Part IV
Division

<table>
<thead>
<tr>
<th>Parts</th>
<th>Chapters</th>
</tr>
</thead>
</table>
| I. Number Representation | 1. Numbers and Arithmetic  
2. Representing Signed Numbers  
3. Redundant Number Systems  
4. Residue Number Systems |
| 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 Representations  
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

<table>
<thead>
<tr>
<th>Edition</th>
<th>Released</th>
<th>Revised</th>
<th>Revised</th>
<th>Revised</th>
<th>Revised</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>May 2008</td>
<td>May 2009</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Second</td>
<td>May 2010</td>
<td>Apr. 2011</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Chapter 13</th>
<th>Basic Division Schemes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 14</td>
<td>High-Radix Dividers</td>
</tr>
<tr>
<td>Chapter 15</td>
<td>Variations in Dividers</td>
</tr>
<tr>
<td>Chapter 16</td>
<td>Division by Convergence</td>
</tr>
</tbody>
</table>
I'm appalled at what they'll put on television nowadays ... It's nothing but senseless viruses and gratuitous dividing!

Be fruitful and multiply . . .

Now, divide.
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

<table>
<thead>
<tr>
<th>Topics in This Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.1 Shift/Subtract Division Algorithms</td>
</tr>
<tr>
<td>13.2 Programmed Division</td>
</tr>
<tr>
<td>13.3 Restoring Hardware Dividers</td>
</tr>
<tr>
<td>13.4 Nonrestoring and Signed Division</td>
</tr>
<tr>
<td>13.5 Division by Constants</td>
</tr>
<tr>
<td>13.6 Radix-2 SRT Division</td>
</tr>
</tbody>
</table>
13.1 Shift/Subtract Division Algorithms

Notation for our discussion of division algorithms:

- **z** Dividend \( z_{2k-1}z_{2k-2} \ldots z_3z_2z_1z_0 \)
- **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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
<th>Cycles/Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load / Store</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Integer Multiply</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Integer Divide</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>Double/Single FP Multiply</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Double/Single FP Add</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Double/Single FP Divide</td>
<td>38</td>
<td>38</td>
</tr>
</tbody>
</table>
Division Recurrence

Division with left shifts

(There is no corresponding right-shift algorithm)

\[ s^{(j)} = 2s^{(j-1)} - q_{k-j} (2^k d) \]

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

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

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

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

Divide fractions like integers; adjust the remainder

No-overflow condition for fractions is:

\[ z_{\text{frac}} < d_{\text{frac}} \]
Examples of Basic Division

<table>
<thead>
<tr>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer division</td>
</tr>
<tr>
<td>===========</td>
</tr>
<tr>
<td>$z$</td>
</tr>
<tr>
<td>$2^4 d$</td>
</tr>
<tr>
<td>$s^{(0)}$</td>
</tr>
<tr>
<td>$2 s^{(0)}$</td>
</tr>
<tr>
<td>$-q_3 2^4 d$</td>
</tr>
<tr>
<td>$s^{(1)}$</td>
</tr>
<tr>
<td>$2 s^{(1)}$</td>
</tr>
<tr>
<td>$-q_2 2^4 d$</td>
</tr>
<tr>
<td>$s^{(2)}$</td>
</tr>
<tr>
<td>$2 s^{(2)}$</td>
</tr>
<tr>
<td>$-q_1 2^4 d$</td>
</tr>
<tr>
<td>$s^{(3)}$</td>
</tr>
<tr>
<td>$2 s^{(3)}$</td>
</tr>
<tr>
<td>$-q_0 2^4 d$</td>
</tr>
<tr>
<td>$s^{(4)}$</td>
</tr>
<tr>
<td>$s$</td>
</tr>
<tr>
<td>$q$</td>
</tr>
</tbody>
</table>

| Fractional division |
| =========== |
| $z_{\text{frac}}$ | .0 1 1 \ 1 0 1 0 1 |
| $d_{\text{frac}}$ | .1 0 1 0 |
| $s^{(0)}$ | .0 1 1 \ 1 0 1 0 1 |
| $2 s^{(0)}$ | .0 1 1 \ 1 0 1 0 1 |
| $-q_{-1} d$ | .1 0 1 0 \ {q_{-1}=1} |
| $s^{(1)}$ | .0 1 0 \ 0 1 0 1 |
| $2 s^{(1)}$ | .0 1 0 \ 0 1 0 1 |
| $-q_{-2} d$ | .0 0 0 0 \ {q_{-2}=0} |
| $s^{(2)}$ | .1 0 0 \ 1 0 1 |
| $2 s^{(2)}$ | .1 0 0 \ 1 0 1 |
| $-q_{-3} d$ | .1 0 1 0 \ {q_{-3}=1} |
| $s^{(3)}$ | .1 0 0 \ 0 1 |
| $2 s^{(3)}$ | .1 0 0 \ 0 1 |
| $-q_{-4} d$ | .1 0 1 0 \ {q_{-4}=1} |
| $s^{(4)}$ | .0 1 1 1 |
| $s_{\text{frac}}$ | 0 \ 0 0 0 0 \ 0 1 1 1 |
| $q_{\text{frac}}$ | .1 0 1 1 |
13.2 Programmed Division

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.

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, $6k + 3$ to $8k + 3$ 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

Fig. 13.5 Shift/subtract sequential restoring divider.
### Example of Restoring Unsigned Division

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

Positive, so set $q_3 = 1$

Negative, so set $q_2 = 0$

and restore

Positive, so set $q_1 = 1$

Positive, so set $q_0 = 1$

Fig. 13.6 Example of restoring unsigned division.

<table>
<thead>
<tr>
<th>$z$</th>
<th>0111</th>
<th>0101</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^4 d$</td>
<td>01010</td>
<td></td>
</tr>
<tr>
<td>$-2^4 d$</td>
<td>10110</td>
<td></td>
</tr>
</tbody>
</table>

| $s^{(0)}$ | 001110101 |
| $2s^{(0)}$ | 01110101 |
| $+(−2^4 d)$ | 10110 |

| $s^{(1)}$ | 00100101 |
| $2s^{(1)}$ | 0100101 |
| $+(−2^4 d)$ | 10110 |

| $s^{(2)}$ | 1111101 |
| $s^{(2)}=2s^{(1)}$ | 0100101 |
| $2s^{(2)}$ | 100101 |
| $+(−2^4 d)$ | 10110 |

| $s^{(3)}$ | 010001 |
| $2s^{(3)}$ | 10001 |
| $+(−2^4 d)$ | 10110 |

| $s^{(4)}$ | 00111 |
| $s$ | 0111 |
| $q$ | 1011 |
Indirect Signed Division

In division with signed operands, $q$ and $s$ are defined by

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

Examples of division with signed operands

$$z = 5 \quad d = 3 \quad \Rightarrow \quad q = 1 \quad s = 2$$
$$z = 5 \quad d = -3 \quad \Rightarrow \quad q = -1 \quad s = 2 \quad \text{(not } q = -2, s = -1)$$
$$z = -5 \quad d = 3 \quad \Rightarrow \quad q = -1 \quad s = -2$$
$$z = -5 \quad d = -3 \quad \Rightarrow \quad q = 1 \quad s = -2$$

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”)

Later events depend on earlier ones in the same cycle, causing a lengthening of the clock cycle.
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 $u - 2^kd$

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

Option 2: Keep the incorrect partial remainder $u - 2^kd$, shift left, and add to get $2(u - 2^kd) + 2^kd = 2u - 2^kd$
Example of Nonrestoring Unsigned Division

Fig. 13.7 Example of nonrestoring unsigned division.
Graphical Depiction of Nonrestoring Division

Example

\((01110101)_\text{two} / (1010)_\text{two}\)

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

Fig. 13.8 Partial remainder variations for restoring and nonrestoring division.
Convergence of the Partial Quotient to $q$

Example

$$(01110101)_2 \div (1010)_2 = (111)_2 = (1011)_2$$

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 \( d \)

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

\[
\text{if } \text{sign}(s) = \text{sign}(d) \text{ then } q_{k-j} = 1 \text{ else } q_{k-j} = -1
\]

Example: \( q = \ldots 0 0 0 1 \ldots \rightarrow \ldots 1 -1 -1 -1 \ldots \)
Quotient Conversion and Final Correction

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

Quotient with digits $-1$ and $1$

Replace $-1$s with $0$s

Shift left, complement MSB, and set LSB to 1 to get the $2$’s-complement quotient

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

Fig. 13.9

Example of nonrestoring signed division.

<table>
<thead>
<tr>
<th></th>
<th>0 0 1 0 0 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d$</td>
<td>1 1 0 0 1</td>
</tr>
<tr>
<td>$-d$</td>
<td>0 0 1 1 1</td>
</tr>
</tbody>
</table>

$s^{(0)} = 0 0 0 1 0 0 0 1$

$2s^{(0)} = 0 0 1 0 0 0 1$

$+2^4 d = 1 1 0 0 1$

$s^{(1)} = 1 1 1 0 1 0 0 1$

$2s^{(1)} = 1 1 0 1 0 0 1$

$+(-2^4 d) = 0 0 1 1 1$

$s^{(2)} = 0 0 0 1 0 1$

$2s^{(2)} = 0 0 1 0 0 1$

$+2^4 d = 1 1 0 0 1$

$s^{(3)} = 1 1 0 1 1 0 1$

$2s^{(3)} = 1 0 1 1 1$

$+(-2^4 d) = 0 0 1 1 1$

$s^{(4)} = 1 1 1 1 0$

$+(-2^4 d) = 0 0 1 1 1$

$s^{(4)} = 0 0 1 0 1$

$p = 0 1 0 1$ Shift, compl MSB

Add 1 to correct

$q = -1 1 1 1$

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

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

Number of shift-adds required is proportional to $\log k$
Example Division by a Constant

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

\[
\frac{z}{d} = \frac{zm}{2^n - 1} = \frac{zm}{2^n (1 - 2^{-n})} = \frac{zm}{2^n} (1 + 2^{-n})(1 + 2^{-2n})(1 + 2^{-4n}) \ldots
\]

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

**Instruction sequence for division by 5**

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

\[
\begin{align*}
q & \leftarrow z + z \text{ shift-left } 1 \quad \{3z \text{ computed}\} \\
q & \leftarrow q + q \text{ shift-right } 4 \quad \{3z(1 + 2^{-4}) \text{ computed}\} \\
q & \leftarrow q + q \text{ shift-right } 8 \quad \{3z(1 + 2^{-4})(1 + 2^{-8}) \text{ computed}\} \\
q & \leftarrow q + q \text{ shift-right } 16 \quad \{3z(1 + 2^{-4})(1 + 2^{-8})(1 + 2^{-16}) \text{ computed}\} \\
q & \leftarrow q \text{ shift-right } 4 \quad \{3z(1 + 2^{-4})(1 + 2^{-8})(1 + 2^{-16})/16 \text{ computed}\}
\end{align*}
\]

Computing \(29 \div 5\) \((z = 29, \ d = 5)\)

\[
\begin{align*}
87 & \leftarrow 29 + 29 \text{ shift-left } 1 \quad \{3z \text{ computed}\} \\
92 & \leftarrow 87 + 87 \text{ shift-right } 4 \quad \{3z(1 + 2^{-4}) \text{ computed}\} \\
92 & \leftarrow 92 + 92 \text{ shift-right } 8 \quad \{3z(1 + 2^{-4})(1 + 2^{-8}) \text{ computed}\} \\
92 & \leftarrow 92 + 92 \text{ shift-right } 16 \quad \{3z(1 + 2^{-4})(1 + 2^{-8})(1 + 2^{-16}) \text{ computed}\} \\
5 & \leftarrow 92 \text{ shift-right } 4 \quad \{3z(1 + 2^{-4})(1 + 2^{-8})(1 + 2^{-16})/16 \text{ computed}\}
\end{align*}
\]

Repeat the process for computing \(30 \div 5\) 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 $-\frac{1}{2}$ and $\frac{1}{2}$ for quotient digit selection:

- $2s \geq +\frac{1}{2}$ means $2s = (0.1xxxxxxxx)_{2's-compl}$
- $2s < -\frac{1}{2}$ means $2s = (1.0xxxxxxxx)_{2's-compl}$

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

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 $-\frac{1}{2}$ and $\frac{1}{2}$ for quotient digit selection.

For $2s \geq +\frac{1}{2}$ or $2s = (0.1xxxxxxx)_{2\text{’s-compl}}$ choose $q_j = 1$

For $2s < -\frac{1}{2}$ or $2s = (1.0xxxxxxx)_{2\text{’s-compl}}$ choose $q_j = -1$

Choose $q_j = 0$ in other cases, that is, for:

- $0 \leq 2s < +\frac{1}{2}$ or $2s = (0.0xxxxxxx)_{2\text{’s-compl}}$
- $-\frac{1}{2} \leq 2s < 0$ or $2s = (1.1xxxxxxx)_{2\text{’s-compl}}$

Observation: What happens when the magnitude of $2s$ 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

\[ z = 0.1000 \ 0101 \]
\[ d = 0.1010 \]
\[ -d = 1.0110 \]

\[ s^{(0)} = 0.0100 \ 0101 \]
\[ 2s^{(0)} = 0.1000 \ 101 \]
\[ +(−d) = 1.0110 \]
\[ s^{(1)} = 1.1110 \ 101 \]
\[ 2s^{(1)} = 1.1101 \ 01 \]
\[ s^{(2)} = 2s^{(1)} = 1.1101 \ 01 \]
\[ 2s^{(2)} = 1.1010 \ 1 \]
\[ s^{(3)} = 2s^{(2)} = 0.1010 \ 1 \]
\[ 2s^{(3)} = 1.0101 \]
\[ +d = 0.1010 \]
\[ s^{(4)} = 1.1111 \]
\[ +d = 0.1010 \]
\[ s^{(4)} = 0.1001 \]
\[ s = 0.0000 \ 0101 \]
\[ q = 0.1001 \]
\[ q = 0.0110 \]

In \([-\frac{1}{2}, \frac{1}{2})\), so okay

**Uncorrected BSD quotient**

Convert and subtract \textit{ulp}
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.
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

<table>
<thead>
<tr>
<th>Section</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>14.1</td>
<td>Basics of High-Radix Division</td>
</tr>
<tr>
<td>14.2</td>
<td>Using Carry-Save Adders</td>
</tr>
<tr>
<td>14.3</td>
<td>Radix-4 SRT Division</td>
</tr>
<tr>
<td>14.4</td>
<td>General High-Radix Dividers</td>
</tr>
<tr>
<td>14.5</td>
<td>Quotient Digit Selection</td>
</tr>
<tr>
<td>14.6</td>
<td>Using $p-d$ Plots in Practice</td>
</tr>
</tbody>
</table>
14.1 Basics of High-Radix Division

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

Division with left shifts

\[ s^{(j)} = r s^{(j-1)} - q_{k-j} (r^k d) \]

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

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?

```
  2 0 4 3  |  1 2 2 5 7 9 6 8
```

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: $5 \cdot 9 = 6 -1$
Examples of High-Radix Division

Radix-4 integer division

<table>
<thead>
<tr>
<th>z</th>
<th>0 1 2 3 1 1 2 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4^4 d</td>
<td>1 2 0 3</td>
</tr>
</tbody>
</table>

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)} | 0 3 0 1 2 |
| 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 $\{-1, 0, 1\}$. 

Choose $-1$  Choose 0  Choose 1

$-1/0$  $s(j)$  $0/+1$  Overlap

$q_{-j} = -1$  $q_{-j} = 0$  $q_{-j} = 1$

$-d$  $d$

$-2d$  $2d$  $2s^{(j-1)}$

$-1/2$  0

$-1/0$ Overlap  $0/+1$ Overlap
Quotient Digit Selection Based on Truncated PR

Sum part of $2s^{(j-1)}$: $u = (u_1u_0 \cdot u_{-1}u_{-2} \ldots)_{2's-compl}$

Carry part of $2s^{(j-1)}$: $v = (v_1v_0 \cdot v_{-1}v_{-2} \ldots)_{2's-compl}$

Approximation to the partial remainder:

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

Max error in approximation

$< \frac{1}{4} + \frac{1}{4} = \frac{1}{2}$

Error in $[0, \frac{1}{2})$

Fig. 14.3

Choose -1

Choose 0

Choose 1

$-2d \quad -d \quad 0 \quad d \quad 2d$

$q_j = -1$

$q_j = 0$

$q_j = 1$

Overlap

Overlap

$-1/0$

$0/+1$

$-1/2$

$2s^{(j-1)}$
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

Infeasible region 

\( p \) cannot be \( \geq 2d \)

Worst-case error margin in comparison

Choose 1

Choose 0

Choose −1

Infeasible region

\( p \) cannot be < −2d

\[ \begin{align*}
0.100 & \quad 0.101 & \quad 0.110 & \quad 0.111 & \quad 1.0 \\
0.11 & \quad 0.01 & \quad 1.0 & \quad 1.0 & \quad 1.0
\end{align*} \]

\[ \begin{align*}
-0.01 & \quad 0.1 & \quad -0.1 & \quad -0.1 & \quad -0.1
\end{align*} \]

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

\[
\text{Shifted sum} = (u_1u_0 \cdot u_{-1}u_{-2} \cdots)^{2\text{'s-compl}} \\
\text{Shifted carry} = (v_1v_0 \cdot v_{-1}v_{-2} \cdots)^{2\text{'s-compl}} \\
\text{Approx shifted PR} = (t_1t_0 \cdot t_{-1}t_{-2})^{2\text{'s-compl}}
\]

Non0 = \( t'_1 \lor t'_0 \lor t'_{-1} = (t_1 t_0 t_{-1})' \)
Sign = \( t_1 (t'_0 \lor t'_{-1}) \)
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—|

\[ s^{(j)} \]

\[ d \]

\[ 4s^{(j-1)} \]

\[ -4d \]

\[ -3 \]

\[ -2 \]

\[ -1 \]

\[ 0 \]

\[ +1 \]

\[ +2 \]

\[ +3 \]

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 $[-3, 3]$. 

Infeasible region ($p$ cannot be $\geq 4d$)
Restricting the Quotient Digit Set in Radix 4

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

\[
s^{(j)} = 4s^{(j-1)} - q_{-j}d \quad \text{with} \quad s^{(0)} = z \quad \text{and} \quad s^{(k)} = 4^k s
\]

\[\text{shift} \quad \text{subtract}\]

For this restriction to be feasible, we must have:

\[s \in [-hd, hd) \text{ for some } h < 1, \text{ and } 4hd - 2d \leq hd\]

This yields \( h \leq 2/3 \) (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 $[-2, 2]$. 

Infeasible region
($p$ cannot be $\geq 8d/3$)

Building the $p$-$d$ Plot with Restricted Radix-4 Digit Set
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.
14.5 Quotient Digit Selection

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

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

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

Fig. 14.13 A part of $p$-$d$ plot showing the overlap region for choosing the quotient digit value $\beta$ or $\beta+1$ in radix-$r$ division with quotient digit set $[-\alpha, \alpha]$.

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

Note: $h = \alpha/(r-1)$
14.6 Using $p$-$d$ Plots in Practice

Smallest $\Delta d$ occurs for the overlap region of $\alpha$ and $\alpha - 1$

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

$$\Delta p = d_{\text{min}} (2h - 1)$$

Fig. 14.15 Establishing upper bounds on the dimensions of uncertainty rectangles.
Example: Lower Bounds on Precision

\[ \Delta d = d_{\text{min}} \frac{2h - 1}{-h + \alpha} \]

\[ \Delta p = d_{\text{min}} (2h - 1) \]

For \( r = 4 \), divisor range \([0.5, 1)\), digit set \([-2, 2]\), we have \( \alpha = 2 \), \( d_{\text{min}} = 1/2 \), \( h = \alpha/(r - 1) = 2/3 \)

\[ \Delta d = \left(\frac{1}{2}\right) \frac{4/3 - 1}{-2/3 + 2} = 1/8 \]

\[ \Delta p = \left(\frac{1}{2}\right) \left(4/3 - 1\right) = 1/6 \]

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 $\Delta d$ and $\Delta p$, one more bit of precision in each direction is always adequate.

Proof: Let $w$ be the spacing of vertical grid lines.

$$w \leq \Delta d/2 \quad \Rightarrow \quad v \leq \Delta p/2 \quad \Rightarrow \quad u \geq \Delta p/2$$
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 \( r = 4 \)
- \( q \) 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

<table>
<thead>
<tr>
<th>Topics in This Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>15.1 Division with Prescaling</td>
</tr>
<tr>
<td>15.2 Overlapped Quotient Digit Selection</td>
</tr>
<tr>
<td>15.3 Combinational and Array Dividers</td>
</tr>
<tr>
<td>15.4 Modular Dividers and Reducers</td>
</tr>
<tr>
<td>15.5 The Special Case of Reciprocation</td>
</tr>
<tr>
<td>15.6 Combined Multiply/Divide Units</td>
</tr>
</tbody>
</table>
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 $d^{\text{max}}$ (say $1 - \varepsilon < d < 1 + \delta$, when $d^{\text{max}} = 1$), 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 $d \in [1/2, 3/4)$, multiply by $1\frac{1}{2}$ [$d$ 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 $d \in$

- $[1/2, 9/16)$, 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 $1 + 1/8$

$[1/2, 9/16) \times 2 = [1, 1 + 1/8)$

$[9/16, 5/8) \times (1 + 1/2) = [1 – 5/32, 1 – 1/16)$

$[5/8, 3/4) \times (1 + 1/2) = [1 – 1/16, 1 + 1/8)$

$[3/4, 1) \times (1 + 1/8) = [1 – 5/32, 1 + 1/8)$

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 $\rightarrow$ 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

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

Similarity to array multiplier is deceiving

Critical path

Cell

FA

XOR

Dividend \( z = z_0 z_1 z_2 z_3 z_4 z_5 z_6 \)

Divisor \( d = d_0 d_1 d_2 d_3 d_4 d_5 d_6 \)

Quotient \( q = q_0 q_1 q_2 q_3 q_4 q_5 q_6 \)

Remainder \( s = s_0 s_1 s_2 s_3 s_4 s_5 s_6 \)
Speedup Methods for Array Dividers

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

Fig. 15.8
15.4 Modular Dividers and Reducers

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

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

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:

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

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

Multiplication: $t = (aR)(xR)R^{-1} \mod m = (ax)R \mod 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

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p^{(0)}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$+x_{0}a$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| $2p^{(1)}$| 0 | 1 | 0 | 1 | 0 |
| $p^{(1)}$| 0 | 1 | 0 | 1 | 0 |
| $+x_{1}a$|   |   |   |   |

| $2p^{(2)}$| 0 | 1 | 1 | 1 | 1 | 0 |
| $p^{(2)}$| 0 | 1 | 1 | 1 | 1 | 0 |
| $+x_{2}a$|   |   |   |   |   |   |

| $2p^{(3)}$| 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| $p^{(3)}$| 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| $+x_{3}a$|   |   |   |   |   |   |   |

| $2p^{(4)}$| 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| $p^{(4)}$| 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |

---

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

### Fig. 15.4

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p^{(0)}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$+x_{0}a$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| $2p^{(1)}$| 0 | 1 | 0 | 1 | 0 |
| $p^{(1)}$| 0 | 1 | 0 | 1 | 0 |
| $+x_{1}a$|   |   |   |   |

| $2p^{(2)}$| 0 | 1 | 1 | 1 | 1 | 0 |
| $p^{(2)}$| 0 | 1 | 1 | 1 | 1 | 0 |
| $+13$|   |   |   |   |   |   |

| $2p^{(3)}$| 0 | 1 | 1 | 1 | 0 |
| $p^{(3)}$| 0 | 1 | 1 | 1 | 0 |
| $+x_{2}a$|   |   |   |   |   |   |

| $2p^{(4)}$| 0 | 1 | 0 | 0 | 0 | 1 |
| $p^{(4)}$| 0 | 1 | 0 | 0 | 0 | 1 |
| $+13$|   |   |   |   |   |   |

---

(a) Ordinary

(b) Mod 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

Multiplier

\[ p = ax \]

\[ y^2 \]

\[(a) \text{ Squaring}\]

(b) Square-rooting?

Divider

\[ q = z / d \]

\[ \sqrt{y} \]

\[(b) \text{ Square-rooting?}\]

(c) Reciprocation

Divider

\[ q = z / d \]

\[ 1 / y \]

\[(c) \text{ Reciprocation}\]

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

\[ Q \approx \frac{1}{d} \text{ with error } \leq 2^{-k/2} \]

\[ t = Q(2 - Qd) \approx \frac{1}{d}; \text{ error } \leq 2^{-k} \]

\[ s^{(j+1)} = 2s^{(j)} - q_{-j}d, \quad \text{with } 2s^{(0)} = 1 \]

\[ t^{(j+1)} = 4t^{(j)} + q_{-j}(4s^{(j)} - q_{-j}d), \quad \text{with } t^{(0)} = 0 \]

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

Dividend: \( z = z_{-6} \cdots z_{-2} \cdots z_{-1} \cdots z_0 \)

Divisor: \( d = d_{-1} \cdots d_{-3} \cdots d_{-5} \cdots d_{-7} \)

Quotient: \( q = q_{-1} \cdots q_{-3} \)

Remainder: \( s = 0.s_{-2} \cdots s_{-3} \cdots s_{-4} \)

Fig. 11.4

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.

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

<table>
<thead>
<tr>
<th>Topics in This Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.1  General Convergence Methods</td>
</tr>
<tr>
<td>16.2  Division by Repeated Multiplications</td>
</tr>
<tr>
<td>16.3  Division by Reciprocation</td>
</tr>
<tr>
<td>16.4  Speedup of Convergence Division</td>
</tr>
<tr>
<td>16.5  Hardware Implementation</td>
</tr>
<tr>
<td>16.6  Analysis of Lookup Table Size</td>
</tr>
</tbody>
</table>
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 $[-d, d)$. 

![Graph showing the convergence of the quotient $q$]
Recurrence Formulas for Convergence Methods

\[ u^{(i+1)} = f(u^{(i)}, v^{(i)}) \quad \text{Constant} \quad u^{(i+1)} = f(u^{(i)}, v^{(i)}, w^{(i)}) \]
\[ v^{(i+1)} = g(u^{(i)}, v^{(i)}) \quad \text{Desired function} \quad v^{(i+1)} = g(u^{(i)}, v^{(i)}, w^{(i)}) \]
\[ w^{(i+1)} = h(u^{(i)}, v^{(i)}, w^{(i)}) \]

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 faster via multiplications faster if 10 or fewer needed  

**Idea:**  
\[ q = \frac{z}{d} = \frac{z x^{(0)} x^{(1)} \ldots x^{(m-1)}}{d x^{(0)} x^{(1)} \ldots x^{(m-1)}} \rightarrow \text{Converges to } q \]  
\[ d x^{(0)} x^{(1)} \ldots x^{(m-1)} \rightarrow \text{Force to 1} \]  

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:

\[ q = \frac{z}{d} = \frac{zx^{(0)}x^{(1)} \ldots x^{(m-1)}}{dx^{(0)}x^{(1)} \ldots x^{(m-1)}} \quad \text{Converges to } q \]

\[ d^{(i+1)} = d^{(i)} x^{(i)} \quad \text{Set } d^{(0)} = d; \text{ make } d^{(m)} \text{ converge to } 1 \]

\[ z^{(i+1)} = z^{(i)} x^{(i)} \quad \text{Set } z^{(0)} = z; \text{ obtain } z/d = q \cong z^{(m)} \]

Question 1: How to select the multipliers \( x^{(i)} \)? \( x^{(i)} = 2 - d^{(i)} \)

This choice transforms the recurrence equations into:

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

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

\[ u^{(i+1)} = f(u^{(i)}, v^{(i)}) \]

\[ v^{(i+1)} = g(u^{(i)}, v^{(i)}) \quad \text{Fits the general form} \]
Determining the Rate of Convergence

\[ d^{(i+1)} = d^{(i)} x^{(i)} \] \hspace{1cm} \text{Set } d^{(0)} = d; \text{ make } d^{(m)} \text{ converge to 1}

\[ z^{(i+1)} = z^{(i)} x^{(i)} \] \hspace{1cm} \text{Set } z^{(0)} = z; \text{ 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 \( 1 - d^{(i)} \leq \varepsilon \), we get \( 1 - d^{(i+1)} \leq \varepsilon^2 \): \textit{Quadratic convergence}

In general, for \( k \)-bit operands, we need

\[ 2m - 1 \text{ multiplications and } m \text{ 2’s complementations} \]

where \( m = \left\lceil \log_2 k \right\rceil \)
Quadratic Convergence

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

<table>
<thead>
<tr>
<th>( i )</th>
<th>( d^{(i)} = d^{(i-1)} x^{(i-1)}, \text{ with } d^{(0)} = d )</th>
<th>( x^{(i)} = 2 - d^{(i)} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( 1 - y = (.1xxx xxxx xxxx xxxx)_{two} \geq 1/2 )</td>
<td>( 1 + y )</td>
</tr>
<tr>
<td>1</td>
<td>( 1 - y^2 = (.11xx xxxx xxxx xxxx)_{two} \geq 3/4 )</td>
<td>( 1 + y^2 )</td>
</tr>
<tr>
<td>2</td>
<td>( 1 - y^4 = (.11111 xxxx xxxx xxxx)_{two} \geq 15/16 )</td>
<td>( 1 + y^4 )</td>
</tr>
<tr>
<td>3</td>
<td>( 1 - y^8 = (.11111111 xxxx xxxx xxxx)_{two} \geq 255/256 )</td>
<td>( 1 + y^8 )</td>
</tr>
<tr>
<td>4</td>
<td>( 1 - y^{16} = (.1111111111111111)_{two} = 1 - ulp )</td>
<td></td>
</tr>
</tbody>
</table>

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

Beginning with a single 1 \((d \geq \frac{1}{2})\), after \( \log_2 k \) iterations we get as close to 1 as is possible in a fractional representation
Graphical Depiction of Convergence to $q$

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

Question 3 (implementation in hardware) to be discussed later
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:

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

Justification:

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

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 $\approx 0.1$
16.4 Speedup of Convergence Division

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

This is not yet very impressive
- 64-bit numbers, 3-ns multiplier ⇒ 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

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

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

$$d \times x^{(0)} x^{(1)} x^{(2)} = (0.1111\,1111\,\ldots)_{\text{two}}$$

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:** $d = 0.1011\,xxxx\ldots$ ($11/16 \leq d < 12/16$)

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

**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.

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

**Fig. 16.4** One step in convergence division with truncated multiplicative factors.
16.5 Hardware Implementation

Repeated multiplications: Each pair of ops involves the same multiplier

\[ d^{(i+1)} = d^{(i)} (2 - d^{(i)}) \]
\[ z^{(i+1)} = z^{(i)} (2 - d^{(i)}) \]

Set \( d^{(0)} = d \); iterate until \( d^{(m)} \approx 1 \)

Set \( z^{(0)} = z \); obtain \( z/d = q \approx z^{(m)} \)

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

<table>
<thead>
<tr>
<th>Address</th>
<th>$d = 0.1 \text{ xxxx } \text{ xxxx}$</th>
<th>$x^{(0+)} = 1. \text{ xxxx } \text{ xxxx}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>55</td>
<td>0011 0111</td>
<td>1010 0101</td>
</tr>
<tr>
<td>64</td>
<td>0100 0000</td>
<td>1001 1001</td>
</tr>
</tbody>
</table>

**Example:** Table entry at address 55 $(311/512 \leq d < 312/512)$

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

\[
(311/512)(1 + .f) \geq 1 - 2^{-8} \quad \text{or} \quad (312/512)(1 + .f) \leq 1 + 2^{-8}
\]

\[
199/311 \leq .f \leq 101/156 \quad \text{or} \quad 163.81 \leq 256 \times .f \leq 165.74
\]

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 \((1.xxx \ldots xxx)_2\), 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.