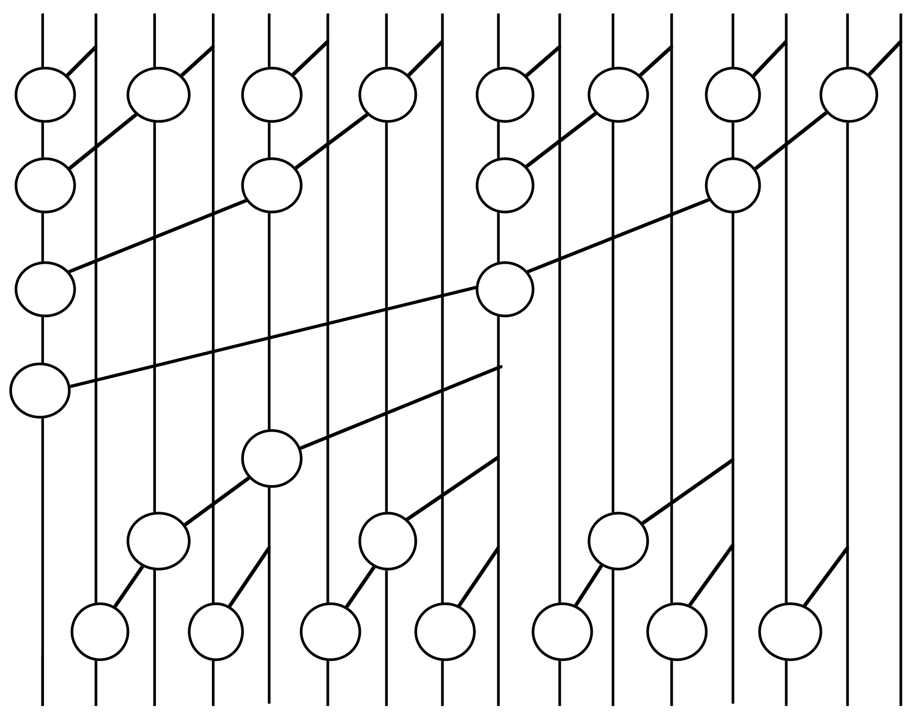
Instructor's Manual for

Computer Arithmetic

ALGORITHMS AND HARDWARE DESIGNS

Vol. 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)

ECE Dept., Univ. of California, Santa Barbara, CA 93106-9560, USA (parhami@ece.ucsb.edu)

Preface to the Instructor's Manual, Volume 2

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 in a format suitable for use as transparency masters, along with other presentation material.

The August 1999 edition of Volume 1 of the Instructor's Manual consists of the following parts:

Volume 1	Part I	Selected solutions and additional problems
	Part II	Question bank, assignments, and projects
	Part III	Additions, corrections, and other updates
	Part IV	Sample course outline, calendar, and forms

A more complete edition of Volume 1 is planned for release in the summer of 2000.

The January 2000 edition of Volume 2 is currently available as a postscript file through the author's Web address (http://www.ece.ucsb.edu/faculty/parhami):

Volume 2	Parts I-VII	Presentation material for the seven text parts

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, California
January 2000

Table of Contents

Preface to the Instructor's Manual, Volume 2 iii

Part I Number Representation 1

1 Numbers and Arithmetic 2
2 Representing Signed Numbers 9
3 Redundant Number Systems 17
4 Residue Number Systems 21

Part II Addition/Subtraction 28

5 Basic Addition and Counting 29
6 Carry-Lookahead Adders 36
7 Variations in Fast Adders 41
8 Multioperand Addition 51

Part III Multiplication 59

9 Basic Multiplication Schemes 60
10 High-Radix Multipliers 72
11 Tree and Array Multipliers 79
12 Variations in Multipliers 90

Part IV Division 99

13 Basic Division Schemes 100
14 High-Radix Dividers 108
15 Variations in Dividers 115
16 Division by Convergence 127

Part V Real Arithmetic 135

17 Floating-Point Representations 136
18 Floating-Point Operations 142
19 Errors and Error Control 149
20 Precise and Certifiable Arithmetic 157

Part VI Function Evaluation 161

21 Square-Rooting Methods 162
22 The CORDIC Algorithms 171
23 Variations in Function Evaluation 180
24 Arithmetic by Table Lookup 191

Part VII Implementation Topics 195

25 High-Throughput Arithmetic 196
26 Low-Power Arithmetic 205
27 Fault-Tolerant Arithmetic 214
28 Past, Present, and Future 225

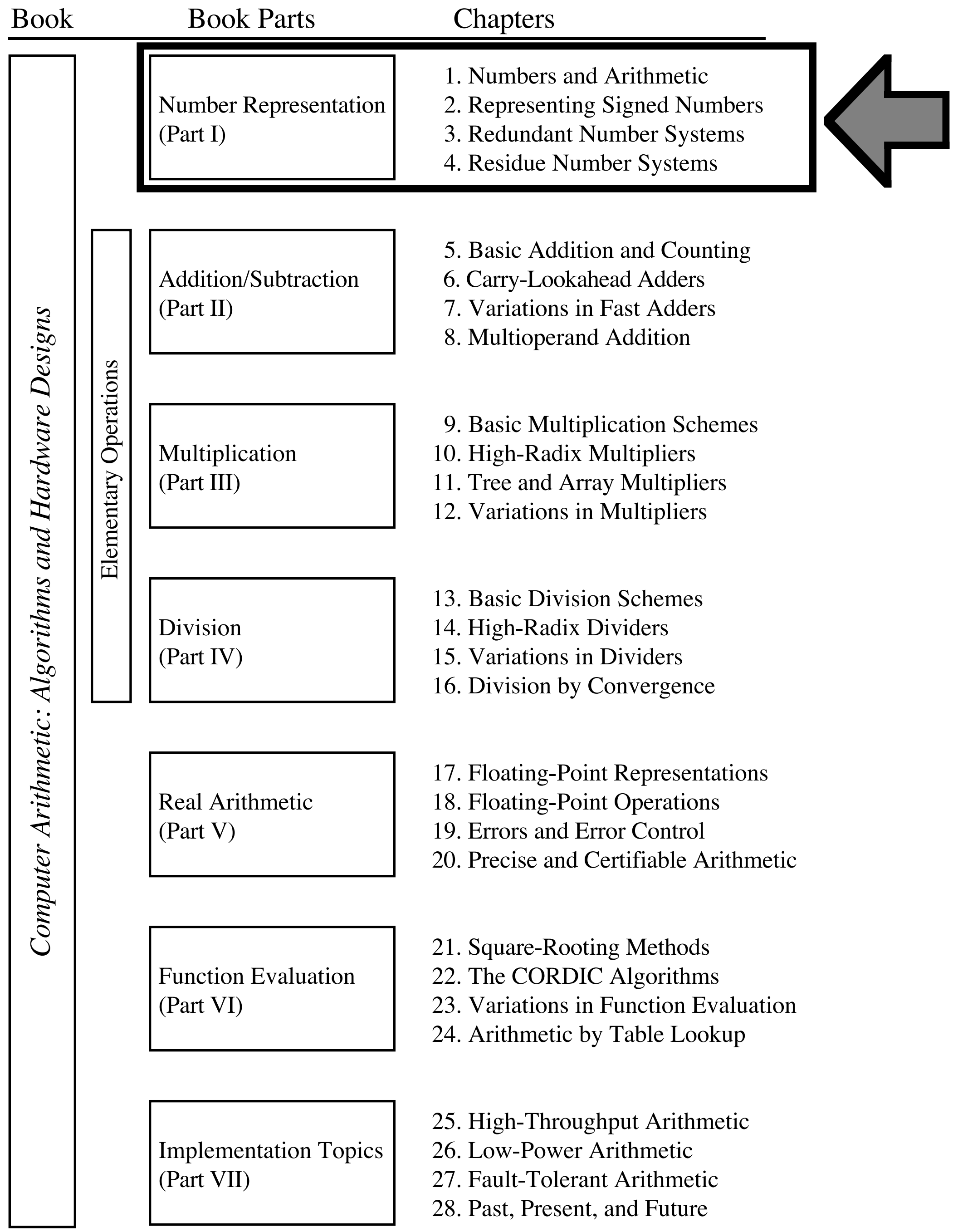


Fig. P.1 The structure of this book in parts and chapters.

Part I Number Representation

Part Goals

Review conventional number systems

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

Chapter 1 Numbers and Arithmetic
Chapter 2 Representing Signed Numbers
Chapter 3 Redundant Number Systems
Chapter 4 Residue Number Systems

1 Numbers and Arithmetic

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?
Various 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?
1.2 A Motivating Example
1.3 Numbers and Their Encodings
1.4 Fixed-Radix Positional Number Systems
1.5 Number Radix Conversion
1.6 Classes of Number Representations

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:

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

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 $+$, $-$, $\times$, $\div$, $\sqrt{\square}$, 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 $+$, $-$, $\times$, $\div$, $\sqrt{\square}$
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 √, x^2, and x^y functions, compute:

$u = \underbrace{\sqrt{\sqrt{\cdots\sqrt{2}}}}_{10\text{ times}}$ = 1.000 677 131 “the 1024th root of 2”

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

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

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

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

$y = \overbrace{(((v^2)^2)\cdots)^2}^{10\text{ times}}$ = 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

Here is one way to determine the hidden digits for u and v:

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

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

A simple analysis:

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

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 $\approx 0.1100\ 1100 \times 2^{-23} \approx 9.5 \times 10^{-8}$

Error in 100-hr operation period
$\approx 9.5 \times 10^{-8} \times 100 \times 60 \times 60 \times 10 = 0.34$ s

Distance traveled by Scud = (0.34 s) × (1676 m/s) ≈ 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 first voyage
after a decade of development costing $7 billion

Cause: "software error in the inertial reference system"

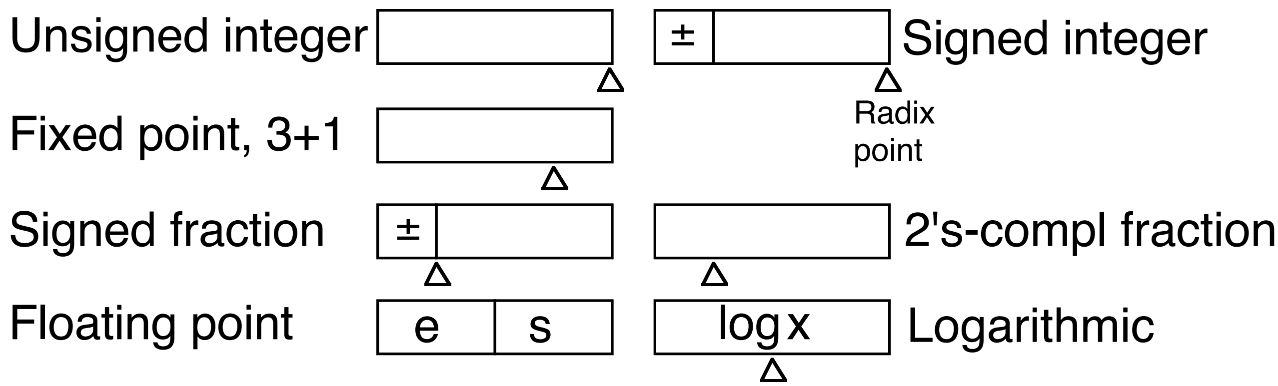
Specifics of the problem: a 64 bit floating point number
relating to the horizontal velocity of the rocket
was being converted to a 16 bit signed integer

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

Encoding of numbers in 4 bits:



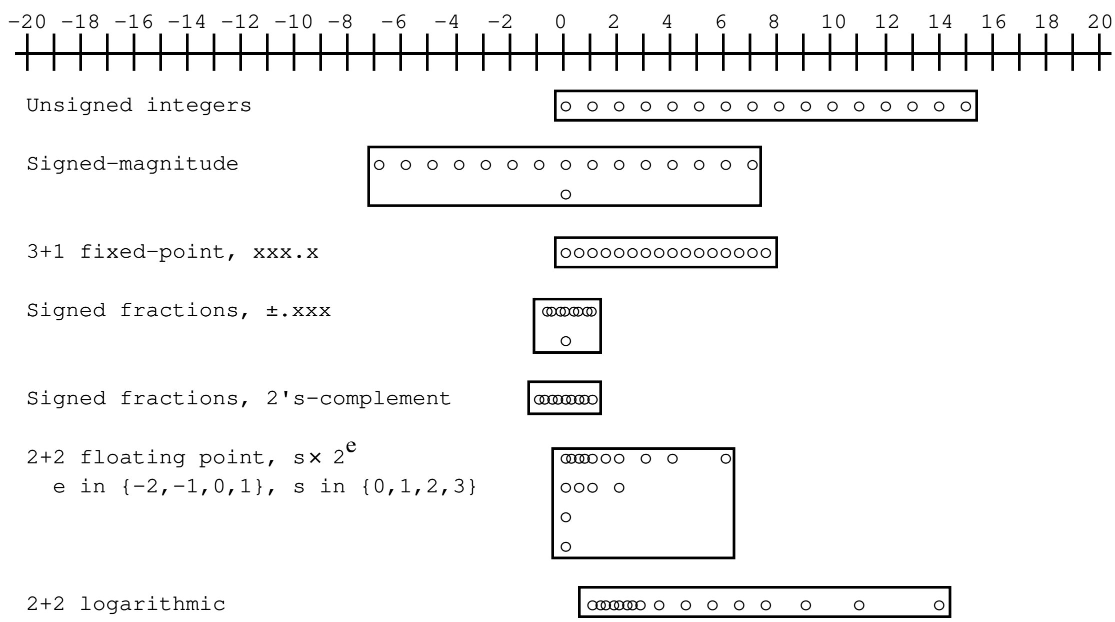


Fig. 1.2 Some of the possible ways of assigning 16 distinct codes to represent numbers.

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

$$u = w \,.\, v$$
$$= (x_{k-1}x_{k-2} \cdots x_1x_0 \,.\, x_{-1}x_{-2} \cdots x_{-l})_r \qquad \text{Old}$$
$$= (X_{K-1}X_{K-2} \cdots X_1X_0 \,.\, X_{-1}X_{-2} \cdots Xx_{-L})_R \qquad \text{New}$$

Radix conversion: arithmetic in the old radix r

Converting the 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 the 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, $(105.486)_{ten} \approx (410.22033)_{five}$

Radix conversion: arithmetic in the new radix R

Converting the whole part w

```
((((2 × 5) + 2) × 5 + 0) × 5 + 3) × 5 + 3
    |-----|     :         :         :         :
       10        :         :         :         :
   |-----------|          :         :         :
         12                :         :         :
  |---------------------|          :         :
            60                       :         :
 |-------------------------------|          :
                303                            :
|-------------------------------------------|
                         1518
```

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

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

```
(((((3 / 5) + 3) / 5 + 0) / 5 + 2) / 5 + 2) / 5
    |-----|     :         :         :         :
     0.6         :         :         :         :
   |-----------|          :         :         :
       3.6                 :         :         :
  |---------------------|          :         :
           0.72                      :         :
 |-------------------------------|          :
              2.144                            :
|-------------------------------------------|
                  2.4288
|-----------------------------------------------------|
                        0.48576
```

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)

2 Representing Signed Numbers

Chapter Goals

Learn different encodings of the sign info

Discuss implications for arithmetic design

Chapter Highlights

Using a sign bit, biasing, complementation

Properties of 2's-complement numbers

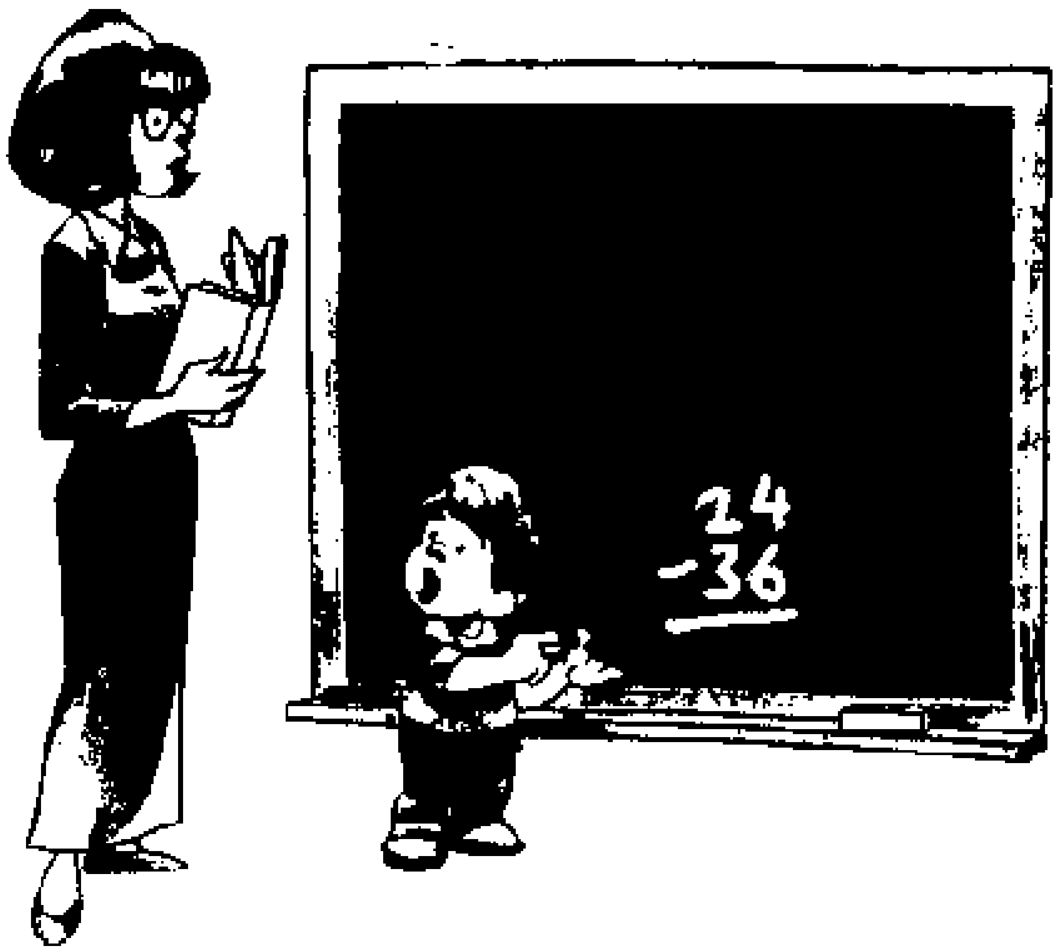
Signed vs unsigned arithmetic

Signed numbers, positions, or digits

Chapter Contents

2.1 Signed-Magnitude Representation
2.2 Biased Representations
2.3 Complement Representations
2.4 Two's- and 1's-Complement Numbers
2.5 Direct and Indirect Signed Arithmetic
2.6 Using Signed Positions or Signed Digits

When Numbers Go into the Red!



"This can't be right ... It goes into the red."

2.1 Signed-Magnitude Representation

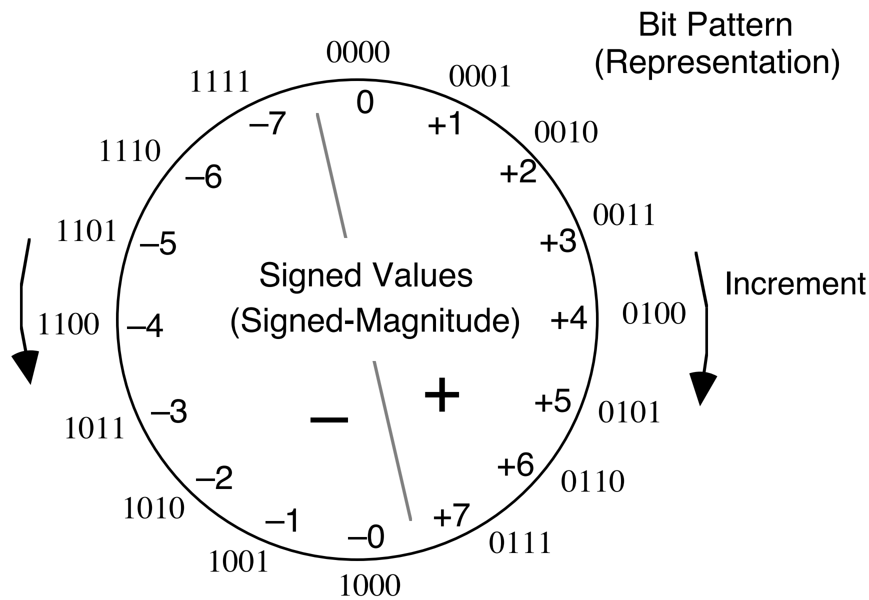


Fig. 2.1 Four-bit signed-magnitude number representation system for integers.

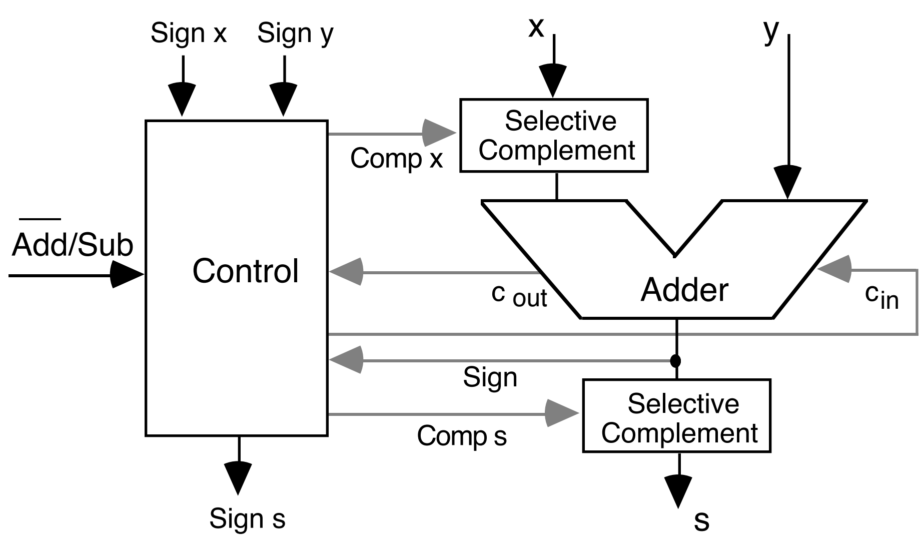


Fig. 2.2 Adding signed-magnitude numbers using precomplementation and postcomplementation.

2.2 Biased Representations

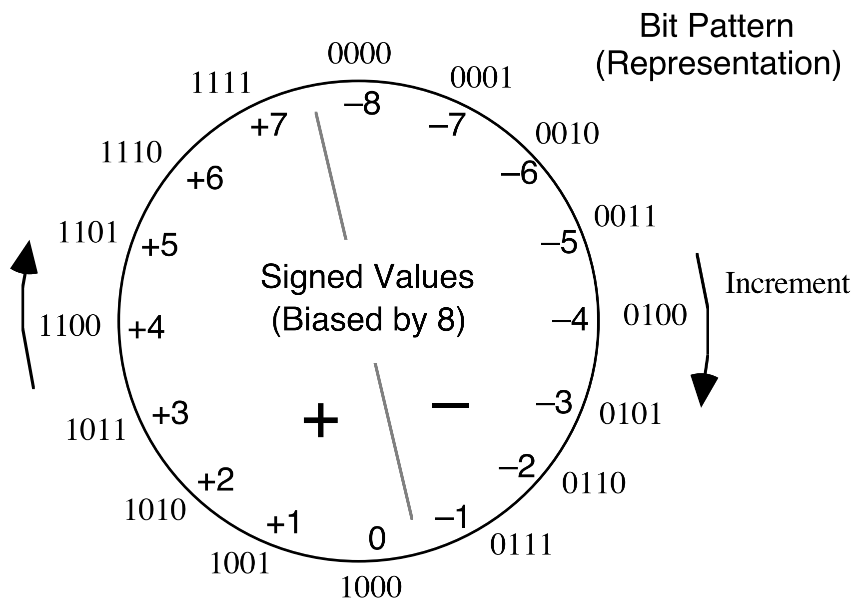


Fig. 2.3 Four-bit biased integer number representation system with a bias of 8.

Addition/subtraction of biased numbers

$$x + y + \text{bias} = (x + \text{bias}) + (y + \text{bias}) - \text{bias}$$

$$x - y + \text{bias} = (x + \text{bias}) - (y + \text{bias}) + \text{bias}$$

A power-of-2 (power-of-2 minus 1) bias simplifies the above

Comparison of biased numbers:

compare like ordinary unsigned numbers

find true difference by ordinary subtraction

2.3 Complement Representations

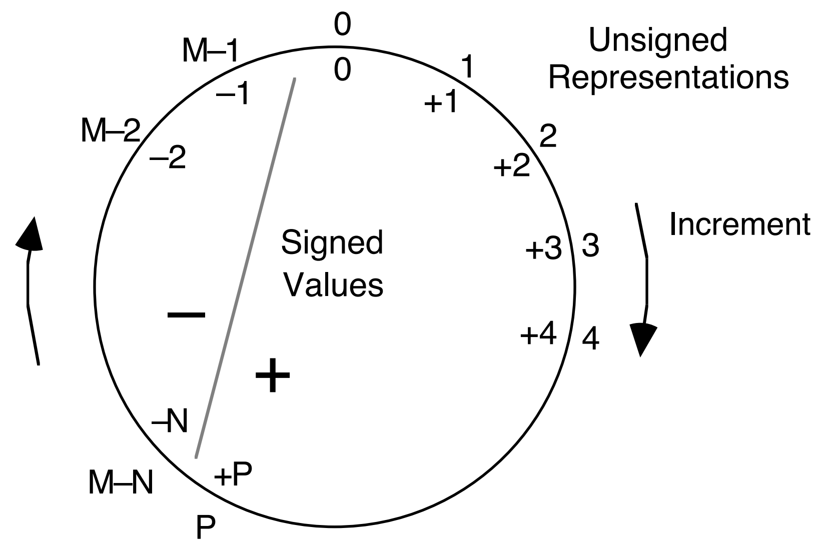


Fig. 2.4 Complement representation of signed integers.

Table 2.1 Addition in a complement number system with complementation constant M and range [–N, +P].

Desired operation	Computation to be performed mod M	Correct result with no overflow	Overflow condition
$(+x) + (+y)$	$x + y$	$x + y$	$x + y > P$
$(+x) + (-y)$	$x + (M - y)$	$x - y$ if $y \le x$ $M - (y - x)$ if $y > x$	N/A
$(-x) + (+y)$	$(M - x) + y$	$y - x$ if $x \le y$ $M - (x - y)$ if $x > y$	N/A
$(-x) + (-y)$	$(M - x) + (M - y)$	$M - (x + y)$	$x + y > N$

Example -- complement system for fixed-point numbers:

complementation constant M = 12.000

fixed-point number range [–6.000, +5.999]

represent –3.258 as 12.000 – 3.258 = 8.742

Auxiliary operations for complement representations

complementation or change of sign (computing M – x)

computations of residues mod M

Thus M must be selected to simplify these operations

Two choices allow just this for fixed-point radix-r arithmetic with k whole digits and l fractional digits

Radix complement $M = r^k$

Digit complement $M = r^k - \text{ulp}$
(diminished radix complement)

ulp (unit in least position) stands for r^{-l}

it allows us to forget about l even for nonintegers

2.4 Two's- and 1's-Complement Numbers

Two's complement = radix complement system for r = 2

$$2^k - x \;=\; [(2^k - \text{ulp}) - x] + \text{ulp} \;=\; x^{\text{compl}} + \text{ulp}$$

Range of representable numbers in a 2's-complement number system with k whole bits:

$$\text{from } -2^{k-1} \text{ to } 2^{k-1} - \text{ulp}$$

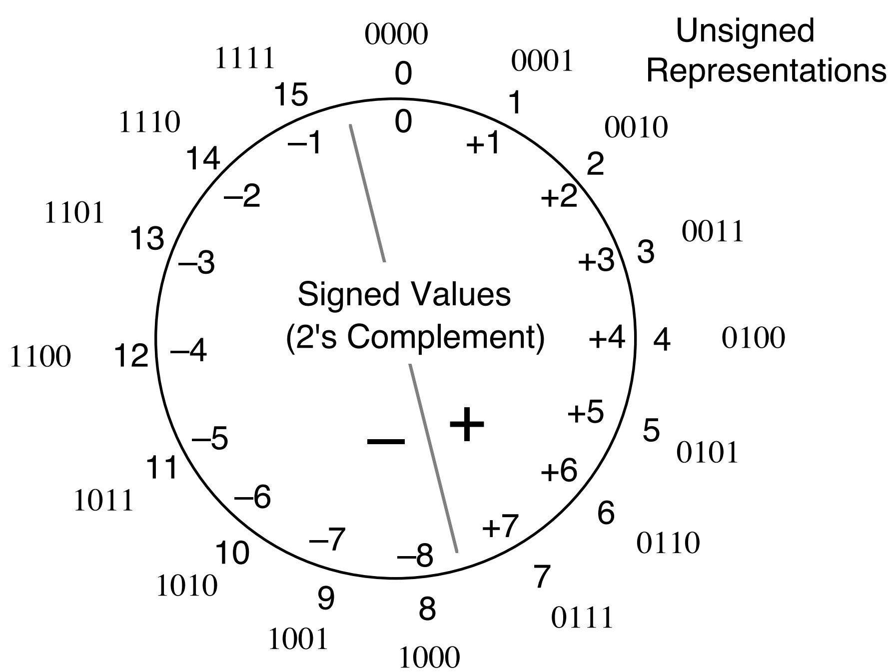


Fig. 2.5 Four-bit 2's-complement number representation system for integers.

Range/precision extension for 2's-complement numbers

$$\cdots\, x_{k-1}x_{k-1}x_{k-1}x_{k-1}x_{k-2} \cdots x_1x_0 \,.\, x_{-1}x_{-2} \cdots x_{-l}\, 0\;0\;0 \cdots$$

One's complement = digit complement system for r = 2

$$(2^k - \text{ulp}) - x \ = \ x^{\text{compl}}$$

Mod-(2^k – ulp) operation is done via end-around carry

$$(x + y) - (2^k - \text{ulp}) = x - y - 2^k + \text{ulp}$$

Range of representable numbers in a one's-complement number system with k whole bits:

$$\text{from} \quad -2^{k-1} \ \text{ to } \ 2^{k-1} - \text{ulp}$$

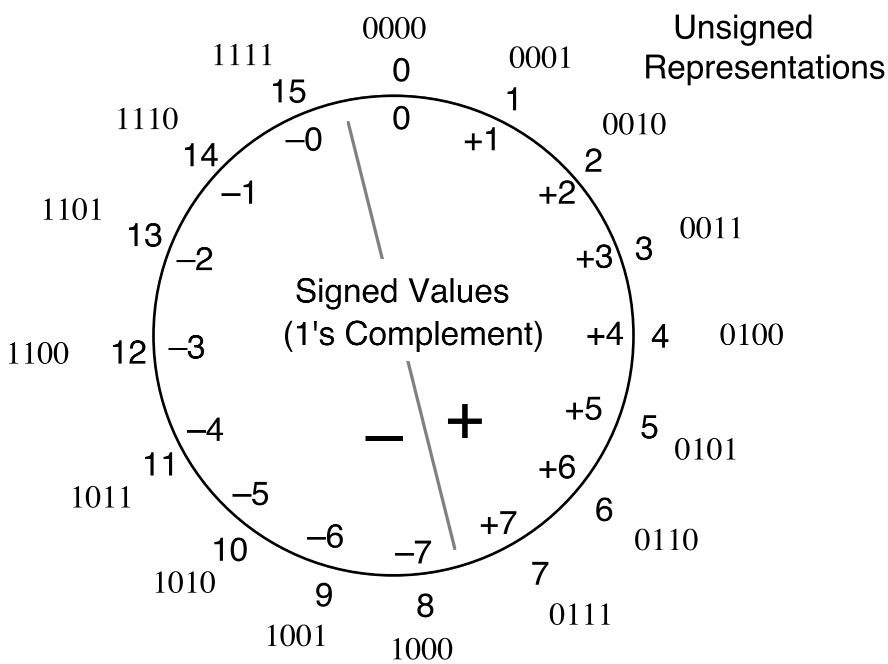


Fig. 2.6 Four-bit 1's-complement number representation system for integers.

Table 2.2 Comparing radix- and digit-complement number representation systems

Feature/Property	Radix complement	Digit complement
Symmetry (P = N?)	Possible for odd r (radices of practical interest are even)	Possible for even r
Unique zero?	Yes	No
Complementation	Complement all digits and add ulp	Complement all digits
Mod-M addition	Drop the carry-out	End-around carry

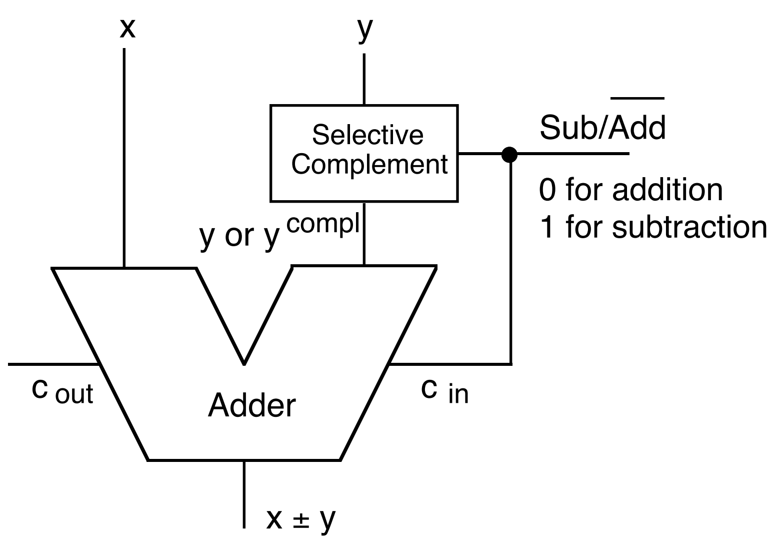


Fig. 2.7 Adder/subtractor architecture for two's-complement numbers.

2.5 Direct and Indirect Signed Arithmetic

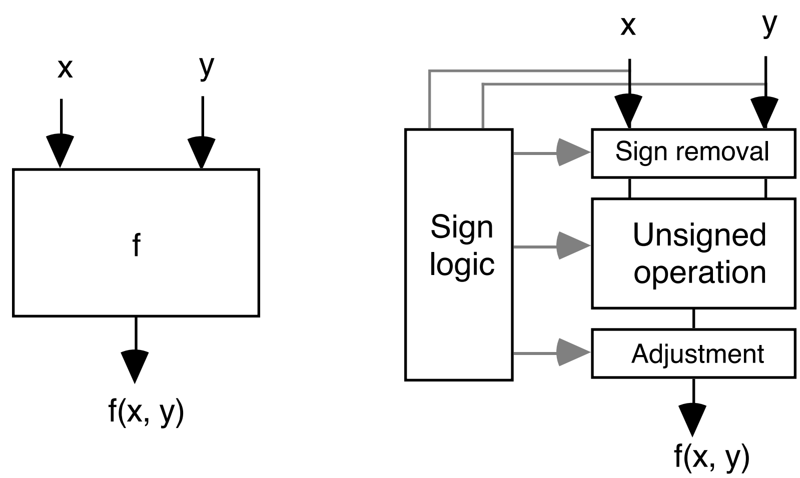


Fig. 2.8 Direct vs indirect operation on signed numbers.

Advantage of direct signed arithmetic

usually faster (not always)

Advantages of indirect signed arithmetic

can be simpler (not always)

allows sharing of signed/unsigned hardware

when both operation types are needed

2.6 Using Signed Positions or Signed Digits

A very important property of 2's-complement numbers that is used extensively in computer arithemetic:

$x = ($	1	0	1	0	0	1	1	0	$)_{\text{two's-compl}}$
	-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
	-128		$+32$			$+4$	$+2$		$= -90$
Check:									
$x = ($	1	0	1	0	0	1	1	0	$)_{\text{two's-compl}}$
$-x = ($	0	1	0	1	1	0	1	0	$)_{\text{two}}$
	-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
		64		$+16$	$+8$		$+2$		$= 90$

Fig. 2.9 Interpreting a 2's-complement number as having a negatively weighted most-significant digit.

Generalization: associate a sign with each digit position

$$\lambda = (\lambda_{k-1}\lambda_{k-2} \cdots \lambda_1\lambda_0 . \lambda_{-1}\lambda_{-2} \cdots \lambda_{-l}) \quad \lambda_i \text{ in } \{-1, 1\}$$

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

$$\lambda = 1 \; 1 \; 1 \; \cdots \; 1 \; 1 \; 1 \; 1 \quad \text{positive-radix}$$
$$\lambda = -1 \; 1 \; 1 \; \cdots \; 1 \; 1 \; 1 \; 1 \quad \text{two's-complement}$$
$$\lambda = \cdots \; -1 \; 1 \; -1 \; 1 \quad \text{negative-radix}$$

Signed digits: associate signs not with digit positions but with the digits themselves

	3	1	2	0	2	3	Original digits in [0, 3]
	–1	1	2	0	2	–1	Rewritten digits in [–1, 2]
1	0	0	0	0	1		Transfer digits in [0, 1]
1	–1	1	2	0	3	–1	Sum digits in [–1, 3]
1	–1	1	2	0	–1	–1	Rewritten digits in [–1, 2]
0	0	0	0	1	0		Transfer digits in [0, 1]
1	–1	1	2	1	–1	–1	Sum digits in [–1, 3]

Fig. 2.10 Converting a standard radix-4 integer to a radix-4 integer with the non-standard digit set [–1, 2].

	3	1	2	0	2	3	Original digits in [0, 3]
	–1	1	–2	0	–2	–1	Interim digits in [–2, 1]
1	0	1	0	1	1		Transfer digits in [0, 1]
1	–1	2	–2	1	–1	–1	Sum digits in [–2, 2]

Fig. 2.11 Converting a standard radix-4 integer to a radix-4 integer with the non-standard digit set [–2, 2].

3 Redundant Number Systems

Chapter Goals

Explore the advantages and drawbacks of using more than r digit values in radix r

Chapter Highlights

Redundancy eliminates long carry chains
Redundancy takes many forms: tradeoffs
Conversions between redundant and nonredundant representations
Redundancy used for endpoint values too?

Chapter Contents

3.1 Coping with the Carry Problem
3.2 Redundancy in Computer Arithmetic
3.3 Digit Sets and Digit-Set Conversions
3.4 Generalized Signed-Digit Numbers
3.5 Carry-Free Addition Algorithms
3.6 Conversions and Support Functions

3.1 Coping with the Carry Problem

The carry problem can be dealt with in several ways:

1. Limit carry propagation to within a small number of bits
2. Detect the end of propagation; don't wait for worst case
3. Speed up propagation via lookahead and other methods
4. Ideal: Eliminate carry propagation altogether!

	5	7	8	2	4	9	
+	6	2	9	3	8	9	Operand digits in [0, 9]
	11	9	17	5	12	18	Position sums in [0, 18]

But how can we extend this beyond a single addition?

		11	9	17	10	12	18	
+		6	12	9	10	8	18	Operand digits in [0, 18]
		17	21	26	20	20	36	Position sums in [0, 36]
		7	11	16	0	10	16	Interim sums in [0, 16]
	1	1	1	2	1	2		Transfer digits in [0, 2]
	1	8	12	18	1	12	16	Sum digits in [0, 18]

Fig. 3.1 Adding radix-10 numbers with digit set [0, 18].

Position sum decomposition $[0, 36] = 10 \times [0, 2] + [0, 16]$

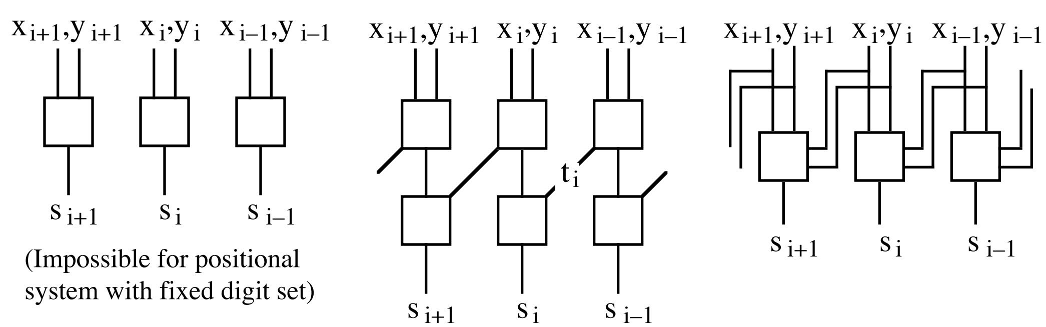
Absorption of transfer digit $[0, 16] + [0, 2] = [0, 18]$

So, redundancy helps us achieve carry-free addition

But how much redundancy is actually needed?

	11	10	7	11	3	8	
+	7	2	9	10	9	8	Operand digits in [0, 11]
	18	12	16	21	12	16	Position sums in [0, 22]
	8	2	6	1	2	6	Interim sums in [0, 9]
1	1	1	2	1	1		Transfer digits in [0, 2]
1	9	3	8	2	3	6	Sum digits in [0, 11]

Fig. 3.3 Adding radix-10 numbers with digit set [0, 11].



(a) Ideal single-stage carry-free. (b) Two-stage carry-free. (c) Single-stage with lookahead.

Fig. 3.2 Ideal and practical carry-free addition schemes.

3.2 Redundancy in Computer Arithmetic

Oldest example of redundancy in computer arithmetic is the stored-carry representation (carry-save addition):

```
      0 0 1 0 0 1    First binary number
    + 0 1 1 1 1 0    Add second binary number
    -------------
      0 1 2 1 1 1    Position sums in [0, 2]
    + 0 1 1 1 0 1    Add third binary number
    -------------
      0 2 3 2 1 2    Position sums in [0, 3]
      | | | | | |
      0 0 1 0 1 0    Interim sums in [0, 1]
     / / / / / /
    0 1 1 1 0 1      Transfer digits in [0, 1]
    -------------
      1 1 2 0 2 0    Position sums in [0, 2]
    + 0 0 1 0 1 1    Add fourth binary number
    -------------
      1 1 3 0 3 1    Position sums in [0, 3]
      | | | | | |
      1 1 1 0 1 1    Interim sums in [0, 1]
     / / / / / /
    0 0 1 0 1 0      Transfer digits in [0, 1]
    -------------
      1 2 1 1 1 1    Sum digits in [0, 2]
```

Fig. 3.4 Addition of 4 binary numbers, with the sum obtained in stored-carry form.

Possible 2-bit encoding for binary stored-carry digits:

0	represented as	0 0		
1	represented as	0 1	or	1 0
2	represented as	1 1		

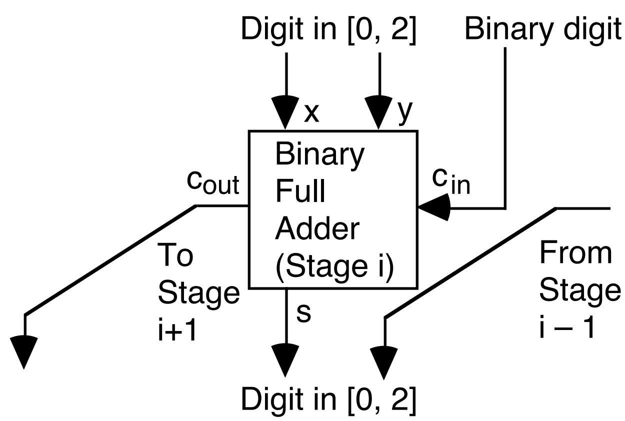


Fig. 3.5 Carry-save addition using an array of independent binary full adders.

3.3 Digit Sets and Digit-Set Conversions

Example 3.1: Convert from digit set [0, 18] to the digit set [0, 9] in radix 10.

	11	9	17	10	12	18	Rewrite 18 as 10 (carry 1) + 8
	11	9	17	10	13	8	13 = 10 (carry 1) + 3
	11	9	17	11	3	8	11 = 10 (carry 1) + 1
	11	9	18	1	3	8	18 = 10 (carry 1) + 8
	11	10	8	1	3	8	10 = 10 (carry 1) + 0
	12	0	8	1	3	8	12 = 10 (carry 1) + 2
1	2	0	8	1	3	8	Answer; all digits in [0, 9]

Example 3.2: Convert from digit set [0, 2] to digit set [0, 1] in radix 2.

	1	1	2	0	2	0	Rewrite 2 as 2 (carry 1) + 0
	1	1	2	1	0	0	2 = 2 (carry 1) + 0
	1	2	0	1	0	0	2 = 2 (carry 1) + 0
	2	0	0	1	0	0	2 = 2 (carry 1) + 0
1	0	0	0	1	0	0	Answer; all digits in [0, 1]

Another way: Decompose the carry-save number into two numbers and add them:

	1	1	1	0	1	0	First number: "Sum" bits
+	0	0	1	0	1	0	Second number: "Carry" bits
1	0	0	0	1	0	0	Sum of the two numbers

Example 3.3: Convert from digit set [0, 18] to the digit set [–6, 5] in radix 10 (same as Example 3.1, but with an asymmetric target digit set)

	11	9	17	10	12	18	**Rewrite 18 as 20 (carry 2) – 2**
	11	9	17	10	14	$^{-}2$	**14 = 10 (carry 1) + 4 [or 20 – 6]**
	11	9	17	11	4	$^{-}2$	**11 = 10 (carry 1) + 1**
	11	9	18	1	4	$^{-}2$	**18 = 20 (carry 1) + $^{-}2$**
	11	11	$^{-}2$	1	4	$^{-}2$	**11 = 10 (carry 1) + 1**
	12	1	$^{-}2$	1	4	$^{-}2$	**12 = 10 (carry 1) + 2**
1	2	1	$^{-}2$	1	4	$^{-}2$	**Answer; all digits in [0, 9]**

Example 3.4: Convert from digit set [0, 2] to digit set [–1, 1] in radix 2 (same as Example 3.2, but with the target digit set [–1, 1] instead of [0, 1])

Carry-free conversion:

	1	1	2	0	2	0	**Given carry-save number**
	$^{-}1$	$^{-}1$	0	0	0	0	**Interim digits in [–1, 0]**
1	1	1	0	1	0		**Transfer digits in [0, 1]**
1	0	0	0	1	0	0	**Answer; all digits in [0, 1]**

3.4 Generalized Signed-Digit Numbers

Radix r

Digit set $[-\alpha, \beta]$ feasibility requirement $\alpha + \beta + 1 \geq r$

Redundancy index $\rho = \alpha + \beta + 1 - r$

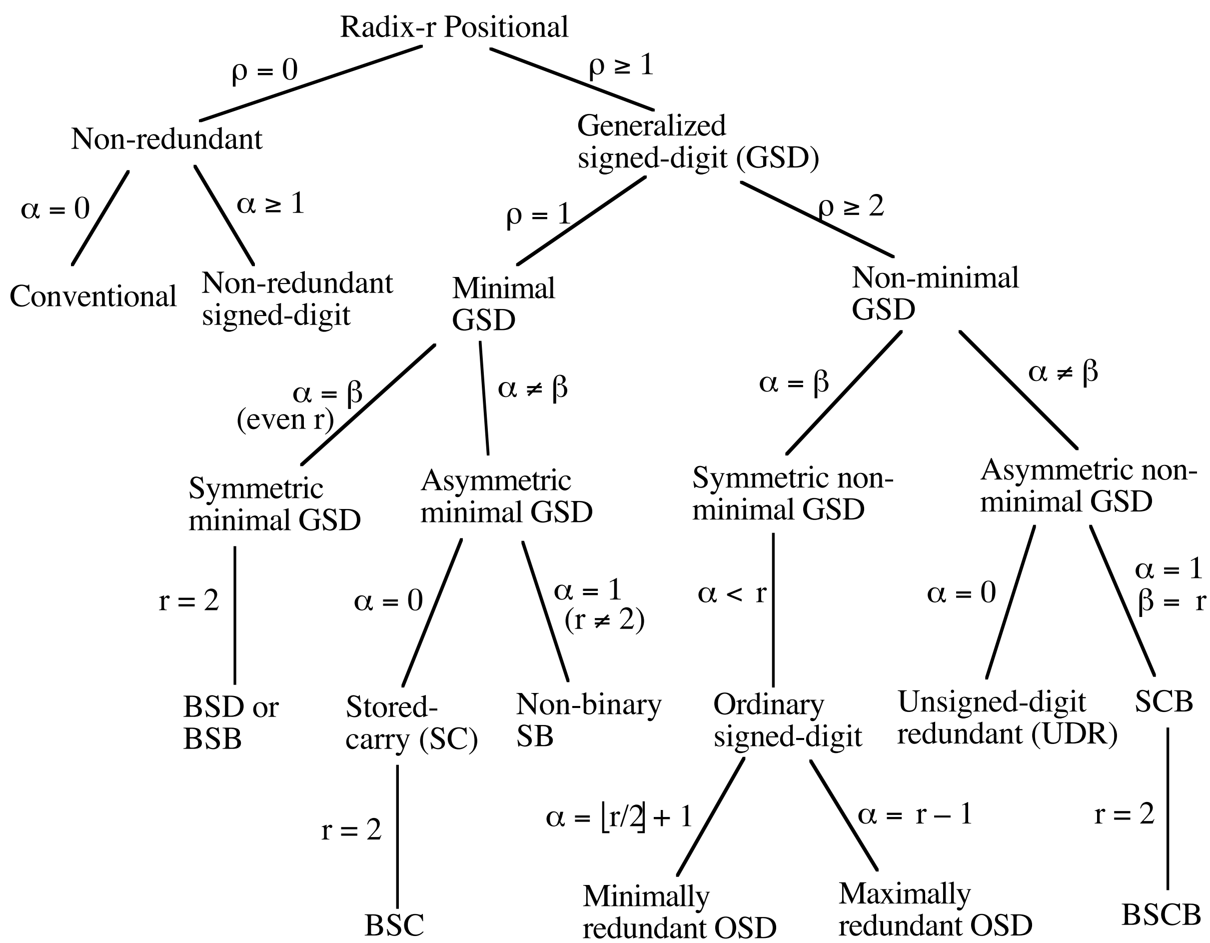


Fig. 3.6 A taxonomy of redundant and non-redundant positional number systems.

Binary vs multivalue-logic encoding of GSD digit sets

x_i	1	−1	0	−1	0	BSD representation of +6
(s,v)	01	11	00	11	00	Sign & value encoding
2’s-compl	01	10	00	10	00	2-bit 2’s-complement
(n,p)	01	10	00	10	00	Negative & positive flags
(n,z,p)	001	100	010	100	010	1-out-of-3 encoding

Fig. 3.7 Four encodings for the BSD digit set [–1, 1].

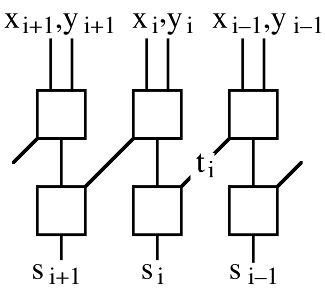
The hybrid example in Fig. 3.8, with a regular pattern of binary (B) and BSD positions, can be viewed as an implementation of a GSD system with

r = 8	Three positions form one digit
digit set [–4, 7]	−1 0 0 to 1 1 1

	BSD	**B**	**B**	**BSD**	**B**	**B**	**BSD**	**B**	**B**	Type
	1	0	1	−1	0	1	−1	0	1	x_i
+	0	1	1	−1	1	0	0	1	0	y_i
	1	1	2	–2	1	1	–1	1	1	p_i
	–1			0			–1			w_i
1			–1			0			0	t_{i+1}
1	−1	1	1	0	1	1	−1	1	1	s_i

Fig. 3.8 Example of addition for hybrid signed-digit numbers.

3.5 Carry-Free Addition Algorithms



<u>Carry-free addition of GSD numbers</u>

Compute the position sums $p_i = x_i + y_i$

Divide p_i into a transfer t_{i+1} and an interim sum $w_i = p_i - rt_{i+1}$

Add the incoming transfers to get the sum digits $s_i = w_i + t_i$

If the transfer digits t_i are in $[-\lambda, \mu]$, we must have:

$$-\alpha + \lambda \quad \le \quad \underbrace{p_i - rt_{i+1}}_{\text{interim sum}} \quad \le \quad \beta - \mu$$

Smallest interim sum if a transfer of $-\lambda$ is to be absorbable

Largest interim sum if a transfer of μ is to be absorbable

These constraints lead to

$$\lambda \ge \frac{\alpha}{r - 1} \qquad \mu \ge \frac{\beta}{r - 1}$$

Constants	$C_{-\lambda}$	$C_{-\lambda+1}$	$C_{-\lambda+2}$	$\cdots$	C_0	C_1	$\cdots$	$C_{\mu-1}$	C_μ	$C_{\mu+1}$
	$-\infty$	$\vert$	$\vert$		$\vert$	$\vert$		$\vert$	$\vert$	$+\infty$
p_i range	[---)	[----)	[---)	$\cdots$	[---)	[---)	$\cdots$	[---)	[----)	
t_{i+1} chosen	$-\lambda$	$-\lambda+1$	$-\lambda+2$		0	1		$\mu-1$	μ	

Fig. 3.9 Choosing the transfer digit t_{i+1} based on comparing the interim sum p_i to the comparison constants C_j.

Example 3.5: $r = 10$, digit set $[-5, 9]$ $\Rightarrow$ $\lambda \geq 5/9$, $\mu \geq 1$

Choose the minimal values:

$\lambda_{min} = \mu_{min} = 1$ i.e., transfer digits are in $[-1, 1]$

$-\infty = C_{-1}$ $\quad -4 \leq C_0 \leq -1$ $\quad 6 \leq C_1 \leq 9$ $\quad C_2 = +\infty$

Deriving the range of C_1: The position sum p_i is in $[-10, 18]$

We can set t_{i+1} to 1 for p_i values as low as 6

We must transfer 1 for p_i values of 9 or more

For $p_i \geq C_1$, where $6 \leq C_1 \leq 9$, we choose $t_{i+1} = 1$

For $p_i < C_0$, we choose $t_{i+1} = -1$, where $-4 \leq C_0 \leq -1$

In all other cases, $t_{i+1} = 0$

If p_i is given as a 6-bit 2's-complement number abcdef, good choices for the constants are $C_0 = -4$, $C_1 = 8$

The logic expressions for the signals g_1 and g_{-1}:

$g_{-1} = a(\bar{c} + \bar{d})$ generate a transfer of -1

$g_1 = \bar{a}(b + c)$ generate a transfer of 1

		3	−4	9	−2	8	x_i in [–5, 9]
+		8	−4	9	8	1	y_i in [–5, 9]
		11	–8	18	6	9	p_i in [–10, 18]
		1	2	8	6	–1	w_i in [–4, 8]
	1	–1	1	0	1		t_{i+1} in [–1, 1]
	1	0	3	8	7	−1	s_i in [–5, 9]

Fig. 3.10 Adding radix-10 numbers with digit set [–5, 9].

The preceding carry-free addition algorithm is applicable if

$r > 2, \rho \geq 3$

$r > 2, \rho = 2, \alpha \neq 1, \beta \neq 1$

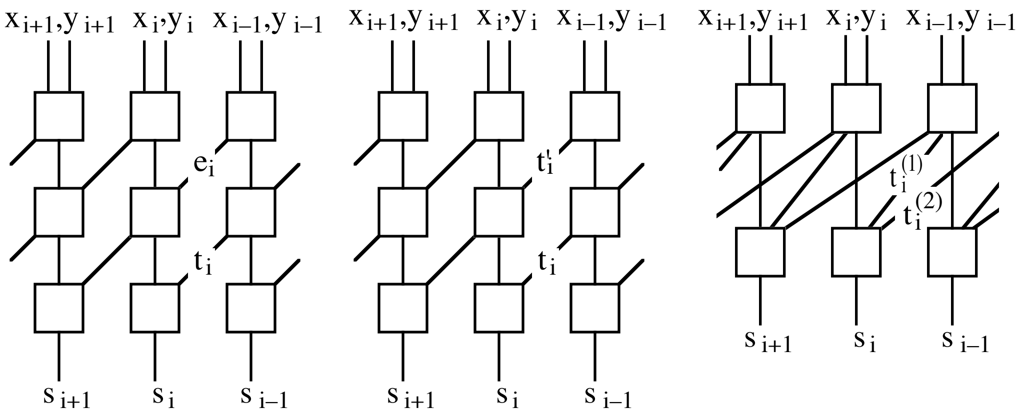
In other words, it is inapplicable for

$r = 2$

$\rho = 1$

$\rho = 2$ with $\alpha = 1$ or $\beta = 1$

Fortunately, in such cases, a limited-carry algorithm is always applicable



(a) Three-stage carry estimate. (b) Three-stage repeated-carry. (c) Two-stage parallel-carries.

Fig. 3.11 Some implementations for limited-carry addition.

	1	−1	0	−1	0	x_i in [−1, 1]
+	0	−1	−1	0	1	y_i in [−1, 1]
	1	−2	−1	−1	1	p_i in [−2, 2]
high	low	high	low	high	high	e_i in {low:[−1, 0], high:[0, 1]}
	1	0	1	−1	−1	w_i in [−1, 1]
0	−1	−1	0	1		t_{i+1} in [−1, 1]
0	0	−1	1	0	−1	s_i in [−1, 1]

Fig. 3.12 Limited-carry addition of radix-2 numbers with digit set [–1, 1] using carry estimates. A position sum of –1 is kept intact when the incoming transfer is in [0, 1], whereas it is rewritten as 1 with a carry of –1 for incoming transfer in [–1, 0]. This guarantees that $t_i \neq w_i$ and thus $-1 \le s_i \le 1$.

		1	1	3	1	2	x_i in [0, 3]
+		0	0	2	2	1	y_i in [0, 3]
		1	1	5	3	3	p_i in [0, 6]
	low	low	high	low	low	low	e_i in {low:[0, 2], high:[1, 3]}
		1	–1	1	1	1	w_i in [–1, 1]
	0	1	2	1	1		t_{i+1} in [0, 3]
	0	2	1	2	2	1	s_i in [0, 3]

Fig. 3.13 Limited-carry addition of radix-2 numbers with the digit set [0, 3] using carry estimates. A position sum of 1 is kept intact when the incoming transfer is in [0, 2], whereas it is rewritten as –1 with a carry of 1 if the incoming transfer is in [1, 3].

3.6 Conversions and Support Functions

BSD-to-binary conversion example

1	**-1**	**0**	**-1**	**0**	**BSD representation of +6**
1	**0**	**0**	**0**	**0**	**Positive part (1 digits)**
0	**1**	**0**	**1**	**0**	**Negative part (-1 digits)**
0	**0**	**1**	**1**	**0**	**Difference = conversion result**

Zero test: zero has a unique code under some conditions

Sign test: needs carry propagation

$$
\begin{array}{cccccc l}
 & x_{k-1} & x_{k-2} & \cdots & x_1 & x_0 & \text{k-digit GSD operands} \\
+ & y_{k-1} & y_{k-2} & \cdots & y_1 & y_0 & \\
\hline
 & p_{k-1} & p_{k-2} & \cdots & p_1 & p_0 & \text{Position sums} \\
 & | & | & & | & | & \\
 & w_{k-1} & w_{k-2} & \cdots & w_1 & w_0 & \text{Interim sum digits} \\
 / & & / & & / & / & \\
t_k & t_{k-1} & \cdots & t_2 & t_1 & & \text{Transfer digits} \\
\hline
 & s_{k-1} & s_{k-2} & \cdots & s_1 & s_0 & \text{k-digit “apparent” sum}
\end{array}
$$

Fig. 3.16. Overflow and its detection in GSD arithmetic.

4 Residue Number Systems

Chapter Goals

Study a way of encoding large numbers
as a collection of smaller numbers
to simplify and speed up some operations

Chapter Highlights

RNS moduli, range, & arithmetic operations
Many sets of moduli are possible: tradeoffs
Conversions between RNS and binary
The Chinese remainder theorem
Why are RNS application domains limited?

Chapter Contents

4.1 RNS Representation and Arithmetic
4.2 Choosing the RNS Moduli
4.3 Encoding and Decoding of Numbers
4.4 Difficult RNS Arithmetic Operations
4.5 Redundant RNS Representations
4.6 Limits of Fast Arithmetic in RNS

4.1 RNS Representation and Arithmetic

Chinese puzzle, 1500 years ago:

What number has the remainders of 2, 3, and 2 when divided by the numbers 7, 5, and 3, respectively?

Pairwise relatively prime moduli: $m_{k-1} > \cdots > m_1 > m_0$

The residue x_i of x wrt the ith modulus m_i is akin to a digit:

$$x_i = x \bmod m_i = \langle x \rangle_{m_i}$$

RNS representation contains a list of k residues or digits:

$$x = (2 \mid 3 \mid 2)_{\mathrm{RNS}(7|5|3)}$$

Default RNS for this chapter RNS(8 | 7 | 5 | 3)

The product M of the k pairwise relatively prime moduli is the dynamic range

$$M = m_{k-1} \times \cdots \times m_1 \times m_0$$

For RNS(8 | 7 | 5 | 3), $M = 8 \times 7 \times 5 \times 3 = 840$

Negative numbers: Complement representation with complementation constant M

$$\langle -x \rangle_{m_i} = \langle M - x \rangle_{m_i}$$

$$21 = (5 \mid 0 \mid 1 \mid 0)_{\mathrm{RNS}}$$

$$-21 = (8-5 \mid 0 \mid 5-1 \mid 0)_{\mathrm{RNS}} = (3 \mid 0 \mid 4 \mid 0)_{\mathrm{RNS}}$$

Here are some example numbers in RNS(8 | 7 | 5 | 3):

$(0 \mid 0 \mid 0 \mid 0)_{RNS}$	Represents 0 or 840 or ···
$(1 \mid 1 \mid 1 \mid 1)_{RNS}$	Represents 1 or 841 or ···
$(2 \mid 2 \mid 2 \mid 2)_{RNS}$	Represents 2 or 842 or ···
$(0 \mid 1 \mid 3 \mid 2)_{RNS}$	Represents 8 or 848 or ···
$(5 \mid 0 \mid 1 \mid 0)_{RNS}$	Represents 21 or 861 or ···
$(0 \mid 1 \mid 4 \mid 1)_{RNS}$	Represents 64 or 904 or ···
$(2 \mid 0 \mid 0 \mid 2)_{RNS}$	Represents –70 or 770 or ···
$(7 \mid 6 \mid 4 \mid 2)_{RNS}$	Represents –1 or 839 or ···

Any RNS can be viewed as a weighted representation.

For RNS(8 | 7 | 5 | 3), the weights associated with the 4 positions are:

105 120 336 280

Example: $(1 \mid 2 \mid 4 \mid 0)_{RNS}$ represents the number

$$\langle 105\times1 + 120\times2 + 336\times4 + 280\times0 \rangle_{840} = \langle 1689 \rangle_{840} = 9$$

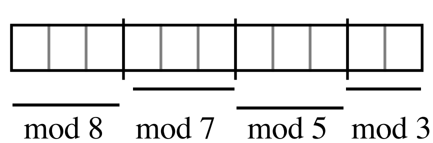


Fig. 4.1 Binary-coded format for RNS(8 | 7 | 5 | 3).

RNS Arithmetic

$(5 \mid 5 \mid 0 \mid 2)_{RNS}$	Represents $x = +5$
$(7 \mid 6 \mid 4 \mid 2)_{RNS}$	Represents $y = -1$
$(4 \mid 4 \mid 4 \mid 1)_{RNS}$	$x + y : \langle 5 + 7 \rangle_8 = 4$, $\langle 5 + 6 \rangle_7 = 4$, etc.
$(6 \mid 6 \mid 1 \mid 0)_{RNS}$	$x - y : \langle 5 - 7 \rangle_8 = 6$, $\langle 5 - 6 \rangle_7 = 6$, etc. (alternatively, find $-y$ and add to x)
$(3 \mid 2 \mid 0 \mid 1)_{RNS}$	$x \times y : \langle 5 \times 7 \rangle_8 = 3$, $\langle 5 \times 6 \rangle_7 = 2$, etc.

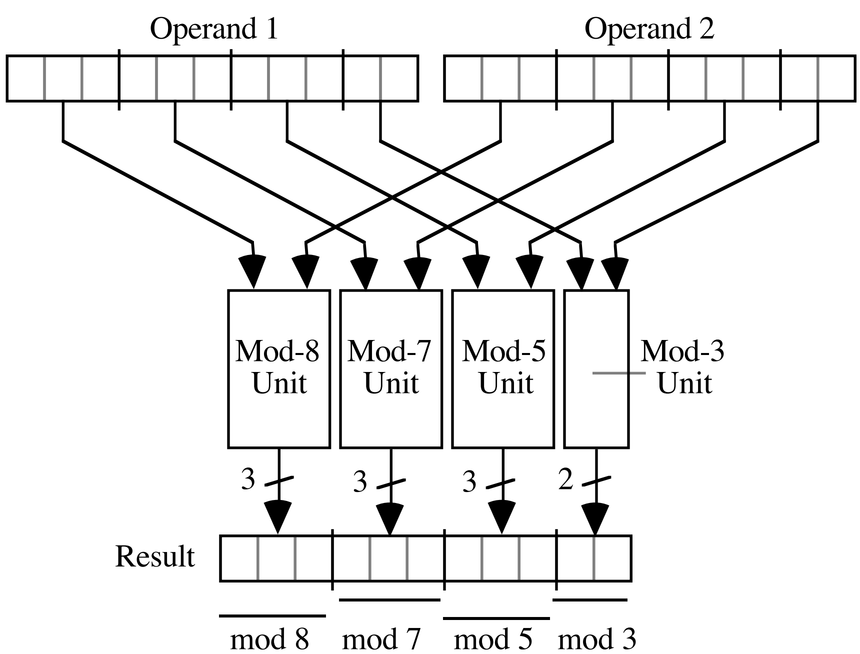


Fig. 4.2 The structure of an adder, subtractor, or multiplier for RNS(8 | 7 | 5 | 3).

4.2 Choosing the RNS moduli

Target range: Decimal values [0, 100 000]

Pick prime numbers in sequence:

$m_0 = 2$, $m_1 = 3$, $m_2 = 5$, etc. After adding $m_5 = 13$:

RNS(13 | 11 | 7 | 5 | 3 | 2)	M = 30 030	Inadequate
RNS(17 | 13 | 11 | 7 | 5 | 3 | 2)	M = 510 510	Too large
RNS(17 | 13 | 11 | 7 | 3 | 2)	M = 102 102	Just right!

5 + 4 + 4 + 3 + 2 + 1 = 19 bits

Combine pairs of moduli 2 & 13 and 3 & 7:

RNS(26 | 21 | 17 | 11) M = 102 102

Include powers of smaller primes before moving to larger primes.

RNS(2^2 | 3)	M = 12	
RNS(3^2 | 2^3 | 7 | 5)	M = 2520	
RNS(11 | 3^2 | 2^3 | 7 | 5)	M = 27 720	
RNS(13 | 11 | 3^2 | 2^3 | 7 | 5)	M = 360 360	Too large
RNS(15 | 13 | 11 | 2^3 | 7)	M = 120 120	

4 + 4 + 4 + 3 + 3 = 18 bits

Maximize the size of the even modulus within the 4-bit residue limit:

RNS(2^4 | 13 | 11 | 3^2 | 7 | 5) M = 720 720 Too large
Can remove 5 or 7

Restrict the choice to moduli of the form 2^a or $2^a - 1$:

$$\text{RNS}(2^{a_{k-2}} \mid 2^{a_{k-2}} - 1 \mid \cdots \mid 2^{a_1} - 1 \mid 2^{a_0} - 1)$$

Such "low-cost" moduli simplify both the complementation and modulo operations

2^{a_i} and 2^{a_j} are relatively prime iff a_i and a_j are relatively prime.

RNS($2^3 \mid 2^3-1 \mid 2^2-1$)	basis: 3, 2	M = 168
RNS($2^4 \mid 2^4-1 \mid 2^3-1$)	basis: 4, 3	M = 1680
RNS($2^5 \mid 2^5-1 \mid 2^3-1 \mid 2^2-1$)	basis: 5, 3, 2	M = 20 832
RNS($2^5 \mid 2^5-1 \mid 2^4-1 \mid 2^4-1$)	basis: 5, 4, 3	M = 104 160

Comparison

RNS($15 \mid 13 \mid 11 \mid 2^3 \mid 7$)	18 bits	M = 120 120
RNS($2^5 \mid 2^5-1 \mid 2^4-1 \mid 2^3-1$)	17 bits	M = 104 160

4.3 Encoding and Decoding of Numbers

Conversion from binary/decimal to RNS

$$\langle (y_{k-1} \cdots y_1 y_0)_{\text{two}} \rangle_{m_i} = \langle \langle 2^{k-1} y_{k-1} \rangle_{m_i} + \cdots + \langle 2y_1 \rangle_{m_i} + \langle y_0 \rangle_{m_i} \rangle_{m_i}$$

Table 4.1 Residues of the first 10 powers of 2

i	2^i	$\langle 2^i \rangle_7$	$\langle 2^i \rangle_5$	$\langle 2^i \rangle_3$
0	1	1	1	1
1	2	2	2	2
2	4	4	4	1
3	8	1	3	2
4	16	2	1	1
5	32	4	2	2
6	64	1	4	1
7	128	2	3	2
8	256	4	1	1
9	512	1	2	2

High-radix version (processing 2 or more bits at a time) is also possible

Conversion from RNS to mixed-radix

$MRS(m_{k-1} \mid \cdots \mid m_2 \mid m_1 \mid m_0)$ is a k-digit positional system with position weights

$m_{k-2} \cdots m_2 m_1 m_0 \quad \cdots \quad m_2 m_1 m_0 \quad m_1 m_0 \quad m_0 \quad 1$

and digit sets

$[0, m_{k-2}-1] \quad \cdots \quad [0, m_2-1] \; [0, m_1-1] \; [0, m_0-1]$

$(0 \mid 3 \mid 1 \mid 0)_{MRS(8|7|5|3)} = 0 \times 105 + 3 \times 15 + 1 \times 3 + 0 \times 1 = 48$

RNS-to-MRS conversion problem:

$y = (x_{k-1} \mid \cdots \mid x_2 \mid x_1 \mid x_0)_{RNS} = (z_{k-1} \mid \cdots \mid z_2 \mid z_1 \mid z_0)_{MRS}$

Mixed-radix representation allows us to compare the magnitudes of two RNS numbers or to detect the sign of a number.

Example: 48 versus 45

RNS representations

$(0 \mid 6 \mid 3 \mid 0)_{RNS}$ vs $(5 \mid 3 \mid 0 \mid 0)_{RNS}$

$(000 \mid 110 \mid 011 \mid 00)_{RNS}$ vs $(101 \mid 011 \mid 000 \mid 00)_{RNS}$

Equivalent mixed-radix representations

$(0 \mid 3 \mid 1 \mid 0)_{MRS}$ vs $(0 \mid 3 \mid 0 \mid 0)_{MRS}$

$(000 \mid 011 \mid 001 \mid 00)_{MRS}$ vs $(000 \mid 011 \mid 000 \mid 00)_{MRS}$

Theorem 4.1 (The Chinese remainder theorem)

The magnitude of an RNS number can be obtained from:

$$x = (x_{k-1} \mid \cdots \mid x_2 \mid x_1 \mid x_0)_{RNS} = \langle \sum_{i=0}^{k-1} M_i \langle \alpha_i x_i \rangle_{m_i} \rangle_M$$

where, by definition, $M_i = M/m_i$ and $\alpha_i = \langle M_i^{-1} \rangle_{m_i}$ is the multiplicative inverse of M_i with respect to m_i

Table 4.2 Values needed in applying the Chinese remainder theorem to RNS(8 | 7 | 5 | 3)

i	m_i	x_i	$\langle M_i \langle \alpha_i x_i \rangle_{m_i} \rangle_M$
3	8	0	0
		1	105
		2	210
		3	315
		4	420
		5	525
		6	630
		7	735
2	7	0	0
		1	120
		2	240
		3	360
		4	480
		5	600
		6	720
1	5	0	0
		1	336
		2	672
		3	168
		4	504
0	3	0	0
		1	280
		2	560

4.4 Difficult RNS Arithmetic Operations

Sign test and magnitude comparison are difficult

Example: of the following RNS(8 | 7 | 5 | 3) numbers

which, if any, are negative?

which is the largest?

which is the smallest?

Assume a range of [–420, 419]

$a = (0 \mid 1 \mid 3 \mid 2)_{RNS}$

$b = (0 \mid 1 \mid 4 \mid 1)_{RNS}$

$c = (0 \mid 6 \mid 2 \mid 1)_{RNS}$

$d = (2 \mid 0 \mid 0 \mid 2)_{RNS}$

$e = (5 \mid 0 \mid 1 \mid 0)_{RNS}$

$f = (7 \mid 6 \mid 4 \mid 2)_{RNS}$

Answer: $d = -70 < c = -8 < f = -1 < a = 8 < e = 21 < b = 64$

Approximate CRT decoding: Divide both sides of the CRT equality by M, we obtain the scaled value of x in [0, 1):

$$x/M = (x_{k-1} \mid \cdots \mid x_2 \mid x_1 \mid x_0)_{RNS}/M = \left\langle \sum_{i=0}^{k-1} m_i^{-1} \langle \alpha_i x_i \rangle_{m_i} \right\rangle_1$$

Terms are added modulo 1, meaning that the whole part of each result is discarded and only the fractional part is kept.

Table 4.3 Values needed in applying approximate CRT decoding to RNS(8 | 7 | 5 | 3).

i	m_i	x_i	$m_i^{-1}\langle \alpha_i x_i \rangle_{m_i}$
3	8	0	.0000
		1	.1250
		2	.2500
		3	.3750
		4	.5000
		5	.6250
		6	.7500
		7	.8750
2	7	0	.0000
		1	.1429
		2	.2857
		3	.4286
		4	.5714
		5	.7143
		6	.8571
1	5	0	.0000
		1	.4000
		2	.8000
		3	.2000
		4	.6000
0	3	0	.0000
		1	.3333
		2	.6667

Example: Use approximate CRT decoding to determine the larger of the two numbers

$$x = (0 \mid 6 \mid 3 \mid 0)_{\text{RNS}} \qquad y = (5 \mid 3 \mid 0 \mid 0)_{\text{RNS}}$$

Reading values from Table 4.3, we get:

$$x/M \approx \langle .0000 + .8571 + .2000 + .0000 \rangle_1 \approx .0571$$

$$y/M \approx \langle .6250 + .4286 + .0000 + .0000 \rangle_1 \approx .0536$$

Thus, $x > y$, subject to approximation errors.

Errors are no problem here because each entry has a maximum error of 0.00005, for a total of at most 0.0002

RNS general division

Use an algorithm that has built-in tolerance to imprecision

Example — SRT algorithm (s is the partial remainder)

$s < 0$	quotient digit $= -1$
$s \approx 0$	quotient digit $= 0$
$s > 0$	quotient digit $= 1$

The partial remainder is decoded approximately
The BSD quotient is converted to RNS on the fly

4.5 Redundant RNS Representations

The mod-m_i residue need not be restricted to $[0, m_i - 1]$
(just as radix-r digits need not be limited to $[0, r - 1]$)

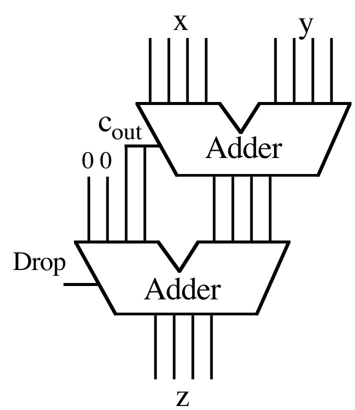


Figure 4.3 Adder design for 4-bit mod-13 pseudo-residues.

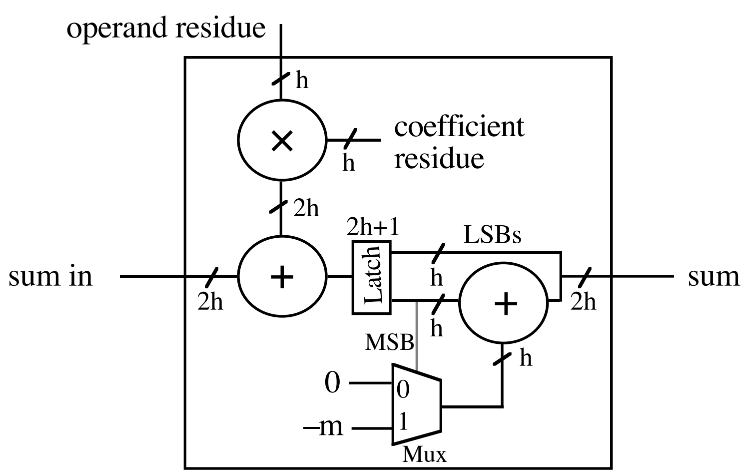


Figure 4.4 A modulo-m multiply-add cell that accumulates the sum into a double-length redundant pseudo-residue.

4.6 Limits of Fast Arithmetic in RNS

Theorem 4.2: The ith prime p_i is asymptotically i ln i

Theorem 4.3: The number of primes in [1, n] is asymptotically n/ln n

Theorem 4.4: The product of all primes in [1, n] is asymptotically e^n.

Table 4.4 The ith prime p_i and the number of primes in [1, n] versus their asymptotic approximations

i	p_i	i ln i	Error (%)	n	No of primes	n/ln n	Error (%)
1	2	0.000	100	5	2	3.107	55
2	3	1.386	54	10	4	4.343	9
3	5	3.296	34	15	6	5.539	8
4	7	5.545	21	20	8	6.676	17
5	11	8.047	27	25	9	7.767	14
10	29	23.03	21	30	10	8.820	12
15	47	40.62	14	40	12	10.84	10
20	71	59.91	16	50	15	12.78	15
30	113	102.0	10	100	25	21.71	13
40	173	147.6	15	200	46	37.75	18
50	229	195.6	15	500	95	80.46	15
100	521	460.5	12	1000	170	144.8	15

Theorem 4.5: It is possible to represent all k-bit binary numbers in RNS with $O(k / \log k)$ moduli such that the largest modulus has $O(\log k)$ bits

Implication: a fast adder would need $O(\log \log k)$ time

Theorem 4.6: The numbers $2^a - 1$ and $2^b - 1$ are relatively prime iff a and b are relatively prime

Theorem 4.7: The sum of the first i primes is asymptotically $O(i^2 \ln i)$.

Theorem 4.8: It is possible to represent all k-bit binary numbers in RNS with $O(\sqrt{k / \log k})$ low-cost moduli of the form $2^a - 1$ such that the largest modulus has $O(\sqrt{k \log k})$ bits.

Implication: a fast adder would need $O(\log k)$ time, thus offering little advantage over standard binary

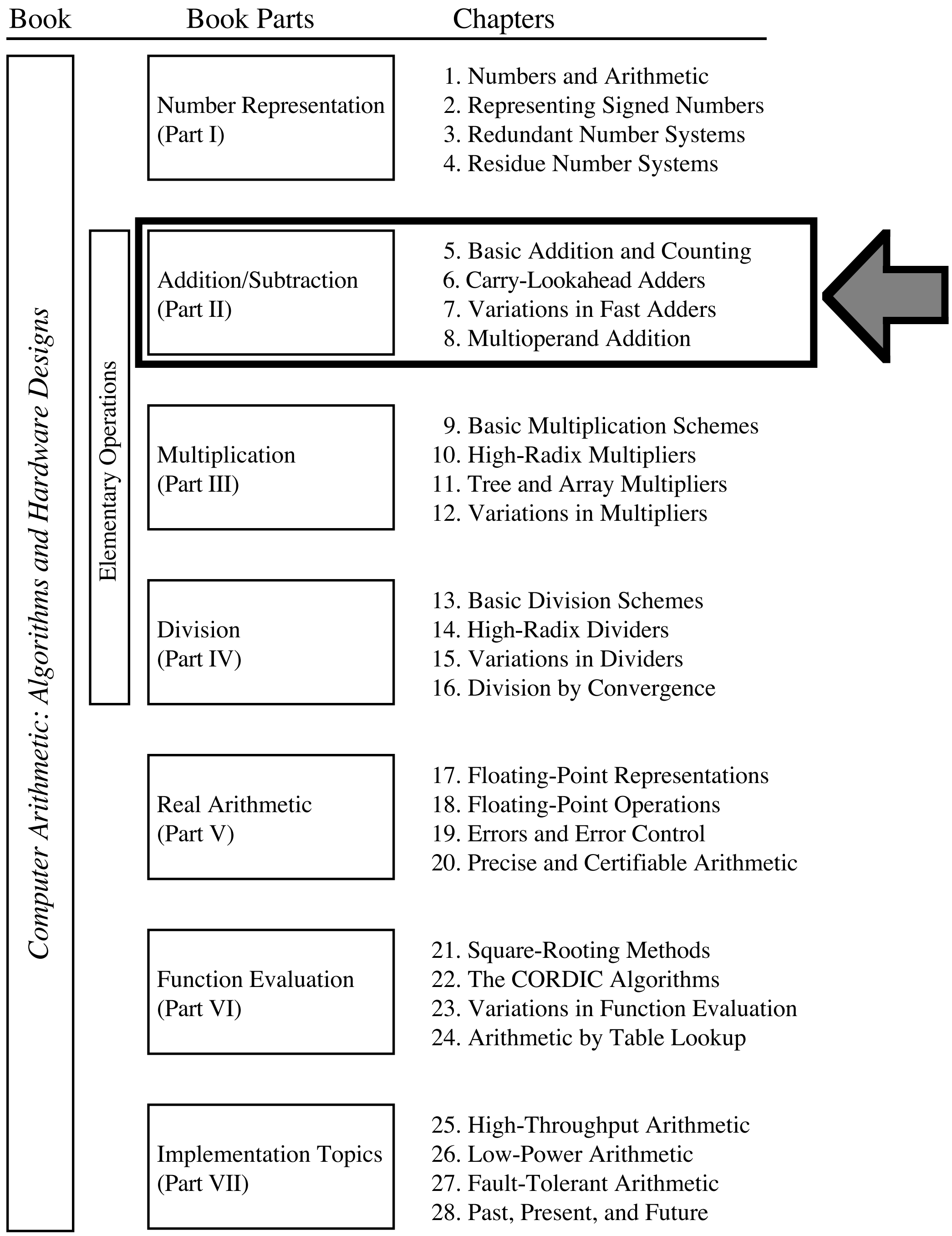


Fig. P.1 The structure of this book in parts and chapters.

Part II Addition/Subtraction

Part Goals

Review basic adders and the carry problem
Learn how to speed up carry propagation
Discuss speed/cost tradeoffs in fast adders

Part Synopsis

Addition is a fundamental operation
(arithmetic and address calculations)
Also a building block for other operations
Subtraction = negation + addition
Carry speedup: lookahead, skip, select, ...
Two-operand vs multioperand addition

Part Contents

Chapter 5 Basic Addition and Counting
Chapter 6 Carry-Lookahead Adders
Chapter 7 Variations in Fast Adders
Chapter 8 Multioperand Addition

5 Basic Addition and Counting

Chapter Goals

Study the design of ripple-carry adders,
discuss why their latency is unacceptable,
and set the foundation for faster adders

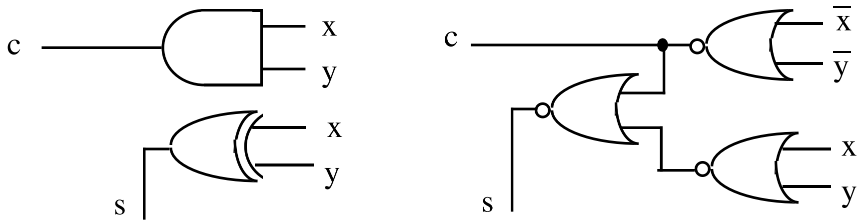
Chapter Highlights

Full-adders are versatile building blocks
Worst-case carry chain in k-bit addition
has an average length of $\log_2 k$
Fast asynchronous adders are simple
Counting is relatively easy to speed up

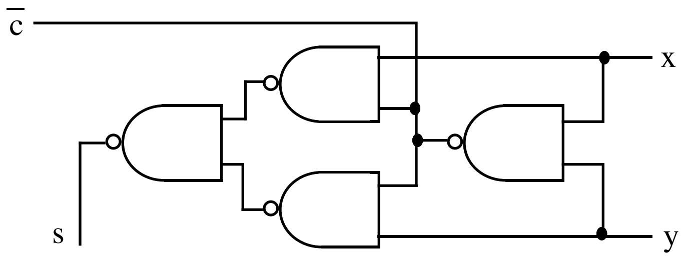
Chapter Contents

5.1 Bit-Serial and Ripple-Carry Adders
5.2 Conditions and Exceptions
5.3 Analysis of Carry Propagation
5.4 Carry Completion Detection
5.5 Addition of a Constant: Counters
5.6 Manchester Carry Chains and Adders

5.1 Bit-serial and ripple-carry adders



(a) AND/XOR half-adder. (b) NOR-gate half-adder.



(c) NAND-gate half-adder with complemented carry.

Fig. 5.1 Three implementations of a half-adder.

Single-bit full-adder (FA)

Inputs

Operand bits x, y and carry-in c_{in} (or x_i, y_i, c_i for stage i)

Outputs

Sum bit s and carry-out c_{out} (or c_{i+1} for stage i)

$$s = x \oplus y \oplus c_{in} \quad \text{(odd parity function)}$$

$$= x\, y\, c_{in} + \bar{x}\, \bar{y}\, c_{in} + \bar{x}\, y\, \bar{c}_{in} + x\, \bar{y}\, \bar{c}_{in}$$

$$c_{out} = x\, y + x\, c_{in} + y\, c_{in} \quad \text{(majority function)}$$

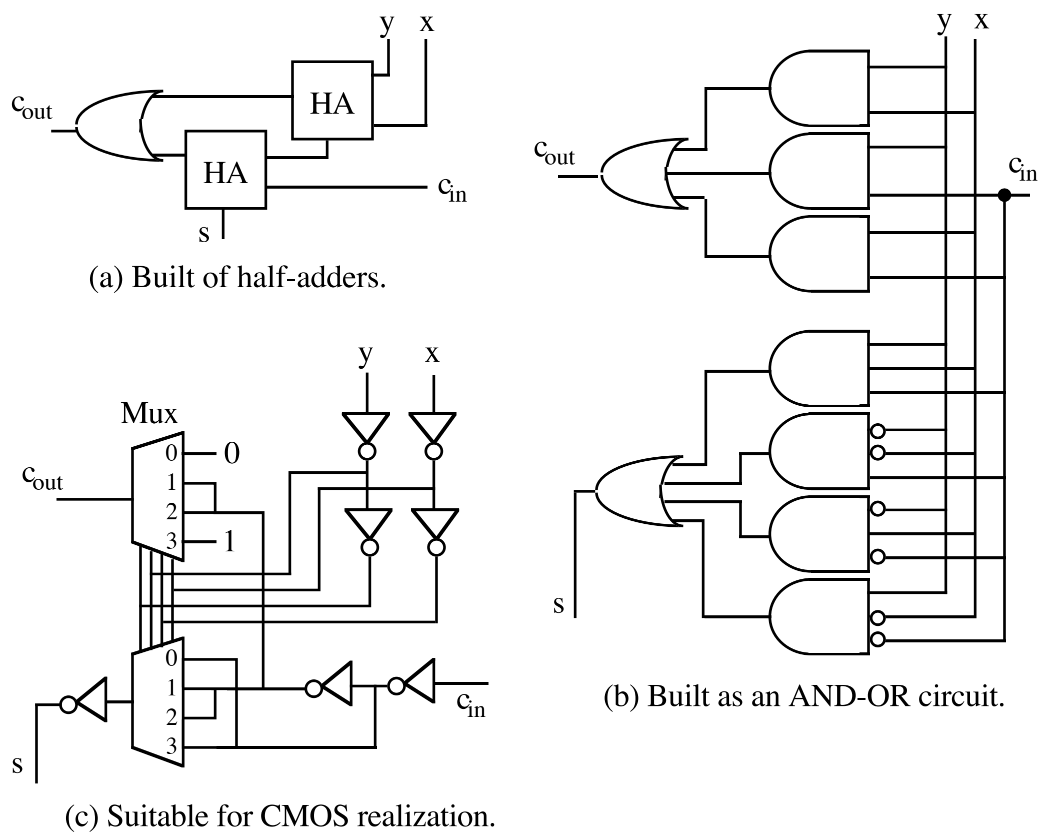
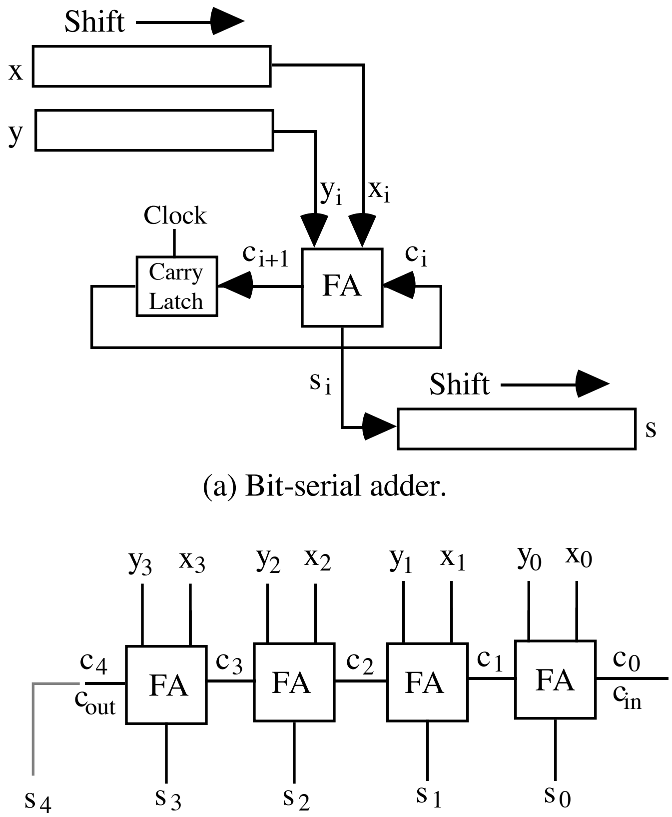


Fig. 5.2 Possible designs for a full-adder in terms of half-adders, logic gates, and CMOS transmission gates.



(a) Bit-serial adder.

(b) Four-bit ripple-carry adder.

Fig. 5.3 Using full-adders in building bit-serial and ripple-carry adders.

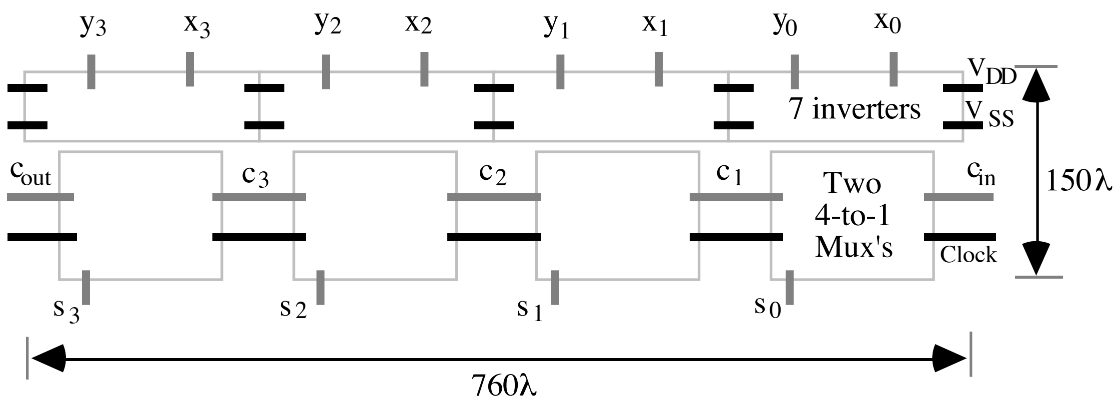


Fig. 5.4 The layout of a 4-bit ripple-carry adder in CMOS implementation [Puck94].

$$T_{\text{ripple-add}} = T_{FA}(x,y \to c_{out}) + (k-2) \times T_{FA}(c_{in} \to c_{out}) + T_{FA}(c_{in} \to s)$$

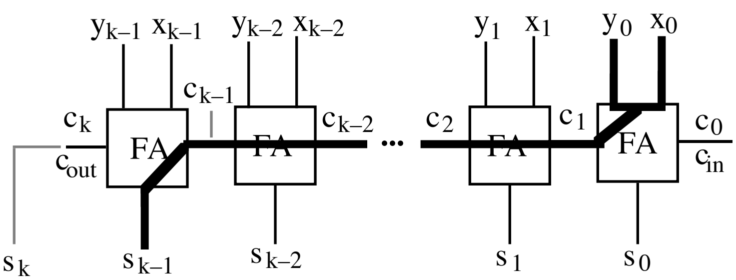


Fig. 5.5 Critical path in a k-bit ripple-carry adder.

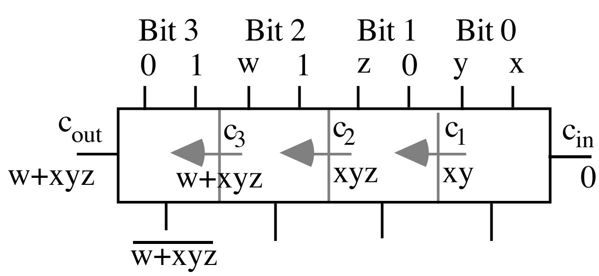


Fig. 5.6 Four-bit binary adder used to realize the logic function f = w + xyz and its complement.

5.2 Conditions and exceptions

In an ALU, it is customary to provide information about certain outcomes during addition or other operations

Flag bits within a condition/exception register:

c_{out}	a carry-out of 1 is produced
overflow	the output is not the correct sum
negative	Indicating that the addition result is negative
zero	Indicating that the addition result is zero

$$\text{overflow}_{2\text{'s-compl}} = x_{k-1}\, y_{k-1}\, \bar{s}_{k-1} + \bar{x}_{k-1}\, \bar{y}_{k-1}\, s_{k-1}$$
$$\text{overflow}_{2\text{'s-compl}} = c_k \oplus c_{k-1} = c_k\, \bar{c}_{k-1} + \bar{c}_k\, c_{k-1}$$

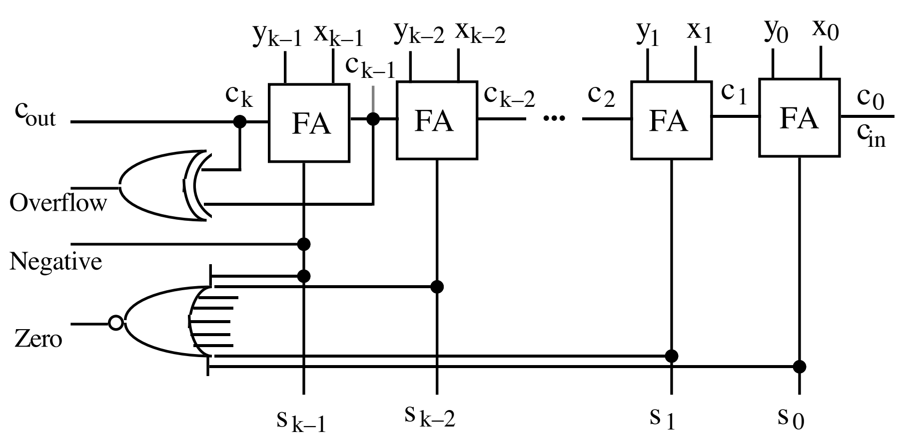


Fig. 5.7 Two's-complement adder with provisions for detecting conditions and exceptions.

5.3 Analysis of carry propagation

```
Bit # 15 14 13 12  11 10  9  8   7  6  5  4   3  2  1  0
      -----------  -----------  -----------  -----------
       1  0  1  1   0  1  1  0   0  1  1  0   1  1  1  0
cout   0  1  0  1   1  0  0  1   1  1  0  0   0  0  1  1  cin
      \__________/\__________________/      \_________/\____/
          4                6                     3       2
```

Carry chains and their lengths

Fig. 5.8 Example addition and its carry propagation chains.

Given binary numbers with random bit values, for each position i we have:

Probability of carry generation = 1/4

Probability of carry annihilation = 1/4

Probability of carry propagation = 1/2

The probability that carry generated at position i propagates up to and including position j – 1 and stops at position j (j > i)

$$2^{-(j-1-i)} \times 1/2 = 2^{-(j-i)}$$

Expected length of the carry chain that starts at bit position i

$$2 - 2^{-(k-i-1)}$$

Average length of the longest carry chain in k-bit addition is less than $\log_2 k$; it is $\log_2(1.25k)$ per experimental results

5.4 Carry Completion Detection

(b_i, c_i) = 00 Carry not yet known

01 Carry known to be 1

10 Carry known to be 0

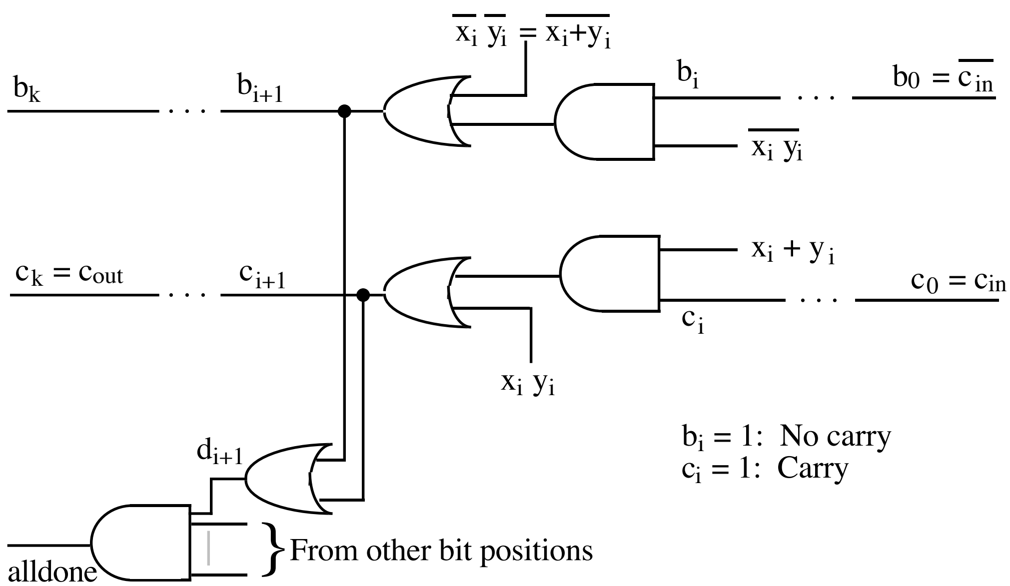


Fig. 5.9 The carry network of an adder with two-rail carries and carry completion detection logic.

5.4 Addition of a Constant: Counters

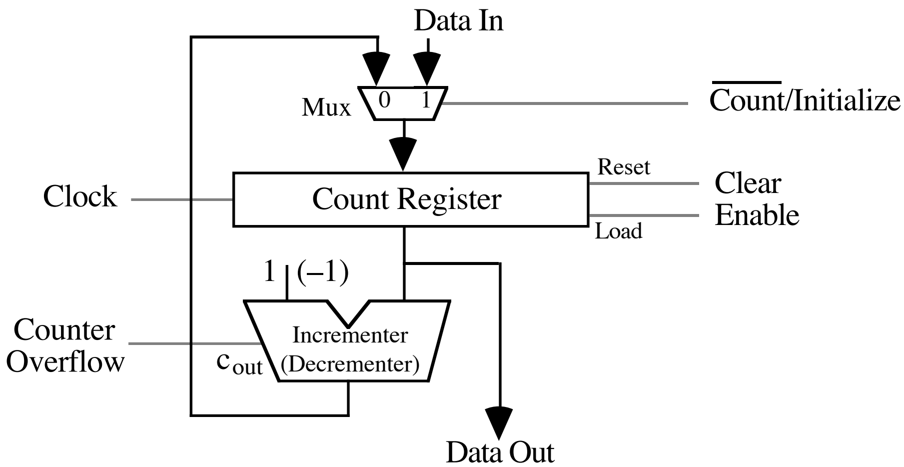


Fig. 5.10 An up (down) counter built of a register, an incrementer (decrementer), and a multiplexer.

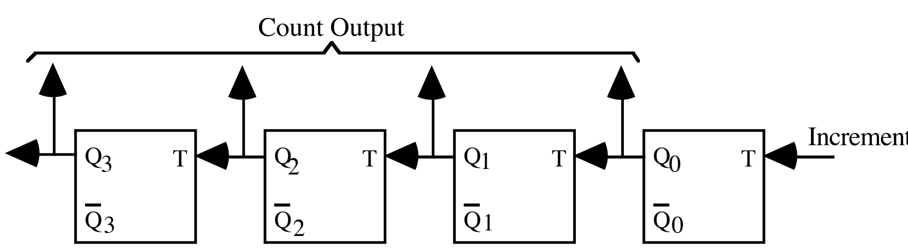


Fig. 5.11 Four-bit asynchronous up counter built only of negative-edge-triggered T flip-flops.

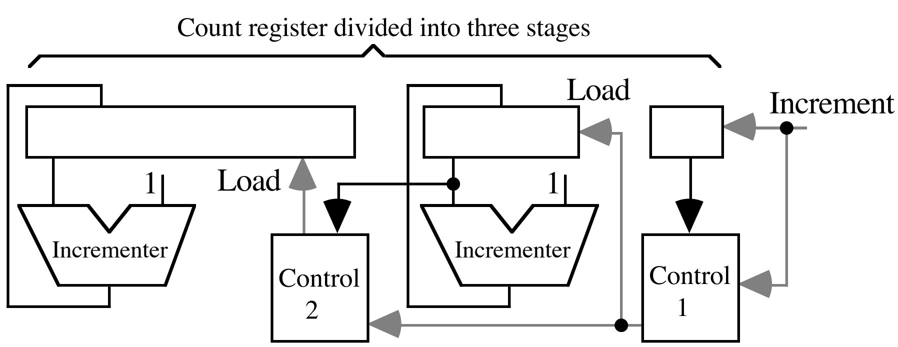


Fig. 5.12 Fast three-stage up counter.

5.6 Manchester Carry Chains and Adders

Sum digit in radix r $\quad s_i \quad = \quad (x_i + y_i + c_i) \bmod r$

Special case of radix 2 $\quad s_i \quad = \quad x_i \oplus y_i \oplus c_i$

Computing the carries is thus our central problem
For this, the actual operand digits are not important
What matters is whether in a given position a carry is
generated, propagated, or annihilated (absorbed)
For binary addition:

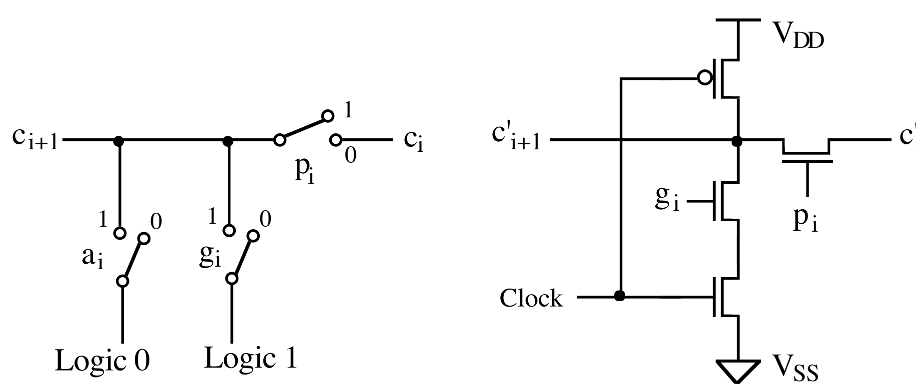
$$g_i = x_i y_i \qquad p_i = x_i \oplus y_i \qquad a_i = \bar{x}_i \bar{y}_i = \overline{x_i + y_i}$$

It is also helpful to define a transfer signal:

$$t_i = g_i + p_i = \bar{a}_i = x_i + y_i$$

Using these signals, the carry recurrence can be written as

$$c_{i+1} = g_i + c_i p_i = g_i + c_i g_i + c_i p_i = g_i + c_i t_i$$



(a) Conceptual representation (b) Possible CMOS realization.

Fig. 5.13 One stage in a Manchester carry chain.

6 Carry-Lookahead Adders

Chapter Goals

Understand the carry-lookahead method
and its many variations
used in the design of fast adders

Chapter Highlights

Single- and multilevel carry lookahead
Various designs for logarithmic-time adders
Relating the carry determination problem
to parallel prefix computation
Implementing fast adders in VLSI

Chapter Contents

6.1. Unrolling the Carry Recurrence
6.2. Carry-Lookahead Adder Design
6.3. Ling Adder and Related Designs
6.4. Carry Determination as Prefix Computation
6.5. Alternative Parallel Prefix Networks
6.6. VLSI Implementation Aspects

6.1 Unrolling the Carry Recurrence

Recall g_i (generate), p_i (propagate), a_i (annihilate/absorb), and t_i (transfer)

$g_i = 1$	iff $x_i + y_i \geq r$	Carry is generated
$p_i = 1$	iff $x_i + y_i = r - 1$	Carry is propagated
$t_i = \bar{a}_i = g_i + p_i$		Carry is not annihilated

These signals, along with the carry recurrence

$$c_{i+1} = g_i + c_i p_i \;=\; g_i + c_i t_i$$

allow us to decouple the problem of designing a fast carry network from details of the number system (radix, digit set)

It does not even matter whether we are adding or subtracting; any carry network can be used as a borrow network by defining the signals to represent borrow generation, borrow propagation, etc.

$$\begin{aligned}
c_i &= g_{i-1} + c_{i-1}p_{i-1} \\
&= g_{i-1} + (g_{i-2} + c_{i-2}p_{i-2})p_{i-1} \\
&= g_{i-1} + g_{i-2}p_{i-1} + c_{i-2}p_{i-2}p_{i-1} \\
&= g_{i-1} + g_{i-2}p_{i-1} + g_{i-3}p_{i-2}p_{i-1} + c_{i-3}p_{i-3}p_{i-2}p_{i-1} \\
&= g_{i-1} + g_{i-2}p_{i-1} + g_{i-3}p_{i-2}p_{i-1} + g_{i-4}p_{i-3}p_{i-2}p_{i-1} \\
&\qquad + c_{i-4}p_{i-4}p_{i-3}p_{i-2}p_{i-1} \\
&= \quad \ldots
\end{aligned}$$

Four-bit CLA adder:

$$c_4 = g_3 + g_2p_3 + g_1p_2p_3 + g_0p_1p_2p_3 + c_0p_0p_1p_2p_3$$
$$c_3 = g_2 + g_1p_2 + g_0p_1p_2 + c_0p_0p_1p_2$$
$$c_2 = g_1 + g_0p_1 + c_0p_0p_1$$
$$c_1 = g_0 + c_0p_0$$

Note the use of $c_4 = g_3 + c_3p_3$ in the following diagram

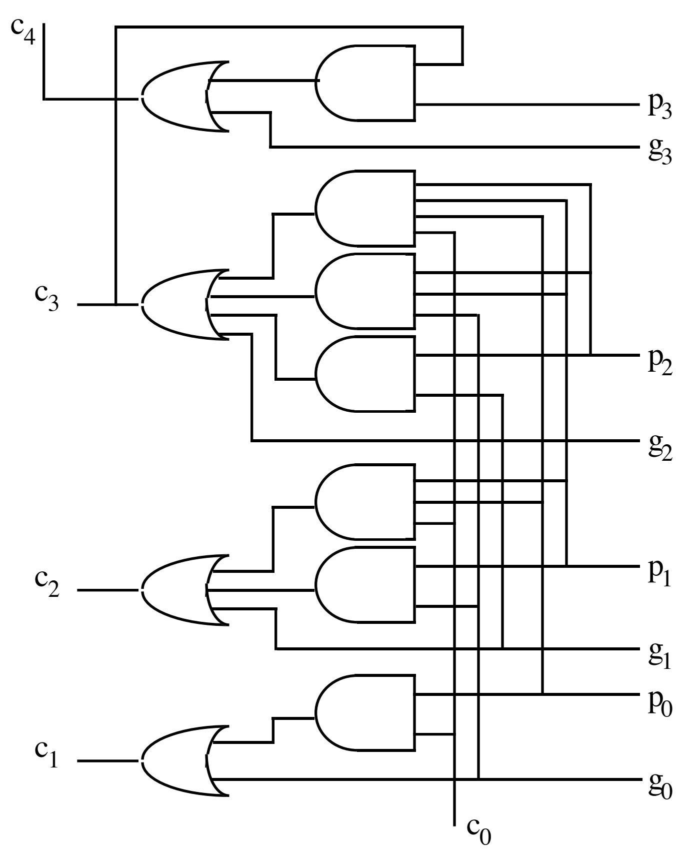


Fig. 6.1 Four-bit carry network with full lookahead.

Full carry lookahead is impractical for wide words

The fully unrolled carry equation for c_{31} consists of 32 product terms, the largest of which contains 32 literals

Thus, the required AND and OR functions must be realized by tree networks, leading to increased latency and cost

Two schemes for managing this complexity:

High-radix addition (i.e., radix 2^g)

increases the latency for generating

the auxiliary signals and sum digits

but simplifies the carry network (optimal radix?)

Multilevel lookahead

Example: 16-bit addition

Radix-16 (four digits)

Two-level carry lookahead (four 4-bit blocks)

Either way, the carries c_4, c_8, and c_{12} are determined first

c_{16}	c_{15}	c_{14}	c_{13}	c_{12}	c_{11}	c_{10}	c_9	c_8	c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0
c_{out}				?				?				?				c_{in}

6.2 Carry-Lookahead Adder Design

$$g_{[i,i+3]} = g_{i+3} + g_{i+2}p_{i+3} + g_{i+1}p_{i+2}p_{i+3} + g_ip_{i+1}p_{i+2}p_{i+3}$$

$$p_{[i,i+3]} = p_i\ p_{i+1}\ p_{i+2}\ p_{i+3}$$

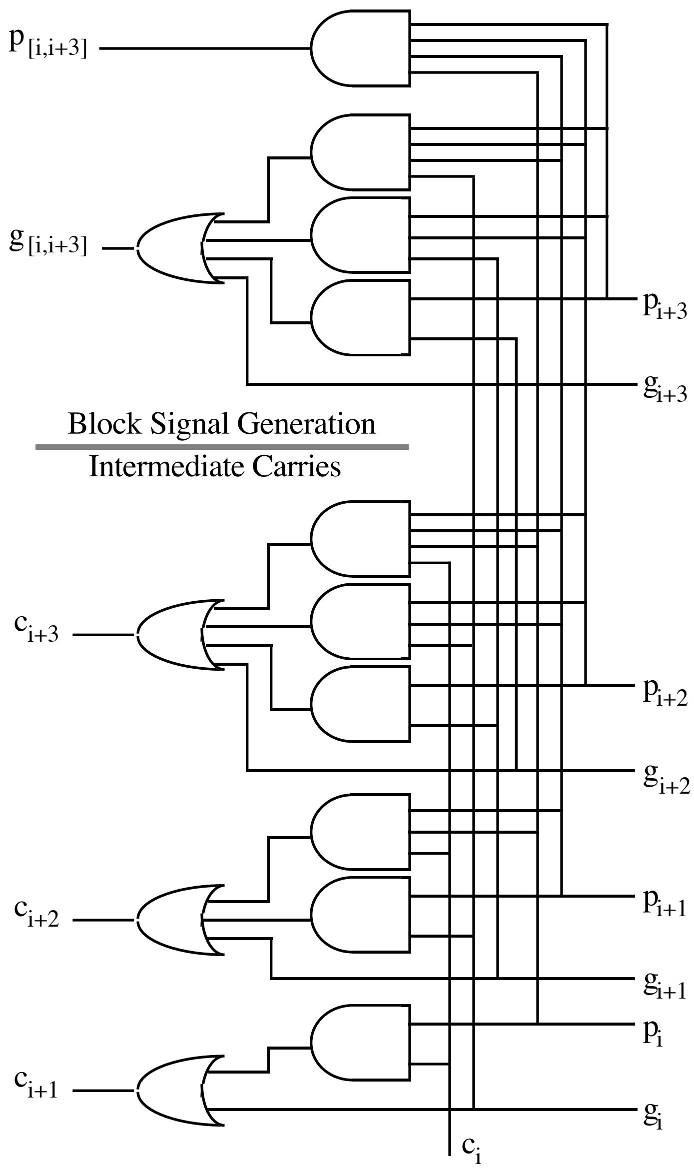


Fig. 6.2 Four-bit lookahead carry generator.

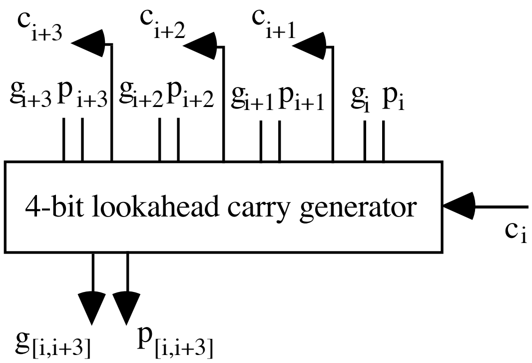


Fig. 6.3 Schematic diagram of a 4-bit lookahead carry generator.

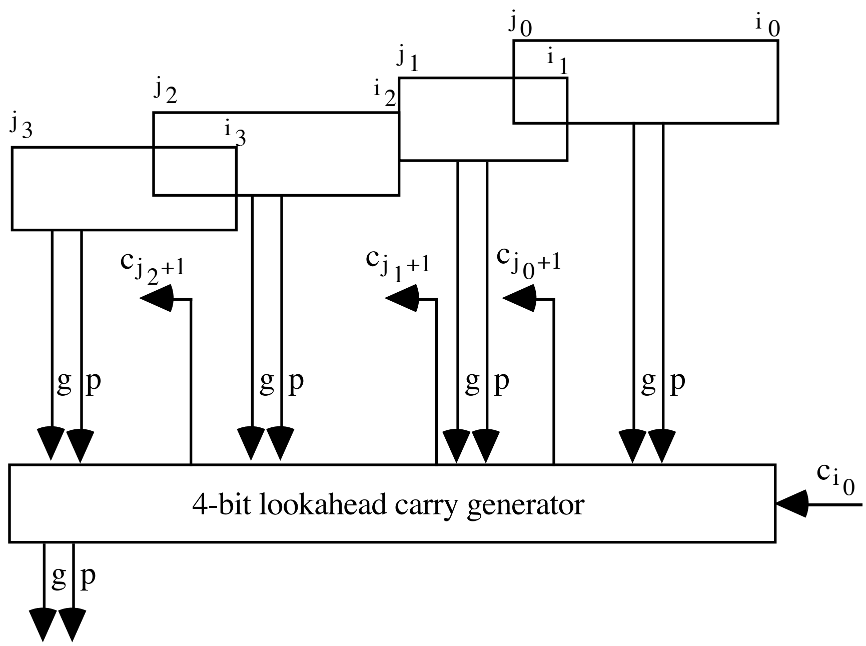


Fig. 6.4 Combining of g and p signals of four (contiguous or overlapping) blocks of arbitrary widths into the g and p signals for the overall block $[i_0, j_3]$.

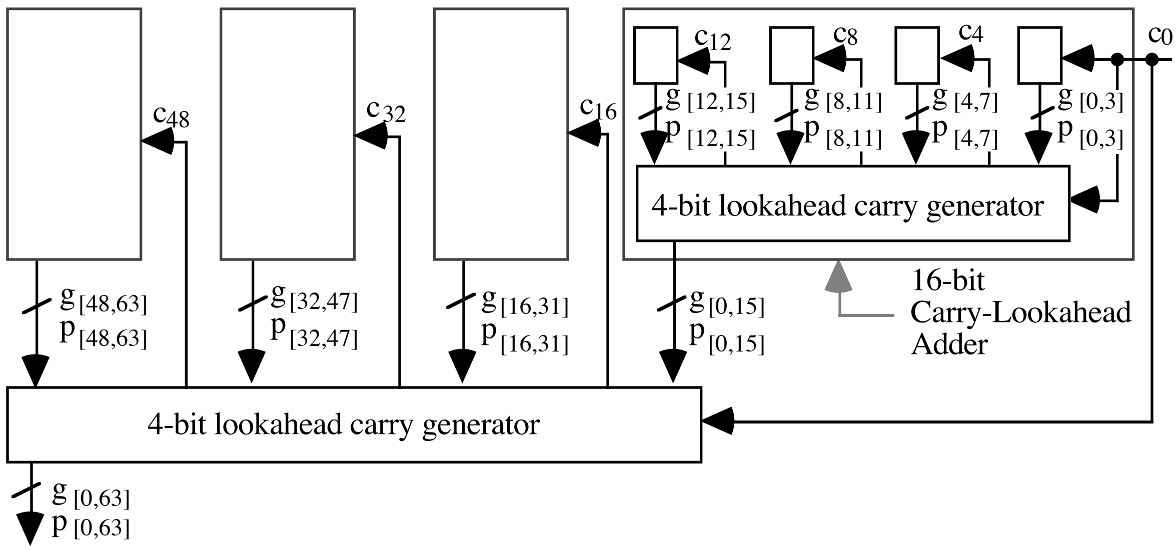


Fig. 6.5 Building a 64-bit carry-lookahead adder from 16 4-bit adders and 5 lookahead carry generators.

Latency through the 16-bit CLA adder consists of finding:

g and p for individual bit positions (1 gate level)

g and p signals for 4-bit blocks (2 gate levels)

block carry-in signals c_4, c_8, and c_{12} (2 gate levels)

internal carries within 4-bit blocks (2 gate levels)

sum bits (2 gate levels)

Total latency for the 16-bit adder = 9 gate levels

(compare to 32 gate levels for a 16-bit ripple-carry adder)

$$T_{\text{lookahead-add}} = 4 \log_4 k + 1 \text{ gate levels}$$

$$c_{out} = x_{k-1} y_{k-1} + \bar{s}_{k-1}(x_{k-1} + y_{k-1})$$

6.3 Ling Adder and Related Designs

Consider the carry recurrence and its unrolling by 4 steps:

$$c_i = g_{i-1} + g_{i-2}t_{i-1} + g_{i-3}t_{i-2}t_{i-1} + g_{i-4}t_{i-3}t_{i-2}t_{i-1} + c_{i-4}t_{i-4}t_{i-3}t_{i-2}t_{i-1}$$

Ling's modification:

propagate $h_i = c_i + c_{i-1}$ instead of c_i

$$h_i = g_{i-1} + g_{i-2} + g_{i-3}\,t_{i-2} + g_{i-4}\,t_{i-3}\,t_{i-2} + h_{i-4}\,t_{i-4}\,t_{i-3}\,t_{i-2}$$

CLA:	5 gates	max 5 inputs	19 gate inputs
Ling:	4 gates	max 5 inputs	14 gate inputs

The advantage of h_i over c_i is even greater with wired-OR:

CLA:	4 gates	max 5 inputs	14 gate inputs
Ling:	3 gates	max 4 inputs	9 gate inputs

Once h_i is known, however, the sum is obtained by a slightly more complex expression compared to $s_i = p_i \oplus c_i$

$$s_i = (t_i \oplus h_{i+1}) + h_i\,g_i\,t_{i-1}$$

Other designs similar to Ling's are possible [Dora88]

6.4 Carry Determination as Prefix Computation

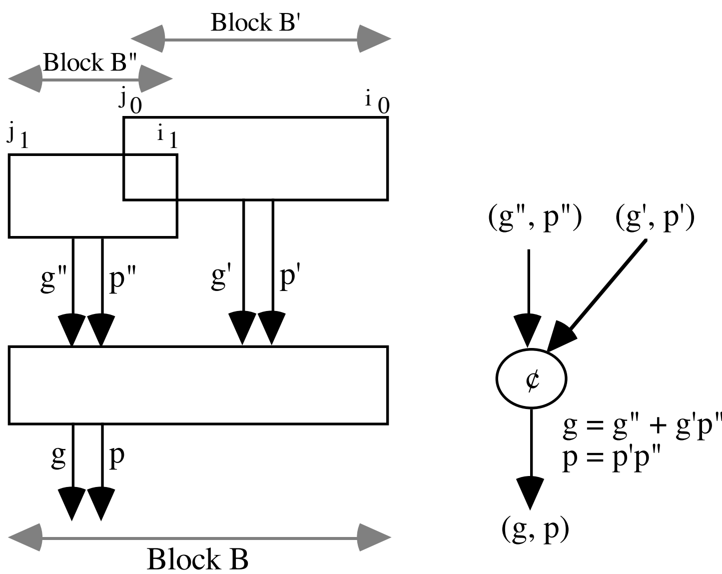


Fig. 6.6 Combining of g and p signals of two (contiguous or overlapping) blocks B' and B" of arbitrary widths into the g and p signals for block B.

The problem of carry determination can be formulated as:

Given (g_0, p_0) (g_1, p_1) $\cdots$ (g_{k-2}, p_{k-2}) (g_{k-1}, p_{k-1})

Find $(g_{[0,0]}, p_{[0,0]})$ $(g_{[0,1]}, p_{[0,1]})$ $\cdots$ $(g_{[0,k-2]}, p_{[0,k-2]})$ $(g_{[0,k-1]}, p_{[0,k-1]})$

The desired pairs can be found by evaluating all prefixes of

$$(g_0, p_0)\ ¢\ (g_1, p_1)\ ¢\ \cdots\ ¢\ (g_{k-2}, p_{k-2})\ ¢\ (g_{k-1}, p_{k-1})$$

Prefix sums analogy:

Given x_0 x_1 x_2 $\cdots$ x_{k-1}

Find x_0 x_0+x_1 $x_0+x_1+x_2$ $\cdots$ $x_0+x_1+\cdots+x_{k-1}$

6.5 Alternative Parallel Prefix Networks

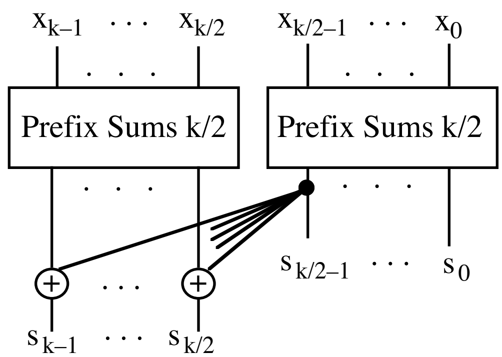


Fig. 6.7 Parallel prefix sums network built of two k/2-input networks and k/2 adders.

Delay recurrence $D(k) = D(k/2) + 1 = \log_2 k$

Cost recurrence $C(k) = 2C(k/2) + k/2 = (k/2) \log_2 k$

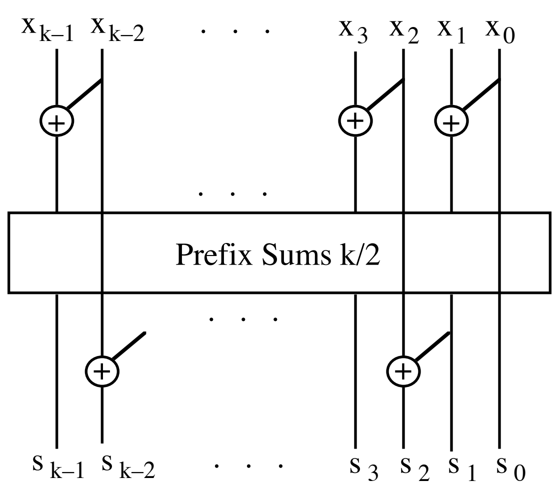


Fig. 6.8 Parallel prefix sums network built of one k/2-input network and k – 1 adders.

Delay recurrence $D(k) = D(k/2) + 2 = 2 \log_2 k - 1$ (–2 really)

Cost recurrence $C(k) = C(k/2) + k - 1 = 2k - 2 - \log_2 k$

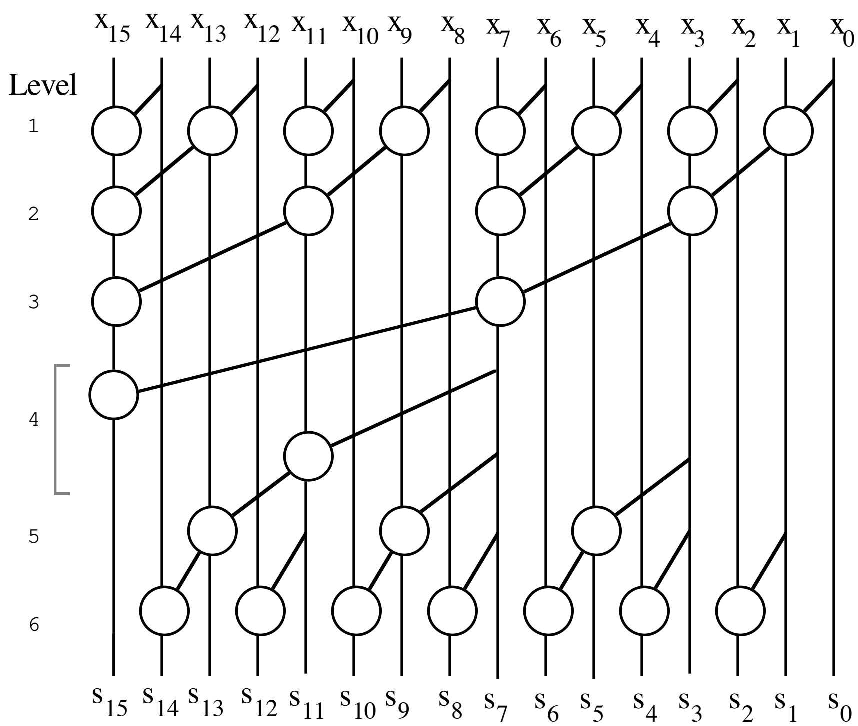


Fig. 6.9 Brent-Kung parallel prefix graph for 16 inputs.

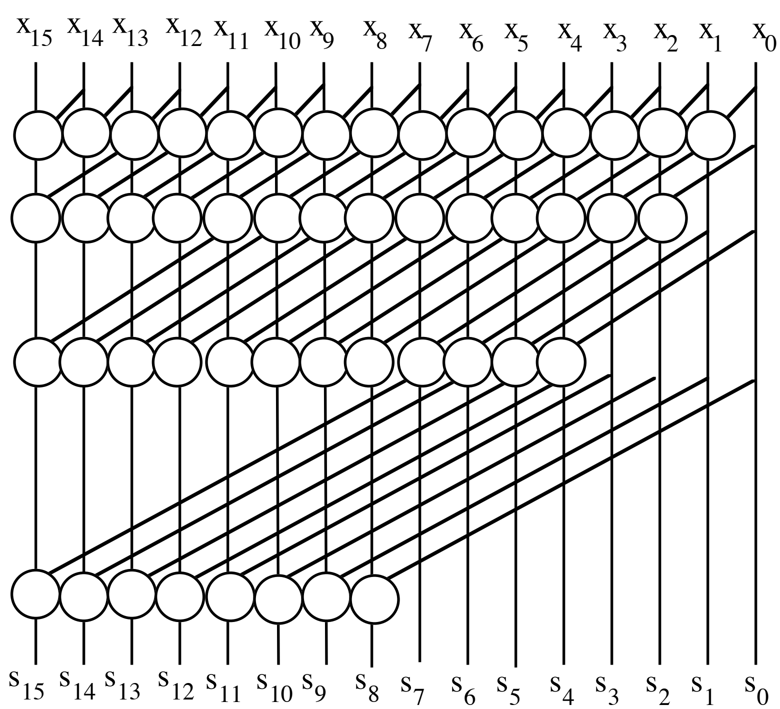
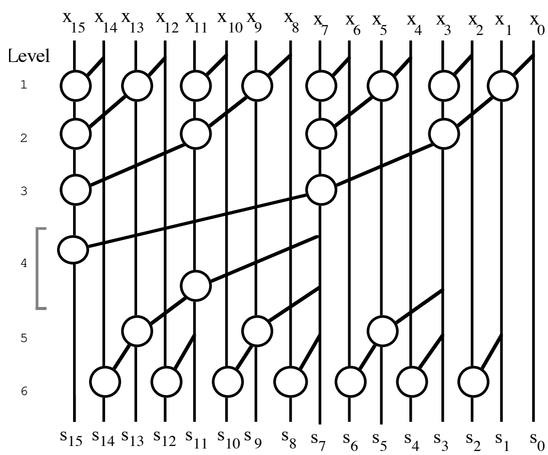
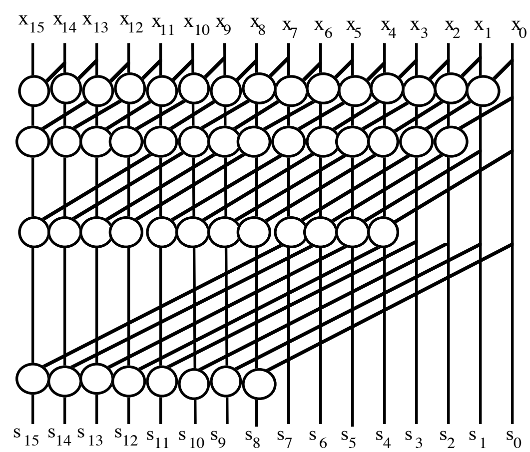


Fig. 6.10 Kogge-Stone parallel prefix graph for 16 inputs.





B-K: Six levels, 26 cells K-S: Four levels, 49 cells

Hybrid: Five levels, 32 cells

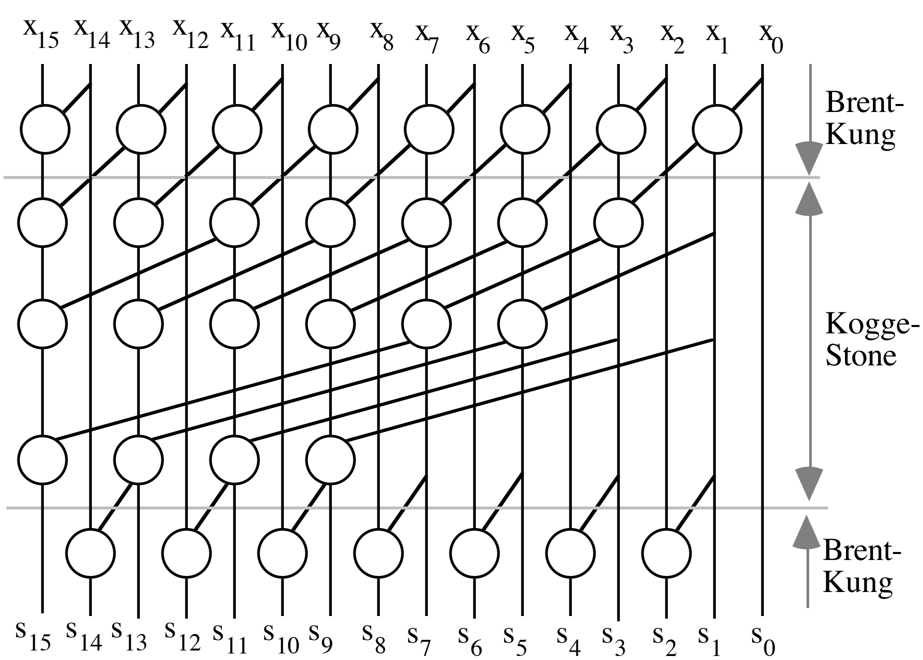


Fig. 6.11 A Hybrid Brent-Kung/Kogge-Stone parallel prefix graph for 16 inputs.

6.6 VLSI Implementation Aspects

Example: Radix-256 addition of 56-bit numbers

as implemented in the AMD Am29050 CMOS micro

The following description is based on the 64-bit version

In radix-256 addition of 64-bit numbers, only the carries

c_8, c_{16}, c_{24}, c_{32}, c_{40}, c_{48}, and c_{56} are needed

First, 4-bit Manchester carry chains (MCCs) of Fig. 6.12a are used to derive g and p signals for 4-bit blocks

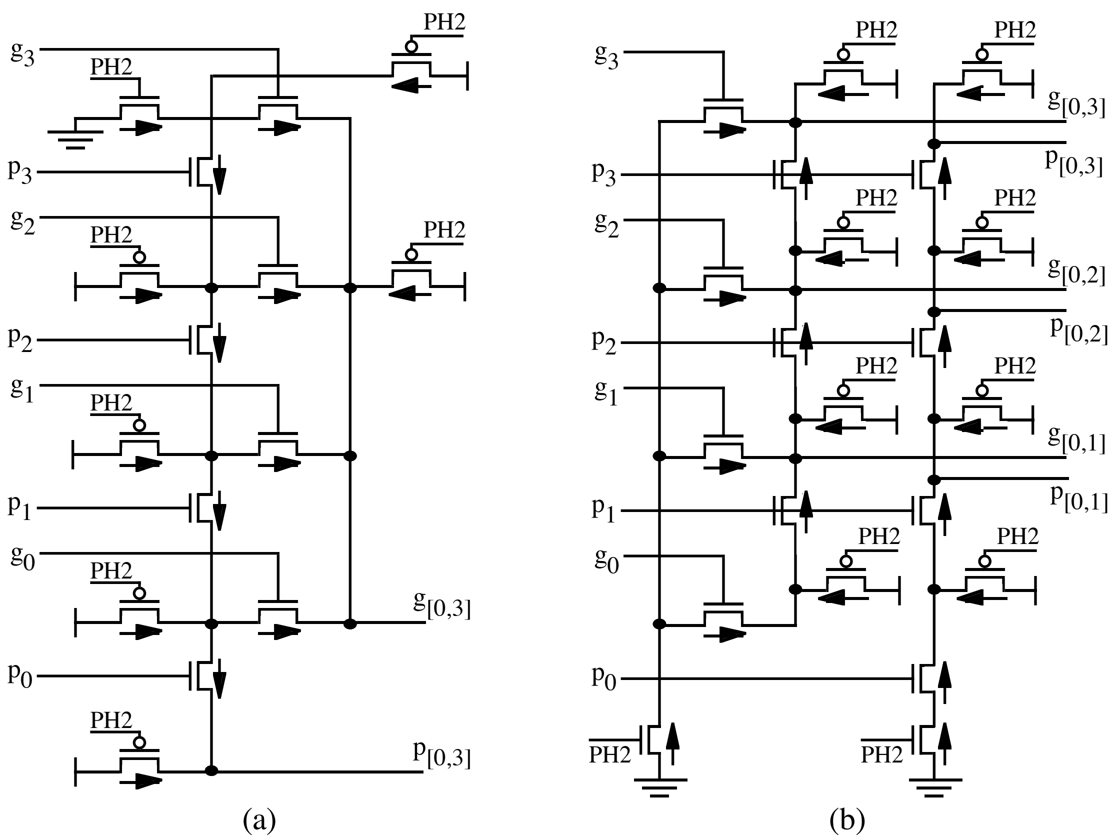
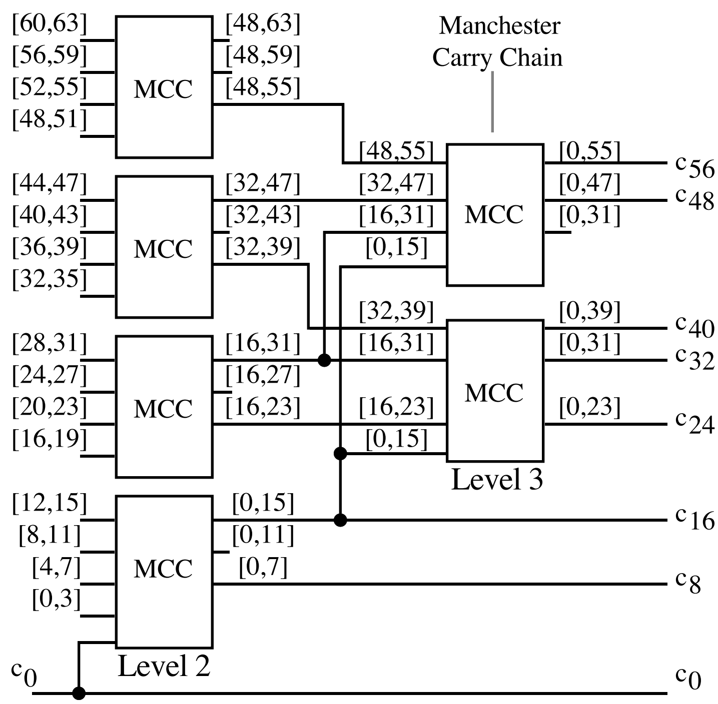


Fig. 6.12 Example four-bit Manchester carry chain designs in CMOS technology [Lync92].

These signal pairs,

denoted [0, 3], [4, 7], ... at the left edge of Fig. 6.13, form the inputs to one 5-bit and three 4-bit MCCs that in turn feed two more MCCs in the third level

The MCCs in Fig. 6.13 are of the type shown in Fig. 6.12b



[i,j] represents the pair of signals $p_{[i,j]}$ and $g_{[i,j]}$.
[0,j] should really be [–1,j], as c_0 is taken to be g_{-1}.

Fig. 6.13 Spanning-tree carry-lookahead network [Lync92]. The 16 MCCs at level 1, that produce generate and propagate signals for 4-bit blocks, are not shown.

7 Variations in Fast Adders

Chapter Goals

Study alternatives to the CLA method
for designing fast adders

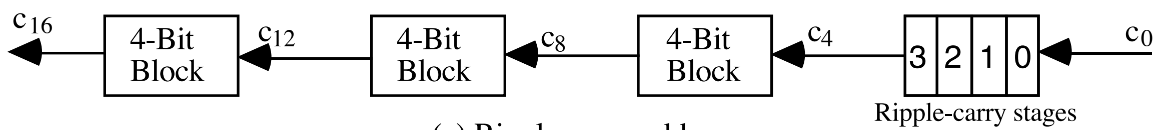
Chapter Highlights

Many methods besides CLA are available
(both competing and complementary)
Best adder design is technology-dependent
(often hybrid rather than pure)
Knowledge of timing allows optimizations

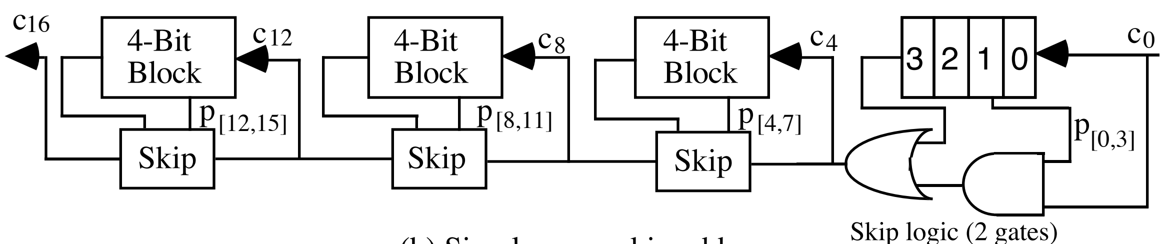
Chapter Contents

7.1 Simple Carry-Skip Adders
7.2 Multilevel Carry-Skip Adders
7.3 Carry-Select Adders
7.4 Conditional-Sum Adder
7.5 Hybrid Adder Designs
7.6 Optimizations in Fast Adders

7.1 Simple Carry-Skip Adders



(a) Ripple-carry adder.



(b) Simple carry-skip adder.

Fig. 7.1 Converting a 16-bit ripple-carry adder into a simple carry-skip adder with 4-bit skip blocks.

$$T_{\text{fixed-skip-add}} = \underset{\text{in block 0}}{(b-1)} + \underset{\text{OR gate}}{0.5} + \underset{\text{skips}}{(k/b-2)} + \underset{\text{in last block}}{(b-1)}$$

$$\approx 2b + k/b - 3.5 \text{ stages}$$

$$\frac{dT_{\text{fixed-skip-add}}}{db} = 2 - k/b^2 = 0 \quad \Rightarrow \quad b^{\text{opt}} = \sqrt{k/2}$$

$$T^{\text{opt}}_{\text{fixed-skip-add}} \approx 2\sqrt{2k} - 3.5$$

Example: $k = 32$, $b^{\text{opt}} = 4$, $T^{\text{opt}}_{\text{fixed-skip-add}} = 12.5$ stages

(contrast with 32 stages for a ripple-carry adder)

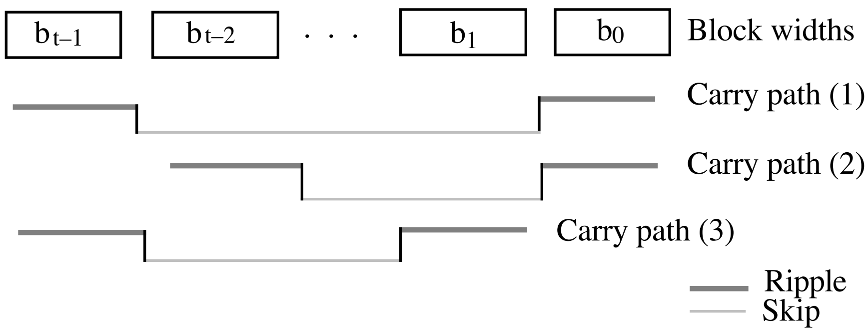


Fig. 7.2 Carry-skip adder with variable-size blocks and three sample carry paths.

Optimal variable-width blocks

$$b \quad b+1 \quad \cdots \quad b+t/2-1 \quad b+t/2-1 \quad \cdots \quad b+1 \quad b$$

The total number of bits in the t blocks is k:

$$2[b + (b+1) + \cdots + (b+t/2-1)] = t(b + t/4 - 1/2) = k$$

$$b = k/t - t/4 + 1/2$$

$$T_{\text{var-skip-add}} = 2(b-1) + 0.5 + t - 2 = 2k/t + t/2 - 2.5$$

$$\frac{dT_{\text{var-skip-add}}}{dt} = -2k/t^2 + 1/2 = 0 \quad \Rightarrow \quad t^{\text{opt}} = 2\sqrt{k}$$

Note: the optimal number of blocks with variable-width blocks is $\sqrt{2}$ times larger than that of fixed-width blocks

$$T^{\text{opt}}_{\text{var-skip-add}} \approx 2\sqrt{k} - 2.5$$

which is roughly a factor of $\sqrt{2}$ smaller than that obtained with optimal fixed-width skip-blocks

7.2 Multilevel carry-skip adders

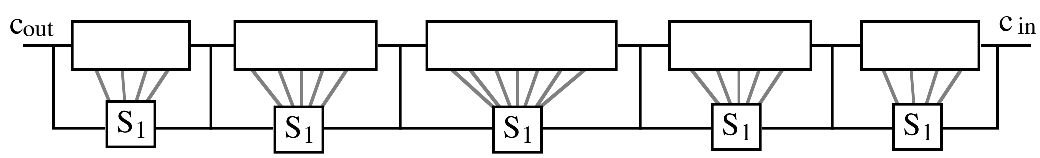


Fig. 7.3 Schematic diagram of a one-level carry-skip adder.

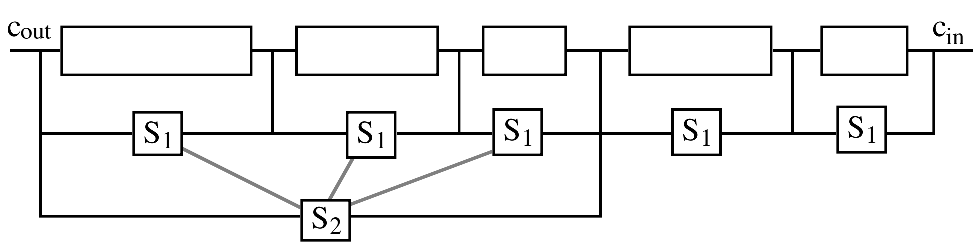


Fig. 7.4 Example of a two-level carry-skip adder.

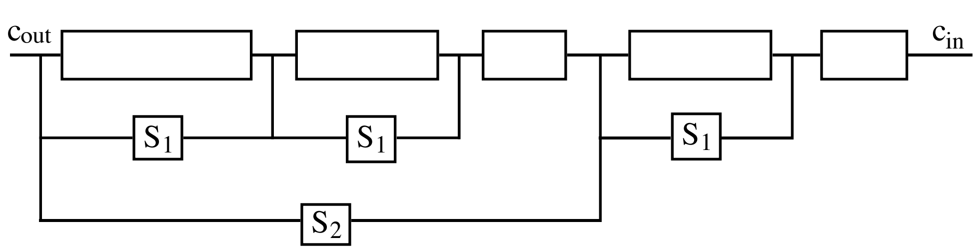


Fig. 7.5 Two-level carry-skip adder optimized by removing the short-block skip circuits.

Example 7.1

Each of the following operations takes one unit of time: generation of g_i and p_i, generation of level-i skip signal from level-(i–1) skip signals, ripple, skip, and computation of sum bit once the incoming carry is known

Build the widest possible single-level carry-skip adder with a total delay not exceeding 8 time units

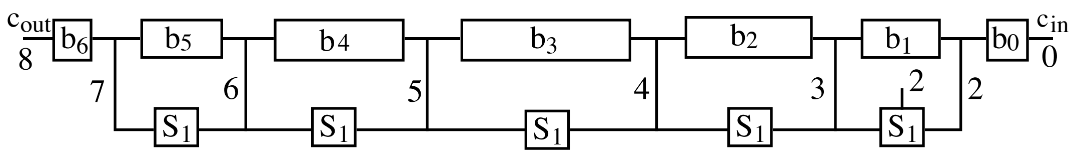


Fig. 7.6 Timing constraints of a single-level carry-skip adder with a delay of 8 units.

Max adder width = 1 + 3 + 4 + 4 + 3 + 2 + 1 = 18 bits

Generalization of Example 7.1:

For a single-level carry-skip adder with total latency of T, where T is even, the block widths are:

1 2 3 . . . T/2 . . . 3 2 1

This yields a total width of $T^2/4$ bits

When T is odd, the block widths become:

1 2 3 . . . (T – 1)/2 (T – 1)/2 . . . 3 2 1

This yields a total width of $(T^2 – 1)/4$

Thus, for any T, the total width is $\lfloor T^2/4 \rfloor$

Example 7.2

Each of the following operations takes one unit of time: generation of g_i and p_i, generation of level-i skip signal from level-(i–1) skip signals, ripple, skip, and computation of sum bit once the incoming carry is known

Build the widest possible two-level carry-skip adder with a total delay not exceeding 8 time units

First determine the number of blocks and timing constraints at the second level

The remaining problem is to build single-level carry-skip adders with $T_{produce} = \beta$ and $T_{assimilate} = \alpha$

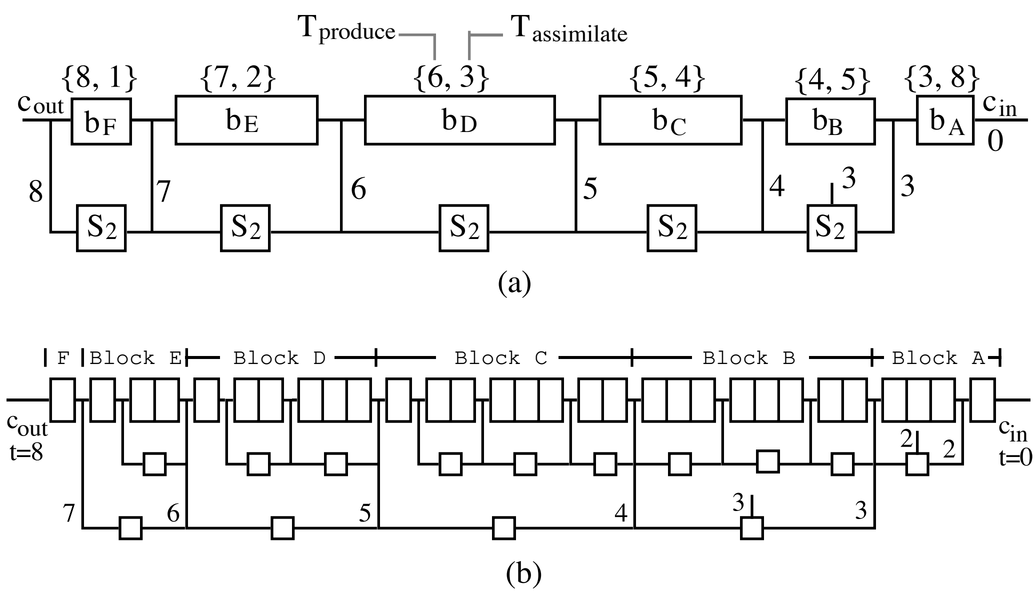


Fig. 7.7 Two-level carry-skip adder with a delay of 8 units: (a) Initial timing constraints, (b) Final design.

Table 7.1 Second-level constraints $T_{produce}$ and $T_{assimilate}$, with associated subblock and block widths, in a two-level carry-skip adder with a total delay of 8 time units (Fig. 7.7)

Block	$T_{produce}$ β	$T_{assimilate}$ α	Number of subblocks $\min(\beta-1, \alpha)$	Subblock widths (bits)	Block width (bits)
A	3	8	2	1, 3	4
B	4	5	3	2, 3, 3	8
C	5	4	4	2, 3, 2, 1	8
D	6	3	3	3, 2, 1	6
E	7	2	2	2, 1	3
F	8	1	1	1	1

Total width: 30 bits

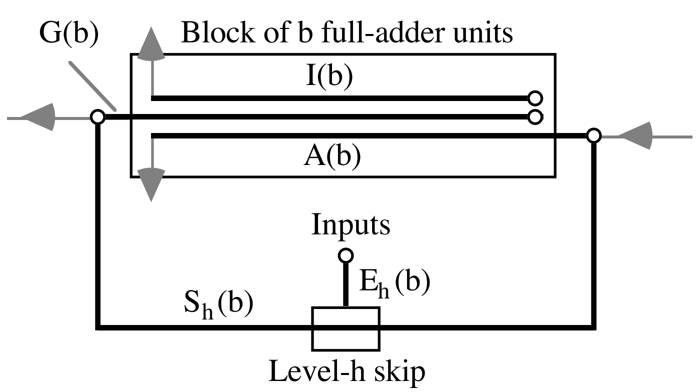
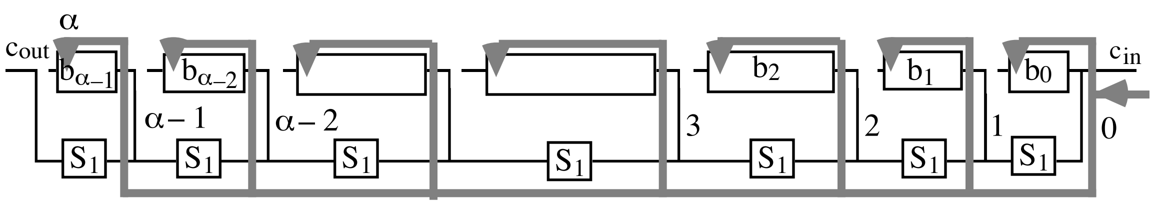


Fig. 7.8 Generalized delay model for carry-skip adders.

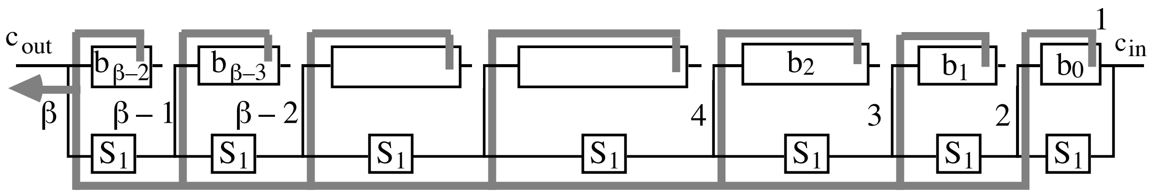
Elaboration on Example 7.2

Given the delay pair $\{\beta, \alpha\}$ for a level-2 block in Fig. 7.7a, the number of level-1 blocks in the corresponding single-level carry-skip adder will be $\gamma = \min(\beta - 1, \alpha)$

This is easily verified from the following two diagrams.



Single-level carry-skip adder with $T_{\text{assimilate}} = \alpha$



Single-level carry-skip adder with $T_{\text{produce}} = \beta$

The width of the ith level-1 block in the level-2 block characterized by $\{\beta, \alpha\}$ is $b_i = \min(\beta - \gamma + i + 1, \alpha - i)$

So, the total width of such a block is:

$$\sum_{i=0}^{\gamma-1} \min(\beta - \gamma + i + 1, \alpha - i)$$

The only exception occurs in the rightmost level-2 block A for which b_0 is one unit less than the value given above in order to accommodate the carry-in signal

7.3 Carry-Select Adders

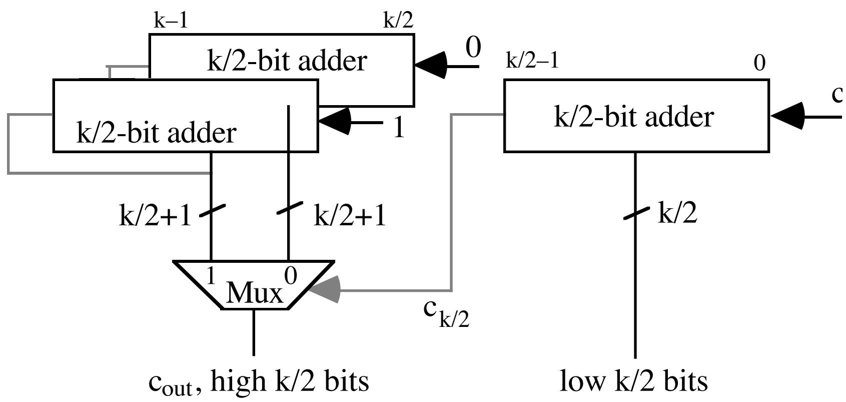


Fig. 7.9 Carry-select adder for k-bit numbers built from three k/2-bit adders.

$$C_{\text{select-add}}(k) = 3C_{\text{add}}(k/2) + k/2 + 1$$
$$T_{\text{select-add}}(k) = T_{\text{add}}(k/2) + 1$$

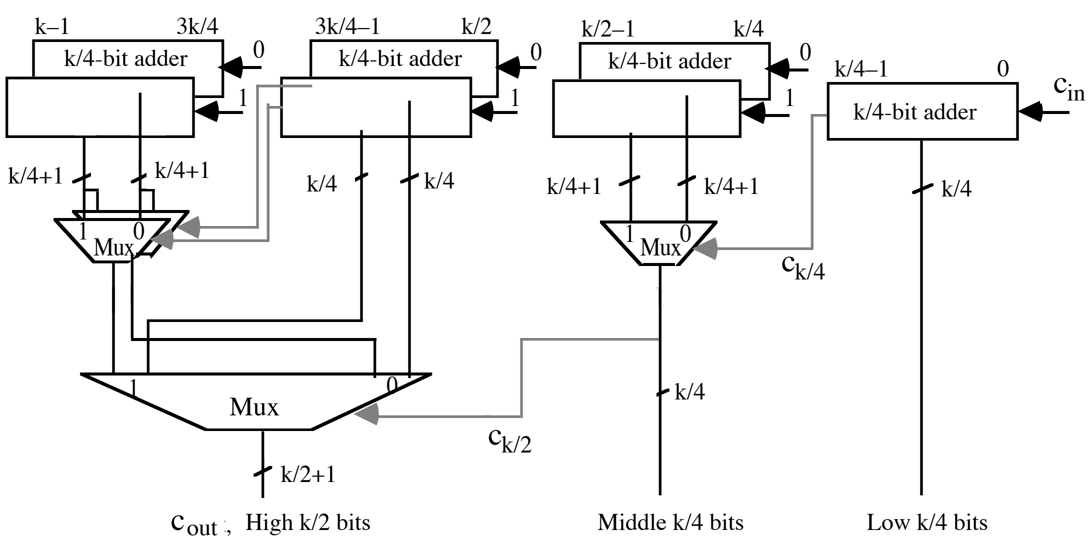


Fig. 7.10 Two-level carry-select adder built of k/4-bit adders.

7.4 Conditional-Sum Adder

Multilevel carry-select idea carried out to the extreme, until we arrive at single-bit blocks.

$$C(k) \approx 2C(k/2) + k + 2 \approx k\,(\log_2 k + 2) + k\,C(1)$$

$$T(k) = T(k/2) + 1 = \log_2 k + T(1)$$

where C(1) and T(1) are the cost and delay of the circuit of Fig. 7.11 used at the top to derive the sum and carry bits with a carry-in of 0 and 1

The term k + 2 in the first recurrence represents an upper bound on the number of single-bit 2-to-1 multiplexers needed for combining two k/2-bit adders into a k-bit adder

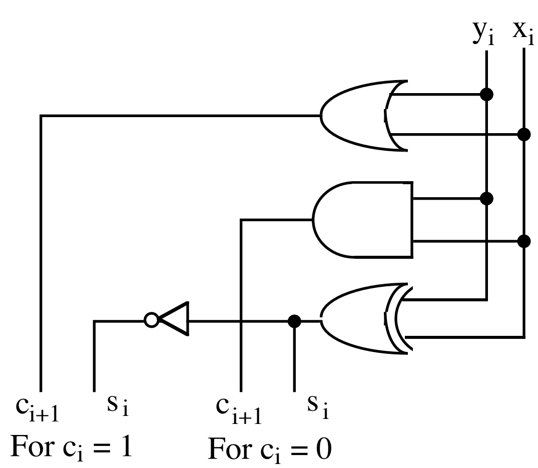


Fig. 7.11 Top-level block for one bit position of a conditional-sum adder.

Table 7.2 Conditional-sum addition of two 16-bit numbers. The width of the block for which the sum and carry bits are known doubles with each additional level, leading to an addition time that grows as the logarithm of the word width k.

		x	0	0	1	0	0	1	1	0	1	1	1	0	1	0	1	0	
		y	0	1	0	0	1	0	1	1	0	1	0	1	1	1	0	1	
Block width	Block carry-in	Block sum and block carry-out	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	c_{in}
1	0	s	0	1	1	0	1	1	0	1	1	0	1	1	0	1	1	1	
		c	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0
	1	s	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0		
		c	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1		
2	0	s	0	1	1	0	1	1	0	1	0	0	1	1	0	1	1	1	
		c	0		0		0		1		1		0		1		0		
	1	s	1	0	1	1	0	0	1	0	0	1	0	0	1	0			
		c	0		0		1		1		1		1		1				
4	0	s	0	1	1	0	0	0	0	1	0	0	1	1	0	1	1	1	
		c	0				1				1				1				
	1	s	0	1	1	1	0	0	1	0	0	1	0	0					
		c	0				1				1								
8	0	s	0	1	1	1	0	0	0	1	0	1	0	0	0	1	1	1	
		c	0								1								
	1	s	0	1	1	1	0	0	1	0									
		c	0																
16	0	s	0	1	1	1	0	0	1	0	0	1	0	0	0	1	1	1	
		c	0																
	1	s																	
		c																	

c_{out}

7.5 Hybrid Adder Designs

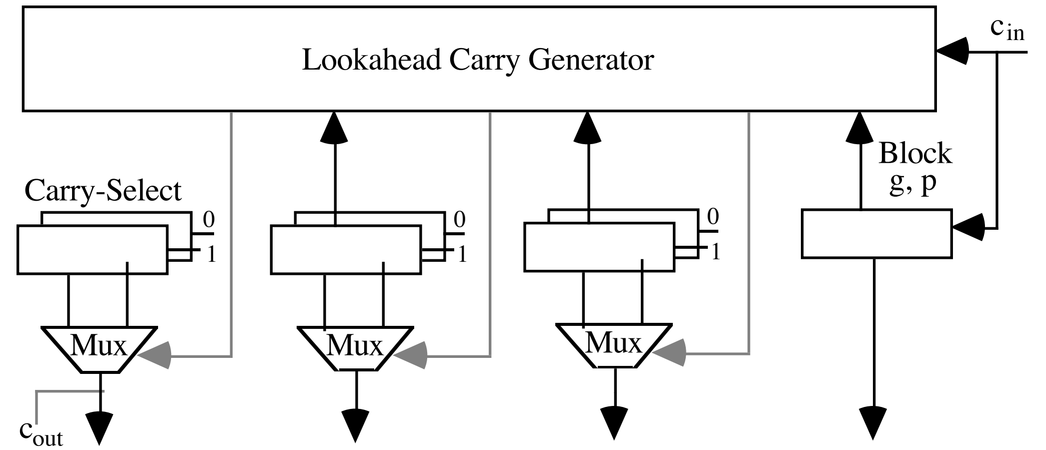


Fig. 7.12 A hybrid carry-lookahead/carry-select adder.

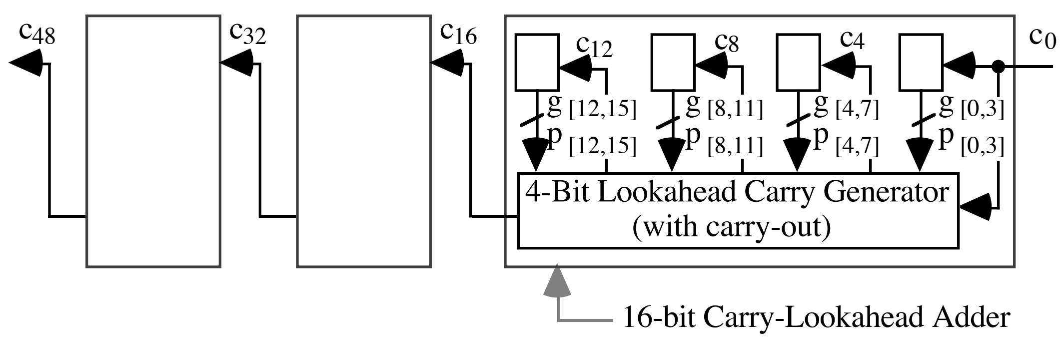


Fig. 7.13 Example 48-bit adder with hybrid ripple-carry/carry-lookahead design.

Other possibilities: hybrid carry-select/ripple-carry
hybrid ripple-carry/carry-select
. . .

7.6 Optimizations in Fast Adders

What looks best at the block diagram or gate level may not be best when a circuit-level design is generated (effects of wire length, signal loading, ...)

Modern practice: optimization at the transistor level

Variable-block carry-lookahead adder

Optimization based on knowledge of given input timing or required output timing

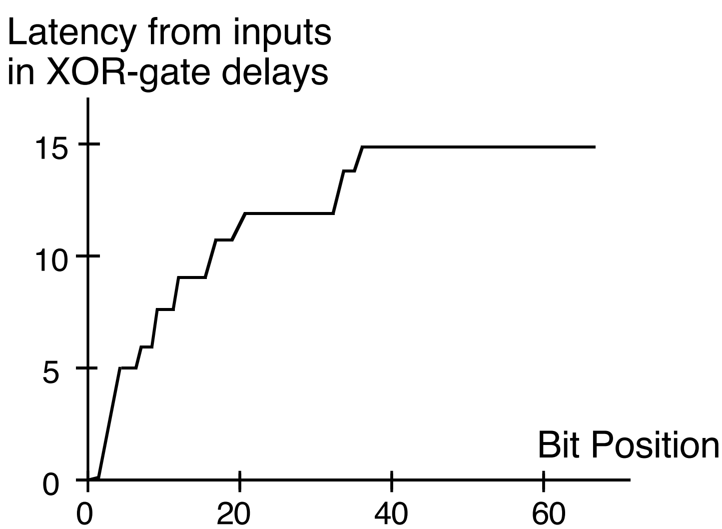


Fig. 7.14 Example arrival times for operand bits in the final fast adder of a tree multiplier [Oklo96].

8 Multi-Operand Addition

Chapter Goals

Learn methods for speeding up the addition of several numbers (as required for multiplication or inner-product computation)

Chapter Highlights

Running total kept in redundant form

Current total + Next number –> New total

Deferred carry assimilation

Wallace/Dadda trees and parallel counters

Chapter Contents

8.1 Using Two-Operand Adders
8.2 Carry-Save Adders
8.3 Wallace and Dadda Trees
8.4 Parallel Counters
8.5 Generalized Parallel Counters
8.6 Adding Multiple Signed Numbers

8.1 Using Two-Operand Adders

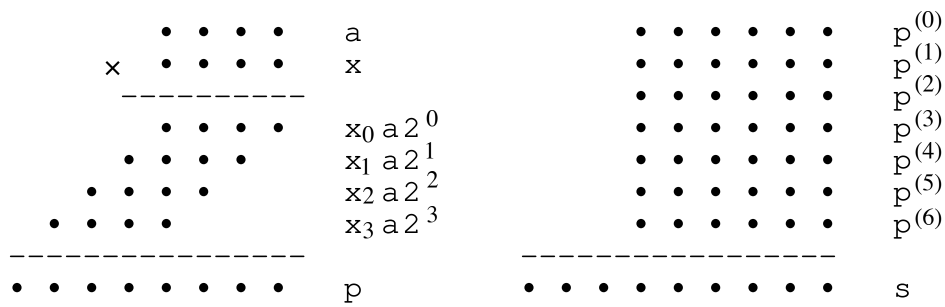


Fig. 8.1 Multioperand addition problems for multiplication or inner-product computation in dot notation.

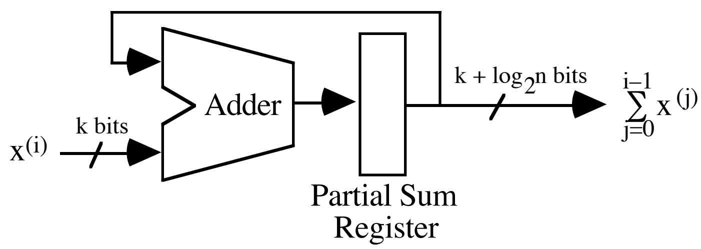


Fig. 8.2 Serial implementation of multi-operand addition with a single 2-operand adder.

$$T_{\text{serial-multi-add}} = O(n \log(k + \log n))$$

Because $\max(k, \log n) < k + \log n \le \max(2k, 2 \log n)$
we have $\log(k + \log n) = O(\log k + \log\log n)$ and:

$$T_{\text{serial-multi-add}} = O(n \log k + n \log\log n)$$

Therefore, addition time grows superlinearly with n when k is fixed and logarithmically with k for a given n

One can pipeline this serial solution to get somewhat better performance.

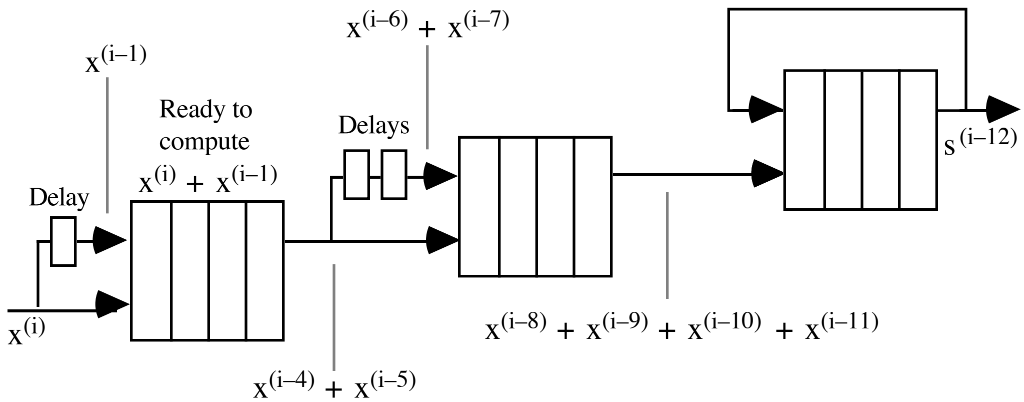


Fig. 8.3 Serial multi-operand addition when each adder is a 4-stage pipeline.

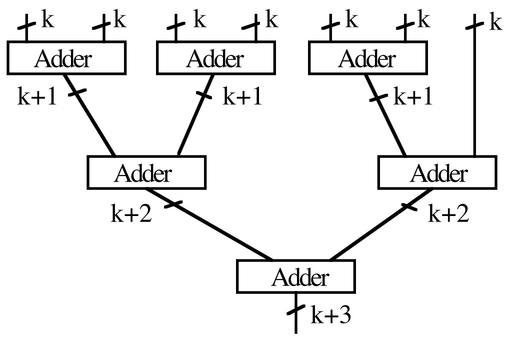


Fig. 8.4 Adding 7 numbers in a binary tree of adders.

$$T_{\text{tree-fast-multi-add}} = O(\log k + \log(k+1) + \cdots + \log(k + \lceil \log_2 n \rceil - 1))$$
$$= O(\log n \log k + \log n \log\log n)$$

$$T_{\text{tree-ripple-multi-add}} = O(k + \log n)$$

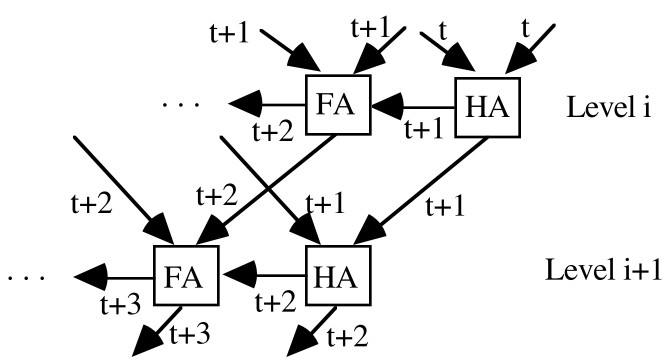


Fig. 8.5 Ripple-carry adders at levels i and i + 1 in the tree of adders used for multi-operand addition.

8.2 Carry-Save Adders

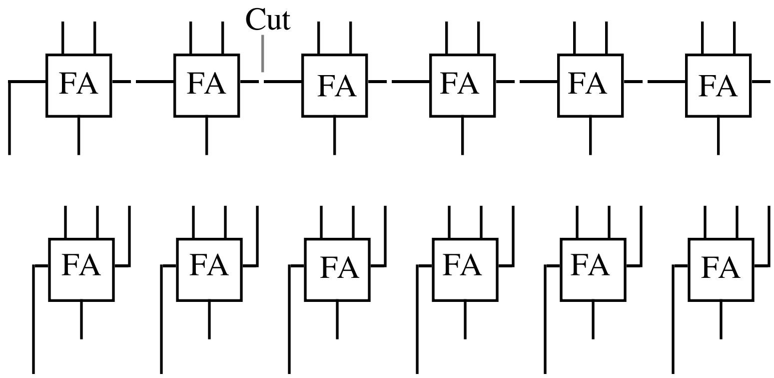


Fig. 8.6 A ripple-carry adder turns into a carry-save adder if the carries are saved (stored) rather than propagated.

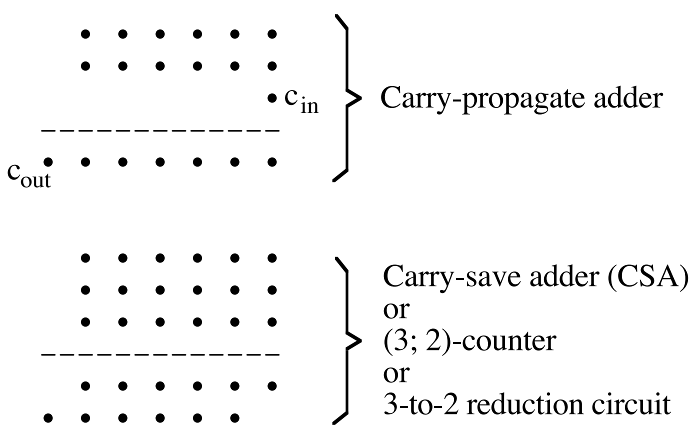


Fig. 8.7 Carry-propagate adder (CPA) and carry-save adder (CSA) functions in dot notation.

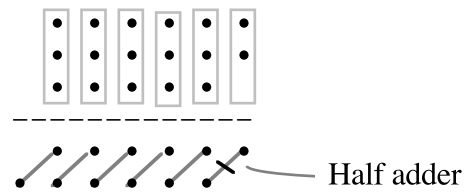


Fig. 8.8 Specifying full- and half-adder blocks, with their inputs and outputs, in dot notation.

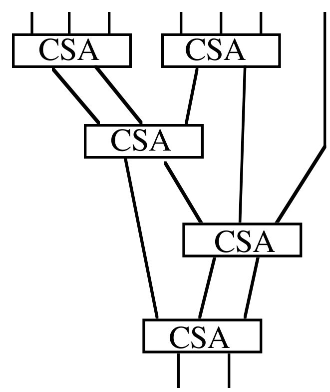
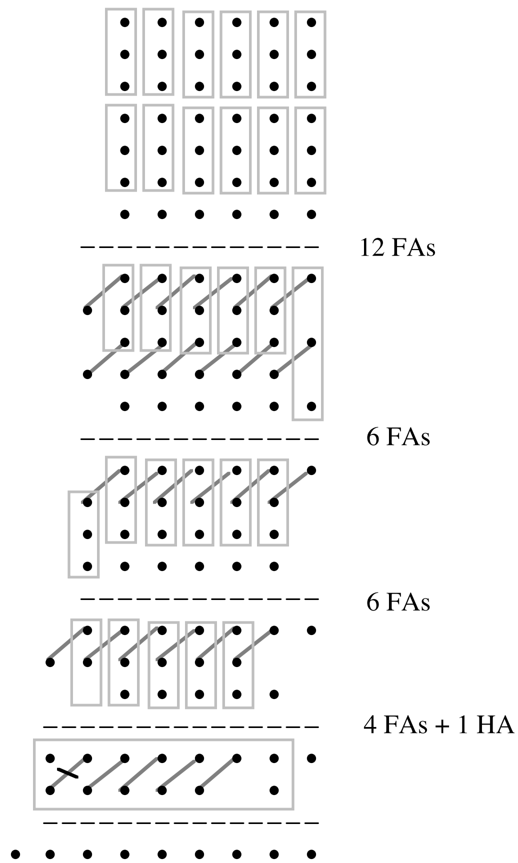


Fig. 8.9 Tree of carry-save adders reducing seven numbers to two.

$$
\begin{aligned}
T_{\text{carry-save-multi-add}} &= O(\text{tree height} + T_{CPA}) \\
&= O(\log n + \log k)
\end{aligned}
$$

$$
C_{\text{carry-save-multi-add}} = (n-1)C_{CSA} + C_{CPA}
$$



Total cost = 7-bit adder + 28 FAs + 1 HA

Fig. 8.10 Addition of seven 6-bit numbers in dot notation.

8	7	6	5	4	3	2	1	0	Bit position
			7	7	7	7	7	7	6 × 2 = 12 FAs
		2	5	5	5	5	5	3	6 FAs
		3	4	4	4	4	4	1	6 FAs
	1	2	3	3	3	3	2	1	4 FAs + 1 HA
	2	2	2	2	2	1	2	1	7-bit adder
	--- Carry-propagate adder ---								
1	1	1	1	1	1	1	1	1	

Fig. 8.11 Representing a seven-operand addition in tabular form.

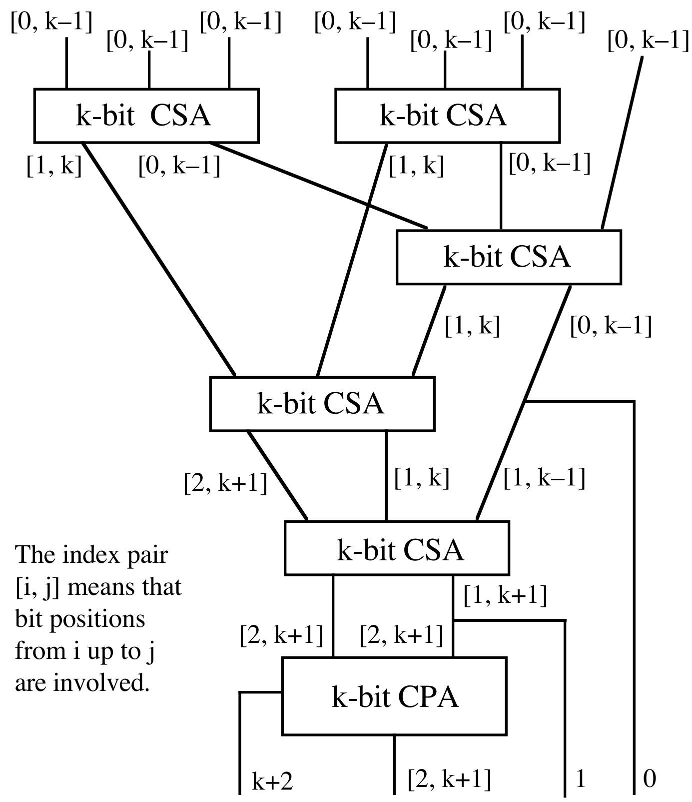


Fig. 8.12 Adding seven k-bit numbers and the CSA/CPA widths required.

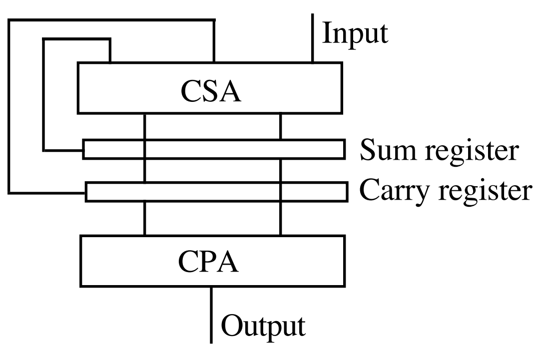
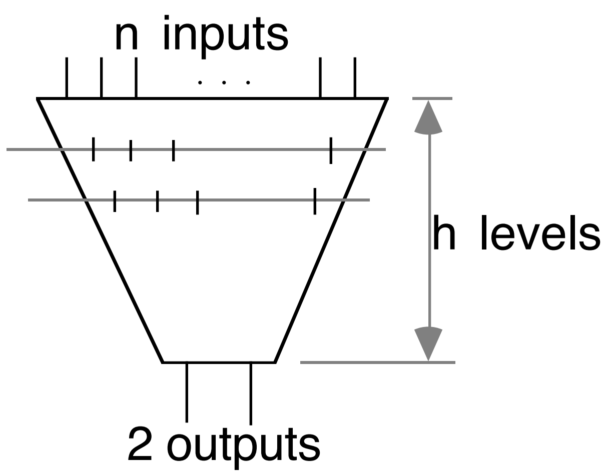


Fig. 8.13 Serial carry-save addition using a single CSA.

8.3 Wallace and Dadda Trees



$$h(n) = 1 + h(\lceil 2n/3 \rceil)$$

$$n(h) = \lfloor 3n(h-1)/2 \rfloor$$

$$2 \times 1.5^{h-1} < n(h) \le 2 \times 1.5^{h}$$

Table 8.1 The maximum number n(h) of inputs for an h-level carry-save-adder tree

h	n(h)	h	n(h)	h	n(h)
0	2	7	28	14	474
1	3	8	42	15	711
2	4	9	63	16	1066
3	6	10	94	17	1599
4	9	11	141	18	2398
5	13	12	211	19	3597
6	19	13	316	20	5395

In a Wallace tree, we reduce the number of operands at the earliest possible opportunity

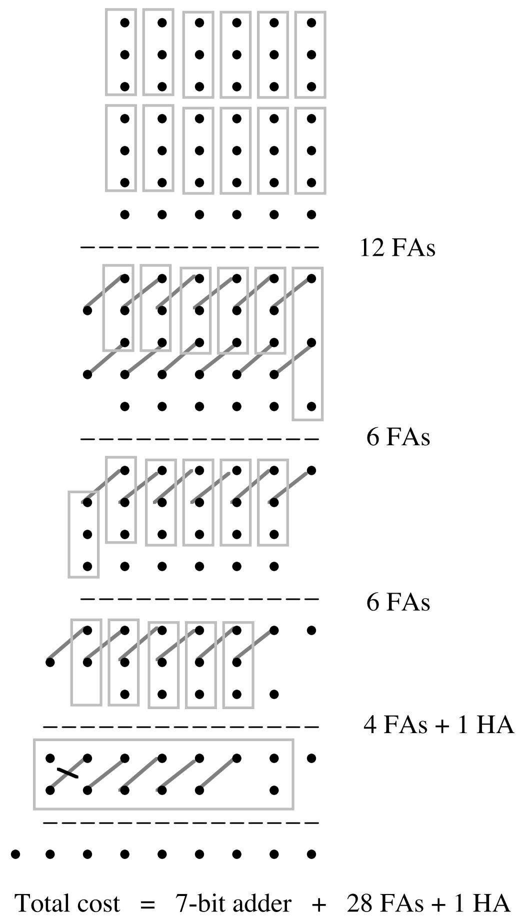
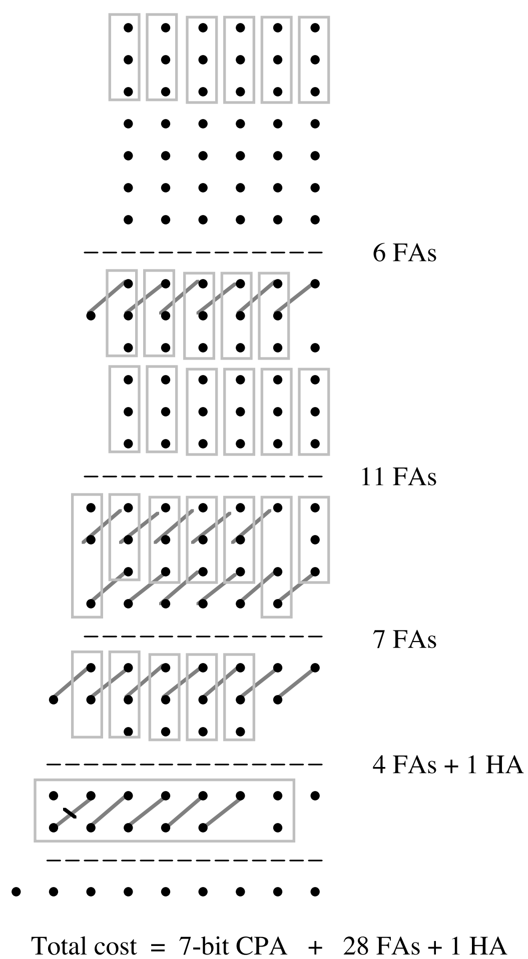


Fig. 8.10 Addition of seven 6-bit numbers in dot notation.

In a Dadda tree, we reduce the number of operands at the latest possible opportunity that leads to no added delay (target the next smaller number in Table 8.1)



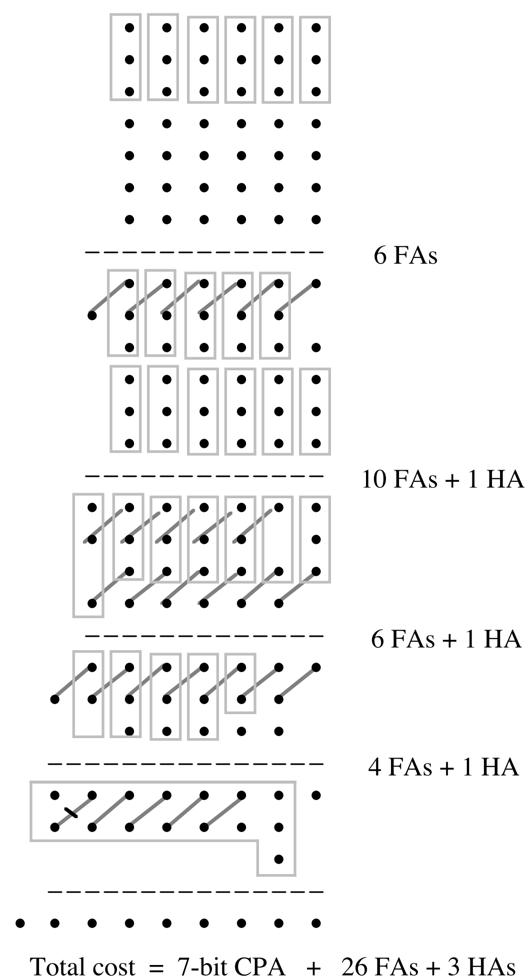


Fig. 8.14 Adding seven 6-bit numbers using Dadda's strategy (left).

Fig. 8.15 Adding seven 6-bit numbers by taking advantage of the final adder's carry-in (right).

8.4 Parallel Counters

Single-bit full-adder = (3; 2)-counter

Circuit reducing 7 bits to their 3-bit sum = (7; 3)-counter

Circuit reducing n bits to their $\lceil \log_2(n + 1) \rceil$-bit sum

$$= (n; \lceil \log_2(n + 1) \rceil)\text{-counter}$$

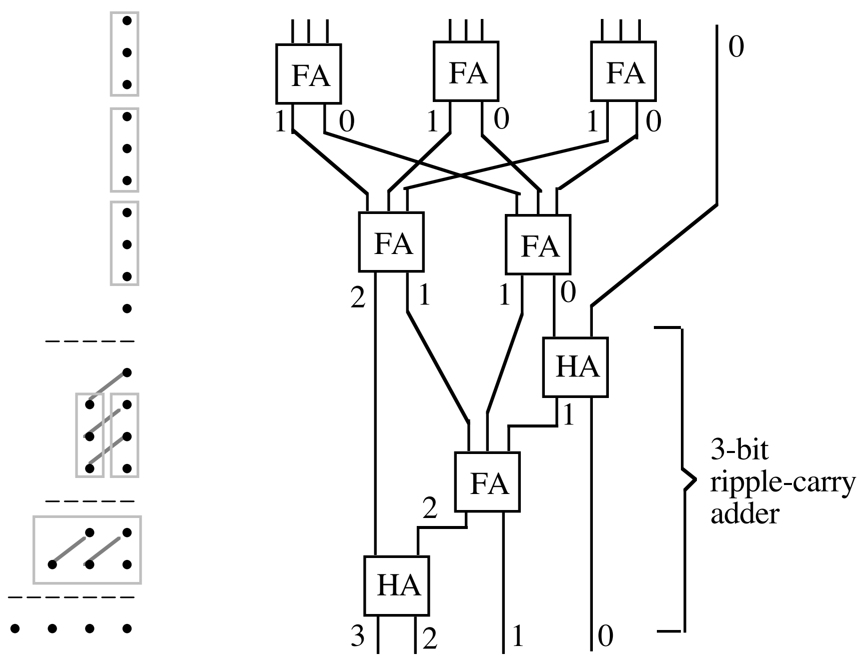


Fig. 8.16 A 10-input parallel counter also known as a (10; 4)-counter.

8.5 Generalized Parallel Counters

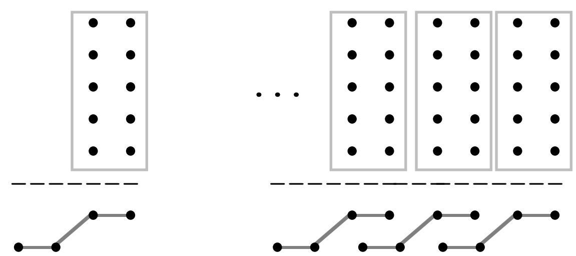
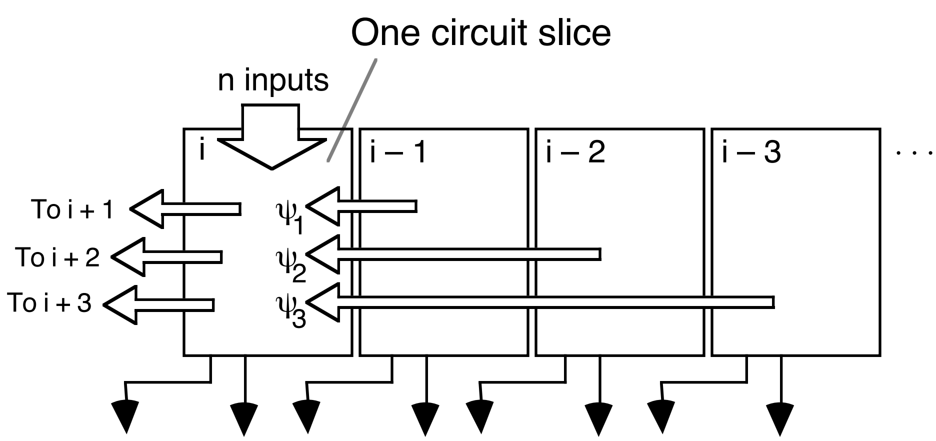


Fig. 8.17 Dot notation for a (5, 5; 4)-counter and the use of such counters for reducing 5 numbers to 2 numbers.

(n; 2)-counters



$$n + \psi_1 + \psi_2 + \psi_3 + \cdots \le 3 + 2\psi_1 + 4\psi_2 + 8\psi_3 + \cdots$$

8.6 Adding Multiple Signed Numbers

Extended positions					Sign	Magnitude positions			
x_{k-1}	x_{k-1}	x_{k-1}	x_{k-1}	x_{k-1}	x_{k-1}	x_{k-2}	x_{k-3}	x_{k-4}	$\cdots$
y_{k-1}	y_{k-1}	y_{k-1}	y_{k-1}	y_{k-1}	y_{k-1}	y_{k-2}	y_{k-3}	y_{k-4}	$\cdots$
z_{k-1}	z_{k-1}	z_{k-1}	z_{k-1}	z_{k-1}	z_{k-1}	z_{k-2}	z_{k-3}	z_{k-4}	$\cdots$

(a)

Extended positions					Sign	Magnitude positions			
1	1	1	1	0	$\bar{x}_{k-1}$	x_{k-2}	x_{k-3}	x_{k-4}	$\cdots$
					$\bar{y}_{k-1}$	y_{k-2}	y_{k-3}	y_{k-4}	$\cdots$
					$\bar{z}_{k-1}$	z_{k-2}	z_{k-3}	z_{k-4}	$\cdots$
					1				

(b)

Fig. 8.18 Adding three 2's-complement numbers using sign extension (a) or by the method based on negatively weighted sign bits (b).

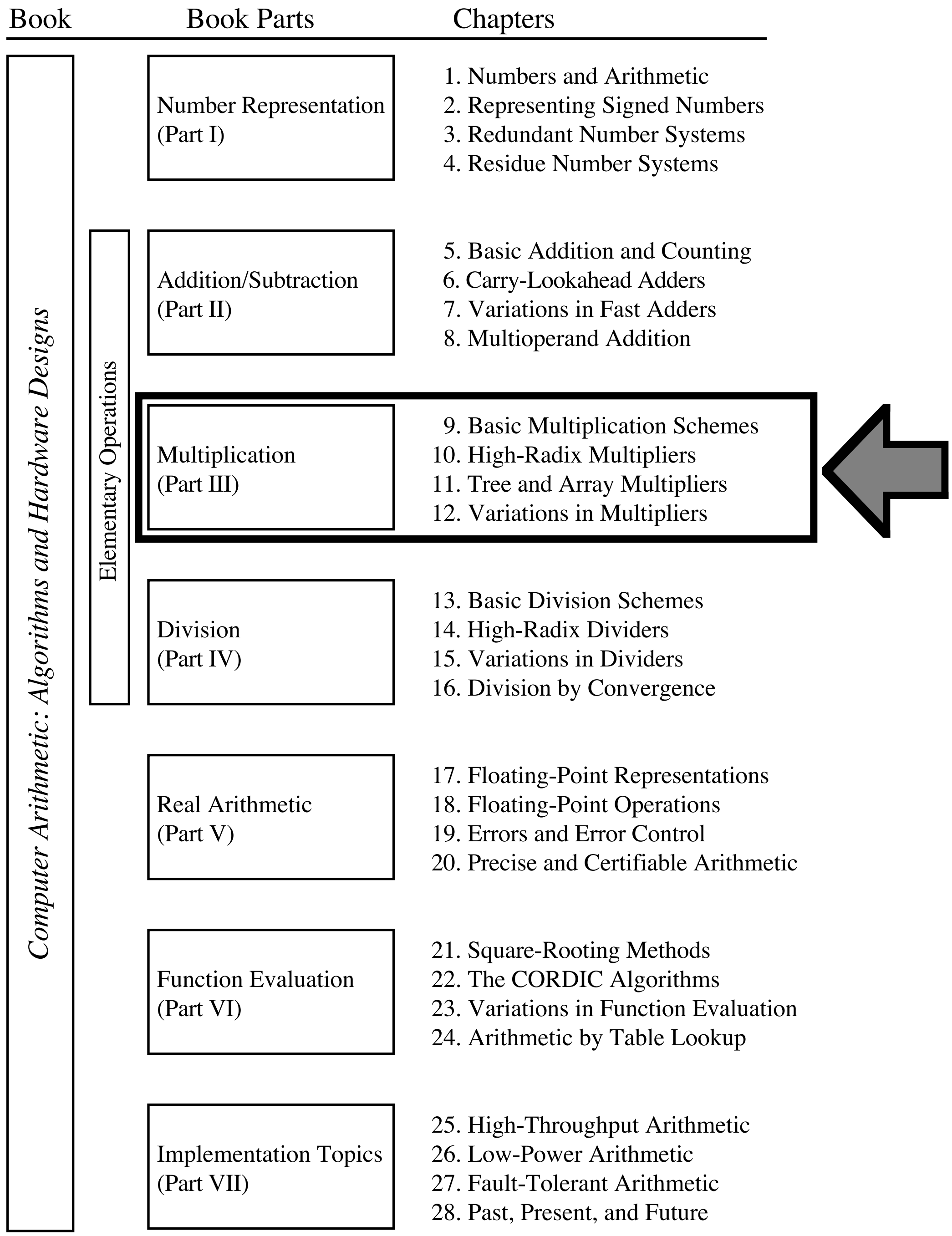


Fig. P.1 The structure of this book in parts and chapters.

Part III Multiplication

Part Goals

Review shift-add multiplication schemes

Learn about faster multipliers

Discuss speed/cost tradeoffs in multipliers

Part Synopsis

Multiplication is an often-used operation
(arithmetic and array index calculations)

Division = reciprocation + multiplication

Multiplication speedup: high-radix, tree, ...

Bit-serial, modular, and array multipliers

Part Contents

Chapter 9 Basic Multiplication Schemes

Chapter 10 High-Radix Multipliers

Chapter 11 Tree and Array Multipliers

Chapter 12 Variations in Multipliers

9 Basic Multiplication Schemes

Chapter Goals

Study shift/add or bit-at-a-time multipliers and set the stage for faster methods and variations to be covered in Chapters 10-12

Chapter Highlights

Multiplication = multioperand addition
Hardware, firmware, software algorithms
Multiplying 2's-complement numbers
The special case of one constant operand

Chapter Contents

9.1. Shift/Add Multiplication Algorithms
9.2. Programmed Multiplication
9.3. Basic Hardware Multipliers
9.4. Multiplication of Signed Numbers
9.5. Multiplication by Constants
9.6. Preview of Fast Multipliers

“At least one good reason
for studying multiplication and division is that
there is an infinite number of ways
of performing these operations
and hence there is an infinite number of PhDs
(or expenses-paid visits to conferences in the USA)
to be won from inventing new forms of multiplier.”

Alan Clements
The Principles of Computer Hardware, 1986

9.1 Shift/Add Multiplication Algorithms

Notation for our discussion of multiplication algorithms:

a	Multiplicand	$a_{k-1}a_{k-2} \cdots a_1a_0$
x	Multiplier	$x_{k-1}x_{k-2} \cdots x_1x_0$
p	Product ($a \times x$)	$p_{2k-1}p_{2k-2} \cdots p_1p_0$

Initially, we assume unsigned operands

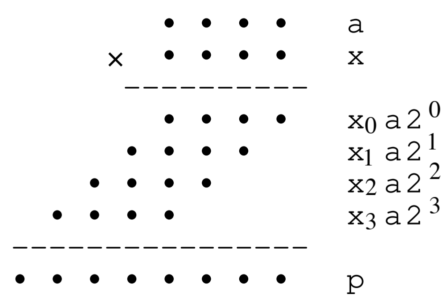


Fig. 9.1 Multiplication of two 4-bit unsigned binary numbers in dot notation.

Multiplication with right shifts: top-to-bottom accumulation

$p^{(j+1)} = (p^{(j)} + x_j\, a\, 2^k)\, 2^{-1}$ with $p^{(0)} = 0$ and

|——add——| $p^{(k)} = p = ax + p^{(0)}2^{-k}$

|—shift right—|

Multiplication with left shifts: bottom-to-top accumulation

$p^{(j+1)} = 2p^{(j)} + x_{k-j-1}a$ with $p^{(0)} = 0$ and

|shift| $p^{(k)} = p = ax + p^{(0)}2^{k}$

|——add——|

Right-shift algorithm	
a	1 0 1 0
x	1 0 1 1
$p^{(0)}$	0 0 0 0
$+x_0a$	1 0 1 0
$2p^{(1)}$	0 1 0 1 0
$p^{(1)}$	0 1 0 1 0
$+x_1a$	1 0 1 0
$2p^{(2)}$	0 1 1 1 1 0
$p^{(2)}$	0 1 1 1 1 0
$+x_2a$	0 0 0 0
$2p^{(3)}$	0 0 1 1 1 1 0
$p^{(3)}$	0 0 1 1 1 1 0
$+x_3a$	1 0 1 0
$2p^{(4)}$	0 1 1 0 1 1 1 0
$p^{(4)}$	0 1 1 0 1 1 1 0

Left-shift algorithm	
a	1 0 1 0
x	1 0 1 1
$p^{(0)}$	0 0 0 0
$2p^{(0)}$	0 0 0 0 0
$+x_3a$	1 0 1 0
$p^{(1)}$	0 1 0 1 0
$2p^{(1)}$	0 1 0 1 0 0
$+x_2a$	0 0 0 0
$p^{(2)}$	0 1 0 1 0 0
$2p^{(2)}$	0 1 0 1 0 0 0
$+x_1a$	1 0 1 0
$p^{(3)}$	0 1 1 0 0 1 0
$2p^{(3)}$	0 1 1 0 0 1 0 0
$+x_0a$	1 0 1 0
$p^{(4)}$	0 1 1 0 1 1 1 0

Fig. 9.2 Examples of sequential multiplication with right and left shifts.

9.2 Programmed Multiplication

R0	0	Rc	Counter
Ra	Multiplicand	Rx	Multiplier
Rp	Product, high	Rq	Product, low

```
{Using right shifts, multiply unsigned m_cand and m_ier,
storing the resultant 2k-bit product in p_high and p_low.
Registers: R0 holds 0         Rc for counter
           Ra for m_cand      Rx for m_ier
           Rp for p_high      Rq for p_low}

{Load operands into registers Ra and Rx}

    mult:  load      Ra with m_cand
           load      Rx with m_ier

{Initialize partial product and counter}

           copy      R0 into Rp
           copy      R0 into Rq
           load      k  into Rc

{Begin multiplication loop}

  m_loop:  shift     Rx right 1  {LSB moves to carry flag}
           branch    no_add if carry = 0
           add       Ra to Rp    {carry flag is set to c_out}
  no_add:  rotate    Rp right 1  {carry to MSB, LSB to carry}
           rotate    Rq right 1  {carry to MSB, LSB to carry}
           decr      Rc          {decrement counter by 1}
           branch    m_loop if Rc ≠ 0

{Store the product}

           store     Rp into p_high
           store     Rq into p_low
  m_done:  ...
```

Fig. 9.3 Programmed multiplication using the right-shift algorithm.

9.3 Basic Hardware Multipliers

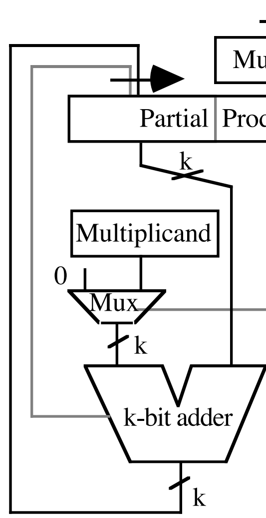


Fig. 9.4 Hardware realization of the sequential multiplication algorithm with additions and right shifts.

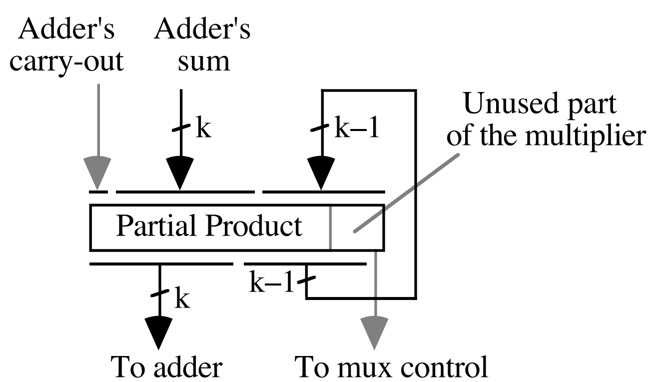


Fig. 9.5 Combining the loading and shifting of the double-width register holding the partial product and the partially used multiplier.

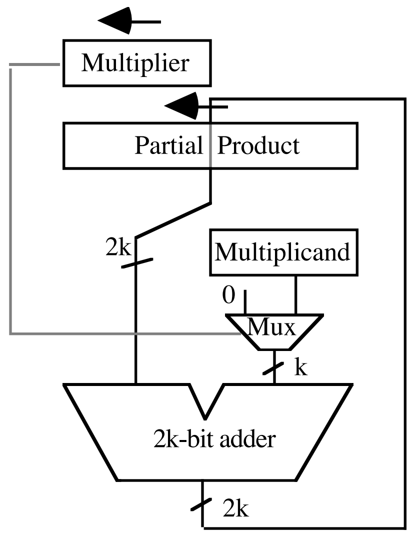


Fig. 9.6 Hardware realization of the sequential multiplication algorithm with left shifts and additions.

9.4 Multiplication of Signed Numbers

a	1 0 1 1 0	
x	0 1 0 1 1	
$p^{(0)}$	0 0 0 0 0	
$+x_0a$	1 0 1 1 0	
$2p^{(1)}$	1 1 0 1 1 0	
$p^{(1)}$	1 1 0 1 1	0
$+x_1a$	1 0 1 1 0	
$2p^{(2)}$	1 1 0 0 0 1	0
$p^{(2)}$	1 1 0 0 0	1 0
$+x_2a$	0 0 0 0 0	
$2p^{(3)}$	1 1 1 0 0 0	1 0
$p^{(3)}$	1 1 1 0 0	0 1 0
$+x_3a$	1 0 1 1 0	
$2p^{(4)}$	1 1 0 0 1 0	0 1 0
$p^{(4)}$	1 1 0 0 1	0 0 1 0
$+x_4a$	0 0 0 0 0	
$2p^{(5)}$	1 1 1 0 0 1	0 0 1 0
$p^{(5)}$	1 1 1 0 0	1 0 0 1 0

Fig. 9.7 Sequential multiplication of 2’s-complement numbers with right shifts (positive multiplier).

a	1 0 1 1 0	
x	1 0 1 0 1	
$p^{(0)}$	0 0 0 0 0	
$+x_0 a$	1 0 1 1 0	
$2p^{(1)}$	1 1 0 1 1 0	
$p^{(1)}$	1 1 0 1 1	0
$+x_1 a$	0 0 0 0 0	
$2p^{(2)}$	1 1 1 0 1 1	0
$p^{(2)}$	1 1 1 0 1	1 0
$+x_2 a$	1 0 1 1 0	
$2p^{(3)}$	1 1 0 0 1 1	1 0
$p^{(3)}$	1 1 0 0 1	1 1 0
$+x_3 a$	0 0 0 0 0	
$2p^{(4)}$	1 1 1 0 0 1	1 1 0
$p^{(4)}$	1 1 1 0 0	1 1 1 0
$+(-x_4 a)$	0 1 0 1 0	
$2p^{(5)}$	0 0 0 1 1 0	1 1 1 0
$p^{(5)}$	0 0 0 1 1	0 1 1 1 0

Fig. 9.8 Sequential multiplication of 2's-complement numbers with right shifts (negative multiplier).

Table 9.1 Radix-2 Booth's recoding

x_i	x_{i-1}	y_i	Explanation
0	0	0	No string of 1s in sight
0	1	1	End of string of 1s in x
1	0	$\bar{1}$	Beginning of string of 1s in x
1	1	0	Continuation of string of 1s in x

Example

	1 0 0 1	1 1 0 1	1 0 1 0	1 1 1 0	Operand x
(1)	$\bar{1}$ 0 1 0	0 $\bar{1}$ 1 0	$\bar{1}$ 1 $\bar{1}$ 1	0 0 $\bar{1}$ 0	Recoded version y

Justification

$$2^j + 2^{j-1} + \cdot \cdot \cdot + 2^{i+1} + 2^i \quad = \quad 2^{j+1} - 2^i$$

a	1 0 1 1 0		
x	1 0 1 0 1		Multiplier
y	$\bar{1}$ 1 $\bar{1}$ 1 $\bar{1}$		Booth-recoded
$p^{(0)}$	0 0 0 0 0		
$+y_0a$	0 1 0 1 0		
$2p^{(1)}$	0 0 1 0 1 0		
$p^{(1)}$	0 0 1 0 1	0	
$+y_1a$	1 0 1 1 0		
$2p^{(2)}$	1 1 1 0 1 1	0	
$p^{(2)}$	1 1 1 0 1	1 0	
$+y_2a$	0 1 0 1 0		
$2p^{(3)}$	0 0 0 1 1 1	1 0	
$p^{(3)}$	0 0 0 1 1	1 1 0	
$+y_3a$	1 0 1 1 0		
$2p^{(4)}$	1 1 1 0 0 1	1 1 0	
$p^{(4)}$	1 1 1 0 0	1 1 1 0	
$+y_4a$	0 1 0 1 0		
$2p^{(5)}$	0 0 0 1 1 0	1 1 1 0	
$p^{(5)}$	0 0 0 1 1	0 1 1 1 0	

Fig. 9.9 Sequential multiplication of 2’s-complement numbers with right shifts using Booth’s recoding.

9.5 Multiplication by Constants

Explicit multiplications, e.g. $y := 12 * x + 1$

Implicit multiplications, e.g. $A[i, j] := A[i, j] + B[i, j]$

Address of $A[i, j] = \text{base} + n * i + j$

Aspects of multiplication by integer constants:

Produce efficient code using as few registers as possible

Find the best code by a time- and space-efficient algorithm

Use binary expansion

Example: multiply R_1 by $113 = (1110001)_{two}$

$R_2 \leftarrow R_1$ shift-left 1
$R_3 \leftarrow R_2 + R_1$
$R_6 \leftarrow R_3$ shift-left 1
$R_7 \leftarrow R_6 + R_1$
$R_{112} \leftarrow R_7$ shift-left 4
$R_{113} \leftarrow R_{112} + R_1$

Only two registers are required; R_1 and another

Shorter sequence using shift-and-add instructions

$R_3 \leftarrow R_1$ shift-left 1 $+ R_1$
$R_7 \leftarrow R_3$ shift-left 1 $+ R_1$
$R_{113} \leftarrow R_7$ shift-left 4 $+ R_1$

Use of subtraction (Booth's recoding) may help

Example: multiply R_1 by $113 = (1110001)_{two} = (100^{-}10001)_{two}$

$$
\begin{aligned}
R_8 &\leftarrow R_1 \text{ shift-left } 3 \\
R_7 &\leftarrow R_8 - R_1 \\
R_{112} &\leftarrow R_7 \text{ shift-left } 4 \\
R_{113} &\leftarrow R_{112} + R_1
\end{aligned}
$$

Use of factoring may help

Example: multiply R_1 by $119 = 7 \times 17 = (8 - 1) \times (16 + 1)$

$$
\begin{aligned}
R_8 &\leftarrow R_1 \text{ shift-left } 3 \\
R_7 &\leftarrow R_8 - R_1 \\
R_{112} &\leftarrow R_7 \text{ shift-left } 4 \\
R_{119} &\leftarrow R_{112} + R_7
\end{aligned}
$$

Shorter sequence using shift-and-add/subtract instructions

$$
\begin{aligned}
R_7 &\leftarrow R_1 \text{ shift-left } 3 - R_1 \\
R_{119} &\leftarrow R_7 \text{ shift-left } 4 + R_7
\end{aligned}
$$

Factors of the form $2^b \pm 1$ translate directly into a shift followed by an add or subtract

Program execution time improvement by an optimizing compiler using the preceding methods: 20-60% [Bern86]

9.6 Preview of Fast Multipliers

Viewing multiplication as a multioperand addition problem, there are but two ways to speed it up

a. Reducing the number of operands to be added:
handling more than one multiplier bit at a time
(high-radix multipliers, Chapter 10)

b. Adding the operands faster:
parallel/pipelined multioperand addition
(tree and array multipliers, Chapter 11)

In Chapter 12, we cover all remaining multiplication topics, including bit-serial multipliers, multiply-add units, and the special case of squaring

10 High-Radix Multipliers

Chapter Goals

Study techniques that allow us to handle more than one multiplier bit in each cycle (two bits in radix 4, three bits in radix 8, . . .)

Chapter Highlights

High radix gives rise to "difficult" multiples
Recoding (change of digit-set) as remedy
Carry-save addition reduces the cycle time
Implementation and optimization methods

Chapter Contents

10.1 Radix-4 Multiplication
10.2 Modified Booth's Recoding
10.3 Using Carry-Save Adders
10.4 Radix-8 and Radix-16 Multipliers
10.5 Multibeat Multipliers
10.6 VLSI Complexity Issues

10.1 Radix-4 Multiplication

Radix-r versions of multiplication recurrences of Section 9.1

Multiplication with right shifts: top-to-bottom accumulation

$$p^{(j+1)} = (p^{(j)} + x_j\, a\, r^k)\, r^{-1} \qquad \text{with} \qquad p^{(0)} = 0 \quad \text{and}$$

l——add——l

l—shift right—l

$$p^{(k)} = p = ax + p^{(0)} r^{-k}$$

Multiplication with left shifts: bottom-to-top accumulation

$$p^{(j+1)} = r\, p^{(j)} + x_{k-j-1} a \qquad \text{with} \qquad p^{(0)} = 0 \quad \text{and}$$

lshiftl

l——add——l

$$p^{(k)} = p = ax + p^{(0)} r^{k}$$

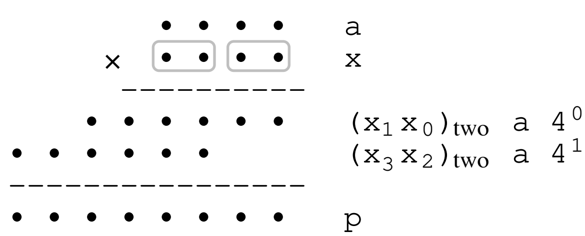


Fig. 10.1 Radix-4, or two-bit-at-a-time, multiplication in dot notation.

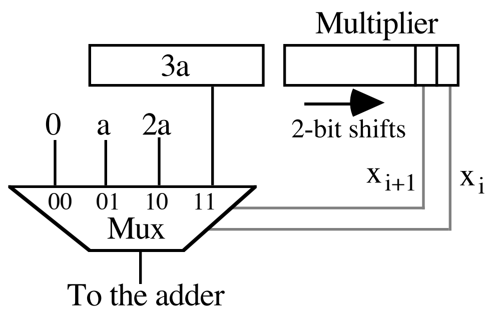


Fig. 10.2 The multiple generation part of a radix-4 multiplier with precomputation of 3a.

a	0 1 1 0
$3a$	0 1 0 0 1 0
x	1 1 1 0
$p^{(0)}$	0 0 0 0
$+(x_1x_0)_{two}a$	0 0 1 1 0 0
$4p^{(1)}$	0 0 1 1 0 0
$p^{(1)}$	0 0 1 1 0 0
$+(x_3x_2)_{two}a$	0 1 0 0 1 0
$4p^{(2)}$	0 1 0 1 0 1 0 0
$p^{(2)}$	0 1 0 1 0 1 0 0

Fig. 10.3 Example of radix-4 multiplication using the 3a multiple.

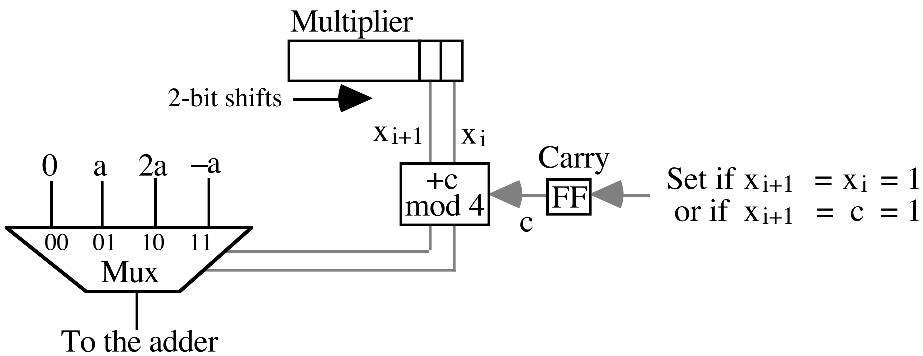


Fig. 10.4 The multiple generation part of a radix-4 multiplier based on replacing 3a with 4a (carry into next higher radix-4 multiplier digit) and –a.

10.2 Modified Booth's Recoding

Table 10.1 Radix-4 Booth's recoding yielding $(z_{k/2} \cdots z_1 z_0)_{four}$

x_{i+1}	x_i	x_{i-1}	y_{i+1}	y_i	$z_{i/2}$	Explanation
0	0	0	0	0	0	No string of 1s in sight
0	0	1	0	1	1	End of a string of 1s in x
0	1	0	0	1	1	Isolated 1 in x
0	1	1	1	0	2	End of a string of 1s in x
1	0	0	-1	0	-2	Beginning of a string of 1s in x
1	0	1	-1	1	-1	End one string, begin new one
1	1	0	0	-1	-1	Beginning of a string of 1s in x
1	1	1	0	0	0	Continuation of string of 1s in x

Example: $(21\ 31\ 22\ 32)_{four}$

	1 0	0 1	1 1	0 1	1 0	1 0	1 1	1 0	Operand x
(1)	-2	2	-1	2	-1	-1	0	-2	Recoded radix-4 version z

a	0 1 1 0	
x	1 0 1 0	
z	$\bar{1}$ $\bar{2}$	Recoded version of x
$p^{(0)}$	0 0 0 0 0 0	
$+z_0a$	1 1 0 1 0 0	
$4p^{(1)}$	1 1 0 1 0 0	
$p^{(1)}$	1 1 1 1 0 1 0 0	
$+z_1a$	1 1 1 0 1 0	
$4p^{(2)}$	1 1 0 1 1 1 0 0	
$p^{(2)}$	1 1 0 1 1 1 0 0	

Fig. 10.5 Example radix-4 multiplication with modified Booth’s recoding of the 2’s-complement multiplier.

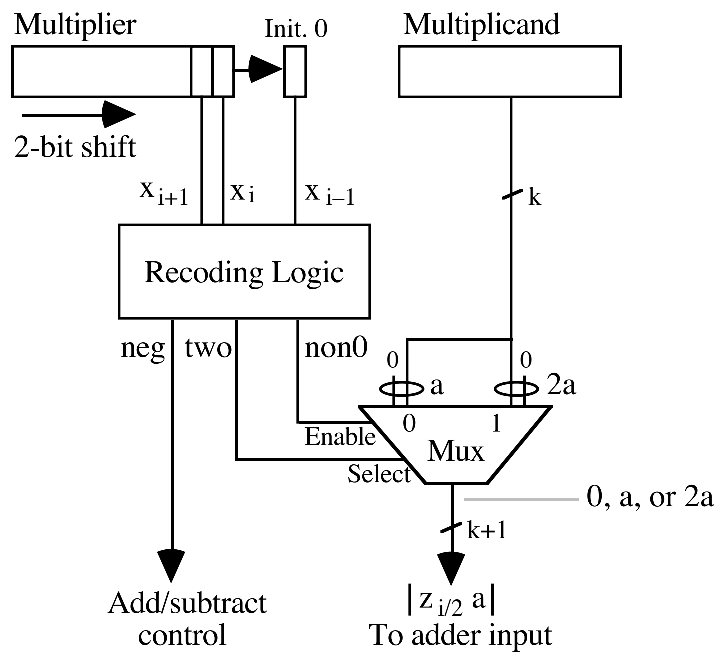


Fig. 10.6 The multiple generation part of a radix-4 multiplier based on Booth's recoding.

10.3 Using Carry-Save Adders

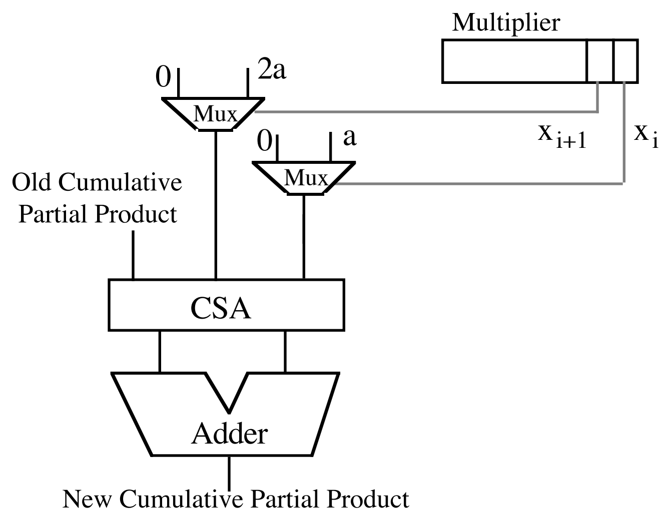


Fig. 10.7 Radix-4 multiplication with a carry-save adder used to combine the cumulative partial product, x_ia, and $2x_{i+1}$a into two numbers.

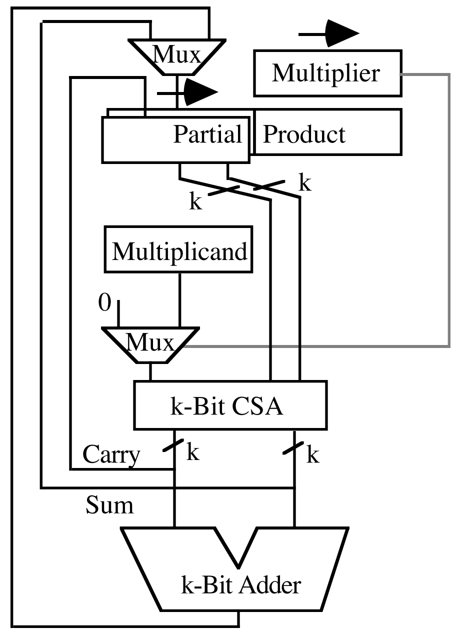


Fig. 10.8 Radix-2 multiplication with the upper half of the cumulative partial product in stored-carry form.

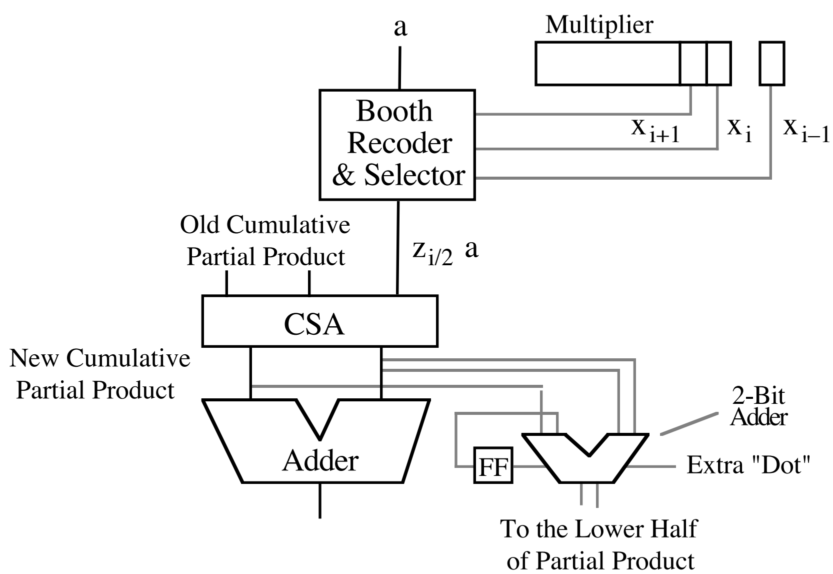


Fig. 10.9 Radix-4 multiplication with a carry-save adder used to combine the stored-carry cumulative partial product and $z_{i/2}a$ into two numbers.

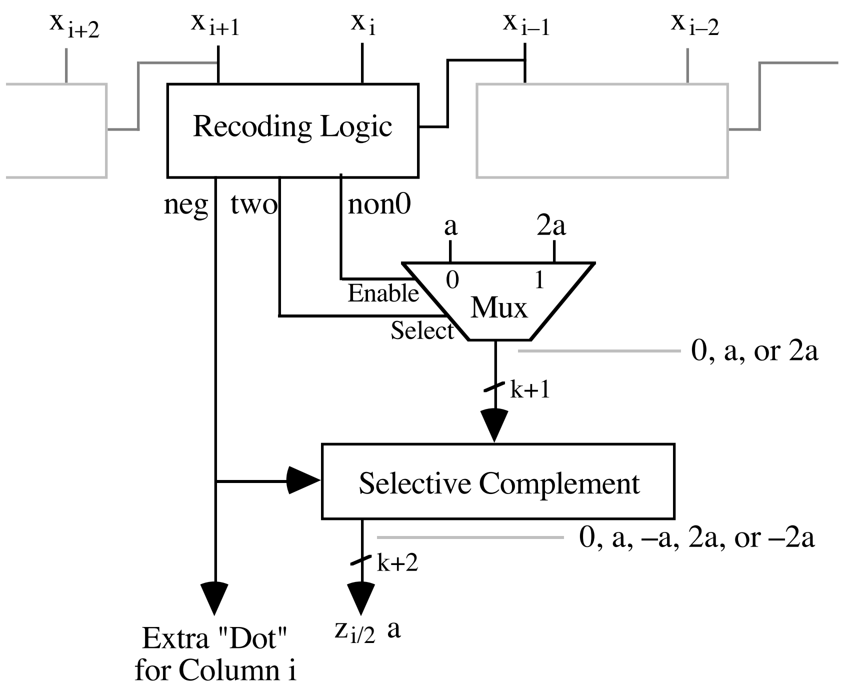


Fig. 10.10 Booth recoding and multiple selection logic for high-radix or parallel multiplication.

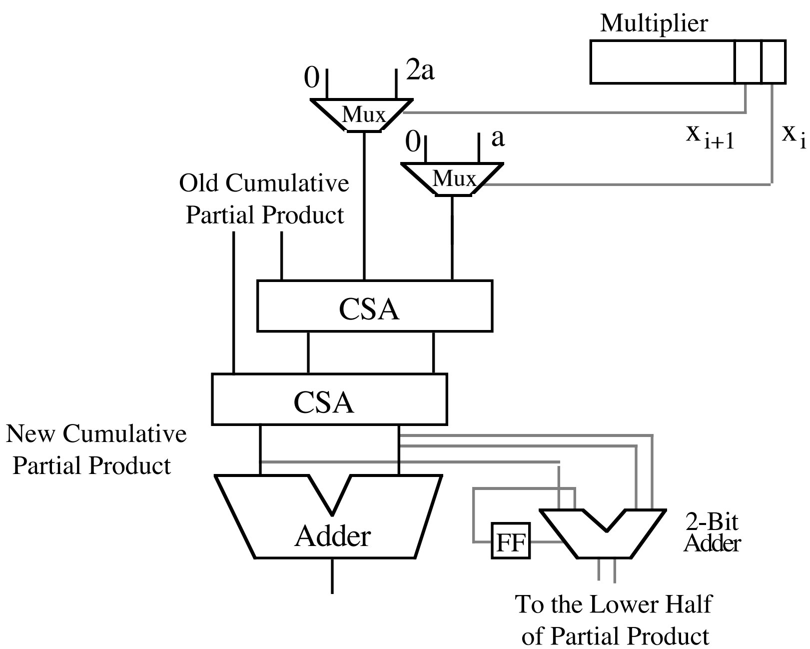


Fig. 10.11 Radix-4 multiplication with two carry-save adders.

10.4 Radix-8 and Radix-16 Multipliers

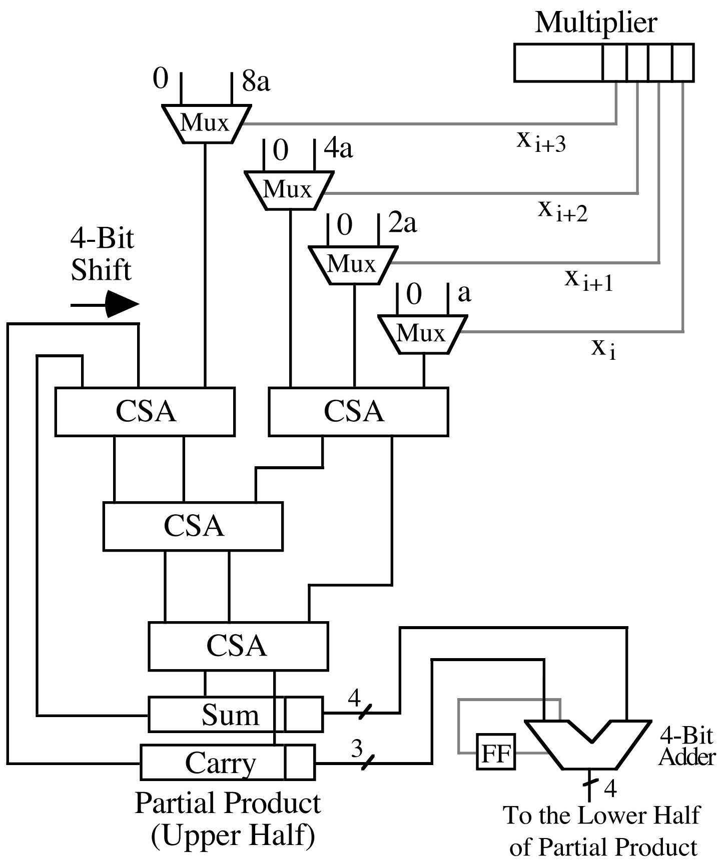


Fig. 10.12 Radix-16 multiplication with the upper half of the cumulative partial product in carry-save form.

Remove the mux corresponding to x_{i+3}

and the CSA right below it to get a radix-8 multiplier

(the cycle time will remain the same, though)

Must also modify the small adder at the lower right

Radix-16 multiplier design can become a radix-32 design if modified Booth's recoding is applied first

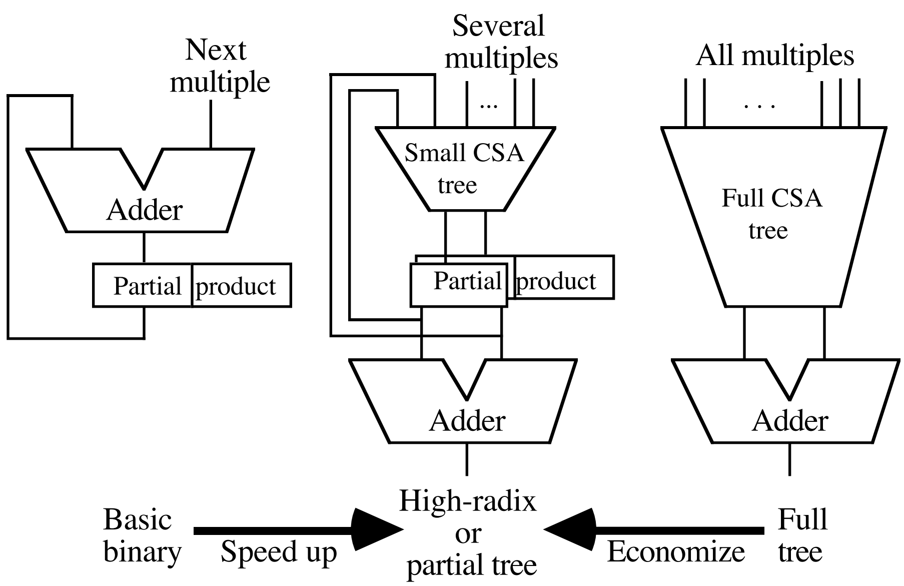


Fig. 10.13 High-radix multipliers as intermediate between sequential radix-2 and full-tree multipliers.

10.5 Multibeat Multipliers

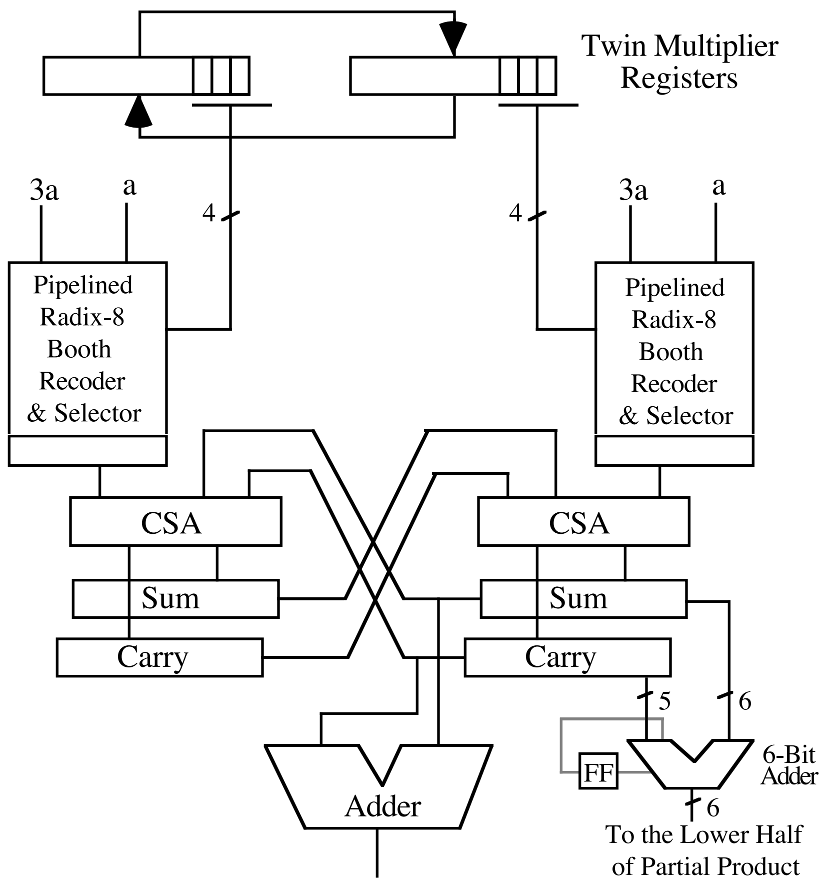


Fig. 10.14 Twin-beat multiplier with radix-8 Booth's recoding.

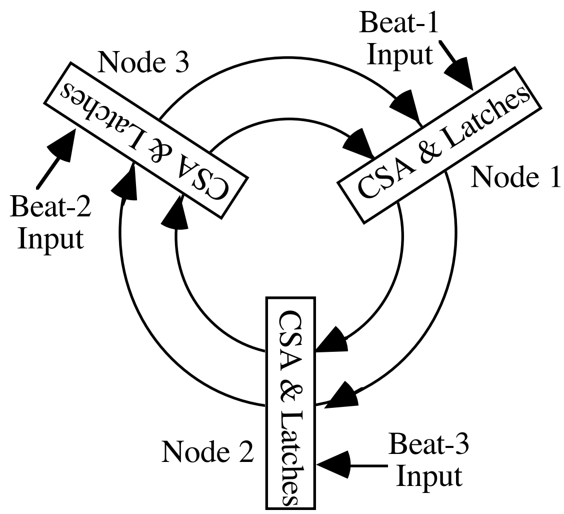


Fig. 10.15 Conceptual view of a three-beat multiplier.

10.6 VLSI Complexity Issues

Radix-2^b multiplication

bk two-input AND gates

O(bk) area for the CSA tree

Total area: $A = O(bk)$

Latency: $T = O((k/b) \log b + \log k)$

Any VLSI circuit computing the product of two k-bit integers must satisfy the following constraints

AT grows at least as fast as $k\sqrt{k}$.

AT^2 is at least proportional to k^2

For the preceding implementations, we have:

$$AT = O(k^2 \log b + bk \log k)$$

$$AT^2 = O((k^3/b) \log^2 b)$$

Suboptimality of radix-b multipliers

Low cost b constant	High speed $b = k$	Optimal wrt AT or AT^2
$AT = O(k^2)$	$AT = O(k^2 \log k)$	$AT = O(k\sqrt{k})$
$AT^2 = O(k^3)$	$AT^2 = O(k^2 \log^2 k)$	$AT^2 = O(k^2)$

Intermediate designs do not yield better AT and AT^2 values
The multipliers remain asymptotically suboptimal for any b

By the AT measure (indicator of cost-effectiveness)
slower radix-2 multipliers are better than
high-radix or tree multipliers

When many independent multiplications are required,
it may be appropriate to use the available chip area
for a large number of slow multipliers
as opposed to a small number of faster units

Latency of high-radix multipliers can actually be reduced
from $O((k/b) \log b + \log k)$ to $O(k/b + \log k)$
through more effective pipelining (Chapter 11)

11 Tree and Array Multipliers

Chapter Goals

Study the design of multipliers for highest possible performance (speed, throughput)

Chapter Highlights

Tree multiplier = reduction tree
+ redundant-to-binary converter

Avoiding full sign extension in multiplying signed numbers

Array multiplier = one-sided reduction tree
+ ripple-carry adder

Chapter Contents

11.1 Full-Tree Multipliers
11.2 Alternative Reduction Trees
11.3 Tree Multipliers for Signed Numbers
11.4 Partial-Tree Multipliers
11.5 Array Multipliers
11.6 Pipelined Tree and Array Multipliers

11.1 Full-Tree Multipliers

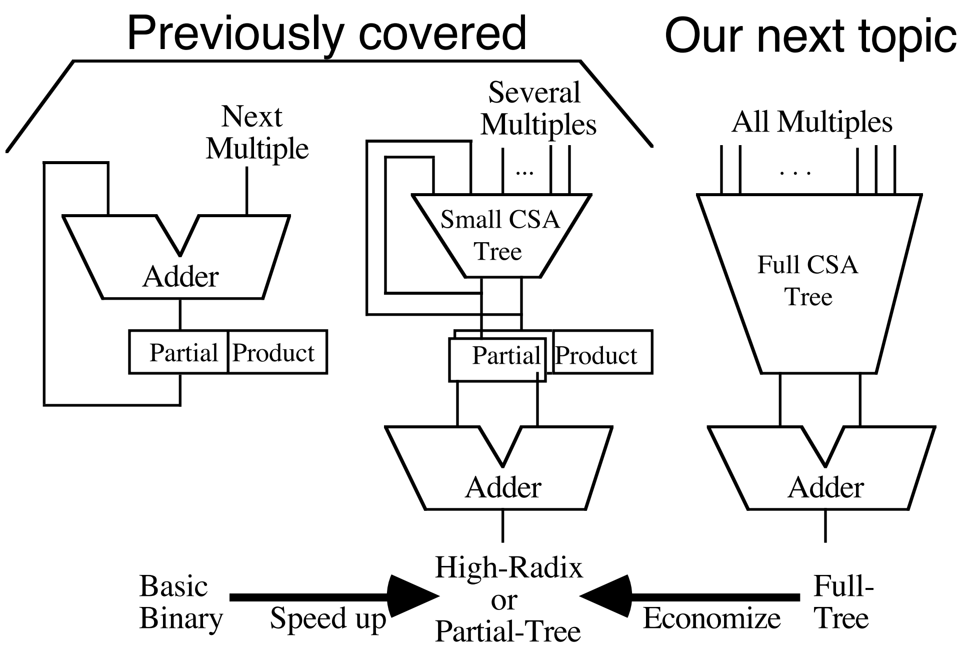


Fig. 10.13 High-radix multipliers as intermediate between sequential radix-2 and full-tree multipliers.

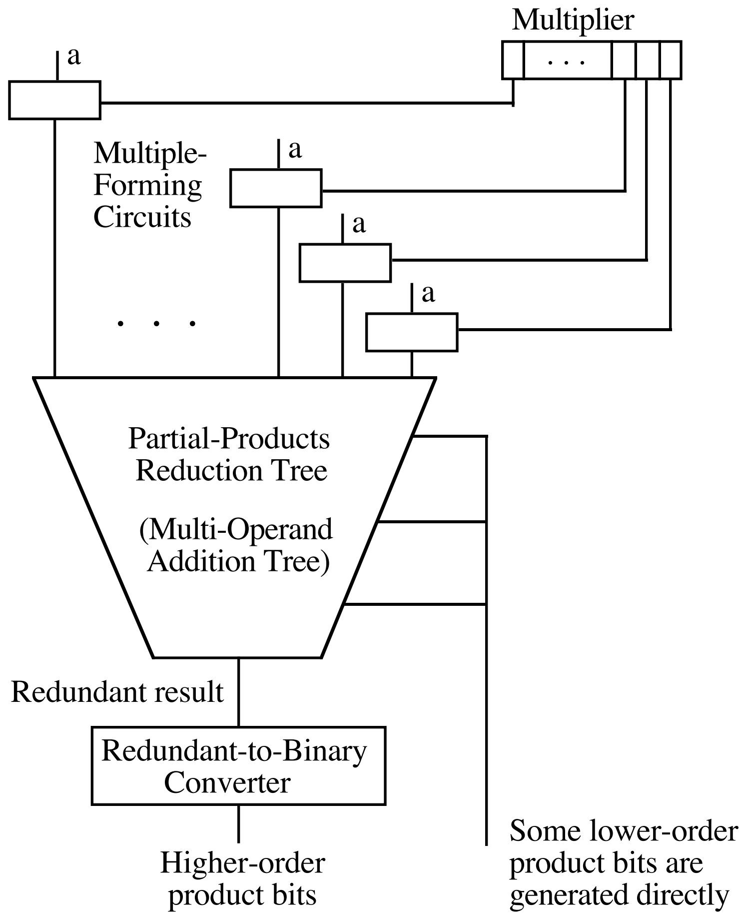


Fig. 11.1 General structure of a full-tree multiplier.

Variations in tree multipliers are distinguished by the designs of the following three elements:

multiple-forming circuits

partial products reduction tree

redundant-to-binary converter

Wallace Tree
(5 FAs + 3 HAs + 4-Bit Adder)

```
   1  2  3  4  3  2  1
         FA FA FA HA
  --------------------
   1  3  2  3  2  1  1
      FA HA FA HA
----------------------
   2  2  2  2  1  1  1
   4-Bit Adder
----------------------
1  1  1  1  1  1  1  1
```

Dadda Tree
(4 FAs + 2 HAs + 6-Bit Adder)

```
   1  2  3  4  3  2  1
         FA FA
  --------------------
   1  3  2  2  3  2  1
      FA HA HA FA
----------------------
   2  2  2  2  1  2  1
     6-Bit Adder
----------------------
1  1  1  1  1  1  1  1
```

Fig. 11.2 Two different binary 4 × 4 tree multipliers.

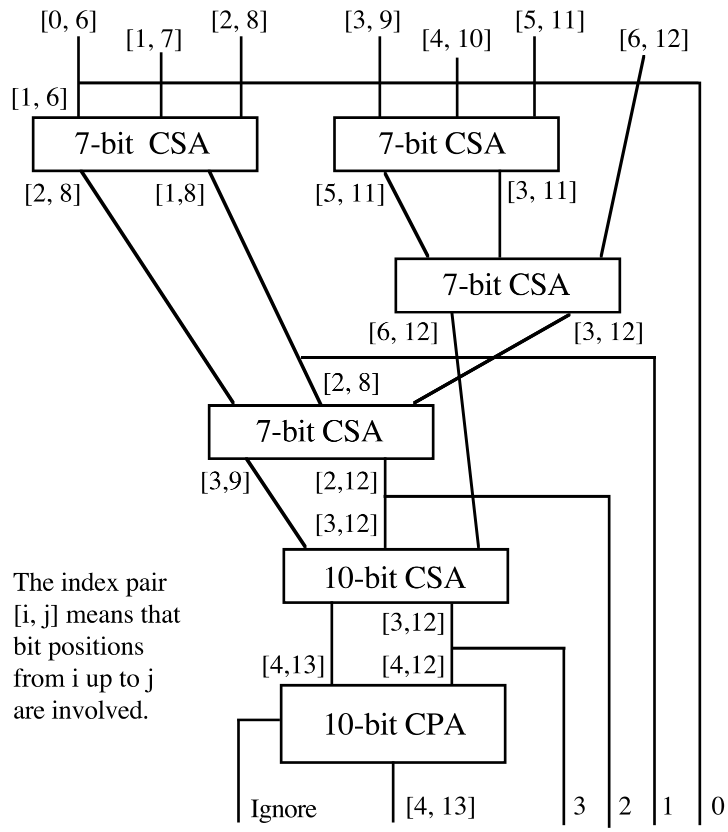


Fig. 11.3 Possible CSA tree for a 7×7 tree multiplier.

CSA trees are generally quite irregular,
thus causing problems in VLSI implementation

11.2 Alternative Reduction Trees

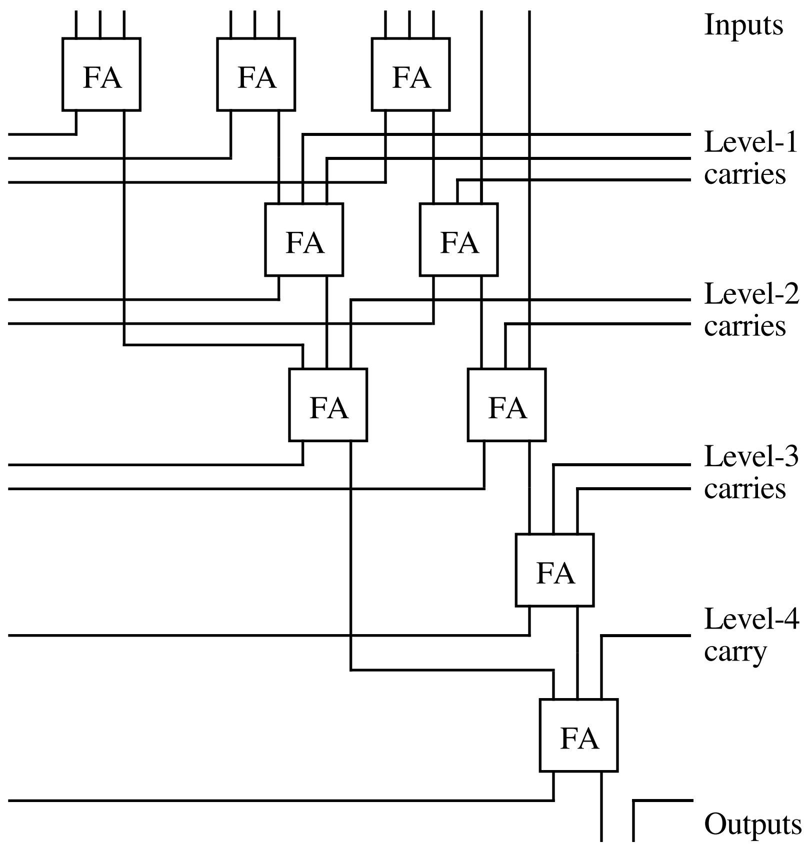


Fig. 11.4 A slice of a balanced-delay tree for 11 inputs.

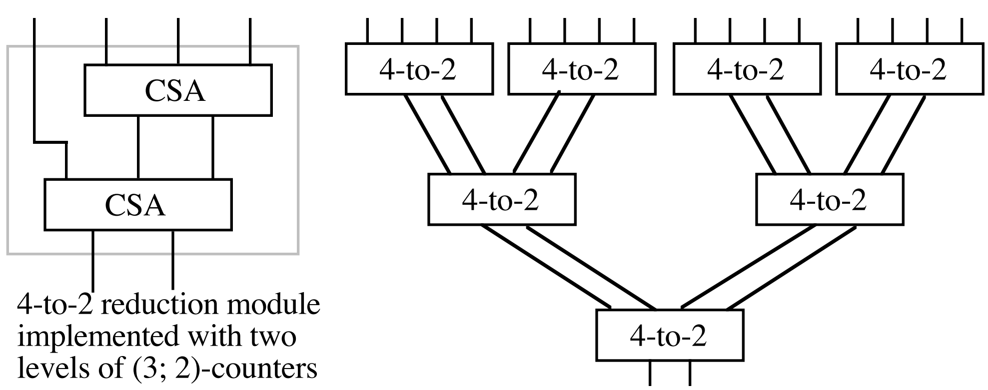


Fig. 11.5 Tree multiplier with a more regular structure based on 4-to-2 reduction modules.

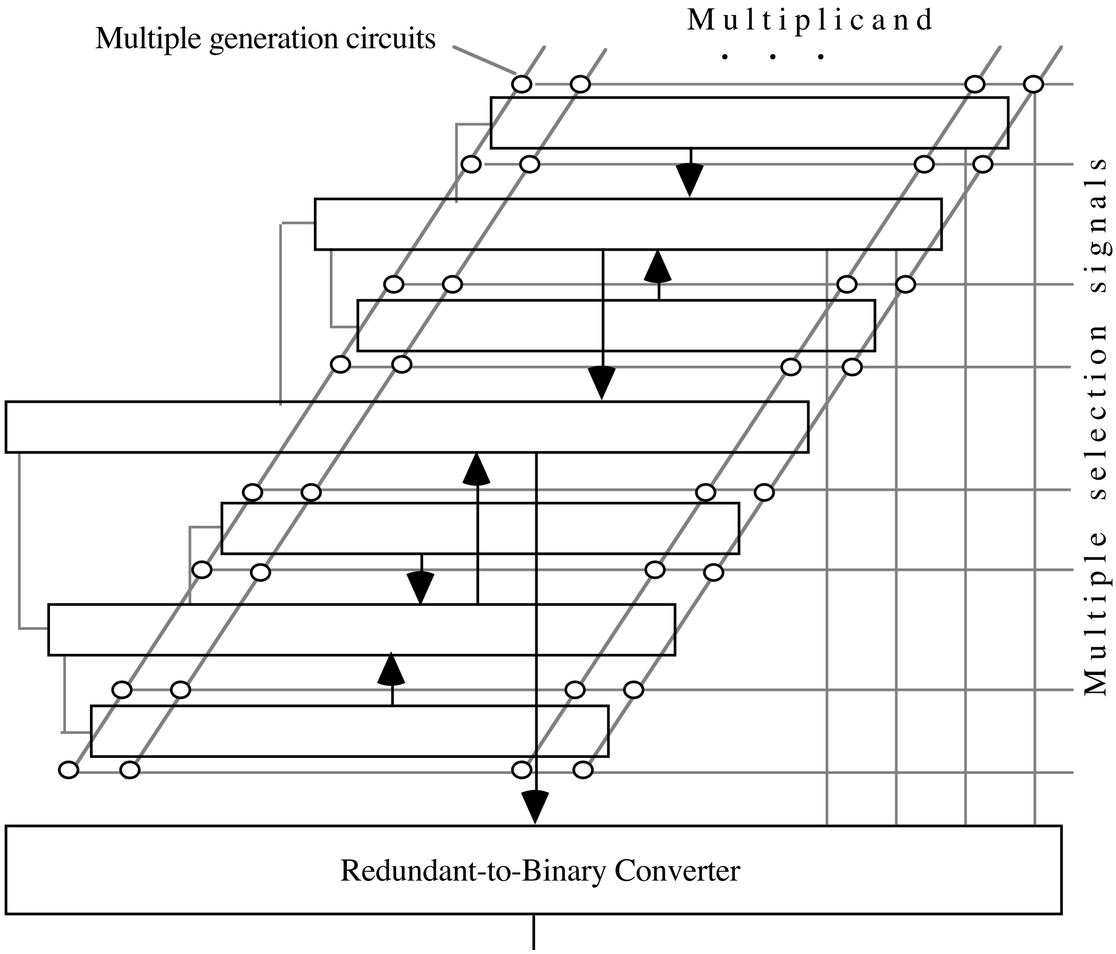


Fig. 11.6 Layout of a partial-products reduction tree composed of 4-to-2 reduction modules. Each solid arrow represents two numbers.

If 4-to-2 reduction is done by using two CSAs,
the binary tree will often have more CSA levels,
but regularity will improve

Use of Booth's recoding reduces the gate complexity
but may not prove worthwhile due to irregularity

11.3 Tree Multipliers for Signed Numbers

Sign extension in multioperand addition (from Fig. 8.18)

Extended positions					Sign	Magnitude positions			
x_{k-1}	x_{k-1}	x_{k-1}	x_{k-1}	x_{k-1}	x_{k-1}	x_{k-2}	x_{k-3}	x_{k-4}	. . .
y_{k-1}	y_{k-1}	y_{k-1}	y_{k-1}	y_{k-1}	y_{k-1}	y_{k-2}	y_{k-3}	y_{k-4}	. . .
z_{k-1}	z_{k-1}	z_{k-1}	z_{k-1}	z_{k-1}	z_{k-1}	z_{k-2}	z_{k-3}	z_{k-4}	. . .

The difference in multiplication is the shifting sign positions

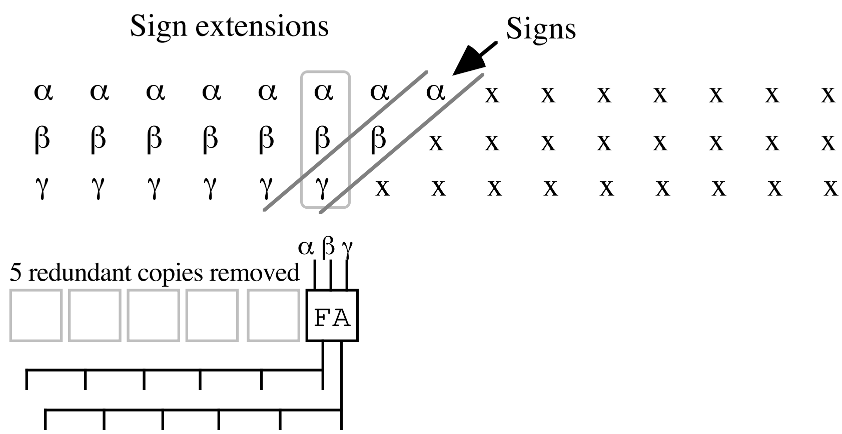


Fig. 11.7 Sharing of full adders to reduce the CSA width in a signed tree multiplier.

Unsigned

					a_4	a_3	a_2	a_1	a_0
				$\times$	x_4	x_3	x_2	x_1	x_0
					a_4x_0	a_3x_0	a_2x_0	a_1x_0	a_0x_0
				a_4x_1	a_3x_1	a_2x_1	a_1x_1	a_0x_1	
			a_4x_2	a_3x_2	a_2x_2	a_1x_2	a_0x_2		
		a_4x_3	a_3x_3	a_2x_3	a_1x_3	a_0x_3			
	a_4x_4	a_3x_4	a_2x_4	a_1x_4	a_0x_4				
p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

2's-complement

					a_4	a_3	a_2	a_1	a_0
				$\times$	x_4	x_3	x_2	x_1	x_0
					$-a_4x_0$	a_3x_0	a_2x_0	a_1x_0	a_0x_0
				$-a_4x_1$	a_3x_1	a_2x_1	a_1x_1	a_0x_1	
			$-a_4x_2$	a_3x_2	a_2x_2	a_1x_2	a_0x_2		
		$-a_4x_3$	a_3x_3	a_2x_3	a_1x_3	a_0x_3			
	a_4x_4	$-a_3x_4$	$-a_2x_4$	$-a_1x_4$	$-a_0x_4$				
p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

Baugh-Wooley

					a_4	a_3	a_2	a_1	a_0
				$\times$	x_4	x_3	x_2	x_1	x_0
					$a_4\bar{x}_0$	a_3x_0	a_2x_0	a_1x_0	a_0x_0
				$a_4\bar{x}_1$	a_3x_1	a_2x_1	a_1x_1	a_0x_1	
			$a_4\bar{x}_2$	a_3x_2	a_2x_2	a_1x_2	a_0x_2		
		$a_4\bar{x}_3$	a_3x_3	a_2x_3	a_1x_3	a_0x_3			
	a_4x_4	$\bar{a}_3x_4$	$\bar{a}_2x_4$	$\bar{a}_1x_4$	$\bar{a}_0x_4$				
	$\bar{a}_4$				a_4				
1	$\bar{x}_4$				x_4				
p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

Modified B-W

					a_4	a_3	a_2	a_1	a_0
				$\times$	x_4	x_3	x_2	x_1	x_0
					$\overline{a_4x_0}$	a_3x_0	a_2x_0	a_1x_0	a_0x_0
				$\overline{a_4x_1}$	a_3x_1	a_2x_1	a_1x_1	a_0x_1	
			$\overline{a_4x_2}$	a_3x_2	a_2x_2	a_1x_2	a_0x_2		
		$\overline{a_4x_3}$	a_3x_3	a_2x_3	a_1x_3	a_0x_3			
	a_4x_4	$\overline{a_3x_4}$	$\overline{a_2x_4}$	$\overline{a_1x_4}$	$\overline{a_0x_4}$				
1				1					
p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

Fig. 11.8 Baugh-Wooley 2’s-complement multiplication.

$$-a_4 x_0 = a_4(1 - x_0) - a_4 = a_4 \bar{x}_0 - a_4$$

11.4 Partial-Tree Multipliers

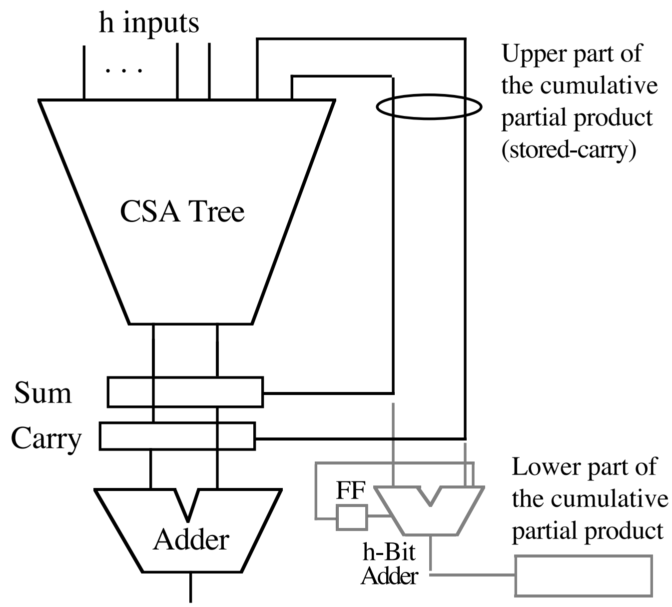


Fig. 11.9 General structure of a partial-tree multiplier.

High-radix versus partial-tree multipliers

difference is quantitative rather than qualitative

for small h, say < 8 bits, the multiplier of Fig. 11.9

is viewed as a high-radix multiplier

when h is a significant fraction of k, say k/2 or k/4,

then we view it as a partial-tree multiplier

11.4 Array Multipliers

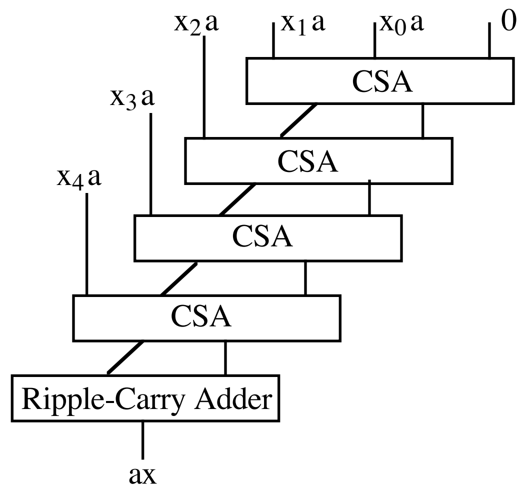


Fig. 11.10 A basic array multiplier uses a one-sided CSA tree and a ripple-carry adder.

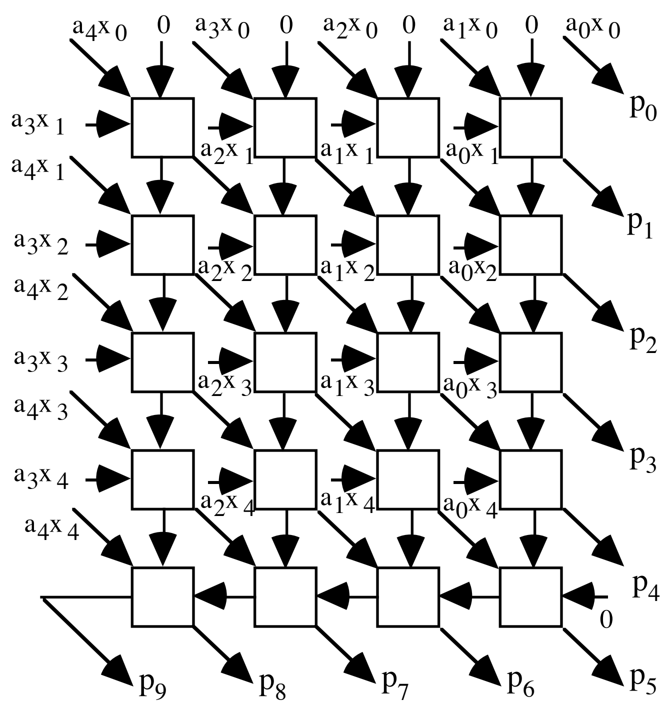


Fig. 11.11 Details of a 5 × 5 array multiplier using FA blocks.

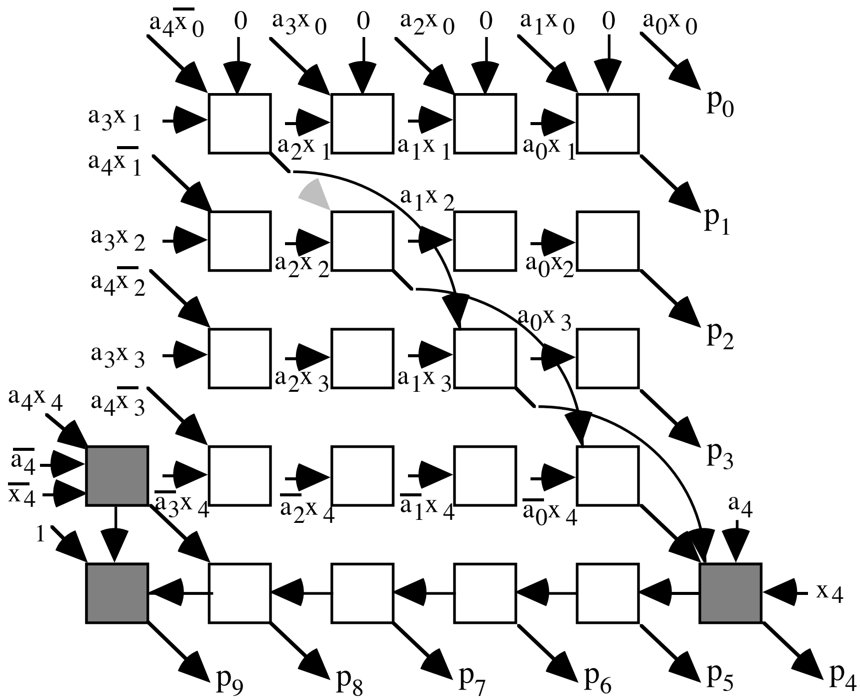


Fig. 11.12 Modifications in a 5 × 5 array multiplier to deal with 2’s-complement inputs using the Baugh-Wooley method or to shorten the critical path.

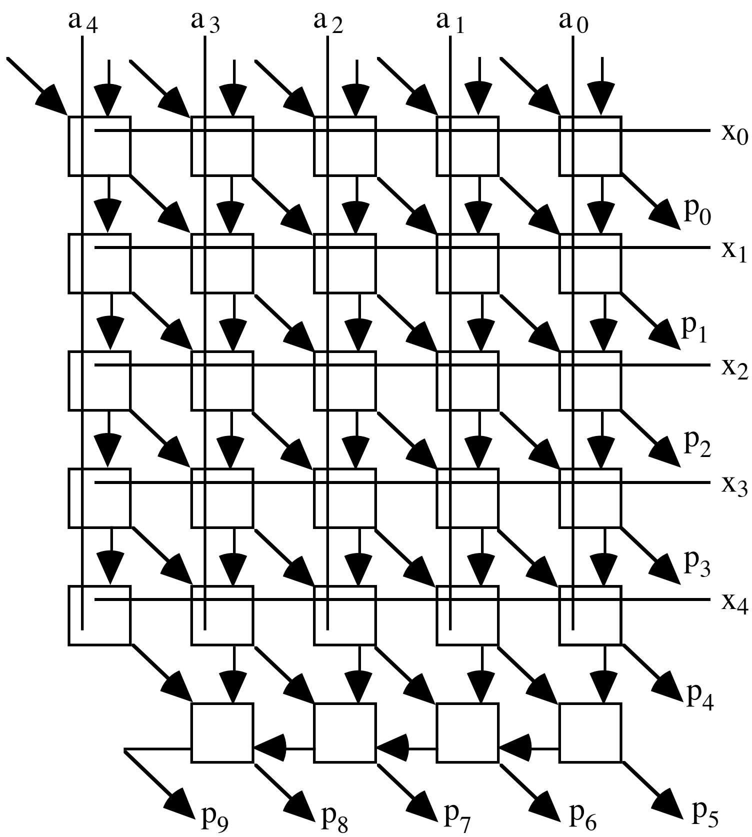


Fig. 11.13 Design of a 5 × 5 array multiplier with two additive inputs and full-adder blocks that include AND gates.

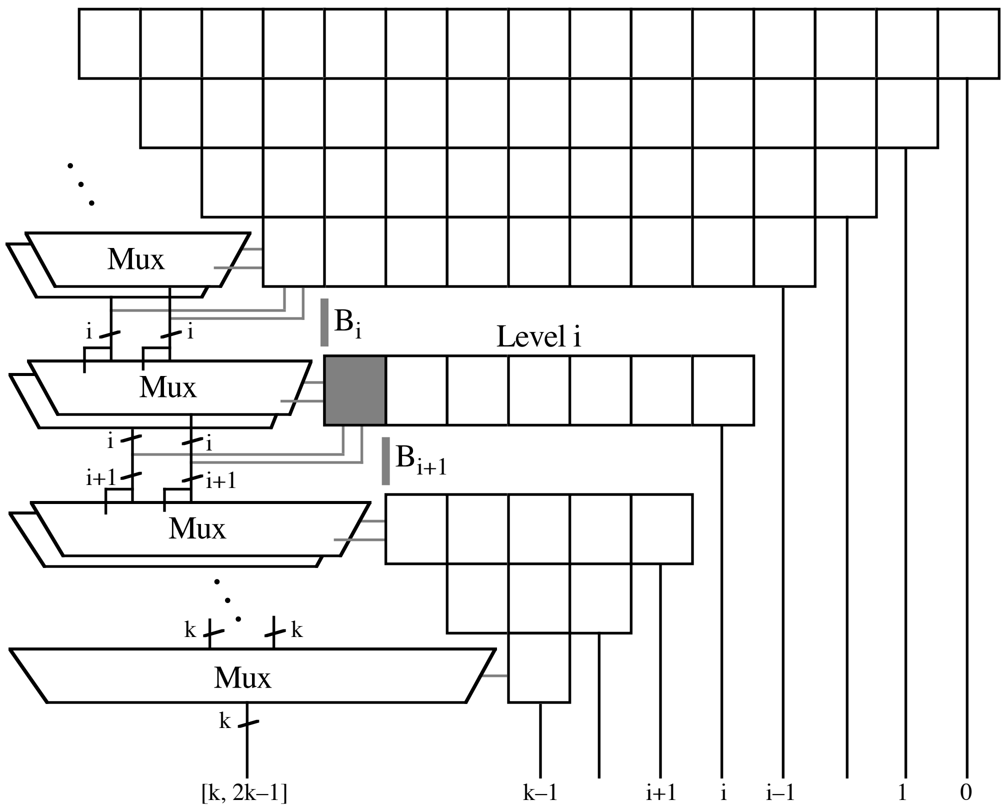


Fig. 11.14 Conceptual view of a modified array multiplier that does not need a final carry-propagate adder.

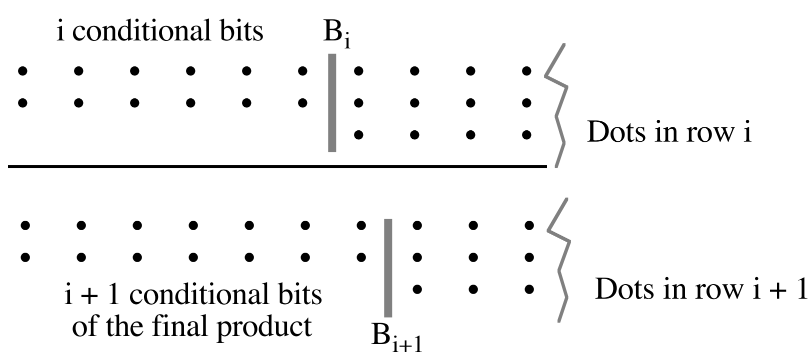


Fig. 11.15 Carry-save addition, performed in level i, extends the conditionally computed bits of the final product.

11.6 Pipelined Tree and Array Multipliers

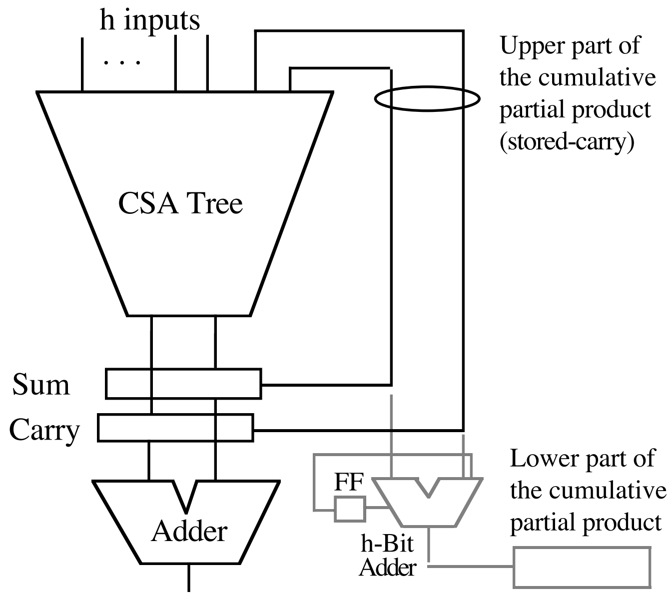


Fig. 11.9 General structure of a partial-tree multiplier.

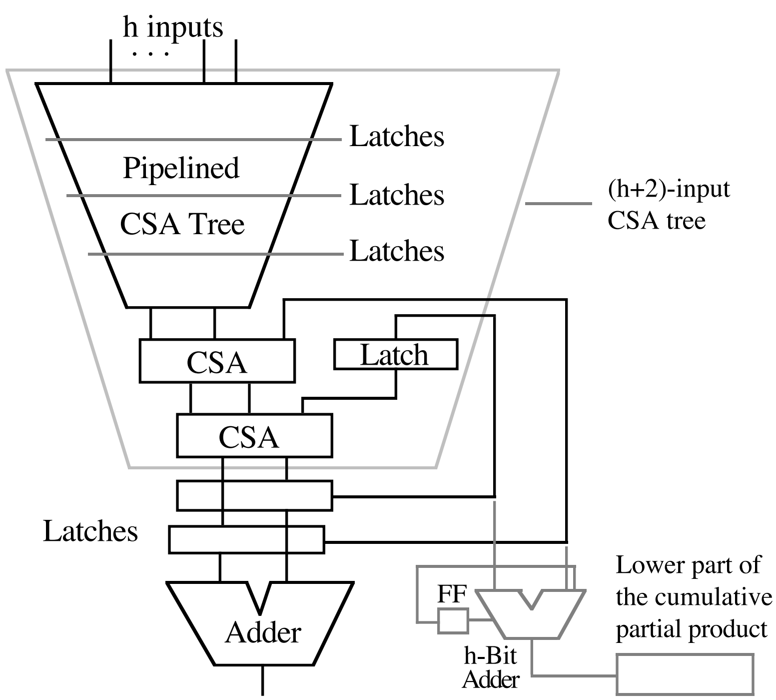


Fig. 11.16 Efficiently pipelined partial-tree multiplier.

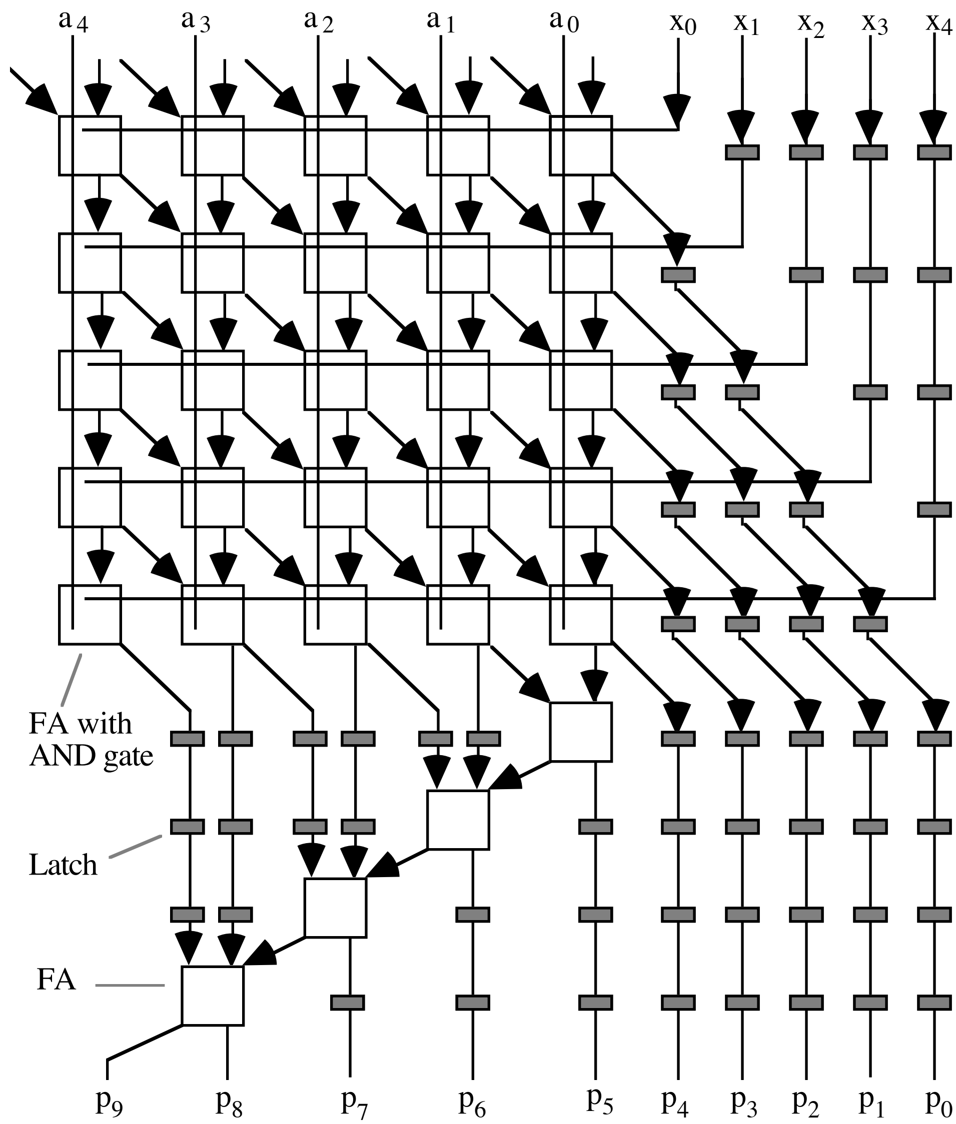


Fig. 11.17 Pipelined 5 × 5 array multiplier using latched FA blocks. The small shaded boxes are latches.

12 Variations in Multipliers

Chapter Goals

Learn additional methods for synthesizing fast multipliers as well as other types of multipliers (bit-serial, modular, etc.)

Chapter Highlights

Synthesizing a multiplier from smaller units
Performing multiply-add as one operation
Bit-serial and (semi)systolic multipliers
Using a multiplier for squaring is wasteful

Chapter Contents

12.1 Divide-and-Conquer Designs
12.2 Additive Multiply Modules
12.3 Bit-Serial Multipliers
12.4 Modular Multipliers
12.5 The Special Case of Squaring
12.6 Combined Multiply-Add Units

12.1 Divide-and-Conquer Designs

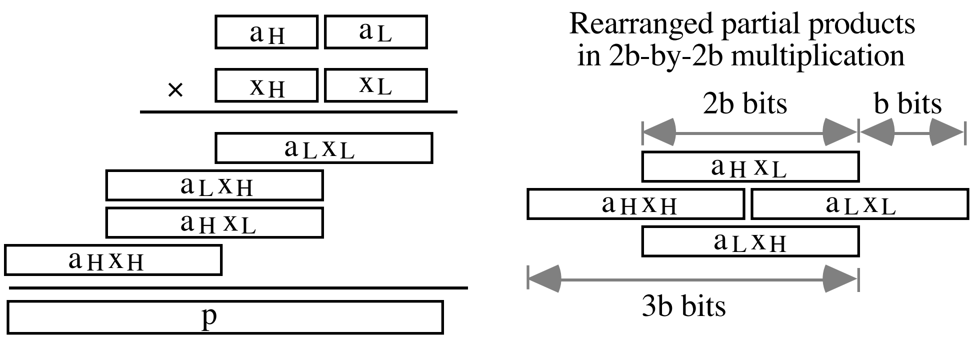


Fig. 12.1 Divide-and-conquer strategy for synthesizing a 2b×2b multiplier from b×b multipliers.

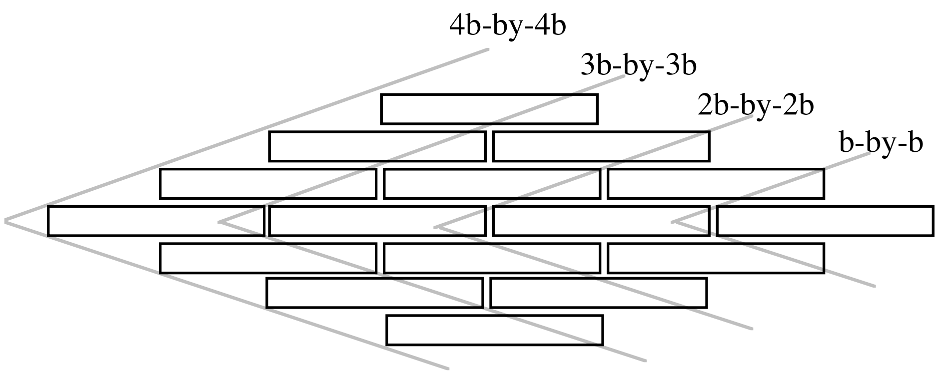


Fig. 12.2 Using b×b multipliers to synthesize 2b×2b, 3b×3b, and 4b×4b multipliers.

2b × 2b	use (3; 2)-counters
3b × 3b	use (5; 2)-counters
4b × 4b	use (7; 2)-counters

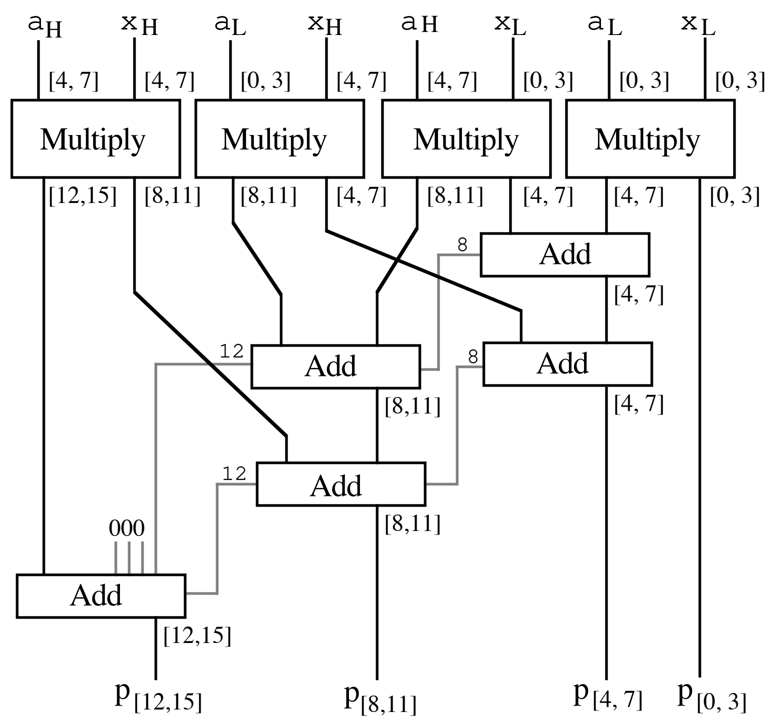


Fig. 12.3 Using 4×4 multipliers and 4-bit adders to synthesize an 8×8 multiplier.

Generalization

$2b \times 2c$	use $b \times c$ multipliers and (3; 2)-counters
$2b \times 4c$	use $b \times c$ multipliers and (5; 2)-counters
$gb \times hc$	use $b \times c$ multipliers and (?; 2)-counters

12.2 Additive Multiply Modules

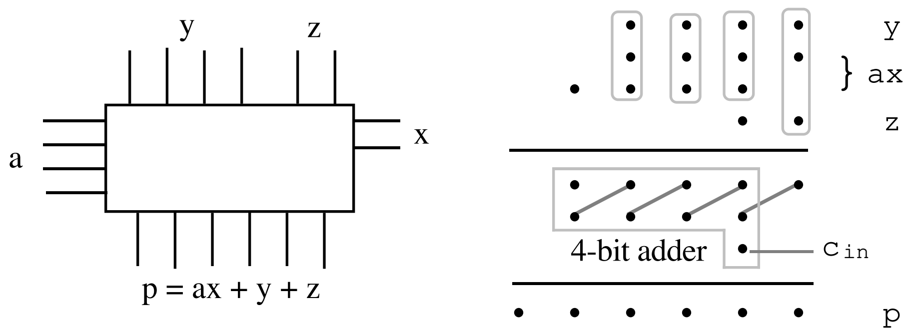


Fig. 12.4 Additive multiply module with 2 × 4 multiplier (ax) plus 4-bit and 2-bit additive inputs (y and z).

b × c AMM

b-bit and c-bit multiplicative inputs
b-bit additive input
c-bit additive input
(b + c)-bit output

$$(2^b - 1) \times (2^c - 1) + (2^b - 1) + (2^c - 1) = 2^{b+c} - 1$$

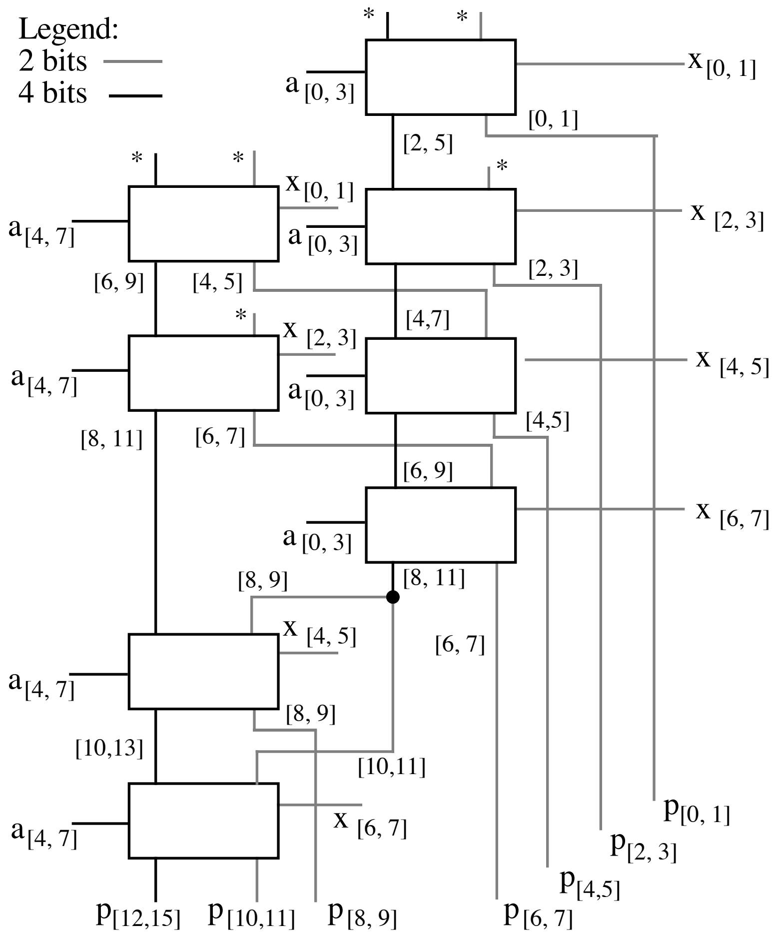


Fig. 12.5 An 8 × 8 multiplier built of 4 × 2 AMMs. Inputs marked with an asterisk carry 0s.

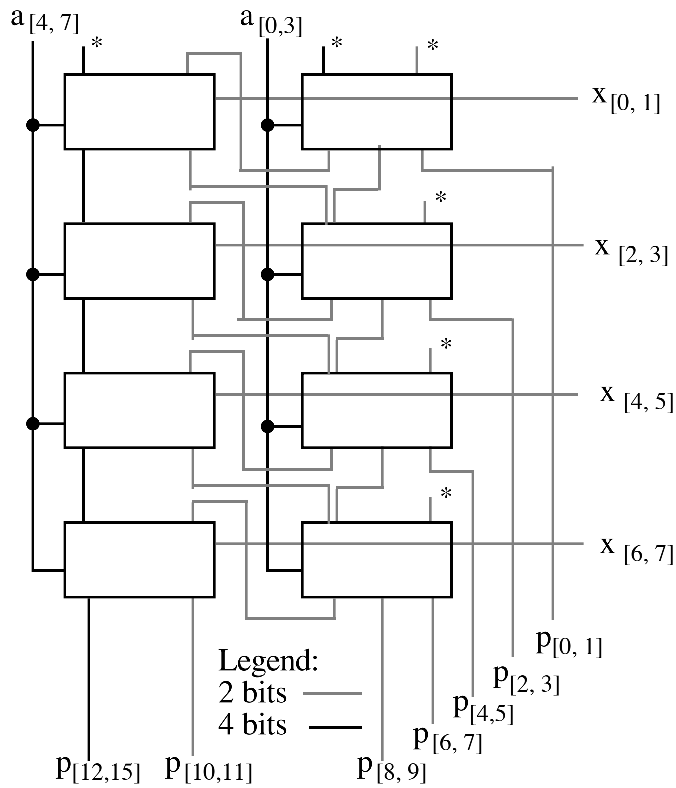


Fig. 12.6 Alternate 8 × 8 multiplier design based on 4 × 2 AMMs. Inputs marked with an asterisk carry 0s.

12.3 Bit-Serial Multipliers

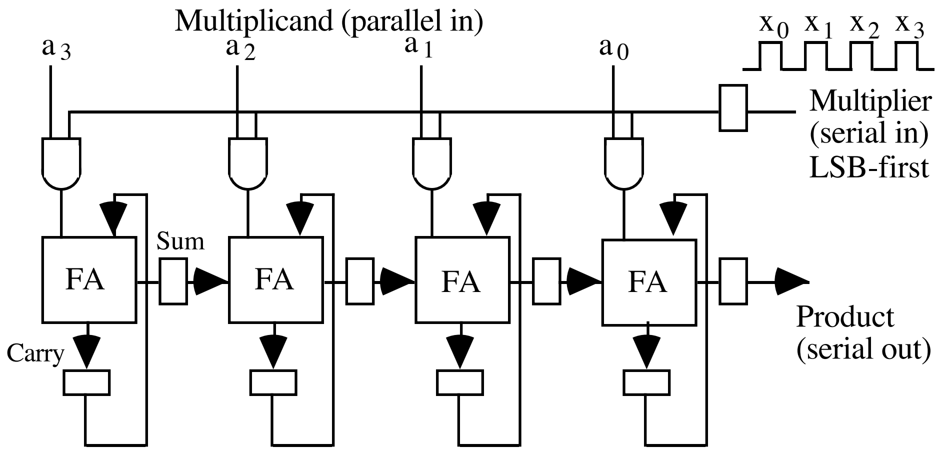


Fig. 12.7 Semi-systolic circuit for 4×4 multiplication in 8 clock cycles.

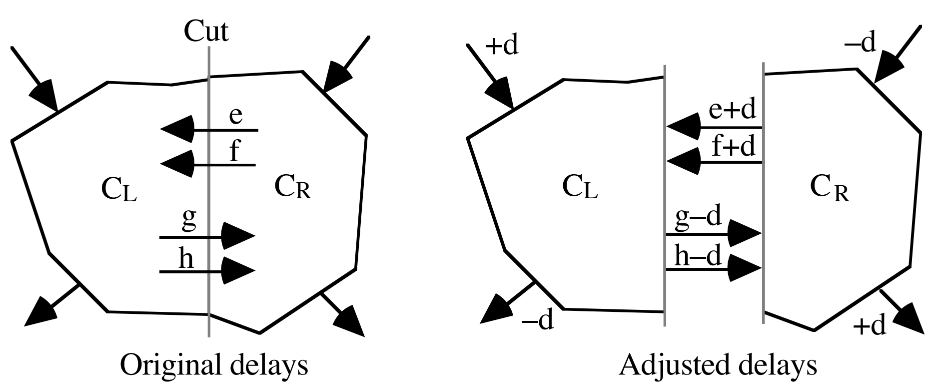


Fig. 12.8 Example of retiming by delaying the inputs to C_L and advancing the outputs from C_L by d units.

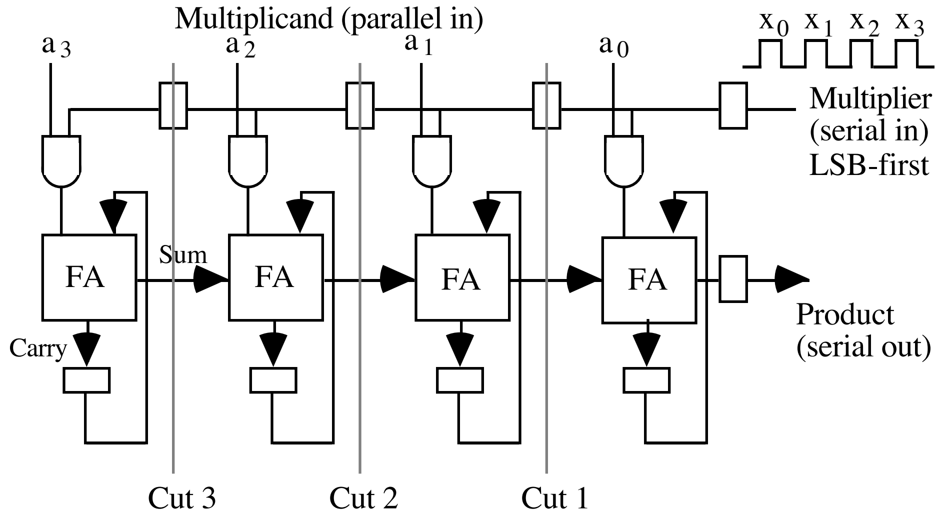


Fig. 12.9 A retimed version of our semi-systolic multiplier.

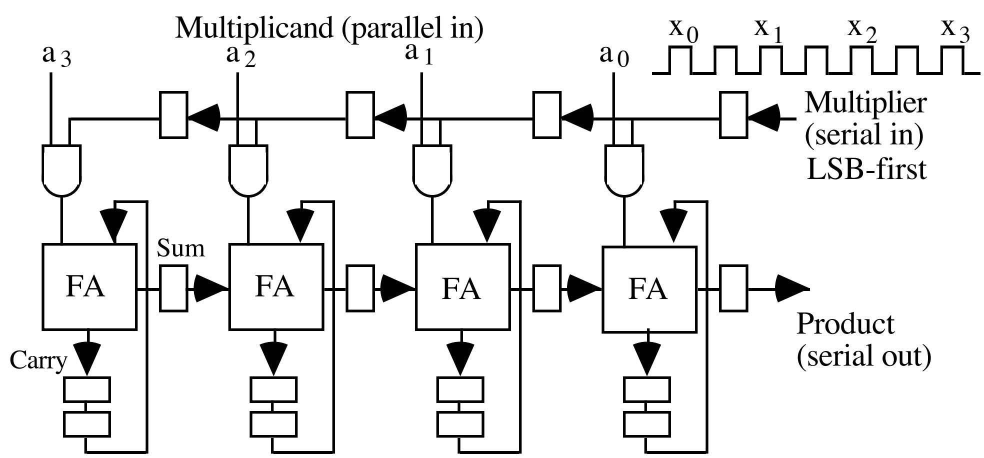


Fig. 12.10 Systolic circuit for 4×4 multiplication in 15 cycles.

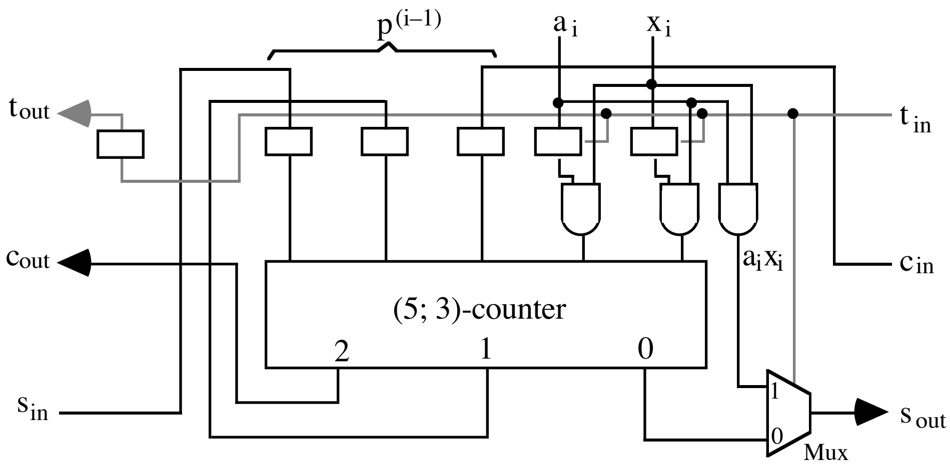


Fig. 12.11 Building block for a latency-free bit-serial multiplier.

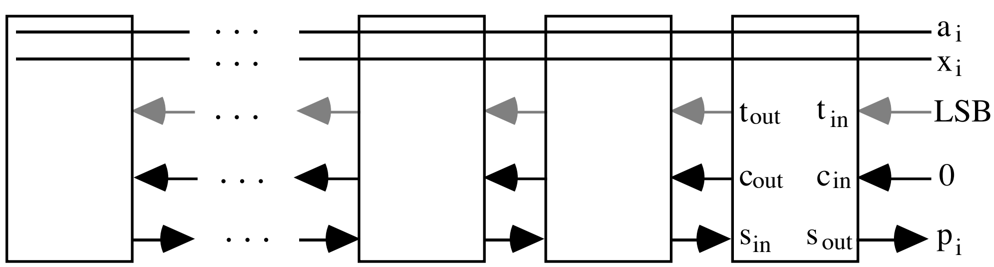


Fig. 12.12 The cellular structure of the bit-serial multiplier based on the cell in Fig. 12.11.

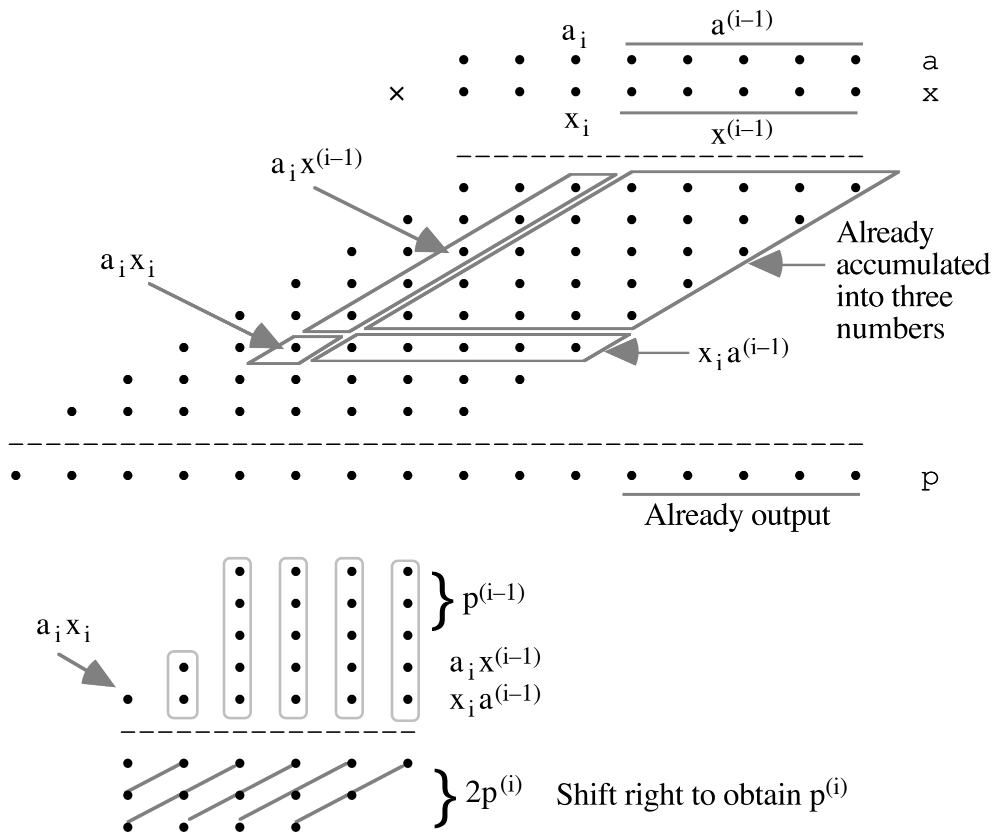


Fig. 12.13 Bit-serial multiplier design in dot notation.

12.4 Modular Multipliers

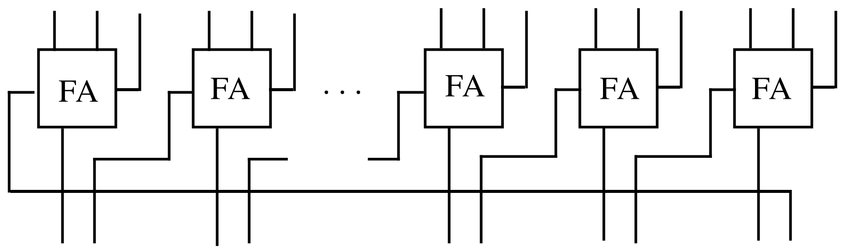


Fig. 12.14 Modulo-(2^b – 1) carry-save adder.

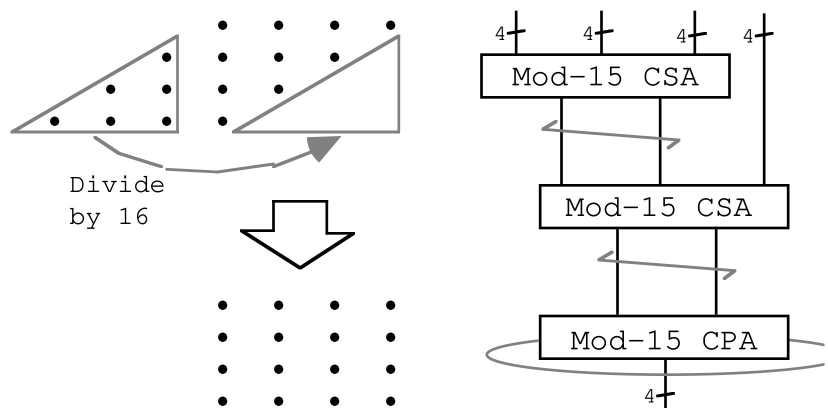


Fig. 12.15 Design of a 4 × 4 modulo-15 multiplier.

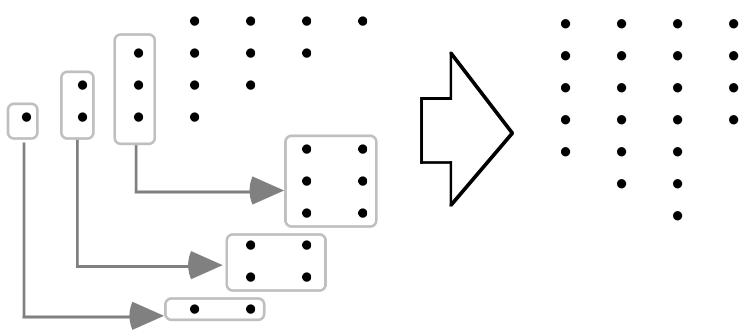


Fig. 12.16 One way to design of a 4 × 4 modulo-13 multiplier.

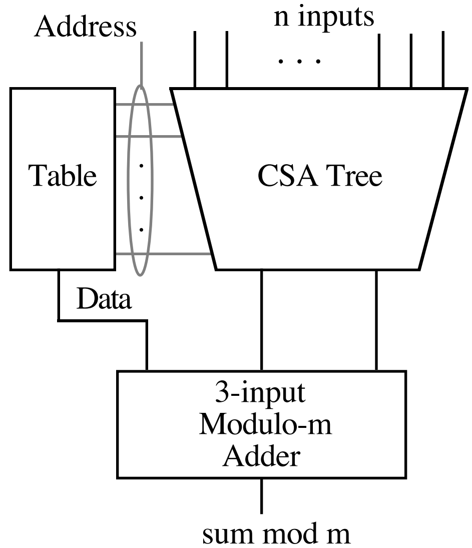


Fig. 12.17 A method for modular multioperand addition.

12.5 The Special Case of Squaring

Multiply x by x					x_4	x_3	x_2	x_1	x_0
				×	x_4	x_3	x_2	x_1	x_0
					x_4x_0	x_3x_0	x_2x_0	$\boxed{x_1x_0}$	$\boxed{x_0x_0}$
				x_4x_1	x_3x_1	x_2x_1	x_1x_1	$\boxed{x_0x_1}$	Reduce to x_0
			x_4x_2	x_3x_2	x_2x_2	x_1x_2	x_0x_2	Move to next column	
		x_4x_3	x_3x_3	x_2x_3	x_1x_3	x_0x_3			
	x_4x_4	x_3x_4	x_2x_4	x_1x_4	x_0x_4				
p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

Reduce the bit matrix					x_4	x_3	x_2	x_1	x_0
				×	x_4	x_3	x_2	x_1	x_0
	x_4x_3	x_4x_2	x_4x_1	x_4x_0	x_3x_0	x_2x_0	x_1x_0	–	x_0
	x_4		x_3x_2	x_3x_1	x_2x_1		x_1		
			x_3		x_2				
p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	0	x_0

Fig. 12.18 Design of a 5-bit squarer.

12.6 Combined Multiply-Add Units

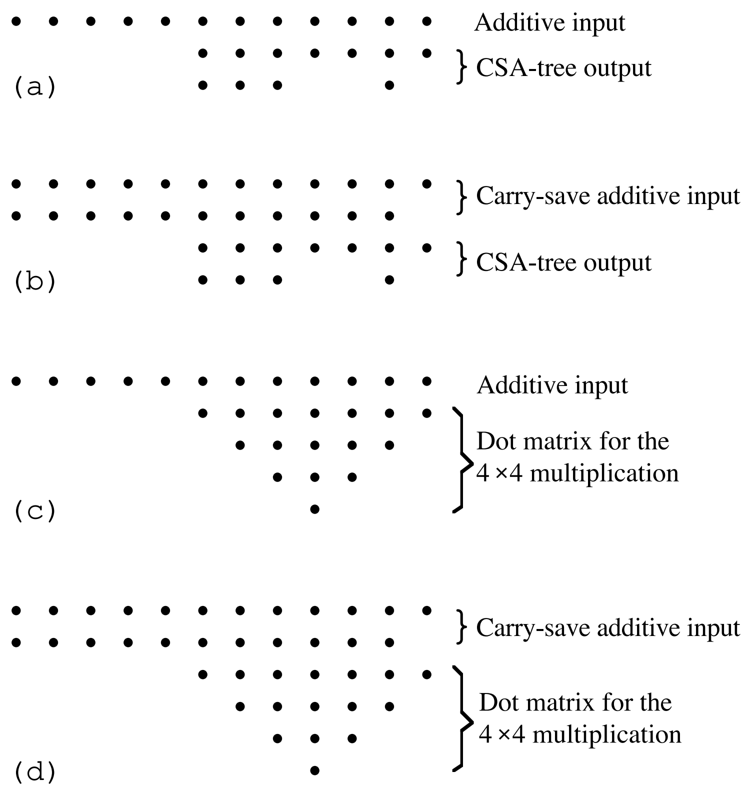


Fig. 12.19 Dot-notation representations of various methods for performing a multiply-add operation in hardware.

Multiply-add versus multiply-accumulate

Multiply-accumulate units often have wider additive inputs

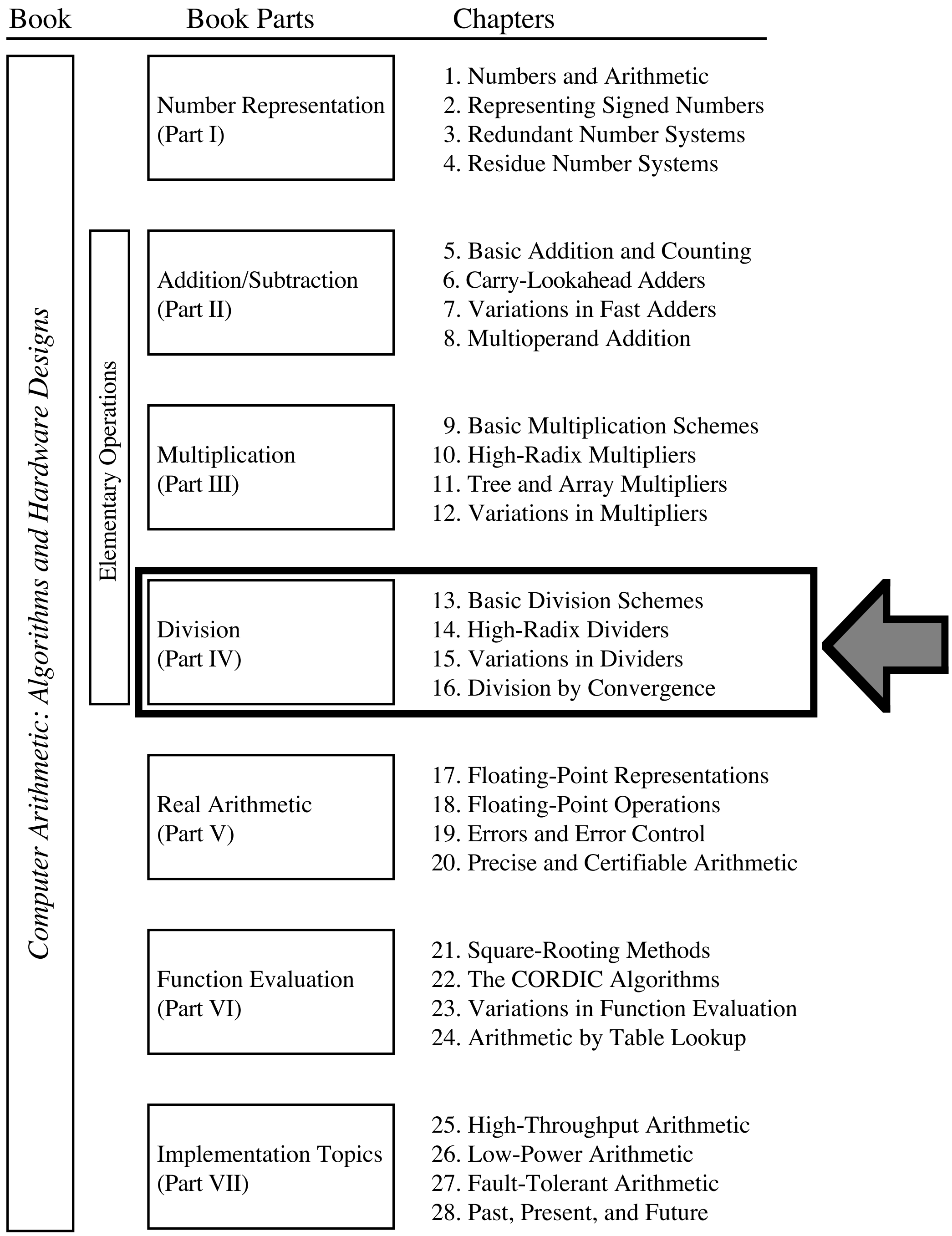


Fig. P.1 The structure of this book in parts and chapters.

Part IV Division

Part Goals

Review shift-subtract division schemes
Learn about faster dividers
Discuss speed/cost tradeoffs in dividers

Part Synopsis

Division: the most complex basic operation
Fortunately, it is also the least common
Division speedup: high-radix, array, ...
Combined multiplication/division hardware
Digit-recurrence vs convergence division

Part Contents

Chapter 13 Basic Division Schemes
Chapter 14 High-Radix Dividers
Chapter 15 Variations in Dividers
Chapter 16 Division by Convergence

13 Basic Division Schemes

Chapter Goals

Study shift/subtract or bit-at-a-time dividers and set the stage for faster methods and variations to be covered in Chapters 14-16

Chapter Highlights

Shift/sub division vs shift/add multiplication
Hardware, firmware, software algorithms
Dividing 2's-complement numbers
The special case of a constant divisor

Chapter Contents

13.1 Shift/Subtract Division Algorithms
13.2 Programmed Division
13.3 Restoring Hardware Dividers
13.4 Nonrestoring and Signed Division
13.5 Division by Constants
13.6 Preview of Fast Dividers

13.1 Shift/Subtract Division Algorithms

Notation for our discussion of division algorithms:

z	Dividend	$z_{2k-1}z_{2k-2} \cdots z_1z_0$
d	Divisor	$d_{k-1}d_{k-2} \cdots d_1d_0$
q	Quotient	$q_{k-1}q_{k-2} \cdots q_1q_0$
s	Remainder $(z - d \times q)$	$s_{k-1}s_{k-2} \cdots s_1s_0$

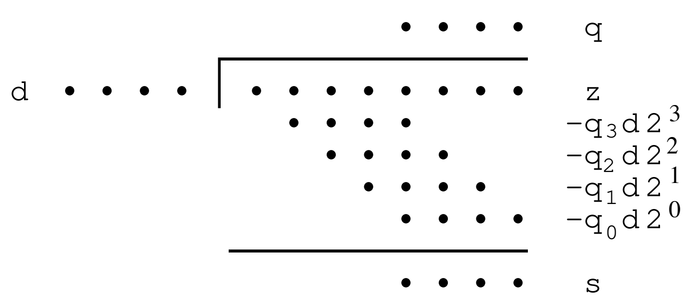


Fig. 13.1 Division of an 8-bit number by a 4-bit number in dot notation.

Division is more complex than multiplication:

Need for quotient digit selection or estimation

Possibility of overflow: the high-order k bits of z must be strictly less than d; this overflow check also detects the divide-by-zero condition.

Fractional division can be reformulated as integer division

Integer division is characterized by $z = d \times q + s$

multiply both sides by 2^{-2k} to get

$$2^{-2k}z = (2^{-k}d) \times (2^{-k}q) + 2^{-2k}s$$

$$z_{frac} = d_{frac} \times q_{frac} + 2^{-k}s_{frac}$$

Divide fractions like integers; adjust the final remainder

No overflow condition in this case is $z_{frac} < d_{frac}$

Sequential division with left shifts

$$s^{(j)} = \underbrace{\underbrace{2s^{(j-1)}}_{\text{shift}} - q_{k-j}(2^k d)}_{\text{subtract}} \qquad \text{with} \qquad s^{(0)} = z \text{ and } s^{(k)} = 2^k s$$

There is no division algorithm with right shifts

Integer division		Fractional division	
z	0 1 1 1 0 1 0 1	z_{frac}	. 0 1 1 1 0 1 0 1
$2^4 d$	1 0 1 0	d_{frac}	. 1 0 1 0
$s^{(0)}$	0 1 1 1 0 1 0 1	$s^{(0)}$	. 0 1 1 1 0 1 0 1
$2s^{(0)}$	0 1 1 1 0 1 0 1	$2s^{(0)}$	0 . 1 1 1 0 1 0 1
$-q_3 2^4 d$	1 0 1 0 $\{q_3 = 1\}$	$-q_{-1} d$	. 1 0 1 0 $\{q_{-1} = 1\}$
$s^{(1)}$	0 1 0 0 1 0 1	$s^{(1)}$	. 0 1 0 0 1 0 1
$2s^{(1)}$	0 1 0 0 1 0 1	$2s^{(1)}$	0 . 1 0 0 1 0 1
$-q_2 2^4 d$	0 0 0 0 $\{q_2 = 0\}$	$-q_{-2} d$	. 0 0 0 0 $\{q_{-2} = 0\}$
$s^{(2)}$	1 0 0 1 0 1	$s^{(2)}$	. 1 0 0 1 0 1
$2s^{(2)}$	1 0 0 1 0 1	$2s^{(2)}$	1 . 0 0 1 0 1
$-q_1 2^4 d$	1 0 1 0 $\{q_1 = 1\}$	$-q_{-3} d$	. 1 0 1 0 $\{q_{-3} = 1\}$
$s^{(3)}$	1 0 0 0 1	$s^{(3)}$	. 1 0 0 0 1
$2s^{(3)}$	1 0 0 0 1	$2s^{(3)}$	1 . 0 0 0 1
$-q_0 2^4 d$	1 0 1 0 $\{q_0 = 1\}$	$-q_{-1} d$	. 1 0 1 0 $\{q_{-4} = 1\}$
$s^{(4)}$	0 1 1 1	$s^{(4)}$	. 0 1 1 1
s	0 1 1 1	s_{frac}	0 . 0 0 0 0 0 1 1 1
q	1 0 1 1	q_{frac}	. 1 0 1 1

Fig. 13.2 Examples of sequential division with integer and fractional operands.

13.2 Programmed Division

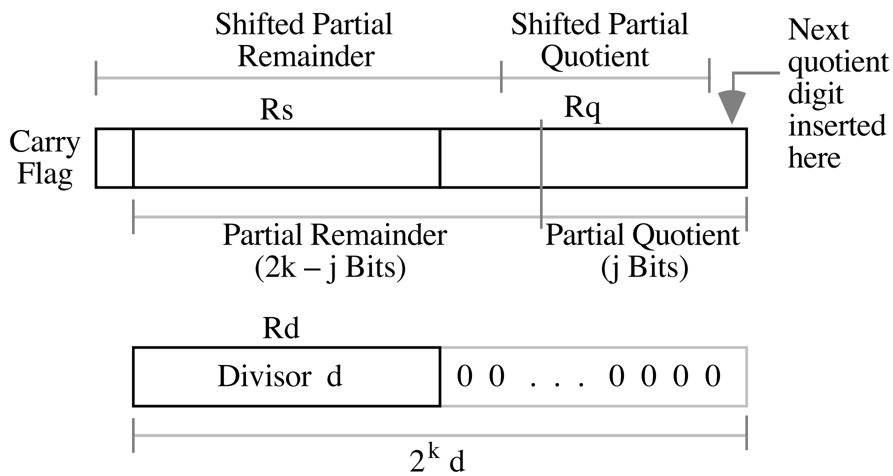


Fig. 13.3 Register usage for programmed division.

```
{Using left shifts, divide unsigned 2k-bit dividend,
z_high|z_low, storing the k-bit quotient and remainder.
Registers: R0 holds 0        Rc for counter
           Rd for divisor    Rs for z_high & remainder
           Rq for z_low & quotient}

{Load operands into registers Rd, Rs, and Rq}

      div: load    Rd with divisor
           load    Rs with z_high
           load    Rq with z_low

{Check for exceptions}

           branch  d_by_0 if Rd = R0
           branch  d_ovfl if Rs > Rd

{Initialize counter}

           load     k  into Rc

{Begin division loop}

   d_loop: shift   Rq left 1   {zero to LSB, MSB to carry}
           rotate  Rs left 1   {carry to LSB, MSB to carry}
           skip    if carry = 1
           branch  no_sub if Rs < Rd
           sub     Rd from Rs
           incr    Rq          {set quotient digit to 1}
   no_sub: decr    Rc          {decrement counter by 1}
           branch  d_loop if Rc ≠ 0

{Store the quotient and remainder}

           store   Rq into quotient
           store   Rs into remainder
   d_by_0: ...
   d_ovfl: ...
   d_done: ...
```

Fig. 13.4 Programmed division using left shifts.

13.3 Restoring Hardware Dividers

Division with signed operands: q and s are defined by

$$z = d \times q + s \qquad \text{sign}(s) = \text{sign}(z) \qquad |s| < |d|$$

Examples of division with signed operands

$z = 5$	$d = 3$	$\Rightarrow$	$q = 1$	$s = 2$
$z = 5$	$d = -3$	$\Rightarrow$	$q = -1$	$s = 2$
$z = -5$	$d = 3$	$\Rightarrow$	$q = -1$	$s = -2$
$z = -5$	$d = -3$	$\Rightarrow$	$q = 1$	$s = -2$

Magnitudes of q and s are unaffected by input signs

Signs of q and s are derivable from signs of z and d

Will discuss direct signed division later

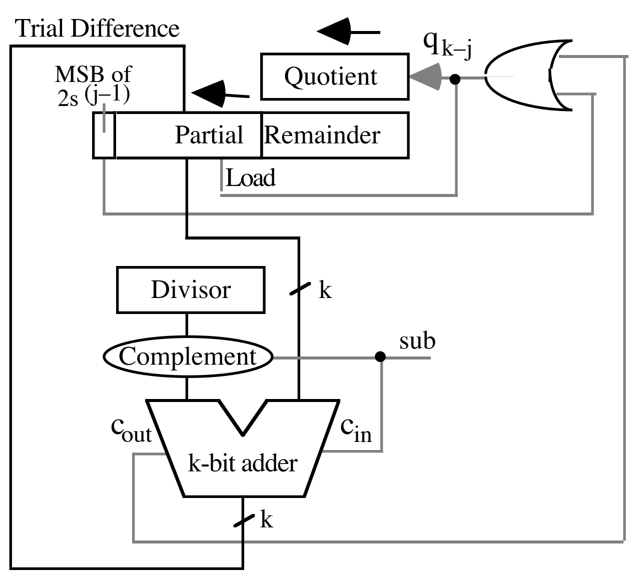


Fig. 13.5 Shift/subtract sequential restoring divider.

z		0 1 1 1	0 1 0 1	No overflow, since: $(0111)_{two} < (1010)_{two}$
$2^4 d$	0	1 0 1 0		
$-2^4 d$	1	0 1 1 0		
$s^{(0)}$	0	0 1 1 1	0 1 0 1	
$2s^{(0)}$	0	1 1 1 0	1 0 1	
$+(-2^4 d)$	1	0 1 1 0		
$s^{(1)}$	0	0 1 0 0	1 0 1	Positive, so set $q_3 = 1$
$2s^{(1)}$	0	1 0 0 1	0 1	
$+(-2^4 d)$	1	0 1 1 0		
$s^{(2)}$	1	1 1 1 1	0 1	Negative, so set $q_2 = 0$ and restore
$s^{(2)} = 2s^{(1)}$	0	1 0 0 1	0 1	
$2s^{(2)}$	1	0 0 1 0	1	
$+(-2^4 d)$	1	0 1 1 0		
$s^{(3)}$	0	1 0 0 0	1	Positive, so set $q_1 = 1$
$2s^{(3)}$	1	0 0 0 1		
$+(-2^4 d)$	1	0 1 1 0		
$s^{(4)}$	0	0 1 1 1		Positive, so set $q_0 = 1$
s			0 1 1 1	
q			1 0 1 1	

Fig. 13.6 Example of restoring unsigned division.

13.4 Nonrestoring and Signed Division

The cycle time in restoring division must accommodate
- shifting the registers
- allowing signals to propagate through the adder
- determining and storing the next quotient digit
- storing the trial difference, if required

Later events depend on earlier ones in the same cycle

Such timing dependencies tend to lengthen the clock cycle

Nonrestoring division algorithm to the rescue!
- assume $q_{k-j} = 1$ and perform a subtraction
- store the difference as the new partial remainder
  - (the partial remainder can become incorrect, hence the name "nonrestoring")

Why it is acceptable to store an incorrect value in the partial-remainder register?

Shifted partial remainder at start of the cycle is u

Subtraction yields the negative result $u - 2^k d$

Option 1: restore the partial remainder to its correct value u, shift, and subtract to get $2u - 2^k d$

Option 2: keep the incorrect partial remainder $u - 2^k d$, shift, and add to get $2(u - 2^k d) + 2^k d = 2u - 2^k d$

z		0 1 1 1	0 1 0 1	No overflow, since:
2^4d	0	1 0 1 0		$(0111)_{two} < (1010)_{two}$
-2^4d	1	0 1 1 0		
$s^{(0)}$	0	0 1 1 1	0 1 0 1	
$2s^{(0)}$	0	1 1 1 0	1 0 1	Positive,
$+(-2^4d)$	1	0 1 1 0		so subtract
$s^{(1)}$	0	0 1 0 0	1 0 1	
$2s^{(1)}$	0	1 0 0 1	0 1	Positive, so set $q_3 = 1$
$+(-2^4d)$	1	0 1 1 0		and subtract
$s^{(2)}$	1	1 1 1 1	0 1	
$2s^{(2)}$	1	1 1 1 0	1	Negative, so set $q_2 = 0$
$+2^4d$	0	1 0 1 0		and add
$s^{(3)}$	0	1 0 0 0	1	
$2s^{(3)}$	1	0 0 0 1		Positive, so set $q_1 = 1$
$+(-2^4d)$	1	0 1 1 0		and subtract
$s^{(4)}$	0	0 1 1 1		Positive, so set $q_0 = 1$
s			0 1 1 1	
q			1 0 1 1	

Fig. 13.7 Example of nonrestoring unsigned division.

Example

$(0\ 1\ 1\ 1\quad 0\ 1\ 0\ 1)_{two}$ / $(1\ 0\ 1\ 0)_{two}$

$(117)_{ten}$ / $(10)_{ten}$

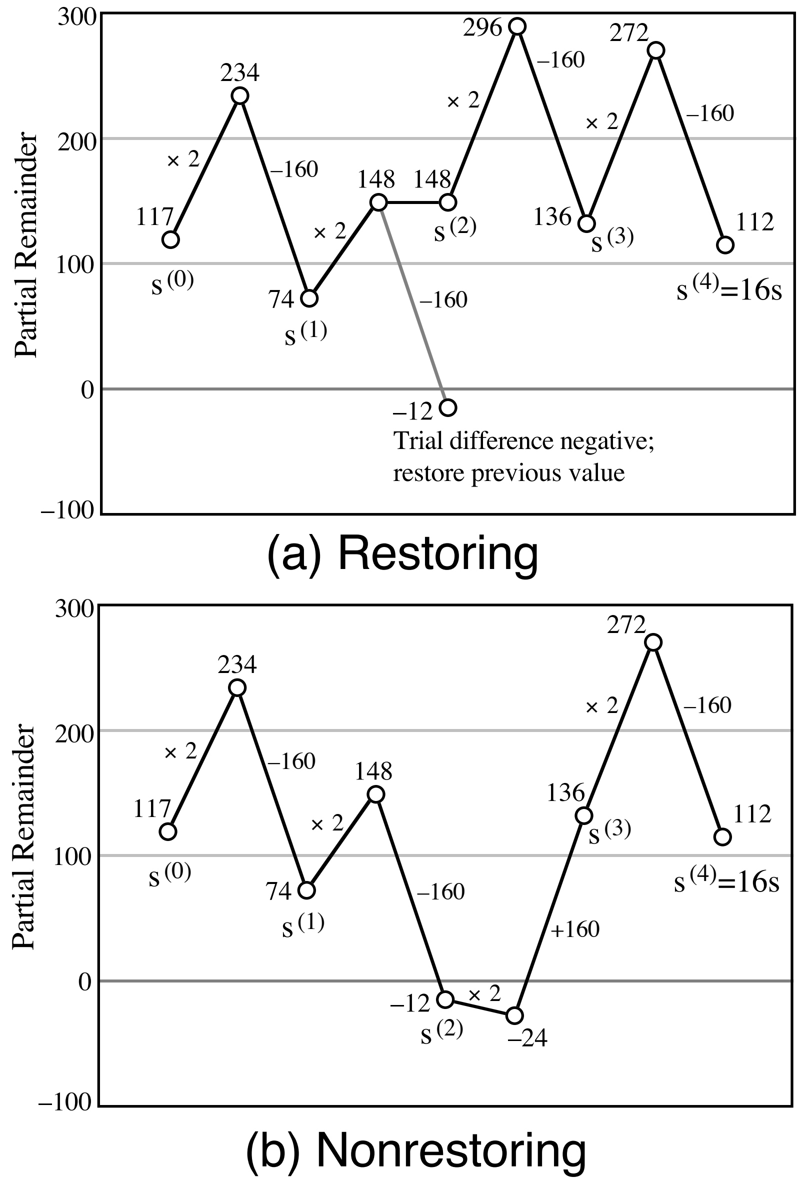


Fig. 13.8 Partial remainder variations for restoring and nonrestoring division.

Restoring division

$q_{k-j} = 0$ means no subtraction (or subtraction of 0)

$q_{k-j} = 1$ means subtraction of d

Nonrestoring division

We always subtract or add

As if the quotient digits are selected from the set {1, $^-1$}

1 corresponds to subtraction

$^-1$ corresponds to addition

Our goal is to end up with a remainder
that matches the sign of the dividend

This idea of trying to match the sign of s with the sign z,
leads to a direct signed division algorithm

If sign(s) = sign(d) then $q_{k-j} = 1$ else $q_{k-j} = {}^-1$

Two problems must be dealt with at the end:

1. Converting the quotient with digits 1 and $^-1$ to binary
2. Adjusting the results if final remainder has wrong sign (correction step involves addition of ±d to remainder and subtraction of ±1 from quotient)

Correction might be required even in unsigned division
(when the final remainder is negative)

z	0 0 1 0 0 0 0 1	Dividend = $(33)_{ten}$
2^4d	1 1 0 0 1	Divisor = $(-7)_{ten}$
-2^4d	0 0 1 1 1	
$s^{(0)}$	0 0 0 1 0 0 0 0 1	
$2s^{(0)}$	0 0 1 0 0 0 0 1	$\text{sign}(s^{(0)}) \neq \text{sign}(d)$,
$+2^4d$	1 1 0 0 1	so set $q_3 = {}^-1$ and add
$s^{(1)}$	1 1 1 0 1 0 0 1	
$2s^{(1)}$	1 1 0 1 0 0 1	$\text{sign}(s^{(1)}) = \text{sign}(d)$,
$+(-2^4d)$	0 0 1 1 1	so set $q_2 = 1$ and sub
$s^{(2)}$	0 0 0 0 1 0 1	
$2s^{(2)}$	0 0 0 1 0 1	$\text{sign}(s^{(2)}) \neq \text{sign}(d)$,
$+2^4d$	1 1 0 0 1	so set $q_1 = {}^-1$ and add
$+(-2^4d)$	1 0 1 1 0	
$s^{(3)}$	1 1 0 1 1 1	
$2s^{(3)}$	1 0 1 1 1	$\text{sign}(s^{(3)}) = \text{sign}(d)$,
$+(-2^4d)$	0 0 1 1 1	so set $q_0 = 1$ and sub
$s^{(4)}$	1 1 1 1 0	$\text{sign}(s^{(4)}) \neq \text{sign}(z)$
$+(-2^4d)$	0 0 1 1 1	Corrective subtraction
$s^{(4)}$	0 0 1 0 1	
s	0 1 0 1	Remainder = $(5)_{ten}$
q	$^-1$ 1 $^-1$ 1	Ucorrected BSD form
$q_{2\text{'s-compl}}$	1 1 0 0	Corrected q = $(-4)_{ten}$

Fig. 13.9 Example of nonrestoring signed division.

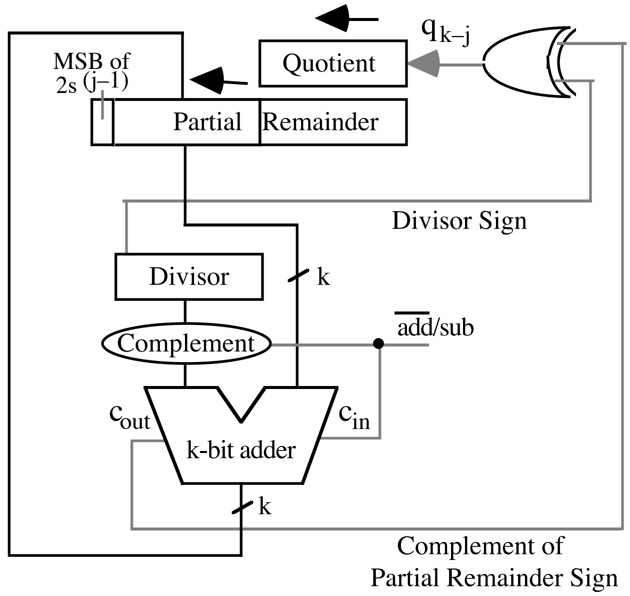


Fig. 13.10 Shift-subtract sequential nonrestoring divider.

13.5 Division by Constants

Method 1: find the reciprocal of the constant and multiply

Method 2: use the property that for each odd integer d there exists an odd integer m such that $d \times m = 2^n - 1$

$$\frac{1}{d} = \frac{m}{2^n - 1} = \frac{m}{2^n(1 - 2^{-n})}$$

$$= \frac{m}{2^n}(1 + 2^{-n})(1 + 2^{-2n})(1 + 2^{-4n}) \cdots$$

Number of shift-adds required is proportional to log k

Example: division by d = 5 with 24 bits of precision

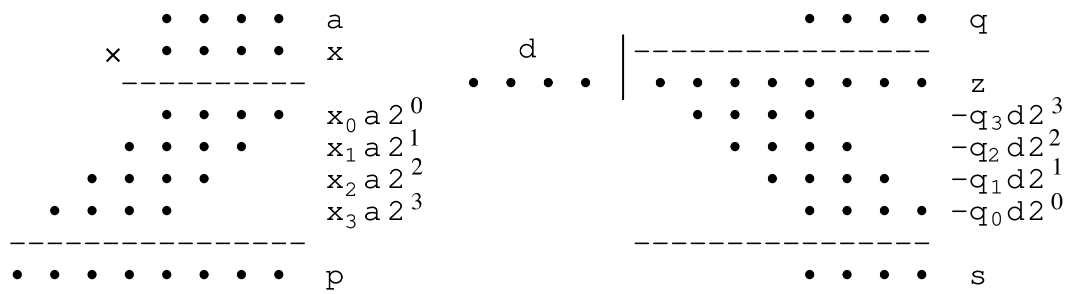
m = 3, n = 4 by inspection

$$\frac{z}{5} = \frac{3z}{2^4 - 1} = \frac{3z}{16(1 - 2^{-4})} = \frac{3z}{16}(1 + 2^{-4})(1 + 2^{-8})(1 + 2^{-16})$$

$q \leftarrow z + z$ shift-left 1	{3z computed}
$q \leftarrow q + q$ shift-right 4	$\{3z(1+2^{-4})\}$
$q \leftarrow q + q$ shift-right 8	$\{3z(1+2^{-4})(1+2^{-8})\}$
$q \leftarrow q + q$ shift-right 16	$\{3z(1+2^{-4})(1+2^{-8})(1+2^{-16})\}$
$q \leftarrow q$ shift-right 4	$\{3z(1+2^{-4})(1+2^{-8})(1+2^{-16})/16\}$

This divide-by-5 algorithm uses 5 shifts and 4 adds

13.6 Preview of Fast Dividers



(a) $k \times k$ integer multiplication **(b)** $2k / k$ integer division

Fig. 13.11 (a) Multiplication and (b) division as multioperand addition problems.

Like multiplication, division is multioperand addition
Thus, there are but two ways to speed it up:

a. Reducing the number of operands
(high-radix dividers)

b. Adding them faster
(use carry-save partial remainder)

There is one complication that makes division more difficult:
the terms to be subtracted from (added to) the dividend are not known a priori but become known as the quotient digits are computed; quotient digits in turn depend on partial remainders

14 High-Radix Dividers

Chapter Goals

Study techniques that allow us to obtain more than one quotient bit in each cycle (two bits in radix 4, three bits in radix 8, . . .)

Chapter Highlights

High radix ⇒ quotient digit selection harder
Redundant quotient representation as cure
Carry-save addition reduces the cycle time
Implementation methods and tradeoffs

Chapter Contents

14.1 Basics of High-Radix Division
14.2 Radix-2 SRT Division
14.3 Using Carry-Save Adders
14.4 Choosing the Quotient Digits
14.5 Radix-4 SRT Division
14.6 General High-Radix Dividers

14.1 Basics of High-Radix Division

Radix-r version of division recurrence of Section 13.1

$$s^{(j)} = r\,s^{(j-1)} - q_{k-j}\,(r^k d) \qquad \text{with} \qquad s^{(0)} = z \quad \text{and } s^{(k)} = r^k s$$

High-radix dividers of practical interest have $r = 2^b$

(and, occasionally, $r = 10$)

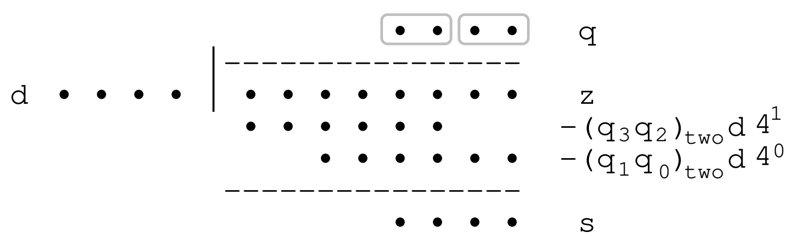


Fig. 14.1 Radix-4 division in dot notation.

Radix-4 integer division

z	0 1 2 3 1 1 2 3	
4^4d	1 0 0 3	
$s^{(0)}$	0 1 2 3 1 1 2 3	
$4s^{(0)}$	0 1 2 3 1 1 2 3	
$-q_3\,4^4d$	0 1 2 0 3	$\{q_3 = 1\}$
$s^{(1)}$	0 0 2 2 1 2 3	
$4s^{(1)}$	0 0 2 2 1 2 3	
$-q_2\,4^4d$	0 0 0 0 0	$\{q_2 = 0\}$
$s^{(2)}$	0 2 2 1 2 3	
$4s^{(2)}$	0 2 2 1 2 3	
$-q_1\,4^4d$	0 1 2 0 3	$\{q_1 = 1\}$
$s^{(3)}$	1 0 0 3 3	
$4s^{(3)}$	1 0 0 3 3	
$-q_0\,4^4d$	0 3 0 1 2	$\{q_0 = 2\}$
$s^{(4)}$	1 0 2 1	
s	1 0 2 1	
q	1 0 1 2	

Radix-10 fractional division

z_{frac}	. 7 0 0 3	
d_{frac}	. 9 9	
$s^{(0)}$	. 7 0 0 3	
$10s^{(0)}$	7 . 0 0 3	
$-q_{-1}d$	6 . 9 3	$\{q_{-1} = 7\}$
$s^{(1)}$	. 0 7 3	
$10s^{(1)}$	0 . 7 3	
$-q_{-2}d$	0 . 0 0	$\{q_{-2} = 0\}$
$s^{(2)}$	. 7 3	
s_{frac}	. 0 0 7 3	
q_{frac}	. 7 0	

Fig. 14.2 Examples of high-radix division with integer and fractional operands.

14.2 Radix-2 SRT Division

Radix-2 nonrestoring division, fractional operands

$$s^{(j)} = 2s^{(j-1)} - q_{-j}\,d \quad \text{with} \quad s^{(0)} = z \;\text{ and }\; s^{(k)} = 2^k s$$

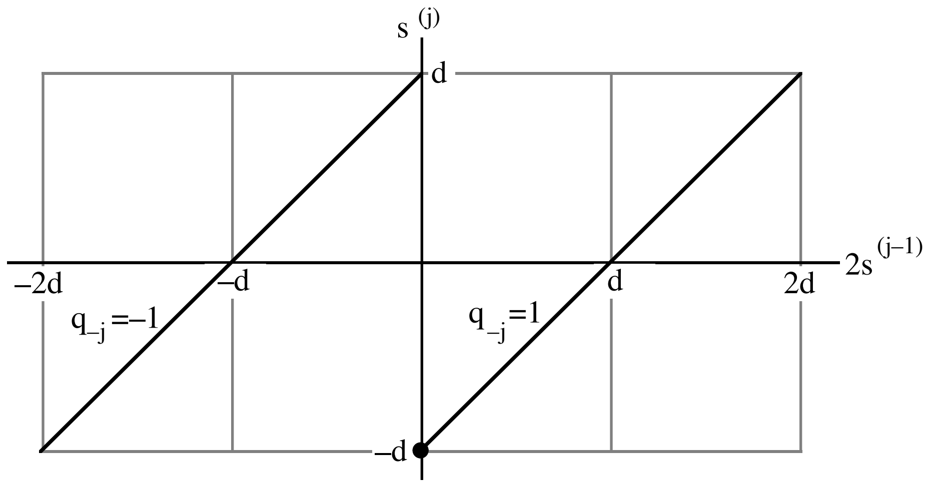


Fig. 14.3 The new partial remainder, $s^{(j)}$, as a function of the shifted old partial remainder, $2s^{(j-1)}$, in radix-2 nonrestoring division.

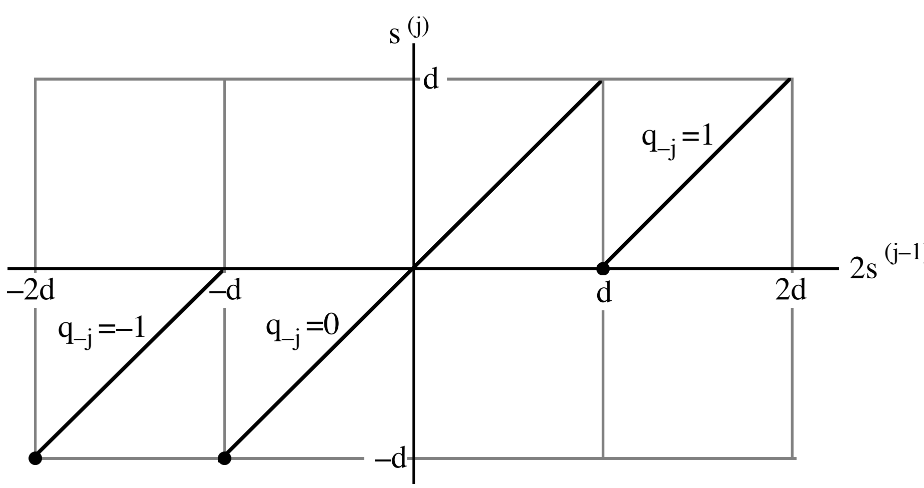


Fig. 14.4 The new partial remainder, $s^{(j)}$, as a function of $2s^{(j-1)}$, with q_{-j} in $\{-1, 0, 1\}$.

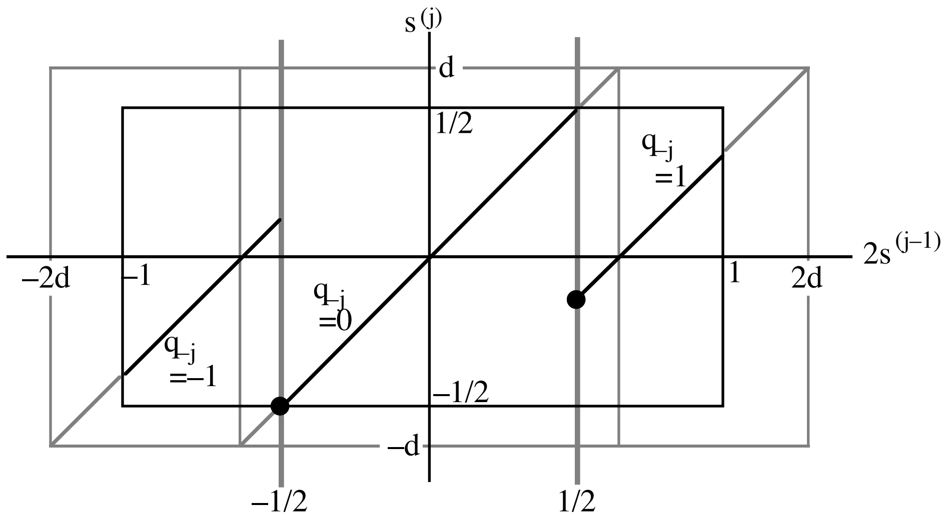


Fig. 14.5 The relationship between new and old partial remainders in radix-2 SRT division.

SRT algorithm (Sweeney, Robertson, Tocher)

$2s^{(j-1)} \geq +1/2 = (0.1)_{2\text{'s-compl}}$

$\Rightarrow\ 2s^{(j-1)} = (0.1u_{-2}u_{-3} \cdots)_{2\text{'s-compl}}$

$2s^{(j-1)} < -1/2 = (1.1)_{2\text{'s-compl}}$

$\Rightarrow\ 2s^{(j-1)} = (1.0u_{-2}u_{-3} \cdots)_{2\text{'s-compl}}$

Skipping over identical leading bits by shifting

$s^{(j-1)} = 0.0000110 \cdots$ Shift left by 4 bits and subtract; append q with 0 0 0 1

$s^{(j-1)} = 1.1110100 \cdots$ Shift left by 3 bits and add; append q with 0 0 $\bar{1}$

Average skipping distance in variable-shift SRT division:

2.67 bits (obtained by statistical analysis)

z	. 0 1 0 0 0 1 0 1	In [–1/2, 1/2), so OK
d	. 1 0 1 0	In [1/2, 1), so OK
$-d$	1 . 0 1 1 0	
$s^{(0)}$	0 . 0 1 0 0 0 1 0 1	
$2s^{(0)}$	0 . 1 0 0 0 1 0 1	$\geq 1/2$, so set $q_{-1} = 1$
$+(-d)$	1 . 0 1 1 0	and subtract
$s^{(1)}$	1 . 1 1 1 0 1 0 1	
$2s^{(1)}$	1 . 1 1 0 1 0 1	In [–1/2, 1/2), so $q_{-2} = 0$
$s^{(2)} = 2s^{(1)}$	1 . 1 1 0 1 0 1	
$2s^{(2)}$	1 . 1 0 1 0 1	$< -1/2$, so set $q_1 = {}^{-}1$
$+d$	0 . 1 0 1 0	and add
$s^{(3)}$	0 . 0 1 0 0 1	
$2s^{(3)}$	0 . 1 0 0 1	$\geq 1/2$, so set $q_0 = 1$
$+(-d)$	1 . 0 1 1 0	and subtract
$s^{(4)} = 2s^{(3)}$	1 . 1 1 1 1	Negative,
$+d$	0 . 1 0 1 0	so add to correct
$s^{(4)}$	0 . 1 0 0 1	
s	0 . 0 0 0 0 1 0 0 1	
q	0 . 1 0 ${}^{-}1$ 1	Ucorrected BSD form
q	0 . 0 1 1 0	Convert & subtract ulp

Fig. 14.6 Example of unsigned radix-2 SRT division.

14.3 Using Carry-Save Adders

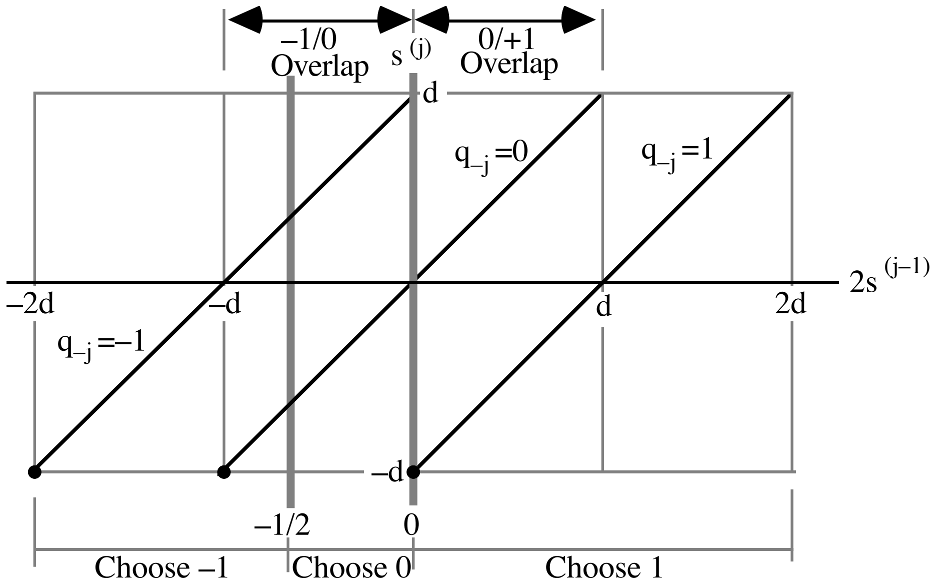


Fig. 14.7 Constant thresholds used for quotient digit selection in radix-2 division with q_{k-j} in {–1, 0, 1}.

Sum part of $2s^{(j-1)}$: $u = (u_1 u_0 . u_{-1} u_{-2} \cdots)_{2\text{'s-compl}}$

Carry part of $2s^{(j-1)}$: $v = (v_1 v_0 . v_{-1} v_{-2} \cdots)_{2\text{'s-compl}}$

$t = u_{[-2,1]} + v_{[-2,1]}$ {Add the 4 MSBs of u and v}

if $t < -1/2$

then $q_{-j} = -1$

else if $t \ge 0$

  then $q_{-j} = 1$

  else $q_{-j} = 0$

  endif

endif

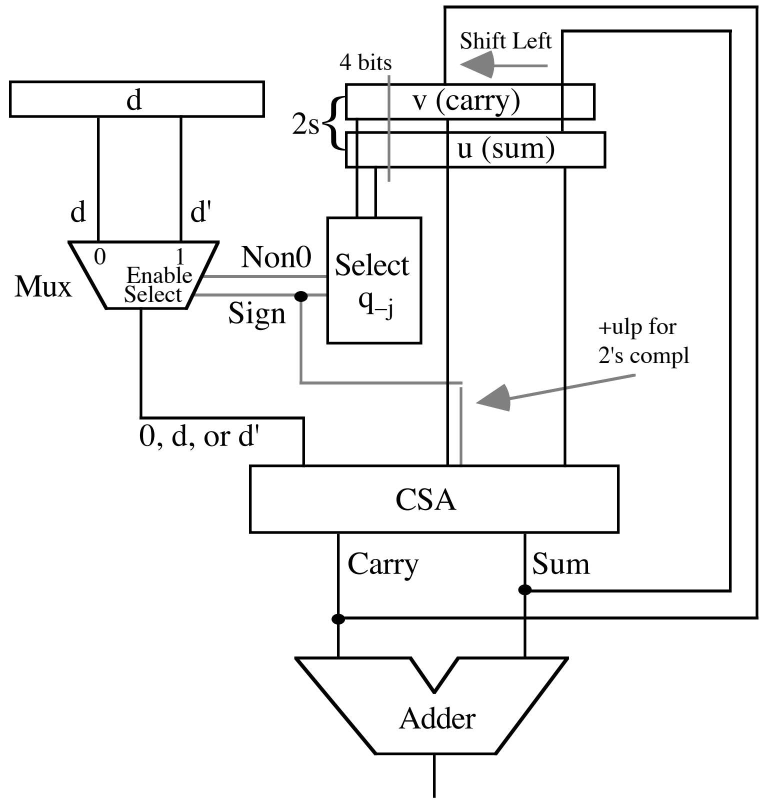


Fig. 14.8 Block diagram of a radix-2 divider with partial remainder in stored-carry form.

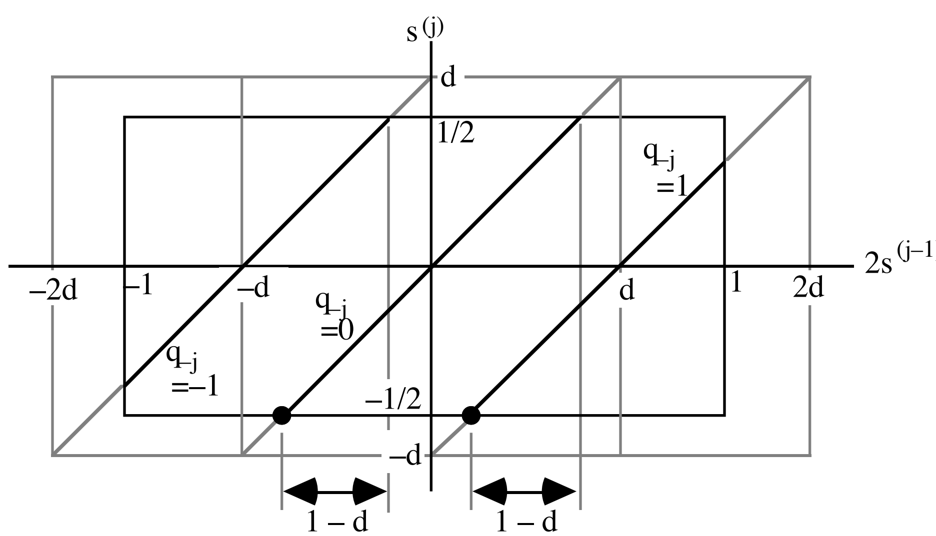


Fig. 14.9 Overlap regions in radix-2 SRT division.

14.4 Choosing the Quotient Digits

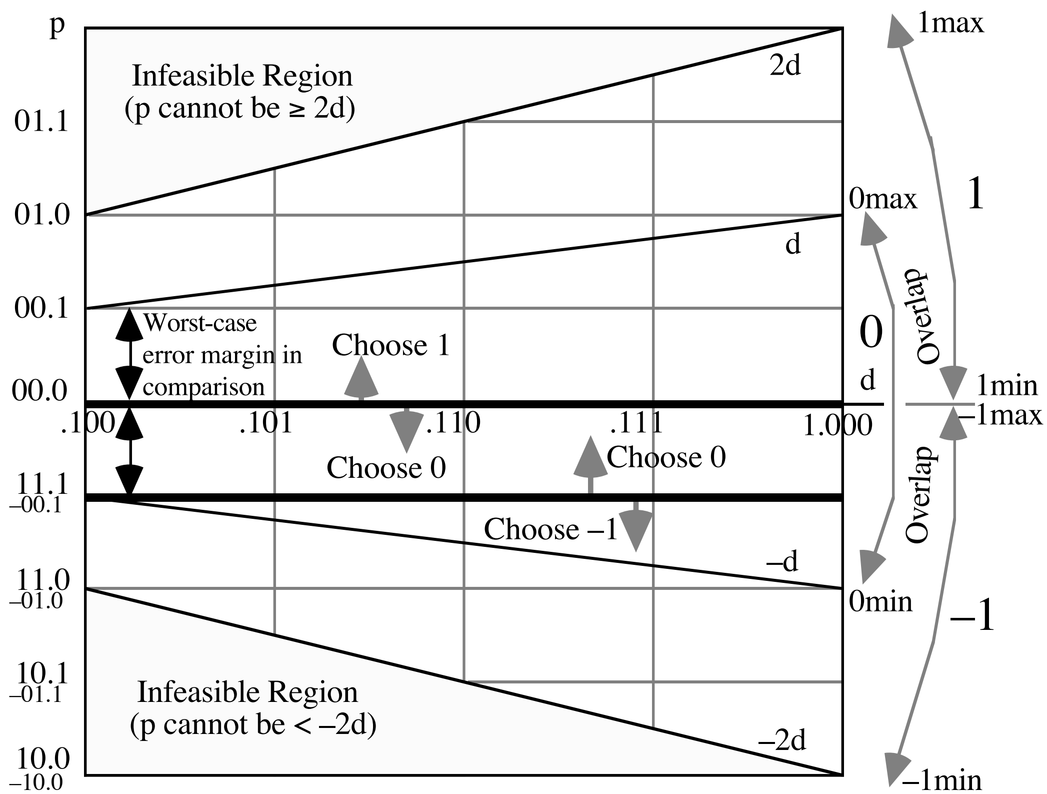


Fig. 14.10 A p-d plot for radix-2 division with $d \in [1/2,1)$, partial remainder in $[-d, d)$, and quotient digits in $[-1, 1]$.

Approx. shifted partial remainder $t = (t_1 t_0 . t_{-1} t_{-2})_{2\text{'s-compl}}$

$$\text{Non0} = \bar{t}_1 + \bar{t}_0 + \bar{t}_{-1} = \overline{t_1\, t_0\, t_{-1}}$$

$$\text{Sign} = t_1 (\bar{t}_0 + \bar{t}_{-1})$$

14.5 Radix-4 SRT division

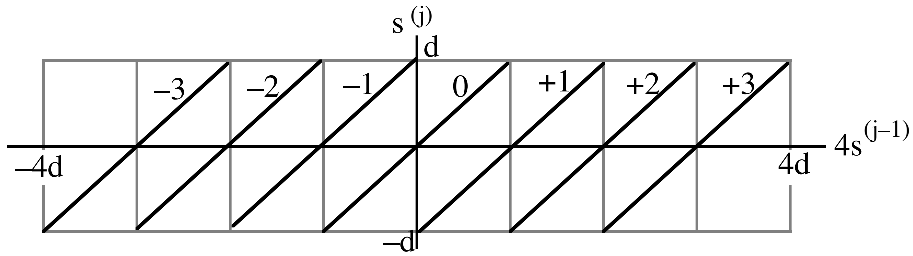


Fig. 14.11 New versus shifted old partial remainder in radix-4 division with q_{-j} in [–3, 3].

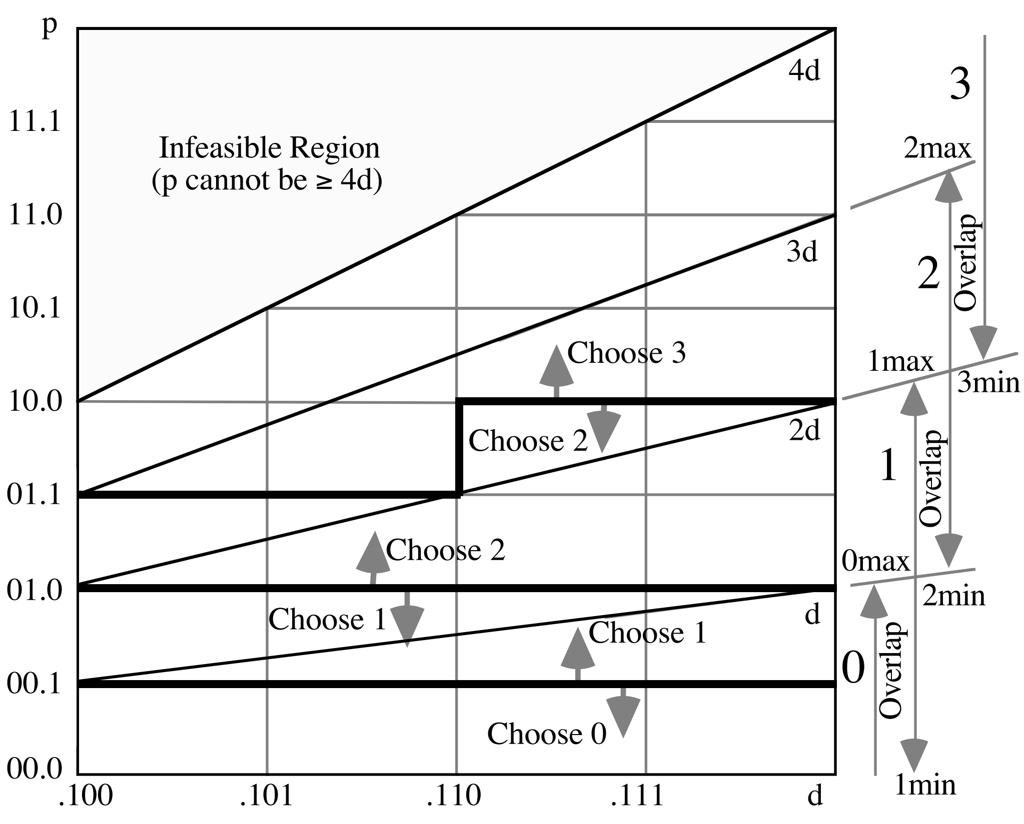


Fig. 14.12 p-d plot for radix-4 SRT division with quotient digit set [–3, 3].

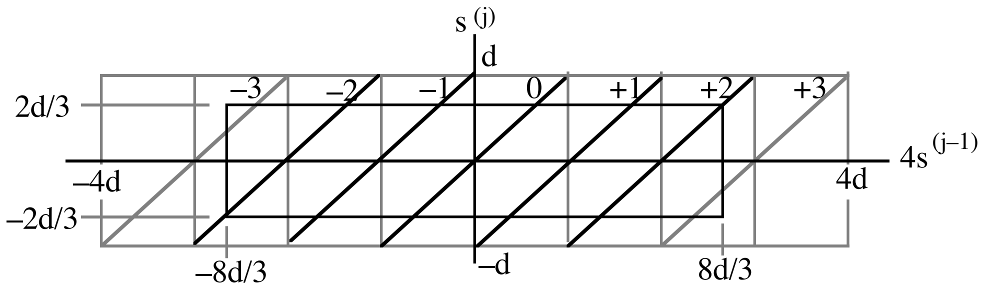


Fig. 14.13 New versus shifted old partial remainder in radix-4 division with q_{-j} in [–2, 2].

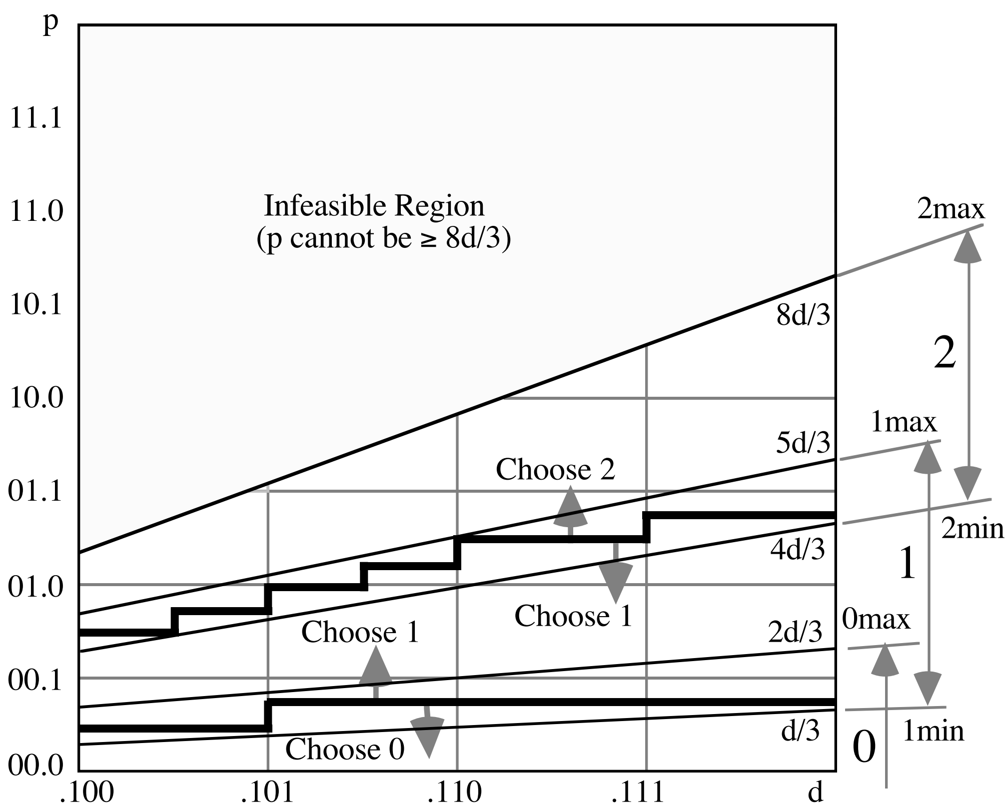


Fig. 14.14 p-d plot for radix-4 SRT division with quotient digit set [–2, 2].

14.6 General High-Radix Dividers

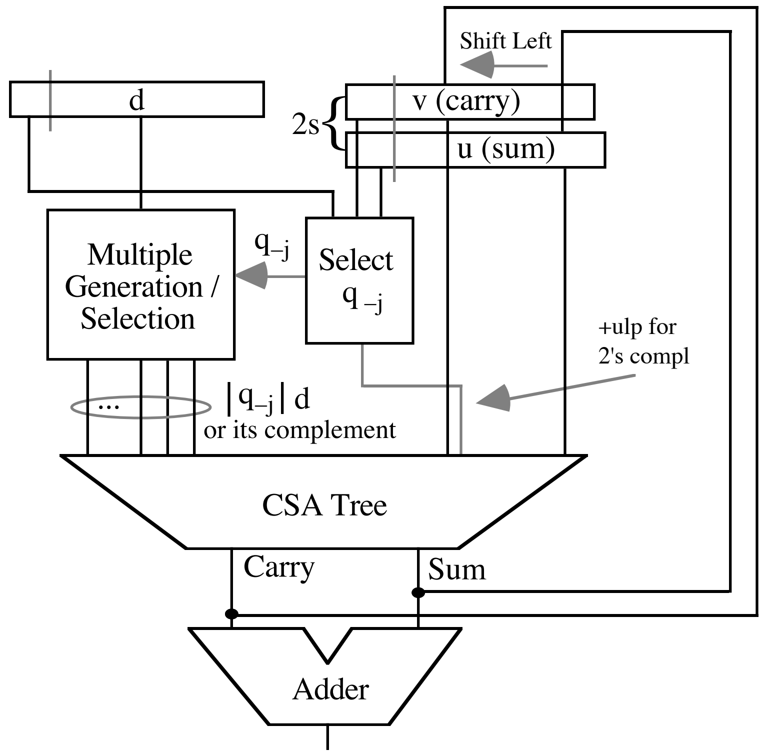


Fig. 14.15 Block diagram of radix-r divider with partial remainder in stored-carry form.

15 Variations in Dividers

Chapter Goals

Discuss practical aspects of designing
high-radix dividers and cover
other types of fast hardware dividers

Chapter Highlights

Building and using p-d plots in practice
Prescaling simplifies quotient digit selection
Parallel hardware (array) dividers
Shared hardware in multipliers and dividers
Square-rooting not special case of division

Chapter Contents

15.1 Quotient-Digit Selection Revisited
15.2 Using p-d Plots in Practice
15.3 Division with Prescaling
15.4 Modular Dividers and Reducers
15.5 Array Dividers
15.6 Combined Multiply/Divide Units

15.1 Quotient-Digit Selection Revisited

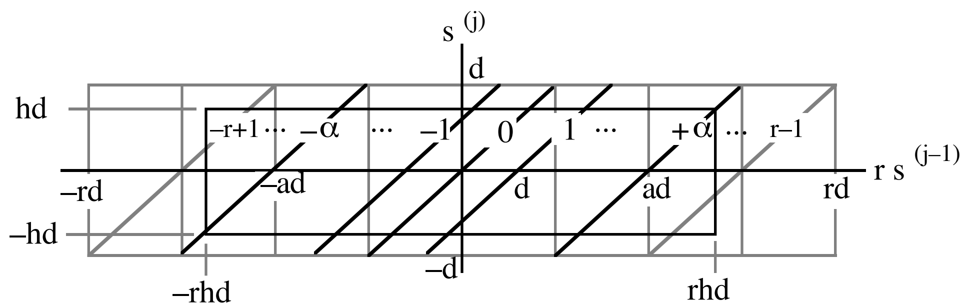


Fig. 15.1 The relationship between new and shifted old partial remainders in radix-r division with quotient digits in [–α, +α].

Radix-r division with quotient digit set [–α, α], $\alpha < r - 1$

Restrict the partial remainder range, say to [–hd, hd)

From the solid rectangle in Fig. 15.1, we get

$$rhd - \alpha d \le hd \qquad \text{or} \qquad h \le \alpha/(r-1)$$

To minimize the range restriction, we choose h = $\alpha/(r - 1)$

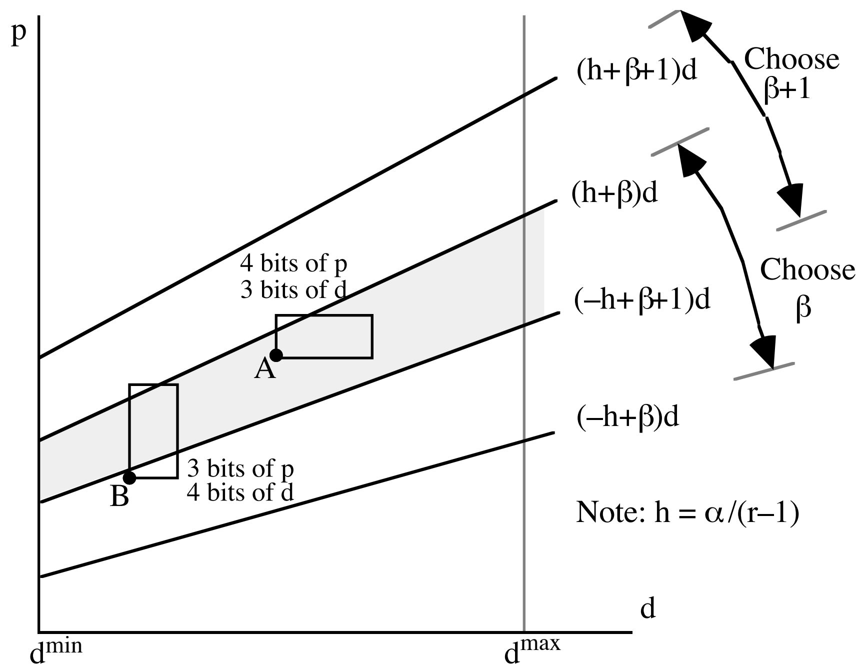


Fig. 15.2 A part of p-d plot showing the overlap region for choosing the quotient digit value β or β+1 in radix-r division with quotient digit set [–α, α].

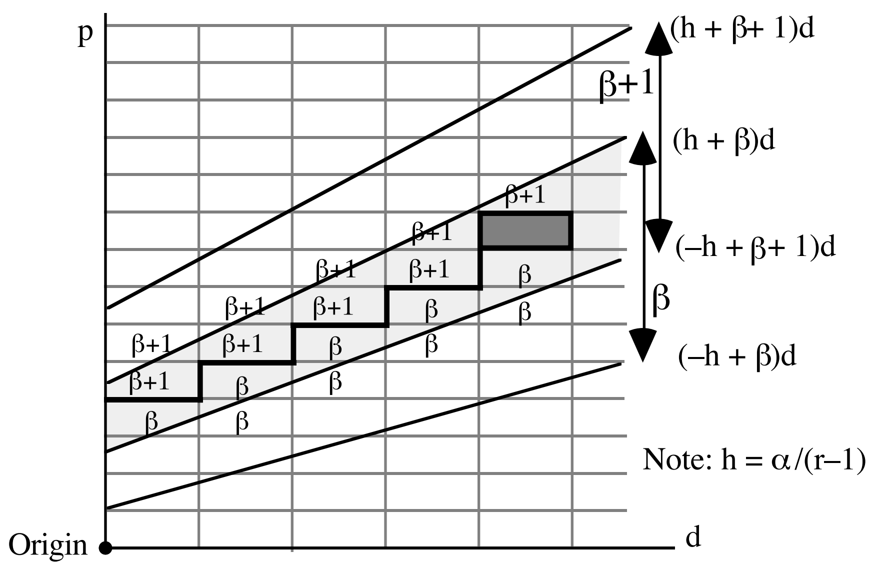


Fig. 15.3 A part of p-d plot showing an overlap region and its staircase-like selection boundary.

15.2 Using p-d Plots in Practice

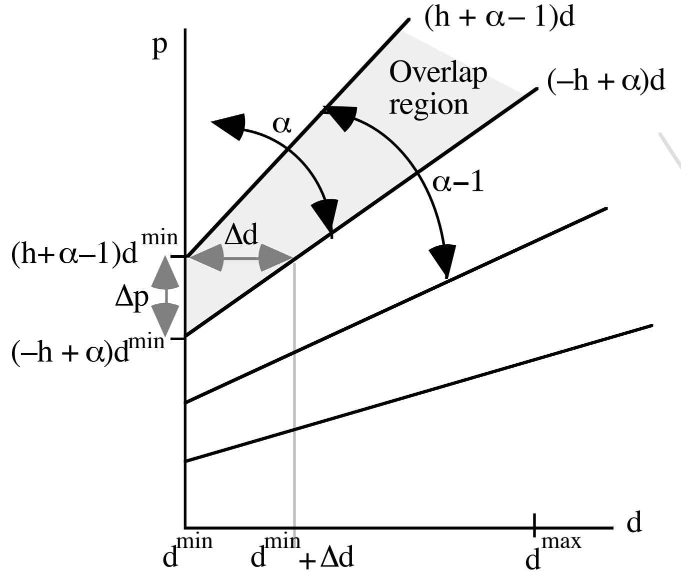


Fig. 15.4 Establishing upper bounds on the dimensions of uncertainty rectangles.

Smallest Δd occurs for the overlap region of α and $\alpha - 1$

$$\Delta d = d^{min} \frac{2h - 1}{-h + \alpha}$$

$$\Delta p = d^{min} (2h - 1)$$

Example: For $r = 4$, divisor range $[0.5, 1)$, digit set $[-2, 2]$, we have $\alpha = 2$, $d^{min} = 1/2$, $h = \alpha/(r-1) = 2/3$

$$\Delta d = (1/2)\,\frac{4/3 - 1}{-2/3 + 2} = 1/8$$

$$\Delta p = (1/2)\,(4/3 - 1) = 1/6$$

Since $1/8 = 2^{-3}$ and $2^{-3} \le 1/6 < 2^{-2}$, we must inspect at least 3 bits of d (2, given its leading 1) and 3 bits of p

These are lower bounds and may prove inadequate

In fact, 3 bits of p and 4 (3) bits of d are required

With p in carry-save form, 4 bits of each component must be inspected (or added to give the high-order 3 bits)

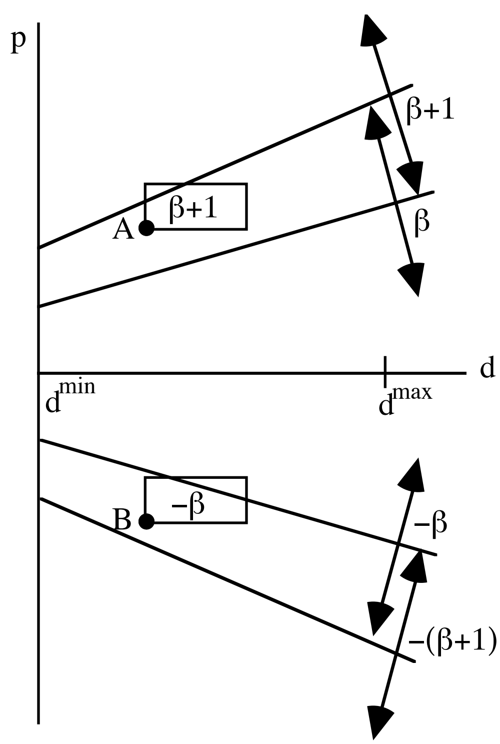
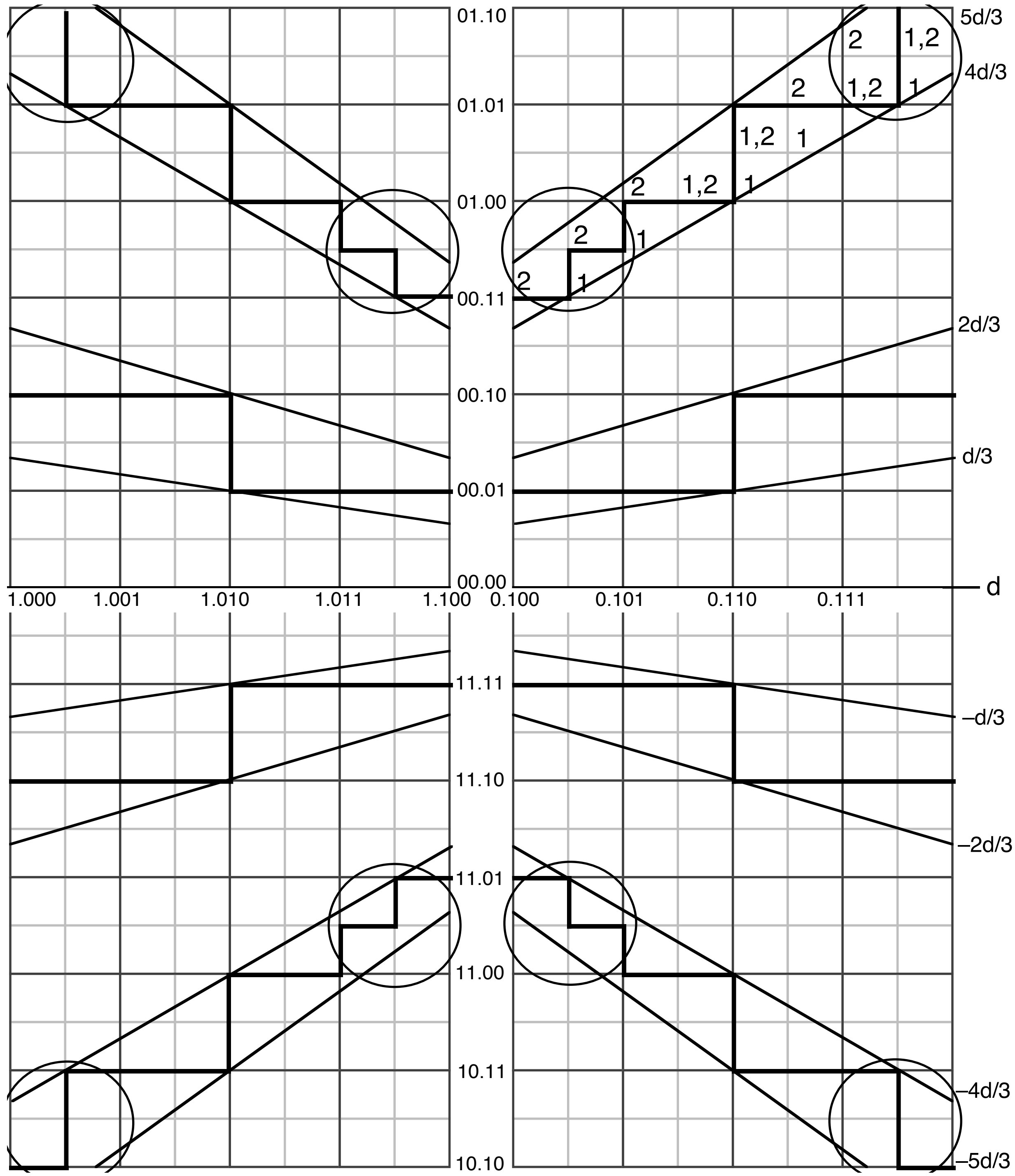


Fig. 15.5 The asymmetry of quotient digit selection process.


01.10
01.01
01.00
00.11
00.10
00.01
00.00
5d/3
4d/3
2d/3
d/3
d
1.000
1.001
1.010
1.011
1.100
0.100
0.101
0.110
0.111
2
1,2
1
11.11
11.10
11.01
11.00
10.11
10.10
−d/3
−2d/3
−4d/3
−5d/3

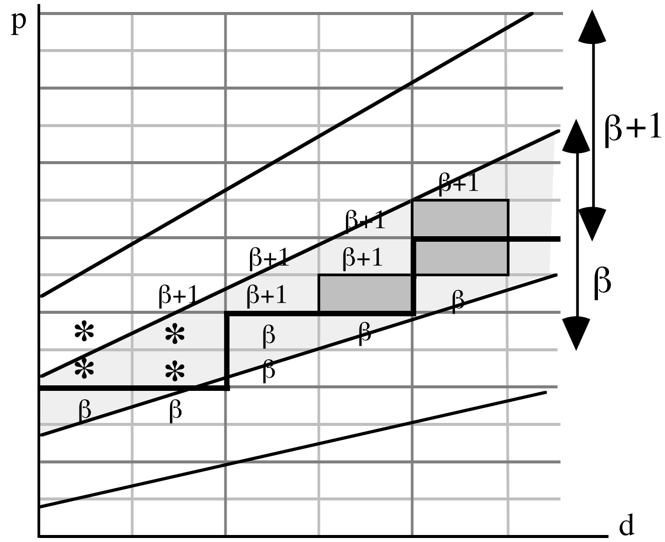
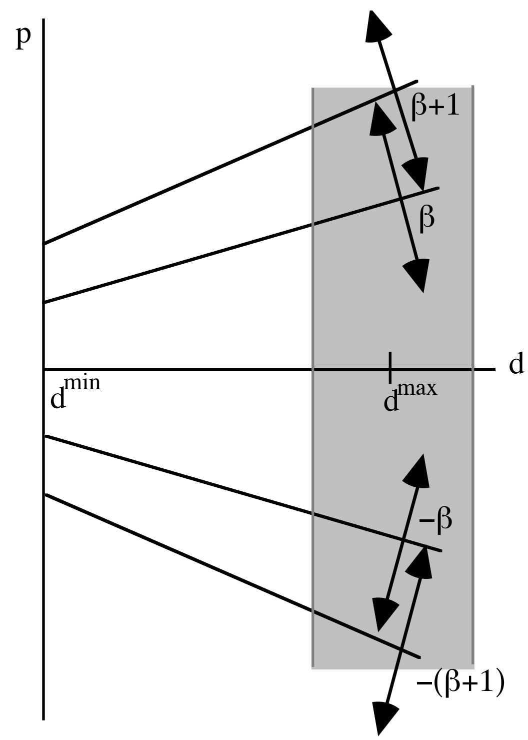


Fig. 15.6 Example of p-d plot allowing larger uncertainty rectangles, if the 4 cases marked with asterisks are handled as exceptions.

15.3 Division with Prescaling

Overlap regions of a p-d plot are wider toward the high end of the divisor range



If we can restrict the magnitude of the divisor to an interval close to d^{max} (say $1 - \varepsilon < d < 1 + \delta$, when $d^{max} = 1$), quotient digit selection may become simpler

This restriction can be achieved by performing the division (zm)/(dm) for a suitably chosen scale factor m ($m > 1$)

Of course, prescaling (multiplying z and d by the scale factor m) should be done without real multiplications

15.4 Modular Dividers and Reducers

Given a dividend z and divisor d, with $d \geq 0$, a modular divider computes

$$q = \lfloor z / d \rfloor \qquad \text{and} \qquad s = z \bmod d = \langle z \rangle_d$$

The quotient q is, by definition, an integer

but the inputs z and d do not have to be integers

Example:

$$\lfloor -3.76 / 1.23 \rfloor = -4 \qquad \text{and} \qquad \langle -3.76 \rangle_{1.23} = 1.16$$

The modular remainder is always positive

A modular reducer computes only the modular remainder

15.5 Array Dividers

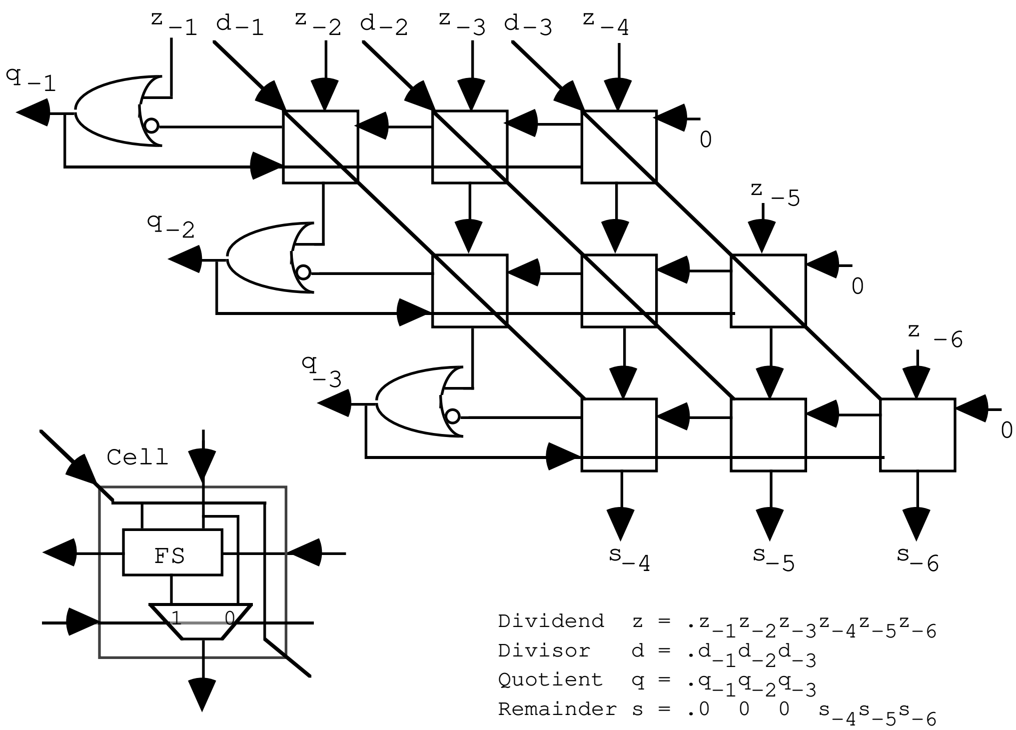


Fig. 15.7 Restoring array divider composed of controlled subtractor cells.

The similarity of the array divider of Fig. 15.7 to an array multiplier is somewhat deceiving

The same number of cells are involved in both designs, and the cells have comparable complexities

However, the critical path in a $k \times k$ array multiplier contains $O(k)$ cells, whereas in Fig. 15.7, the critical path passes through all k^2 cells (the borrow signal ripples in each row)

Thus, an array divider is quite slow, and, given its high cost, not very cost-effective

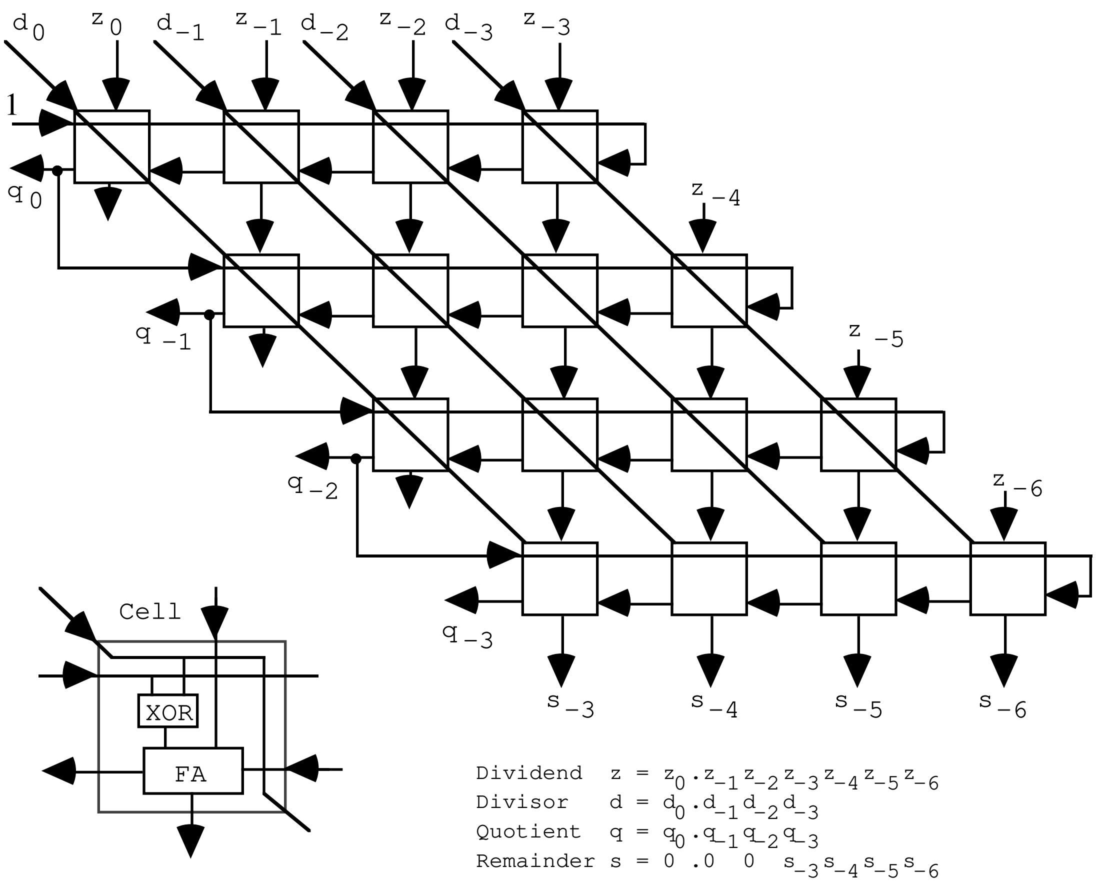


Fig. 15.8 Nonrestoring array divider built of controlled add/subtract cells.

Speedup method:

Pass the partial remainder between rows in carry-save form

However, we still need to know the carry-out or borrow-out from each row to determine the action in the following row: subtract/no-change (Fig. 15.7) or subtract/add (Fig. 15.8)

This can be done by using a carry- (borrow-) lookahead circuit laid out between successive rows

Not used in practice

15.6 Combined Multiply/Divide Units

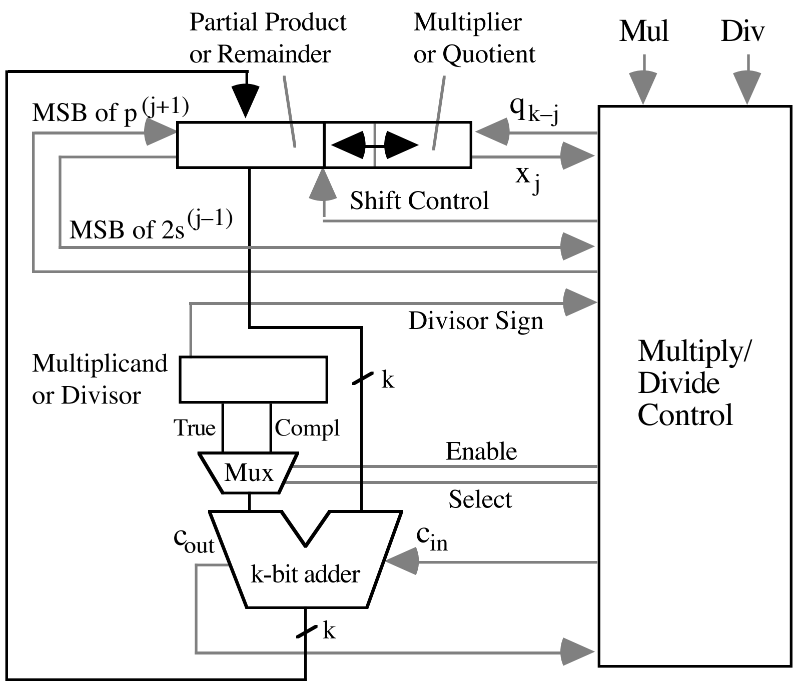


Fig. 15.9 Sequential radix-2 multiply/divide unit.

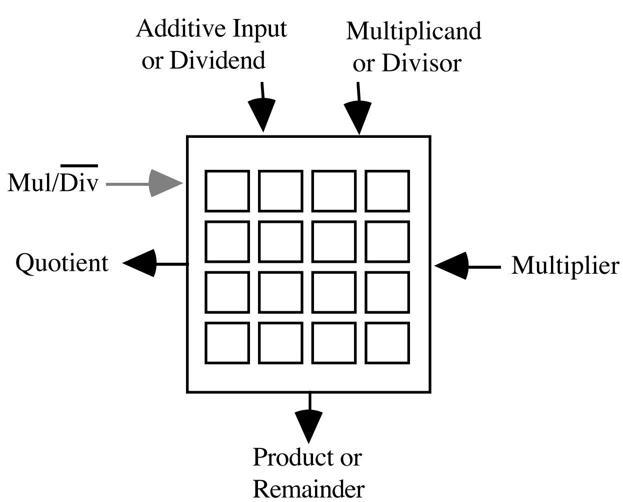


Fig. 15.10 I/O specification of a universal circuit that can act as an array multiplier or array divider.

16 Division by Convergence

Chapter Goals

Show how by using multiplication as the basic operation in each iteration of division, the number of iterations can be reduced

Chapter Highlights

Digit-recurrence as a convergence method
Convergence by Newton-Raphson iteration
Computing the reciprocal of a number
Hardware implementation and fine tuning

Chapter Contents

16.1 General Convergence Methods
16.2 Division by Repeated Multiplications
16.3 Division by Reciprocation
16.4 Speedup of Convergence Division
16.5 Hardware Implementation
16.6 Analysis of Lookup Table Size

16.1 General Convergence Methods

$$u^{(i+1)} = f(u^{(i)}, v^{(i)})$$
$$v^{(i+1)} = g(u^{(i)}, v^{(i)})$$

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

We direct the iterations such that one value, say u, converges to some constant.

The value of v (and/or w) then converges to the desired function(s)

The complexity of this method depends on two factors:

a. Ease of evaluating f and g (and h)

b. Rate of convergence (no. of iterations needed)

16.2 Division by Repeated Multiplications

$$q = \frac{z}{d} = \frac{z\, x^{(0)} x^{(1)} \cdots x^{(m-1)}}{d\, x^{(0)} x^{(1)} \cdots x^{(m-1)}} \quad \begin{array}{l}\text{Converges to } q \\ \text{Made to converge to } 1\end{array}$$

To turn the above into a division algorithm,
we face three questions:

1. How to select the multipliers $x^{(i)}$?
2. How many iterations (pairs of multiplications)?
3. How to implement in hardware?

Formulate as convergence computation, for d in [1/2, 1)

$d^{(i+1)} = d^{(i)}x^{(i)}$ Set $d^{(0)} = d$; make $d^{(m)}$ converge to 1

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

Q1: How to select the multipliers $x^{(i)}$?

$$x^{(i)} = 2 - d^{(i)}$$

This choice transforms the recurrence equations into:

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

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

Q2: How quickly does $d^{(i)}$ converge to 1?

$$d^{(i+1)} = d^{(i)}(2 - d^{(i)}) = 1 - (1 - d^{(i)})^2$$

$$1 - d^{(i+1)} = (1 - d^{(i)})^2$$

If $1 - d^{(i)} \le \varepsilon$, we have $1 - d^{(i+1)} \le \varepsilon^2$: quadratic convergence
In general, for k-bit operands, we need

$2m - 1$ multiplications and m 2's complementations

where $m = \lceil \log_2 k \rceil$

Table 16.1 Quadratic convergence in computing z/d by repeated multiplications, where $1/2 \le d = 1 - y < 1$

i	$d^{(i)} = d^{(i-1)}x^{(i-1)}$, with $d^{(0)} = d$	$x^{(i)} = 2 - d^{(i)}$
0	$1 - y = (.1\text{xxx xxxx xxxx xxxx})_{\text{two}} \ge 1/2$	$1 + y$
1	$1 - y^2 = (.11\text{xx xxxx xxxx xxxx})_{\text{two}} \ge 3/4$	$1 + y^2$
2	$1 - y^4 = (.1111\ \text{xxxx xxxx xxxx})_{\text{two}} \ge 15/16$	$1 + y^4$
3	$1 - y^8 = (.1111\ 1111\ \text{xxxx xxxx})_{\text{two}} \ge 255/256$	$1 + y^8$
4	$1 - y^{16} = (.1111\ 1111\ 1111\ 1111)_{\text{two}} = 1 - \text{ulp}$	

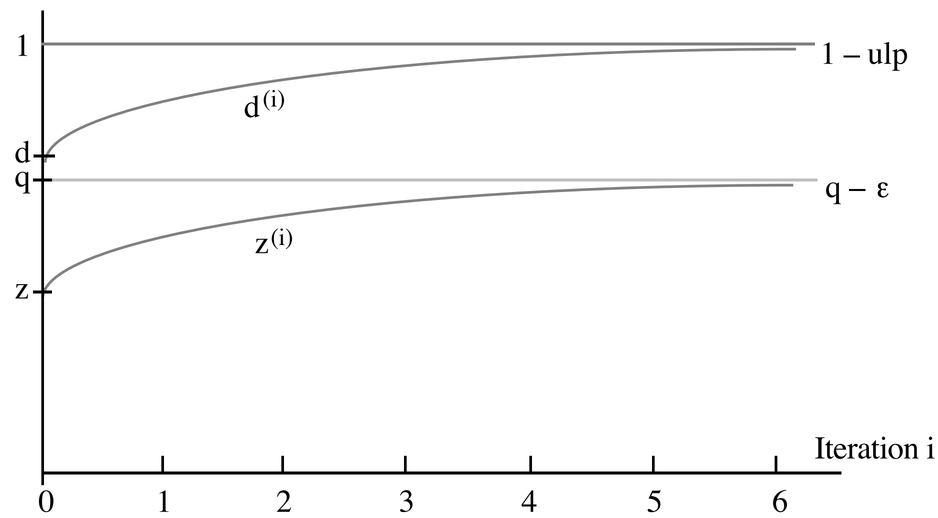


Fig. 16.1. Graphical representation of convergence in division by repeated multiplications.

16.3 Division by Reciprocation

To find $q = z/d$, compute $1/d$ and multiply it by z.
Particularly efficient if several divisions byd are required

Newton-Raphson iteration to determine a root of $f(x) = 0$

Start with some initial estimate $x^{(0)}$ for the root

Iteratively refine the estimate using the recurrence

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

Justification: $\tan \alpha^{(i)} = f'(x^{(i)}) = f(x^{(i)}) / (x^{(i)} - x^{(i+1)})$

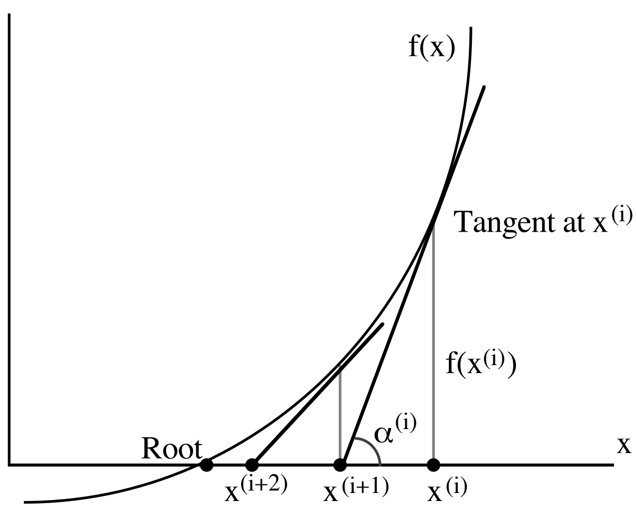


Fig. 16.2 Convergence to a root of f(x) = 0 in the Newton-Raphson method.

To compute 1/d, find the root of $f(x) = 1/x - d$

$f'(x) = -1/x^2$, leading to the recurrence:

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

Two multiplications and a 2's complementation per iteration

Let $\delta^{(i)} = 1/d - x^{(i)}$ be the error at the ith iteration. Then:

$$\delta^{(i+1)} = 1/d - x^{(i+1)} = 1/d - x^{(i)} (2 - x^{(i)}d)$$
$$= d(1/d - x^{(i)})^2 = d(\delta^{(i)})^2$$

Since $d < 1$, we have $\delta^{(i+1)} < (\delta^{(i)})^2$

Choosing the initial value $x^{(0)}$

$0 < x^{(0)} < 2/d \Rightarrow |\delta^{(0)}| < 1/d \Rightarrow$ guaranteed convergence

For d in [1/2, 1):

simple choice $x^{(0)} = 1.5 \Rightarrow |\delta^{(0)}| \le 0.5$

better approx. $x^{(0)} = 4(\sqrt{3} - 1) - 2d = 2.9282 - 2d$

16.4 Speedup of Convergence Division

Division can be done via $2\lceil \log_2 k \rceil - 1$ multiplications

This is not yet very impressive

64-bit numbers, 5-ns multiplier $\Rightarrow$ 55-ns division

Three types of speedup are possible:

Reducing the number of multiplications

Using narrower multiplications

Performing the multiplications faster

Convergence is slow in the beginning

It takes 6 multiplications to get 8 bits of convergence and another 5 to go from 8 bits to 64 bits

$$d\underbrace{x^{(0)}x^{(1)}x^{(2)}}_{x^{(0+)} \text{ read from table}} = (0.1111\ 1111 \cdots)_{two}$$

A $2^w \times w$ lookup table is necessary and sufficient for w bits of convergence after the first pair of multiplications

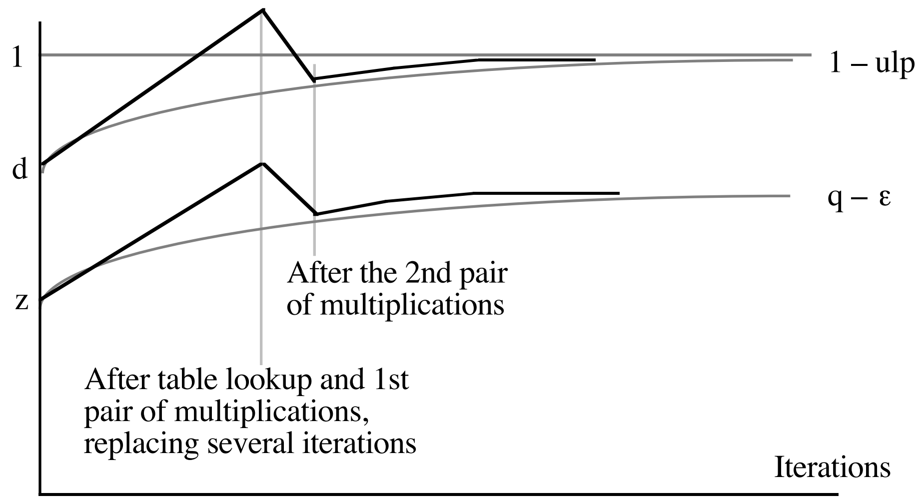


Fig. 16.3 Convergence in division by repeated multiplications with initial table lookup.

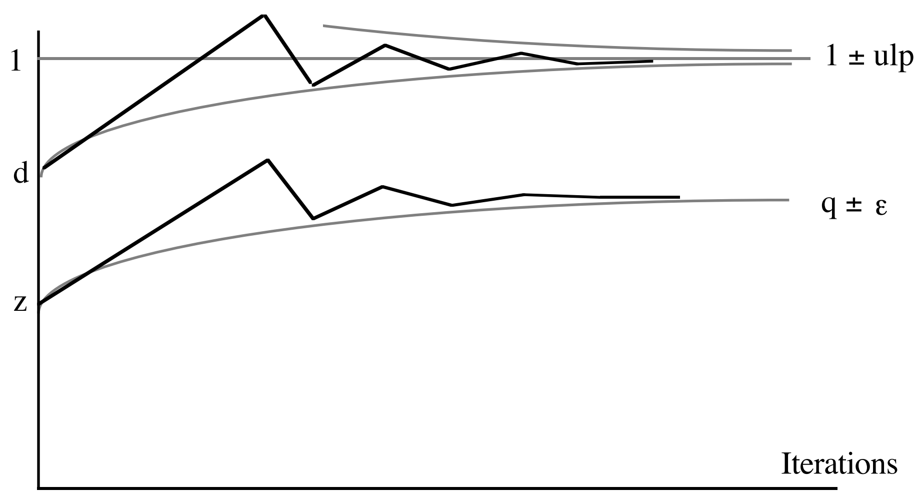


Fig. 16.4 Convergence in division by repeated multiplications with initial table lookup and the use of truncated multiplicative factors.

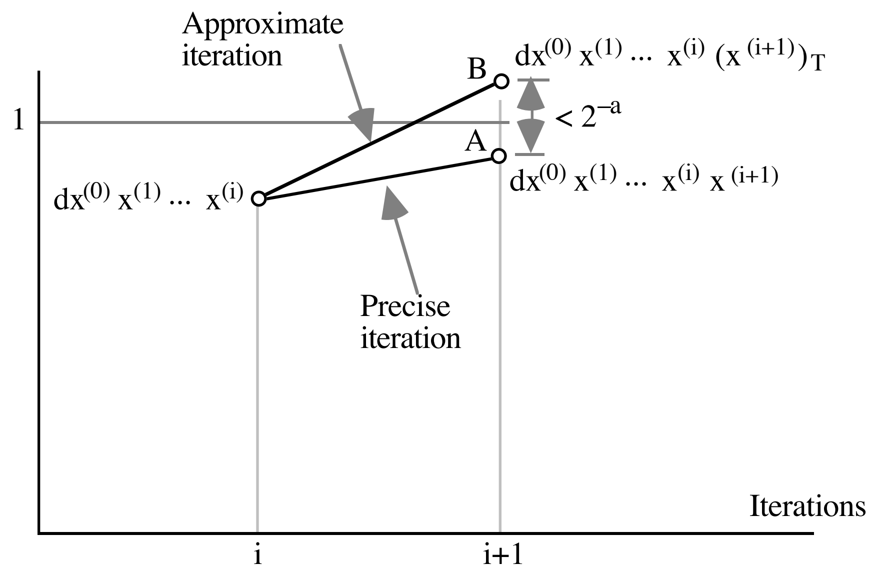


Fig. 16.5 One step in convergence division with truncated multiplicative factors.

Example (64-bit multiplication)

Table of size 256 × 8 = 2K bits for the lookup step

Then we need pairs of multiplications with the multiplier being 9 bits, 17 bits, and 33 bits wide

The final step involves a single 64 × 64 multiplication

16.5 Hardware Implementation

Repeated multiplications

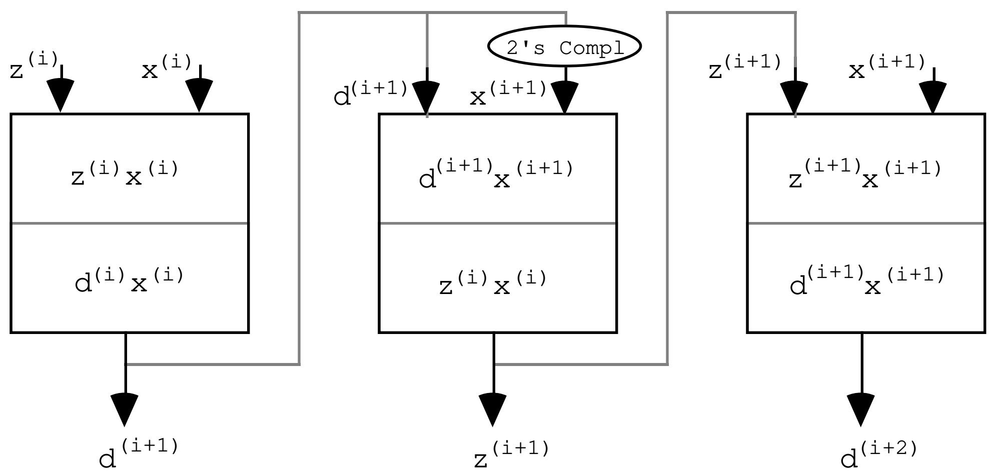


Fig. 16.6 Two multiplications fully overlapped in a 2-stage pipelined multiplier.

Reciprocation

Can begin with a good approximation to the reciprocal by

consulting a large table

table lookup, along with interpolation

augmenting the multiplier tree

16.6 Analysis of Lookup Table Size

Table 16.2 Sample entries in the lookup table replacing the first 4 multiplications in division by repeated multiplications

Address	$d = 0.1\ xxxx\ \ xxxx$	$x^{(0+)} = 1.\ xxxx\ \ xxxx$
55	0011 0111	1010 0101
64	0100 0000	1001 1001

Example: derivation of the table entry for 55

$$\frac{311}{512} \le d < \frac{312}{512}$$

For 8 bits of convergence, the table entry f must satisfy

$$\frac{311}{512}(1 + .f) \ge 1 - 2^{-8} \qquad\qquad \frac{312}{512}(1 + .f) \le 1 + 2^{-8}$$

Thus

$$\frac{199}{311} \le .f \le \frac{101}{156} \qquad \text{or for the integer } f = 256 \times .f$$

$$163.81 \le f \le 165.74$$

Two choices: $164 = (1010\ 0100)_{two}$ or $165 = (1010\ 0101)_{two}$

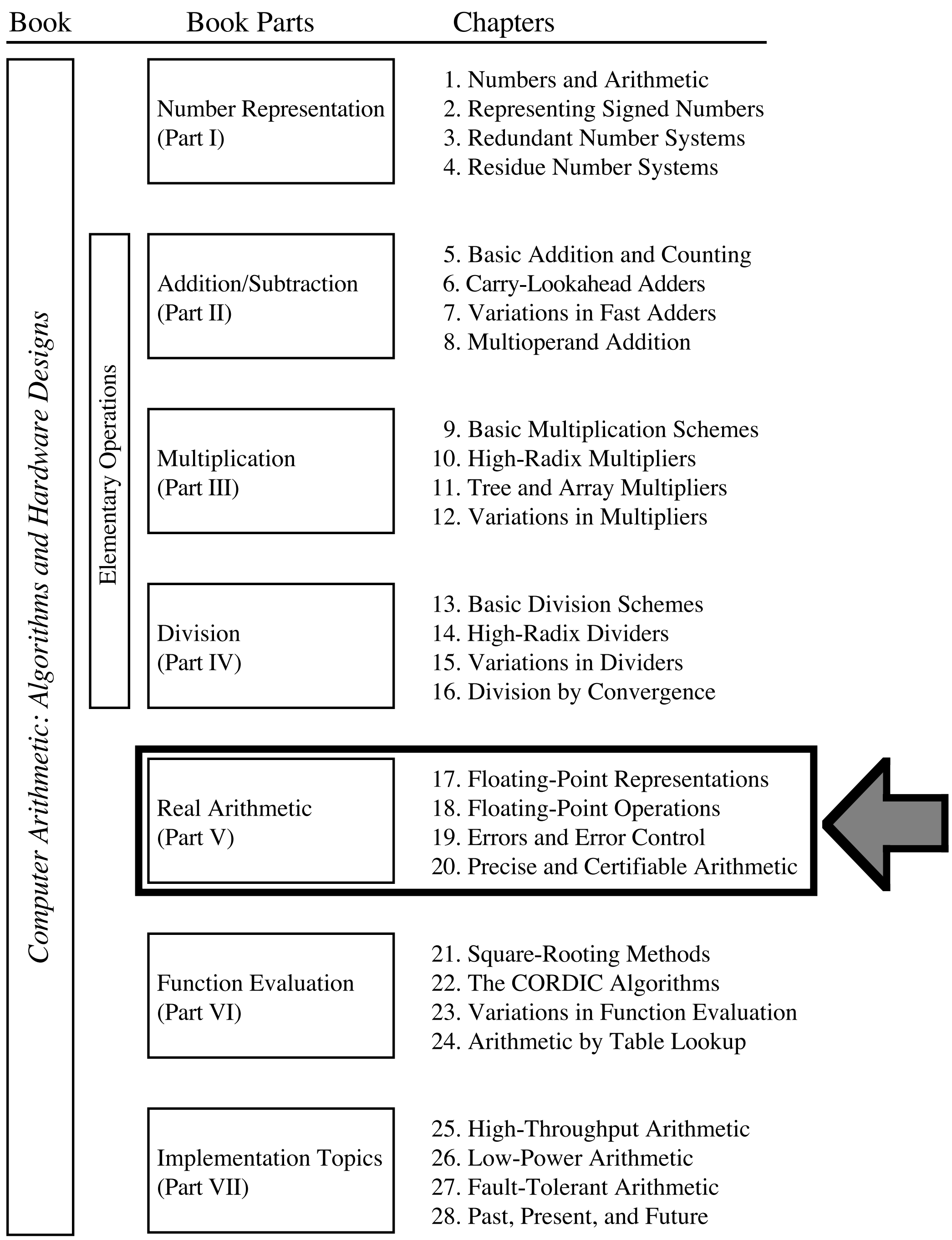


Fig. P.1 The structure of this book in parts and chapters.

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

Chapter 17 Floating-Point Representations

Chapter 18 Floating-Point Operations

Chapter 19 Errors and Error Control

Chapter 20 Precise and Certifiable Arithmetic

17 Floating-Point Representations

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
17.2 The ANSI/IEEE Floating-Point Standard
17.3 Basic Floating-Point Algorithms
17.4 Conversions and Exceptions
17.5 Rounding Schemes
17.6 Logarithmic Number Systems

17.1 Floating-Point Numbers

No finite number system can represent all real numbers

Various systems can be used for a subset of real numbers

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

Fixed-point numbers

$x = (0000\ 0000\ .\ 0000\ 1001)_{two}$ Small number

$y = (1001\ 0000\ .\ 0000\ 0000)_{two}$ Large number

Floating-point numbers

$x = \pm s \times b^e$ or $\pm$ significand $\times$ base^{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)

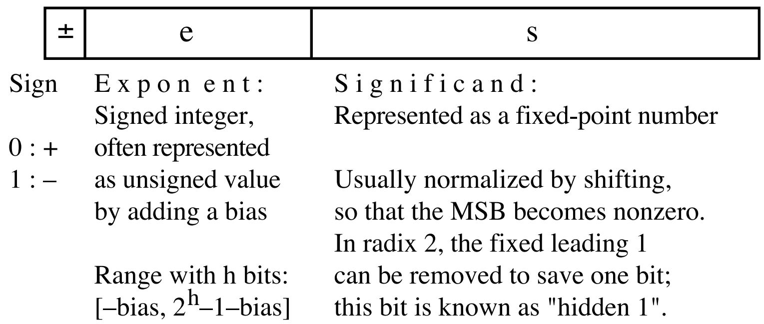


Fig. 17.1 Typical floating-point number format.

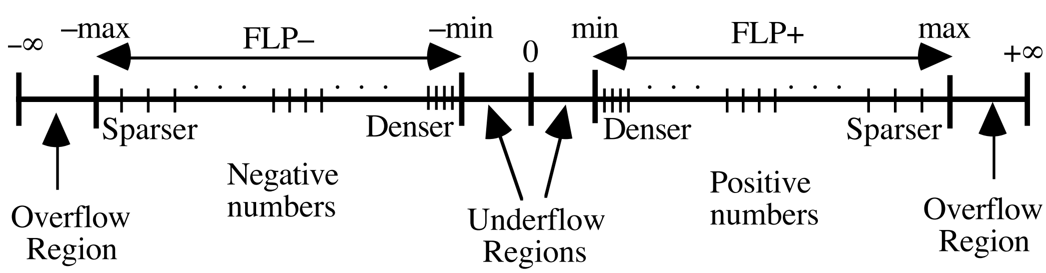


Fig. 17.2 Subranges and special values in floating-point number representations.

17.2 The ANSI/IEEE Floating-Point Standard

Sign	Biased exponent	Significand s = 1.f (the 1 is hidden)
±	e + bias	f

32-bit:	8 bits, bias= 127	23+1 bits, single-precision or short format
64-bit:	11 bits, bias=1023	52+1 bits, double-precision or long format

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	$2^{-126} \approx 1.2 \times 10^{-38}$	$2^{-1022} \approx 2.2 \times 10^{-308}$
max	$\approx 2^{128} \approx 3.4 \times 10^{38}$	$\approx 2^{1024} \approx 1.8 \times 10^{308}$

Operations on special operands:

Ordinary number $\div$ $(+\infty)$ = ± 0

$(+\infty)$ $\times$ Ordinary number = $\pm\infty$

NaN + Ordinary number = NaN

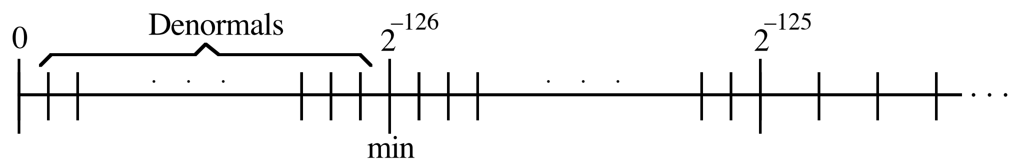


Fig. 17.4 Denormals in the IEEE single-precision format.

The IEEE floating-point standard also defines

The four basic arithmetic operations (+, –, ×, ÷) and $\sqrt{x}$ (must match the results that would be obtained if intermediate computations were infinite-precision)

Extended formats for greater internal precision

Single-extended: $\geq$ 11 bits for exponent
$\geq$ 32 bits for significand
bias unspecified, but
exp range $\supset$ [–1022, 1023]

Double-extended: $\geq$ 15 bits for exponent
$\geq$ 64 bits for significand
exp range $\supset$ [–16 382, 16 383]

17.3 Basic Floating-Point Algorithms

Addition/Subtraction

Assume e1 ≥ e2; need alignment shift, or 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

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

Rounding introduces additional complications

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 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

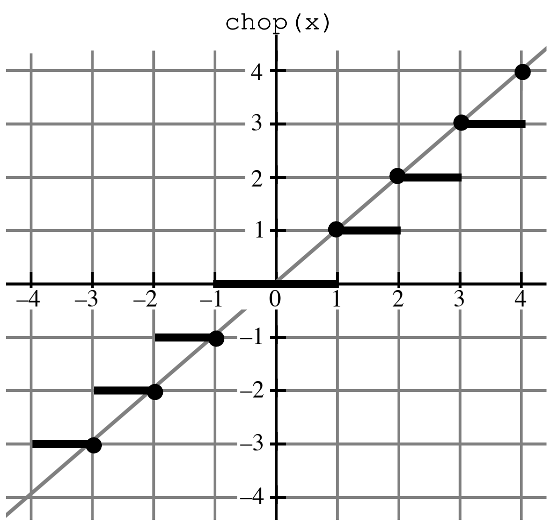
addition	$(+\infty) + (-\infty)$
multiplication	$0 \times \infty$
division	$0 / 0$ or ∞ / ∞
square-root	operand < 0

17.5 Rounding Schemes

$$x_{k-1}x_{k-2} \cdots x_1 x_0 \,.\, x_{-1}x_{-2} \cdots x_{-l} \xrightarrow{\text{Round}} y_{k-1}y_{k-2} \cdots y_1 y_0 \,.$$

Special case: truncation or chopping

$$x_{k-1}x_{k-2} \cdots x_1 x_0 \,.\, x_{-1}x_{-2} \cdots x_{-l} \xrightarrow{\text{Chop}} x_{k-1}x_{k-2} \cdots x_1 x_0 \,.$$



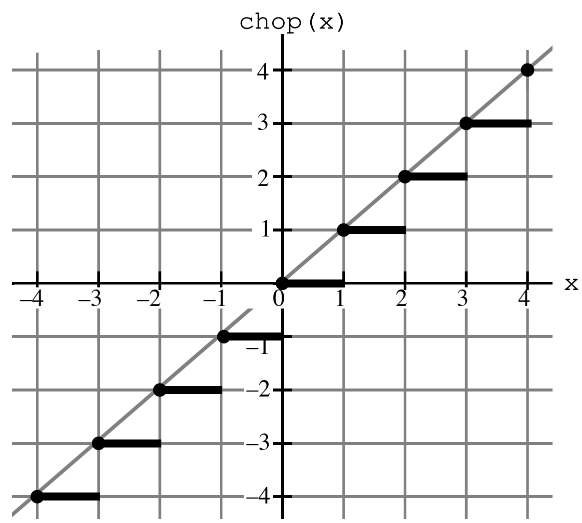


Fig. 17.5 Truncation or chopping of a signed-magnitude number (same as round towards 0).

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

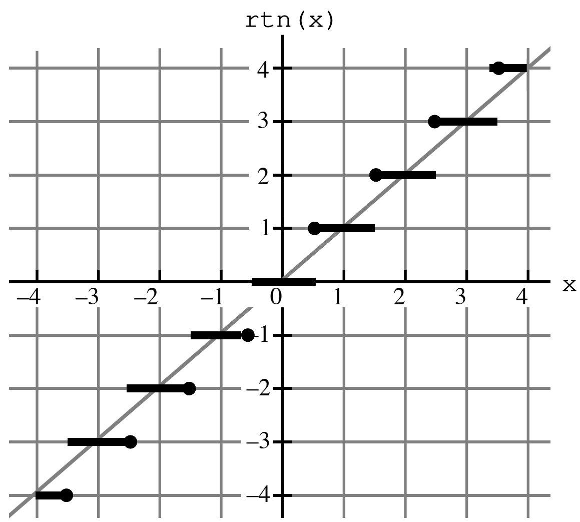


Fig. 17.7 Rounding of a signed-magnitude value to the nearest number.

Ordinary rounding has a slight upward bias

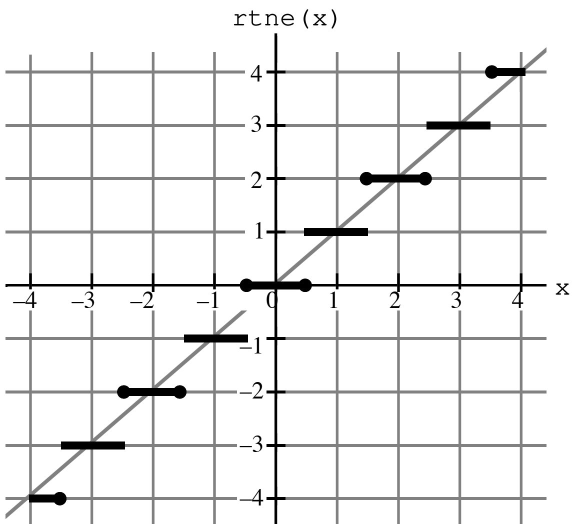
Assume that $(x_{k-1}x_{k-2} \cdots x_1x_0 . x_{-1}x_{-2})_{two}$ is to be rounded to an integer $(y_{k-1}y_{k-2} \cdots y_1y_0 .)_{two}$

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

If these four cases occur with equal probability, then the average error is 0.125

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



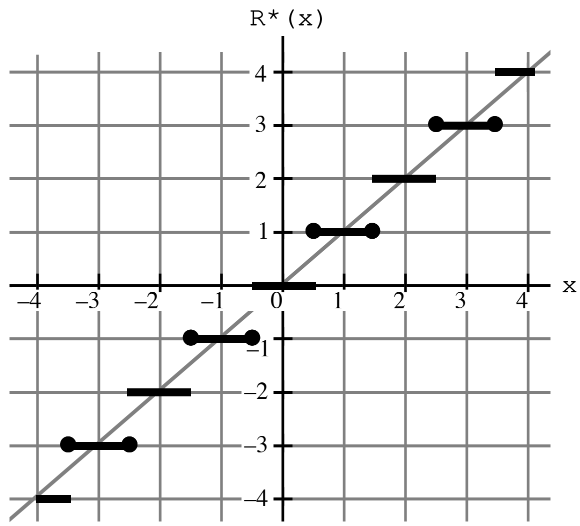


Fig. 17.8 Rounding to the nearest even number.

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

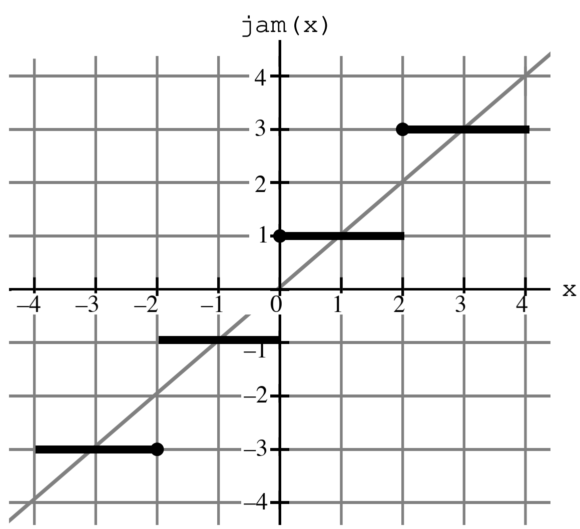


Fig. 17.10 Jamming or von Neumann rounding.

32×4-ROM-Round

$$x_{k-1} \cdots x_4 \underbrace{x_3 x_2 x_1 x_0 \, . \, x_{-1}}_{\text{ROM Address}} \cdots x_{-l} \;\longrightarrow\; x_{k-1} \cdots x_4 \underbrace{y_3 y_2 y_1 y_0}_{\text{ROM Data}} \, .$$

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 $x_3 = x_2 = x_1 = x_0 = 1$

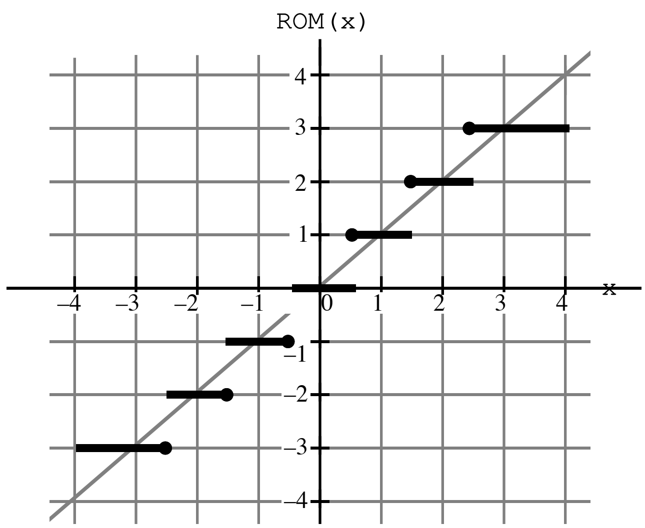


Fig. 17.11 ROM rounding with an 8×2 table.

We may need computation 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

upward-directed rounding (round toward +∞) and

downward-directed rounding (round toward –∞)

(optional features of the IEEE floating-point standard)

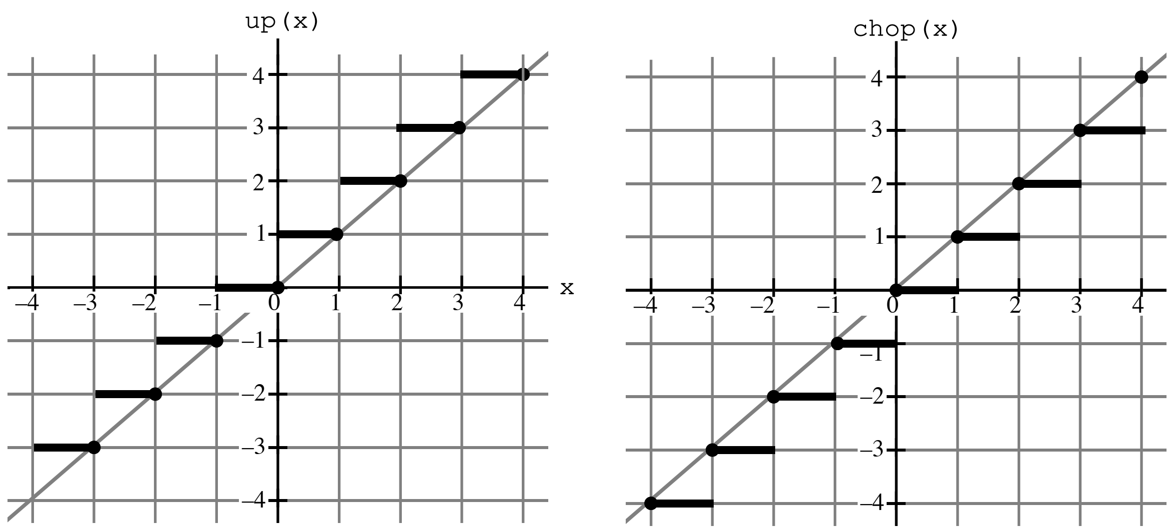


Fig. 17.12 Upward-directed rounding or rounding toward +∞ (see Fig. 17.6 for downward-directed rounding, or rounding toward –∞).

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

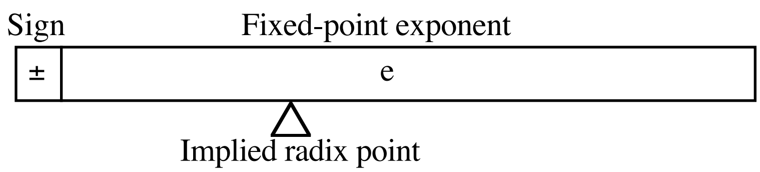


Fig. 17.13 Logarithmic number representation with sign and fixed-point exponent.

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

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

Simple multiply and divide; more complex add and subtract

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

1 1 0 1 1 0 Δ 0 0 1 0 1 1

Sign Radix point

The above represents $-2^{-9.828125} \approx -(0.0011)_{ten}$

number range $\approx [-2^{16}, 2^{16}]$, with min $= 2^{-16}$

18 Floating-Point Operations

Chapter Goals

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

Chapter Highlights

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

Adders need preshift, postshift, rounding

Multipliers and dividers are easy to design

Chapter Contents

18.1 Floating-Point Adders/Subtractors
18.2 Pre- and Postshifting
18.3 Rounding and Exceptions
18.4 Floating-Point Multipliers
18.5 Floating-Point Dividers
18.6 Logarithmic Arithmetic Unit

18.1 Floating-Point Adders/Subtractors

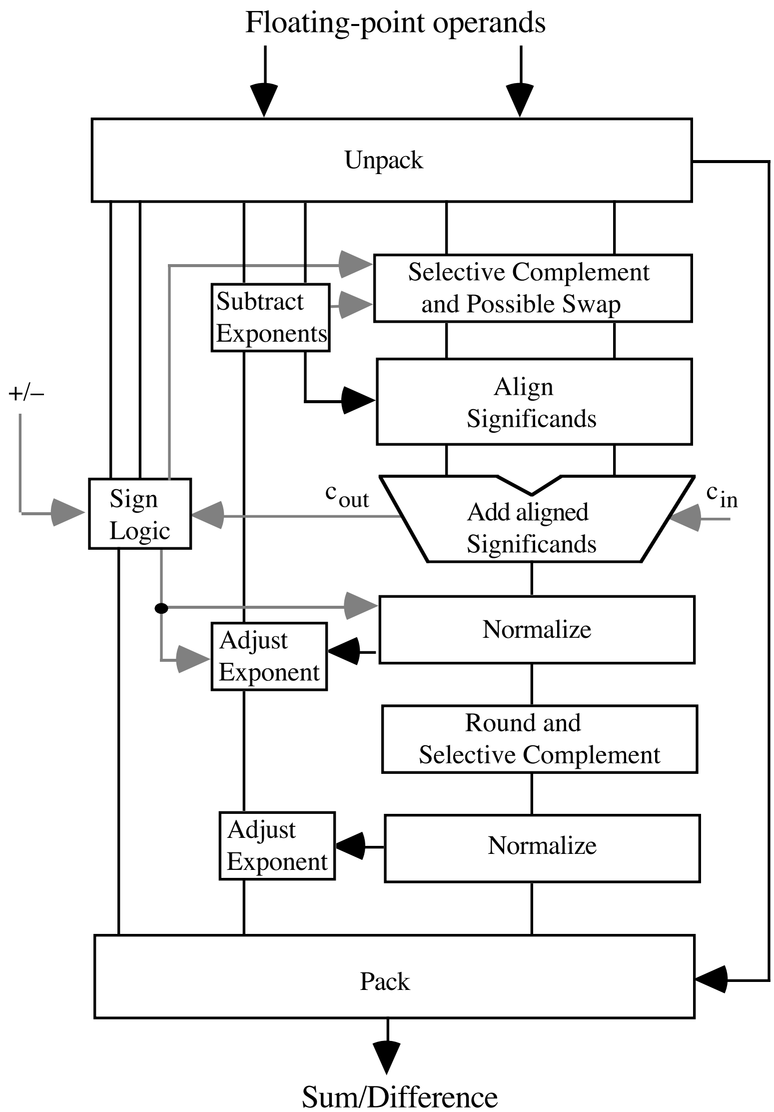


Fig. 18.1 Block diagram of a floating-point adder/subtractor.

Unpacking of the operands involves:

- Separating sign, exponent, and significand
- Reinstating the hidden 1
- Converting the operands to the internal format
- Testing for special operands and exceptions

Packing of the result involves:

- Combining sign, exponent, and significand
- Hiding (removing) the leading 1
- Testing for special outcomes and exceptions

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

Other key parts of a floating-point adder:

- significand aligner or preshifter: Section 18.2
- result normalizer or postshifter, including
  - leading 0s detector/predictor: Section 18.2
- rounding unit: Section 18.3
- sign logic: Problem 18.2

18.2 Pre- and Postshifting

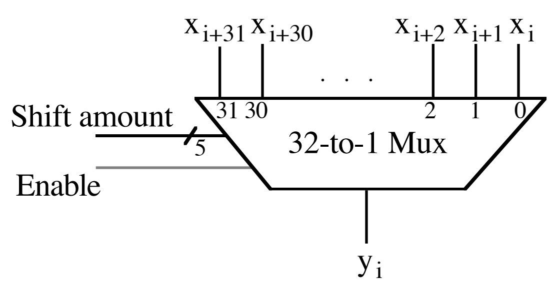


Fig. 18.2 One bit-slice of a single-stage pre-shifter.

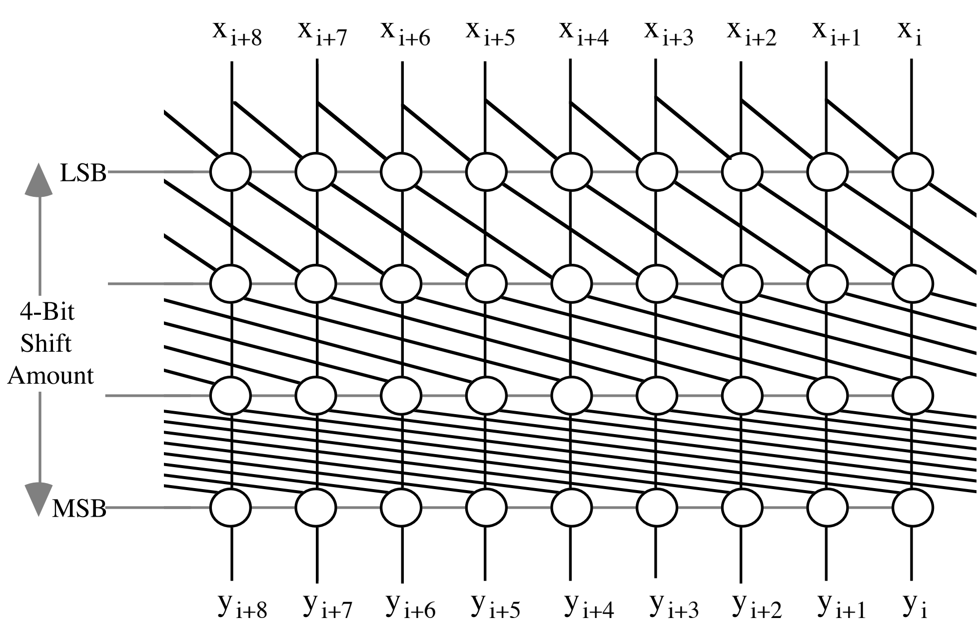


Fig. 18.3 Four-stage combinational shifter for preshifting an operand by 0 to 15 bits.

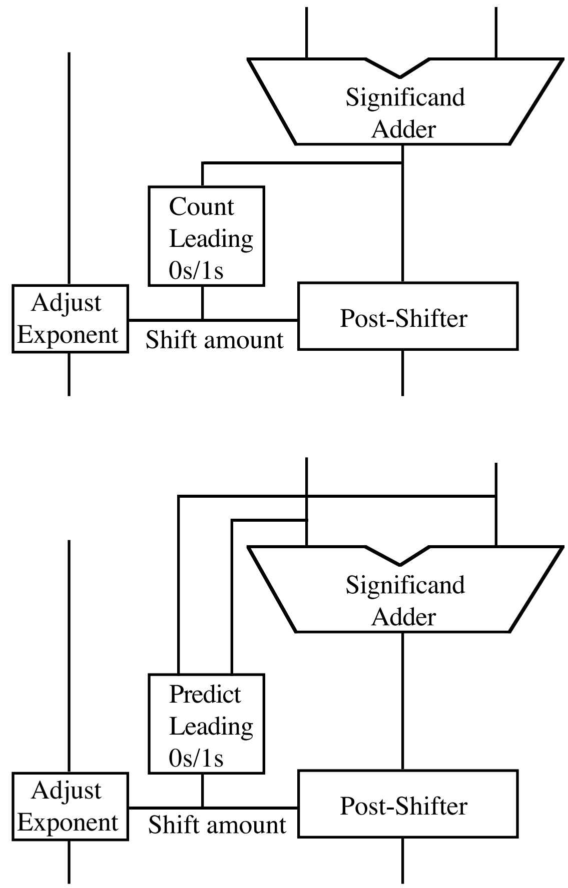


Fig. 18.4 Leading zeros/ones counting versus prediction.

Leading zeros prediction

Adder inputs: $(0x_0.x_{-1}x_{-2}\cdots)_{2\text{'s-compl}}$, $(1y_0.y_{-1}y_{-2}\cdots)_{2\text{'s-compl}}$

How leading 0s/1s can be generated

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

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

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

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

18.3 Rounding and Exceptions

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

G Guard bit

R Round bit

S Sticky bit

Why only three extra bits at the right are adequate

Amount of alignment right-shift

1 bit: G holds the bit that is shifted out, no precision is lost

2 bits or more:

shifted significand has a magnitude in [0, 1/2)

unshifted significand has a magnitude in [1, 2)

difference of aligned significands

has a magnitude in [1/2, 2)

normalization left-shift will be by at most one bit

If a normalization left-shift actually takes place:

R = 0, round down, discarded part < ulp/2

R = 1, round up, discarded part ≥ ulp/2

The only remaining question is establishing if the discarded part is exactly equal to ulp/2, as this information is needed in some rounding schemes

Providing this information is the role of S which is set to the logical OR of all the bits that are right-shifted through it

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

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

Round to nearest even:

Do nothing if $Z_{-l-1} = 0$ or $Z_{-l} = Z_{-l-2} = Z_{-l-3} = 0$

Add ulp $= 2^{-l}$ otherwise

No rounding needed in case of multibit left-shift,

because full precision is preserved in this case

Overflow and underflow exceptions are detected by the exponent adjustment blocks in Fig. 18.1

Overflow can occur only for normalizing right-shift

Uderflow is possible only with normalizing left shifts

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

Zero detection is a special case of leading 0s detection

Determining when the “inexact” exception must be signalled is left as an exercise

18.4 Floating-Point Multipliers

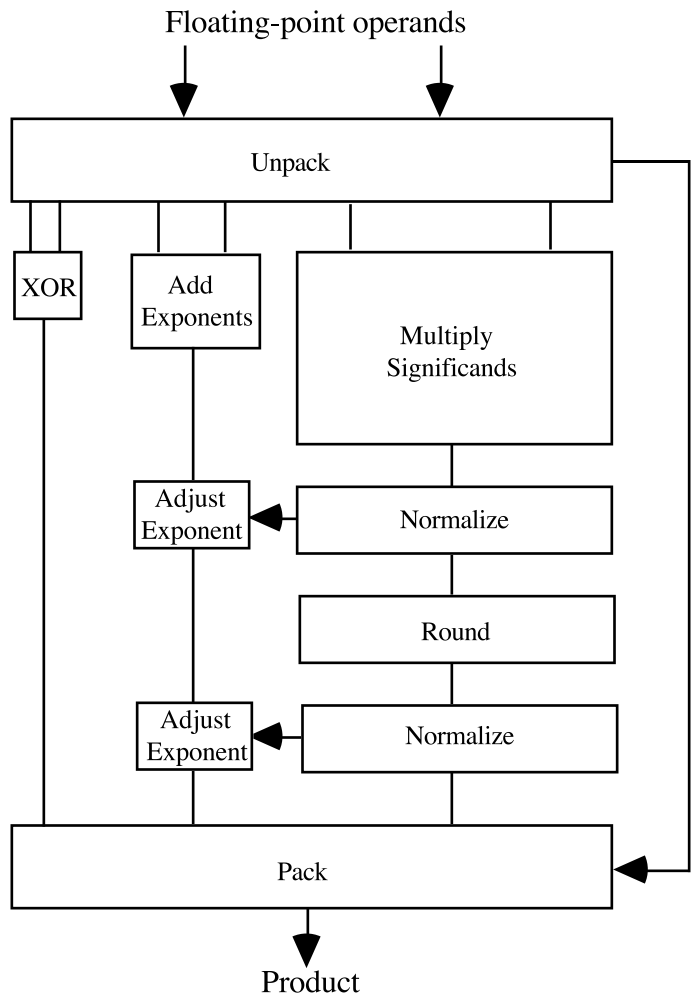


Fig. 18.5 Block diagram of a floating-point multiplier.

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

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

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

18.5 Floating-Point Dividers

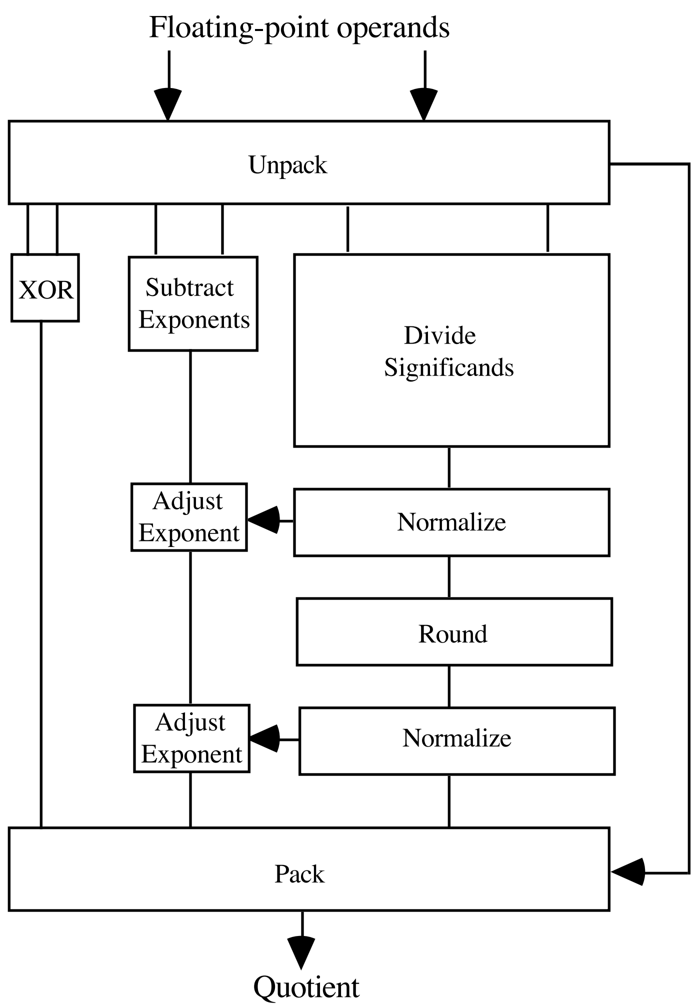


Fig. 18.6 Block diagram of a floating-point divider.

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

The remainder does the job of the sticky bit

18.6 Logarithmic Arithmetic Unit

Add/subtract: (Sx, Lx) ± (Sy, Ly) = (Sz, Lz)

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

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

Given $\Delta = -(\log x - \log y)$, the term

$$\log(1 \pm y/x) = \log(1 \pm \log^{-1}\Delta)$$

is obtained from a table (two tables ϕ^+ and ϕ^- are needed)

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

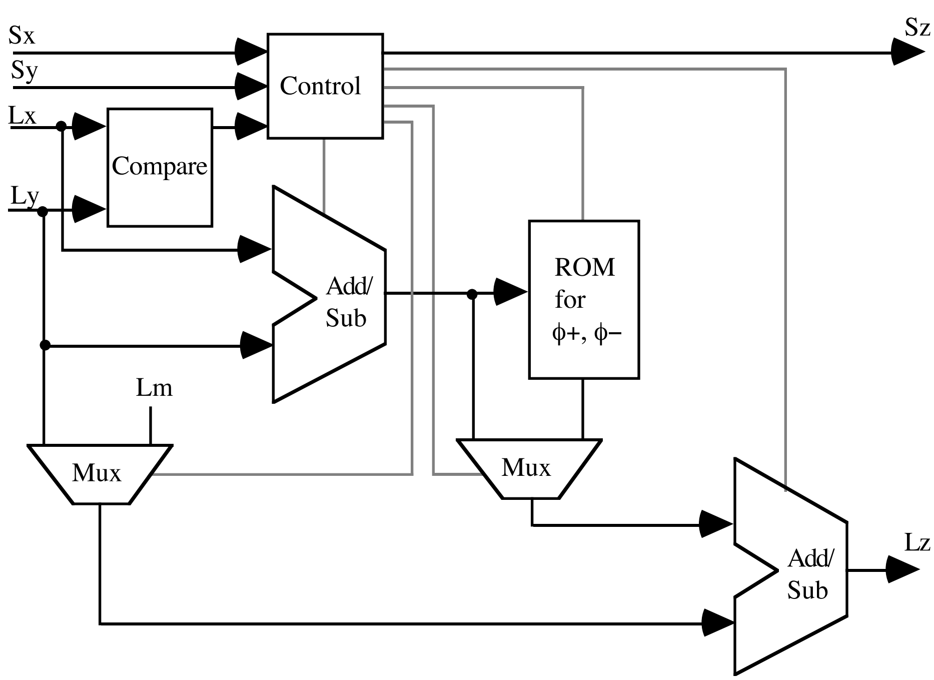


Fig. 18.7 Arithmetic unit for a logarithmic number system.

19 Errors and Error Control

Chapter Goals

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

Chapter Highlights

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

Chapter Contents

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

Disasters Caused by Computer Arithmetic Errors
From: http://www.math.psu.edu/dna/455.f96/disasters.html

Example: Failure of Patriot Missile (1991 Feb. 25)
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)
Specifically, the time in tenths of second
as measured by the system's internal clock
was multiplied by 1/10 to get the time in seconds
Registers are 24 bits wide
1/10 = 0.000 1100 1100 1100 1100 1100 (chopped to 24 b)
Error $\approx$ 0.1100 1100 $\times 2^{-24} \approx$ 0.000 000 095
Error in 100-hr operation
$\approx 0.000000095 \times 100 \times 60 \times 60 \times 10 = 0.34$ s
Distance traveled by SCUD = (0.34 s) $\times$ (1676 m/s) $\approx$ 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) parts
of the code contributed to the problem,
since it meant that inaccuracies did not cancel out

19.1 Sources of Computational Errors

Floating-point arithmetic 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

Example 19.1: Compute 1/99 – 1/100

(decimal floating-point format, 4-digit significand in [1, 10), single-digit signed exponent)

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

$x = 1/99 \approx 1.010 \times 10^{-2}$ Error $\approx 10^{-6}$ 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 $\approx 10^{-6}$ 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 $1/r \le s < 1$, and x_{fp} be its representation in FLP(r, p, A)

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

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

Arithmetic in FLP(r, p, A)

An infinite-precision result is obtained and then chopped, rounded, . . . , to the available precision

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

Consider multiplication, division, addition, and subtraction of <u>positive operands</u> x_{fp} and y_{fp} in FLP(r, p, A)

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

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

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

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

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

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

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

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

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

This situation 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 FLP(r, p, chop(g)) with $g \ge 1$ and $-x < y < 0 < x$, we have:

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

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

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

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:

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

$$\text{Relative error} = \frac{10^{-3} - 0.544 \times 10^{-4}}{0.544 \times 10^{-4}} \approx 17.38$$

(i.e., the result is 1738% larger than the correct sum!)

With 1 guard digit, we get:

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

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

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

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

$= 0.123\ 41\times10^5 +_{fp} (-0.123\ 40\times10^5 +_{fp} 0.143\ 21\times10^1)$

$= 0.123\ 41 \times 10^5 -_{fp} 0.123\ 39 \times 10^5 = \underline{0.200\ 00 \times 10^1}$

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

$= (0.123\ 41\times10^5 -_{fp} 0.123\ 40\times10^5) +_{fp} 0.143\ 21\times10^1$

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

The two results differ by about 20%!

A possible remedy: unnormalized arithmetic.

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

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

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

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

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

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

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:

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

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

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

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

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

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

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

Other 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 $a \approx b + c$, Kahan's version returns accurate results:

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

19.3 Worst-Case Error Accumulation

In a sequence of computations, arithmetic or round-off errors might accumulate

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 = $\text{ulp}^2/2$

Total worst-case error = $1024 \times \text{ulp}^2/2$

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:

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

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

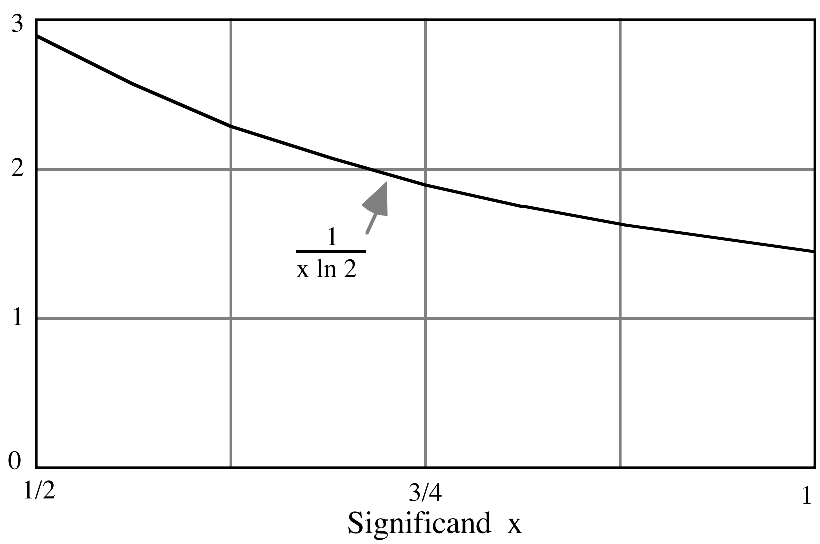


Fig. 19.1 Probability density function for the distribution of normalized significands in FLP(r=2, p, A).

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

$.1234\times10^5 +_{fp} .0000\times10^{10} = .0000\times10^{10}$

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 produce comparable results, then we are probably safe

d. Interval arithmetic

An interval $[x_{lo}, x_{hi}]$ represents x, $x_{lo} \le x \le x_{hi}$

With x_{lo}, x_{hi}, y_{lo}, $y_{hi} > 0$, to find $z = x / y$, we compute

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

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}, and x_{alt} can be from a_{fp}, b_{fp}, and x_{fp}

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

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

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 × ulp additional error in each of the inputs a, b, and x

20 Precise and Certifiable Arithmetic

Chapter Goals

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

Chapter Highlights

More precise computation via
multi- or variable-precision arithmetic
Result certification via
exact or error-bounded arithmetic
Precise/exact arithmetic with low overhead

Chapter Contents

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

20.1 High Precision and Certifiability

There are two aspects of precision to discuss:

Results possessing adequate precision
Being able to provide assurance of the same

We consider three distinct approaches
for coping with precision issues:

1. Obtaining completely trustworthy results
via exact arithmetic

2. Making the arithmetic highly precise
in order to raise our confidence
in the validity of the results:
multi- or variable-precision arithmetic

3. Performing ordinary or high-precision calculations,
while keeping track of potential error accumulation
(can lead to fail-safe operation)

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

20.2 Exact Arithmetic

Continued fractions

Any unsigned rational number x = p/q has a unique continued-fraction expansion

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

with $a_0 \geq 0$, $a_m \geq 2$, and $a_i \geq 1$ for $1 \leq i \leq m-1$

Example: continued-fraction representation for 277/642

$$\frac{277}{642} = 0 + \cfrac{1}{2 + \cfrac{1}{3 + \cfrac{1}{6 + \cfrac{1}{1 + \cfrac{1}{3 + \cfrac{1}{3}}}}}} = [0/2/3/6/1/3/3]$$

0

1/2

3/7

19/44

Approximations for finite representation can be obtained by limiting the number of "digits" in the continued-fraction representation.

Fixed-slash number systems

Numbers are represented by the ratio p/q of two integers

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

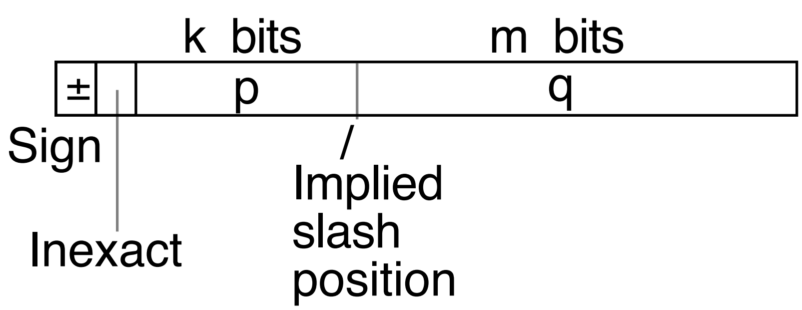


Fig. 20.1 Example fixed-slash number representation format.

The space waste due to multiple representations such as 3/5 = 6/10 = 9/15 = · · · is no more than one bit, because:

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

Floating-slash number systems

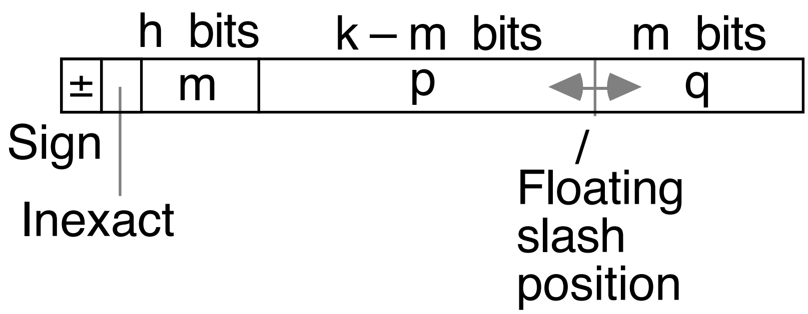


Fig. 20.2 Example floating-slash representation format.

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

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

20.3 Multiprecision Arithmetic

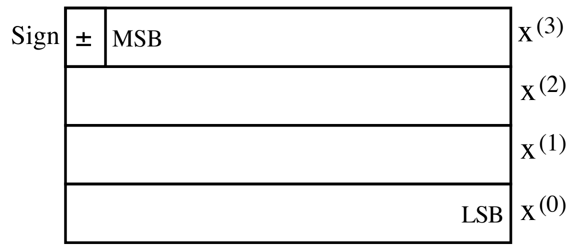


Fig. 20.3 Example quadruple-precision integer format.

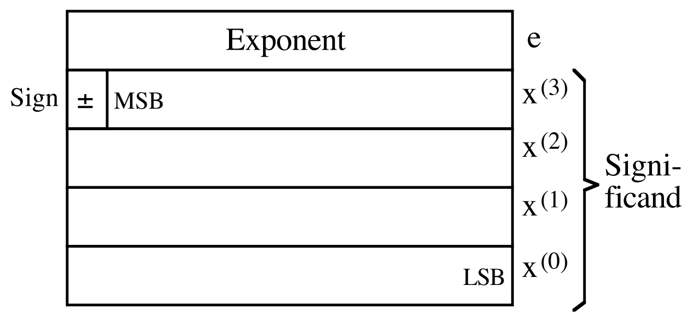


Fig. 20.4 Example quadruple-precision floating-point format.

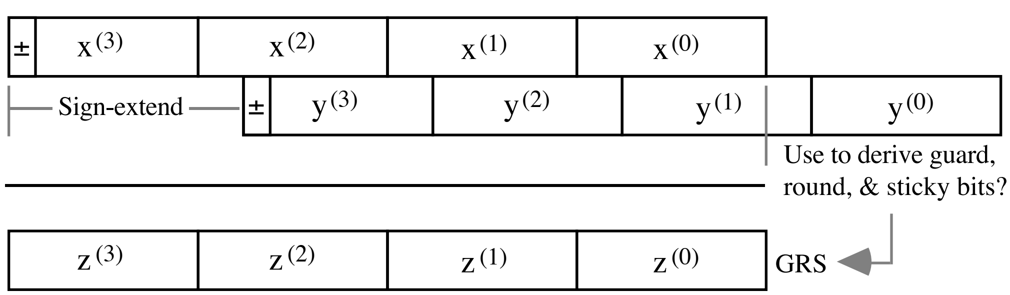


Fig. 20.5 Quadruple-precision significands aligned for the floating-point addition $z = x +_{fp} y$.

20.4 Variable-Precision Arithmetic

Same as multiprecision, but with dynamic adjustments

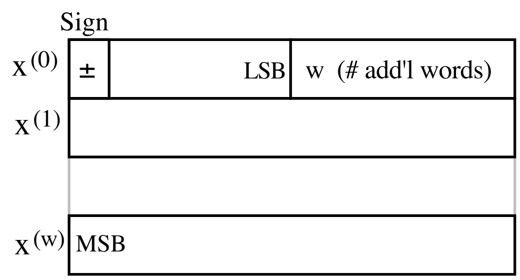


Fig. 20.6 Example variable-precision integer format.

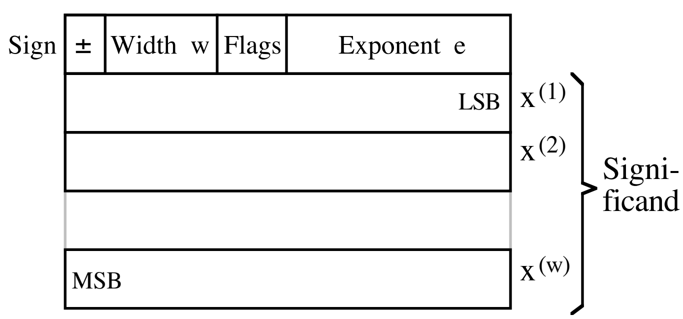


Fig. 20.7 Example variable-precision floating-point format.

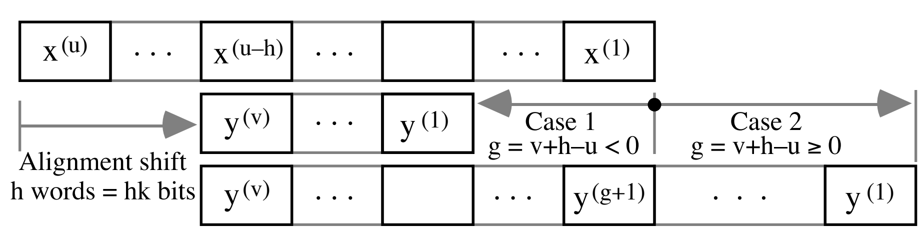


Fig. 20.8 Variable-precision floating-point addition.

20.5 Error-Bounding via Interval Arithmetic

$[a, b]$, $a < b$, is an interval enclosing a number x, $a < x < b$

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

The interval $[a, b]$, $a > b$, represents the empty interval Φ

Intervals can be combined and compared in a natural way

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

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

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

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

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

Interval arithmetic operations are intuitive and efficient

Additive inverse $-x$ of an interval $x = [x_{lo}, x_{hi}]$

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

Multiplicative inverse of an interval $x = [x_{lo}, x_{hi}]$

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

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

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

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

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

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

From the viewpoint of arithmetic calculations, a very important property of interval arithmetic is stated in the following theorem

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

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

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

In particular, with reasonable assumptions about machine arithmetic, the following theorem holds

Theorem 20.2: Consider the execution of an algorithm on real numbers using machine interval arithmetic with precision p in radix r; i.e., in FLP(r, p, $\nabla|\Delta$). If the same algorithm is executed using the precision q, with $q > p$, the bounds for both the absolute error and relative error are reduced by the factor r^{q-p}

(the absolute or relative error itself may not be reduced by this factor; the guarantee applies only to the upper bound)

Strategy for obtaining results with a desired error bound ε:

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

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

is guaranteed to yield the desired accuracy.

20.6 Adaptive and Lazy Arithmetic

Need-based incremental precision adjustment to avoid high-precision calculations that are dictated by worst-case requirements

Interestingly, the opposite of multi-precision arithmetic, which we may call fractional-precision arithmetic, is also of some interest. Example: Intel's MMX

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

if cond1 and cond2 then action

the evaluation of cond2 may be totally skipped if cond1 evaluates to "false"

More generally, lazy evaluation means

postponing all computations or actions

until they become irrelevant or unavoidable

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

Redundant number representations offer some advantages for lazy arithmetic

Since arithmetic on redundant numbers can be performed using MSD-first algorithms, it is possible to produce a small number of digits of the result by using correspondingly less computational effort, until more precision is needed

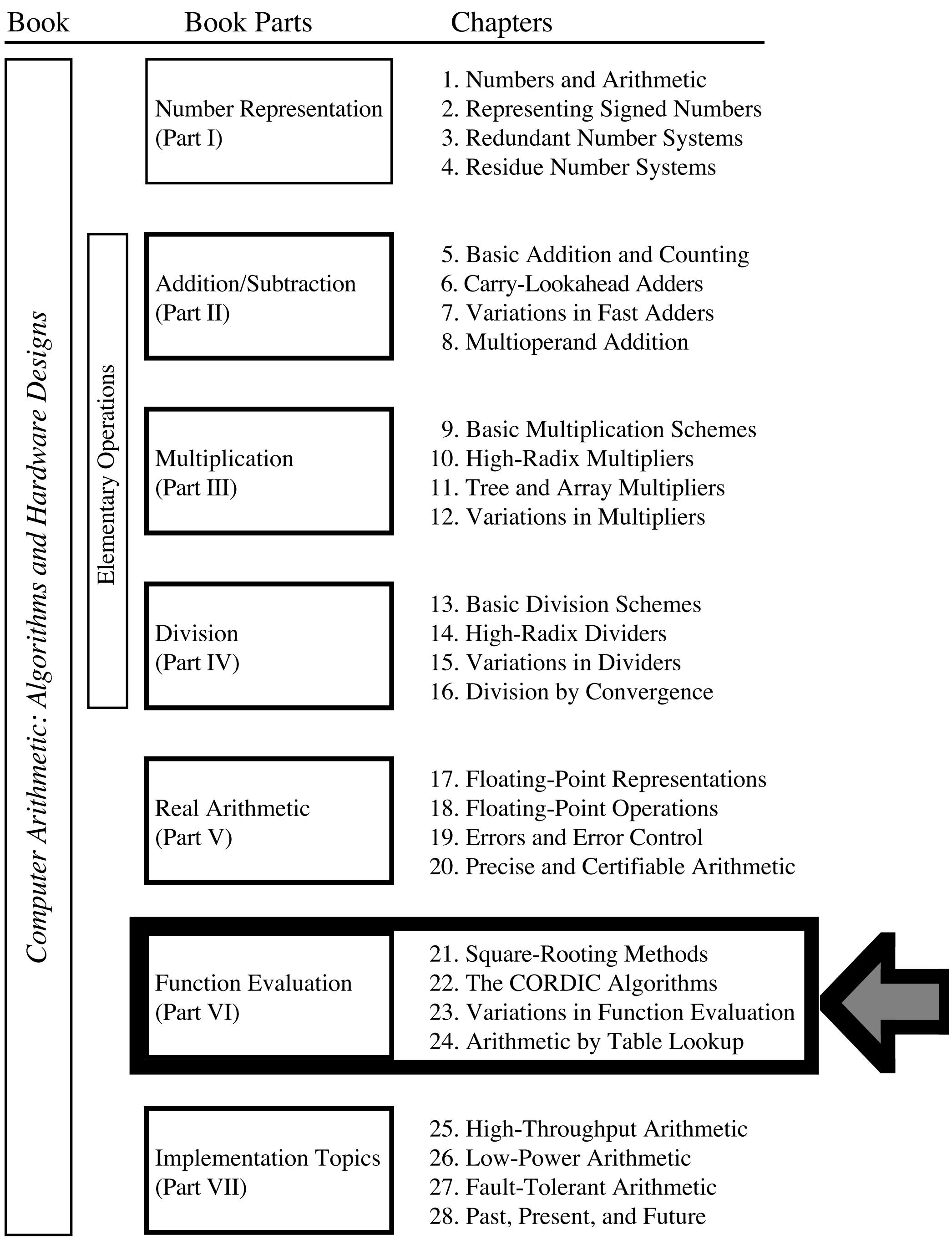


Fig. P.1 The structure of this book in parts and chapters.

Part VI Function Evaluation

Part Goals

Learn algorithms
and implementation methods
for evaluating useful functions

Part Synopsis

Divisionlike square-rooting algorithms
Evaluating $\sqrt{x}$, sin x, tanh x, ln x, e^x, . . . via
series expansion, using +, −, ×, /
convergence computation
Tables: the ultimate in simplicity/flexibility

Part Contents

Chapter 21 Square-Rooting Methods
Chapter 22 The CORDIC Algorithms
Chapter 23 Variations in Function Evaluation
Chapter 24 Arithmetic by Table Lookup

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 ANSI/IEEE standard
Digit-recurrence or divisionlike algorithms
Convergence square-rooting
Square-rooting not special case of division

Chapter Contents

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 Parallel Hardware Square-Rooters

21.1 The Pencil-and-Paper Algorithm

Notation for our discussion of square-rooting algorithms:

z	Radicand	$z_{2k-1}z_{2k-2} \cdots z_1z_0$
q	Square root	$q_{k-1}q_{k-2} \cdots q_1q_0$
s	Remainder $(z - q^2)$	$s_ks_{k-1}s_{k-2} \cdots s_1s_0$

Remainder range: $0 \le s \le 2q$ $k + 1$ digits

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

so q cannot be the correct square-root of z

q_2 | q_1 | q_0	q		$q^{(0)} = 0$
$\sqrt{9 \mid 5\ 2 \mid 4\ 1}$ =	z	$q_2=3$	$q^{(1)} = 3$
9			
0 5 2	$6q_1 \times q_1 \le 52$	$q_1=0$	$q^{(2)} = 30$
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	$s = (377)_{ten}$		$q = (308)_{ten}$

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

Root digit selection

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

Attach next digit q_{k-i-1}; root becomes $q^{(i+1)} = 10q^{(i)} + q_{k-i-1}$

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

$100(q^{(i)})^2 = (10q^{(i)})^2$ subtracted from PR in previous steps

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

```
  q3 | q2 | q1 | q0     q                            q(0) = 0
 ____|____|____|____
√ 0 1| 1 1| 0 1| 1 0    z = (118)ten       q3=1     q(1) = 1
  0 1
  ---------
  0 0 1 1               ≥ 101?    no       q2=0     q(2) = 10
    0 0 0
    ---------
    0 1 1 0 1           ≥ 1001?   yes      q1=1     q(3) = 101
      1 0 0 1
    ---------
      0 1 0 0 1 0       ≥ 10101?  no       q0=0     q(4) = 1010
        0 0 0 0 0
        ---------
        1 0 0 1 0       s = (18)ten        q = (1010)two = (10)ten
```

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

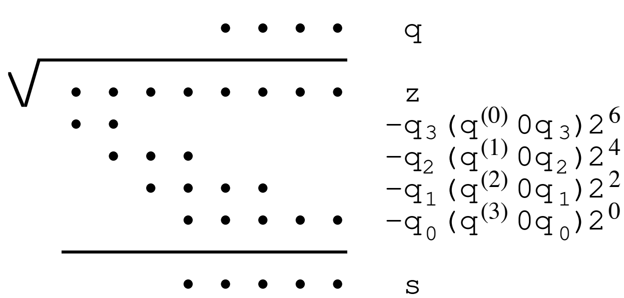


Fig. 21.3 Binary square-rooting in dot notation.

21.2 Restoring Shift/Subtract Algorithm

Consistent with the ANSI/IEEE floating-point standard, we formulate our algorithms for a radicand satisfying $1 \le z < 4$ (1-bit left shift may be needed to make the exponent even)

Notation:

z	Radicand in [1, 4)	$z_1 z_0 . z_{-1} z_{-2} \cdots z_{-l}$
q	Square root in [1, 2)	$1 . q_{-1} q_{-2} \cdots q_{-l}$
s	Scaled remainder	$s_1 s_0 . s_{-1} s_{-2} \cdots s_{-l}$

Binary square-rooting is defined by the recurrence

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

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

where $q^{(j)}$ stands for 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

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

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

z	0 1 . 1 1 0 1 1 0		(118/64)
$s^{(0)} = z - 1$	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
$s^{(1)} = 2s^{(0)}$	0 0 1 . 1 0 1 1 0 0	Restore	
$2s^{(1)}$	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
$s^{(3)} = 2s^{(2)}$	0 1 0 . 0 1 0 0 0 0	Restore	
$2s^{(3)}$	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
$s^{(6)} = 2s^{(5)}$	0 1 0 . 0 1 1 1 0 0	Restore	(156/64)
s (true remainder)	0 . 0 0 0 0 1 0	0 1 1 1 0 0	
q	1 . 0 1 0 1 1 0		(86/64)

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

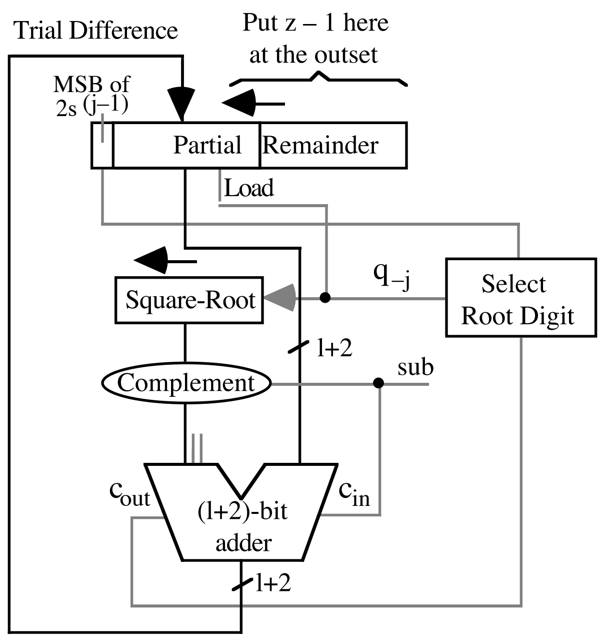


Fig. 21.5 Sequential shift/subtract restoring square-rooter.

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 $q_{-l-1} = 1$

The midway case ($q_{-l-1} = 1$) with only 0s to its right, is impossible

Example: in Fig. 21.4, 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

z	0 1 . 1 1 0 1 1 0		(118/64)
$s^{(0)} = z - 1$	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	$q_{-2}=\bar{1}$	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	$q_{-4}=\bar{1}$	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 (–17/64)	
$+[2\times(1.01011)+2^{-6}]$	1 0 . 1 0 1 1 0 1	Correct	
$s^{(6)}$ (corrected)	0 1 0 . 0 1 1 1 0 0		(156/64)
s (true remainder)	0 . 0 0 0 0 1 0 0 1 1 1 0 0		
q (signed-digit)	1 . 1 $\bar{1}$ 1 $\bar{1}$ 1 1		(87/64)
q (corrected bin)	1 . 0 1 0 1 1 0		(86/64)

Fig. 21.6 Example of nonrestoring binary square-rooting.

Details of binary nonrestoring square-rooting

Root digits are chosen from the set {-1, 1}; on-the-fly conversion to binary

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

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

from the PR. For $q_{-j} = {}^{-}1$, we must subtract

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

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

$2q^{(j-1)} + 2^{-j} = 2[q^{(j-1)} + 2^{-j-1}]$ is formed by appending 01 to the right end of $q^{(j-1)}$ and shifting

But computing the term $2q^{(j-1)} - 2^{-j}$ is problematic

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:

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

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

Additional rule for SRT-like algorithm

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

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 $q^{(j-1)} - r^{-j+1}$, suitably updating them in each step.

Example: r = 4, root digit set [–2, 2]

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
$q_{-j} = \bar{1}$	Add	$2q^{(j-1)} - 2^{-2j}$	shift	Q*111
$q_{-j} = \bar{2}$	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

$q_{-j} = \bar{1} \Rightarrow$ Q := Q*11 Q* := Q*10

$q_{-j} = \bar{2} \Rightarrow$ Q := Q*10 Q* := Q*01

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

Using carry-save addition

As in division, root digit selection can be based on a few bits of the partial remainder and of the partial root

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 for root digit estimation

With proper care, the same lookup table can be used for quotient digit selection in division and root digit selection in square-rooting

To see how, compare the 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 the magnitudes of the partial remainders for division and square-rooting comparable, thus allowing the use of the same tables, we can perform radix-4 square-rooting using the digit set

$$\{-1, -\tfrac{1}{2}, 0, \tfrac{1}{2}, 1\}$$

Conversion from the digit set above to the digit set [–2, 2], or directly to binary, is possible with no extra computation

21.5 Square-Rooting by Convergence

Newton-Raphson method

Choose $f(x) = x^2 - z$ which has 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 needs division, addition, and a one-bit shift

Convergence is quadratic

For $0.5 \le z \le 1$, 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 $z = 0.5$

Table-lookup can yield a better starting estimate for $\sqrt{z}$

For example, if the initial estimate is accurate to within 2^{-8}, then 3 iterations would be sufficient to increase the accuracy of the root to 64 bits.

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

$x^{(0)}$	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}

Convergence square-rooting without division

Rewrite the square-root recurrence as:

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

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: the recurrence above, but with the reciprocal found iteratively; thus interlacing the two computations

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

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

Convergence is less than quadratic but better than linear

Example 21.2: Compute the square root of $z = (1.4)_{ten}$

$$\begin{array}{lllll}
x^{(0)} & = & y^{(0)} & = & 1.0 \\
x^{(1)} & = & 0.5(x^{(0)} + 1.4y^{(0)}) & = & 1.200\,000\,000 \\
y^{(1)} & = & y^{(0)}(2 - x^{(0)}y^{(0)}) & = & 1.000\,000\,000 \\
x^{(2)} & = & 0.5(x^{(1)} + 1.4y^{(1)}) & = & 1.300\,000\,000 \\
y^{(2)} & = & y^{(1)}(2 - x^{(1)}y^{(1)}) & = & 0.800\,000\,000 \\
x^{(3)} & = & 0.5(x^{(2)} + 1.4y^{(2)}) & = & 1.210\,000\,000 \\
y^{(3)} & = & y^{(2)}(2 - x^{(2)}y^{(2)}) & = & 0.768\,000\,000 \\
x^{(4)} & = & 0.5(x^{(3)} + 1.4y^{(3)}) & = & 1.142\,600\,000 \\
y^{(4)} & = & y^{(3)}(2 - x^{(3)}y^{(3)}) & = & 0.822\,312\,960 \\
x^{(5)} & = & 0.5(x^{(4)} + 1.4y^{(4)}) & = & 1.146\,919\,072 \\
y^{(5)} & = & y^{(4)}(2 - x^{(4)}y^{(4)}) & = & 0.872\,001\,394 \\
x^{(6)} & = & 0.5(x^{(5)} + 1.4y^{(5)}) & = & 1.183\,860\,512 \approx \sqrt{1.4}
\end{array}$$

Convergence square-rooting without division (cont.)

A variant is based on computing $1/\sqrt{z}$ and then multiplying the result by z

Use the $f(x) = 1/x^2 - z$ that has a root at $x = 1/\sqrt{z}$ to get

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

Each iteration requires 3 multiplications and 1 addition, but quadratic convergence leads to only a few iterations

The Cray-2 supercomputer uses this method

An initial estimate $x^{(0)}$ for $1/\sqrt{z}$ is plugged in to get $x^{(1)}$

$1.5x^{(0)}$ and $0.5(x^{(0)})^3$ are read out from a table

$x^{(1)}$ is accurate to within half the machine precision, so, a second iteration and a multiplication by z complete the process

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

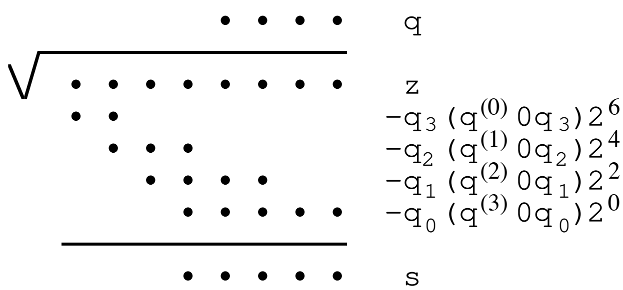
Table lookup provides the starting value $x^{(0)} = 1.3$ for $1/\sqrt{z}$

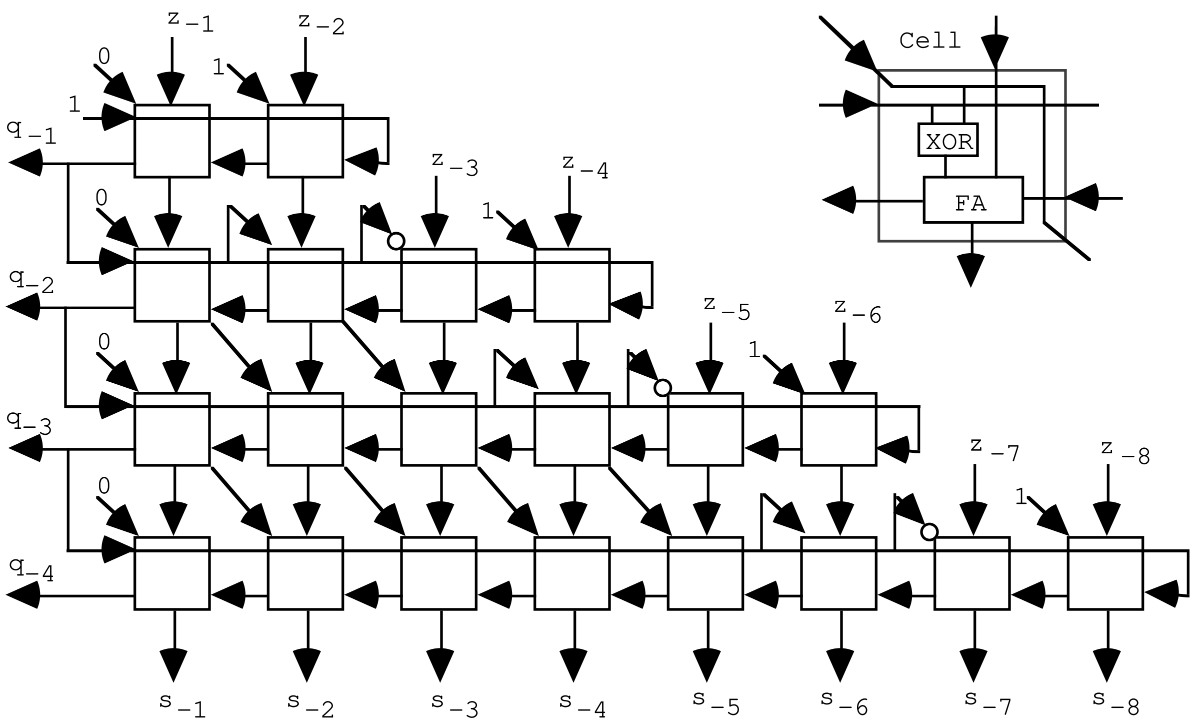
Two iterations, plus a multiplication by z, yield a fairly accurate result

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

21.6 Parallel Hardware Square Rooters

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





Radicand $z = .z_{-1}z_{-2}z_{-3}z_{-4}z_{-5}z_{-6}z_{-7}z_{-8}$
Root $q = .q_{-1}q_{-2}q_{-3}q_{-4}$
Remainder $s = .s_{-1}s_{-2}s_{-3}s_{-4}s_{-5}s_{-6}s_{-7}s_{-8}$

Fig. 21.7 Nonrestoring array square-rooter built of controlled add/subtract cells.

22 The CORDIC Algorithms

Chapter Goals

Learning an elegant and useful convergence method for evaluating trigonometric and many other functions

Chapter Highlights

Basic CORDIC idea: rotate the 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

Chapter Contents

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

Rotate the vector $OE^{(i)}$ with end point at $(x^{(i)}, y^{(i)})$ by $\alpha^{(i)}$

$$x^{(i+1)} = x^{(i)}\cos\alpha^{(i)} - y^{(i)}\sin\alpha^{(i)}$$
$$= (x^{(i)} - y^{(i)}\tan\alpha^{(i)})/(1 + \tan^2\alpha^{(i)})^{1/2}$$
$$y^{(i+1)} = y^{(i)}\cos\alpha^{(i)} + x^{(i)}\sin\alpha^{(i)} \qquad \text{[Real rotation]}$$
$$= (y^{(i)} + x^{(i)}\tan\alpha^{(i)})/(1 + \tan^2\alpha^{(i)})^{1/2}$$
$$z^{(i+1)} = z^{(i)} - \alpha^{(i)}$$

Goal: eliminate the divisions by $(1 + \tan^2\alpha^{(i)})^{1/2}$ and choose $\alpha^{(i)}$ so that $\tan\alpha^{(i)}$ is a power of 2

Elimination of the divisions by $(1 + \tan^2\alpha^{(i)})^{1/2}$

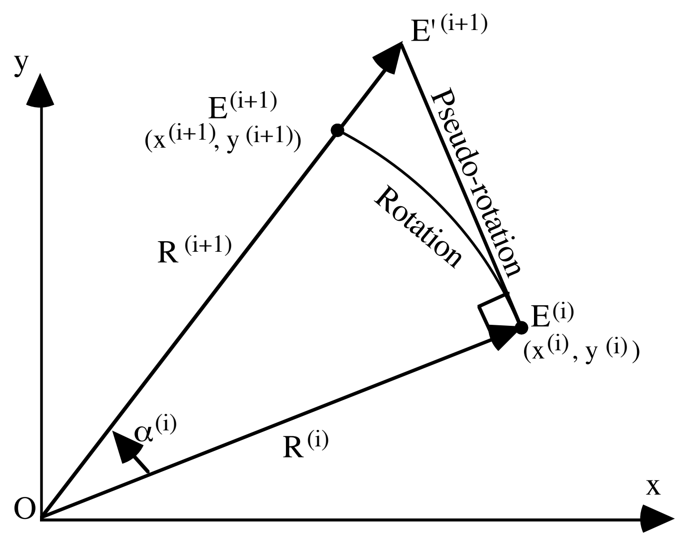


Fig. 22.1 A pseudorotation step in CORDIC.

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

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

The coordinates of the new end point $E'^{(i+1)}$ after pseudorotation is derived by multiplying the coordinates of $E^{(i+1)}$ by the expansion factor

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

$$y^{(i+1)} = y^{(i)} + x^{(i)} \tan \alpha^{(i)} \qquad \text{[Pseudorotation]}$$

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

Assuming $x^{(0)} = x$, $y^{(0)} = y$, and $z^{(0)} = z$, after m real rotations by the angles $\alpha^{(1)}, \alpha^{(2)}, \ldots, \alpha^{(m)}$, we have:

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

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

$$z^{(m)} = z - (\sum\alpha^{(i)})$$

After m pseudorotations by the angles $\alpha^{(1)}, \alpha^{(2)}, \ldots, \alpha^{(m)}$:

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

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

$$z^{(m)} = z - (\sum\alpha^{(i)})$$

where $K = \Pi(1 + \tan^2\alpha^{(i)})^{1/2}$

22.2 Basic CORDIC Iterations

Pick $\alpha^{(i)}$ such that $\tan \alpha^{(i)} = d_i 2^{-i}$, $d_i \in \{-1, 1\}$

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

$$y^{(i+1)} = y^{(i)} + d_i x^{(i)} 2^{-i} \qquad \text{[CORDIC iteration]}$$

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

If we always pseudo-rotate by the same set of angles (with + or – signs), then the expansion factor K is a constant that can be precomputed.

$$30.0 \approx 45.0 - 26.6 + 14.0 - 7.1 + 3.6 + 1.8 - 0.9 + 0.4 - 0.2 + 0.1$$
$$= 30.1$$

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

i	$e^{(i)}$
0	45.0
1	26.6
2	14.0
3	7.1
4	3.6
5	1.8
6	0.9
7	0.4
8	0.2
9	0.1

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

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

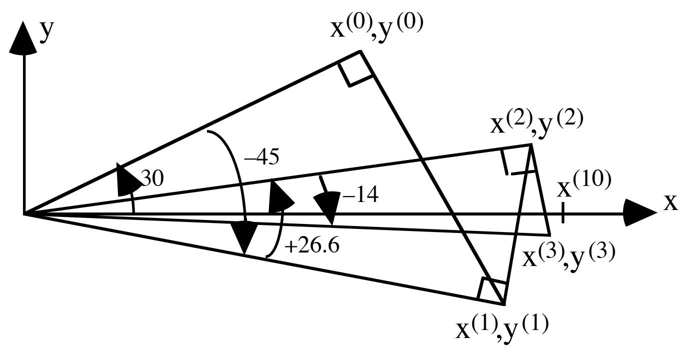


Fig. 22.2 The first 3 of 10 pseudo-rotations leading from $(x^{(0)}, y^{(0)})$ to $(x^{(10)}, 0)$ in rotating by +30°.

CORDIC Rotation Mode

In CORDIC terminology, the preceding selection rule for d_i, which makes z converge to 0, is known as "rotation mode".

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

After m iterations in rotation mode, when $z^{(m)} \approx 0$, we have $\sum\alpha^{(i)} = z$, and the CORDIC equations [*] become:

$$x^{(m)} = K(x \cos z - y \sin z)$$
$$y^{(m)} = K(y \cos z + x \sin z) \qquad \text{[Rotation mode]}$$
$$z^{(m)} = 0$$

Rule: Choose $d_i \in \{-1, 1\}$ such that $z \to 0$

The constant K is K = 1.646 760 258 121 ···

Start with x = 1/K = 0.607 252 935 ··· and y = 0;
as $z^{(m)}$ tends to 0 with CORDIC iterations in rotation mode, $x^{(m)}$ and $y^{(m)}$ converge to cos z and sin z

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

In rotation mode, convergence of z to 0 is possible since each angle in Table 22.1 is more than half of the previous angle or, equivalently, each angle is less than the sum of all the angles following it

The domain of convergence is $-99.7° \le z \le 99.7°$, where 99.7° is the sum of all the angles in Table 14.1

Fortunately, this range includes angles from –90° to +90°, or $[-\pi/2, \pi/2]$ in radians, which is adequate

CORDIC Vectoring Mode

Let us now make y tend to 0 by choosing $d_i = -\mathrm{sign}(x^{(i)}y^{(i)})$

After m steps in "vectoring mode" we have $\tan(\sum\alpha^{(i)}) = -y/x$

$$\begin{aligned} x^{(m)} &= K(x\cos(\textstyle\sum\alpha^{(i)}) - y\sin(\textstyle\sum\alpha^{(i)})) \\ &= K(x - y\tan(\textstyle\sum\alpha^{(i)}))/(1 + \tan^2(\textstyle\sum\alpha^{(i)}))^{1/2} \\ &= K(x + y^2/x)/(1 + y^2/x^2)^{1/2} \\ &= K(x^2 + y^2)^{1/2} \end{aligned}$$

The CORDIC equations [*] thus become:

$$\begin{aligned} x^{(m)} &= K(x^2 + y^2)^{1/2} \\ y^{(m)} &= 0 \qquad\qquad \text{[Vectoring mode]} \\ z^{(m)} &= z + \tan^{-1}(y/x) \end{aligned}$$

Rule: Choose $d_i \in \{-1, 1\}$ such that $y \to 0$

Compute $\tan^{-1}y$ by starting with x = 1 and z = 0

This computation always converges. However, one can take advantage of

$$\tan^{-1}(1/y) = \pi/2 - \tan^{-1}y$$

to limit the range of fixed-point numbers encountered

Computation of $\sin^{-1}$ and $\cos^{-1}$: later

22.3 CORDIC Hardware

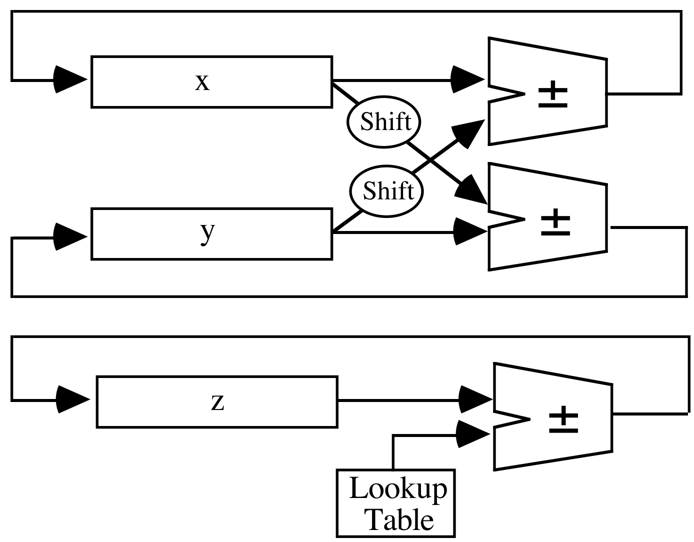


Fig. 22.3 Hardware elements needed for the CORDIC method.

22.4 Generalized CORDIC

The basic CORDIC method of Section 22.2 can be generalized to provide a more powerful tool for function evaluation. Generalized CORDIC is defined as follows:

$$x^{(i+1)} = x^{(i)} - \mu d_i\, y^{(i)} 2^{-i}$$
$$y^{(i+1)} = y^{(i)} + d_i\, x^{(i)} 2^{-i} \qquad \text{[Generalized CORDIC iteration]}$$
$$z^{(i+1)} = z^{(i)} - d_i\, e^{(i)}$$

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

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

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

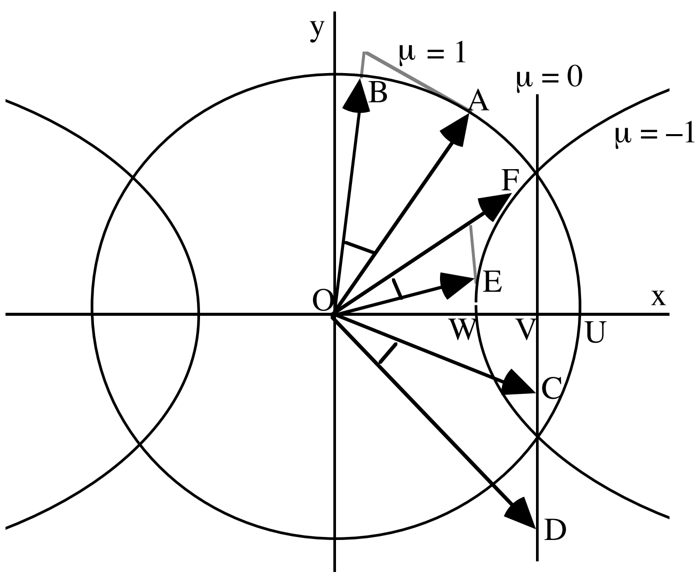


Fig. 24.4 Circular, linear, and hyperbolic CORDIC.

22.5 Using the CORDIC Method

	Rotation mode: $d_i = \text{sign}(z^{(i)})$, $z^{(i)} \to 0$	Vectoring mode: $d_i = -\text{sign}(x^{(i)} y^{(i)})$, $y^{(i)} \to 0$
$\mu = 1$ Circular $e^{(i)} = \tan^{-1} 2^{-i}$	CORDIC: $x \to K(x \cos z - y \sin z)$ $y \to K(y \cos z + x \sin z)$ $z \to 0$ For cos & sin, set $x = 1/K$, $y = 0$ $\tan z = \sin z / \cos z$	CORDIC: $x \to K\sqrt{x^2 + y^2}$ $y \to 0$ $z \to z + \tan^{-1}(y/x)$ 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 $e^{(i)} = 2^{-i}$	CORDIC: $x \to x$ $y \to y + xz$ $z \to 0$ For multiplication, set $y = 0$	CORDIC: $x \to x$ $y \to 0$ $z \to z + y/x$ For division, set $z = 0$
$\mu = -1$ Hyperbolic $e^{(i)} = \tanh^{-1} 2^{-i}$	CORDIC: $x \to K'(x \cosh z - y \sinh z)$ $y \to K'(y \cosh z + x \sinh z)$ $z \to 0$ For cosh & sinh, set $x = 1/K'$, $y = 0$ $\tanh z = \sinh z / \cosh z$ $\exp(z) = \sinh z + \cosh z$ $w^t = \exp(t \ln w)$	CORDIC: $x \to K'\sqrt{x^2 - y^2}$ $y \to 0$ $z \to z + \tanh^{-1}(y/x)$ For $\tanh^{-1}$, set $x = 1$, $z = 0$ $\ln w = 2 \tanh^{-1} \lvert (w - 1)/(w + 1) \rvert$ $\sqrt{w} = \sqrt{(w + 1/4)^2 - (w - 1/4)^2}$ $\cosh^{-1} w = \ln(w + \sqrt{1 - w^2})$ $\sinh^{-1} w = \ln(w + \sqrt{1 + w^2})$
	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.	

$$
\begin{aligned}
x^{(i+1)} &= x^{(i)} - \mu\, d_i\, (2^{-i} y^{(i)}) & \mu &\in \{-1, 0, 1\},\ d_i \in \{-1, 1\} \\
y^{(i+1)} &= y^{(i)} + d_i\, (2^{-i} x^{(i)}) & K &= 1.646\,760\,258\,121 \cdots \\
z^{(i+1)} &= z^{(i)} - d_i\, e^{(i)} & K' &= 0.828\,159\,360\,960\,2 \cdots
\end{aligned}
$$

Fig. 22.5 Summary of generalized CORDIC algorithms.

Speedup Methods for CORDIC

Skipping some rotations

Must keep track of the expansion factor via the recurrence:

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

Given the additional work, variable-factor CORDIC is often not cost-effective compared to constant-factor CORDIC

Early termination

A possible speed-up method in constant-factor CORDIC is to do k/2 iterations as usual and then combine the remaining k/2 iterations into a single step, involving multiplication, by using:

$$\begin{aligned} x^{(k/2+1)} &= x^{(k/2)} - y^{(k/2)}z^{(k/2)} \\ y^{(k/2+1)} &= y^{(k/2)} + x^{(k/2)}z^{(k/2)} \\ z^{(k/2+1)} &= z^{(k/2)} - z^{(k/2)} = 0 \end{aligned}$$

This is possible because for very small values of z, we have $\tan^{-1}z \approx z \approx \tan z$. The expansion factor K presents no problem since for $e^{(i)} < 2^{-k/2}$, the contribution of the ignored terms that would have been multiplied by K is provably less than ulp

High-radix CORDIC

In a radix-4 CORDIC, d_i assumes values in {–2, –1, 1, 2} (perhaps with 0 also included) rather than in {–1, 1}.

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

22.6 An Algebraic Formulation

Because

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

$\cos z$ 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 methods (convergence and otherwise) for some of the same functions computable through CORDIC

Chapter Highlights

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

Chapter Contents

23.1 Additive/Multiplicative Normalization
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 Additive/Multiplicative Normalization

Convergence methods characterized by recurrences

$$u^{(i+1)} = f(u^{(i)}, v^{(i)}) \qquad u^{(i+1)} = f(u^{(i)}, v^{(i)}, w^{(i)})$$
$$v^{(i+1)} = g(u^{(i)}, v^{(i)}) \qquad v^{(i+1)} = g(u^{(i)}, v^{(i)}, w^{(i)})$$
$$w^{(i+1)} = h(u^{(i)}, v^{(i)}, w^{(i)})$$

Making u converge to a constant = "normalization"

Additive normalization = u normalized by adding values to it

Multiplicative normalization = u multiplied by values

Availability of cost-effective fast adders and multipliers
 make these two types of convergence methods useful

Multipliers are slower and more costly than adders,
 so we avoid multiplicative normalization
 when additive normalization would do

Multiplicative methods often offer faster convergence,
 thus making up for the slower steps

Multiplicative terms of 1 ± 2^a are desirable (shift-add)

Examples we have seen before:
 Additive normalization: CORDIC
 Multiplicative normalization: convergence division

23.2 Computing Logarithms

A multiplicative normalization method using shift-add

$$x^{(i+1)} = x^{(i)}c^{(i)} = x^{(i)}(1 + d_i\, 2^{-i}) \qquad d_i \in \{-1, 0, 1\}$$
$$y^{(i+1)} = y^{(i)} - \ln c^{(i)} = y^{(i)} - \ln(1 + d_i\, 2^{-i})$$

where $\ln(1 + d_i\, 2^{-i})$ is read out from a table

Begin with $x^{(0)} = x$, $y^{(0)} = y$; Choose d_i such that $x^{(m)} \to 1$

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

To find $\ln x$, start with $y = 0$

The algorithm's domain of convergence is easily obtained:

$$1/\prod(1 + 2^{-i}) \le x \le 1/\prod(1 - 2^{-i}) \quad \text{or} \quad 0.21 \le x \le 3.45$$

For large i, we have $\ln(1 \pm 2^{-i}) \approx \pm 2^{-i}$

So, we need k iterations to find ln x with k bits of precision

This method can be used directly for x in [1, 2)

Any x outside [1, 2) can be written as $x = 2^q s$, with $1 \le s < 2$

$$\ln x = \ln(2^q s) = q \ln 2 + \ln s = 0.693\,147\,180\, q + \ln s$$

A radix-4 version of this algorithm can be easily developed

A clever method based on squaring

Let $y = \log_2 x$ be a fractional number $(.y_{-1}y_{-2} \cdot\cdot\cdot y_{-l})_{two}$

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

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

If $y_{-1} = 0$, we are back to the original situation

If $y_{-1} = 1$, we divide both sides of the equation by 2 to get

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

Subsequent bits of y can be determined in a similar way.

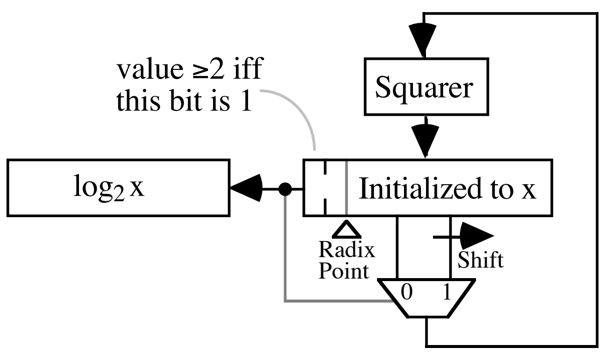


Fig. 23.1 Hardware elements needed for computing $\log_2 x$.

Generalization to base-b logarithms: $y = \log_b x$ implies

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

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

23.3 Exponentiation

An additive normalization method for computing e^x

$$x^{(i+1)} = x^{(i)} - \ln c^{(i)} = x^{(i)} - \ln(1 + d_i 2^{-i})$$
$$y^{(i+1)} = y^{(i)} c^{(i)} = y^{(i)}(1 + d_i 2^{-i}) \qquad d_i \in \{-1, 0, 1\}$$

As before, $\ln(1 + d_i 2^{-i})$ is read out from a table
Begin with $x^{(0)} = x$, $y^{(0)} = y$; Choose d_i such that $x^{(m)} \to 0$

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

$$y^{(m)} = y \prod c^{(i)} = y\, e^{\ln \Pi c^{(i)}} = y\, e^{\Sigma \ln c^{(i)}} \approx y\, e^x$$

The algorithm's domain of convergence is easily obtained:

$$\sum \ln(1 - 2^{-i}) \le x \le \sum \ln(1 + 2^{-i}) \quad \text{or} \quad -1.24 \le x \le 1.56$$

Half of the required k iterations can be eliminated by noting:

$$\ln(1 + \varepsilon) = \varepsilon - \varepsilon^2/2 + \varepsilon^3/3 - \cdots \approx \varepsilon \quad \text{for } \varepsilon^2 < \text{ulp}$$

So when $x^{(j)} = 0.00 \cdots 00xx \cdots xx$, with k/2 leading zeros, we have $\ln(1 + x^{(j)}) \approx x^{(j)}$, allowing us to terminate by

$$x^{(j+1)} = x^{(j)} - x^{(j)} = 0$$
$$y^{(j+1)} = y^{(j)}(1 + x^{(j)})$$

A radix-4 version of this e^x algorithm can be developed

The general exponentiation function x^y

can be computed by combining the logarithm and exponential functions and a single multiplication:

$$x^y = (e^{\ln x})^y = e^{y \ln x}$$

When y is a positive integer, exponentiation can be done by repeated multiplication

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

Example:

$$x^{25} = ((((x)^2x)^2)^2)^2x$$

which implies 4 squarings and 2 multiplications. Noting that

$$25 = (1\ 1\ 0\ 0\ 1)_{two}$$

leads us to a general procedure

To raise x to the power y, where y is a positive integer:

Initialize the partial result to 1

Scan the binary representation of y starting at its MSB

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)

23.5 Use of Approximating Functions

Convert the problem of evaluating the function f to that of evaluating a different function g that approximates f, perhaps with a few pre- and post-processing operations

Approximating polynomials attractive because they need only additions and multiplications

Polynomial approximations can be obtained based on various schemes; e.g., Taylor-Maclaurin series expansion

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

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

The error due to omitting terms of degree greater than m is:

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

Setting a = 0 yields the Maclaurin-series expansion

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

and its corresponding error bound:

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

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

Table 23.1 Polynomial approximations for some useful functions

Function	Polynomial approximation	Conditions
$1/x$	$1 + y + y^2 + y^3 + \cdots + y^i + \cdots$	$0 < x < 2$ and $y = 1-x$
$\sqrt{x}$	$1 - \frac{1}{2}y - \frac{1}{2\times4}y^2 - \frac{1\times3}{2\times4\times6}y^3 - \cdots - \frac{1\times3\times5\times\ldots\times(2i-3)}{2\times4\times6\times\ldots\times2i}y^i - \cdots$	$y = 1-x$
e^x	$1 + \frac{1}{1!}x + \frac{1}{2!}x^2 + \frac{1}{3!}x^3 + \cdots + \frac{1}{i!}x^i + \cdots$	
$\ln x$	$-y - \frac{1}{2}y^2 - \frac{1}{3}y^3 - \frac{1}{4}y^4 - \cdots - \frac{1}{i}y^i - \cdots$	$0 < x \le 2$ and $y = 1 - x$
$\ln x$	$2[z + \frac{1}{3}z^3 + \frac{1}{5}z^5 + \cdots + \frac{1}{2i+1}z^{2i+1} + \cdots]$	$x > 0$ and $z = \frac{x-1}{x+1}$
$\sin x$	$x - \frac{1}{3!}x^3 + \frac{1}{5!}x^5 - \frac{1}{7!}x^7 + \cdots + (-1)^i \frac{1}{(2i+1)!}x^{2i+1} + \cdots$	
$\cos x$	$1 - \frac{1}{2!}x^2 + \frac{1}{4!}x^4 - \frac{1}{6!}x^6 + \cdots + (-1)^i \frac{1}{(2i)!}x^{2i} + \cdots$	
$\tan^{-1}x$	$x - \frac{1}{3}x^3 + \frac{1}{5}x^5 - \frac{1}{7}x^7 + \cdots + (-1)^i \frac{1}{2i+1}x^{2i+1} + \cdots$	$-1 < x < 1$
$\sinh x$	$x + \frac{1}{3!}x^3 + \frac{1}{5!}x^5 + \frac{1}{7!}x^7 + \cdots + \frac{1}{(2i+1)!}x^{2i+1} + \cdots$	
$\cosh x$	$1 + \frac{1}{2!}x^2 + \frac{1}{4!}x^4 + \frac{1}{6!}x^6 + \cdots + \frac{1}{(2i)!}x^{2i} + \cdots$	
$\tanh^{-1}x$	$x + \frac{1}{3}x^3 + \frac{1}{5}x^5 + \frac{1}{7}x^7 + \cdots + \frac{1}{2i+1}x^{2i+1} + \cdots$	$-1 < x < 1$

A divide-and-conquer strategy for function evaluation

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

$$x = x_H + 2^{-t} x_L \qquad \underset{t+2 \text{ bits}}{0 \le x_H < 4} \qquad \underset{l-t \text{ bits}}{0 \le x_L < 1}$$

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

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

where $f^{(j)}(x)$ is the jth derivative of f(x). If one takes just the first two terms, a linear approximation is obtained

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

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

Approximation by the ratio of two polynomials

Example, yielding good results for many elementary functions:

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

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

23.6 Merged Arithmetic

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

For ultrahigh performance, one can build hardware structures to compute the function of interest directly without breaking it down into conventional operations

Example: merged arithmetic for inner product computation

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

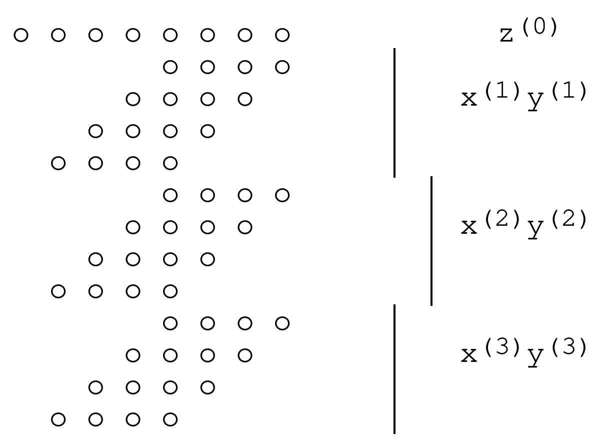


Fig. 23.2 Merged-arithmetic computation of an inner product followed by accumulation.

	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

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

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)

Chapter Contents

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. Tradeoffs in Cost, Speed, and Accuracy
24.6. Piecewise Lookup Tables

24.1 Direct and Indirect Table Lookup

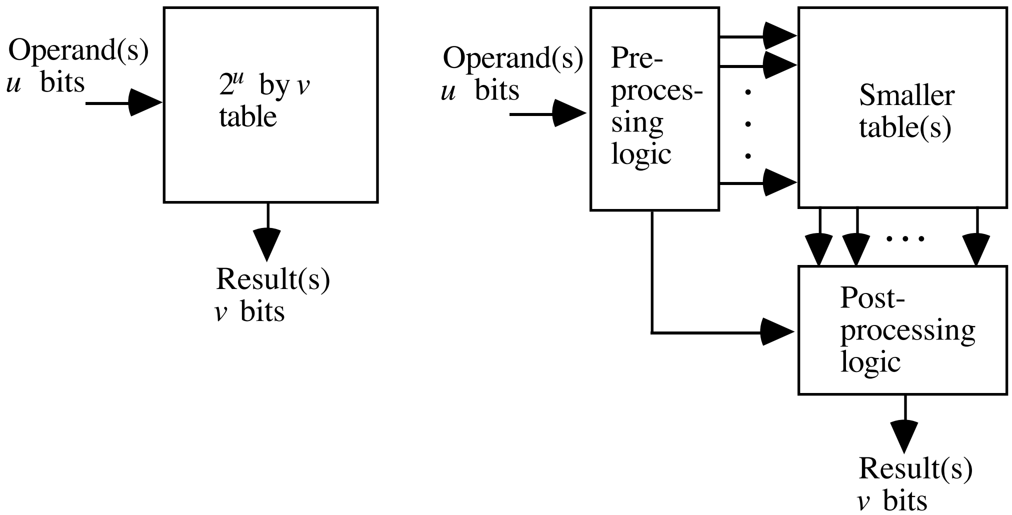


Fig. 24.1 Direct table lookup versus table-lookup with pre- and post-processing.

The boundary between the two uses of tables (in supporting or primary role) is quite fuzzy

The pure logic and pure tabular approaches are extreme points in a continuum of hybrid solutions

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 intermediate solutions can be derived starting at either endpoint

24.2 Binary-to-Unary Reduction

An approach to reducing the table size is using an auxiliary unary function in order to evaluate a desired binary function

Example 1: Addition in a logarithmic number system

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

Example 2: Multiplication via squaring

$$xy = [(x + y)^2/4 - (x - y)^2/4]$$

Simplification

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

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

Upon computing $x + y$ and $x - y$ in the preprocessing stage, we can drop the least significant bit of each result and then consult squaring tables of size $2^k \times (2k - 1)$

Post-processing requires a carry-save adder (to reduce the 3 values to 2) followed by a carry-propagate adder

24.3 Tables in Bit-Serial Arithmetic

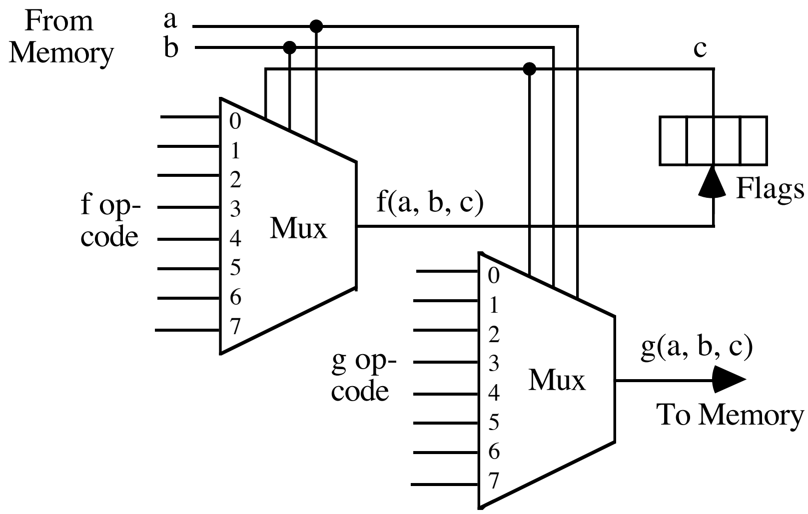


Fig. 24.2 Bit-serial ALU with 2 tables implemented as multiplexers.

In Fig. 24.2, a and b come from a 64K-bit memory (16-bit addresses) and c from a 4-bit "flags" register (2-bit address)

The f output is stored as a flag bit (2-bit address) and the g output replaces the a operand in a third clock cycle

Three additional bits are used to specify a flag bit and a value (0 or 1) for conditionalizing the operation (to allow data-dependent ALU operations)

To perform integer addition with the CM-2 ALU

a and b: numbers to be added

c: flag bit holding the carry from one position into next

f op code: "00010111" (majority or ab + bc + ca)

g op-code: "01010101" (3-input XOR)

Second-order digital filter example

$$y^{(i)} = a^{(0)}x^{(i)} + a^{(1)}x^{(i-1)} + a^{(2)}x^{(i-2)} - b^{(1)}y^{(i-1)} - b^{(2)}y^{(i-2)}$$

Expand the equation for $y^{(i)}$ in terms of the bits in the operands $x = (x_0.x_{-1}x_{-2}\cdots x_{-l})_{\text{two}}$ and $y = (y_0.y_{-1}y_{-2}\cdots y_{-l})_{\text{two}}$

$$\begin{aligned} y^{(i)} = {} & a^{(0)}(-x_0^{(i)} + \textstyle\sum_{j=-l}^{-1} 2^j x_j^{(i)}) \\ & + a^{(1)}(-x_0^{(i-1)} + \textstyle\sum_{j=-l}^{-1} 2^j x_j^{(i-1)}) + a^{(2)}(-x_0^{(i-2)} + \textstyle\sum_{j=-l}^{-1} 2^j x_j^{(i-2)}) \\ & - b^{(1)}(-y_0^{(i-1)} + \textstyle\sum_{j=-l}^{-1} 2^j x_j^{(i-1)}) - b^{(2)}(-y_0^{(i-2)} + \textstyle\sum_{j=-l}^{-1} 2^j x_j^{(i-2)}) \end{aligned}$$

Define $f(s, t, u, v, w) = a^{(0)}s + a^{(1)}t + a^{(2)}u - b^{(1)}v - b^{(2)}w$

$$\begin{aligned} y^{(i)} = {} & \textstyle\sum_{j=-l}^{-1} 2^j f(x_j^{(i)}, x_j^{(i-1)}, x_j^{(i-2)}, y_j^{(i-1)}, y_j^{(i-2)}) \\ & - f(x_0^{(i)}, x_0^{(i-1)}, x_0^{(i-2)}, y_0^{(i-1)}, y_0^{(i-2)}) \end{aligned}$$

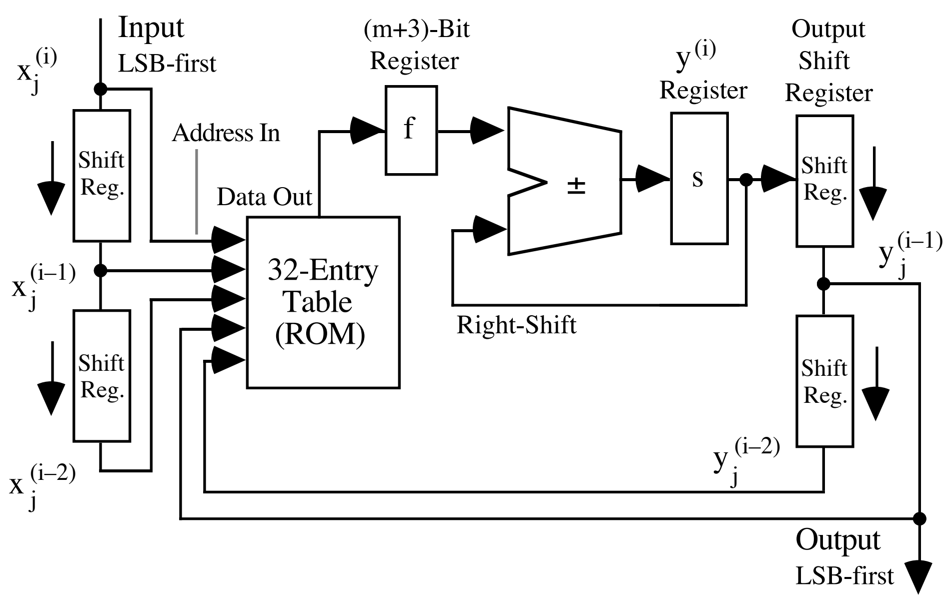


Fig. 24.3 Bit-serial tabular realization of a second-order filter.

24.4 Interpolating Memory

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

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

By choosing the end points to be consecutive multiples of a power of 2, the division and two of the additions can be reduced to trivial operations

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:

$$\log_2 x \approx x - 1 = \text{the fractional part of } x$$

An improvemed linear interpolation formula

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

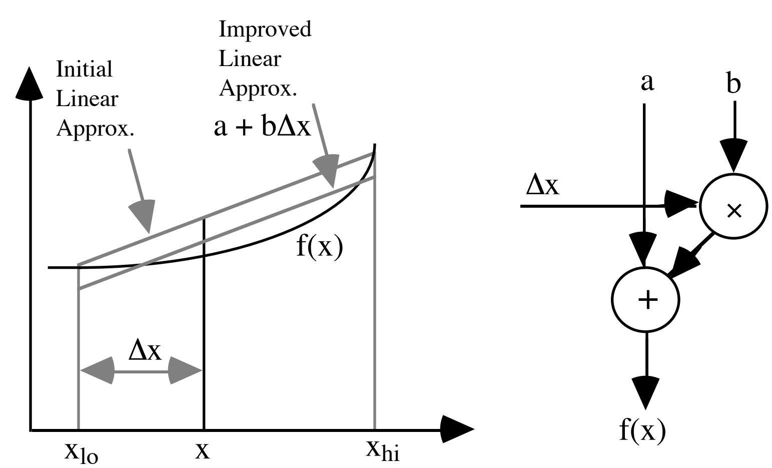


Fig. 24.4 Linear interpolation for computing f(x) and its hardware realization.

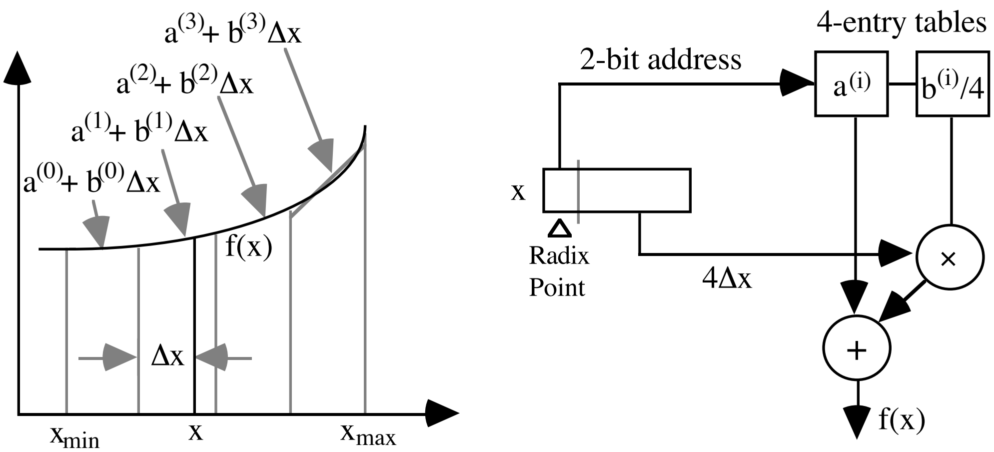


Fig. 24.5 Linear interpolation for computing f(x) using 4 subintervals.

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

i	x_{lo}	x_{hi}	$a^{(j)}$	$b^{(j)}/4$	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

24.5 Trade-offs in Cost, Speed, and Accuracy

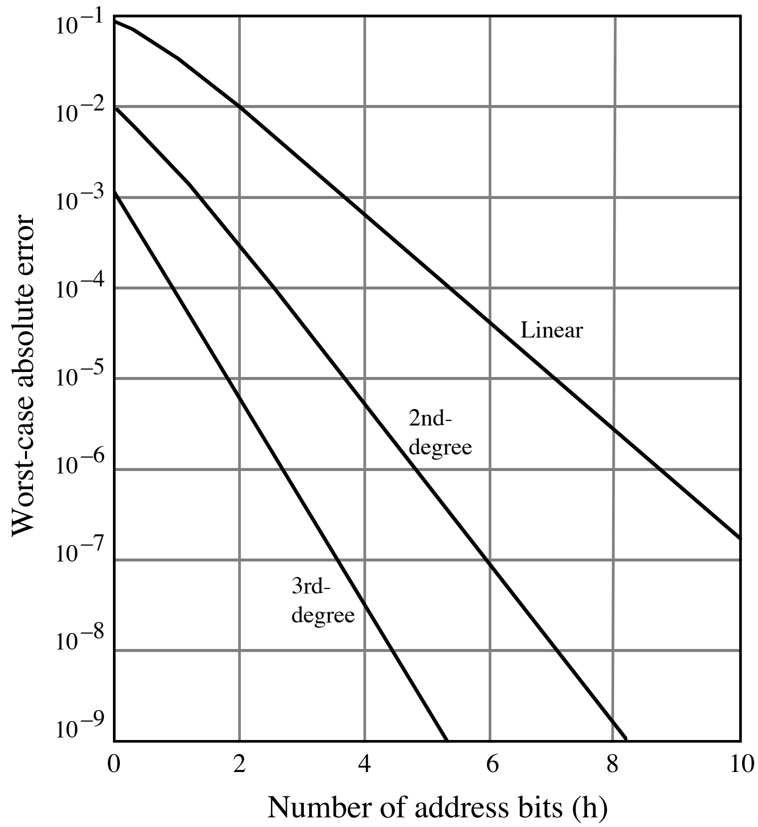


Fig. 24.6 Maximum absolute error in computing $\log_2 x$ as a function of number h of address bits for the tables with linear, quadratic (second-degree), and cubic (third-degree) interpolations [Noet89].

24.6 Piecewise Lookup Tables

Function of a short (single) IEEE floating-point number

Divide the 26-bit significand x (with 2 whole and 24 fractional bits) into four sections:

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

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

Taylor polynomial for f(x):

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

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

$$f(x) \approx f(t + \lambda u) + \frac{\lambda}{2}[f(t + \lambda u + \lambda v) - f(t + \lambda u - \lambda v)]$$

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

With this method, computing f(x) reduces to:

a. Deriving the 14-bit values $t + \lambda u + \lambda v$, $t + \lambda u - \lambda v$, $t + \lambda u + \lambda w$, $t + \lambda u - \lambda w$ using 4 additions ($t + \lambda u$ needs no computation)

b. Reading the five values of f from table(s)

c. Reading the last term $\lambda^4[\frac{v^2}{2} f^{(2)}(t) - \frac{v^3}{6} f^{(3)}(t)]$ from a table

d. Performing a 6-operand addition

Analytical evaluation has shown that the error in the preceding computation is provably less than ulp/2 = 2^{-24}

Computing z mod p (modular reduction)

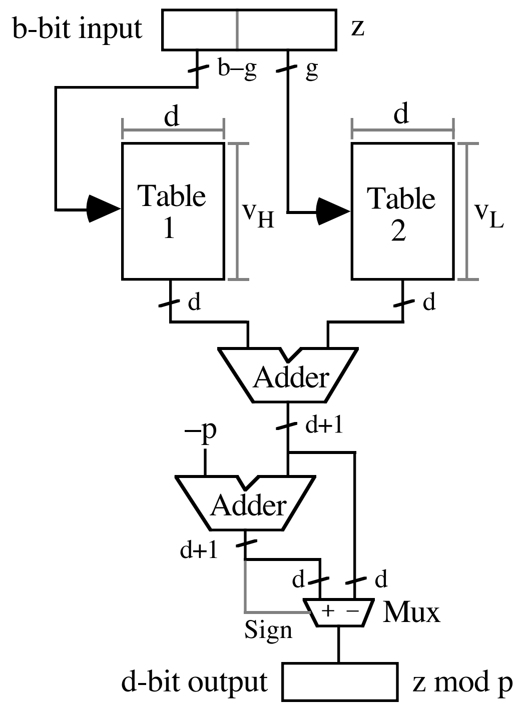


Fig. 24.7 Two-table modular reduction scheme based on divide-and-conquer.

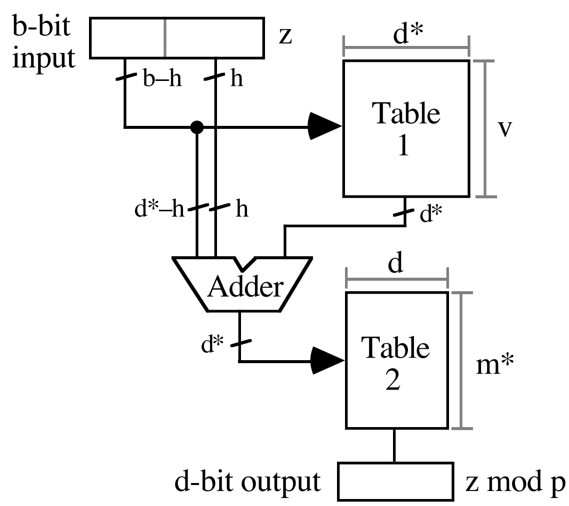


Fig. 24.8 Two-table modular reduction based on successive refinement.

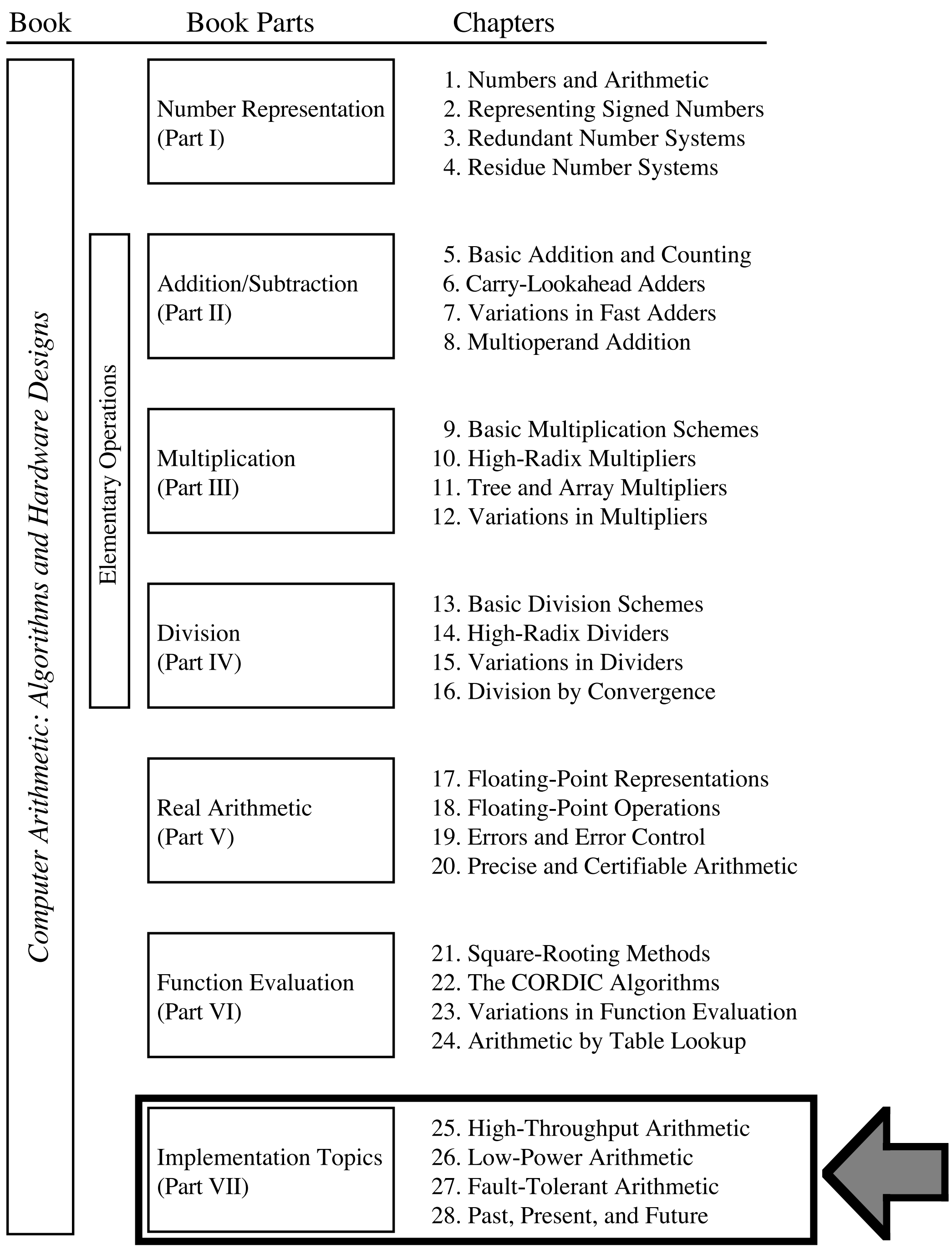


Fig. P.1 The structure of this book in parts and chapters.

Part VII Implementation Topics

Part Goals

Sample more advanced implementation methods and ponder some of the practical aspects of computer arithmetic

Part Synopsis

Speed/latency is often not the only concern

Other attributes of interest include throughput, size, power, reliability

Case studies: arithmetic in micros to supers

Lessons from the past, future outlook

Part Contents

Chapter 25 High-Throughput Arithmetic

Chapter 26 Low-Power Arithmetic

Chapter 27 Fault-Tolerant Arithmetic

Chapter 28 Past, Present, and Future

25 High-Throughput Arithmetic

Chapter Goals

Learn how to improve the performance of an arithmetic unit via higher throughput rather than reduced latency

Chapter Highlights

To improve overall performance, one has to

- look beyond individual operations
- trade off latency for throughput

E.g., a multiply may take 20 clock cycles, but a new one can begin every cycle

Data availability and hazards limit the depth

Chapter Contents

25.1. Pipelining of Arithmetic Functions
25.2. Clock Rate and Throughput
25.3. The Earle Latch
25.4. Parallel and Digit-Serial Pipelines
25.5. On-Line or Digit-Pipelined Arithmetic
25.6. Systolic Arithmetic Units

25.1 Pipelining of Arithmetic Functions

Throughput = number of operations per unit time

Pipelining period = time interval between the application of successive input data for proper overlapped computation

Latency, though secondary, is still important because:

a. There is occasional need for doing single operations not immediately followed by others of the same type

b. Dependencies or conditional operations (hazards) may force us to insert bubbles into or drain the pipeline

At times, a pipelined implementation may improve the latency of a multi-step arithmetic computation while also reducing its hardware cost

In such a case, pipelining is obviously the preferred method

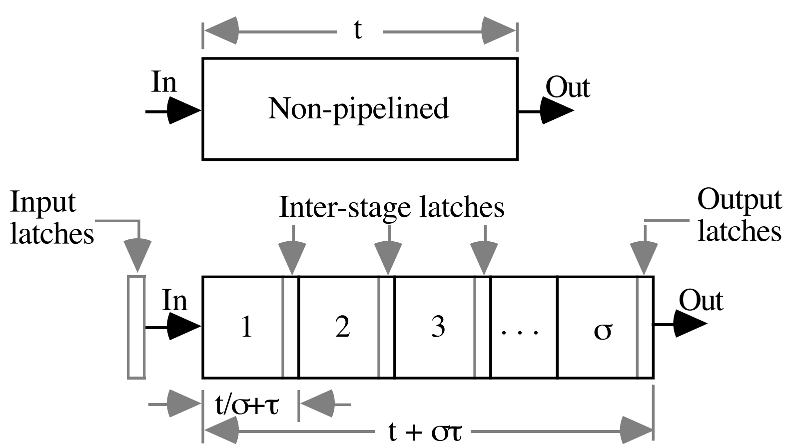


Fig. 26.1 An arithmetic function unit and its σ-stage pipelined version.

Analysis of pipelining

Consider an arithmetic circuit with cost g and latency t.

Simplifying assumptions for our analysis:

1. Pipelining time overhead per stage is τ (latching delay)
2. Pipelining cost overhead per stage is γ (latching cost)
3. The function is divisible into σ equal stages for any σ

Then, for the pipelined implementation:

Latency	T	=	$t + \sigma\tau$
Throughput	R	=	$\frac{1}{T/\sigma} = \frac{1}{t/\sigma + \tau}$
Cost	G	=	$g + \sigma\gamma$

Throughput approaches its maximum of $1/\tau$ for very large σ

In practice, however, it does not pay to reduce t/σ below a certain threshold; typically 4 logic gate levels

Assuming a stage delay of 4δ, we have $\sigma = t/(4\delta)$ and:

Latency	T	=	$t(1 + \frac{\tau}{4\delta})$
Throughput	R	=	$\frac{1}{T/\sigma} = \frac{1}{4\delta + \tau}$
Cost	G	=	$g(1 + \frac{t\gamma}{4g\delta})$

Cost-effectiveness

If throughput isn't the single most important factor,

then one might try to maximize a composite figure of merit

Throughput per unit cost may represent cost-effectiveness:

$$E = \frac{R}{G} = \frac{\sigma}{(t + \sigma\tau)(g + \sigma\gamma)}$$

To maximize E, we compute $dE/d\sigma$:

$$\frac{dE}{d\sigma} = \frac{tg - \sigma^2\tau\gamma}{(t + \sigma\tau)^2(g + \sigma\gamma)^2}$$

Equating $dE/d\sigma$ with 0 yields:

$$\sigma^{opt} = \sqrt{\frac{t\,g}{\tau\,\gamma}}$$

We see that the optimal number of pipeline stages for maximal cost-effectiveness is

directly related to the latency and cost of the function

(it pays to have many pipeline stages if the function implemented is very slow or complex)

inversely related to pipelining delay & cost overheads

(few pipeline stages are in order if the time and/or cost overhead of pipelining is too high)

All in all, not a surprising result!

25.2 Clock Rate and Throughput

Consider a σ-stage pipeline with stage delay t_{stage}

One set of inputs are applied to the pipeline at time t_1

At $t_1 + t_{stage} + \tau$, results of this set are safely stored in latches

Apply the next set of inputs at time t_2 satisfying

$$t_2 \geq t_1 + t_{stage} + \tau$$

to ensure proper pipeline operation. Thus:

$$\text{Clock period} = \Delta t = t_2 - t_1 \geq t_{stage} + \tau$$

Pipeline throughput is the inverse of the clock period:

$$\text{Throughput} = \frac{1}{\text{Clock period}} \leq \frac{1}{t_{stage} + \tau}$$

This analysis assumes that a single clock signal is distributed to all circuit elements and that all latches are clocked at precisely the same time

Uncontrolled or random clock skew causes the clock signal to arrive at point B before or after its arrival at point A

With proper design of the clock distribution network, we can place an upper bound $\pm\varepsilon$ on the uncontrolled clock skew at the input and output latches of a pipeline stage

Then, the clock period is lower bounded as:

$$\text{clock period} = \Delta t = t_2 - t_1 \geq t_{stage} + \tau + 2\varepsilon$$

Wave Pipelining

Note that the stage delay t_{stage} is really not a constant but varies from t_{min} to t_{max}

t_{min} represents fast paths (with fewer or faster gates)

t_{max} represents slow paths

Suppose that one set of inputs is applied at time t_1

At $t_1 + t_{max} + \tau$, results of this set are safely stored in latches

If that the next inputs are applied at time t_2, we must have:

$$t_2 + t_{min} \geq t_1 + t_{max} + \tau$$

This places a lower bound on the clock period:

$$\text{clock period} = \Delta t = t_2 - t_1 \geq t_{max} - t_{min} + \tau$$

Thus, we can approach the maximum possible throughput of $1/\tau$ without necessarily requiring very small stage delay

All we need is a very small delay variance $t_{max} - t_{min}$

Hence, there are two distinct strategies for increasing the throughput of a pipelined function unit:

(1) the traditional method of reducing t_{max}, and

(2) the counter-intuitive method of increasing t_{min}

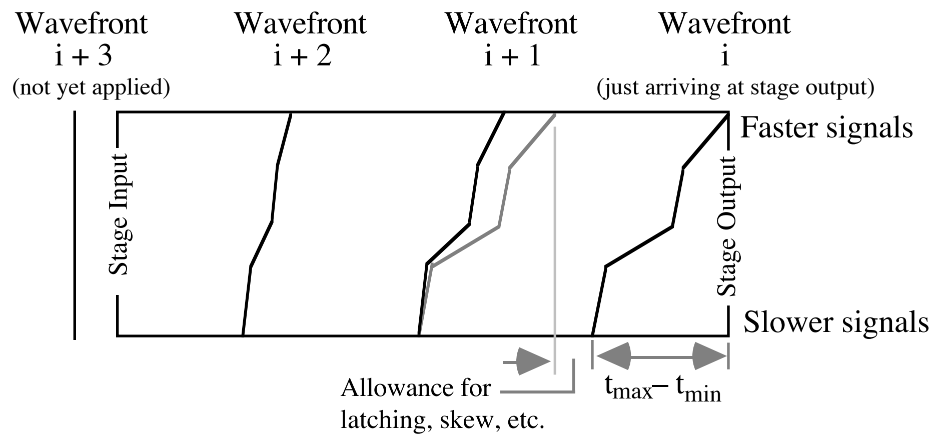


Fig. 26.2 Wave pipelining allows multiple computational wavefronts to coexist in a single pipeline stage.

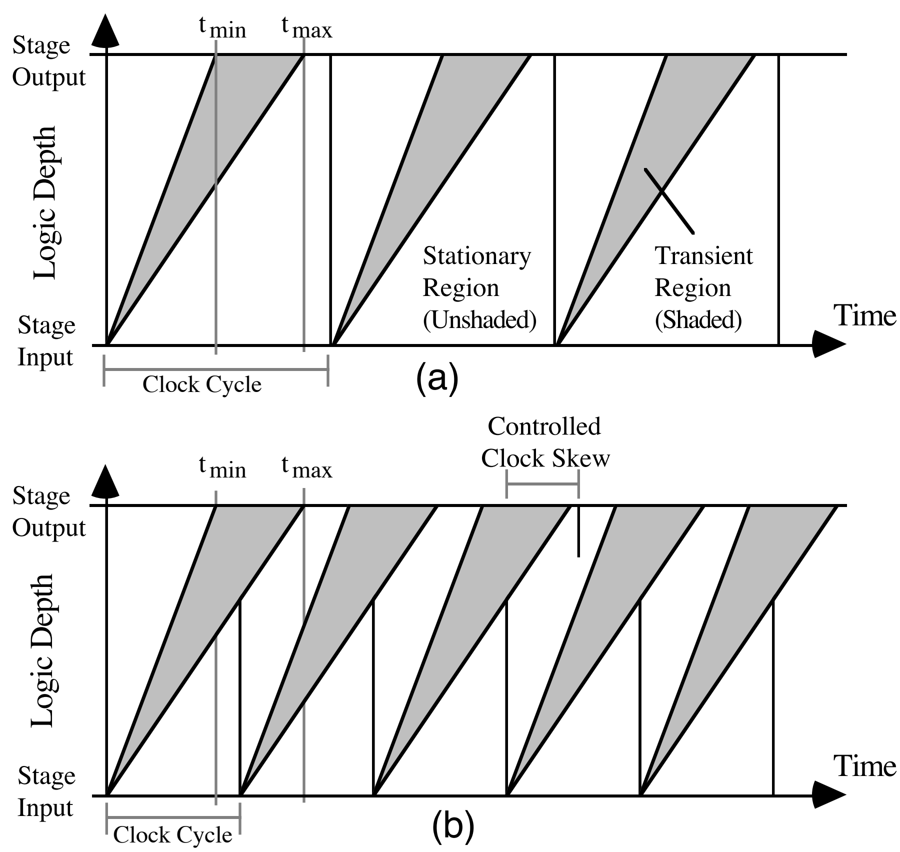


Fig. 26.3 An alternate view of the throughput advantage of wave pipelining (b) over ordinary pipelining (a).

Controlled clock skew

Suppose that $t_{max} - t_{min} = 0$, leading to clock period $\Delta t \geq \tau$.

A new input enters the pipeline stage every Δt time units

and the stage latency is $t_{max} + \tau$

The clock application at the output latch must be skewed by $(t_{max} + \tau) \bmod \Delta t$ to ensure proper sampling of the results

Example: $t_{max} + \tau = 12$ ns and $\Delta t = 5$ ns

A clock skew of +2 ns is required at the stage output latches relative to the input latches

Generally $t_{max} - t_{min} > 0$ and perhaps different for each stage

$$\Delta t \geq \max_{i=1}^{\sigma}[t_{max}^{(i)} - t_{max}^{(i)} + \tau]$$

and the controlled clock skew at the output of stage i will be:

$$S^{(i)} = \sum_{j=1}^{i}[t_{max}^{(i)} - t_{min}^{(i)} + \tau] \bmod \Delta t$$

We still need to worry about uncontrolled or random skew:

$$\text{clock period} = \Delta t = t_2 - t_1 \geq t_{max} - t_{min} + \tau + 4\varepsilon$$

Reason for including the term 4ε :

The clocking of the first input set may lag by ε, while

that of the second set leads by ε (net difference = 2ε)

The reverse condition may exist at the output

Note that uncontrolled clock skew has a larger effect on the performance of wave pipelining than on standard pipelining, especially in relative terms

25.3 The Earle Latch

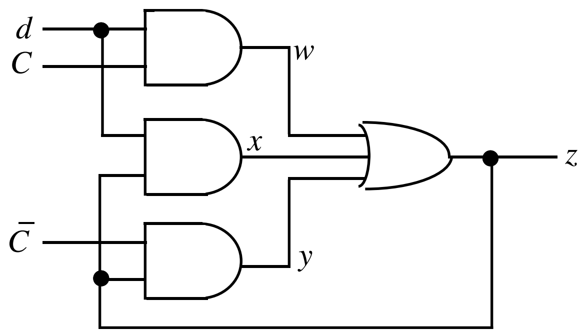


Fig. 25.4 Two-level AND-OR realization of the Earl latch.

Previously, we derived constraints on the maximum clock rate $1/\Delta t$

The clock period Δt has two parts: clock high, and clock low

$$\Delta t \quad = \quad C_{high} + C_{low}$$

Consider a pipeline stage that is preceded and followed by Earle latches.

C_{high}, must satisfy the inequalities

$$3\delta_{max} - \delta_{min} + S_{max}(C\uparrow,\ \bar{C}\downarrow) \ \le\ C_{high} \ \le\ 2\delta_{min} + t_{min}$$

The clock pulse must be wide enough to ensure that valid data is stored in the output latch and to avoid logic hazard should C slightly lead $\bar{C}$

Clock must go low before the fastest signals from the next input data set can affect the input z of the latch

δ_{max} and δ_{min} are maximum and minimum gate delays;

$S_{max}(C\uparrow,\ \bar{C}\downarrow) \ge 0$ is max skew between $C\uparrow$ and $\bar{C}\downarrow$

Merged logic and latch

A key property of the Earle latch is that it can be merged with the 2-level AND-OR logic that precedes it.

Example: to latch

$$d = vw + xy$$

we substitute for d in the equation for the Earle latch

$$z = dC + dz + \bar{C}z$$

to get a "logic+latch" circuit implementing z = vw + xy

$$\begin{aligned} z &= (vw + xy)C + (vw + xy)z + \bar{C}z \\ &= vwC + xyC + vwz + xyz + \bar{C}z \end{aligned}$$

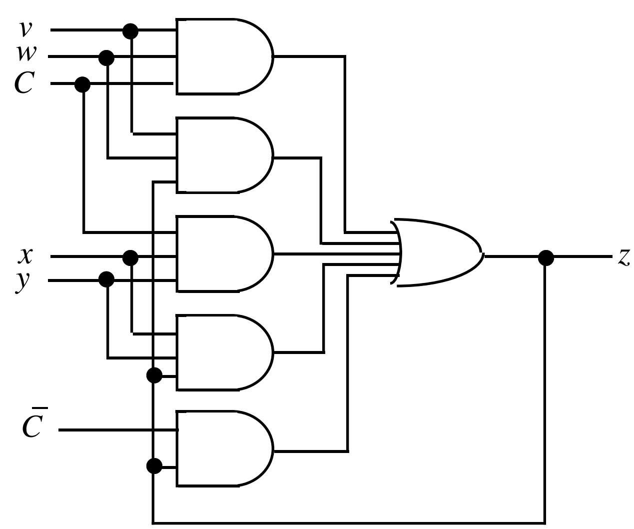


Fig. 25.5 Two-level AND-OR latched realization of the function z = vw + xy.

25.4 Parallel and Digit-Serial Pipelines

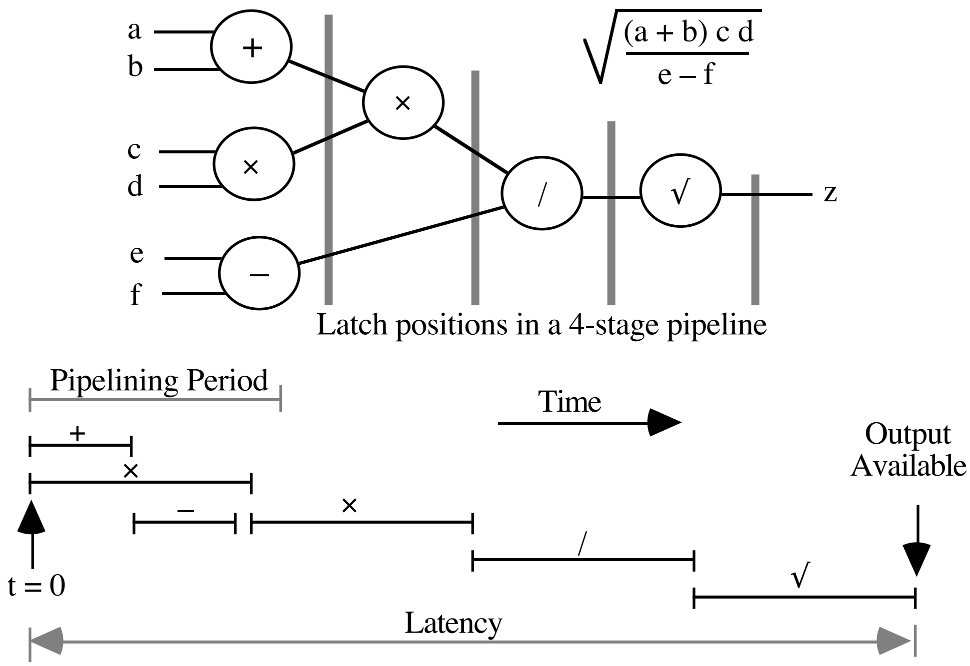


Fig. 25.6 Flow-graph representation of an arithmetic expression and timing diagram for its evaluation with digit-parallel computation.

Bit-serial addition and multiplication can be done LSB-first

But, division and square-rooting are MSB-first operations

Besides, division can't be done in pipelined bit-serial fashion, because the MSB of the quotient q in general depends on all the bits of the dividend and divisor

Example: consider the decimal division .1234/.2469

$$\frac{.1xxx}{.2xxx} = .?xxx \qquad \frac{.12xx}{.24xx} = .?xxx \qquad \frac{.123x}{.246x} = .?xxx$$

Solution: redundant number representation!

25.5 On-Line or Digit-Pipelined Arithmetic

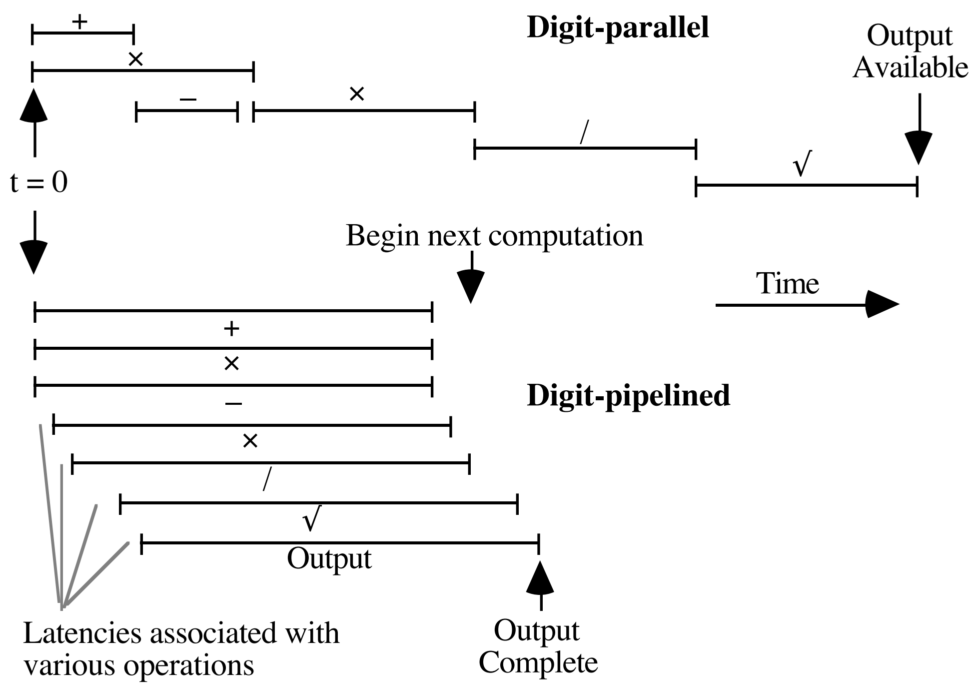


Fig. 25.7 Digit-parallel versus digit-pipelined computation.

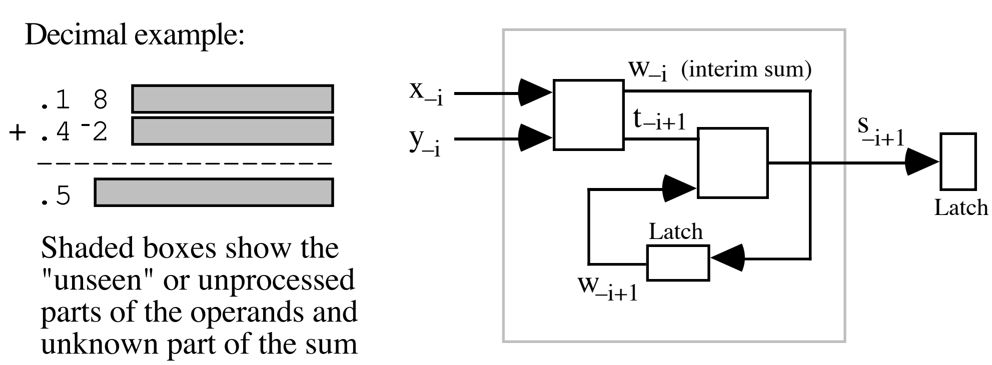


Fig. 25.8 Digit-pipelined MSD-first carry-free addition.

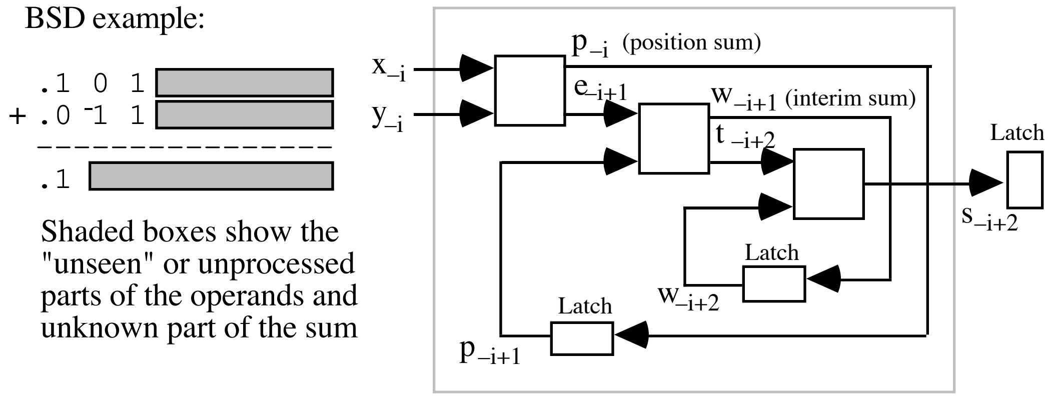


Fig. 25.9 Digit-pipelined MSD-first limited-carry addition.

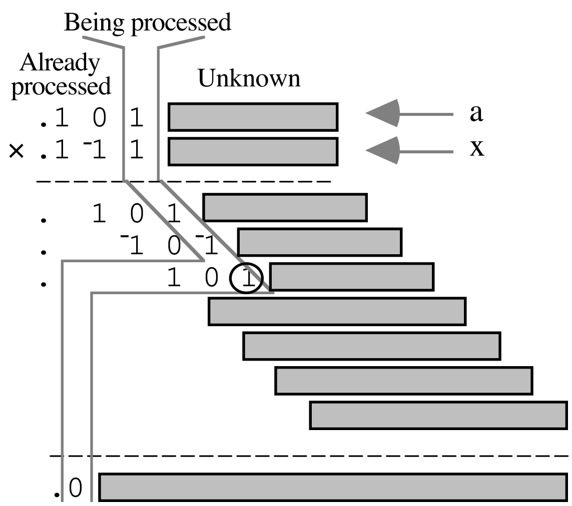


Fig. 25.10 Digit-pipelined MSD-first multiplication process.

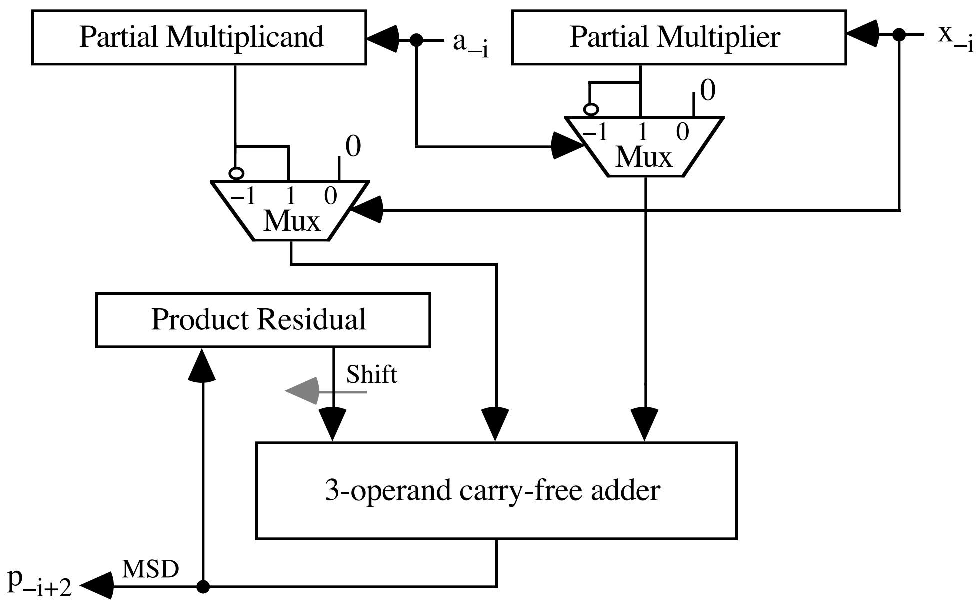


Fig. 25.11 Digit-pipelined MSD-first BSD multiplier.

Table 25.1 Example of digit-pipelined division showing that three cycles of delay are necessary before quotient digits can be output (radix = 4, digit set = [–2, 2])

Cycle	Dividend	Divisor	q Range	q_{-1} Range
1	$(.0 \cdots)_{four}$	$(.1 \cdots)_{four}$	(–2/3, 2/3)	[–2, 2]
2	$(.0\ 0 \cdots)_{four}$	$(.1\,{}^{-}2 \cdots)_{four}$	(–2/4, 2/4)	[–2, 2]
3	$(.0\ 0\ 1 \cdots)_{four}$	$(.1\,{}^{-}2\,{}^{-}2 \cdots)_{four}$	(1/16, 5/16)	[0, 1]
4	$(.0\ 0\ 1\ 0 \cdots)_{four}$	$(.1\,{}^{-}2\,{}^{-}2\,{}^{-}2 \cdots)_{four}$	(10/64, 14/64)	1

Table 25.2 Examples of digit-pipelined square-root computation showing that 1-2 cycles of delay are necessary before root digits can be output (radix = 10, digit set = [–6, 6], and radix = 2, digit set = [–1, 1]).

Cycle	Radicand	q Range	q_{-1} Range
1	$(.3 \cdots)_{ten}$	$(\sqrt{7/30}, \sqrt{11/30})$	[5, 6]
2	$(.3\ 4 \cdots)_{ten}$	$(\sqrt{1/3}, \sqrt{26/75})$	6
1	$(.0 \cdots)_{two}$	$(0, \sqrt{1/2})$	[0, 1]
2	$(.0\ 1 \cdots)_{two}$	$(0, \sqrt{1/2})$	[0, 1]
3	$(.0\ 1\ 1 \cdots)_{two}$	$(1/2, \sqrt{1/2})$	1

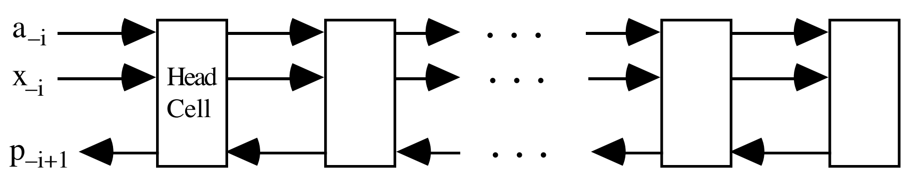


Fig. 25.12 High-level design of a systolic radix-4 digit-pipelined multiplier.

26 Low-Power Arithmetic

Chapter Goals

Learn how to improve the power efficiency of arithmetic circuits by means of algorithmic and logic design strategies

Chapter Highlights

Reduced power dissipation needed due to

- limited source (portable, embedded)
- difficulty of heat disposal

Algorithm & logic-level methods: discussed

Technology & circuit methods: ignored here

Chapter Contents

26.1. The Need for Low-Power Design
26.2. Sources of Power Consumption
26.3. Reduction of Power Waste
26.4. Reduction of Activity
26.5. Transformations and Tradeoffs
26.6. Some Emerging Methods

26.1 The Need for Low-Power Design

Portable and wearable electronic devices

Nickel-cadmium batteries: 40-50 W-hr per kg of weight

Practical battery weight < 1 kg (< 0.1 kg if wearable device)

Total power ≈ 3-5 W for a full day's work between recharges

Modern high-performance mircoprocessors use 10s Watts

  Power is proportional to die area × clock frequency

  Cooling of micros not yet a problem; but for MPPs . . .

New battery technologies cannot keep pace with demand

Demand for more speed and functionality (multimedia, etc.)

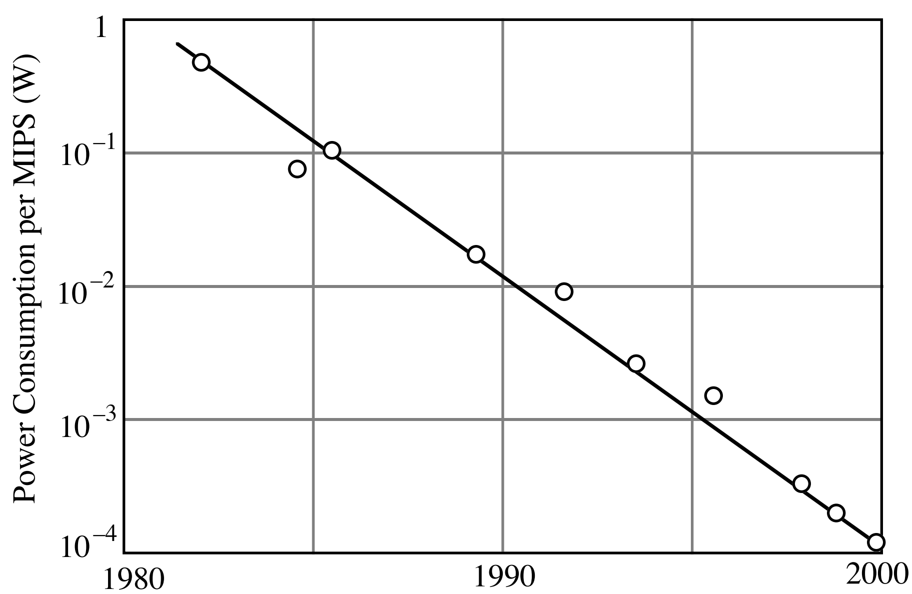


Fig. 26.1 Power consumption trend in DSPs [Raba98].

26.2 Sources of Power Consumption

Both average and peak power are important

Peak power impacts power distribution and signal integrity

Typically, low-power design aims at reducing both

Power dissipation in CMOS digital circuits

Static: leakage current in imperfect switches (< 10%)

Dynamic: due to (dis)charging of parasitic capacitance

$P_{avg} \approx \alpha f C V^2$

f: data rate (clock frequency) α: "activity"

Example: A 32-bit off-chip bus operates at 5 V and 100 MHz and drives a capacitance of 30 pF per bit. If random values were placed on the bus in every cycle, we would have $\alpha = 0.5$. To account for data correlation and idle bus cycles, assume $\alpha = 0.2$. Then:

$$P_{avg} \approx \alpha f C V^2 = 0.2 \times 10^8 \times (32 \times 30 \times 10^{-12}) \times 5^2 = 0.48\text{ W}$$

Once we fix the data rate f, there are but three ways to reduce the power requirements:

1. Using a lower supply voltageV
2. Reducing the parasitic capacitance C
3. Lowering the switching activity α

26.3 Reduction of Power Waste

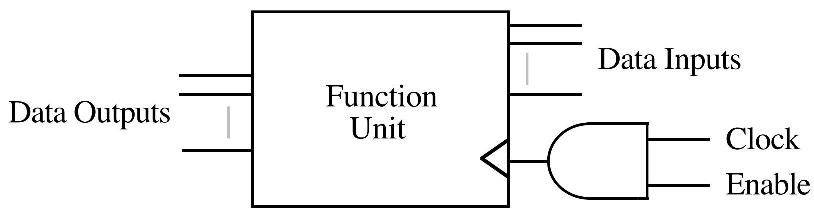


Fig. 26.2 Saving power through clock gating.

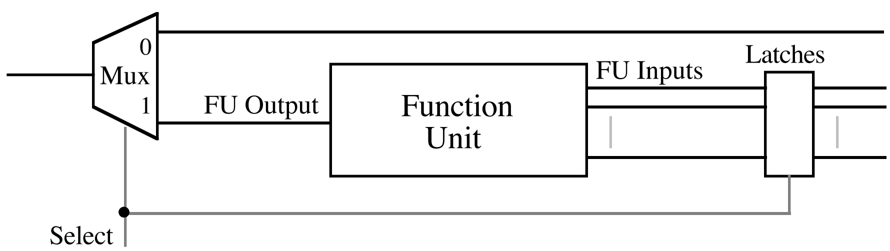


Fig. 26.3 Saving power via guarded evaluation.

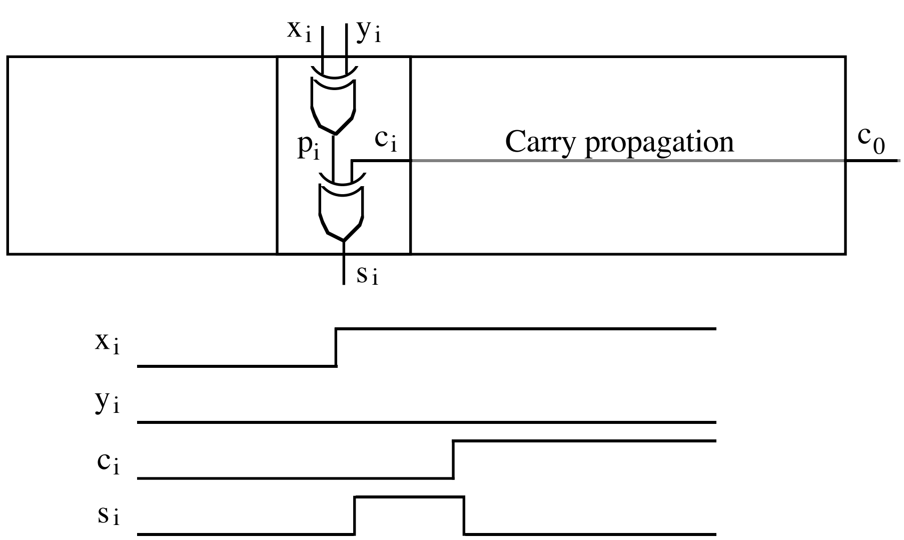


Fig. 26.4 Example of glitching in a ripple-carry adder.

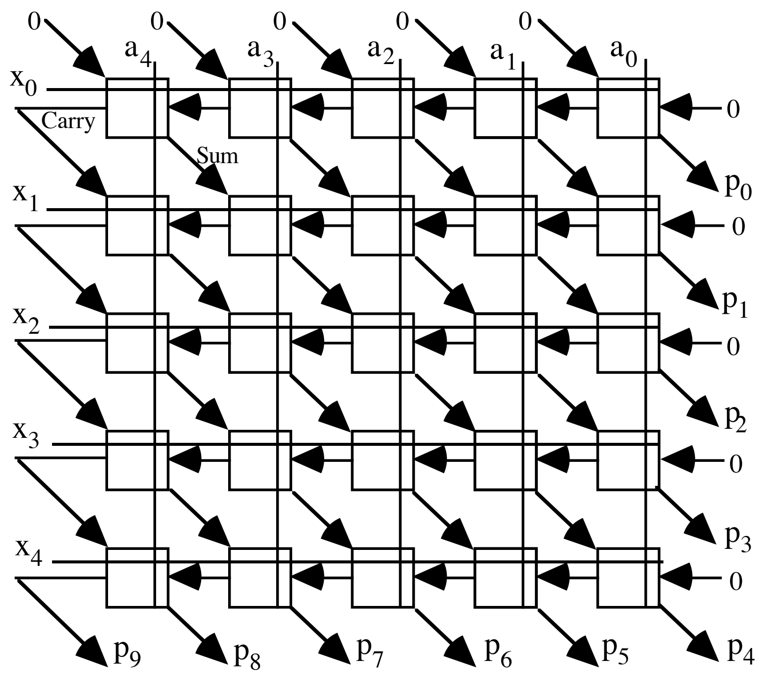
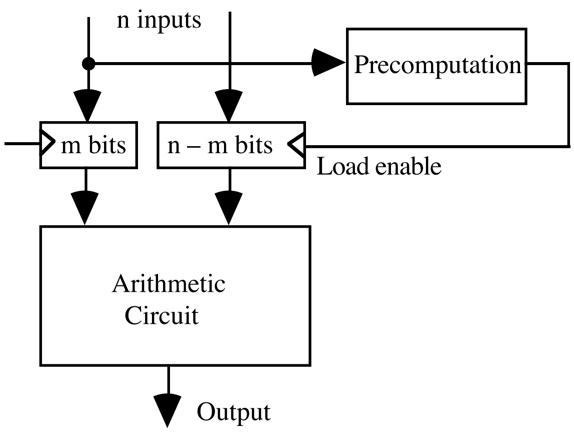


Fig. 26.5 An array multiplier with gated FA cells.

26.4 Reduction of Activity



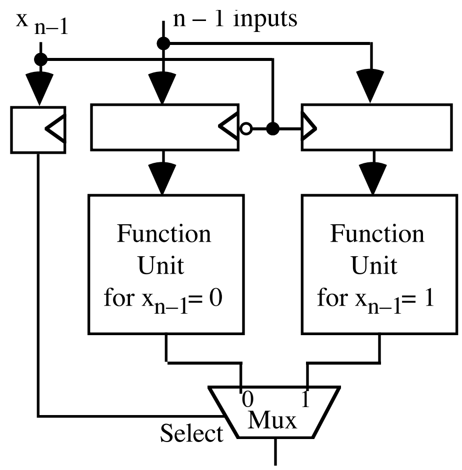
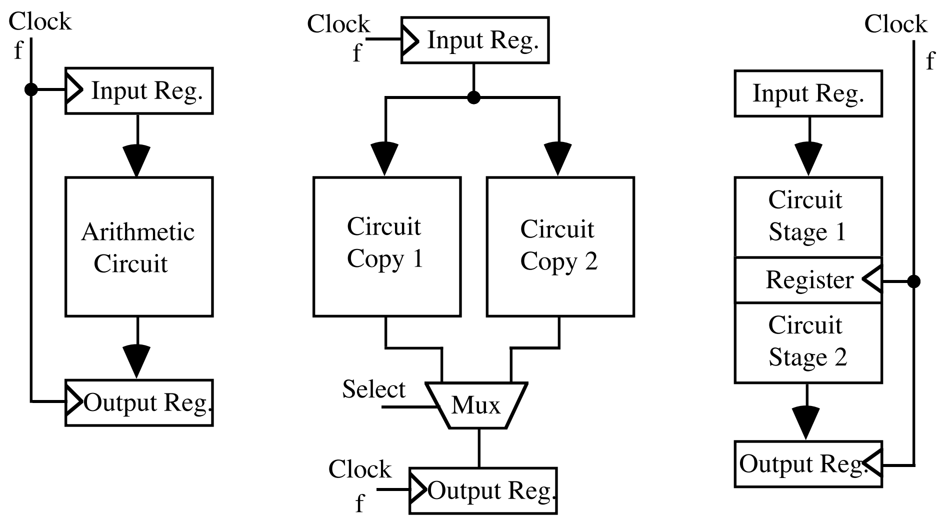


Fig. 26.6 Reduction of activity by precomputation.

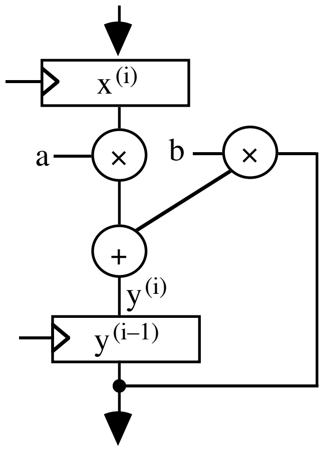
Fig. 26.7 Reduction of activity via Shannon expansion.

26.5 Transformations and Tradeoffs



Frequency = f	Frequency = 0.5f	Frequency = f
Capacitance = C	Capacitance = 2.2C	Capacitance = 1.2C
Voltage = V	Voltage = 0.6V	Voltage = 0.6V
Power = P	Power = 0.396P	Power = 0.432P

Fig. 26.8 Reduction of power via parallelism or pipelining.



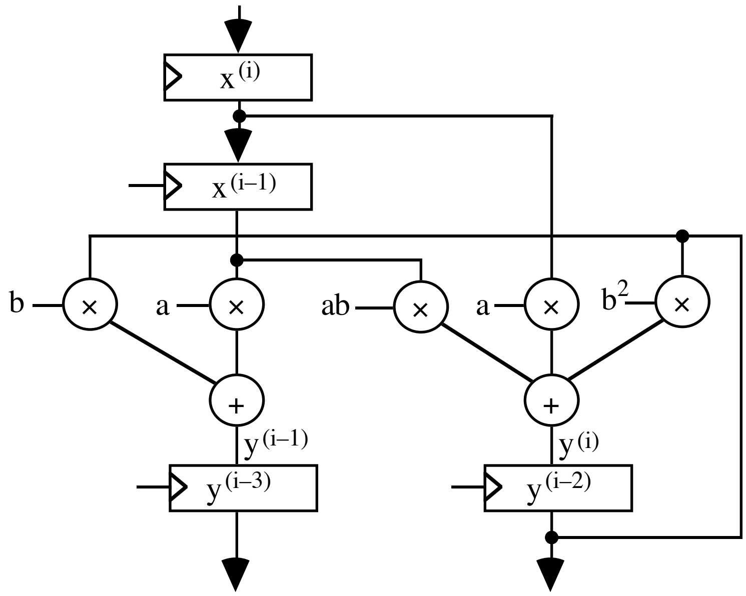
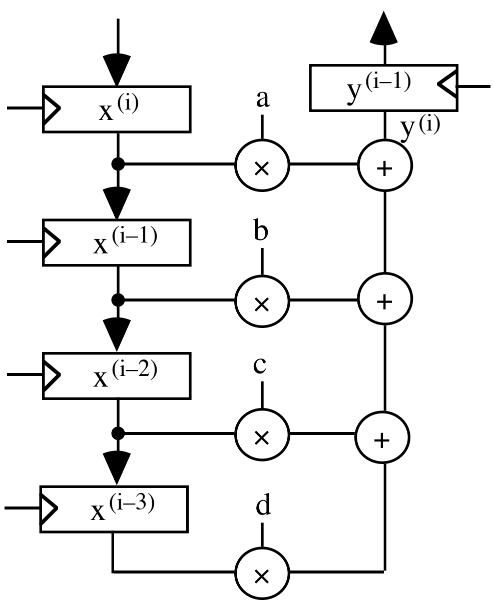


Fig. 26.9 Direct realization of a first-order IIR filter.

Fig. 26.10 Realization of a first-order filter, unrolled once.



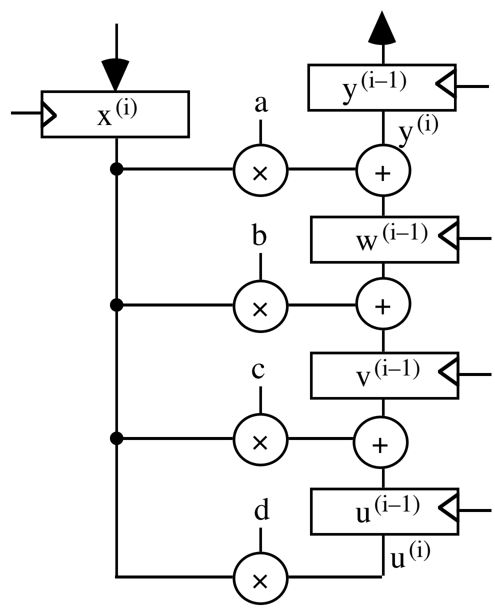


Fig. 26.11 Possible realization of a fourth-order FIR filter.

Fig. 26.12 Realization of the retimed fourth-order FIR filter.

26.6 Some Emerging Methods

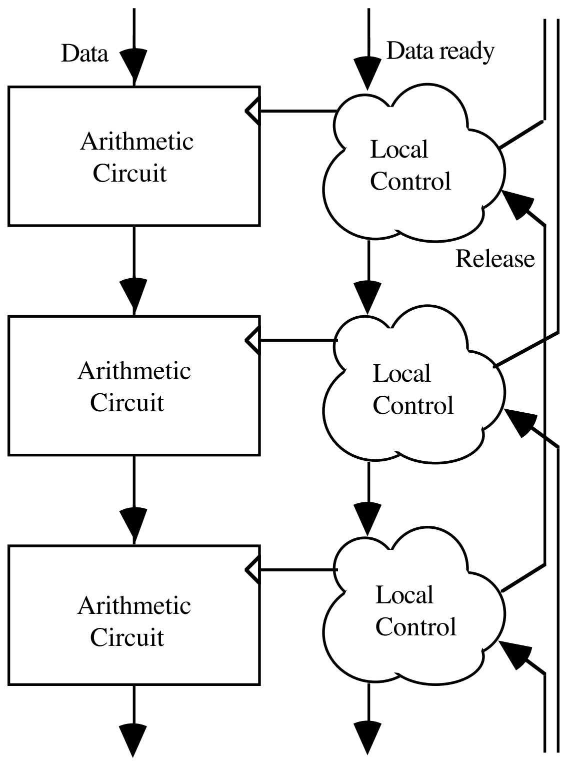


Fig. 26.13 Part of an asynchronous chain of computations.

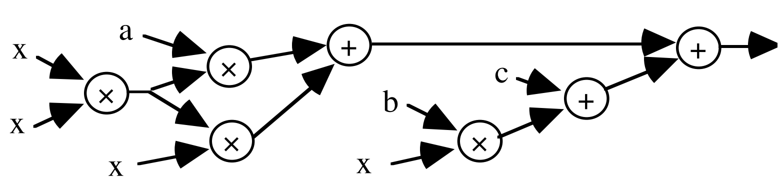


Fig. for problem 26.5

27 Fault-Tolerant Arithmetic

Chapter Goals

Learn about errors due to hardware faults
or hostile environmental conditions,
and how to deal with or circumvent them

Chapter Highlights

Modern components are very robust, but ...
put millions/billions of them together
and something is bound to go wrong
Can arithmetic be protected via encoding?
Reliable circuits and robust algorithms

Chapter Contents

27.1 Faults, Errors, and Error Codes
27.2 Arithmetic Error-Detecting Codes
27.3 Arithmetic Error-Correcting Codes
27.4 Self-Checking Function Units
27.5 Algorithm-Based Fault Tolerance
27.6 Fault-Tolerant RNS Arithmetic

27.1 Faults, Errors, and Error Codes

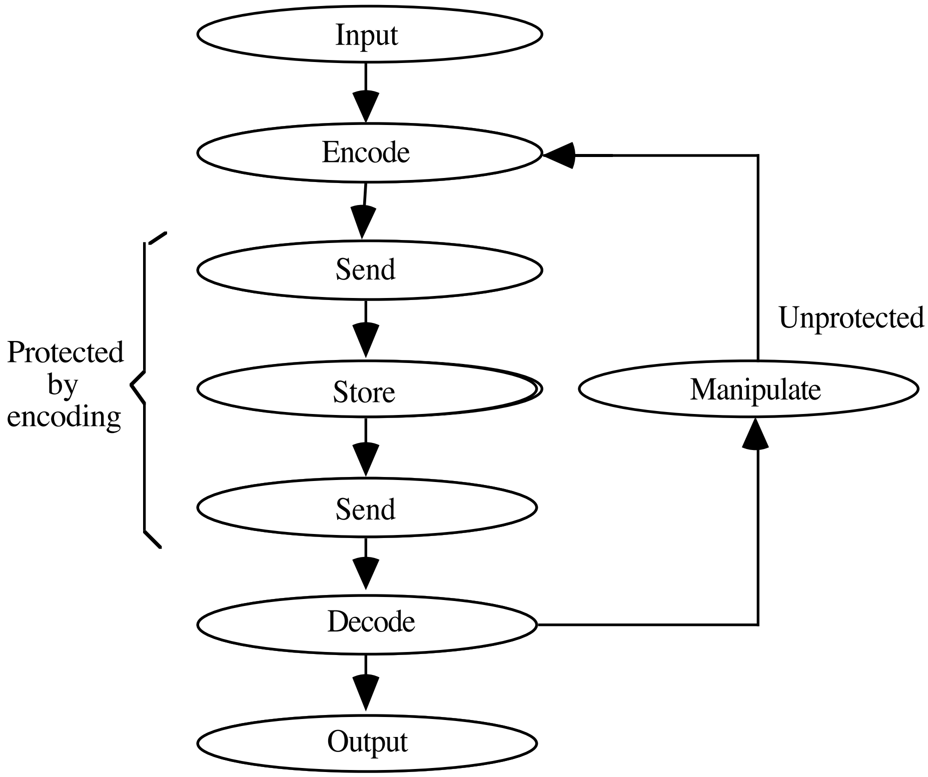


Fig. 27.1 A common way of applying information coding techniques.

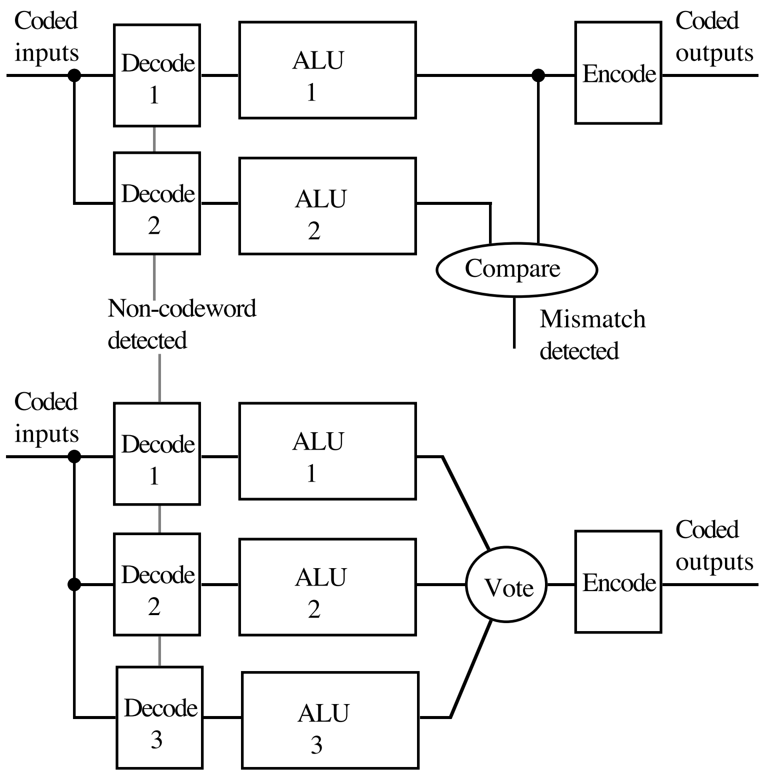


Fig. 27.2 Arithmetic fault detection or fault tolerance (masking) with replicated units.

Unsigned addition		0010 0111 0010 0001
	+	0101 1000 1101 0011
Correct sum		0111 1111 1111 0100
Erroneous sum		1000 0000 0000 0100

↑ Stage generating an erroneous carry of 1

Fig. 27.3 How a single carry error can produce an arbitrary number of bit-errors (inversions).

The arithmetic weight of an error

Minimum number of signed powers of 2 that must be added to the correct value in order to produce the erroneous result (or vice versa). Examples:

Correct	0111 1111 1111 0100	1101 1111 1111 0100
Erroneous	1000 0000 0000 0100	0110 0000 0000 0100
Difference (error)	$16 = 2^4$	$-32752 = -2^{15} + 2^4$
Error (min-weight BSD)	0000 0000 0001 0000	$^{-}$1000 0000 0001 0000
Arithmetic weight of error	1	2
Error type	Single, positive	Double, negative

27.2 Arithmetic Error-Detecting Codes

Arithmetic error-detecting codes:

1. Are characterized by arithmetic weights of detectable errors
2. Allow direct arithmetic on coded operands

a. Product codes or AN codes

Represent N by the product AN (A = check modulus)

For odd A, all weight-1 arithmetic errors are detected

Arithmetic errors of weight 2 and higher may go undetected

e.g., the error $32736 = 2^{15} - 2^5$

undetectable with A = 3, 11, or 31

Error detection: check divisibility by A

Encoding/decoding: multiply/divide by A

Arithmetic also requires multiplication and division by A

Thus, we are led to low-cost product codes with $A = 2^a - 1$

Multiplication by $A = 2^a - 1$: done by shift-subtract

Division by $A = 2^a - 1$: done a bits at a time as follows

Given $y = (2^a - 1)x$, find x by computing $2^a x - y$

. . . xxxx 0000	–	. . . xxxx xxxx	=	. . . xxxx xxxx
Unknown $2^a x$		Known $(2^a - 1)x$		Unknown x

Theorem 27.1: Any unidirectional error with arithmetic weight not exceeding $a - 1$ is detectable by a low-cost product code using the check modulus $A = 2^a - 1$

Product codes are nonseparate (nonseparable) codes

Data and redundant check info are intermixed

Arithmetic on AN-coded operands

Addition/subtraction is done directly:

$$Ax \pm Ay = A(x \pm y)$$

Direct multiplication results in:

$$Aa \times Ax = A^2ax$$

So the result must be corrected through division by A

For division, if $z = qd + s$, we have:

$$Az = q(Ad) + As \qquad \text{thus, } q \text{ is unprotected}$$

Possible cure: premultiply the dividend Az by A

The result will need correction

Square rooting leads to a problem similar to division

$$\lfloor \sqrt{A^2x} \rfloor = \lfloor A\sqrt{x} \rfloor \text{ which is not the same as } A \lfloor \sqrt{x} \rfloor$$

b. Residue codes

Represent N by the pair (N, C(N)), where C(N) = N mod A

Residue codes are separate (separable) codes

Separate data and check parts make decoding trivial

Encoding: given N, compute C(N) = N mod A

Low-cost residue codes use $A = 2^a - 1$

Arithmetic on residue-coded operands

Add/subtract: data and check parts are handled separately

$$(x, C(x)) \pm (y, C(y)) = (x \pm y, (C(x) \pm C(y)) \bmod A)$$

Multiply

$$(a, C(a)) \times (x, C(x)) = (a \times x, (C(a) \times C(x)) \bmod A)$$

Divide/square-root: difficult

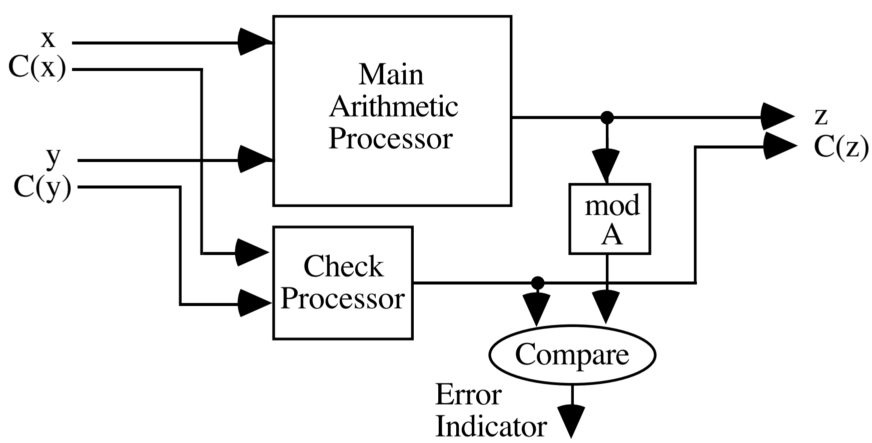


Fig. 27.4 Arithmetic processor with residue checking.

27.3 Arithmetic Error-Correcting Codes

Table 27.1 Error syndromes for weight-1 arithmetic errors in the (7, 15) biresidue code

Positive Error	Error Syndrome mod 7	mod 15	Negative Error	Error Syndrome mod 7	mod 15
1	1	1	–1	6	14
2	2	2	–2	5	13
4	4	4	–4	3	11
8	1	8	–8	6	7
16	2	1	–16	5	14
32	4	2	–32	3	13
64	1	4	–64	6	11
128	2	8	–128	5	7
256	4	1	–256	3	14
512	1	2	–512	6	13
1024	2	4	–1024	5	11
2048	4	8	–2048	3	7
4096	1	1	–4096	6	14
8192	2	2	–8192	5	13
16384	4	4	–16384	3	11
32768	1	8	–32768	6	7

Biresidue code with relatively prime low-cost check moduli $A = 2^a - 1$ and $B = 2^b - 1$ supports $a \times b$ bits of data for weight-1 error correction

Representational redundancy = $(a + b)/(ab) = 1/a + 1/b$

27.4 Self-Checking Function Units

Self-checking unit: any fault from a prescribed set

does not affect the correct output (masked) or

leads to a noncodeword output (detectable)

An invalid result is

detected immediately by a code checker or

propagated downstream by the next self-checking unit

To build self-checking (SC) units, we need self-checking code checkers that never validate a noncodeword, even when they are faulty

Example: SC checker for inverse residue code (N, C' (N))
N mod A should be the bitwise complement of C' (N)
Verifying that signal pairs (x_i, y_i) are all (1, 0) or (0, 1)
= finding the AND of Boolean values encoded as
1: (1, 0) or (0, 1) 0: (0, 0) or (1, 1)

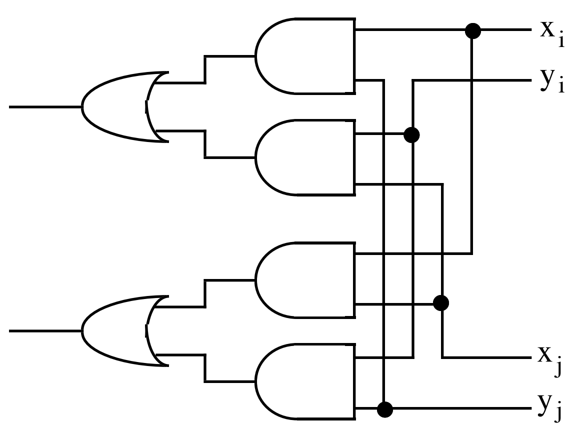


Fig. 27.5 Two-input AND circuit, with 2-bit inputs (x_i, y_i) and (x_j, y_j), for use in a self-checking code checker.

27.5 Algorithm-Based Fault Tolerance

Alternative to error detection at the level of basic operation:

Accept that arithmetic operations may yield incorrect results

Detect/correct errors at data-structure or application level

Example: multiplication of matrices X and Y yielding P

Row checksum, column checksum, and full checksum matrices (mod 8)

$$M = \begin{bmatrix} \square\,21 & 6 \\ \square\,53 & 4 \\ \square\,32 & 7 \end{bmatrix} \qquad M_r = \begin{bmatrix} \square\,21 & 6 & 1 \\ \square\,53 & 4 & 4 \\ \square\,32 & 7 & 4 \end{bmatrix}$$

$$M_c = \begin{bmatrix} \square\,21 & 6 \\ \square\,53 & 4 \\ \square\,32 & 7 \\ \square\,26 & 1 \end{bmatrix} \qquad M_f = \begin{bmatrix} \square\,21 & 6 & 1 \\ \square\,53 & 4 & 4 \\ \square\,32 & 7 & 4 \\ \square\,26 & 1 & 1 \end{bmatrix}$$

Fig. 27.6 A 3×3 matrix M with its row, column, and full checksum matrices M_r, M_c, and M_f.

Theorem 27.3: If $P = X \times Y$, we have $P_f = X_c \times Y_r$

With floating-point values, the equalities are approximate

Theorem 27.4: In a full-checksum matrix, any single erroneous element can be corrected and any three errors can be detected

27.6 Fault-Tolerant RNS Arithmetic

Residue number systems allow very elegant and effective error detection and correction schemes by means of redundant residues (extra moduli)

Example: RNS(8 | 7 | 5 | 3), Dynamic range M = 8 × 7 × 5 × 3 = 840; redundant modulus: 11. Any error confined to a single residue is detectable

The redundant modulus must be the largest one, say m

Error detection scheme:

(1) Use other residues to compute the residue of the number mod m (this process is known as base extension)

(2) Compare the computed and actual mod-m residues

The beauty of this method is that arithmetic algorithms are totally unaffected; error detection is made possible by simply extending the dynamic range of the RNS

Example: RNS(8 | 7 | 5 | 3), redundant moduli: 13 and 11

25 = (12, 3, 1, 4, 0, 1), erroneous version = (12, 3, 1, 6, 0, 1)

Transform (–, –, 1, 6, 0, 1) to (5, 1, 1, 6, 0, 1) by means of base extension

The difference between the first two components of the corrupted and reconstructed numbers is (+7, +2) which is the error syndrome

28 Past, Present, and Future

Chapter Goals

Wrap things up, provide perspective, and examine arithmetic in a few key systems

Chapter Highlights

One must look at arithmetic in context of

- computational requirements
- technological constraints
- overall system design goals
- past and future developments

Current trends and research directions?

Chapter Contents

28.1 Historical Perspective
28.2 An Early High-Performance Machine
28.3 A Modern Vector Supercomputer
28.4 Digital Signal Processors
28.5 A Widely Used Microprocessor
28.6 Trends and Future Outlook

28.1 Historical Perspective

1940s

Machine arithmetic was crucial in proving the feasibility of computing with stored-program electronic devices

Hardware for addition, use of complement representation, and shift-add multiplication and division algorithms were developed and fine-tuned

A seminal report by A.W. Burkes, H.H. Goldstein, and J. von Neumann contained ideas on choice of number radix, carry propagation chains, fast multiplication via carry-save addition, and restoring division

State of computer arithmetic circa 1950: overview paper by R.F. Shaw [Shaw50]

1950s

The focus shifted from feasibility to algorithmic speedup methods and cost-effective hardware realizations

By the end of the decade, virtually all important fast-adder designs had already been published or were in the final phases of development

Rresidue arithmetic, SRT division, and CORDIC algorithms were proposed and implemented

Snapshot of the field circa 1960: overview paper by O.L. MacSorley [MacS61]

1960s

Tree multipliers, array multipliers, high-radix dividers, convergence division, redundant signed-digit arithmetic were introduced

Implementation of floating-point arithmetic operations in hardware or firmware (in microprogram) became prevalent

Many innovative ideas originated from the design of early supercomputers, when the demand for high performance, along with the still high cost of hardware, led designers to novel and cost-effective solutions. Examples:

IBM System/360 Model 91 [Ande67]

CDC 6600 [Thor70]

1970s

Advent of microprocessors and vector supercomputers

Early LSI chips were quite limited in the number of transistors or logic gates that they could accommodate; thus microprogrammed control was a natural choice for single-chip processors which were not yet expected to offer high performance

At the high end of performance spectrum, pipelining methods were perfected to allow the throughput of arithmetic units to keep up with computational demand in vector supercomputers. Example:

Cray 1 supercomputer and its successors

1980s

Widespread use of VLSI circuits triggered a reconsideration of virtually all arithmetic designs in light of interconnection cost and pin limitations

For example, carry-lookahead adders, that appeared to be ill-suited to VLSI, were shown to be efficiently realizable after suitable modifications

Similar ideas were applied to more efficient VLSI implementation of tree and array multipliers

Bit-serial and on-line arithmetic were advanced to deal with severe pin limitations in VLSI packages

Arithmetic-intensive signal processing functions became driving forces for low-cost and/or high-performance embedded hardware: DSP chips

1990s

No breakthrough design concept

Increasing demand for performance led to fine-tuning of arithmetic algorithms and implementations; hence, a wide array of hybrid designs

Increasing use of table lookup and tight integration of arithmetic unit and other parts of the processor for maximum performance

Clock speeds reached and surpassed 100, 200, 300, 400, and 500 MHz in rapid succession; pipelining used to ensure smooth flow of data through the system. Example:

Intel's Pentium Pro (P6), predecessor of Pentium II

28.2 An Early High-Performance Machine

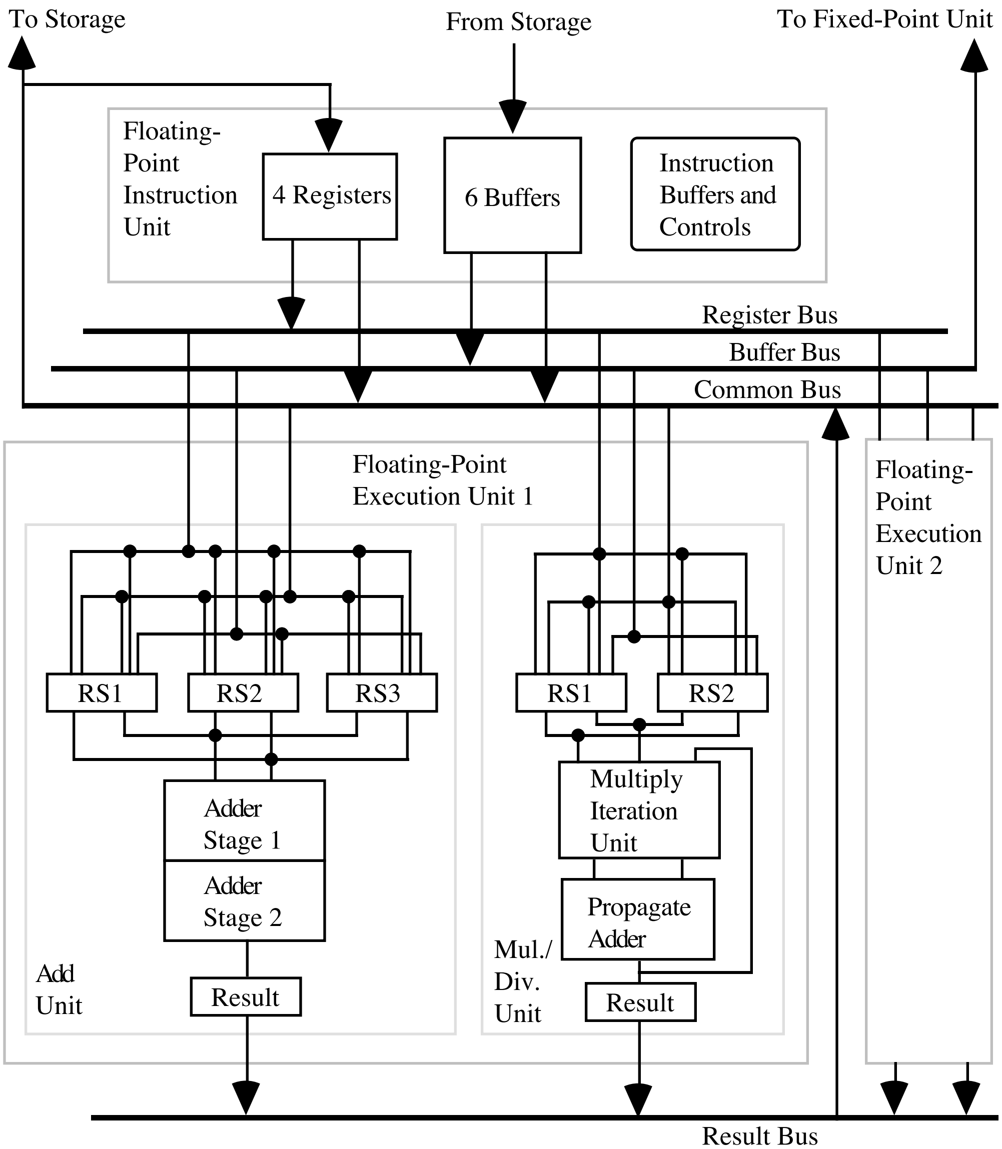


Fig. 28.1 Overall structure of the IBM System/360 Model 91 floating-point execution unit.

28.3 A Modern Vector Supercomputer

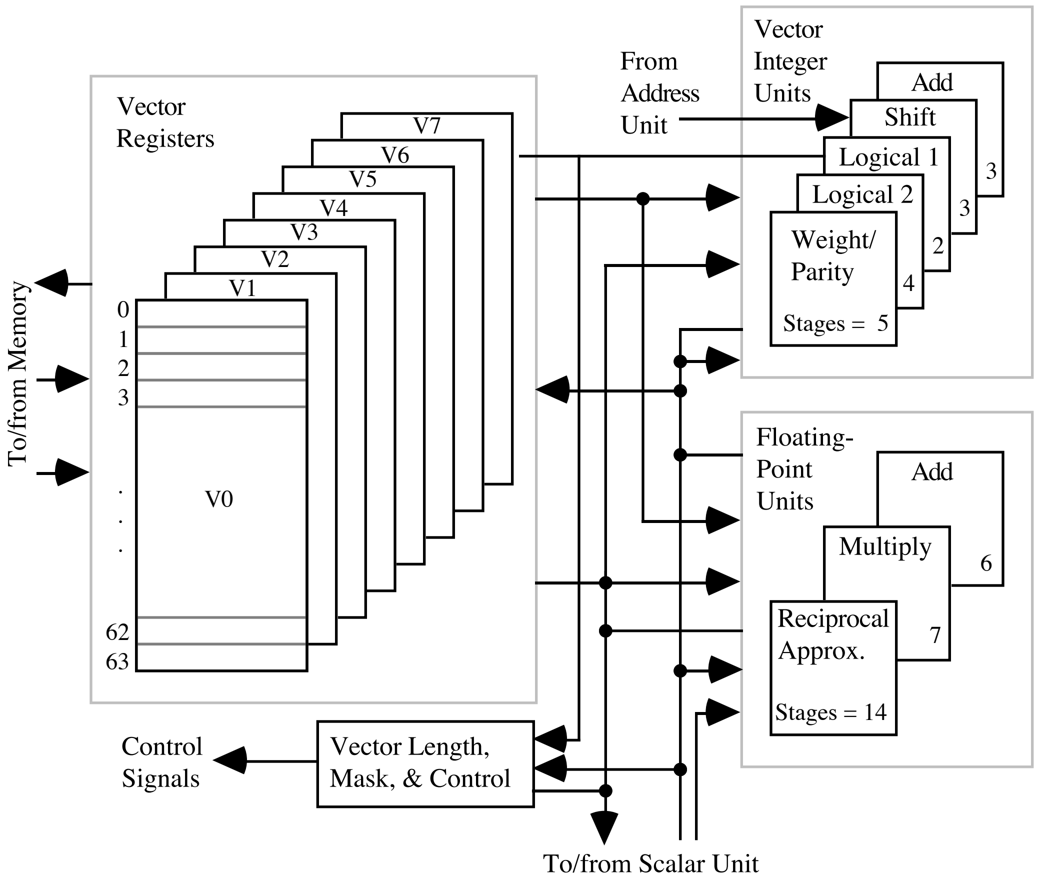


Fig. 28.2 The vector section of one of the processors in the Cray X-MP/Model 24 supercomputer.

Pipeline setup and shutdown overheads

Vector unit not efficient for short vectors (break-even point)

Pipeline chaining

28.4 Digital Signal Processors

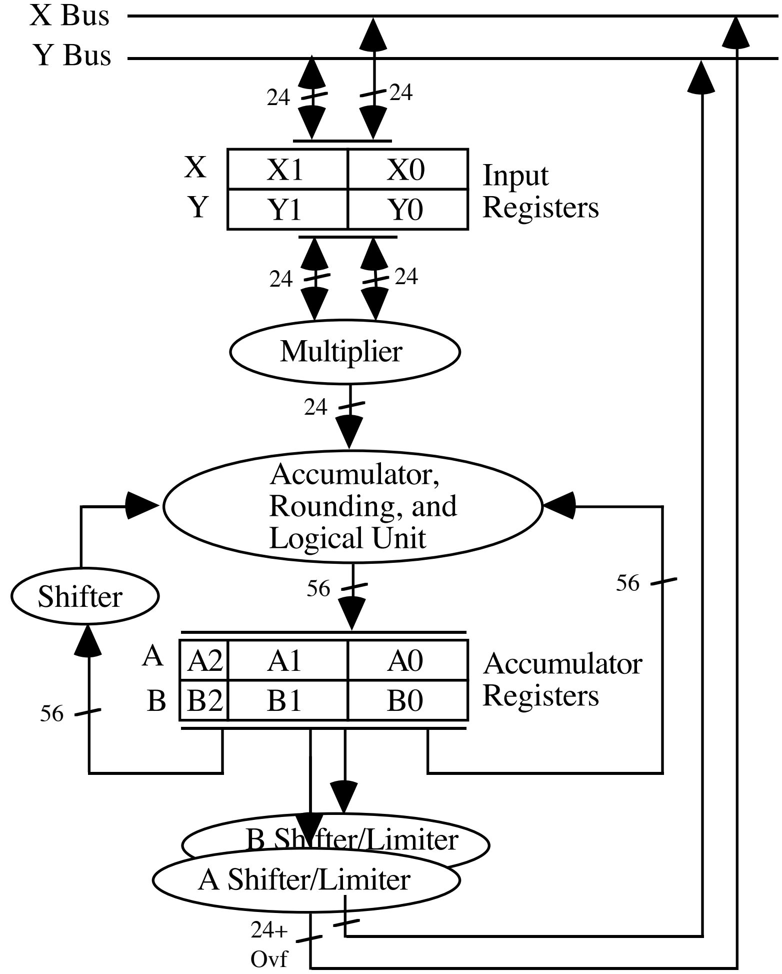


Fig. 28.3 Block diagram of the data ALU in Motorola's DSP56002 (fixed-point) processor.

Example DSP instructions

ADD	A, B	{ A + B → B }
SUB	X, A	{ A – X → A }
MPY	±X1, X0, B	{ ±X1 × X0 → B }
MAC	±Y1, X1, A	{ A ± Y1 × X1 → A }
AND	X1, A	{ A AND X1 → A }

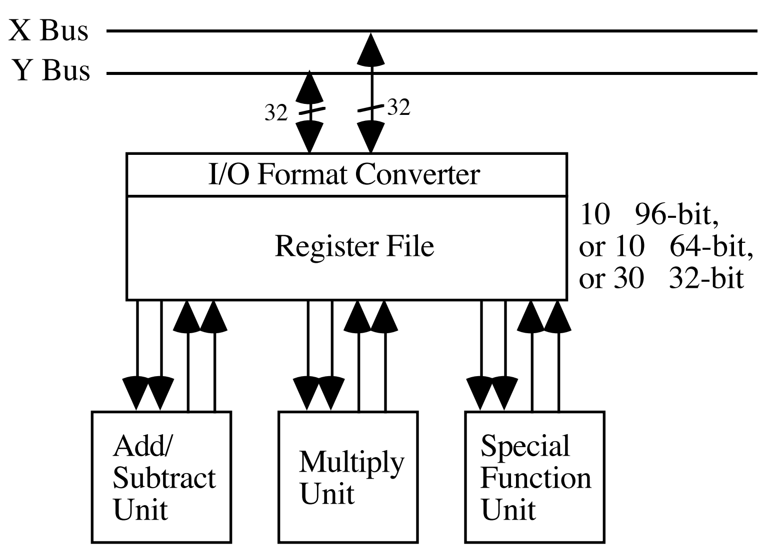
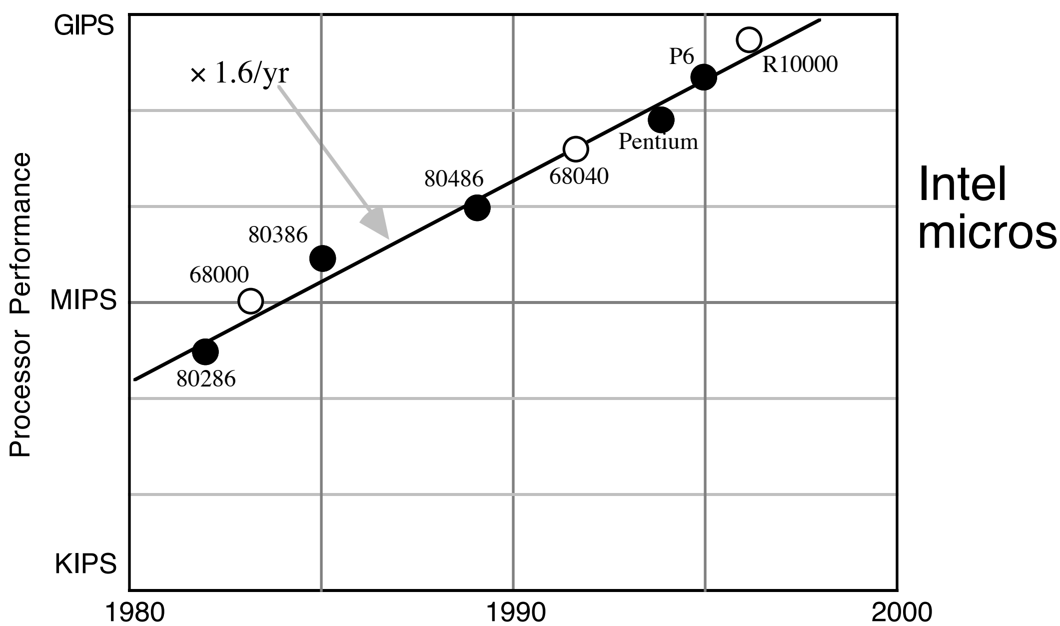


Fig. 28.4 Block diagram of the data ALU in Motorola's DSP96002 (floating-point) processor.

28.5 A Widely Used Microprocessor



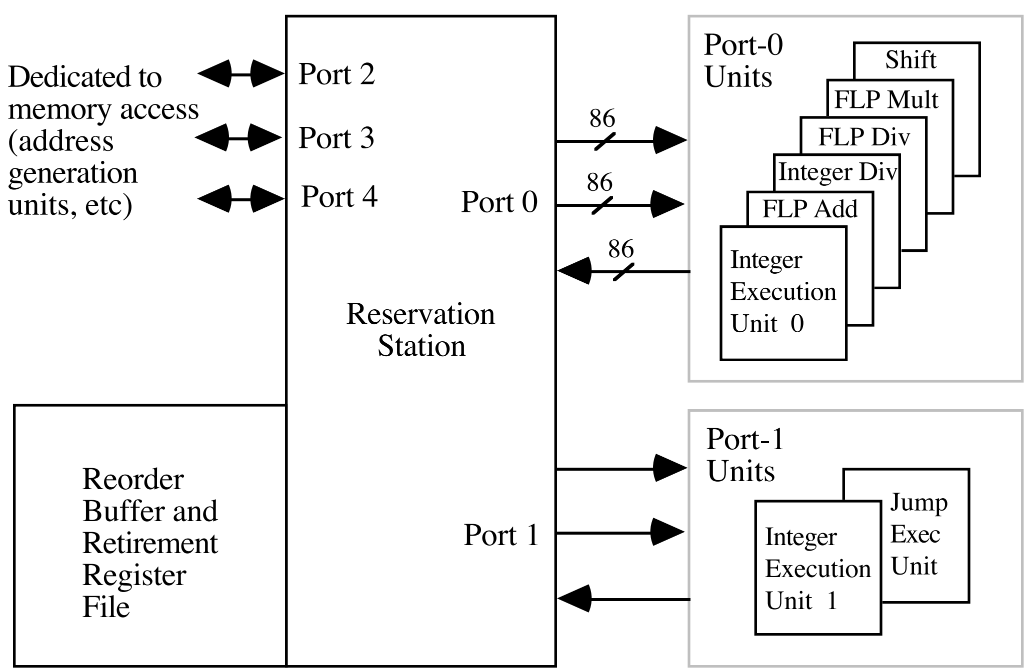


Fig. 28.5 Key parts of the CPU in the Intel Pentium Pro (P6) microprocessor.

28.6 Trends and Future Outlook

Design: Shift of attention from algorithms to optimizations at the level of transistors and wires; this explains the proliferation of hybrid designs

Applications: Shift from high-speed or high-throughput designs in mainframes to low cost and low power for embedded systems

Renewed interest in bit- and digit-serial arithmetic as mechanisms to reduce the VLSI area and to improve packageability and testability

Synchronous versus asynchronous design (asynchrounous design has some overhead, but an equivalent overhead is being paid for clock distribution and/or systolization)

New technologies and design paradigms may alter the way in which we view or design arithmetic circuits

- Neuron-type computational elements
- Optical computing (redundant representations)
- Multivalued logic (match to high-radix arithmetic)
- Configurable logic

Arithmetic complexity theory

THE END!

"You're up to date. Take my advice and try to keep it that way. It'll be tough to do; make no mistake about it. The phone will ring and it'll be the administrator — talking about budgets. The doctors will come in, and they'll want this bit of information and that. Then you'll get the salesman. Until at the end of the day you'll wonder what happened to it and what you've accomplished; what you've achieved.

"That's the way the next day can go, and the next, and the one after that. Until you find a year has slipped by, and another, and another. And then suddenly, one day, you'll find everything you knew is out of date. That's when it's too late to change.

"Listen to an old man who's been through it all, who made the mistake of falling behind. Don't let it happen to you! Lock yourself in a closet if you have to! Get away from the phone and the files and paper, and read and learn and listen and keep up to date. Then they can never touch you, never say, 'He's finished, all washed up; he belongs to yesterday.' "

Arthur Hailey, The Final Diagnosis

