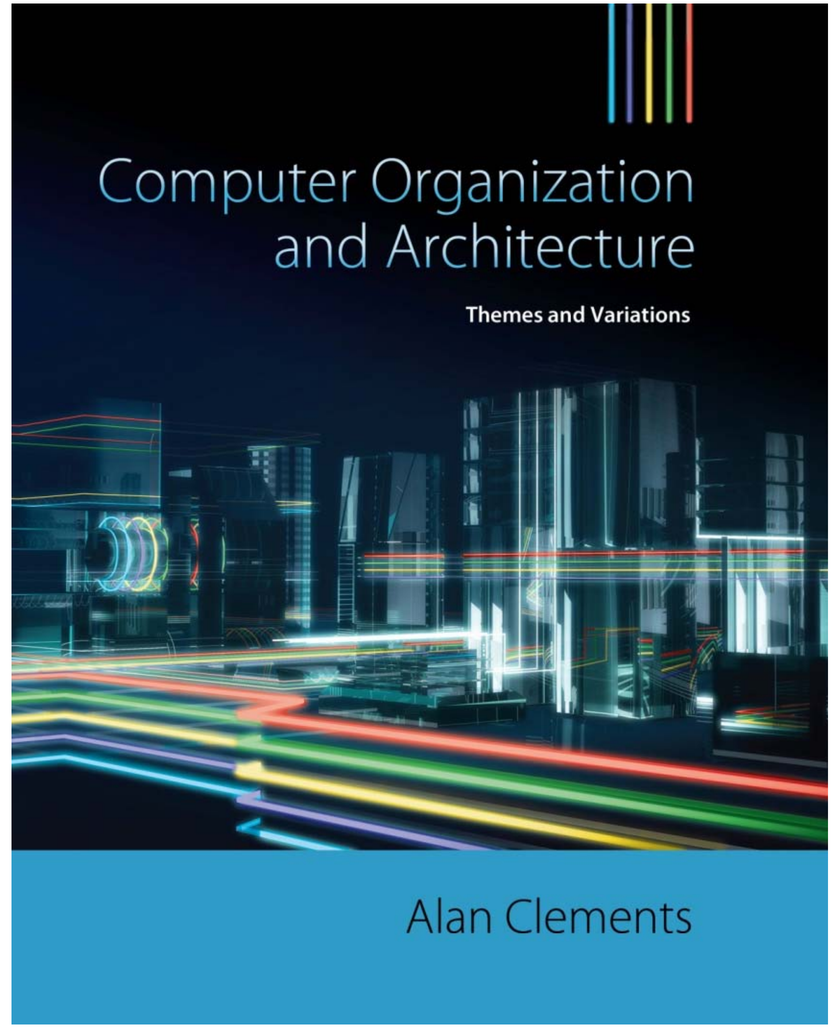
Part 3

CHAPTER 2

Computer Arithmetic and Digital Logic

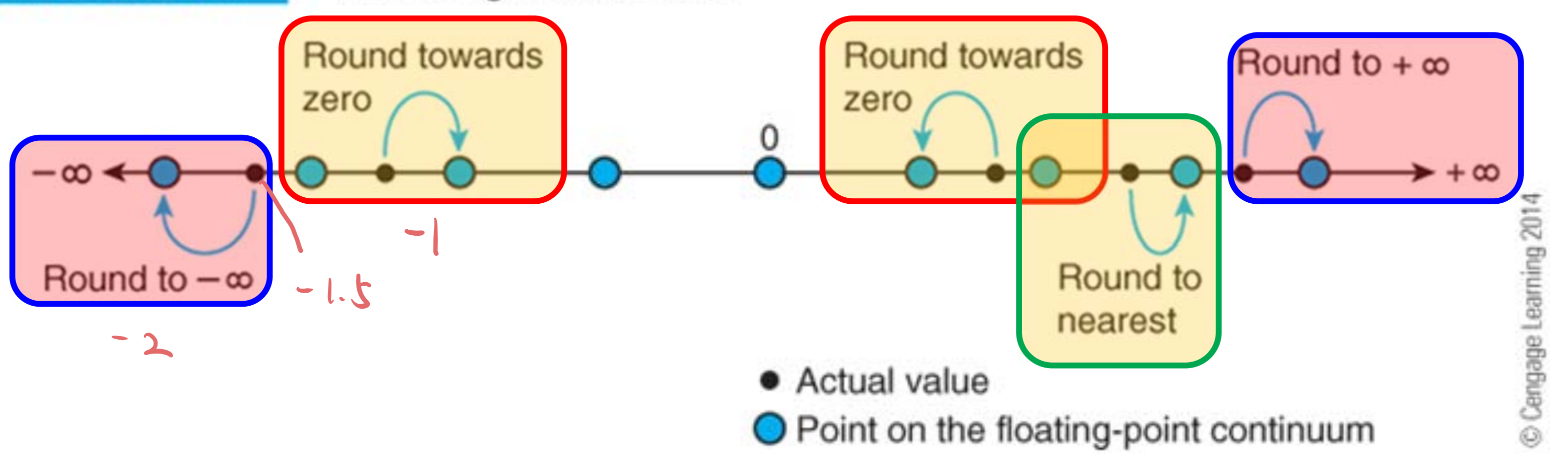


CENGAGE Learning

Rounding and Errors

- Floating-point arithmetic can lead to an increase in the number of bits in the fractional part
- To keep the number of fractional bits constant, rounding is needed
  - Error will be induced
- The rounding mechanisms include
  - *Truncation* (i.e., *dropping unwanted bits*) by *rounding towards zero*; a.k.a., *rounding down*
  - *Rounding towards positive or negative infinity*, the nearest valid floating-point number in the direction positive or negative infinity, respectively, is chosen to decide the rounding; a.k.a., *rounding up*.
  - *Rounding to nearest*, the closest floating-point representation to the actual number is used.

FIGURE 2.11 Rounding mechanisms



Rounding and Errors

- Decimal rounding examples:

  Rounding towards zero
  - +4.7 *truncation*, i.e., *rounded towards zero* ➔ **+4**
  - –4.7 *truncation*, i.e., *rounded towards zero* ➔ **–4**

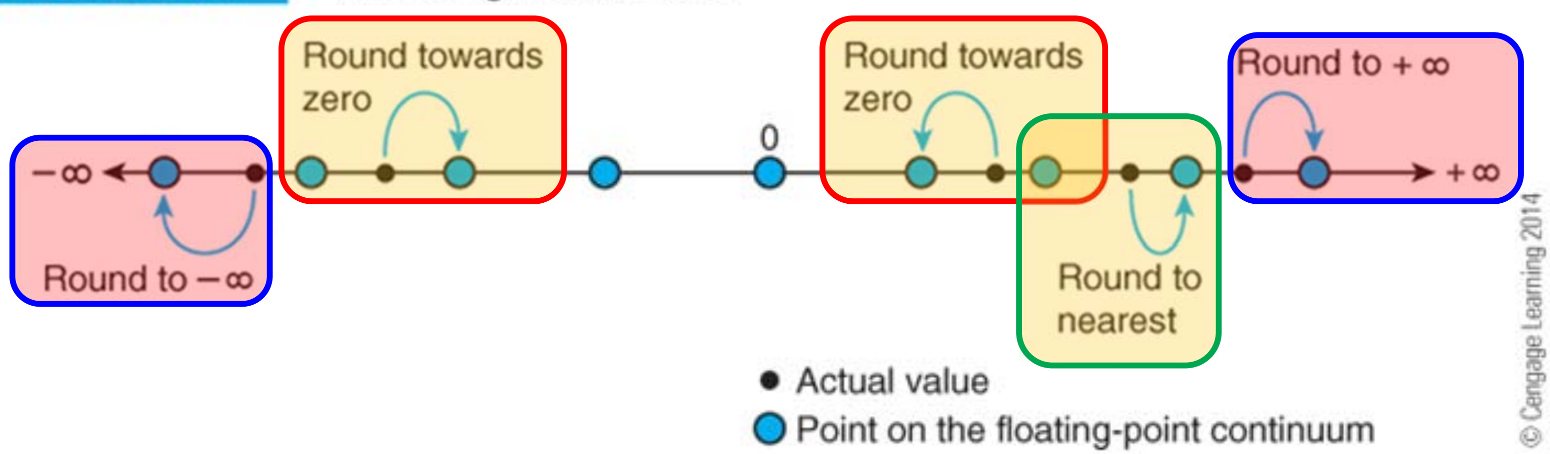
  In *truncation*, we just get rid of the extra digits (regardless the number is positive or negative). The end result is *rounding towards zero*.

  Rounding towards ± infinity (i.e., *rounding up*)

  It is the opposite of *rounding towards zero*
  - +4.7 *rounded towards ± infinity* ➔ **+5**
  - –4.7 *rounded towards ± infinity* ➔ **–5**

  Rounding to nearest
  - +4.7 *rounded to nearest* ➔ **+5**
  - –4.7 *rounded to nearest* ➔ **–5**
  - +4.3 *rounded to nearest* ➔ **+4**
  - –4.3 *rounded to nearest* ➔ **–4**

FIGURE 2.11 Rounding mechanisms



科学计数法 Normalization

- A number is normalized when it is written in *scientific notation* with a ***single non-zero decimal digit before the decimal point*** (i.e., ***the integer part consists of a single non-zero digit***).

Example 1:

- The number 123.456_{10} is not normalized, as the integer part is not a single non-zero digit.
- To normalize it, you need to move the decimal point two position to the left and to compensate this move by multiplying the number by 100, i.e.,
  - ✓ $1.23456_{10} \times 10^{2}$

Example 2:

- The number 0.00123_{10} is not normalized, as the integer part is not a single non-zero digit.
- To normalize it, you need to move the decimal point three position to the right and to compensate this move by dividing the number by 1000, i.e.,
  - ✓ $1.23_{10} \times 10^{-3}$

- In **base b**, a **normalized number** will have the form $\pm \mathbf{b_0 . b_1 b_2 b_3 \ldots \times b^n}$ where $\mathbf{b_0 \neq 0}$, and $\mathbf{b_0, b_1, b_2, b_3} \ldots$ are integers between **0** and **b -1**

Floating-point Numbers

- Floating-point arithmetic lets you handle the very large and very small numbers found in scientific applications.
- Floating-point is also called *scientific notation*, because scientists use it to represent large numbers (e.g., 1.2345×10^{20}) and small numbers that are very close to zero, but not zero (e.g., $0.45679999 \times 10^{-50}$).
- A floating-point value is encoded as ***two*** components: *a number* and the *location of the radix point* within the number.
- A binary floating-point number is represented by

$$\textit{mantissa} \times 2^{\textit{exponent}}$$

  - for example, 101010.111110_2 can be represented by $1.01010111110_2 \times 2^5$, where
    - the ***significan**t **d**igits* (or simply *significand*) is 1.01010111110 and
    - the *exponent* is 5 (00000101_2 in 8-bit binary arithmetic).
- The term *mantissa* has been replaced by *significand* to indicate the number of *significant bits* in a floating-point number.
- Because a floating-point number is defined as the *product* of *two values*, a floating-point value is not unique; for example $10.110_2 \times 2^4 = 1.011_2 \times 2^5$.

Normalization of Floating-point Numbers

- In the [*IEEE-754 Standard for Floating-Point Arithmetic*], the ***significand*** term is *always normalized* (***unless*** *it represents a zero* or *underflow*)
- A *normalized* binary ***significand*** always has a leading ***1*** (i.e., ***1*** in the MSB)
- The *normalized* absolute ***non-zero*** value of the ***IEEE-754*** FP numbers are always in *the range*

The book is missing the –ve sign here

The minimum absolute value

$$[1.000\ldots0_2 \times 2^{-e} \text{ to } 1.111\ldots1_2 \times 2^{e}]$$

The maximum absolute value

- The *floating-point* normalization leads to the highest available ***precision***, as all significant bits are utilized.
  - the un-normalized 8-bit significand 0.0000101 has only three significant bits, whereas
  - the normalized 8-bit significand 1.0100011 has eight significant bits.

three not four

  - If a floating-point calculation is to yield the value $0.110\ldots_2 \times 2^e$, the result would be normalized to give $1.10\ldots_2 \times 2^{e-1}$.
  - Similarly, the result $10.1\ldots_2 \times 2^e$ would be normalized to $1.01\ldots_2 \times 2^{e+1}$.

Significand and Exponent Encoding

- The *significand* of an ***IEEE-754*** floating-point number is *represented in sign and magnitude* form.

- The *exponent* is *represented in a biased* form, by *adding a constant* to the *true exponent*.

- Suppose an 8-bit exponent is used and all exponents are biased by 127.
  - If the *true exponent* is 0, it will be encoded as 0 + 127 = 127.
  - If the *true exponent* is –2, it will be encoded as –2 + 127 = 125.
  - If the *true exponent* is +2, it will be encoded as +2 + 127 = 129.

- A real number such as 1010.1111 is normalized to get $+1.0101111 \times 2^3$.

  - The *true exponent* is +3, which is encoded as a *biased exponent* of 3 + 127; that is 130_{10} or 10000010_2 in binary form.

- Likewise, if a *biased exponent* is 130_{10}, the *true exponent* is 130 – 127 = 3

Significand and Exponent Encoding

- A 32-bit single-precision ***IEEE-754*** floating-point number is represented by the bit sequence

  S EEEEEEEE 1.FFFFFFFFFFFFFFFFFFFFFFF

  - ***S*** is the *sign bit*,
    - 0 means positive significand,
    - 1 means negative significand
  - ***E*** is an eight-bit *biased exponent* that tells you how to shift the binary point, and
  - ***F*** is a 23-bit ***fractional*** *significand*.
  - ***The leading 1 in front of the significand is omitted when the number is encoded.***

1=E-127.

sign bit

In this case, B is 127, i.e., excess-127 code

- A floating-point number X is defined as:

E=0

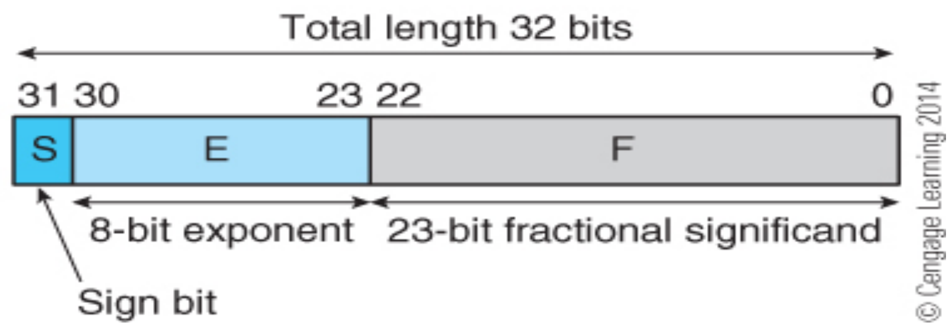
E=255.

$$1 \le E \le 254 \;\leftrightarrow\; X_{10} = (-1)^S \times \boxed{\underbrace{2^{(E-B)}}_{=>\,2^k}} \times 1.F$$

$1.F_2 \times 2^k$

When 1 ≤ *E* ≤ 254, the *significand* = 1 + the fractional significand *F*

FIGURE 2.7 Structure of a 32-bit IEEE floating-point number



Significand and Exponent Encoding

- If the exponent *EEEEEEEE* > *0*, the *significand* of an ***IEEE-754*** floating-point number is *normalized* in the range *1.0000…00* to *1.1111…11*,
- If the exponent *EEEEEEEE* = *0*, the *significand* is represented ***without** normalization*.

Used when it is ***impossible*** to normalize the number.

  - In such case, the floating-point number *X* is defined as:

S 00000000 0.FFFFFFFFFFFFFFFFFFFFFFF

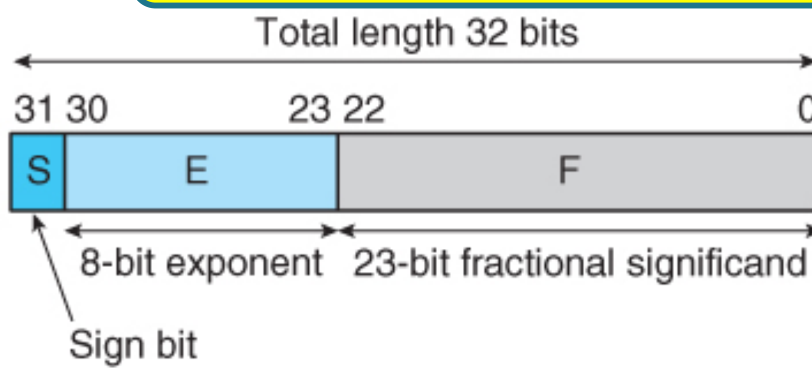
$$E = 0 \leftrightarrow X = (-1)^{S} \times 2^{(0-(B-1))} \times 0.F$$

When E = 0, the *significand* = the fractional significand *F*

In this case, B – 1 is 126, i.e., excess-126 code

where,

- *S* is the sign bit,
  - 0 means positive significand,
  - 1 means negative significand
- *E = 0*
  - the exponent was biased by ***B – 1***
- *F* is the fractional significand
  - As *E = 0*, the significand was encoded ***without** normalization*, i.e., 0.F *without* an *implicit leading one*



- When *E = 0*, *F ≠ 0* ➔ ± ***Denormalized underflow number***

Significand and Exponent Encoding

- The floating-point value of ***zero*** is represented by

$$0.00...00 \times 2^{\text{most negative exponent}}$$

E=0 F=0.

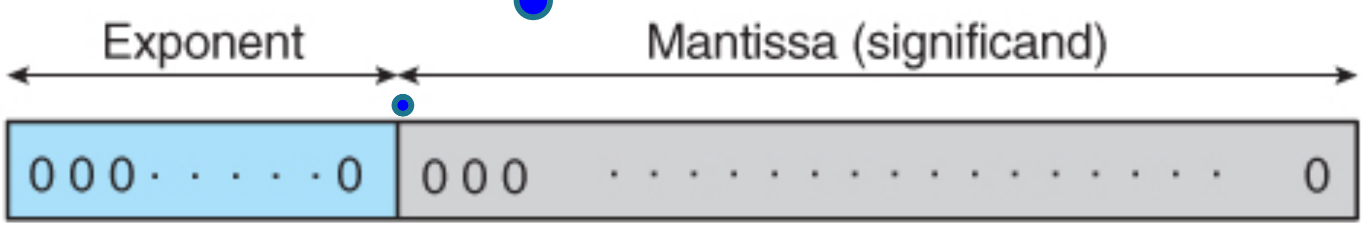
i.e., the ***zero*** is represented by
  - a ***zero*** *significand* and
  - a ***zero*** *exponent*

as Figure 2.6 demonstrates.

In this floating-point representation,
how many zeros do we have?

2 : ±0

FIGURE 2.6 Representing the value zero in floating-point arithmetic



Significand and Exponent Encoding

TABLE 2.7 IEEE Floating-Point Formats

float type in Java and C

double type in Java and C

	Single Precision	Double Precision (Single Extended)
Field width in bits		
S = sign	1	1
E = exponent	8	11
L = leading bit	1 (not stored)	1 (not stored)
F = fraction	23	52
Total width	32	64
Exponent		
Maximum E	255	2047
Minimum E	0	0
Bias	127	1023
E_{max}	127	1023
E_{min}	−126	−1022

Biased values

Unbiased values

2''

The L value =1, if and only if E ≠ 0
The L value =0, if and only if E = 0

If E ≠ 0, True exponent = ***biased exponent – bias***

If E = 0, True exponent = ***0 – (bias – 1)***

1-bias

The book flipped the meaning of S. It is S=0 for +ve and =1 for –ve.

S = sign bit (0 for a negative number, 1 for a positive number)
L = leading bit (always 1 in a normalized, non-zero significand)
F = fractional part of the significand
The range of exponents is from the minimum E + 1 to the maximum E – 1
The number is represented by $-1^S \times 2^{E - \text{exponent}} \times L.F$
Zero is represented by the minimum exponent, L = 0, and F = 0
The maximum exponent, E_{max} + 1 represents signed infinity

1 ➔ 254 for **NORMALIZED** numbers

When E = 255

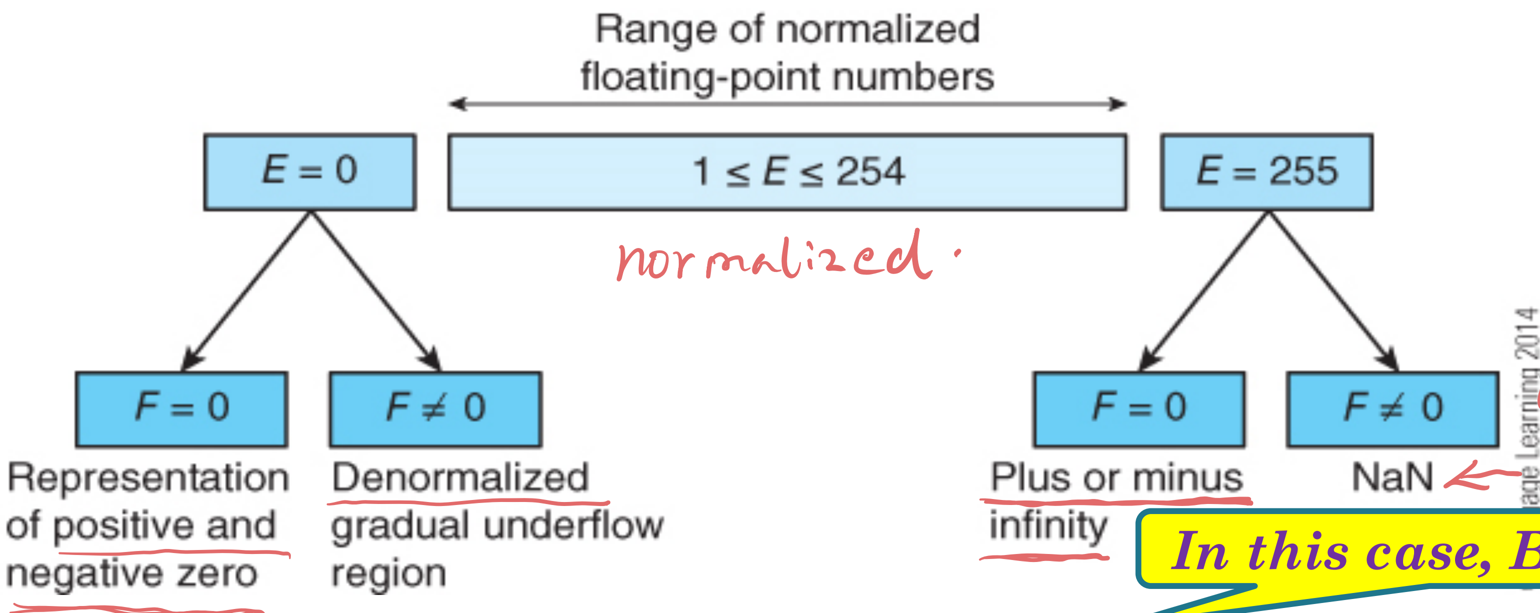
In the IEEE single precision representation,
the ***largest*** ***normalized*** absolute number is
$2^{+127} \times 1.111\ldots1_2 \approx 2^{+128} = 10^{+38.5318394} \approx 3.4\times10^{+38}$.

the ***smallest*** ***normalized*** absolute number is
$2^{-126} \times 1.000\ldots0_2 = 2^{-126} = 10^{-37.9297794} \approx 1.17\times10^{-38}$.

This slide is modified from the original slid

Significand and Exponent Encoding

FIGURE 2.8 IEEE floating-point number space for a single-precision number



In this case, B – 1 is 126

$$E = 0 \Leftrightarrow X = (-1)^S \times 2^{(0-(B-1))} \times 0.F$$

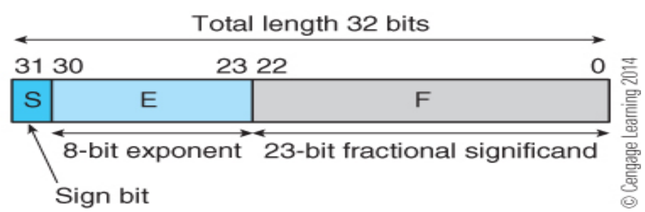
- *Underflow* occurs when the result of a calculation is a smaller number (in magnitude) than the smallest value representable as a *normalized* floating point number in the target data type.
- Replacing an *underflow* case by a **zero** might be *ok* from the *addition* point of view, but it is *not ok* from the *multiplication* point of view.
- **NaN** means *Not a Number*, e.g., $0 \div 0$, $\infty \div \infty$, $0 \times \infty$, or $\infty - \infty$
- In **NaN**, the value of F is ignored by applications.

From Binary to 32-bit IEEE-754 FP

- ***Example 1(a):***
  Convert $-1111000011100.0011110000111_2$ into a *32-bit single-precision IEEE-754 FP* value.
  - The number is negative ➔ ***S = 1***
  - The *significand* is $11110000111100.00111100001111_2$
  - The normalized *significand* is
    $1.11100001111000011110000111 1_2 \times \boxed{2^{13}}$ true exponent part
  - To encode the ***F*** value, we will *ignore* the leading 1 and we will only consider the first 23 bits after the binary point, i.e., $11100001111000011110000\boxed{1111}_2$
  - The ignored part of the *significand* is *rounded to the nearest*, hence the value of ***F*** = $11100001111000011110001_2$
  - The *biased exponent* is the *true exponent* plus 127; that is, $13 + 127 = 140_{10} = 1000\ 1100_2$
    Hence, ***E*** = $1000\ 1100_2$
  - The final number is $1100\ 0110\ 0111\ 0000\ 1111\ 0000\ 1111\ 0001_2$, or $C670F0F1_{16}$.

0 = 0000
1 = 0001
2 = 0010
3 = 0011
4 = 0100
5 = 0101
6 = 0110
7 = 0111
8 = 1000
9 = 1001
A = 1010
B = 1011
C = 1100
D = 1101
E = 1110
F = 1111

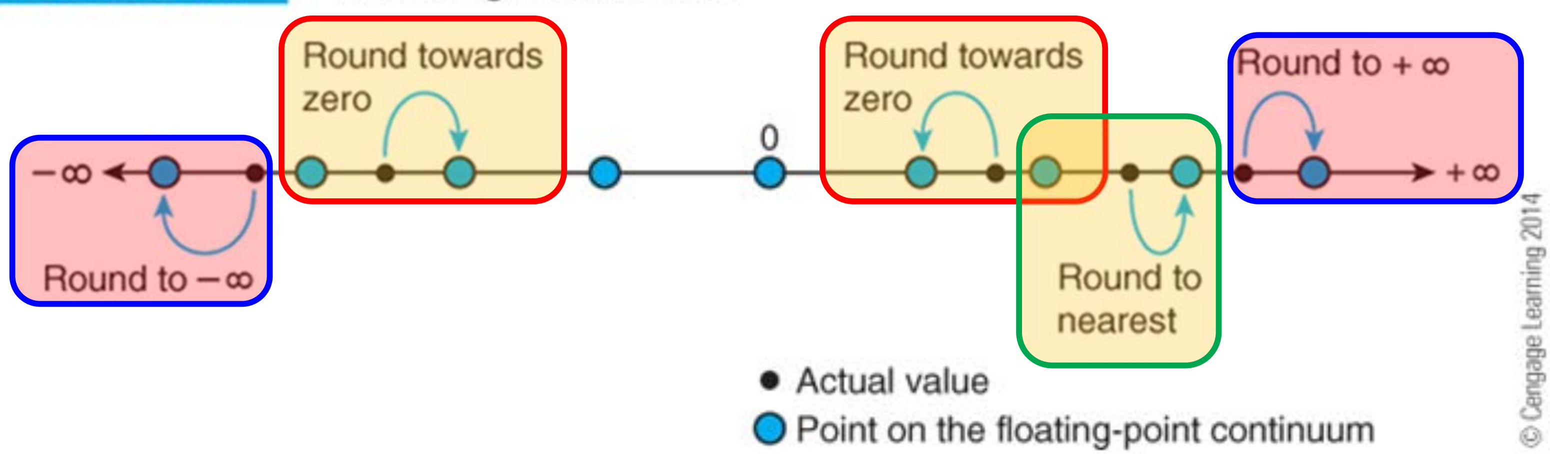
FIGURE 2.7 Structure of a 32-bit IEEE floating-point number



Rounding and Errors

- *When the number to be rounded is midway between two points on the floating-point line, IEEE rounding to the nearest selects the value whose least-significant digit is zero (i.e., rounding to an even binary significand).*
  - *For example:*
    - $0.1110000111100001111000{\color{red}0}{\color{gray}1000}_2$ will be rounded to $0.1110000111100001111000{\color{red}0}_2$
    - $0.1110000111100001111000{\color{red}1}{\color{gray}1000}_2$ will be rounded to $0.111000011110000111100{\color{red}10}_2$

FIGURE 2.11 Rounding mechanisms



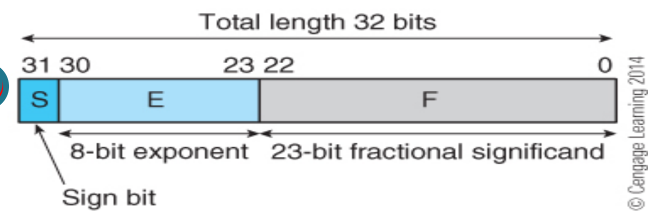
From 32-bit IEEE-754 FP to Binary

- ***Example 1(b):*** *Convert* $C670F0F1_{16}$ from a *32-bit single-precision IEEE-754 FP* value into a binary value
  ---This is the same value as in example 1(a)
  - Unpack the number into *sign bit*, *biased exponent*, and *fractional significand*:

    $C670F0F1_{16} \rightarrow 1100\ 0110\ 0111\ 0000\ 1111\ 0000\ 1111\ 0001_2$
    - S = 1
    - E = 100 0110 0
    - F =111 0000 1111 0000 1111 0001
  - As the sign bit is 1, the number is negative.
  - Subtract $\boxed{127}$ from the *biased exponent* $100\ 0110\ 0_2$ to get the *true exponent* ➔ $1000\ 1100_2 - \underbrace{0111\ 1111_2}_{127_{10}} = 0000\ 1101_2 = 13_{10}$.
  - The fractional significand is $.111\ 0000\ 1111\ 0000\ 1111\ 0001_2$.
  - Reinserting the *leading one* gives $1.111\ 0000\ 1111\ 0000\ 1111\ 0001_2$.
  - The number is $-1.111\ 0000\ 1111\ 0000\ 1111\ 0001_2 \times 2^{13}$
    $= -1111\ 0000\ 1111\ 00.00\ 1111\ 0001_2$

Note that the correct answer is:
$-1111\ 0000\ 1111\ 00.00\ 1111\ 0001_2$ *not*
$-1111\ 0000\ 1111\ 00.00\ 1111\ 00001111_2$
This is due to the rounding error.

FIGURE 2.7 Structure of a 32-bit IEEE floating-point number



From 32-bit IEEE-754 FP to Decimal

- ***Example 2:*** *Convert* $1111\ 1110\ 0110\ 0000\ 0000\ 0000\ 0000\ 0000_2$ from a *32-bit single-precision IEEE-754 FP* value into a decimal value.

  -1

  0111 1101

  - Unpack the number into *sign bit*, *biased exponent*, and *fractional significand*.

    $\boxed{-1.11_2 \times 2^{125}}$

    - S = 1
    - E = 1111 1100
    - F =110 0000 0000 0000 0000 0000
  - As the sign bit is 1, the number is negative.
  - Subtract 127 from the *biased exponent* $1111\ 1100_2$ to get the *true exponent* ➔ $1111\ 1100_2 - 0111\ 1111_2 = 0111\ 1101_2 = 125_{10}$.
  - The fractional significand is $.110\ 0000\ 0000\ 0000\ 0000\ 0000_2$.
  - Reinserting the *leading one* gives $1.110\ 0000\ 0000\ 0000\ 0000\ 0000_2$.
  - The number is $-1.11_2 \times 2^{125} = -1.75_{10} \times 2^{125}$

$2^{125} = 10^z$ ➔ $\log_{10}(2^{125}) = z$ ➔ $z = 125 \times 0.30103 = 37.62875$

$2^{125} = 10^{37.62875} = 10^{37} \times 10^{0.62875} = 10^{37} \times 4.253535$

$-1.75 \times 2^{125} = -1.75 \times 10^{37} \times 4.253535 = -7.44368625 \times 10^{37}$

From 32-bit IEEE-754 FP to Decimal

-2 2^{-126} 1.11

- ***Example 3:*** *Convert* 1000 0000 0110 0000 0000 0000 0000 0000_2 from a *32-bit single-precision IEEE-754 FP* value into a decimal value.
  - Unpack the number into *sign bit*, *biased exponent*, and *fractional significand*.

    $-0.11_2 \times 2^{-126} \Rightarrow 0.75_{10} \times 2^{-126}$.

    - S = 1
    - E = 0000 0000
    - F =110 0000 0000 0000 0000 0000
  - As the sign bit is 1, the number is negative.
  - As E = 0 ➔ *true exponent = 0 – (127 – 1) = –126*
  - The fractional significand is .110 0000 0000 0000 0000 0000_2.
  - As E = 0, the fractional significand is *not normalized*.
  - As E = 0 and F ≠ 0, it means that this is an *underflow* case.
  - The number is $-0.11_2 \times 2^{-126} = -0.75 \times 2^{-126}$

The L value =0, as E = 0

$2^{-126} = 10^z$ ➔ $\log_{10}(2^{-126}) = z$ ➔ $z = -126 \times 0.30103 = -37.92978$

$2^{-126} = 10^{-37.92978} = 10^{-37} \times 10^{-0.92978} = 10^{-37} \times 0.11755$

$-0.75 \times 2^{-126} = -0.75 \times 10^{-37} \times 0.11755 = -0.0881625 \times 10^{-37}$

$= -8.81625 \times 10^{-39}$ < the ***smallest normalized*** value (1.17×10^{-38})

From 32-bit IEEE-754 FP to Decimal

127+127+1.

- ***Example 4:*** *Convert* 0111 1111 1000 0000 0000 0000 0000 0000_2 from a *32-bit single-precision IEEE-754 FP* value into a decimal value.
  - Unpack the number into *sign bit*, *biased exponent*, and *fractional significand*.
    - S = 0
    - E = 1111 1111 $2^{8}+2^{7}+2$ => +∞
    - F =000 0000 0000 0000 0000 0000
  - As the sign bit is 0, the number is positive.
  - As E = 255 ➔ either an infinity case or a NaN case
  - The fractional significand is .000 0000 0000 0000 0000 0000_2.
  - As the *bias exponent* is 255 and the F value is ***zero***, it means that this is an **+infinity** case, e.g., a number that is larger than ***3.4028235 ×10^+38^***

if significand is not 0,

then it will be NaN.

From 32-bit IEEE-754 FP to Decimal

- ***Example 5:*** *Convert* 1111 1111 1110 0000 0000 0000 0000 0000_2 from a *32-bit single-precision IEEE-754 FP* value into a decimal value.
  - Unpack the number into *sign bit*, *biased exponent*, and *fractional significand*.
    - S = 1
    - E = 1111 1111
    - F =110 0000 0000 0000 0000 0000

-NaN.

  - As the sign bit is 1, the number is negative.
  - As E = 255 ➔ either an infinity case or a NaN case
  - The fractional significand is .110 0000 0000 0000 0000 0000_2.
  - As the *bias exponent* is 255 and the F value is ***NOT zero***, it means that this is a **NaN** case (*Not a Number*), e.g., the result of a ***0*** ÷ ***0***, ∞ ÷ ∞, ***0*** **×** ∞, or ∞ **—** ∞ operation.
  - In **NaN** cases, the value of F is ignored.
  - The value **–NaN**

From 32-bit IEEE-754 FP to Decimal

- ***Example 6:*** *Convert* C46C0000$_{16}$ from *32-bit single-precision IEEE-754 FP* value into a decimal value.

13

  - Convert the hexadecimal number into binary form
    C46C0000$_{16}$ = 1100 0100 0110 1100 0000 0000 0000 0000$_2$.

    C 4 6 C

  - Unpack the number into *sign bit*, *biased exponent*, and *fractional significand*.
    - S = 1
    - E = 1000 1000
    - F =110 1100 0000 0000 0000 0000

-1.

100 01000
- 011 11111
0001001

2^9.

0.110 1100$_2$ =

=> -1.1101100$_2$ × 2^9.

it is 9
not 7

  - As the sign bit is 1, the number is negative.
  - We subtract 127 from the *biased exponent* 1000 1000$_2$ to get the *true exponent* ➔ 1000 1000$_2$ – 0111 1111$_2$ = 0000 1001$_2$ = 9$_{10}$.
  - The fractional significand is .110 1100 0000 0000 0000 0000$_2$.
  - Reinserting the leading one gives 1.110 1100 0000 0000 0000 0000$_2$.
  - The number is –1.110 1100 0000 0000 0000 0000$_2$ × 2^9,
    or –1110 1100 00.00 0000 0000 0000$_2$ (i.e., –944.0$_{10}$).

all numbers

0 = 0000
1 = 0001
2 = 0010
3 = 0011
4 = 0100
5 = 0101
6 = 0110
7 = 0111
8 = 1000
9 = 1001
A = 1010
B = 1011
C = 1100
D = 1101
E = 1110
F = 1111

From Binary to 32-bit IEEE-754 FP

-129
+126
-2.

1.0000 0000 0000 0000 0000 0011 1 × 2 -129.

- ***Example 7:*** *Convert* 0.0000 1000 0000 0000 0000 0000 0001 $11_2 \times 2^{-124}$ into a *32-bit single-precision IEEE-754 FP* value.
  - The number is positive ➔ ***S = 0***
  - The *fractional* part is 0.0000 1000 0000 0000 0000 0000 0001 11_2
    The normalized *fractional* part is
    1.000 0000 0000 0000 0000 0001 $11_2 \times 2^{-5}$
  - Hence the number will be 1.000 0000 0000 0000 0000 0001 $11_2 \times 2^{-129}$
  - As the *exponent* is less than –126, the *fractional* part ***can NOT be*** represented as a ***normalized*** number (the number is ***too small***)
  - Instead, we will attempt to represent it as an ***un-normalized*** underflow number with *exponent* = –126
  - The number = 0.001 0000 0000 0000 0000 0000 0011 $1_2 \times 2^{-126}$ rounded to 0.
  - The encoded ***F*** value (23 bits) will be ***001 0000 0000 0000 0000 0000***
  - As ***F*** is *un-normalized*, the *biased exponent* will be the *true exponent* plus 127 – 1; normalized => 127.
    that is, –126 + 127 – 1 = 0; Hence, ***E*** = 0000 0000_2
  - The final number is 0 $\boxed{000\ 0000\ 0}$001 0000 0000 0000 0000 0000_2, or 00100000_{16}.

rounded

59

From Binary to 32-bit IEEE-754 FP

$1.11_2 \times 2^{-152}$. 0/0000 0000/

- ***Example 8:*** *Convert* 0.0000 0000 0000 0000 0000 0000 0001 $11_2 \times 2^{-124}$ into a *32-bit single-precision IEEE-754 FP* value.
  - The number is positive ➔ ***S = 0*** $-152+127-1=-26$.
  - The *fractional* part is 0.0000 0000 0000 0000 0000 0000 0001 11_2
    The normalized *fractional* part is $1.11_2 \times 2^{-28}$
  - Hence the number will be $1.11_2 \times 2^{-152}$
  - As the *exponent* is less than –126, the *fractional* part ***can NOT be*** represented as a ***normalized*** number (the number is ***too small***)
  - Instead, we will attempt to represent it as an ***un-normalized*** *underflow number* with *exponent* = –126
  - The number = 0.000 0000 0000 0000 0000 0000 0011 $1_2 \times 2^{-126}$ rounded
  - The encoded ***F*** value (23 bits) will be ***000 0000 0000 0000 0000 0000***
  - As ***F*** is *un-normalized* the *biased exponent* will be the *true exponent* plus 127 – 1; that is, –126 + 127 – 1 = 0; Hence, ***E*** = $0000\ 0000_2$
  - The final number is 0000 0000 0000 0000 0000 0000 0000 0000_2, or 00000000_{16}.
    I.e., the number is encoded as ***ZERO***

From Binary to 32-bit IEEE-754 FP

|-126|.

- **Example 9:** *Convert* 0.0000 0000 0000 0000 0000 0000 0111 11$_2$ × 2^{-124} into a *32-bit single-precision IEEE-754 FP* value.
  - The number is positive ➔ ***S* = 0**

    $1.1111_2 \times 2^{-150}$ 0.0000 =>
  - The *fractional* part is 0.0000 0000 0000 0000 0000 0000 0111 11$_2$
    The normalized *fractional* part is $1.1111_2 \times 2^{-26}$

    1.0000 0000 0000 0000 . . . 1.
  - Hence the number will be $1.1111_2 \times 2^{-150}$
  - As the *exponent* is less than –126, the *fractional* part ***can NOT be*** represented as a ***normalized*** number (the number is ***too small***)
  - Instead, we will attempt to represent it as an ***un-normalized*** *underflow number* with *exponent* = –126
  - The number = 0.000 0000 0000 0000 0000 0000 1111 1$_2$ × 2^{-126} rounded
  - The encoded ***F*** value (23 bits) will be ***000 0000 0000 0000 0000 0001***
  - As ***F*** is *un-normalized* the *biased exponent* will be the *true exponent* plus 127 – 1;
    that is, –126 + 127 – 1 = 0; Hence, ***E*** = 0000 0000$_2$
  - The final number is 0000 0000 0000 0000 0000 0000 0000 0001$_2$, or 00000001$_{16}$ ---the ***smallest non-zero absolute*** *un-normalized underflow number* ***(1.4012985×10^{-45})***

From Binary to 32-bit IEEE-754 FP

- ***Example 10:*** *Convert* 1111.1111 1111 1111 1111 1111 011$_2$ × 2^{124} into a *32-bit single-precision IEEE-754 FP* value.
  - The number is positive ➔ ***S = 0***
  - The *fractional* part is 1111.1111 1111 1111 1111 1111 011$_2$
    The normalized *fractional* part is
    1.111 1111 1111 1111 1111 1111 011$_2$ × 2^3
  - Hence the number will be 1.111 1111 1111 1111 1111 1111 011$_2$ × 2^{127}
  - To encode the ***F*** value, we will ignore the leading 1 and we will only consider the first 23 bits after the binary point, i.e., ***111 1111 1111 1111 1111 1111***… rounded
  - The *biased exponent* is the *true exponent* plus 127; that is, 127 + 127 = 254; Hence, ***E*** = 1111 1110$_2$
  - The final number is 0111 1111 0111 1111 1111 1111 1111 1111$_2$, or 7F7FFFFF$_{16}$.
    1111 1111 → infinity / NaN.
  - This number is the ***largest absolute*** *normalized number* *(3.4028235×10^{+38})*

From Binary to 32-bit IEEE-754 FP

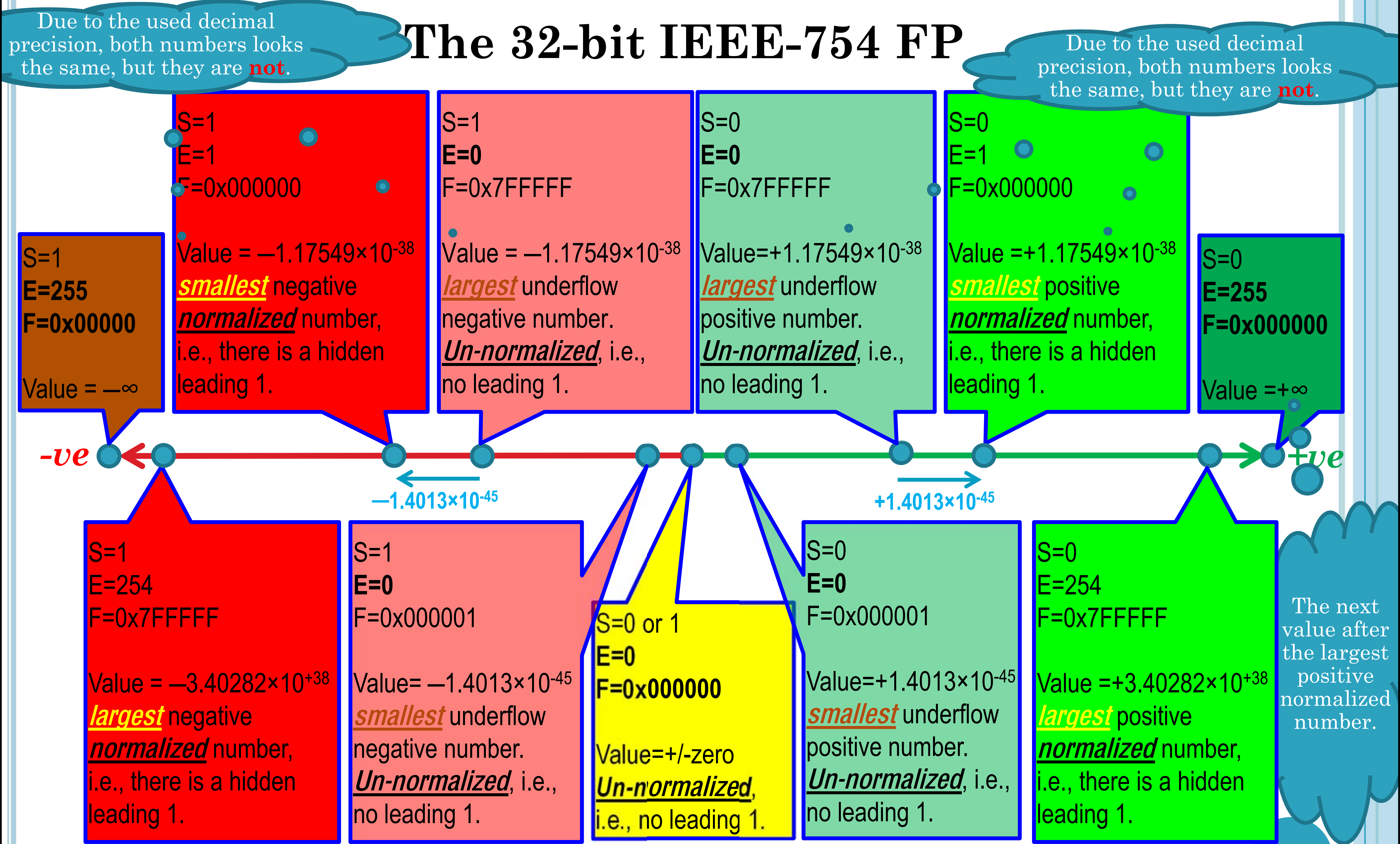
- ***Example 11:*** *Convert* $1111.1111\,1111\,1111\,1111\,1111\,111_2 \times 2^{124}$ into a *32-bit single-precision IEEE-754 FP* value.
  - The number is positive ➔ ***S = 0***
  - The *fractional* part is $1111.1111\,1111\,1111\,1111\,1111\,111_2$
    The normalized *fractional* part is
    $1.111\,1111\,1111\,1111\,1111\,1111\,111_2 \times 2^3$
  - Hence the number will be $1.111\,1111\,1111\,1111\,1111\,1111\,111_2 \times 2^{127}$
  - To encode the ***F*** value, we will only consider the first 23 bits after the binary point
  - Note that, the rounding here will add 1 to the fraction to make it $10.000\,0000\,0000\,0000\,0000\,0000_2 \times 2^{127}$
  - As a result of this, the number needs to renormalized again $1.0000\,0000\,0000\,0000\,0000\,0000_2 \times 2^{128}$
  - The true exponent of the normalized number is > 127, hence the number will be encoded as **+infinity**, i.e.,
    - the ***F*** value will be ***000 0000 0000 0000 0000 0000***
    - the ***E*** value will be $1111\,1111_2$
  - The final number is $0111\,1111\,1000\,0000\,0000\,0000\,0000\,0000_2$, i.e., **+infinity** ($7F800000_{16}$)

From Decimal to 32-bit IEEE-754 FP

- ***Example 12:*** *Convert* 4100.125_{10} into a *32-bit single-precision IEEE-754 FP* value.
  - Convert 4100.125_{10} into a fixed-point binary
    - $4100_{10} = 1\ 0000\ 0000\ 0100_2$ and
    - $0.125_{10} = 0.001_2$.
    - Therefore, $4100.125_{10} = 1000\ 0000\ 0010\ 0.001_2$.
  - Normalize $1000\ 0000\ 0010\ 0.001_2$ to $1.000\ 0000\ 0010\ 0001_2 \times 2^{12}$.
  - The sign bit, S, is 0 because the number is positive
  - The *biased exponent* is the *true exponent* plus 127; that is, $12_{10} + 127_{10} = 139_{10} = 1000\ 1011_2$
  - The fractional significand is 000 0000 0010 0001 **0000 0000**
    - *the leading 1 is stripped* and
    - *the significand is **expanded** to 23 bits*.
  - The final number is $0100\ 0101\ 1000\ 0000\ 0010\ 0001\ 0000\ 0000_2$, or 45802100_{16}.

0 = 0000
1 = 0001
2 = 0010
3 = 0011
4 = 0100
5 = 0101
6 = 0110
7 = 0111
8 = 1000
9 = 1001
A = 1010
B = 1011
C = 1100
D = 1101
E = 1110
F = 1111

The 32-bit IEEE-754 FP



Floating-Point Arithmetic

- Consider an example using an *unsigned* *normalized 8-bit (**1 + 7 bits**)* *significand* and an ***unbiased*** *exponent* with $A = 1.010\ 1001_2 \times 2^4$ and $B = 1.100\ 1100_2 \times 2^3$

- To multiply these numbers,
  - you *multiply* the significands and
  - *add* the exponents

- $A \times B = 1.010\ 1001_2 \times 2^4 \times 1.100\ 1100_2 \times 2^3$

$= 1.010\ 1001_2 \times 1.100\ 1100_2 \times 2^{3+4}$

$= 10.00\ 0110\ 1010\ 1100_2 \times 2^7$

After normalization:

$= 1.000\ 0110\ 1010\ 1100_2 \times 2^8.$

After rounding using:

truncation, i.e., *rounding towards zero*:

➔ $1.000\ 0110_2 \times 2^8 = (268_{10})$

rounding up, i.e., *rounding toward infinity*:

➔ $1.000\ 0111_2 \times 2^8 = (270_{10})$

How about rounding to the nearest?

$A = 1.010\ 1001_2 \times 2^4$
$= 1010\ 1.001_2$
$= 21.125_{10}$

$B = 1.100\ 1100_2 \times 2^3$
$= 1100.\ 1100_2$
$= 12.75_{10}$

$A \times B =$ **269.34375_{10}**

$269_{10} = 1\ 0000\ 1101_2$
$0.34375_{10} = 0.010\ 1100_2$

$A \times B =$
$1000\ 0110\ 1.010\ 1100_2$

Why is not it rounded to 269_{10}?

Floating-Point Arithmetic

- Now let's look at the addition.
- If these two floating-point numbers ($A = 1.010\ 1001_2 \times 2^4$ and $B = 1.100\ 1100_2 \times 2^3$) were to be added *by hand*, we would ***automatically align the binary points*** of A and B as follows.

$$\begin{array}{r} 10101.001_2 \\ +\ 1100.1100_2 \\ \hline 100001.1110_2 \end{array}$$

$A = 1.010\ 1001_2 \times 2^4$
$= 1010\ 1.001_2$
$= 21.125_{10}$

$B = 1.100\ 1100_2 \times 2^3$
$= 1100.\ 1100_2$
$= 12.75_{10}$

A + B = **33.875_{10}**

$33_{10} = 1\ 00001_2$
$0.875_{10} = 0.111_2$

A + B =
100001.111_2

Floating-Point Arithmetic

- However, as these numbers are held in a *normalized* floating-point format the computer has to carry out the following steps to ***equalize exponents***:

$$A = \ \ 1.0101001_2 \times 2^4$$

$$B = \underline{+1.1001100_2} \times 2^3$$

1. *Identify* the number with *the smaller exponent*.
2. *Make the smaller exponent equal to the larger exponent* by dividing the significand of the smaller number by the same factor by which its exponent was increased, ***i.e., un-normalizing the small number to have the same exponent value as the large number.***
   ($1.100\ 1100_2 \times 2^3 \rightarrow 0.110\ 0110\ 0_2 \times 2^4 \rightarrow 0.110\ 0110_2 \times 2^4$).
3. *Add (or subtract) the significands.*
4. If necessary, *normalize the result.*

- We can now add *A* to the denormalized *B*.

$$A = \ \ 1.010\ 1001_2 \times 2^4$$

$$B = \underline{+\ 0.110\ 0110_2 \times 2^4}$$

$$10.000\ 1111_2 \times 2^4 \rightarrow 1.000\ 0111\ 1_2 \times 2^5 = \mathbf{33.875_{10}}$$

- After rounding using *truncation*, i.e., *rounding towards zero*:

$$\rightarrow 1.000\ 0111_2 \times 2^5 = (33.75_{10})$$

rounding up, i.e., *rounding toward infinity*:

$$\rightarrow 1.000\ 1000_2 \times 2^5 = (34_{10})$$

How about rounding to the nearest?

Floating-Point Arithmetic

- Consider another example using an *unsigned normalized 8-bit (**1+7 bits**) significand* and an ***unbiased** exponent* with $A=1.010\ 1001_2 \times 2^4$ & $C=1.100\ 1100_2 \times 2^{13}$

$$\begin{array}{rrl} A & = & 1.0101001_2 \times 2^4 \\ C & = & \underline{+1.1001100_2} \times 2^{13} \end{array}$$

1. *Identify* the number with *the smaller exponent*.
2. *Make the smaller exponent equal to the larger exponent* by dividing the significand of the smaller number by the same factor by which its exponent was increased, ***i.e., un-normalizing the small number to have the same exponent value as the large number.*** ($1.010\ 1001_2 \times 2^4 \rightarrow 0.000\ 0000\ 010101001_2 \times 2^{13} \rightarrow 0.000\ 0000_2 \times 2^{13}$)
3. *Add (or subtract) the significands.*

- We can now add C to the un-normalized A.

$$\begin{array}{rrl} A & = & 0.000\ 0000_2 \times 2^{13} \\ C & = & \underline{+\ 1.100\ 1100_2 \times 2^{13}} \\ & & 1.100\ 1100_2 \times 2^{13} \rightarrow C \end{array}$$

- If the difference between the two exponents of the ***normalized*** two numbers is *greater than* the number of significant bits (i.e., 7 + 1) ➔ the addition result of these two numbers will be the larger of them.

The result is the larger number of the two.

Equalize exponents

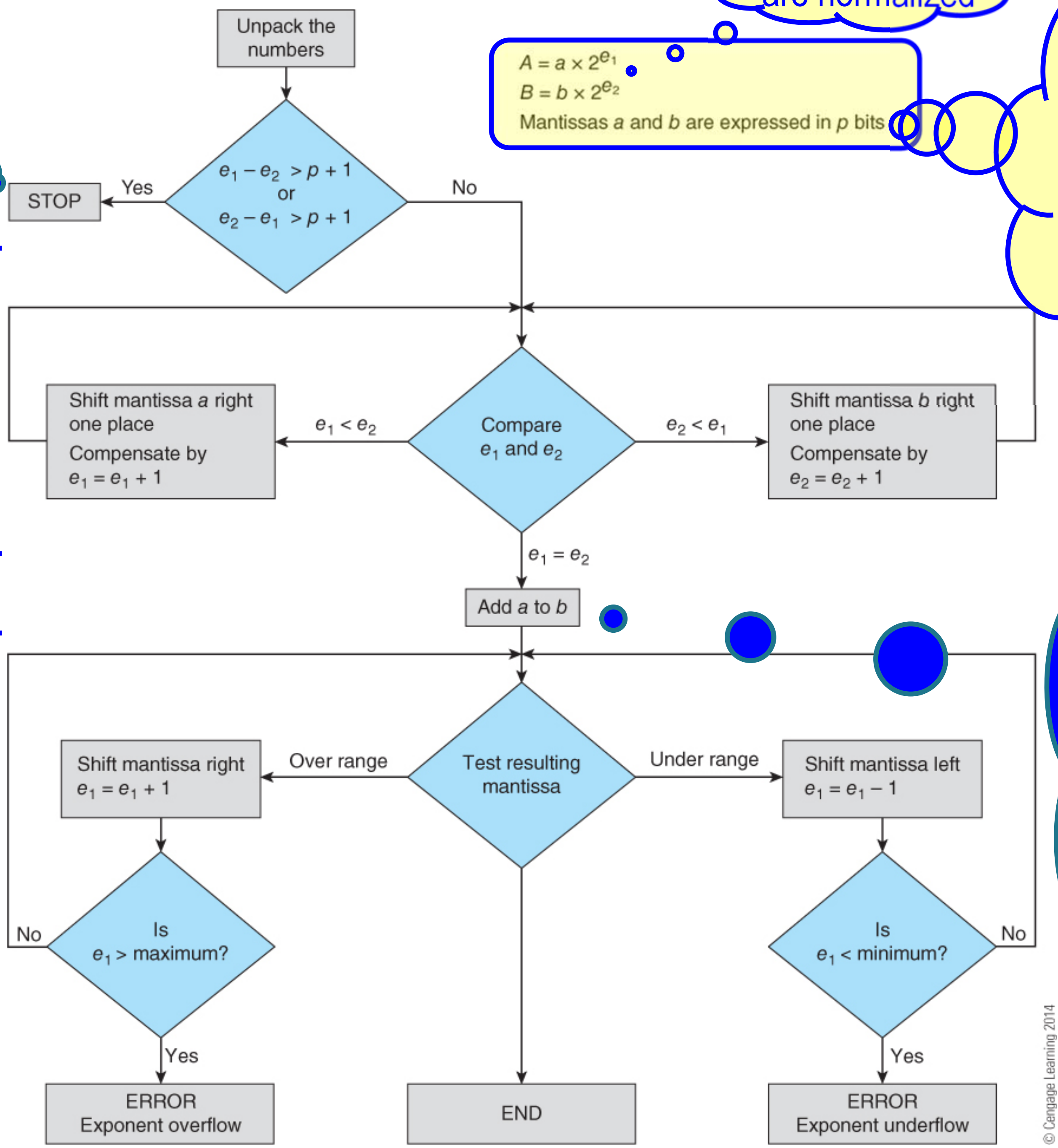
post normalization, if necessary

Both A and B are normalized

$A = a \times 2^{e_1}$
$B = b \times 2^{e_2}$
Mantissas a and b are expressed in p bits

p here is the number of bits without the omitted leading 1, i.e., the number of significant fractional bits, i.e., in single-precision IEEE FP standard, **p** = 23

FIGURE 2.10 Flowchart for floating-point addition and subtraction



For subtraction, you need to perform the two's complement on the negative number.

Then, the result is converted back to sign-magnitude again to be ready for the normalization step, if needed.