2_4 Floating Point

2.4 Floating Point

2.4.1 Fractional Binary Numbers

- Decimal notation:
  - $d_m d_{m-1} \ldots d_1 d_0 . d_{-1} d_{-2} \ldots d_{-n}$
  - value $d = \sum_{i=-n}^{m} 10^i \times d_i$
  - Example: $12.34_{10} = 1 \times 10^1 + 2 \times 10^0 + 3 \times 10^{-1} + 4 \times 10^{-2}$
  - For decimal point symbol ('.'), digits to the left are weighted by nonnegative powers of 10, giving integral values, while digits to the right are weighted by negative powers of 10.
  - For finite-length encodings, decimal notation cannot represent numbers such as $\frac{1}{3}$ and $\frac{5}{7}$
- Fractional binary notation (小数的二进制表示):
  - $b_m b_{m-1} \ldots b_1 b_0 . b_{-1} b_{-2} \ldots b_{-n}$
  - value $b = \sum_{i=-n}^{m} 2^i \times b_i$
  - Example: $101.11_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2}$
  - For binary point symbol ('.'), bits on the left being weighted by nonnegative powers of 2, and those on the right being weighted by negative powers of 2.
  - Fractional binary notation can only represent numbers that can be written $x \times 2^y$. Other values can only be approximated, such as $\frac{1}{5}$
  - Example - how to represent $\frac{1}{5}$ approximately.

Representation	Value	Decimal
0.0_2	$\frac{0}{2}$	0.0_{10}
0.01_2	$\frac{1}{4}$	0.25_{10}
0.010_2	$\frac{2}{8}$	0.25_{10}
0.0011_2	$\frac{3}{16}$	0.1875_{10}
0.00110_2	$\frac{6}{32}$	0.1875_{10}
0.001101_2	$\frac{13}{64}$	0.203125_{10}
0.0011010_2	$\frac{26}{128}$	0.203125_{10}
0.00110011_2	$\frac{51}{256}$	0.19921875_{10}

Practice Problem 2.45

Fill in the missing information in the following table:

Fractional value	Binary representation	Decimal representation
$\frac{1}{8}$	0.001	0.125
$\frac{3}{4}$		
$\frac{5}{16}$		
	10.1011	
	1.001	
		5.875
		3.1875

Solution:

Fractional value Binary representation Decimal representation

$\frac{3}{4} = 2^{-1} + 2^{-2}$ 0.11 0.75

$\frac{5}{16} = 2^{-2} + 2^{-4}$ 0.0101 0.3125

$2\frac{11}{16}$ 10.1011 $1 \times 2^1 + 1 \times 2^{-1} + 1 \times 2^{-3} + 1 \times 2^{-4}$ = 2.688

$1 \times 2^0 + 1 \times 2^{-3} = 1\frac{1}{8}$ 1.001 1.125

$5\frac{7}{8}$ $1 \times 2^2 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}$= 101.111 5.875

$1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-3} + 1 \times 2^{-4} = 3\frac{3}{16}$ 11.0011 3.1875

Practice Problem 2.46

The imprecision of floating-point arithmetic can have disastrous effects. On February 25, 1991, during the first Gulf War (海湾战争), an American Patriot Missile battery (美国爱国者导弹炮台) in Dharan, Saudi Arabia, failed to intercept an incoming Iraqi Scud missile (伊拉克飞毛腿导弹). The Scud struck an American Army barracks and killed 28 soldiers. The US General Accounting Office (GAO) conducted a detailed analysis of the failure and determined that the underlying cause was an imprecision in a numeric calculation. In this exercise, you will reproduce part of the GAO's analysis.

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 $\frac{1}{10}$. In particular, the binary representation of $\frac{1}{10}$ is the nonterminating sequence $0.000110011[0011]\ldots_2$,where the portion in brackets is repeated indefinitely. The program approximated 0.1, as a value x, by considering just the first 23 bits of the sequence to the right of the binary point: x = 0.00011001100110011001100.

1. What is the binary representation of 0.1 - x?

   $0.1_{10} = 0.000110011[0011]\ldots_2$

   $x = 0.00011001100110011001100$

   $0.1 - x = 0.0000000000000000000000[1100]$
2. What is the approximate decimal value of 0.1 - x?

   $0.1 - x = 2^{-20} \times \frac{1}{10}$ = 9.537 x 10^{-8}
3. The clock starts at 0 when the system is first powered up and keeps counting up from there. In this case, the system had been running for around 100 hours. What was the difference between the actual time and the time computed by the software?

   $(0.1 - x) \times 60 \times 60 \times 10 \times 100 = 2^{-20} \times \frac{1}{10} \times 60 \times 60 \times 10 \times 100 = 0.343$
4. The system predicts where an incoming missile will appear based on its velocity and the time of the last radar detection. Given that a Scud travels at around 2,000 meters per second, how far off was its prediction?

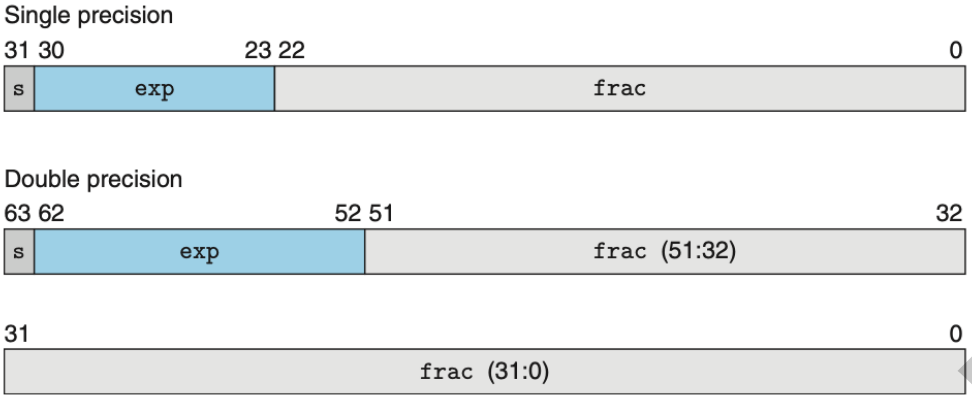
   2000x0.343 = 687

2.4.2 IEEE Floating-Point Representation

- We would like to represent numbers in a form x × 2^y by giving the values of x and y.
- The IEEE floating-point standard represents a number in a form

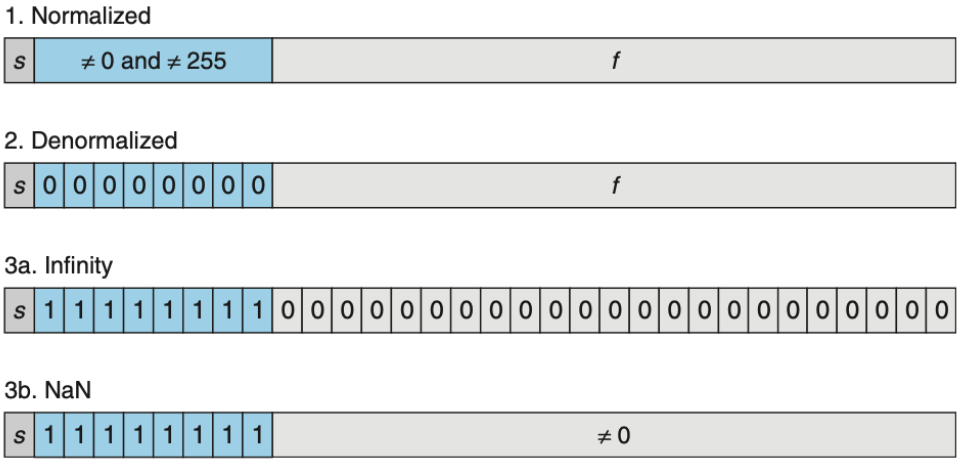
  $V = (-1)^s \times M \times 2^E$:
  - The sign s determines whether the number is negative (s = 1) or positive (s = 0).
  - The significand (有效数) M is a fractional binary number that ranges either between 1 and 2 - ε or between 0 and 1 - ε.
  - The exponent (指数) E weights the value by a (possibly negative) power of 2.
- Floating-point numbers are represented by three fields:
  - The single sign bit s directly encodes the sign s.
  - The k-bit exponent field $exp = e_{k-1} \ldots e_1 e_0$ encodes the exponent E.

- The n-bit fraction field $frac = f_{n-1}...f_1f_0$ encodes the significand M, but the value encoded also depends on whether or not the exponent field equals 0.
- Standard floating-point formats:
  - Single precision floating-point format - 单精度浮点型
    - s = 1, k = 8, n = 23
  - Double precision floating-point format - 双精度浮点型
    - s = 1, k = 11, n = 52



- The value encoded by a given bit representation can be divided into three different cases:
  - Case 1: Normalized Values (规格化) - most common case
    - Pre-condition: $exp \neq [00..0] or [11..1]$
    - We know formula: $V = (-1)^s \times M \times 2^E$
    - Then exponent value: $E = e - bias$
      - $e = [e_{k-1}...e_1e_0]$
      - $bias = 2^{k-1} - 1$
        - For single precision: $bias = 2^7 - 1 = 127$
        - For double precision: $bias = 2^{10} - 1 = 1023$
    - Then significand: $M = 1 + f$ - implied leading 1 representation (隐含以1起始的表示)
      - $f = 0.f_{n-1}...f_1f_0$, $0 \leq f < 1$
      - Then: $1 \leq M < 2$
    - To summarize, in normalized form, $V = (-1)^s \times (1 + 0.f_{n-1}...f_1f_0) \times 2^{[e_{k-1}...e_1e_0] - (2^{k-1}-1)}$
  - Case 2: Denormalized Values (非规格化) - provides a way to represent numeric value 0
    - Pre-condition: $exp = [00..0]$
    - We know formula: $V = (-1)^s \times M \times 2^E$
    - Then exponent value: $E = 1 - bias$
      - $bias = 2^{k-1} - 1$
    - Then $M = f = 0.f_{n-1}...f_1f_0$ - the fraction field
    - To summarize, in denormalized form, $V = (-1)^s \times 0.f_{n-1}...f_1f_0 \times 2^{1-(2^{k-1}-1)}$
    - Denormalized form provides a way to represent numeric value 0
      - When $s = 0$ and $f = 0$, denormalized value = +0.0
      - When $s = 1$ and $f = 0$, denormalized value = -0.0
    - Denormalized form also provides a way to represent numbers that are very close to 0.0. They provide a property known as *gradual underflow* (逐渐溢出) in which possible numeric values are spaced evenly near 0.0.
  - Case 3: Special Values
    - Pre-condition: $exp = [11..1]$
    - If $f = [00..0]$ and $s = 0$, then $V = +\infty$
      - Infinity can represent overflow, that multiply two very large numbers.
    - If $f = [00..0]$ and $s = 1$, then $V = -\infty$
      - Infinity can represent overflow, that divide by zero.
    - If $f = nonzero$, then $V = NaN$ (not a number)

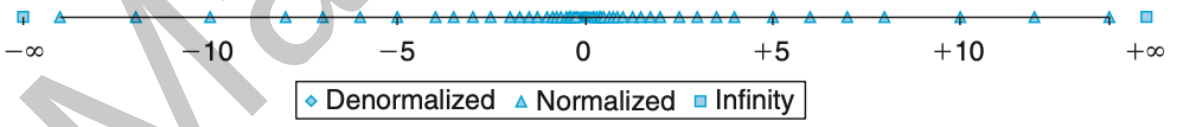
- Such values are returned as the result of an operation where the result cannot be given as a real number or as infinity, as when computing $\sqrt{-1}$ or $\infty - \infty$



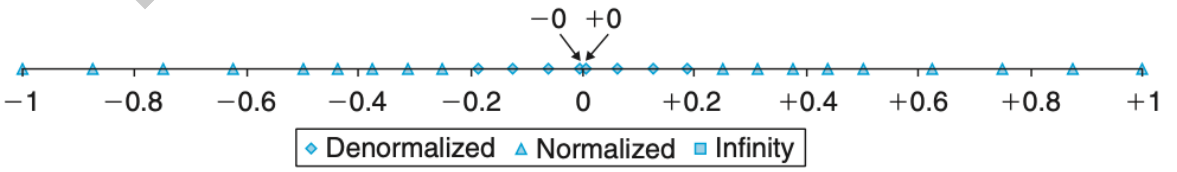
2.4.3 Example Numbers

Example 0

- Assume $w = 6, k = 3, n = 2$
  - bias = $2^{k-1} - 1 = 3$
- In normalized form ($exp \neq [00...0] or [11...1]$), $V = (-1)^s \times (1 + 0.f_{n-1} \dots f_1 f_0) \times 2^{[e_{k-1} \dots e_1 e_0] - (2^{k-1} - 1)} = (-1)^s \times 1.f_1 f_0 \times 2^{[e_2 e_1 e_0] - 3}$
  - Samples:
    - $[000100] = (-1)^0 \times 1.00 \times 2^{[001]-3} = \frac{1}{4}$
      - s = 0, e = [001], f = 0.0
    - $[011111] = NaN$
      - s = 0, e = [111], f = [11]
- In denormalized form ($exp = [00...0]$), $V = (-1)^s \times 0.f_n f_{n-1} \dots f_1 f_0 \times 2^{1-(2^{k-1}-1)} = (-1)^s \times 0.f_1 f_0 \times 2^{-2}$
  - Samples:
    - $[100000] = (-1)^1 \times 0.00 \times 2^{-2} = -0.0$
      - s = 1, e = [000], f = [00]
    - $[100010] = (-1)^1 \times 0.10 \times 2^{-2} = -\frac{1}{8}$
      - s = 1, e = [000], f = [10]



(a) Complete range



(b) Values between −1.0 and +1.0

Example 1

- Assume $w = 8, k = 4, n = 3$
  - bias = $2^{k-1} - 1 = 7$

- In normalized form ($exp \neq [00...0] or [11...1]$),
  $V = (-1)^s \times (1 + 0.f_{n-1}...f_1f_0) \times 2^{[e_{k-1}...e_1e_0]-(2^{k-1}-1)} = (-1)^s \times 1.f_2f_1f_0 \times 2^{[e_3e_2e_1e_0]-7}$
- In denormalized form ($exp = [00...0]$), $V = (-1)^s \times 0.f_nf_{n-1}...f_1f_0 \times 2^{1-(2^{k-1}-1)} = (-1)^s \times 0.f_2f_1f_0 \times 2^{-6}$
- Samples:
  - Zero
    - [0 0000 000] = $(-1)^0 \times 0.f_2f_1f_0 \times 2^{-6}$ = +0.0
    - [1 0000 000] = $(-1)^1 \times 0.f_2f_1f_0 \times 2^{-6}$ = -0.0
  - Smallest positive denormalized
    - [0 0000 001] = $(-1)^0 \times 0.001 \times 2^{-6} = 2^{-9} = \frac{1}{512}$
  - Largest negative denormalized
    - [1 0000 001] = $(-1)^1 \times 0.001 \times 2^{-6} = 2^{-9} = -\frac{1}{512}$
  - Largest positive denormalized
    - [0 0000 111] = $(-1)^0 \times 0.111 \times 2^{-6} = 7 \times 2^{-9} = \frac{7}{512}$
  - Smallest negative denormalized
    - [1 0000 111] = $(-1)^1 \times 0.111 \times 2^{-6} = 7 \times 2^{-9} = -\frac{7}{512}$
  - Smallest positive normalized
    - [0 0001 000] = $(-1)^0 \times 1.000 \times 2^{[0001]-7} = 2^{-6} = \frac{1}{64}$
  - Largest negative normalized
    - [1 0001 000] = $(-1)^1 \times 1.000 \times 2^{[0001]-7} = -2^{-6} = -\frac{1}{64}$
  - Positive one
    - [0 0111 000] = $(-1)^s \times (1 + 0.f_2f_1f_0) \times 2^{e-bias}$ = $(-1)^0 \times 1.000 \times 2^{[0111]-7} = 1.0$
  - Negative one
    - [1 0111 000] = $(-1)^1 \times 1.000 \times 2^{[0111]-7} = -1.0$
  - Largest normalized
    - [0 1110 111] = $(-1)^0 \times 1.111 \times 2^{[1110]-7} = 240.0$
  - Smallest normalized
    - [1 1110 111] = $(-1)^1 \times 1.111 \times 2^{[1110]-7} = -240.0$
  - Positve infinity
    - [0 1111 000] = $+\infty$
  - Negative infinity
    - [1 1111 000] = $-\infty$

Description	Bit representation	Exponent			Fraction		Value		
		e	E	2^E	f	M	$2^E \times M$	V	Decimal
Zero	0 0000 000	0	−6	$\frac{1}{64}$	$\frac{0}{8}$	$\frac{0}{8}$	$\frac{0}{512}$	0	0.0
Smallest positive	0 0000 001	0	−6	$\frac{1}{64}$	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{1}{512}$	$\frac{1}{512}$	0.001953
	0 0000 010	0	−6	$\frac{1}{64}$	$\frac{2}{8}$	$\frac{2}{8}$	$\frac{2}{512}$	$\frac{1}{256}$	0.003906
	0 0000 011	0	−6	$\frac{1}{64}$	$\frac{3}{8}$	$\frac{3}{8}$	$\frac{3}{512}$	$\frac{3}{512}$	0.005859
	⋮								
Largest denormalized	0 0000 111	0	−6	$\frac{1}{64}$	$\frac{7}{8}$	$\frac{7}{8}$	$\frac{7}{512}$	$\frac{7}{512}$	0.013672
Smallest normalized	0 0001 000	1	−6	$\frac{1}{64}$	$\frac{0}{8}$	$\frac{8}{8}$	$\frac{8}{512}$	$\frac{1}{64}$	0.015625
	0 0001 001	1	−6	$\frac{1}{64}$	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{512}$	$\frac{9}{512}$	0.017578
	⋮								
	0 0110 110	6	−1	$\frac{1}{2}$	$\frac{6}{8}$	$\frac{14}{8}$	$\frac{14}{16}$	$\frac{7}{8}$	0.875
	0 0110 111	6	−1	$\frac{1}{2}$	$\frac{7}{8}$	$\frac{15}{8}$	$\frac{15}{16}$	$\frac{15}{16}$	0.9375
One	0 0111 000	7	0	1	$\frac{0}{8}$	$\frac{8}{8}$	$\frac{8}{8}$	1	1.0
	0 0111 001	7	0	1	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{8}$	$\frac{9}{8}$	1.125
	0 0111 010	7	0	1	$\frac{2}{8}$	$\frac{10}{8}$	$\frac{10}{8}$	$\frac{5}{4}$	1.25
	⋮								
	0 1110 110	14	7	128	$\frac{6}{8}$	$\frac{14}{8}$	$\frac{1792}{8}$	224	224.0
Largest normalized	0 1110 111	14	7	128	$\frac{7}{8}$	$\frac{15}{8}$	$\frac{1920}{8}$	240	240.0
Infinity	0 1111 000	—	—	—	—	—	—	∞	—

- Some tips:
  - The smooth transition between the largest denormalized number $\frac{7}{512}$ and the smallest normalized number $\frac{8}{512}$.
  - The largest normalized value = 240.0. Going beyond this overflows to $+\infty$
  - The IEEE format was designed so that floating-point numbers could be sorted using an integer sorting routine.

Practice Problem 2.47

Consider a 5-bit floating-point representation based on the IEEE floating-point format, with one sign bit, two exponent bits (k = 2), and two fraction bits (n = 2). The exponent bias is $2^{2-1}-1=1$.

The table that follows enumerates the entire nonnegative range for this 5-bit floating-point representation. Fill in the blank table entries using the following directions:

e: The value represented by considering the exponent field to be an unsigned integer

E: The value of the exponent after biasing

2^E: The numeric weight of the exponent

f : The value of the fraction

M: The value of the significand

$2^E \times M$: The (unreduced) fractional value of the number

V: The reduced fractional value of the number

$Decimal$: The decimal representation of the number

Express the values of $2^E, f, M, 2^E \times M$, and V either as integers (when possible) or as fractions of the form $\frac{x}{y}$, where y is a power of 2. You need not fill in entries marked —.

Bits	e	E	2^E	f	M	$2^E \times M$	V	Decimal
0 00 00								
0 00 01								
0 00 10								
0 00 11								
0 01 00								
0 01 01	1	0	1	$\frac{1}{4}$	$\frac{5}{4}$	$\frac{5}{4}$	$\frac{5}{4}$	1.25

Bits	e	E	2^E	f	M	$2^E \times M$	V	Decimal
0 01 10								
0 01 11								
0 10 00								
0 10 01								
0 10 10								
0 10 11								
0 11 00	—	—	—	—	—	—		—
0 11 01	—	—	—	—	—	—		—
0 11 10	—	—	—	—	—	—		—
0 11 11	—	—	—	—	—	—		—

Solution:

k = 2, n = 2, bias = 1

Denormalized:

[0 00 00]: e = [00], E = 1-bias = 0, 2^E = 1, f = 0.00, M = f = 0.00, V = +0.0

[0 00 01]: e = [00], E = 0, 2^E = 1, f = 0.01_2, M = f = 0.01_2, V = $0.01_2 \times 1$ = 0.25

[0 00 10]: e = [00], E = 0, 2^E = 1, f = 0.10_2, M = f = 0.10_2, V = $0.10_2 \times 1$ = 0.5

[0 00 11]: e = [00], E = 0, 2^E = 1, f = 0.11_2, M = f = 0.11_2, V = $0.11_2 \times 1$ = 0.75

Normalized:

[0 01 00]: e = [01], E = e - bias = $[01] - bias$ = 0, 2^E = 1, f = 0.00, M = 1+f = 1.00, V = $(-1)^0 \times 1.00_2 \times 1$ = 1.0

[0 01 01]: e = [01], E = 0, 2^E = 1, f = 0.01_2, M = 1.01_2, V = $(-1)^0 \times 1.01_2 \times 1$ = 1.25

[0 01 10]: e = [01], E = 0, 2^E = 1, f = 0.10_2, M = 1.1_2, V = $(-1)^0 \times 1.1_2 \times 1$ = 1.5

[0 01 11]: e = [01], E = 0, 2^E = 1, f = 0.11_2, M = 1.11_2, V = $(-1)^0 \times 1.11_2 \times 1$ = 1.75

[0 10 00]: e = [10], E = e-bias = 1, 2^E = 2, f = 0.0_2, M = 1.0_2, V = $(-1)^0 \times 1.0_2 \times 2$ = 2.0

[0 10 01]: e = [10], E = 1, 2^E = 2, f = 0.01_2, M = 1.01_2, V = $(-1)^0 \times 1.01_2 \times 2$ = 2.5

[0 10 10]: e = [10], E = 1, 2^E = 2, f = 0.10_2, M = 1.10_2, V = $(-1)^0 \times 1.10_2 \times 2$ = 3

[0 10 11]: e = [10], E = 1, 2^E = 2, f = 0.11_2, M = 1.11_2, V = $(-1)^0 \times 1.11_2 \times 2$ = 3.5

Special Values:

[0 11 00]: e = [11], f = 0.0, = $+\infty$
[0 11 01]: NaN
[0 11 10]: NaN
[0 11 11]: NaN

- Single and double precision floating-point numbers.
  - Single precision floating-point (单精度浮点型):
    - w =32, k = 8, n = 23
    - Normalized form (e != [0...0] or [1...1]):
      $V = (-1)^s \times (1 + 0.f_{22} \ldots f_1 f_0) \times 2^{[e_7 \ldots e_1 e_0] - (2^{8-1}-1)} = (-1)^s \times 1.f_{22} \ldots f_1 f_0 \times 2^{[e_7 \ldots e_1 e_0] - 127_{10}}$
    - Denormalized form (e == [0...0]):

$V = (-1)^s \times 0.f_{22} \dots f_1 f_0 \times 2^{1-(2^{8-1}-1)} = (-1)^s \times 0.f_{22} \dots f_1 f_0 \times 2^{-126}$

- Positive Layout (ascending): $bias = 2^{k-1} - 1 = 127$
  - 0 - zero: [0 0...0 00...00] = +0.0
  - 2^{-149} - smallest positive denormalized: [0 0...0 00...01] = $(-1)^0 \times 0.0 \dots 01_2 \times 2^{-126} = 2^{-149}$
  - 2^{-126} - largest positive denormalized: [0 0...0 11...11] = $(-1)^0 \times 0.1 \dots 11_2 \times 2^{-126} = (1 - \frac{1}{2^{23}}) \times 2^{-126} = (1 - \epsilon) \times 2^{-126} \approx 2^{-126}$
  - 2^{-126} - smallest positive normalized: [0 0...01 00...00] = $(-1)^0 \times (1 + 0.0 \dots 00) \times 2^{[0 \dots 01] - (2^{8-1}-1)} = 2^{-126}$
  - 1 - one: [0 01...11 0...00] = 1
  - 2^{128} - largest positive normalized: [0 11...10 1...11] = $(-1)^0 \times (1 + 0.1 \dots 11) \times 2^{[1 \dots 10] - (2^{8-1}-1)} = (2 - \epsilon) \times 2^{127} \approx 2^{128}$
  - $+\infty$ - positive infinity: [0 11...11 00...00]
- Double precision floating-point (双精度浮点型):
  - w = 64, k = 11, n = 52
  - Normalized form: $V = (-1)^s \times (1 + 0.f_{51} \dots f_1 f_0) \times 2^{[e_{10} \dots e_1 e_0] - (2^{11-1}-1)} = (-1)^s \times 1.f_{51} \dots f_1 f_0 \times 2^{[e_{10} \dots e_1 e_0] - 1023_{10}}$
  - Denormalized form:

    $V = (-1)^s \times 0.f_{51} \dots f_1 f_0 \times 2^{1-(2^{11-1}-1)} = (-1)^s \times 0.f_{51} \dots f_1 f_0 \times 2^{-1022}$
  - 0 - zero
  - 2^{-1074} - smallest denormalized
  - 2^{-1022} - largest denormalized
  - 2^{-1022} - smallest normalized
  - 1 - one
  - 2^{1024} - largest normalized
  - $+\infty$ - positive infinity

			Single precision		Double precision	
Description	exp	frac	Value	Decimal	Value	Decimal
Zero	$00 \cdots 00$	$0 \cdots 00$	0	0.0	0	0.0
Smallest denormalized	$00 \cdots 00$	$0 \cdots 01$	$2^{-23} \times 2^{-126}$	1.4×10^{-45}	$2^{-52} \times 2^{-1022}$	4.9×10^{-324}
Largest denormalized	$00 \cdots 00$	$1 \cdots 11$	$(1-\epsilon) \times 2^{-126}$	1.2×10^{-38}	$(1-\epsilon) \times 2^{-1022}$	2.2×10^{-308}
Smallest normalized	$00 \cdots 01$	$0 \cdots 00$	1×2^{-126}	1.2×10^{-38}	1×2^{-1022}	2.2×10^{-308}
One	$01 \cdots 11$	$0 \cdots 00$	1×2^0	1.0	1×2^0	1.0
Largest normalized	$11 \cdots 10$	$1 \cdots 11$	$(2-\epsilon) \times 2^{127}$	3.4×10^{38}	$(2-\epsilon) \times 2^{1023}$	1.8×10^{308}

- One useful exercise for understanding floating-point representations is to **convert sample integer values into floating-point form.**
  - Example:
    - we know $12345_{10} = [11000000111001]$
    - suppose single-precison float: k = 8, n = 23
    - then we get: $12345.0_{10} = (-1)^0 \times (1 + 0.10000001110010000000000_2) \times 2^{13}$ - normalized
      - $[e_7 \dots e_0] - (2^7 - 1) = 13$ -> $[e_7 \dots e_0] = [10001100]$
    - To summarize: single precision float result is [0 10001100 10000001110010000000000] which contains the same part of bit pattern [1000000111001] as integer bit pattern.

Practice Problem 2.48

As mentioned in Problem 2.6, the integer 3,510,593 has hexadecimal representation 0x00359141, while the single-precision floating-point number 3,510,593.0 has hexadecimal representation 0x4A564504. Derive this floating-point representation and explain the correlation between the bits of the integer and floating-point representations.

Solution:

0x00359141 = [0000 0000 0011 0101 1001 0001 0100 0001]

= $(-1)^0 \times 1.10101100100010100000{1}_2 \times 2^{21}$ in the format of IEEE: $V = (-1)^s \times M \times 2^E$

E = 21 = e - bias, M = $1.10101100100010100000100_2$

For single-precision floating, bias = 127, so e = $148_{10} = [10010100]$

M = 1+ f, so f = $[10101100100010100000100]$

Therefore, after converting to single-precision floating-point format, it should be $[01001010010101100100010100000100]$

which is exactly same as 0x4A564504

0x4A564504 = [0100 1010 0101 0110 0100 0101 0000 0100]

From the derivation process, we get to know that $[f_{n-1} \dots f_1 f_0]$ should be same for both representations, which is $[10101100100010100000{1}]$

Practice Problem 2.49

1. For a floating-point format with an n-bit fraction, give a formula for the **smallest positive integer** that cannot be represented exactly (because it would require an (n + 1)-bit fraction to be exact). Assume the exponent field size k is large enough that the range of representable exponents does not provide a limitation for this problem.

   Solution:

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

   Because it requires smallest positive integer, it must be normalized form:
   $V = (1 + 0.f_{n-1} \dots f_1 f_0) \times 2^{[e_{k-1} \dots e_1 e_0] - (2^{k-1} - 1)}$

   Logically, if smallest, it would be $1.0_{n-1} \dots .0_1 0_0 1 \times 2^{n+1} = 2^{n+1} + 1$
2. What is the numeric value of this integer for single-precision format (n = 23)?

   $2^{24} + 1$

2.4.4 Rounding (舍入)

- Floating-point arithmetic can only approximate real arithmetic, since the representation has limited range and precision.
- **Rounding operation:** for a value x, we generally want a systematic method of finding the "closest" matching value x that can be represented in the desired floating-point format.
- How to do rounding? - 4 rounding modes:
  1. Round-to-even (round-to-nearest):
     - Firstly attempts to find a closest match.
     - Secondly if it's halfway between 2 possible results, round-to-even mode adopts the convention that it rounds the number either upward or downward such that the least significant bit of the result is even.
     - Example:

```
1.60  -> 2
1.40  -> 1
1.50  -> LSB of 2 is 0 which is even -> 2
2.50  -> LSB of 2 is 0 which is even -> 2
-1.50 -> -2
```

  2. Round-toward-zero: rounds positive numbers downward and negative numbers upward.

```
1.60  -> 1
1.40  -> 1
1.50  -> 1
2.50  -> 2
```

```
5           -1.50 -> -1
```

3. Round-down: rounds both positive and negative numbers downward.

```
1           1.60  -> 1
2           1.40  -> 1
3           1.50  -> 1
4           2.50  -> 2
5           -1.50 -> -2
```

4. Round-up: rounds both positive and negative numbers upward.

```
1           1.60  -> 2
2           1.40  -> 2
3           1.50  -> 2
4           2.50  -> 3
5           -1.50 -> -1
```

- Round-to-even rounding can be applied to binary fractional numbers.
  - Least significant bit (LSB) value 0 to be even and 1 to be odd.
  - Examples - rounding to nearest quarter (2 bits to the right of the binary point):
    - 10.00011_2 round down to 10.00_2 - not half way
    - 10.00110_2 round up to 10.01_2 - not half way
    - 10.11100_2 round up to 11.00_2 - half way, LSB == 1
      - the 3rd bit after point is 1 and the bits after 3rd bit is 0, which means it's half-way.
      - 2 cases:
        - if round-up, the rounding value = 11.00 whose LSB is 0, which matches.
        - if round-down, the rounding value = 10.11 whose LSB is 1.
    - 10.10100_2 round down to 10.10_2 - half way, LSB == 0
      - the 3rd bit after point is 1 and the bits after 3rd bit is 0, which means it's half-way.
      - 2 cases:
        - if round-up, the rounding value = 10.11 whose LSB is 1.
        - if round-down, the rounding value = 10.10 whose LSB is 0, which matches.

Practice Problem 2.50

Show how the following binary fractional values would be rounded to the nearest half (1 bit to the right of the binary point), according to the round-to-even rule. In each case, show the numeric values, both before and after rounding.
A. 10.111_2 B. 11.010_2 C. 11.000_2 D. 10.110_2

A. 10.111_2 - not half way = 11.0_2
B. 11.010_2 - half way = 11.0_2
2 cases:
round up - 11.1_2 whose LSB is 1
round down - 11.0_2 whose LSB is 0 (even).
C. 11.000_2 - not half way = 11.0_2
D. 10.110_2 - half way = 11.0_2
2 cases:
round up - 11.0_2 whose LSB is 0 (even).
round down - 10.1_2 whose LSB is 1.

Practice Problem 2.51

We saw in Problem 2.46 that the Patriot missile software approximated 0.1 as x = $0.0001100110011001100110_2$. Suppose instead that they had used IEEE round-to-even mode to determine an approximation x' to 0.1 with 23 bits to the right of the binary point.

A. What is the binary representation of x'?

Not half: $0.000110011001100110011001100(11)_2$->$0.00011001100110011001101_2$

B. What is the approximate decimal value of x' − 0.1?

$0.000000000000000000000000[1100]_2$

C. How far off would the computed clock have been after 100 hours of operation?

$0.000000000000000000000000[1100]_2 \times 100 \times 60 \times 60 \times 10 \approx 0.086$

D. How far off would the program's prediction of the position of the Scud missile have been?

171

Practice Problem 2.52

Consider the following two 7-bit floating-point representations based on the IEEE floating-point format. Neither has a sign bit - they can only represent nonnegative numbers.

1. Format A
   - There are k = 3 exponent bits. The exponent bias is 3.
   - There are n = 4 fraction bits.
2. Format B

   There are k = 4 exponent bits. The exponent bias is 7.

   There are n = 3 fraction bits.

   Below, you are given some bit patterns in format A, and your task is to convert them to the closest value in format B. If necessary, you should apply the round-to-even rounding rule. In addition, give the values of numbers given by the format A and format B bit patterns. Give these as whole numbers (e.g., 17) or as fractions (e.g., 17/64).

Format A - Bits	Value	Format B - Bits	Value
011 0000	1.0	0111 000	1.0
101 1110	$7\frac{1}{2}$	1001 111	$7\frac{1}{2}$
010 1001	$\frac{25}{32}$	0110 100	$\frac{3}{4}$
110 1111	$15\frac{1}{2}$	1011 000	16
000 0001	$\frac{1}{64}$	0001 000	$\frac{1}{64}$

Solution:

Format A:

Normalized - $V = 1.f_3f_2f_1f_0 \times 2^{[e_2e_1e_0]-3}$

Denormalized - $V = 0.f_3f_2f_1f_0 \times 2^{-2}$

Format B:

Normalized - $V = 1.f_2f_1f_0 \times 2^{[e_3e_2e_1e_0]-7}$

Denormalized - $V = 0.f_2f_1f_0 \times 2^{-6}$

011 0000:

Format A: $V = 1.0000 \times 2^{3-3} = 1.0$

Convert to format B: $1.0 = 1.000 \times 2^{[0111]-7} = [0111000]_2$

101 1110:

Format A: $V = 1.1110 \times 2^{[101]-3} = 1.1110 \times 2^2 = 111.1_2$

Convert to format B: $1.1110 \times 2^2 = 1.111 \times 2^{[1001]-7} = [1001111]_2$

010 1001:

Format A: $V = 1.1001 \times 2^{[010]-3} = 1.1001 \times 2^{-1} = \frac{25}{32}$

Convert to format B:

$$1.1001 \times 2^{-1} \approx 1.100 \times 2^{[0110]-7} = [0110100]_2 = \frac{3}{4}$$

110 1111:
Format A: $V = 1.1111 \times 2^{[110]-3} = 1.1111 \times 2^3 = 15\frac{1}{2}$
Convert to format B:

$$1.1111 \times 2^3 \approx 10.000 \times 2^3 = 1.000 \times 2^4 = 1.000 \times 2^{[1011]-7} = [1011000]_2 = 16$$

000 0001:
Format A (denormalized): $V = 0.0001 \times 2^{-2} = \frac{1}{64}$
Convert to format B (normalized):

$$0.0001 \times 2^{-2} = 1.000 \times 2^{[0001]-7} = [0001000]_2 = \frac{1}{64}$$

Tip: denormalized -> normalized

2.4.5 Floating-Point Operations

- IEEE standard's method of specifying the behavior of floating-point operations is that it is independent of any particular hardware or software realization.
- Floating-point Addition:
  - Define $x +^f y$ to be $Round(x + y)$
  - Addition over real numbers also forms an abelian group.
    - Most values have inverses under floating-point addition: $x +^f (-x) = 0$
    - The exceptions are infinities and NaNs:
      - $+\infty + (-\infty) = NaN$
      - for any x: $NaN +^f x = NaN$
  - Is commutative: $x +^f y = y +^f x$
  - Not associative:
    - $(3.14 + 1e10) - 1e10 = 0.0$ while $3.14 + (1e10 - 1e10) = 3.14$
    - Scientific notation: $1e10 = 1 \times 10^{10}$
  - Is monotonic (单调性):
    - For any values of a,b and x other than NaN: if $a \geq b$, then $x +^f a \geq x +^f b$
- Floating-point Multiplication:
  - Define $x *^f y = Round(x \times y)$
  - Is commutative: $x *^f y = y *^f x$
  - Not associative: $(1e20 * 1e20) * 1e-20 = +\infty$ while $1e20 * (1e20 * 1e-20) = 1e20$
  - Not distributive over addition:
    - With single-precision floating point: $1e20 * (1e20 - 1e20) = 0.0$ while $1e20 * 1e20 - 1e20 * 1e20 = NaN$
  - Is monotonic (单调性):
    - if $a \geq b$ and $c \geq 0$, then $a *^f c \geq b *^f c$
    - if $a \geq b$ and $c \leq 0$, then $a *^f c \leq b *^f c$
  - As long as $a \neq NaN$, $a *^f a \geq 0$

2.4.6 Floating Point in C

- C provide two different floating-point data types: **float and double.** The machines use **round-to-even** round mode.

Practice Problem 2.53

Fill in the following macro definitions to generate the double-precision values +∞, −∞, and −0:

```
1   #define POS_INFINITY
2   #define NEG_INFINITY
```

```
3    #define NEG_ZERO
```

Solution:
We know that the largest number of Normalized in double-precision mode is 2^{1024}
Then we get $2^{1024} < 8^{342} < 10^{350} = 1e350$

```
1    #define POS_INFINITY 1e350
2    #define NEG_INFINITY (-POS_INFINITY)
3    #define NEG_ZERO (-1.0/POS_INFINITY)
```

- Casting from int to float, the number cannot overflow, but it may be rounded.
  - For integer, n = 32
  - For float, w = 32, k = 8, n = 23
  - $V = (-1)^s \times 1.f_{22} \dots f_1 f_0 \times 2^{[e_7 \dots e_1 e_0]_2 - 127_{10}} = 1.111 \dots 11 \times 2^{127}$
- From int or float to double, the exact numeric value can be preserved because double has both greater range, as well as greater precision.
- From double to float, the value can overflow to +∞ or −∞, since the range is smaller. Otherwise, it may be rounded, because the precision is smaller.
- From float or double to int, the value will be rounded toward zero.
  - 2.9 casting from float to integer, it will be 2
  - -2.9 casting from float to integer, it will be -2
  - But it may also be overflow.

Practice Problem 2.54

Assume variables x, f, and d are of type int, float, and double, respectively. Their values are arbitrary, except that neither f nor d equals +∞, −∞, or NaN. For each of the following C expressions, either argue that it will always be true or give a value for the variables such that it is not true.
A. x == (int)(double)x
True. integer -> double: numeric values preserved
B. x == (int)(float)x
FALSE. integer -> float: numeric value may be rounded.
if $x = 2^{24} + 1$, then $(float)x == 2^{24}$
C. d == (double)(float)d
FALSE. double -> float: numeric value may be overflow.
if $d = 1e30$, then $(float)d == +\infty$
D. f == (float)(double)f
TRUE.
E. f == -(-f)
TRUE. Just inverse the signed bit.
F. 1.0/2 == 1/2.0
TRUE. Convert to float.
G. $d * d >= 0.0$
TRUE. The worst case is $+\infty$
H. (f+d)-f == d
FALSE.

Bit-Level Integer Coding Rules

- Assumptions:
  - Integers are represented in two's-complement form.
    - Right shifts of signed data are performed arithmetically.
    - Data type int is w bits long. For some of the problems, you will be given a specific value for w, but otherwise your code should work as long as w is a

multiple of 8. You can use the expression sizeof(int)<<3 to compute w.

- Forbidden:
  - Conditionals (if or ?:), loops, switch statements, function calls, and macro invocations.
    - Division, modulus, and multiplication.
    - Relative comparison operators (<, >, <=, and >=).
- Allowed operations:
  - All bit-level and logic operations.
    - Left and right shifts, but only with shift amounts between 0 and w - 1.
    - Addition and subtraction.
    - Equality (==) and inequality (!=) tests. (Some of the problems do not allow these.)
    - Integer constants INT_MIN and INT_MAX.
    - Casting between data types int and unsigned, either explicitly or implicitly.
    - Make your code readable by choosing descriptive variable names and using comments to describe the logic behind your solutions.

```c
/* Get most significant byte from x */
    int get_msb(int x) {

        /* Shift by w-8 */
        int shift_val = (sizeof(int)-1)<<3;
        /* Arithmetic shift */
        int xright = x >> shift_val;
        /* Zero all but LSB */
        return xright & 0xFF;

    }
```