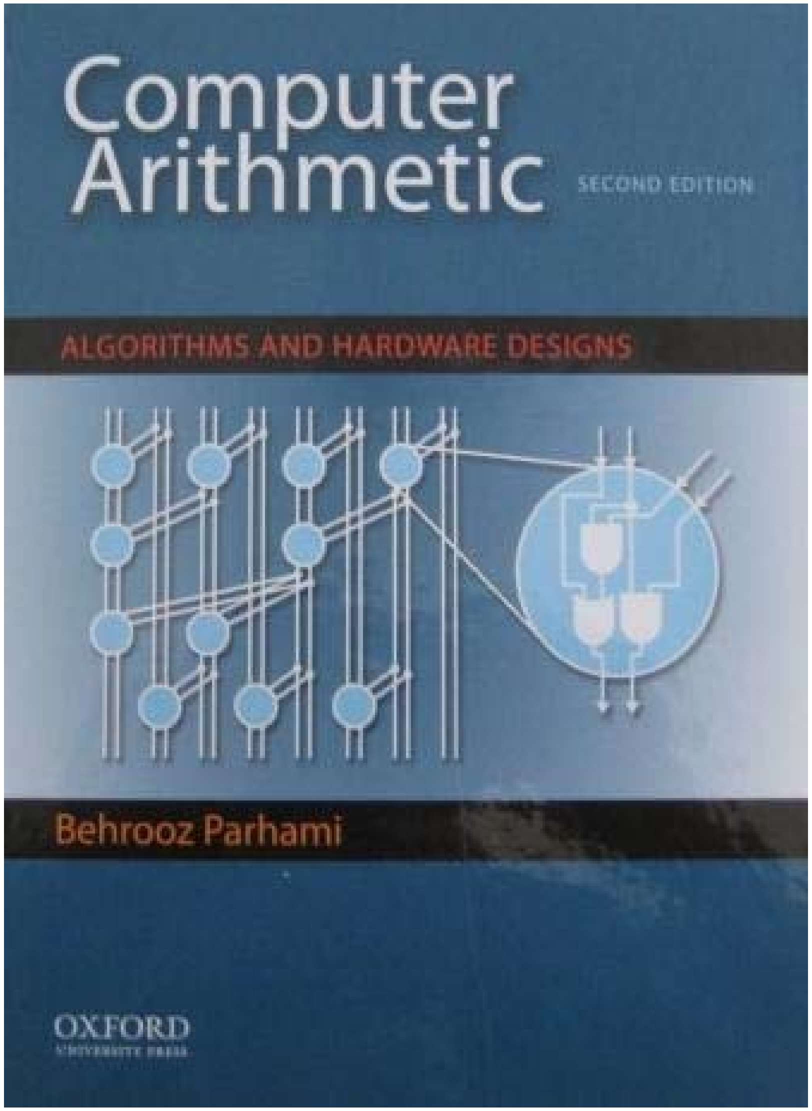


Part VI
Function Evaluation

	Parts	Chapters
	I. Number Representation	1. Numbers and Arithmetic 2. Representing Signed Numbers 3. Redundant Number Systems 4. Residue Number Systems
Elementary Operations	II. Addition / Subtraction	5. Basic Addition and Counting 6. Carry-Lookahead Adders 7. Variations in Fast Adders 8. Multioperand Addition
Elementary Operations	III. Multiplication	9. Basic Multiplication Schemes 10. High-Radix Multipliers 11. Tree and Array Multipliers 12. Variations in Multipliers
Elementary Operations	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.

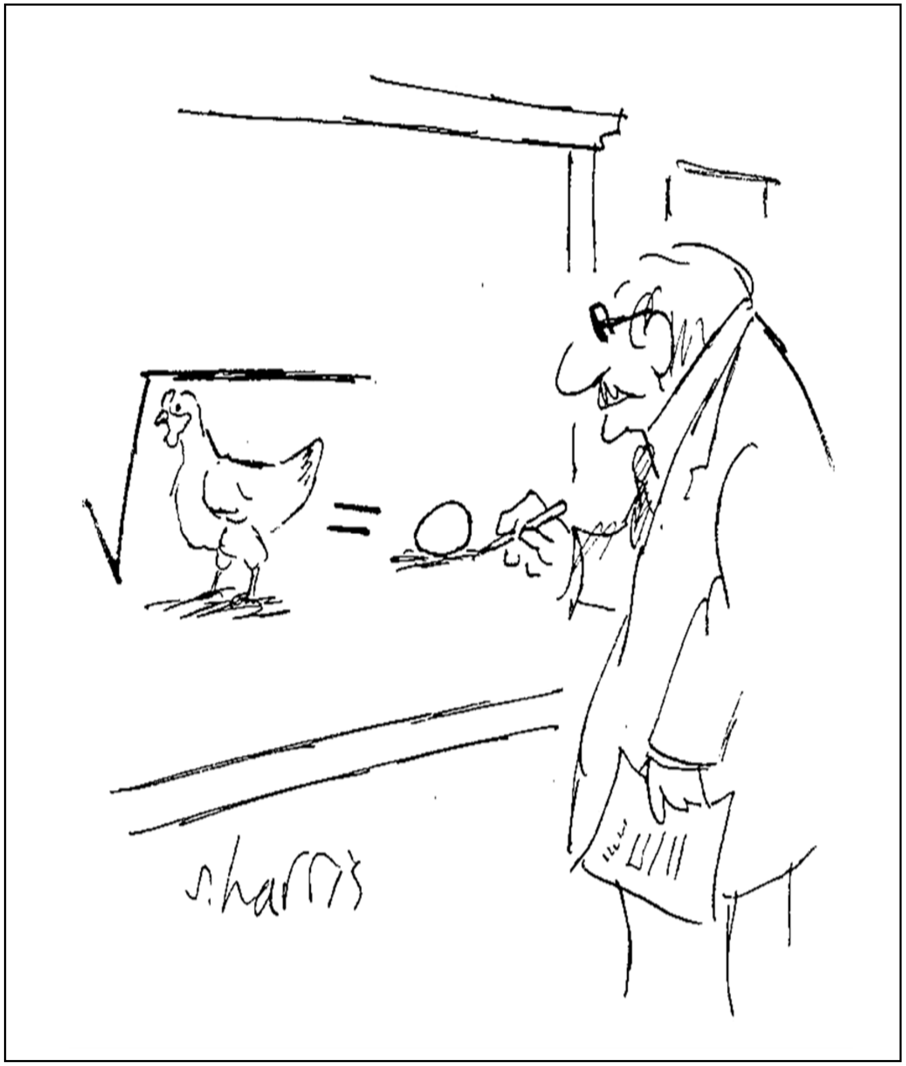
Edition	Released	Revised	Revised	Revised	Revised
First	Jan. 2000	Sep. 2001	Sep. 2003	Oct. 2005	June 2007
		May 2008	May 2009		
Second	May 2010	May 2011	May 2012	May 2015	

VI Function Evaluation

Learn hardware algorithms for evaluating useful functions

- Divisionlike square-rooting algorithms
- Evaluating sin *x*, tanh *x*, ln *x*, . . . by series expansion
- Function evaluation via convergence computation
- Use of tables: the ultimate in simplicity and flexibility

Topics in This Part	
Chapter 21	Square-Rooting Methods
Chapter 22	The CORDIC Algorithms
Chapter 23	Variation in Function Evaluation
Chapter 24	Arithmetic by Table Lookup


s.harris


s.harris
"THIS IS THE PART I ALWAYS HATE."

21 Square-Rooting Methods

Chapter Goals

Learning algorithms and implementations for both digit-at-a-time and convergence square-rooting

Chapter Highlights

Square-rooting part of IEEE 754 standard
Digit-recurrence (divisionlike) algorithms
Convergence or iterative schemes
Square-rooting not special case of division

Square-Rooting Methods: Topics

Topics in This Chapter
21.1 The Pencil-and-Paper Algorithm
21.2 Restoring Shift / Subtract Algorithm
21.3 Binary Nonrestoring Algorithm
21.4 High-Radix Square-Rooting
21.5 Square-Rooting by Convergence
21.6 Fast Hardware Square-Rooters

21.1 The Pencil-and-Paper Algorithm

Notation for our discussion of division algorithms:

z	Radicand	$z_{2k-1}z_{2k-2} \; . \; . \; . \; z_3z_2z_1z_0$
q	Square root	$q_{k-1}q_{k-2} \; . \; . \; . \; q_1q_0$
s	Remainder, $z - q^2$	$s_ks_{k-1}s_{k-2} \; . \; . \; . \; s_1s_0$

Remainder range, $0 \le s \le 2q$ ($k + 1$ digits)

Justification: $s \ge 2q + 1$ would lead to $z = q^2 + s \ge (q + 1)^2$

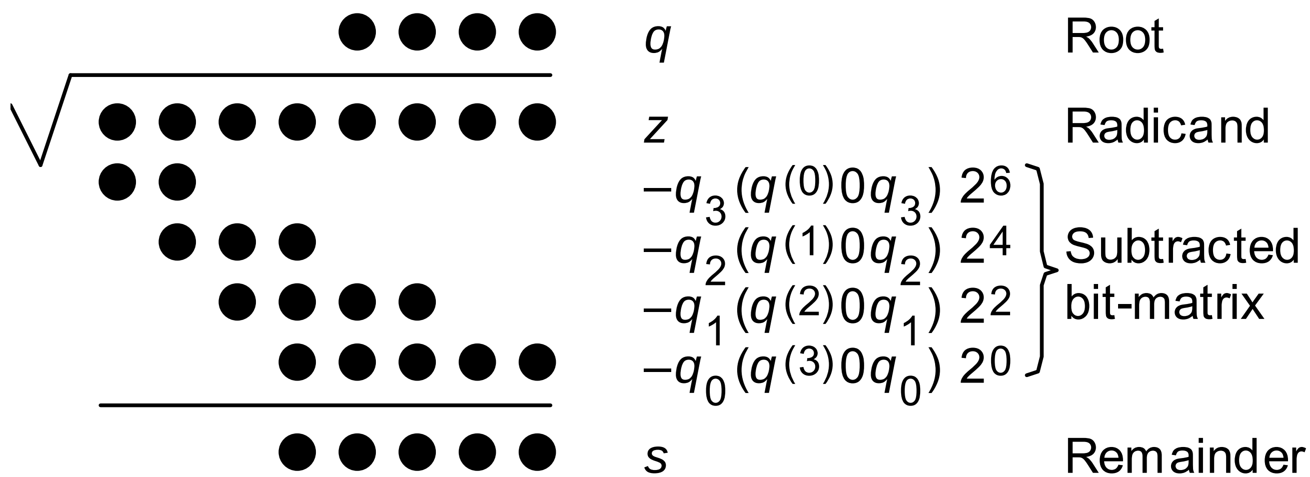


Fig. 21.3 Binary square-rooting in dot notation.

Example of Decimal Square-Rooting

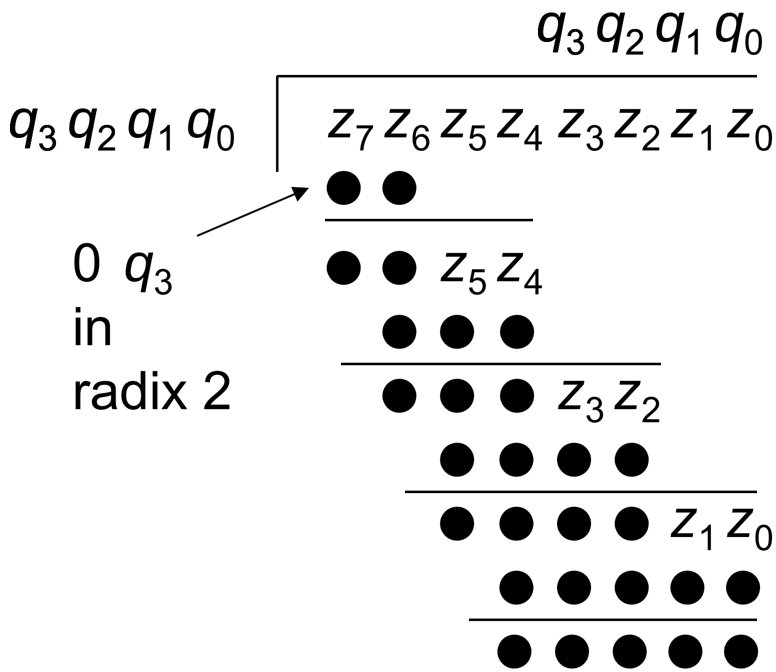
Check: $308^2 + 377 = 94{,}864 + 377 = 95{,}241$

			Root digit	Partial root
q_2 q_1 q_0	$\leftarrow q$			$q^{(0)} = 0$
$\sqrt{9 \mid 5\ 2 \mid 4\ 1}$	$\leftarrow z$		$q_2 = 3$	$q^{(1)} = 3$
9				
0 5 2	$6q_1 \times q_1 \le 52$	"sixty plus q_1"	$q_1 = 0$	$q^{(2)} = 30$
0 0				
5 2 4 1	$60q_0 \times q_0 \le 5241$		$q_0 = 8$	$q^{(3)} = 308$
4 8 6 4				
0 3 7 7	$\leftarrow s = (377)_{ten}$			$q = (308)_{ten}$

Fig. 21.1 Extracting the square root of a decimal integer using the pencil-and-paper algorithm.

Square-Rooting as Division with Unknown Divisor

$$\begin{array}{r|l} & \quad\quad\quad\; q_3\, q_2\, q_1\, q_0 \\ \hline q_3\, q_2\, q_1\, q_0 & \; z_7\, z_6\, z_5\, z_4\, z_3\, z_2\, z_1\, z_0 \end{array}$$



q_3 depends only on $z_7\, z_6$

Justification: For $\varepsilon \neq 0$,
the square of $(q_3 + \varepsilon)r^3$
is $q_3^{\,2} r^6 + \varepsilon(2q_3 + \varepsilon)r^6$,
leading to a change in $z_7\, z_6$

Similarly, q_2 depends only on $z_7\, z_6\, z_5\, z_4$, and so on

Root Digit Selection Rule

The root thus far is denoted by $q^{(i)} = (q_{k-1}q_{k-2} \ldots q_{k-i})_{\text{ten}}$

Attaching the next digit q_{k-i-1}, partial root becomes $q^{(i+1)} = 10\,q^{(i)} + q_{k-i-1}$

The square of $q^{(i+1)}$ is $100(q^{(i)})^2 + 20\,q^{(i)}\,q_{k-i-1} + (q_{k-i-1})^2$

$100(q^{(i)})^2 = (10\,q^{(i)})^2$ subtracted from partial remainder in previous steps

Must subtract $(10(2\,q^{(i)}) + q_{k-i-1}) \times q_{k-i-1}$ to get the new partial remainder

More generally, in radix r, must subtract $(r\,(2\,q^{(i)}) + q_{k-i-1}) \times q_{k-i-1}$

In radix 2, must subtract $(4\,q^{(i)} + q_{k-i-1}) \times q_{k-i-1}$, which is

$4\,q^{(i)} + 1$ for $q_{k-i-1} = 1$, and 0 otherwise
Thus, we use $(q_{k-1}q_{k-2} \ldots q_{k-l}\;0\;1)_{\text{two}}$ in a trial subtraction

Example of Binary Square-Rooting

Check: $10^2 + 18 = 118 = (0111\ 0110)_{\text{two}}$

Computation	Test	Answer	Root digit	Partial root
$q_3 \quad q_2 \quad q_1 \quad q_0 \quad \leftarrow \quad q$				$q^{(0)} = 0$
$\sqrt{0\ 1 \mid 1\ 1 \mid 0\ 1 \mid 1\ 0}$	$\geq 01?$	Yes	$q_3 = 1$	$q^{(1)} = 1$
$0\ 1$				
$0\ 0\ 1\ 1$	$\geq \underline{1}01?$	No	$q_2 = 0$	$q^{(2)} = 10$
$0\ 0\ 0$				
$0\ 1\ 1\ 0\ 1$	$\geq \underline{10}01?$	Yes	$q_1 = 1$	$q^{(3)} = 101$
$1\ 0\ 0\ 1$				
$0\ 1\ 0\ 0\ 1\ 0$	$\geq \underline{101}01?$	No	$q_0 = 0$	$q^{(4)} = 1010$
$0\ 0\ 0\ 0\ 0$				
$1\ 0\ 0\ 1\ 0 \quad \leftarrow \quad s = (18)_{\text{ten}}$				$q = (1010)_{\text{two}} = (10)_{\text{ten}}$

Fig. 21.2 Extracting the square root of a binary integer using the pencil-and-paper algorithm.

21.2 Restoring Shift/Subtract Algorithm

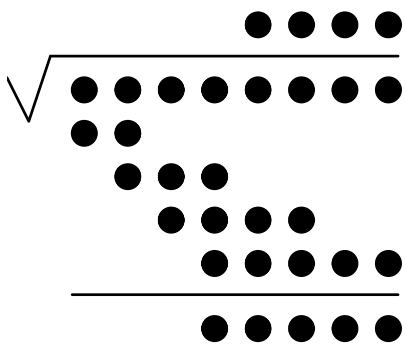


Fig. 21.3

Consistent with the IEEE 754 floating-point standard, we formulate our algorithms for a radicand in the range $1 \le z < 4$ (after possible 1-bit shift for an odd exponent)

$1 \le z < 4$	Radicand	$z_1 z_0 \,.\, z_{-1} z_{-2} \ldots z_{-l}$
$1 \le q < 2$	Square root	$1 \,.\, q_{-1} q_{-2} \ldots q_{-l}$
$0 \le s < 4$	Remainder	$s_1 s_0 \,.\, s_{-1} s_{-2} \ldots s_{-l}$

Binary square-rooting is defined by the recurrence

$$s^{(j)} = 2s^{(j-1)} - q_{-j}(2q^{(j-1)} + 2^{-j} q_{-j}) \quad \text{with } s^{(0)} = z - 1,\ q^{(0)} = 1,\ s^{(j)} = s$$

where $q^{(j)}$ is the root up to its $(-j)$th digit; thus $q = q^{(l)}$

To choose the next root digit $q_{-j} \in \{0, 1\}$, subtract from $2s^{(j-1)}$ the value

$$2q^{(j-1)} + 2^{-j} = (1\, q_{-1}^{(j-1)} \,.\, q_{-2}^{(j-1)} \ldots\ q_{-j+1}^{(j-1)}\ 0\ 1)_{\text{two}}$$

A negative trial difference means $q_{-j} = 0$

Finding the Sq. Root of $z = 1.110110$ via the Restoring Algorithm

Fig. 21.4 Example of sequential binary square-rooting using the restoring algorithm.

		Root digit	Partial root
z (radicand = 118/64)	0 1 . 1 1 0 1 1 0		
$s^{(0)} = z - 1$	0 0 0 . 1 1 0 1 1 0	$q_0 = 1$	1.
$2s^{(0)}$	0 0 1 . 1 0 1 1 0 0		
$-[2 \times (1.)+2^{-1}]$	1 0 . 1		
$s^{(1)}$	1 1 1 . 0 0 1 1 0 0	$q_{-1} = 0$	1.0
$s^{(1)} = 2s^{(0)}$ Restore	0 0 1 . 1 0 1 1 0 0		
$2s^{(1)}$	0 1 1 . 0 1 1 0 0 0		
$-[2 \times (1.0)+2^{-2}]$	1 0 . 0 1		
$s^{(2)}$	0 0 1 . 0 0 1 0 0 0	$q_{-2} = 1$	1.01
$2s^{(2)}$	0 1 0 . 0 1 0 0 0 0		
$-[2 \times (1.01)+2^{-3}]$	1 0 . 1 0 1		
$s^{(3)}$	1 1 1 . 1 0 1 0 0 0	$q_{-3} = 0$	1.010
$s^{(3)} = 2s^{(2)}$ Restore	0 1 0 . 0 1 0 0 0 0		
$2s(3)$	1 0 0 . 1 0 0 0 0 0		
$-[2 \times (1.010)+2^{-4}]$	1 0 . 1 0 0 1		
$s^{(4)}$	0 0 1 . 1 1 1 1 0 0	$q_{-4} = 1$	1.0101
$2s^{(4)}$	0 1 1 . 1 1 1 0 0 0		
$-[2 \times (1.0101)+2^{-5}]$	1 0 . 1 0 1 0 1		
$s^{(5)}$	0 0 1 . 0 0 1 1 1 0	$q_{-5} = 1$	1.01011
$2s^{(5)}$	0 1 0 . 0 1 1 1 0 0		
$-[2\times(1.01011)+2^{-6}]$	1 0 . 1 0 1 1 0 1		
$s^{(6)}$	1 1 1 . 1 0 1 1 1 1	$q_{-6} = 0$	1.010110
$s^{(6)} = 2s^{(5)}$ Restore	0 1 0 . 0 1 1 1 0 0		
s (remainder = 156/64)	0 . 0 0 0 0 1 0 0 1 1 1 0 0		
q (root = 86/64)	1 . 0 1 0 1 1 0		

$q_{-7} = 1$, so round up

Hardware for Restoring Square-Rooting

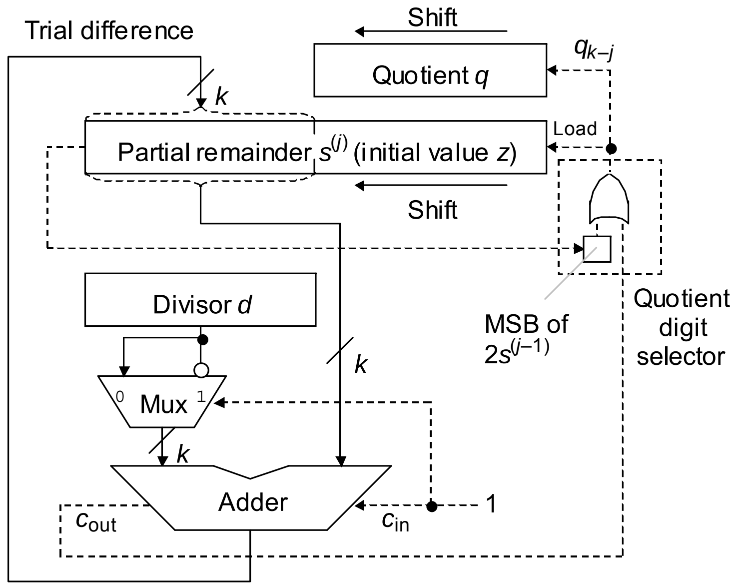


Fig. 13.5 Shift/subtract sequential restoring divider (for comparison).

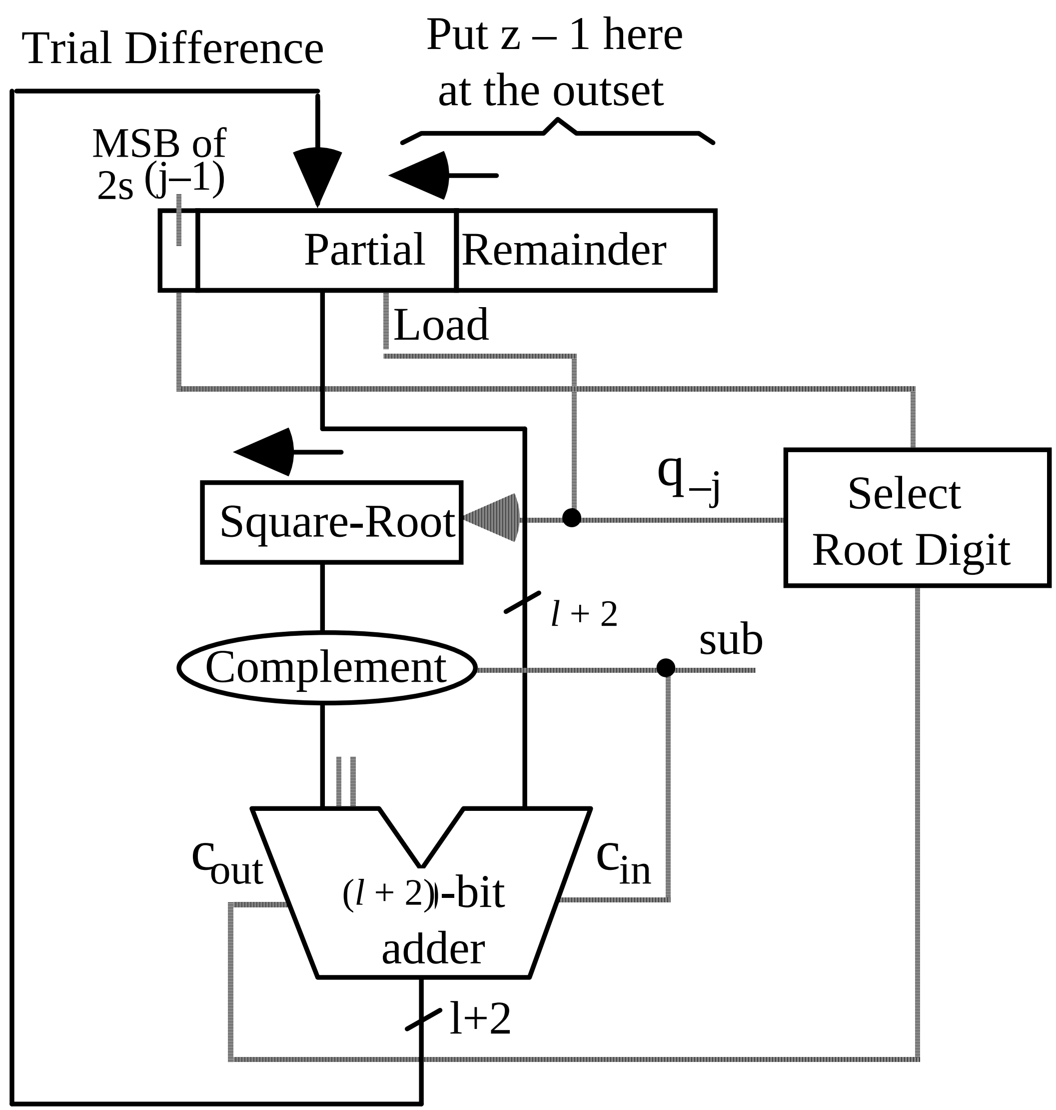


Fig. 21.5 Sequential shift/subtract restoring square-rooter.

Rounding the Square Root

In fractional square-rooting, the remainder is not needed

To round the result, we can produce an extra digit q_{-l-1}:

Truncate for $q_{-l-1} = 0$, round up for $q_{-l-1} = 1$

Midway case, $q_{-l-1} = 1$ followed by all 0s, impossible (Prob. 21.11)

Example: In Fig. 21.4, we had

$(01.110110)_{two} = (1.010110)_{two}^2 + (10.011100)/64$

An extra iteration produces $q_{-7} = 1$
So the root is rounded up to $q = (1.010111)_{two} = 87/64$

The rounded-up value is closer to the root than the truncated version

Original: $118/64 = (86/64)^2 + 156/(64)^2$
Rounded: $118/64 = (87/64)^2 - 17/(64)^2$

21.3 Binary Nonrestoring Algorithm

As in nonrestoring division, nonrestoring square-rooting implies:

Root digits in {$^{-}1$, 1}
On-the-fly conversion to binary
Possible final correction

The case $q_{-j} = 1$ (nonnegative partial remainder), is handled as in the restoring algorithm; i.e., it leads to the trial subtraction of

$$q_{-j}\,[2q^{(j-1)} + 2^{-j} q_{-j}] \;=\; 2q^{(j-1)} + 2^{-j}$$

For $q_{-j} = {}^{-}1$, we must subtract

$$q_{-j}\,[2q^{(j-1)} + 2^{-j} q_{-j}] \;=\; -[2q^{(j-1)} - 2^{-j}]$$

which is equivalent to adding $\underline{2q^{(j-1)} - 2^{-j}}$

This term cannot be formed by concatenation

Slight complication, compared with nonrestoring division

Finding the Sq. Root of z = 1.110110 via the Nonrestoring Algorithm

		Root digit	Partial root
z (radicand = 118/64)	0 1 . 1 1 0 1 1 0		
$s^{(0)} = z - 1$	0 0 0 . 1 1 0 1 1 0	$q_0 = 1$	1.
$2s^{(0)}$	0 0 1 . 1 0 1 1 0 0	$q_{-1} = 1$	1.1
$-[2 \times (1.)+2^{-1}]$	1 0 . 1		
$s^{(1)}$	1 1 1 . 0 0 1 1 0 0	$q_{-2} = {}^{-}1$	1.01
$2s^{(1)}$	1 1 0 . 0 1 1 0 0 0		
$+[2 \times (1.1)-2^{-2}]$	1 0 . 1 1		
$s^{(2)}$	0 0 1 . 0 0 1 0 0 0	$q_{-3} = 1$	1.011
$2s^{(2)}$	0 1 0 . 0 1 0 0 0 0		
$-[2 \times (1.01)+2^{-3}]$	1 0 . 1 0 1		
$s^{(3)}$	1 1 1 . 1 0 1 0 0 0	$q_{-4} = {}^{-}1$	1.0101
$2s(3)$	1 1 1 . 0 1 0 0 0 0		
$+[2 \times (1.011)-2^{-4}]$	1 0 . 1 0 1 1		
$s^{(4)}$	0 0 1 . 1 1 1 1 0 0	$q_{-5} = 1$	1.01011
$2s^{(4)}$	0 1 1 . 1 1 1 0 0 0		
$-[2 \times (1.0101)+2^{-5}]$	1 0 . 1 0 1 0 1		
$s^{(5)}$	0 0 1 . 0 0 1 1 1 0	$q_{-6} = 1$	1.010111
$2s^{(5)}$	0 1 0 . 0 1 1 1 0 0		
$-[2\times(1.01011)+2^{-6}]$	1 0 . 1 0 1 1 0 1		
$s^{(6)}$	1 1 1 . 1 0 1 1 1 1	Negative; Correct	(–17/64)
$+[2\times(1.01011)-2^{-6}]$	1 0 . 1 0 1 1 0 1		
$s^{(6)}$ Corrected	0 1 0 . 0 1 1 1 0 0		(156/64)
s (remainder = 156/64)	0 . 0 0 0 0 1 0 0 1 1 1 0 0		($156/64^2$)
q (binary)	1 . 0 1 0 1 1 1		(87/64)
q (corrected binary)	1 . 0 1 0 1 1 0		(86/64)

Fig. 21.6 Example of nonrestoring binary square-rooting.

Some Details for Nonrestoring Square-Rooting

Depending on the sign of the partial remainder, add:

(positive) Add $2q^{(j-1)} + 2^{-j}$ Concatenate 01 to the end of $q^{(j-1)}$

(negative) Sub. $2q^{(j-1)} - 2^{-j}$ Cannot be formed by concatenation

Solution: We keep $q^{(j-1)}$ and $q^{(j-1)} - 2^{-j+1}$ in registers Q (partial root) and Q* (diminished partial root), respectively. Then:

$q_{-j} = 1$ Subtract $2q^{(j-1)} + 2^{-j}$ formed by shifting Q 01

$q_{-j} = -1$ Add $2q^{(j-1)} - 2^{-j}$ formed by shifting Q*11

Updating rules for Q and Q* registers:

$q_{-j} = 1 \Rightarrow$ Q := Q 1 Q* := Q 0

$q_{-j} = -1 \Rightarrow$ Q := Q*1 Q* := Q*0

Additional rule for SRT-like algorithm that allow $q_{-j} = 0$ as well:

$q_{-j} = 0 \Rightarrow$ Q := Q 0 Q* := Q*1

21.4 High-Radix Square-Rooting

Basic recurrence for fractional radix-r square-rooting:

$$s^{(j)} = rs^{(j-1)} - q_{-j}(2q^{(j-1)} + r^{-j}q_{-j})$$

As in radix-2 nonrestoring algorithm, we can use two registers Q and Q* to hold $q^{(j-1)}$ and its diminished version $q^{(j-1)} - r^{-j+1}$, respectively, suitably updating them in each step

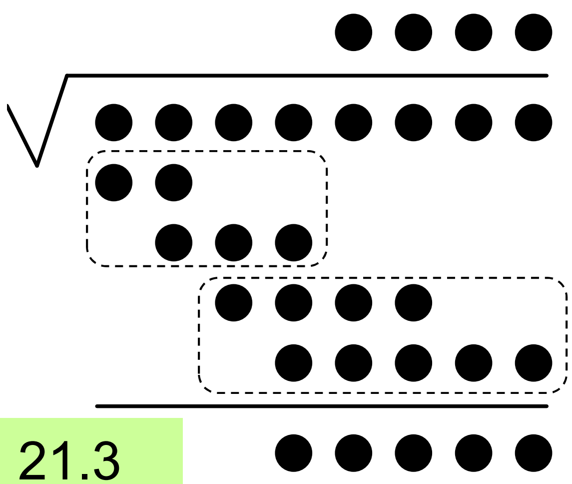
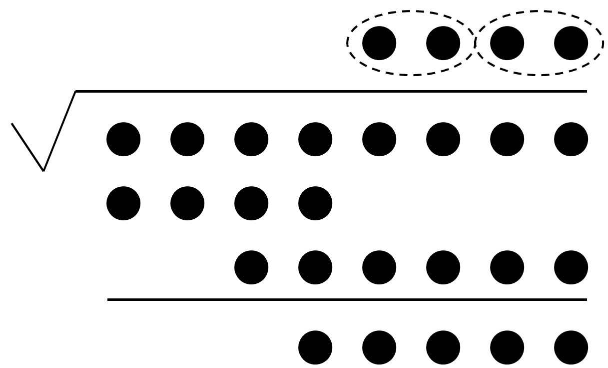


Fig. 21.3



Radix-4 square-rooting in dot notation

An Implementation of Radix-4 Square-Rooting

$r = 4$, root digit set [–2, 2]

$s^{(j)} = rs^{(j-1)} - q_{-j}(2q^{(j-1)} + r^{-j}q_{-j})$

Q* holds $q^{(j-1)} - 4^{-j+1} = q^{(j-1)} - 2^{-2j+2}$. Then, one of the following values must be subtracted from, or added to, the shifted partial remainder $rs^{(j-1)}$

$q_{-j} = 2$	Subtract	$4q^{(j-1)} + 2^{-2j+2}$	double-shift	Q 010
$q_{-j} = 1$	Subtract	$2q^{(j-1)} + 2^{-2j}$	shift	Q 001
$q_{-j} = {}^{-}1$	Add	$2q^{(j-1)} - 2^{-2j}$	shift	Q*111
$q_{-j} = {}^{-}2$	Add	$4q^{(j-1)} - 2^{-2j+2}$	double-shift	Q*110

Updating rules for Q and Q* registers:

$q_{-j} = 2$	$\Rightarrow$	Q := Q 10	Q* := Q 01
$q_{-j} = 1$	$\Rightarrow$	Q := Q 01	Q* := Q 00
$q_{-j} = 0$	$\Rightarrow$	Q := Q 00	Q* := Q*11
$q_{-j} = {}^{-}1$	$\Rightarrow$	Q := Q*11	Q* := Q*10
$q_{-j} = {}^{-}2$	$\Rightarrow$	Q := Q*10	Q* := Q*01

Note that the root is obtained in binary form (no conversion needed!)

Keeping the Partial Remainder in Carry-Save Form

As in fast division, root digit selection can be based on a few bits of the shifted partial remainder $4s^{(j-1)}$ and of the partial root $q^{(j-1)}$

This would allow us to keep s in carry-save form

One extra bit of each component of s (sum and carry) must be examined

Can use the same lookup table for quotient digit and root digit selection

To see how, compare recurrences for radix-4 division and square-rooting:

Division: $s^{(j)} = 4s^{(j-1)} - q_{-j}\,d$

Square-rooting: $s^{(j)} = 4s^{(j-1)} - q_{-j}\,(2q^{(j-1)} + 4^{-j}q_{-j})$

To keep magnitudes of partial remainders for division and square-rooting comparable, we can perform radix-4 square-rooting using the digit set

$$\{-1, -½ , 0 , ½ , 1\}$$

Can convert from the digit set above to the digit set [–2, 2], or directly to binary, with no extra computation

21.5 Square-Rooting by Convergence

Newton-Raphson method

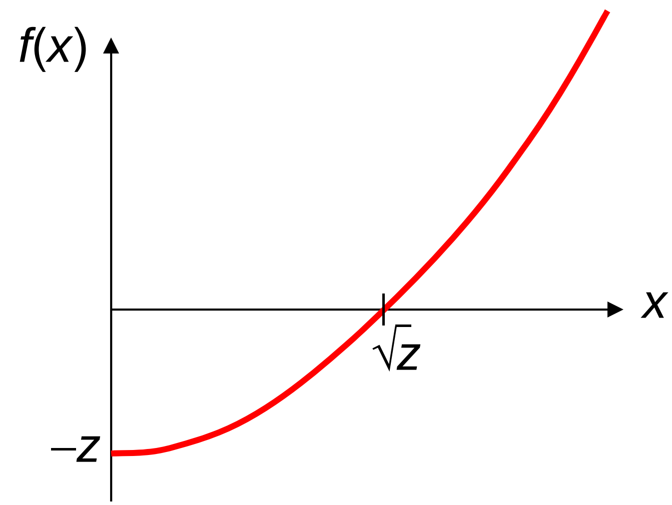
Choose $f(x) = x^2 - z$ with a root at $x = \sqrt{z}$

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

$$x^{(i+1)} = 0.5(x^{(i)} + z / x^{(i)})$$

Each iteration: division, addition, 1-bit shift

Convergence is quadratic



For $0.5 \le z < 1$, a good starting approximation is $(1 + z)/2$

This approximation needs no arithmetic

The error is 0 at $z = 1$ and has a max of 6.07% at $z = 0.5$

The hardware approximation method of Schwarz and Flynn, using the tree circuit of a fast multiplier, can provide a much better approximation (e.g., to 16 bits, needing only two iterations for 64 bits of precision)

Initial Approximation Using Table Lookup

Table-lookup can yield a better starting estimate $x^{(0)}$ for $\sqrt{z}$

For example, with an initial estimate accurate to within 2^{-8}, three iterations suffice to increase the accuracy of the root to 64 bits

$$x^{(i+1)} = 0.5(x^{(i)} + z / x^{(i)})$$

Example 21.1: Compute the square root of $z = (2.4)_{\text{ten}}$

$x^{(0)}$	read out from table	$= 1.5$	accurate to 10^{-1}
$x^{(1)}$	$= 0.5(x^{(0)} + 2.4 / x^{(0)})$	$= 1.550\,000\,000$	accurate to 10^{-2}
$x^{(2)}$	$= 0.5(x^{(1)} + 2.4 / x^{(1)})$	$= 1.549\,193\,548$	accurate to 10^{-4}
$x^{(3)}$	$= 0.5(x^{(2)} + 2.4 / x^{(2)})$	$= 1.549\,193\,338$	accurate to 10^{-8}

Check: $(1.549\,193\,338)^2 = 2.399\,999\,999$

Convergence Square-Rooting without Division

Rewrite the square-root recurrence as:

$$x^{(i+1)} = 0.5(x^{(i)} + z / x^{(i)})$$

$$x^{(i+1)} = x^{(i)} + 0.5(1/x^{(i)})(z - (x^{(i)})^2) = x^{(i)} + 0.5\gamma(x^{(i)})(z - (x^{(i)})^2)$$

where $\gamma(x^{(i)})$ is an approximation to $1/x^{(i)}$ obtained by a simple circuit or read out from a table

Because of the approximation used in lieu of the exact value of $1/x^{(i)}$, convergence rate will be less than quadratic

Alternative: Use the recurrence above, but find the reciprocal iteratively; thus interlacing the two computations

Using the function $f(y) = 1/y - x$ to compute $1/x$, we get:

$$x^{(i+1)} = 0.5(x^{(i)} + z\, y^{(i)})$$
$$y^{(i+1)} = y^{(i)}(2 - x^{(i)} y^{(i)})$$

3 multiplications, 2 additions, and a 1-bit shift per iteration

Convergence is less than quadratic but better than linear

Example for Division-Free Square-Rooting

$$x^{(i+1)} = 0.5(x^{(i)} + z\,y^{(i)})$$
$$y^{(i+1)} = y^{(i)}(2 - x^{(i)}y^{(i)})$$

x converges to $\sqrt{z}$
y converges to $1/\sqrt{z}$

Example 21.2: Compute $\sqrt{1.4}$, beginning with $x^{(0)} = y^{(0)} = 1$

$$\begin{array}{llllll}
x^{(1)} & = & 0.5(x^{(0)} + 1.4\,y^{(0)}) & = & 1.200\,000\,000 & \\
y^{(1)} & = & y^{(0)}(2 - x^{(0)}y^{(0)}) & = & 1.000\,000\,000 & \\
x^{(2)} & = & 0.5(x^{(1)} + 1.4\,y^{(1)}) & = & 1.300\,000\,000 & \\
y^{(2)} & = & y^{(1)}(2 - x^{(1)}y^{(1)}) & = & 0.800\,000\,000 & \\
x^{(3)} & = & 0.5(x^{(2)} + 1.4\,y^{(2)}) & = & 1.210\,000\,000 & \\
y^{(3)} & = & y^{(2)}(2 - x^{(2)}y^{(2)}) & = & 0.768\,000\,000 & \\
x^{(4)} & = & 0.5(x^{(3)} + 1.4\,y^{(3)}) & = & 1.142\,600\,000 & \\
y^{(4)} & = & y^{(3)}(2 - x^{(3)}y^{(3)}) & = & 0.822\,312\,960 & \\
x^{(5)} & = & 0.5(x^{(4)} + 1.4\,y^{(4)}) & = & 1.146\,919\,072 & \\
y^{(5)} & = & y^{(4)}(2 - x^{(4)}y^{(4)}) & = & 0.872\,001\,394 & \\
x^{(6)} & = & 0.5(x^{(5)} + 1.4\,y^{(5)}) & = & 1.183\,860\,512 & \cong \sqrt{1.4}
\end{array}$$

Check: $(1.183\,860\,512)^2 = 1.401\,525\,712$

Another Division-Free Convergence Scheme

Based on computing $1/\sqrt{z}$, which is then multiplied by z to obtain $\sqrt{z}$
The function $f(x) = 1/x^2 - z$ has a root at $x = 1/\sqrt{z}$ $\quad (f'(x) = -2/x^3)$

$$x^{(i+1)} = 0.5x^{(i)}(3 - z(x^{(i)})^2)$$

3 multiplications, 1 addition, and a 1-bit shift per iteration

Quadratic convergence

Example 21.3: Compute the square root of $z = (.5678)_{\text{ten}}$

$x^{(0)}$	read out from table	=	1.3
$x^{(1)}$	$= 0.5x^{(0)}(3 - 0.5678(x^{(0)})^2)$	=	1.326 271 700
$x^{(2)}$	$= 0.5x^{(1)}(3 - 0.5678(x^{(1)})^2)$	=	1.327 095 128
$\sqrt{z}$	$\cong z \times x^{(2)}$	=	0.753 524 613

Cray 2 supercomputer used this method. Initially, instead of $x^{(0)}$, the two values $1.5x^{(0)}$ and $0.5(x^{(0)})^3$ are read out from a table, requiring only 1 multiplication in the first iteration. The value $x^{(1)}$ thus obtained is accurate to within half the machine precision, so only one other iteration is needed (in all, 5 multiplications, 2 additions, 2 shifts)

21.6 Fast Hardware Square-Rooters

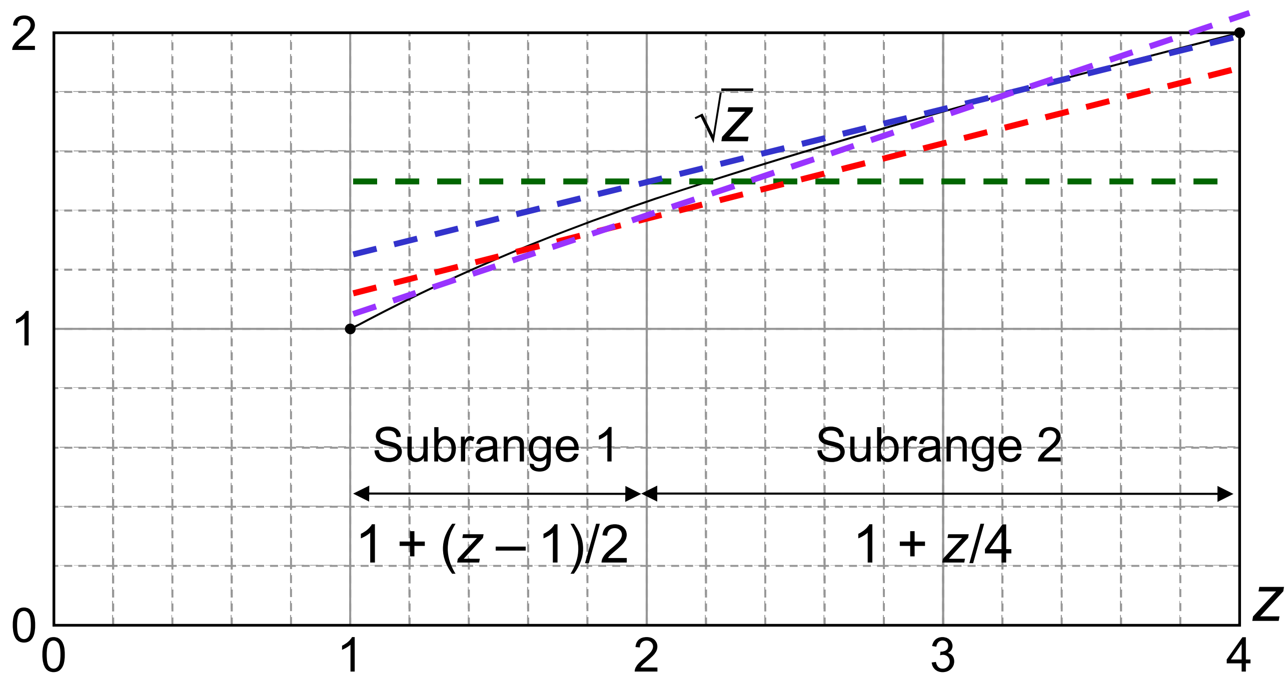
Combinational hardware square-rooter serve two purposes:
1. Approximation to start up or speed up convergence methods
2. Replace digit recurrence or convergence methods altogether

$\sqrt{z} \approx 1.5$

$\sqrt{z} \approx 1 + z/4$

$\sqrt{z} \approx 7/8 + z/4$

$\sqrt{z} \approx 17/24 + z/3$
Best linear approx.

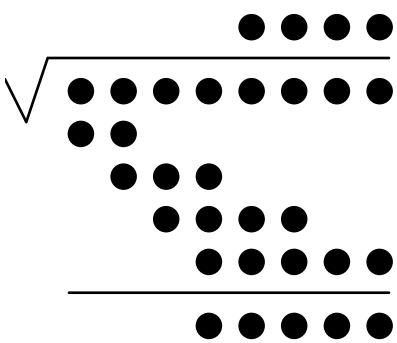


More subranges
Better approx in each

Fig. 21.7 Plot of the function $\sqrt{z}$ for $1 \le z < 4$.

Nonrestoring Array Square-Rooters

Array square-rooters can be derived from the dot-notation representation in much the same way as array dividers



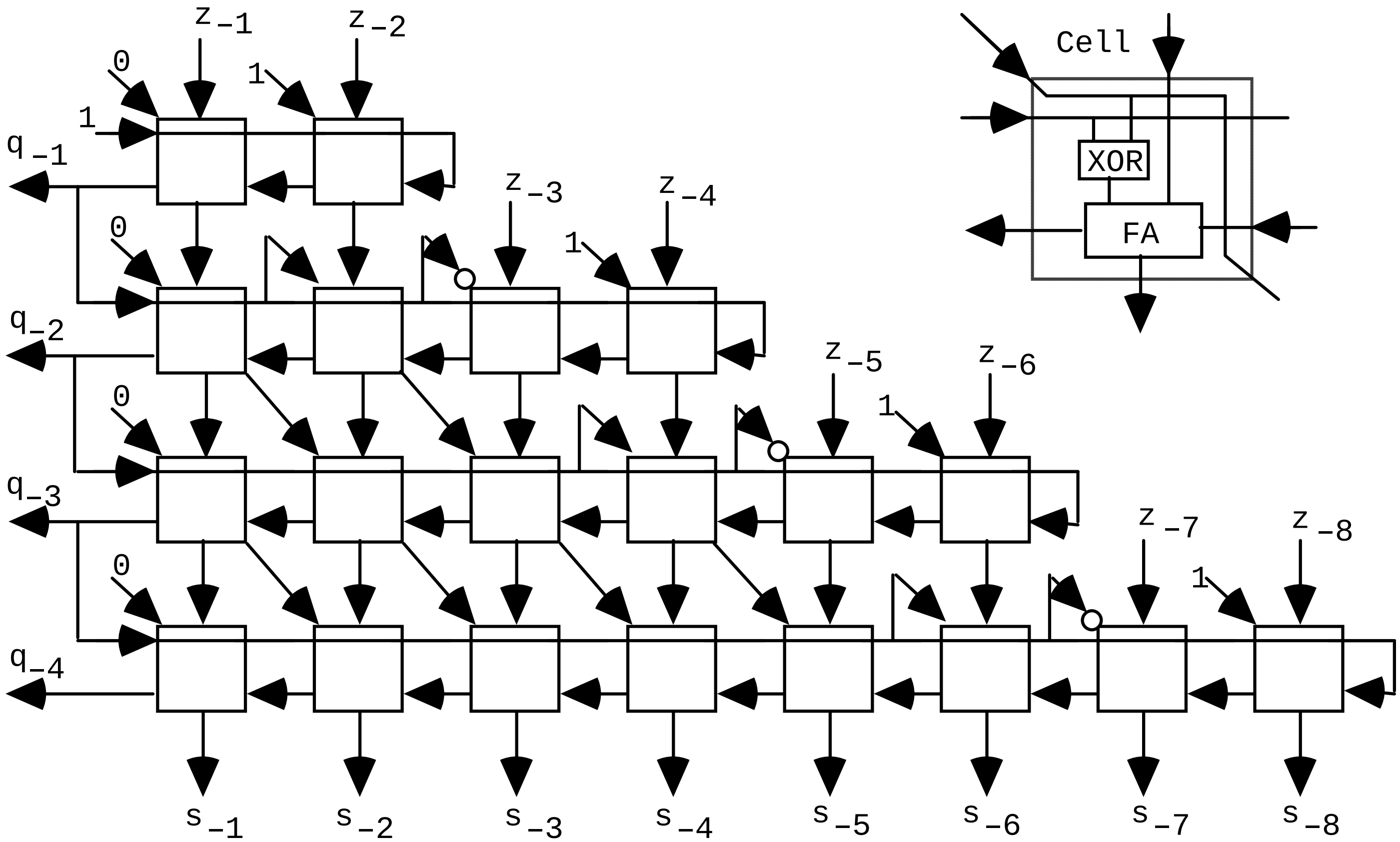
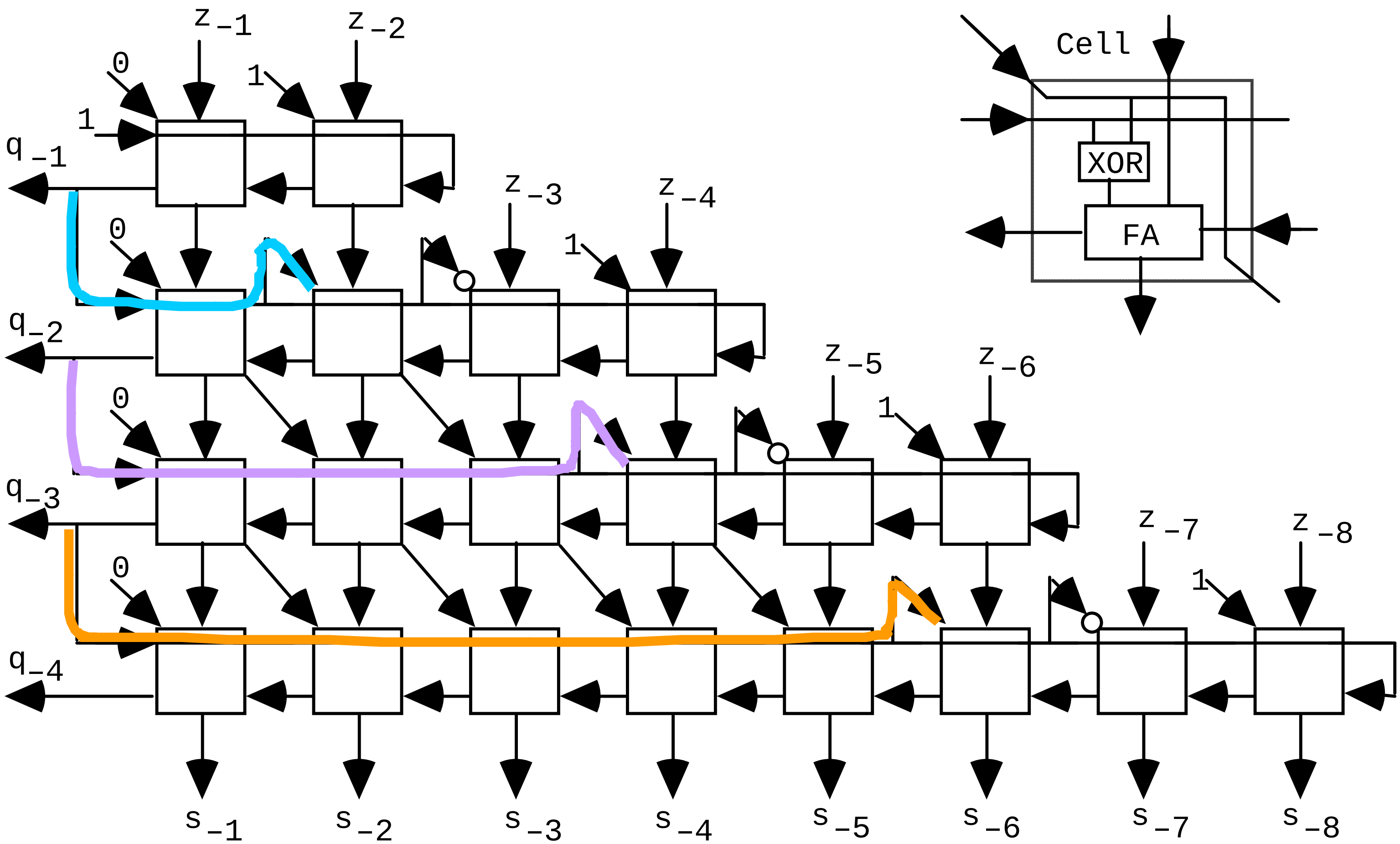


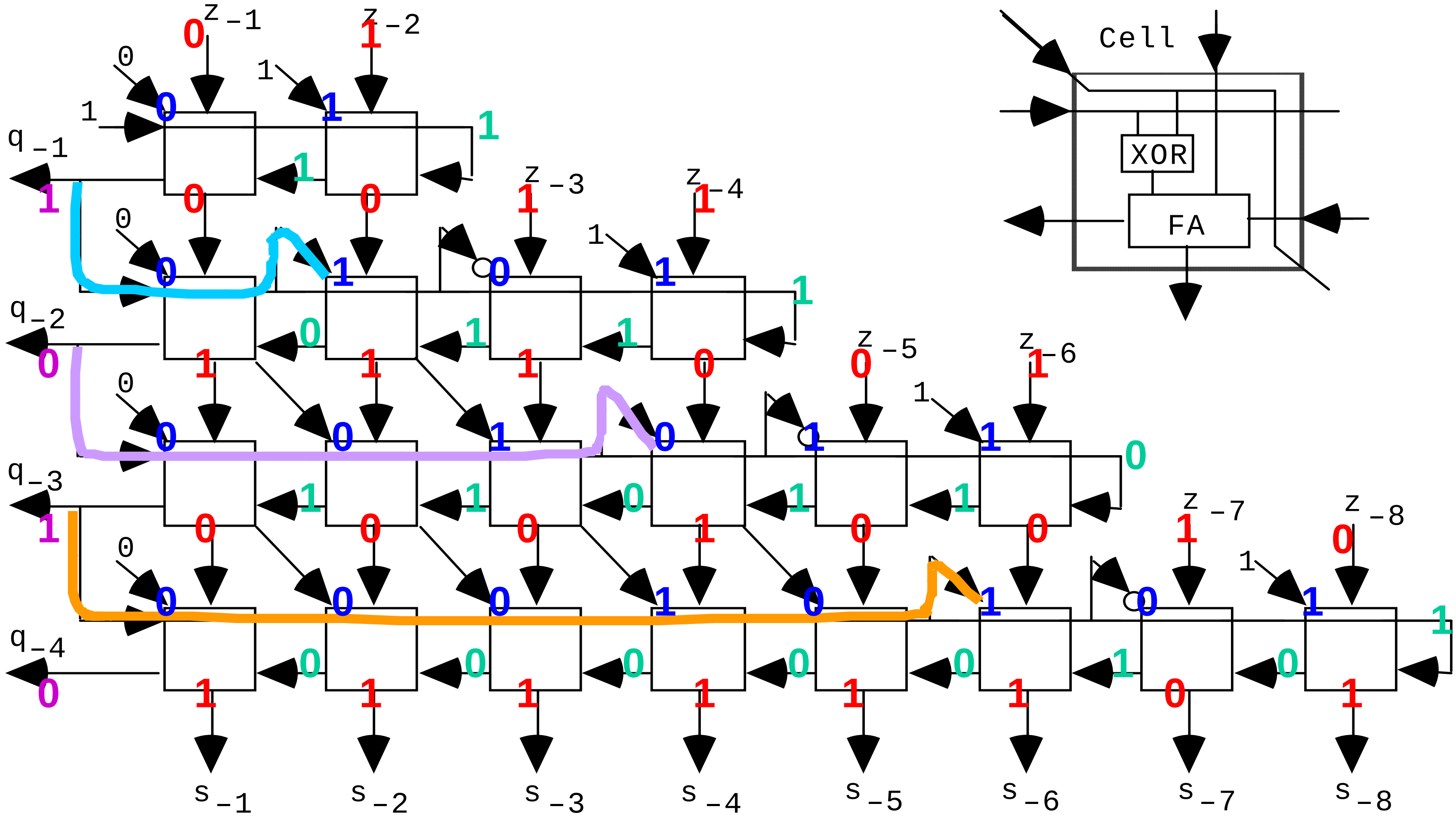
Fig. 21.8 Nonrestoring array square-rooter built of controlled add/subtract cells incorporating full adders (FAs) and XOR gates.

Understanding the Array Square-Rooter Design



Partial root, transferred diagonally from row to row, is appended with: 01 if the last root digit was 1; with 11 if the last root digit was 0

Nonrestoring Array Square-Rooter in Action



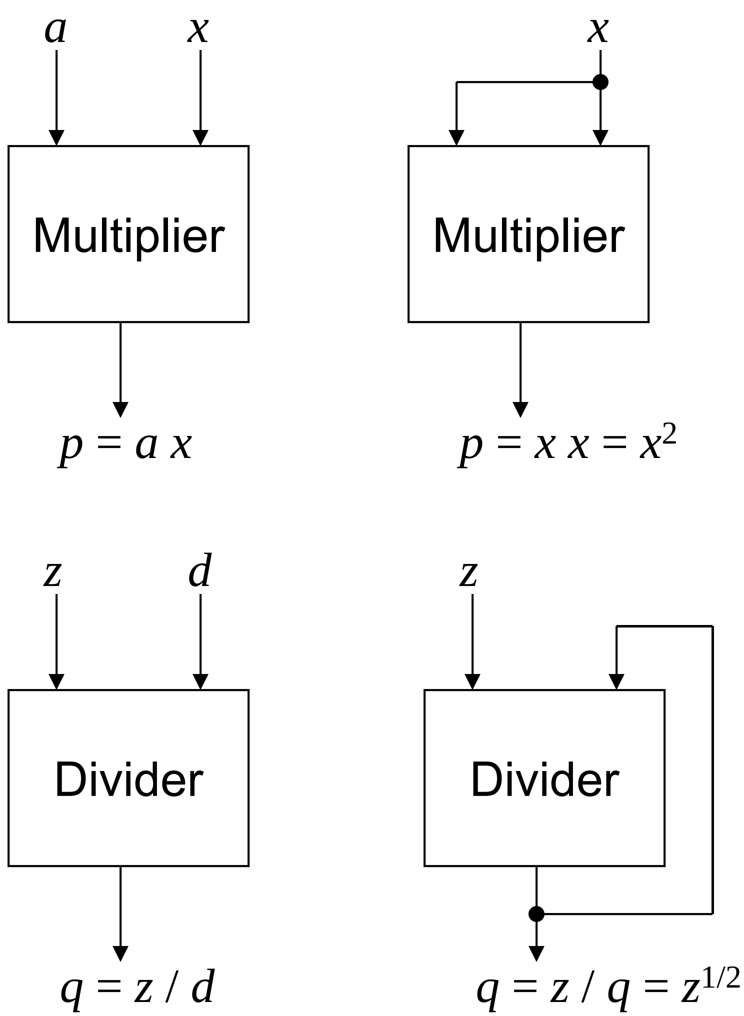
Check: 118/256 = $(10/16)^2 + (–3/256)$? Note that the answer is approximate (to within 1 *ulp*) due to there being no final correction

Digit-at-a-Time Version of the Previous Example

In this example, z is ¼ of that in Fig. 21.6. Subtraction (addition) uses the term $2q + 2^{-i}$ $(2q - 2^{-i})$.

		Root digit	Partial root
z = 118/256	. 0 1 1 1 0 1 1 0		
$s^{(0)} = z$	0 0 . 0 1 1 1 0 1 1 0		
$2s^{(0)}$	0 0 0 . 1 1 1 0 1 1 0		
$-(2q + 2^{-1})$	1 1 . 1		
$s^{(1)}$	0 0 . 0 1 1 0 1 1 0	$q_{-1} = 1$	$q = .1$
$2s^{(1)}$	0 0 0 . 1 1 0 1 1 0		
$-(2q + 2^{-2})$	1 0 . 1 1		
$s^{(2)}$	1 1 . 1 0 0 1 1 0	$q_{-2} = 0$	$q = .10$
$2s^{(2)}$	1 1 1 . 0 0 1 1 0		
$+(2q - 2^{-3})$	0 0 . 1 1 1		
$s^{(3)}$	0 0 . 0 0 0 1 0	$q_{-3} = 1$	$q = .101$
$2s^{(3)}$	0 0 0 . 0 0 1 0		
$-(2q + 2^{-4})$	1 0 . 1 0 1 1		
$s^{(4)}$	1 0 . 1 1 0 1	$q_{-4} = 0$	$q = .1010$

Square Rooting Is Not a Special Case of Division



Multiplier, with both inputs connected to same value, becomes a squarer

But, direct realization of squarer leads to simpler and faster circuit

Divider can’t be used as square-rooter via feedback connection

Direct square-rooter realization does not lead to simpler or faster circuit

22 The CORDIC Algorithms

Chapter Goals

Learning a useful convergence method for evaluating trigonometric and other functions

Chapter Highlights

Basic CORDIC idea: rotate a vector with end point at $(x,y) = (1,0)$ by the angle z to put its end point at $(\cos z, \sin z)$

Other functions evaluated similarly

Complexity comparable to division

The CORDIC Algorithms: Topics

Topics in This Chapter
22.1 Rotations and Pseudorotations
22.2 Basic CORDIC Iterations
22.3 CORDIC Hardware
22.4 Generalized CORDIC
22.5 Using the CORDIC Method
22.6 An Algebraic Formulation

22.1 Rotations and Pseudorotations

Evaluation of trigonometric, hyperbolic, and other common functions, such as log and exp, is needed in many computations

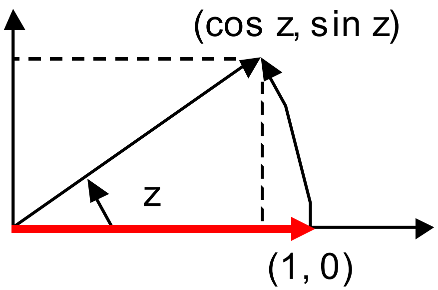
It comes as a surprise to most people that such elementary functions can be evaluated in time that is comparable to division time or a fairly small multiple of it

Some groups advocate including these functions in IEEE 754, thus requiring that they be evaluated exactly, except for the final rounding

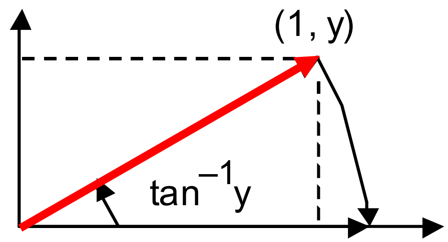
Progress has been made toward such properly rounded elementary functions, but the cost of achieving this goal is still prohibitive

CORDIC is a low-cost method that achieves the reasonable accuracy of about 1 *ulp*, but does not guarantee proper rounding

Key Ideas on which CORDIC Is Based



start at (1, 0)
rotate by z
get cos z, sin z



start at (1, y)
rotate until y = 0
rotation amount is $\tan^{-1} y$

COordinate **R**otation **DI**gital **C**omputer used this method in the1950s; modern electronic calculators also use it

If we have a computationally efficient way of rotating a vector, we can evaluate cos, sin, and $\tan^{-1}$ functions

Rotation by an arbitrary angle is difficult, so we:

Perform psuedorotations that require simpler operations
Use special angles to synthesize the desired angle *z*

$$z = \alpha^{(1)} + \alpha^{(2)} + \ldots + \alpha^{(m)}$$

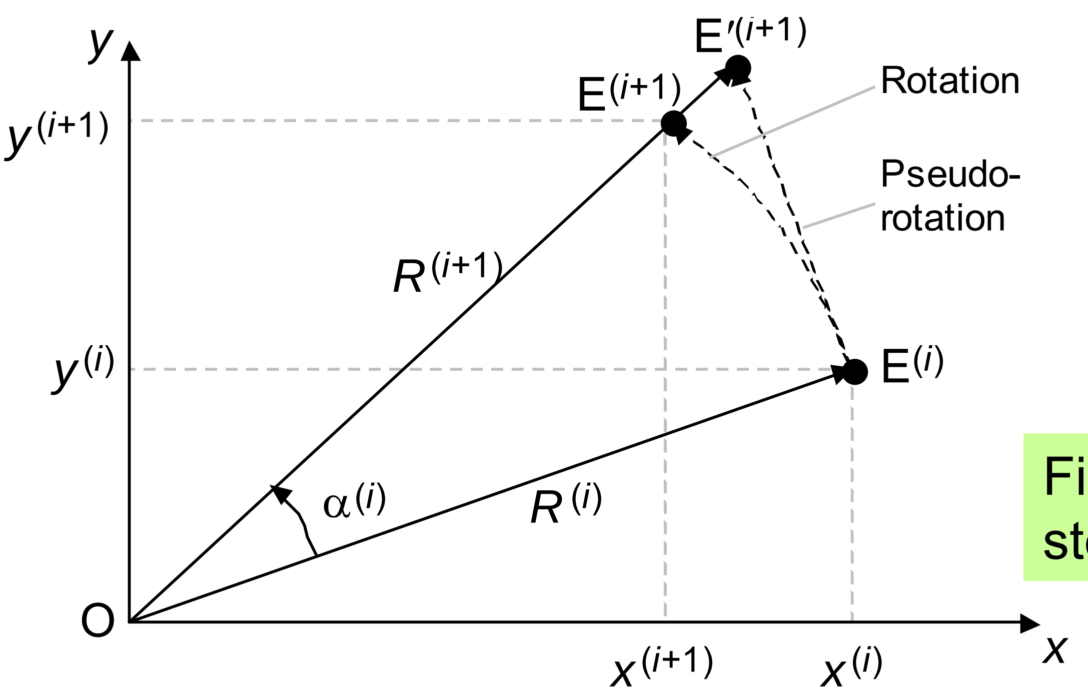
Rotating a Vector $(x^{(i)}, y^{(i)})$ by the Angle $\alpha^{(i)}$

$$x^{(i+1)} = x^{(i)} \cos \alpha^{(i)} - y^{(i)} \sin \alpha^{(i)} = (x^{(i)} - y^{(i)} \tan \alpha^{(i)}) / (1 + \tan^2 \alpha^{(i)})^{1/2}$$

$$y^{(i+1)} = y^{(i)} \cos \alpha^{(i)} + x^{(i)} \sin \alpha^{(i)} = (y^{(i)} + x^{(i)} \tan \alpha^{(i)}) / (1 + \tan^2 \alpha^{(i)})^{1/2}$$

$$z^{(i+1)} = z^{(i)} - \alpha^{(i)}$$

Recall that $\cos \theta = 1 / (1 + \tan^2 \theta)^{1/2}$



Our strategy: Eliminate the terms $(1 + \tan^2 \alpha^{(i)})^{1/2}$ and choose the angles $\alpha^{(i)}$ so that $\tan \alpha^{(i)}$ is a power of 2; need two shift-adds

Fig. 22.1 A pseudorotation step in CORDIC

Pseudorotating a Vector $(x^{(i)}, y^{(i)})$ by the Angle $\alpha^{(i)}$

$$\left.\begin{aligned} x^{(i+1)} &= x^{(i)} - y^{(i)} \tan \alpha^{(i)} \\ y^{(i+1)} &= y^{(i)} + x^{(i)} \tan \alpha^{(i)} \\ z^{(i+1)} &= z^{(i)} - \alpha^{(i)} \end{aligned}\right\}$$

Pseudorotation: Whereas a real rotation does not change the length $R_{(i)}$ of the vector, a pseudorotation step increases its length to:

$$R^{(i+1)} = R^{(i)} / \cos \alpha^{(i)} = R^{(i)} (1 + \tan^2 \alpha^{(i)})^{1/2}$$

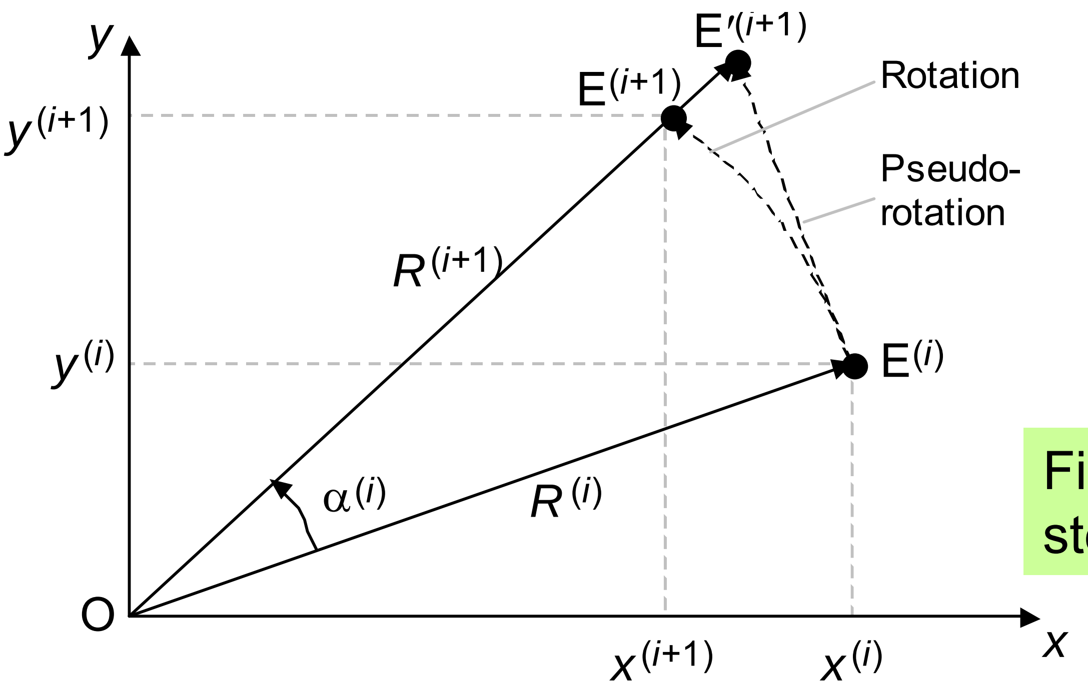


Fig. 22.1 A pseudorotation step in CORDIC

A Sequence of Rotations or Pseudorotations

$$x^{(m)} = x \cos(\Sigma\alpha^{(i)}) - y \sin(\Sigma\alpha^{(i)})$$
$$y^{(m)} = y \cos(\Sigma\alpha^{(i)}) + x \sin(\Sigma\alpha^{(i)})$$
$$z^{(m)} = z - (\Sigma\alpha^{(i)})$$

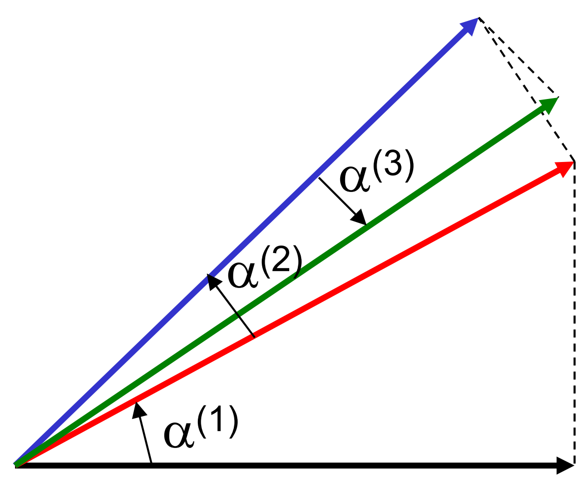
After m real rotations by $\alpha^{(1)}, \alpha^{(2)}, \ldots, \alpha^{(m)}$, given $x^{(0)} = x$, $y^{(0)} = y$, and $z^{(0)} = z$

$$x^{(m)} = K(x \cos(\Sigma\alpha^{(i)}) - y \sin(\Sigma\alpha^{(i)}))$$
$$y^{(m)} = K(y \cos(\Sigma\alpha^{(i)}) + x \sin(\Sigma\alpha^{(i)}))$$
$$z^{(m)} = z - (\Sigma\alpha^{(i)})$$

After m pseudorotations by $\alpha^{(1)}, \alpha^{(2)}, \ldots, \alpha^{(m)}$, given $x^{(0)} = x$, $y^{(0)} = y$, and $z^{(0)} = z$

where $K = \prod(1 + \tan^2 \alpha^{(i)})^{1/2}$ is a constant if angles of rotation are always the same, differing only in sign or direction

Question: Can we find a set of angles so that any angle can be synthesized from all of them with appropriate signs?



22.2 Basic CORDIC Iterations

$$\left.\begin{aligned} x^{(i+1)} &= x^{(i)} - d_i\, y^{(i)}\, 2^{-i} \\ y^{(i+1)} &= y^{(i)} + d_i\, x^{(i)}\, 2^{-i} \\ z^{(i+1)} &= z^{(i)} - d_i \tan^{-1} 2^{-i} \end{aligned}\right\}$$

$$= z^{(i)} - d_i\, e^{(i)}$$

CORDIC iteration: In step i, we pseudorotate by an angle whose tangent is $d_i\, 2^{-i}$ (the angle $e^{(i)}$ is fixed, only direction d_i is to be picked)

i	$e^{(i)}$ in degrees (approximate)	$e^{(i)}$ in radians (precise)
0	45.0	0.785 398 163
1	26.6	0.463 647 609
2	14.0	0.244 978 663
3	7.1	0.124 354 994
4	3.6	0.062 418 810
5	1.8	0.031 239 833
6	0.9	0.015 623 728
7	0.4	0.007 812 341
8	0.2	0.003 906 230
9	0.1	0.001 953 123

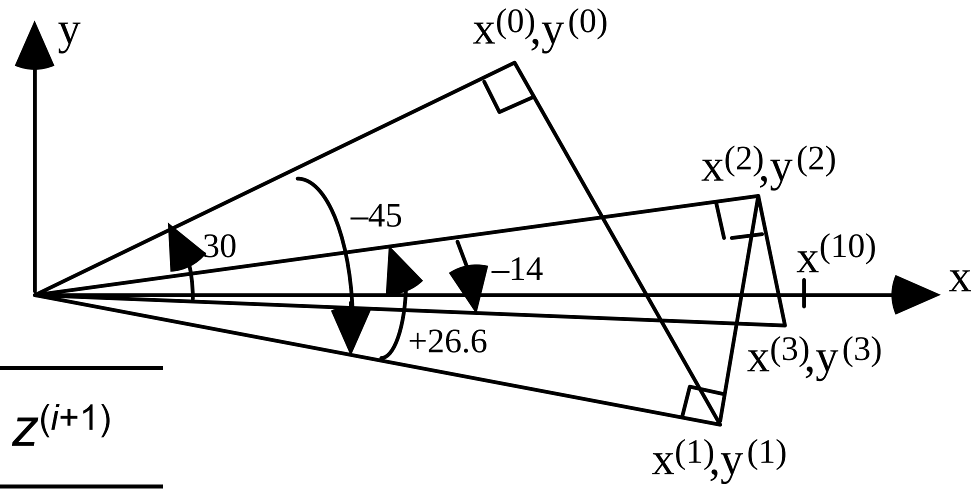
Table 22.1 Value of the function $e^{(i)} = \tan^{-1} 2^{-i}$, in degrees and radians, for $0 \le i \le 9$

Example: 30° angle

$$\begin{aligned} 30.0 \;\cong\; & 45.0 - 26.6 + 14.0 \\ & - 7.1 + 3.6 + 1.8 \\ & - 0.9 + 0.4 - 0.2 \\ & + 0.1 \\ =\; & 30.1 \end{aligned}$$

Choosing the Angles to Force z to Zero

$$x^{(i+1)} = x^{(i)} - d_i\, y^{(i)}\, 2^{-i}$$
$$y^{(i+1)} = y^{(i)} + d_i\, x^{(i)}\, 2^{-i}$$
$$z^{(i+1)} = z^{(i)} - d_i \tan^{-1} 2^{-i}$$
$$= z^{(i)} - d_i\, e^{(i)}$$



i	$z^{(i)}$	–	$d_i\, e^{(i)}$	=	$z^{(i+1)}$
					+30.0
0	+30.0	–	45.0	=	–15.0
1	–15.0	+	26.6	=	+11.6
2	+11.6	–	14.0	=	–2.4
3	–2.4	+	7.1	=	+4.7
4	+4.7	–	3.6	=	+1.1
5	+1.1	–	1.8	=	–0.7
6	–0.7	+	0.9	=	+0.2
7	+0.2	–	0.4	=	–0.2
8	–0.2	+	0.2	=	+0.0
9	+0.0	–	0.1	=	–0.1

Fig. 22.2 The first three of 10 pseudorotations leading from $(x^{(0)}, y^{(0)})$ to $(x^{(10)}, 0)$ in rotating by +30°.

Table 22.2 Choosing the signs of the rotation angles in order to force z to 0

Why Any Angle Can Be Formed from Our List

Analogy: Paying a certain amount while using all currency denominations (in positive or negative direction) exactly once; red values are fictitious.

$20 $10 $5 $3 $2 $1 $.50 $.25 $.20 $.10 $.05 $.03 $.02 $.01

Example: Pay $12.50

$20 – $10 + $5 – $3 + $2 – $1 – $.50 + $.25 – $.20 – $.10 + $.05 + $.03 – $.02 – $.01

Convergence is possible as long as each denomination is no greater than the sum of all denominations that follow it.

Domain of convergence: –$42.16 to +$42.16

We can guarantee convergence with actual denominations if we allow multiple steps at some values:

$20 $10 $5 $2 $2 $1 $.50 $.25 $.10 $.10 $.05 $.01 $.01 $.01 $.01

Example: Pay $12.50

$20 – $10 + $5 – $2 – $2 + $1 + $.50+$.25–$.10–$.10–$.05+$.01–$.01+ $.01–$.01

We will see later that in hyperbolic CORDIC, convergence is guaranteed only if certain “angles” are used twice.

Angle Recoding

The selection of angles during pseudorotations can be viewed as recoding the angle in a specific number system

For example, an angle of 30° is recoded as the following digit string, with each digit being 1 or –1:

45.0	26.6	14.0	7.1	3.6	1.8	0.9	0.4	0.2	0.1
1	–1	1	–1	1	1	–1	1	–1	1

The money-exchange analogy also lends itself to this recoding view

For example, a payment of $12.50 is recoded as:

$20	**$10**	**$5**	**$3**	**$2**	**$1**	**$.50**	**$.25**	**$.20**	**$.10**	**$.05**	**$.03**	**$.02**	**$.01**
1	**–1**	**1**	**–1**	**1**	**–1**	**–1**	**1**	**–1**	**–1**	**1**	**1**	**–1**	**–1**

Using CORDIC in Rotation Mode

$$x^{(i+1)} = x^{(i)} - d_i\, y^{(i)}\, 2^{-i}$$
$$y^{(i+1)} = y^{(i)} + d_i\, x^{(i)}\, 2^{-i}$$
$$\left.\begin{aligned} z^{(i+1)} &= z^{(i)} - d_i \tan^{-1} 2^{-i} \\ &= z^{(i)} - d_i\, e^{(i)} \end{aligned}\right\}$$

Make z converge to 0 by choosing $d_i = \text{sign}(z^{(i)})$

$$x^{(m)} = \cancel{K}(\cancel{x} \cos z - \overset{0}{\cancel{y \sin z}})$$
$$y^{(m)} = \cancel{K}(\underset{0}{\cancel{y \cos z}} + \cancel{x} \sin z)$$
$$z^{(m)} = 0$$

where $K = 1.646\ 760\ 258\ 121\ \ldots$

For k bits of precision in results, k CORDIC iterations are needed, because $\tan^{-1} 2^{-i} \cong 2^{-i}$ for large i

Start with
$x = 1/K = 0.607\ 252\ 935\ \ldots$
and $y = 0$
to find cos z and sin z

Convergence of z to 0 is possible because each of the angles in our list is more than half the previous one or, equivalently, each is less than the sum of all the angles that follow it

Domain of convergence is $-99.7^\circ \le z \le 99.7^\circ$, where 99.7° is the sum of all the angles in our list; the domain contains $[-\pi/2, \pi/2]$ radians

Using CORDIC in Vectoring Mode

$$x^{(i+1)} = x^{(i)} - d_i\, y^{(i)}\, 2^{-i}$$
$$y^{(i+1)} = y^{(i)} + d_i\, x^{(i)}\, 2^{-i}$$
$$z^{(i+1)} = z^{(i)} - d_i \tan^{-1} 2^{-i}$$
$$= z^{(i)} - d_i\, e^{(i)}$$

Make y converge to 0 by choosing $d_i = -\mathrm{sign}(x^{(i)} y^{(i)})$

$$x^{(m)} = K(x^2 + y^2)^{1/2}$$
$$y^{(m)} = 0$$
$$z^{(m)} = \underset{0}{\cancel{z}} + \tan^{-1}(y/\cancel{x})$$

where $K = 1.646\ 760\ 258\ 121\ \ldots$

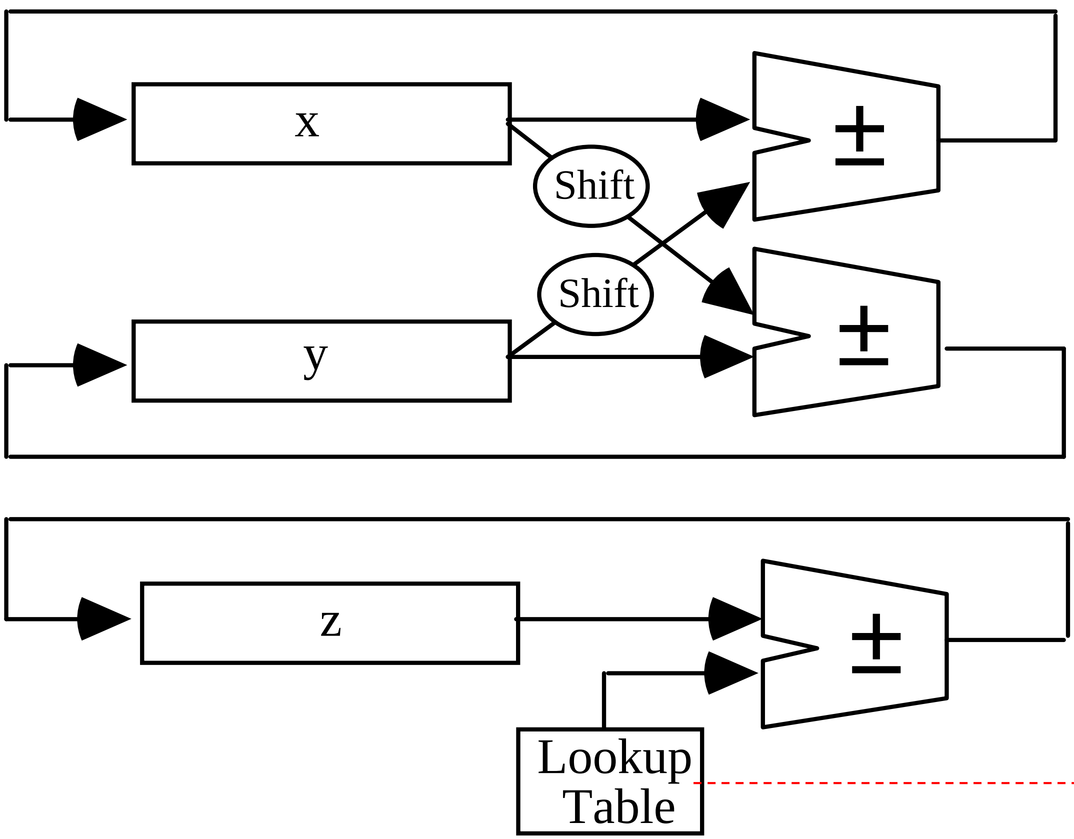
For k bits of precision in results, k CORDIC iterations are needed, because $\tan^{-1} 2^{-i} \cong 2^{-i}$ for large i

Start with
$x = 1$ and $z = 0$
to find $\tan^{-1} y$

Even though the computation above always converges, one can use the relationship $\tan^{-1}(1/y) = \pi/2 - \tan^{-1} y$ to limit the range of fixed-point numbers encountered

Other trig functions: $\tan z$ obtained from $\sin z$ and $\cos z$ via division; inverse sine and cosine ($\sin^{-1} z$ and $\cos^{-1} z$) discussed later

22.3 CORDIC Hardware



$$x^{(i+1)} = x^{(i)} - d_i\, y^{(i)}\, 2^{-i}$$
$$y^{(i+1)} = y^{(i)} + d_i\, x^{(i)}\, 2^{-i}$$
$$z^{(i+1)} = z^{(i)} - d_i \tan^{-1} 2^{-i} = z^{(i)} - d_i\, e^{(i)}$$

If very high speed is not needed (as in a calculator), a single adder and one shifter would suffice

k table entries for *k* bits of precision

Fig. 22.3 Hardware elements needed for the CORDIC method.

22.4 Generalized CORDIC

$$x^{(i+1)} = x^{(i)} - \mu\, d_i\, y^{(i)}\, 2^{-i}$$

$$y^{(i+1)} = y^{(i)} + d_i\, x^{(i)}\, 2^{-i}$$

$$z^{(i+1)} = z^{(i)} - d_i\, e^{(i)}$$

$\mu = 1$ Circular rotations (basic CORDIC) $e^{(i)} = \tan^{-1} 2^{-i}$

$\mu = 0$ Linear rotations $e^{(i)} = 2^{-i}$

$\mu = -1$ Hyperbolic rotations $e^{(i)} = \tanh^{-1} 2^{-i}$

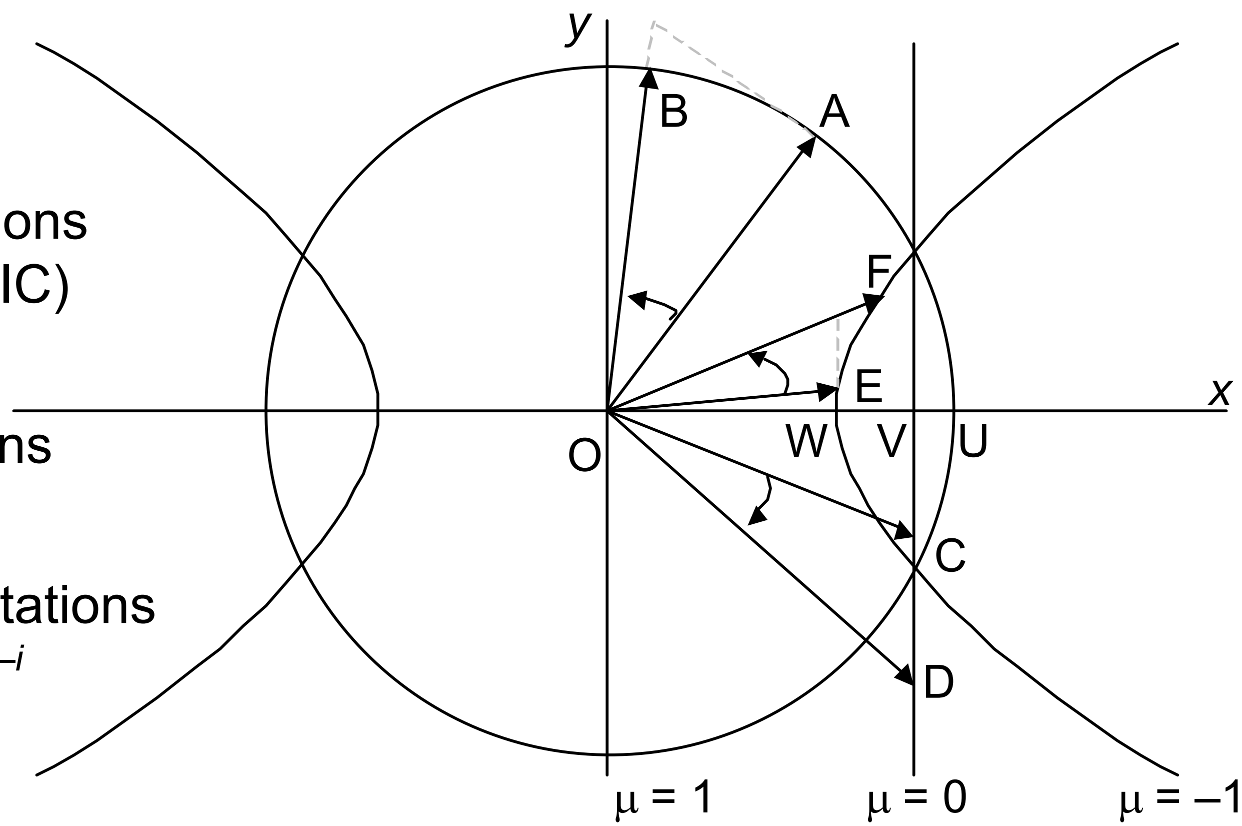


Fig. 22.4 Circular, linear, and hyperbolic CORDIC.

22.5 Using the CORDIC Method

$x^{(i+1)} = x^{(i)} - \mu d_i y^{(i)} 2^{-i}$

$y^{(i+1)} = y^{(i)} + d_i x^{(i)} 2^{-i}$

$z^{(i+1)} = z^{(i)} - d_i e^{(i)}$

$\mu \in \{-1, 0, 1\}$

$d_i \in \{-1, 1\}$

$K = 1.646\,760\,258\,121\ldots$

$1/K = .607\,252\,935\,009\ldots$

$K' = .828\,159\,360\,960\,2\ldots$

$1/K' = 1.207\,497\,067\,763\ldots$

Mode →	Rotation: $d_i = \mathrm{sign}(z^{(i)})$, $z^{(i)} \to 0$	Vectoring: $d_i = -\mathrm{sign}(x^{(i)} y^{(i)})$, $y^{(i)} \to 0$
$\mu = 1$ Circular $e^{(i)} = \tan^{-1} 2^{-i}$	$x \to$ CORDIC $\to K(x \cos z - y \sin z)$ $y \to$ CORDIC $\to K(y \cos z + x \sin z)$ $z \to$ CORDIC $\to 0$ For cos & sin, set $x = 1/K$, $y = 0$ $\tan z = \sin z / \cos z$	$x \to$ CORDIC $\to K\sqrt{x^2 + y^2}$ $y \to$ CORDIC $\to 0$ $z \to$ CORDIC $\to z + \tan^{-1}(y/x)$ For $\tan^{-1}$, set $x = 1$, $z = 0$ $\cos^{-1} w = \tan^{-1}[\sqrt{1 - w^2} / w]$ $\sin^{-1} w = \tan^{-1}[w / \sqrt{1 - w^2}]$
$\mu = 0$ Linear $e^{(i)} = 2^{-i}$	$x \to$ CORDIC $\to x$ $y \to$ CORDIC $\to y + xz$ $z \to$ CORDIC $\to 0$ For multiplication, set $y = 0$	$x \to$ CORDIC $\to x$ $y \to$ CORDIC $\to 0$ $z \to$ CORDIC $\to z + y/x$ For division, set $z = 0$
$\mu = -1$ Hyperbolic $e^{(i)} = \tanh^{-1} 2^{-i}$	$x \to$ CORDIC $\to K'(x \cosh z - y \sinh z)$ $y \to$ CORDIC $\to K'(y \cosh z + x \sinh z)$ $z \to$ CORDIC $\to 0$ For cosh & sinh, set $x = 1/K'$, $y = 0$ $\tanh z = \sinh z / \cosh z$ $\exp(z) = \sinh z + \cosh z$ $w^t = \exp(t \ln w)$	$x \to$ CORDIC $\to K'\sqrt{x^2 - y^2}$ $y \to$ CORDIC $\to 0$ $z \to$ CORDIC $\to z + \tanh^{-1}(y/x)$ For $\tanh^{-1}$, set $x = 1$, $z = 0$ $\ln w = 2 \tanh^{-1} \lvert (w-1)/(w+1) \rvert$ $\sqrt{w} = \sqrt{(w + 1/4)^2 - (w - 1/4)^2}$ $\cosh^{-1} w = \ln(w + \sqrt{1 - w^2})$ $\sinh^{-1} w = \ln(w + \sqrt{1 + w^2})$
Note →	In executing the iterations for $\mu = -1$, steps 4, 13, 40, 121, . . . , j , $3j + 1$, . . . must be repeated. These repetitions are incorporated in the constant K' below.	

Fig. 22.5 Summary of generalized CORDIC algorithms.

CORDIC Speedup Methods

$$x^{(i+1)} = x^{(i)} - \mu d_i y^{(i)} 2^{-i}$$
$$y^{(i+1)} = y^{(i)} + d_i x^{(i)} 2^{-i}$$
$$z^{(i+1)} = z^{(i)} - d_i e^{(i)}$$

Skipping some rotations

Must keep track of expansion via the recurrence:

$$(K^{(i+1)})^2 = (K^{(i)})^2 (1 \pm 2^{-2i})$$

This additional work makes *variable-factor* CORDIC less cost-effective than *constant-factor* CORDIC

$$x^{(k)} = x^{(k/2)} - y^{(k/2)} z^{(k/2)}$$
$$y^{(k)} = y^{(i)} + x^{(k/2)} z^{(k/2)}$$
$$z^{(k)} = z^{(k/2)} - z^{(k/2)}$$

Early termination

Do the first $k/2$ iterations as usual, then combine the remaining $k/2$ into a single multiplicative step:

For very small z, we have $\tan^{-1} z \cong z \cong \tan z$

Expansion factor not an issue because contribution of the ignored terms is provably less than *ulp*

$d_i \in \{-2, -1, 1, 2\}$ or $\{-2, -1, 0, 1, 2\}$

High-radix CORDIC

The hardware for the radix-4 version of CORDIC is quite similar to Fig. 22.3

22.6 An Algebraic Formulation

Because

$$\cos z + j \sin z = e^{jz} \qquad \text{where} \qquad j = \sqrt{-1}$$

$\cos z$ and $\sin z$ can be computed via evaluating the complex exponential function e^{jz}

This leads to an alternate derivation of CORDIC iterations

Details in the text

23 Variations in Function Evaluation

Chapter Goals

Learning alternate computation methods (convergence and otherwise) for some functions computable through CORDIC

Chapter Highlights

Reasons for needing alternate methods:
Achieve higher performance or precision
Allow speed/cost tradeoffs
Optimizations, fit to diverse technologies

Variations in Function Evaluation: Topics

Topics in This Chapter
23.1 Normalization and Range Reduction
23.2 Computing Logarithms
23.3 Exponentiation
23.4 Division and Square-Rooting, Again
23.5 Use of Approximating Functions
23.6 Merged Arithmetic

23.1 Normalization and Range Reduction

$$u^{(i+1)} = f(u^{(i)}, v^{(i)}) \longrightarrow \text{Constant} \longleftarrow u^{(i+1)} = f(u^{(i)}, v^{(i)}, w^{(i)})$$

$$v^{(i+1)} = g(u^{(i)}, v^{(i)}) \longrightarrow \text{Desired function} \longleftarrow v^{(i+1)} = g(u^{(i)}, v^{(i)}, w^{(i)})$$

$$w^{(i+1)} = h(u^{(i)}, v^{(i)}, w^{(i)})$$

Guide the iteration such that one of the values converges to a constant (usually 0 or 1); this is known as *normalization*

The other value then converges to the desired function

Additive normalization: Normalize u via addition of terms to it

Multiplicative normalization: Normalize u via multiplication of terms

Additive normalization is more desirable, unless the multiplicative terms are of the form 1 ± 2^a (shift-add) or multiplication leads to much faster convergence compared with addition

Convergence Methods You Already Know

$$x^{(i+1)} = x^{(i)} - \mu\, d_i\, y^{(i)}\, 2^{-i}$$
$$y^{(i+1)} = y^{(i)} + d_i\, x^{(i)}\, 2^{-i}$$
$$z^{(i+1)} = z^{(i)} - d_i\, e^{(i)}$$

CORDIC
Example of additive normalization

Force y or z to 0 by adding terms to it

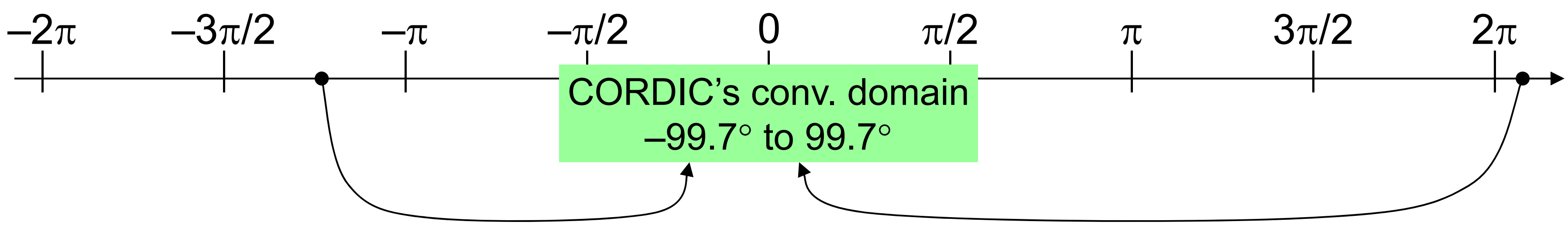
Force d to 1 by multiplying terms with it

$d^{(i+1)} = d^{(i)}(2 - d^{(i)})$ Set $d^{(0)} = d$; iterate until $d^{(m)} \cong 1$

$z^{(i+1)} = z^{(i)}(2 - d^{(i)})$ Set $z^{(0)} = z$; obtain $z/d = q \cong z^{(m)}$

Division by repeated multiplications
Example of multiplicative normalization

Range Reduction



$\cos(z - \pi) = -\cos z$

Adding π to the argument flips the function sign

$\cos(2j\pi + z) = \cos z$

Subtracting multiples of 2π from the argument does not change the function value

Must be careful: A slight error in the value of π is amplified when a large multiple of 2π is added to, or subtracted from, the argument

Example: Compute $\cos(1.125 \times 2^{47})$

Additive range reduction: see the CORDIC example above

Multiplicative range reduction: applicable to the log function, e.g.

23.2 Computing Logarithms

$d_i \in \{-1, 0, 1\}$

$x^{(i+1)} = x^{(i)} c^{(i)} = x^{(i)} (1 + d_i 2^{-i})$ Force $x^{(m)}$ to 1

$y^{(i+1)} = y^{(i)} - \ln c^{(i)} = y^{(i)} - \ln(1 + d_i 2^{-i})$ $y^{(m)}$ converges to $\cancel{y}$ + ln x

0

Read out from table

Why does this multiplicative normalization method work?

$x^{(m)} = x \prod c^{(i)} \cong 1 \qquad \Rightarrow \qquad \prod c^{(i)} \cong 1/x$

$y^{(m)} = y - \sum \ln c^{(i)} = y - \ln(\prod c^{(i)}) = y - \ln(1/x) \cong y + \ln x$

Convergence domain: $1/\prod(1+2^{-i}) \le x \le 1/\prod(1-2^{-i})$ or $0.21 \le x \le 3.45$

Number of iterations: k, for k bits of precision; for large i, $\ln(1 \pm 2^{-i}) \cong \pm 2^{-i}$

Use directly for $x \in [1, 2)$. For $x = 2^q s$, we have:
$\ln x = q \ln 2 + \ln s = 0.693\,147\,180\, q + \ln s$

Radix-4 version can be devised

Computing Binary Logarithms via Squaring

For $x \in [1, 2)$, $\log_2 x$ is a fractional number $y = (.y_{-1}y_{-2}y_{-3} \ldots y_{-l})_{two}$

$$x = 2^y = 2^{(.y_{-1}y_{-2}y_{-3} \ldots y_{-l})_{two}}$$

$$x^2 = 2^{2y} = 2^{(y_{-1}.y_{-2}y_{-3} \ldots y_{-l})_{two}} \quad \Rightarrow \quad y_{-1} = 1 \text{ iff } x^2 \geq 2$$

Once y_{-1} has been determined, if $y_{-1} = 0$, we are back at the original situation; otherwise, divide both sides of the equation above by 2 to get:

$$x^2/2 = 2^{(1.y_{-2}y_{-3} \ldots y_{-l})_{two}}/2 = 2^{(.y_{-2}y_{-3} \ldots y_{-l})_{two}}$$

Generalization to base b:

$$x = b^{(.y_{-1}y_{-2}y_{-3} \ldots y_{-l})_{two}}$$

$$y_{-1} = 1 \text{ iff } x^2 \geq b$$

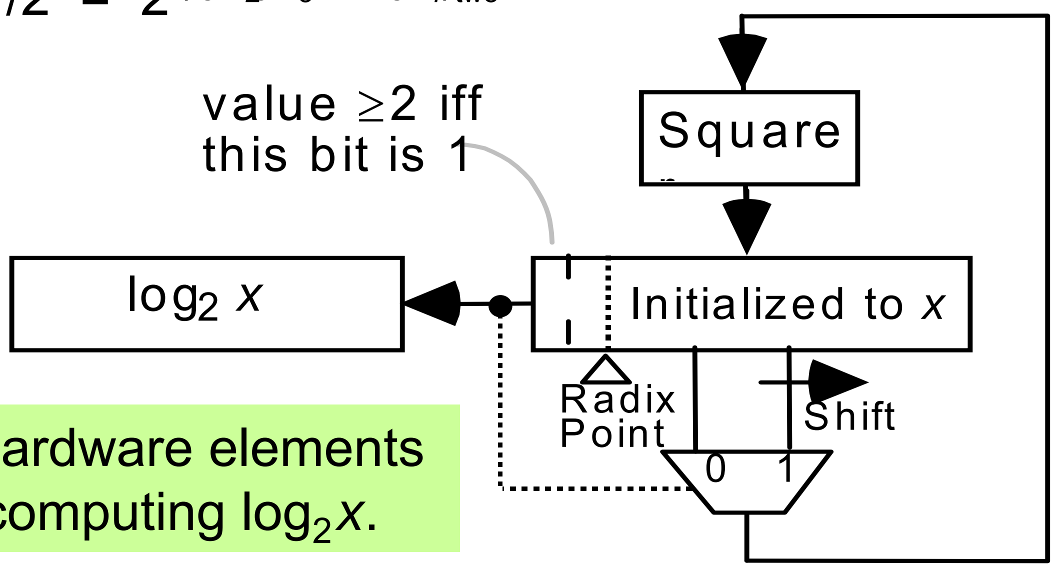


Fig. 23.1 Hardware elements needed for computing $\log_2 x$.

23.3 Exponentiation

Computing e^x

Read out from table

$x^{(i+1)} = x^{(i)} - \ln c^{(i)} = x^{(i)} - \ln(1 + d_i 2^{-i})$

$y^{(i+1)} = y^{(i)} c^{(i)} = y^{(i)}(1 + d_i 2^{-i})$

$d_i \in \{-1, 0, 1\}$

Force $x^{(m)}$ to 0

$y^{(m)}$ converges to $\cancel{y}$ e^x

1

Why does this additive normalization method work?

$x^{(m)} = x - \sum \ln c^{(i)} \cong 0 \qquad \Rightarrow \qquad \sum \ln c^{(i)} \cong x$

$y^{(m)} = y \prod c^{(i)} = y \exp(\ln \prod c^{(i)}) = y \exp(\sum \ln c^{(i)}) \cong y e^x$

Convergence domain: $\sum \ln(1 - 2^{-i}) \le x \le \sum \ln(1 + 2^{-i})$ or $-1.24 \le x \le 1.56$

Number of iterations: k, for k bits of precision; for large i, $\ln(1 \pm 2^{-i}) \cong \pm 2^{-i}$

Can eliminate half the iterations because
$\ln(1 + \varepsilon) = \varepsilon - \varepsilon^2/2 + \varepsilon^3/3 - \ldots \cong \varepsilon$ for $\varepsilon^2 < ulp$
and we may write $y^{(k)} = y^{(k/2)}(1 + x^{(k/2)})$

Radix-4 version can be devised

General Exponentiation, or Computing x^y

$x^y = (e^{\ln x})^y = e^{y \ln x}$ So, compute natural log, multiply, exponentiate

Method is prone to inaccuracies

When y is an integer, we can exponentiate by repeated multiplication (need to consider only positive y; for negative y, compute reciprocal)

In particular, when y is a constant, the methods used are reminiscent of multiplication by constants (Section 9.5)

Example: $x^{25} = ((((x)^2x)^2)^2)^2x$ [4 squarings and 2 multiplications]

Noting that $25 = (1\ 1\ 0\ 0\ 1)_{\text{two}}$, leads to a general procedure

Computing x^y, when y is an unsigned integer

Initialize the partial result to 1
Scan the binary representation of y, starting at its MSB, and repeat
If the current bit is 1, multiply the partial result by x
If the current bit is 0, do not change the partial result
Square the partial result before the next step (if any)

Faster Exponentiation via Recoding

Example: $x^{31} = ((((x)^2x)^2x)^2x)^2x$ [4 squarings and 4 multiplications]

Note that $31 = (1\ 1\ 1\ 1\ 1)_{\text{two}} = (1\ 0\ 0\ 0\ 0\ {}^{-}1)_{\text{two}}$

$x^{31} = (((((x)^2)^2)^2)^2)^2 / x$ [5 squarings and 1 division]

Computing x^y, when y is an integer encoded in BSD format

Initialize the partial result to 1
Scan the binary representation of y, starting at its MSB, and repeat
If the current digit is 1, multiply the partial result by x
If the current digit is 0, do not change the partial result
If the current digit is ${}^{-}1$, divide the partial result by x
Square the partial result before the next step (if any)

Radix-4 example: $31 = (1\ 1\ 1\ 1\ 1)_{\text{two}} = (1\ 0\ 0\ 0\ 0\ {}^{-}1)_{\text{two}} = (2\ 0\ {}^{-}1)_{\text{four}}$

$x^{31} = (((x^2)^4)^4 / x$ [Can you formulate the general procedure?]

23.4 Division and Square-Rooting, Again

Computing $q = z / d$

$s^{(i+1)} = s^{(i)} - \gamma^{(i)} d$

$q^{(i+1)} = q^{(i)} + \gamma^{(i)}$

In digit-recurrence division, $\gamma^{(i)}$ is the next quotient digit and the addition for q turns into concatenation; more generally, $\gamma^{(i)}$ can be any estimate for the difference between the partial quotient $q^{(i)}$ and the final quotient q

Because $s^{(i)}$ becomes successively smaller as it converges to 0, scaled versions of the recurrences above are usually preferred. In the following, $s^{(i)}$ stands for $s^{(i)} r^i$ and $q^{(i)}$ for $q^{(i)} r^i$:

$s^{(i+1)} = rs^{(i)} - \gamma^{(i)} d$ Set $s^{(0)} = z$ and keep $s^{(i)}$ bounded

$q^{(i+1)} = rq^{(i)} + \gamma^{(i)}$ Set $q^{(0)} = 0$ and find $q^* = q^{(m)} r^{-m}$

In the scaled version, $\gamma^{(i)}$ is an estimate for $r(r^{i-m} q - q^{(i)}) = r(r^i q^* - q^{(i)})$, where $q^* = r^{-m} q$ represents the true quotient

Square-Rooting via Multiplicative Normalization

Idea: If z is multiplied by a sequence of values $(c^{(i)})^2$, chosen so that the product $z\prod(c^{(i)})^2$ converges to 1, then $z\prod c^{(i)}$ converges to $\sqrt{z}$

$x^{(i+1)} = x^{(i)}(1 + d_i 2^{-i})^2 = x^{(i)}(1 + 2d_i 2^{-i} + d_i^2 2^{-2i})$ $\qquad x^{(0)} = z$, $x^{(m)} \cong 1$

$y^{(i+1)} = y^{(i)}(1 + d_i 2^{-i})$ $\qquad y^{(0)} = z$, $y^{(m)} \cong \sqrt{z}$

What remains is to devise a scheme for choosing d_i values in {–1, 0, 1}

$d_i = 1$ for $x^{(i)} < 1 - \varepsilon = 1 - \alpha 2^{-i}$ $\qquad d_i = -1$ for $x^{(i)} > 1 + \varepsilon = 1 + \alpha 2^{-i}$

To avoid the need for comparison with a different constant in each step, a scaled version of the first recurrence is used in which $u^{(i)} = 2^i(x^{(i)} - 1)$:

$u^{(i+1)} = 2(u^{(i)} + 2d_i) + 2^{-i+1}(2d_i u^{(i)} + d_i^2) + 2^{-2i+1}d_i^2 u^{(i)}$ $\qquad u^{(0)} = z - 1$, $u^{(m)} \cong 0$

$y^{(i+1)} = y^{(i)}(1 + d_i 2^{-i})$ $\qquad y^{(0)} = z$, $y^{(m)} \cong \sqrt{z}$

Radix-4 version can be devised: Digit set [–2, 2] or {–1, –½, 0, ½, 1}

Square-Rooting via Additive Normalization

Idea: If a sequence of values $c^{(i)}$ can be obtained such that $z - (\sum c^{(i)})^2$ converges to 0, then $\sum c^{(i)}$ converges to $\sqrt{z}$

$x^{(i+1)} = z - (y^{(i+1)})^2 = z - (y^{(i)} + c^{(i)})^2 = x^{(i)} + 2d_i y^{(i)} 2^{-i} - d_i^2 2^{-2i}$ $\quad x^{(0)} = z,\ x^{(m)} \cong 0$

$y^{(i+1)} = y^{(i)} + c^{(i)} = y^{(i)} - d_i 2^{-i}$ $\quad y^{(0)} = 0,\ y^{(m)} \cong \sqrt{z}$

What remains is to devise a scheme for choosing d_i values in {–1, 0, 1}

$d_i = 1$ for $x^{(i)} < -\varepsilon = -\alpha 2^{-i}$ $\quad d_i = -1$ for $x^{(i)} > +\varepsilon = +\alpha 2^{-i}$

To avoid the need for comparison with a different constant in each step, a scaled version of the first recurrence may be used in which $u^{(i)} = 2^i x^{(i)}$:

$u^{(i+1)} = 2(u^{(i)} + 2d_i y^{(i)} - d_i^2 2^{-i})$ $\quad u^{(0)} = z$, $u^{(i)}$ bounded

$y^{(i+1)} = y^{(i)} - d_i 2^{-i}$ $\quad y^{(0)} = 0,\ y^{(m)} \cong \sqrt{z}$

Radix-4 version can be devised: Digit set [–2, 2] or {–1, –½, 0, ½, 1}

23.5 Use of Approximating Functions

Convert the problem of evaluating the function *f* to that of function *g* approximating *f*, perhaps with a few pre- and postprocessing operations

Approximating polynomials need only additions and multiplications

Polynomial approximations can be derived from various schemes

The Taylor-series expansion of $f(x)$ about $x = a$ is

$$f(x) = \sum_{j=0 \text{ to } \infty} f^{(j)}(a)\,(x - a)^j / j!$$

The error due to omitting terms of degree > *m* is:

$$f^{(m+1)}(a + \mu(x - a))\,(x - a)^{m+1} / (m + 1)! \qquad 0 <$$

Setting $a = 0$ yields the Maclaurin-series expansion

$$f(x) = \sum_{j=0 \text{ to } \infty} f^{(j)}(0)\, x^j / j!$$

and its corresponding error bound:

$$f^{(m+1)}(\mu x)\, x^{m+1} / (m + 1)! \qquad 0 < \mu < 1$$

Efficiency in computation can be gained via Horner's method and incremental evaluation

Some Polynomial Approximations (Table 23.1)

Func	Polynomial approximation	Conditions
$1/x$	$1 + y + y^2 + y^3 + \cdots + y^i + \cdots$	$0 < x < 2,\ y = 1 - x$
e^x	$1 + x/1! + x^2/2! + x^3/3! + \cdots + x^i/i! + \cdots$	
$\ln x$	$-y - y^2/2 - y^3/3 - y^4/4 - \cdots - y^i/i - \cdots$	$0 < x \le 2,\ y = 1 - x$
$\ln x$	$2\,[z + z^3/3 + z^5/5 + \cdots + z^{2i+1}/(2i+1) + \cdots]$	$x > 0,\ z = \frac{x-1}{x+1}$
$\sin x$	$x - x^3/3! + x^5/5! - x^7/7! + \cdots + (-1)^i x^{2i+1}/(2i+1)! + \cdots$	
$\cos x$	$1 - x^2/2! + x^4/4! - x^6/6! + \cdots + (-1)^i x^{2i}/(2i)! + \cdots$	
$\tan^{-1} x$	$x - x^3/3 + x^5/5 - x^7/7 + \cdots + (-1)^i x^{2i+1}/(2i+1) + \cdots$	$-1 < x < 1$
$\sinh x$	$x + x^3/3! + x^5/5! + x^7/7! + \cdots + x^{2i+1}/(2i+1)! + \cdots$	
$\cosh x$	$1 + x^2/2! + x^4/4! + x^6/6! + \cdots + x^{2i}/(2i)! + \cdots$	
$\tanh^{-1} x$	$x + x^3/3 + x^5/5 + x^7/7 + \cdots + x^{2i+1}/(2i+1) + \cdots$	$-1 < x < 1$

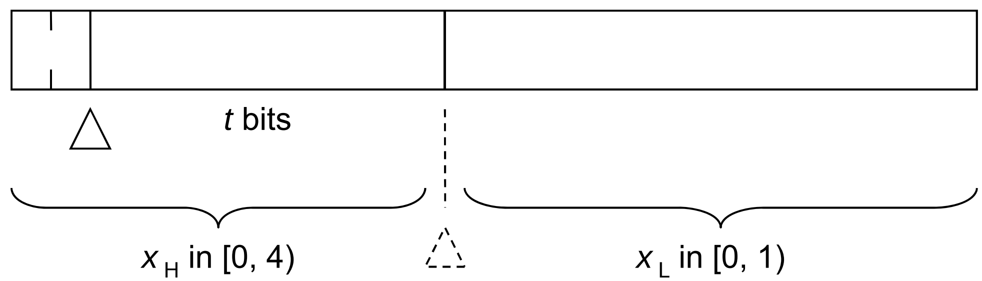
Function Evaluation via Divide-and-Conquer

Let x in [0, 4) be the $(l + 2)$-bit significand of a floating-point number or its shifted version. Divide x into two chunks x_H and x_L:

$x = x_H + 2^{-t} x_L$

$0 \le x_H < 4$ $t + 2$ bits

$0 \le x_L < 1$ $l - t$ bits



The Taylor-series expansion of $f(x)$ about $x = x_H$ is

$$f(x) = \sum_{j=0 \text{ to } \infty} f^{(j)}(x_H)\,(2^{-t} x_L)^j / j!$$

A linear approximation is obtained by taking only the first two terms

$$f(x) \cong f(x_H) + 2^{-t} x_L f'(x_H)$$

If t is not too large, f and/or f' (and other derivatives of f, if needed) can be evaluated via table lookup

Approximation by the Ratio of Two Polynomials

Example, yielding good results for many elementary functions

$$f(x) \cong \frac{a^{(5)}x^5 + a^{(4)}x^4 + a^{(3)}x^3 + a^{(2)}x^2 + a^{(1)}x + a^{(0)}}{b^{(5)}x^5 + b^{(4)}x^4 + b^{(3)}x^3 + b^{(2)}x^2 + b^{(1)}x + b^{(0)}}$$

Using Horner's method, such a "rational approximation" needs 10 multiplications, 10 additions, and 1 division

23.6 Merged Arithmetic

Our methods thus far rely on word-level building-block operations such as addition, multiplication, shifting, . . .

Sometimes, we can compute a function of interest directly without breaking it down into conventional operations

Example: merged arithmetic for inner product computation

$z = z^{(0)} + x^{(1)}y^{(1)} + x^{(2)}y^{(2)} + x^{(3)}y^{(3)}$

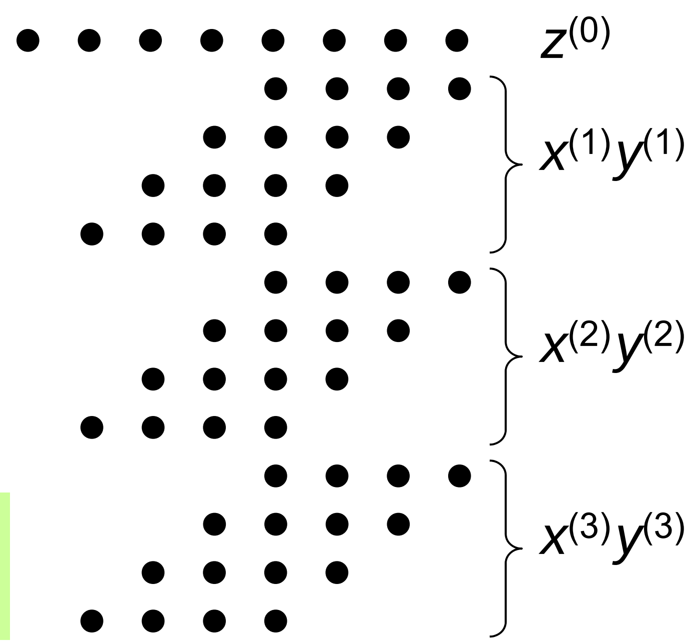


Fig. 23.2 Merged-arithmetic computation of an inner product followed by accumulation.

Example of Merged Arithmetic Implementation

Example: Inner product computation

$z = z^{(0)} + x^{(1)}y^{(1)} + x^{(2)}y^{(2)} + x^{(3)}y^{(3)}$

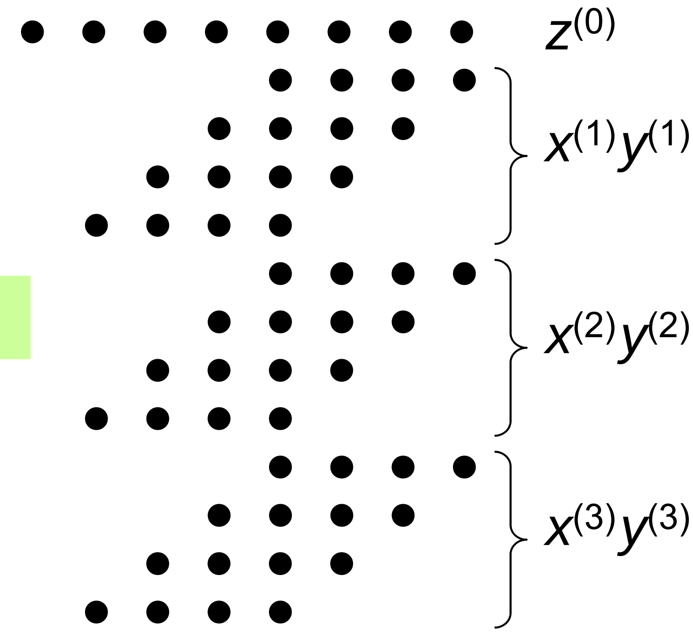


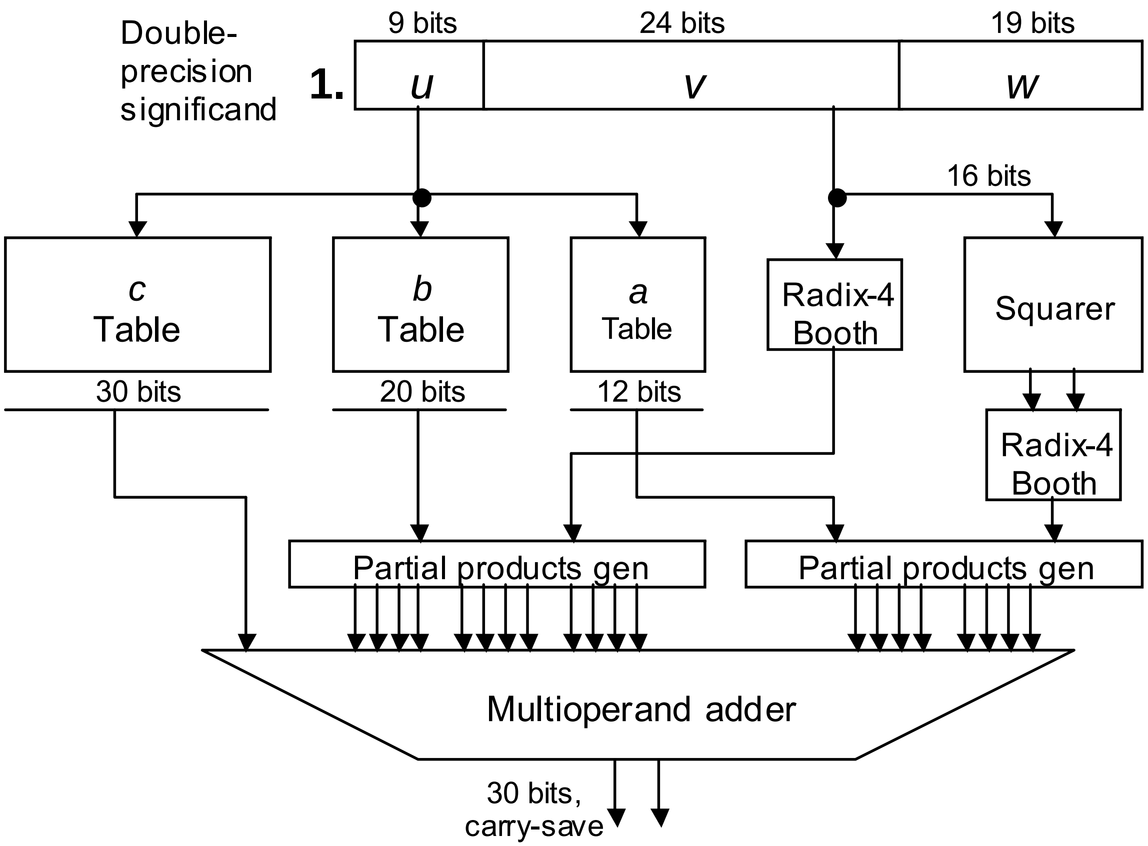
Fig. 23.2

Fig. 23.3 Tabular representation of the dot matrix for inner-product computation and its reduction.

	1	4	7	10	13	10	7	4	16 FAs
	2	4	6	8	8	6	4	2	10 FAs + 1 HA
	3	4	4	6	6	3	3	1	9 FAs
1	2	3	4	4	3	2	1	1	4 FAs + 1 HA
1	3	2	3	3	2	1	1	1	3 FAs + 2 HAs
2	2	2	2	2	1	1	1	1	5-bit CPA

Another Merged Arithmetic Example

Approximation of reciprocal (1/*x*) and reciprocal square root (1/√*x*) functions with 29-30 bits of precision, so that a long floating-point result can be obtained with just one iteration at the end [Pine02]



$f(x) = c + bv + av^2$

2 adds

2 mult's

1 square

Comparable to a multiplier

24 Arithmetic by Table Lookup

Chapter Goals

Learning table lookup techniques
for flexible and dense VLSI realization
of arithmetic functions

Chapter Highlights

We have used tables to simplify or speedup
q digit selection, convergence methods, . . .
Now come tables as primary computational
mechanisms (as stars, not supporting cast)

Arithmetic by Table Lookup: Topics

Topics in This Chapter
24.1 Direct and Indirect Table Lookup
24.2 Binary-to-Unary Reduction
24.3 Tables in Bit-Serial Arithmetic
24.4 Interpolating Memory
24.5 Piecewise Lookup Tables
24.6 Multipartite Table Methods

24.1 Direct and Indirect Table Lookup

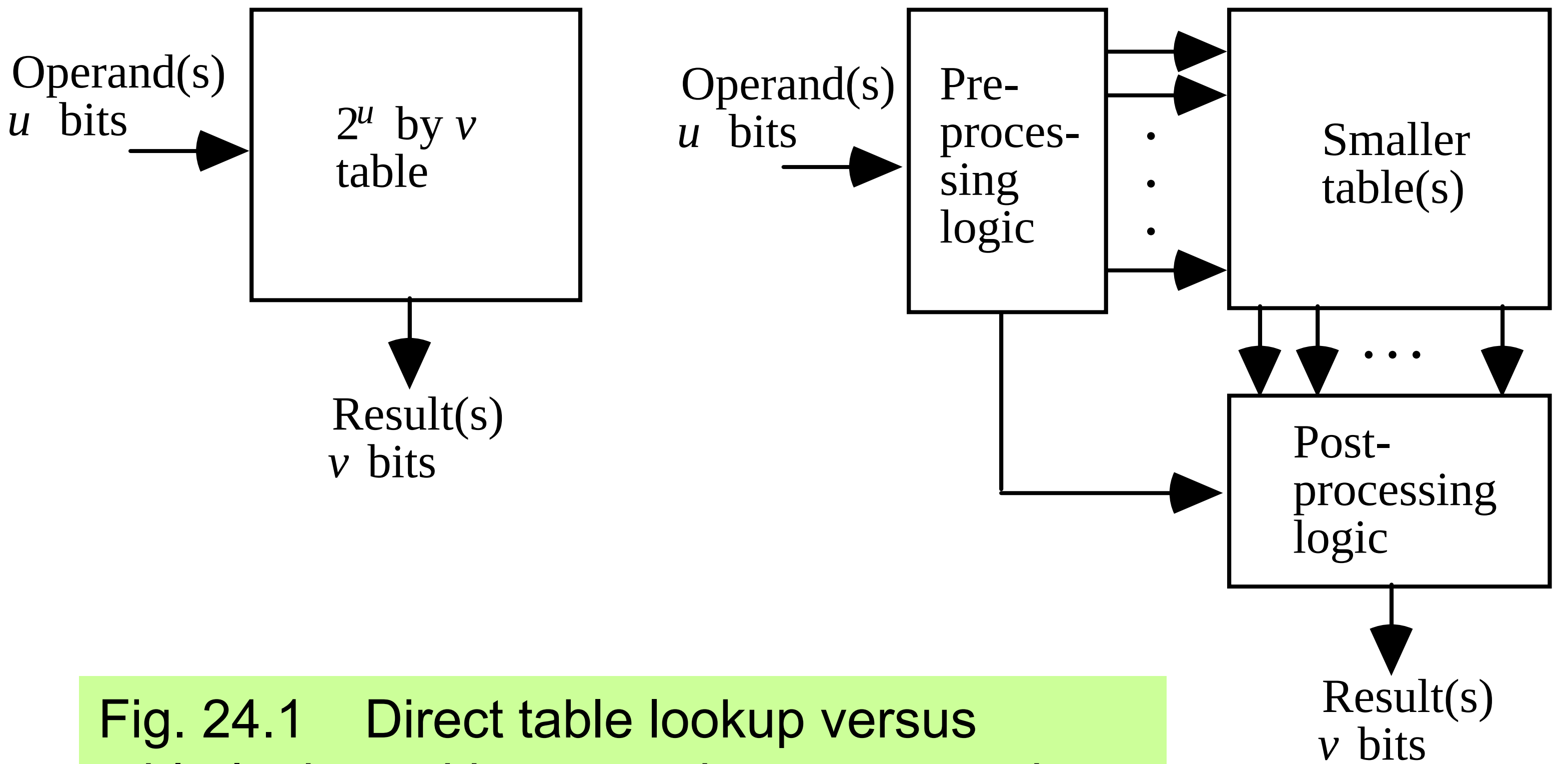


Fig. 24.1 Direct table lookup versus table-lookup with pre- and post-processing.

Tables in Supporting and Primary Roles

Tables are used in two ways:

In supporting role, as in initial estimate for division

As main computing mechanism

Boundary between two uses is fuzzy

Pure logic → Hybrid solutions ← Pure tabular

Previously, we started with the goal of designing logic circuits for particular arithmetic computations and ended up using tables to facilitate or speed up certain steps

Here, we aim for a tabular implementation and end up using peripheral logic circuits to reduce the table size

Some solutions can be derived starting at either endpoint

24.2 Binary-to-Unary Reduction

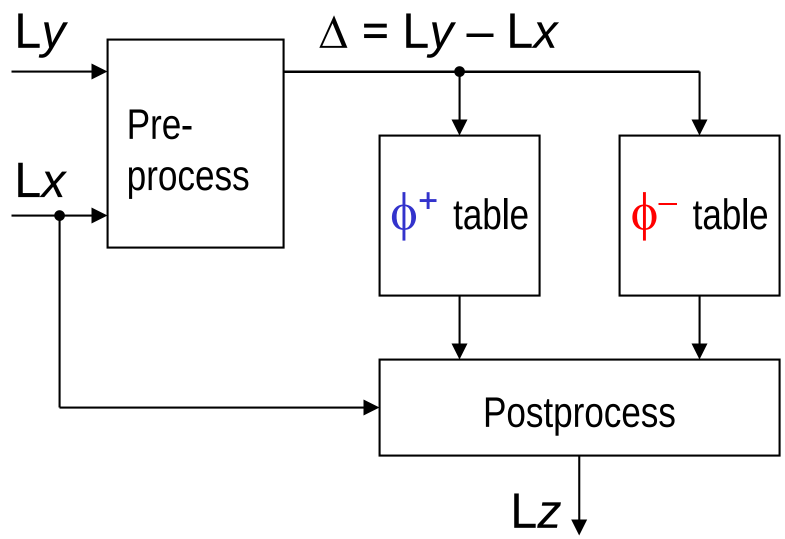
Strategy: Reduce the table size by using an auxiliary unary function to evaluate a desired binary function

Example 1: Addition/subtraction in a logarithmic number system; i.e., finding $Lz = \log(x \pm y)$, given Lx and Ly

Solution: Let $\Delta = Ly - Lx$

$$\begin{aligned} Lz &= \log(x \pm y) \\ &= \log(x(1 \pm y/x)) \\ &= \log x + \log(1 \pm y/x) \\ &= Lx + \log(1 \pm \log^{-1}\Delta) \end{aligned}$$

$Lx + \phi^{+}(\Delta)$ $\qquad$ $Lx + \phi^{-}(\Delta)$



Another Example of Binary-to-Unary Reduction

Example 2: Multiplication via squaring, $xy = (x + y)^2/4 - (x - y)^2/4$

Simplification and implementation details

If x and y are k bits wide, $x + y$ and $x - y$ are $k + 1$ bits wide, leading to two tables of size $2^{k+1} \times 2k$ (total table size = $2^{k+3} \times k$ bits)

Can be realized with one adder and one table

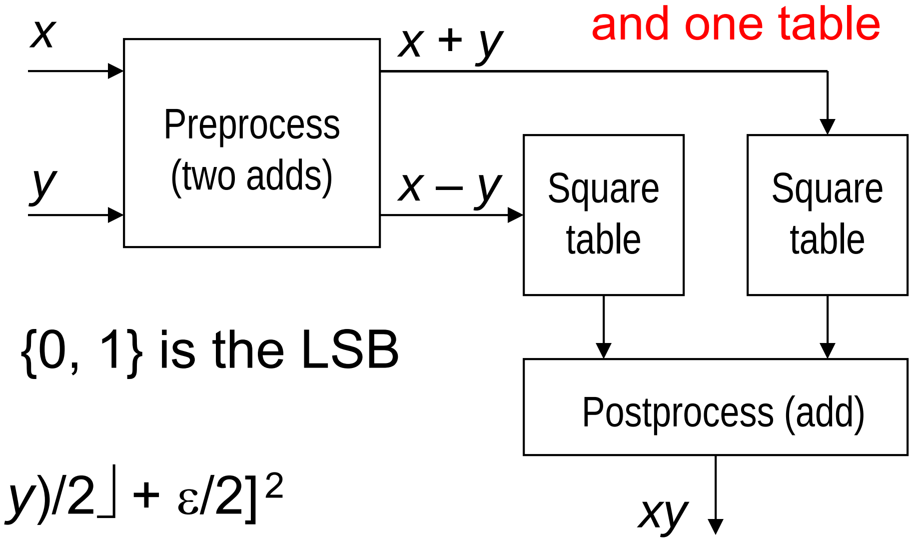


Fig. 24.2

$(x \pm y)/2 = \lfloor (x \pm y)/2 \rfloor + \varepsilon/2 \qquad \varepsilon \in \{0, 1\}$ is the LSB

$$\begin{aligned}(x + y)^2/4 - (x - y)^2/4 &= [\lfloor (x + y)/2 \rfloor + \varepsilon/2]^2 - [\lfloor (x - y)/2 \rfloor + \varepsilon/2]^2 \\ &= \lfloor (x + y)/2 \rfloor^2 - \lfloor (x - y)/2 \rfloor^2 + \varepsilon y\end{aligned}$$

Pre-process: compute $x + y$ and $x - y$; drop their LSBs
Table lookup: consult two squaring table(s) of size $2^k \times (2k - 1)$
Post-process: carry-save adder, followed by carry-propagate adder

(table size after simplification = $2^{k+1} \times (2k - 1) \cong 2^{k+2} \times k$ bits)

24.3 Tables in Bit-Serial Arithmetic

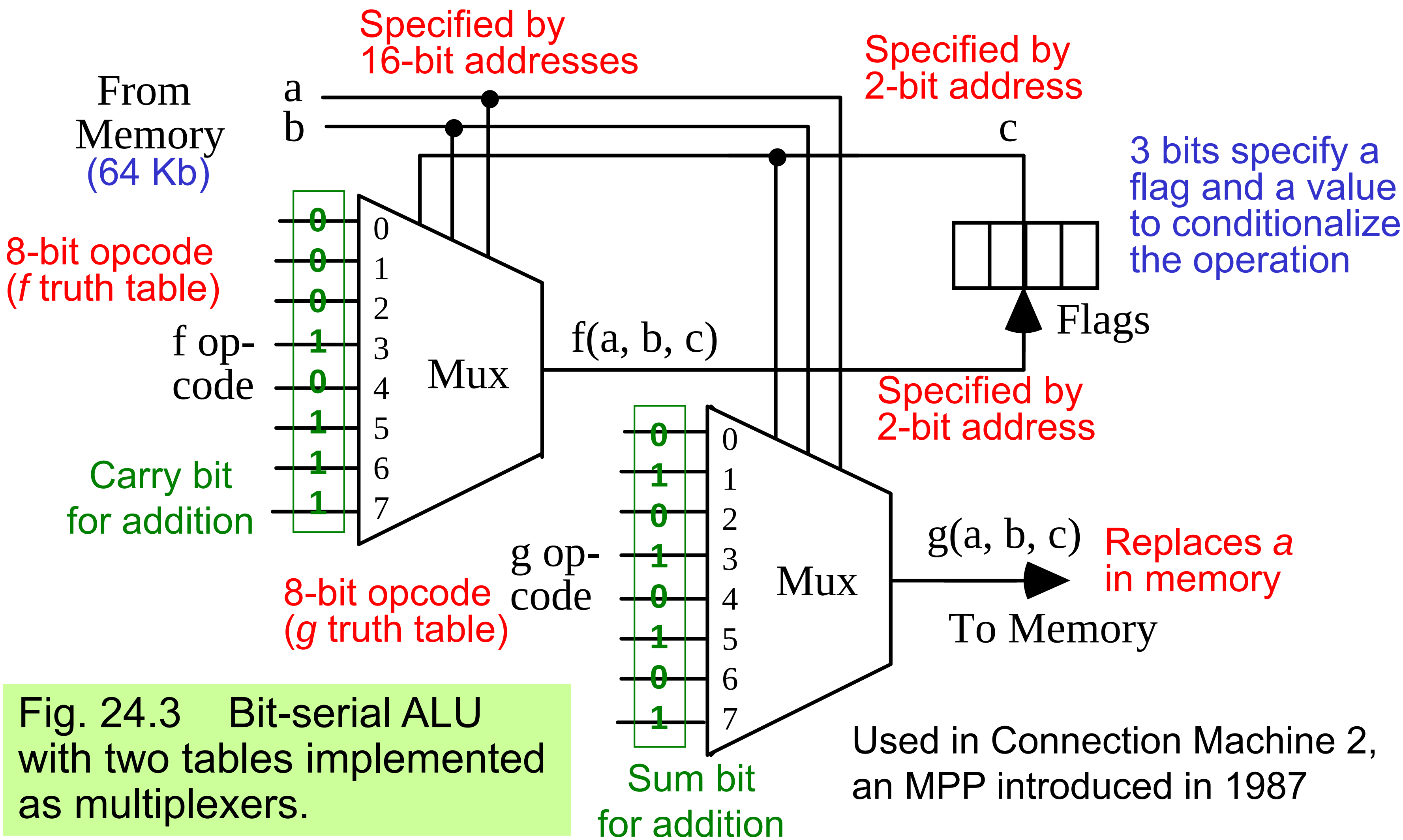
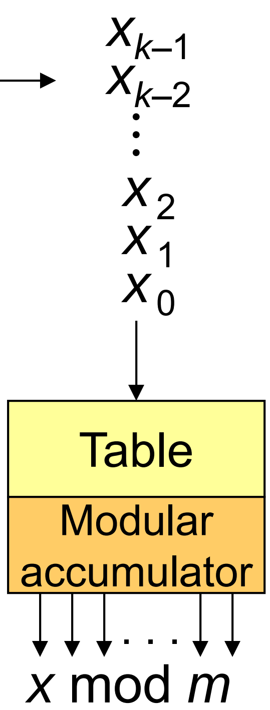


Fig. 24.3 Bit-serial ALU with two tables implemented as multiplexers.

Used in Connection Machine 2, an MPP introduced in 1987

Other Table-Based Bit-Serial Arithmetic Examples

See Section 4.3: Conversion from binary/decimal to RNS



Evaluation of linear expressions (assume unsigned values)

$$z = ax + by = a \sum x_i 2^i + b \sum y_i 2^i$$
$$= \sum (ax_i + by_i) 2^i$$

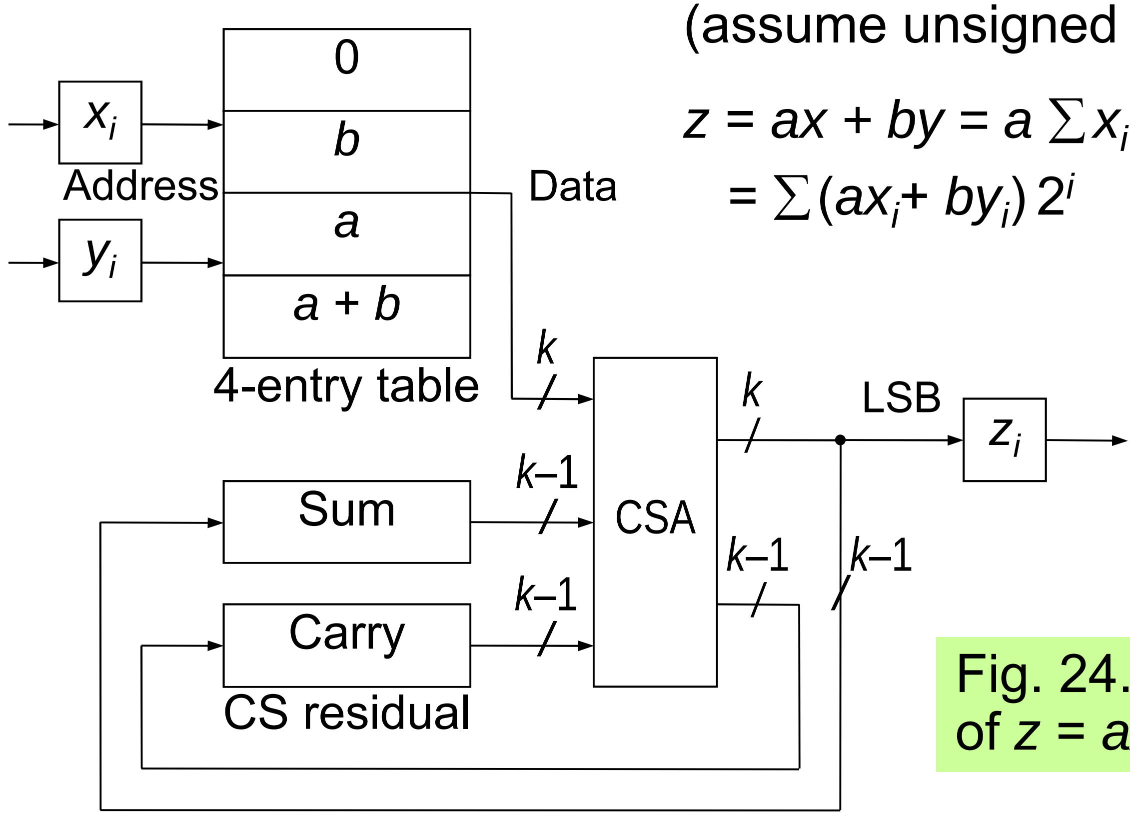


Fig. 24.4 Bit-serial evaluation of $z = ax + by$.

24.4 Interpolating Memory

Linear interpolation: Computing $f(x)$, $x \in [x_{lo}, x_{hi}]$, from $f(x_{lo})$ and $f(x_{hi})$

$$f(x) = f(x_{lo}) + \frac{x - x_{lo}}{x_{hi} - x_{lo}} [f(x_{hi}) - f(x_{lo})]$$

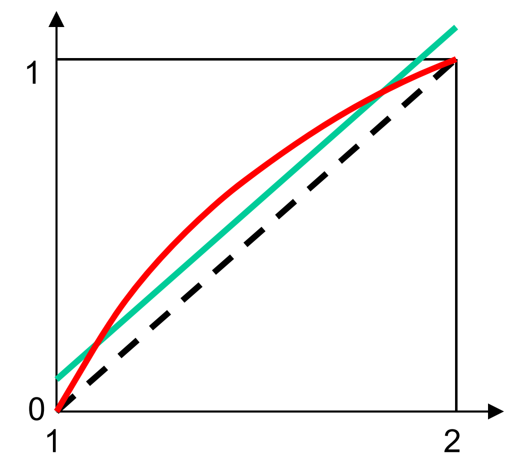
4 adds, 1 divide, 1 multiply

If the x_{lo} and x_{hi} endpoints are consecutive multiples of a power of 2, the division and two of the additions become trivial

Example: Evaluating $\log_2 x$ for $x \in [1, 2)$

$f(x_{lo}) = \log_2 1 = 0$, $f(x_{hi}) = \log_2 2 = 1$; thus:

$\log_2 x \cong x - 1 =$ Fractional part of x



An improved linear interpolation formula

$$\log_2 x \cong \frac{\ln 2 - \ln(\ln 2) - 1}{2 \ln 2} + (x - 1) = 0.043\,036 + \Delta x$$

Hardware Linear Interpolation Scheme

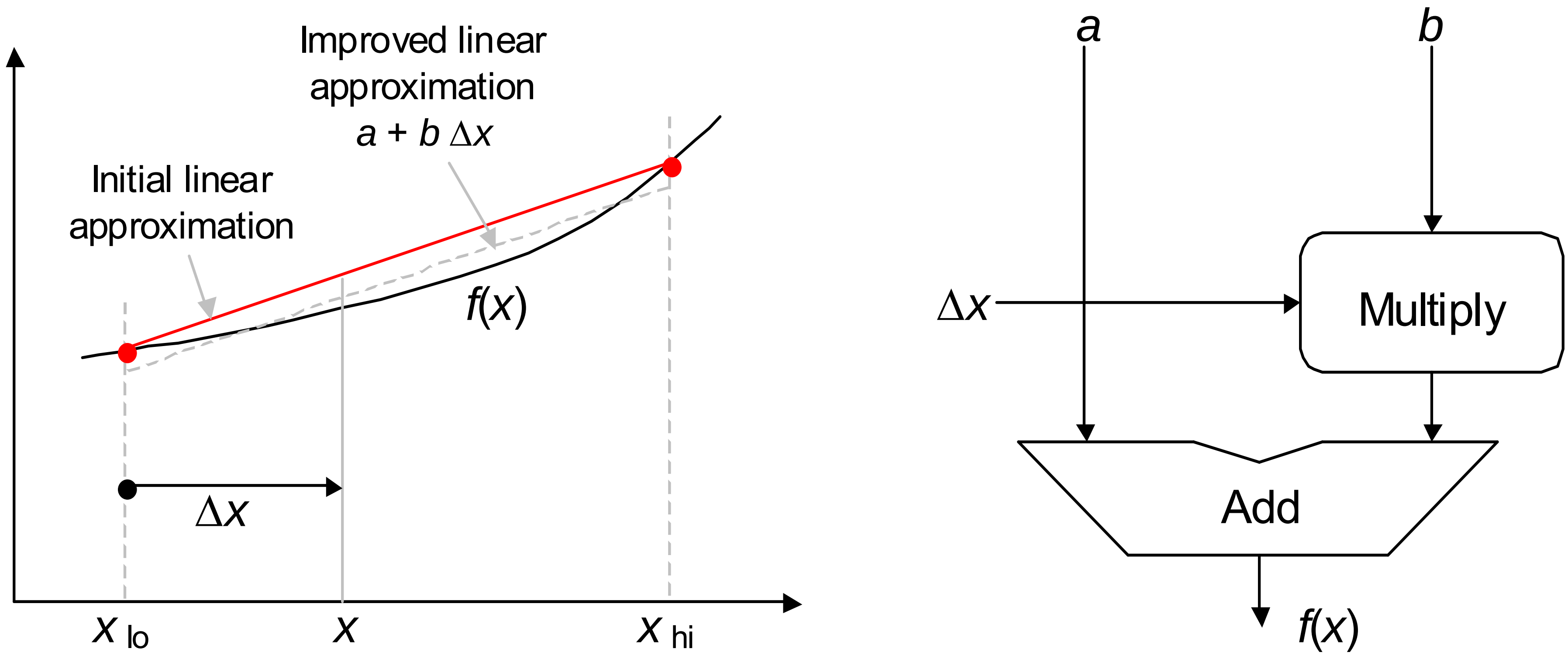


Fig. 24.5 Linear interpolation for computing $f(x)$ and its hardware realization.

Linear Interpolation with Four Subintervals

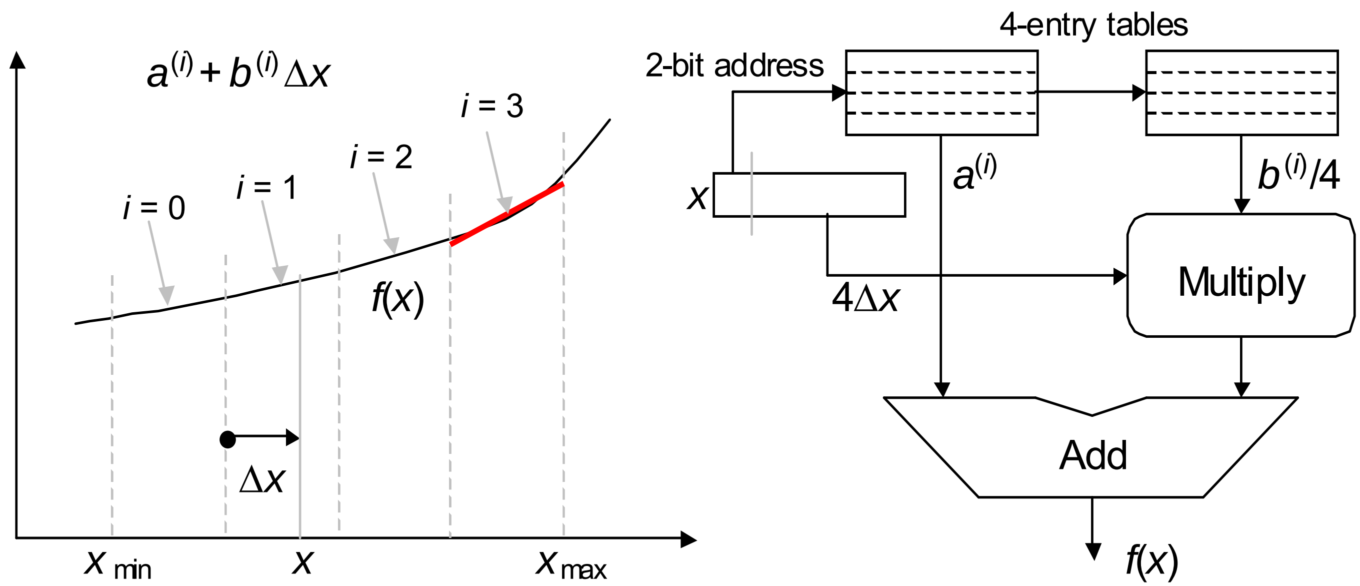


Fig. 24.6 Linear interpolation for computing $f(x)$ using 4 subintervals.

Table 24.1 Approximating $\log_2 x$ for x in $[1, 2)$ using linear interpolation within 4 subintervals.

i	x_{lo}	x_{hi}	$a^{(i)}$	$b^{(i)}/4$	Max error
0	1.00	1.25	0.004 487	0.321 928	± 0.004 487
1	1.25	1.50	0.324 924	0.263 034	± 0.002 996
2	1.50	1.75	0.587 105	0.222 392	± 0.002 142
3	1.75	2.00	0.808 962	0.192 645	± 0.001 607

Tradeoffs in Cost, Speed, and Accuracy

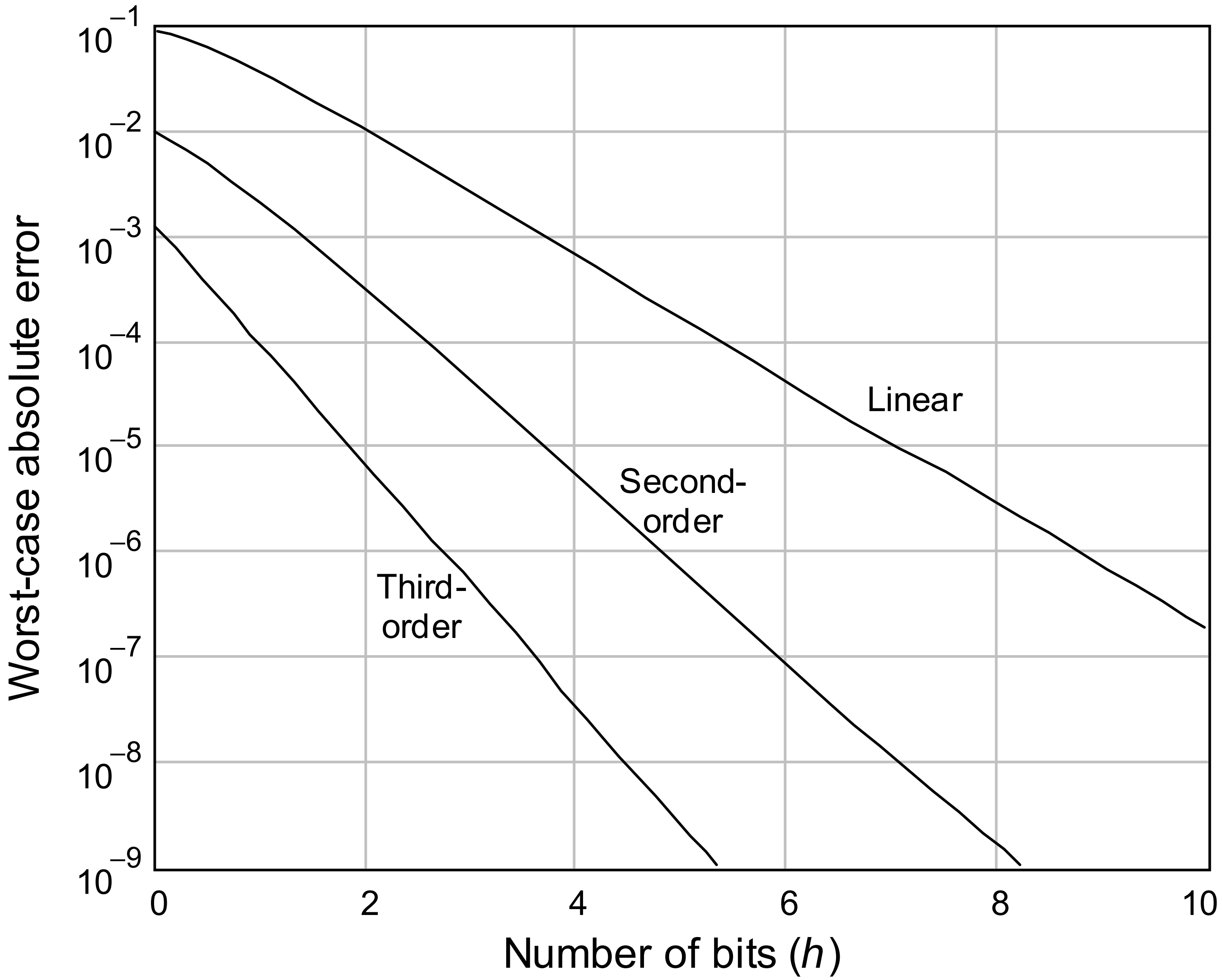


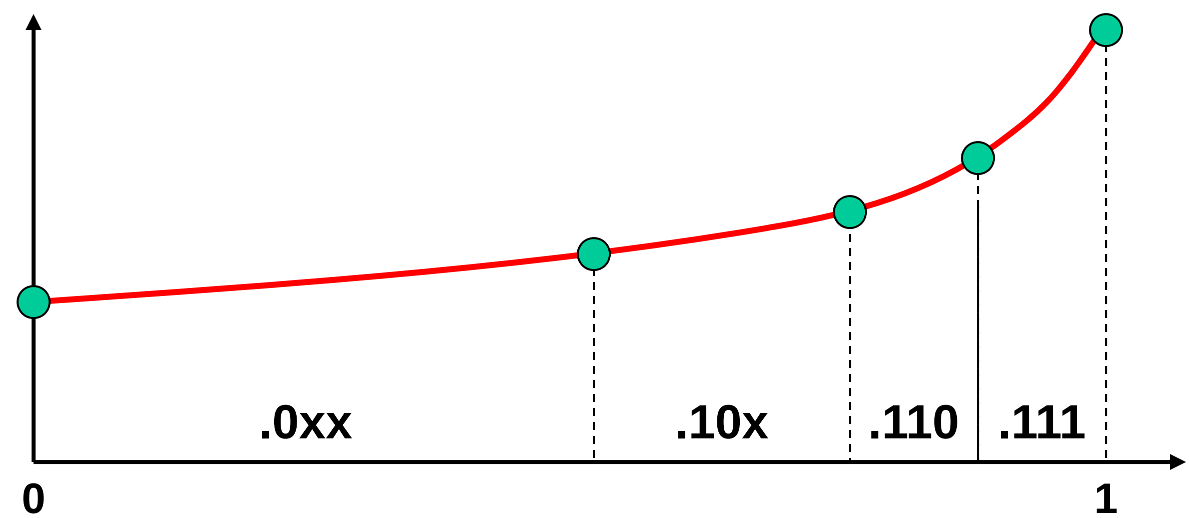
Fig. 24.7 Maximum absolute error in computing $\log_2 x$ as a function of number h of address bits for the tables with linear, quadratic (second-degree), and cubic (third-degree) interpolations [Noet89].

Interpolation with Nonuniform Intervals

One way to use interpolation with nonuniform intervals to successively divide ranges and subranges of interest into 2 parts, with finer divisions used where the function exhibits greater curvature (nonlinearity)

In this way, a number of leading bits can be used to decide which subrange is applicable

The [0,1) range divided into 4 nonuniform intervals

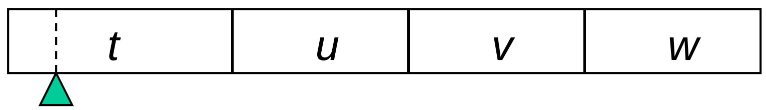


24.5 Piecewise Lookup Tables

To compute a function of a short (single) IEEE floating-point number:

Divide the 26-bit significand x (2 whole + 24 fractional bits) into 4 sections

$$x = t + \lambda u + \lambda^2 v + \lambda^3 w$$
$$= t + 2^{-6} u + 2^{-12} v + 2^{-18} w$$



where u, v, w are 6-bit fractions in [0, 1) and t, with up to 8 bits, is in [0, 4)

Taylor polynomial for $f(x)$:

$$f(x) = \sum_{i=0 \text{ to } \infty} f^{(i)}(t + \lambda u)\,(\lambda^2 v + \lambda^3 w)^i / i!$$

Ignore terms smaller than $\lambda^5 = 2^{-30}$

$$f(x) \cong f(t + \lambda u)$$
$$+ (\lambda/2)\,[f(t + \lambda u + \lambda v) - f(t + \lambda u - \lambda v)]$$
$$+ (\lambda^2/2)\,[f(t + \lambda u + \lambda w) - f(t + \lambda u - \lambda w)]$$
$$+ \lambda^4\,[(v^2/2) f^{(2)}(t) - (v^3/6) f^{(3)}(t)]$$

Use 4 additions to form these terms

Read 5 values of f from tables

Read this last term from a table

Perform 6-operand addition

Modular Reduction, or Computing z mod p

Divide the argument z into a $(b - g)$-bit upper part (x) and a g-bit lower part (y), where x ends with g zeros

$(x + y)$ mod p = (x mod p + y mod p) mod p

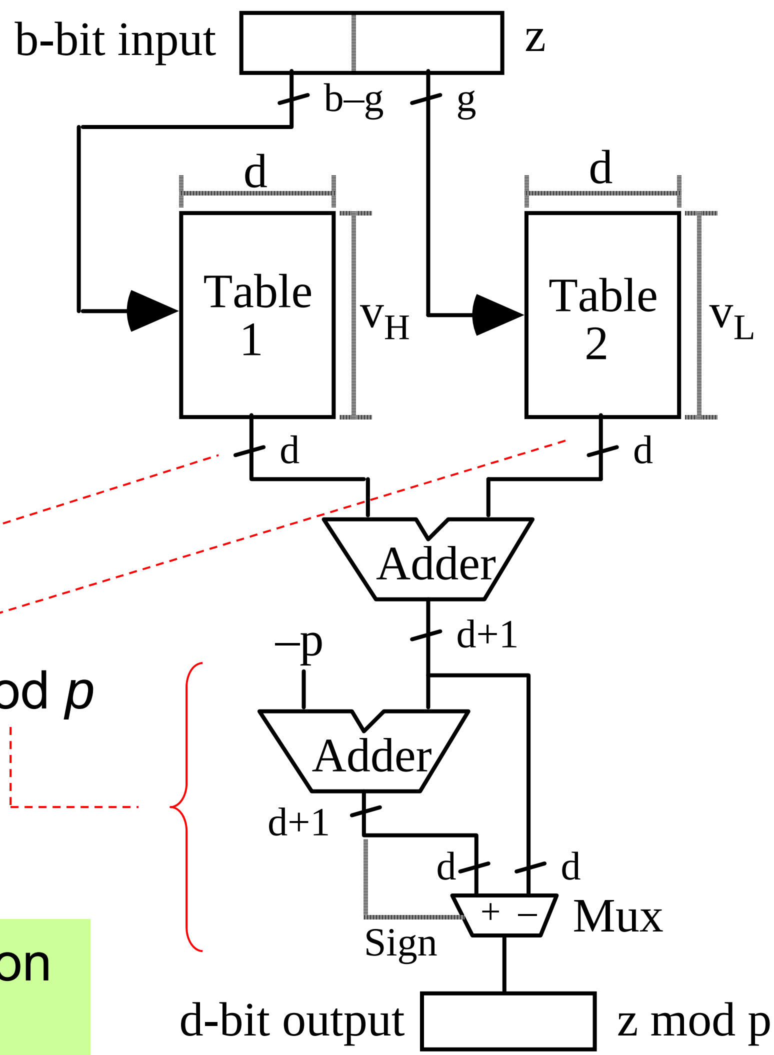
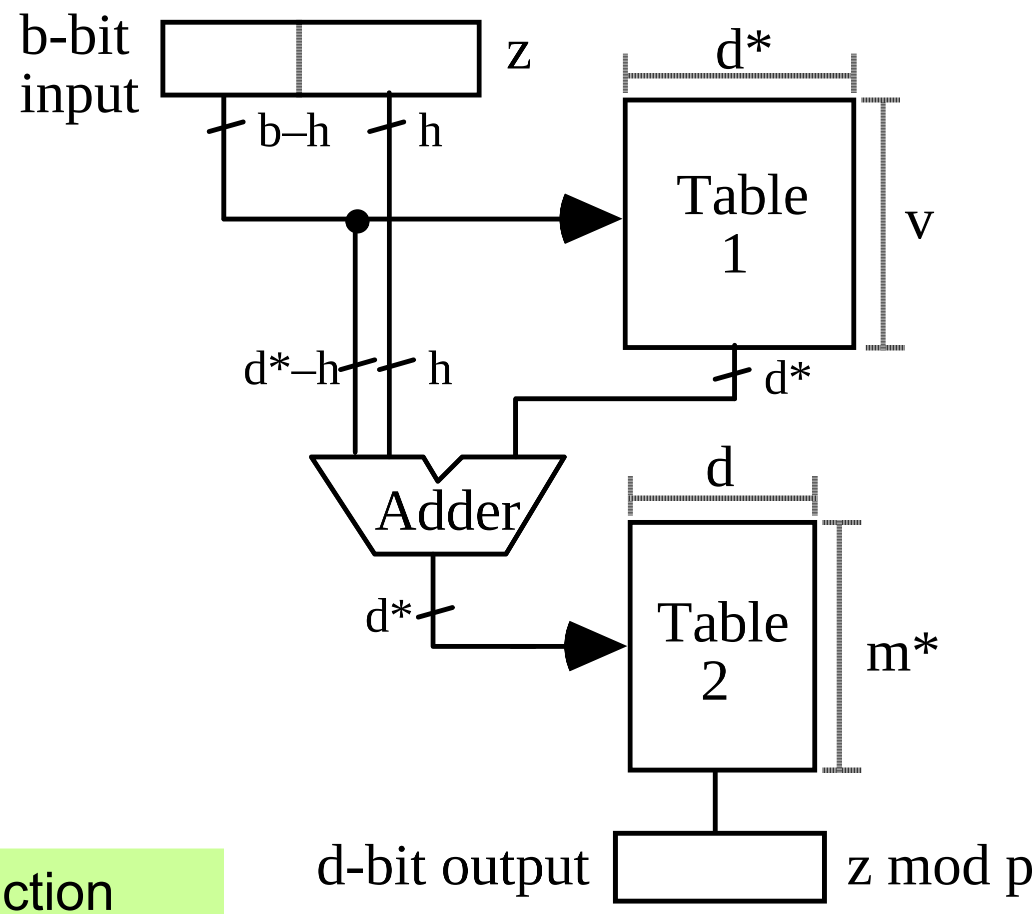


Fig. 24.8a Two-table modular reduction scheme based on divide-and-conquer.

Another Two-Table Modular Reduction Scheme

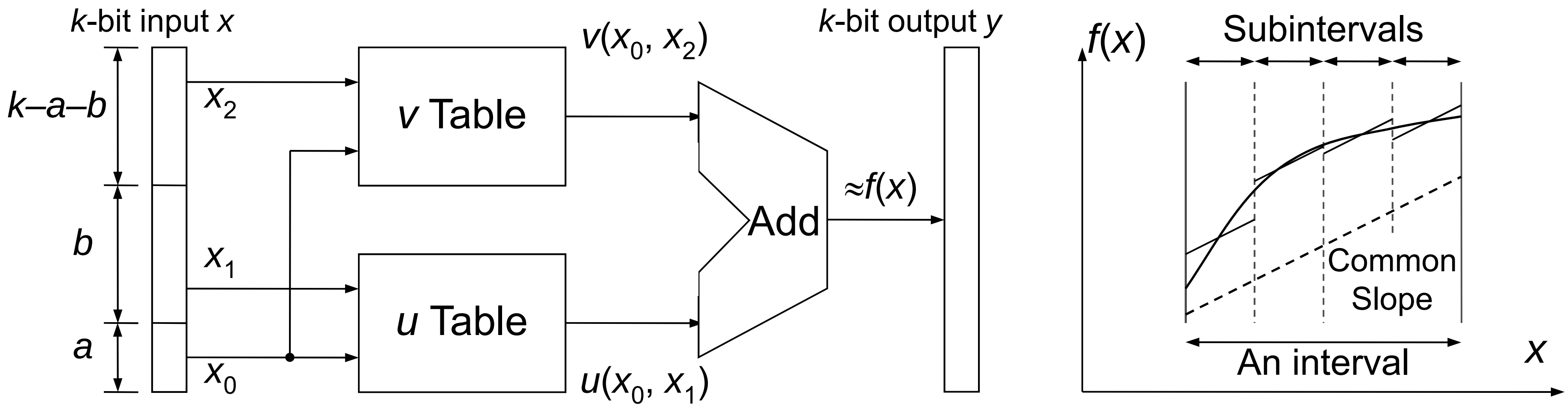
Divide the argument *z* into a (*b* – *h*)-bit upper part (*x*) and an *h*-bit lower part (*y*), where *x* ends with *h* zeros



Explanation to be added

Fig. 24.8b Modular reduction based on successive refinement.

24.6 Multipartite Table Methods



(a) Hardware realization

(b) Linear approximation

Fig. 24.9 The bipartite table method.

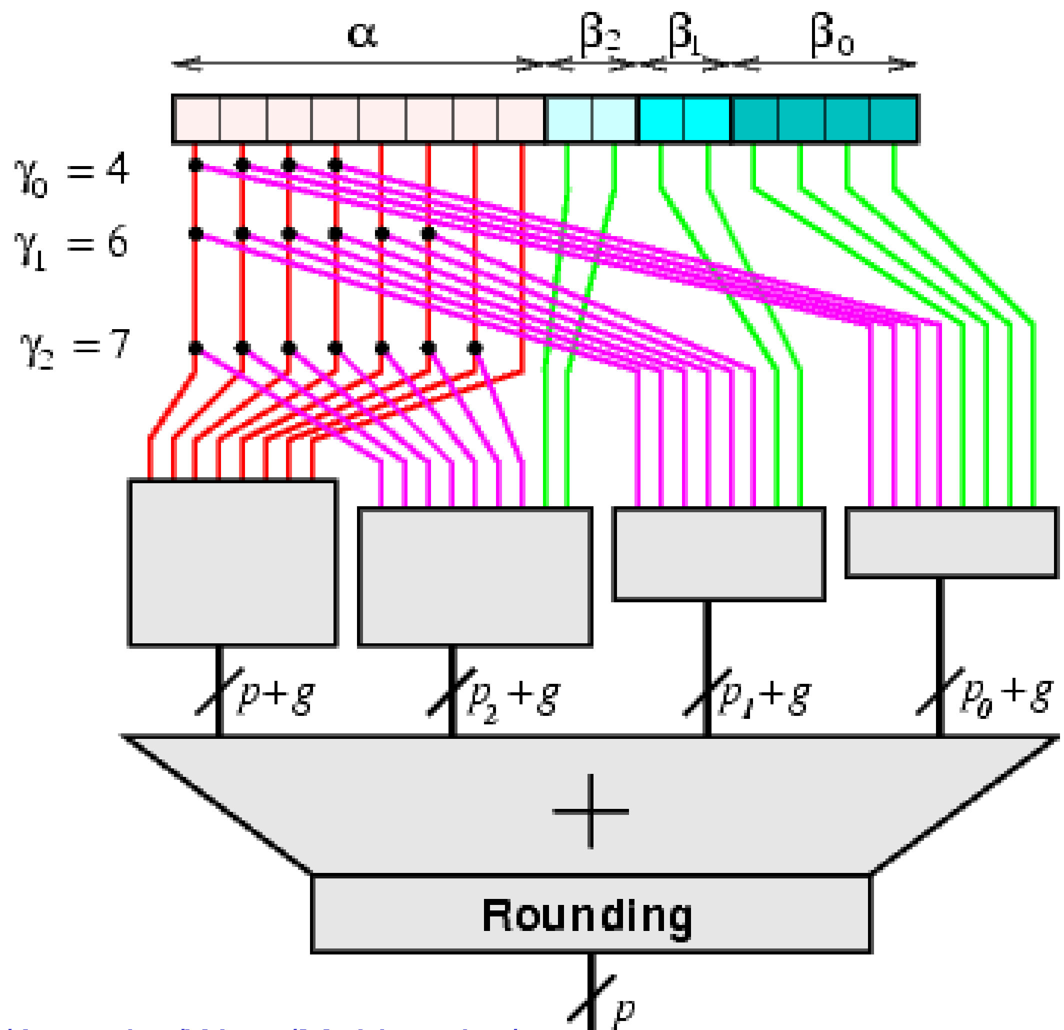
Divide the domain of interest into 2^a intervals, each of which is further divided into 2^b smaller subintervals

The trick: Use linear interpolation with an initial value determined for each subinterval and a common slope for each larger interval

Total table size is $2^{a+b} + 2^{k-b}$, in lieu of 2^k; width of table entries has been ignored in this comparison

Generalizing to Tripartite and Higher-Order Tables

Two-part tables have been generalized to multipart (3-part, 4-part, . . .) tables



Source of figure: www.ens-lyon.fr/LIP/Arenaire/Ware/Multipartite/