### 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. 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 University Press, 2000, ISBN 0-19-512583-5). 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>
</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

<table>
<thead>
<tr>
<th>Topics in This Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 13</td>
</tr>
<tr>
<td>Basic Division Schemes</td>
</tr>
<tr>
<td>Chapter 14</td>
</tr>
<tr>
<td>High-Radix Dividers</td>
</tr>
<tr>
<td>Chapter 15</td>
</tr>
<tr>
<td>Variations in Dividers</td>
</tr>
<tr>
<td>Chapter 16</td>
</tr>
<tr>
<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.
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

<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. Preview of Fast Dividers</td>
</tr>
</tbody>
</table>
13.1 Shift/Subtract Division Algorithms

Notation for our discussion of division algorithms:

- \( z \) Dividend
- \( d \) Divisor
- \( q \) Quotient
- \( s \) Remainder, \( z - (d \times q) \)

Initially, we assume unsigned operands

![Division Diagram](image)

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

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

(There is no corresponding right-shift algorithm)

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

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

![Diagram of Register Usage for Programmed Division](image)

- Rs: Register for Shifted Partial Remainder
- Rq: Register for Shifted Partial Quotient
- Carry Flag
- Partial Remainder (2k – j Bits)
- Partial Quotient (j Bits)
- Rd: Register for Divisor d
- Divisor d inserted here
- Next quotient digit inserted here
- \(2^k d\)

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

\[ \text{Fig. 13.3} \]
Register usage for programmed division.

\[ \text{Fig. 13.4} \]
Programmed division using left shifts.
Time Complexity of Programmed Division

Assume $k$-bit words

$k$ iterations of the main loop
6 or 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.
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 (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
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.
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: \((0111)_{\text{two}} < (1010)_{\text{two}}\)

Positive, so subtract

Positive, so set \(q_3 = 1\) and subtract

Negative, so set \(q_2 = 0\) and add

Positive, so set \(q_1 = 1\) and subtract

Positive, so set \(q_0 = 1\)

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

Example

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

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

Fig. 13.8 Partial remainder variations for restoring and nonrestoring division.
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 \( z \), leads to a direct signed division algorithm

\[
\text{if } \text{sign}(s) = \text{sign}(d) \text{ then } q_{k-j} = 1 \text{ else } q_{k-j} = -1
\]
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 0s

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

Final correction step if \( \text{sign}(s) \neq \text{sign}(z) \):
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.

\[
\begin{array}{c|c}
\ hline
z & 00100001 \\
2^4d & 11001 \\
-2^4d & 00111 \\
\ hline
s^{(0)} & 000100001 \\
2s^{(0)} & 00100001 \\
+2^4d & 11001 \\
\ hline
s^{(1)} & 11101001 \\
2s^{(1)} & 1101001 \\
+(-2^4d) & 00111 \\
\ hline
s^{(2)} & 0000101 \\
2s^{(2)} & 000101 \\
+2^4d & 11001 \\
\ hline
s^{(3)} & 1101111 \\
2s^{(3)} & 101111 \\
+(-2^4d) & 00111 \\
\ hline
s^{(4)} & 11110 \\
+(-2^4d) & 00111 \\
\ hline
s^{(4)} & 00101 \\
\ hline
s & 0101 \\
q & 11011 \\
p & 0101 \\
\ hline
\end{array}
\]

\[p = 0101\] Shift, compl MSB

\[s = 0101\]

\[q = 11011\] Add 1 to correct

\[p = 0101\]

\[q = 11011\]

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

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

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

\[
q \leftarrow z + z \text{ shift-left 1 } \quad \{3z \text{ computed}\}
\]
\[
q \leftarrow q + q \text{ shift-right 4 } \quad \{3z(1 + 2^{-4}) \text{ computed}\}
\]
\[
q \leftarrow q + q \text{ shift-right 8 } \quad \{3z(1 + 2^{-4})(1 + 2^{-8}) \text{ computed}\}
\]
\[
q \leftarrow q + q \text{ shift-right 16 } \quad \{3z(1 + 2^{-4})(1 + 2^{-8})(1 + 2^{-16}) \text{ computed}\}
\]
\[
q \leftarrow q \text{ shift-right 4 } \quad \{3z(1 + 2^{-4})(1 + 2^{-8})(1 + 2^{-16})/16 \text{ computed}\}
\]
13.6 Preview of Fast Dividers

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 $> 2 \Rightarrow$ quotient digit selection harder
Remedy: redundant quotient representation
Carry-save addition reduces cycle time
Implementation methods and tradeoffs
14.1 Basics of High-Radix Division

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

Division with left shifts

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

<table>
<thead>
<tr>
<th>shift</th>
</tr>
</thead>
</table>

\[ s^{(k)} = r^ks \]

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

\[ q \quad \text{Quotient} \]
\[ z \quad \text{Dividend} \]
\[ -(q_3q_2)_2d \quad 4^1 \]
\[ -(q_1q_0)_2d \quad 4^0 \]
\[ s \quad \text{Remainder} \]

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

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

---

<table>
<thead>
<tr>
<th>$s^{(0)}$</th>
<th>0 1 2 3 1 1 2 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$4s^{(0)}$</td>
<td>0 1 2 3 1 1 2 3</td>
</tr>
<tr>
<td>$-q_3 4^4d$</td>
<td>0 1 2 0 3 ${q_3 = 1}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$s^{(1)}$</th>
<th>0 0 2 2 1 2 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$4s^{(1)}$</td>
<td>0 0 2 2 1 2 3</td>
</tr>
<tr>
<td>$-q_2 4^4d$</td>
<td>0 0 0 0 0 ${q_2 = 0}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$s^{(2)}$</th>
<th>0 2 2 1 2 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$4s^{(2)}$</td>
<td>0 2 2 1 2 3</td>
</tr>
<tr>
<td>$-q_1 4^4d$</td>
<td>0 1 2 0 3 ${q_1 = 1}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$s^{(3)}$</th>
<th>1 0 0 3 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$4s^{(3)}$</td>
<td>1 0 0 3 3</td>
</tr>
<tr>
<td>$-q_0 4^4d$</td>
<td>0 3 0 1 2 ${q_0 = 2}$</td>
</tr>
</tbody>
</table>

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

---

### Radix-10 fractional division

<table>
<thead>
<tr>
<th>$z_{\text{frac}}$</th>
<th>.7 0 0 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_{\text{frac}}$</td>
<td>.9 9</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>$s^{(0)}$</th>
<th>.7 0 0 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10s^{(0)}$</td>
<td>7.0 0 3</td>
</tr>
<tr>
<td>$-q_{-1}d$</td>
<td>6.9 3 ${q_{-1} = 7}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$s^{(1)}$</th>
<th>.0 7 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10s^{(1)}$</td>
<td>0.7 3</td>
</tr>
<tr>
<td>$-q_{-2}d$</td>
<td>0.0 0 ${q_{-2} = 0}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$s^{(2)}$</th>
<th>.7 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_{\text{frac}}$</td>
<td>.0 0 7 3</td>
</tr>
<tr>
<td>$q_{\text{frac}}$</td>
<td>.7 0</td>
</tr>
</tbody>
</table>

---

Fig. 14.2 Examples of high-radix division with integer and fractional operands.
14.2 Radix-2 SRT Division

Before discussing high-radix division, we try to solve the more pressing problem of using carry-save methods to speed up each iteration.

\[
s^{(j)} = 2s^{(j-1)} - q_j d
\]
with \(s^{(0)} = z\)
\(s^{(k)} = 2^k s\)
\(q_j \in \{-1, 1\}\)

Fig. 14.3 The new partial remainder, \(s^{(j)}\), as a function of the shifted old partial remainder, \(2s^{(j-1)}\), in radix-2 nonrestoring division.
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 \( s^{(0)} = z \)

\[
s^{(k)} = 2^ks
\]

\( q_{-j} \in \{-1, 0, 1\} \)

Fig. 14.4  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 $2s = (0.1\ldots)_{2's-compl}$
- $2s < -\frac{1}{2}$ means $2s = (1.0\ldots)_{2's-compl}$

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

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

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

$s^{(j)} \in [-\frac{1}{2}, \frac{1}{2})$

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

Fig. 14.5  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 $2s = (0.1\ldots)_{2's-compl}$ choose $q_{-j} = 1$

For $2s < -\frac{1}{2}$ or $2s = (1.0\ldots)_{2's-compl}$ choose $q_{-j} = -1$

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

$0 \leq 2s < +\frac{1}{2}$ or $2s = (0.0\ldots)_{2's-compl}$

$-\frac{1}{2} \leq 2s < 0$ or $2s = (1.1\ldots)_{2's-compl}$

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

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

$2s = (1.1110\ldots)_{2's-compl}$ 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

| \( z \) | 0.100 0101 |
| \( d \) | 0.1010 |
| \( -d \) | 1.0110 |

| \( s^{(0)} \) | 0.0100 0101 |
| 2\( s^{(0)} \) | 0.1000 101 |
| +(-\( d \)) | 1.0110 |

| \( s^{(1)} \) | 1.1110 101 |
| 2\( s^{(1)} \) | 1.1101 01 |

\( s^{(2)} = 2s^{(1)} \) | 1.1101 01 |
| 2\( s^{(2)} \) | 1.1010 1 |

\( s^{(3)} = 2s^{(2)} \) | 0.1010 1 |
| 2\( s^{(3)} \) | 1.0101 |
| +\( d \) | 0.1010 |

\( s^{(4)} \) | 1.1111 |
| +\( d \) | 0.1010 |

\( s^{(4)} \) | 0.1001 |
| \( s \) | 0.0000 0101 |
| \( q \) | 0.100-1 |
| \( q \) | 0.0110 |

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

\( \geq \frac{1}{2} \), so set \( q_{-1} = 1 \) and subtract

In \([-\frac{1}{2}, \frac{1}{2}]\), so set \( q_{-2} = 0 \)

In \([-\frac{1}{2}, \frac{1}{2}]\), so set \( q_{-3} = 0 \)

\(< -\frac{1}{2} \), so set \( q_{-4} = -1 \) and add

Negative, so add to correct

Uncorrected BSD quotient

Convert and subtract \( ulp \)
14.3 Using Carry-Save Adders

Fig. 14.7 Constant thresholds used for quotient digit selection in radix-2 division with $q_{k-j}$ in \{-1, 0, 1\}.
Quotient Digit Selection Based on Truncated PR

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

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

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 $[0, \frac{1}{2})$
Divider with Partial Remainder in Carry-Save Form

Fig. 14.8 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. 14.9  Overlap regions in radix-2 SRT division.

$q_j = -1$
$q_j = 0$
$q_j = 1$

$1 - d$
$1 - d$

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

$2d$
$-2d$

$1/2$

$d$

$s^{(j)}$
14.4 Choosing the Quotient Digits

Fig. 14.10  A $p$-$d$ plot for radix-2 division with $d \in [1/2,1)$, partial remainder in $[-d, d)$, and quotient digits in $[-1, 1]$. 
Design of the Quotient Digit Selection Logic

Shifted sum = \((u_1u_0 \cdot u_{-1}u_{-2} \cdots)\) \(_2\)s-compl

Shifted carry = \((v_1v_0 \cdot v_{-1}v_{-2} \cdots)\) \(_2\)s-compl

Approx shifted PR = \((t_1t_0 \cdot t_{-1}t_{-2})\) \(_2\)s-compl

Non0 = \(t_1' \lor t_0' \lor t_{-1}' = (t_1 t_0 t_{-1})'\)

Sign = \(t_1 (t_0' \lor t_{-1}')\)
14.5 Radix-4 SRT Division

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

$$s^{(i)} = 4s^{(i-1)} - q_{-j} d$$

with $s^{(0)} = z$ and $s^{(k)} = 4^k s$

---shift---

---subtract---

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. 14.12 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 $q_{-j} \in [-2, 2]$

$$s^{(j)} = 4s^{(j-1)} - q_{-j}d$$

with $s^{(0)} = z$ and $s^{(k)} = 4^k s$

$|\text{shift}|$

$|\text{subtract}|$

For this restriction to be feasible, we must have:

$s \in [-hd, hd)$ for some $h < 1$, 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. 14.14 A $p$-$d$ plot for radix-4 SRT division with quotient digit set $[-2, 2]$. 

Infeasible region ($p$ cannot be $\geq 8d/3$)
14.6 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. 14.15 Block diagram of radix-$r$ divider with partial remainder in stored-carry form.
15 Variations in Dividers

Chapter Goals
Discuss practical aspects of designing high-radix division schemes and cover other types of fast hardware dividers

Chapter Highlights
Building and using $p-d$ plots in practice
Prescaling simplifies $q$ digit selection
Parallel hardware (array) dividers
Shared hardware in multipliers/dividers
Square-rooting not special case of division
### Topics in This Chapter

1. **15.1. Quotient Digit Selection Revisited**
2. **15.2. Using \( p-d \) Plots in Practice**
3. **15.3. Division with Prescaling**
4. **15.4. Modular Dividers and Reducers**
5. **15.5. Array Dividers**
6. **15.6. Combined Multiply/Divide Units**
15.1 Quotient Digit Selection Revisited

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 $rhd - \alpha d \leq hd$ or $h \leq \alpha/(r - 1)$
To minimize the range restriction, we choose $h = \alpha/(r - 1)$

Fig. 15.1 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. 15.2  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 $[-\alpha, \alpha]$.

Choose $\beta + 1$

$(h + \beta + 1)d$

$(h + \beta)d$

Overlap region

$(-h + \beta + 1)d$

$(h + \beta)d$

Choose $\beta$

$4$ bits of $p$

$3$ bits of $d$

$3$ bits of $p$

$4$ bits of $d$

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

Standard $p$

$\phantom{.}xx.\phantom{.}xxxx$

Carry-save $p$

$\phantom{.}xx.\phantom{.}xxxxx$

$\phantom{.}xx.\phantom{.}xxxxx$

$\phantom{.}xx.\phantom{.}xxxxx$
Table Entries in the Quotient Digit Selection Logic

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

Note: $h = \alpha/(r-1)$
15.2 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. 15.4 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 \( \alpha = 2 \),
\( d_{\text{min}} = 1/2, h = \alpha/(r - 1) = 2/3 \)

\[ \Delta d = (1/2) \frac{4/3 - 1}{-2/3 + 2} = 1/8 \]
\[ \Delta p = (1/2)(4/3 - 1) = 1/6 \]

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

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

$$w \leq \Delta d/2 \quad \Rightarrow \quad v \leq \Delta p/2 \quad \Rightarrow \quad u \geq \Delta p/2$$
Fig. 15.5 The asymmetry of quotient digit selection process.

Fig. 15.6 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
15.3 Division with Prescaling

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

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

Thus, we perform the division $\frac{zm}{dm}$ 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.
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.
15.5 Array Dividers

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

XOR

FA

Dividend \( z = z_0 z_{-1} z_{-2} z_{-3} z_{-4} z_{-5} \)

Divisor \( d = d_0 d_{-1} d_{-2} d_{-3} \)

Quotient \( q = q_0 q_{-1} q_{-2} q_{-3} \)

Remainder \( s = 0 \cdot 0 \cdot 0 \cdot s_{-3} s_{-4} s_{-5} s_{-6} \)
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.

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

Critical path

Fig. 15.8
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.
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

## Topics in This Chapter

<table>
<thead>
<tr>
<th>Section</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.1</td>
<td>General Convergence Methods</td>
</tr>
<tr>
<td>16.2</td>
<td>Division by Repeated Multiplications</td>
</tr>
<tr>
<td>16.3</td>
<td>Division by Reciprocation</td>
</tr>
<tr>
<td>16.4</td>
<td>Speedup of Convergence Division</td>
</tr>
<tr>
<td>16.5</td>
<td>Hardware Implementation</td>
</tr>
<tr>
<td>16.6</td>
<td>Analysis of Lookup Table Size</td>
</tr>
</tbody>
</table>
16.1 General 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:**
\[
q = \frac{z}{d} = \frac{zx^{(0)}x^{(1)} \ldots x^{(m-1)}}{dx^{(0)}x^{(1)} \ldots x^{(m-1)}} \rightarrow \text{Converges to } q
\]

\[
\rightarrow \text{Force to 1}
\]

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

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

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

This choice transforms the recurrence equations into:
\[ d^{(i+1)} = d^{(i)} (2 - d^{(i)}) \quad \text{Set } d^{(0)} = d; \text{ iterate until } d^{(m)} \equiv 1 \]
\[ z^{(i+1)} = z^{(i)} (2 - d^{(i)}) \quad \text{Set } z^{(0)} = z; \text{ obtain } z/d = q \equiv z^{(m)} \]
\[ u^{(i+1)} = f(u^{(i)}, v^{(i)}) \]
\[ v^{(i+1)} = g(u^{(i)}, v^{(i)}) \quad \text{Fits the general form} \]
Determining the Rate of Convergence

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

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

Question 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 \( 1 - d^{(i+1)} \leq \varepsilon^2 \): Quadratic convergence

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

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

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>$d^{(i)} = d^{(i-1)} \cdot x^{(i-1)}$, with $d^{(0)} = d$</th>
<th>$x^{(i)} = 2 - d^{(i)}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$1 - y = (.1xxx \ xxxx \ xxxx \ xxxx)_\text{two} \geq 1/2$</td>
<td>$1 + y$</td>
</tr>
<tr>
<td>1</td>
<td>$1 - y^2 = (.11xx \ xxxx \ xxxx \ xxxx)_\text{two} \geq 3/4$</td>
<td>$1 + y^2$</td>
</tr>
<tr>
<td>2</td>
<td>$1 - y^4 = (.11111 \ xxxx \ xxxx \ xxxx)_\text{two} \geq 15/16$</td>
<td>$1 + y^4$</td>
</tr>
<tr>
<td>3</td>
<td>$1 - y^8 = (.11111 \ 1111 \ xxxx \ xxxx)_\text{two} \geq 255/256$</td>
<td>$1 + y^8$</td>
</tr>
<tr>
<td>4</td>
<td>$1 - y^{16} = (.11111 \ 1111 \ 1111 \ 1111)_\text{two} = 1 - \text{ulp}$</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
\[
x^{(i+1)} = x^{(i)} - f(x^{(i)}) / f'(x^{(i)})
\]

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

Fig. 16.2 Convergence to a root of \( f(x) = 0 \) in the Newton-Raphson method.
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 $\cong 0.1$
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 $\Rightarrow$ 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

$q = \frac{z}{d} = \frac{zx^{(0)}_1 x^{(1)}_1 \ldots x^{(m-1)}_1}{dx^{(0)}_1 x^{(1)}_1 \ldots x^{(m-1)}_1}$

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.

\[ d \cdot x^{(0)} \cdot x^{(1)} \cdot x^{(2)} = (0.1111 \ 1111 \ldots)_{\text{two}} \]

Better approx

A more accurate approximation can be obtained by reading a 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: \( d = 0.1011 \ xxxx \ldots \quad (11/16 \leq d < 12/16) \)
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.

After table lookup and 1st pair of multiplications, replacing several iterations

After the 2nd pair of multiplications

Iterations
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 xxxx} )</th>
<th>(x^{(0+)} = 1. \text{ xxxx 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

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

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

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.