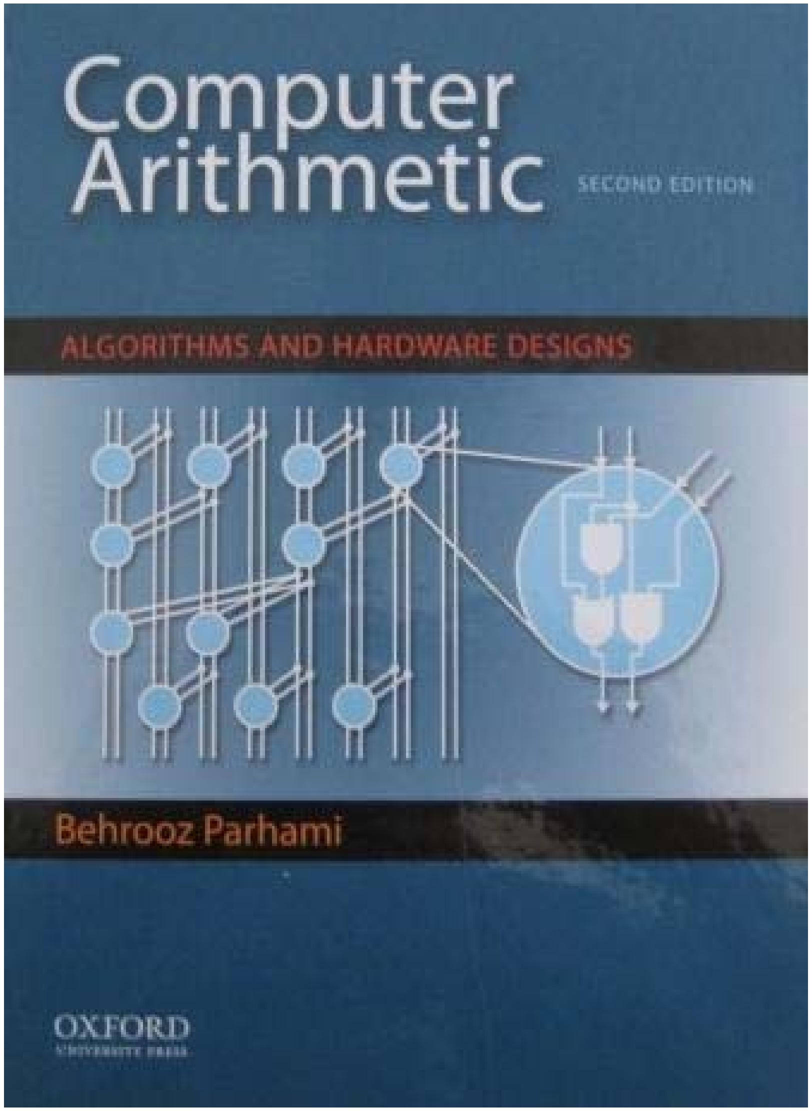


Part IV
Division

	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
	III. Multiplication	9. Basic Multiplication Schemes 10. High-Radix Multipliers 11. Tree and Array Multipliers 12. Variations in Multipliers
	IV. Division	13. Basic Division Schemes 14. High-Radix Dividers 15. Variations in Dividers 16. Division by Convergence
	V. Real Arithmetic	17. Floating-Point Reperesentations 18. Floating-Point Operations 19. Errors and Error Control 20. Precise and Certifiable Arithmetic
	VI. Function Evaluation	21. Square-Rooting Methods 22. The CORDIC Algorithms 23. Variations in Function Evaluation 24. Arithmetic by Table Lookup
	VII. Implementation Topics	25. High-Throughput Arithmetic 26. Low-Power Arithmetic 27. Fault-Tolerant Arithmetic 28. Reconfigurable Arithmetic

Appendix: Past, Present, and Future

About This Presentation

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

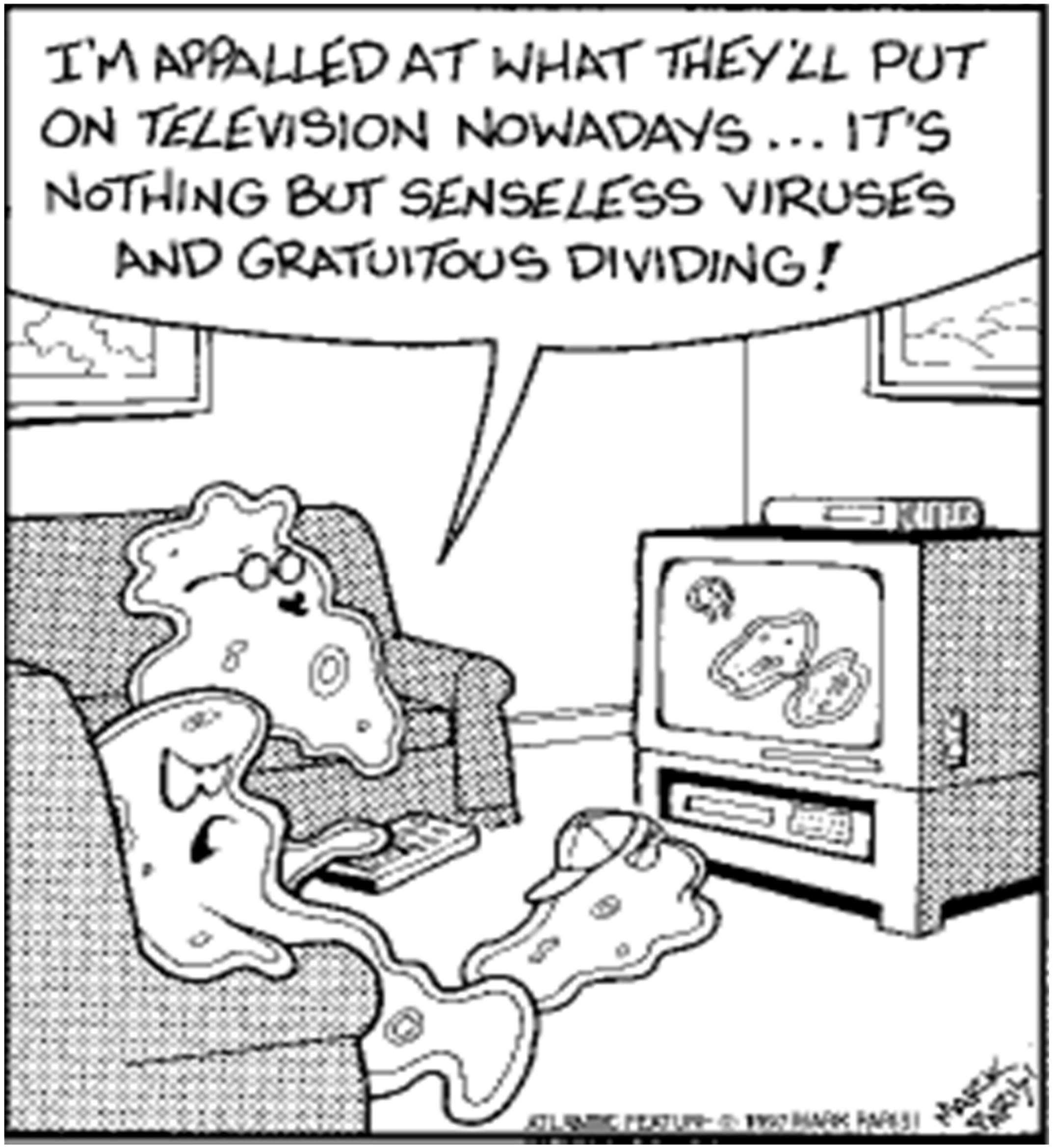
Edition	Released	Revised	Revised	Revised	Revised
First	Jan. 2000	Sep. 2001	Sep. 2003	Oct. 2005	May 2007
		May 2008	May 2009		
Second	May 2010	Apr. 2011	May 2012	May 2015	

IV Division

Review Division schemes and various speedup methods

- Hardest basic operation (fortunately, also the rarest)
- Division speedup methods: high-radix, array, . . .
- Combined multiplication/division hardware
- Digit-recurrence vs convergence division schemes

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


I'M APPALLED AT WHAT THEY'LL PUT ON TELEVISION NOWADAYS ... IT'S NOTHING BUT SENSELESS VIRUSES AND GRATUITOUS DIVIDING!
ATLANTIC FEATURE © 1997 MARK PARISI
MARK PARISI

Be fruitful and multiply . . .

Now, divide.

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/subtract divide vs shift/add multiply
Hardware, firmware, software algorithms
Dividing 2's-complement numbers
The special case of a constant divisor

Basic Division Schemes: Topics

Topics in This Chapter
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 Radix-2 SRT Division

13.1 Shift/Subtract Division Algorithms

Notation for our discussion of division algorithms:

z	Dividend	$z_{2k-1}z_{2k-2} \quad . \quad . \quad . \quad z_3z_2z_1z_0$
d	Divisor	$d_{k-1}d_{k-2} \ldots d_1d_0$
q	Quotient	$q_{k-1}q_{k-2} \ldots q_1q_0$
s	Remainder, $z - (d \times q)$	$s_{k-1}s_{k-2} \ldots s_1s_0$

Initially, we assume unsigned operands

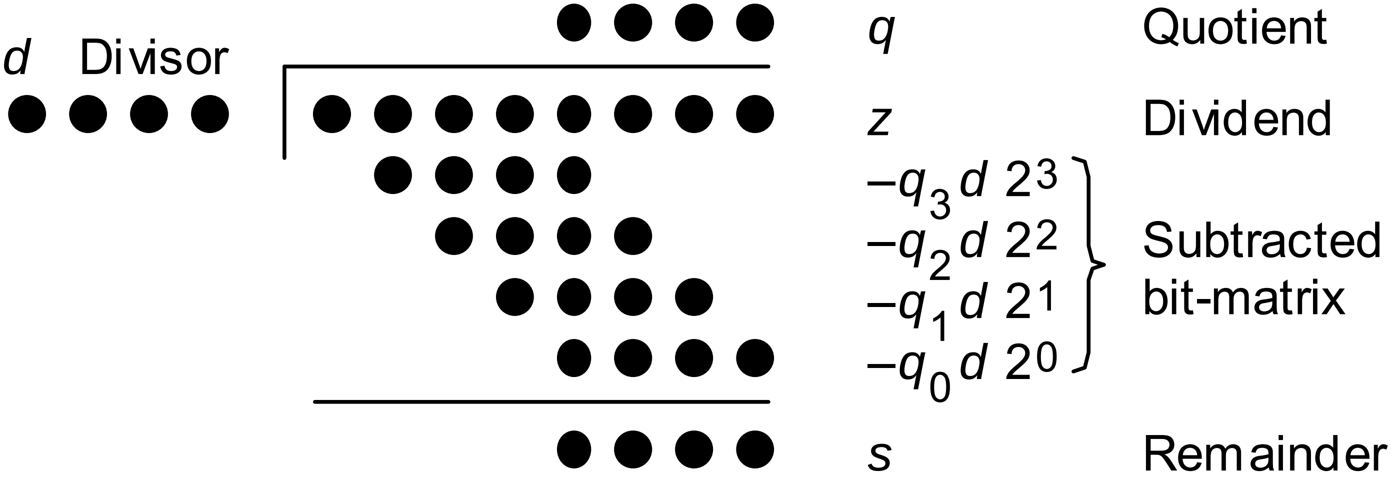
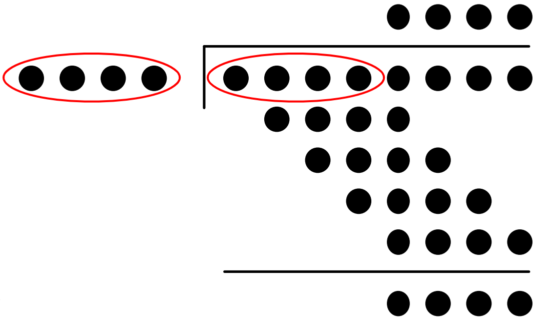


Fig. 13.1 Division of an 8-bit number by a 4-bit number in dot notation.

Division versus Multiplication

Division is more complex than multiplication:
Need for quotient digit selection or estimation

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



Pentium III latencies

Instruction	**Latency**	**Cycles/Issue**
Load / Store	**3**	**1**
Integer Multiply	**4**	**1**
Integer Divide	**36**	**36**
Double/Single FP Multiply	**5**	**2**
Double/Single FP Add	**3**	**1**
Double/Single FP Divide	**38**	**38**

The ratios haven't changed much in later Pentiums, Atom, or AMD products*

*Source: T. Granlund, "Instruction Latencies and Throughput for AMD and Intel x86 Processors," Feb. 2012

Division Recurrence

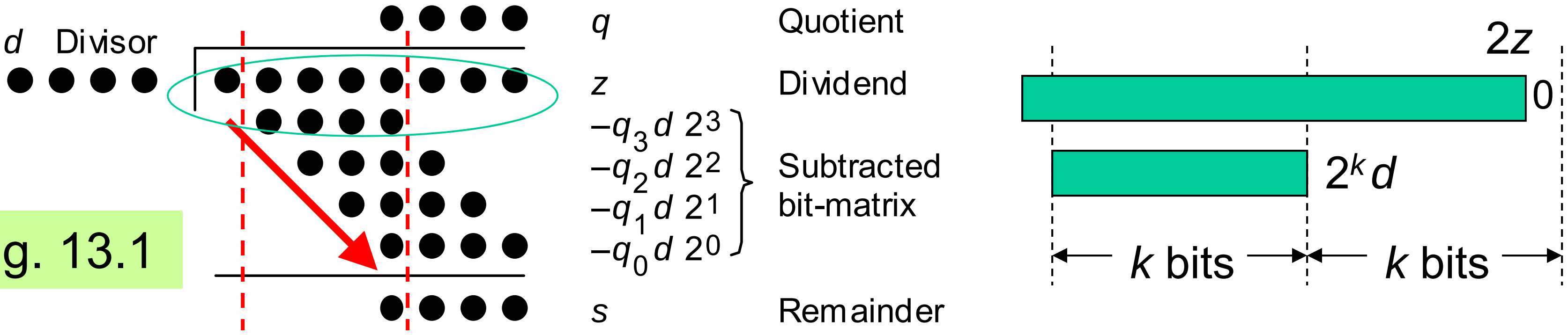


Fig. 13.1

Division with left shifts (There is no corresponding right-shift algorithm)

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

with $s^{(0)} = z$ and

$s^{(k)} = 2^k s$

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

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

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

Divide fractions like integers; adjust the remainder

No-overflow condition for fractions is:

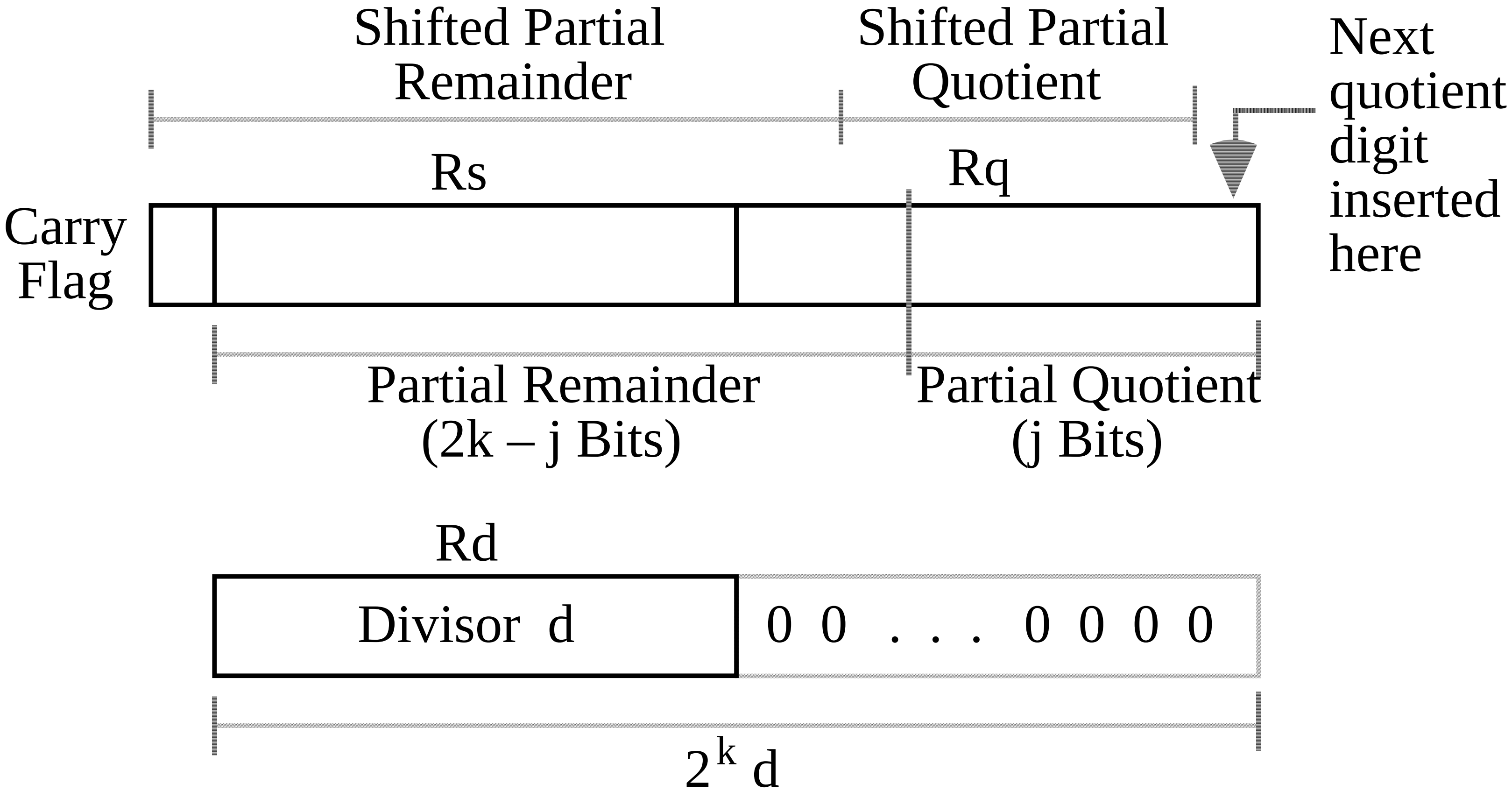
$$z_{\text{frac}} < d_{\text{frac}}$$

Examples of Basic Division

Integer division	Decimal			Fractional division		
z	117	0 1 1 1 0 1 0 1		z_{frac}	. 0 1 1 1 0 1 0 1	
$2^4 d$	10	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	$\{q_0 = 1\}$	$-q_{-4} d$	. 1 0 1 0	$\{q_{-4}=1\}$
$s^{(4)}$		0 1 1 1		$s^{(4)}$	. 0 1 1 1	
s	7	0 1 1 1		s_{frac}	0 . 0 0 0 0 0 1 1 1	
q	11	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



Shifted Partial Remainder
Shifted Partial Quotient
Next quotient digit inserted here
Rs
Rq
Carry Flag
Partial Remainder (2k – j Bits)
Partial Quotient (j Bits)
Rd
Divisor d
0 0 . . . 0 0 0 0
$2^k d$

Fig. 13.3 Register usage for programmed division.

Assembly Language Program for 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. 13.3 Register usage for programmed division.

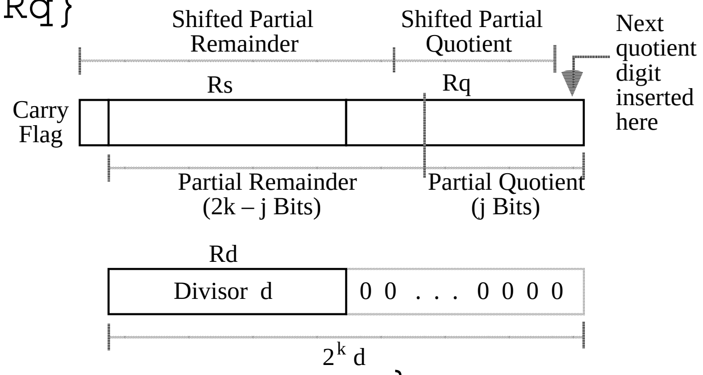

Shifted Partial Remainder
Shifted Partial Quotient
Next quotient digit inserted here
Rs
Rq
Carry Flag
Partial Remainder (2k – j Bits)
Partial Quotient (j Bits)
Rd
Divisor d
0 0 . . . 0 0 0 0
2^k d

Fig. 13.4 Programmed division using left shifts.

Time Complexity of Programmed Division

Assume *k*-bit words

k iterations of the main loop
6-8 instructions per iteration, depending on the quotient bit

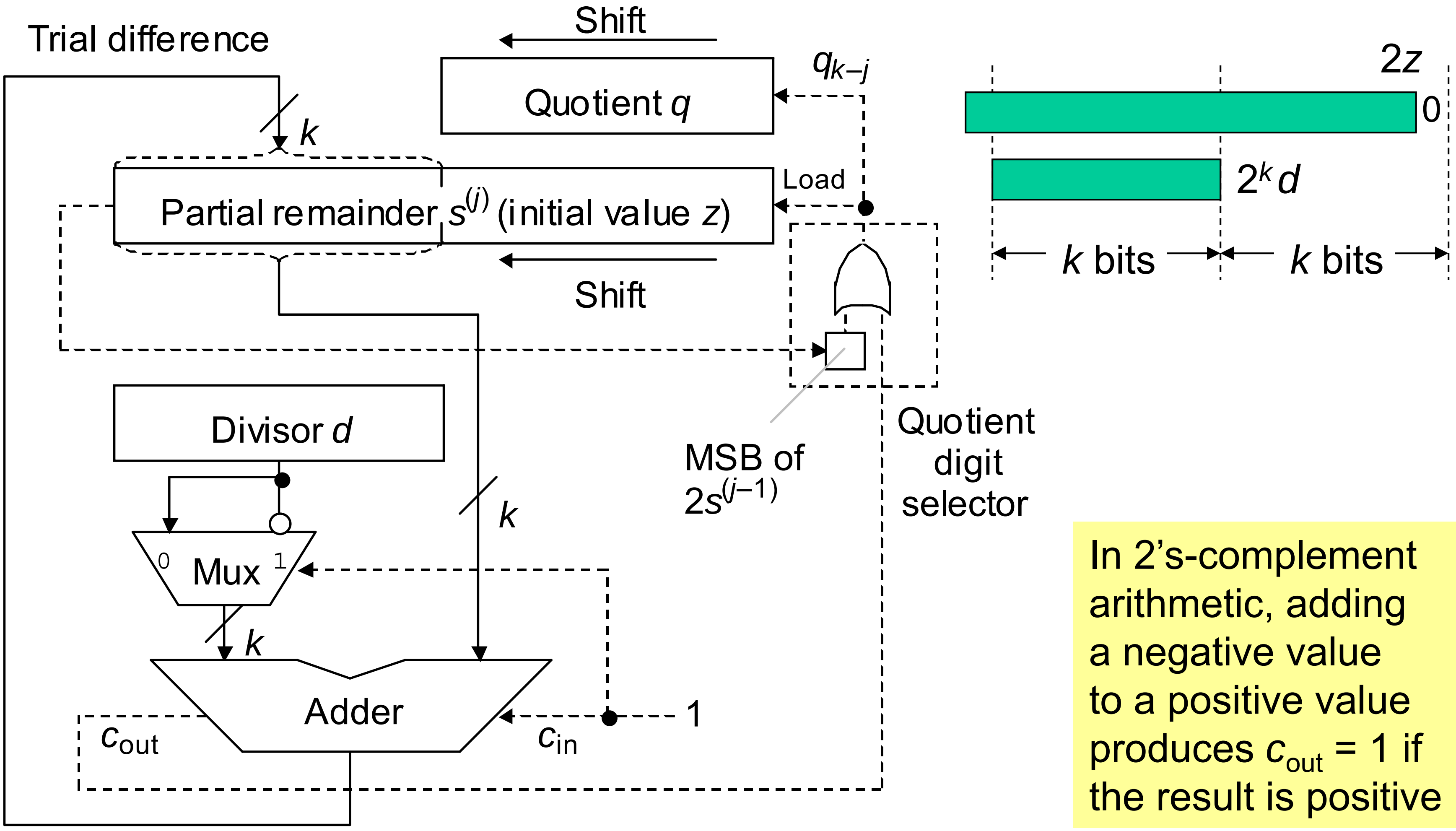
Thus, 6*k* + 3 to 8*k* + 3 machine instructions,
ignoring operand loads and result store

k = 32 implies 220^+ instructions on average

This is too slow for many modern applications!

Microprogrammed division would be somewhat better

13.3 Restoring Hardware Dividers



In 2's-complement arithmetic, adding a negative value to a positive value produces c_{out} = 1 if the result is positive

Fig. 13.5 Shift/subtract sequential restoring divider.

Example of Restoring Unsigned Division

No overflow, because $(0111)_{two} < (1010)_{two}$

z		0 1 1 1	0 1 0 1	
2^4d	0	1 0 1 0		
$–2^4d$	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	
$+(–2^4d)$	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	
$+(–2^4d)$	1	0 1 1 0		
$s^{(2)}$	1	1 1 1 1	0 1	Negative, so set $q_2 = 0$
$s^{(2)}=2s^{(1)}$	0	1 0 0 1	0 1	and restore
$2s^{(2)}$	1	0 0 1 0	1	
$+(–2^4d)$	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		
$+(–2^4d)$	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.

Indirect Signed Division

In 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

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

$z = 5 \quad d = –3 \quad \Rightarrow \quad q = –1 \quad s = 2$ (not $q = –2$, $s = –1$)

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

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

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

13.4 Nonrestoring and Signed Division

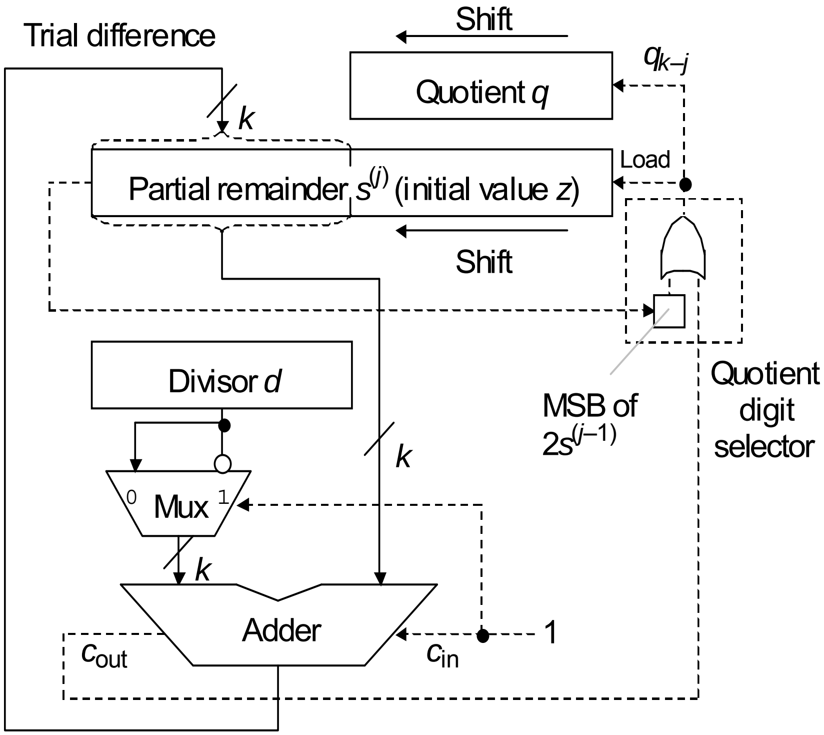
The cycle time in restoring division must accommodate:

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, causing a lengthening of the clock cycle

Nonrestoring division to the rescue!

Assume $q_{k-j} = 1$ and subtract
Store the result as the new PR
(the partial remainder can become incorrect, hence the name "nonrestoring")



Justification for Nonrestoring Division

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

Shifted partial remainder at start of the cycle is u

Suppose subtraction yields the negative result $u - 2^k d$

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

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

Example of Nonrestoring Unsigned Division

No overflow: $(0111)_{two} < (1010)_{two}$

				Decimal	
z		0 1 1 1	0 1 0 1	117	
2^4d	0	1 0 1 0		10×16	
-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, so subtract
$+(-2^4d)$	1	0 1 1 0			
$s^{(1)}$	0	0 1 0 0	1 0 1		
$2s^{(1)}$	0	1 0 0 1	0 1		Positive, so set $q_3 = 1$ and subtract
$+(-2^4d)$	1	0 1 1 0			
$s^{(2)}$	1	1 1 1 1	0 1		
$2s^{(2)}$	1	1 1 1 0	1		Negative, so set $q_2 = 0$ and add
$+2^4d$	0	1 0 1 0			
$s^{(3)}$	0	1 0 0 0	1		
$2s^{(3)}$	1	0 0 0 1			Positive, so set $q_1 = 1$ and subtract
$+(-2^4d)$	1	0 1 1 0			
$s^{(4)}$	0	0 1 1 1			Positive, so set $q_0 = 1$
s			0 1 1 1	7	
q			1 0 1 1	11	

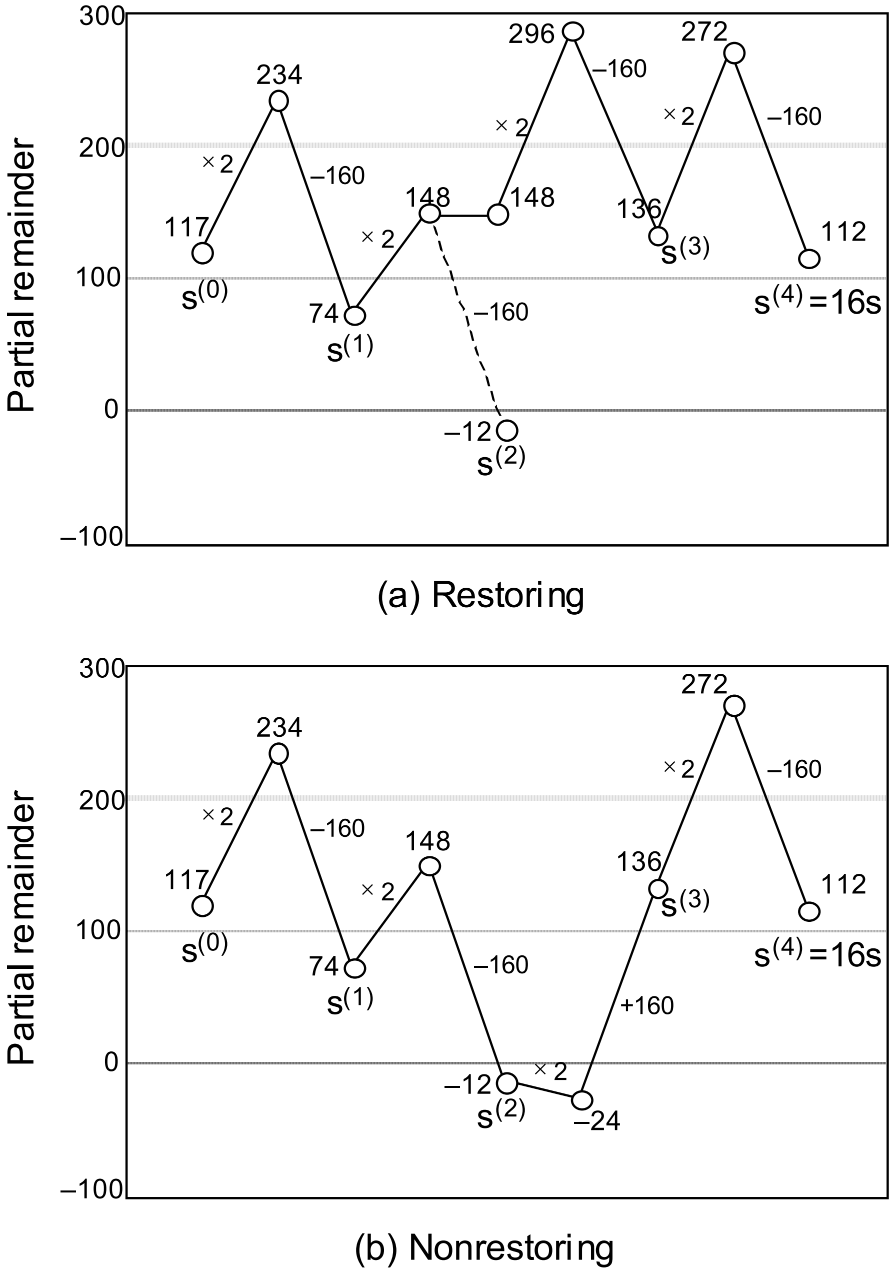
Fig. 13.7 Example of nonrestoring unsigned division.

Graphical Depiction of Nonrestoring Division

Example

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

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



(a) Restoring

(b) Nonrestoring

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

Convergence of the Partial Quotient to q

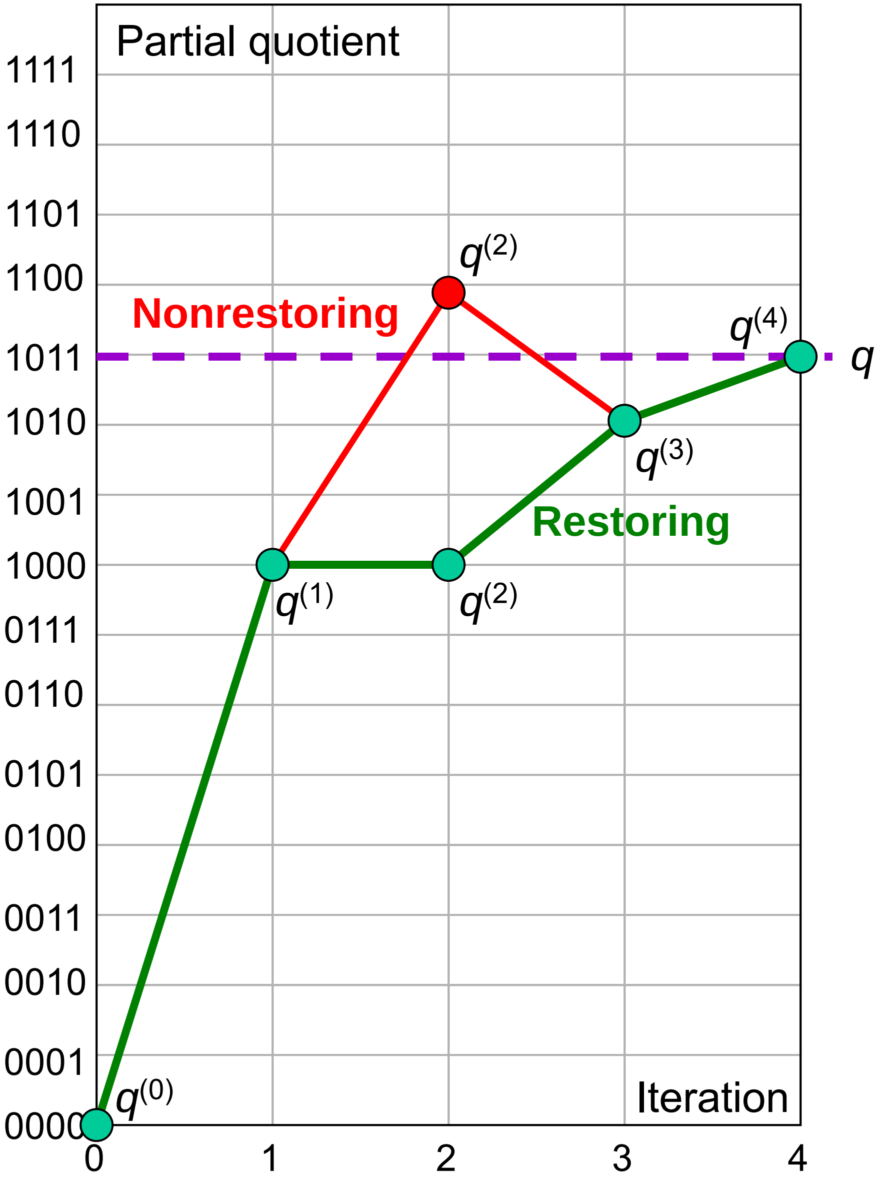
Example

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

$(117)_{\text{ten}}/(10)_{\text{ten}} = (11)_{\text{ten}} = (1011)_{\text{two}}$

In restoring division, the partial quotient converges to q from below

In nonrestoring division, the partial quotient may overshoot q, but converges to it after some oscillations



Nonrestoring Division with Signed Operands

Restoring division

$q_{k-j} = 0$ means no subtraction (or subtraction of 0)
$q_{k-j} = 1$ means subtraction of d

Example: $q =$ . . . 0 0 0 1 . . .
. . . 1 −1 −1 −1 . . .

Nonrestoring division

We always subtract or add
It is as if quotient digits are selected from the set {1, −1}:
1 corresponds to subtraction −1 corresponds to addition

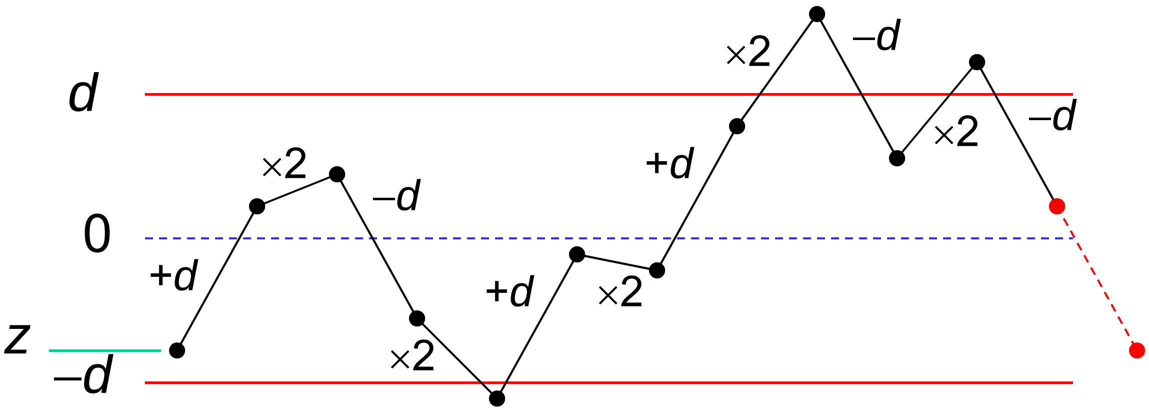
Our goal is to end up with a remainder that matches the sign of the dividend

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

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

Quotient Conversion and Final Correction

Partial remainder variation and selected quotient digits during nonrestoring division with $d > 0$



Quotient with digits $^-1$ and 1	$^-1$	1	$^-1$	$^-1$	1	1	
Replace $^-1$s with 0s	0	1	0	0	1	1	
Shift left, complement MSB, and set LSB to 1 to get the 2’s-complement quotient	1	1	0	0	1	1	1

Check: –32 + 16 – 8 – 4 + 2 + 1 = –25 = –64 + 32 + 4 + 2 + 1

1	1	0	1	0	0	0

Final correction step if sign(s) ≠ sign(z):
Add d to, or subtract d from, s; subtract 1 from, or add 1 to, q

Example of Nonrestoring Signed Division

z		0 0 1 0	0 0 0 1
2^4d	1	1 0 0 1	
-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
$+2^4d$	1	1 0 0 1	
$s^{(1)}$	1	1 1 0 1	0 0 1
$2s^{(1)}$	1	1 0 1 0	0 1
$+(-2^4d)$	0	0 1 1 1	
$s^{(2)}$	0	0 0 0 1	0 1
$2s^{(2)}$	0	0 0 1 0	1
$+2^4d$	1	1 0 0 1	
$s^{(3)}$	1	1 0 1 1	1
$2s^{(3)}$	1	0 1 1 1	
$+(-2^4d)$	0	0 1 1 1	
$s^{(4)}$	1	1 1 1 0	
$+(-2^4d)$	0	0 1 1 1	
$s^{(4)}$	0	0 1 0 1	
s			0 1 0 1
q			$^{-}1\ 1\ ^{-}1\ 1$

sign($s^{(0)}$) ≠ sign(d),
so set **$q_3 = {}^{-}1$** and add

sign($s^{(1)}$) = sign(d),
so set **$q_2 = 1$** and subtract

sign($s^{(2)}$) ≠ sign(d),
so set **$q_1 = {}^{-}1$** and add

sign($s^{(3)}$) = sign(d),
so set **$q_0 = 1$** and subtract

sign($s^{(4)}$) ≠ sign(z),
so perform corrective subtraction

Fig. 13.9 Example of nonrestoring signed division.

p =	0 1 0 1	Shift, compl MSB
1	1 0 1 1	Add 1 to correct
	1 1 0 0	Check: 33/(–7) = –4

Nonrestoring Hardware Divider

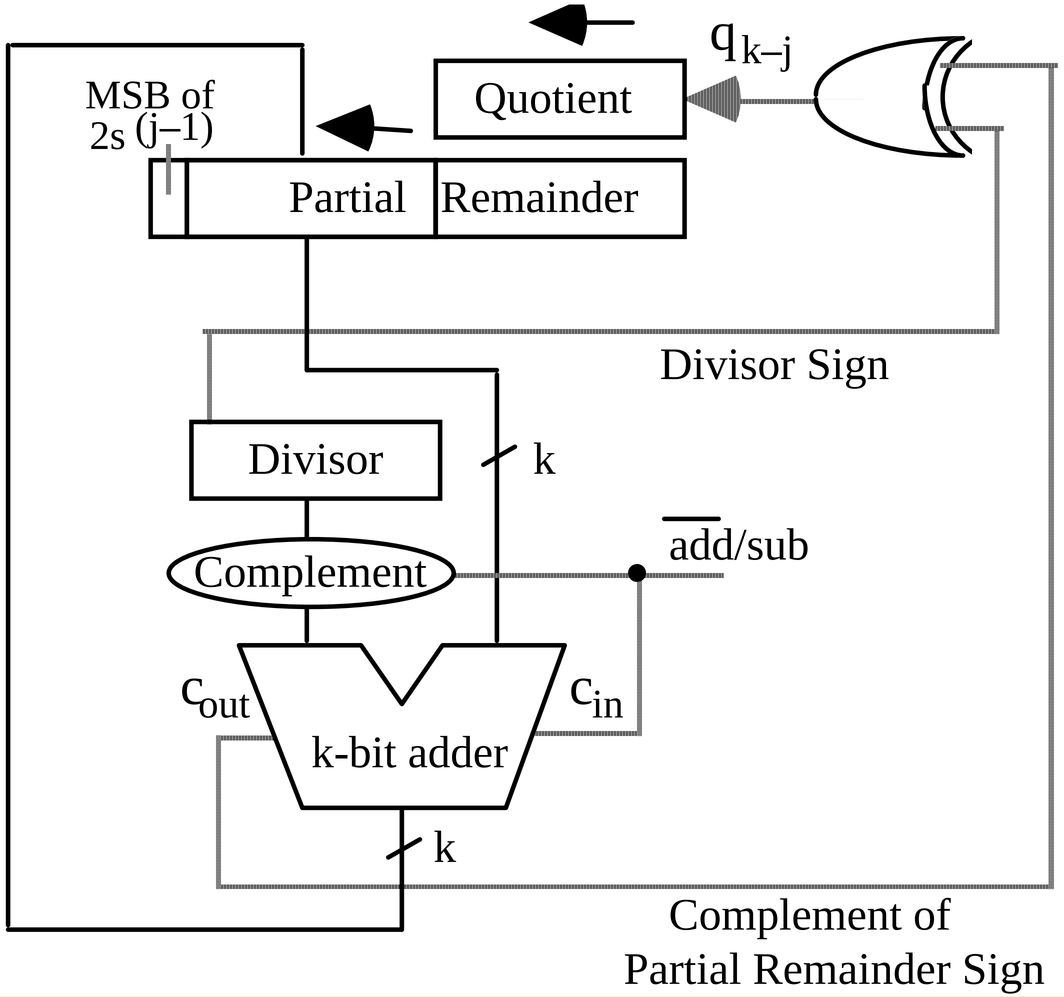


Fig. 13.10 Shift-subtract sequential nonrestoring divider.

13.5 Division by Constants

Software and hardware aspects:

As was the case for multiplications by constants, optimizing compilers may replace some divisions by shifts/adds/subs; likewise, in custom VLSI circuits, hardware dividers may be replaced by simpler adders

Method 1: Find the reciprocal of the constant and multiply (particularly efficient if several numbers must be divided by the same divisor)

Method 2: Use the property that for each odd integer *d*, there exists an odd integer *m* such that $d \times m = 2^n - 1$; hence, $d = (2^n - 1)/m$ and

$$\frac{z}{d} = \frac{zm}{2^n - 1} = \frac{zm}{2^n(1-2^{-n})} = \overset{\text{Multiplication by constant}}{\frac{zm}{2^n}} \overbrace{(1+2^{-n})(1+2^{-2n})(1+2^{-4n})\cdots}^{\text{Shift-adds}}$$

Number of shift-adds required is proportional to log *k*

Example Division by a Constant

Example: Dividing the number *z* by 5, assuming 24 bits of precision.
We have $d = 5$, $m = 3$, $n = 4$; $5 \times 3 = 2^4 - 1$

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

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

Instruction sequence for division by 5

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

5 shifts
4 adds

Numerical Examples for Division by 5

Instruction sequence for division by 5

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

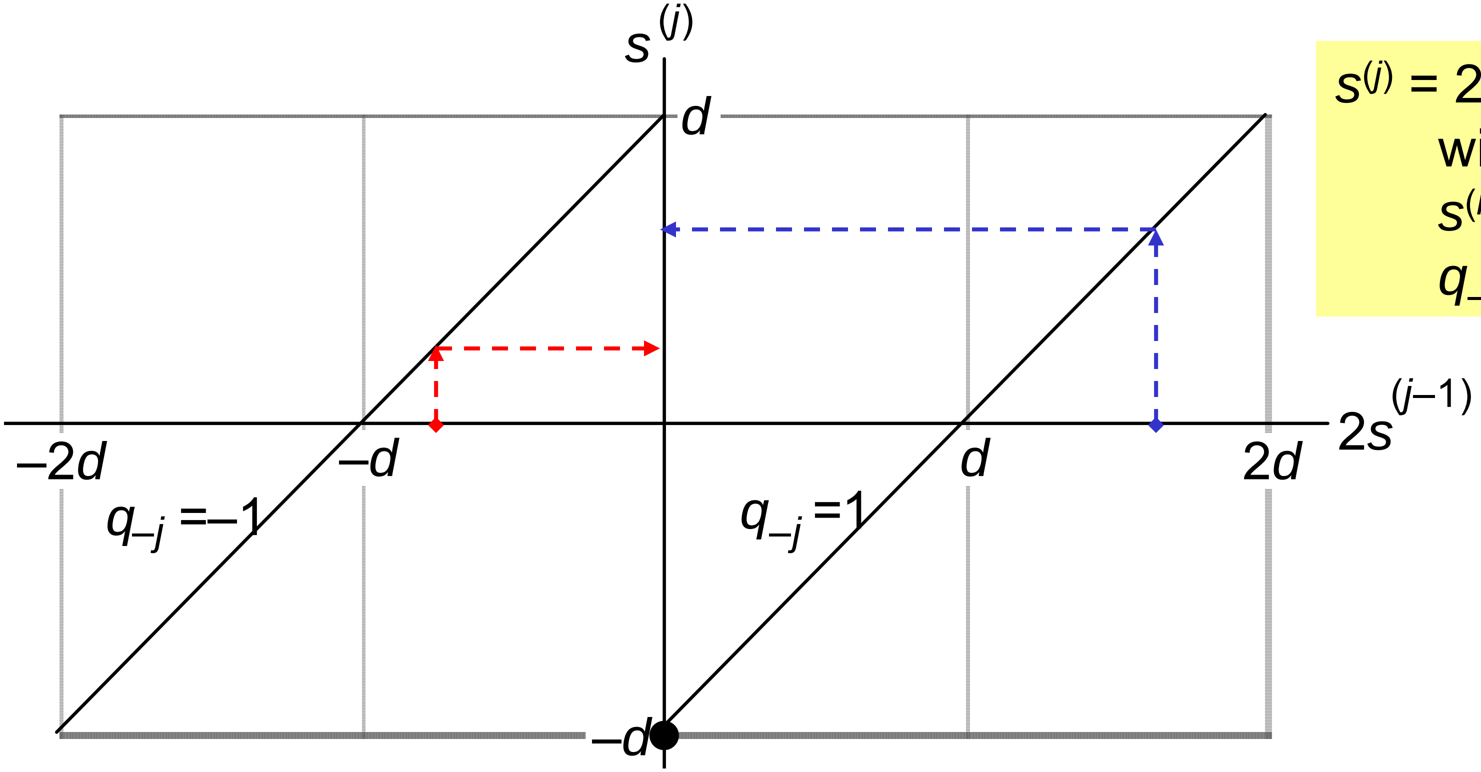
Computing 29 ÷ 5 (z = 29, d = 5)

87 ← 29 + 29 shift-left 1 {3z computed}
92 ← 87 + 87 shift-right 4 {$3\text{z}(1+2^{-4})$ computed}
92 ← 92 + 92 shift-right 8 {$3\text{z}(1+2^{-4})(1+2^{-8})$ computed}
92 ← 92 + 92 shift-right 16 {$3\text{z}(1+2^{-4})(1+2^{-8})(1+2^{-16})$ computed}
5 ← 92 shift-right 4 {$3\text{z}(1+2^{-4})(1+2^{-8})(1+2^{-16})/16$ computed}

Repeat the process for computing 30 ÷ 5 and comment on the outcome

13.6 Radix-2 SRT Division

SRT division takes its name from Sweeney, Robertson, and Tocher, who independently discovered the method



$s^{(j)} = 2s^{(j-1)} - q_{-j}\, d$

with $s^{(0)} = z$

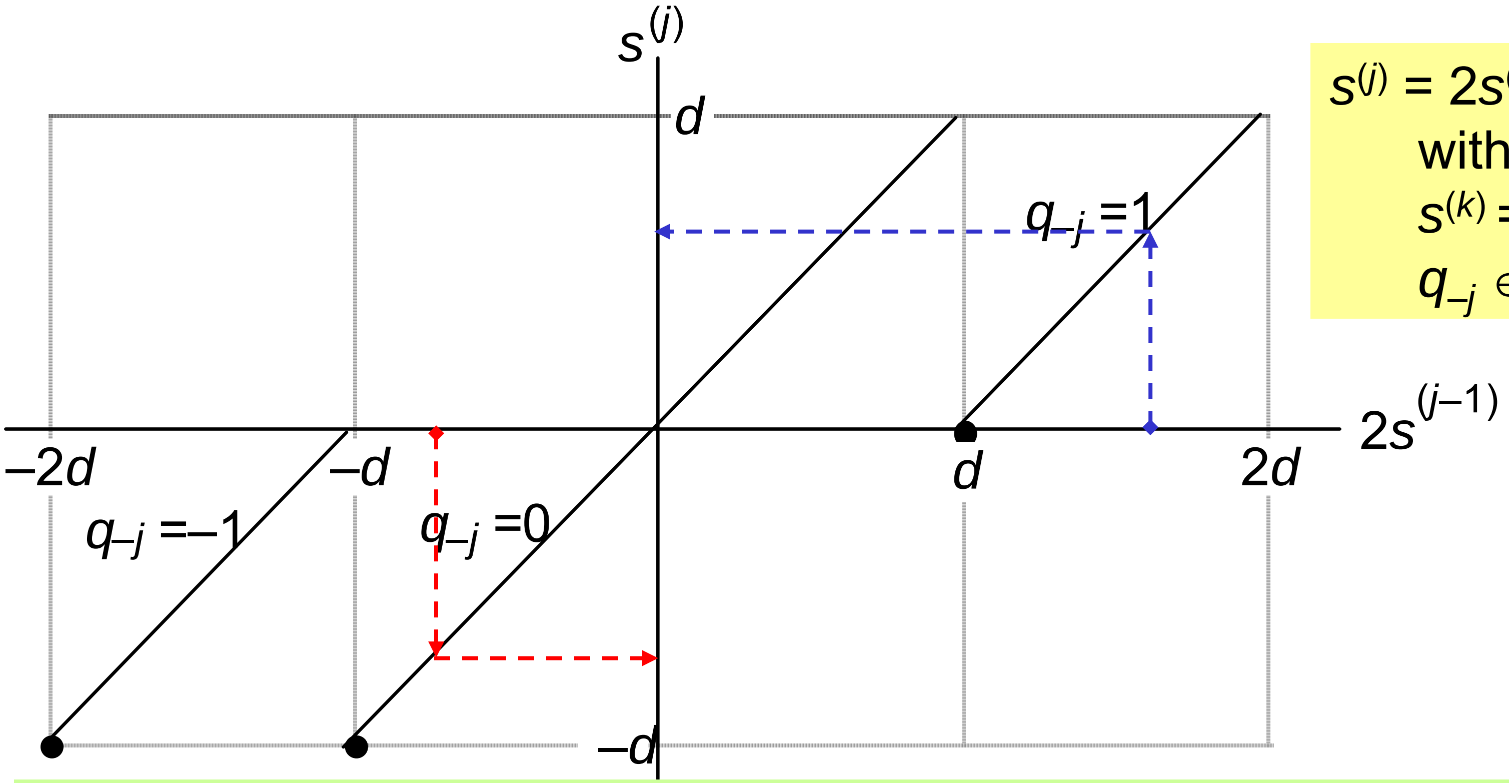
$s^{(k)} = 2^k s$

$q_{-j} \in \{-1, 1\}$

Fig. 13.11 The new partial remainder, $s^{(j)}$, as a function of the shifted old partial remainder, $2s^{(j-1)}$, in radix-2 nonrestoring division.

Allowing 0 as a Quotient Digit in Nonrestoring Division

This method was useful in early computers, because the choice $q_{-j} = 0$ requires shifting only, which was faster than shift-and-subtract



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

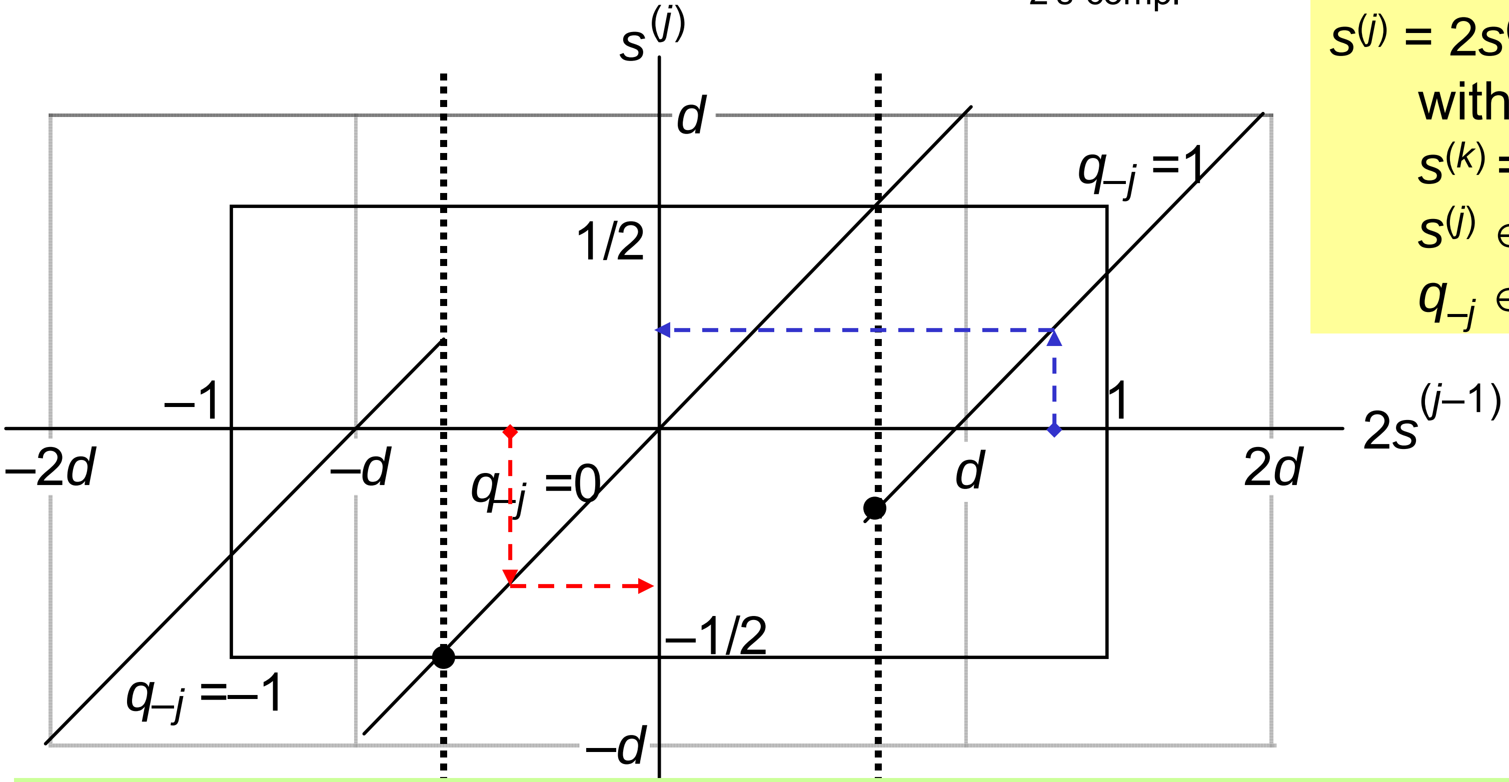
Fig. 13.12 The new partial remainder, $s^{(j)}$, as a function of the shifted old partial remainder, $2s^{(j-1)}$, with q_{-j} in $\{-1, 0, 1\}$.

The Radix-2 SRT Division Algorithm

We use the comparison constants –½ and ½ for quotient digit selection

$2s \geq +½$ means $2s = (0.1\text{xxxxxxxx})_{\text{2's-compl}}$

$2s < -½$ means $2s = (1.0\text{xxxxxxxx})_{\text{2's-compl}}$



$s^{(j)} = 2s^{(j-1)} - q_{-j}\,d$

with $s^{(0)} = z$

$s^{(k)} = 2^k s$

$s^{(j)} \in [-½, ½)$

$q_{-j} \in \{-1, 0, 1\}$

Fig. 13.13 The relationship between new and old partial remainders in radix-2 SRT division.

Radix-2 SRT Division with Variable Shifts

We use the comparison constants –½ and ½ for quotient digit selection

For $2s \ge +½$ or $2s = (0.1xxxxxxxx)_{2\text{'s-compl}}$ choose $q_{-j} = 1$

For $2s < -½$ or $2s = (1.0xxxxxxxx)_{2\text{'s-compl}}$ choose $q_{-j} = {}^{-}1$

Choose $q_{-j} = 0$ in other cases, that is, for:

$0 \le 2s < +½$ or $2s = (0.0xxxxxxxx)_{2\text{'s-compl}}$

$-½ \le 2s < 0$ or $2s = (1.1xxxxxxxx)_{2\text{'s-compl}}$

Observation: What happens when the magnitude of 2*s* is fairly small?

$2s = (0.00001xxxx)_{2\text{'s-compl}}$ — Choosing $q_{-j} = 0$ would lead to the same condition in the next step; generate 5 quotient digits 0 0 0 0 1

$2s = (1.1110xxxxx)_{2\text{'s-compl}}$ — Generate 4 quotient digits 0 0 0 ⁻1

Use leading 0s or leading 1s detection circuit to determine how many quotient digits can be spewed out at once

Statistically, the average skipping distance will be 2.67 bits

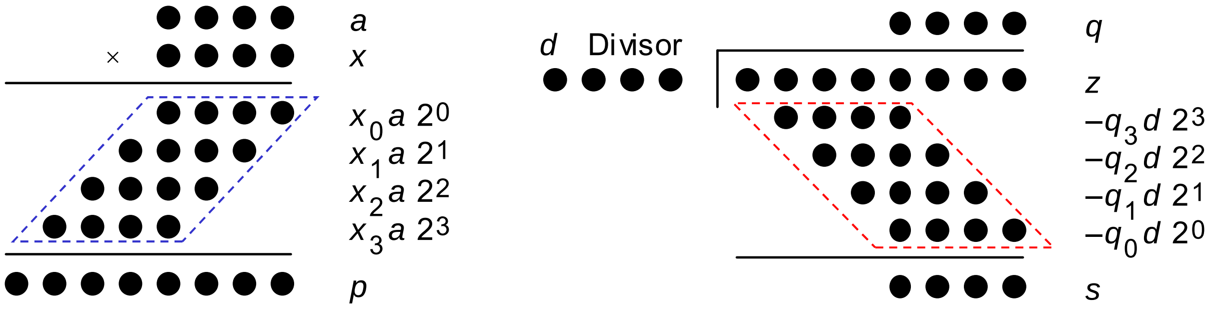
Example Unsigned Radix-2 SRT Division

		In [–½, ½), so okay
z	.0100 0101	
d	0.1010	
$-d$	1.0110	
$s^{(0)}$	0.0100 0101	
$2s^{(0)}$	0.1000 101	≥ ½, so set $q_{-1} = 1$ and subtract
$+(-d)$	1.0110	
$s^{(1)}$	1.1110 101	
$2s^{(1)}$	1.1101 01	In [–½, ½), so set $q_{-2} = 0$
$s^{(2)} = 2s^{(1)}$	1.1101 01	
$2s^{(2)}$	1.1010 1	In [–½, ½), so set $q_{-3} = 0$
$s^{(3)} = 2s^{(2)}$	0.1010 1	
$2s^{(3)}$	1.0101	< –½, so set $q_{-4} = {}^{-}1$ and add
$+d$	0.1010	
$s^{(4)}$	1.1111	Negative, so add to correct
$+d$	0.1010	
$s^{(4)}$	0.1001	
s	0.0000 0101	
q	$0.100{}^{-}1$	Uncorrected BSD quotient
q	0.0110	Convert and subtract *ulp*

0.1	Choose 1
1.0	Choose $^{-}1$
0.0/1.1	Choose 0

Fig. 13.14 Example of unsigned radix-2 SRT division.

Preview of Fast Dividers



(a) $k \times k$ integer multiplication

(b) $2k / k$ integer division

Multiplication and 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 (divide in a higher radix)
b. Adding them faster (keep partial remainder in carry-save form)

There is one complication that makes division inherently more difficult:
The terms to be subtracted from (added to) the dividend are not known a priori but become known as 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 in radix 8, . . .)

Chapter Highlights

Radix > 2 ⇒ quotient digit selection harder
Remedy: redundant quotient representation
Carry-save addition reduces cycle time
Quotient digit selection
Implementation methods and tradeoffs

High-Radix Dividers: Topics

Topics in This Chapter

14.1 Basics of High-Radix Division

14.2 Using Carry-Save Adders

14.3 Radix-4 SRT Division

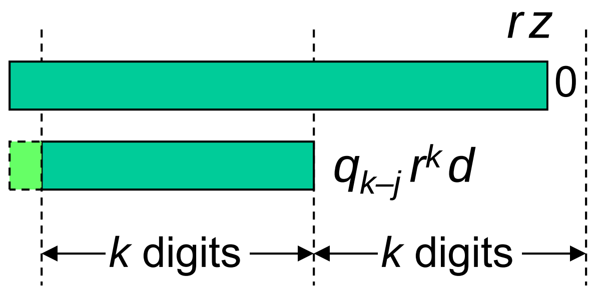
14.4 General High-Radix Dividers

14.5 Quotient Digit Selection

14.6 Using *p*-*d* Plots in Practice

14.1 Basics of High-Radix Division

Radices of practical interest are powers of 2, and perhaps 10



Division with left shifts

$$s^{(j)} = \underbrace{\underbrace{r s^{(j-1)}}_{\text{shift}} - q_{k-j}(r^k d)}_{\text{subtract}}$$

with $s^{(0)} = z$ and
$s^{(k)} = r^k s$

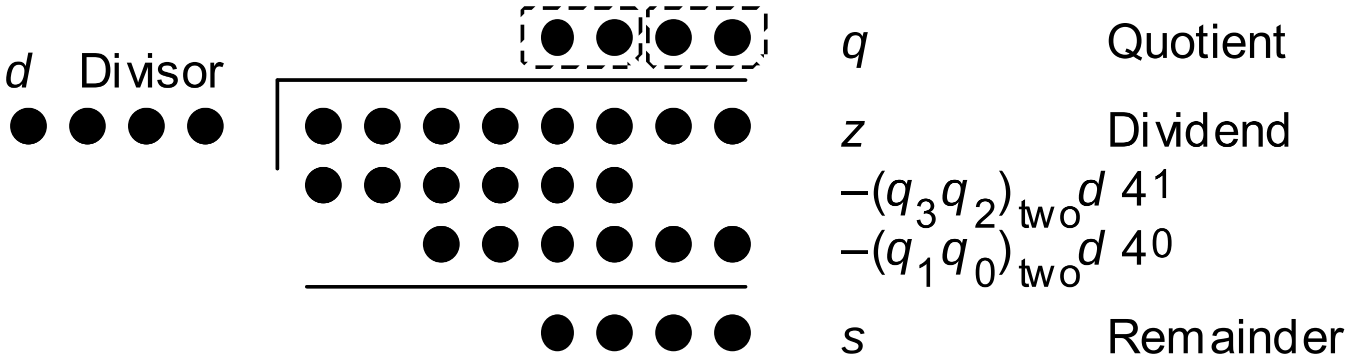
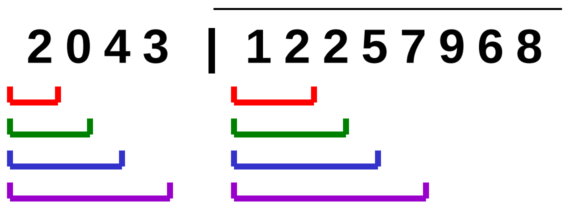

d Divisor
q Quotient
z Dividend
$-(q_3 q_2)_{\text{two}} d\, 4^1$
$-(q_1 q_0)_{\text{two}} d\, 4^0$
s Remainder

Fig. 14.1 Radix-4 division in dot notation

Difficulty of Quotient Digit Selection

What is the first quotient digit in the following radix-10 division?



2043 | 12257968

12 / 2 = 6

122 / 20 = 6

1225 / 204 = 6

12257 / 2043 = 5

The problem with the pencil-and-paper division algorithm is that there is no room for error in choosing the next quotient digit

In the worst case, all *k* digits of the divisor and *k* + 1 digits in the partial remainder are needed to make a correct choice

Suppose we used the redundant signed digit set [–9, 9] in radix 10

Then, we could choose 6 as the next quotient digit, knowing that we can recover from an incorrect choice by using negative digits: $5\ 9 = 6\ {}^{-}1$

Examples of High-Radix Division

Radix-4 integer division

z	0 1 2 3 1 1 2 3	
$4^4 d$	1 2 0 3	
$s^{(0)}$	0 1 2 3 1 1 2 3	
$4s^{(0)}$	0 1 2 3 1 1 2 3	
$-q_3 4^4 d$	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	
$-q_2 4^4 d$	0 0 0 0 0	$\{q_2 = 0\}$
$s^{(2)}$	0 2 2 1 2 3	
$4s^{(2)}$	0 2 2 1 2 3	
$-q_1 4^4 d$	0 1 2 0 3	$\{q_1 = 1\}$
$s^{(3)}$	1 0 0 3 3	
$4s^{(3)}$	1 0 0 3 3	
$-q_0 4^4 d$	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 Using Carry-Save Adders

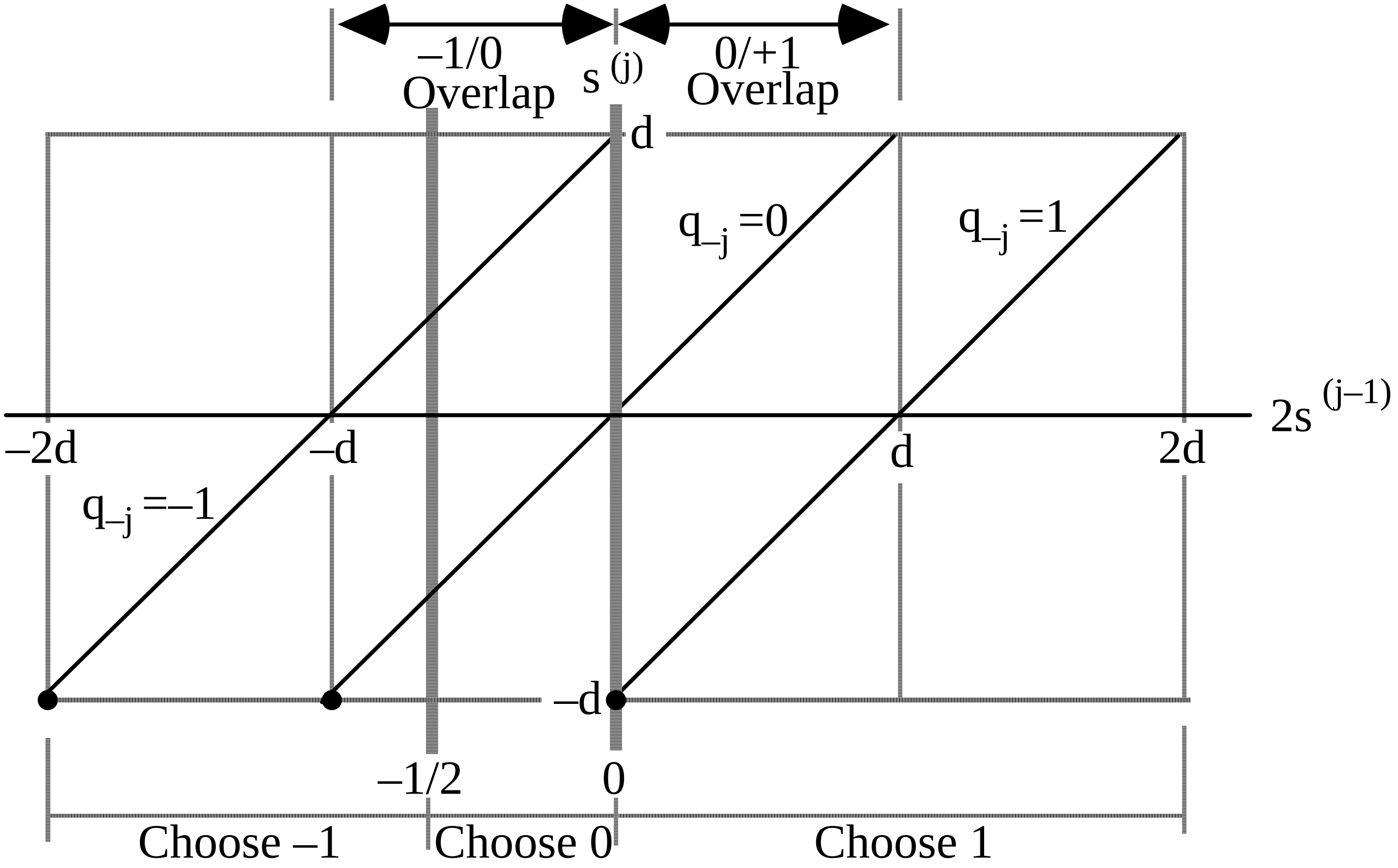


Fig. 14.3 Constant thresholds used for quotient digit selection in radix-2 division with q_{k-j} in {–1, 0, 1} .

Quotient Digit Selection Based on Truncated PR

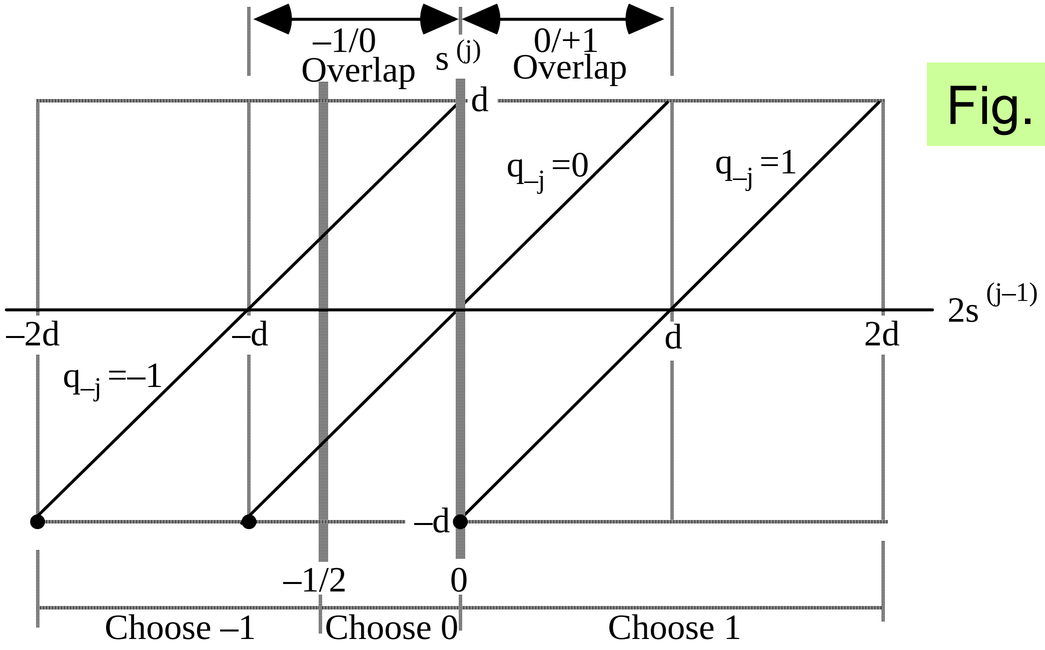


Fig. 14.3

```
$t := u_{[-2,1]} + v_{[-2,1]}$
if $t < -½$
then $q_{-j} = -1$
else if $t \geq 0$
        then $q_{-j} = 1$
        else $q_{-j} = 0$
        endif
endif
```

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

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

Approximation to the partial remainder:

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

Max error in approximation

< ¼ + ¼ = ½

Error in [0, ½)

Divider with Partial Remainder in Carry-Save Form

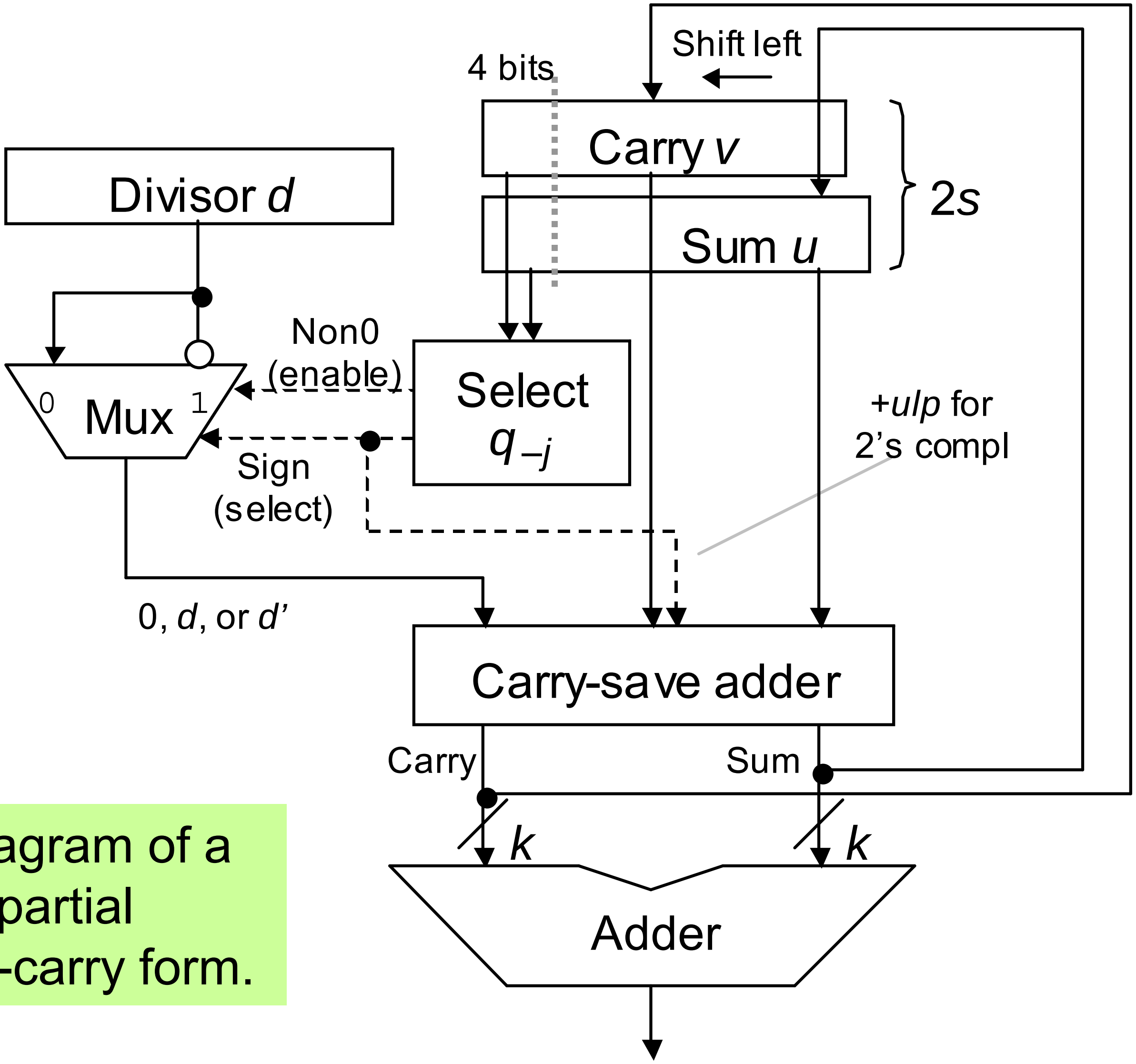


Fig. 14.4 Block diagram of a radix-2 divider with partial remainder in stored-carry form.

Why We Cannot Use Carry-Save PR with SRT Division

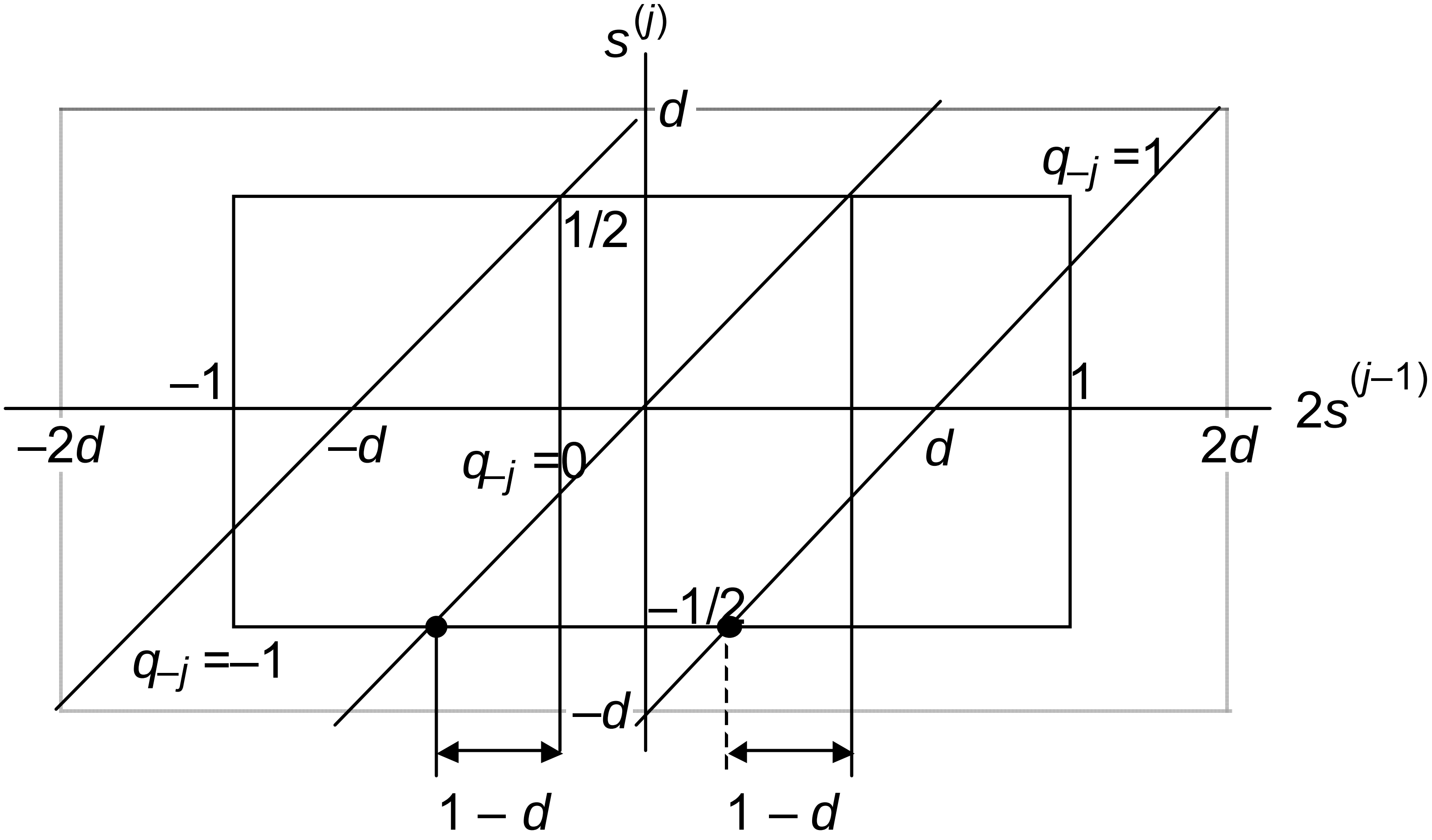
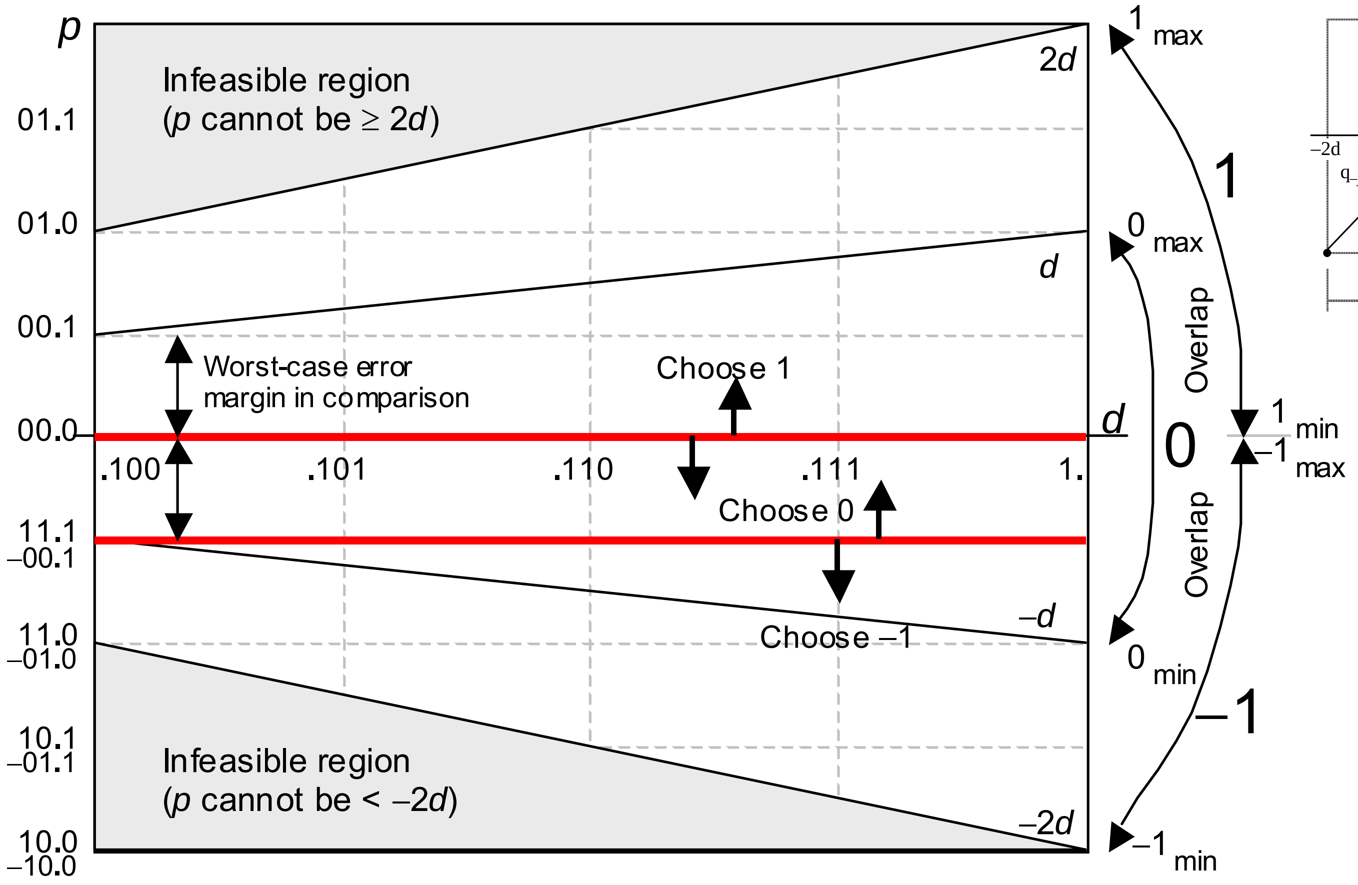


Fig. 14.5 Overlap regions in radix-2 SRT division.

14.4 Choosing the Quotient Digits



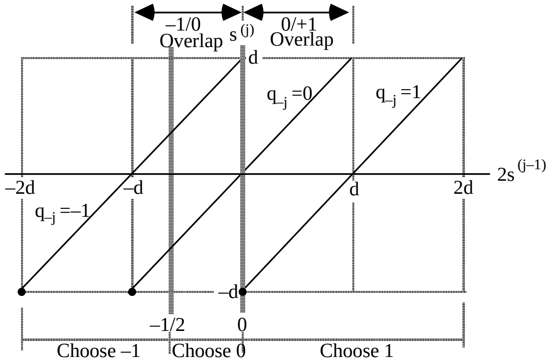
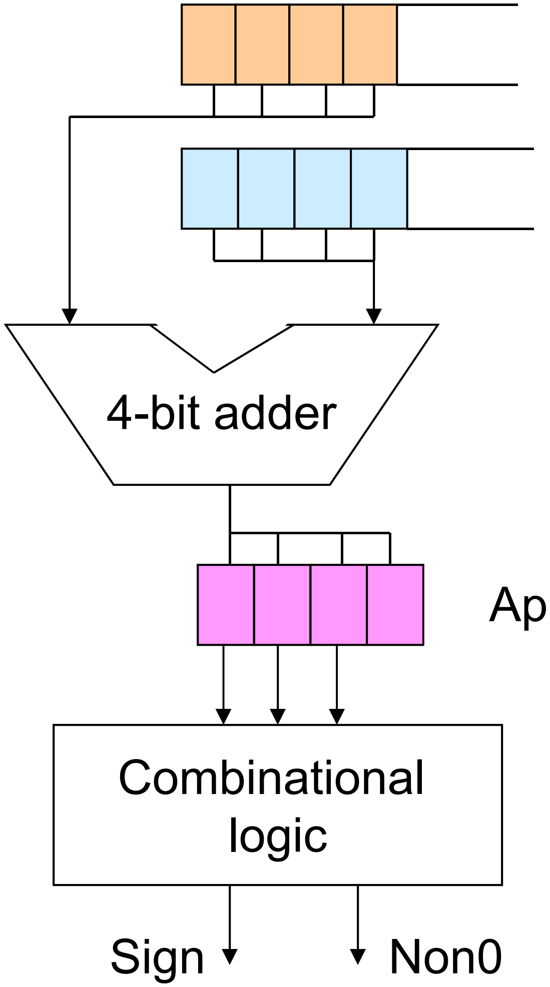


Fig. 14.3

Fig. 14.6 A *p*-*d* plot for radix-2 division with $d \in [1/2,1)$, partial remainder in $[–d, d)$, and quotient digits in $[–1, 1]$.

Design of the Quotient Digit Selection Logic



Shifted sum = $(u_1 u_0 \,.\, u_{-1} u_{-2} \,.\,.\,.\,)_{\text{2's-compl}}$

Shifted carry = $(v_1 v_0 \,.\, v_{-1} v_{-2} \,.\,.\,.\,)_{\text{2's-compl}}$

```
$t$ := $u_{[-2,1]} + v_{[-2,1]}$
if $t < -½$
then $q_{-j} = -1$
else if $t \geq 0$
        then $q_{-j} = 1$
        else $q_{-j} = 0$
        endif
endif
```

Approx shifted PR = $(t_1 t_0 \,.\, t_{-1} t_{-2})_{\text{2's-compl}}$

$$\text{Non0} \;=\; t_1' \vee t_0' \vee t_{-1}' = (t_1\, t_0\, t_{-1})'$$

$$\text{Sign} \;=\; t_1\,(t_0' \vee t_{-1}')$$

14.3 Radix-4 SRT Division

Radix-4 fractional division with left shifts and $q_{-j} \in [-3, 3]$

$s^{(j)} = 4s^{(j-1)} - q_{-j}\,d$ with $s^{(0)} = z$ and $s^{(k)} = 4^k s$

|-shift-|
|—subtract—|

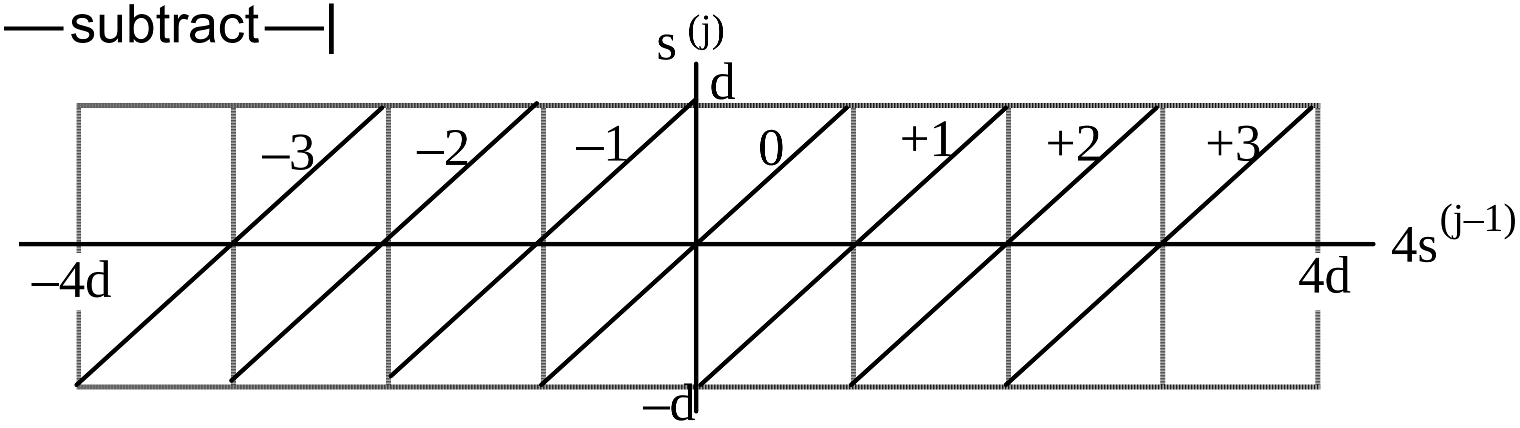


Fig. 14.7 New versus shifted old partial remainder in radix-4 division with q_{-j} in [–3, 3].

Two difficulties:

How do you choose from among the 7 possible values for q_{-j}?
If the choice is +3 or –3, how do you form $3d$?

Building the *p*-*d* Plot for Radix-4 Division

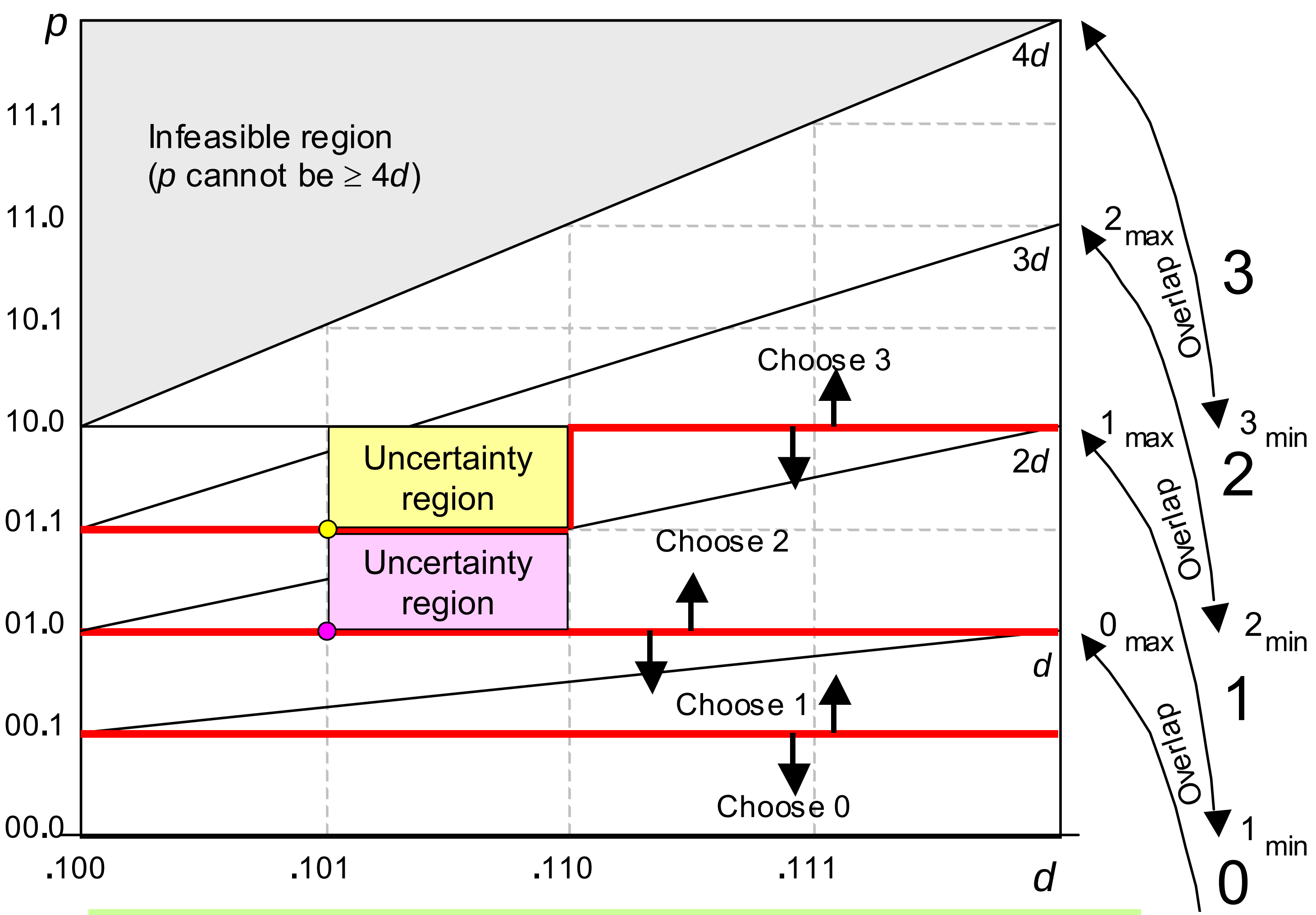


Fig. 14.8 A *p*-*d* plot for radix-4 SRT division with quotient digit set [–3, 3].

Restricting the Quotient Digit Set in Radix 4

Radix-4 fractional division with left shifts and $q_{-j} \in [-2, 2]$

$$s^{(j)} = \underbrace{\underbrace{4\,s^{(j-1)}}_{\text{shift}} - q_{-j}\,d}_{\text{subtract}} \qquad \text{with} \quad s^{(0)} = z \quad \text{and} \quad s^{(k)} = 4^k s$$

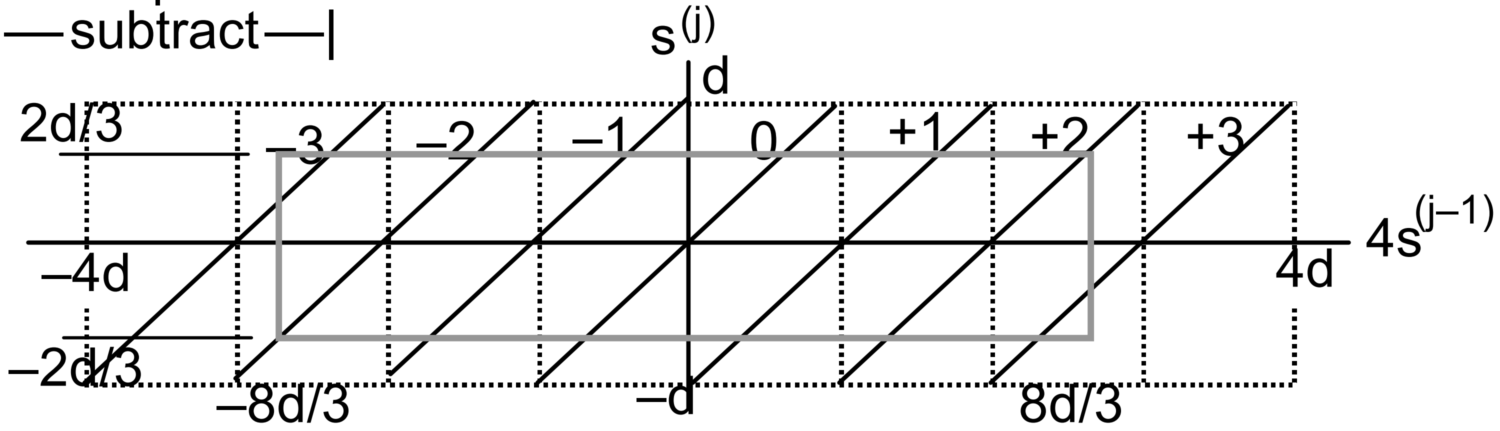


Fig. 14.9 New versus shifted old partial remainder in radix-4 division with q_{-j} in [–2, 2].

For this restriction to be feasible, we must have:

$s \in [-hd, hd)$ for some $h < 1$, and $4hd - 2d \le hd$

This yields $h \le 2/3$ (choose $h = 2/3$ to minimize the restriction)

Building the *p*-*d* Plot with Restricted Radix-4 Digit Set

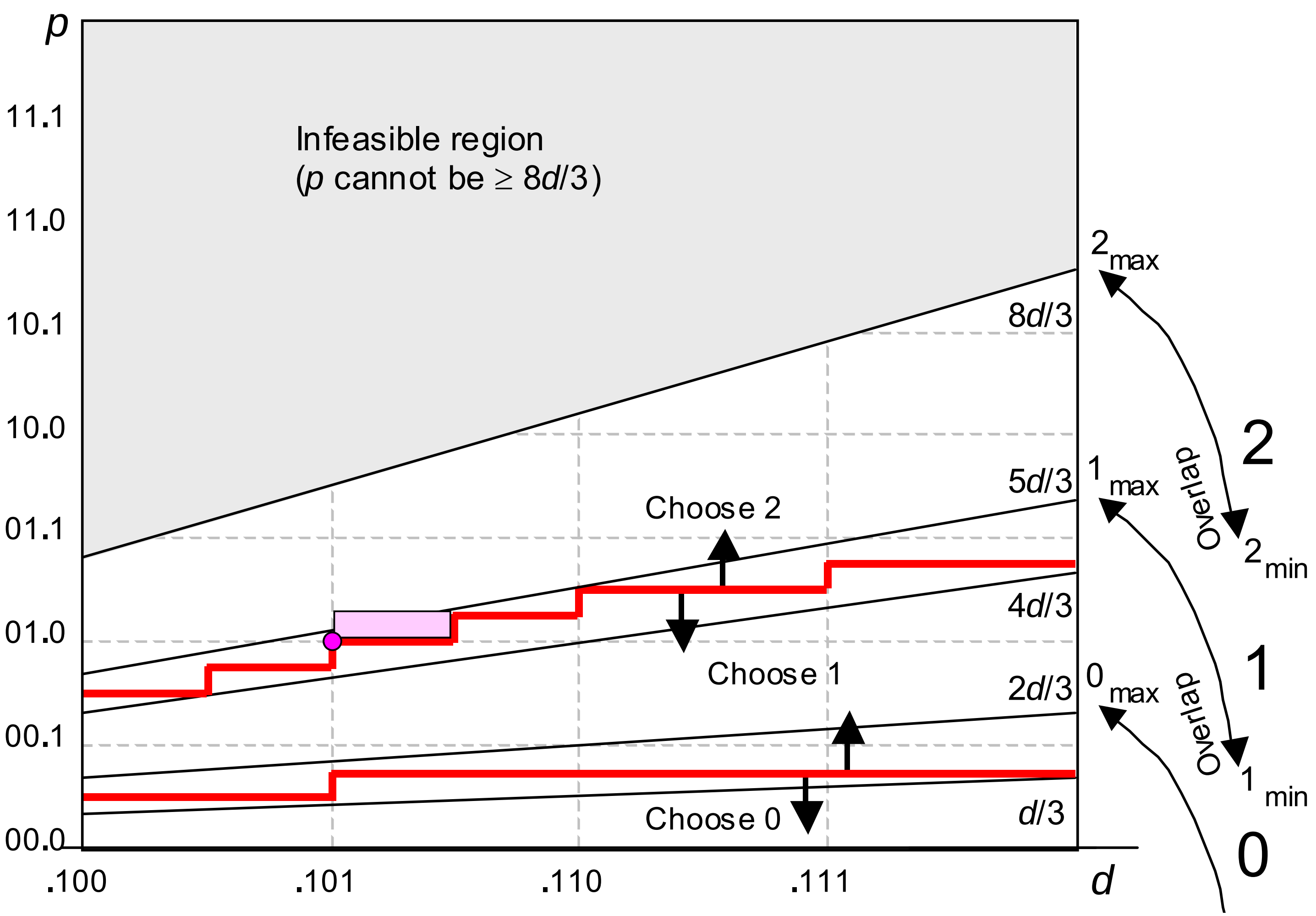
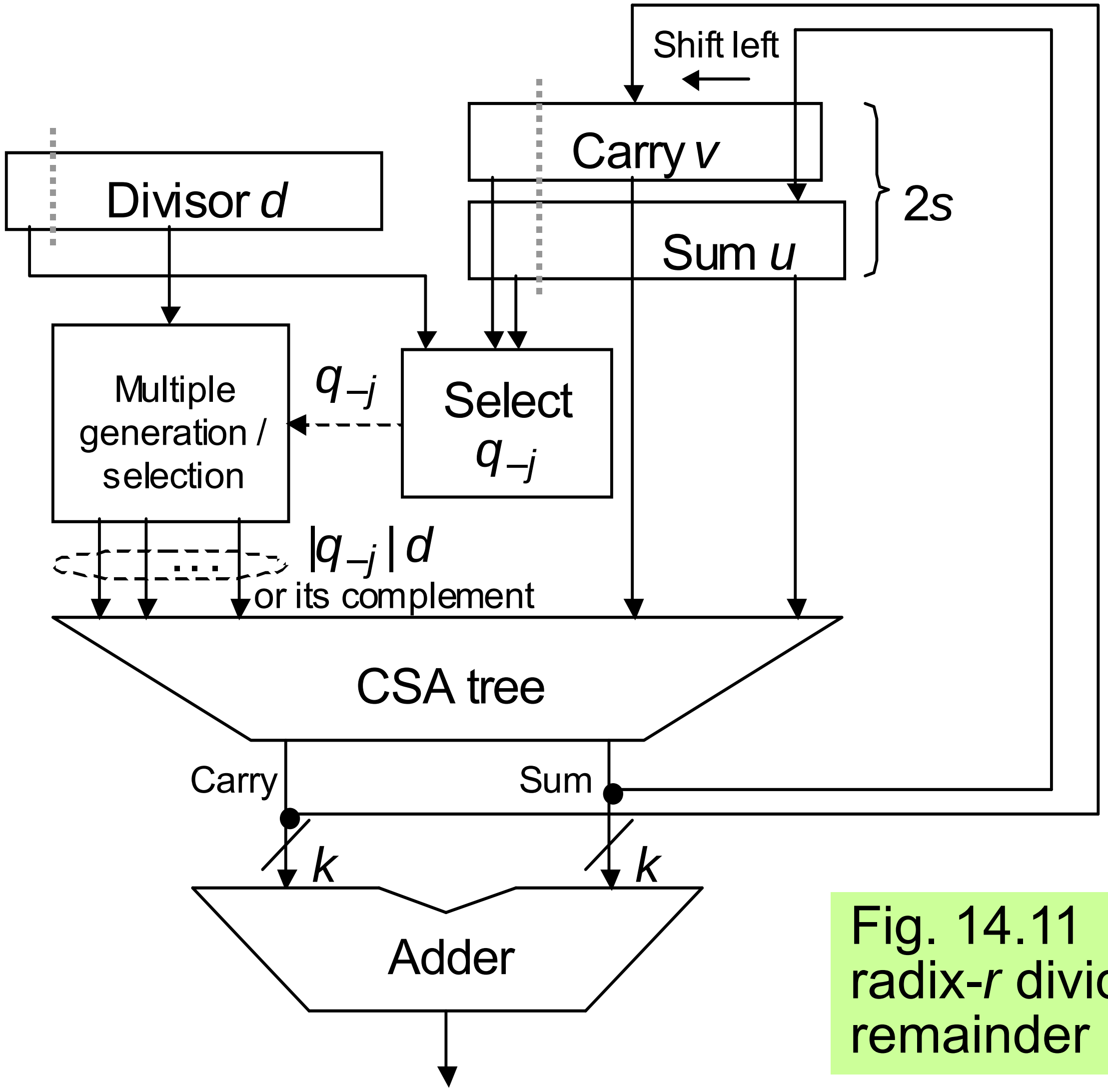


Fig. 14.10 A *p*-*d* plot for radix-4 SRT division with quotient digit set [–2, 2].

14.4 General High-Radix Dividers



Process to derive the details:

Radix r

Digit set $[-\alpha, \alpha]$ for q_{-j}

Number of bits of p (v and u) and d to be inspected

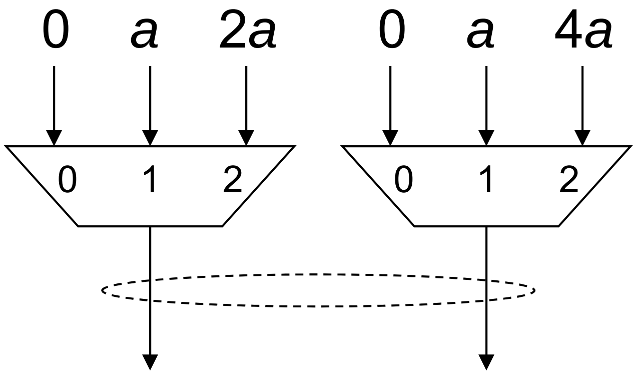
Quotient digit selection unit (table or logic)

Multiple generation/selection scheme

Conversion of redundant q to 2's complement

Fig. 14.11 Block diagram of radix-r divider with partial remainder in stored-carry form.

Multiple Generation for High-Radix Division



Example: Digit set [–6, 6] for r = 8

Option 1: precompute 3a and 5a

Option 2: generate a multiple $|q_{-j}|a$ as a set of two numbers, one chosen from {0, a, 2a} and another from {0, a, 4a}

14.5 Quotient Digit Selection

Radix-r division with quotient digit set $[-\alpha, \alpha]$, $\alpha < r - 1$
Restrict the partial remainder range, say to $[-hd, hd)$
From the solid rectangle in Fig. 15.1, we get $rhd - \alpha d \le hd$ or $h \le \alpha/(r - 1)$
To minimize the range restriction, we choose $h = \alpha/(r - 1)$

Example: $r = 4$, $\alpha = 2$ ➔ $h = 2/3$

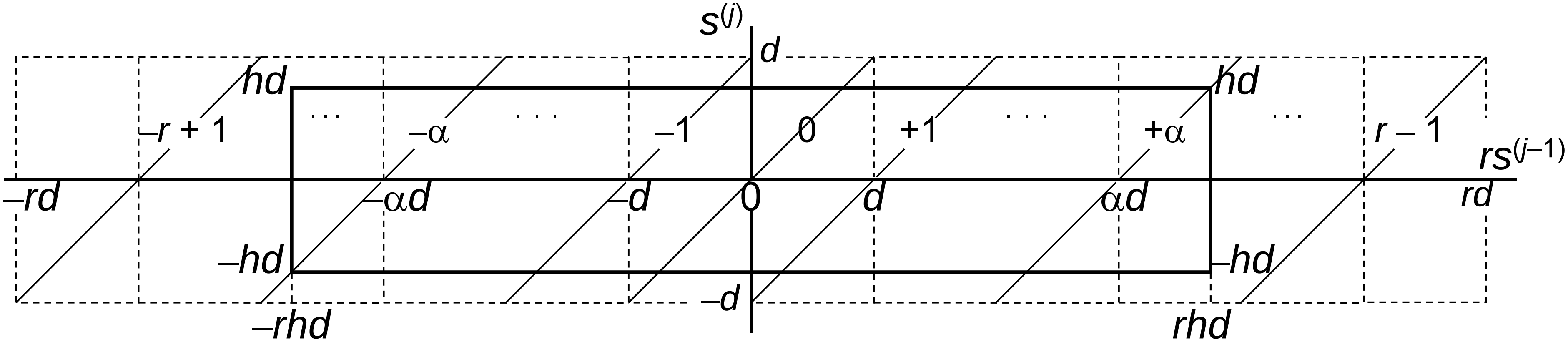
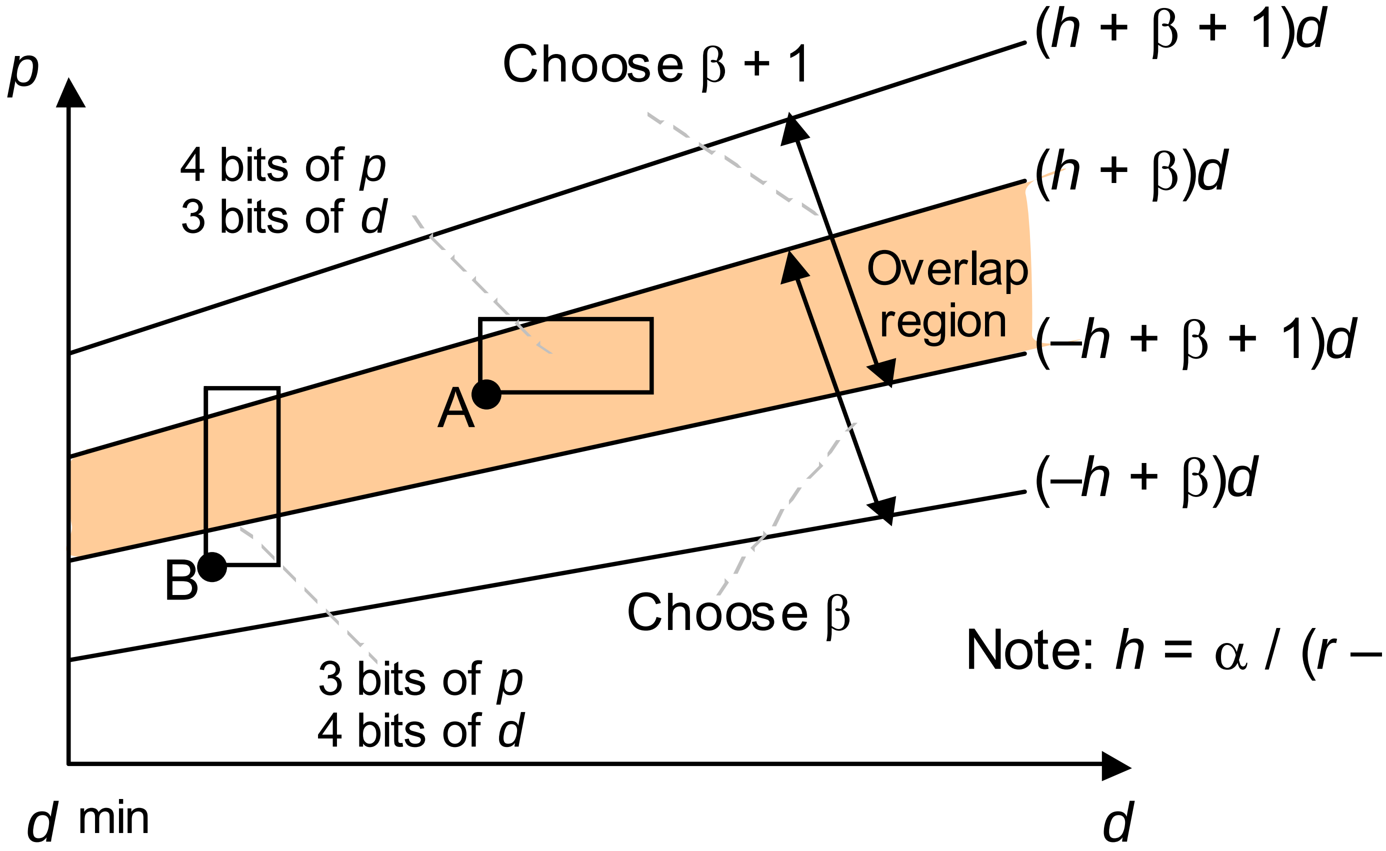


Fig. 14.12 The relationship between new and shifted old partial remainders in radix-r division with quotient digits in $[-\alpha, +\alpha]$.

Why Using Truncated *p* and *d* Values Is Acceptable



Standard *p*
xx.xxxx

Carry-save *p*
xx.xxxxx
xx.xxxxx

Note: $h = \alpha / (r - 1)$

Fig. 14.13 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].

Table Entries in the Quotient Digit Selection Logic

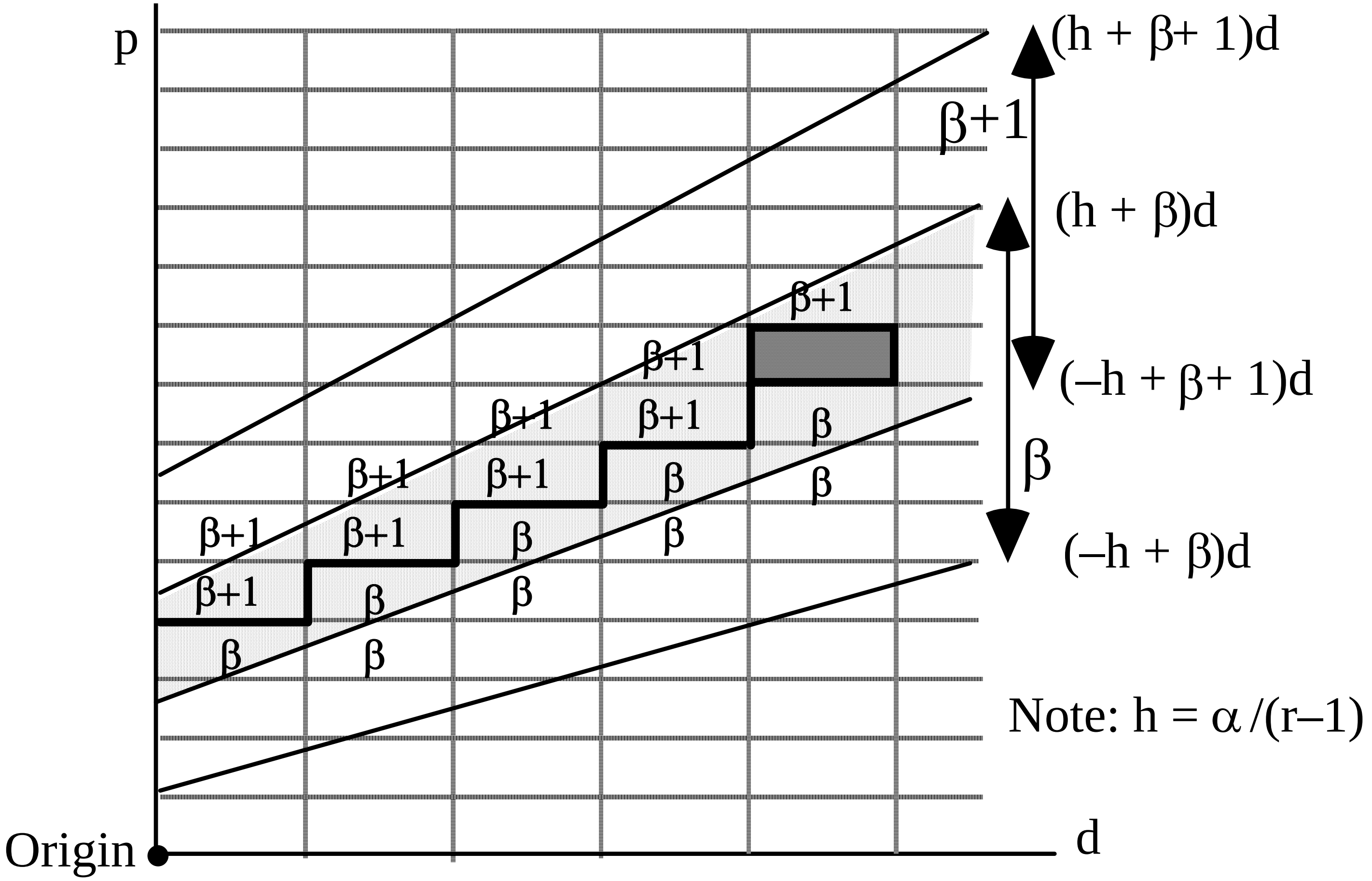
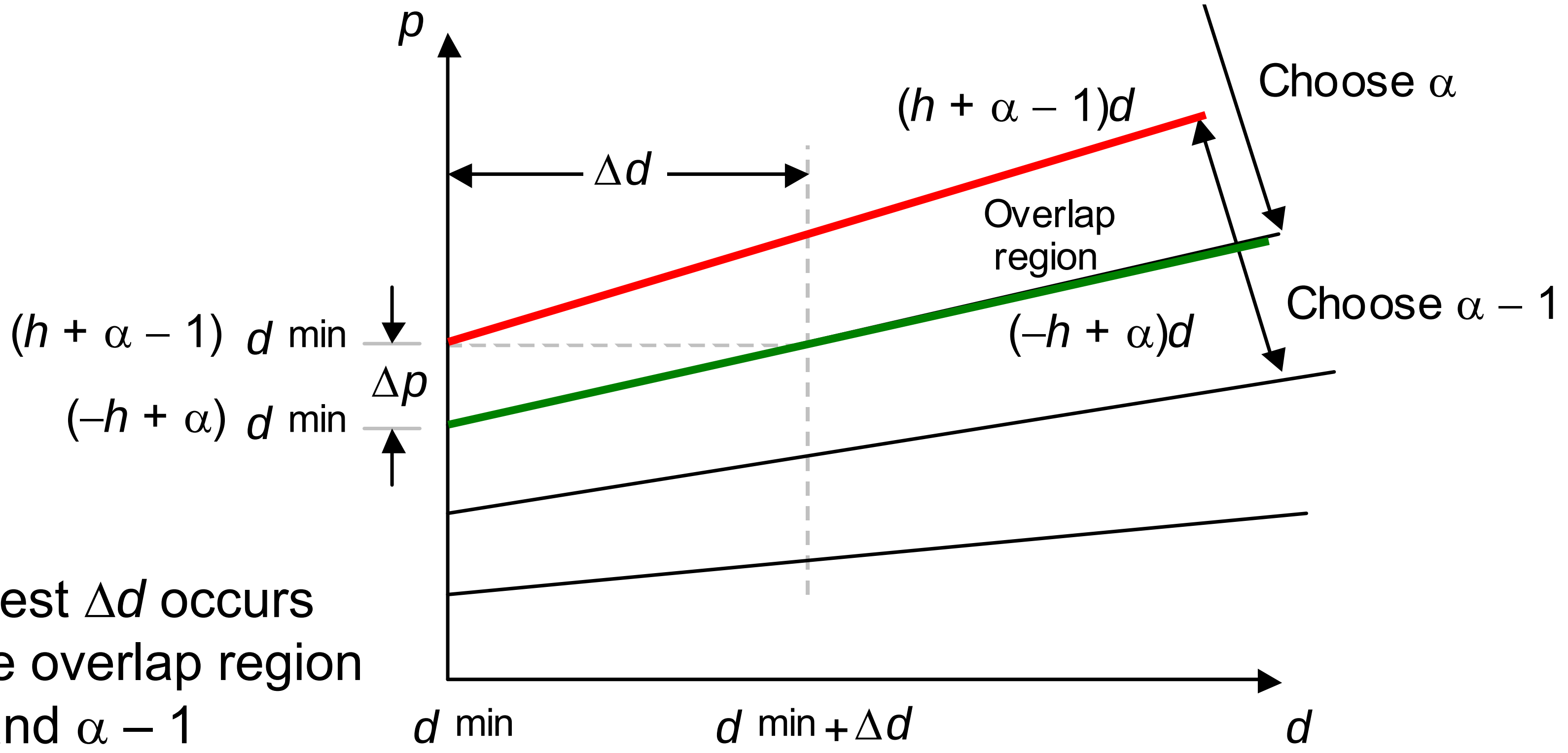


Fig. 14.14 A part of *p*-*d* plot showing an overlap region and its staircase-like selection boundary.

14.6 Using *p*-*d* Plots in Practice



Smallest Δ*d* occurs for the overlap region of α and α – 1

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

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

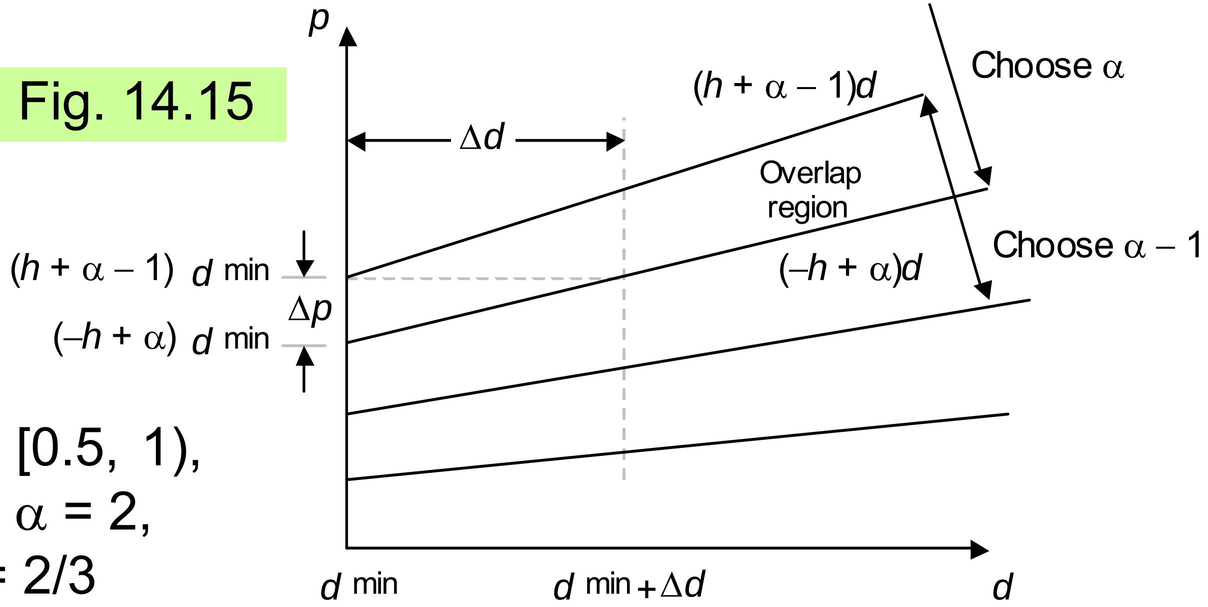
Fig. 14.15 Establishing upper bounds on the dimensions of uncertainty rectangles.

Example: Lower Bounds on Precision

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

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

Fig. 14.15



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

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

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

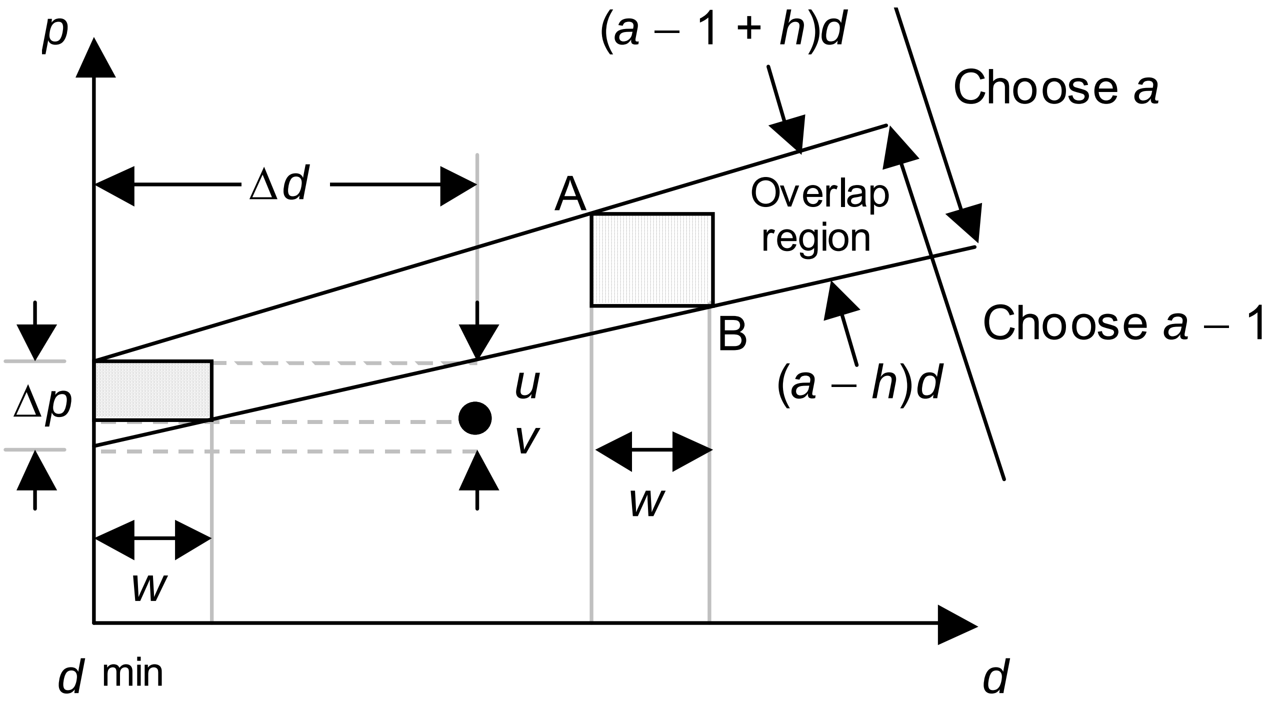
Because 1/8 = 2^{-3} and $2^{-3} \leq 1/6 < 2^{-2}$, we must inspect at least 3 bits of d (2, given its leading 1) and 3 bits of p

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

Upper Bounds for Precision



Theorem: Once lower bounds on precision are determined based on Δ*d* and Δ*p*, one more bit of precision in each direction is always adequate

Proof: Let *w* be the spacing of vertical grid lines
$w \le \Delta d/2$ ⇒ $v \le \Delta p/2$ ⇒ $u \ge \Delta p/2$

Some Implementation Details

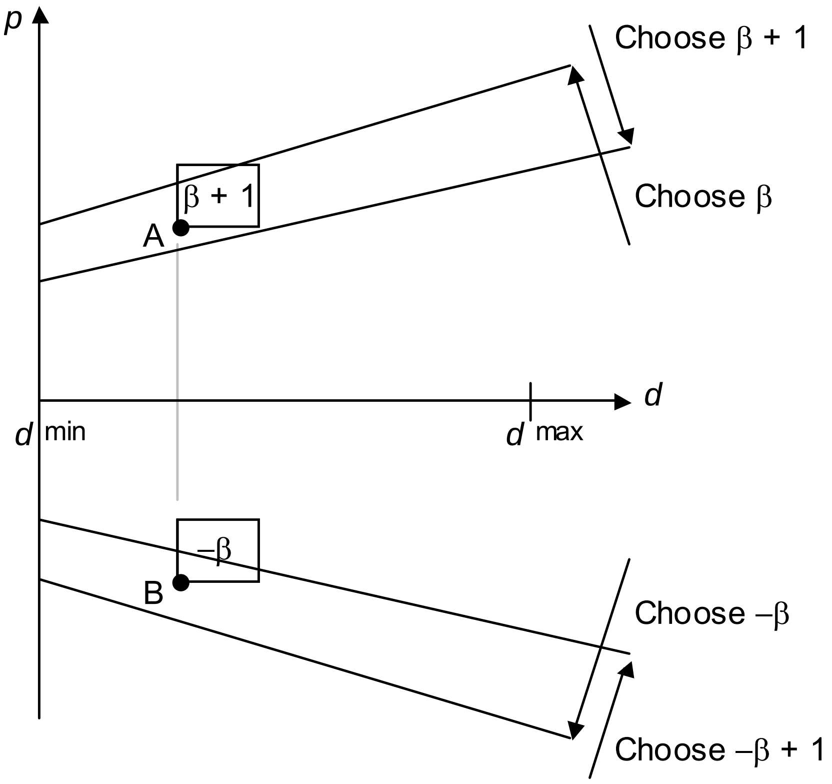


Fig. 14.16 The asymmetry of quotient digit selection process.

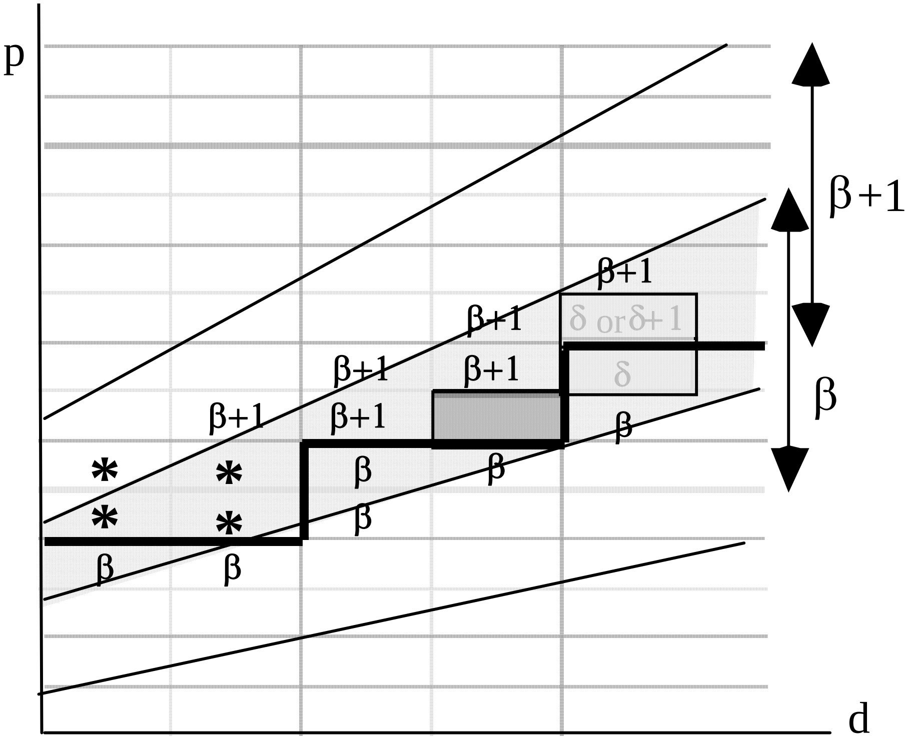
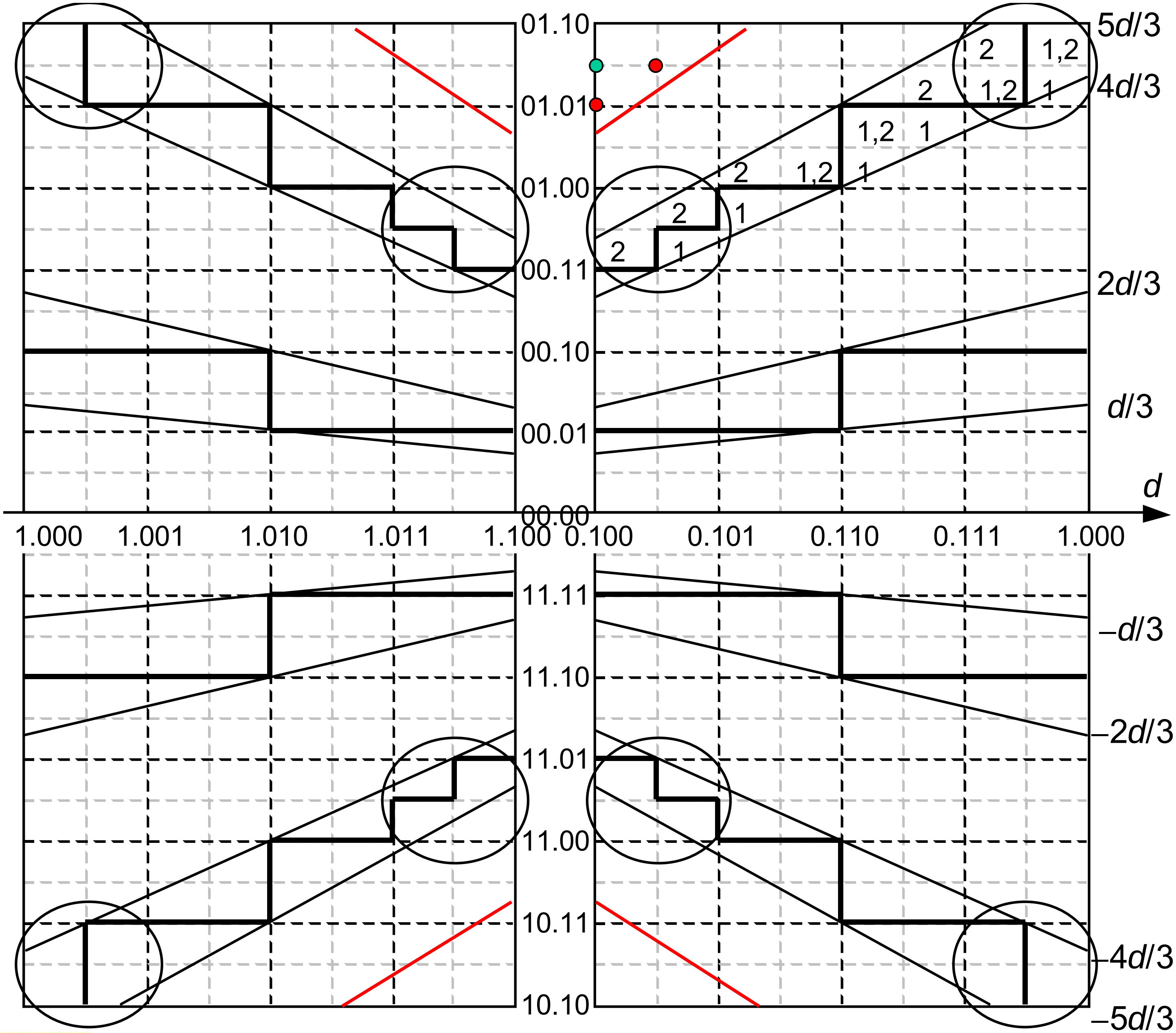


Fig. 14.17 Example of *p*-*d* plot allowing larger uncertainty rectangles, if the 4 cases marked with asterisks are handled as exceptions.

A Complete *p*-*d* Plot

Radix $r = 4$
q_{-j} in [–2, 2]
d in [1/2, 1)
p in [–8/3, 8/3]

Explanation of the Pentium division bug



15 Variations in Dividers

Chapter Goals

Discuss some variations in implementing division schemes and cover combinational, modular, and merged hardware dividers

Chapter Highlights

Prescaling simplifies *q* digit selection
Overlapped *q* digit selection
Parallel hardware (array) dividers
Shared hardware in multipliers/dividers
Square-rooting not special case of division

Variations in Dividers: Topics

Topics in This Chapter
15.1 Division with Prescaling
15.2 Overlapped Quotient Digit Selection
15.3 Combinational and Array Dividers
15.4 Modular Dividers and Reducers
15.5 The Special Case of Reciprocation
15.6 Combined Multiply/Divide Units

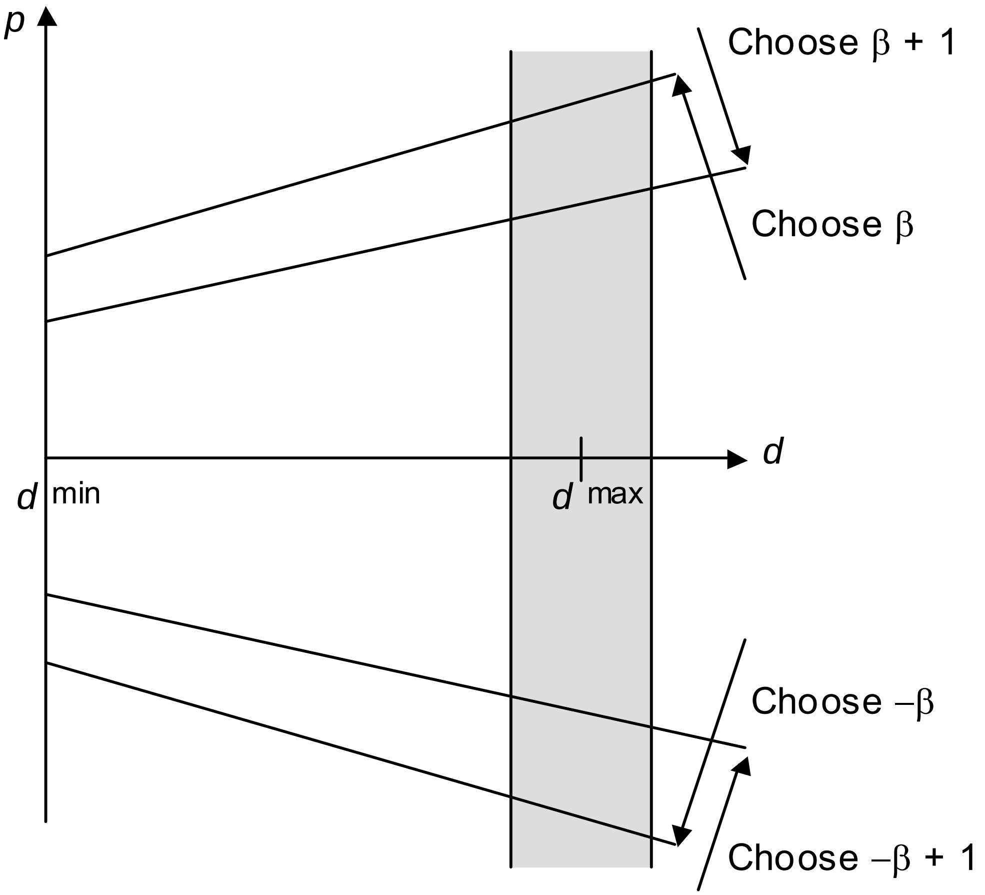
15.1 Division with Prescaling

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

Thus, we perform the division $(zm)/(dm)$ for a suitably chosen scale factor m ($m > 1$)

Prescaling (multiplying *z* and *d* by *m*) should be done without real multiplications



Restricting the divisor to the shaded area simplifies quotient digit selection.

Examples of Prescaling

Example 1: Unsigned divisor *d* in [1/2, 1)

When *d* ∈ [1/2, 3/4), multiply by 1½ [*d* begins 0.10…]

The prescaled divisor will be in [1 – 1/4, 1 + 1/8)

Example 2: Unsigned divisor *d* in [1/2, 1)

Case *d* ∈

[1/2, 9/16), it begins with 0.1000…, multiply by 2
[9/16, 5/8), it begins with 0.1001…, multiply by 1 + 1/2
[5/8, 3/4), it begins with 0.101…, multiply by 1 + 1/2
[3/4, 1), it begins with 0.11…, multiply by 1 + 1/8

[1/2, 9/16) × 2 = [1, 1 + 1/8)
[9/16, 5/8) × (1 + 1/2) = [1 – 5/32, 1 – 1/16)
[5/8, 3/4) × (1 + 1/2) = [1 – 1/16, 1 + 1/8)
[3/4, 1) × (1 + 1/8) = [1 – 5/32, 1 + 1/8)

The prescaled divisor will be in [1 – 5/32, 1 + 1/8)

15.2 Overlapped Quotient Digit Selection

Alternative to high-radix design when *q* digit selection is too complex

Compute the next partial remainder and resulting *q* digit for all possible choices of the current *q* digit

This is the same idea as carry-select addition

Speculative computation (throw transistors at the delay problem) is common in modern systems

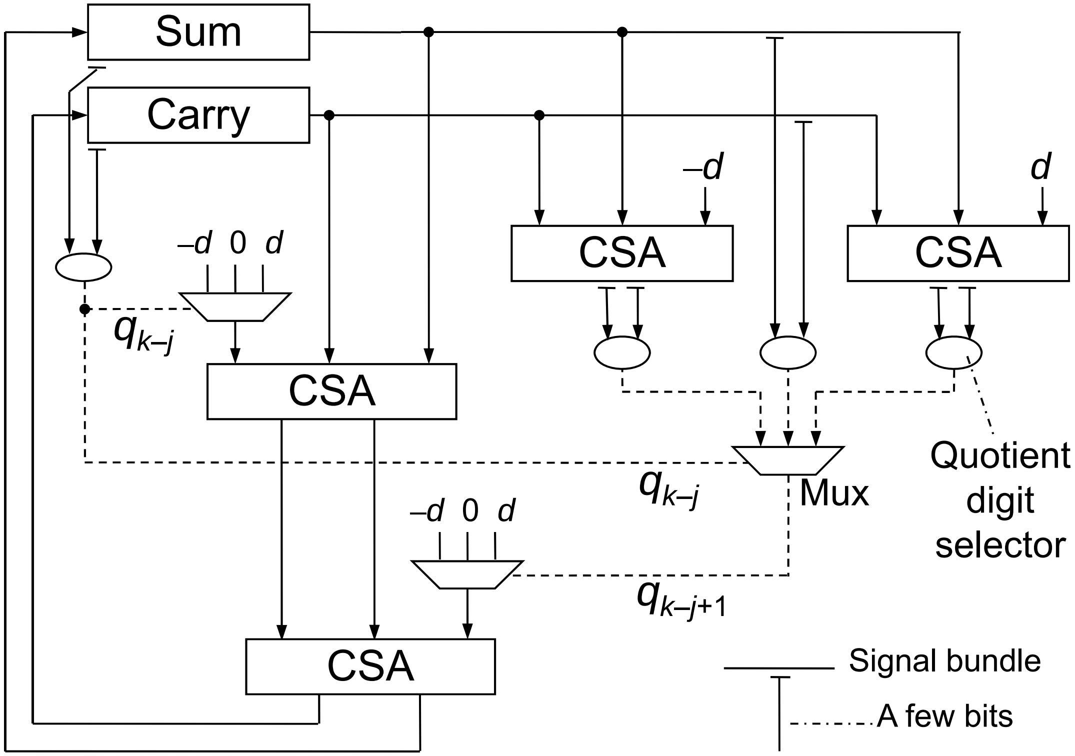


Fig. 15.1 Overlapped radix-2 quotient digit selection for radix-4 division. A dashed line represents a signal pair that denotes a quotient digit value in [–1, 1].

15.3 Combinational and Array Dividers

Can take the notion of overlapped q digit selection to the extreme of selecting all q digits at once → Exponential complexity

By contrast, a fully combinational tree multiplier
has $O(\log k)$ latency and $O(k^2)$ cost
$O(k \log k)$ conjectured

Can we do as well as multipliers, or at least better than exponential cost, for logarithmic-time dividers?

Complexity theory results: It is possible to design dividers
with $O(\log k)$ latency and $O(k^4)$ cost
with $O(\log k \log \log k)$ latency and $O(k^2)$ cost

These theoretical constructions have not led to practical designs

Restoring Array Divider

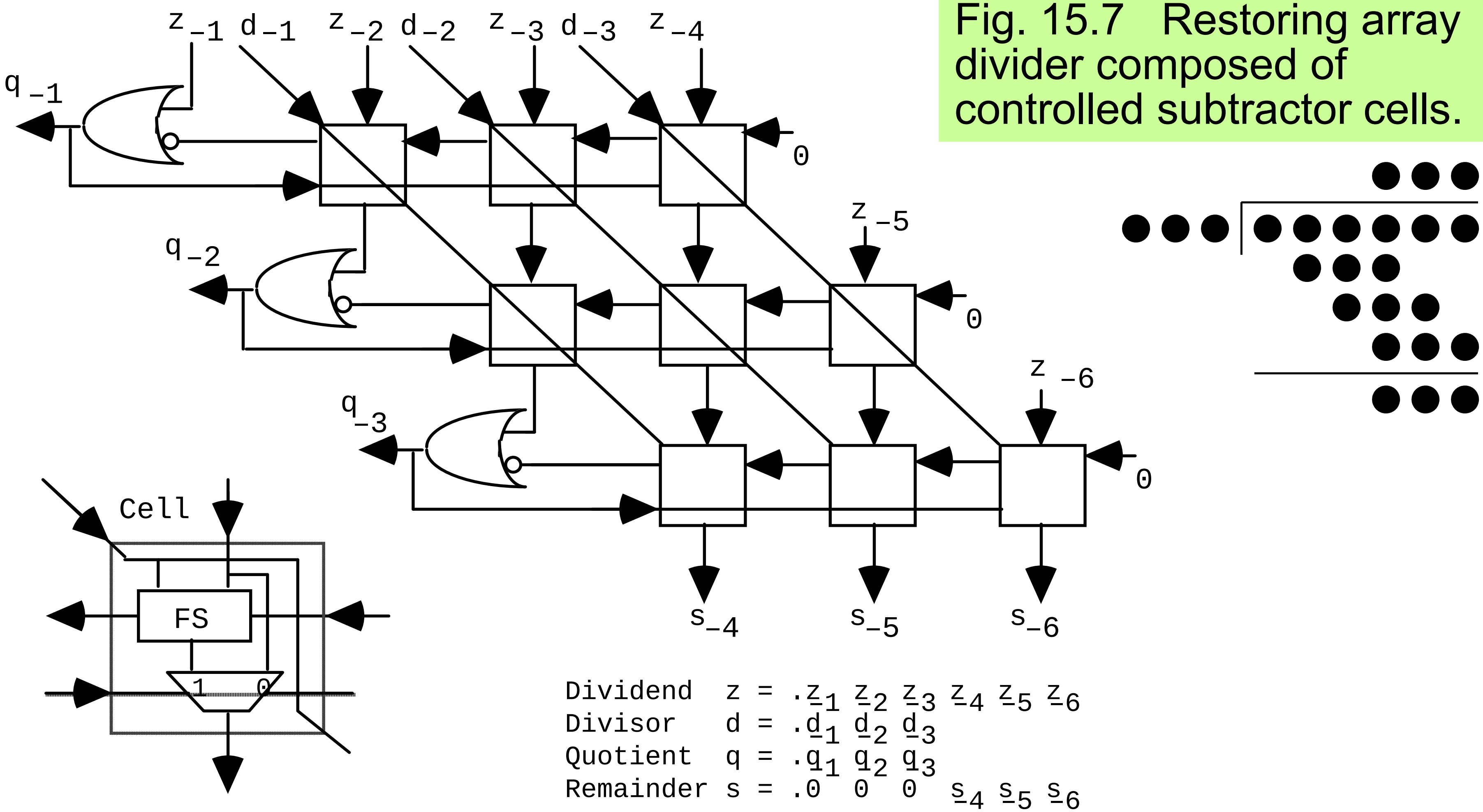


Fig. 15.7 Restoring array divider composed of controlled subtractor cells.

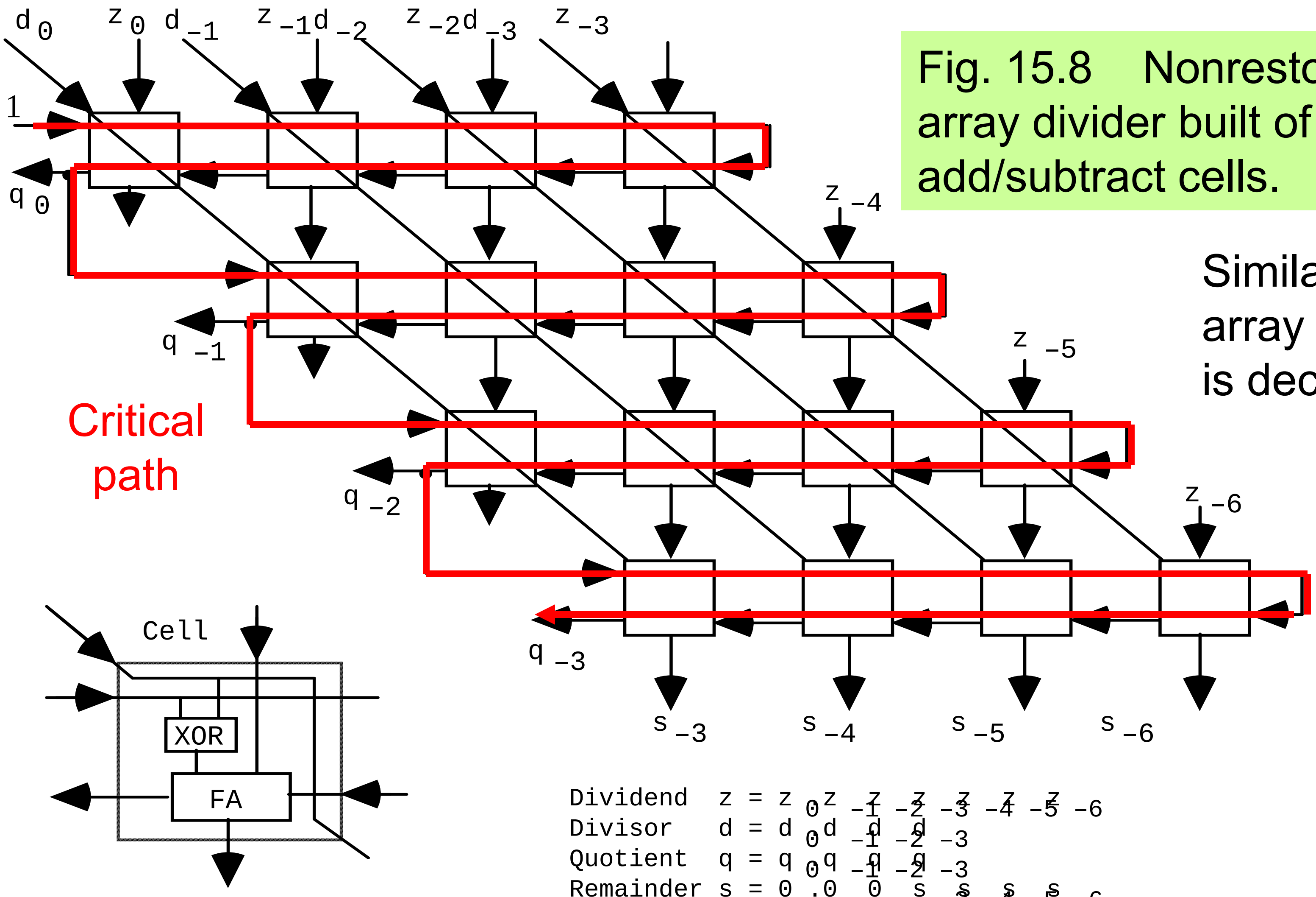
Dividend $z = .z_{-1}\ z_{-2}\ z_{-3}\ z_{-4}\ z_{-5}\ z_{-6}$

Divisor $d = .d_{-1}\ d_{-2}\ d_{-3}$

Quotient $q = .q_{-1}\ q_{-2}\ q_{-3}$

Remainder $s = .0\ \ 0\ \ 0\ \ s_{-4}\ s_{-5}\ s_{-6}$

Nonrestoring Array Divider



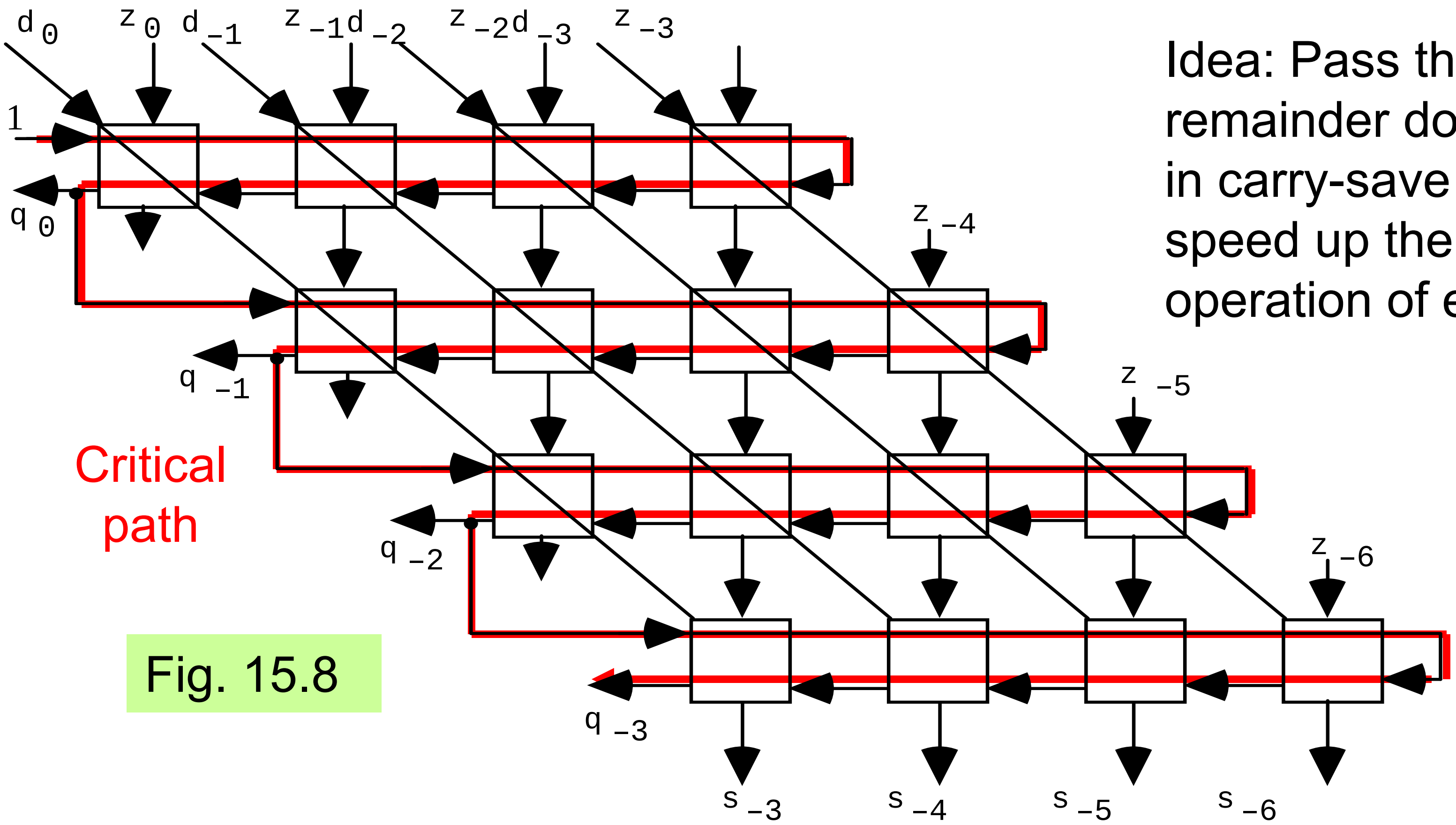
Critical path

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

Similarity to array multiplier is deceiving

```
Dividend  z = z0 z-1 z-2 z-3 z-4 z-5 z-6
Divisor   d = d0 d-1 d-2 d-3
Quotient  q = q0 q-1 q-2 q-3
Remainder s = 0 .0  0  s-3 s-4 s-5 s-6
```

Speedup Methods for Array Dividers



Critical path

Fig. 15.8

Idea: Pass the partial remainder downward in carry-save form to speed up the operation of each row

However, we still need to know the carry/borrow-out from each row

Solution: Insert a carry-lookahead circuit between successive rows

Not very cost-effective; thus not used in practice

15.4 Modular Dividers and Reducers

Given 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; the modular remainder is always positive

Example:

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

The quotient and remainder of ordinary division are –3 and –0.07

A modular reducer computes only the modular remainder and is in many cases simpler than a full-blown divider

Montgomery Modular Reduction

Very efficient for reducing large numbers (100s of bits wide)
The radix-2 version below is suitable for low-cost hardware realization
Software versions are based on radix 2^{32} or 2^{64} (1 word = 1 digit)

Problem: Compute $q = ax \bmod m$, where $m < 2^k$

Straightforward solution: Compute *ax* as usual; then reduce mod *m*

Incremental reduction after adding each partial product is more efficient

Assume *a*, *x*, *q*, and other values are *k*-bit pseudoresidues (can be > *m*)

Pick *R* such that $R = 1 \bmod m$
Montgomery multiplication computes $axR^{-1} \bmod m$, instead of $ax \bmod m$
Represent any number *y* as $yR \bmod m$ (known as the M-code for *y*)
$R = 1 \bmod m$ ensures that numbers in $[0, m - 1]$ have distinct M-codes

Multiplication: $t = (aR)(xR)R^{-1} \bmod m = (ax)R \bmod m$ = M-code for *ax*
Initial conversion: Find *yR* by applying Montgomery's method to *y* and R^2
Final reconversion: Find *y* from $t = yR$ by M-multiplying 1 and *t*

Example Montgomery Modular Multiplication

Fig. 15.4

a		1	0	1	0				
2^4x		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

(a) Ordinary

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	Even
$p^{(1)}$		0	1	0	1	
$+x_1a$		1	0	1	0	
$2p^{(2)}$	0	1	1	1	1	Odd
$+13$		1	1	0	1	
$2p^{(2)}$	1	1	1	0	0	
$p^{(2)}$		1	1	1	0	
$+x_2a$		0	0	0	0	
$2p^{(3)}$	0	1	1	1	0	Even
$p^{(3)}$		0	1	1	1	
$+x_3a$		1	0	1	0	
$2p^{(4)}$	1	0	0	0	1	Odd
$+13$		1	1	0	1	
$2p^{(4)}$	1	1	1	1	0	
$p^{(4)}$		1	1	1	1	

(b) Mod 13

Example: $r = 2$; $m = 13$;
$R = 16 = r^4$; $R^{-1} = 9 \bmod 13$
(because $16 \times 9 = 1 \bmod 13$)

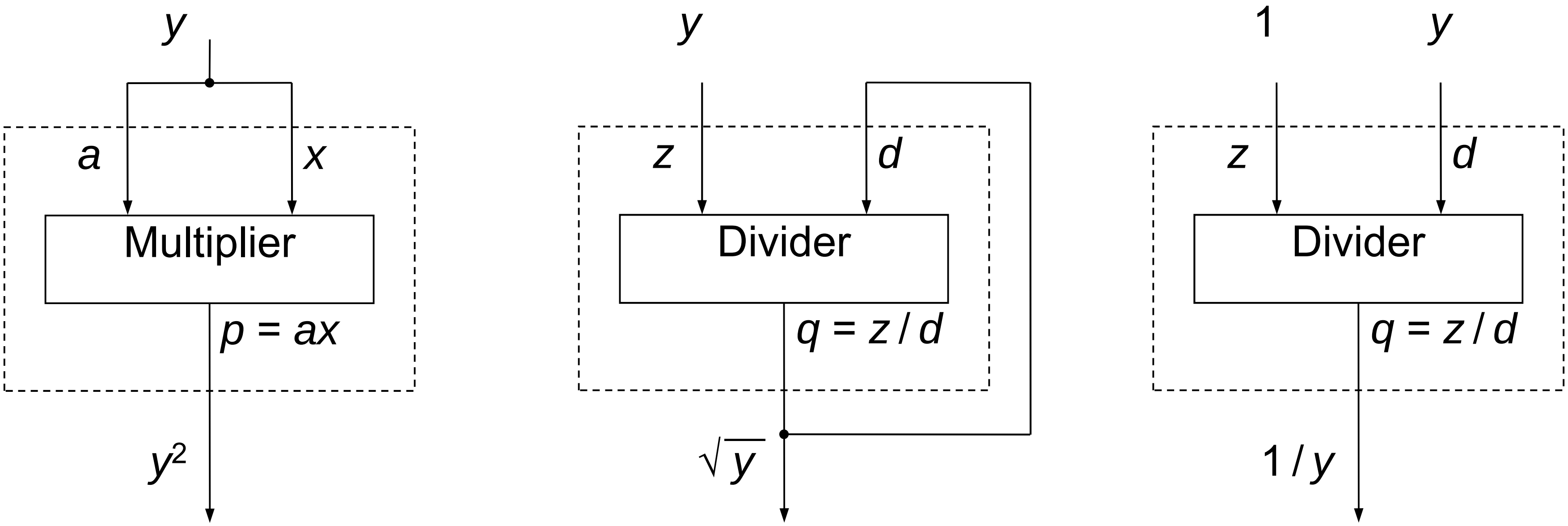
Advantages of Montgomery’s Method

Standard reduction is based on subtracting a multiple of *m* from the result depending on the most significant bit(s)

However, MSBs are not readily known if we use carry-save numbers

In Montgomery reduction, the decision is based on LSB(s), thus allowing the use of carry-save arithmetic as well as parallel processing

15.5 The Special Case of Reciprocation



(a) Squaring

(b) Square-rooting?

(c) Reciprocation

Fig. 15.5 Square-rooting is not a special case of division, but reciprocation is.

Key question: Is reciprocation any faster than division?
Answer: Not if a conventional digit recurrence algorithm is used

Doubling the Speed of Reciprocation

$Q \approx 1/d$ with error $\leq 2^{-k/2}$

$t = Q(2 - Qd) \approx 1/d$; error $\leq 2^{-k}$

$s^{(j+1)} = 2s^{(j)} - q_{-j}\,d$, with $2s^{(0)} = 1$

$t^{(j+1)} = 4t^{(j)} + q_{-j}(4s^{(j)} - q_{-j}\,d)$, with $t^{(0)} = 0$

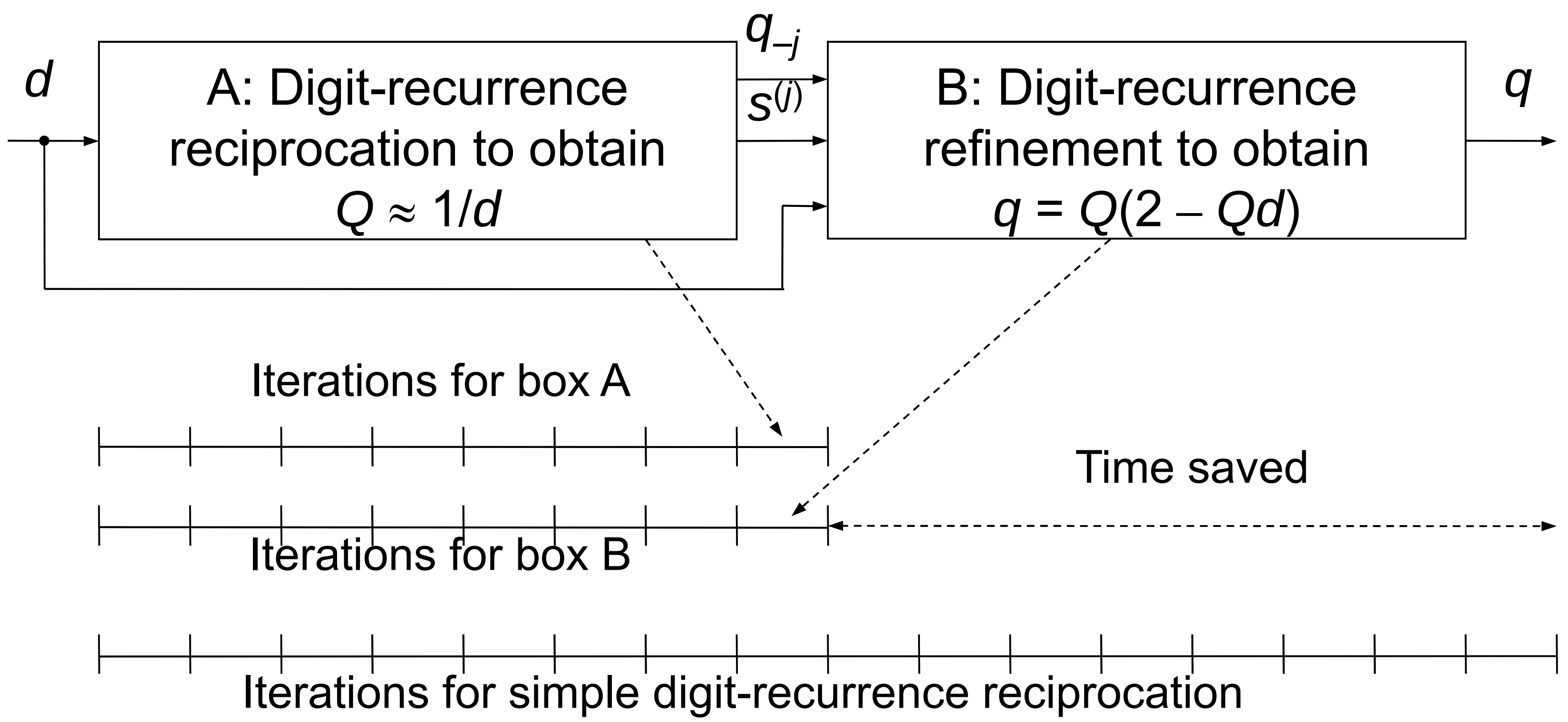

d
A: Digit-recurrence reciprocation to obtain $Q \approx 1/d$
q_{-j}
$s^{(j)}$
B: Digit-recurrence refinement to obtain $q = Q(2 - Qd)$
q
Iterations for box A
Time saved
Iterations for box B
Iterations for simple digit-recurrence reciprocation

Fig. 15.6 Hybrid evaluation of the reciprocal 1/*d* by an approximate reciprocation stage and a refinement stage that operate concurrently.

15.6 Combined Multiply/Divide Units

Similarity of blocks in multipliers and dividers (only shift direction is different)

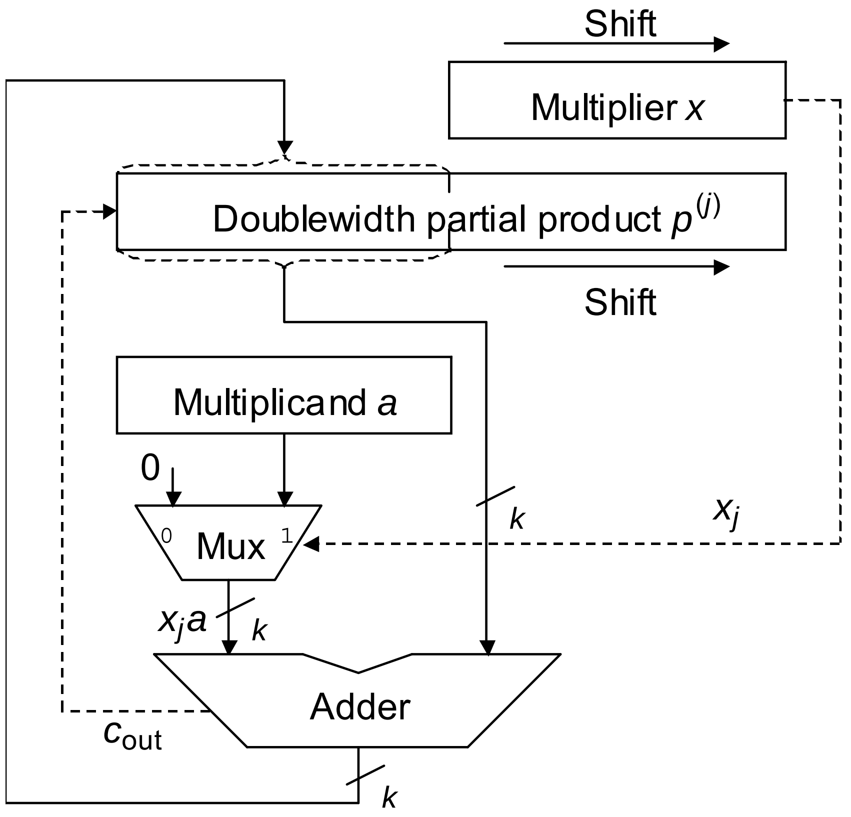


Fig. 9.4

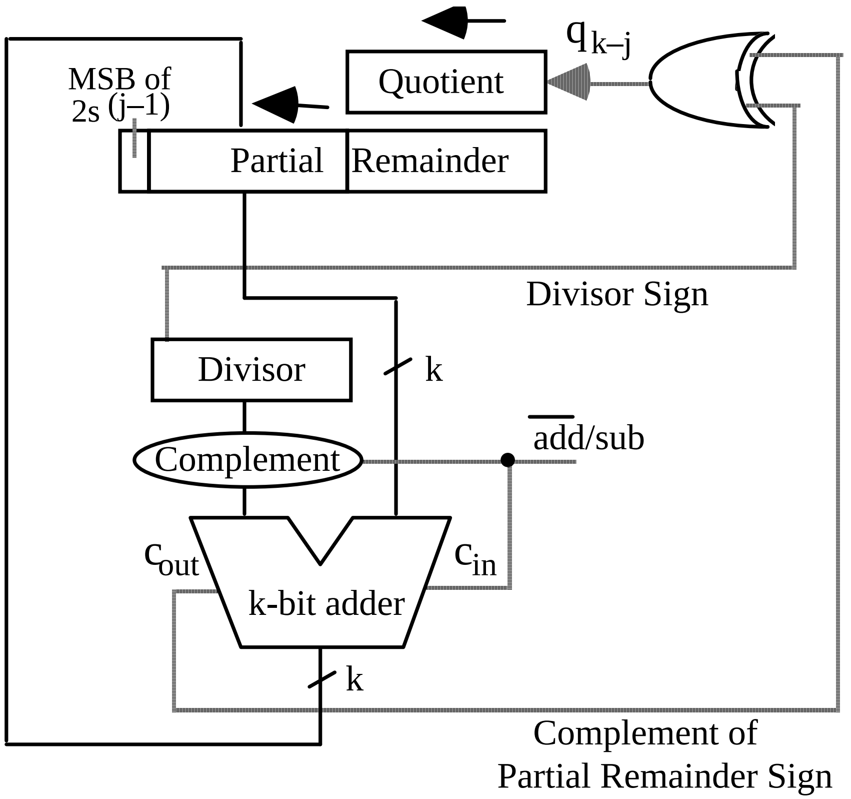


Fig. 13.10

Single Unit for Sequential Multiplication and Division

The control unit proceeds through necessary steps for multiplication or division (including using the appropriate shift direction)

The slight speed penalty owing to a more complex control unit is insignificant

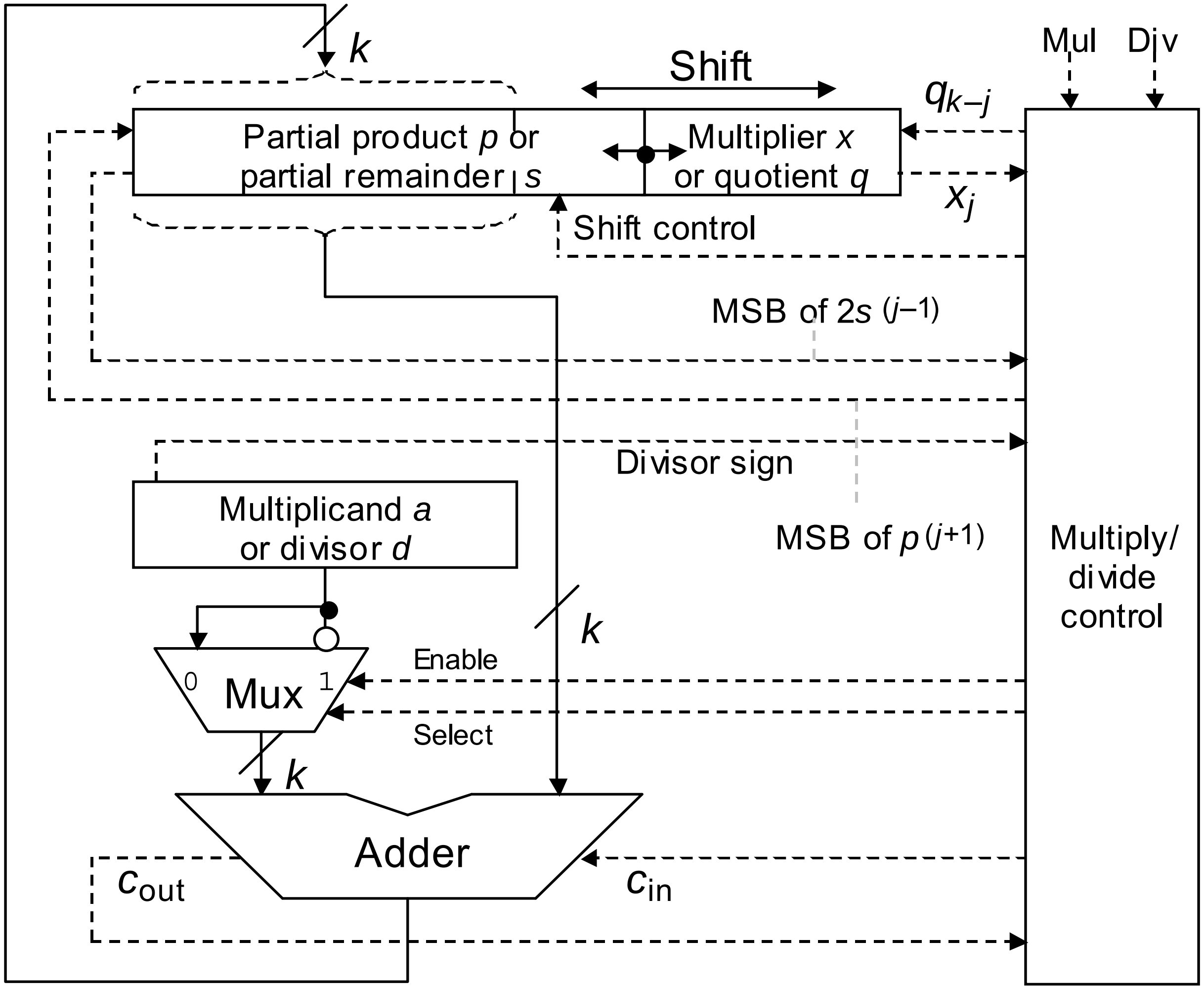


Fig. 15.9 Sequential radix-2 multiply/divide unit.

Similarities of Array Multipliers and Array Dividers

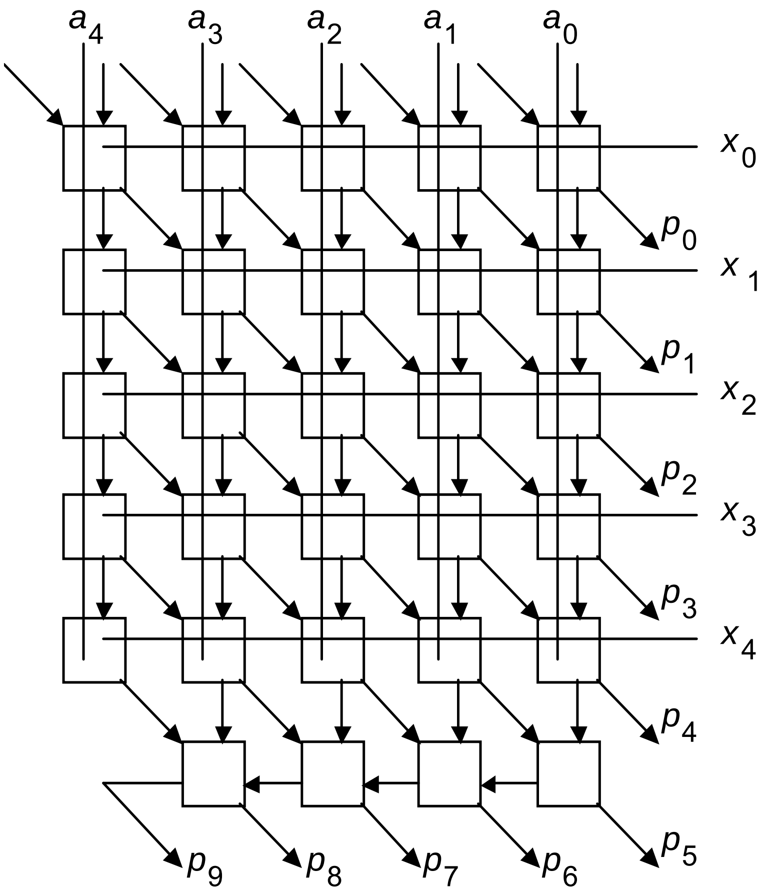
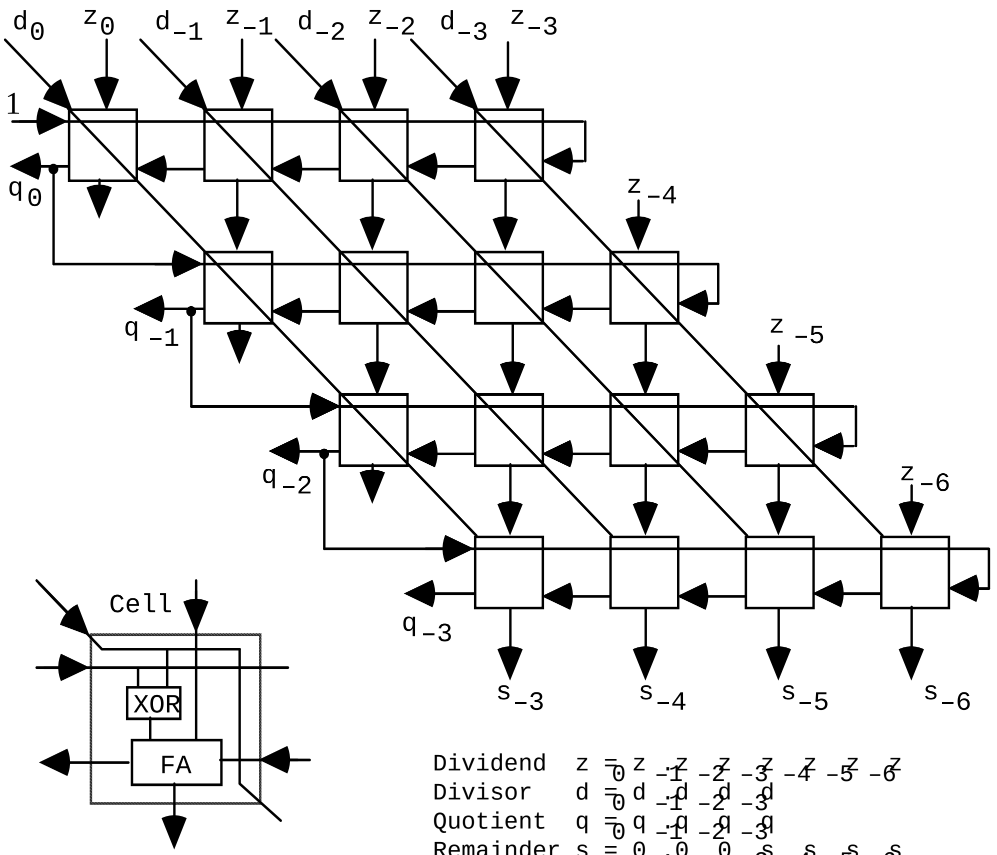


Fig. 11.4



Dividend $z = z_0 . z_{-1} z_{-2} z_{-3} z_{-4} z_{-5} z_{-6}$

Divisor $d = d_0 . d_{-1} d_{-2} d_{-3}$

Quotient $q = q_0 . q_{-1} q_{-2} q_{-3}$

Remainder $s = 0 .0 \; 0 \; s_{-3} s_{-4} s_{-5} s_{-6}$

Fig. 15.8

Single Unit for Array Multiplication and Division

Each cell within the array can act as a modified adder or modified subtractor based on control input values

In some designs, squaring and square-rooting functions are also included within the same array

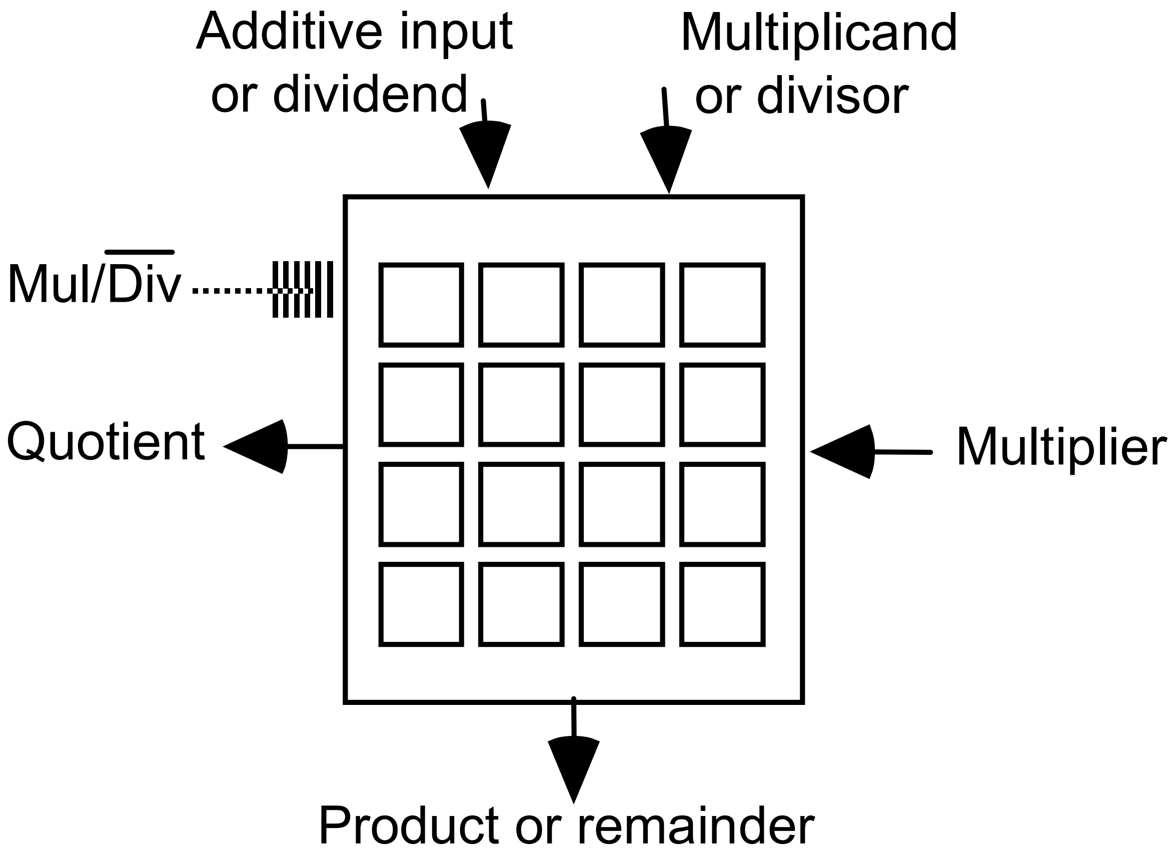

Additive input or dividend
Multiplicand or divisor
$\mathrm{Mul}/\overline{\mathrm{Div}}$
Quotient
Multiplier
Product or remainder

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 division step, the number of iterations can be reduced

Chapter Highlights

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

Division by Convergence: Topics

Topics in This Chapter
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

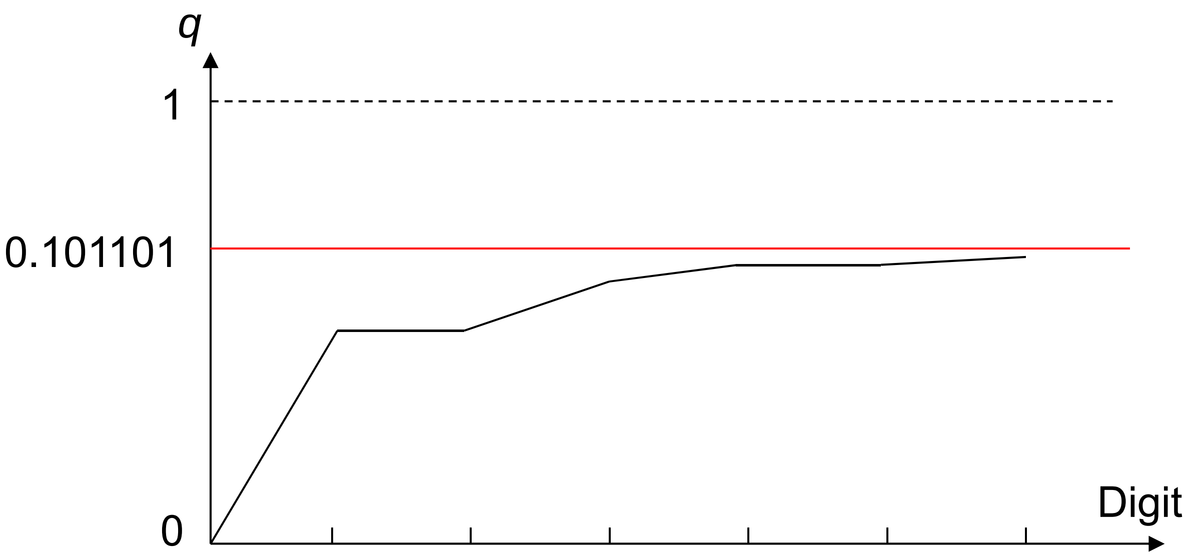
16.1 General Convergence Methods

Sequential digit-at-a-time (binary or high-radix) division can be viewed as a convergence scheme

As each new digit of $q = z / d$ is determined, the quotient value is refined, until it reaches the final correct value

Convergence is from below in restoring division and oscillating in nonrestoring division

Meanwhile, the remainder $s = z - q \times d$ approaches 0; the scaled remainder is kept in a certain range, such as $[-d, d)$



Elaboration on Scaled Remainder in Division

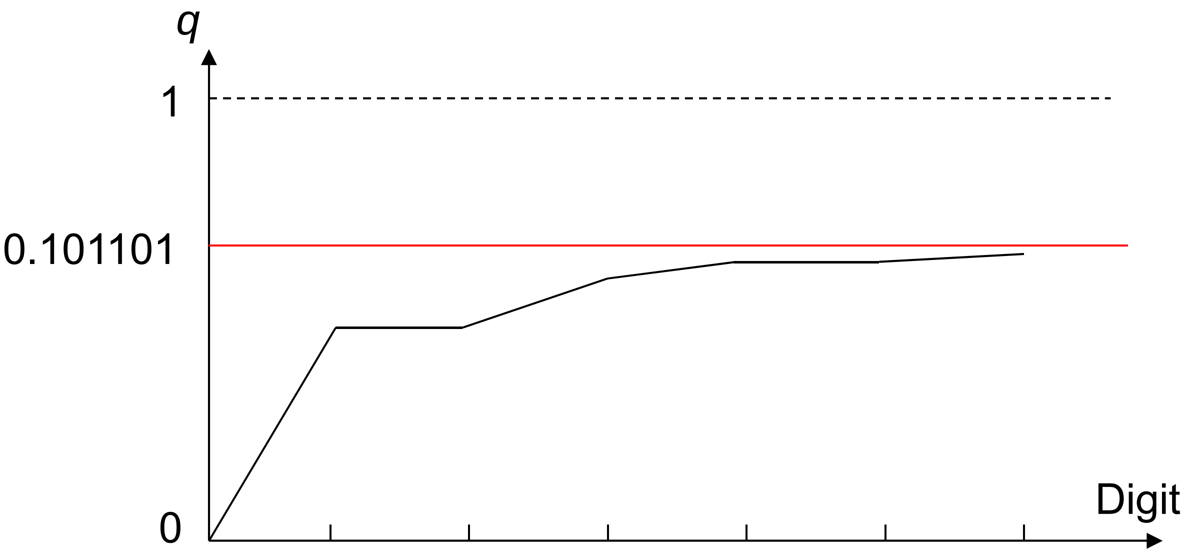
The partial remainder $s^{(j)}$ in division recurrence isn't the true remainder but a version scaled by 2^j

Division with left shifts

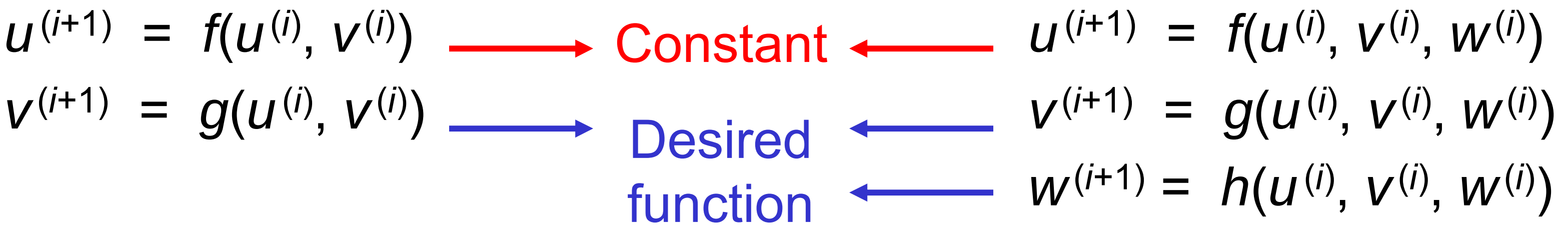
$$s^{(j)} = \underbrace{\underbrace{2s^{(j-1)}}_{\text{shift}} - q_{k-j}(2^k d)}_{\text{subtract}}$$

with $s^{(0)} = z$ and $s^{(k)} = 2^k s$

Quotient digit selection keeps the scaled remainder bounded (say, in the range $-d$ to d) to ensure the convergence of the true remainder to 0



Recurrence Formulas for Convergence Methods



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

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

Constant

Desired function

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

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

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

Guide the iteration such that one of the values converges to a constant (usually 0 or 1)

The other value then converges to the desired function

The complexity of this method depends on two factors:

a. Ease of evaluating *f* and *g* (and *h*)
b. Rate of convergence (number of iterations needed)

16.2 Division by Repeated Multiplications

Motivation: Suppose add takes 1 clock and multiply 3 clocks;
64-bit divide takes 64 clocks in radix 2, 32 in radix 4

→ Divide via multiplications faster if 10 or fewer needed

Idea:

$$q = \frac{z}{d} = \frac{zx^{(0)}x^{(1)}\cdots x^{(m-1)}}{dx^{(0)}x^{(1)}\cdots x^{(m-1)}} \begin{array}{l} \longrightarrow \text{Converges to } q \\ \longrightarrow \text{Force to 1} \end{array}$$

Remainder often not needed, but can be obtained by another multiplication if desired: $s = z - qd$

To turn the identity 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?

Formulation as a Convergence Computation

Idea:

$$q = \frac{z}{d} = \frac{zx^{(0)}x^{(1)}\cdots x^{(m-1)}}{dx^{(0)}x^{(1)}\cdots x^{(m-1)}}$$

Numerator $\longrightarrow$ Converges to q

Denominator $\longrightarrow$ Force to 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 $z/d = q \cong z^{(m)}$

Question 1: 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 $d^{(m)} \cong 1$

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

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

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

Fits the general form

Determining the Rate of Convergence

$d^{(i+1)} = d^{(i)}(2 - d^{(i)})$ Set $d^{(0)} = d$; make $d^{(m)}$ converge to 1

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

Question 2: How quickly does $d^{(i)}$ converge to 1?

We can relate the error in step $i + 1$ to the error in step i:

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

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

For $1 - d^{(i)} \le \varepsilon$, we get $1 - d^{(i+1)} \le \varepsilon^2$: *Quadratic convergence*

In general, for k-bit operands, we need

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

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

Quadratic Convergence

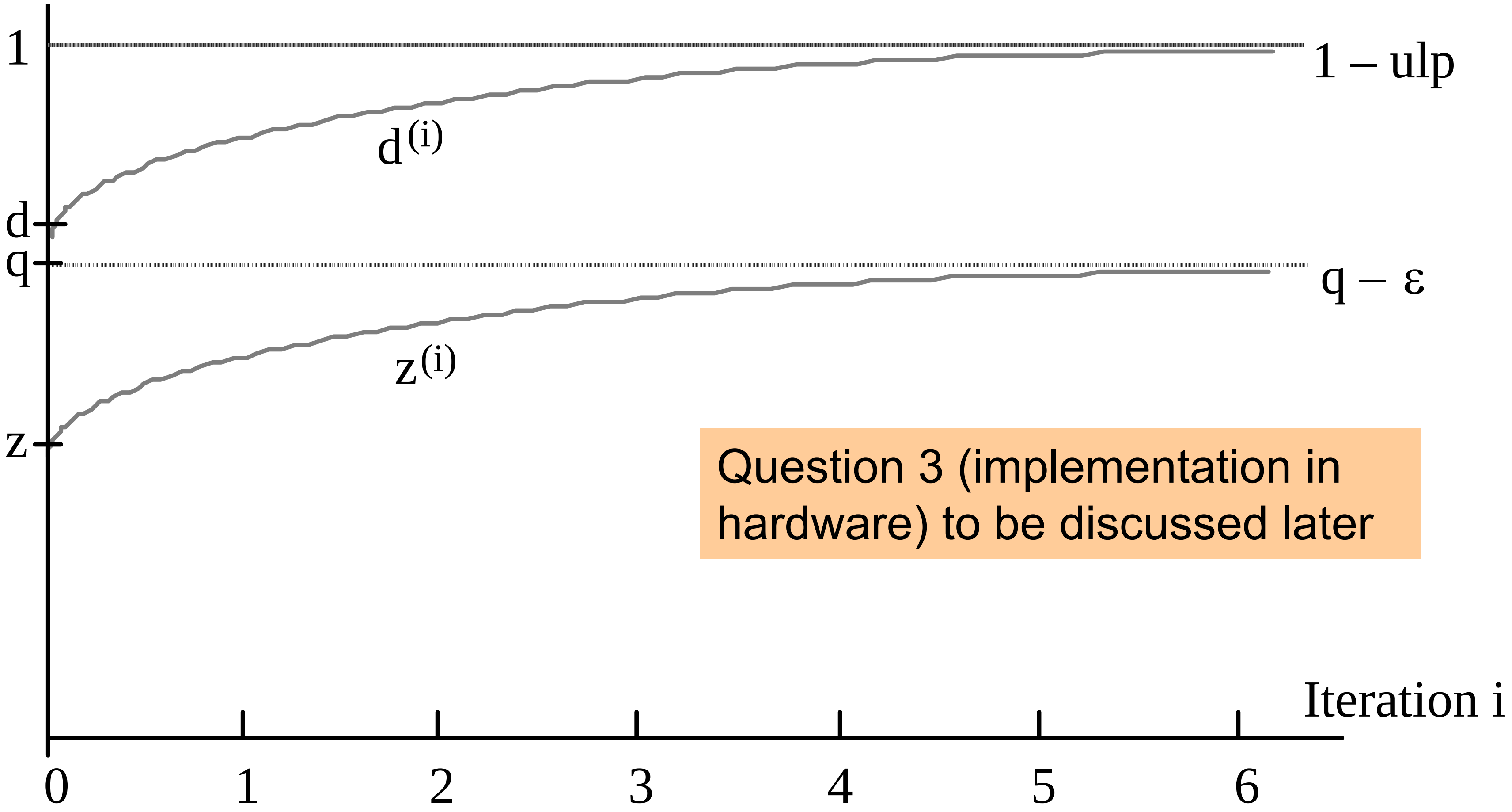
Table 16.1 Quadratic convergence in computing z/d by repeated multiplications, where $1/2 \le d = 1 - y < 1$

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	$1 - y^{16} = (.1111\ 1111\ 1111\ 1111)_{\text{two}} = 1 - ulp$	

Each iteration doubles the number of guaranteed leading 1s (convergence to 1 is from below)

Beginning with a single 1 ($d \ge ½$), after $\log_2 k$ iterations we get as close to 1 as is possible in a fractional representation

Graphical Depiction of Convergence to *q*



Question 3 (implementation in hardware) to be discussed later

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

16.3 Division by Reciprocation

The Newton-Raphson method can be used for finding a root of $f(x) = 0$

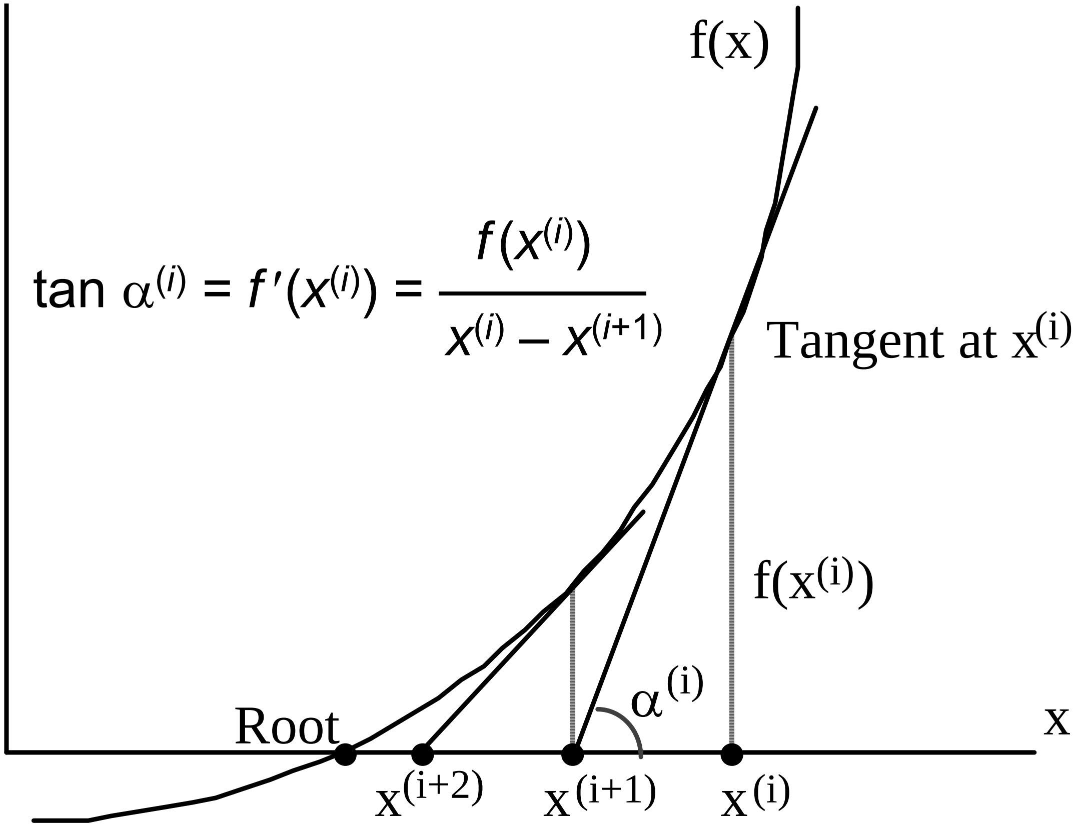
Start with an initial estimate $x^{(0)}$ for the root

Iteratively refine the estimate via the recurrence

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

Justification:

$$\begin{aligned}\tan \alpha^{(i)} &= f'(x^{(i)}) \\ &= f(x^{(i)}) / (x^{(i)} - x^{(i+1)})\end{aligned}$$



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

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

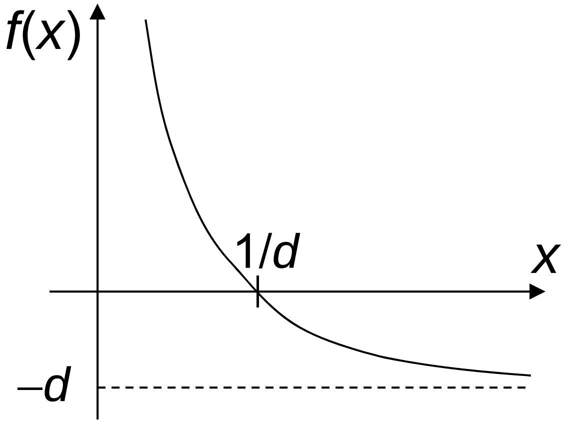
Computing $1/d$ by Convergence

$1/d$ is the root of $f(x) = 1/x - d$

$f'(x) = -1/x^2$

Substitute in the Newton-Raphson recurrence $x^{(i+1)} = x^{(i)} - f(x^{(i)}) / f'(x^{(i)})$ to get:

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



One iteration = Two multiplications + One 2's complementation

Error analysis: Let $\delta^{(i)} = 1/d - x(i)$ be the error at the *i*th iteration

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

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

Choosing the Initial Approximation to 1/*d*

With $x^{(0)}$ in the range $0 < x^{(0)} < 2/d$, convergence is guaranteed

Justification: $|\delta^{(0)}| = |x^{(0)} - 1/d| < 1/d$

$$\delta^{(1)} = |x^{(1)} - 1/d| = d(\delta^{(0)})^2 = (d\delta^{(0)})\delta^{(0)} < \delta^{(0)}$$

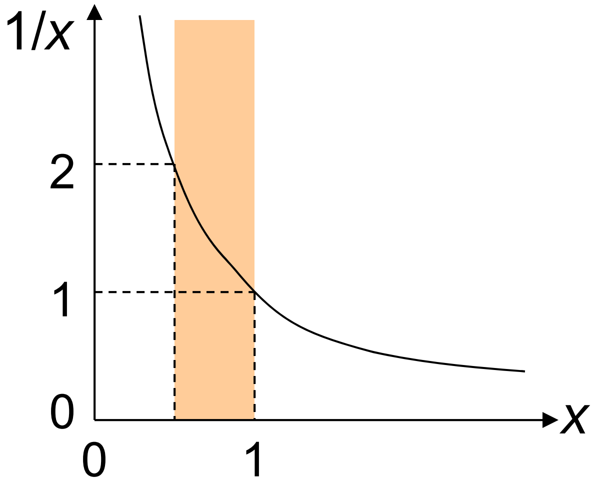
For *d* in [1/2, 1):

Simple choice $x^{(0)} = 1.5$

Max error = 0.5 < 1/*d*

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

Max error ≅ 0.1



16.4 Speedup of Convergence Division

$$q = \frac{z}{d} = \frac{zx^{(0)}x^{(1)}\ldots x^{(m-1)}}{dx^{(0)}x^{(1)}\ldots x^{(m-1)}}$$

Compute $y = 1/d$
Do the multiplication yz

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

This is not yet very impressive

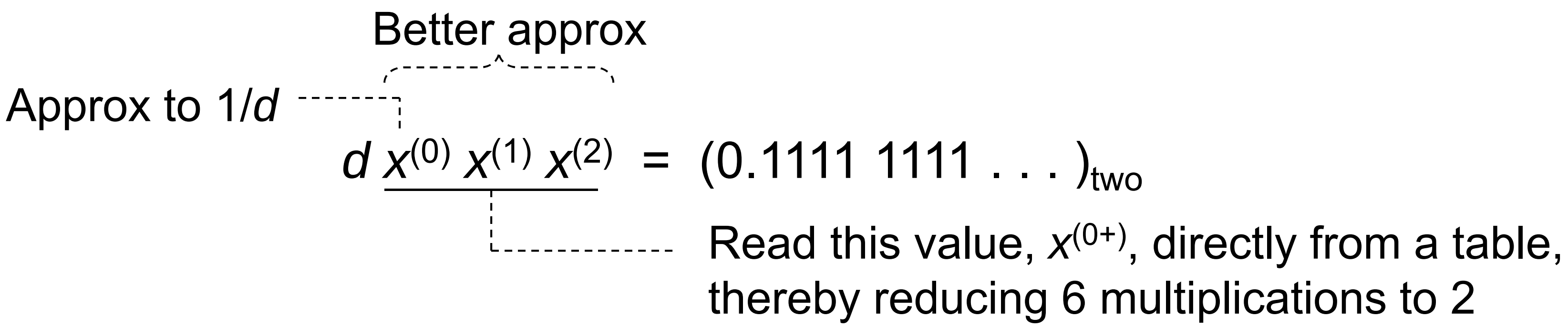
64-bit numbers, 3-ns multiplier $\Rightarrow$ 33-ns division

Three types of speedup are possible:

Fewer multiplications (reduce m)
Narrower multiplications (reduce the width of some $x^{(i)}$s)
Faster multiplications

Initial Approximation via Table Lookup

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



Better approx

Approx to 1/*d*

$$d\,\overbrace{\underline{x^{(0)}\,x^{(1)}\,x^{(2)}}}\;=\;(0.1111\ 1111\ .\ .\ .\ )_{\text{two}}$$

Read this value, $x^{(0+)}$, directly from a table, thereby reducing 6 multiplications to 2

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

Example with 4-bit lookup: $d = 0.1011\ \text{xxxx}\ .\ .\ .$ $(11/16 \le d < 12/16)$

Inverses of the two extremes are $16/11 \cong 1.0111$ and $16/12 \cong 1.0101$

So, 1.0110 is a good estimate for 1/*d*

$1.0110 \times 0.1011 = (11/8) \times (11/16) = 121/128 = 0.1111001$

$1.0110 \times 0.1100 = (11/8) \times (3/4) = 33/32 = 1.000010$

Visualizing the Convergence with Table Lookup

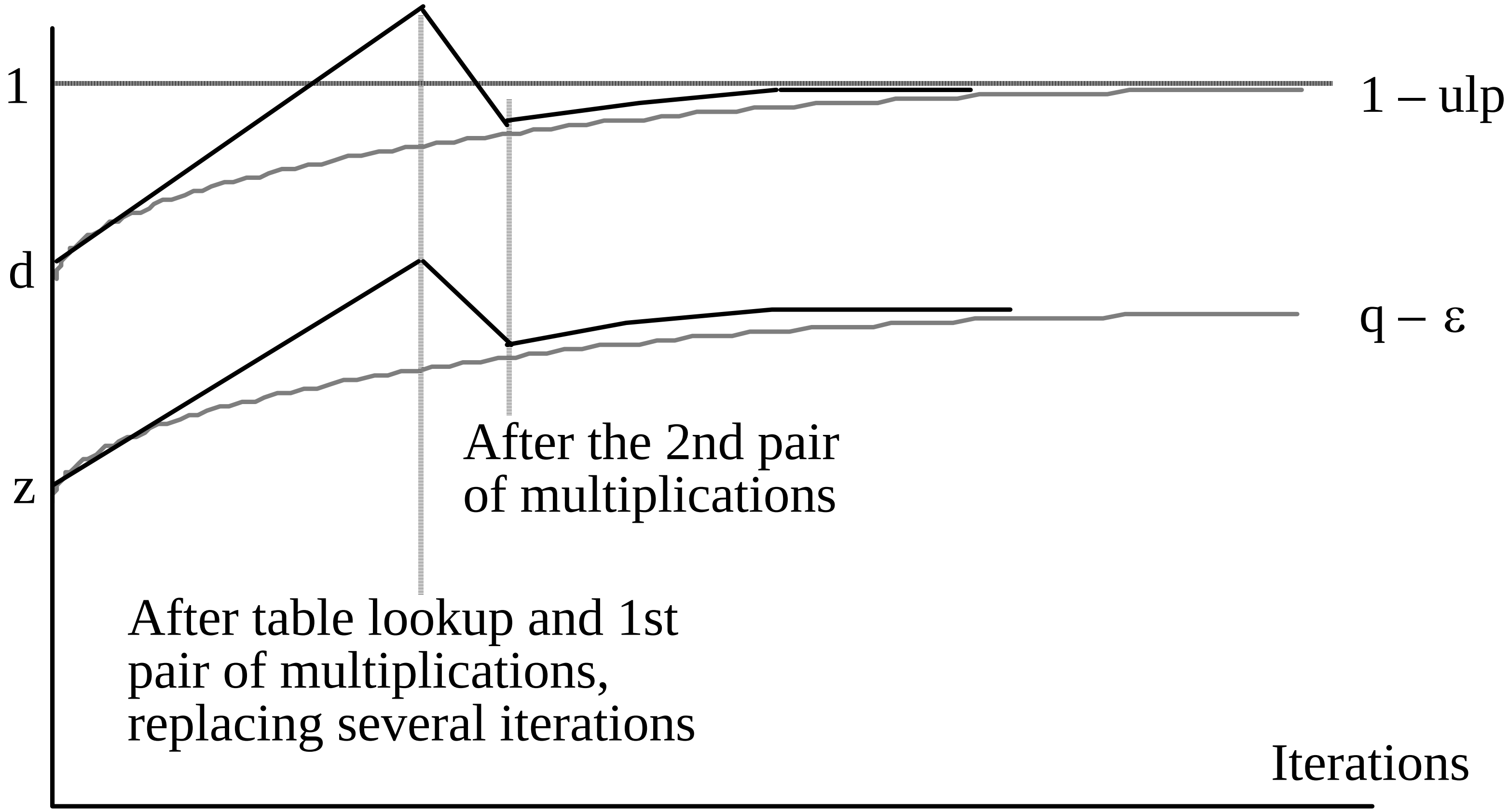


Fig. 16.3 Convergence in division by repeated multiplications with initial table lookup.

Convergence Does Not Have to Be from Below

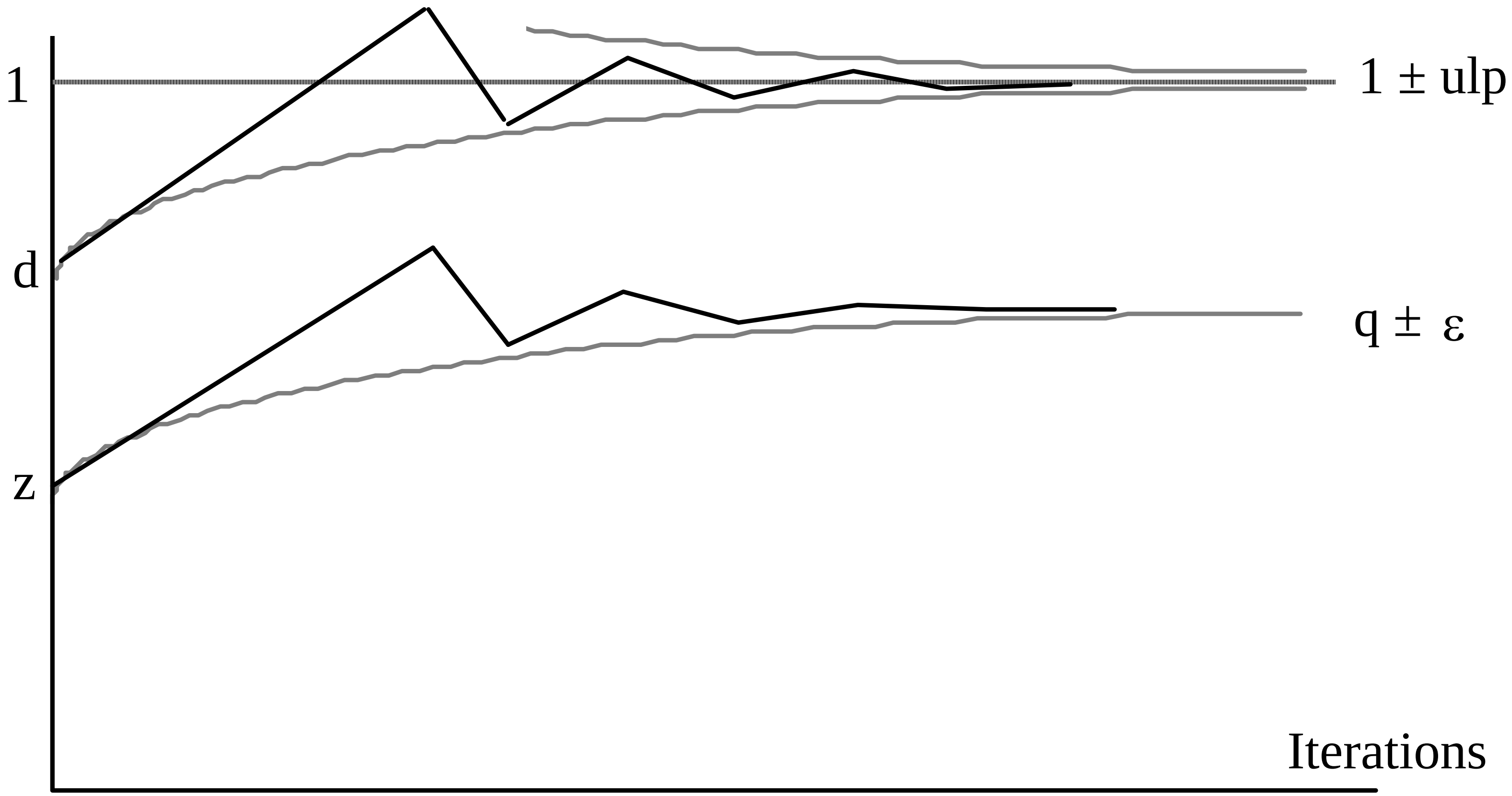
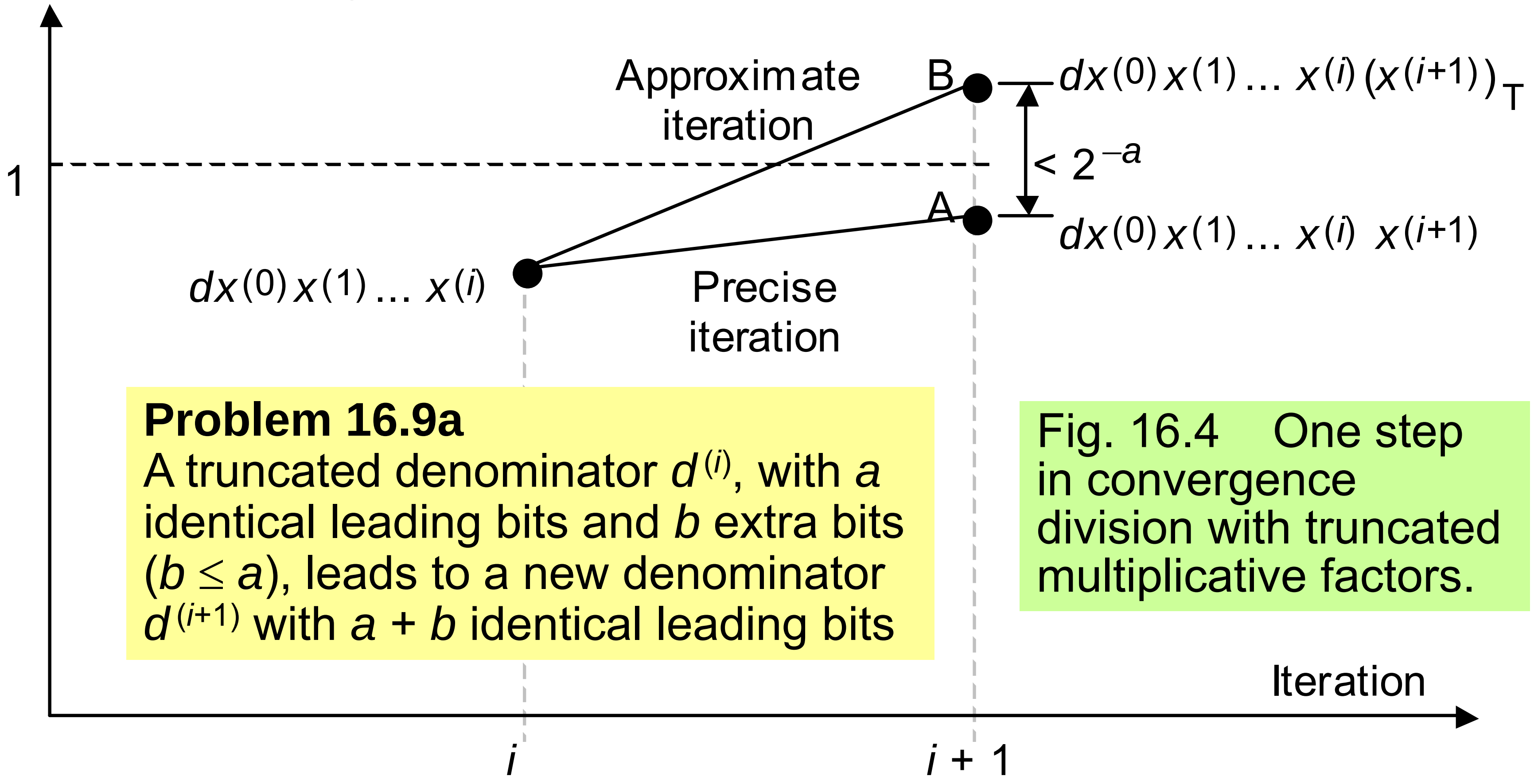


Fig. 16.4 Convergence in division by repeated multiplications with initial table lookup and the use of truncated multiplicative factors.

Using Truncated Multiplicative Factors


Approximate iteration
B
$dx^{(0)}x^{(1)} \ldots x^{(i)}(x^{(i+1)})_T$
1
$< 2^{-a}$
A
$dx^{(0)}x^{(1)} \ldots x^{(i)}\ x^{(i+1)}$
$dx^{(0)}x^{(1)} \ldots x^{(i)}$
Precise iteration
Iteration
i
$i + 1$

Problem 16.9a
A truncated denominator $d^{(i)}$, with a identical leading bits and b extra bits ($b \leq a$), leads to a new denominator $d^{(i+1)}$ with $a + b$ identical leading bits

Fig. 16.4 One step in convergence division with truncated multiplicative factors.

Example (64-bit multiplication)
Initial step: Table of size 256 × 8 = 2K bits
Middle steps: Multiplication pairs, with 9-, 17-, and 33-bit multipliers
Final step: Full 64 × 64 multiplication

16.5 Hardware Implementation

Repeated multiplications: Each pair of ops involves the same multiplier

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

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

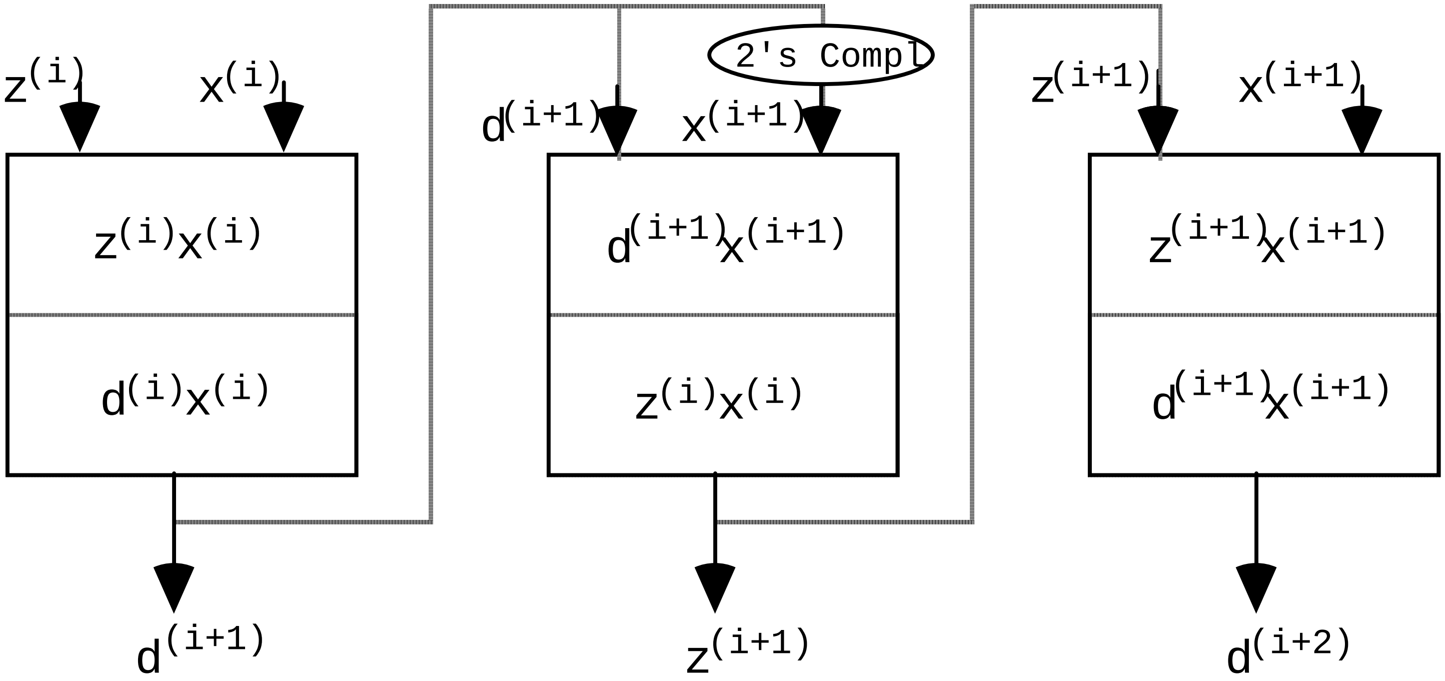


Fig. 16.6 Two multiplications fully overlapped in a 2-stage pipelined multiplier.

Implementing Division with Reciprocation

Reciprocation: Multiplication pairs are data-dependent, so they cannot be pipelined or performed in parallel

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

Options for speedup via a better initial approximation

Consult a larger table
Resort to a bipartite or multipartite table (see Chapter 24)
Use table lookup, followed with interpolation
Compute the approximation via multioperand addition

Unless several multiplications by the same multiplier are needed, division by repeated multiplications is more efficient

However, given a fast method for reciprocation (see Section 24.6), using a reciprocation unit with a standard multiplier is often preferred

16.6 Analysis of Lookup Table Size

Table 16.2 Sample entries in the lookup table replacing the first four 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: Table entry at address 55 $(311/512 \le d < 312/512)$

For 8 bits of convergence, the table entry *f* must satisfy

$$(311/512)(1 + .f) \ge 1 - 2^{-8} \qquad (312/512)(1 + .f) \le 1 + 2^{-8}$$

$$199/311 \le .f \le 101/156$$

$$163.81 \le f = 256 \times .f \le 165.74$$

Two choices: $164 = (1010\ 0100)_{\text{two}}$ or $165 = (1010\ 0101)_{\text{two}}$

A General Result for Table Size

Theorem 16.1: To get $w \geq 5$ bits of convergence after the first iteration of division by repeated multiplications, w bits of d (beyond the mandatory 1) must be inspected. The factor $x^{(0+)}$ read out from table is of the form $(1.\text{xxx} \ldots \text{xxx})_{\text{two}}$, with w bits after the radix point

Proof strategy for sufficiency: Represent the table entry 1.*f* as the integer $v = 2^w \times .f$ and derive upper / lower bound expressions for it. Then, show that at least one integer exists between v_{lb} and v_{ub}

Proof strategy for necessity: Show that derived conditions cannot be met if the table is of size 2^{k-1} (no matter how wide) or if it is of width $k - 1$ (no matter how large)

Excluded cases, *w* < 5: Practically uninteresting (allow smaller table)

General radix *r* : Same analysis method, and results, apply