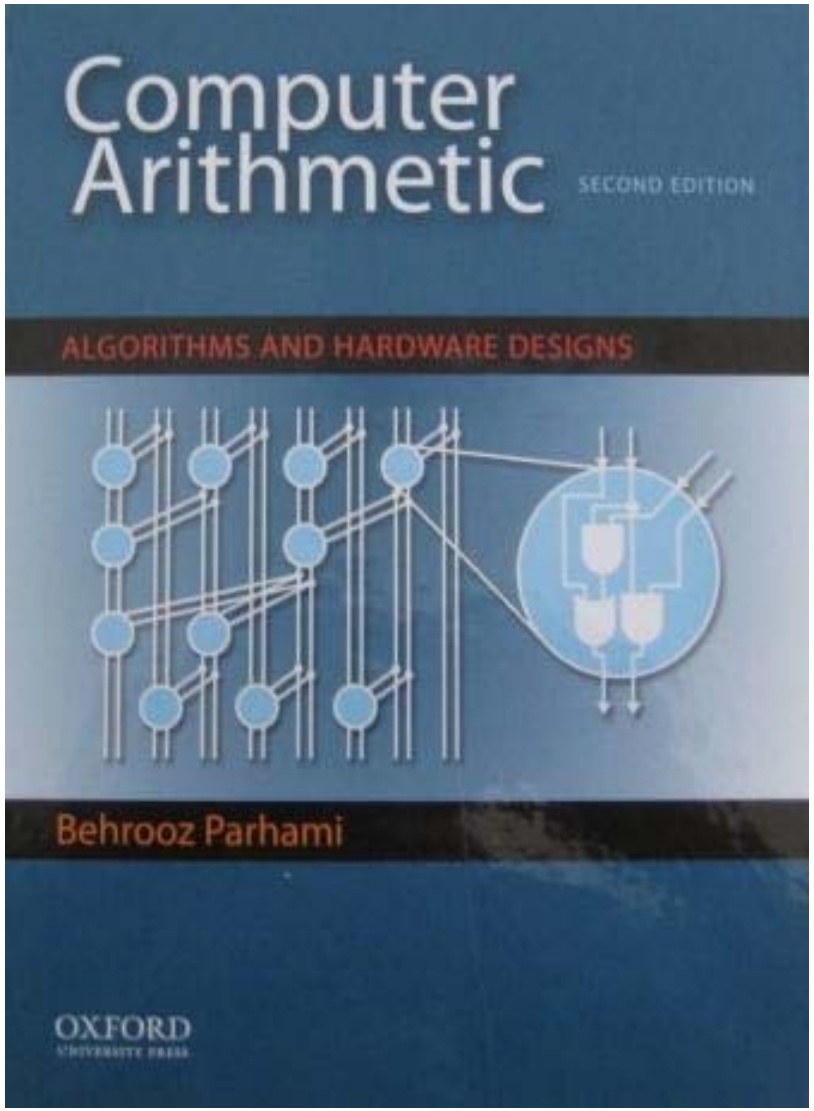


Part V
Real Arithmetic

	Parts	Chapters
Elementary Operations	I. Number Representation	1. Numbers and Arithmetic 2. Representing Signed Numbers 3. Redundant Number Systems 4. Residue Number Systems
	II. Addition / Subtraction	5. Basic Addition and Counting 6. Carry-Lookahead Adders 7. Variations in Fast Adders 8. Multioperand Addition
	III. Multiplication	9. Basic Multiplication Schemes 10. High-Radix Multipliers 11. Tree and Array Multipliers 12. Variations in Multipliers
	IV. Division	13. Basic Division Schemes 14. High-Radix Dividers 15. Variations in Dividers 16. Division by Convergence
	V. Real Arithmetic	17. Floating-Point Reperesentations 18. Floating-Point Operations 19. Errors and Error Control 20. Precise and Certifiable Arithmetic
	VI. Function Evaluation	21. Square-Rooting Methods 22. The CORDIC Algorithms 23. Variations in Function Evaluation 24. Arithmetic by Table Lookup
	VII. Implementation Topics	25. High-Throughput Arithmetic 26. Low-Power Arithmetic 27. Fault-Tolerant Arithmetic 28. Reconfigurable Arithmetic

Appendix: Past, Present, and Future

About This Presentation

This presentation is intended to support the use of the textbook *Computer Arithmetic: Algorithms and Hardware Designs* (Oxford U. Press, 2nd ed., 2010, ISBN 978-0-19-532848-6). It is updated regularly by the author as part of his teaching of the graduate course ECE 252B, Computer Arithmetic, at the University of California, Santa Barbara. Instructors can use these slides freely in classroom teaching and for other educational purposes. Unauthorized uses are strictly prohibited. © Behrooz Parhami

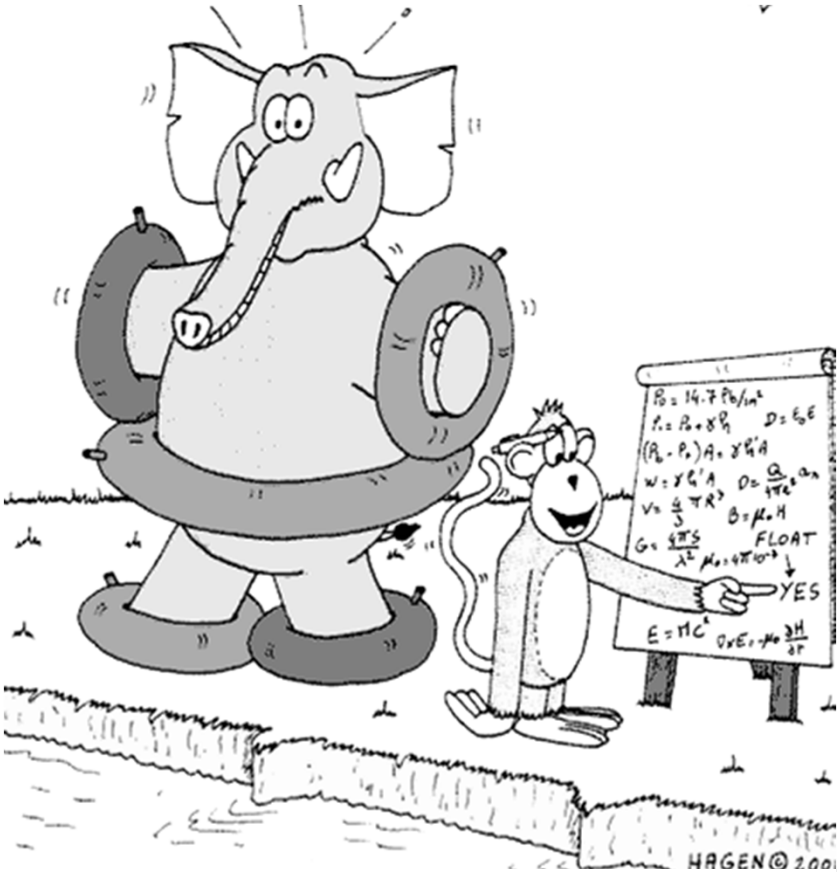
Edition	Released	Revised	Revised	Revised	Revised
First	Jan. 2000	Sep. 2001	Sep. 2003	Oct. 2005	May 2007
		May 2008	May 2009		
Second	May 2010	Apr. 2011	May 2012	May 2015	

V Real Arithmetic

Review floating-point numbers, arithmetic, and errors:

- How to combine wide range with high precision
- Format and arithmetic ops; the IEEE standard
- Causes and consequence of computation errors
- When can we trust computation results?

Topics in This Part	
Chapter 17	Floating-Point Representations
Chapter 18	Floating-Point Operations
Chapter 19	Errors and Error Control
Chapter 20	Precise and Certifiable Arithmetic



"According to my calculation,
you should float now ... I think ..."

"It's an inexact science."

17 Floating-Point Representations

Chapter Goals

Study a representation method offering both wide range (e.g., astronomical distances) and high precision (e.g., atomic distances)

Chapter Highlights

Floating-point formats and related tradeoffs
The need for a floating-point standard
Finiteness of precision and range
Fixed-point and logarithmic representations as special cases at the two extremes

Floating-Point Representations: Topics

Topics in This Chapter
17.1 Floating-Point Numbers
17.2 The IEEE Floating-Point Standard
17.3 Basic Floating-Point Algorithms
17.4 Conversions and Exceptions
17.5 Rounding Schemes
17.6 Logarithmic Number Systems

17.1 Floating-Point Numbers

No finite number system can represent all real numbers
Various systems can be used for a subset of real numbers

Fixed-point	$\pm w.f$	Low precision and/or range
Rational	$\pm p/q$	Difficult arithmetic
Floating-point	$\pm s \times b^e$	Most common scheme
Logarithmic	$\pm \log_b x$	Limiting case of floating-point

Fixed-point numbers

$x = (0000\ 0000.0000\ 1001)_{two}$ Small number
$y = (1001\ 0000.0000\ 0000)_{two}$ Large number

Square of neither number representable

Floating-point numbers

$x = \pm s \times b^e$ or $\pm \text{significand} \times \text{base}^{\text{exponent}}$

$x = 1.001 \times 2^{-5}$
$y = 1.001 \times 2^{+7}$

A floating-point number comes with two signs:

Number sign, usually appears as a separate bit
Exponent sign, usually embedded in the biased exponent

Floating-Point Number Format and Distribution

Fig. 17.1 Typical floating-point number format.

Sign

0 : +

1 : –

Exponent: Signed integer, often represented as unsigned value by adding a bias

Range with h bits: [–bias, 2^h–1–bias]

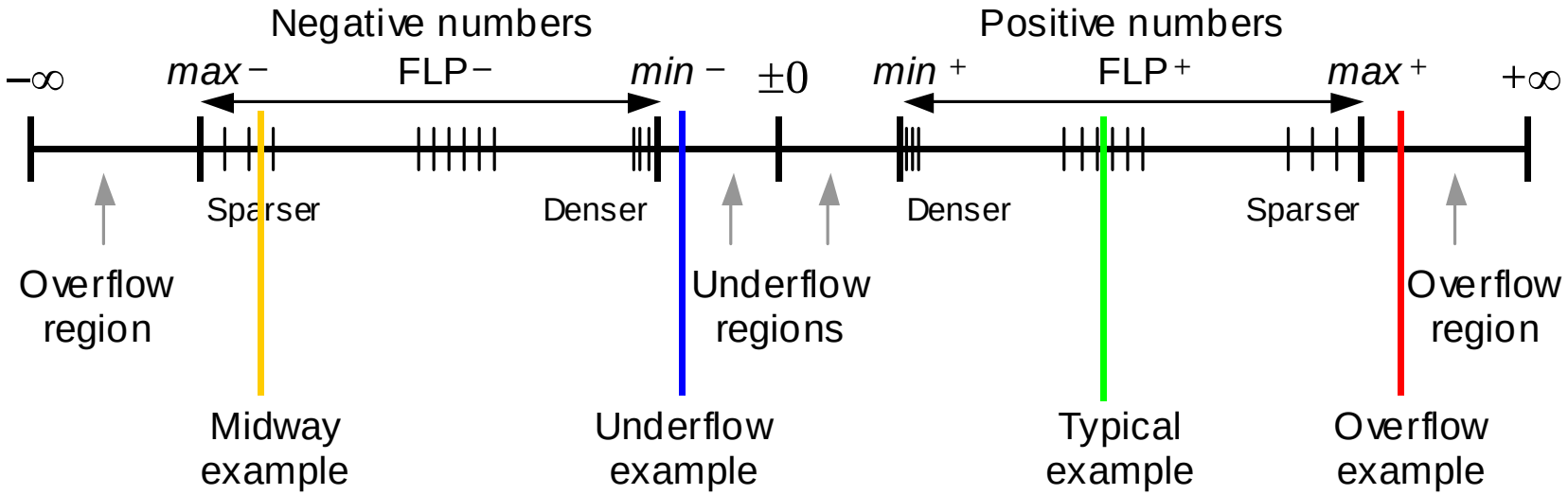
Significand: Represented as a fixed-point number

Usually normalized by shifting, so that the MSB becomes nonzero. In radix 2, the fixed leading 1 can be removed to save one bit; this bit is known as "hidden 1".

1.001×2^{-5}

$1.001 \times 2^{+7}$

Fig. 17.2 Subranges and special values in floating-point number representations.



Floating-Point Before the IEEE Standard

Computer manufacturers tended to have their own hardware-level formats

This created many problems, as floating-point computations could produce vastly different results (not just differing in the last few significant bits)

To get a sense for the wide variations in floating-point formats, visit:

http://www.mrob.com/pub/math/floatformats.html

In computer arithmetic, we talked about IBM, CDC, DEC, Cray, … formats and discussed their relative merits

First IEEE standard for binary floating-point arithmetic was adopted in 1985 after years of discussion

The 1985 standard was continuously discussed, criticized, and clarified for a couple of decades

In 2008, after several years of discussion, a revised standard was issued

17.2 The IEEE Floating-Point Standard

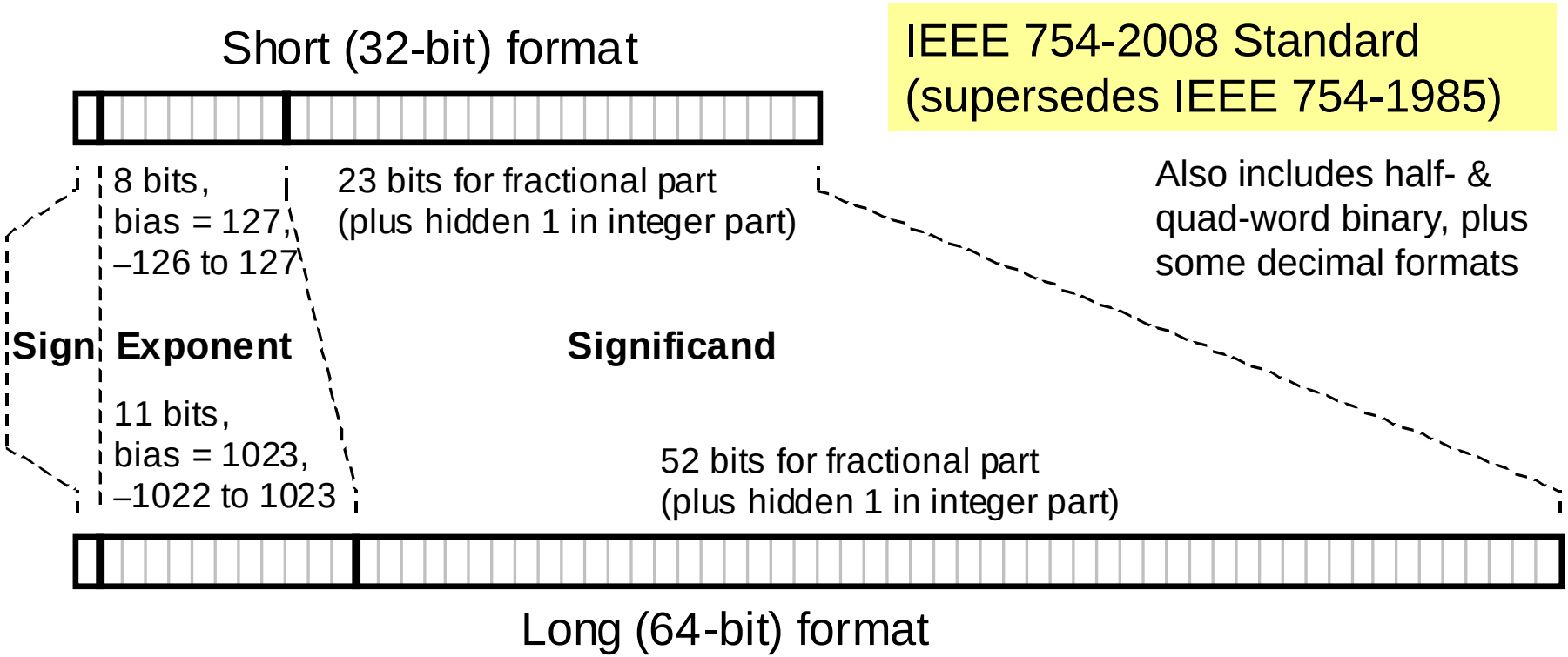


Fig. 17.3 The IEEE standard floating-point number representation formats.

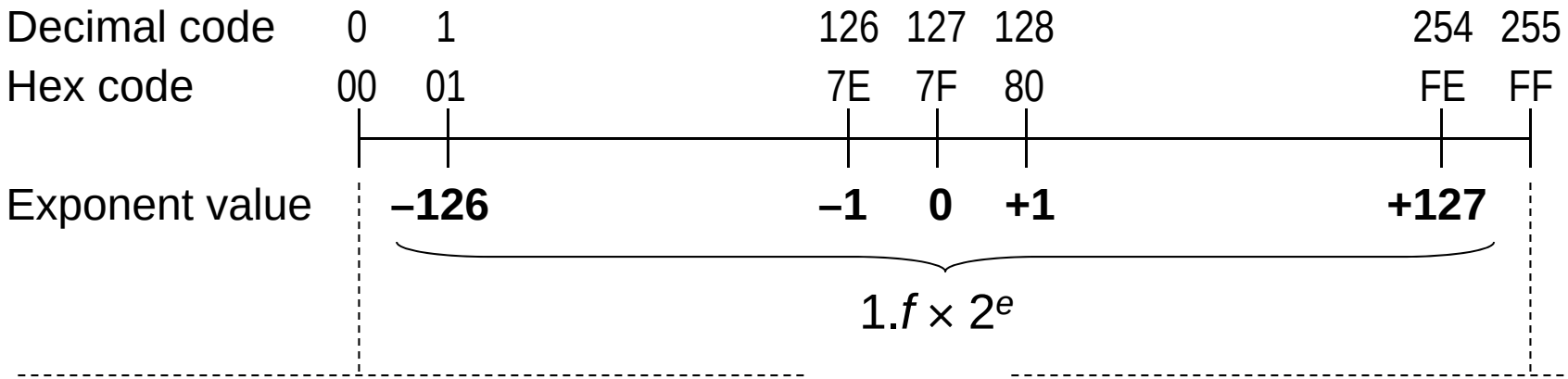
Overview of IEEE 754-2008 Standard Formats

Table 17.1 Some features of the IEEE 754-2008 standard floating-point number representation formats

Feature	Single / Short	Double / Long
Word width (bits)	32	64
Significand bits	23 + 1 hidden	52 + 1 hidden
Significand range	$[1, 2 - 2^{-23}]$	$[1, 2 - 2^{-52}]$
Exponent bits	8	11
Exponent bias	127	1023
Zero (± 0)	$e + bias = 0$, $f = 0$	$e + bias = 0$, $f = 0$
Denormal	$e + bias = 0$, $f \neq 0$ represents $\pm 0.f \times 2^{-126}$	$e + bias = 0$, $f \neq 0$ represents $\pm 0.f \times 2^{-1022}$
Infinity ($\pm\infty$)	$e + bias = 255$, $f = 0$	$e + bias = 2047$, $f = 0$
Not-a-number (NaN)	$e + bias = 255$, $f \neq 0$	$e + bias = 2047$, $f \neq 0$
Ordinary number	$e + bias \in [1, 254]$ $e \in [-126, 127]$ represents $1.f \times 2^e$	$e + bias \in [1, 2046]$ $e \in [-1022, 1023]$ represents $1.f \times 2^e$
min	$2^{-126} \cong 1.2 \times 10^{-38}$	$2^{-1022} \cong 2.2 \times 10^{-308}$
max	$\cong 2^{128} \cong 3.4 \times 10^{38}$	$\cong 2^{1024} \cong 1.8 \times 10^{308}$

Exponent Encoding

Exponent encoding in 8 bits for the single/short (32-bit) IEEE 754 format



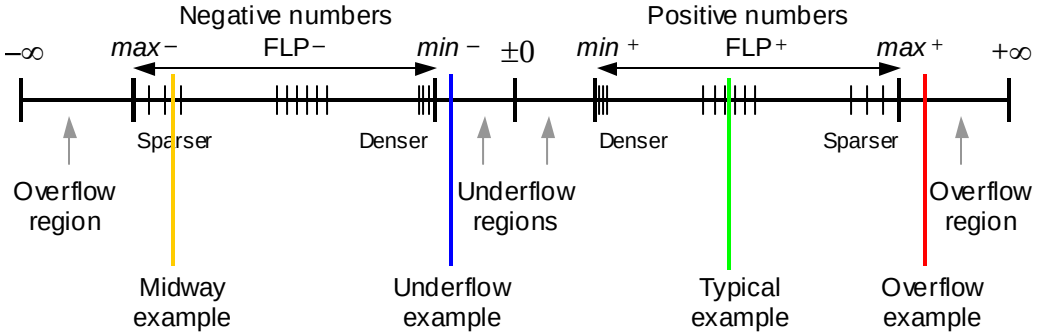
$f = 0$: Representation of ± 0

$f \neq 0$: Representation of subnormals, $0.f \times 2^{-126}$

$f = 0$: Representation of $\pm\infty$

$f \neq 0$: Representation of NaNs

Exponent encoding in 11 bits for the double/long (64-bit) format is similar



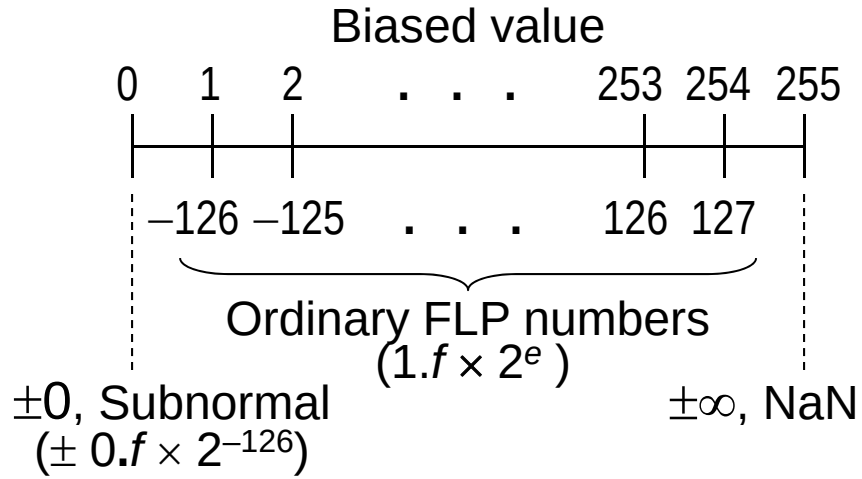
Special Operands and Subnormals

Operations on special operands:

Ordinary number ÷ (+∞) = ±0

(+∞) × Ordinary number = ±∞

NaN + Ordinary number = NaN



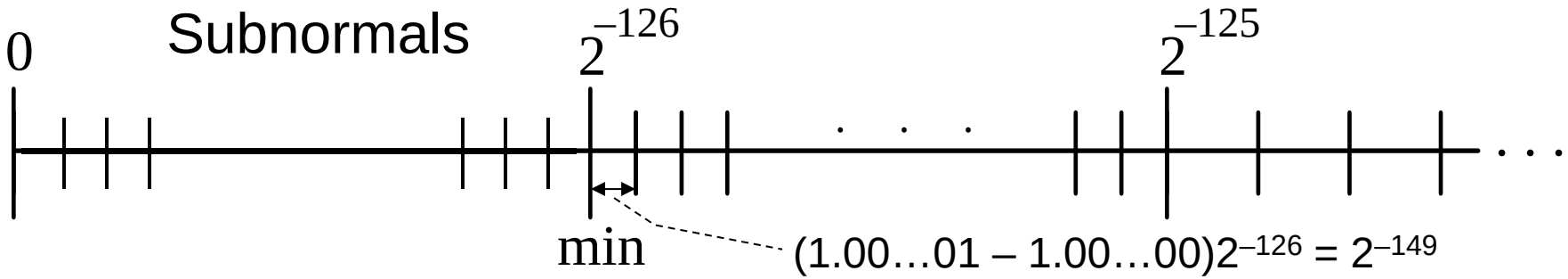
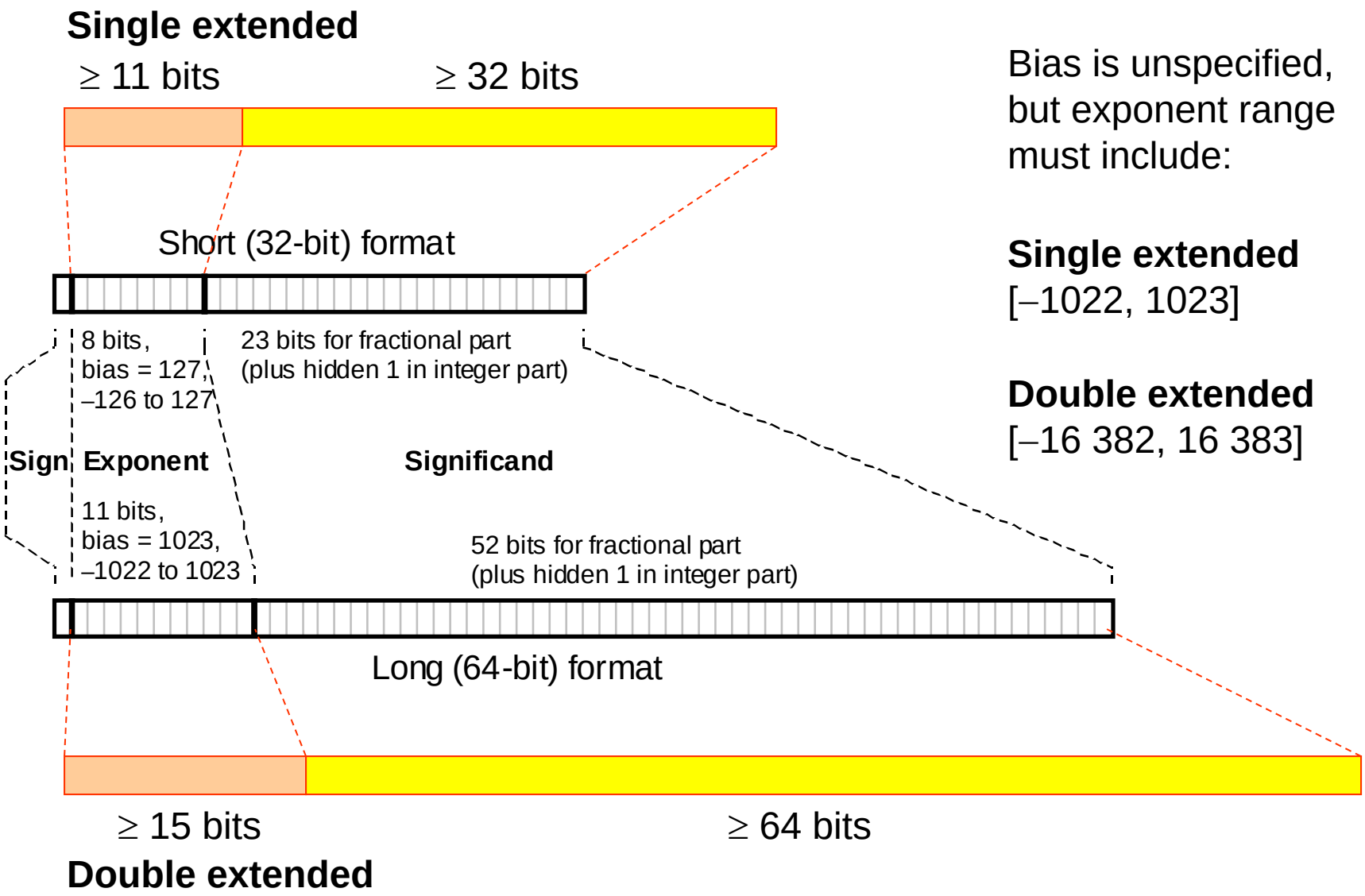


Fig. 17.4 Subnormals in the IEEE single-precision format.

Extended Formats



Bias is unspecified, but exponent range must include:

Single extended
[–1022, 1023]

Double extended
[–16 382, 16 383]

Requirements for Arithmetic

Results of the 4 basic arithmetic operations (+, –, ×, ÷) as well as square-rooting must match those obtained if all intermediate computations were infinitely precise

That is, a floating-point arithmetic operation should introduce no more imprecision than the error attributable to the final rounding of a result that has no exact representation (this is the best possible)

Example:

$(1 + 2^{-1}) \times (1 + 2^{-23})$

Exact result $1 + 2^{-1} + 2^{-23} + 2^{-24}$

Chopped result $1 + 2^{-1} + 2^{-23}$ Error = –½ *ulp*

Rounded result $1 + 2^{-1} + 2^{-22}$ Error = +½ *ulp*

17.3 Basic Floating-Point Algorithms

Addition

Assume $e1 \geq e2$; *alignment shift* (*preshift*) is needed if $e1 > e2$

$$(\pm s1 \times b^{e1}) + (\pm s2 \times b^{e2}) = (\pm s1 \times b^{e1}) + (\pm s2 / b^{e1-e2}) \times b^{e1}$$
$$= (\pm s1 \pm s2 / b^{e1-e2}) \times b^{e1} = \pm s \times b^{e}$$

Example:

Numbers to be added:

$x = 2^5 \times 1.00101101$

$y = 2^1 \times 1.11101101$ ← Operand with smaller exponent to be preshifted

Operands after alignment shift:

$x = 2^5 \times 1.00101101$

$y = 2^5 \times 0.000111101101$

Result of addition:

$s = 2^5 \times 1.010010111101$ ← Extra bits to be rounded off

$s = 2^5 \times 1.01001100$ ← Rounded sum

Rounding, overflow, and underflow issues discussed later

Floating-Point Multiplication and Division

Multiplication

$$(\pm s1 \times b^{e1}) \times (\pm s2 \times b^{e2}) \;=\; (\pm s1 \times s2) \times b^{e1+e2}$$

Because $s1 \times s2 \in [1, 4)$, postshifting may be needed for normalization

Overflow or underflow can occur during multiplication or normalization

Division

$$(\pm s1 \times b^{e1}) \,/\, (\pm s2 \times b^{e2}) \;=\; (\pm s1 / s2) \times b^{e1-e2}$$

Because $s1/s2 \in (0.5, 2)$, postshifting may be needed for normalization

Overflow or underflow can occur during division or normalization

Floating-Point Square-Rooting

For e even: $\sqrt{s \times b^e} = \sqrt{s} \times b^{e/2}$

For e odd: $\sqrt{bs \times b^{e-1}} = \sqrt{bs} \times b^{(e-1)/2}$

After the adjustment of s to bs and e to $e - 1$, if needed, we have:

$$\sqrt{s^* \times b^{e^*}} = \sqrt{s^*} \times b^{e^*/2}$$

Even

In [1, 4) for IEEE 754

In [1, 2) for IEEE 754

Overflow or underflow is impossible; no postnormalization needed

17.4 Conversions and Exceptions

Conversions from fixed- to floating-point

Conversions between floating-point formats

Conversion from high to lower precision: Rounding

The IEEE 754-2008 standard includes five rounding modes:

- Round to nearest, ties away from 0 (rtna)
- Round to nearest, ties to even (rtne) [default rounding mode]
- Round toward zero (inward)
- Round toward $+\infty$ (upward)
- Round toward $-\infty$ (downward)

Exceptions in Floating-Point Arithmetic

Divide by zero

Overflow

Underflow

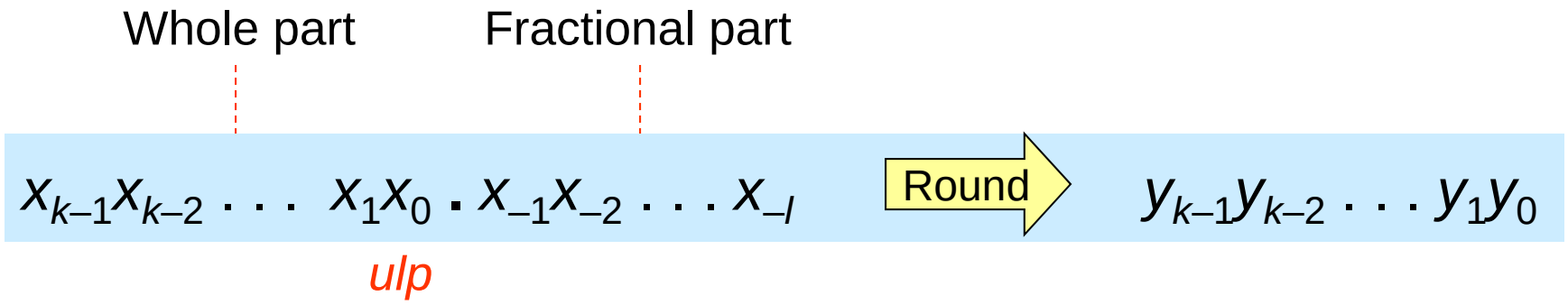
Inexact result: Rounded value not the same as original

Invalid operation: examples include

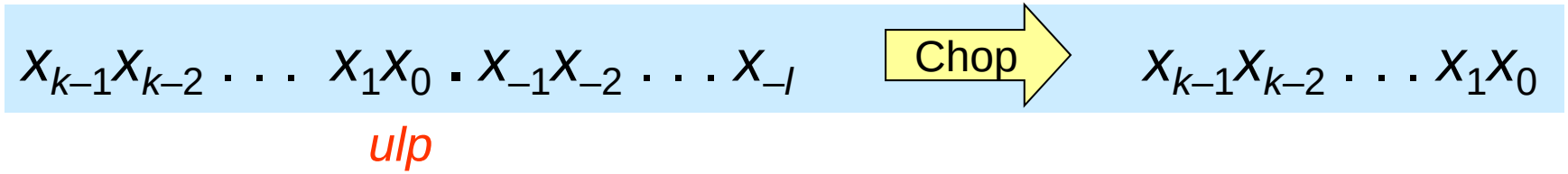
Addition	$(+\infty) + (-\infty)$
Multiplication	$0 \times \infty$
Division	$0/0$ or ∞/∞
Square-rooting	operand < 0

Produce NaN as their results

17.5 Rounding Schemes



The simplest possible rounding scheme: chopping or truncation



Truncation or Chopping

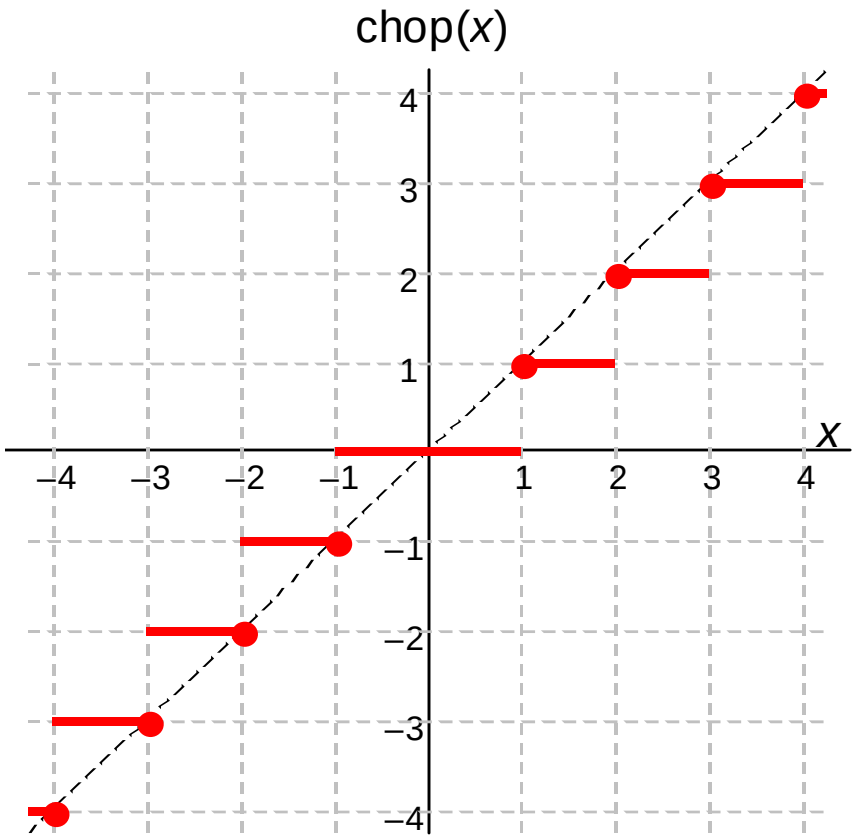


Fig. 17.5 Truncation or chopping of a signed-magnitude number (same as round toward 0).

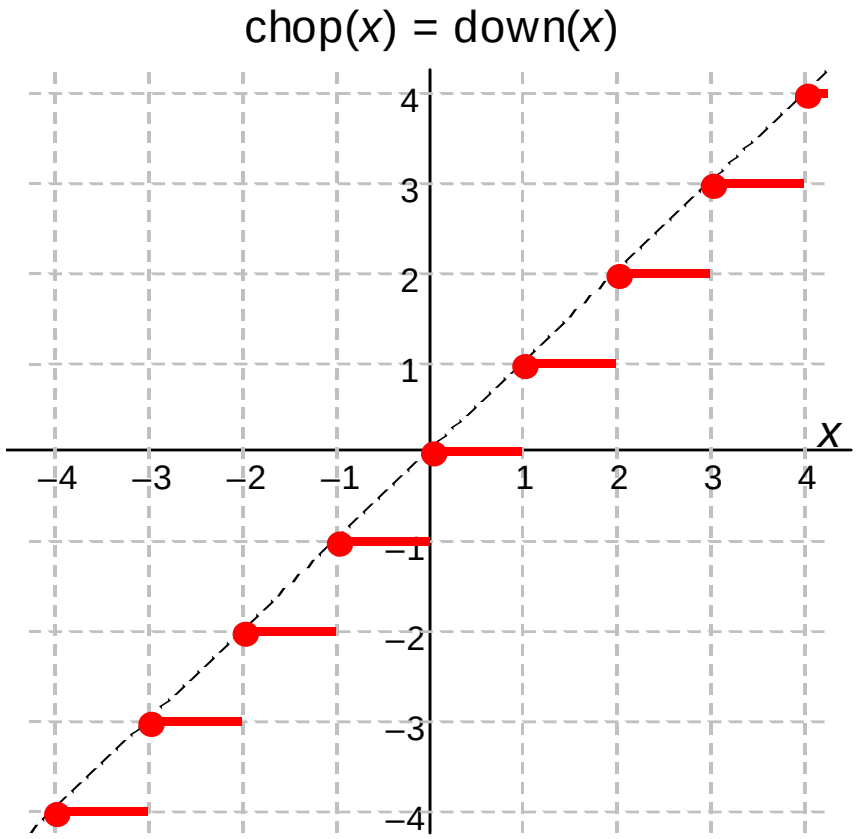
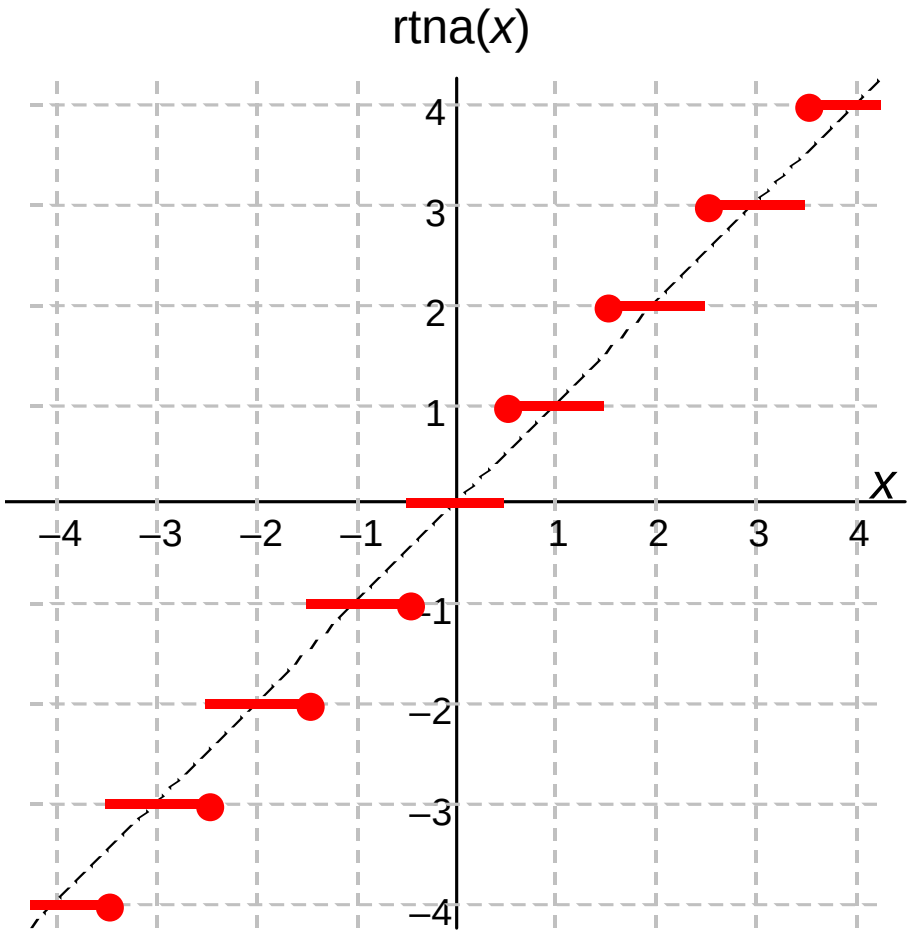


Fig. 17.6 Truncation or chopping of a 2's-complement number (same as downward-directed rounding).

Round to Nearest Number



Rounding has a slight upward bias.

Consider rounding
$(x_{k-1}x_{k-2} \ldots x_1x_0.x_{-1}x_{-2})_{two}$
to an integer $(y_{k-1}y_{k-2} \ldots y_1y_0.)_{two}$

The four possible cases, and their representation errors are:

$x_{-1}x_{-2}$	Round	Error
00	down	0
01	down	–0.25
10	up	0.5
11	up	0.25

With equal prob., mean = 0.125

Fig. 17.7 Rounding of a signed-magnitude value to the nearest number.

For certain calculations, the probability of getting a midpoint value can be much higher than 2^{-l}

Round to Nearest Even Number

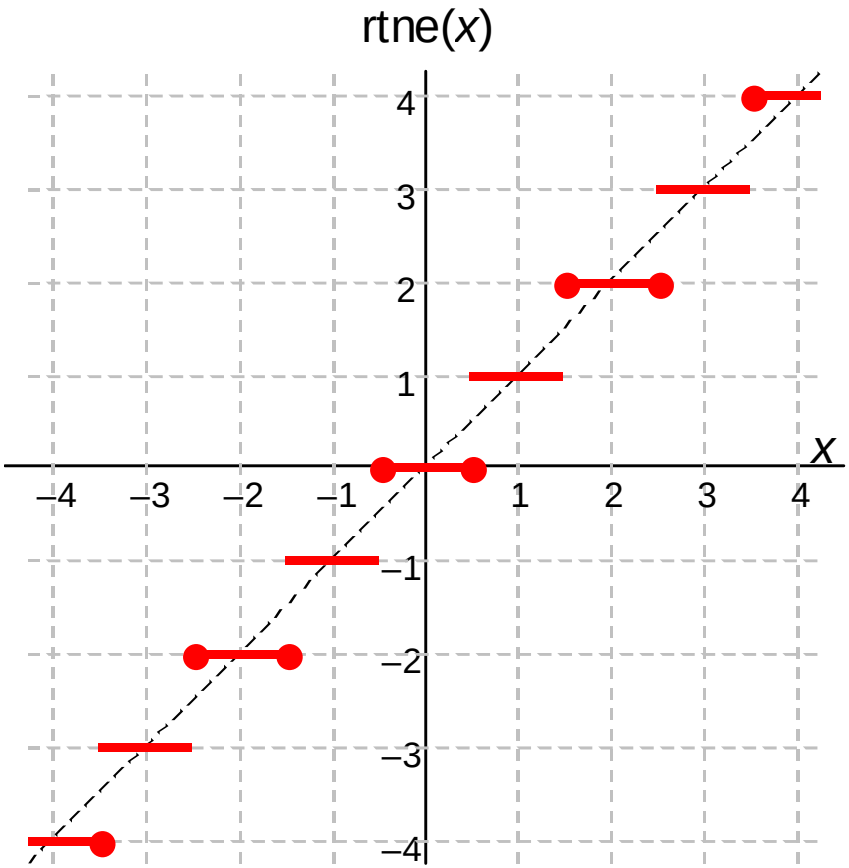


Fig. 17.8 Rounding to the nearest even number.

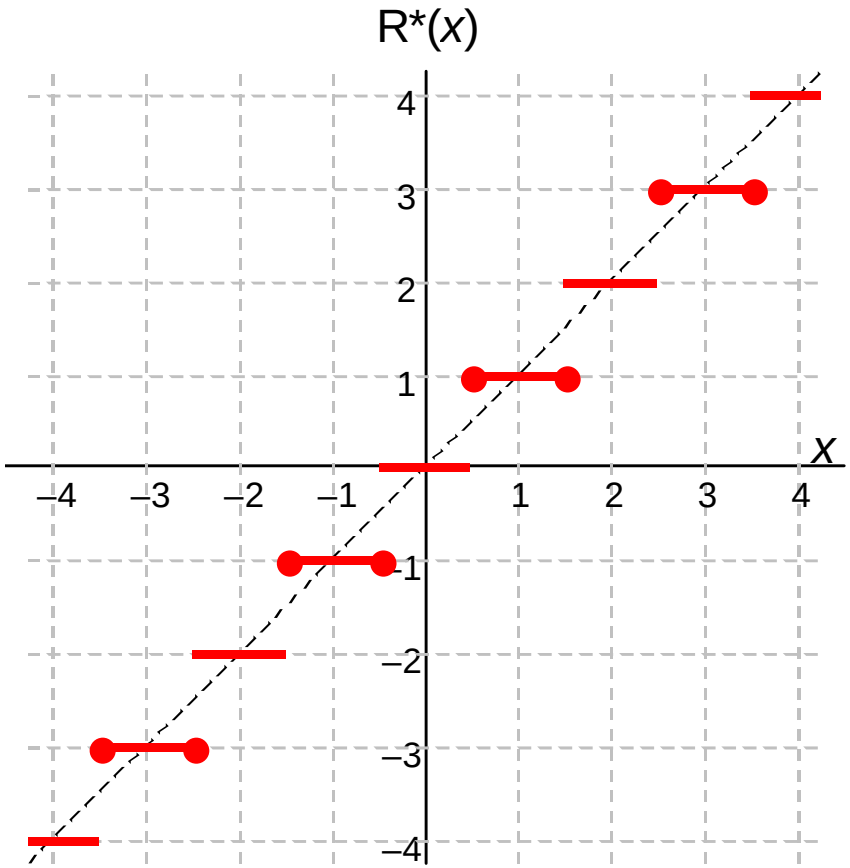


Fig. 17.9 R* rounding or rounding to the nearest odd number.

A Simple Symmetric Rounding Scheme

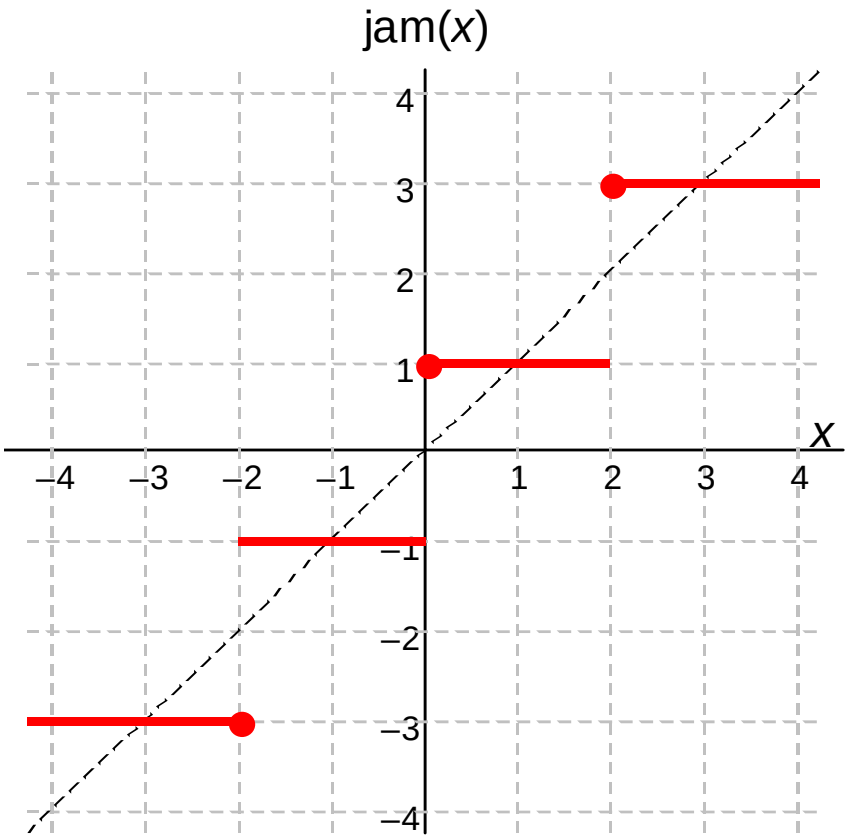


Fig. 17.10 Jamming or von Neumann rounding.

Chop and force the LSB of the result to 1

Simplicity of chopping, with the near-symmetry or ordinary rounding

Max error is comparable to chopping (double that of rounding)

ROM Rounding

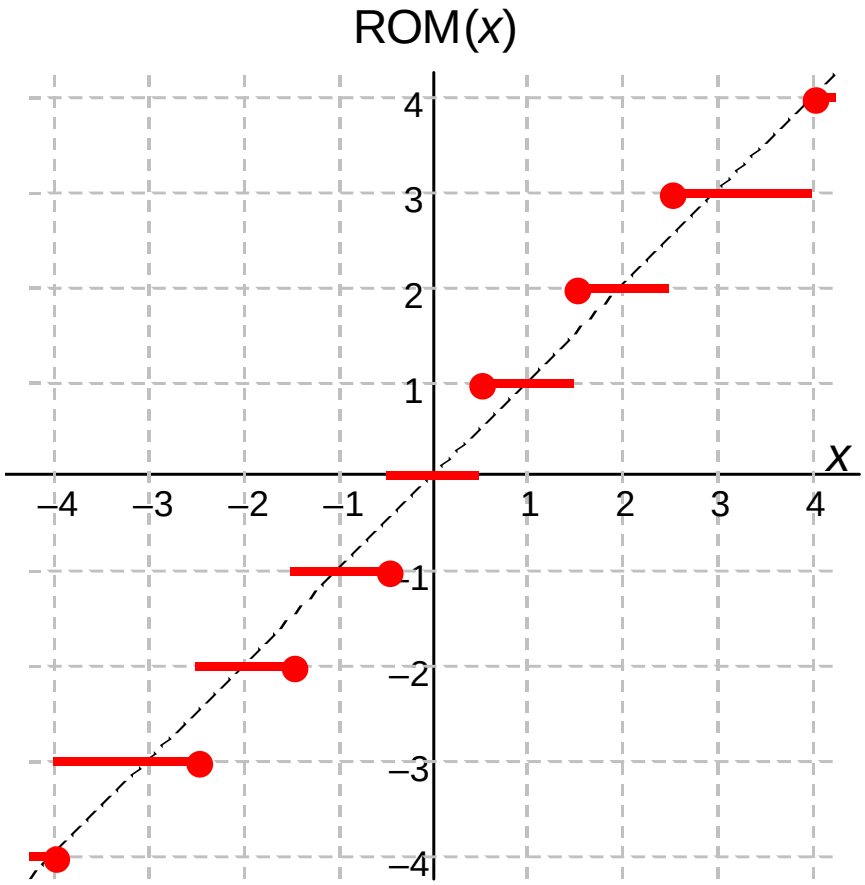


Fig. 17.11 ROM rounding with an 8×2 table.

Example: Rounding with a 32×4 table

Rounding result is the same as that of the round to nearest scheme in 31 of the 32 possible cases, but a larger error is introduced when

$x_3 = x_2 = x_1 = x_0 = x_{-1} = 1$

$x_{k-1} \ldots x_4 x_3 x_2 x_1 x_0 . x_{-1} x_{-2} \ldots x_{-l}$ → ROM → $x_{k-1} \ldots x_4 y_3 y_2 y_1 y_0$

ROM address

ROM data

Directed Rounding: Motivation

We may need result errors to be in a known direction

Example: in computing upper bounds,
larger results are acceptable,
but results that are smaller than correct values
could invalidate the upper bound

This leads to the definition of *directed rounding* modes
upward-directed rounding (round toward $+\infty$) and
downward-directed rounding (round toward $-\infty$)
(required features of IEEE floating-point standard)

Directed Rounding: Visualization

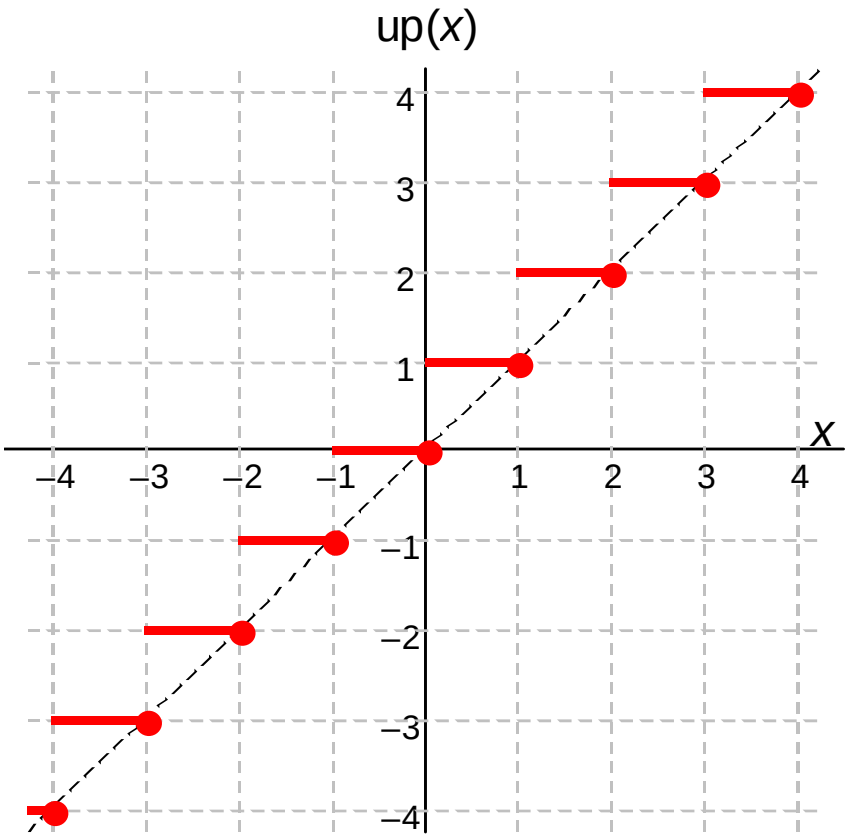


Fig. 17.12 Upward-directed rounding or rounding toward $+\infty$.

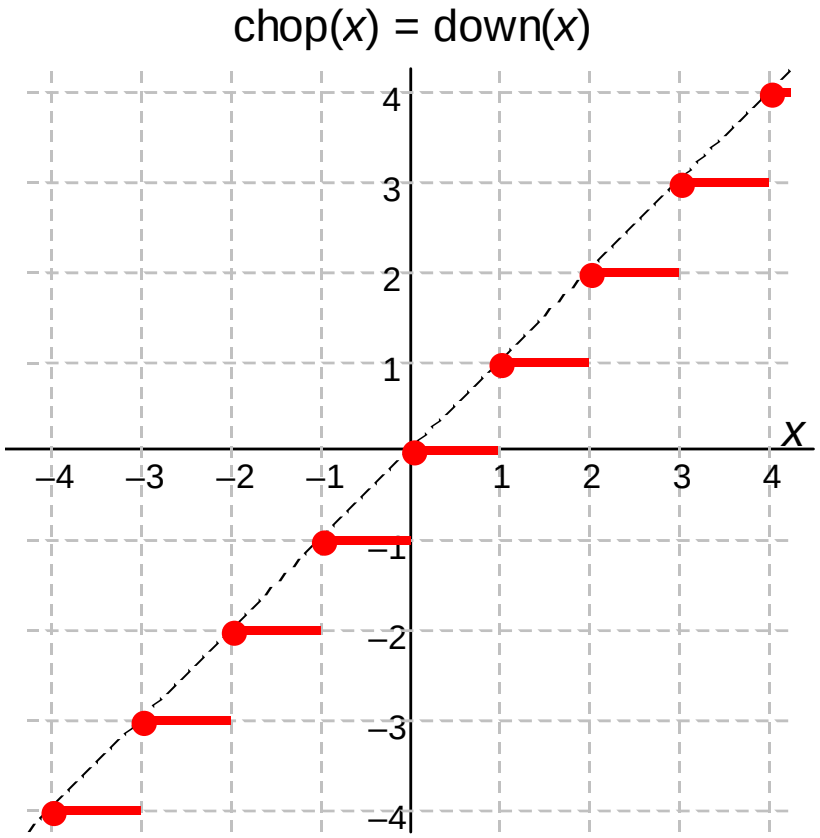


Fig. 17.6 Truncation or chopping of a 2's-complement number (same as downward-directed rounding).

17.6 Logarithmic Number Systems

Sign-and-logarithm number system: Limiting case of FLP representation

$$x = \pm b^e \times 1 \qquad e = \log_b |x|$$

We usually call b the logarithm base, not exponent base

Using an integer-valued e wouldn't be very useful, so we consider e to be a fixed-point number

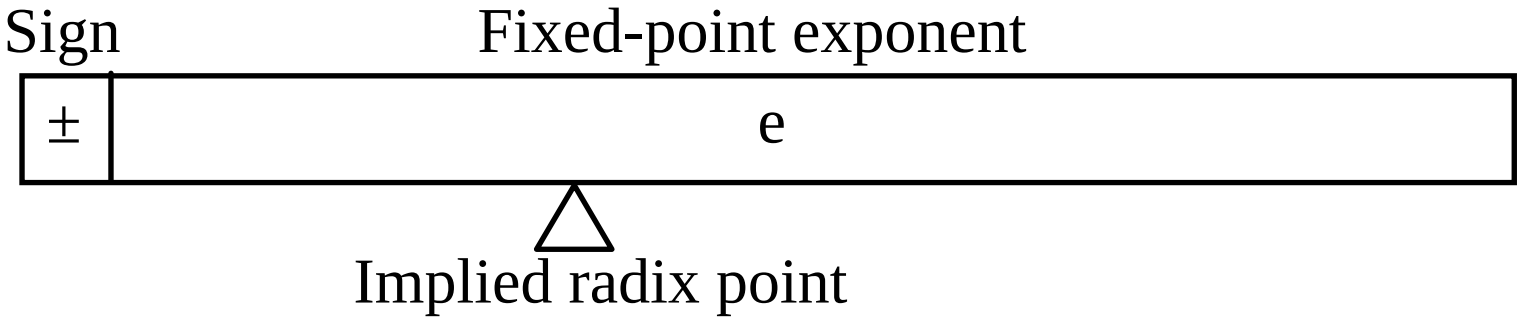


Fig. 17.13 Logarithmic number representation with sign and fixed-point exponent.

Properties of Logarithmic Representation

The logarithm is often represented as a 2's-complement number

$(Sx, Lx) = (\text{sign}(x), \log_2 |x|)$

Simple multiplication and division; harder add and subtract

$L(xy) = Lx + Ly$ $\qquad\qquad$ $L(x/y) = Lx - Ly$

Example: 12-bit, base-2, logarithmic number system

1 1 0 1 1 0 0 0 1 0 1 1

Δ

Sign Radix point

The bit string above represents $-2^{-9.828125} \cong -(0.0011)_{\text{ten}}$

Number range $\cong (-2^{16}, 2^{16})$; *min* = 2^{-16}

Advantages of Logarithmic Representation

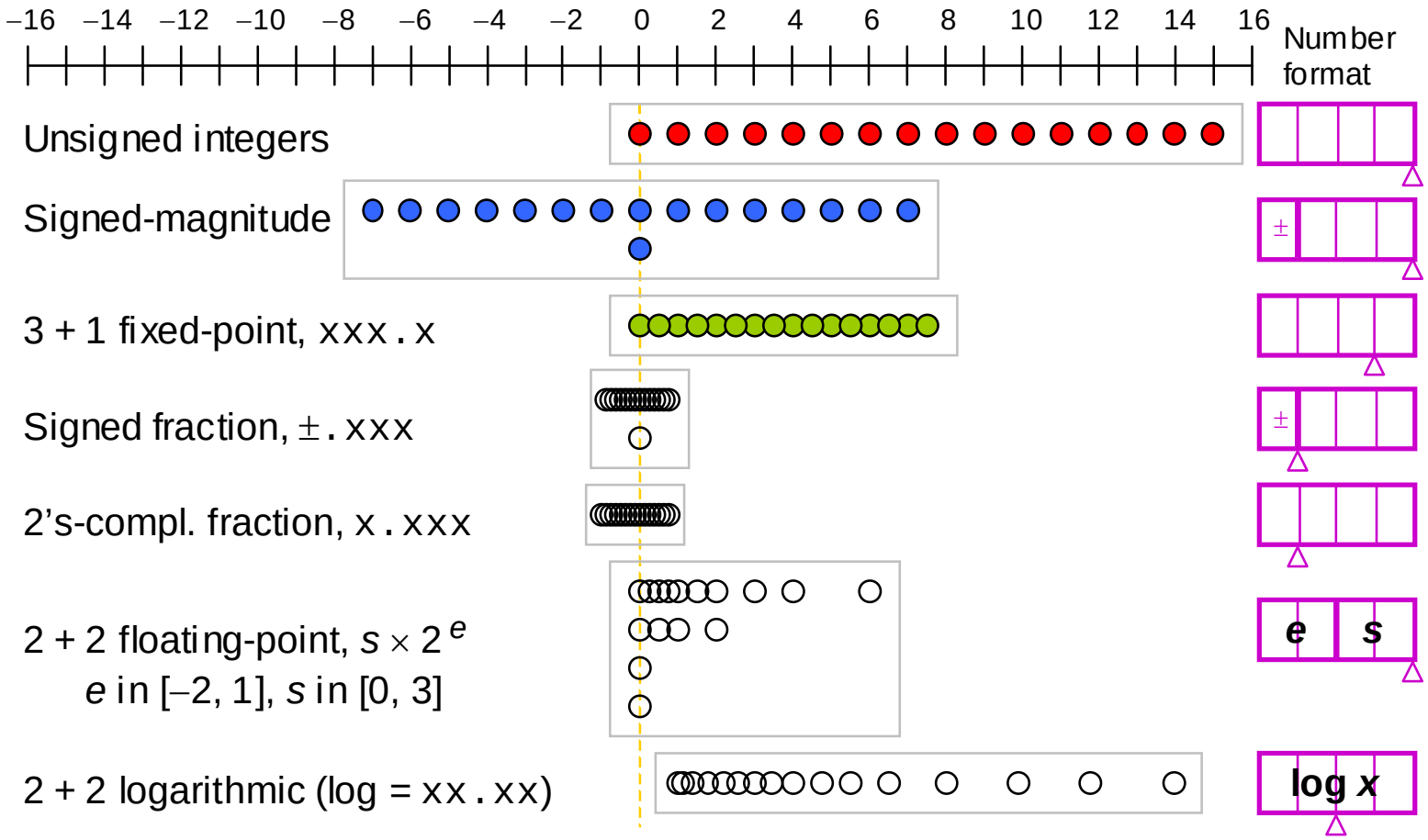


Fig. 1.2 Some of the possible ways of assigning 16 distinct codes to represent numbers.

18 Floating-Point Operations

Chapter Goals

See how adders, multipliers, and dividers are designed for floating-point operands (square-rooting postponed to Chapter 21)

Chapter Highlights

Floating-point operation = preprocessing + exponent and significand arithmetic + postprocessing (+ exception handling)

Adders need preshift, postshift, rounding

Multipliers and dividers are easy to design

Floating-Point Operations: Topics

Topics in This Chapter
18.1 Floating-Point Adders / Subtractors
18.2 Pre- and Postshifting
18.3 Rounding and Exceptions
18.4 Floating-Point Multipliers and Dividers
18.5 Fused-Multiply-Add Units
18.6 Logarithmic Arithmetic Units

18.1 Floating-Point Adders/Subtractors

Floating-Point Addition Algorithm

Assume $e1 \geq e2$; *alignment shift* (*preshift*) is needed if $e1 > e2$

$$(\pm s1 \times b^{e1}) + (\pm s2 \times b^{e2}) = (\pm s1 \times b^{e1}) + (\pm s2 / b^{e1-e2}) \times b^{e1}$$
$$= (\pm s1 \pm s2 / b^{e1-e2}) \times b^{e1} = \pm s \times b^{e}$$

Example:

Numbers to be added:

$x = 2^5 \times 1.00101101$
$y = 2^1 \times 1.11101101$ ← Operand with smaller exponent to be preshifted

Operands after alignment shift:

$x = 2^5 \times 1.00101101$
$y = 2^5 \times 0.000111101101$

Result of addition:

$s = 2^5 \times 1.010010111101$ ← Extra bits to be rounded off
$s = 2^5 \times 1.01001100$ ← Rounded sum

Like signs:
Possible 1-position normalizing right shift

Different signs:
Left shift, possibly by many positions

Overflow/underflow
during addition or normalization

FLP Addition Hardware

Isolate the sign, exponent, significand
Reinstate the hidden 1
Convert operands to internal format
Identify special operands, exceptions

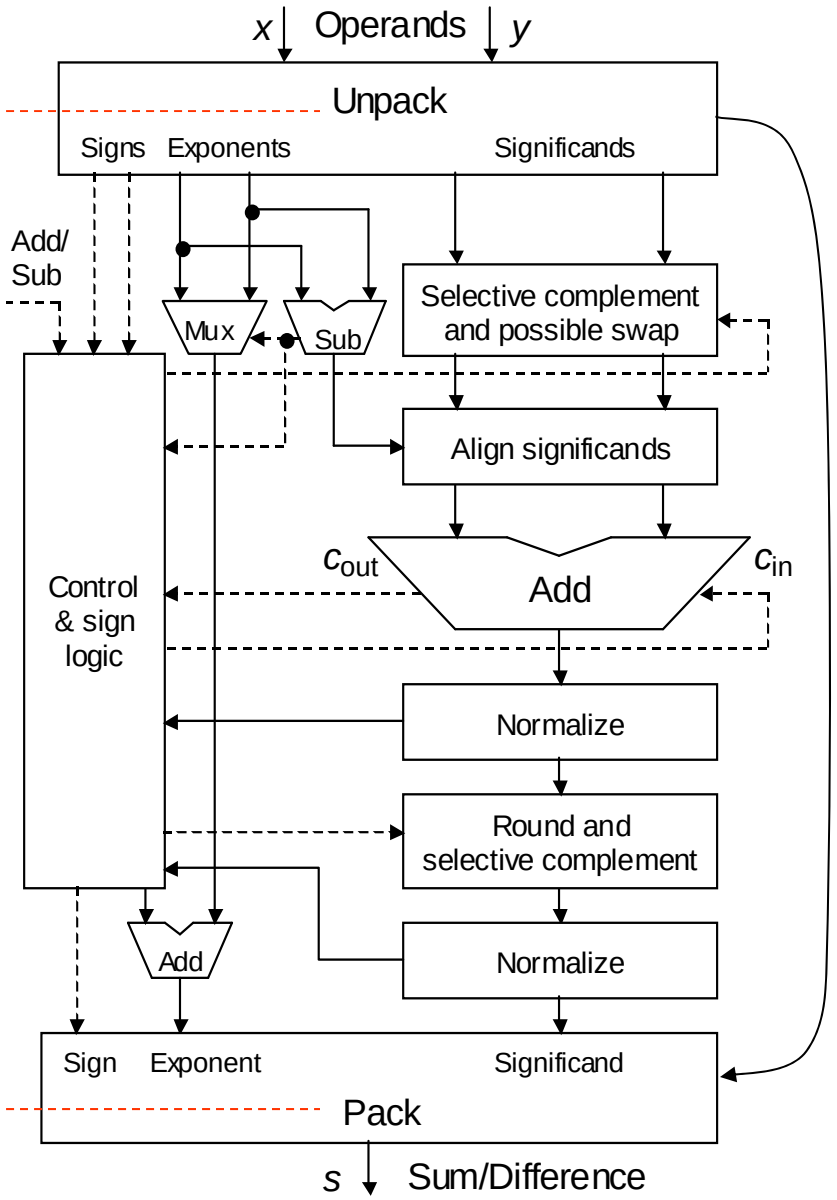


Fig. 18.1 Block diagram of a floating-point adder/subtractor.

Other key parts of the adder:
Significand aligner (preshifter): Sec. 18.2
Result normalizer (postshifter), including leading 0s detector/predictor: Sec. 18.2
Rounding unit: Sec. 18.3
Sign logic: Problem 18.2

Converting internal to external representation, if required, must be done at the rounding stage

Combine sign, exponent, significand
Hide (remove) the leading 1
Identify special outcomes, exceptions

Types of Post-Normalization

$$(\pm s1 \times b^{e1}) + (\pm s2 \times b^{e2}) = (\pm s1 \times b^{e1}) + (\pm s2/b^{e1-e2}) \times b^{e1}$$
$$= (\pm s1 \pm s2/b^{e1-e2}) \times b^{e1} = \pm s \times b^{e}$$

Magnitude in [0, 4)

In [0, 1)	In [1, 2)	In [2, 4)
Arbitrary left shift	None	1-bit right shift

18.2 Pre- and Postshifting

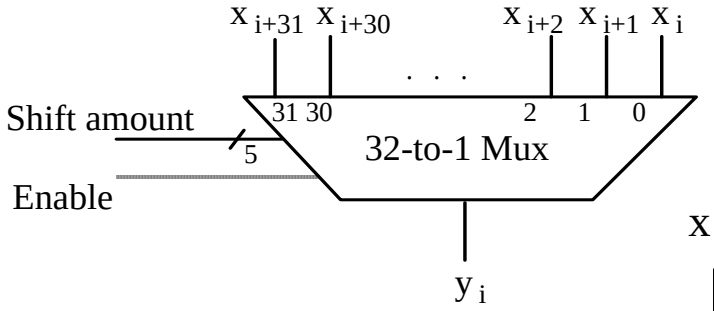


Fig. 18.2 One bit-slice of a single-stage pre-shifter.

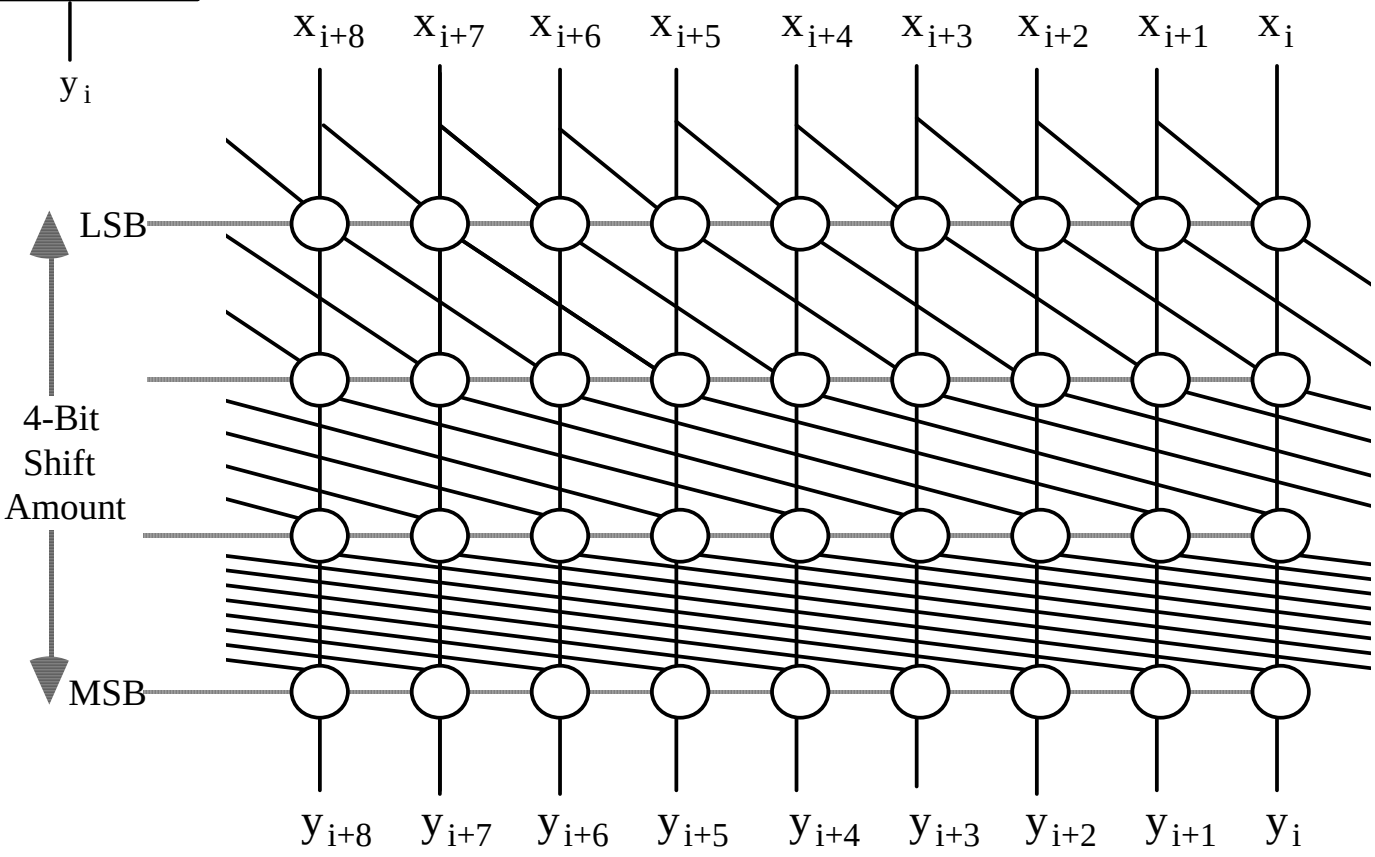


Fig. 18.3 Four-stage combinational shifter for preshifting an operand by 0 to 15 bits.

Leading Zeros/Ones Detection or Prediction

Leading zeros prediction, with adder inputs $(0x_0.x_{-1}x_{-2}\ldots)_{2\text{'s-compl}}$ and $(0y_0.y_{-1}y_{-2}\ldots)_{2\text{'s-compl}}$

Ways in which leading 0s/1s are generated:

p p . . . p p g a a . . . a a g . . .
p p . . . p p g a a . . . a a p . . .
p p . . . p p a g g . . . g g a . . .
p p . . . p p a g g . . . g g p . . .

Prediction might be done in two stages:

- Coarse estimate, used for coarse shift
- Fine tuning of estimate, used for fine shift

In this way, prediction can be partially overlapped with shifting

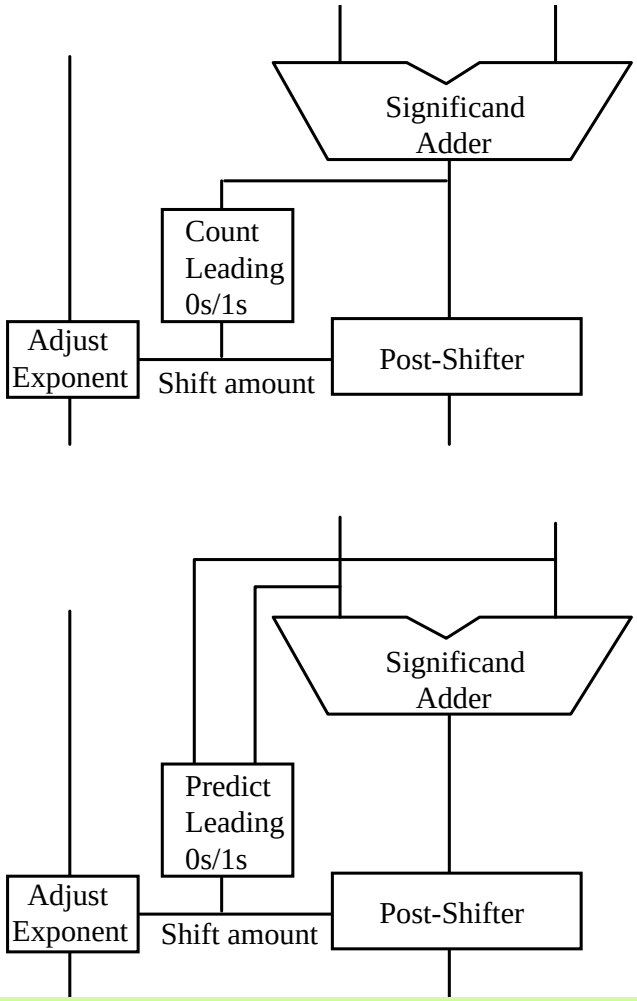


Fig. 18.4 Leading zeros/ones counting versus prediction.

18.3 Rounding and Exceptions

Adder result = $(c_{out} z_1 z_0 \,.\, z_{-1} z_{-2} \ldots z_{-l}\ \mathbf{G}\ \mathbf{R}\ \mathbf{S})_{2\text{'s-compl}}$

Guard bit

Round bit

Sticky bit

OR of all bits shifted past *R*

Why only 3 extra bits?

Amount of alignment right-shift

- One bit: *G* holds the bit that is shifted out, no precision is lost
- Two bits or more: Shifted significand has a magnitude in [0, 1/2)
  - Unshifted significand has a magnitude in [1, 2)
  - Difference of aligned significands has a magnitude in (1/2, 2)
  - Normalization left-shift will be by at most one bit

(1/2, 1) Shift left

[1, 2) No shift

If a normalization left-shift actually takes place:

- *R* = 0, round down, discarded part < *ulp*/2
- *R* = 1, round up, discarded part ≥ *ulp*/2

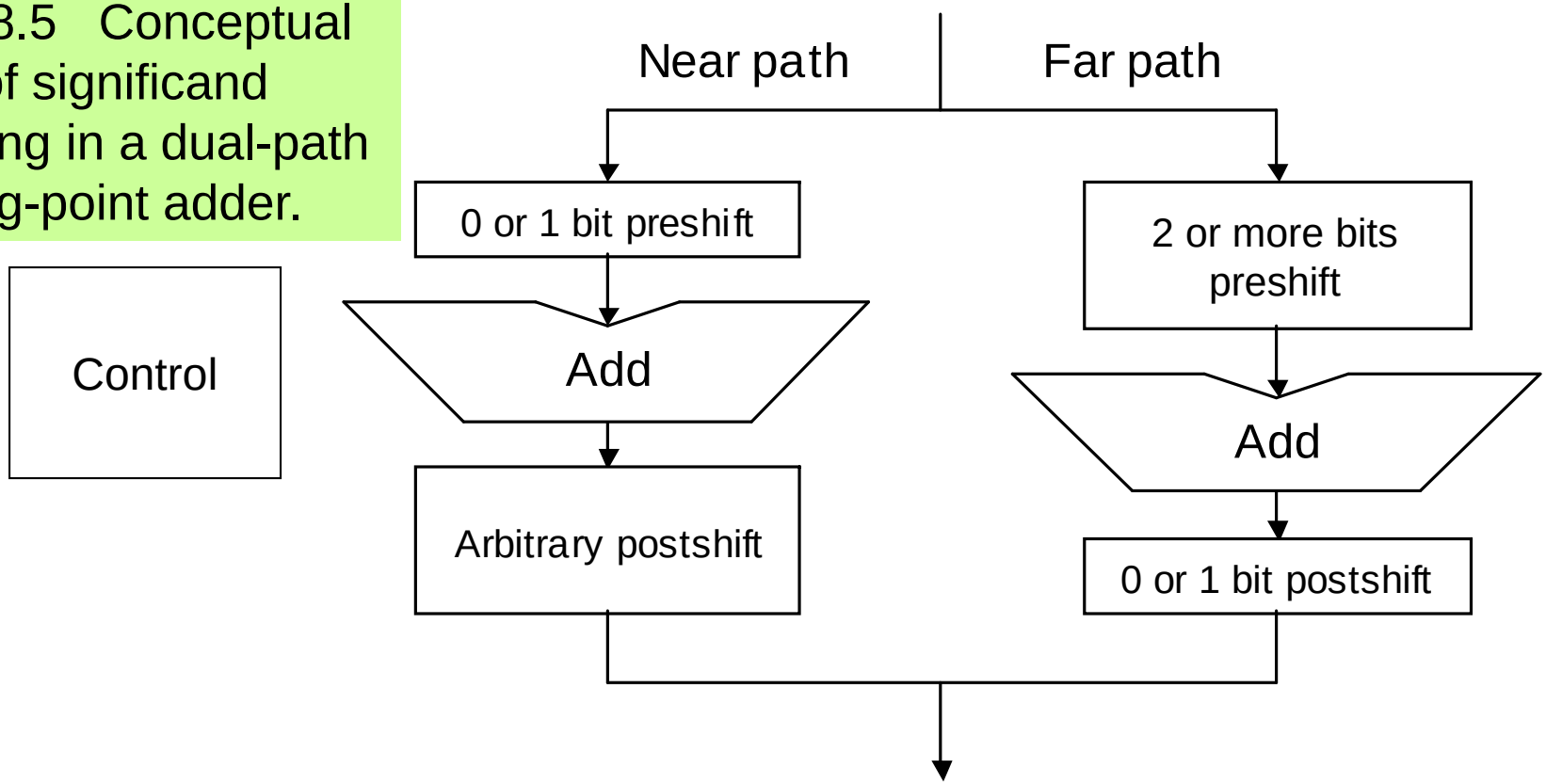
The only remaining question is establishing whether the discarded part is exactly *ulp*/2 (for round to nearest even); *S* provides this information

Floating-Point Adder with Dual Data Paths

Amount of alignment right-shift

One bit: Arbitrary left shift may be needed due to cancellation
Two bits or more: Normalization left-shift will be by at most one bit

Fig. 18.5 Conceptual view of significand handling in a dual-path floating-point adder.



Implementation of Rounding for Addition

The effect of 1-bit normalization shifts on the rightmost few bits of the significand adder output is as follows:

Before postshifting (z)	. . .	z_{-l+1}	z_{-l} |	G	R	S
1-bit normalizing right-shift	. . .	z_{-l+2}	z_{-l+1} |	z_{-l}	G	$R \vee S$
1-bit normalizing left-shift	. . .	z_{-l}	G |	R	S	0
After normalization (Z)	. . .	Z_{-l+1}	Z_{-l} |	Z_{-l-1}	Z_{-l-2}	Z_{-l-3}

Note that no rounding is needed in case of multibit left-shift, because full precision is preserved in this case

Round to nearest even:

Do nothing if $Z_{-l-1} = 0$ or $Z_{-l} = Z_{-l-2} = Z_{-l-3} = 0$
Add $ulp = 2^{-l}$ otherwise

Exceptions in Floating-Point Addition

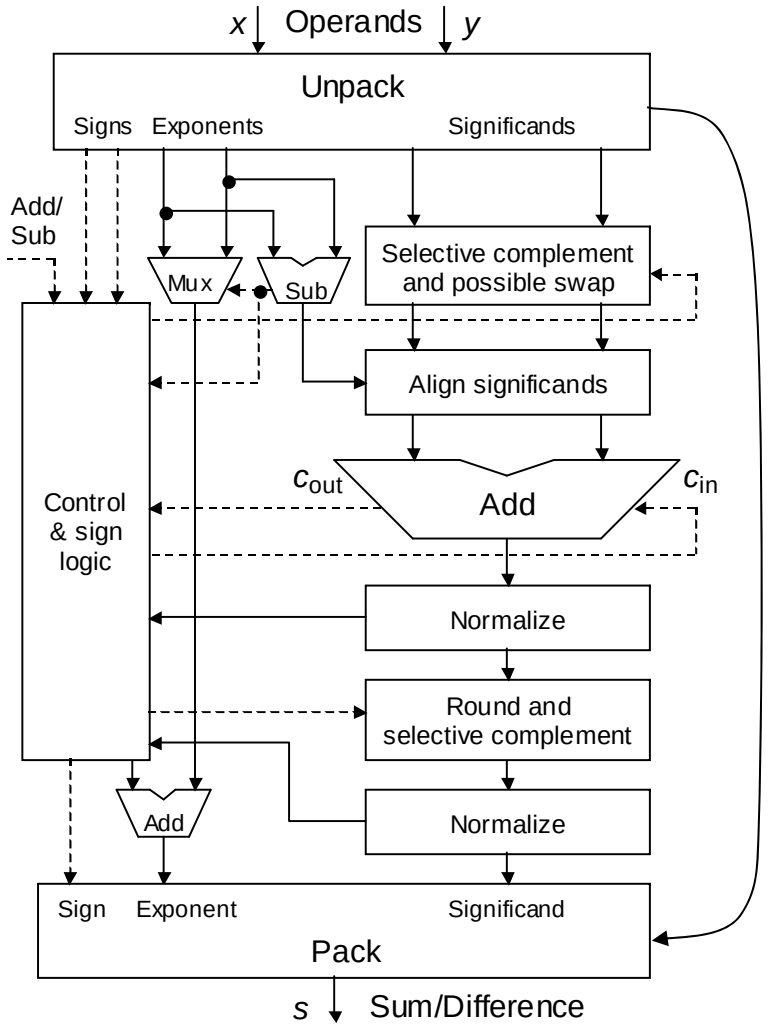
Overflow/underflow detected by exponent adjustment block in Fig. 18.1

- Overflow can occur only for normalizing right-shift
- Underflow possible only with normalizing left shifts

Exceptions involving NaNs and invalid operations handled by unpacking and packing blocks in Fig. 18.1

Zero detection: Special case of leading 0s detection

Determining when “inexact” exception must be signaled left as an exercise



18.4 Floating-Point Multipliers and Dividers

$$(\pm s1 \times b^{e1}) \times (\pm s2 \times b^{e2}) = (\pm s1 \times s2) \times b^{e1+e2}$$

$s1 \times s2 \in [1, 4)$: may need postshifting

Overflow or underflow can occur during multiplication or normalization

Speed considerations

Many multipliers produce the lower half of the product (rounding info) early

Need for normalizing right-shift is known at or near the end

Hence, rounding can be integrated in the generation of the upper half, by producing two versions of these bits

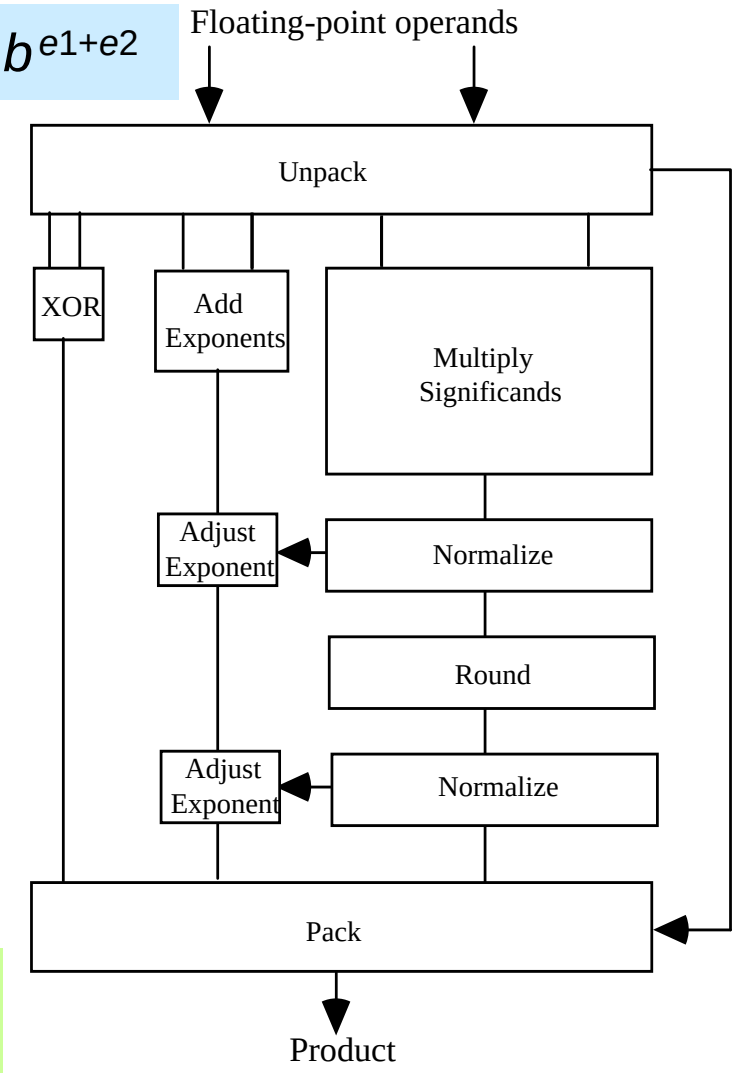


Fig. 18.6 Block diagram of a floating-point multiplier (divider).

Floating-Point Dividers

$$(\pm s1 \times b^{e1}) / (\pm s2 \times b^{e2}) = (\pm s1/s2) \times b^{e1-e2}$$

$s1/s2 \in (0.5, 2)$: may need postshifting

Overflow or underflow can occur during division or normalization

Note: Square-rooting never leads to overflow or underflow

Rounding considerations

Quotient must be produced with two extra bits (*G* and *R*), in case of the need for a normalizing left shift

The remainder acts as the sticky bit

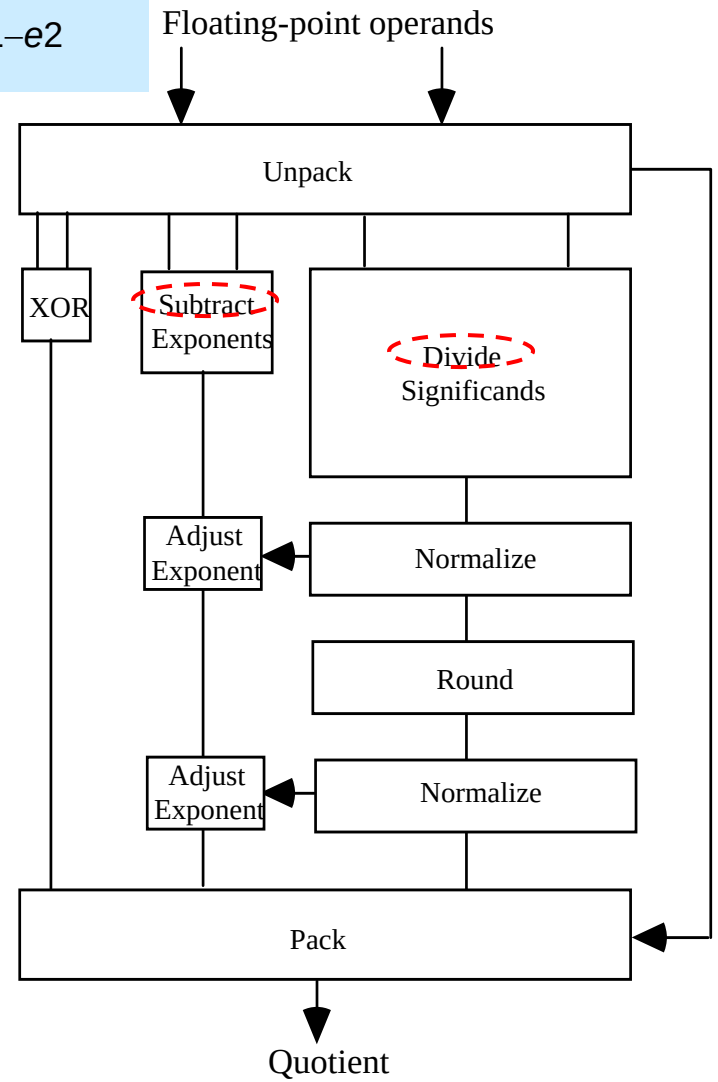


Fig. 18.6 Block diagram of a floating-point multiplier (divider).

18.5 Fused-Multiply-Add Units

Multiply-add operation: $p = ax + b$

The most useful operation beyond the five basic ones

Application 1: Polynomial evaluation

$f(z) = c^{(n-1)}z^{n-1} + c^{(n-2)}z^{n-2} + \ldots + c^{(1)}z + c^{(0)}$

$s := s\,z + c^{(j)}$ for j from $n - 1$ downto 0; initialize s to 0

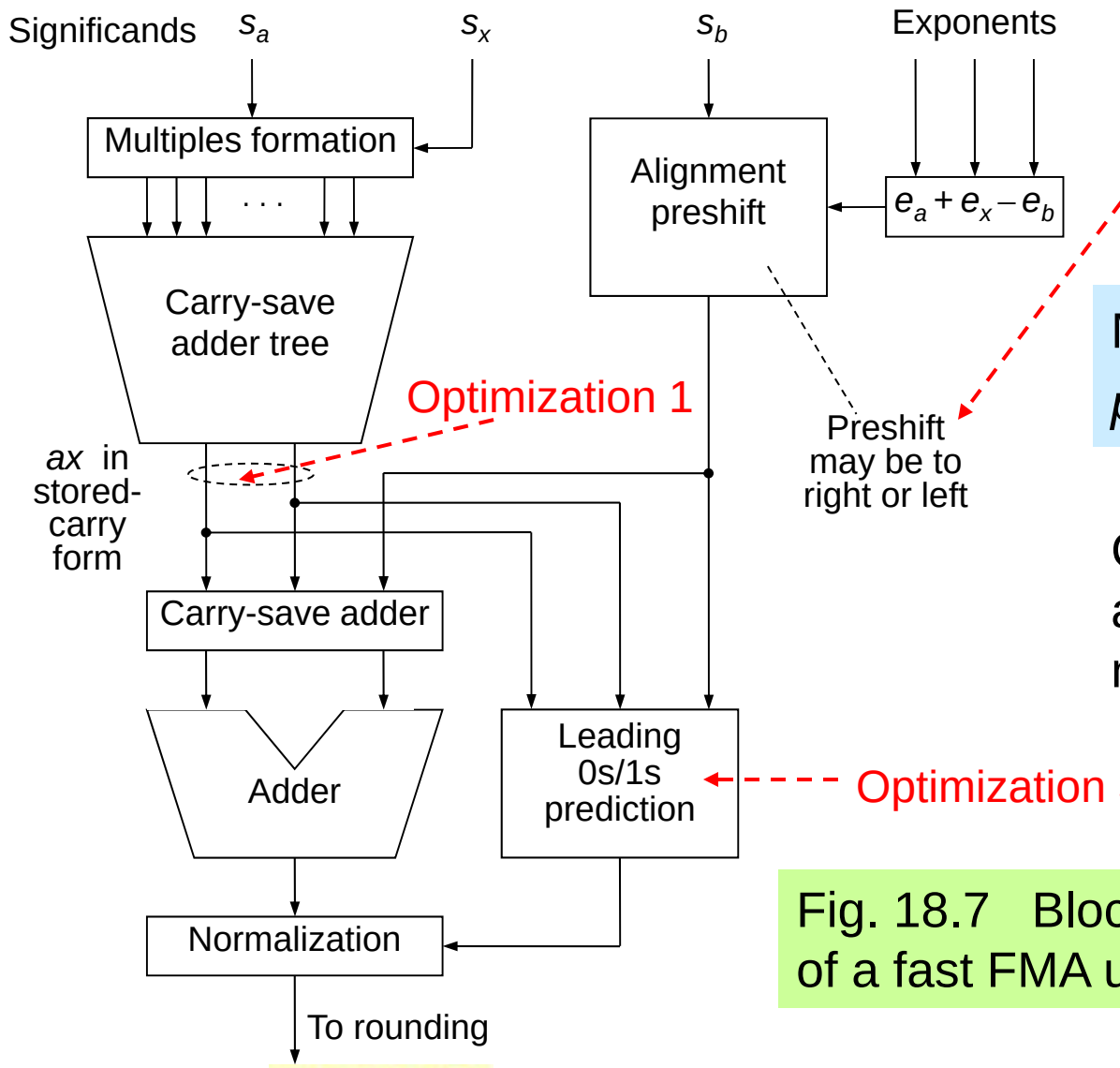
Application 2: Dot-product computation

$u \cdot v = u^{(0)}v^{(0)} + u^{(1)}v^{(1)} + \ldots + u^{(n-1)}v^{(n-1)}$

$s := s + u^{(j)}v^{(j)}$ for j from 0 upto $n - 1$; initialize s to 0

Straightforward implementation: Use a multiplier that keeps its entire double-width product, followed by a double-width adder

Design of a Fast FMA Unit



Multiply-add operation:

$p = ax + b$

Can act as a simple adder ($x = 1$) or multiplier ($b = 0$)

Fig. 18.7 Block diagram of a fast FMA unit.

18.6 Logarithmic Arithmetic Unit

Multiply/divide algorithm in LNS

$$\log(x\,y) = \log x + \log y$$
$$\log(x/y) = \log x - \log y$$

Add/subtract algorithm in LNS

$$(Sx, Lx) \pm (Sy, Ly) = (Sz, Lz)$$

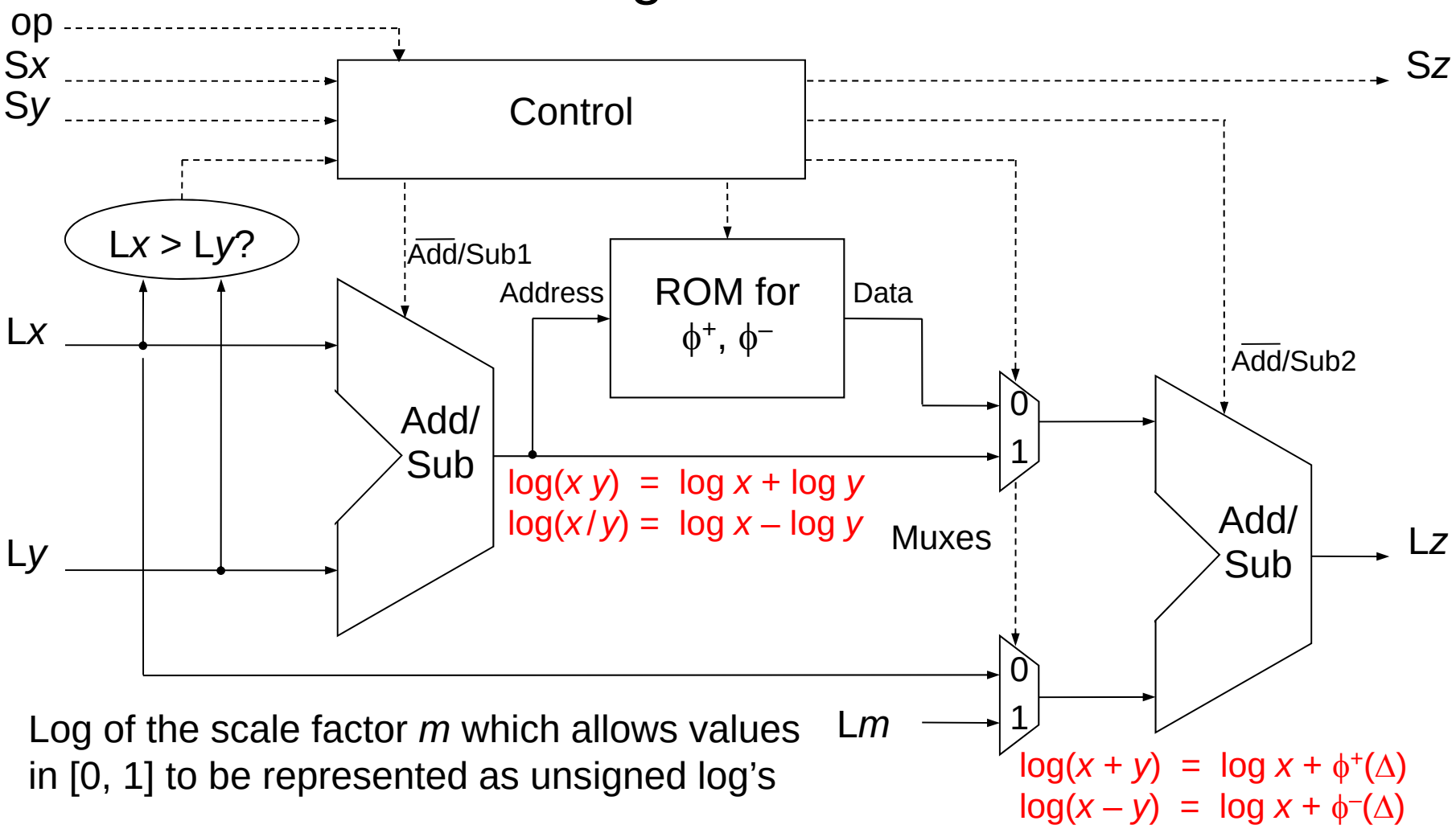
Assume $x > y > 0$ (other cases are similar)

$$Lz = \log z = \log(x \pm y) = \log(x(1 \pm y/x)) = \log x + \log(1 \pm y/x)$$

Given $\Delta = -(\log x - \log y)$, the term $\log(1 \pm y/x) = \log(1 \pm \log^{-1}\Delta)$ is obtained from a table (two tables ϕ^+ and ϕ^- needed)

$$\log(x + y) = \log x + \phi^+(\Delta)$$
$$\log(x - y) = \log x + \phi^-(\Delta)$$

Four-Function Logarithmic Arithmetic Unit



Log of the scale factor *m* which allows values in [0, 1] to be represented as unsigned log's

Fig. 18.8 Arithmetic unit for a logarithmic number system.

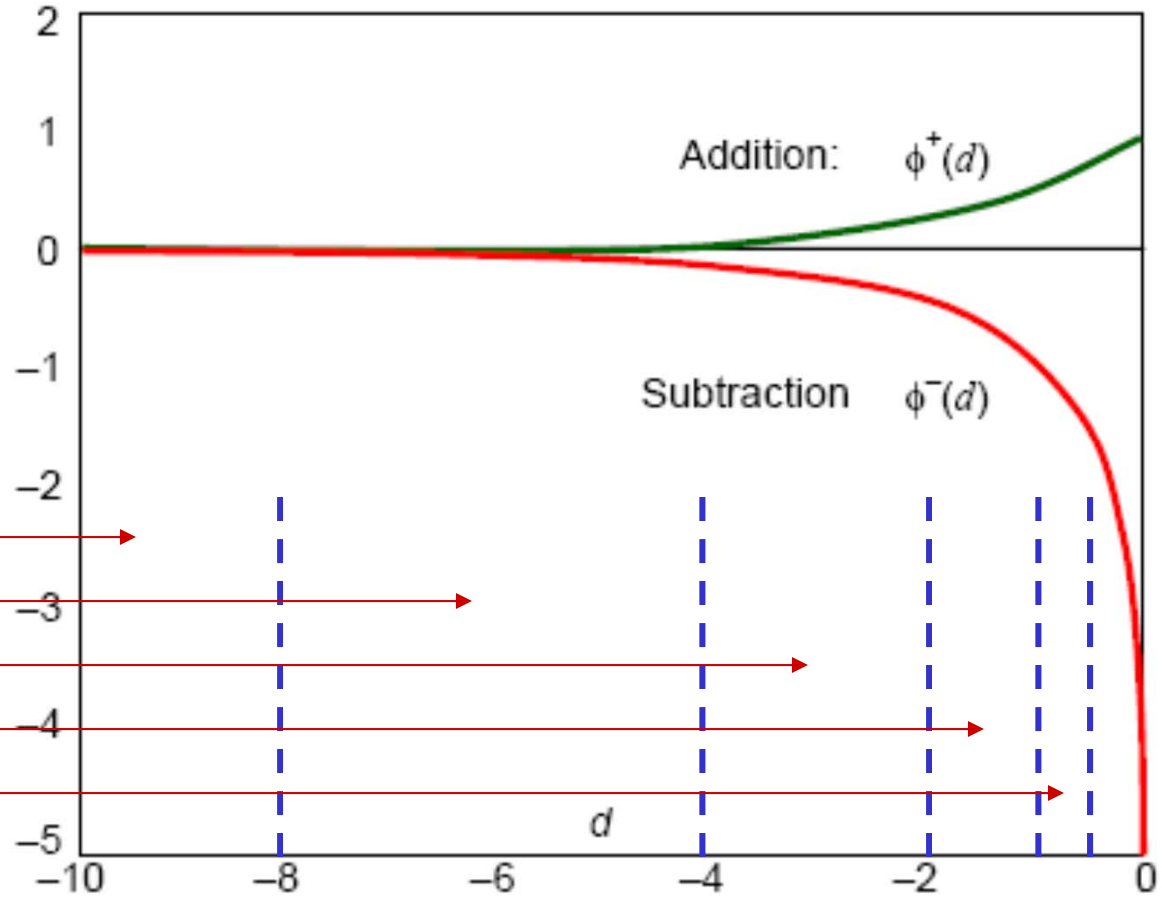
LNS Arithmetic for Wider Words

$\log(x + y) = \log x + \phi^+(\Delta)$
$\log(x - y) = \log x + \phi^-(\Delta)$

ϕ^+ is well-behaved; easy to interpolate
ϕ^- causes difficulties in [–1, 0]

Use nonuniform segmentation for direct table lookup or for a scheme based on linear interpolation

10xxx.xxxxxxx
110xx.xxxxxxx
1110x.xxxxxxx
11110.xxxxxxx
11111.0xxxxxx
. . .



19 Errors and Error Control

Chapter Goals

Learn about sources of computation errors,
consequences of inexact arithmetic,
and methods for avoiding or limiting errors

Chapter Highlights

Representation and computation errors
Absolute versus relative error
Worst-case versus average error
Why $3 \times (1/3)$ does not necessarily yield 1
Error analysis and bounding

Errors and Error Control: Topics

Topics in This Chapter
19.1 Sources of Computational Errors
19.2 Invalidated Laws of Algebra
19.3 Worst-Case Error Accumulation
19.4 Error Distribution and Expected Errors
19.5 Forward Error Analysis
19.6 Backward Error Analysis

19.1 Sources of Computational Errors

FLP approximates exact computation with real numbers

Two sources of errors to understand and counteract:

Representation errors

e.g., no machine representation for 1/3, $\sqrt{2}$, or π

Arithmetic errors

e.g., $(1 + 2^{-12})^2 = 1 + 2^{-11} + 2^{-24}$
not representable in IEEE 754 short format

We saw early in the course that errors due to finite precision can lead to disasters in life-critical applications

Example Showing Representation and Arithmetic Errors

Example 19.1: Compute 1/99 – 1/100, using a decimal floating-point format with 4-digit significand in [1, 10) and single-digit signed exponent

Precise result = $1/9900 \cong 1.010 \times 10^{-4}$ (error $\cong 10^{-8}$ or 0.01%)

Chopped to 3 decimals

$x = 1/99 \cong 1.010 \times 10^{-2}$ Error $\cong 10^{-6}$ or 0.01%

$y = 1/100 = 1.000 \times 10^{-2}$ Error = 0

$z = x -_{fp} y = 1.010 \times 10^{-2} - 1.000 \times 10^{-2} = 1.000 \times 10^{-4}$

Error $\cong 10^{-6}$ or 1%

Notation for a General Floating-Point System

Number representation in FLP(*r*, *p*, A)

Radix r (assume to be the same as the exponent base b)
Precision p in terms of radix-r digits
Approximation scheme A $\in$ {chop, round, rtne, chop(g), . . .}

Let $x = r^e s$ be an unsigned real number, normalized such that $1/r \leq s < 1$, and assume x_{fp} is the representation of x in FLP(r, p, A)

$x_{fp} = r^e s_{fp} = (1 + \eta)x$ $\quad$ η is the relative representation error

A = chop	$-ulp < s_{fp} - s \leq 0$	$-r \times ulp < \eta \leq 0$
A = round	$-ulp/2 < s_{fp} - s \leq ulp/2$	$\lvert\eta\rvert \leq r \times ulp/2$

Arithmetic in FLP(*r*, *p*, A)

Obtain an infinite-precision result, then chop, round, . . .

Real machines approximate this process by keeping $g > 0$ guard digits, thus doing arithmetic in FLP(r, p, chop(g))

Error Analysis for Multiplication and Division

Errors in floating-point multiplication

Consider the positive operands x_{fp} and y_{fp}

$$\begin{aligned} x_{fp} \times_{fp} y_{fp} &= (1 + \eta) x_{fp} y_{fp} \\ &= (1 + \eta)(1 + \sigma)(1 + \tau) xy \\ &= (1 + \eta + \sigma + \tau + \eta\sigma + \eta\tau + \sigma\tau + \eta\sigma\tau) xy \\ &\cong (1 + \eta + \sigma + \tau) xy \end{aligned}$$

Errors in floating-point division

Again, consider positive operands x_{fp} and y_{fp}

$$\begin{aligned} x_{fp} /_{fp} y_{fp} &= (1 + \eta)\, x_{fp} / y_{fp} \\ &= (1 + \eta)(1 + \sigma) x / [(1 + \tau) y] \\ &= (1 + \eta)(1 + \sigma)(1 - \tau)(1 + \tau^2)(1 + \tau^4)(\ldots) x/y \\ &\cong (1 + \eta + \sigma - \tau) x/y \end{aligned}$$

Error Analysis for Addition and Subtraction

Errors in floating-point addition

Consider the positive operands x_{fp} and y_{fp}

$$\begin{aligned} x_{fp} +_{fp} y_{fp} &= (1 + \eta)(x_{fp} + y_{fp}) \\ &= (1 + \eta)(x + \sigma x + y + \tau y) \\ &= (1 + \eta)\left(1 + \frac{\sigma x + \tau y}{x + y}\right)(x + y) \end{aligned}$$

Magnitude of this ratio is upper-bounded by $max(|\sigma|, |\tau|)$, so the overall error is no more than $|\eta| + max(|\sigma|, |\tau|)$

Errors in floating-point subtraction

Again, consider positive operands x_{fp} and y_{fp}

$$\begin{aligned} x_{fp} -_{fp} y_{fp} &= (1 + \eta)(x_{fp} - y_{fp}) \\ &= (1 + \eta)(x + \sigma x - y - \tau y) \\ &= (1 + \eta)\left(1 + \frac{\sigma x - \tau y}{x - y}\right)(x - y) \end{aligned}$$

This term also unbounded for subtraction

Magnitude of this ratio can be very large if x and y are both large but $x - y$ is relatively small (recall that τ can be negative)

Cancellation Error in Subtraction

$$x_{fp} -_{fp} y_{fp} = (1 + \eta)\left(1 + \frac{\sigma x - \tau y}{x - y}\right)(x - y)$$ Subtraction result

Example 19.2: Decimal FLP system, $r = 10$, $p = 6$, no guard digit

$x = 0.100\ 000\ 000 \times 10^3$ $\quad$ $y = -0.999\ 999\ 456 \times 10^2$

$x_{fp} = .100\ 000 \times 10^3$ $\quad$ $y_{fp} = -.999\ 999 \times 10^2$

$x + y = 0.544 \times 10^{-4}$ and $x_{fp} + y_{fp} = 0.1 \times 10^{-3}$

$x_{fp} +_{fp} y_{fp} = 0.100\ 000 \times 10^3 -_{fp} 0.099\ 999 \times 10^3 = 0.100\ 000 \times 10^{-2}$

Relative error $= (10^{-3} - 0.544 \times 10^{-4}) / (0.544 \times 10^{-4}) \cong 17.38 = 1738\%$

Now, ignore representation errors, so as to focus on the effect of η (measure relative error with respect to $x_{fp} + y_{fp}$, not $x + y$)

Relative error $= (10^{-3} - 10^{-4}) / 10^{-4} = 9 = 900\%$

Bringing Cancellation Errors in Check

$$x_{fp} -_{fp} y_{fp} = (1 + \eta)(1 + \frac{\sigma x - \tau y}{x - y})(x - y)$$ Subtraction result

Example 19.2 (cont.): Decimal FLP system, $r = 10$, $p = 6$, 1 guard digit

$x = 0.100\ 000\ 000 \times 10^3$ $\quad$ $y = -0.999\ 999\ 456 \times 10^2$

$x_{fp} = .100\ 000 \times 10^3$ $\quad$ $y_{fp} = -.999\ 999 \times 10^2$

$x + y = 0.544 \times 10^{-4}$ and $x_{fp} + y_{fp} = 0.1 \times 10^{-3}$

$x_{fp} +_{fp} y_{fp} = 0.100\ 000 \times 10^3 -_{fp} 0.099\ 999\ \mathbf{9} \times 10^3 = 0.100\ 000 \times 10^{-3}$

Relative error $= (10^{-4} - 0.544 \times 10^{-4}) / (0.544 \times 10^{-4}) \cong 0.838 = 83.8\%$

Now, ignore representation errors, so as to focus on the effect of η (measure relative error with respect to $x_{fp} + y_{fp}$, not $x + y$)

Relative error = 0 $\quad$ Significantly better than 900%!

How Many Guard Digits Do We Need?

$$x_{fp} -_{fp} y_{fp} = (1 + \eta)\left(1 + \frac{\sigma x - \tau y}{x - y}\right)(x - y) \qquad \text{Subtraction result}$$

Theorem 19.1: In the floating-point system FLP(r, p, chop(g)) with $g \geq 1$ and $-x < y < 0 < x$, we have:

$x_{fp} +_{fp} y_{fp} = (1 + \eta)(x_{fp} + y_{fp})$ with $-r^{-p+1} < \eta < r^{-p-g+2}$

Corollary: In FLP(r, p, chop(1))

$x_{fp} +_{fp} y_{fp} = (1 + \eta)(x_{fp} + y_{fp})$ with $|\eta| < -r^{-p+1}$

So, a single guard digit is sufficient to make the relative arithmetic error in floating-point addition or subtraction comparable to relative representation error with truncation

19.2 Invalidated Laws of Algebra

Many laws of algebra do not hold for floating-point arithmetic (some don't even hold approximately)

This can be a source of confusion and incompatibility

Associative law of addition: $\boldsymbol{a + (b + c) = (a + b) + c}$

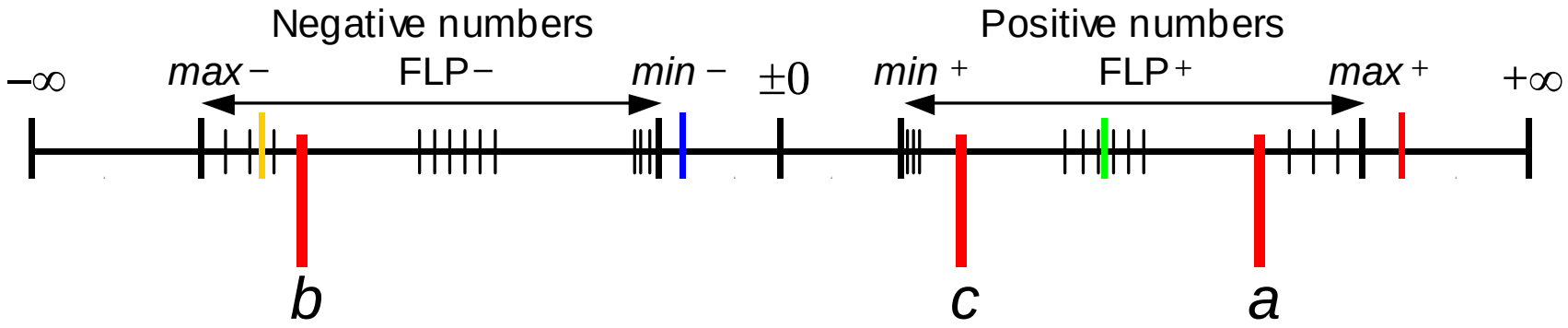
$a = 0.123\ 41 \times 10^5 \quad b = -0.123\ 40 \times 10^5 \quad c = 0.143\ 21 \times 10^1$

$$
\begin{aligned}
&a +_{fp} (b +_{fp} c) \\
&\quad = 0.123\ 41 \times 10^5 +_{fp} (-0.123\ 40 \times 10^5 +_{fp} 0.143\ 21 \times 10^1) \\
&\quad = 0.123\ 41 \times 10^5 -_{fp} 0.123\ 39 \times 10^5 \\
&\quad = 0.200\ 00 \times 10^1
\end{aligned}
$$

$$
\begin{aligned}
&(a +_{fp} b) +_{fp} c \\
&\quad = (0.123\ 41 \times 10^5 -_{fp} 0.123\ 40 \times 10^5) +_{fp} 0.143\ 21 \times 10^1 \\
&\quad = 0.100\ 00 \times 10^1 +_{fp} 0.143\ 21 \times 10^1 \\
&\quad = 0.243\ 21 \times 10^1
\end{aligned}
$$

Results differ by more than 20%!

Elaboration on the Non-Associativity of Addition



Associative law of addition: $a + (b + c) = (a + b) + c$

s_1 s_2

$a = 0.123\ 41 \times 10^5$ $b = -0.123\ 40 \times 10^5$ $c = 0.143\ 21 \times 10^1$

When we first compute $s_1 = b + c$, the small value of c barely makes a dent, yielding a value for $a + s_1$ that is not much affected by c

When we first compute $s_2 = a + b$, the result will be nearly 0, making the effect of c on the final sum $s_2 + c$ more pronounced

Do Guard Digits Help with Laws of Algebra?

Invalidated laws of algebra are intrinsic to FLP arithmetic; problems are reduced, but don't disappear, with guard digits

Let's redo our example with 2 guard digits

Associative law of addition: $\boldsymbol{a + (b + c) = (a + b) + c}$

$a = 0.123\ 41 \times 10^5 \quad b = -0.123\ 40 \times 10^5 \quad c = 0.143\ 21 \times 10^1$

$a +_{fp} (b +_{fp} c)$

$= 0.123\ 41 \times 10^5 +_{fp} (-0.123\ 40 \times 10^5 +_{fp} 0.143\ 21 \times 10^1)$

$= 0.123\ 41 \times 10^5 -_{fp} 0.123\ 385\ 7 \times 10^5$

$= 0.243\ 00 \times 10^1$

Difference of about 0.1% is better, but still too high!

$(a +_{fp} b) +_{fp} c$

$= (0.123\ 41 \times 10^5 -_{fp} 0.123\ 40 \times 10^5) +_{fp} 0.143\ 21 \times 10^1$

$= 0.100\ 00 \times 10^1 +_{fp} 0.143\ 21 \times 10^1$

$= 0.243\ 21 \times 10^1$

Unnormalized Floating-Point Arithmetic

One way to reduce problems resulting from invalidated laws of algebra is to avoid normalizing computed floating-point results

Let's redo our example with unnormalized arithmetic

Associative law of addition: $\boldsymbol{a + (b + c) = (a + b) + c}$

$a = 0.123\ 41 \times 10^5 \quad b = -0.123\ 40 \times 10^5 \quad c = 0.143\ 21 \times 10^1$

$$a +_{fp} (b +_{fp} c)$$
$$= 0.123\ 41 \times 10^5 +_{fp} (-0.123\ 40 \times 10^5 +_{fp} 0.143\ 21 \times 10^1)$$
$$= 0.123\ 41 \times 10^5 -_{fp} 0.123\ 39 \times 10^5$$
$$= 0.000\ 02 \times 10^5$$

$$(a +_{fp} b) +_{fp} c$$
$$= (0.123\ 41 \times 10^5 -_{fp} 0.123\ 40 \times 10^5) +_{fp} 0.143\ 21 \times 10^1$$
$$= 0.000\ 01 \times 10^5 +_{fp} 0.143\ 21 \times 10^1$$
$$= 0.000\ 02 \times 10^5$$

Results are the same and also carry a kind of warning

Other Invalidated Laws of Algebra with FLP Arithmetic

Associative law of multiplication	$a \times (b \times c) = (a \times b) \times c$
Cancellation law (for $a > 0$)	$a \times b = a \times c$ implies $b = c$
Distributive law	$a \times (b + c) = (a \times b) + (a \times c)$
Multiplication canceling division	$a \times (b / a) = b$

Before the IEEE 754 floating-point standard became available and widely adopted, these problems were exacerbated by the use of many incompatible formats

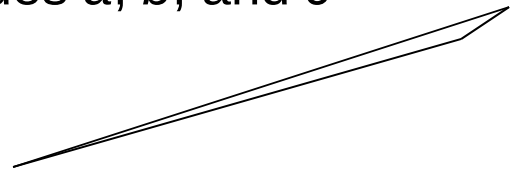
Effects of Algorithms on Result Precision

Example 19.3: The formula $x = -b \pm d$, with $d = (b^2 - c)^{1/2}$, yielding the roots of the quadratic equation $x^2 + 2bx + c = 0$, can be rewritten as $x = -c/(b \pm d)$

When c is small compared with b^2, the root $-b + d$ will have a large error due to cancellation; in such a case, use $-c/(b + d)$ for that root

Confirmation that $-b + d = -c/(b + d) \Rightarrow -c = d^2 - b^2$

Example 19.4: The area of a triangle with sides a, b, and c (assume $a \geq b \geq c$) is given by the formula



$$A = [s(s - a)(s - b)(s - c)]^{1/2}$$

where $s = (a+b+c)/2$. When the triangle is very flat (needlelike), such that $a \cong b + c$, Kahan's version returns accurate results:

$$A = \tfrac{1}{4}[(a + (b + c))(c - (a - b))(c + (a - b))(a + (b - c))]^{1/2}$$

19.3 Worst-Case Error Accumulation

In a sequence of operations, round-off errors might add up

The larger the number of cascaded computation steps (that depend on results from previous steps), the greater the chance for, and the magnitude of, accumulated errors

With rounding, errors of opposite signs tend to cancel each other out in the long run, but one cannot count on such cancellations

Practical implications:

Perform intermediate computations with a higher precision than what is required in the final result

Implement multiply-accumulate in hardware (DSP chips)

Reduce the number of cascaded arithmetic operations; So, using computationally more efficient algorithms has the double benefit of reducing the execution time as well as accumulated errors

Example: Inner-Product Calculation

Consider the computation $z = \sum x^{(i)} y^{(i)}$, for $i \in [0, 1023]$

Max error per multiply-add step = $ulp/2 + ulp/2 = ulp$

Total worst-case absolute error = $1024\, ulp$
(equivalent to losing 10 bits of precision)

A possible cure: keep the double-width products in their entirety and add them to compute a double-width result which is rounded to single-width at the very last step

Multiplications do not introduce any round-off error
Max error per addition = $ulp^2/2$
Total worst-case error = $1024 \times ulp^2/2 + ulp/2$

Therefore, provided that overflow is not a problem, a highly accurate result is obtained

Kahan's Summation Algorithm

To compute $s = \sum x^{(i)}$, for $i \in [0, n-1]$, more accurately:

$s \leftarrow x^{(0)}$
$c \leftarrow 0$ {c is a correction term}
for $i = 1$ to $n - 1$ do
    $y \leftarrow x^{(i)} - c$ {subtract correction term}
    $z \leftarrow s + y$
    $c \leftarrow (z - s) - y$ {find next correction term}
    $s \leftarrow z$
endfor

19.4 Error Distribution and Expected Errors

Probability density function for the distribution of radix-r floating-point significands is $1/(x \ln r)$

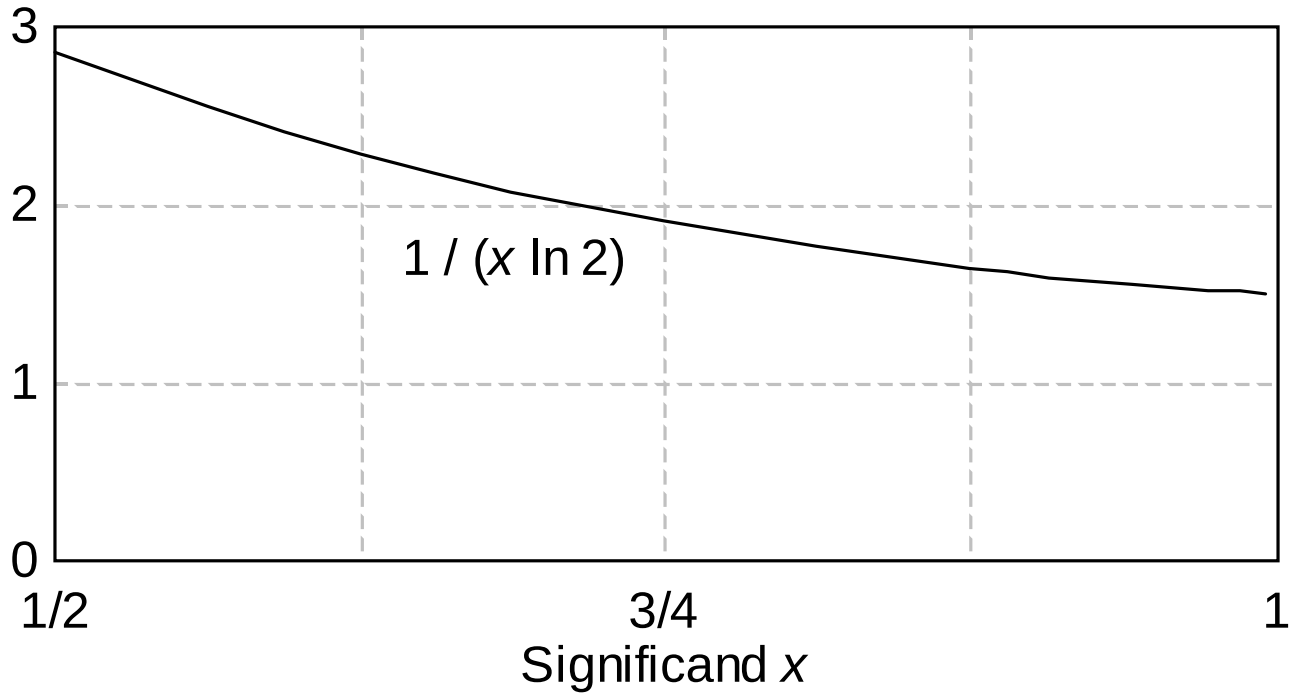


Fig. 19.1 Probability density function for the distribution of normalized significands in FLP(r = 2, p, A).

Maximum Relative Representation Error

MRRE = maximum relative representation error

$$\text{MRRE(FLP}(r, p, \text{chop})) = r^{-p+1}$$
$$\text{MRRE(FLP}(r, p, \text{round})) = r^{-p+1}/2$$

From a practical standpoint, the distribution of errors and their expected values may be more important

Limiting ourselves to positive significands, we define:

$$\text{ARRE(FLP}(r, p, \text{A})) = \int_{1/r}^{1} \frac{|x_{fp} - x|}{x} \frac{dx}{x \ln r}$$

$1/(x \ln r)$ is a probability density function

19.5 Forward Error Analysis

Consider the computation $y = ax + b$ and its floating-point version

$$y_{fp} = (a_{fp} \times_{fp} x_{fp}) +_{fp} b_{fp} = (1 + \eta)y$$

Can we establish any useful bound on the magnitude of the relative error η, given the relative errors in the input operands a_{fp}, b_{fp}, x_{fp}?

The answer is “no”

Forward error analysis =

Finding out how far y_{fp} can be from $ax + b$,
or at least from $a_{fp}x_{fp} + b_{fp}$, in the worst case

Some Error Analysis Methods

Automatic error analysis

Run selected test cases with higher precision and observe differences between the new, more precise, results and the original ones

Significance arithmetic

Roughly speaking, same as unnormalized arithmetic, although there are fine distinctions. The result of the unnormalized decimal addition $.1234 \times 10^5 +_{fp} .0000 \times 10^{10} = .0000 \times 10^{10}$ warns us about precision loss

Noisy-mode computation

Random digits, rather than 0s, are inserted during normalizing left shifts
If several runs of the computation in noisy mode yield comparable results, then we are probably safe

Interval arithmetic

An interval $[x_{lo}, x_{hi}]$ represents x, $x_{lo} \le x \le x_{hi}$. With x_{lo}, x_{hi}, y_{lo}, $y_{hi} > 0$, to find $z = x / y$, we compute $[z_{lo}, z_{hi}] = [x_{lo} /_{\nabla fp} y_{hi}, x_{hi} /_{\Delta fp} y_{lo}]$
Drawback: Intervals tend to widen after many computation steps

19.6 Backward Error Analysis

Backward error analysis replaces the original question

How much does $y_{fp} = a_{fp} \times_{fp} x_{fp} + b_{fp}$ deviate from y?

with another question:

What input changes produce the same deviation?

In other words, if the exact identity $y_{fp} = a_{alt} x_{alt} + b_{alt}$ holds for alternate parameter values a_{alt}, b_{alt}, and x_{alt}, we ask how far a_{alt}, b_{alt}, x_{alt} can be from a_{fp}, x_{fp}, x_{fp}

Thus, computation errors are converted or compared to additional input errors

Example of Backward Error Analysis

$$
\begin{aligned}
y_{fp} &= a_{fp} \times_{fp} x_{fp} +_{fp} b_{fp} \\
&= (1 + \mu)[a_{fp} \times_{fp} x_{fp} + b_{fp}] && \text{with } |\mu| < r - p + 1 = r \times ulp \\
&= (1 + \mu)[(1 + \nu)\, a_{fp}\, x_{fp} + b_{fp}] && \text{with } |\nu| < r - p + 1 = r \times ulp \\
&= (1 + \mu)\, a_{fp}\, (1 + \nu)\, x_{fp} + (1 + \mu)\, b_{fp} \\
&= (1 + \mu)(1 + \sigma) a\, (1 + \nu)(1 + \delta) x + (1 + \mu)(1 + \gamma) b \\
&\cong (1 + \sigma + \mu) a\, (1 + \delta + \nu) x + (1 + \gamma + \mu) b
\end{aligned}
$$

So the approximate solution of the original problem is the exact solution of a problem close to the original one

The analysis assures us that the effect of arithmetic errors on the result y_{fp} is no more severe than that of $r \times ulp$ additional error in each of the inputs a, b, and x

20 Precise and Certifiable Arithmetic

Chapter Goals

Discuss methods for doing arithmetic
when results of high accuracy
or guaranteed correctness are required

Chapter Highlights

More precise computation through
multi- or variable-precision arithmetic
Result certification by means of
exact or error-bounded arithmetic
Precise / exact arithmetic with low overhead

Precise and Certifiable Arithmetic: Topics

Topics in This Chapter
20.1 High Precision and Certifiability
20.2 Exact Arithmetic
20.3 Multiprecision Arithmetic
20.4 Variable-Precision Arithmetic
20.5 Error-Bounding via Interval Arithmetic
20.6 Adaptive and Lazy Arithmetic

20.1 High Precision and Certifiability

There are two aspects of precision to discuss:

Results possessing adequate precision

Being able to provide assurance of the same

We consider 3 distinct approaches for coping with precision issues:

1. Obtaining completely trustworthy results via exact arithmetic
2. Making the arithmetic highly precise to raise our confidence in the validity of the results: multi- or variable-precision arith
3. Doing ordinary or high-precision calculations, while tracking potential error accumulation (can lead to fail-safe operation)

We take the hardware to be completely trustworthy
Hardware reliability issues dealt with in Chapter 27

20.2 Exact Arithmetic

Continued fractions

Any unsigned rational number $x = p/q$ has a unique continued-fraction expansion with $a_0 \geq 0$, $a_m \geq 2$, and $a_i \geq 1$ for $1 \leq i \leq m - 1$

$$x = \frac{p}{q} = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots \cfrac{1}{a_{m-1} + \cfrac{1}{a_m}}}}}$$

$$\frac{277}{642} = 0 + \cfrac{1}{2 + \cfrac{1}{3 + \cfrac{1}{6 + \cfrac{1}{1 + \cfrac{1}{3 + \cfrac{1}{3}}}}}} = [\underbrace{0}_{0}/\underbrace{2}_{1/2}/\underbrace{3/6/1/3}_{3/7}/\underbrace{3}_{19/44}]$$

Example: Continued fraction representation of 277/642

Can get approximations for finite representation by limiting the number of "digits" in the continued-fraction representation

Fixed-Slash Number Systems

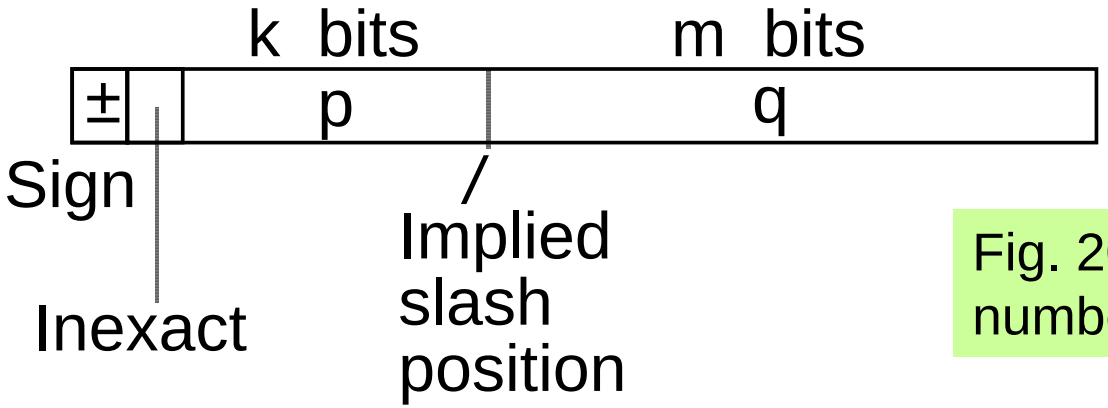


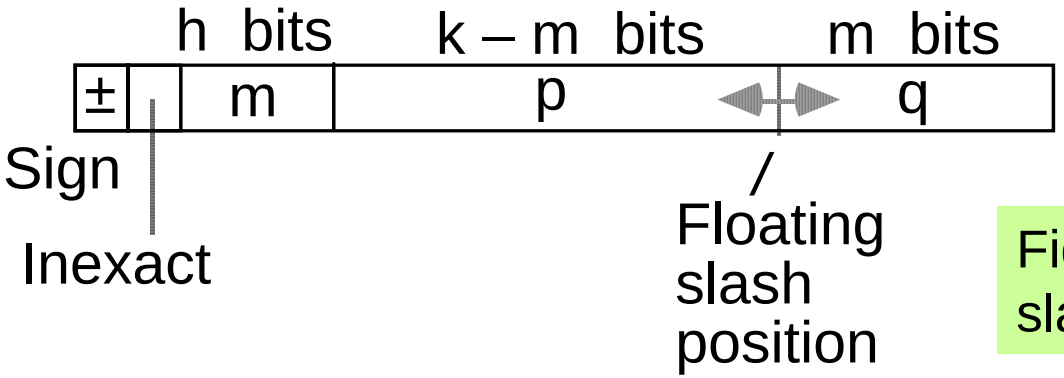
Fig. 20.1 Example fixed-slash number representation format.

Rational number	if $p > 0$ $q > 0$	"rounded" to nearest value
± 0	if $p = 0$ q odd	
$\pm\infty$	if p odd $q = 0$	
NaN (not a number)	otherwise	

Waste due to multiple representations such as 3/5 = 6/10 = 9/15 = . . . is no more than one bit, because:

$$\lim_{n\to\infty} |\{p/q \mid 1 \le p,q \le n, \gcd(p, q) = 1\}|/n^2 = 6/\pi^2 = 0.608$$

Floating-Slash Number Systems



Represents p/q

Fig. 20.2 Example floating-slash representation format.

Set of numbers represented:

$$\{\pm p/q \mid p,q \geq 1,\ \gcd(p, q) = 1,\ \lfloor \log_2 p \rfloor + \lfloor \log_2 q \rfloor \leq k - 2\}$$

Again the following mathematical result, due to Dirichlet, shows that the space waste is no more than one bit:

$$\lim_{n\to\infty} |\{p/q \mid pq \leq n,\ \gcd(p,q) = 1\}| \ / \ |\{p/q \mid pq \leq n,\ p,q \geq 1\}| = 6/\pi^2 = 0.608$$

20.3 Multiprecision Arithmetic

Fig. 20.3 Example quadruple-precision integer format.

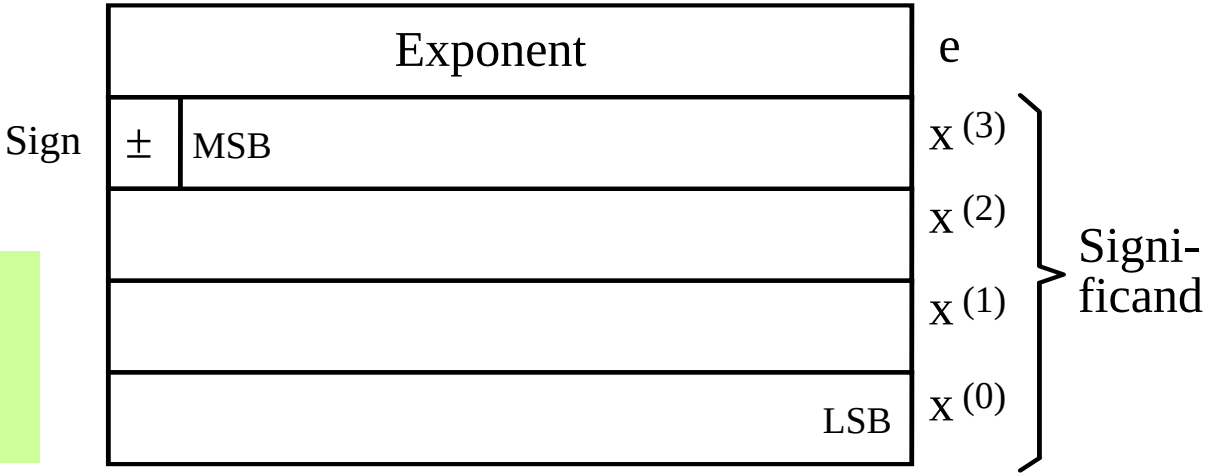


Fig. 20.4 Example quadruple-precision floating-point format.

Multiprecision Floating-Point Addition

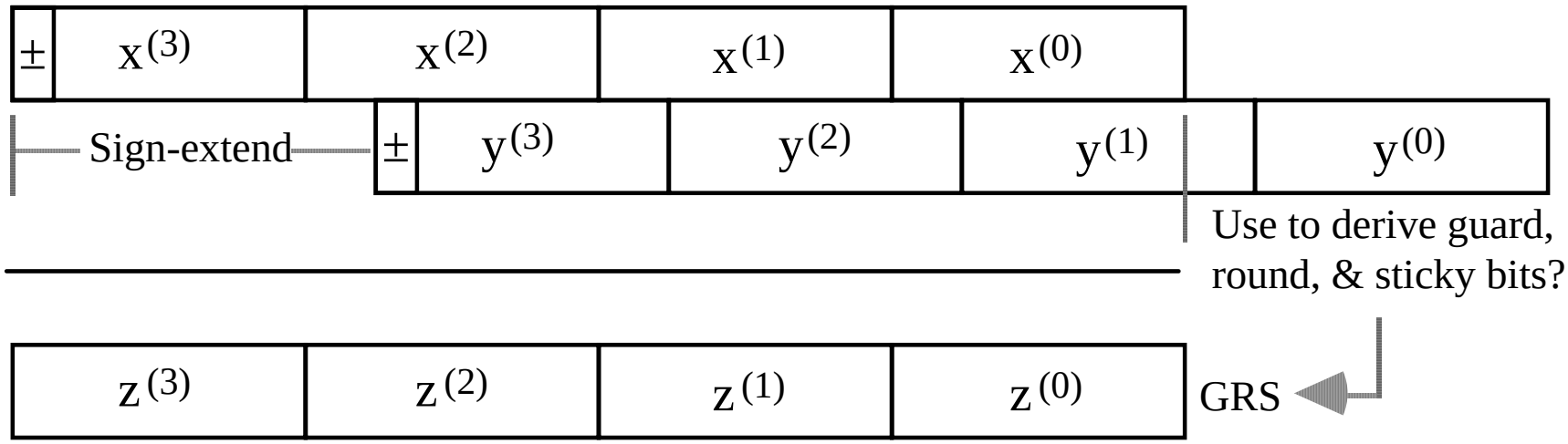


Fig. 20.5 Quadruple-precision significands aligned for the floating-point addition $z = x +_{fp} y$.

Quad-Precision Arithmetic Using Two Doubles

$x_H = 1.011100\ldots101 \times 2^{20}$

$x_L = 1.110101\ldots110 \times 2^{-33}$

$x = x_H + x_L$

$x = 1.011100\ldots101\ 1110101\ldots110 \times 2^{20}$

Key idea used: One can obtain an accurate sum for two floating-point numbers by computing their regular sum $s = x +_{fp} y$ and an error term $e = y - (s - x)$

The following website provides links to downloadable software packages for double-double and quad-double arithmetic

http://crd.lbl.gov/~dhbailey/mpdist/

20.4 Variable-Precision Arithmetic

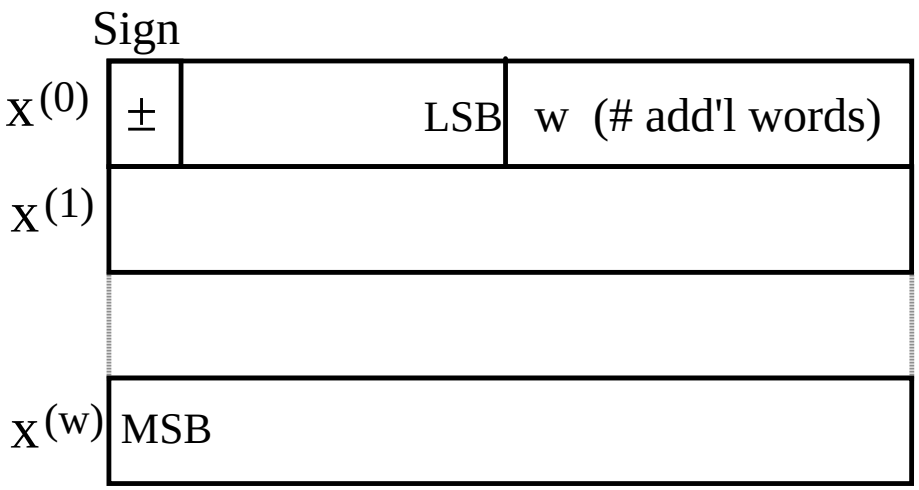


Fig. 20.6 Example variable-precision integer format.

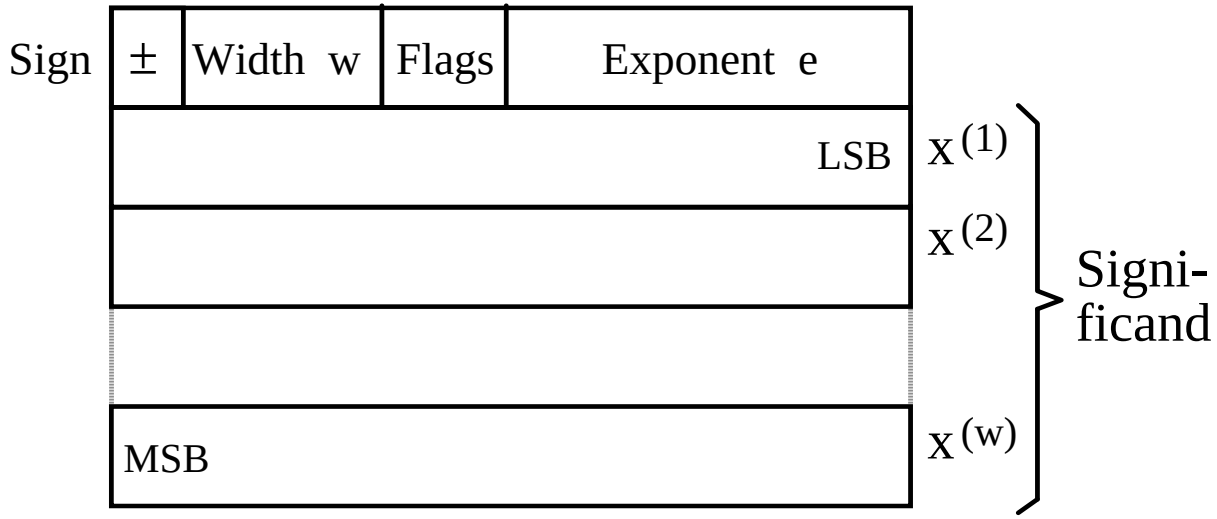


Fig. 20.7 Example variable-precision floating-point format.

Variable-Precision Floating-Point Addition

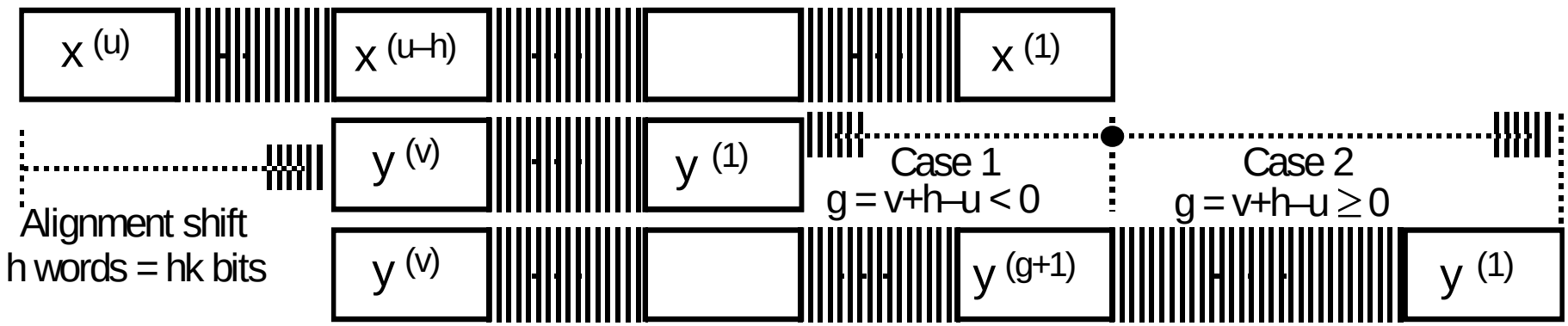


Fig. 20.8 Variable-precision floating-point addition.

20.5 Error-Bounding via Interval Arithmetic

Interval definition

$[a, b]$, $a \le b$, is an interval enclosing x, $a \le x \le b$

(intervals model uncertainty in real-valued parameters)

$[a, a]$ represents the real number $x = a$

$[a, b]$, $a > b$, is the empty interval

Combining and comparing intervals

$$[x_{lo}, x_{hi}] \cap [y_{lo}, y_{hi}] = [max(x_{lo}, y_{lo}), min(x_{hi}, y_{hi})]$$

$$[x_{lo}, x_{hi}] \cup [y_{lo}, y_{hi}] = [min(x_{lo}, y_{lo}), max(x_{hi}, y_{hi})]$$

$$[x_{lo}, x_{hi}] \supseteq [y_{lo}, y_{hi}] \text{ iff } x_{lo} \le y_{lo} \text{ and } x_{hi} \ge y_{hi}$$

$$[x_{lo}, x_{hi}] = [y_{lo}, y_{hi}] \text{ iff } x_{lo} = y_{lo} \text{ and } x_{hi} = y_{hi}$$

$$[x_{lo}, x_{hi}] < [y_{lo}, y_{hi}] \text{ iff } x_{hi} < y_{lo}$$

Arithmetic Operations on Intervals

Additive and multiplicative inverses

$$-[x_{lo}, x_{hi}] = [-x_{hi}, -x_{lo}]$$

$1 / [x_{lo}, x_{hi}] = [1/x_{hi}, 1/x_{lo}]$, provided that $0 \notin [x_{lo}, x_{hi}]$

When $0 \in [x_{lo}, x_{hi}]$, the multiplicative inverse is $[-\infty, +\infty]$

The four basic arithmetic operations

$$[x_{lo}, x_{hi}] + [y_{lo}, y_{hi}] = [x_{lo} + y_{lo}, x_{hi} + y_{hi}]$$

$$[x_{lo}, x_{hi}] - [y_{lo}, y_{hi}] = [x_{lo} - y_{hi}, x_{hi} - y_{lo}]$$

$$[x_{lo}, x_{hi}] \times [y_{lo}, y_{hi}] = [min(x_{lo}y_{lo}, x_{lo}y_{hi}, x_{hi}y_{lo}, x_{hi}y_{hi}), \; max(x_{lo}y_{lo}, x_{lo}y_{hi}, x_{hi}y_{lo}, x_{hi}y_{hi})]$$

$$[x_{lo}, x_{hi}] / [y_{lo}, y_{hi}] = [x_{lo}, x_{hi}] \times [1/y_{hi}, 1/y_{lo}]$$

Getting Narrower Result Intervals

Theorem 20.1: If $f(x^{(1)}, x^{(2)}, \ldots, x^{(n)})$ is a rational expression in the interval variables $x^{(1)}, x^{(2)}, \ldots, x^{(n)}$, that is, f is a finite combination of $x^{(1)}, x^{(2)}, \ldots, x^{(n)}$ and a finite number of constant intervals by means of interval arithmetic operations, then $x^{(i)} \supset y^{(i)}$, $i = 1, 2, \ldots, n$, implies:

$$f(x^{(1)}, x^{(2)}, \ldots, x^{(n)}) \supset f(y^{(1)}, y^{(2)}, \ldots, y^{(n)})$$

Thus, arbitrarily narrow result intervals can be obtained by simply performing arithmetic with sufficiently high precision

With reasonable assumptions about machine arithmetic, we have:

Theorem 20.2: Consider the execution of an algorithm on real numbers using machine interval arithmetic in FLP(r, p, $\nabla|\Delta$). If the same algorithm is executed using the precision q, with $q > p$, the bounds for both the absolute error and relative error are reduced by the factor r^{q-p} (the absolute or relative error itself may not be reduced by this factor; the guarantee applies only to the upper bound)

A Strategy for Accurate Interval Arithmetic

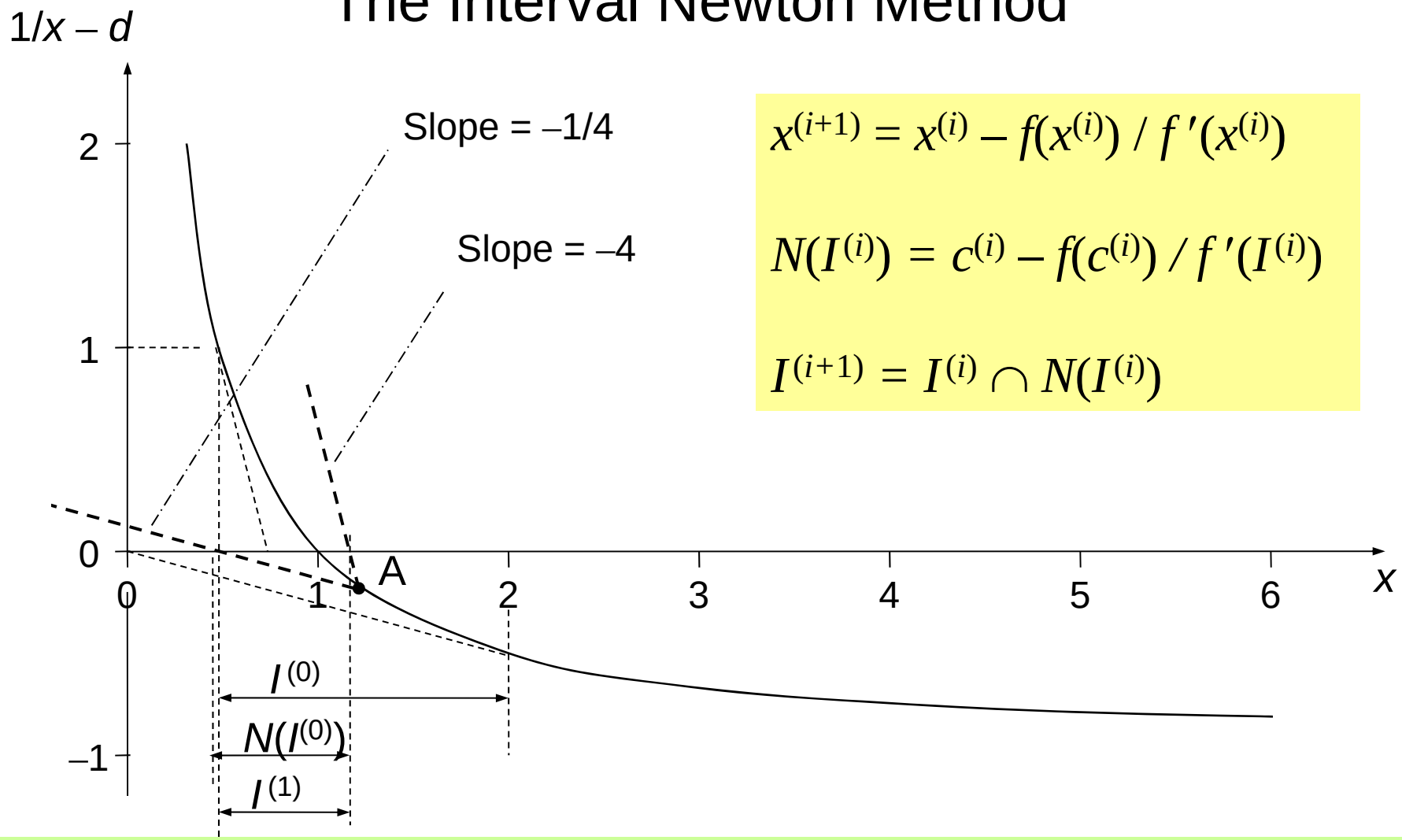
Theorem 20.2: Consider the execution of an algorithm on real numbers using machine interval arithmetic in FLP(r, p, $\nabla|\Delta$). If the same algorithm is executed using the precision q, with $q > p$, the bounds for both the absolute error and relative error are reduced by the factor r^{q-p} (the absolute or relative error itself may not be reduced by this factor; the guarantee applies only to the upper bound)

Let w_{max} be the maximum width of a result interval when interval arithmetic is used with p radix-r digits of precision. If $w_{max} \leq \varepsilon$, then we are done. Otherwise, interval calculations with the higher precision

$$q = p + \lceil \log_r w_{max} - \log_r \varepsilon \rceil$$

is guaranteed to yield the desired accuracy.

The Interval Newton Method



$$x^{(i+1)} = x^{(i)} - f(x^{(i)}) / f'(x^{(i)})$$

$$N(I^{(i)}) = c^{(i)} - f(c^{(i)}) / f'(I^{(i)})$$

$$I^{(i+1)} = I^{(i)} \cap N(I^{(i)})$$

Fig. 20.9 Illustration of the interval Newton method for computing 1/*d*.

Laws of Algebra in Interval Arithmetic

As in FLP arithmetic, laws of algebra may not hold for interval arithmetic

For example, one can readily construct an example where for intervals *x*, *y* and *z*, the following two expressions yield different interval results, thus demonstrating the violation of the distributive law:

$x(y + z)$ $\qquad\qquad$ $xy + xz$

Can you find other laws of algebra that may be violated?

20.6 Adaptive and Lazy Arithmetic

Need-based incremental precision adjustment to avoid high-precision calculations dictated by worst-case errors

Lazy evaluation is a powerful paradigm that has been and is being used in many different contexts. For example, in evaluating composite conditionals such as

if *cond1* and *cond2* then *action*

evaluation of *cond2* may be skipped if *cond1* yields “false”
More generally, lazy evaluation means

postponing all computations or actions
until they become irrelevant or unavoidable

Opposite of lazy evaluation (speculative or aggressive execution) has been applied extensively

Lazy Arithmetic with Redundant Representations

Redundant number representations offer some advantages for lazy arithmetic

Because redundant representations support MSD-first arithmetic, it is possible to produce a small number of result digits by using correspondingly less computational effort, until more precision is actually needed