Part III
Multiplication

<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
III  Multiplication

Review multiplication schemes and various speedup methods
- Multiplication is heavily used (in arith & array indexing)
- Division = reciprocation + multiplication
- Multiplication speedup: high-radix, tree, recursive
- Bit-serial, modular, and array multipliers

Topics in This Part

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 9</td>
<td>Basic Multiplication Schemes</td>
</tr>
<tr>
<td>Chapter 10</td>
<td>High-Radix Multipliers</td>
</tr>
<tr>
<td>Chapter 11</td>
<td>Tree and Array Multipliers</td>
</tr>
<tr>
<td>Chapter 12</td>
<td>Variations in Multipliers</td>
</tr>
</tbody>
</table>
“Well, well, for a rabbit, you’re not very good at multiplying, are you?”
9 Basic Multiplication Schemes

Chapter Goals

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

Chapter Highlights

Multiplication = multioperand addition
Hardware, firmware, software algorithms
Multiplying 2’s-complement numbers
The special case of one constant operand
Basic Multiplication Schemes: Topics

<table>
<thead>
<tr>
<th>Topics in This Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1 Shift/Add Multiplication Algorithms</td>
</tr>
<tr>
<td>9.2 Programmed Multiplication</td>
</tr>
<tr>
<td>9.3 Basic Hardware Multipliers</td>
</tr>
<tr>
<td>9.4 Multiplication of Signed Numbers</td>
</tr>
<tr>
<td>9.5 Multiplication by Constants</td>
</tr>
<tr>
<td>9.6 Preview of Fast Multipliers</td>
</tr>
</tbody>
</table>
9.1 Shift/Add Multiplication Algorithms

Notation for our discussion of multiplication algorithms:

\[
\begin{align*}
    a & \quad \text{Multiplicand} & a_{k-1}a_{k-2} \ldots a_1a_0 \\
    x & \quad \text{Multiplier} & x_{k-1}x_{k-2} \ldots x_1x_0 \\
    p & \quad \text{Product} (a \times x) & p_{2k-1}p_{2k-2} \ldots p_3p_2p_1p_0
\end{align*}
\]

Initially, we assume unsigned operands

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

Multiplication with right shifts: top-to-bottom accumulation

\[ p^{(j+1)} = (p^{(j)} + x_j a 2^k) 2^{-1} \]

with \( p^{(0)} = 0 \) and
\[ p^{(k)} = p = ax + p^{(0)}2^{-k} \]

Multiplication with left shifts: bottom-to-top accumulation

\[ p^{(j+1)} = 2p^{(j)} + x_{k-j-1}a \]

with \( p^{(0)} = 0 \) and
\[ p^{(k)} = p = ax + p^{(0)}2^k \]
Why Premultiply the Multiplicand by $2^k$?

Addition takes place between the dashed lines in the figure below. The 0th PP is eventually shifted right by $k$ bits, the 1st by $k - 1$ bits, … Even though the cumulative PP widens by 1 bit at each step, the addition is always $k$ bits wide.

$$p^{(j+1)} = (p^{(j)} + x_j a 2^k) 2^{-1}$$

with $p^{(0)} = 0$ and

$$p^{(0)} = 0$$

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

Right-shift algorithm

<table>
<thead>
<tr>
<th></th>
<th>1 0 1 0</th>
<th>1 0 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>1 0 1 0</td>
<td></td>
</tr>
<tr>
<td>$x$</td>
<td>1 0 1 1</td>
<td></td>
</tr>
<tr>
<td>$p^{(0)}$</td>
<td>0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>$+x_0a$</td>
<td>1 0 1 0</td>
<td></td>
</tr>
<tr>
<td>$2p^{(1)}$</td>
<td>0 1 0 1 0</td>
<td></td>
</tr>
<tr>
<td>$p^{(1)}$</td>
<td>0 1 0 1 0</td>
<td></td>
</tr>
<tr>
<td>$+x_1a$</td>
<td>1 0 1 0</td>
<td></td>
</tr>
<tr>
<td>$2p^{(2)}$</td>
<td>0 1 1 1 1 0</td>
<td></td>
</tr>
<tr>
<td>$p^{(2)}$</td>
<td>0 1 1 1 1 0</td>
<td></td>
</tr>
<tr>
<td>$+x_2a$</td>
<td>0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>$2p^{(3)}$</td>
<td>0 0 1 1 1 1 0</td>
<td></td>
</tr>
<tr>
<td>$p^{(3)}$</td>
<td>0 0 1 1 1 1 0</td>
<td></td>
</tr>
<tr>
<td>$+x_3a$</td>
<td>1 0 1 0</td>
<td></td>
</tr>
<tr>
<td>$2p^{(4)}$</td>
<td>0 1 1 0 1 1 0</td>
<td></td>
</tr>
<tr>
<td>$p^{(4)}$</td>
<td>0 1 1 0 1 1 0</td>
<td></td>
</tr>
</tbody>
</table>

Left-shift algorithm

<table>
<thead>
<tr>
<th></th>
<th>1 0 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>1 0 1 0</td>
</tr>
<tr>
<td>$x$</td>
<td>1 0 1 1</td>
</tr>
<tr>
<td>$p^{(0)}$</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>$2p^{(0)}$</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>$+x_0a$</td>
<td>1 0 1 0</td>
</tr>
<tr>
<td>$p^{(1)}$</td>
<td>0 1 0 1 0 0</td>
</tr>
<tr>
<td>$2p^{(1)}$</td>
<td>0 1 0 1 0 0</td>
</tr>
<tr>
<td>$+x_1a$</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>$p^{(2)}$</td>
<td>0 1 0 1 0 0</td>
</tr>
<tr>
<td>$2p^{(2)}$</td>
<td>0 1 0 1 0 0</td>
</tr>
<tr>
<td>$+x_2a$</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>$p^{(3)}$</td>
<td>0 1 0 1 0 0</td>
</tr>
<tr>
<td>$2p^{(3)}$</td>
<td>0 1 0 1 0 0</td>
</tr>
<tr>
<td>$+x_3a$</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>$p^{(4)}$</td>
<td>0 1 1 0 1 1 0</td>
</tr>
</tbody>
</table>

\[ p^{(j+1)} = (p^{(j)} + x_j a 2^k) 2^{-1} \]

Check:
\[ 10 \times 11 = 64 + 32 + 8 + 4 + 2 = 110 \]
Examples of Basic Multiplication (Continued)

<table>
<thead>
<tr>
<th></th>
<th>Right-shift algorithm</th>
<th>Left-shift algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-----------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>$a$</td>
<td>1 0 1 0</td>
<td>1 0 1 0</td>
</tr>
<tr>
<td>$x$</td>
<td>1 0 1 1</td>
<td>1 0 1 1</td>
</tr>
<tr>
<td>$p^{(0)}$</td>
<td>0 0 0 0</td>
<td>$p^{(0)}$</td>
</tr>
<tr>
<td>$+$</td>
<td>$x_0 a$</td>
<td>$2p^{(0)}$</td>
</tr>
<tr>
<td></td>
<td>1 0 1 0</td>
<td>$+$ $x_3 a$</td>
</tr>
<tr>
<td>$2p^{(1)}$</td>
<td>0 1 0 1 0</td>
<td>$p^{(1)}$</td>
</tr>
<tr>
<td>$p^{(1)}$</td>
<td>0 1 0 1 0</td>
<td>$2p^{(1)}$</td>
</tr>
<tr>
<td>$+$</td>
<td>$x_1 a$</td>
<td>$+$ $x_2 a$</td>
</tr>
<tr>
<td></td>
<td>1 0 1 0</td>
<td>2 $p^{(2)}$</td>
</tr>
<tr>
<td>$2p^{(2)}$</td>
<td>0 1 1 1 1 0</td>
<td>$+$ $x_1 a$</td>
</tr>
<tr>
<td>$+$</td>
<td>$x_3 a$</td>
<td>$p^{(3)}$</td>
</tr>
<tr>
<td></td>
<td>1 0 1 0</td>
<td>$2p^{(3)}$</td>
</tr>
<tr>
<td>$2p^{(3)}$</td>
<td>0 1 1 0 1 1 0</td>
<td>$+$ $x_0 a$</td>
</tr>
<tr>
<td>$+$</td>
<td>$x_2 a$</td>
<td>$p^{(4)}$</td>
</tr>
<tr>
<td></td>
<td>1 0 1 0</td>
<td>$2p^{(4)}$</td>
</tr>
<tr>
<td>$p^{(4)}$</td>
<td>0 1 1 0 1 1 0</td>
<td></td>
</tr>
</tbody>
</table>
9.2 Programmed Multiplication

{Using right shifts, multiply unsigned \( m_{\text{cand}} \) and \( m_{\text{ier}} \), storing the resultant \( 2^k \)-bit product in \( p_{\text{high}} \) and \( p_{\text{low}} \).

Registers: \( R0 \) holds 0 \( \quad \) Rc for counter
\( Ra \) for \( m_{\text{cand}} \) \( \quad \) Rx for \( m_{\text{ier}} \)
\( Rp \) for \( p_{\text{high}} \) \( \quad \) Rq for \( p_{\text{low}} \}

{Load operands into registers \( Ra \) and \( Rx \})
mult: load \( Ra \) with \( m_{\text{cand}} \)
      load \( Rx \) with \( m_{\text{ier}} \)

{Initialize partial product and counter}
copy \( R0 \) into \( Rp \)
copy \( R0 \) into \( Rq \)
load \( k \) into \( Rc \)

{Begin multiplication loop}
m_loop: shift \( Rx \) right 1 \( \{ \text{LSB moves to carry flag} \} \)
branch no_add if carry = 0
add \( Ra \) to \( Rp \) \( \{ \text{carry flag is set to cout} \} \)
no_add: rotate \( Rp \) right 1 \( \{ \text{carry to MSB, LSB to carry} \} \)
rotate \( Rq \) right 1 \( \{ \text{carry to MSB, LSB to carry} \} \)
decr \( Rc \) \( \{ \text{decrement counter by 1} \} \)
branch m_loop if \( Rc \neq 0 \)

{Store the product}
store \( Rp \) into \( p_{\text{high}} \)
store \( Rq \) into \( p_{\text{low}} \)

m_done: ...

Fig. 9.3 Programmed multiplication (right-shift algorithm).
Time Complexity of Programmed Multiplication

Assume \( k \)-bit words

\( k \) iterations of the main loop
6-7 instructions per iteration, depending on the multiplier bit

Thus, \( 6k + 3 \) to \( 7k + 3 \) machine instructions, ignoring operand loads and result store

\( k = 32 \) implies \( 200^+ \) instructions on average

This is too slow for many modern applications!

Microprogrammed multiply would be somewhat better
9.3 Basic Hardware Multipliers

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

\[ p^{(j+1)} = (p^{(j)} + x_j 2^k) 2^{-1} \]

- **Add**
- **Shift right**
Example of Hardware Multiplication

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

\[ p^{(j+1)} = (p^{(j)} + x_j a 2^k) 2^{-1} \]

\[ \text{add} \]

\[ \text{shift right} \]
Performing Add and Shift in One Clock Cycle

Fig. 9.5 Combining the loading and shifting of the double-width register holding the partial product and the partially used multiplier.
Sequential Multiplication with Left Shifts

Fig. 9.4b  Hardware realization of the sequential multiplication algorithm with left shifts and additions.
9.4 Multiplication of Signed Numbers

---

Fig. 9.6 Sequential multiplication of 2’s-complement numbers with right shifts (positive multiplier).

Negative multiplicand, positive multiplier:

No change, other than looking out for proper sign extension
The Case of a Negative Multiplier

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

### Negative multiplicand, negative multiplier:

In last step (the sign bit), subtract rather than add.

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

<table>
<thead>
<tr>
<th>$p^{(0)}$</th>
<th>0 0 0 0 0</th>
<th>Check: $-10 \times -11 = 110$</th>
</tr>
</thead>
<tbody>
<tr>
<td>+$x_0a$</td>
<td>1 0 1 1 0</td>
<td></td>
</tr>
</tbody>
</table>

| $2p^{(1)}$ | 1 1 0 1 1 0 |
| $p^{(1)}$  | 1 1 0 1 1 0 |
| +$x_1a$   | 0 0 0 0 0   |

| $2p^{(2)}$ | 1 1 1 0 1 1 0 |
| $p^{(2)}$  | 1 1 1 0 1 1 0 |
| +$x_2a$   | 1 0 1 1 0   |

| $2p^{(3)}$ | 1 1 0 0 1 1 1 0 |
| $p^{(3)}$  | 1 1 0 0 1 1 1 0 |
| +$x_3a$   | 0 0 0 0 0   |

| $2p^{(4)}$ | 1 1 1 0 0 1 1 1 0 |
| $p^{(4)}$  | 1 1 1 0 0 1 1 1 0 |
| +($-x_4a$) | 0 1 0 1 0   |

| $2p^{(5)}$ | 0 0 0 1 1 0 1 1 1 0 |
| $p^{(5)}$  | 0 0 0 1 1 0 1 1 1 0 |

Check: $-10 \times -11 = 110 = 64 + 32 + 8 + 4 + 2$
Signed 2’s-Complement Hardware Multiplier

Fig. 9.8 The 2’s-complement sequential hardware multiplier.
Booth’s Recoding

Table 9.1  Radix-2 Booth’s recoding

<table>
<thead>
<tr>
<th>$x_i$</th>
<th>$x_{i-1}$</th>
<th>$y_i$</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>No string of 1s in sight</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>End of string of 1s in $x$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>-1</td>
<td>Beginning of string of 1s in $x$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Continuation of string of 1s in $x$</td>
</tr>
</tbody>
</table>

Example

1 0 0 1 1 1 0 1 1 0 1 0 1 1 1 0  
(1) -1 0 1 0 0 -1 1 0 -1 1 -1 1 0 0 -1 0  
Operand $x$  
Recoded version $y$

Justification

$$2^j + 2^{j-1} + \ldots + 2^{i+1} + 2^i = 2^{i+1} - 2^i$$
Example Multiplication with Booth’s Recoding

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

<table>
<thead>
<tr>
<th>(x_i)</th>
<th>(x_{i-1})</th>
<th>(y_i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

| \(a\)   | 1 0 1 1 0 |
| \(x\)   | 1 0 1 0 1 |
| \(y\)   | -1 1 -1 1 -1 |

|= Booth-recoded|

| \(p^{(0)}\) | 0 0 0 0 0 |
| \(+y_0a\)   | 0 1 0 1 0 |

Check:
\(-10 \times -11 = 110\)

\[
\begin{align*}
2p^{(1)} & = 0 0 1 0 1 0 \\
p^{(1)} & = 0 0 1 0 1 0 \\
+\ y_1a & = 1 0 1 1 0 \\
\end{align*}
\]

\[
\begin{align*}
2p^{(2)} & = 1 1 1 0 1 1 0 \\
p^{(2)} & = 1 1 1 0 1 1 0 \\
+\ y_2a & = 0 1 0 1 0 \\
\end{align*}
\]

\[
\begin{align*}
2p^{(3)} & = 0 0 0 1 1 1 1 0 \\
p^{(3)} & = 0 0 0 1 1 1 1 0 \\
+\ y_3a & = 1 0 1 1 0 \\
\end{align*}
\]

\[
\begin{align*}
2p^{(4)} & = 1 1 1 0 0 1 1 0 \\
p^{(4)} & = 1 1 1 0 0 1 1 0 \\
y_4a & = 0 1 0 1 0 \\
\end{align*}
\]

\[
\begin{align*}
2p^{(5)} & = 0 0 0 1 1 0 1 1 0 \\
p^{(5)} & = 0 0 0 1 1 0 1 1 1 0 \\
\end{align*}
\]

---

\(a\) \ 1 0 1 1 0
\(x\) \ 1 0 1 0 1
\(y\) \ -1 1 -1 1 -1

 applyMiddleware: false

---
9.5 Multiplication by Constants

Explicit, e.g. \( y := 12 \times x + 1 \)

Address of \( A[i, j] = \text{base} + n \times i + j \)

Software aspects:

Optimizing compilers replace multiplications by shifts/adds/subs
Produce efficient code using as few registers as possible
Find the best code by a time/space-efficient algorithm

Hardware aspects:

Synthesize special-purpose units such as filters
\[
y[t] = a_0 x[t] + a_1 x[t - 1] + a_2 x[t - 2] + b_1 y[t - 1] + b_2 y[t - 2]
\]
Multiplication Using Binary Expansion

Example: Multiply R1 by the constant $113 = (1 1 1 0 0 0 1)_2$

\[
\begin{align*}
R2 & \leftarrow R1 \text{ shift-left 1} \\
R3 & \leftarrow R2 + R1 \\
R6 & \leftarrow R3 \text{ shift-left 1} \\
R7 & \leftarrow R6 + R1 \\
R112 & \leftarrow R7 \text{ shift-left 4} \\
R113 & \leftarrow R112 + R1
\end{align*}
\]

Shift, add

Shift

$R_i$: Register that contains $i$ times (R1)

This notation is for clarity; only one register other than R1 is needed

Shorter sequence using shift-and-add instructions

\[
\begin{align*}
R3 & \leftarrow R1 \text{ shift-left 1} + R1 \\
R7 & \leftarrow R3 \text{ shift-left 1} + R1 \\
R113 & \leftarrow R7 \text{ shift-left 4} + R1
\end{align*}
\]
Multiplication via Recoding

Example: Multiply R1 by $113 = (1 1 1 0 0 0 1)_2 = (1 0 0 1 0 0 0 1)_2$

\[
\begin{align*}
R8 & \gets R1 \text{ shift-left 3} \\
R7 & \gets R8 - R1 \\
R112 & \gets R7 \text{ shift-left 4} \\
R113 & \gets R112 + R1
\end{align*}
\]

Shorter sequence using shift-and-add/subtract instructions

\[
\begin{align*}
R7 & \gets R1 \text{ shift-left 3} - R1 \\
R113 & \gets R7 \text{ shift-left 4} + R1
\end{align*}
\]

6 shift or add (3 shift-and-add) instructions needed without recoding

The canonic signed-digit representation of a number contains no consecutive nonzero digits: average number of shift-adds is $O(k/3)$
Multiplication via Factorization

Example: Multiply R1 by 119 = 7 \times 17

= (8 - 1) \times (16 + 1)

\begin{align*}
R8 & \leftarrow R1 \text{ shift-left 3} \\
R7 & \leftarrow R8 - R1 \\
R112 & \leftarrow R7 \text{ shift-left 4} \\
R119 & \leftarrow R112 + R7
\end{align*}

Shorter sequence using shift-and-add/subtract instructions

\begin{align*}
R7 & \leftarrow R1 \text{ shift-left 3} - R1 \\
R119 & \leftarrow R7 \text{ shift-left 4} + R7
\end{align*}

119 = (1110111)_{\text{two}} = (100000-1000-1)_{\text{two}}

More instructions may be needed without factorization

Requires a scratch register for holding the 7 multiple
Multiplication by Multiple Constants

Example: Multiplying a number by 45, 49, and 65

\[
\begin{align*}
R9 & \leftarrow R1 \text{ shift-left } 3 + R1 \\
R45 & \leftarrow R9 \text{ shift-left } 2 + R9 \\
R7 & \leftarrow R1 \text{ shift-left } 3 - R1 \\
R49 & \leftarrow R7 \text{ shift-left } 3 - R7 \\
R65 & \leftarrow R1 \text{ shift-left } 6 + R1
\end{align*}
\]

Separate solutions:

- 5 shift-add/subtract operations

A combined solution for all three constants

\[
\begin{align*}
R65 & \leftarrow R1 \text{ shift-left } 6 + R1 \\
R49 & \leftarrow R65 - R1 \text{ left-shift } 4 \\
R45 & \leftarrow R49 - R1 \text{ left-shift } 2
\end{align*}
\]

A programmable block can perform any of the three multiplications.
9.6 Preview of Fast Multipliers

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

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

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

In Chapter 12, we cover all remaining multiplication topics:

- Bit-serial multipliers
- Modular multipliers
- Multiply-add units
- Squaring as a special case
Chapter Goals

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

Chapter Highlights

High radix gives rise to “difficult” multiples
Recoding (change of digit-set) as remedy
Carry-save addition reduces cycle time
Implementation and optimization methods
High-Radix Multipliers: Topics

Topics in This Chapter

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

Multiplication with right shifts in radix \( r \): top-to-bottom accumulation

\[
p^{(j+1)} = (p^{(j)} + x_j a r^k) r^{-1}
\]

with
\[
p^{(0)} = 0 \quad \text{and} \quad p^{(k)} = p = ax + p^{(0)}r^{-k}
\]

Multiplication with left shifts in radix \( r \): bottom-to-top accumulation

\[
p^{(j+1)} = r p^{(j)} + x_{k-j-1} a
\]

with
\[
p^{(0)} = 0 \quad \text{and} \quad p^{(k)} = p = ax + p^{(0)}r^k
\]
Radix-4 Multiplication in Dot Notation

Number of cycles is halved, but now the “difficult” multiple $3a$ must be dealt with.

Fig. 9.1

Fig. 10.1 Radix-4, or two-bit-at-a-time, multiplication in dot notation
A Possible Design for a Radix-4 Multiplier

Precomputed via shift-and-add \((3a = 2a + a)\)

\(k/2 + 1\) cycles, rather than \(k\)

One extra cycle over \(k/2\)

not too bad, but we would like to avoid it if possible

Solving this problem for radix 4 may also help when dealing with even higher radices

Fig. 10.2 The multiple generation part of a radix-4 multiplier with precomputation of \(3a\).
### Example Radix-4 Multiplication Using 3a

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>a</strong></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td><strong>3a</strong></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td><strong>x</strong></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

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

\[
\begin{align*}
 p^{(0)} & \quad 0 1 1 0 0 \\
 +(x_1x_0)_\text{two}a & \quad 0 0 1 1 0 0 \\
\hline
 4p^{(1)} & \quad 0 0 1 1 0 0 \\
 p^{(1)} & \quad 0 0 1 1 0 0 \\
 +(x_3x_2)_\text{two}a & \quad 0 1 0 0 1 0 \\
\hline
 4p^{(2)} & \quad 0 1 0 1 0 1 0 0 \\
 p^{(2)} & \quad 0 1 0 1 0 0 0 0 \\
\end{align*}
\]
A Second Design for a Radix-4 Multiplier

Fig. 10.4  The multiple generation part of a radix-4 multiplier based on replacing \( 3a \) with \( 4a \) (carry into next higher radix-4 multiplier digit) and \( -a \).
### 10.2 Modified Booth’s Recoding

**Table 10.1 Radix-4 Booth’s recoding yielding \((z_{k/2} \ldots z_{1}z_{0})_{\text{four}}\)**

<table>
<thead>
<tr>
<th>(x_{i+1})</th>
<th>(x_i)</th>
<th>(x_{i-1})</th>
<th>(y_{i+1})</th>
<th>(y_i)</th>
<th>(z_{i/2})</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>No string of 1s in sight</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>End of string of 1s</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Isolated 1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>End of string of 1s</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>-2</td>
<td>Beginning of string of 1s</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
<td>End a string, begin new one</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>Beginning of string of 1s</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Continuation of string of 1s</td>
</tr>
</tbody>
</table>

**Example**

\[
\begin{array}{ccccccccccccccc}
1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\
(1) & -1 & 0 & 1 & 0 & 0 & -1 & 1 & 0 & -1 & 1 & -1 & 1 & 0 & 0 & -1 & 0 \\
(1) & -2 & 2 & -1 & 2 & -1 & -1 & -1 & 0 & -2
\end{array}
\]

Operand \(x\)

Recoded version \(y\)

Radix-4 version \(z\)
Example Multiplication via Modified Booth’s Recoding

---

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>a</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>x</strong></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td><strong>z</strong></td>
<td>-1</td>
<td>-2</td>
<td>Radix-4</td>
<td></td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>p^{(0)}</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+z_0a</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>4p^{(1)}</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>p^{(1)}</strong></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>+z_1a</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>4p^{(2)}</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>p^{(2)}</strong></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

---

![Fig. 10.5](image)

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

Fig. 10.6 The multiple generation part of a radix-4 multiplier based on Booth’s recoding.
10.3 Using Carry-Save Adders

Fig. 10.7  Radix-4 multiplication with a carry-save adder used to combine the cumulative partial product, $x_i a$, and $2x_{i+1} a$ into two numbers.
Keeping the Partial Product in Carry-Save Form

Fig. 10.8  Radix-2 multiplication with the upper half of the cumulative partial product kept in stored-carry form.
Fig. 10.9   Radix-4 multiplication with a CSA used to combine the stored-carry cumulative partial product and \( z_{i/2}a \) into two numbers.
Radix-4 Booth’s Recoding for Parallel Multiplication

Fig. 10.10  Booth recoding and multiple selection logic for high-radix or parallel multiplication.
Yet Another Design for Radix-4 Multiplication

Fig. 10.11  Radix-4 multiplication, with the cumulative partial product, $x_i a$, and $2x_{i+1} a$ combined into two numbers by two CSAs.

(4; 2)-counter

Old Cumulative Partial Product

New Cumulative Partial Product

CSA

Adder

2-Bit Adder

To the Lower Half of Partial Product

FF

Mux

Multiplier

$0 \quad 2a$

$0 \quad a$

$x_i+1 \quad x_i$
10.4 Radix-8 and Radix-16 Multipliers

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

Remove this mux & CSA and replace the 4-bit shift (adder) with a 3-bit shift (adder) to get a radix-8 multiplier (cycle time will remain the same, though)

A radix-16 multiplier design becomes a radix-256 multiplier if radix-4 Booth’s recoding is applied first (the muxes are replaced by Booth recoding and multiple selection logic)

Fig. 10.12
Fig. 10.13  High-radix multipliers as intermediate between sequential radix-2 and full-tree multipliers.
10.5 Multibeat Multipliers

Fig. 10.15   Two-phase clocking for sequential logic.

Observation: Half of the clock cycle goes to waste
Twin-Beat and Three-Beat Multipliers

This radix-64 multiplier runs at the clock rate of a radix-8 design (2X speed)

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

Fig. 10.16 Conceptual view of a three-beat multiplier.
10.6 VLSI Complexity Issues

A radix-$2^b$ multiplier requires:

- $bk$ two-input AND gates to form the partial products bit-matrix
- $O(bk)$ area for the CSA tree
- At least $\Theta(k)$ area for the final carry-propagate adder

Total area: $A = O(bk)$
Latency: $T = O((k/b) \log b + \log k)$

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

- $AT$ grows at least as fast as $k^{3/2}$
- $AT^2$ is at least proportional to $k^2$

The preceding radix-$2^b$ implementations are suboptimal, because:

- $AT = O(k^2 \log b + bk \log k)$
- $AT^2 = O((k^3/b) \log^2 b)$
Comparing High- and Low-Radix Multipliers

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

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

<table>
<thead>
<tr>
<th></th>
<th>Low-Cost $b = O(1)$</th>
<th>High Speed $b = O(k)$</th>
<th>$AT$- or $AT^2$-Optimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>$AT$</td>
<td>$O(k^2)$</td>
<td>$O(k^2 \log k)$</td>
<td>$O(k^{3/2})$</td>
</tr>
<tr>
<td>$AT^2$</td>
<td>$O(k^3)$</td>
<td>$O(k^2 \log^2 k)$</td>
<td>$O(k^2)$</td>
</tr>
</tbody>
</table>

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

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

Thus, when an application requires many independent multiplications, it is more cost-effective to use a large number of slower multipliers

High-radix multiplier latency can be reduced from $O((k/b) \log b + \log k)$ to $O(k/b + \log k)$ through more effective pipelining (Chapter 11)
Chapter Goals

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

Chapter Highlights

Tree multiplier = reduction tree
+ redundant-to-binary converter
Avoiding full sign extension in multiplying signed numbers

Array multiplier = one-sided reduction tree
+ ripple-carry adder
Tree and Array Multipliers: Topics

<table>
<thead>
<tr>
<th>Topics in This Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.1. Full-Tree Multipliers</td>
</tr>
<tr>
<td>11.2. Alternative Reduction Trees</td>
</tr>
<tr>
<td>11.3. Tree Multipliers for Signed Numbers</td>
</tr>
<tr>
<td>11.4. Partial-Tree and Truncated Multipliers</td>
</tr>
<tr>
<td>11.5. Array Multipliers</td>
</tr>
<tr>
<td>11.6. Pipelined Tree and Array Multipliers</td>
</tr>
</tbody>
</table>
11.1 Full-Tree Multipliers

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

**Fig. 11.1** General structure of a full-tree multiplier.
Full-Tree versus Partial-Tree Multiplier

Schematic diagrams for full-tree and partial-tree multipliers.
Variations in Full-Tree Multiplier Design

Designs are distinguished by variations in three elements:

1. Multiple-forming circuits
2. Partial products reduction tree
3. Redundant-to-binary converter

Fig. 11.1
Example of Variations in CSA Tree Design

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

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

Corrections shown in red

Fig. 11.2 Two different binary 4 × 4 tree multipliers.
Details of a CSA Tree

Fig. 11.3 Possible CSA tree for a $7 \times 7$ tree multiplier.

CSA trees are quite irregular, causing some difficulties in VLSI realization.

Thus, our motivation to examine alternate methods for partial products reduction.
11.2 Alternative Reduction Trees

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

\[ 11 + \psi_1 = 2\psi_1 + 3 \]

Therefore, \( \psi_1 = 8 \) carries are needed.
Due to its recursive structure, a binary tree is more regular than a 3-to-2 reduction tree when laid out in VLSI.
Example Multiplier with 4-to-2 Reduction Tree

Even if 4-to-2 reduction is implemented using two CSA levels, design regularity potentially makes up for the larger number of logic levels.

Similarly, using Booth’s recoding may not yield any advantage, because it introduces irregularity.

Fig. 11.6 Layout of a partial-products reduction tree composed of 4-to-2 reduction modules. Each solid arrow represents two numbers.
11.3 Tree Multipliers for Signed Numbers

From Fig. 8.19a  Sign extension in multioperand addition.

The difference in multiplication is the shifting sign positions

Fig. 11.7  Sharing of full adders to reduce the CSA width in a signed tree multiplier.
Using the Negative-Weight Property of the Sign Bit

Sign extension is a way of converting negatively weighted bits (negabits) to positively weighted bits (posibits) to facilitate reduction, but there are other methods of accomplishing the same without introducing a lot of extra bits.

Baugh and Wooley have contributed two such methods.

Fig. 11.8 Baugh-Wooley 2’s-complement multiplication.
The Baugh-Wooley Method and Its Modified Form

The Baugh-Wooley Method and Its Modified Form

**c. Baugh-Wooley**

\[-a_4x_0 = a_4(1 - x_0) - a_4 = a_4x_0' - a_4\]

\[-a_4 \quad a_4x_0' \]

\[a_4 \quad a_4 \]

In next column

\[-a_4x_0 = a_4(1 - x_0) - 1 = (a_4x_0) - 1\]

\[-1 \quad (a_4x_0)'\]

\[1 \quad 1\]

In next column

**d. Modified B-W**

\[-a_4x_0 = (1 - a_4x_0) - 1 = (a_4x_0)' - 1\]

\[-1 \quad (a_4x_0)'\]

\[1 \quad 1\]
Alternate Views of the Baugh-Wooley Methods

+ 0 0 \(-a_4x_3\) \(-a_4x_2\) \(-a_4x_1\) \(-a_4x_0\)  
+ 0 0 \(-a_3x_4\) \(-a_2x_4\) \(-a_1x_4\) \(-a_0x_4\)  

\(- 0 0 a_4x_3\) \(a_4x_2\) \(a_4x_1\) \(a_4x_0\)  
\(- 0 0 a_3x_4\) \(a_2x_4\) \(a_1x_4\) \(a_0x_4\)  

+ 1 1 \(\frac{a_4}{a_4}x_3\) \(\frac{a_4}{a_4}x_2\) \(\frac{a_4}{a_4}x_1\) \(\frac{a_4}{a_4}x_0\)  
+ 1 1 \(\frac{a_3}{a_3}x_4\) \(\frac{a_2}{a_2}x_4\) \(\frac{a_1}{a_1}x_4\) \(\frac{a_0}{a_0}x_4\)  

\(\frac{a_4}{a_4}\) \(\frac{a_4}{a_4}\) \(\frac{a_4}{a_4}\) \(\frac{a_4}{a_4}\)  
\(\frac{a_4}{a_4}\) \(\frac{a_4}{a_4}\) \(\frac{a_4}{a_4}\) \(\frac{a_4}{a_4}\)  

\(a_4\) \(a_4\) \(a_4\) \(a_4\)  
\(x_4\) \(x_4\) \(x_4\) \(x_4\)  

---

a. Unsigned  
\[ \begin{array}{ccccccc}
1 & a_4 & x_3 & a_3 & x_2 & a_2 & x_1 & a_1 & x_0 \\
- a_4 & x_4 & x_3 & x_2 & x_1 & x_0 \\
\end{array} \]  

b. 2’s-complement  
\[ \begin{array}{ccccccc}
1 & a_4 & x_4 & a_3 & x_3 & a_2 & x_2 & a_1 & x_1 \\
- a_4 & x_4 & x_3 & x_2 & x_1 & x_0 \\
\end{array} \]  

c. Baugh-Wooley  
\[ \begin{array}{ccccccc}
1 & a_4 & x_3 & a_3 & x_2 & a_2 & x_1 & a_1 & x_0 \\
- a_4 & x_4 & x_3 & x_2 & x_1 & x_0 \\
\end{array} \]  

d. Modified B-W  
\[ \begin{array}{ccccccc}
1 & a_4 & x_3 & a_3 & x_2 & a_2 & x_1 & a_1 & x_0 \\
- a_4 & x_4 & x_3 & x_2 & x_1 & x_0 \\
\end{array} \]
11.4 Partial-Tree and Truncated Multipliers

High-radix versus partial-tree multipliers: The difference is quantitative, not qualitative

For small $h$, say $\leq 8$ bits, we view the multiplier of Fig. 11.9 as high-radix

When $h$ is a significant fraction of $k$, say $k/2$ or $k/4$, then we tend to view it as a partial-tree multiplier

Better design through pipelining to be covered in Section 11.6

Fig. 11.9 General structure of a partial-tree multiplier.
Why Truncated Multipliers?

Nearly half of the hardware in array/tree multipliers is there to get the last bit right (1 dot = one FPGA cell)

Max error = $\frac{8}{2} + \frac{7}{4} + \frac{6}{8} + \frac{5}{16} + \frac{4}{32} + \frac{3}{64} + \frac{2}{128} + \frac{1}{256} = 7.004$ ulp

Mean error = $1.751$ ulp

Fig. 11.10 The idea of a truncated multiplier with 8-bit fractional operands.
Truncated Multipliers with Error Compensation

We can introduce additional “dots” on the left-hand side to compensate for the removal of dots from the right-hand side.

Constant compensation

Variable compensation

Constant and variable error compensation for truncated multipliers.

- Max error = +4 \(ulp\)
- Max error \(\approx -3 \ulp\)
- Mean error = \(? \ulp\)

- Max error = +? \(ulp\)
- Max error \(\approx -? \ulp\)
- Mean error = ? \(ulp\)
11.5 Array Multipliers

Fig. 11.11 A basic array multiplier uses a one-sided CSA tree and a ripple-carry adder.

Fig. 11.12 Details of a $5 \times 5$ array multiplier using FA blocks.
Modifications in a 5 x 5 array multiplier to deal with 2’s-complement inputs using the Baugh-Wooley method or to shorten the critical path.
Array Multiplier Built of Modified Full-Adder Cells

Fig. 11.14  Design of a 5 \times 5 array multiplier with two additive inputs and full-adder blocks that include AND gates.
Array Multiplier without a Final Carry-Propagate Adder

Fig. 11.15 Conceptual view of a modified array multiplier that does not need a final carry-propagate adder.

Fig. 11.16 Carry-save addition, performed in level \( i \), extends the conditionally computed bits of the final product.

All remaining bits of the final product produced only 2 gate levels after \( p_{k-1} \).
11.6 Pipelined Tree and Array Multipliers

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

Fig. 11.17 Efficiently pipelined partial-tree multiplier.
Pipelined Array Multipliers

With latches after every FA level, the maximum throughput is achieved.

Latches may be inserted after every \( h \) FA levels for an intermediate design.

Example: 3-stage pipeline

Fig. 11.18 Pipelined 5\( \times \)5 array multiplier using latched FA blocks. The small shaded boxes are latches.
12 Variations in Multipliers

Chapter Goals

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

Chapter Highlights

Building a multiplier from smaller units
Performing multiply-add as one operation
Bit-serial and (semi)systolic multipliers
Using a multiplier for squaring is wasteful
Variations in Multipliers: Topics

<table>
<thead>
<tr>
<th>Topics in This Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.1 Divide-and-Conquer Designs</td>
</tr>
<tr>
<td>12.2 Additive Multiply Modules</td>
</tr>
<tr>
<td>12.3 Bit-Serial Multipliers</td>
</tr>
<tr>
<td>12.4 Modular Multipliers</td>
</tr>
<tr>
<td>12.5 The Special Case of Squaring</td>
</tr>
<tr>
<td>12.6 Combined Multiply-Add Units</td>
</tr>
</tbody>
</table>
12.1 Divide-and-Conquer Designs

Building wide multiplier from narrower ones

Fig. 12.1 Divide-and-conquer (recursive) strategy for synthesizing a $2b \times 2b$ multiplier from $b \times b$ multipliers.

Rearranged partial products in $2b$-by-$2b$ multiplication

$2b$ bits $b$ bits

$3b$ bits
General Structure of a Recursive Multiplier

- $2b \times 2b$ use $(3; 2)$-counters
- $3b \times 3b$ use $(5; 2)$-counters
- $4b \times 4b$ use $(7; 2)$-counters

Fig. 12.2 Using $b \times b$ multipliers to synthesize $2b \times 2b$, $3b \times 3b$, and $4b \times 4b$ multipliers.
Using $b \times c$, rather than $b \times b$ Building Blocks

- $2b \times 2c$: use $b \times c$ multipliers and (3; 2)-counters
- $2b \times 4c$: use $b \times c$ multipliers and (5; 2)-counters
- $gb \times hc$: use $b \times c$ multipliers and (?; 2)-counters
Wide Multiplier Built of Narrow Multipliers and Adders

Fig. 12.3 Using $4 \times 4$ multipliers and 4-bit adders to synthesize an $8 \times 8$ multiplier.
Karatsuba Multiplication

$2b \times 2b$ multiplication requires four $b \times b$ multiplications:

$$(2^b a_H + a_L) \times (2^b x_H + x_L) = 2^{2b} a_H x_H + 2^b (a_H x_L + a_L x_H) + a_L x_L$$

Karatsuba noted that one of the four multiplications can be removed at the expense of introducing a few additions:

$$(2^b a_H + a_L) \times (2^b x_H + x_L) =$$

$$2^{2b} a_H x_H + 2^b [(a_H + a_L) \times (x_H + x_L) - a_H x_H - a_L x_L] + a_L x_L$$

Benefit is quite significant for extremely wide operands

$$(4/3)^5 = 4.2 \quad (4/3)^{10} = 17.8 \quad (4/3)^{20} = 315.3 \quad (4/3)^{50} = 1,765,781$$
Computational Complexity of Multiplication

Arnold Schonhage and Volker Strassen (via FFT); best until 2007

- $O(\log k)$ time
- $O(k \log k \log \log k)$ complexity

In 2007, Martin Furer managed to replace the $\log \log k$ term with an asymptotically smaller term (for astronomically large numbers)

It is an open problem whether there exist logarithmic-delay multipliers with linear cost
(it is widely believed that there are not)

In the absence of a linear cost multiplication circuit, multiplication must be viewed as a more difficult problem than addition

In 2019, David Harvey and Joris van der Hoeven developed an $O(k \log k)$ multiplication algorithm, which is believed to be the best possible theoretically (but not practical at present)
12.2 Additive Multiply Modules

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

$b \times c$ AMM

\[
(2^b - 1) \times (2^c - 1) + (2^b - 1) + (2^c - 1) = 2^{b+c} - 1
\]
Fig. 12.5 An 8 × 8 multiplier built of 4 × 2 AMMs. Inputs marked with an asterisk carry 0s.
Multiplier Built of AMMs: Alternate Design

This design is more regular than that in Fig. 12.5 and is easily expandable to larger configurations; its latency, however, is greater.

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

Bit-serial adder (LSB first)

Bit-serial multiplier
(Must follow the \(k\)-bit inputs with \(k\) 0s; alternatively, view the product as being only \(k\) bits wide)

What goes inside the box to make a bit-serial multiplier? Can the circuit be designed to support a high clock rate?
Fig. 12.7  Semi-systolic circuit for $4 \times 4$ multiplication in 8 clock cycles.

This is called “semisystolic” because it has a large signal fan-out of $k$ ($k$-way broadcasting) and a long wire spanning all $k$ positions.
Systolic Retiming as a Design Tool

A semisystolic circuit can be converted to a systolic circuit via retiming, which involves advancing and retarding signals by means of delay removal and delay insertion in such a way that the relative timings of various parts are unaffected.

**Fig. 12.8** Example of retiming by delaying the inputs to $C_L$ and advancing the outputs from $C_L$ by $d$ units.
Alternate Explanation of Systolic Retiming

Transferring delay from the outputs of a subsystem to its inputs does not change the behavior of the overall system.
A First Attempt at Retiming

Fig. 12.7

Fig. 12.9  A retimed version of our semi-systolic multiplier.
Deriving a Fully Systolic Multiplier

Fig. 12.7  Systolic circuit for $4 \times 4$ multiplication in 15 cycles.

Fig. 12.10  Systolic circuit for $4 \times 4$ multiplication in 15 cycles.
A Direct Design for a Bit-Serial Multiplier

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

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

Fig. 12.13 Bit-serial multiplier design in dot notation.
12.4 Modular Multipliers

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

Fig. 12.15 Design of a 4 × 4 modulo-15 multiplier.
Other Examples of Modular Multiplication

Fig. 12.16 One way to design of a $4 \times 4$ modulo-13 multiplier.

Fig. 12.17 A method for modular multioperand addition.
12.5 The Special Case of Squaring

Multiply $x$ by $x$

<table>
<thead>
<tr>
<th></th>
<th>$x_4$</th>
<th>$x_3$</th>
<th>$x_2$</th>
<th>$x_1$</th>
<th>$x_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_4$</td>
<td>$x_0$</td>
<td>$x_1$</td>
<td>$x_2$</td>
<td>$x_3$</td>
<td>$x_4$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>$p_9$</th>
<th>$p_8$</th>
<th>$p_7$</th>
<th>$p_6$</th>
<th>$p_5$</th>
<th>$p_4$</th>
<th>$p_3$</th>
<th>$p_2$</th>
<th>$p_1$</th>
<th>$p_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_4$</td>
<td>$x_0$</td>
<td>$x_1$</td>
<td>$x_2$</td>
<td>$x_3$</td>
<td>$x_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x_3$</td>
<td>$x_2$</td>
<td>$x_1$</td>
<td>$x_0$</td>
<td>$x_3$</td>
<td>$x_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x_2$</td>
<td>$x_1$</td>
<td>$x_0$</td>
<td>$x_2$</td>
<td>$x_3$</td>
<td>$x_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x_1$</td>
<td>$x_0$</td>
<td>$x_1$</td>
<td>$x_2$</td>
<td>$x_3$</td>
<td>$x_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x_0$</td>
<td>$x_1$</td>
<td>$x_2$</td>
<td>$x_3$</td>
<td>$x_4$</td>
<td>$x_0$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Simplify

Fig. 12.18 Design of a 5-bit squarer.
Divide-and-Conquer Squarers

Building wide squarers from narrower ones

Divide-and-conquer (recursive) strategy for synthesizing a $2b \times 2b$ squarer from $b \times b$ squarers and multiplier.
12.6 Combined Multiply-Add Units

(a) Additive input → CSA tree output

(b) CSA tree output

(c) Additive input → Dot matrix for the $4 \times 4$ multiplication

(d) Carry-save additive input → Dot matrix for the $4 \times 4$ multiplication

Multiply-add versus multiply-accumulate

Multiply-accumulate units often have wider additive inputs

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