Floating Point Representation
Lecture 09

Floating Point Representation

- A floating-point representation encodes rational numbers of the form $V = x \times 2^y$.
- It is useful for performing computations involving very large numbers ($|V| >> 0$), numbers very close to 0 ($|V| << 1$), and more generally as an approximation to real arithmetic.

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

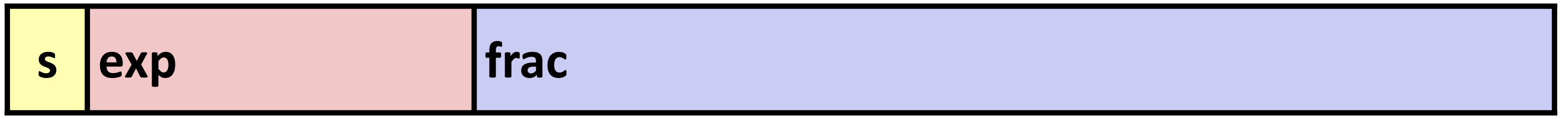
Floating Point Representation

- **Numerical Form:**

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

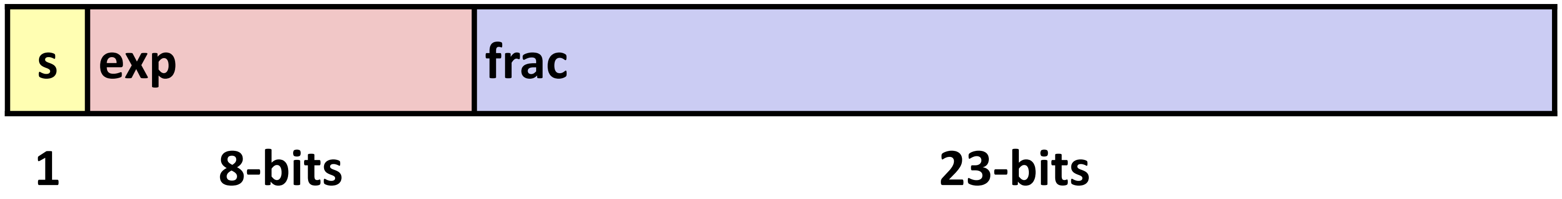
  - **Sign bit *s*** determines whether number is negative or positive
  - **Significand *M*** normally a fractional value in range [1.0,2.0).
  - **Exponent *E*** weights value by power of two

- **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)

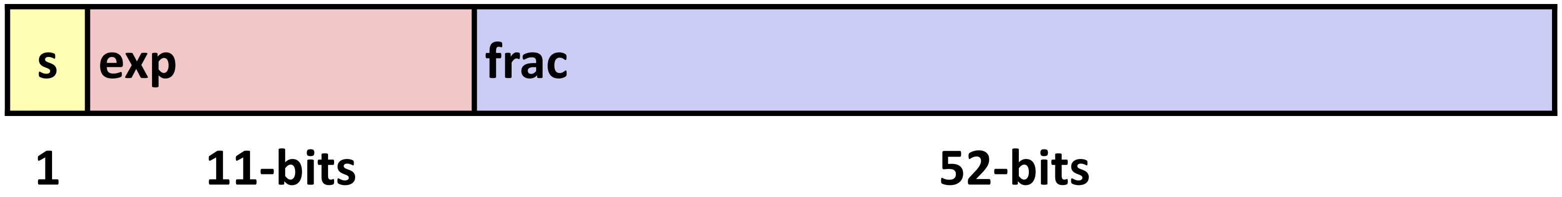


Precision options

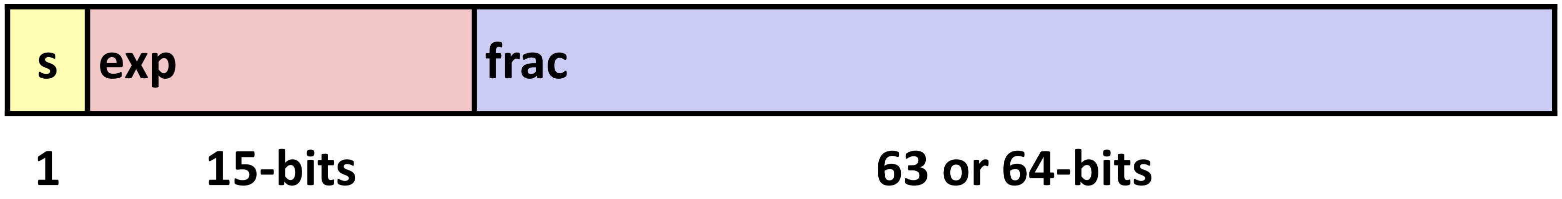
- Single precision: 32 bits

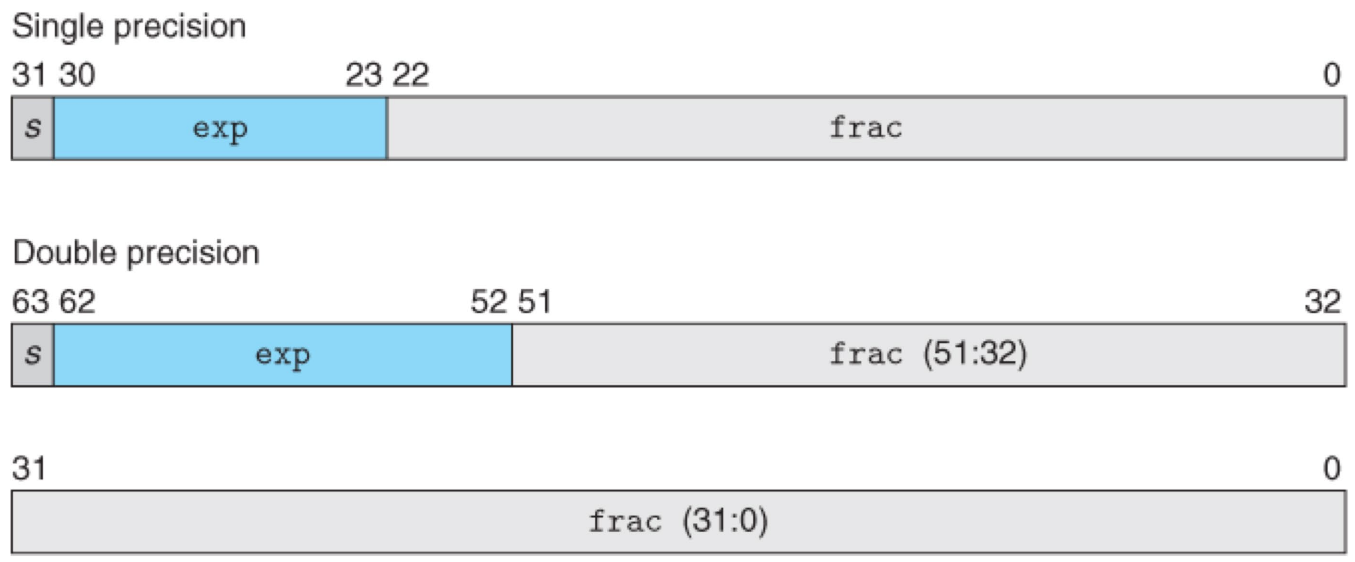


- Double precision: 64 bits



- Extended precision: 80 bits (Intel only)





Floating-point numbers are represented by three fields. For the two most common formats, these are packed in 32-bit (single-precision) or 64-bit (double-precision) words.

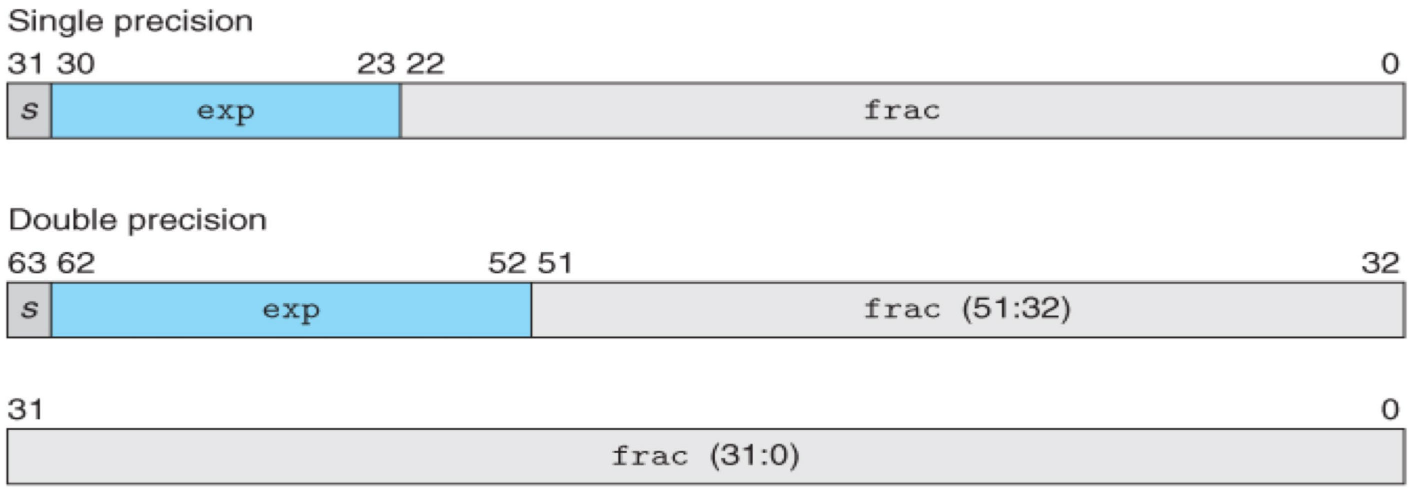
The bit representation of a floating-point number is divided into three fields to encode these values:

The single sign bit s directly encodes the sign *s.*

The *k*-bit exponent field exp = $e_{k-1} \cdots e_1 e_0$ encodes the exponent *E.*

The *n*-bit fraction field frac = $f_{n-1} \cdots f_1 f_0$ encodes the significand *M*, but the value encoded also depends on whether or not the exponent field equals 0.

- shows the packing of these three fields into words for the two most common formats. In the single-precision floating-point format (a float in C), fields s, exp, and frac are 1, *k* = 8, and *n*= 23 bits each, yielding a 32-bit representation. In the double-precision floating-point format (a double in C), fields s, exp, and frac are 1, *k* = 11, and *n* = 52 bits each, yielding a 64-bit representation.



Case 1: Normalized Values

- This is the most common case. It occurs when the bit pattern of exp is neither all zeros (numeric value 0) nor all ones (numeric value 255 for single precision, 2047 for double).
- In this case, the exponent field is interpreted as representing a signed integer in *biased* form. That is, the exponent value is $E = e - Bias$, where e is the unsigned number having bit representation $e_{k-1} \ldots e_1 e_0$ and *Bias* is a bias value equal to $2^{k-1} - 1$ (127 for single precision and 1023 for double). This yields exponent ranges from −126 to +127 for single precision and −1022 to +1023 for double precision.
- The fraction field frac is interpreted as representing the fractional value f, where $0 \leq f < 1$, having binary representation $0.f_{n-1} \ldots f_1 f_0$, that is, with the binary point to the left of the most significant bit.

“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

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

$E = Exp - Bias$

- **Value:** `float F = 15213.0;`
  - $15213_{10} = 11101101101101_2$

    $= 1.1101101101101_2 \times 2^{13}$

- **Significand**

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

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

- **Exponent**

  $E \quad = \quad 13$

  $Bias \quad = \quad 127$

  $Exp \quad = \quad 140 \quad = \quad 10001100_2$

- **Result:**

0	10001100	11011011011010000000000
s	**exp**	**frac**

Case 2: Denormalized Values

- When the exponent field is all zeros, the represente number is in *denormalized* form. In this case, the exponent value is $E = 1 - Bias$, and the significand value is $M = f$, that is, the value of the fraction field without an implied leading 1.
- Denormalized numbers serve two purposes. First, they provide a way to represent numeric value 0, since with a normalized number we must always have $M \geq 1$, and hence we cannot represent 0. In fact, the floating-point representation of +0.0 has a bit pattern of all zeros: the sign bit is 0, the exponent field is all zeros (indicating a denormalized value), and the fraction field is all zeros, giving $M = f = 0$. Curiously, when the sign bit is 1, but the other fields are all zeros, we get the value −0.0. With IEEE floating-point format, the values −0.0 and +0.0 are considered different in some ways and the same in others.
- A second function of denormalized numbers is to represent numbers that are very close to 0.0

Denormalized Values

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

$E = 1 - Bias$

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

Case 3: Special Values

- A final category of values occurs when the exponent field is all ones. When the fraction field is all zeros, the resulting values represent infinity, either $+\infty$ when $s = 0$ or $-\infty$ when $s = 1$. Infinity can represent results that *overflow*, as when we multiply two very large numbers, or when we divide by zero.
- When the fraction field is nonzero, the resulting value is called **a *"NaN,"* short for "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 $\infty - \infty$. They can also be useful in some applications for representing uninitialized data.

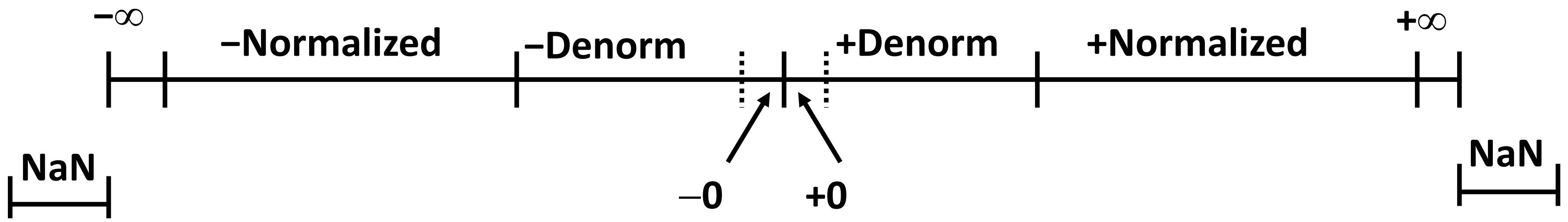
Case 3: Special Values

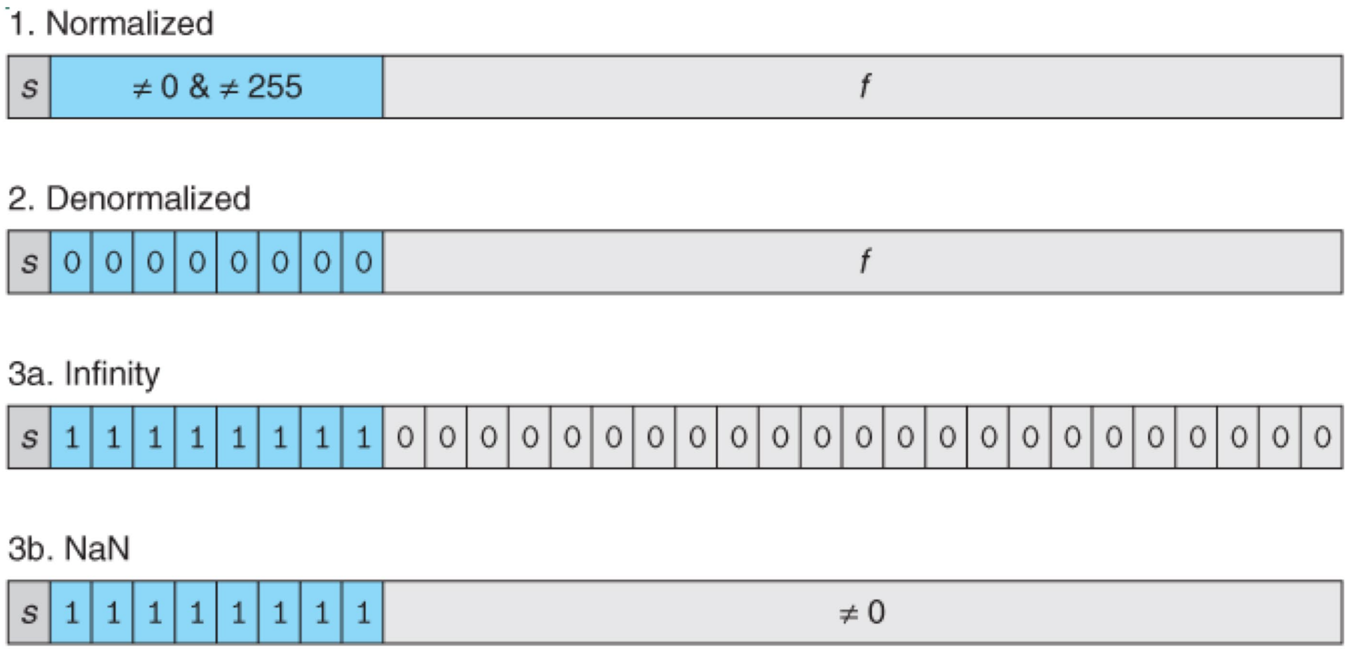
- **Condition: `exp` = `111…1`**

- **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 = −∞

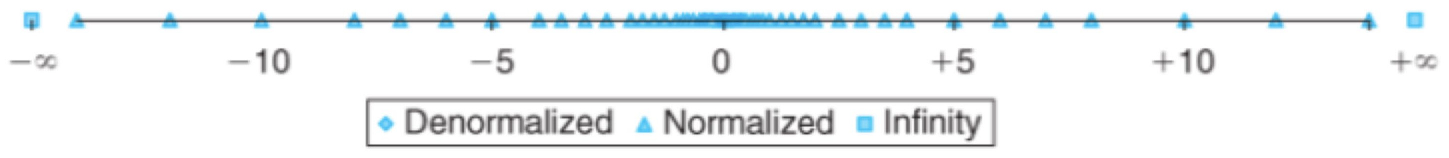
- **Case: `exp` = `111…1`, `frac` ≠ `000…0`**
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., sqrt(–1), ∞ − ∞, ∞ × 0

Visualization: Floating Point Encodings

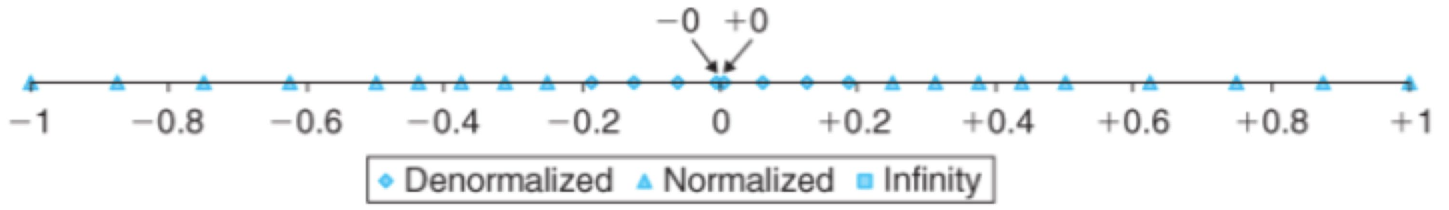



1. Normalized
s
≠ 0 & ≠ 255
f
2. Denormalized
s
0 0 0 0 0 0 0 0
f
3a. Infinity
s
1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
3b. NaN
s
1 1 1 1 1 1 1 1
≠ 0

Example Numbers



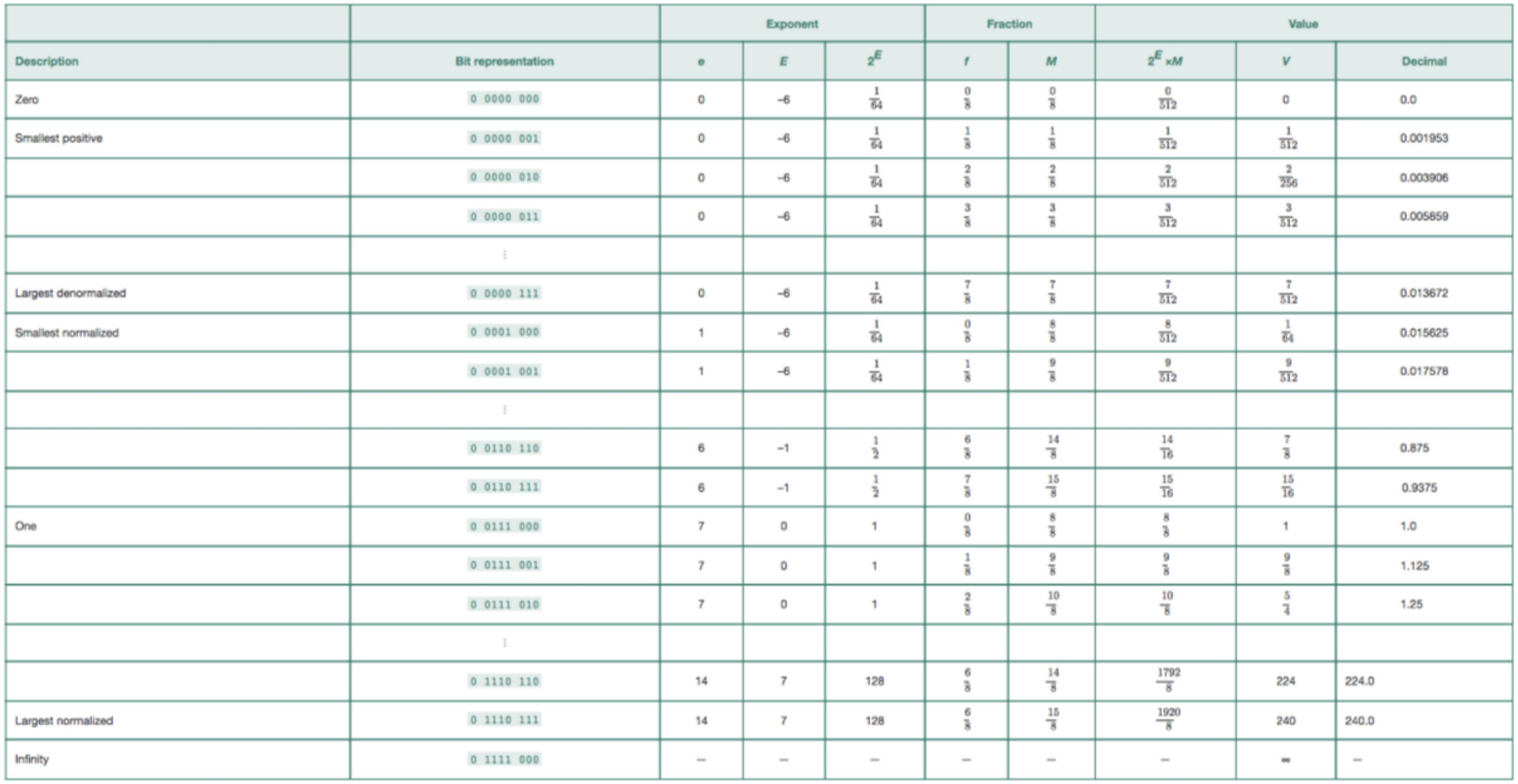
(a) Complete range



(b) Values between −1.0 and +1.0

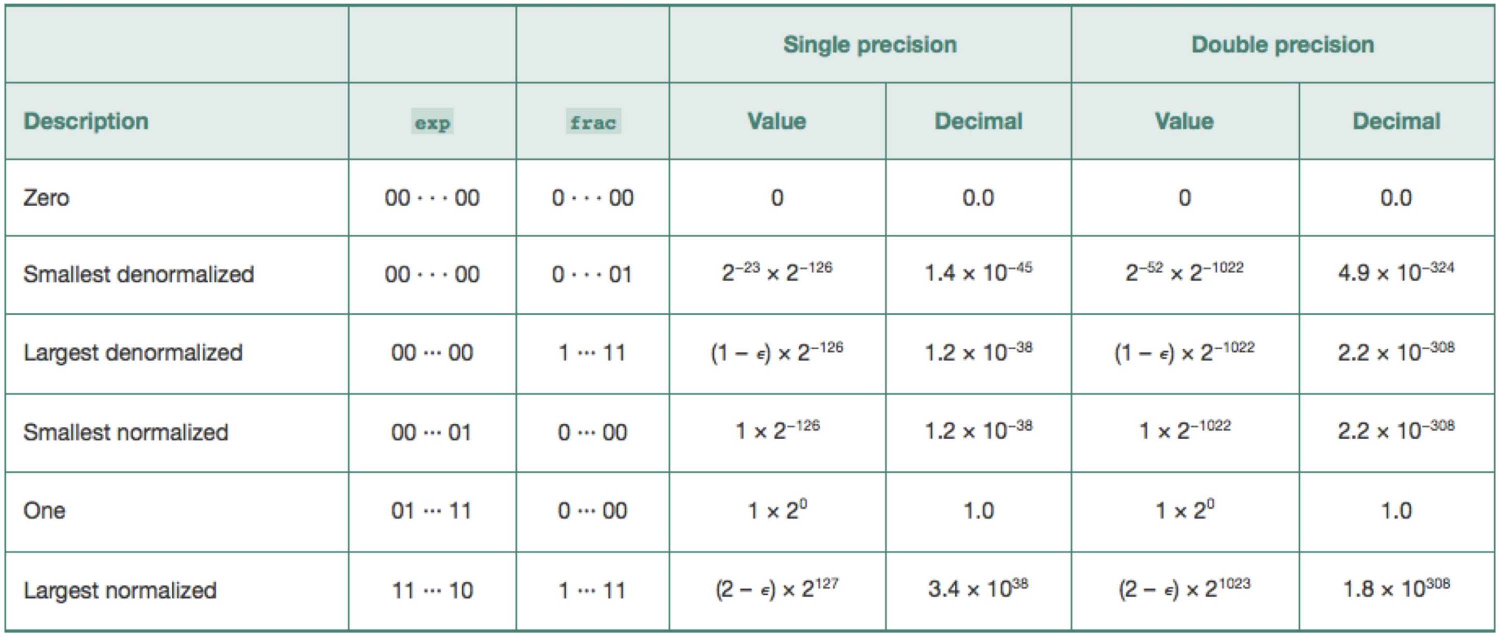
shows the set of values that can be represented in a hypothetical 6-bit format having *k*= 3 exponent bits and *n* = 2 fraction bits. The bias is $2^{3-1} - 1 = 3$. Part (a) of the figure shows all representable values (other than *NaN*). The two infinities are at the extreme ends. The normalized numbers with maximum magnitude are ±14. The denormalized numbers are clustered around 0. These can be seen more clearly in part (b) of the figure, where we show just the numbers between –1.0 and +1.0. The two zeros are special cases of denormalized numbers. Observe that the representable numbers are not uniformly distributed—they are denser nearer the origin.

Example Numbers



		Exponent			Fraction		Value		
Description	Bit representation	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	$\frac{1}{64}$	$\frac{2}{8}$	$\frac{2}{8}$	$\frac{2}{512}$	$\frac{2}{256}$	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	$\frac{6}{8}$	$\frac{15}{8}$	$\frac{1920}{8}$	240	240.0
Infinity	`0 1111 000`	—	—	—	—	—	—	∞	—

Example Numbers

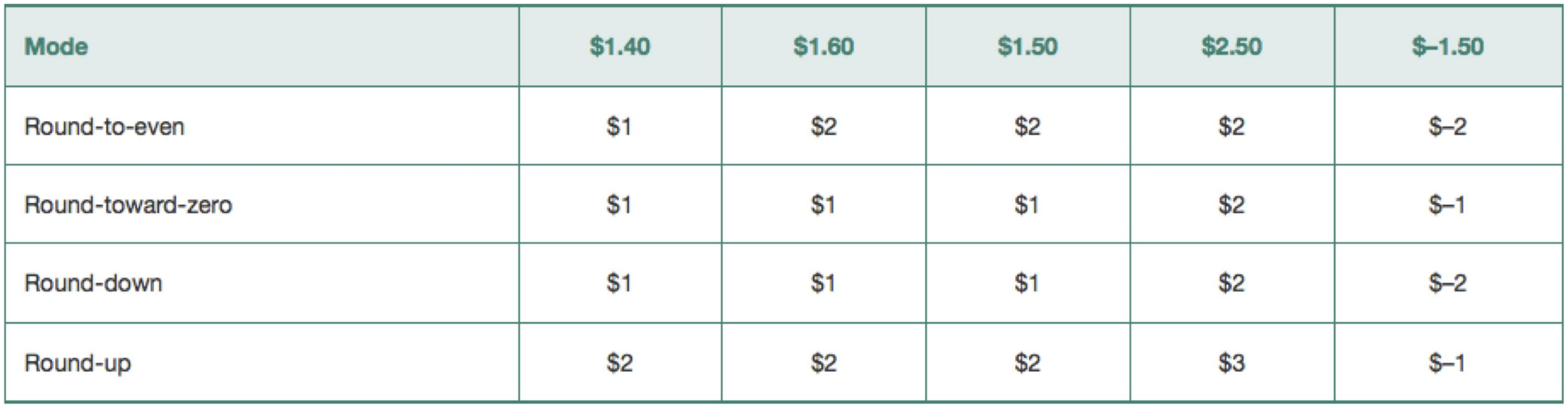


			Single precision		Double precision	
Description	`exp`	`frac`	Value	Decimal	Value	Decimal
Zero	00 · · · 00	0 · · · 00	0	0.0	0	0.0
Smallest denormalized	00 · · · 00	0 · · · 01	$2^{-23} \times 2^{-126}$	1.4×10^{-45}	$2^{-52} \times 2^{-1022}$	4.9×10^{-324}
Largest denormalized	00 ··· 00	1 ··· 11	$(1 - \epsilon)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

Rounding

- Floating-point arithmetic can only approximate real arithmetic, since the representation has limited range and precision. Thus, 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. This is the task of the rounding operation. One key problem is to define the direction to round a value that is halfway between two possibilities. For example, if I have $1.50 and want to round it to the nearest dollar, should the result be $1 or $2?
- An alternative approach is to maintain a lower and an upper bound on the actual number. For example, we could determine representable values $x-$ and $x+$ such that the value x is guaranteed to lie between them: $x- \le x \le x+$.
- The IEEE floating-point format defines four different rounding modes. The **default method finds a closest match**, while the other three can be used for computing upper and lower bounds.

Rounding



Mode	\$1.40	\$1.60	\$1.50	\$2.50	\$–1.50
Round-to-even	\$1	\$2	\$2	\$2	\$–2
Round-toward-zero	\$1	\$1	\$1	\$2	\$–1
Round-down	\$1	\$1	\$1	\$2	\$–2
Round-up	\$2	\$2	\$2	\$3	\$–1

Round-to-even

- Round-to-even at first seems like it has a rather arbitrary goal—why is there any reason to prefer even numbers? Why not consistently round values halfway between two representable values upward? The problem with such a convention is that one can easily imagine scenarios in which rounding a set of data values would then introduce a statistical bias into the computation of an average of the values. The average of a set of numbers that we rounded by this means would be slightly higher than the average of the numbers themselves. Conversely, if we always rounded numbers halfway between downward, the average of a set of rounded numbers would be slightly lower than the average of the numbers themselves. Rounding toward even numbers avoids this statistical bias in most real-life situations. It will round upward about 50% of the time and round downward about 50% of the time.

Round-to-even

- Round-to-even rounding can be applied even when we are not rounding to a whole number. We simply consider whether the least significant digit is even or odd. For example, suppose we want to round decimal numbers to the nearest hundredth. We would round 1.2349999 to 1.23 and 1.2350001 to 1.24, regardless of rounding mode, since they are not halfway between 1.23 and 1.24. On the other hand, we would round both 1.2350000 and 1.2450000 to 1.24, since 4 is even.

Rounding

- **Rounding Modes (illustrate 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

Floating Point in C

- When casting values between int, float, and double formats, the program changes the numeric values and the bit representations as follows (assuming data type int is 32 bits):
- From int to float, the number cannot overflow, but it may be rounded.
- From int or float to double, the exact numeric value can be preserved because double has both greater range (i.e., the range of representable values), as well as greater precision (i.e., the number of significant bits).
- 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. For example, 1.999 will be converted to 1, while −1.999 will be converted to −1. Furthermore, the value may overflow. The C standards do not specify a fixed result for this case. Intel-compatible microprocessors designate the bit pattern [10 *. . .* 00] (*TMinw* for word size *w*) as an *integer indefinite* value. Any conversion from floating point to integer that cannot assign a reasonable integer approximation yields this value. Thus, the expression (int) +1e10 yields -21483648, generating a negative value from a positive one.

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**
- **Not the same as real arithmetic**
  - Violates associativity/distributivity
  - Makes life difficult for compilers & serious numerical applications programmers

Interesting Numbers {single,double}

Description	*exp*	*frac*	*Numeric Value*
Zero	**00…00**	**00…00**	**0.0**
Smallest Pos. Denorm.	**00…00**	**00…01**	$2^{-\{23,52\}} \times 2^{-\{126,1022\}}$
▪ Single ≈ 1.4×10^{-45}			
▪ Double ≈ 4.9×10^{-324}			
Largest Denormalized	**00…00**	**11…11**	$(1.0 - \varepsilon) \times 2^{-\{126,1022\}}$
▪ Single ≈ 1.18×10^{-38}			
▪ Double ≈ 2.2×10^{-308}			
Smallest Pos. Normalized	**00…01**	**00…00**	$1.0 \times 2^{-\{126,1022\}}$
▪ Just larger than largest denormalized			
One	**01…11**	**00…00**	**1.0**
Largest Normalized	**11…10**	**11…11**	$(2.0 - \varepsilon) \times 2^{\{127,1023\}}$
▪ Single ≈ 3.4×10^{38}			
▪ Double ≈ 1.8×10^{308}			

The high cost of floating-point conversion to integers

- Converting large floating-point numbers to integers is a common source of programming errors. Such an error had disastrous consequences for the maiden voyage of the Ariane 5 rocket, on June 4, 1996. Just 37 seconds after liftoff, the rocket veered off its flight path, broke up, and exploded.
- Communication satellites valued at $500 million were on board the rocket. A later investigation showed that the computer controlling the inertial navigation system had sent invalid data to the computer controlling the engine nozzles. Instead of sending flight control information, it had sent a diagnostic bit pattern indicating that an overflow had occurred during the conversion of a 64-bit floating-point number to a 16-bit signed integer.
- The value that overflowed measured the horizontal velocity of the rocket, which could be more than five times higher than that achieved by the earlier Ariane 4 rocket. In the design of the Ariane 4 software, they had carefully analyzed the numeric values and determined that the horizontal velocity would never overflow a 16-bit number. Unfortunately, they simply reused this part of the software in the Ariane 5 without checking the assumptions on which it had been based.
- Floating-point arithmetic must be used very carefully, because it has only limited range and precision and because it does not obey common mathematical properties such as associativity.

The high cost of floating point imprecision

- The imprecision of floating-point arithmetic can have disastrous effects.
- 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. 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.