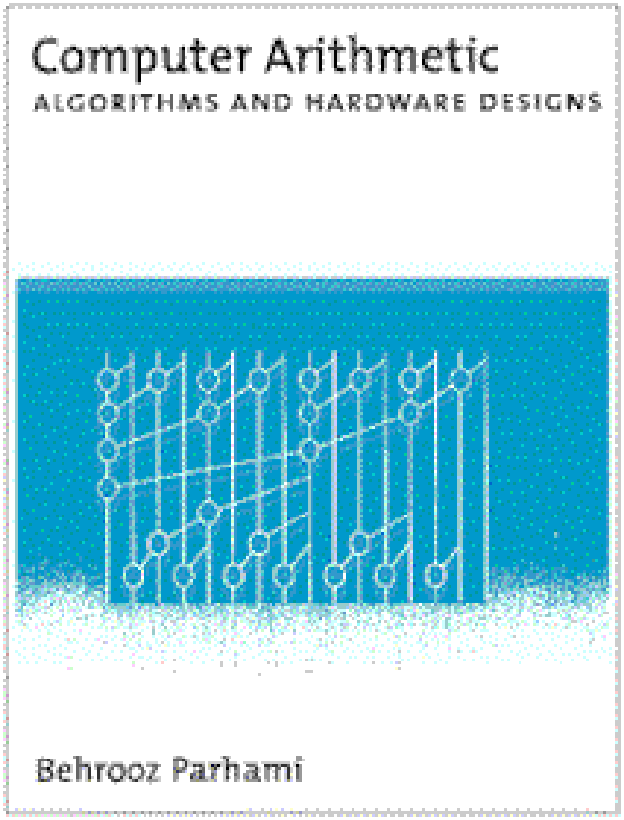
INSTRUCTOR'S MANUAL FOR

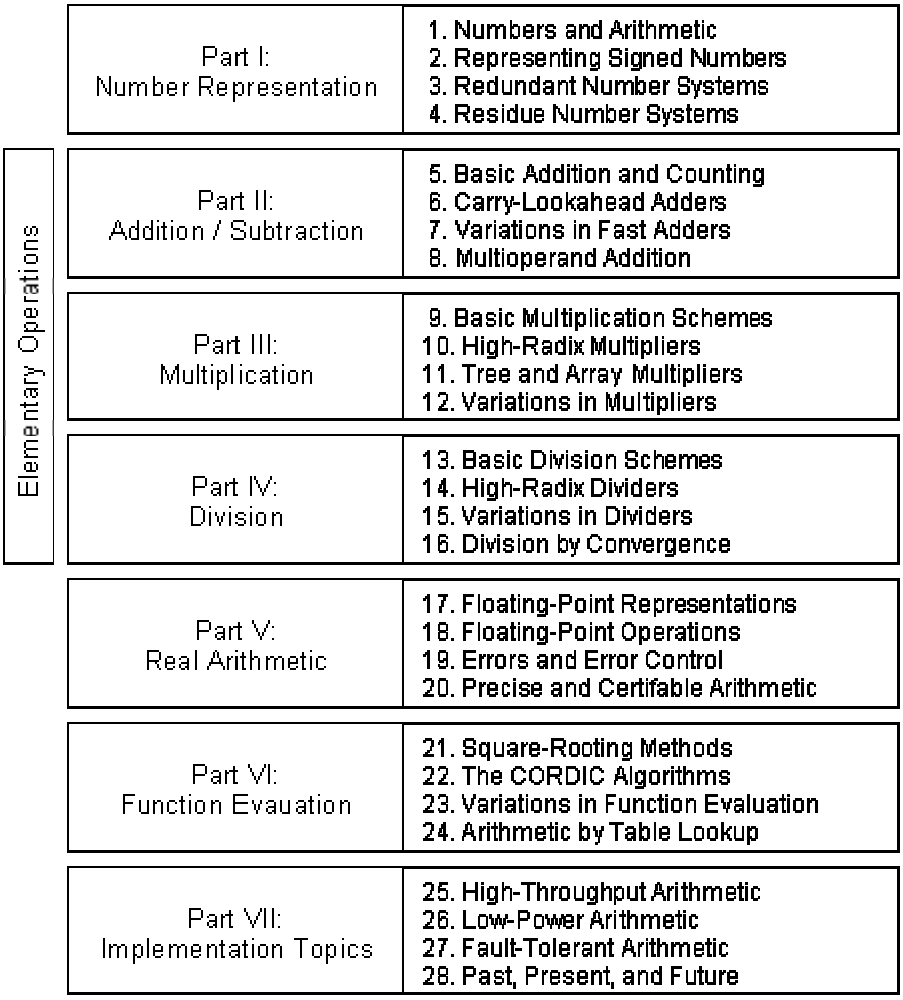


Volume 2: Presentation Material

Behrooz Parhami

Department of Electrical and Computer Engineering
University of California
Santa Barbara, CA 93106-9560, USA

E-mail: parhami@ece.ucsb.edu



This instructor's manual is for

Computer Arithmetic: Algorithms and Hardware Designs, by Behrooz Parhami

ISBN 0-19-512583-5, QA76.9.C62P37

For information and errata, see http://www.ece.ucsb.edu/Faculty/Parhami/text_comp_arit.htm

Preface to the Instructor's Manual

This instructor's manual consists of two volumes. Volume 1 presents solutions to selected problems and includes additional problems (many with solutions) that did not make the cut for inclusion in the text *Computer Arithmetic: Algorithms and Hardware Designs* (Oxford University Press, 2000) or that were designed after the book went to print. Volume 2 contains enlarged versions of the figures and tables in the text as well as additional material, presented in a format that is suitable for use as transparency masters.

The fall 2001 edition Volume 1, which consists of the following parts, is available to qualified instructors through the publisher:

Volume 1	Part I	Selected solutions and additional problems
	Part II	Question bank, assignments, and projects

The fall 2001 edition of Volume 2, which consists of the following parts, is available as a large file in postscript format through the book's Web page:

Volume 2	Parts I-VII	Lecture slides and other presentation material

The book's Web page, given below, also contains an errata and a host of other material (please note the upper-case "F" and "P" and the underscore symbol after "text" and "comp":

http://www.ece.ucsb.edu/Faculty/Parhami/text_comp_arit.htm

The author would appreciate the reporting of any error in the textbook or in this manual, suggestions for additional problems, alternate solutions to solved problems, solutions to other problems, and sharing of teaching experiences. Please e-mail your comments to

parhami@ece.ucsb.edu

or send them by regular mail to the author's postal address:

Department of Electrical and Computer Engineering
University of California
Santa Barbara, CA 93106-9560, USA

Contributions will be acknowledged to the extent possible.

Behrooz Parhami
Santa Barbara, Fall 2001

Table of Contents

Part I Number Representation

Part Goals

- Review fixed-point number systems
  (floating-point covered in Part V)
- Learn how to handle signed numbers
- Discuss some unconventional methods

Part Synopsis

- Number representation is is a key element
  affecting hardware cost and speed
- Conventional, redundant, residue systems
- Intermediate vs endpoint representations
- Limits of fast arithmetic

Part Contents

1 Numbers and Arithmetic

Go to TOC

Chapter Goals

- Define scope and provide motivation
- Set the framework for the rest of the book
- Review positional fixed-point numbers

Chapter Highlights

- What goes on inside your calculator?
- Ways of encoding numbers in *k* bits
- Radix and digit set: conventional, exotic
- Conversion from one system to another

Chapter Contents

1.1 What Is Computer Arithmetic?

Pentium Division Bug (1994-95): Pentium's radix-4 SRT algorithm occasionally produced an incorrect quotient
First noted in 1994 by T. Nicely who computed sums of reciprocals of twin primes:

$$1/5 + 1/7 + 1/11 + 1/13 + \ldots + 1/p + 1/(p + 2) + \ldots$$

Worst-case example of division error in Pentium:

$$c = \frac{4\,195\,835}{3\,145\,727} = \begin{cases} 1.333\,820\,44... & \text{Correct quotient} \\ 1.333\,739\,06... & \text{circa 1994 Pentium double FLP value; accurate to only 14 bits (worse than single!)} \end{cases}$$

Humor, circa 1995

Top Ten New Intel Slogans for the Pentium:

9.999 997 325 It's a FLAW, dammit, not a bug
8.999 916 336 It's close enough, we say so
7.999 941 461 Nearly 300 correct opcodes
6.999 983 153 You don't need to know what's inside
5.999 983 513 Redefining the PC — and math as well
4.999 999 902 We fixed it, really
3.999 824 591 Division considered harmful
2.999 152 361 Why do you think it's called "floating" point?
1.999 910 351 We're looking for a few good flaws
0.999 999 999 The errata inside

Hardware (our focus in this book)

Design of efficient digital circuits for primitive and other arithmetic operations such as +, –, ×, ÷, √, log, sin, and cos

Issues:
- Algorithms
- Error analysis
- Speed/cost tradeoffs
- Hardware implementation
- Testing, verification

Software

Numerical methods for solving systems of linear equations, partial differential equations, etc.

Issues:
- Algorithms
- Error analysis
- Computational complexity
- Programming
- Testing, verification

General-Purpose

Flexible data paths
Fast primitive operations like +, –, ×, ÷, √
Benchmarking

Special-Purpose

Tailored to application areas such as:
Digital filtering
Image processing
Radar tracking

Fig. 1.1 The scope of computer arithmetic.

1.2 A Motivating Example

Using a calculator with $\sqrt{}$, x^2, and x^y functions, compute:

$u = \underbrace{\sqrt{\sqrt{\ldots\sqrt{2}}}}_{10 \text{ times}}$ = 1.000 677 131 "1024th root of 2"

$v = 2^{1/1024}$ = 1.000 677 131

Save u and v; If you can't, recompute when needed.

$x = \underbrace{(((u^2)^2)\cdots)^2}_{10 \text{ times}}$ = 1.999 999 963

$x' = u^{1024}$ = 1.999 999 973

$y = \underbrace{(((v^2)^2)\cdots)^2}_{10 \text{ times}}$ = 1.999 999 983

$y' = v^{1024}$ = 1.999 999 994

Perhaps v and u are not really the same value.

$w = v - u = 1 \times 10^{-11}$ Nonzero due to hidden digits

$(u - 1) \times 1000$ = 0.677 130 680 [Hidden ··· (0) 68]

$(v - 1) \times 1000$ = 0.677 130 690 [Hidden ··· (0) 69]

A simple analysis:

$$v^{1024} = (u + 10^{-11})^{1024} \cong u^{1024} + 1024 \times 10^{-11} u^{1023}$$
$$\cong u^{1024} + 2 \times 10^{-8}$$

Finite Precision Can Lead to Disaster

Example: Failure of Patriot Missile (1991 Feb. 25)

Source http://www.math.psu.edu/dna/455.f96/disasters.html

American Patriot Missile battery in Dharan, Saudi Arabia,
failed to intercept incoming Iraqi Scud missile
The Scud struck an American Army barracks, killing 28

Cause, per GAO/IMTEC-92-26 report: "software problem"
(inaccurate calculation of the time since boot)

Specifics of the problem: time in tenths of second
as measured by the system's internal clock
was multiplied by 1/10 to get the time in seconds
Internal registers were 24 bits wide
1/10 = 0.0001 1001 1001 1001 1001 100 (chopped to 24 b)
Error $\cong 0.1100\ 1100 \times 2^{-23} \cong 9.5 \times 10^{-8}$
Error in 100-hr operation period
$\cong 9.5 \times 10^{-8} \times 100 \times 60 \times 60 \times 10 = 0.34$ s
Distance traveled by Scud = (0.34 s) $\times$ (1676 m/s) $\cong$ 570 m
This put the Scud outside the Patriot's "range gate"

Ironically, the fact that the bad time calculation
had been improved in some (but not all) code parts
contributed to the problem,
since it meant that inaccuracies did not cancel out

Finite Range Can Lead to Disaster

Example: Explosion of Ariane Rocket (1996 June 4)

Source http://www.math.psu.edu/dna/455.f96/disasters.html

Unmanned Ariane 5 rocket
launched by the European Space Agency
veered off its flight path, broke up, and exploded
only 30 seconds after lift-off (altitude of 3700 m)

The $500 million rocket (with cargo) was on its 1st voyage
after a decade of development costing $7 billion

Cause: "software error in the inertial reference system"

Specifics of the problem: a 64 bit floating point number
relating to the horizontal velocity of the rocket
was being converted to a 16 bit signed integer

An SRI* software exception arose during conversion
because the 64-bit floating point number
had a value greater than what could be represented
by a 16-bit signed integer (max 32 767)

*SRI stands for Système de Référence Inertielle
or Inertial Reference System

1.3 Numbers and Their Encodings

Numbers versus their representations (*numerals*)

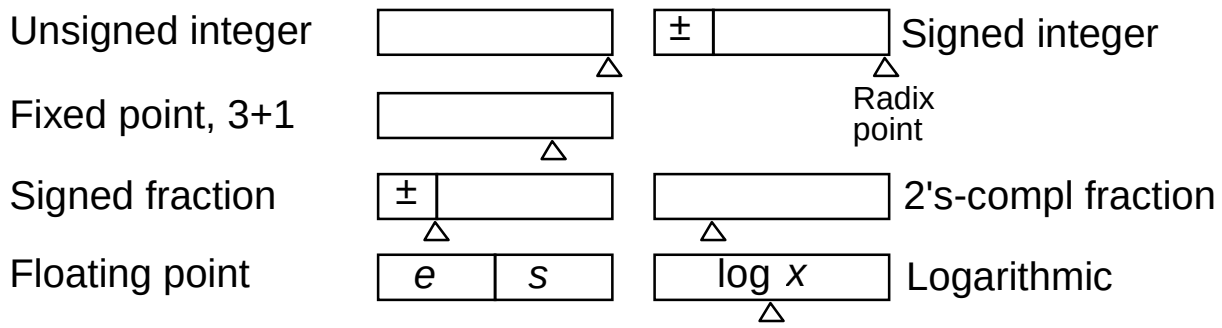
The number "twenty-seven" can be represented in different ways using numerals or *numeration systems*:

||||| ||||| ||||| ||||| ||||| ||	sticks or *unary* code	
27	radix-10 or *decimal* code	$(27)_{ten}$
11011	radix-2 or *binary* code	$(11011)_{two}$
XXVII	Roman numerals	

Encoding of digit sets as binary strings: BCD example

Digit	BCD representation
0	0 0 0 0
1	0 0 0 1
2	0 0 1 0
3	0 0 1 1
4	0 1 0 0
5	0 1 0 1
6	0 1 1 0
7	0 1 1 1
8	1 0 0 0
9	1 0 0 1

Encoding of numbers in 4 bits:



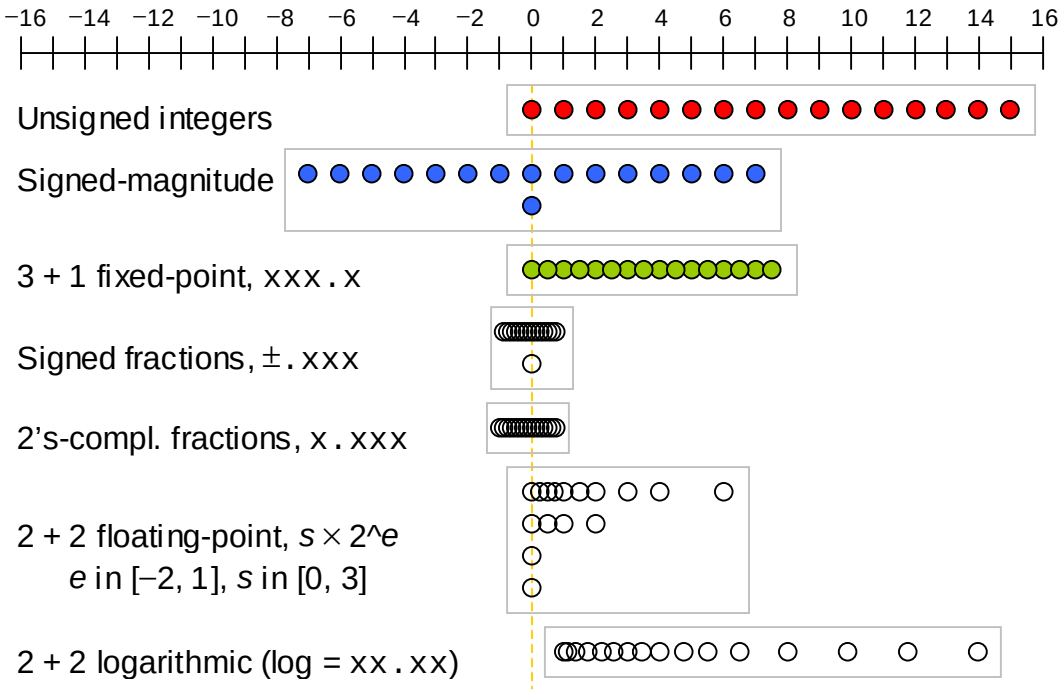


Fig. 1.2 Some of the possible ways of assigning 16 distinct codes to represent numbers.

1.4 Fixed-Radix Positional Number Systems

$$(x_{k-1}x_{k-2}\cdots x_1x_0 . x_{-1}x_{-2}\cdots x_{-l})_r = \sum_{i=-l}^{k-1} x_i r^i$$

One can generalize to:
arbitrary radix (not necessarily integer, positive, constant)
arbitrary digit set, usually $\{-\alpha, -\alpha+1, \cdots, \beta-1, \beta\} = [-\alpha, \beta]$

Example 1.1. Balanced ternary number system:
radix $r = 3$, digit set = $[-1, 1]$

Example 1.2. Negative-radix number systems:
radix $-r$, $r \geq 2$, digit set = $[0, r-1]$

The special case with radix –2 and digit set [0, 1] is known as the negabinary number system

Example 1.3. Digit set [–4, 5] for $r = 10$:
$(3\ \ {}^{-}1\ \ 5)_{ten}$ represents $295 = 300 - 10 + 5$

Example 1.4. Digit set [–7, 7] for $r = 10$:
$(3\ \ {}^{-}1\ \ 5)_{ten} = (3\ \ 0\ \ {}^{-}5)_{ten} = (1\ \ {}^{-}7\ \ 0\ \ {}^{-}5)_{ten}$

Example 1.7. Quater-imaginary number system:
radix $r = 2j$, digit set [0, 3].

1.5 Number Radix Conversion

$u = w \,.\, v$

$= (x_{k-1}x_{k-2} \cdots x_1x_0 \,.\, x_{-1}x_{-2} \cdots x_{-l})_r$ Old

$= (X_{K-1}X_{K-2} \cdots X_1X_0 \,.\, X_{-1}X_{-2} \cdots Xx_{-L})_R$ New

Radix conversion: arithmetic in the old radix r

Converting whole part *w*: $(105)_{ten} = (?)_{five}$

Repeatedly divide by five

Quotient	Remainder
105	0
21	1
4	4
0	

Therefore, $(105)_{ten} = (410)_{five}$

Converting fractional part *v*: $(105.486)_{ten} = (410.?)_{five}$

Repeatedly multiply by five

Whole Part	Fraction
	.486
2	.430
2	.150
0	.750
3	.750
3	.750

Therefore, $(105.486)_{ten} \cong (410.22033)_{five}$

Radix conversion: arithmetic in the new radix R

Converting the whole part *w*

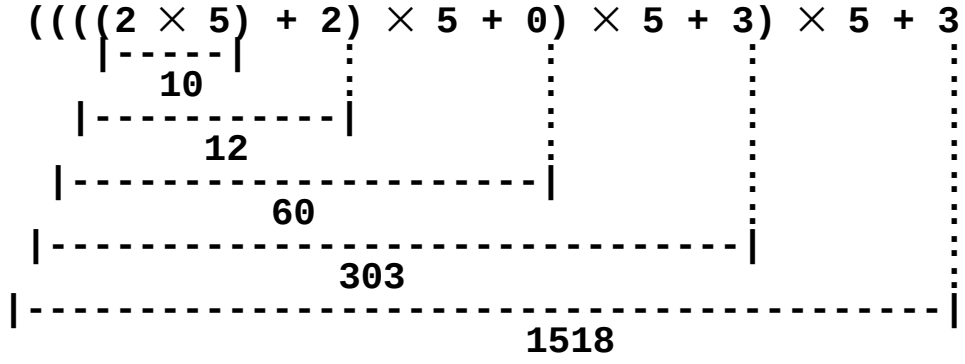


Fig. 1.A Horner's rule used to convert $(22033)_{five}$ to decimal.

Converting fractional part *v*: $(410.22033)_{five} = (105.?)_{ten}$

$(0.22033)_{five} \times 5^5 = (22033)_{five} = (1518)_{ten}$

$1518 / 5^5 = 1518 / 3125 = 0.48576$

Therefore, $(410.22033)_{five} = (105.48576)_{ten}$

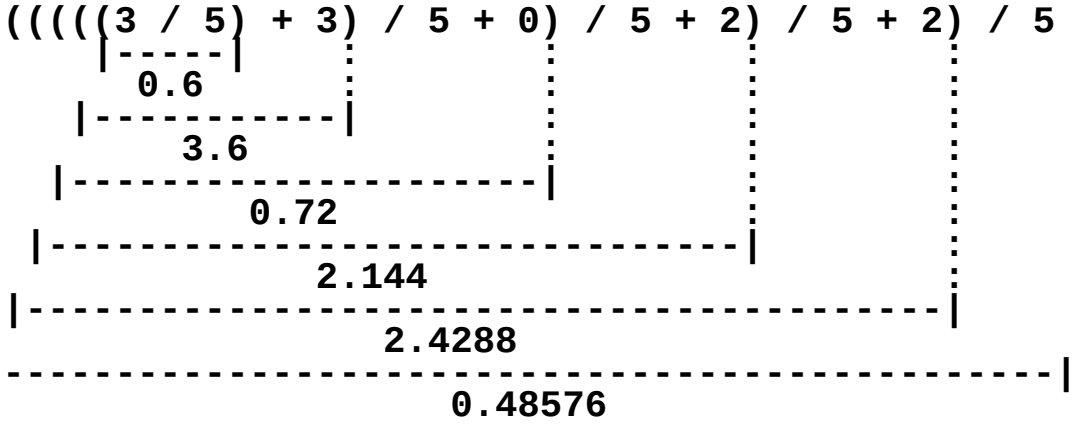


Fig. 1.3 Horner's rule used to convert $(0.22033)_{five}$ to decimal.

1.6 Classes of Number Representations

Integers (fixed-point), unsigned: Chapter 1

Integers (fixed-point), signed

- signed-magnitude, biased, complement: Chapter 2
- signed-digit: Chapter 3
  - (but the key point of Chapter 3 is use of redundancy for faster arithmetic, not how to represent signed values)
- residue number system: Chapter 4
  - (again, the key to Chapter 4 is use of parallelism for faster arithmetic, not how to represent signed values)

Real numbers, floating-point: Chapter 17

- covered in Part V, just before real-number arithmetic

Real numbers, exact: Chapter 20

- continued-fraction, slash, ... (for error-free arithmetic)

Part V Real Arithmetic

Part Goals

- Review floating-point representations
- Learn about floating-point arithmetic
- Discuss error sources and error bounds

Part Synopsis

- Combining wide range and high precision
- Floating-point formats and operations
- The ANSI/IEEE standard
- Errors: causes and consequences
- When can we trust computation results?

Part Contents

17 Floating-Point Representations

Go to TOC

Chapter Goals

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

Chapter Highlights

Floating-point formats and tradeoffs
Why a floating-point standard?
Finiteness of precision and range
The two extreme special cases:
fixed-point and logarithmic numbers

Chapter Contents

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

Floating-point numbers

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

Two signs are involved in a floating-point number.

1. The significand or number sign,
   usually represented by a separate sign bit

2. The exponent sign,
   usually embedded in the biased exponent
   (when the bias is a power of 2,
   the exponent sign is the complement of its MSB)

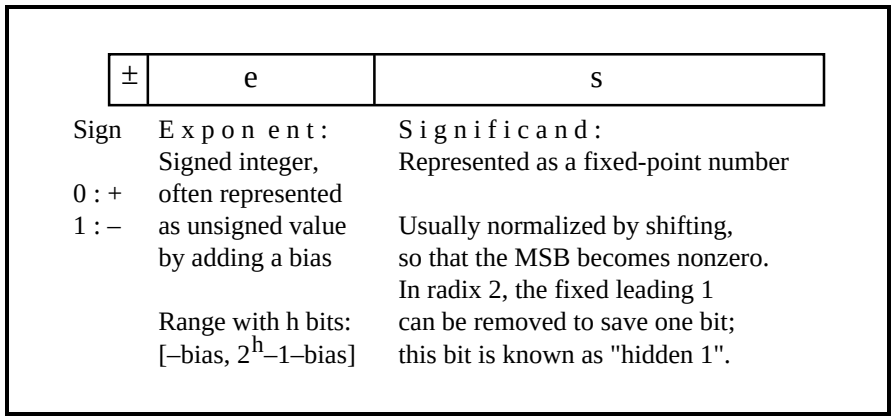


Fig. 17.1 Typical floating-point number format.

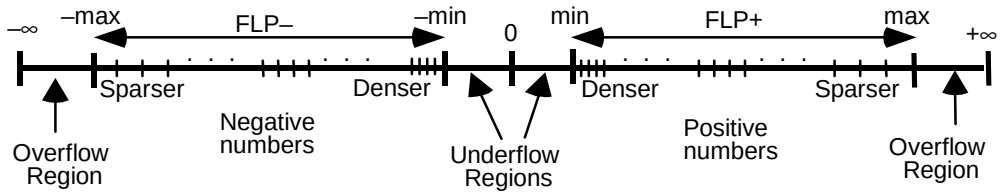


Fig. 17.2 Subranges and special values in floating-point number representations.

17.2 The ANSI/IEEE Floating-Point Standard

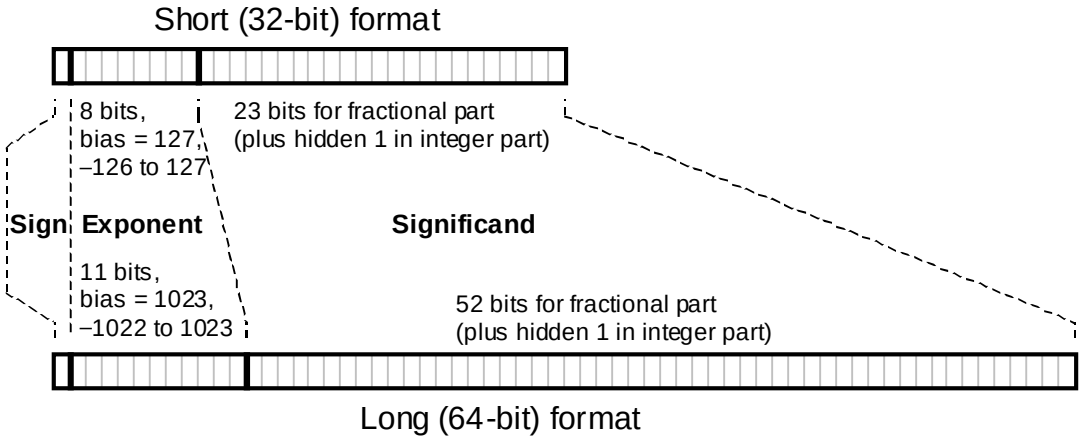


Fig. 17.3 The ANSI/IEEE standard floating-point number representation formats.

Table 17.1 Some features of the ANSI/IEEE 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	$\cong 2^{128} \cong 3.4 \times 10^{38}$	$\cong 2^{1024} \cong 1.8 \times 10^{308}$

Operations on special operands:

Ordinary number ÷ (+∞) = ±0
(+∞) × Ordinary number = ±∞
NaN + Ordinary number = NaN

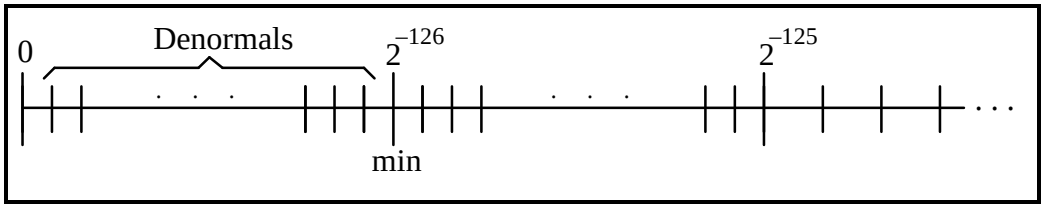


Fig. 17.4 Denormals in the IEEE single-precision format.

The IEEE floating-point standard also defines

The four basic arithmetic op's (+, –, ×, ÷) and $\sqrt{x}$ must match the results that would be obtained if intermediate computations were infinitely precise

Extended formats for greater internal precision

Single-extended: ≥ 11 bits for exponent
≥ 32 bits for significand
bias unspecified, but
exp range ⊇ [–1022, 1023]

Double-extended: ≥ 15 bits for exponent
≥ 64 bits for significand
exp range ⊇ [–16 382, 16 383]

17.3 Basic Floating-Point Algorithms

Addition/Subtraction

Assume $e1 \geq e2$; need *alignment shift* (*preshift*) if $e1 > e2$:

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

Like signs: 1-digit normalizing right shift may be needed
Different signs: shifting by many positions may be needed
Overflow/underflow during addition or normalization

Multiplication

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

Postshifting for normalization, exponent adjustment
Overflow/underflow during multiplication or normalization

Division

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

Square-rooting

First make the exponent even, if necessary

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

In all algorithms, rounding complications are ignored here

17.4 Conversions and Exceptions

Conversions from fixed- to floating-point

Conversions between floating-point formats

Conversion from high to lower precision: Rounding

ANSI/IEEE standard includes four rounding modes:

Round to nearest even [default rounding mode]
Round toward zero (inward)
Round toward $+\infty$ (upward)
Round toward $-\infty$ (downward)

Exceptions

divide by zero
overflow
underflow
inexact result: rounded value not same as original
invalid operation: examples include

addition	$(+\infty) + (-\infty)$
multiplication	$0 \times \infty$
division	$0 / 0$ or ∞ / ∞
square-root	operand < 0

17.5 Rounding Schemes

Round

$$x_{k-1}x_{k-2}\cdots x_1x_0 . x_{-1}x_{-2}\cdots x_{-l} \quad \Rightarrow \quad y_{k-1}y_{k-2}\cdots y_1y_0 .$$

Special case: truncation or chopping

Chop

$$x_{k-1}x_{k-2}\cdots x_1x_0 . x_{-1}x_{-2}\cdots x_{-l} \quad \Rightarrow \quad x_{k-1}x_{k-2}\cdots x_1x_0 .$$

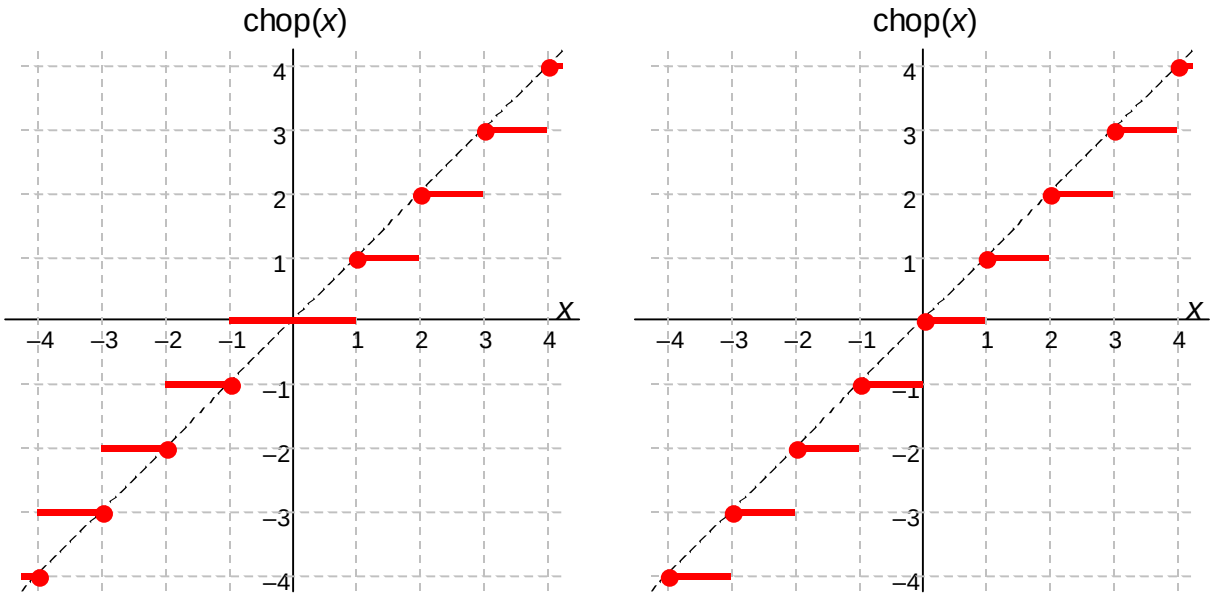


Fig. 17.5 Truncation or chopping of a signed-magnitude number (same as round toward 0).

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

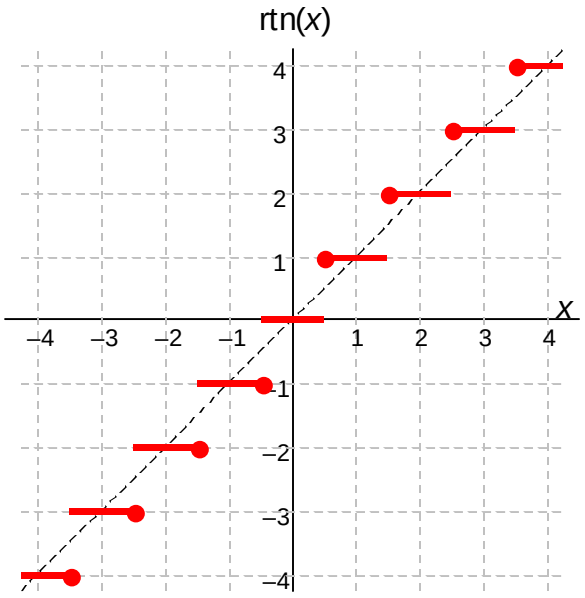


Fig. 17.7 Rounding of a signed-magnitude value to the nearest number.

Ordinary rounding has a slight upward bias

Assume that $(x_{k-1}x_{k-2} \cdot \cdot \cdot x_1x_0 \,.\, x_{-1}x_{-2})_{\text{two}}$ is to be rounded to an integer $(y_{k-1}y_{k-2} \cdot \cdot \cdot y_1y_0 \,.)_{\text{two}}$

The four possible cases, and their representation errors:

$x_{-1}x_{-2} = 00$	round down	error = 0
$x_{-1}x_{-2} = 01$	round down	error = –0.25
$x_{-1}x_{-2} = 10$	round up	error = 0.5
$x_{-1}x_{-2} = 11$	round up	error = 0.25

Assume 4 cases are equiprobable $\Rightarrow$ mean error = 0.125

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

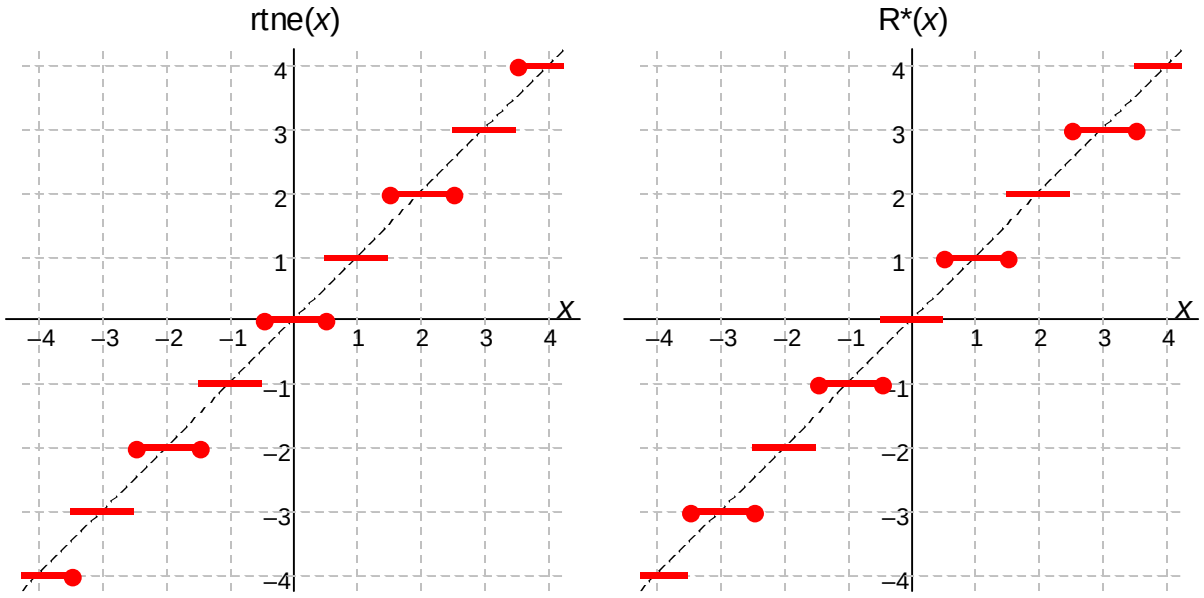


Fig. 17.8 Rounding to the nearest even number.

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

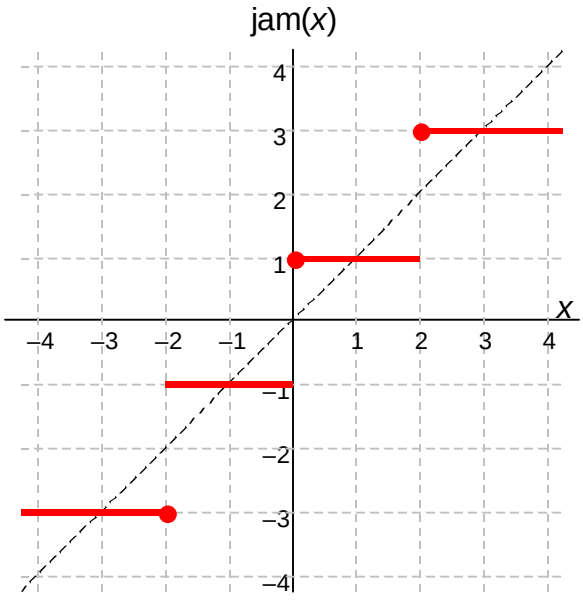


Fig. 17.10 Jamming or von Neumann rounding.

ROM rounding

32×4-ROM-Round

$$x_{k-1} \cdots x_4 \overbrace{x_3 x_2 x_1 x_0}^{\text{ROM Address}} . x_{-1} \cdots x_{-l} \Longrightarrow x_{k-1} \cdots x_4 \overbrace{y_3 y_2 y_1 y_0}^{\text{ROM Data}} .$$

The rounding result is the same as that of the round to nearest scheme in 15 of the 16 possible cases, but a larger error is introduced when $x_3 = x_2 = x_1 = x_0 = 1$

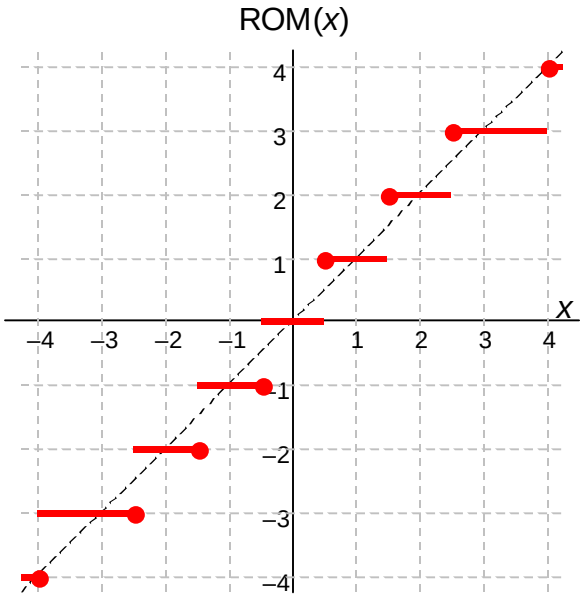


Fig. 17.11 ROM rounding with an 8 × 2 table.

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 +∞) and
downward-directed rounding (round toward –∞)
(required features of the IEEE floating-point standard)

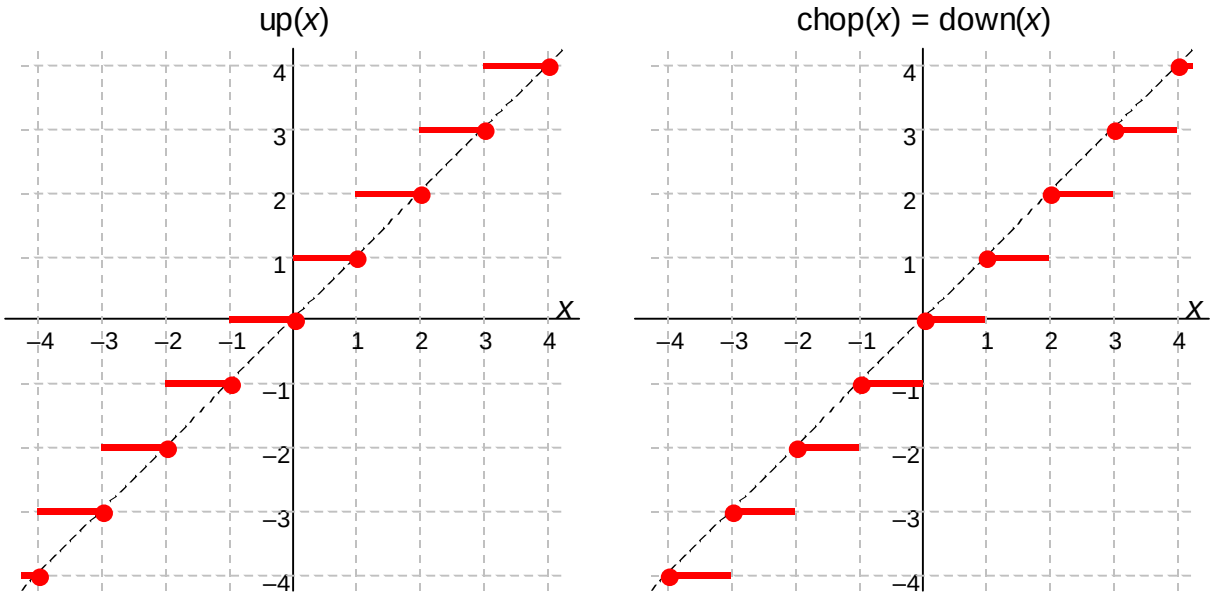


Fig. 17.12 Upward-directed rounding or rounding toward +∞ (see Fig. 17.6 for downward-directed rounding, or rounding toward –∞).

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 floating-point representation

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

b usually called the logarithm base, not exponent base

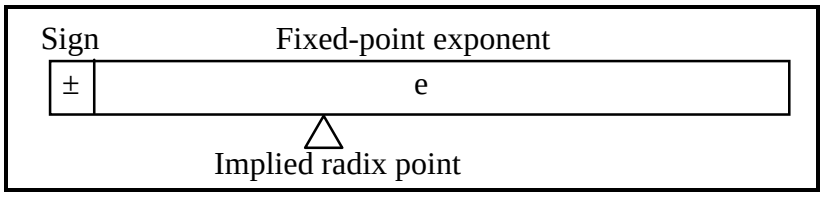


Fig. 17.13 Logarithmic number representation with sign and fixed-point exponent.

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

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

Simple multiply and divide; harder add and subtract

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 above represents $-2^{-9.828125} \cong -(0.0011)_{ten}$
number range $\cong [-2^{16}, 2^{16}]$, with $min = 2^{-16}$

19 Errors and Error Control

Go to TOC

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 $3 \times (1/3)$ is not necessarily 1?
Error analysis and bounding

Chapter Contents

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 format

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

Example 19.1: Compute 1/99 – 1/100
(decimal floating-point format, 4-digit significand in [1, 10), single-digit signed exponent)
precise result = $1/9900 \cong 1.010 \times 10^{-4}$ (error $\cong 10^{-8}$ or 0.01%)
$x = 1/99 \cong 1.010 \times 10^{-2}$ 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 floating-point system 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 x_{fp} be its representation in FLP(r, p, A)

$x_{fp} = r^e s_{fp} = (1 + \eta)x$

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 FLP(r, p, A)

Consider multiplication, division, addition, and subtraction for *positive operands* x_{fp} and y_{fp} in FLP(r, p, A)

Due to representation errors, $x_{fp} = (1 + \sigma)x$, $y_{fp} = (1 + \tau)y$

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

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

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

$$= (1 + \eta)\left(1 + \frac{\sigma x + \tau y}{x + y}\right)(x + y)$$

Since $|\sigma x + \tau y| \leq max(|\sigma|, |\tau|)(x + y)$, the magnitude of the worst-case relative error in the computed sum is roughly bounded by $|\eta| + max(|\sigma|, |\tau|)$

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

$$= (1 + \eta)\left(1 + \frac{\sigma x - \tau y}{x - y}\right)(x - y)$$

The term $(\sigma x - \tau y)/(x - y)$ can be very large if x and y are both large but $x - y$ is relatively small

This is known as *cancellation* or *loss of significance*

Fixing the problem

The part of the problem that is due to η being large can be fixed by using guard digits

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

$$x +_{fp} y = (1 + \eta)(x + y) \quad \text{with} \quad -r^{-p+1} < \eta < r^{-p-g+2}$$

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

$$x +_{fp} y = (1 + \eta)(x + y) \quad \text{with } |\eta| < r^{-p+1}$$

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

Example 19.2: Decimal floating-point system ($r = 10$) with $p = 6$ and 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 $x_{fp} + y_{fp} = 10^{-4}$, but:

$$x_{fp} +_{fp} y_{fp} = .100\ 000 \times 10^3 -_{fp} .099\ 999 \times 10^3$$
$$= .100\ 000 \times 10^{-2}$$

Relative error = $(10^{-3} - 0.544 \times 10^{-4})/(0.544 \times 10^{-4}) \cong 17.38$
(i.e., the result is 1738% larger than the correct sum!)

With 1 guard digit, we get:

$$x_{fp} +_{fp} y_{fp} = 0.100\ 000\ 0 \times 10^3 -_{fp} 0.099\ 999\ 9 \times 10^3$$
$$= 0.100\ 000 \times 10^{-3}$$

Relative error = 80.5% relative to the exact sum $x + y$
but the error is 0% with respect to $x_{fp} + y_{fp}$

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: $a + (b + c) = (a + b) + c$

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

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

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

$= 0.123\ 41 \times 10^5 -_{fp} 0.123\ 39 \times 10^5$

$= \boxed{0.200\ 00 \times 10^1}$

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

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

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

$= \boxed{0.243\ 21 \times 10^1}$

The two results differ by about 20%!

A possible remedy: unnormalized arithmetic

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

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

$= 0.123\ 41 \times 10^5 -_{fp} 0.123\ 39 \times 10^5 = \boxed{0.000\ 02 \times 10^5}$

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

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

$= 0.000\ 01 \times 10^5 +_{fp} 0.143\ 21 \times 10^1 = \boxed{0.000\ 02 \times 10^5}$

Not only are the two results the same but they carry with them a kind of warning about the extent of potential error

Let's see if using 2 guard digits helps:

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

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

$= 0.123\ 41 \times 10^5 -_{fp} 0.123\ 385\ 7 \times 10^5 = \boxed{0.243\ 00 \times 10^1}$

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

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

$= 0.100\ 00 \times 10^1 +_{fp} 0.143\ 21 \times 10^1 = \boxed{0.243\ 21 \times 10^1}$

The difference is now about 0.1%; still too high

Using more guard digits will improve the situation but does not change the fact that laws of algebra cannot be assumed to hold in floating-point arithmetic

Examples of other laws of algebra that do not hold:

Associative law of multiplication

$$a \times (b \times c) = (a \times b) \times c$$

Cancellation law (for $a > 0$)

$$a \times b = a \times c \text{ implies } b = c$$

Distributive law

$$a \times (b + c) = (a \times b) + (a \times c)$$

Multiplication canceling division

$$a \times (b / a) = b$$

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

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

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

$$A = \sqrt{s(s-a)(s-b)(s-c)}$$

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

$$A = \frac{1}{4}\sqrt{(a+(b+c))(c-(a-b))(c+(a-b))(a+(b-c))}$$

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

Example: inner-product calculation $z = \sum_{i=0}^{1023} x^{(i)} y^{(i)}$

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 = $1024 \times ulp^2/2$

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

Moral of the preceding examples:

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

Kahan's summation algorithm or formula

To compute $s = \sum_{i=0}^{n-1} x^{(i)}$, proceed as follows

$s \leftarrow x^{(0)}$

$c \leftarrow 0$ {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

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, however, the distribution of errors and their expected values may be more important

Limiting ourselves to positive significands, we define:

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

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

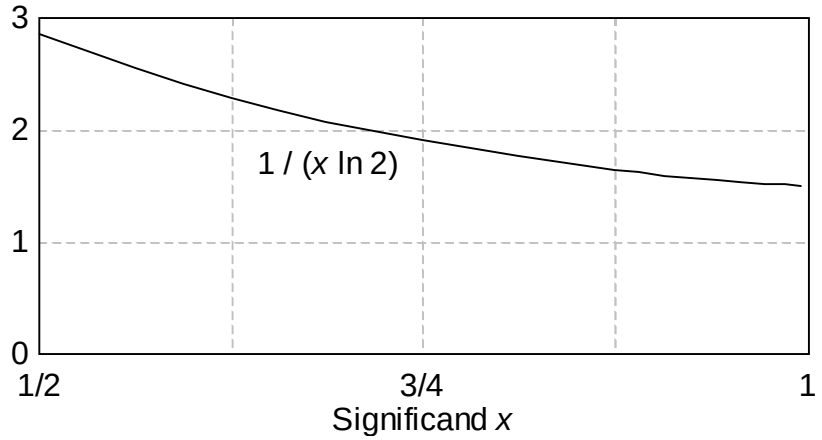


Fig. 19.1 Probability density function for the distribution of normalized significands in FLP($r = 2$, p, A).

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}, and 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

a. Automatic error analysis

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

b. Significance arithmetic

Roughly speaking, same as unnormalized arithmetic,
although there are some fine distinctions
The result of the unnormalized decimal addition

$.1234 \times 10^5 +_{fp} .0000 \times 10^{10} = .0000 \times 10^{10}$

warns us that precision has been lost

c. 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

d. Interval arithmetic

An interval $[x_{lo}, x_{hi}]$ represents x, $x_{lo} \leq x \leq x_{hi}$
With x_{lo}, x_{hi}, y_{lo}, $y_{hi} > 0$, to find $z = x / y$, we compute

$[z_{lo}, z_{hi}] = [x_{lo} /_{\nabla fp} y_{hi}, x_{hi} /_{\Delta fp} y_{lo}]$

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} deviate from the correct result 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}, b_{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

We are, thus, assured 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

