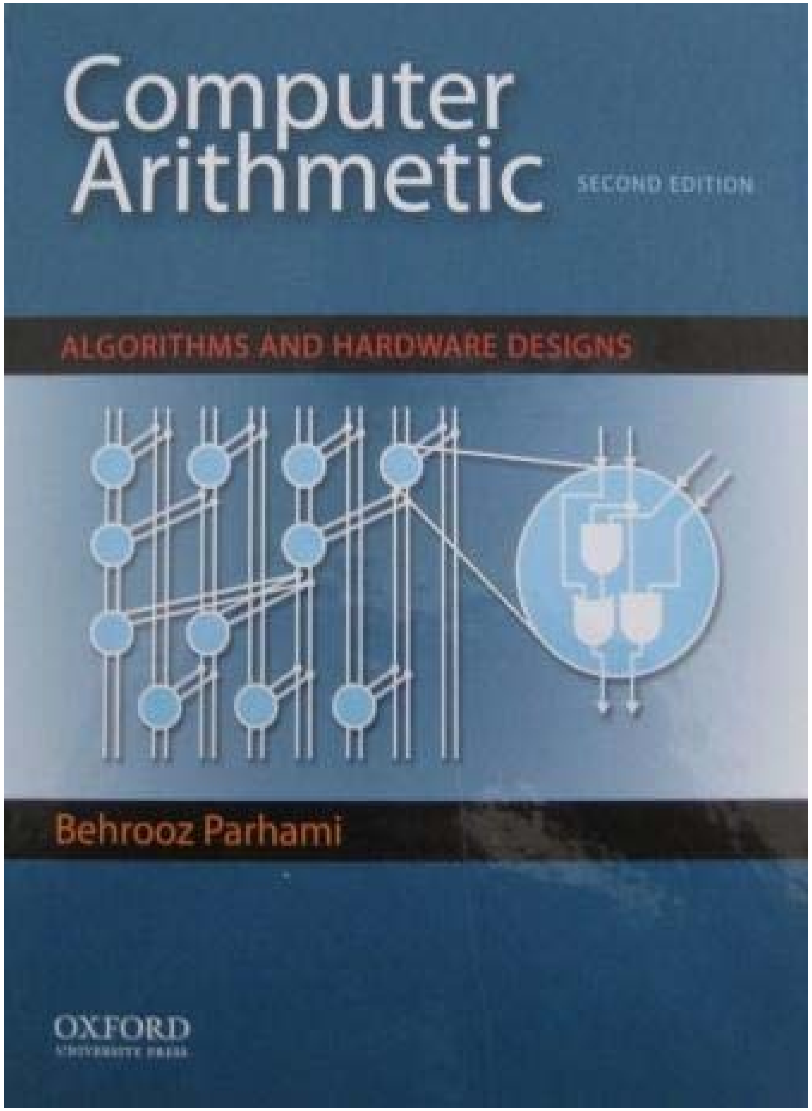


Part I
Number Representation

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

Appendix: Past, Present, and Future

About This Presentation

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

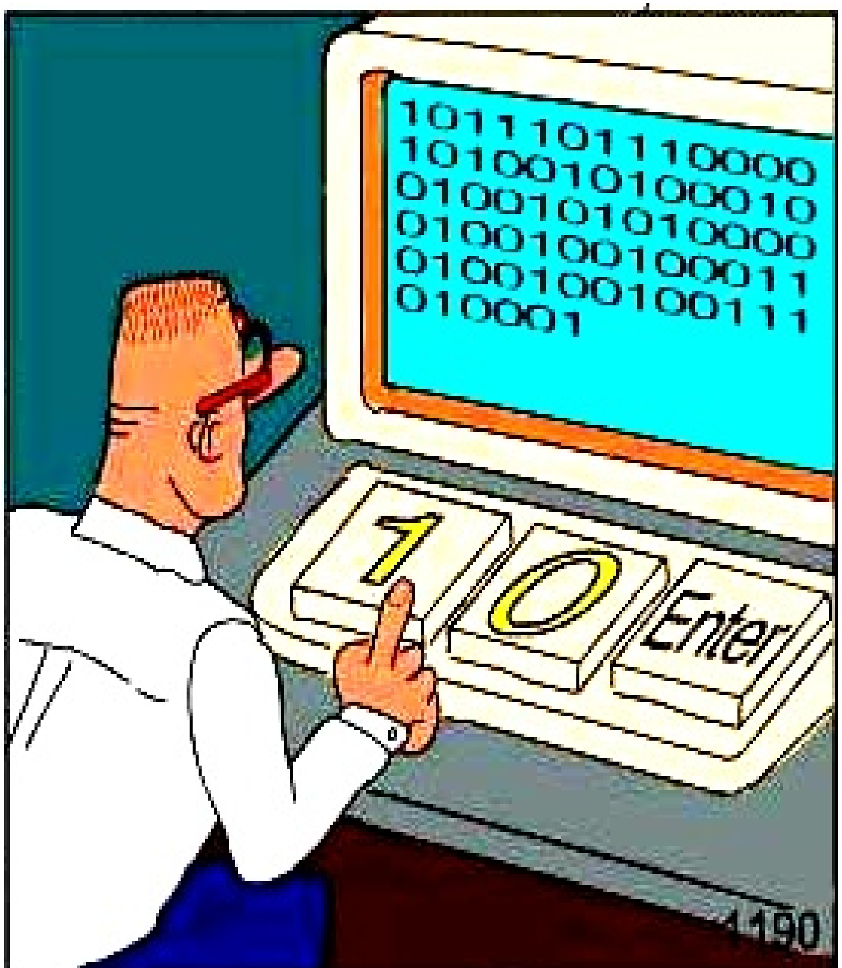
Edition	Released	Revised	Revised	Revised	Revised
First	Jan. 2000	Sep. 2001	Sep. 2003	Sep. 2005	Apr. 2007
	Apr. 2008	Apr. 2009			
Second	Apr. 2010	Mar. 2011			

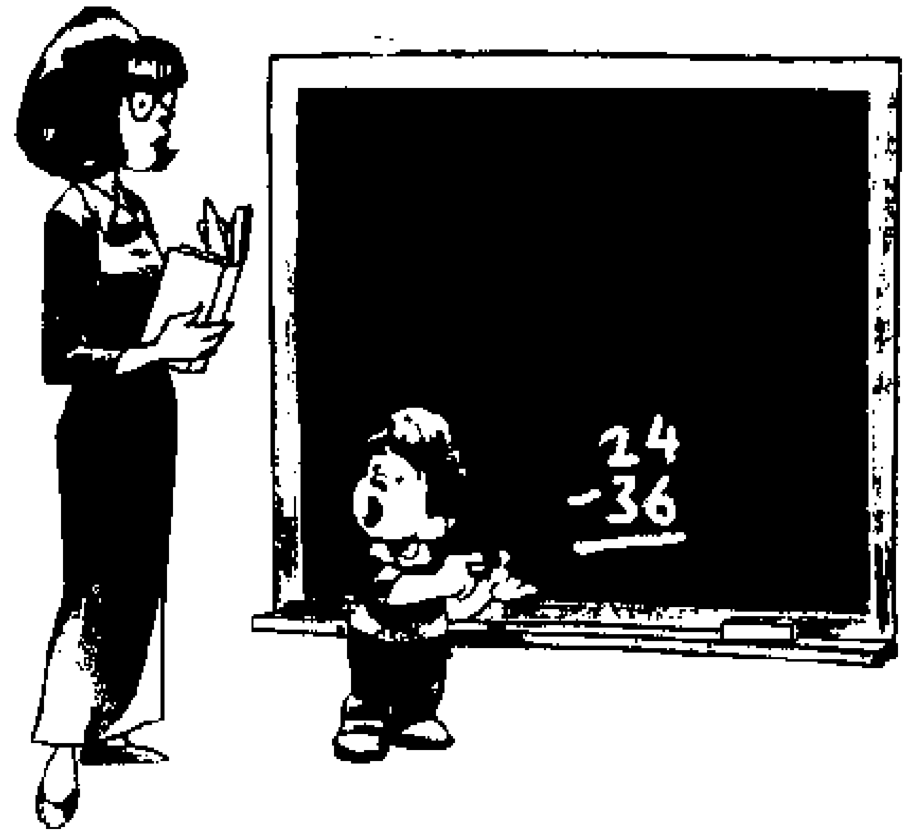
I Background and Motivation

Number representation arguably the most important topic:

- Effects on system compatibility and ease of arithmetic
- 2's-complement, redundant, residue number systems
- Limits of fast arithmetic
- Floating-point numbers to be covered in Chapter 17

Topics in This Part	
Chapter 1	Numbers and Arithmetic
Chapter 2	Representing Signed Numbers
Chapter 3	Redundant Number Systems
Chapter 4	Residue Number Systems





“This can’t be right . . . It goes into the red!”

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?
Ways of encoding numbers in *k* bits
Radices and digit sets: conventional, exotic
Conversion from one system to another
Dot notation: a useful visualization tool

Numbers and Arithmetic: Topics

Topics in This Chapter
1.1 What is Computer Arithmetic?
1.2 Motivating Examples
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 gave incorrect quotient
First noted in 1994 by Tom 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 Intel Slogans for the Pentium

Humor, circa 1995 (in the wake of the floating-point division bug)

- 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

Aspects of, and Topics in, Computer Arithmetic

Hardware (our focus in this book)

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

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

General-purpose

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

Special-purpose

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

Software

Numerical methods for solving systems of linear equations, partial differential eq'ns, and so on

Issues: Algorithms
Error analysis
Computational complexity
Programming
Testing, verification

Fig. 1.1 The scope of computer arithmetic.

1.2 A Motivating Example

Using a calculator with √, x^2, and x^y functions, compute:

$u = \sqrt{}\sqrt{} \ldots \sqrt{} 2$	=	1.000 677 131	"1024th root of 2"
$v = 2^{1/1024}$	=	1.000 677 131	

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

$x = (((u^2)^2)\ldots)^2$	=	1.999 999 963
$x' = u^{1024}$	=	1.999 999 973
$y = (((v^2)^2)\ldots)^2$	=	1.999 999 983
$y' = v^{1024}$	=	1.999 999 994

Perhaps v and u are not really the same value

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

$(u - 1) \times 1000$	=	0.677 130 680	[Hidden ... (0) 68]
$(v - 1) \times 1000$	=	0.677 130 690	[Hidden ... (0) 69]

Finite Precision Can Lead to Disaster

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

Source http://www.ima.umn.edu/~arnold/disasters/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)

Problem specifics:

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) $\times$ (1676 m/s) $\approx$ 570 m

Inadequate Range Can Lead to Disaster

Example: Explosion of Ariane Rocket (1996 June 4)

Source http://www.ima.umn.edu/~arnold/disasters/disasters.html

Unmanned Ariane 5 rocket of the European Space Agency veered off its flight path, broke up, and exploded only 30 s 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”

Problem specifics:

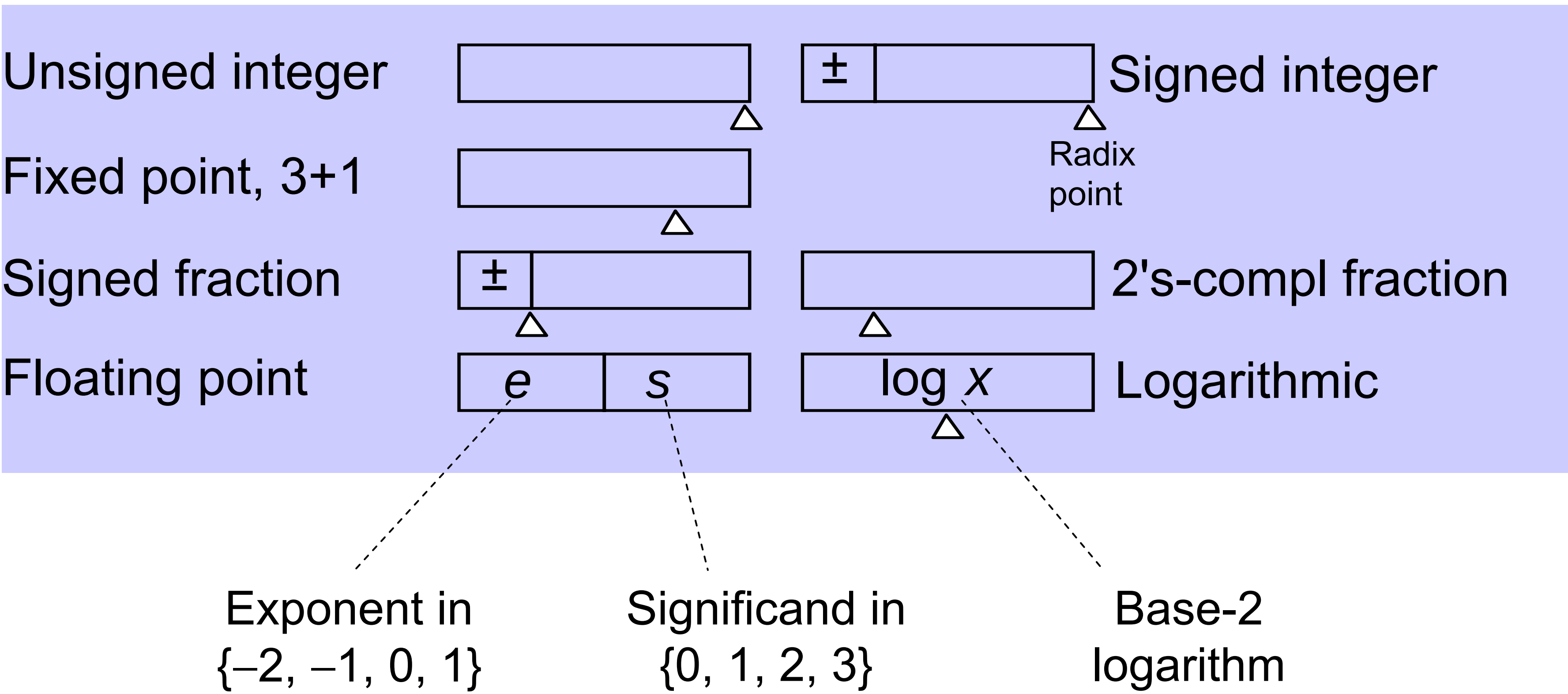
A 64 bit floating point number relating to the horizontal velocity of the rocket was being converted to a 16 bit signed integer

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 = Système de Référence Inertielle or Inertial Reference System

1.3 Numbers and Their Encodings

Some 4-bit number representation formats



Encoding Numbers in 4 Bits

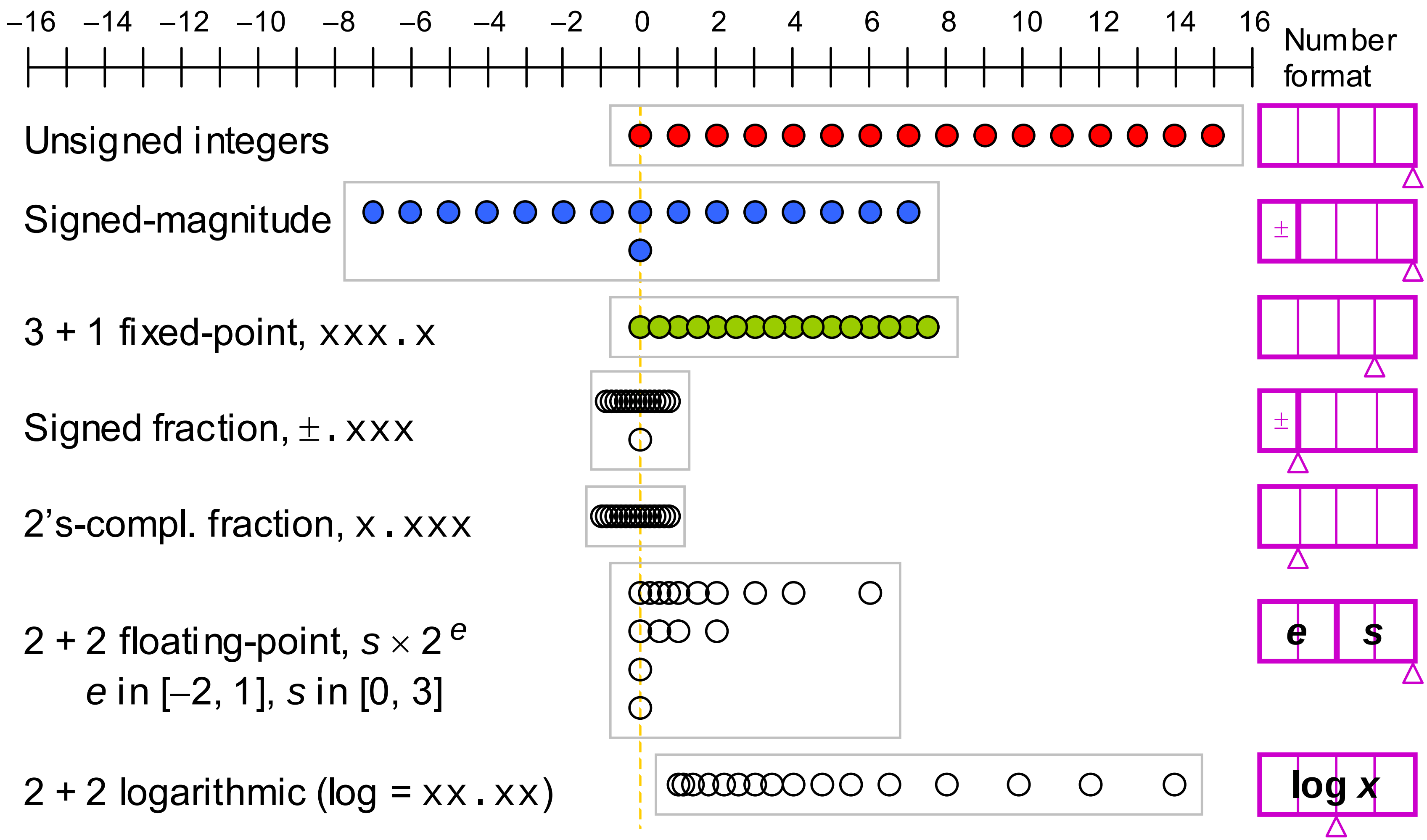


Fig. 1.2 Some of the possible ways of assigning 16 distinct codes to represent numbers. Small triangles denote the radix point locations.

1.4 Fixed-Radix Positional Number Systems

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

One can generalize to:

Arbitrary radix (not necessarily integer, positive, constant)

Arbitrary digit set, usually $\{-\alpha, -\alpha+1, \ldots, \beta-1, \beta\} = [-\alpha, \beta]$

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

More Examples of Number Systems

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

Whole part   Fractional part

$$
\begin{aligned}
u &= w \,.\, v \\
&= (x_{k-1}x_{k-2} \ldots x_1 x_0 \,.\, x_{-1}x_{-2} \ldots x_{-l})_r && \text{Old} \\
&= (X_{K-1}X_{K-2} \ldots X_1 X_0 \,.\, X_{-1}X_{-2} \ldots X_{-L})_R && \text{New}
\end{aligned}
$$

Example: $(31)_{\text{eight}} = (25)_{\text{ten}}$

Radix conversion, using arithmetic in the old radix r

Convenient when converting from $r = 10$

Radix conversion, using arithmetic in the new radix R

Convenient when converting to $R = 10$

Radix Conversion: Old-Radix Arithmetic

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

Repeatedly divide by five

Quotient	Remainder
105	0
21	1
4	4
0	

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

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

Repeatedly multiply by five

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

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

Radix Conversion: New-Radix Arithmetic

Converting whole part *w*: $(22033)_{\text{five}} = (?)_{\text{ten}}$

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

Horner's rule or formula

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

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

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

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

Horner's rule is also applicable: Proceed from right to left and use division instead of multiplication

Horner's Rule for Fractions

Converting fractional part v: $(0.22033)_{five} = (?)_{ten}$

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

Horner's rule or formula

Fig. 1.3 Horner's rule used to convert $(0.220\ 33)_{five}$ 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, including carry/borrow-save: Chapter 3
  - (but the key point of Chapter 3 is using 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
- Part V deals with real arithmetic

Real numbers, exact: Chapter 20
- Continued-fraction, slash, . . .

For the most part you need:

– 2's complement numbers
– Carry-save representation
– IEEE floating-point format

However, knowing the rest of the material (including RNS) provides you with more options when designing custom and special-purpose hardware systems

Dot Notation: A Useful Visualization Tool

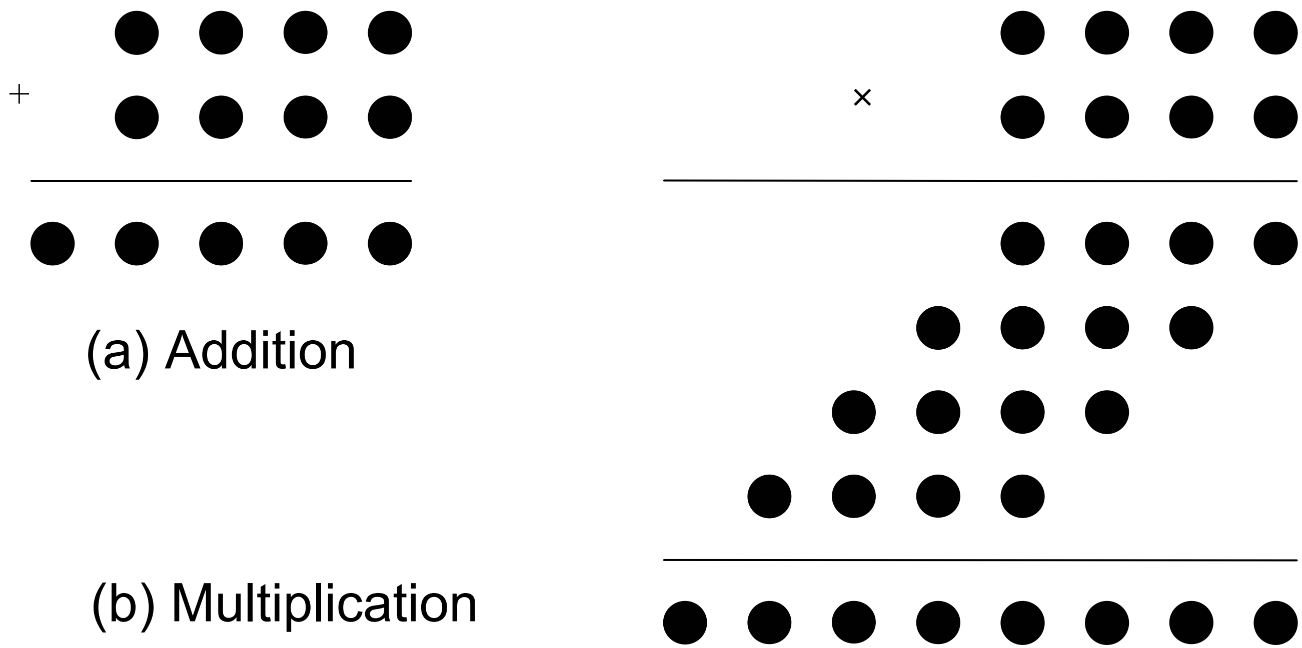


Fig. 1.4 Dot notation to depict number representation formats and arithmetic algorithms.

2 Representing Signed Numbers

Chapter Goals

Learn different encodings of the sign info
Discuss implications for arithmetic design

Chapter Highlights

Using sign bit, biasing, complementation
Properties of 2's-complement numbers
Signed vs unsigned arithmetic
Signed numbers, positions, or digits
Extended dot notation: posibits and negabits

Representing Signed Numbers: Topics

Topics in This Chapter
2.1 Signed-Magnitude Representation
2.2 Biased Representations
2.3 Complement Representations
2.4 2's- and 1's-Complement Numbers
2.5 Direct and Indirect Signed Arithmetic
2.6 Using Signed Positions or Signed Digits

2.1 Signed-Magnitude Representation

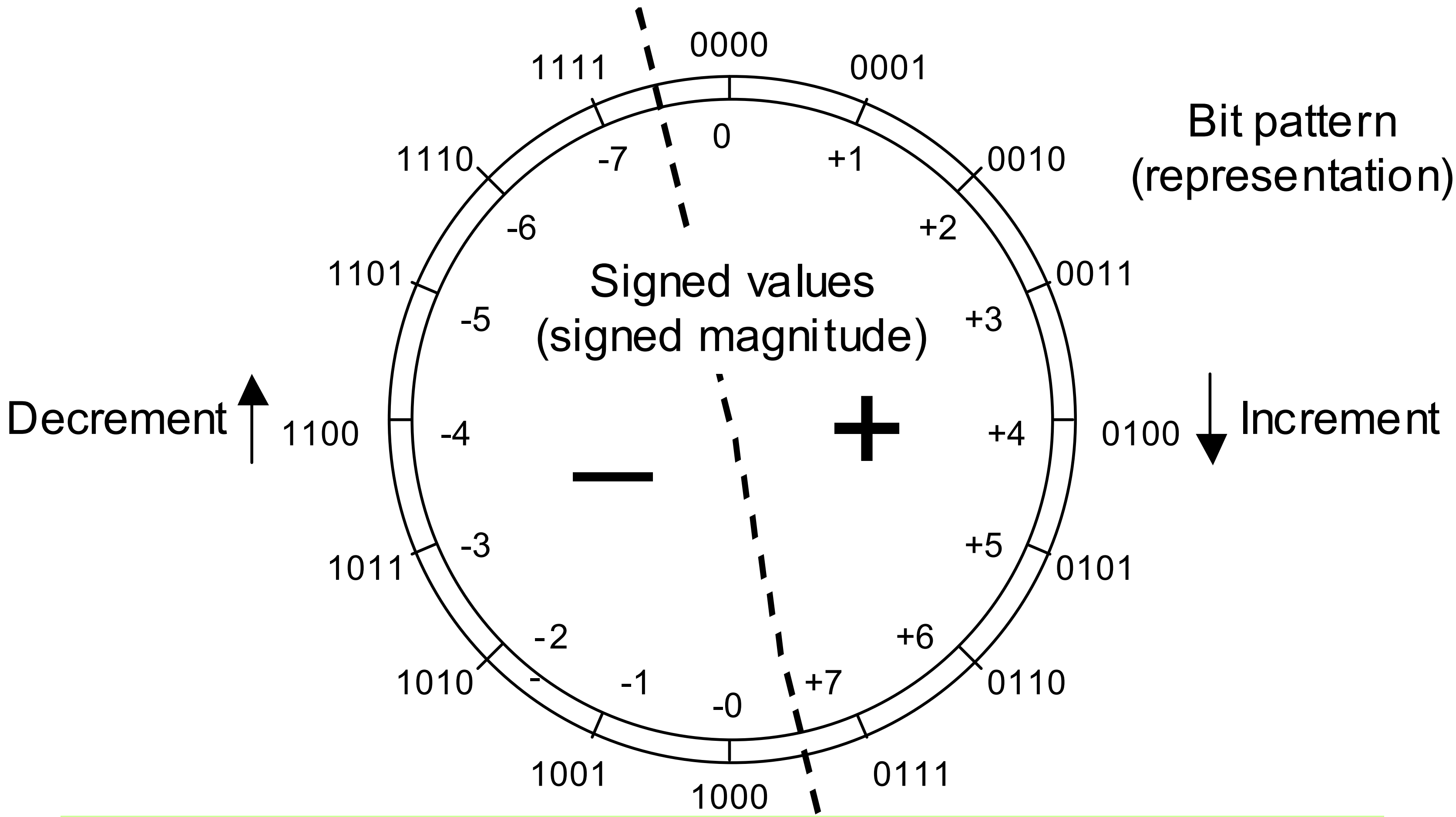


Fig. 2.1 A 4-bit signed-magnitude number representation system for integers.

Signed-Magnitude Adder

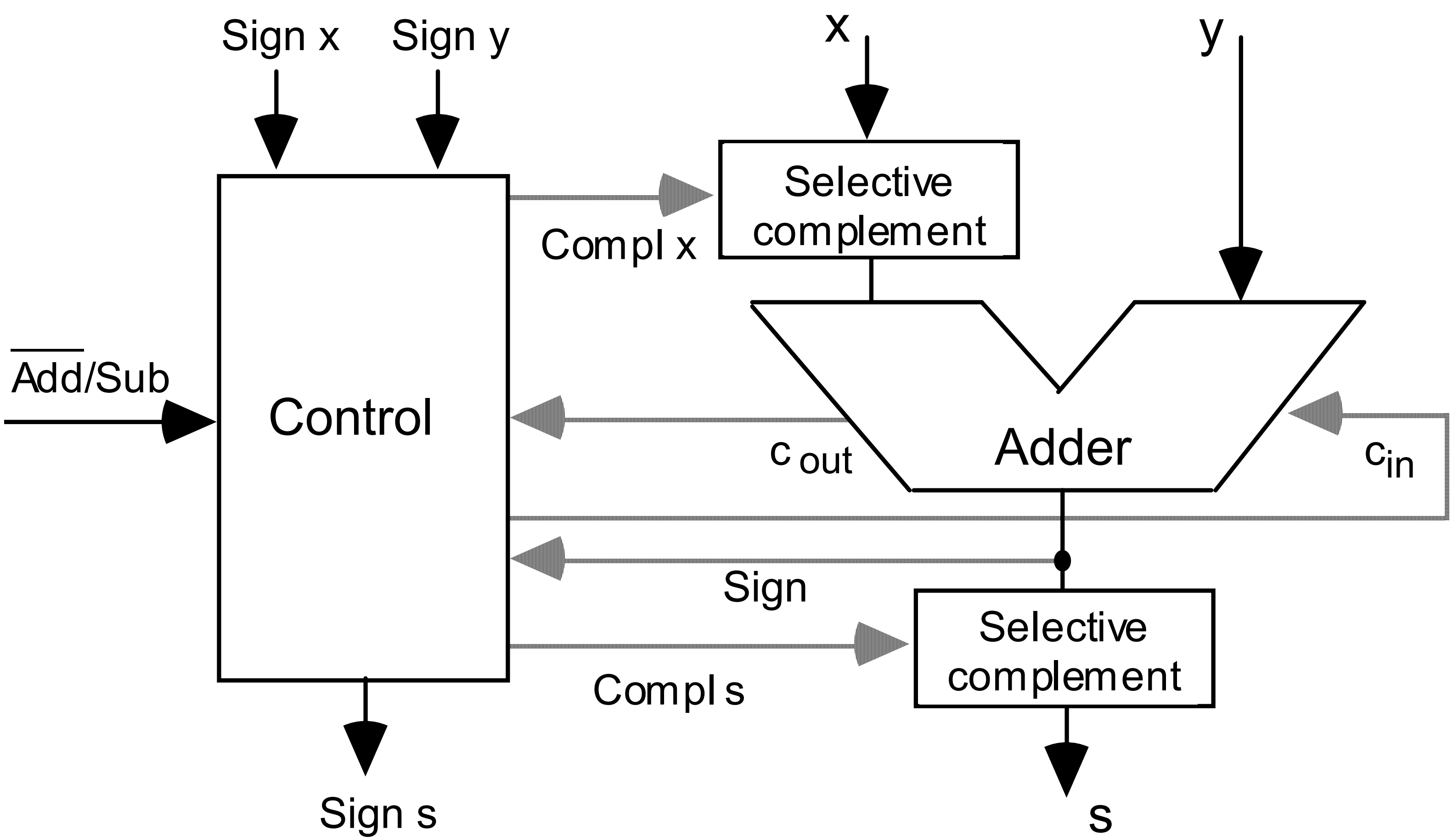


Fig. 2.2 Adding signed-magnitude numbers using precomplementation and postcomplementation.

2.2 Biased Representations

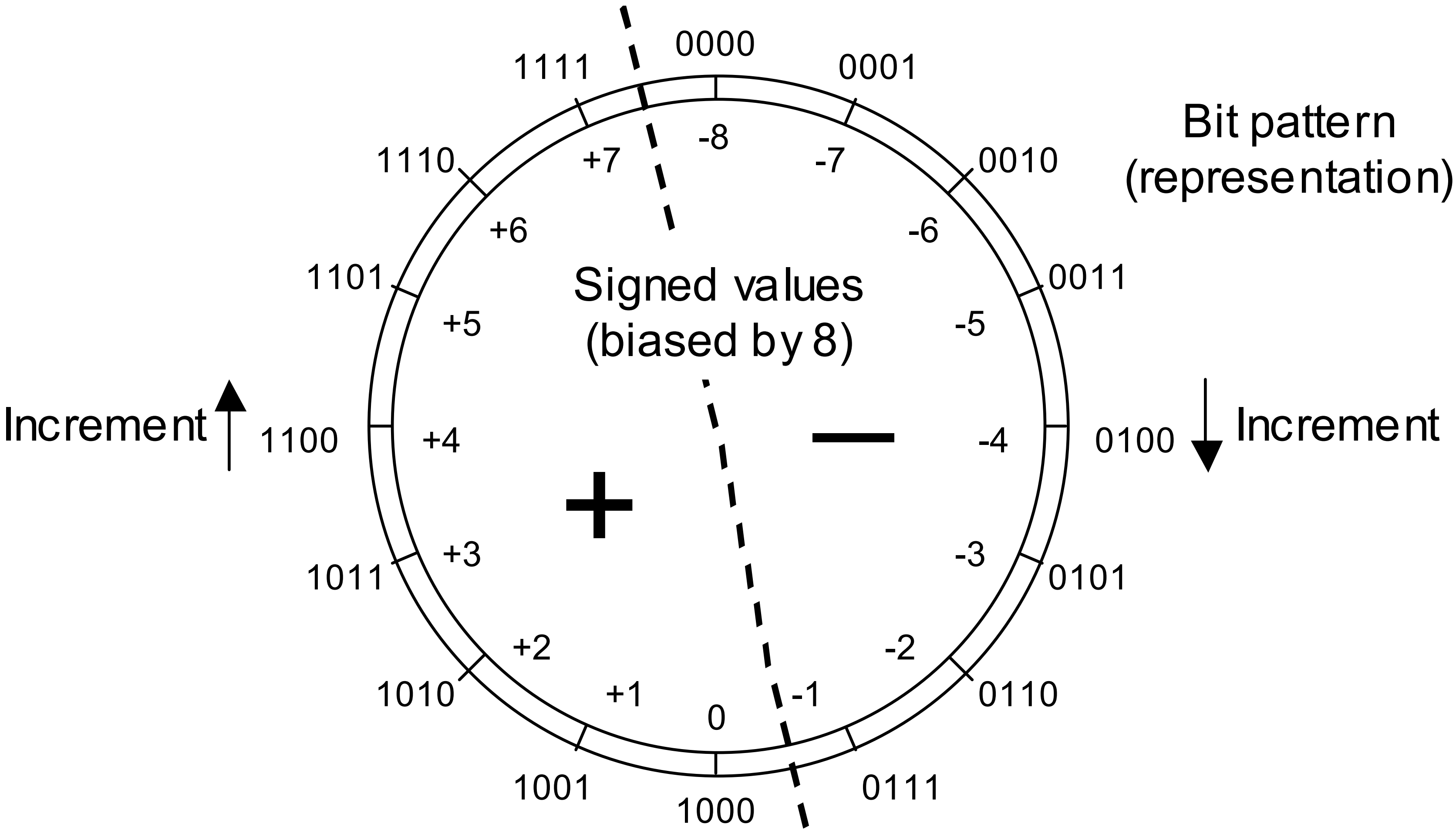


Fig. 2.3 A 4-bit biased integer number representation system with a bias of 8.

Arithmetic with Biased Numbers

Addition/subtraction of biased numbers

$$x + y + bias = (x + bias) + (y + bias) - bias$$
$$x - y + bias = (x + bias) - (y + bias) + bias$$

A power-of-2 (or $2^a - 1$) bias simplifies addition/subtraction

Comparison of biased numbers:

Compare like ordinary unsigned numbers
find true difference by ordinary subtraction

We seldom perform arbitrary arithmetic on biased numbers
Main application: Exponent field of floating-point numbers

2.3 Complement Representations

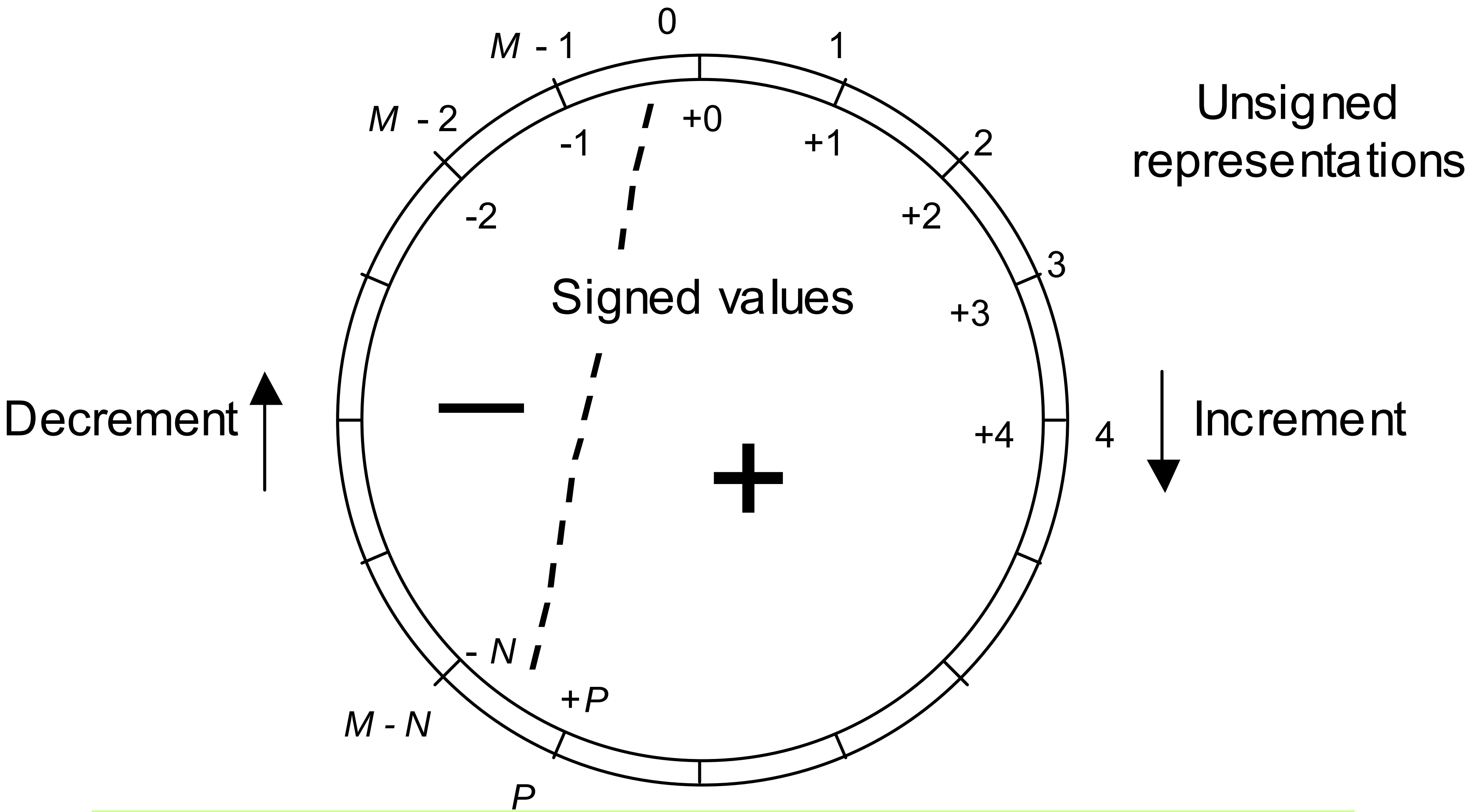


Fig. 2.4 Complement representation of signed integers.

Arithmetic with Complement Representations

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 and Two Special Cases

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

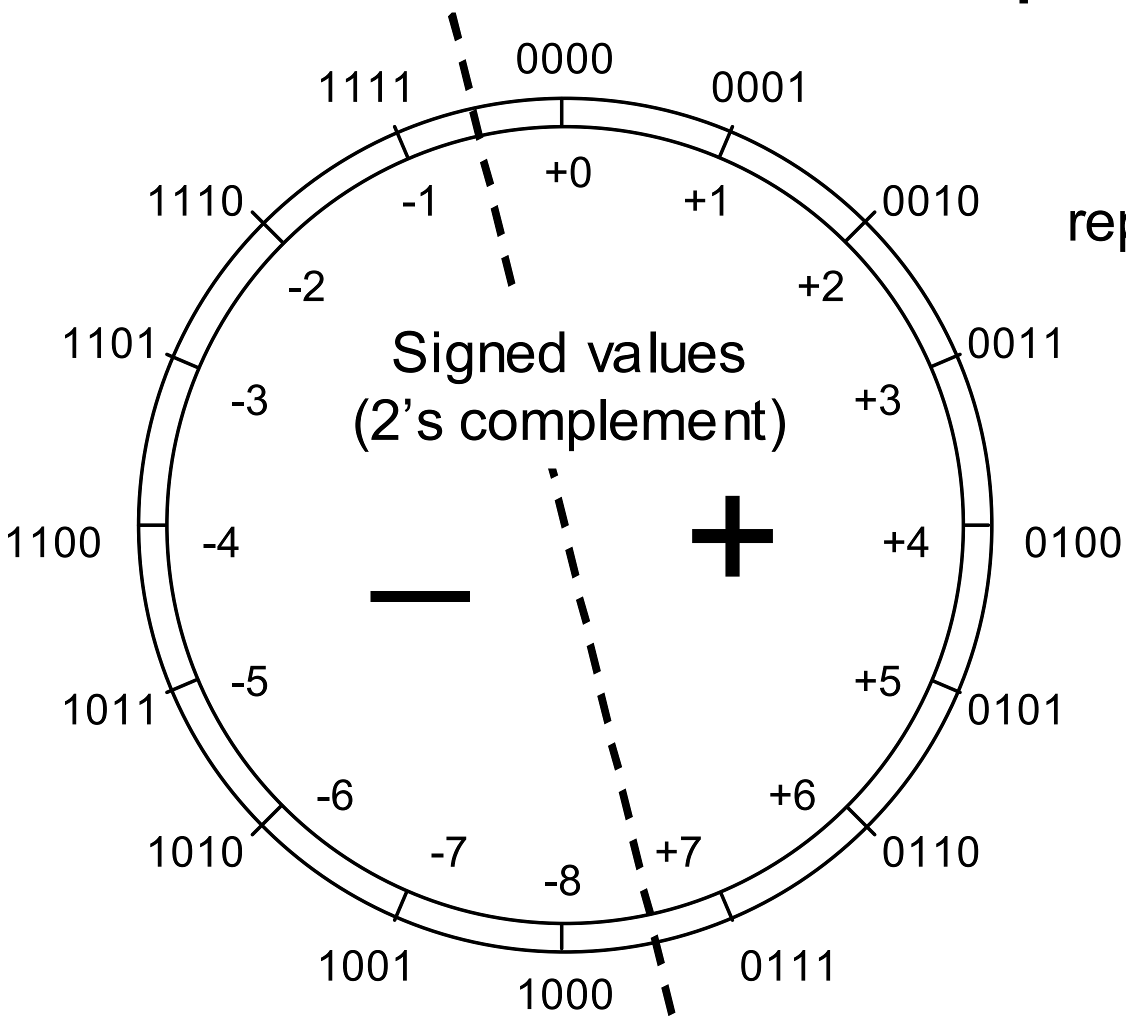
Two choices allow just this for fixed-point radix-r arithmetic with k whole digits and l fractional digits

Radix complement $M = r^k$

Digit complement $M = r^k - ulp$ (aka diminished radix compl)

ulp (unit in least position) stands for r^{-l}
Allows us to forget about l, even for nonintegers

2.4 2's- and 1's-Complement Numbers



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

$M = 2^k$

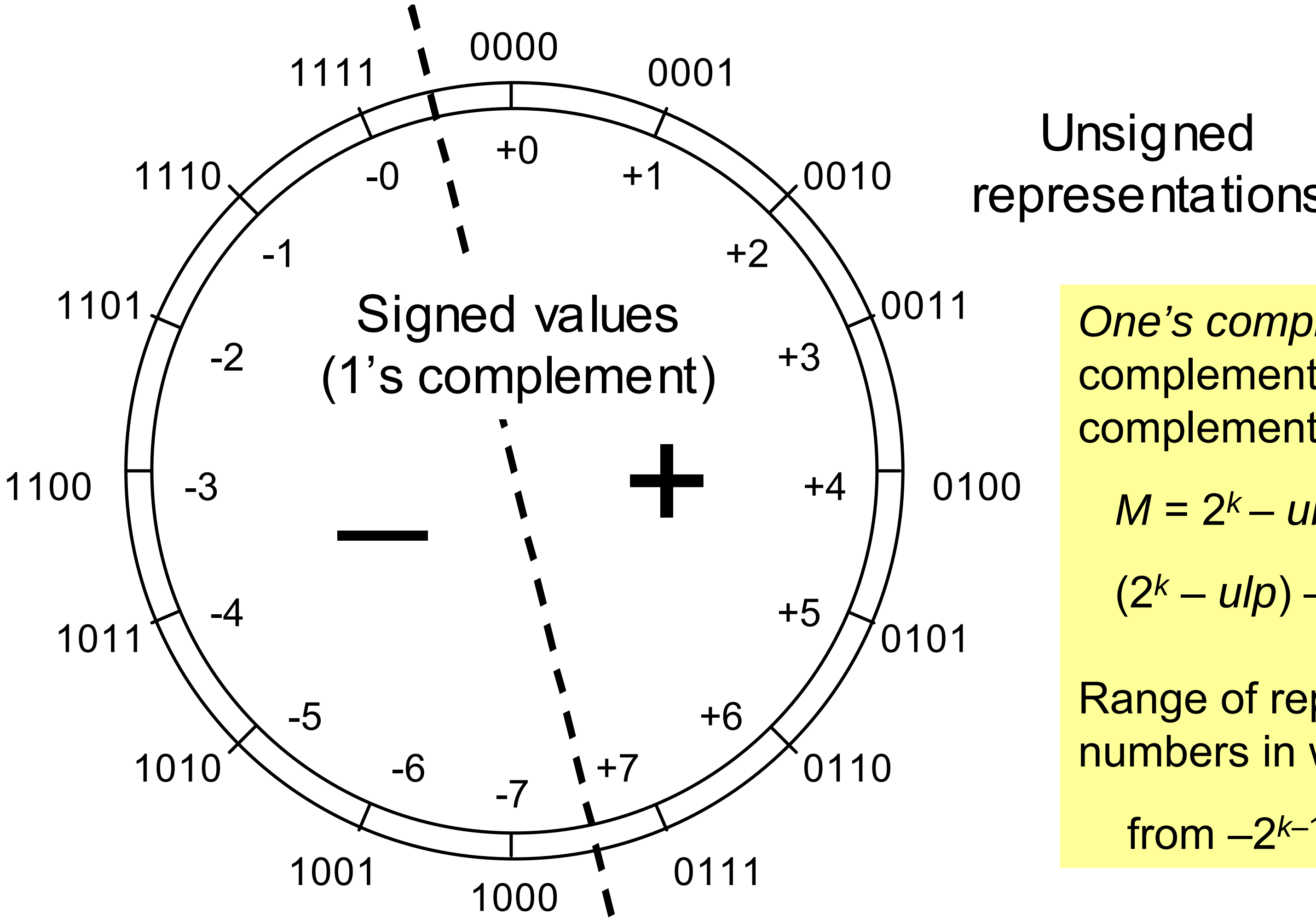
$$2^k - x = [(2^k - ulp) - x] + ulp$$
$$= x^{compl} + ulp$$

Range of representable numbers in with k whole bits:

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

Fig. 2.5 A 4-bit 2's-complement number representation system for integers.

1's-Complement Number Representation



One's complement = digit complement (diminished radix complement) system for $r = 2$

$M = 2^k - ulp$

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

Range of representable numbers in with k whole bits:

from $-2^{k-1} + ulp$ to $2^{k-1} - ulp$

Fig. 2.6 A 4-bit 1's-complement number representation system for integers.

Some Details for 2's- and 1's Complement

Range/precision extension for 2's-complement numbers

$$. . . x_{k-1}\, x_{k-1}\, x_{k-1}\, \boxed{x_{k-1}\, x_{k-2}\, . . .\, x_1\, x_0\, .\, x_{-1}\, x_{-2}\, . . .\, x_{-l}}\, 0\ \ 0\ \ 0\ . . .$$

← Sign extension → Sign bit LSD ← Extension →

Range/precision extension for 1's-complement numbers

$$. . . x_{k-1}\, x_{k-1}\, x_{k-1}\, \boxed{x_{k-1}\, x_{k-2}\, . . .\, x_1\, x_0\, .\, x_{-1}\, x_{-2}\, . . .\, x_{-l}}\, x_{k-1}\, x_{k-1}\, x_{k-1}\ . . .$$

← Sign extension → Sign bit LSD ← Extension →

Mod-2^k operation needed in 2's-complement arithmetic is trivial:
Simply drop the carry-out (subtract 2^k if result is 2^k or greater)

Mod-(2^k – *ulp*) operation needed in 1's-complement arithmetic is done via end-around carry

$(x + y) - (2^k - ulp) = (x - y - 2^k) + ulp$ Connect c_{out} to c_{in}

Which Complement System Is Better?

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, there are two 0s
Complementation	Complement all digits and add *ulp*	Complement all digits
Mod-M addition	Drop the carry-out	End-around carry

Why 2's-Complement Is the Universal Choice

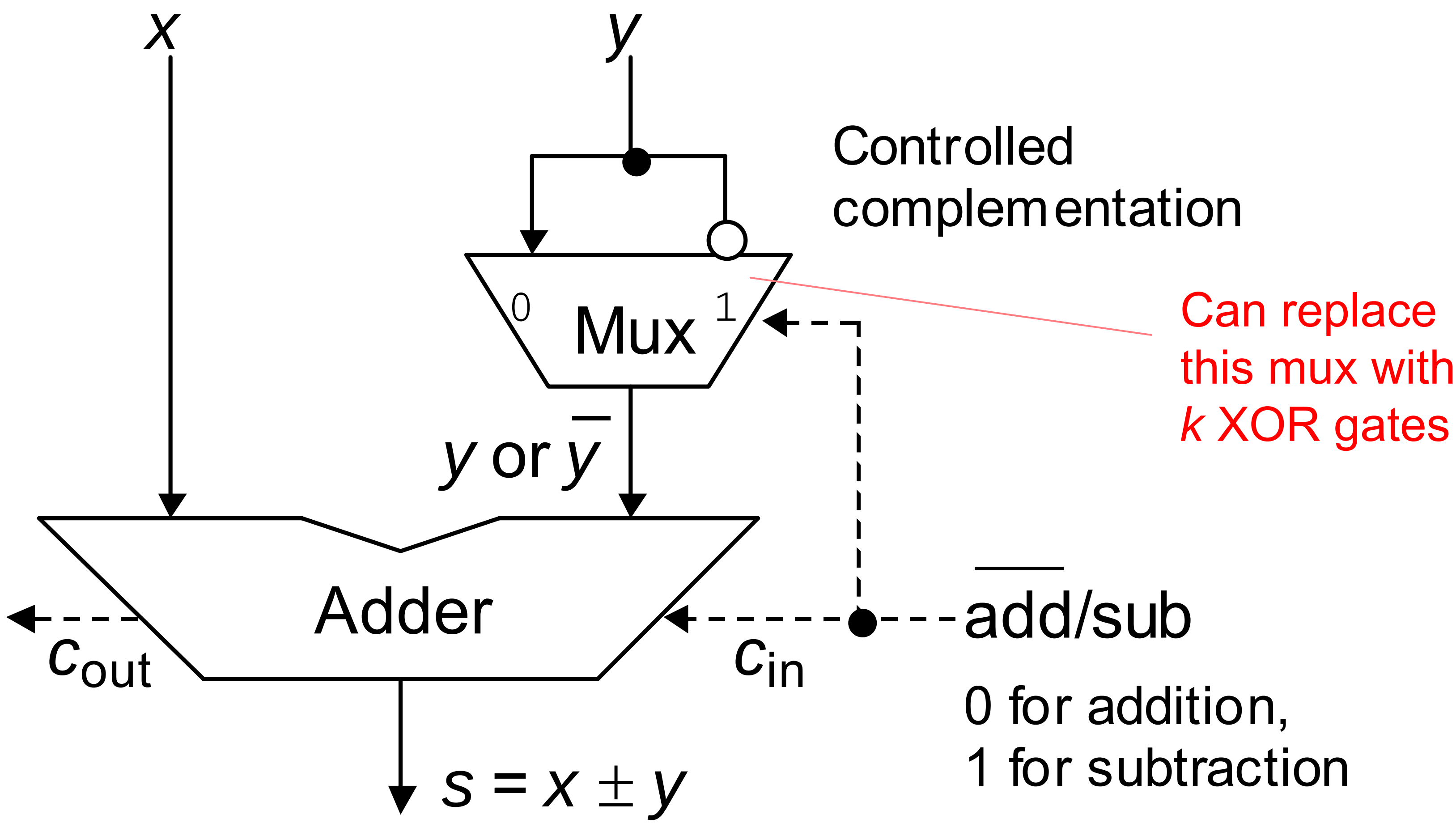


Fig. 2.7 Adder/subtractor architecture for 2's-complement numbers.

Signed-Magnitude vs 2's-Complement

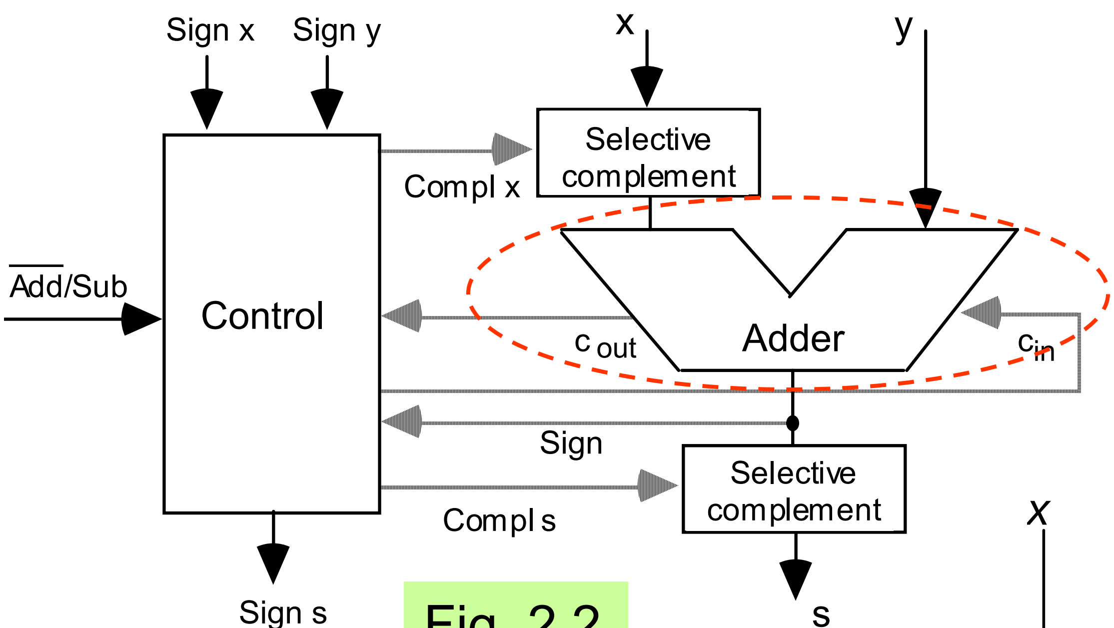


Fig. 2.2

Signed-magnitude adder/subtractor is significantly more complex than a simple adder

2's-complement adder/subtractor needs very little hardware other than a simple adder

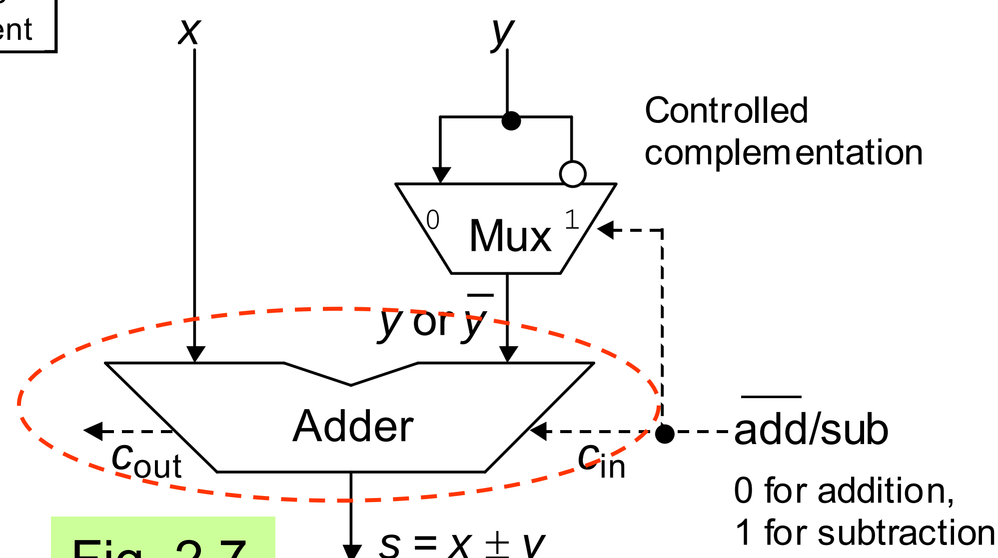


Fig. 2.7

2.5 Direct and Indirect Signed Arithmetic

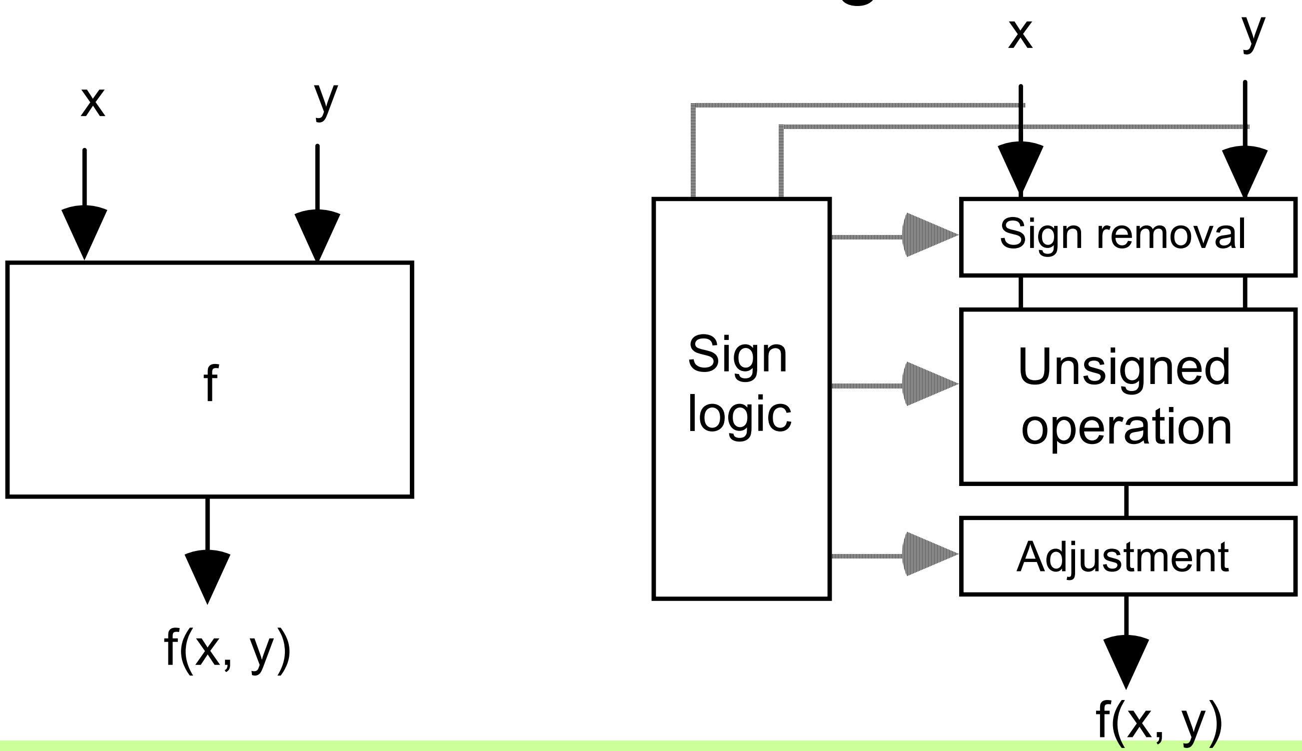


Fig. 2.8 Direct versus indirect operation on signed numbers.

Direct signed arithmetic is usually faster (not always)

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 key property of 2's-complement numbers that facilitates direct signed arithmetic:

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.

Associating a Sign with Each Digit

Signed-digit representation: Digit set [$-\alpha$, β] instead of [0, r – 1]

Example: Radix-4 representation with digit set [–1, 2] rather than [0, 3]

	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 nonstandard digit set [–1, 2].

Redundant Signed-Digit Representations

Signed-digit representation: Digit set $[-\alpha, \beta]$, with $\rho = \alpha + \beta + 1 - r > 0$

Example: Radix-4 representation with digit set [–2, 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 nonstandard digit set [–2, 2].

Here, the transfer does not propagate, so conversion is “carry-free”

Extended Dot Notation: Posibits and Negabits

Posibit, or simply bit: positively weighted
Negabit: negatively weighted

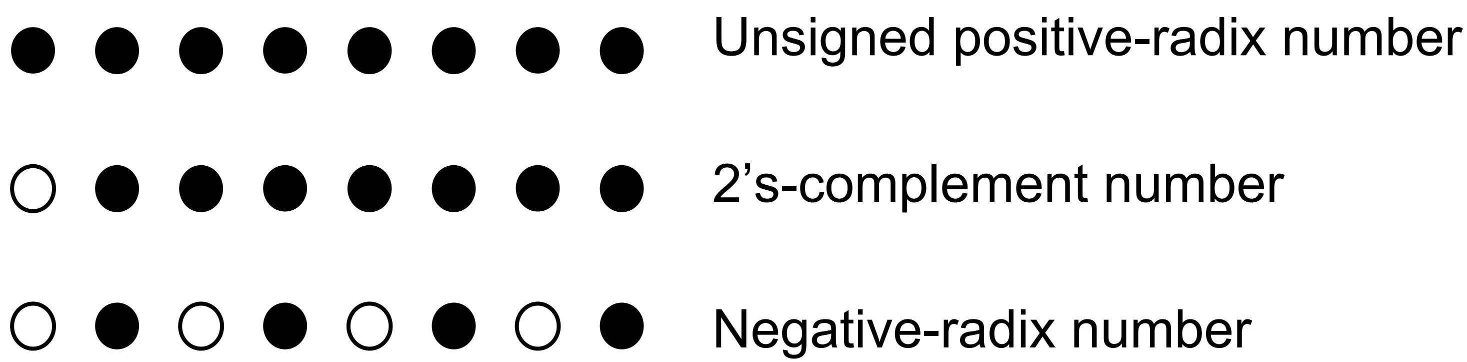


Fig. 2.12 Extended dot notation depicting various number representation formats.

Extended Dot Notation in Use

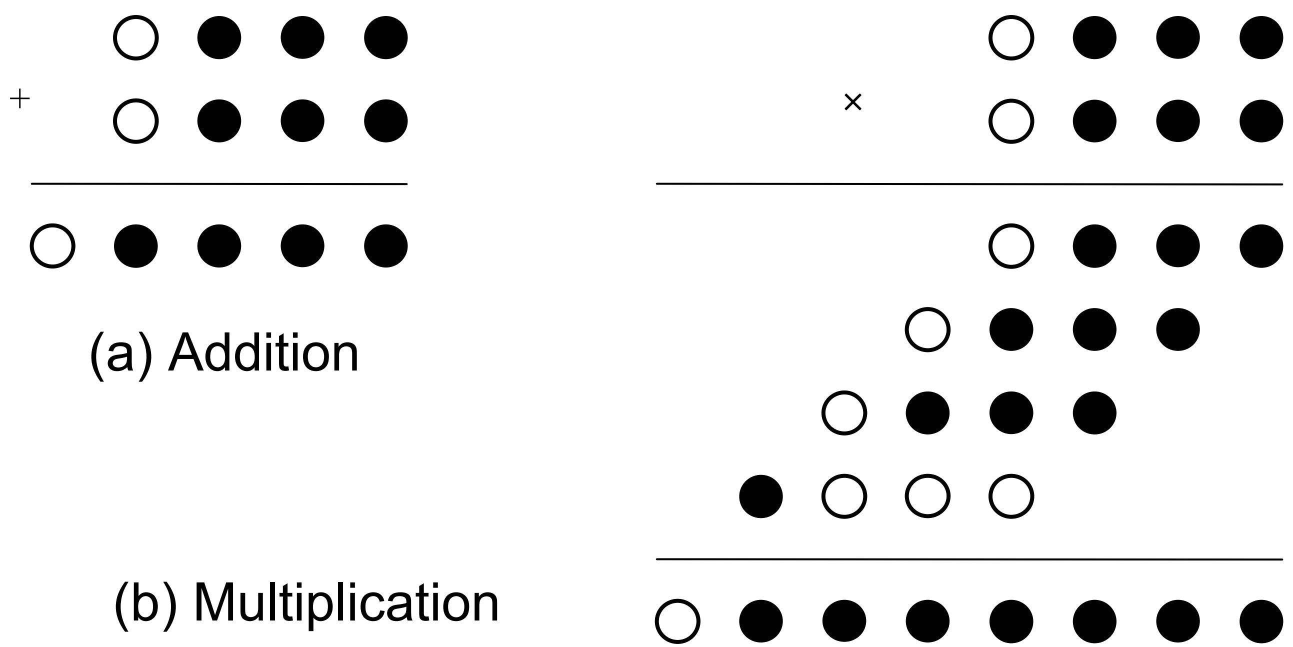


Fig. 2.13 Example arithmetic algorithms represented in extended dot notation.

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: trade-offs
Redundant/nonredundant conversions
Redundancy used for end values too?
Extended dot notation with redundancy

Redundant Number Systems: Topics

Topics in This Chapter
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

Ways of dealing with the carry propagation problem:

1. Limit propagation to within a small number of bits (Chapters 3-4)
2. Detect end of propagation; don’t wait for worst case (Chapter 5)
3. Speed up propagation via lookahead etc. (Chapters 6-7)
4. Ideal: Eliminate carry propagation altogether! (Chapter 3)

	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?

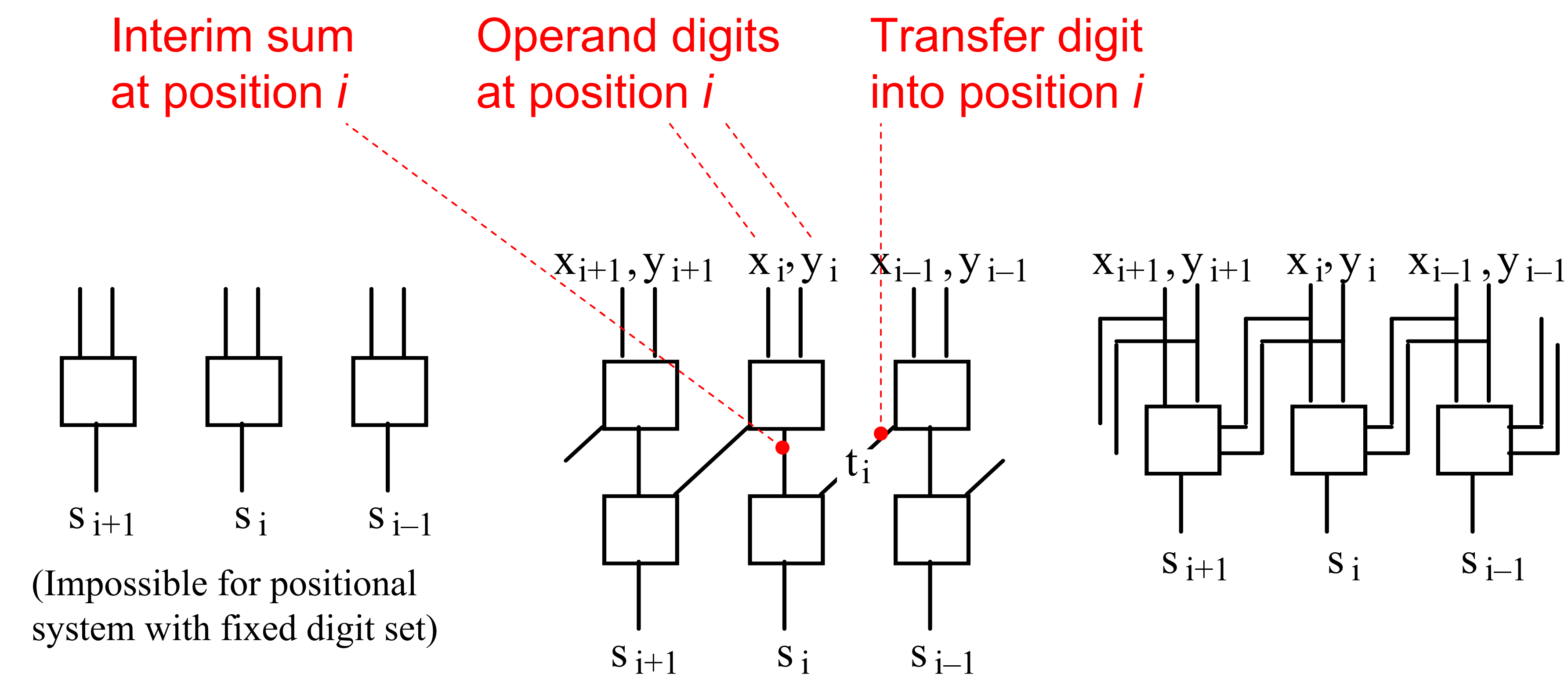
Addition of Redundant Numbers

Position sum decomposition	[0, 36]	$= 10 \times [0, 2] + [0, 16]$
Absorption of transfer digit	[0, 16]	$+ \; [0, 2] \; = \; [0, 18]$

	11	9	17	10	12	18	Operand digits in [0, 18]
+	6	12	9	10	8	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].

Meaning of Carry-Free Addition



(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.

Redundancy Index

So, redundancy helps us achieve carry-free addition $-\alpha$ β

But how much redundancy is actually needed? Is [0, 11] enough for r = 10?

Redundancy index $\rho = \alpha + \beta + 1 - r$ For example, 0 + 11 + 1 – 10 = 2

	11	10	7	11	3	8	Operand digits in [0, 11]
+	7	2	9	10	9	8	
	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].

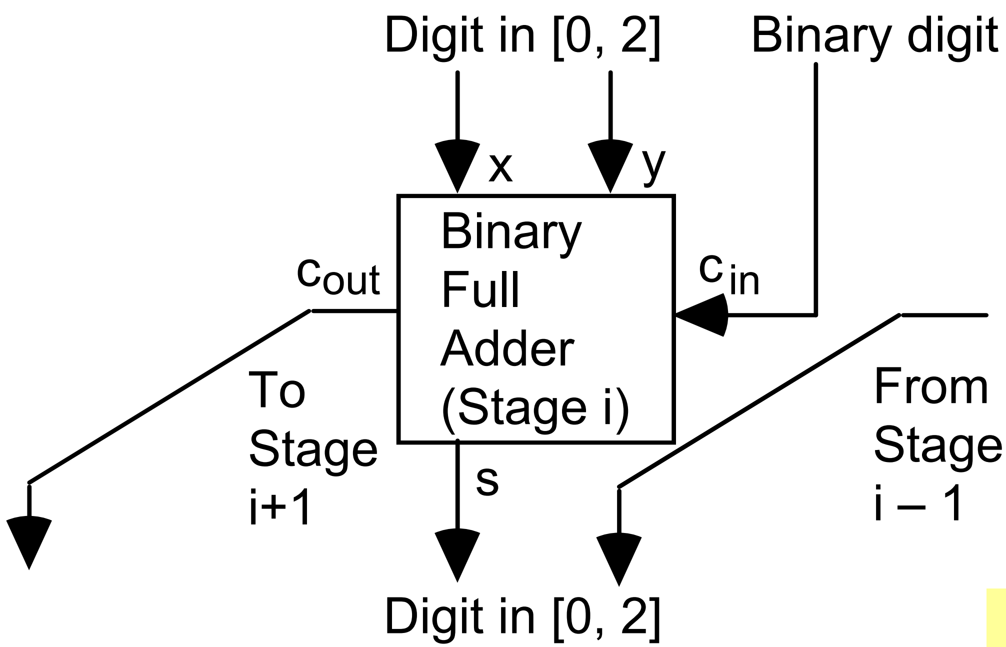
3.2 Redundancy in Computer Arithmetic

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 four binary numbers, with the sum obtained in stored-carry form.

Hardware for Carry-Save Addition



Two-bit encoding for binary stored-carry digits used in this implementation:

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

Fig. 3.5 Using an array of independent binary full adders to perform carry-save addition.

Because in carry-save addition, three binary numbers are reduced to two binary numbers, this process is sometimes referred to as 3-2 compression

Carry-Save Addition in Dot Notation

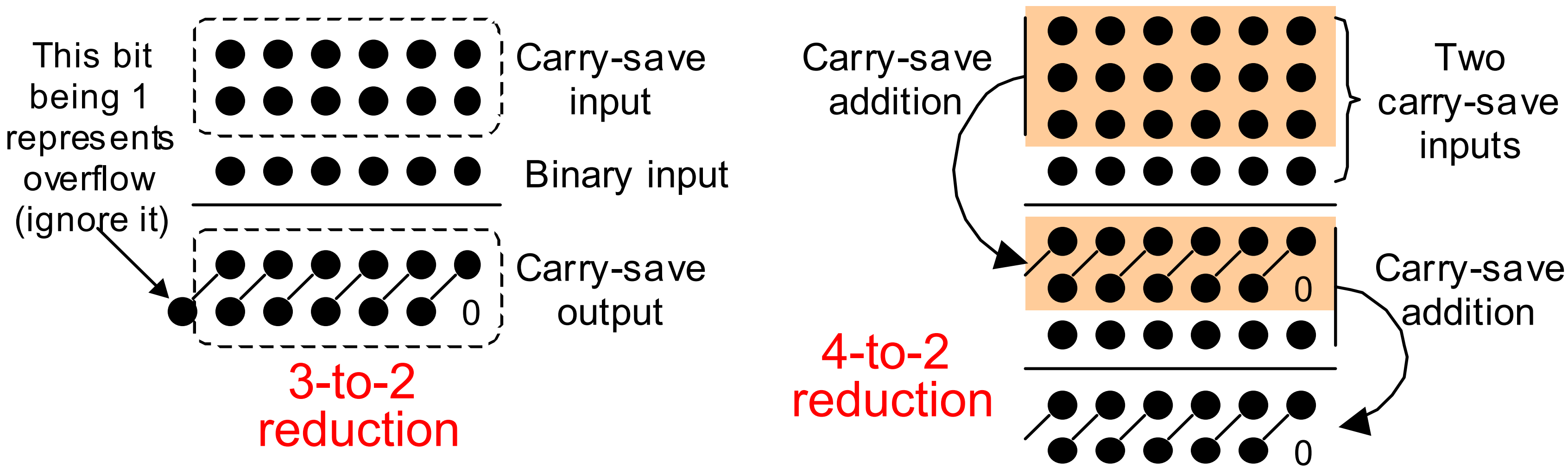


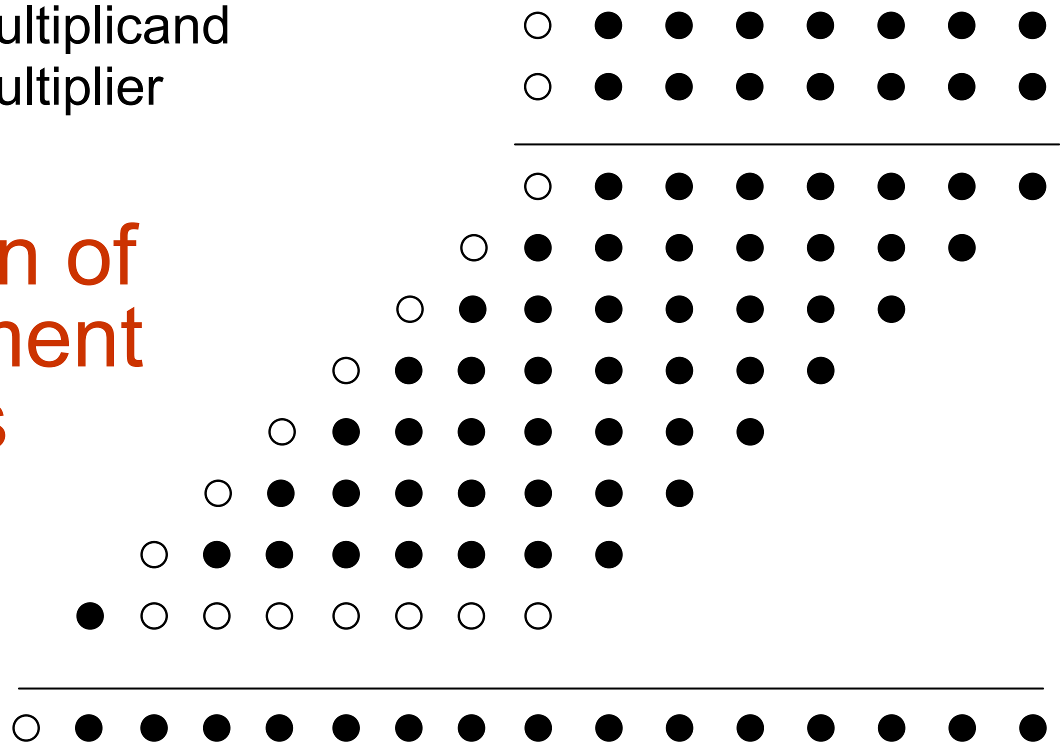
Fig. 9.3 From text on computer architecture (Parhami, Oxford/2005)

We sometimes find it convenient to use an extended dot notation, with heavy dots (●) for posibits and hollow dots (○) for negabits

Eight-bit, 2's-complement number	○ ● ● ● ● ● ● ●
Negative-radix number	○ ● ○ ● ○ ● ○ ●
BSD number with ⟨*n*, *p*⟩ encoding of the digit set [–1, 1]	○ ○ ○ ○ ○ ○ ○ ○ ● ● ● ● ● ● ● ●

Example for the Use of Extended Dot Notation

2’s-complement multiplicand
2’s-complement multiplier



Multiplication of 2’s-complement numbers

Option 1:
sign extension

$$\begin{array}{ccccc} & & & & -x \\ -x & x & x & x & x \end{array}$$

Option 2: Baugh-Wooley method

$$\begin{array}{c} -x \\ -y \end{array} \Rightarrow \begin{array}{cc} & 1-x \\ -1 & 1-y \end{array}$$

3.3 Digit Sets and Digit-Set Conversions

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

	11	9	17	10	12	18	18 = 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]

Note: Conversion from redundant to nonredundant representation always involves carry propagation

Thus, the process is sequential and slow

Conversion from Carry-Save to Binary

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

	1	1	2	0	2	0	2 = 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	1st number: sum bits
+	0	0	1	0	1	0	2nd number: carry bits
1	0	0	0	1	0	0	Sum

Conversion Between Redundant Digit Sets

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

	11	9	17	10	12	18	18 = 20 (carry 2) – 2
	11	9	17	10	14	–2	14 = 10 (carry 1) + 4
	11	9	17	11	4	–2	11 = 10 (carry 1) + 1
	11	9	18	1	4	–2	18 = 20 (carry 2) – 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 [–6, 5]

On line 2, we could have written 14 = 20 (carry 2) – 6; this would have led to a different, but equivalent, representation

In general, several representations may exist for a redundant digit set

Carry-Free Conversion to a Redundant Digit Set

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

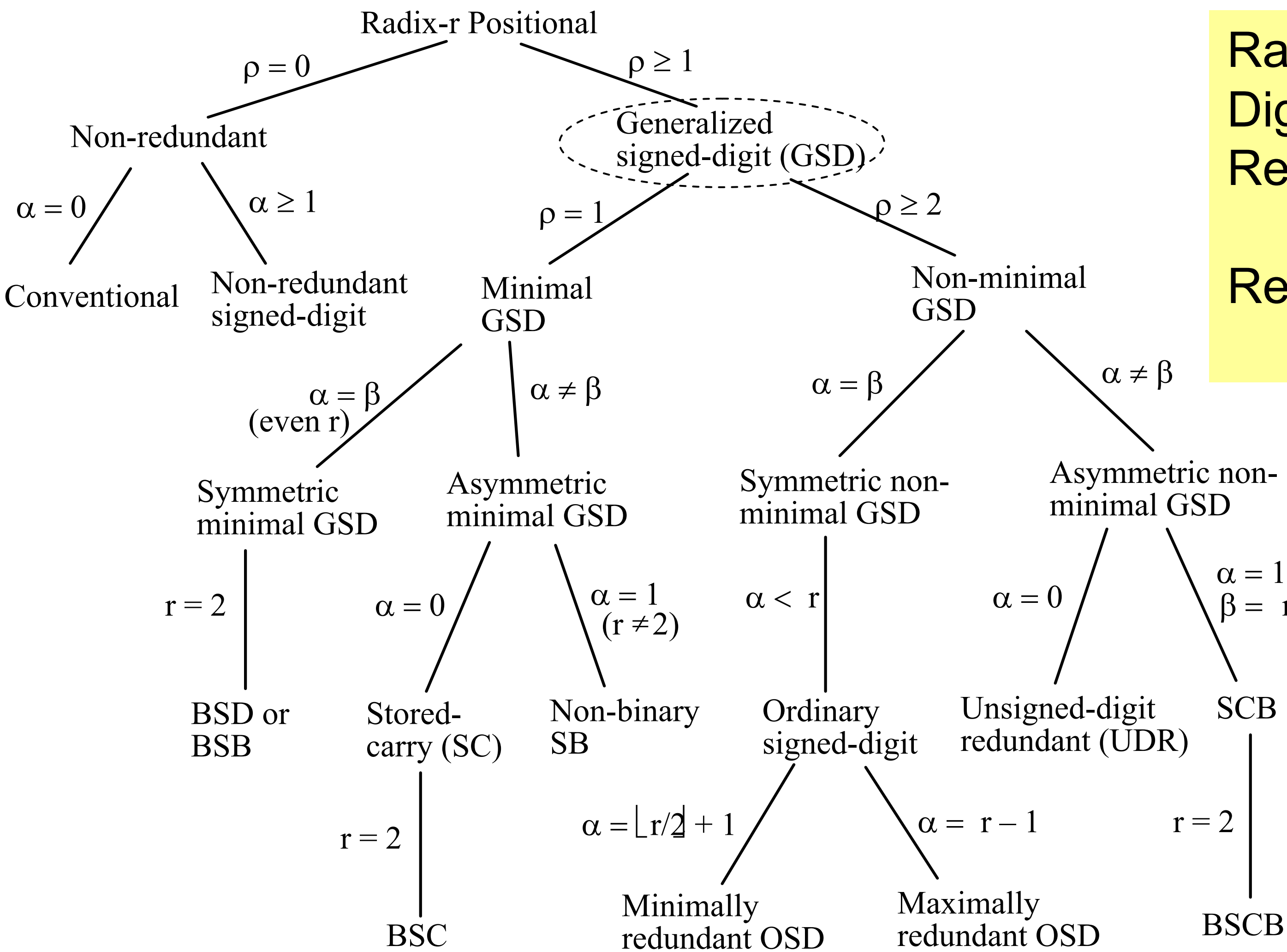
Carry-free conversion:

	1	1	2	0	2	0	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 [–1, 1]

We rewrite 2 as 2 (carry 1) + 0, and 1 as 2 (carry 1) – 1

A carry of 1 is always absorbed by the interim digit that is in {–1, 0}

3.4 Generalized Signed-Digit Numbers



Radix r

Digit set $[-\alpha, \beta]$

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.

Encodings for Signed Digits

x_i	1	$^{-}1$	0	$^{-}1$	0	BSD representation of +6
$\langle s, v \rangle$	01	11	00	11	00	Sign and value encoding
2’s-compl	01	11	00	11	00	2-bit 2’s-complement
$\langle n, p \rangle$	01	10	00	10	00	Negative & positive flags
$\langle n, z, p \rangle$	001	100	010	100	010	1-out-of-3 encoding

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

Two of the encodings above can be shown in extended dot notation

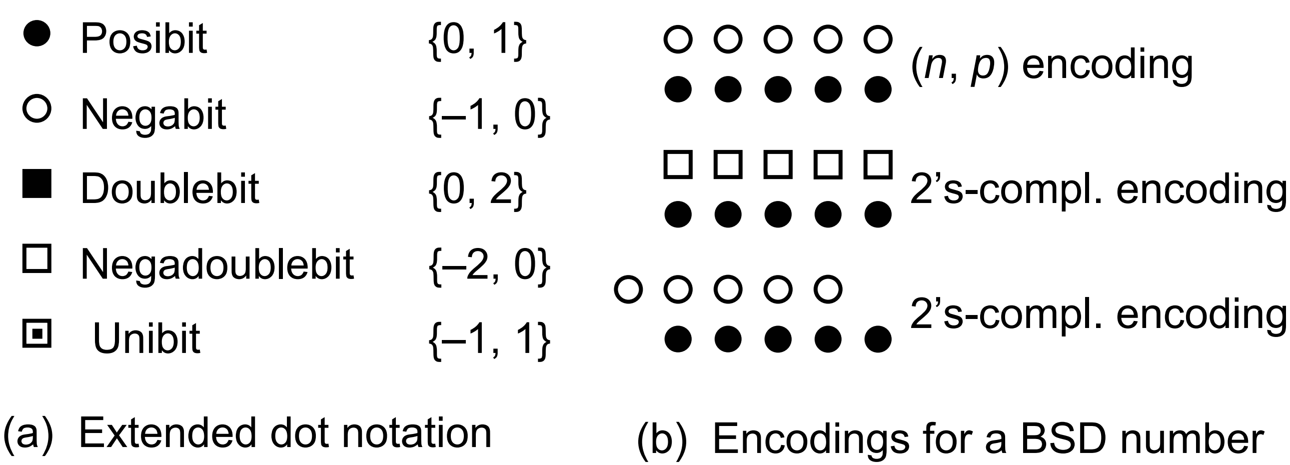


Fig. 3.8 Extended dot notation and its use in visualizing some BSD encodings.

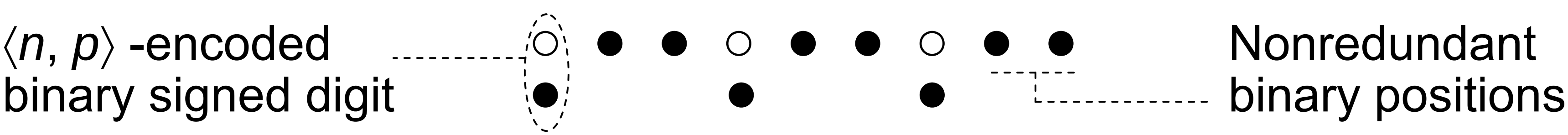
Hybrid Signed-Digit Numbers

	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				t_{i+1}
1	–1	1	1	0	1	1	–1	1	1	s_i

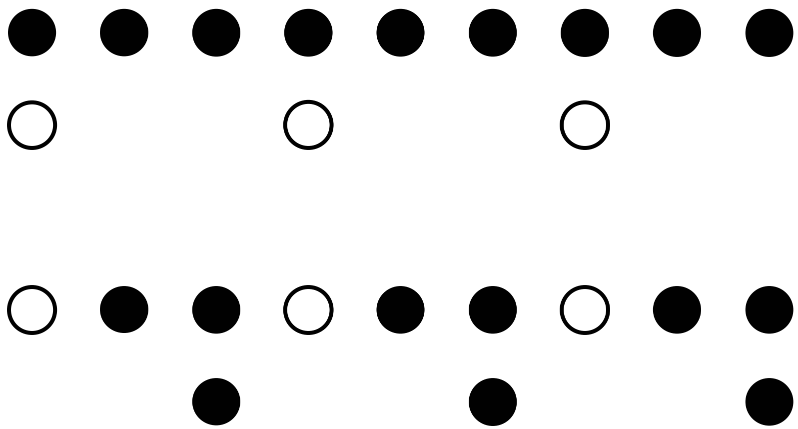
Radix-8 GSD with digit set [–4,7]

Fig. 3.9 Example of addition for hybrid signed-digit numbers.

The hybrid-redundant representation above in extended dot notation:



Hybrid Redundancy in Extended Dot Notation



Radix-8 digit set [–4, 7]

Radix-8 digit set [–4, 4]

Fig. 3.10 Two hybrid-redundant representations in extended dot notation.

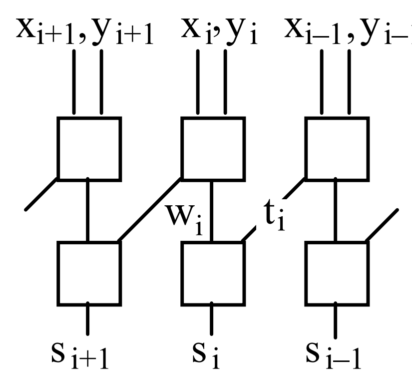
3.5 Carry-Free Addition Algorithms

<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 interim sum $w_i = p_i - rt_{i+1}$

Add 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:

$$-\alpha + \lambda \quad \leq \quad p_i - rt_{i+1} \quad \leq \quad \beta - \mu$$

interim sum

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:

$\lambda \geq \alpha / (r - 1)$

$\mu \geq \beta / (r - 1)$

Is Carry-Free Addition Always Applicable?

No: It requires one of the following two conditions

a. $r > 2$, $\rho \geq 3$

b. $r > 2$, $\rho = 2$, $\alpha \neq 1$, $\beta \neq 1$ — e.g., not [–1, 10] in radix 10

In other words, it is inapplicable for

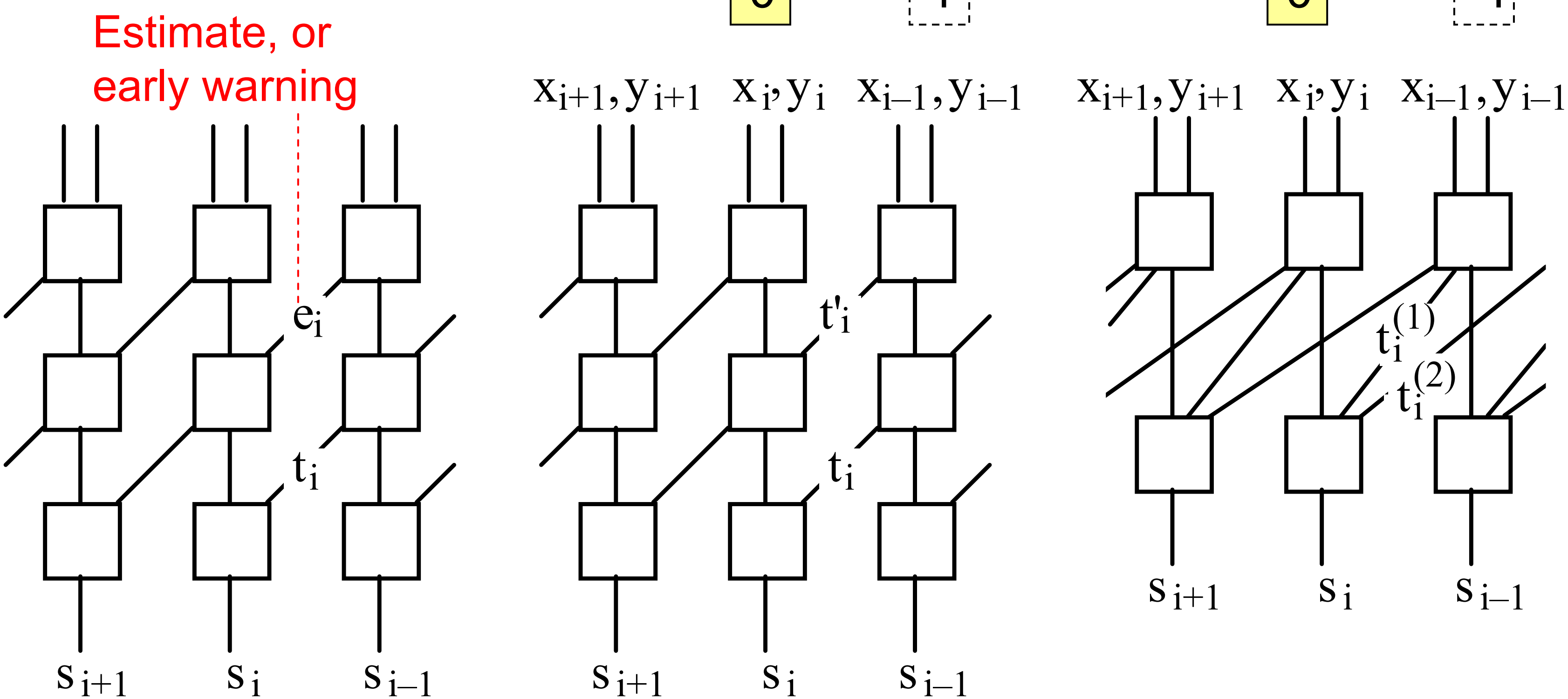
$r = 2$	Perhaps most useful case
$\rho = 1$	e.g., carry-save
$\rho = 2$ with $\alpha = 1$ or $\beta = 1$	e.g., carry/borrow-save

BSD fails on at least two criteria!

Fortunately, in the latter cases, a limited-carry addition algorithm is always applicable

Limited-Carry Addition

Example: BSD addition



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

Fig. 3.12 Some implementations for limited-carry addition.

Limited-Carry BSD 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	low	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. 3.13 Limited-carry addition of radix-2 numbers with digit set [–1, 1] using carry estimates. A position sum –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 $-1 \le s_i \le 1$.

3.6 Conversions and Support Functions

Example 3.10: Conversion from/to BSD to/from standard binary

1	–1	0	–1	0	BSD representation of +6
1	0	0	0	0	Positive part
0	1	0	1	0	Negative part
0	0	1	1	0	Difference = Conversion result

The negative and positive parts above are particularly easy to obtain if the BSD number has the $\langle n, p \rangle$ encoding

Conversion from redundant to nonredundant representation always requires full carry propagation

Conversion from nonredundant to redundant is often trivial

Other Arithmetic Support Functions

Zero test: Zero has a unique code under some conditions

Sign test: Needs carry propagation

Overflow: May be real or apparent (result may be representable)

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

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

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

Residue Number Systems: Topics

Topics in This Chapter
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 Representations and Arithmetic

Puzzle, due to the Chinese scholar Sun Tzu, 1500^+ years ago:

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

Residues (akin to digits in positional systems) uniquely identify the number, hence they constitute a representation

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

The residue x_i of x wrt the ith modulus m_i (similar to a digit):

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

RNS representation contains a list of k residues or digits:

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

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

RNS Dynamic Range

Product M of the k pairwise relatively prime moduli is the *dynamic range*

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

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

We can take the range of RNS(8|7|5|3) to be [–420, 419] or any other set of 840 consecutive integers

Negative numbers: Complement relative to M

$$\langle -x \rangle_{m_i} = \langle M - x \rangle_{m_i}$$
$$21 = (5 \mid 0 \mid 1 \mid 0)_{\text{RNS}}$$
$$-21 = (8 - 5 \mid 0 \mid 5 - 1 \mid 0)_{\text{RNS}} = (3 \mid 0 \mid 4 \mid 0)_{\text{RNS}}$$

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

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

RNS as Weighted Representation

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

105 120 336 280

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

$$\langle 105\times 1 + 120\times 2 + 336\times 4 + 280\times 0\rangle_{840} = \langle 1689\rangle_{840} = 9$$

For RNS(7 | 5 | 3), the weights of the 3 positions are:

15 21 70

Example -- Chinese puzzle: $(2 \mid 3 \mid 2)_{RNS(7|5|3)}$ represents the number

$$\langle 15 \times 2 + 21 \times 3 + 70 \times 2\rangle_{105} = \langle 233\rangle_{105} = 23$$

We will see later how the weights can be determined for a given RNS

RNS Encoding and Arithmetic Operations

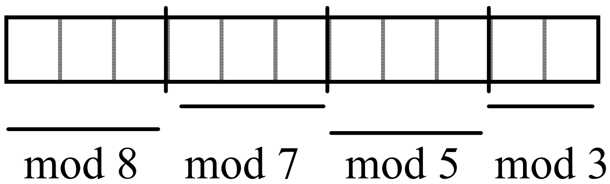


Fig. 4.1 Binary-coded format for RNS(8 | 7 | 5 | 3).

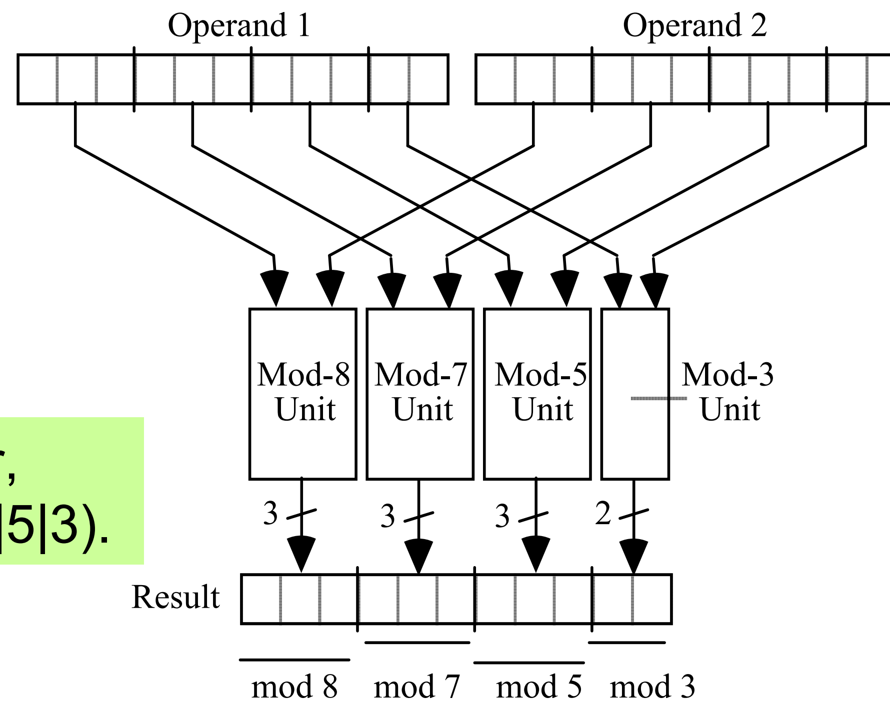


Fig. 4.2 The structure of an adder, subtractor, or multiplier for RNS(8|7|5|3).

Arithmetic in RNS(8 | 7 | 5 | 3)

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

4.2 Choosing the RNS Moduli

Target range for our RNS: Decimal values [0, 100 000]

Strategy 1: To minimize the largest modulus, and thus ensure high-speed arithmetic, pick prime numbers in sequence

Pick $m_0 = 2$, $m_1 = 3$, $m_2 = 5$, 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	

Fine tuning: Combine pairs of moduli 2 & 13 (26) and 3 & 7 (21)
RNS(26 | 21 | 17 | 11) $M = 102\ 102$

An Improved Strategy

Target range for our RNS: Decimal values [0, 100 000]

Strategy 2: Improve strategy 1 by including powers of smaller primes before proceeding to the next larger prime

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
	(remove one 3, combine 3 & 5)
RNS(15 | 13 | 11 | 2^3 | 7)	M = 120 120
	4 + 4 + 4 + 3 + 3 = 18 bits

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

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

We can now remove 5 or 7; not an improvement in this example

Low-Cost RNS Moduli

Target range for our RNS: Decimal values [0, 100 000]

Strategy 3: To simplify the modular reduction (mod m_i) operations, choose only moduli of the forms 2^a or $2^a - 1$, aka "low-cost moduli"

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

We can have only one even modulus

$2^{a_i} - 1$ and $2^{a_j} - 1$ 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($2^5 \mid 2^5-1 \mid 2^4-1 \mid 2^3-1$)	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

Low- and Moderate-Cost RNS Moduli

Target range for our RNS: Decimal values [0, 100 000]

Strategy 4: To simplify the modular reduction (mod m_i) operations, choose moduli of the forms 2^a, $2^a - 1$, or $2^a + 1$

$$\mathrm{RNS}(2^{a_{k-1}} \mid 2^{a_{k-2}} \pm 1 \mid \ldots \mid 2^{a_1} \pm 1 \mid 2^{a_0} \pm 1)$$

We can have only one even modulus

$2^{a_i} - 1$ and $2^{a_j} + 1$ are relatively prime

Neither 5 nor 3 is acceptable

$\mathrm{RNS}(2^5 \mid 2^4-1 \mid 2^4+1 \mid 2^3-1)$ $\quad M = 57\,120$

$\mathrm{RNS}(2^5 \mid 2^4+1 \mid 2^3+1 \mid 2^3-1 \mid 2^2-1)$ $\quad M = 102\,816$

The modulus $2^a + 1$ is not as convenient as $2^a - 1$
(needs an extra bit for residue, and modular operations are not as simple)

Diminished-1 representation of values in $[0, 2^a]$ is a way to simplify things
Represent 0 by a special flag bit and nonzero values by coding one less

4.3 Encoding and Decoding of Numbers

Conversion from binary/decimal to RNS

Example 4.1: Represent the number $y = (1010\ 0100)_{two} = (164)_{ten}$ in RNS(8 | 7 | 5 | 3)

The mod-8 residue is easy to find

$x_3 = \langle y \rangle_8 = (100)_{two} = 4$

We have $y = 2^7+2^5+2^2$; thus

$x_2 = \langle y \rangle_7 = \langle 2 + 4 + 4 \rangle_7 = 3$

$x_1 = \langle y \rangle_5 = \langle 3 + 2 + 4 \rangle_5 = 4$

$x_0 = \langle y \rangle_3 = \langle 2 + 2 + 1 \rangle_3 = 2$

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

Conversion from RNS to Mixed-Radix Form

MRS(m_{k-1} | . . . | m_2 | m_1 | m_0) is a k-digit positional system with weights

$m_{k-2}...m_2m_1m_0$ . . . $m_2m_1m_0$ m_1m_0 m_0 1

and digit sets

$[0, m_{k-1}-1]$ . . . $[0,m_3-1]$ $[0,m_2-1]$ $[0,m_1-1]$ $[0,m_0-1]$

Example: $(0 \mid 3 \mid 1 \mid 0)_{\text{MRS}(8|7|5|3)} = 0\times105 + 3\times15 + 1\times3 + 0\times1 = 48$

RNS-to-MRS conversion problem:

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

MRS representation allows magnitude comparison and sign detection

Example: 48 versus 45

$(0 \mid 6 \mid 3 \mid 0)_{\text{RNS}}$	vs	$(5 \mid 3 \mid 0 \mid 0)_{\text{RNS}}$
$(000 \mid 110 \mid 011 \mid 00)_{\text{RNS}}$	vs	$(101 \mid 011 \mid 000 \mid 00)_{\text{RNS}}$

Equivalent mixed-radix representations

$(0 \mid 3 \mid 1 \mid 0)_{\text{MRS}}$	vs	$(0 \mid 3 \mid 0 \mid 0)_{\text{MRS}}$
$(000 \mid 011 \mid 001 \mid 00)_{\text{MRS}}$	vs	$(000 \mid 011 \mid 000 \mid 00)_{\text{MRS}}$

Conversion from RNS to Binary/Decimal

Theorem 4.1 (The Chinese remainder theorem)

$$x = (x_{k-1} \mid \ldots \mid x_2 \mid x_1 \mid x_0)_{\text{RNS}} = \langle \textstyle\sum_i M_i \langle \alpha_i x_i \rangle_{m_i} \rangle_M$$

where $M_i = M/m_i$ and $\alpha_i = \langle M_i^{-1} \rangle_{m_i}$ (multiplicative inverse of M_i wrt m_i)

Implementing CRT-based RNS-to-binary conversion

$$x = \langle \textstyle\sum_i M_i \langle \alpha_i x_i \rangle_{m_i} \rangle_M = \langle \textstyle\sum_i f_i(x_i) \rangle_M$$

We can use a table to store the f_i values –- $\sum_i m_i$ entries

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
		⋮	⋮

Intuitive Justification for CRT

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

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

$$(1 \mid 0 \mid 0)_{\text{RNS}(7|5|3)} = \text{multiple of 15 that is 1 mod 7} = 15$$
$$(0 \mid 1 \mid 0)_{\text{RNS}(7|5|3)} = \text{multiple of 21 that is 1 mod 5} = 21$$
$$(0 \mid 0 \mid 1)_{\text{RNS}(7|5|3)} = \text{multiple of 35 that is 1 mod 3} = 70$$

$$\begin{aligned}(2 \mid 3 \mid 2)_{\text{RNS}(7|5|3)} &= (2 \mid 0 \mid 0) + (0 \mid 3 \mid 0) + (0 \mid 0 \mid 2) \\ &= 2 \times (1 \mid 0 \mid 0) + 3 \times (0 \mid 1 \mid 0) + 2 \times (0 \mid 0 \mid 1) \\ &= 2 \times 15 + 3 \times 21 + 2 \times 70 \\ &= 30 + 63 + 140 \\ &= 233 = 23 \bmod 105\end{aligned}$$

Therefore, $x = (23)_{\text{ten}}$

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]

$$a = (0 \mid 1 \mid 3 \mid 2)_{\mathrm{RNS}}$$
$$b = (0 \mid 1 \mid 4 \mid 1)_{\mathrm{RNS}}$$
$$c = (0 \mid 6 \mid 2 \mid 1)_{\mathrm{RNS}}$$
$$d = (2 \mid 0 \mid 0 \mid 2)_{\mathrm{RNS}}$$
$$e = (5 \mid 0 \mid 1 \mid 0)_{\mathrm{RNS}}$$
$$f = (7 \mid 6 \mid 4 \mid 2)_{\mathrm{RNS}}$$

Answers:

$$d < c < f < a < e < b$$

$$-70 < -8 < -1 < 8 < 21 < 64$$

Approximate CRT Decoding

Theorem 4.1 (The Chinese remainder theorem, scaled version)

Divide both sides of CRT equality by M to get scaled version of x in [0, 1)

$$x \quad = (x_{k-1} \mid \ldots \mid x_2 \mid x_1 \mid x_0)_{\text{RNS}} = \langle \textstyle\sum_i \; M_i \langle \alpha_i x_i \rangle_{m_i} \rangle_M$$

$$x/M = \langle \textstyle\sum_i \; \langle \alpha_i x_i \rangle_{m_i} / m_i \rangle_1 = \langle \textstyle\sum_i g_i(x_i) \rangle_1$$

where mod-1 summation implies that we discard the integer parts

Errors can be estimated and kept in check for the particular application

Table 4.3 Values needed in applying the approximate Chinese remainder theorem decoding to RNS(8 | 7 | 5 | 3)

i	m_i	x_i	$\langle \alpha_i x_i \rangle_{m_i} / m_i$
3	8	0	.0000
		1	.1250
		2	.2500
		3	.3750
		⋮	⋮

General RNS Division

General RNS division, as opposed to division by one of the moduli (aka scaling), is difficult; hence, use of RNS is unlikely to be effective when an application requires many divisions

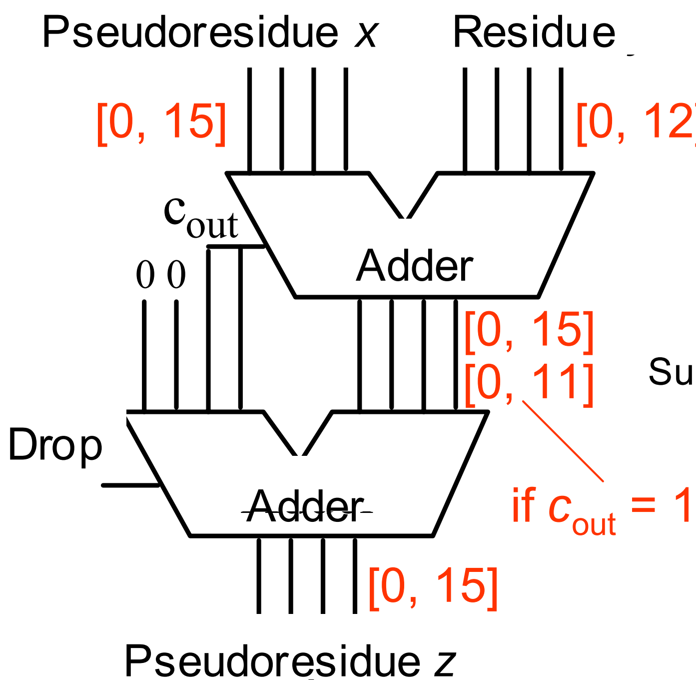
Scheme proposed in 1994 PhD thesis of Ching-Yu Hung (UCSB): Use an algorithm that has built-in tolerance to imprecision, and apply the approximate CRT decoding to choose quotient digits

Example — SRT algorithm (s is the partial remainder)

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

The BSD quotient can be converted to RNS on the fly

4.5 Redundant RNS Representations



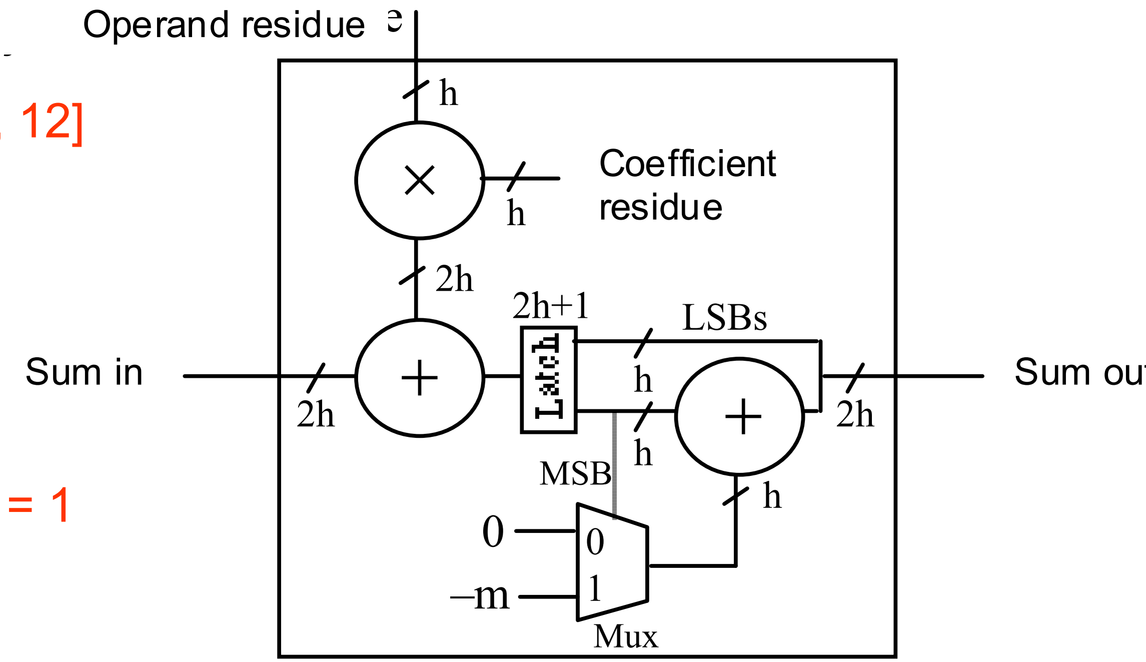


Fig. 4.3 Adding a 4-bit ordinary mod-13 residue *x* to a 4-bit pseudoresidue *y*, producing a 4-bit mod-13 pseudoresidue *z*.

Fig. 4.4 A modulo-*m* multiply-add cell that accumulates the sum into a double-length redundant pseudoresidue.

4.6 Limits of Fast Arithmetic in RNS

Known results from number theory

Theorem 4.2: The *i*th 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

Implications to speed of arithmetic in RNS

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

That is, with fast log-time adders, addition needs $O(\log \log k)$ time

Limits for Low-Cost RNS

Known results from number theory

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 $O(i^2 \ln i)$

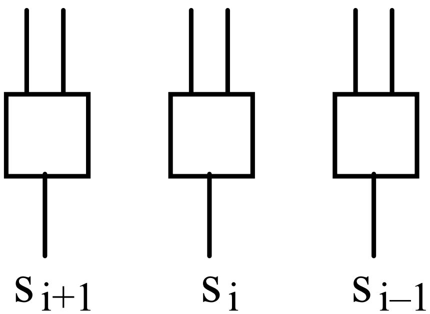
Implications to speed of arithmetic in low-cost RNS

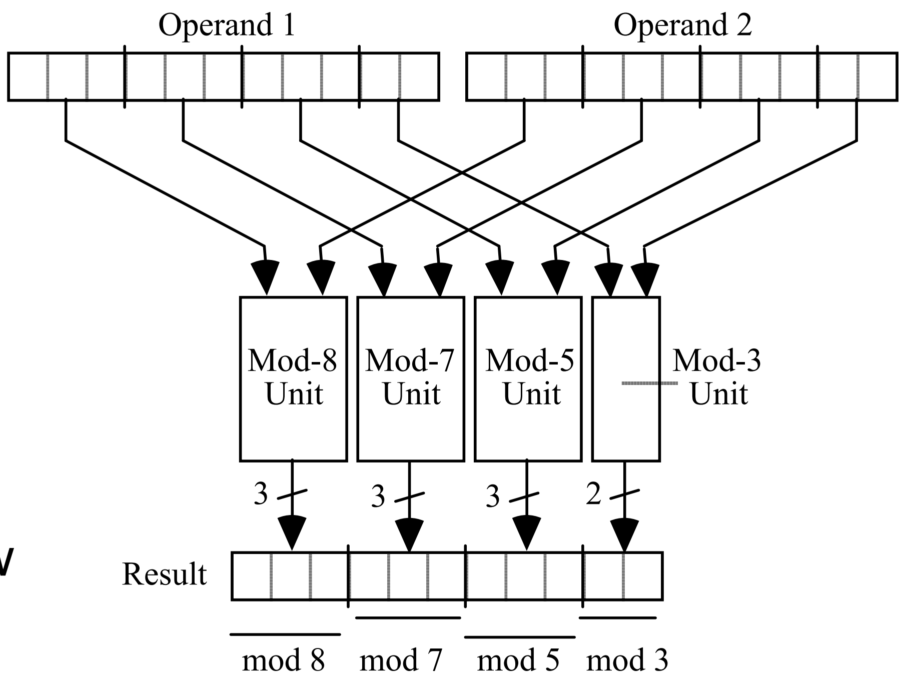
Theorem 4.8: It is possible to represent all k-bit binary numbers in RNS with $O((k / \log k)^{1/2})$ low-cost moduli of the form $2^a - 1$ such that the largest modulus has $O((k \log k)^{1/2})$ bits

Because a fast adder needs $O(\log k)$ time, asymptotically, low-cost RNS offers little speed advantage over standard binary

Disclaimer About RNS Representations

RNS representations are sometimes referred to as "carry-free"





Positional representation does not support totally carry-free addition; but it appears that RNS does allow digitwise arithmetic

However . . . even though each RNS digit is processed independently (for +, –, ×), the size of the digit set is dependent on the desired range (grows at least double-logarithmically with the range *M*, or logarithmically with the word width *k* in the binary representation of the same range)