

Part III Multiplication

		Parts	Chapters
Elementary Operations		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 Reperesentations 18. Floating-Point Operations 19. Errors and Error Control 20. Precise and Certifiable Arithmetic
		VI. Function Evaluation	21. Square-Rooting Methods 22. The CORDIC Algorithms 23. Variations in Function Evaluation 24. Arithmetic by Table Lookup
		VII. Implementation Topics	25. High-Throughput Arithmetic 26. Low-Power Arithmetic 27. Fault-Tolerant Arithmetic 28. Reconfigurable Arithmetic

Appendix: Past, Present, and Future

About This Presentation

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

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

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

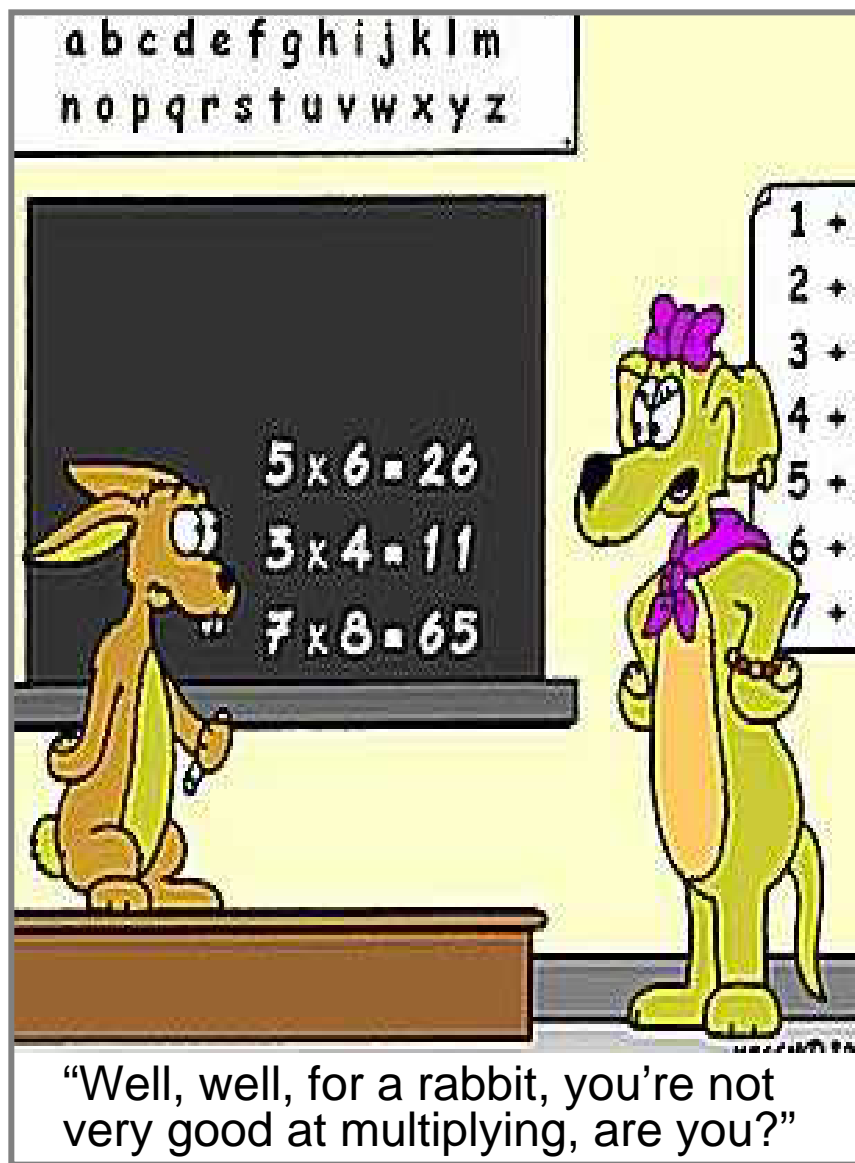
Topics in This Part

Chapter 9 Basic Multiplication Schemes

Chapter 10 High-Radix Multipliers

Chapter 11 Tree and Array Multipliers

Chapter 12 Variations in Multipliers



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

Topics in This Chapter

9.1 Shift/Add Multiplication Algorithms

9.2 Programmed Multiplication

9.3 Basic Hardware Multipliers

9.4 Multiplication of Signed Numbers

9.5 Multiplication by Constants

9.6 Preview of Fast Multipliers

9.1 Shift/Add Multiplication Algorithms

Notation for our discussion of multiplication algorithms:

a	Multiplicand	$a_{k-1}a_{k-2} \dots a_1a_0$
x	Multiplier	$x_{k-1}x_{k-2} \dots x_1x_0$
p	Product ($a \times x$)	$p_{2k-1}p_{2k-2} \dots p_3p_2p_1p_0$

Initially, we assume unsigned operands

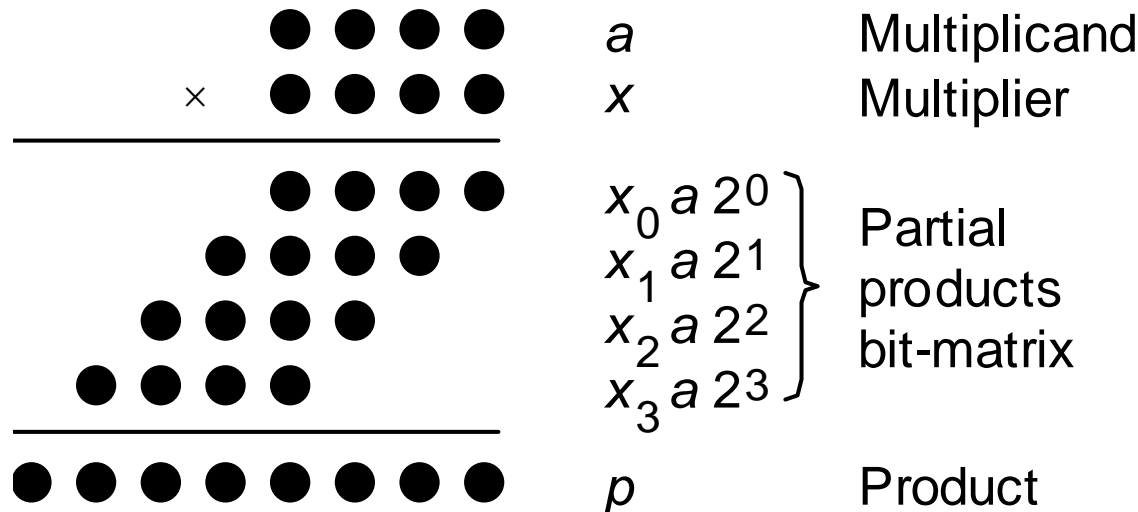
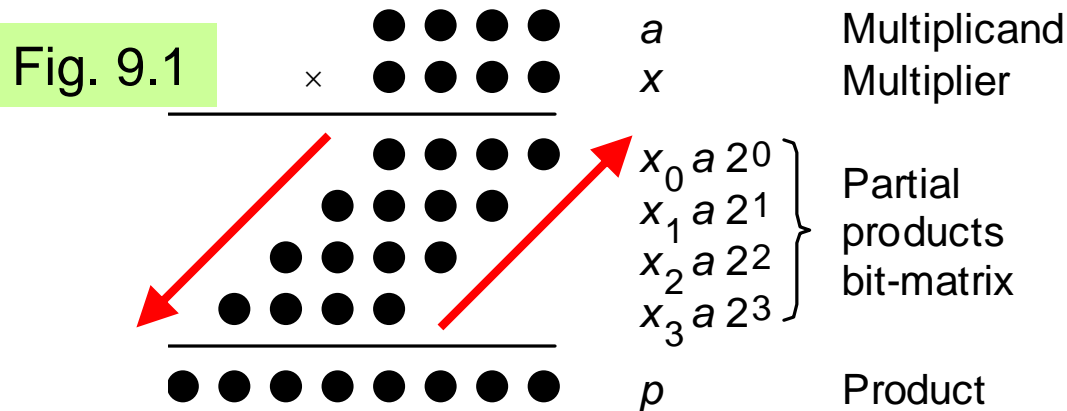


Fig. 9.1 Multiplication of two 4-bit unsigned binary numbers in dot notation.

Multiplication Recurrence



Preferred

Multiplication with right shifts: top-to-bottom accumulation

$$p^{(j+1)} = (p^{(j)} + x_j a 2^k) 2^{-1} \quad \text{with} \quad p^{(0)} = 0 \quad \text{and} \quad p^{(k)} = p = ax + p^{(0)} 2^{-k}$$

|——add——|
|——shift right——|

Multiplication with left shifts: bottom-to-top accumulation

$$p^{(j+1)} = 2 p^{(j)} + x_{k-j-1} a \quad \text{with} \quad p^{(0)} = 0 \quad \text{and} \quad p^{(k)} = p = ax + p^{(0)} 2^k$$

|shift|
|——add——|

Examples of Basic Multiplication

Right-shift algorithm

a	1 0 1 0	←	1 0 1 0
x			1 0 1 1
$p^{(0)}$	0 0 0 0		
$+x_0 a$	1 0 1 0		
$2p^{(1)}$	0 1 0 1 0		
$p^{(1)}$	0 1 0 1	0	
$+x_1 a$	1 0 1 0		
$2p^{(2)}$	0 1 1 1 1 0		
$p^{(2)}$	0 1 1 1	1 0	
$+x_2 a$	0 0 0 0		
$2p^{(3)}$	0 0 1 1 1 1 0		
$p^{(3)}$	0 0 1 1	1 1 0	
$+x_3 a$	1 0 1 0		
$2p^{(4)}$	0 1 1 0 1 1 1 0		
$p^{(4)}$	0 1 1 0	1 1 1 0	

Left-shift algorithm

a	1 0 1 0
x	1 0 1 1
$p^{(0)}$	0 0 0 0
$2p^{(0)}$	0 0 0 0
$+x_3 a$	1 0 1 0
$p^{(1)}$	0 1 0 1 0
$2p^{(1)}$	0 1 0 1 0 0
$+x_2 a$	0 0 0 0
$p^{(2)}$	0 1 0 1 0 0
$2p^{(2)}$	0 1 0 1 0 0 0
$+x_1 a$	0 0 0 0
$p^{(3)}$	0 1 0 1 0 0 0
$2p^{(3)}$	0 1 0 1 0 0 0 0
$+x_0 a$	1 0 1 0
$p^{(4)}$	0 1 1 0 1 1 1 0

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

|——add——|
|——shift right——|

Fig. 9.2
Examples
of
sequential
multipli-
cation with
right and
left shifts.

Check:
 10×11
 $= 110$
 $= 64 + 32 +$
 $8 + 4 + 2$

Examples of Basic Multiplication (Continued)

Right-shift algorithm

=====																
a									1	0	1	0				
x									1	0	1	1				
=====																
$p^{(0)}$									0	0	0	0				
$+x_0a$									1	0	1	0				
=====																
$2p^{(1)}$									0	1	0	1	0			
$p^{(1)}$									0	1	0	1	0			
$+x_1a$									1	0	1	0	0			
=====																
$2p^{(2)}$									0	1	1	1	1	0		
$p^{(2)}$									0	1	1	1	1	0		
$+x_2a$									1	0	1	0	0	0		
=====																
$2p^{(3)}$									0	1	1	1	1	0		
$p^{(3)}$									0	1	1	1	1	0		
$+x_3a$									1	0	1	0	0	0		
=====																
$2p^{(4)}$									0	1	1	0	1	1	1	0
$p^{(4)}$									0	1	1	0	1	1	1	0
=====																

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

|shift|

|-----add-----|

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

|shift|
|-----add-----|

Left-shift algorithm

=====											
a								1	0	1	0
x								1	0	1	1
=====											
$p^{(0)}$								0	0	0	0
$2p^{(0)}$							0	0	0	0	0
$+x_3a$								1	0	1	0
=====											
$p^{(1)}$							0	1	0	1	0
$2p^{(1)}$						0	1	0	1	0	0
$+x_2a$								0	0	0	0
=====											
$p^{(2)}$							0	1	0	1	0
$2p^{(2)}$						0	1	0	1	0	0
$+x_1a$								1	0	1	0
=====											
$p^{(3)}$							0	1	1	0	0
$2p^{(3)}$						0	1	1	0	0	0
$+x_0a$								1	0	1	0
=====											
$p^{(4)}$							0	1	1	0	1
=====											

Fig. 9.2
Examples
of
sequential
multipli-
cation with
right and
left shifts.

Check:
 10×11
 $= 110$
 $= 64 + 32 +$
 $8 + 4 + 2$

9.2 Programmed Multiplication

{Using right shifts, multiply unsigned m_cand and m_ier, storing the resultant 2k-bit product in p_high and p_low.

Registers: R0 holds 0 Rc for counter

Ra for m_cand Rx for m_ier

Rp for p_high Rq for p_low

{Load operands into registers Ra and Rx}

R0	0	Rc	Counter
Ra	Multiplicand	Rx	Multiplier
Rp	Product, high	Rq	Product, low

mult: load Ra with m_cand

load Rx with m_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 {LSB moves to carry flag}

branch no_add if carry = 0

add Ra to Rp {carry flag is set to cout}

no_add: rotate Rp right 1 {carry to MSB, LSB to carry}

rotate Rq right 1 {carry to MSB, LSB to carry}

decr Rc {decrement counter by 1}

branch m_loop if Rc ≠ 0

{Store the product}

store Rp into p_high

store Rq into p_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

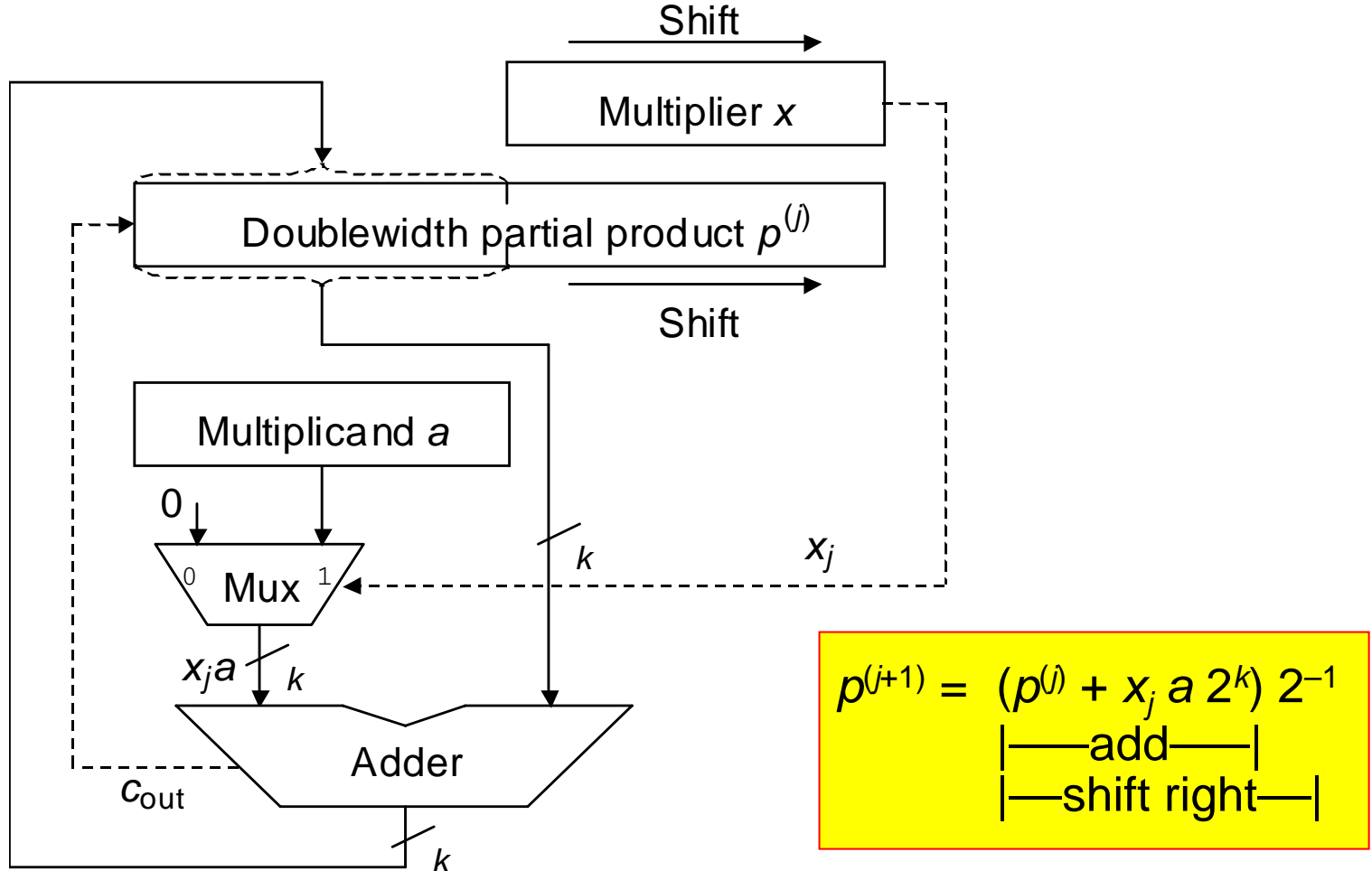


Fig. 9.4 Hardware realization of the sequential multiplication algorithm with additions and right shifts.

Example of Hardware Multiplication

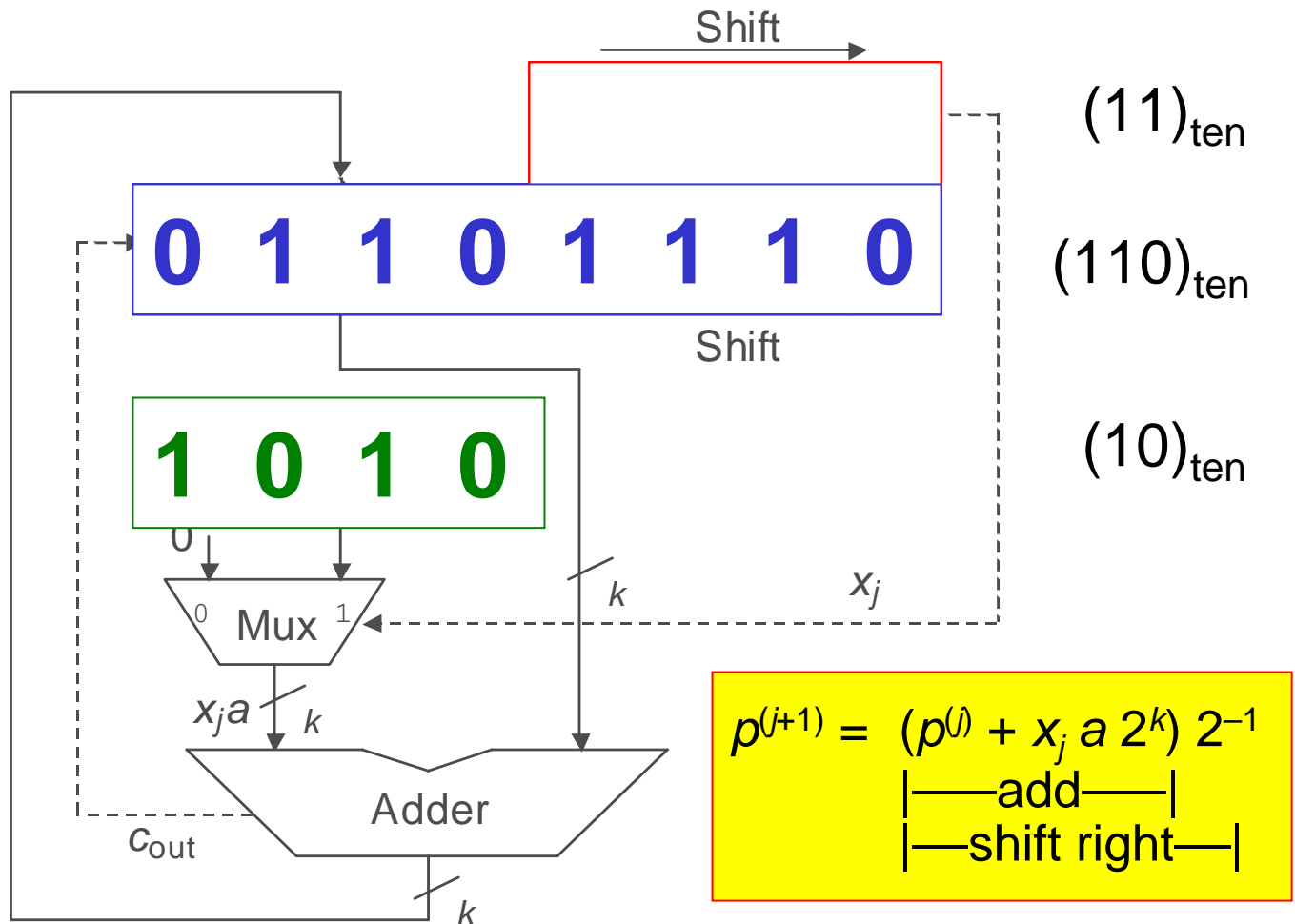


Fig. 9.4a Hardware realization of the sequential multiplication algorithm with additions and right shifts.

Performing Add and Shift in One Clock Cycle

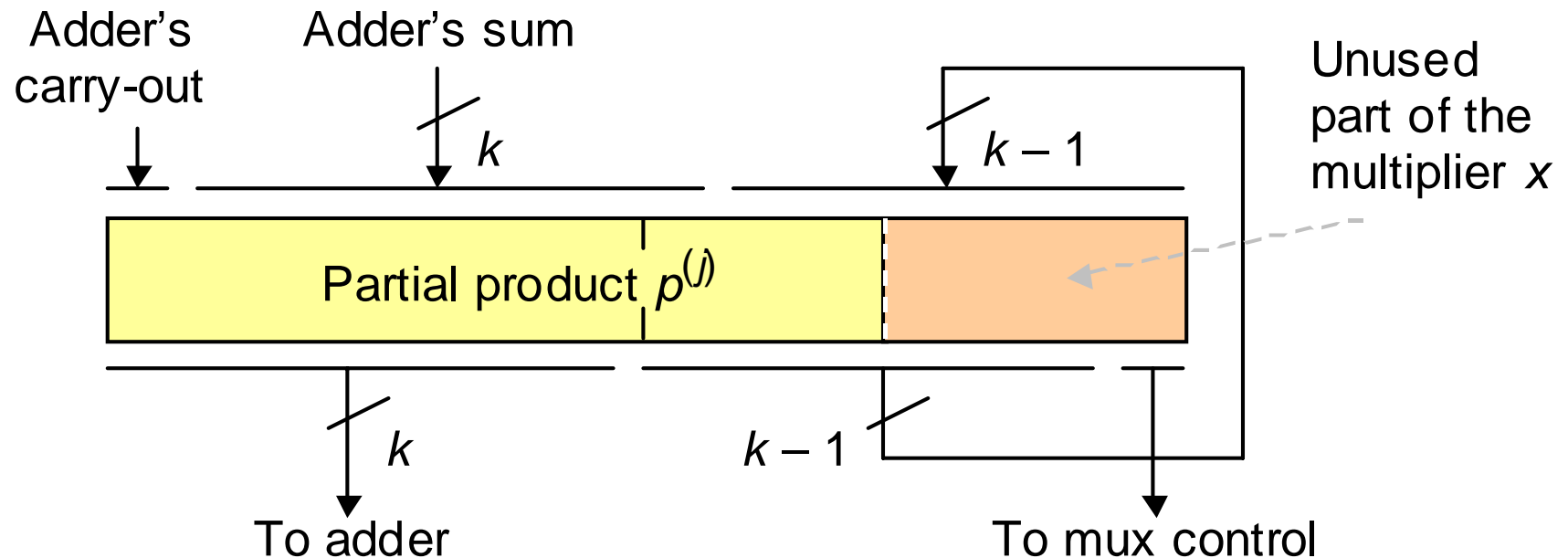


Fig. 9.5 Combining the loading and shifting of the double-width register holding the partial product and the partially used multiplier.

Sequential Multiplication with Left Shifts

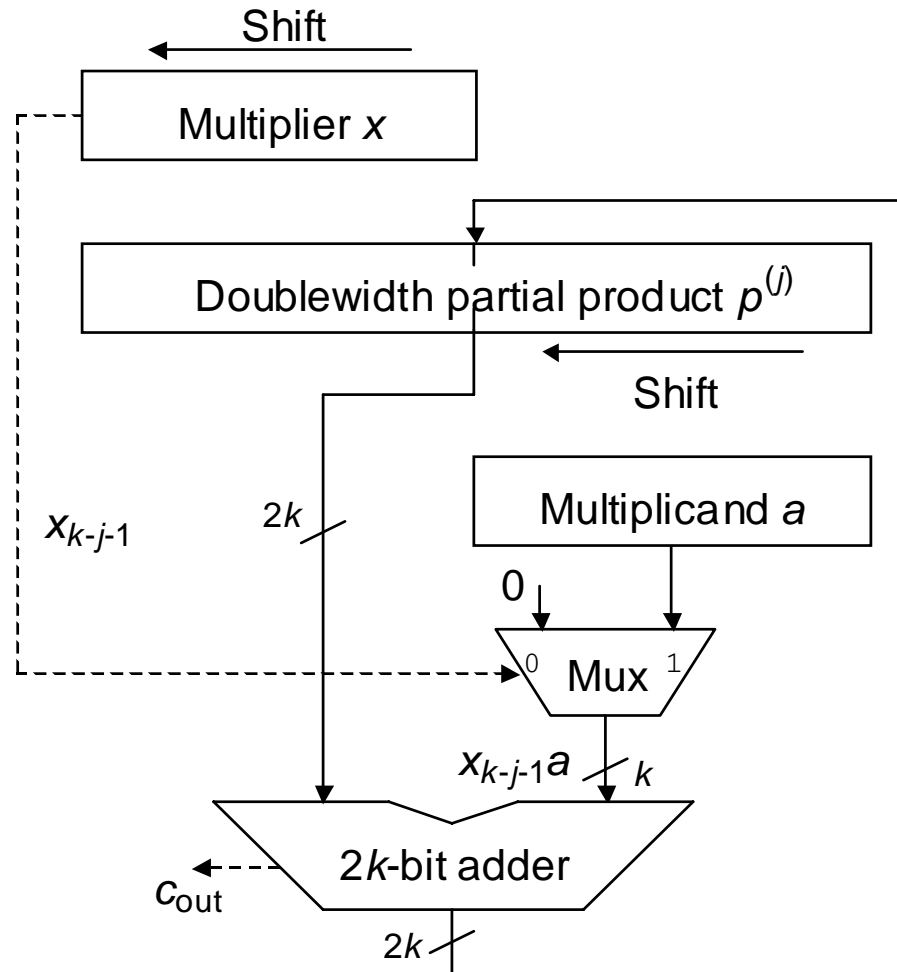


Fig. 9.4b Hardware realization of the sequential multiplication algorithm with left shifts and additions.

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

=====									
a		1	0	1	1	0			
x		0	1	0	1	1			
=====									
$p^{(0)}$		0	0	0	0	0			
$+x_0a$		1	0	1	1	0			
<hr/>									
$2p^{(1)}$	1	1	0	1	1	0			
$p^{(1)}$		1	1	0	1	1	0		
$+x_1a$		1	0	1	1	0			
<hr/>									
$2p^{(2)}$	1	1	0	0	0	1	0		
$p^{(2)}$		1	1	0	0	0	1	0	
$+x_2a$		0	0	0	0	0			
<hr/>									
$2p^{(3)}$	1	1	1	0	0	0	1	0	
$p^{(3)}$		1	1	1	0	0	0	1	0
$+x_3a$		1	0	1	1	0			
<hr/>									
$2p^{(4)}$	1	1	0	0	1	0	0	1	0
$p^{(4)}$		1	1	0	0	1	0	0	1
$+x_4a$		0	0	0	0	0			
<hr/>									
$2p^{(5)}$	1	1	1	0	0	1	0	0	1
$p^{(5)}$		1	1	1	0	0	1	0	0
=====									

Check:

-10 × 11

= -110

= -512 +

256 +

128 +

16 + 2

Check:
 -10×11
 $= -110$
 $= -512 +$
 $256 +$
 $128 +$
 $16 + 2$

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

a	1	0	1	1	0						
x	1	0	1	0	1						
$p^{(0)}$	0	0	0	0	0						
$+x_0a$	1	0	1	1	0						
$2p^{(1)}$	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×-11
 $= 110$
 $= 64 + 32 + 8 + 4 + 2$

Signed 2's-Complement Hardware Multiplier

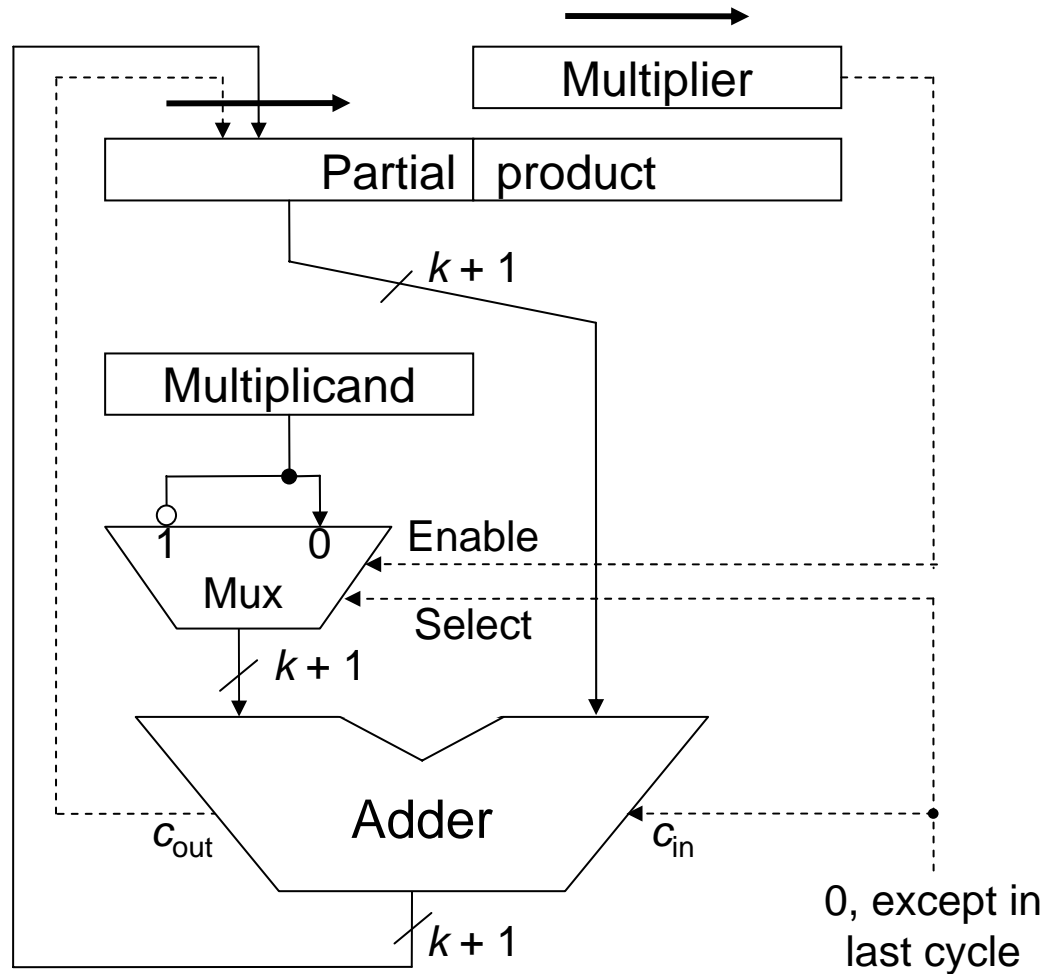


Fig. 9.8 The 2's-complement sequential hardware multiplier.

Booth's Recoding

Table 9.1 Radix-2 Booth's recoding

x_i	x_{i-1}	y_i	Explanation
0	0	0	No string of 1s in sight
0	1	1	End of string of 1s in x
1	0	-1	Beginning of string of 1s in x
1	1	0	Continuation of string of 1s in x

Example

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

Justification

$$2^j + 2^{j-1} + \dots + 2^{i+1} + 2^i = 2^{j+1} - 2^i$$

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.

x_i	x_{i-1}	y_i
0	0	0
0	1	1
1	0	-1
1	1	0

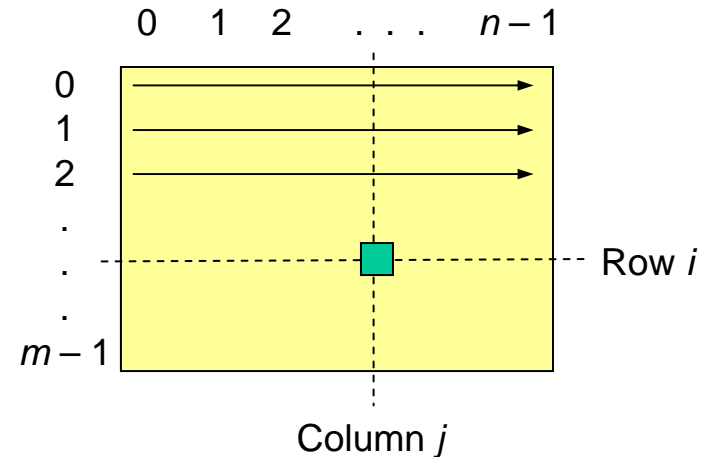
a	1	0	1	1	0		
x	1	0	1	0	1		Multiplier
y	-1	1	-1	1	-1		Booth-recoded
$p^{(0)}$	0	0	0	0	0		Check:
$+y_0a$	0	1	0	1	0		-10×-11
$2p^{(1)}$	0	0	1	0	1	0	$= 110$
$p^{(1)}$	0	0	1	0	1	0	$= 64 + 32 +$
$+y_1a$	1	0	1	1	0		$8 + 4 + 2$
$2p^{(2)}$	1	1	1	0	1	1	0
$p^{(2)}$	1	1	1	0	1	1	0
$+y_2a$	0	1	0	1	0		
$2p^{(3)}$	0	0	0	1	1	1	1
$p^{(3)}$	0	0	0	1	1	1	1
$+y_3a$	1	0	1	1	0		0
$2p^{(4)}$	1	1	1	0	0	1	1
$p^{(4)}$	1	1	1	0	0	1	1
y_4a	0	1	0	1	0		0
$2p^{(5)}$	0	0	0	1	1	0	1
$p^{(5)}$	0	0	0	1	1	0	1

9.5 Multiplication by Constants

Explicit, e.g. $y := 12 * x + 1$

Implicit, e.g. $A[i, j] := A[i, j] + B[i, j]$

Address of $A[i, j] = \text{base} + n * i + j$



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:

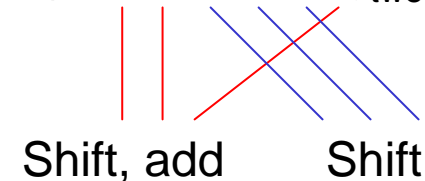
Synthesize special-purpose units such as filters

$$y[t] = a_0x[t] + a_1x[t-1] + a_2x[t-2] + b_1y[t-1] + b_2y[t-2]$$

Multiplication Using Binary Expansion

Example: Multiply R1 by the constant $113 = (1\ 1\ 1\ 0\ 0\ 0\ 1)_{\text{two}}$

R2	←	R1 shift-left 1
R3	←	R2 + R1
R6	←	R3 shift-left 1
R7	←	R6 + R1
R112	←	R7 shift-left 4
R113	←	R112 + R1



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

R3	←	R1 shift-left 1 + R1
R7	←	R3 shift-left 1 + R1
R113	←	R7 shift-left 4 + R1

Multiplication via Recoding

Example: Multiply R1 by $113 = (1\ 1\ 1\ 0\ 0\ 0\ 1)_{\text{two}} = (1\ 0\ 0\ -1\ 0\ 0\ 0\ 1)_{\text{two}}$

R8 \leftarrow R1 shift-left 3
 R7 \leftarrow R8 - R1
 R112 \leftarrow R7 shift-left 4
 R113 \leftarrow R112 + R1

Shift, add Shift
 Shift, subtract

Shorter sequence using shift-and-add/subtract instructions

R7 \leftarrow R3 shift-left 3 - R1
 R113 \leftarrow R7 shift-left 4 + R1

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

Multiplication via Factorization

Example: Multiply R1 by $119 = 7 \times 17 = (8 - 1) \times (16 + 1)$

R8 \leftarrow R1 shift-left 3
R7 \leftarrow R8 - R1
R112 \leftarrow R7 shift-left 4
R119 \leftarrow R112 + R7

Shorter sequence using shift-and-add/subtract instructions

R7 \leftarrow R3 shift-left 3 - R1
R119 \leftarrow R7 shift-left 4 + R7

Requires a scratch register
for holding the 7 multiple

$$119 = (1\ 1\ 1\ 0\ 1\ 1\ 1)_{\text{two}} = (1\ 0\ 0\ 0\ 1\ 0\ 0\ 1)_{\text{two}}$$

More instructions may be needed without factorization

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

10 High-Radix Multipliers

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

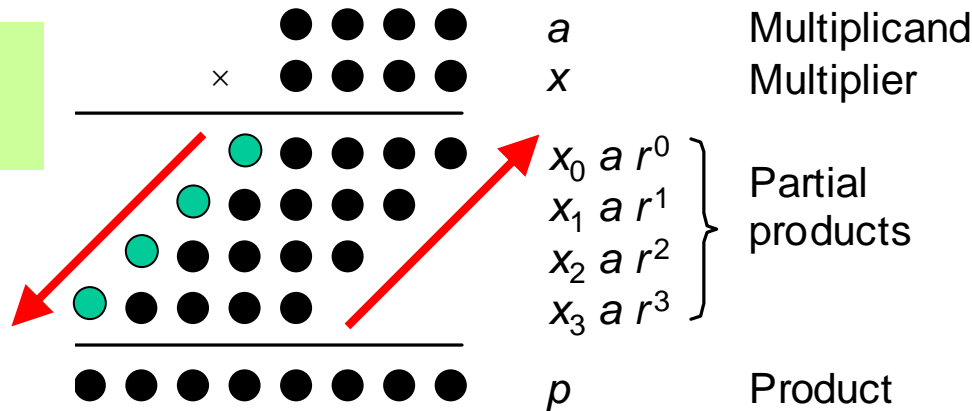
10.4 Radix-8 and Radix-16 Multipliers

10.5 Multibit Multipliers

10.6 VLSI Complexity Issues

10.1 Radix-4 Multiplication

Fig. 9.1
(modified)



Preferred

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

$$p^{(j+1)} = \underbrace{(p^{(j)} + x_j a r^k)}_{\text{—add—}} \underbrace{r^{-1}}_{\text{—shift right—}} \quad \text{with} \quad p^{(0)} = 0 \quad \text{and} \quad p^{(k)} = p = ax + p^{(0)} r^{-k}$$

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

$$p^{(j+1)} = \underbrace{r p^{(j)}}_{\text{—shift—}} + \underbrace{x_{k-j-1} a}_{\text{—add—}} \quad \text{with} \quad p^{(0)} = 0 \quad \text{and} \quad p^{(k)} = p = ax + p^{(0)} r^k$$

Radix-4 Multiplication in Dot Notation

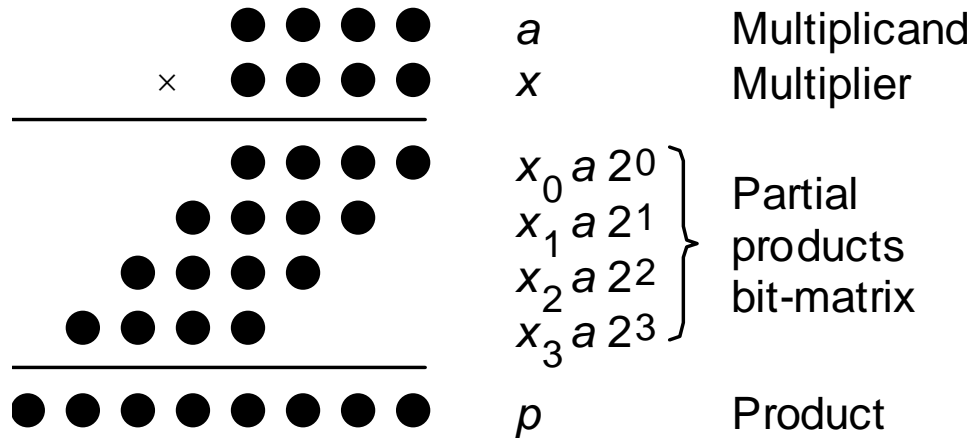
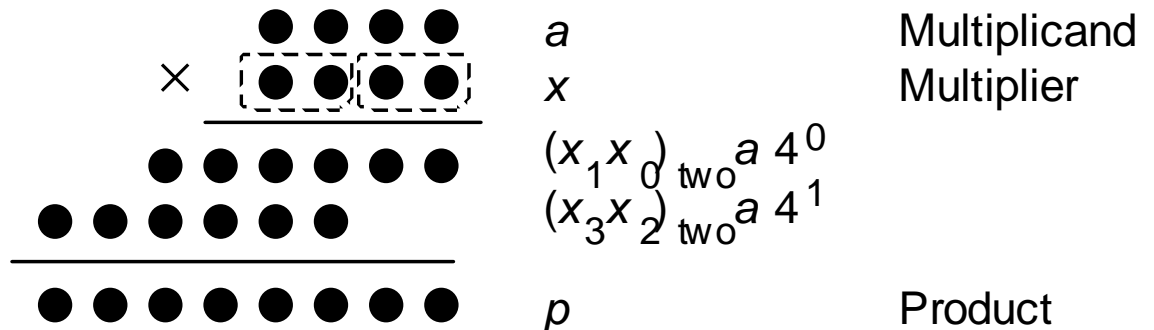


Fig. 9.1

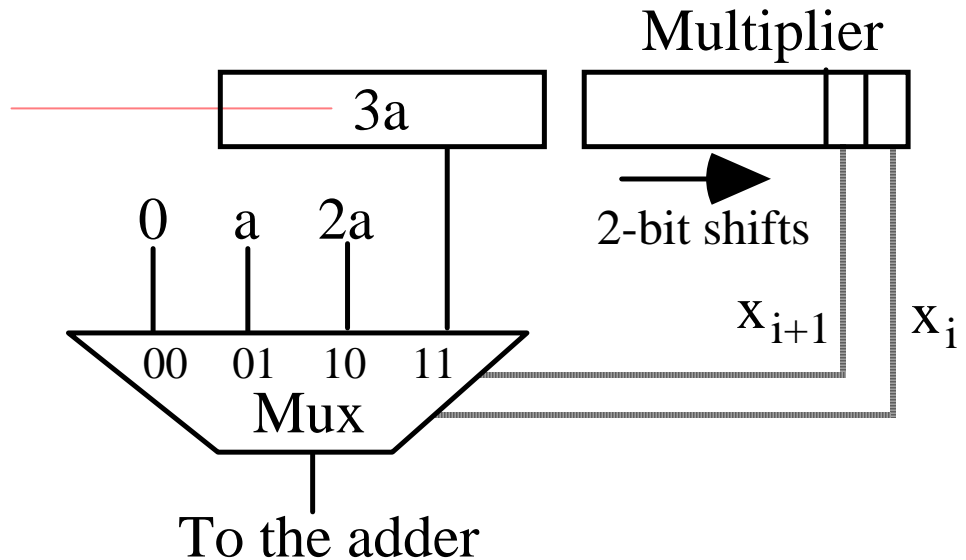
Fig. 10.1 Radix-4, or two-bit-at-a-time, multiplication in dot notation

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



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

=====									
a			0	1	1	0			
$3a$		0	1	0	0	1	0		
x			1	1	1	0			
=====									
$p^{(0)}$			0	0	0	0			
$+(x_1x_0)_{\text{two}}a$		0	0	1	1	0	0		
=====									
$4p^{(1)}$		0	0	1	1	0	0		
$p^{(1)}$			0	0	1	1		0	0
$+(x_3x_2)_{\text{two}}a$		0	1	0	0	1	0		
=====									
$4p^{(2)}$		0	1	0	1	0	1		0
$p^{(2)}$			0	1	0	1		0	1
=====									

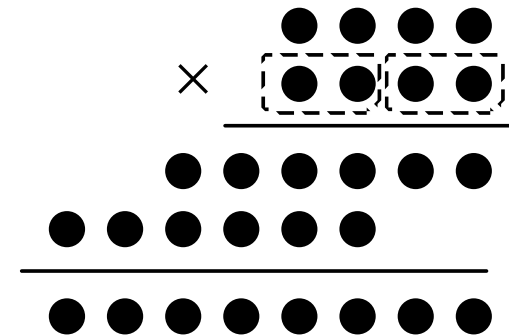
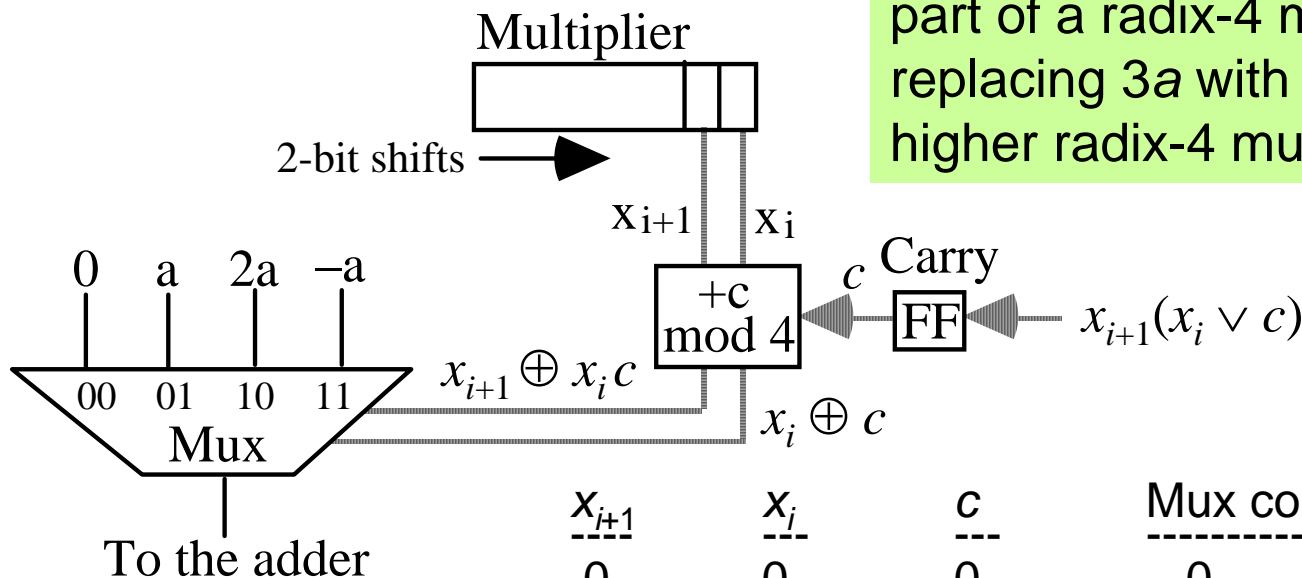


Fig. 10.3 Example of radix-4 multiplication using the 3a multiple.

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 $-a$.



x_{i+1}	x_i	c	Mux control		Set carry
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	1	0	0
1	0	1	1	1	1
1	1	0	1	1	1
1	1	1	0	0	1

10.2 Modified Booth's Recoding

Table 10.1 Radix-4 Booth's recoding yielding $(z_{k/2} \dots z_1 z_0)_{\text{four}}$

x_{i+1}	x_i	x_{i-1}	y_{i+1}	y_i	$z_{i/2}$	Explanation
0	0	0	0	0	0	No string of 1s in sight
0	0	1	0	1	1	End of string of 1s
0	1	0	0	1	1	Isolated 1
0	1	1	1	0	2	End of string of 1s
1	0	0	-1	0	-2	Beginning of string of 1s
1	0	1	-1	1	-1	End a string, begin new one
1	1	0	0	-1	-1	Beginning of string of 1s
1	1	1	0	0	0	Continuation of string of 1s

Context
Recoded radix-2 digits
Radix-4 digit

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
	<hr/>				<hr/>				<hr/>				<hr/>			
(1)	-2		2		-1		2		-1		-1		0		-2	

Operand x

Recoded version y

Radix-4 version z

Example Multiplication via Modified Booth's Recoding

=====									
a				0	1	1	0		
x				1	0	1	0		
z				-1		-2		Radix-4	
=====									
$p^{(0)}$				0	0	0	0	0	0
$+z_0 a$				1	1	0	1	0	0
<hr/>									
$4p^{(1)}$				1	1	0	1	0	0
$p^{(1)}$				1	1	1	1	0	1
								0	0
$+z_1 a$				1	1	1	0	1	0
<hr/>									
$4p^{(2)}$				1	1	0	1	1	1
								0	0
$p^{(2)}$				1	1	0	1	1	1
								0	0
=====									

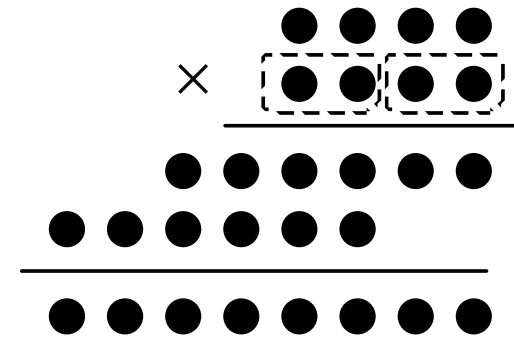


Fig. 10.5 Example of radix-4 multiplication with modified Booth's recoding of the 2's-complement multiplier.

Multiple Generation with Radix-4 Booth's Recoding

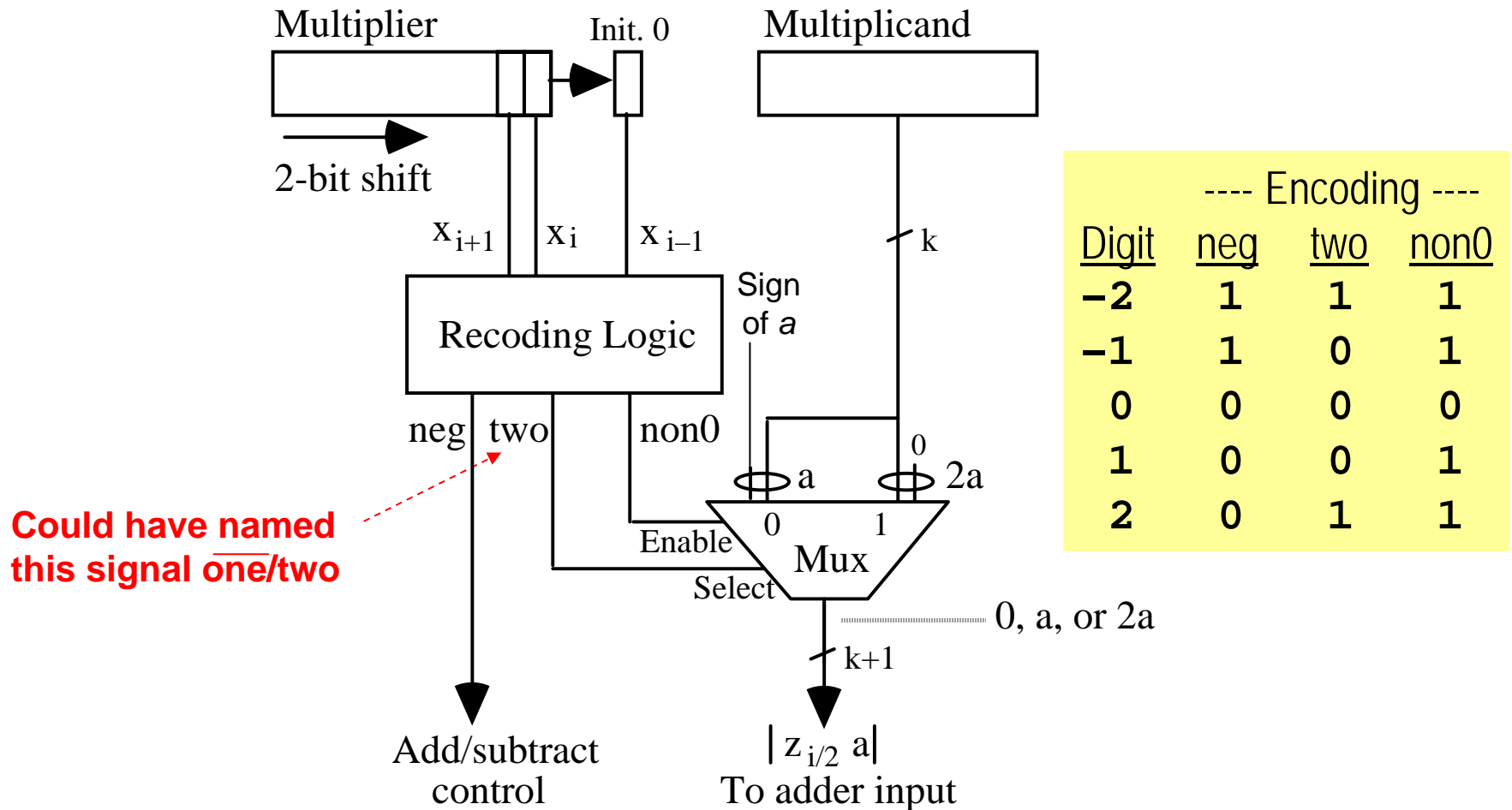


Fig. 10.6 The multiple generation part of a radix-4 multiplier based on Booth's recoding.

10.3 Using Carry-Save Adders

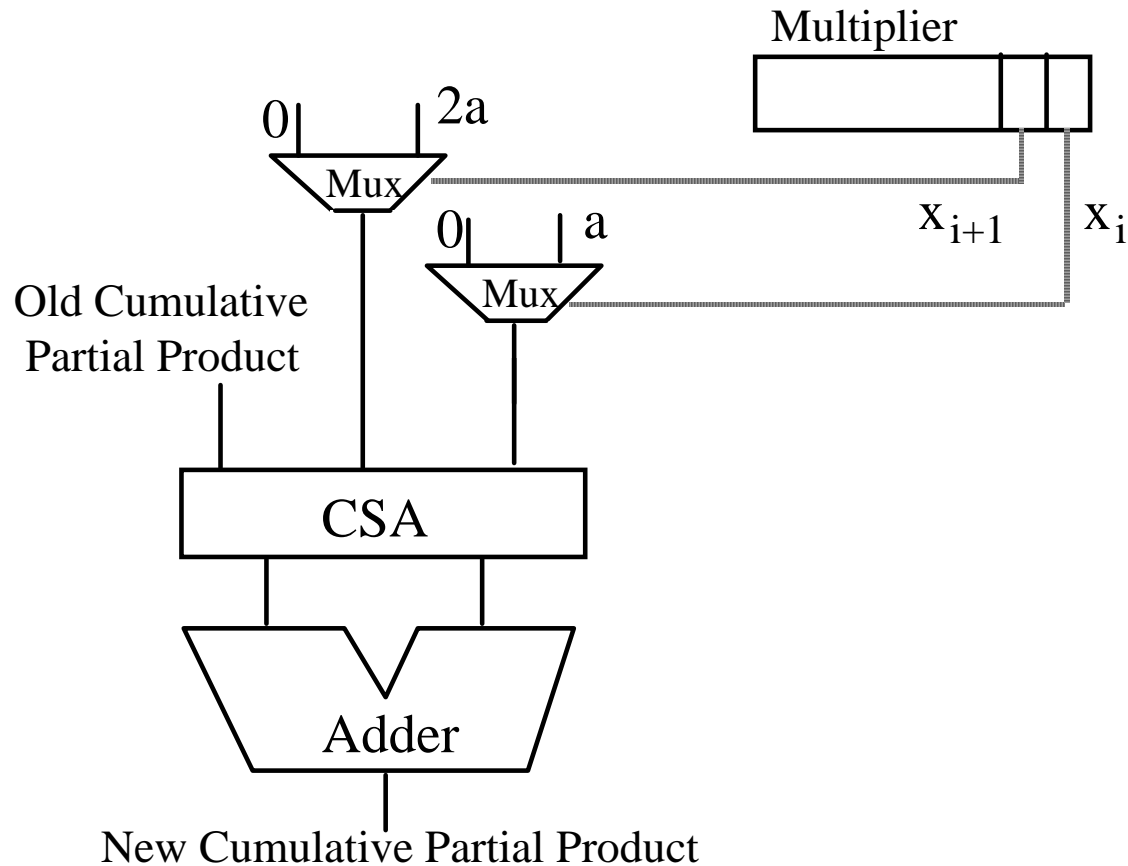
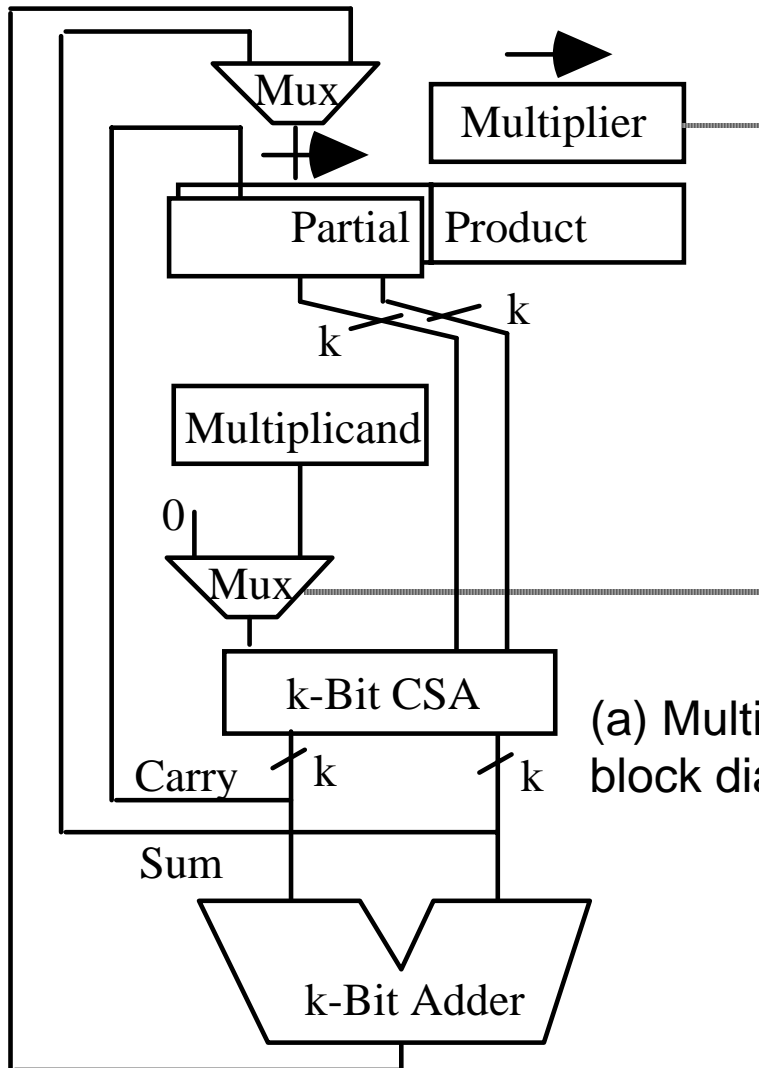
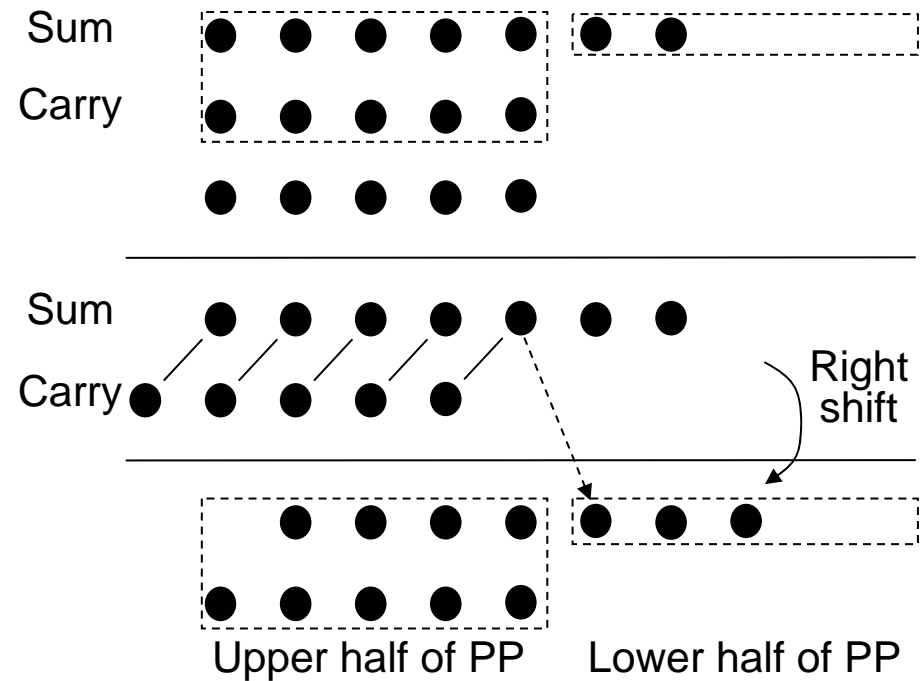


Fig. 10.7 Radix-4 multiplication with a carry-save adder used to combine the cumulative partial product, $x_i a$, and $2x_{i+1} a$ into two numbers.

Keeping the Partial Product in Carry-Save Form



(a) Multiplier block diagram



(b) Operation in a typical cycle

Fig. 10.8 Radix-2 multiplication with the upper half of the cumulative partial product kept in stored-carry form.

Carry-Save Multiplier with Radix-4 Booth's Recoding

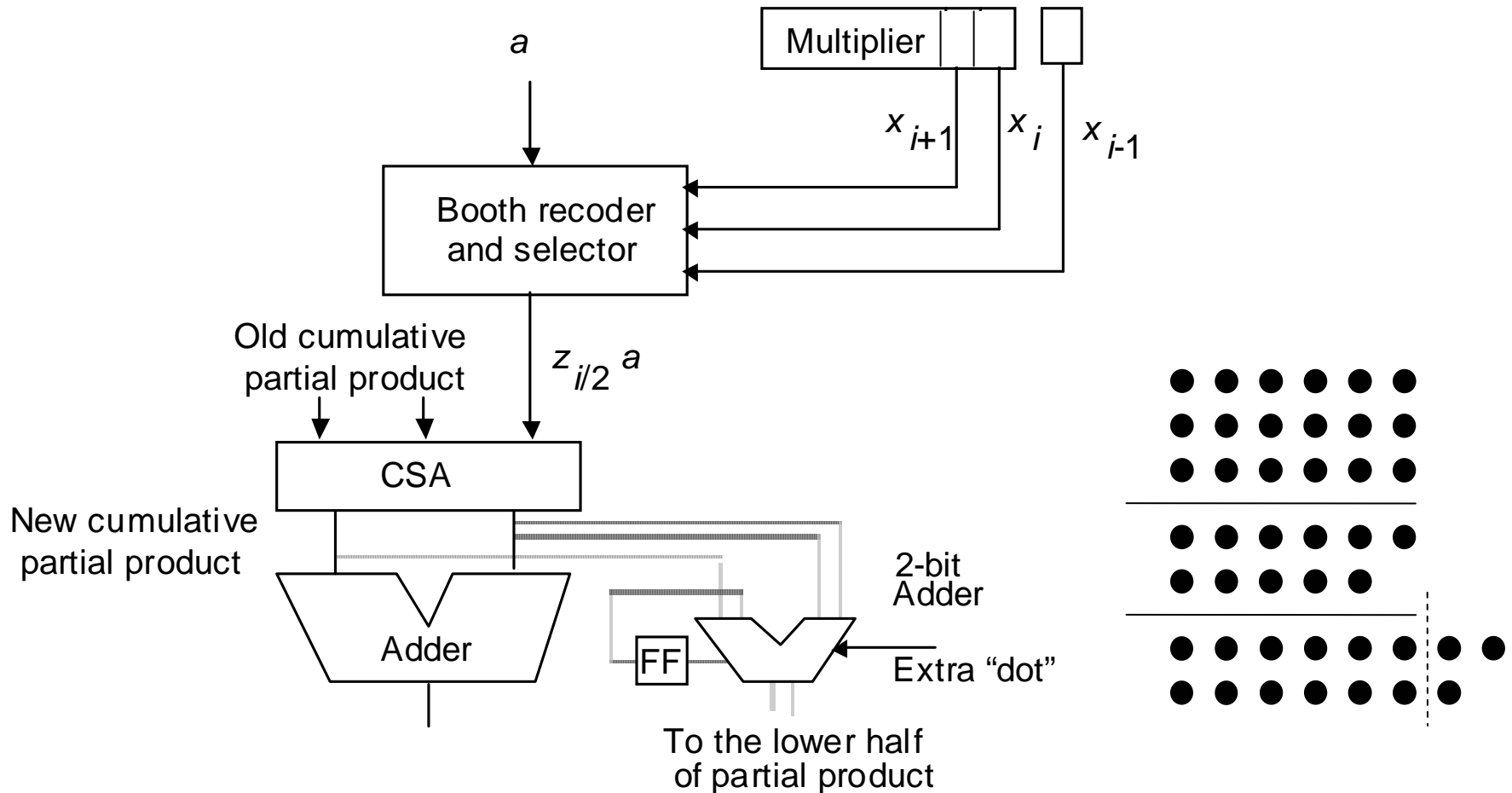


Fig. 10.9 Radix-4 multiplication with a CSA used to combine the stored-carry cumulative partial product and $z_{i/2}a$ into two numbers.

Radix-4 Booth's Recoding for Parallel Multiplication

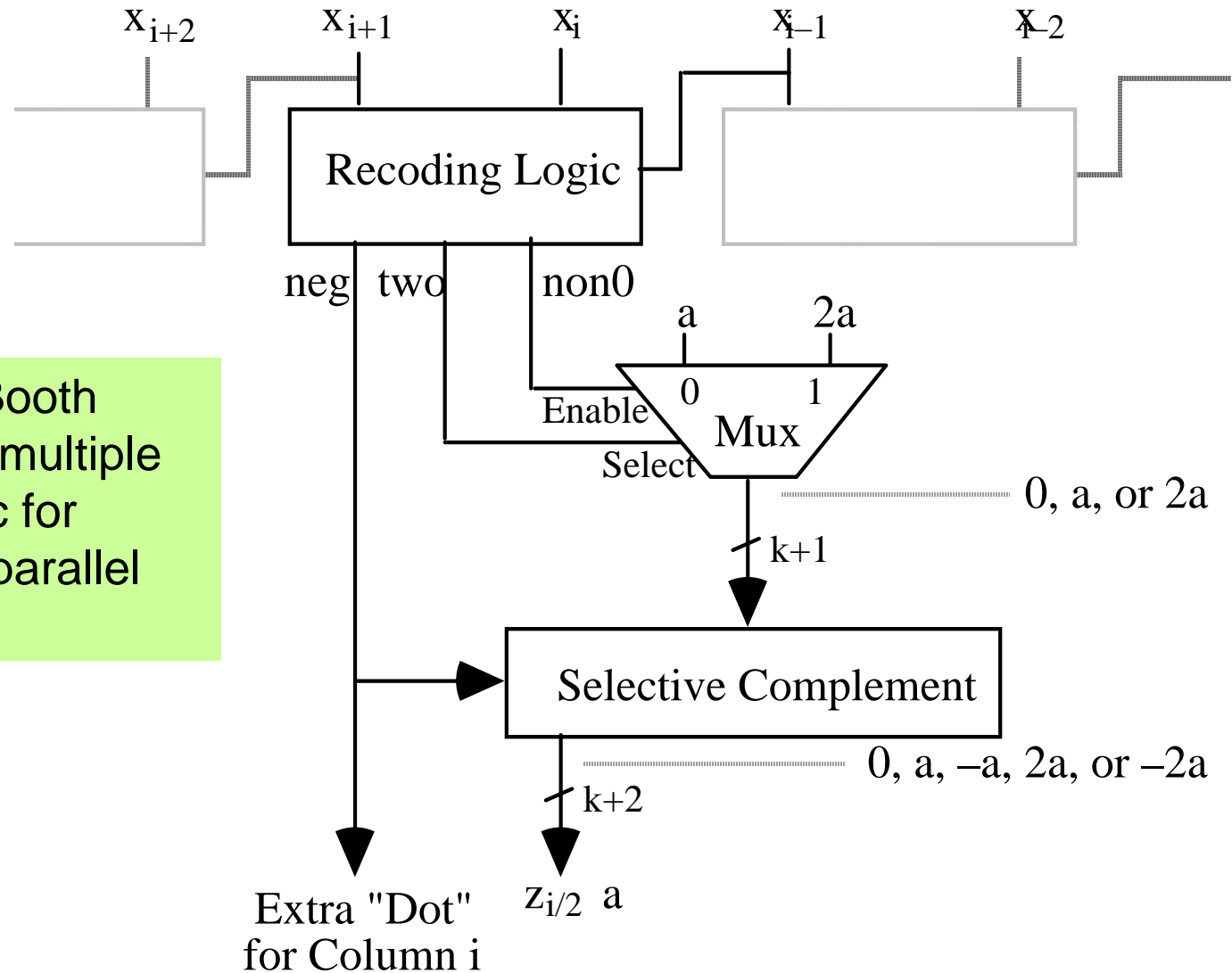


Fig. 10.10 Booth recoding and multiple selection logic for high-radix or parallel multiplication.

Yet Another Design for Radix-4 Multiplication

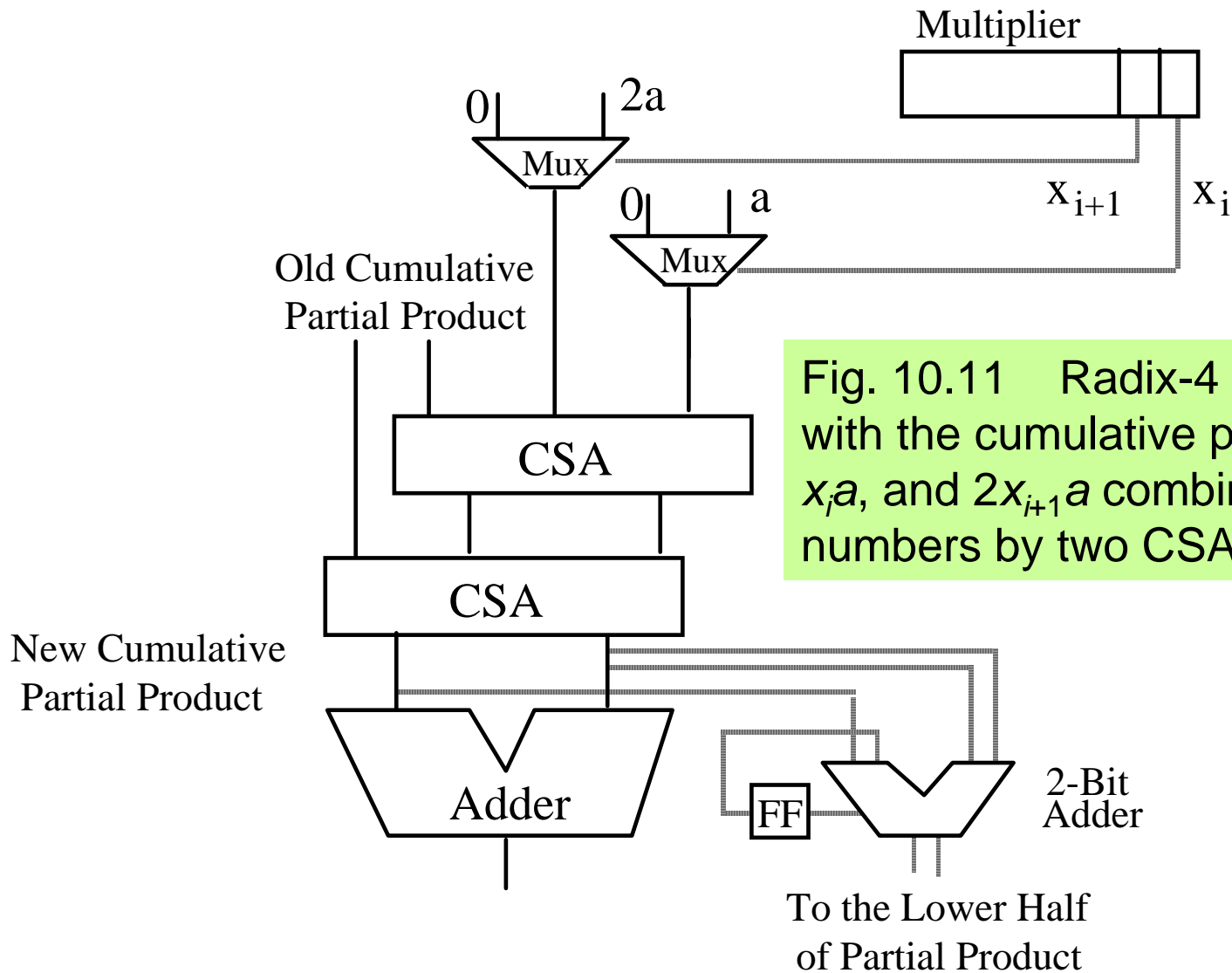


Fig. 10.11 Radix-4 multiplication, with the cumulative partial product, $x_i a$, and $2x_{i+1} a$ combined into two numbers by two CSAs.

10.4 Radix-8 and Radix-16 Multipliers

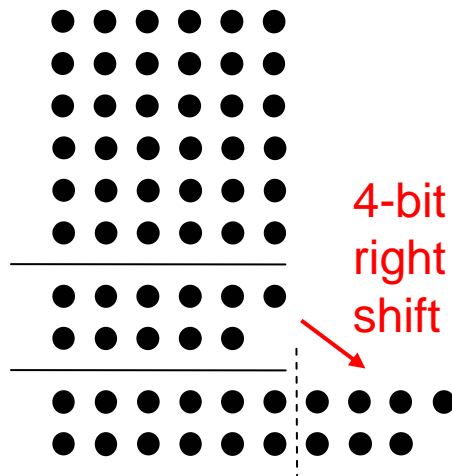
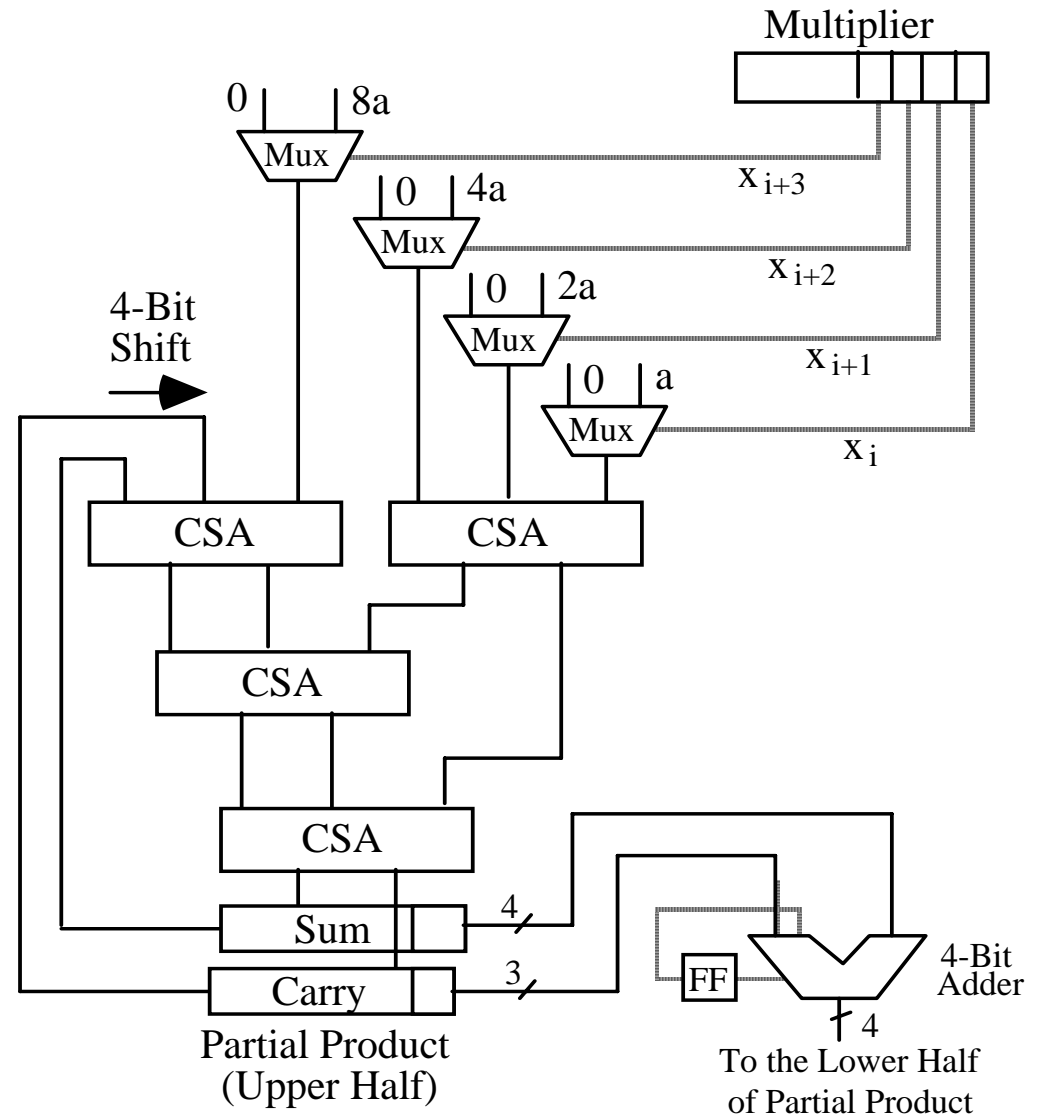


Fig. 10.12 Radix-16 multiplication with the upper half of the cumulative partial product in carry-save form.



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)

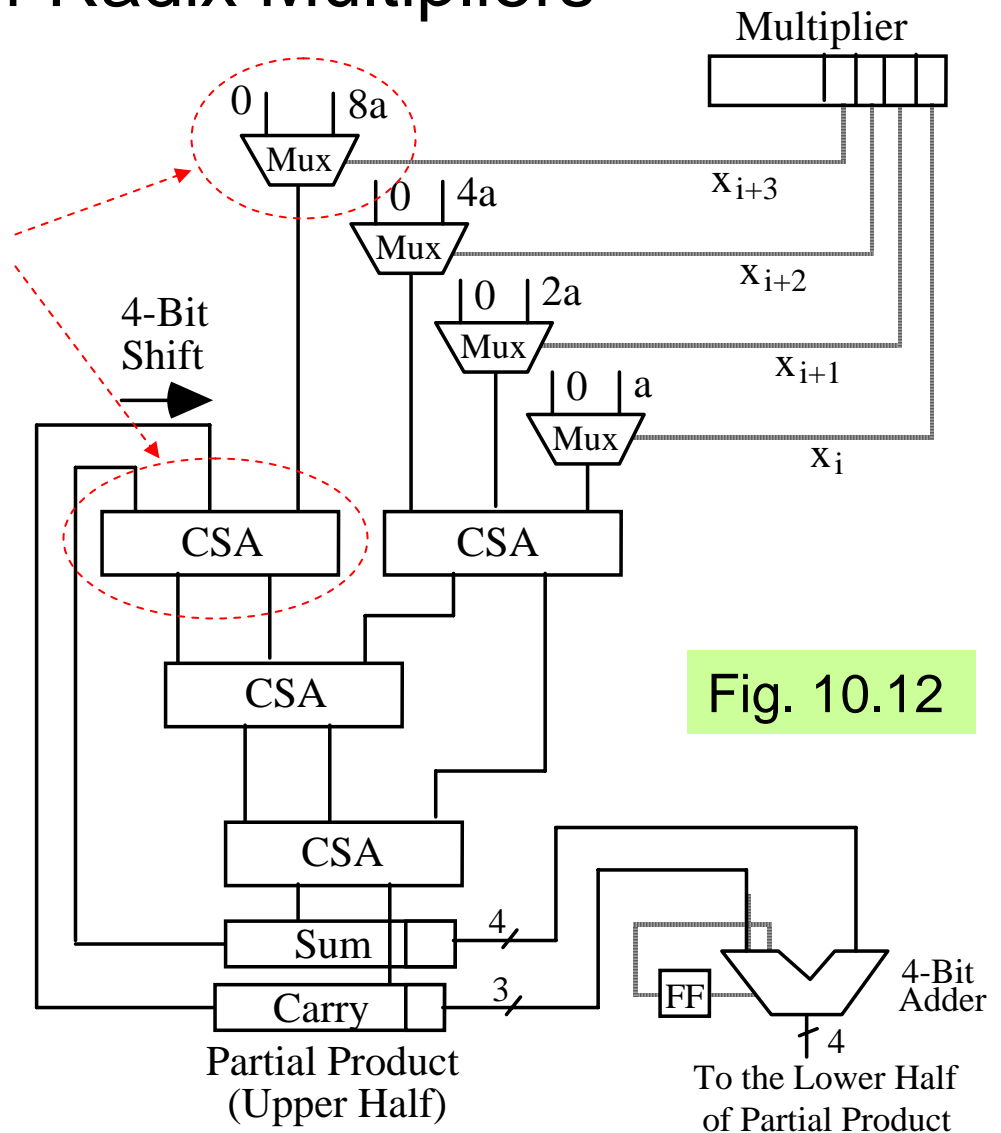


Fig. 10.12

A Spectrum of Multiplier Design Choices

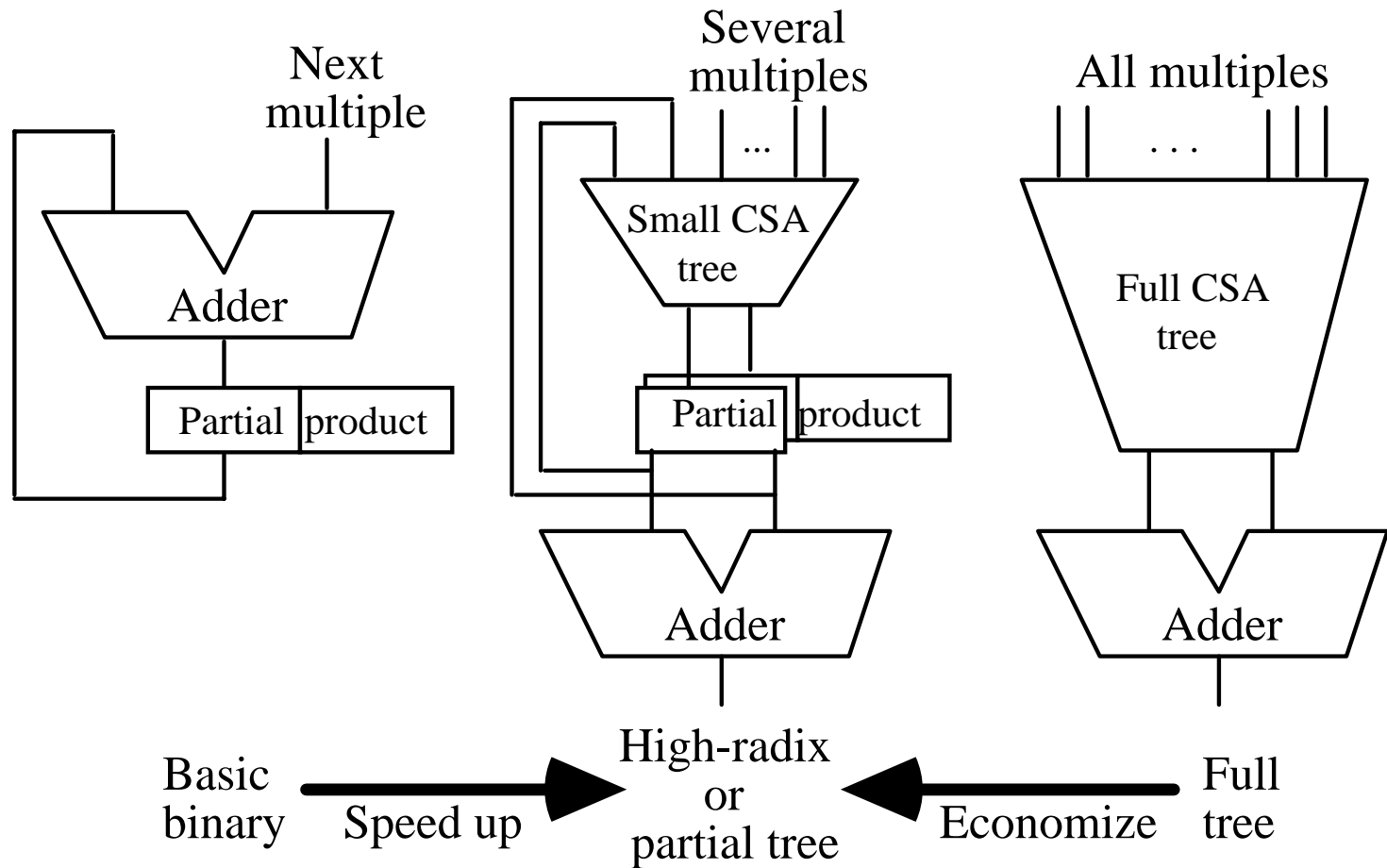
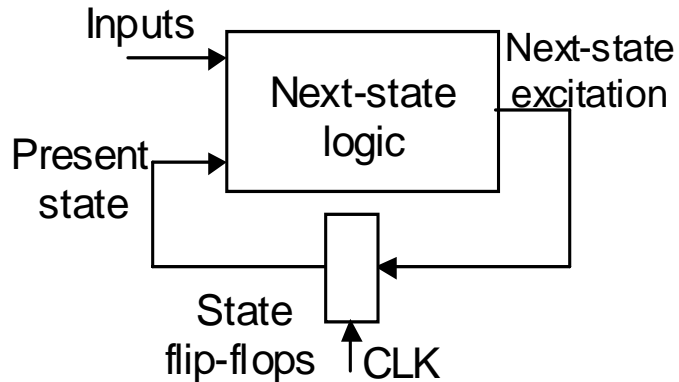
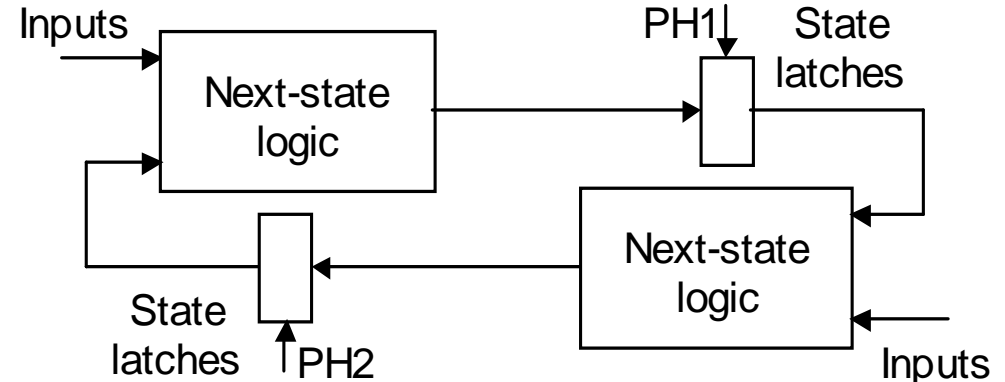


Fig. 10.13 High-radix multipliers as intermediate between sequential radix-2 and full-tree multipliers.

10.5 Multibit Multipliers

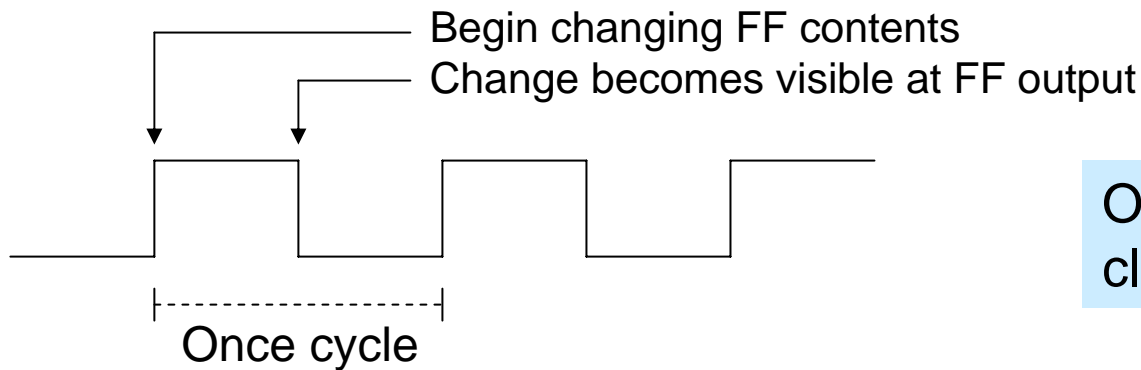


(a) Sequential machine with FFs



(b) Sequential machine with latches and 2-phase clock

Fig. 10.15 Two-phase clocking for sequential logic.



Observation: Half of the clock cycle goes to waste

Twin-Beat and Three-Beat Multipliers

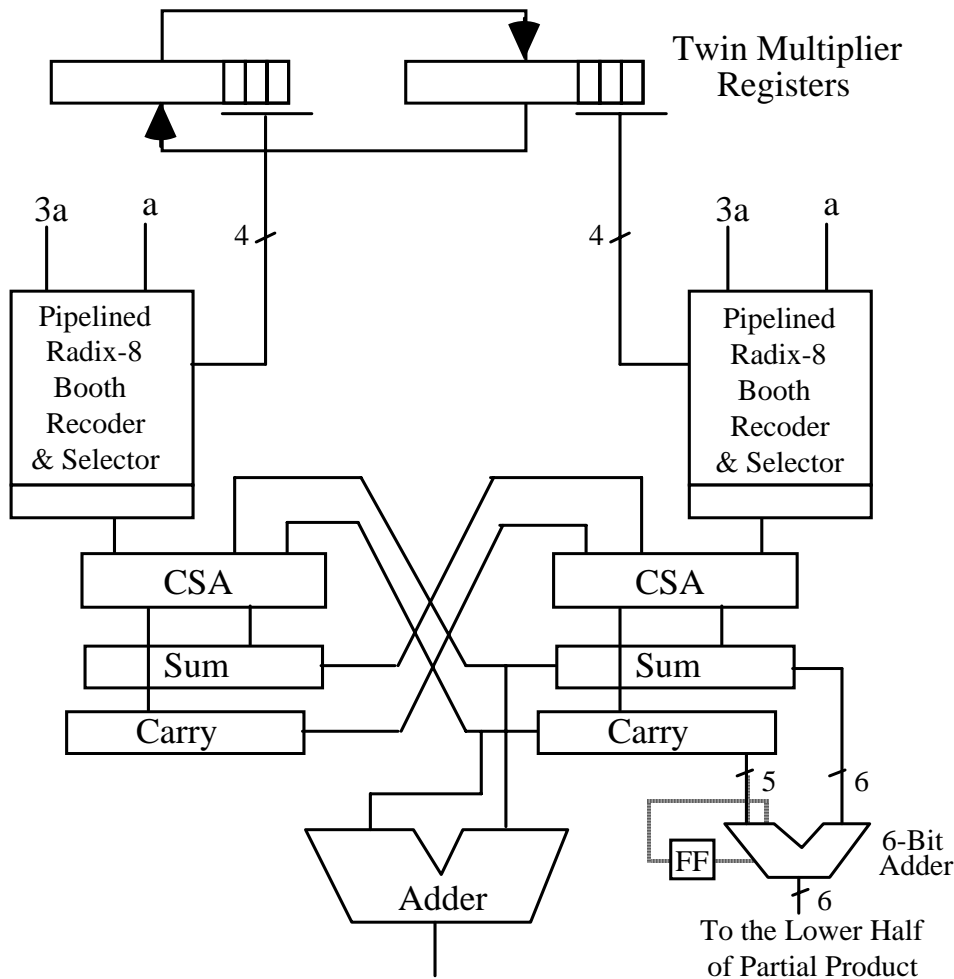


Fig. 10.14 Twin-beat multiplier with radix-8 Booth's recoding.

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

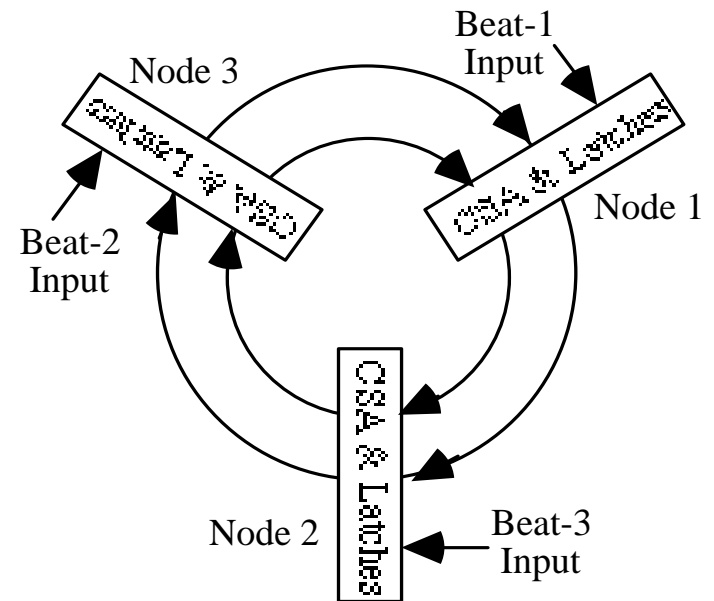


Fig. 10.16 Conceptual view of a three-beat multiplier.

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

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

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

	Low-Cost $b = O(1)$	High Speed $b = O(k)$	AT - or AT^2 - Optimal
AT	$O(k^2)$	$O(k^2 \log k)$	$O(k^{3/2})$
AT^2	$O(k^3)$	$O(k^2 \log^2 k)$	$O(k^2)$

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)

11 Tree and Array Multipliers

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

Topics in This Chapter

11.1. Full-Tree Multipliers

11.2. Alternative Reduction Trees

11.3. Tree Multipliers for Signed Numbers

11.4. Partial-Tree and Truncated Multipliers

11.5. Array Multipliers

11.6. Pipelined Tree and Array Multipliers

11.1 Full-Tree Multipliers

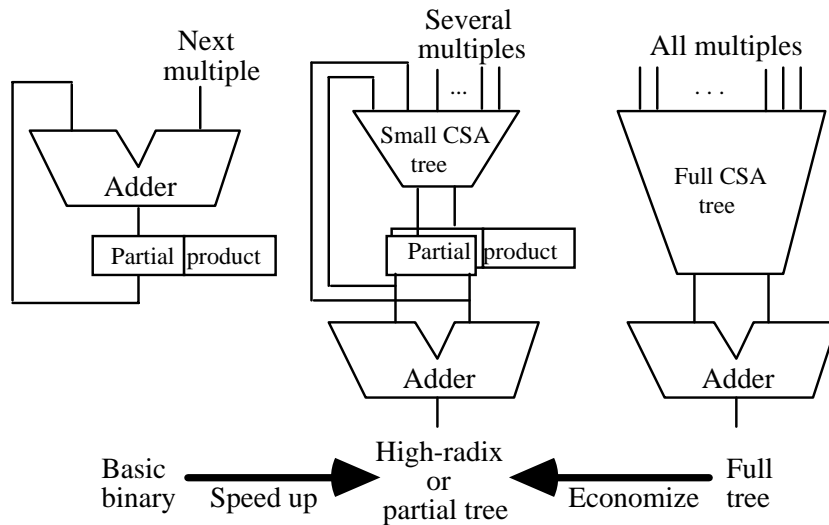


Fig. 10.13 High-radix multipliers as intermediate between sequential radix-2 and full-tree multipliers.

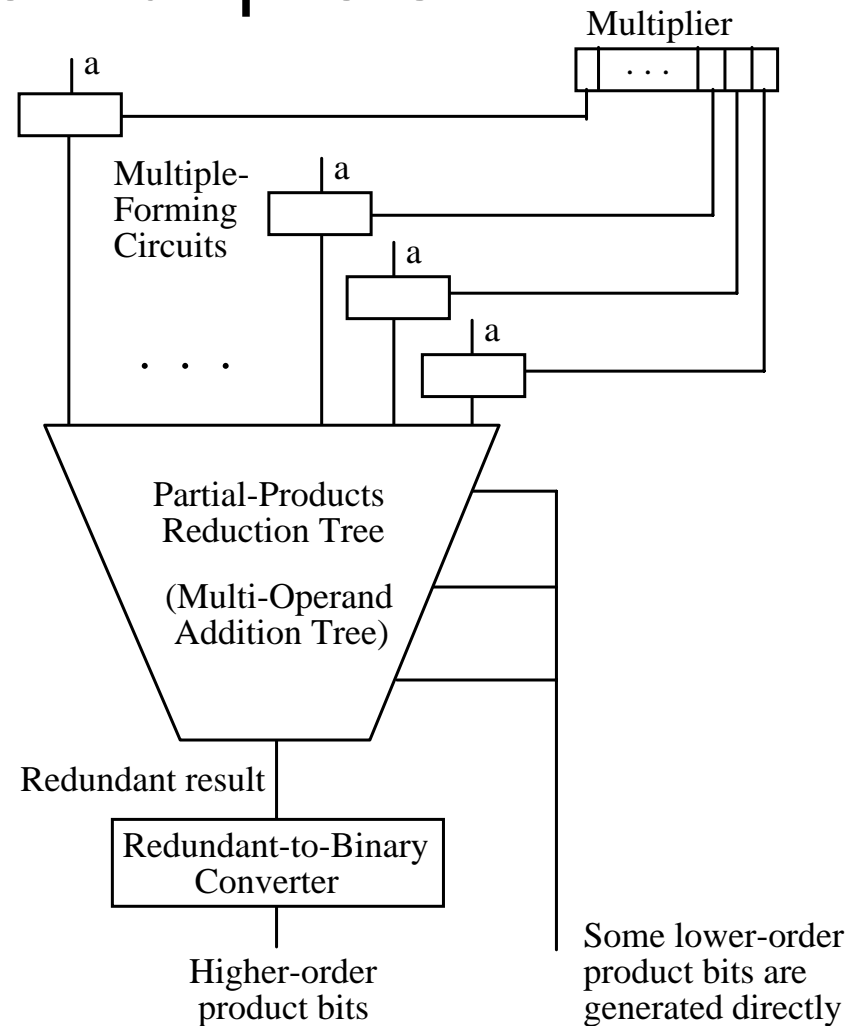
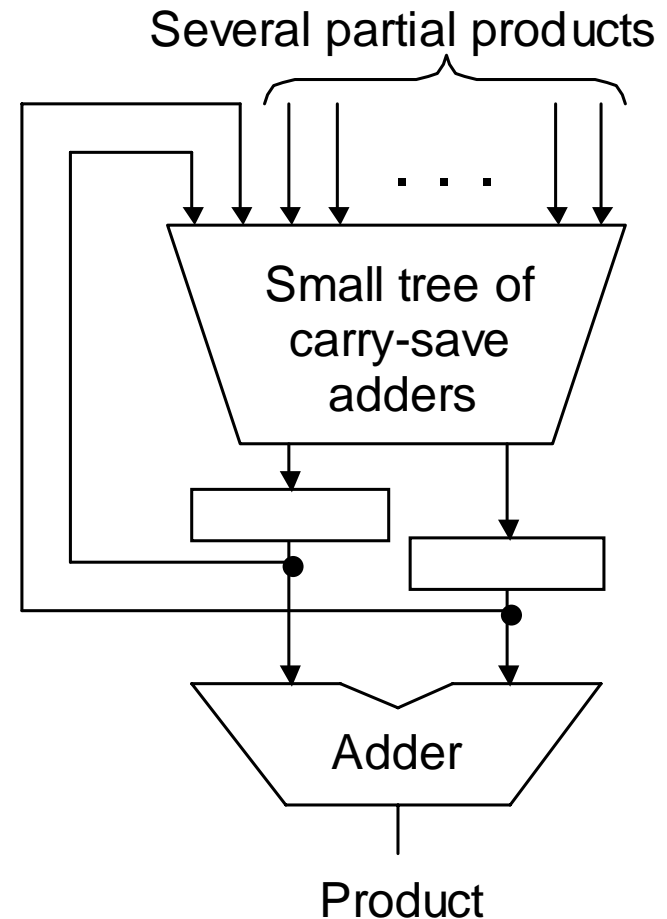
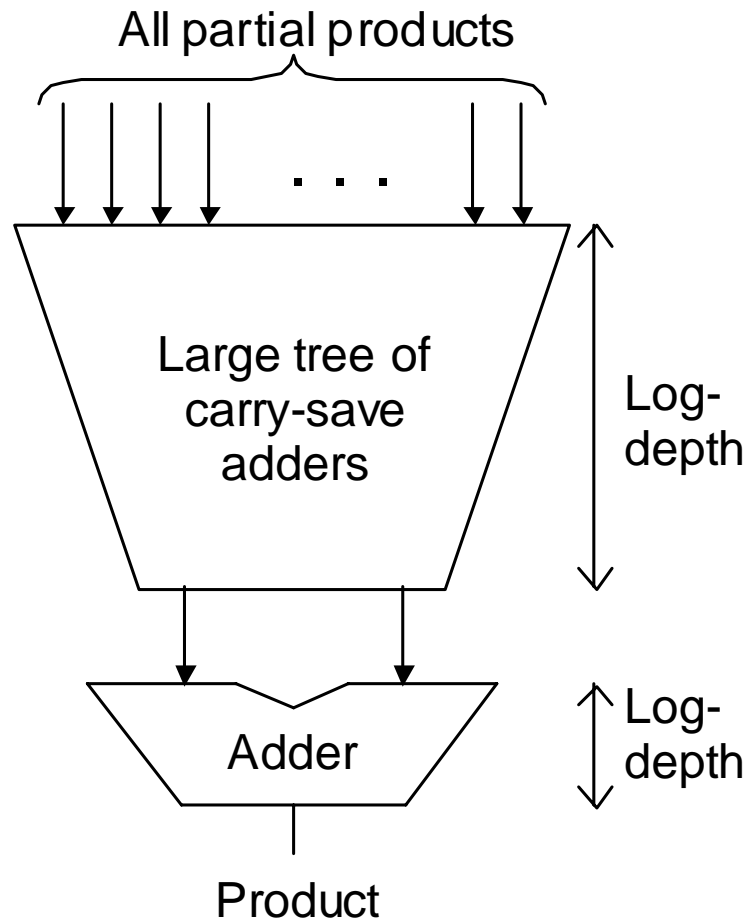


Fig. 11.1 General structure of a full-tree multiplier.

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

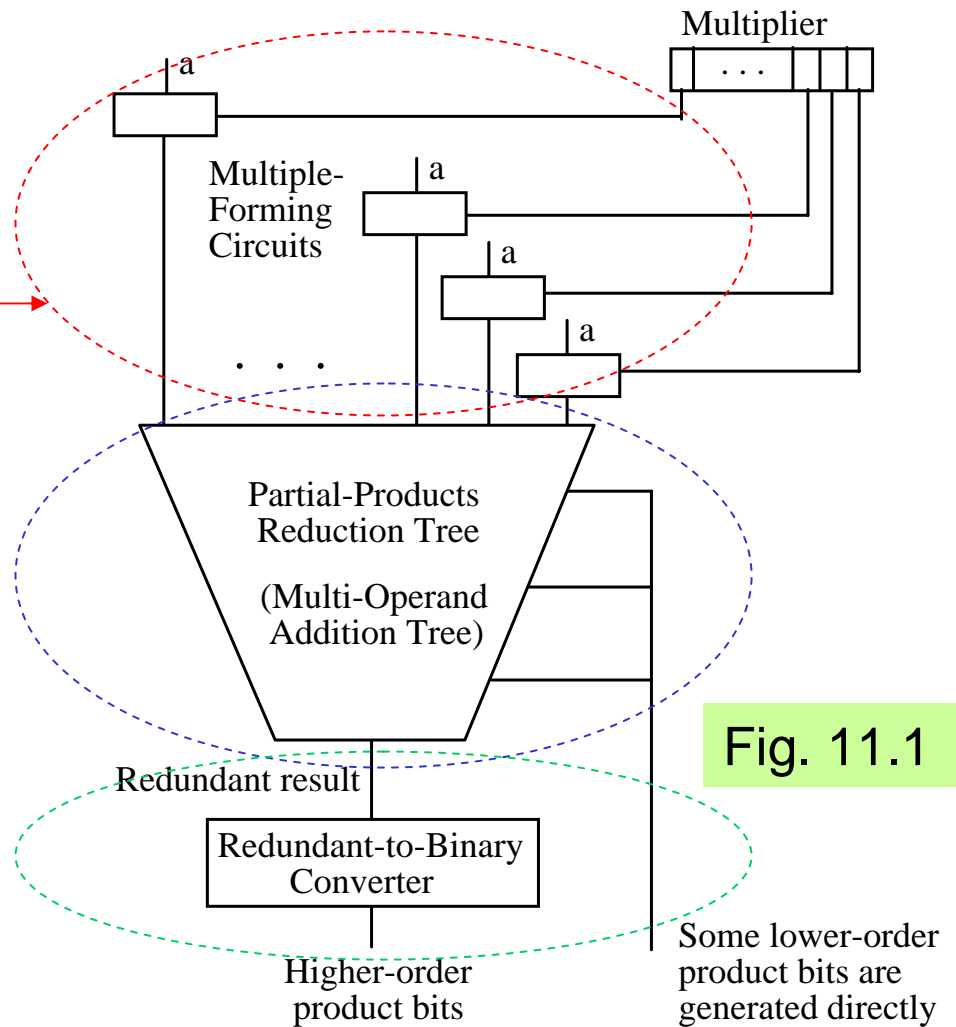
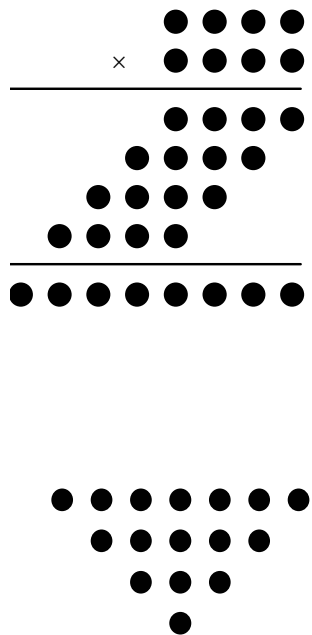
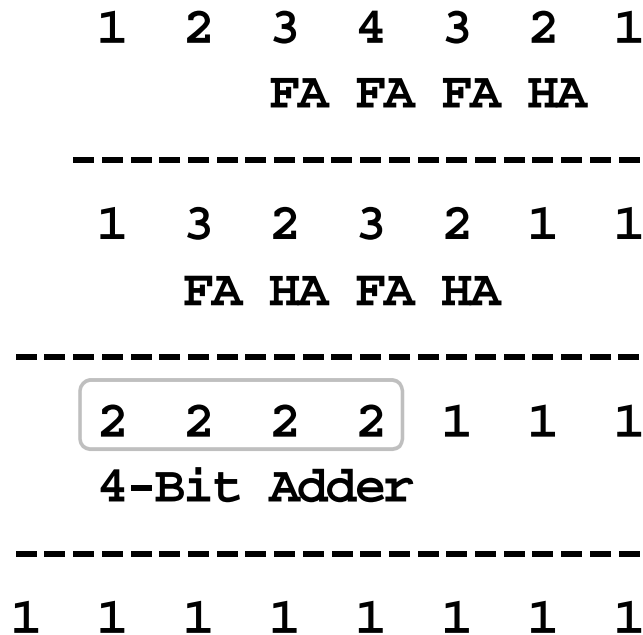


Fig. 11.1

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)

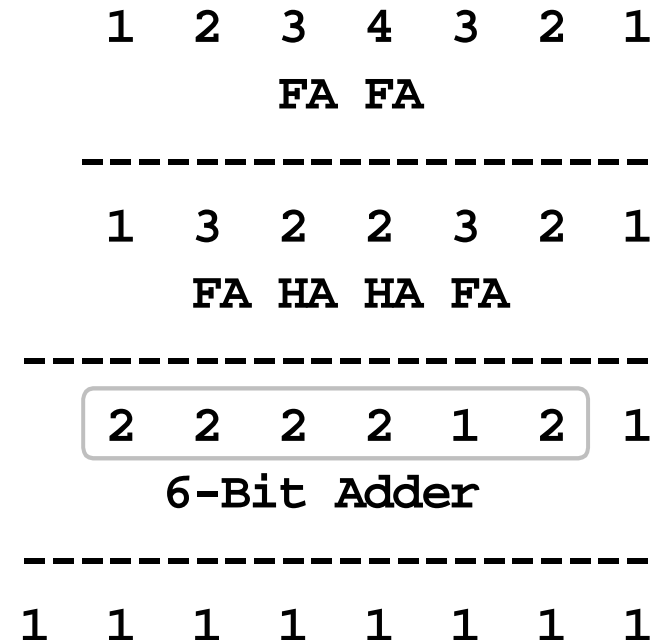


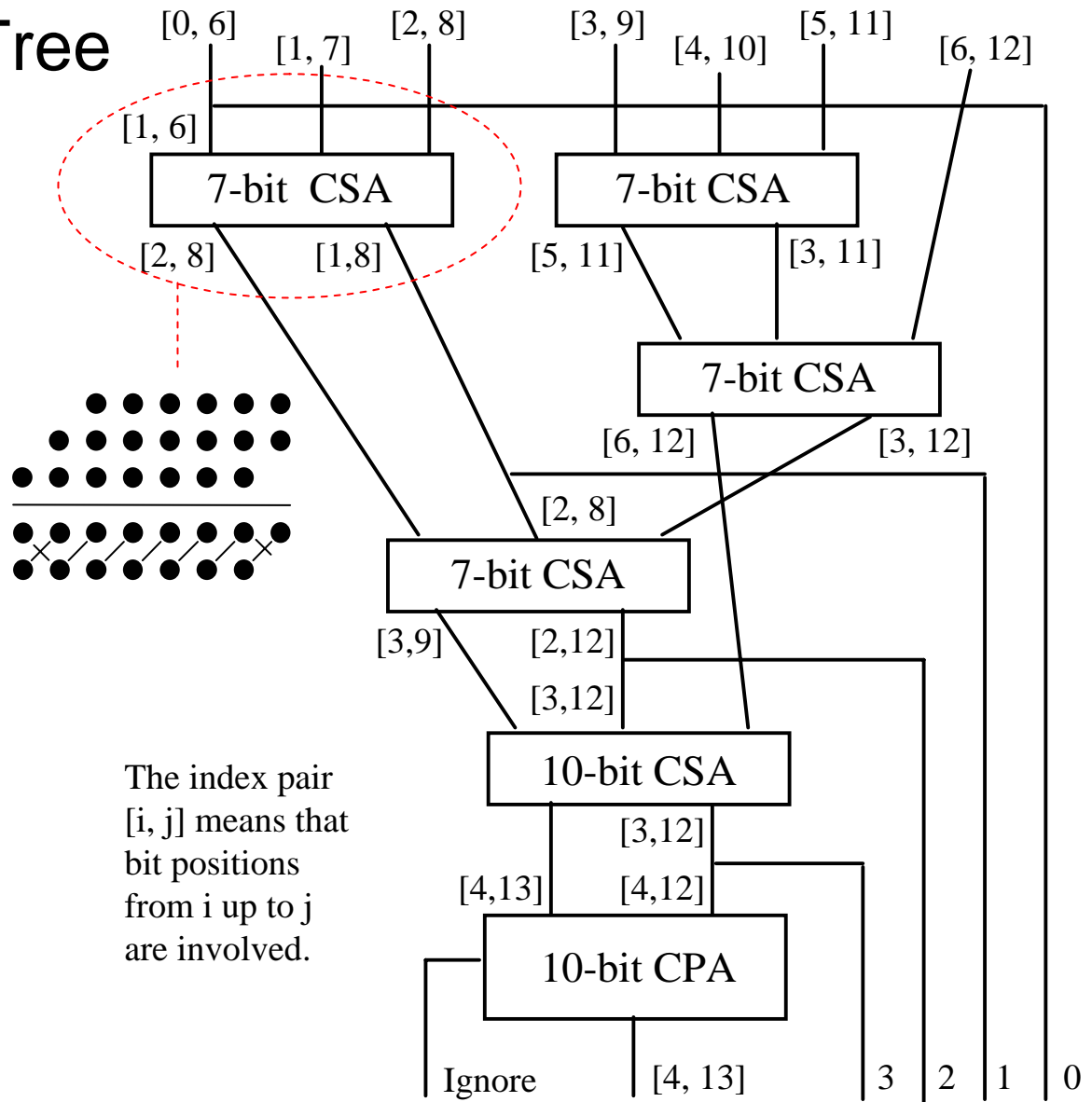
Fig. 11.2 Two different binary 4×4 tree multipliers.

Details of a CSA Tree

Fig. 11.3 Possible CSA tree for a 7×7 tree multiplier.

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

Thus, our motivation to examine alternate methods for partial products reduction

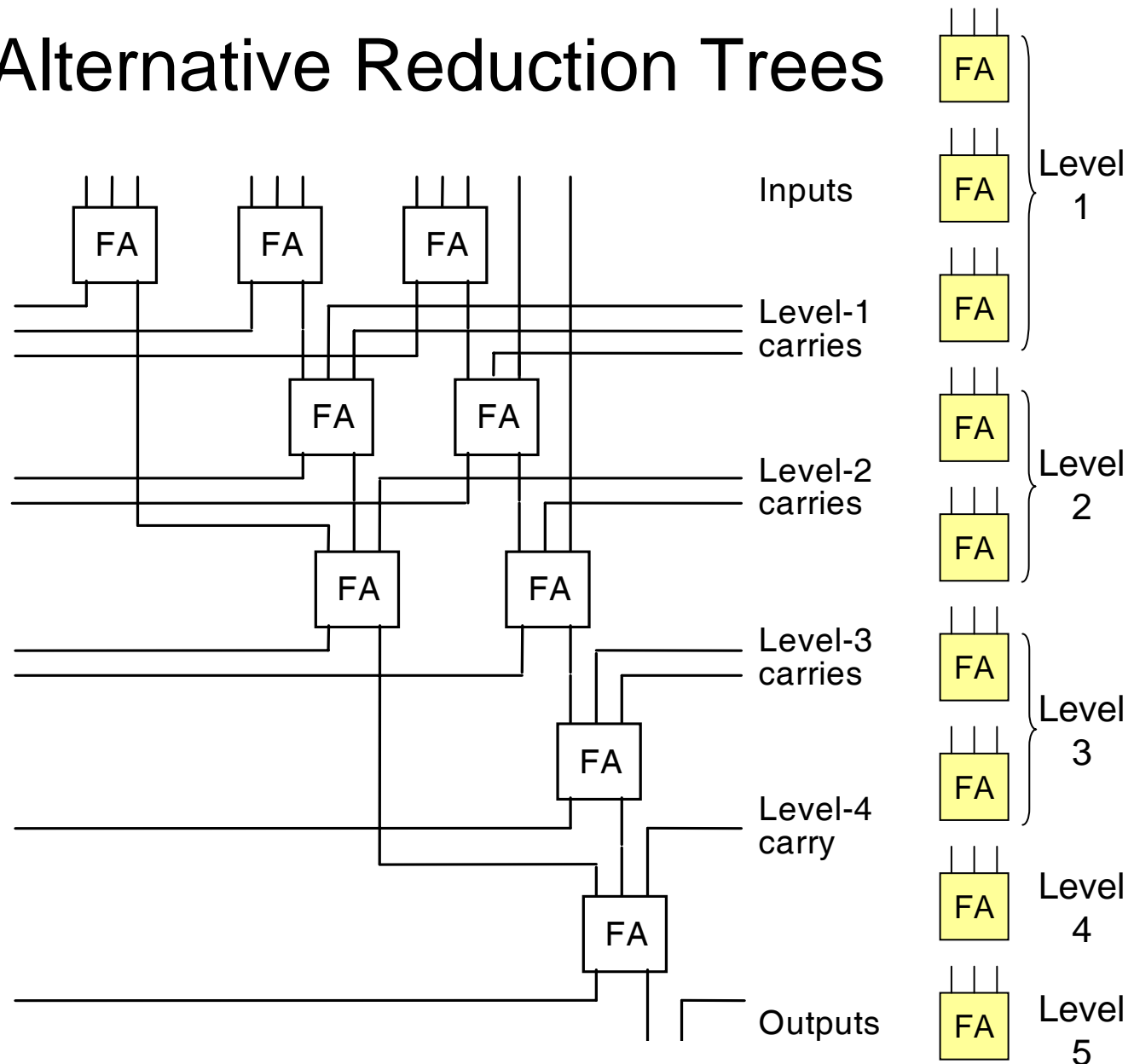


11.2 Alternative Reduction Trees

$$11 + \psi_1 = 2\psi_1 + 3$$

Therefore, $\psi_1 = 8$
carries are needed

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



Binary Tree of 4-to-2 Reduction Modules

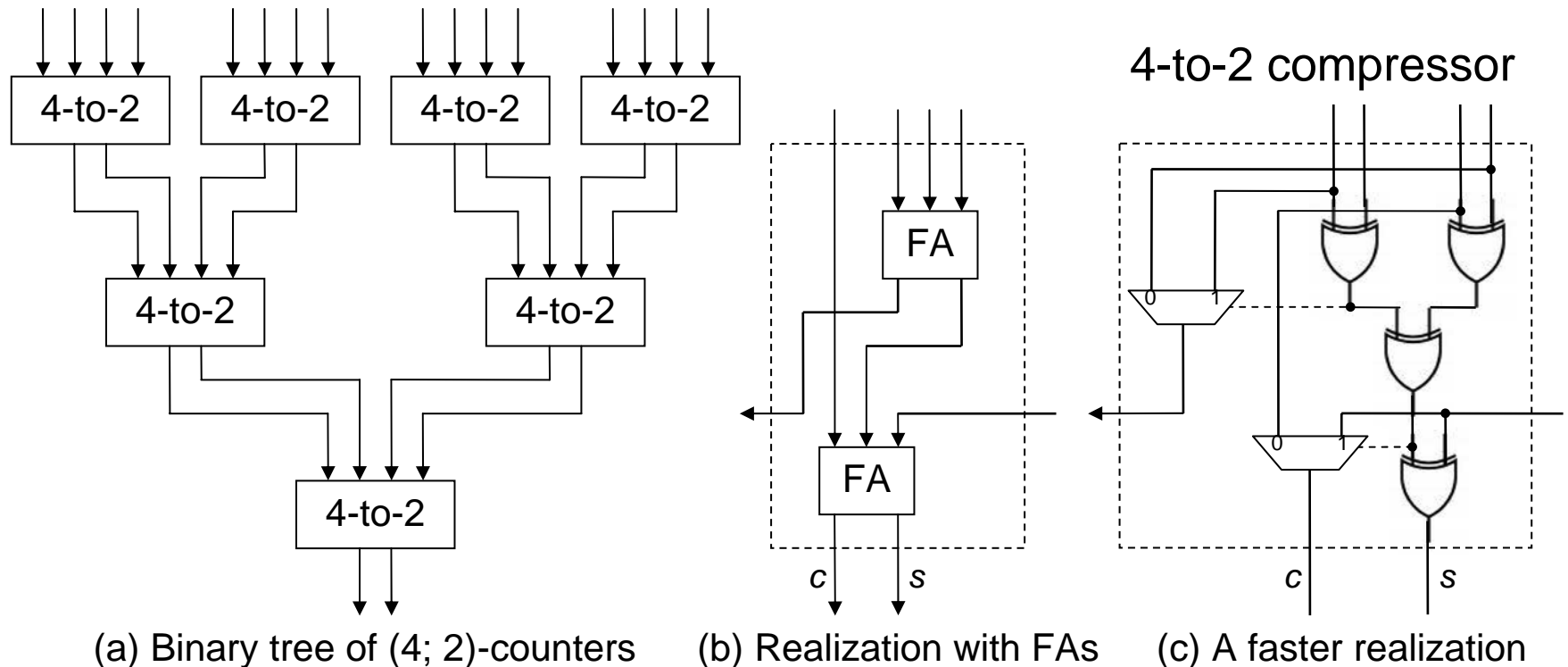


Fig. 11.5 Tree multiplier with a more regular structure based on 4-to-2 reduction modules.

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

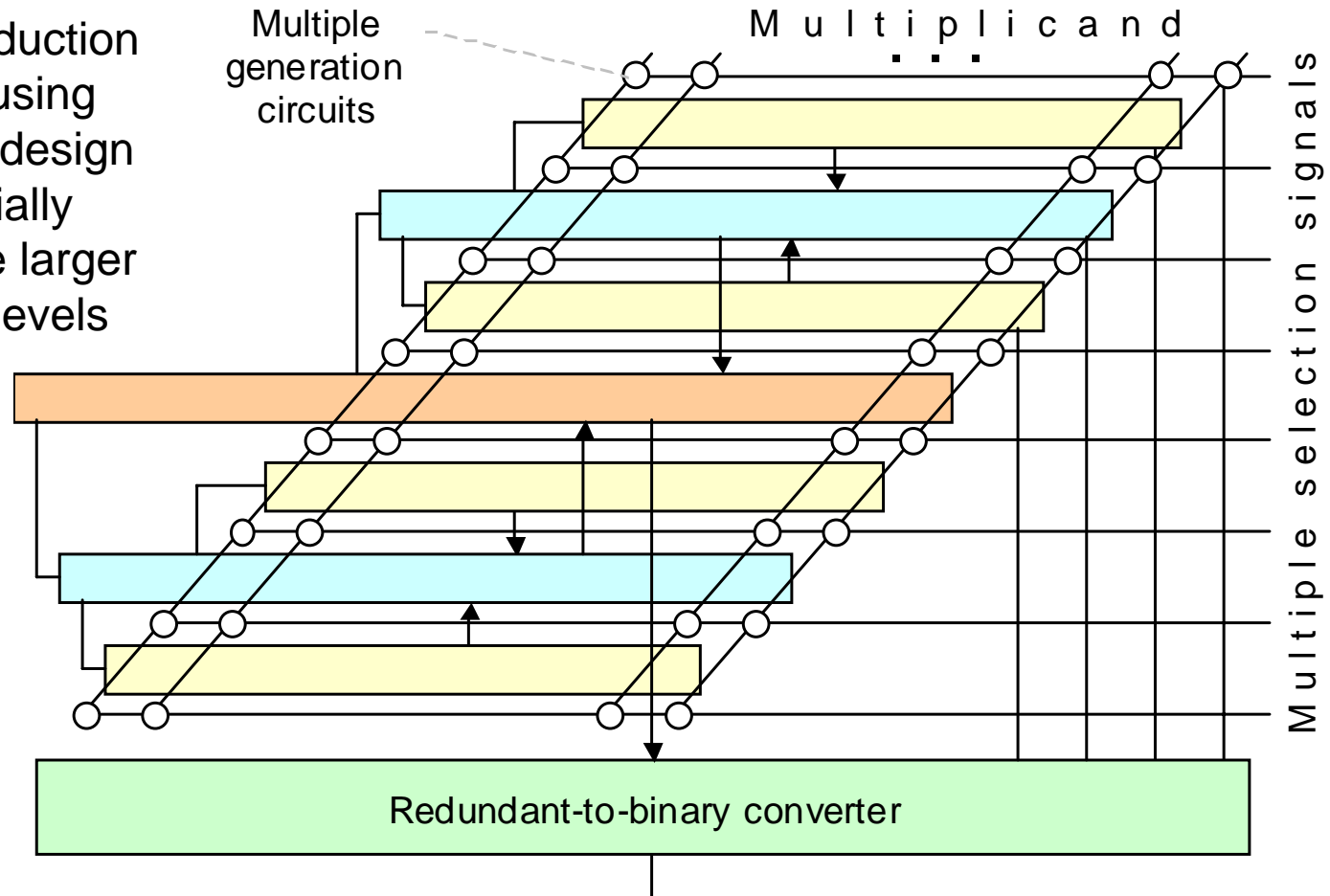
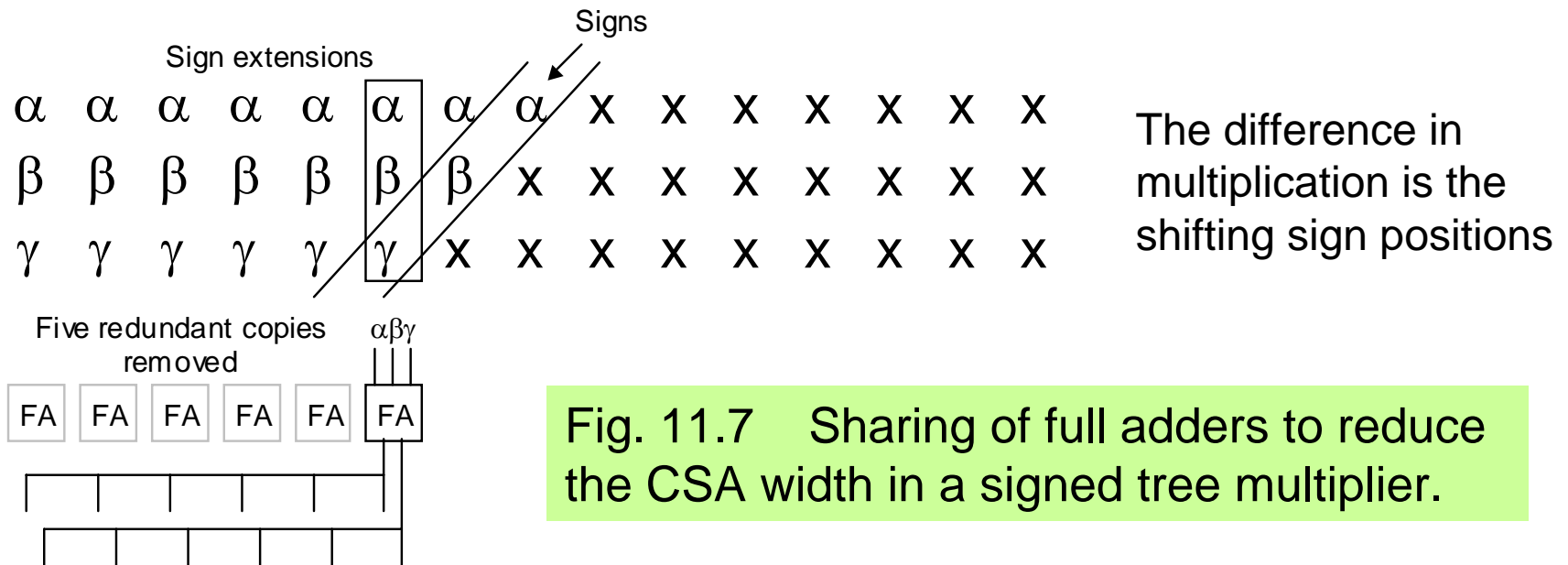


Fig. 11.6 Layout of a partial-products reduction tree composed of 4-to-2 reduction modules. Each solid arrow represents two numbers.

11.3 Tree Multipliers for Signed Numbers

----- Extended positions -----					Sign	Magnitude positions -----				
X_{k-1}	X_{k-1}	X_{k-1}	X_{k-1}	X_{k-1}	X_{k-1}	X_{k-2}	X_{k-3}	X_{k-4}	...	
Y_{k-1}	Y_{k-1}	Y_{k-1}	Y_{k-1}	Y_{k-1}	Y_{k-1}	Y_{k-2}	Y_{k-3}	Y_{k-4}	...	
Z_{k-1}	Z_{k-1}	Z_{k-1}	Z_{k-1}	Z_{k-1}	Z_{k-1}	Z_{k-2}	Z_{k-3}	Z_{k-4}	...	

From Fig. 8.19a Sign extension in multioperand addition.



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.

a. Unsigned					\times	a_4	a_3	a_2	a_1	a_0
						x_4	x_3	x_2	x_1	x_0

						a_4x_0	a_3x_0	a_2x_0	a_1x_0	a_0x_0
						a_4x_1	a_3x_1	a_2x_1	a_1x_1	a_0x_1
						a_4x_2	a_3x_2	a_2x_2	a_1x_2	a_0x_2
						a_4x_3	a_3x_3	a_2x_3	a_1x_3	a_0x_3
						a_4x_4	a_3x_4	a_2x_4	a_1x_4	a_0x_4

p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0	
b. 2's-complement					\times	a_4	a_3	a_2	a_1	a_0
						x_4	x_3	x_2	x_1	x_0

						a_4x_0	a_3x_0	a_2x_0	a_1x_0	a_0x_0
						$-a_4x_1$	a_3x_1	a_2x_1	a_1x_1	a_0x_1
						$-a_4x_2$	a_3x_2	a_2x_2	a_1x_2	a_0x_2
						$-a_4x_3$	a_3x_3	a_2x_3	a_1x_3	a_0x_3
						$-a_4x_4$	a_3x_4	a_2x_4	a_1x_4	a_0x_4

p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0	
c. Baugh-Wooley					\times	a_4	a_3	a_2	a_1	a_0
						x_4	x_3	x_2	x_1	x_0

						a_4x_0	a_3x_0	a_2x_0	a_1x_0	a_0x_0
						$-a_4x_1$	a_3x_1	a_2x_1	a_1x_1	a_0x_1
						a_4x_2	a_3x_2	a_2x_2	a_1x_2	a_0x_2
						$-a_4x_3$	a_3x_3	a_2x_3	a_1x_3	a_0x_3
						a_4x_4	a_3x_4	a_2x_4	a_1x_4	a_0x_4

p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0	
d. Modified B-W					\times	a_4	a_3	a_2	a_1	a_0
						x_4	x_3	x_2	x_1	x_0

						a_4x_0	a_3x_0	a_2x_0	a_1x_0	a_0x_0
						$-a_4x_1$	a_3x_1	a_2x_1	a_1x_1	a_0x_1
						a_4x_2	a_3x_2	a_2x_2	a_1x_2	a_0x_2
						$-a_4x_3$	a_3x_3	a_2x_3	a_1x_3	a_0x_3
						a_4x_4	a_3x_4	a_2x_4	a_1x_4	a_0x_4

p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0	

The Baugh-Wooley Method and Its Modified Form

Fig. 11.8

c. Baugh-Wooley

$$\begin{aligned} -a_4x_0 &= a_4(1 - x_0) - a_4 \\ &= a_4x_0' - a_4 \end{aligned}$$

$$\begin{array}{cc} -a_4 & a_4x_0' \\ \uparrow & \\ a_4 & \end{array}$$

In next column

	\times	$\overline{a_4}$ x_4	a_3 x_3	a_2 x_2	a_1 x_1	a_0 x_0				
		$\overline{a_4x_0}$	a_3x_0	a_2x_0	a_1x_0	a_0x_0				
		$\overline{a_4x_1}$	a_3x_1	a_2x_1	a_1x_1	a_0x_1				
		$\overline{a_4x_2}$	a_3x_2	a_2x_2	a_1x_2	a_0x_2				
		$\overline{a_4x_3}$	a_3x_3	a_2x_3	a_1x_3	a_0x_3				
		$\overline{a_4x_4}$	a_3x_4	a_2x_4	a_1x_4	a_0x_4				
1		$\overline{a_4}$ x_4								
p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0	

d. Modified B-W

$$\begin{aligned} -a_4x_0 &= (1 - a_4x_0) - 1 \\ &= (a_4x_0)' - 1 \end{aligned}$$

$$\begin{array}{cc} -1 & (a_4x_0)' \\ \uparrow & \\ 1 & \end{array}$$

In next column

	\times	$\overline{a_4}$ x_4	a_3 x_3	a_2 x_2	a_1 x_1	a_0 x_0				
		$\overline{a_4x_0}$	a_3x_0	a_2x_0	a_1x_0	a_0x_0				
		$\overline{a_4x_1}$	a_3x_1	a_2x_1	a_1x_1	a_0x_1				
		$\overline{a_4x_2}$	a_3x_2	a_2x_2	a_1x_2	a_0x_2				
		$\overline{a_4x_3}$	a_3x_3	a_2x_3	a_1x_3	a_0x_3				
		$\overline{a_4x_4}$	a_3x_4	a_2x_4	a_1x_4	a_0x_4				
1		$\overline{a_4}$ x_4								
p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0	

Alternate Views of the Baugh-Wooley Methods

$$\begin{array}{rcccccc} + & 0 & 0 & -a_4x_3 & -a_4x_2 & -a_4x_1 & -a_4x_0 \\ + & 0 & 0 & -a_3x_4 & -a_2x_4 & -a_1x_4 & -a_0x_4 \end{array}$$

$$\begin{array}{cccccc} - & 0 & 0 & a_4x_3 & a_4x_2 & a_4x_1 & a_4x_0 \\ - & 0 & 0 & a_3x_4 & a_2x_4 & a_1x_4 & a_0x_4 \end{array}$$

$$\begin{array}{ccccccc} + & 1 & 1 & \overline{a_4x_3} & \overline{a_4x_2} & \overline{a_4x_1} & \overline{a_4x_0} \\ + & 1 & 1 & \overline{a_3x_4} & \overline{a_2x_4} & \overline{a_1x_4} & \overline{a_0x_4} \end{array}$$

$$\begin{array}{rcccccc}
 & & & & & 1 \\
 & & & & & 1 \\
 \hline
 + & \cancel{a_4} & \cancel{a_4} & a_4 \bar{x}_3 & a_4 \bar{x}_2 & a_4 \bar{x}_1 & a_4 \bar{x}_0 \\
 + & \cancel{x_4} & \cancel{x_4} & \bar{a}_3 x_4 & \bar{a}_2 x_4 & \bar{a}_1 x_4 & \bar{a}_0 x_4 \\
 & & & & & & a_4 \\
 & & & & & & x_4 \\
 \hline
 \end{array}$$

a. Unsigned					\times	a_4 x_4	a_3 x_3	a_2 x_2	a_1 x_1	a_0 x_0
						$a_4 x_0$	$a_3 x_0$	$a_2 x_0$	$a_1 x_0$	$a_0 x_0$
					$a_4 x_1$	$a_3 x_1$	$a_2 x_1$	$a_1 x_1$	$a_0 x_1$	
					$a_4 x_2$	$a_3 x_2$	$a_2 x_2$	$a_1 x_2$	$a_0 x_2$	
					$a_4 x_3$	$a_3 x_3$	$a_2 x_3$	$a_1 x_3$	$a_0 x_3$	
					$a_4 x_4$	$a_3 x_4$	$a_2 x_4$	$a_1 x_4$	$a_0 x_4$	
p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0	

[illegible]

c. Baugh-Wooley

					\times	$\frac{a_4}{x_4}$	$\frac{a_3}{x_3}$	$\frac{a_2}{x_2}$	$\frac{a_1}{x_1}$	$\frac{a_0}{x_0}$
						$\frac{a_4 x_0}{x_4}$	$\frac{a_3 x_0}{x_3}$	$\frac{a_2 x_0}{x_2}$	$\frac{a_1 x_0}{x_1}$	$\frac{a_0 x_0}{x_0}$
				$\frac{a_4 x_1}{x_4}$		$\frac{a_3 x_1}{x_3}$	$\frac{a_2 x_1}{x_2}$	$\frac{a_1 x_1}{x_1}$	$\frac{a_0 x_1}{x_0}$	
			$\frac{a_4 x_2}{x_4}$	$\frac{a_3 x_2}{x_3}$		$\frac{a_2 x_2}{x_2}$	$\frac{a_1 x_2}{x_1}$	$\frac{a_0 x_2}{x_0}$		
		$\frac{a_4 x_3}{x_4}$	$\frac{a_3 x_3}{x_3}$	$\frac{a_2 x_3}{x_2}$		$\frac{a_1 x_3}{x_1}$	$\frac{a_0 x_3}{x_0}$			
	$\frac{a_4 x_4}{x_4}$	$\frac{a_3 x_4}{x_3}$	$\frac{a_2 x_4}{x_2}$	$\frac{a_1 x_4}{x_1}$		$\frac{a_0 x_4}{x_0}$				
1	$\frac{a_4}{x_4}$					$\frac{a_4}{x_4}$				
p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0	

[illegible]

11.4 Partial-Tree and Truncated Multipliers

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

For small h , say ≤ 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

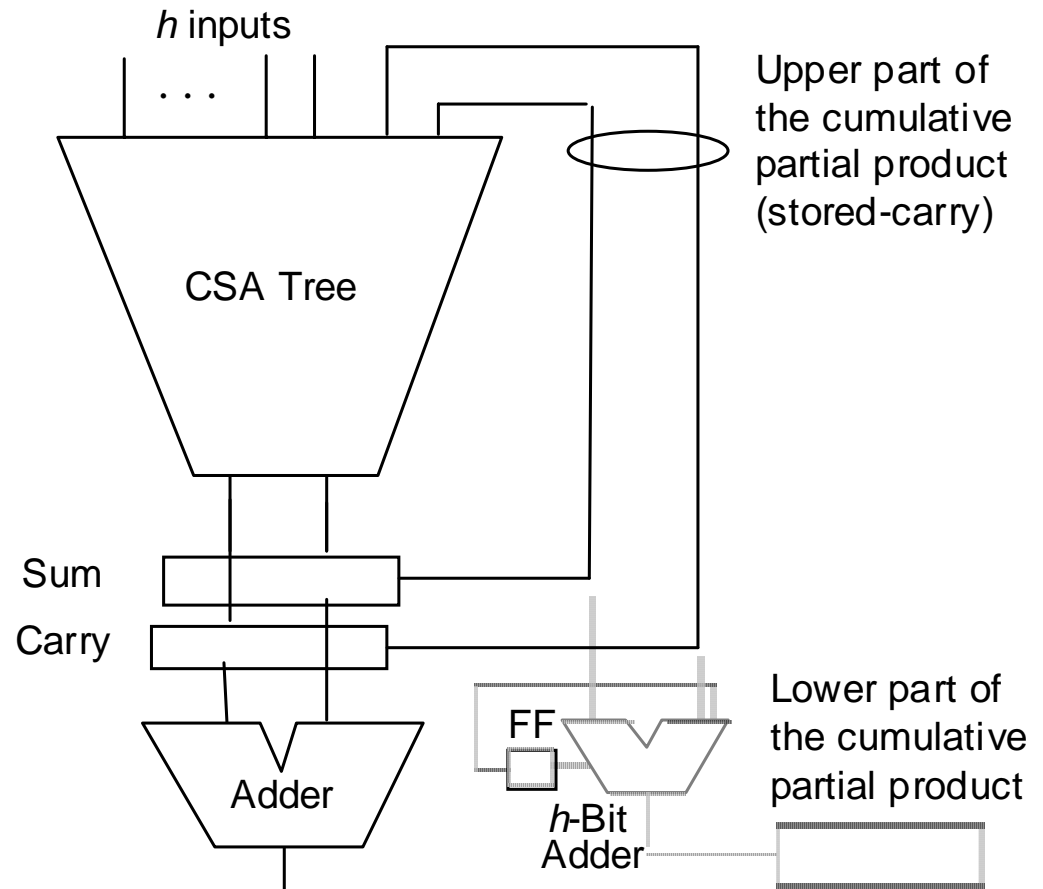


Fig. 11.9 General structure of a partial-tree multiplier.

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)

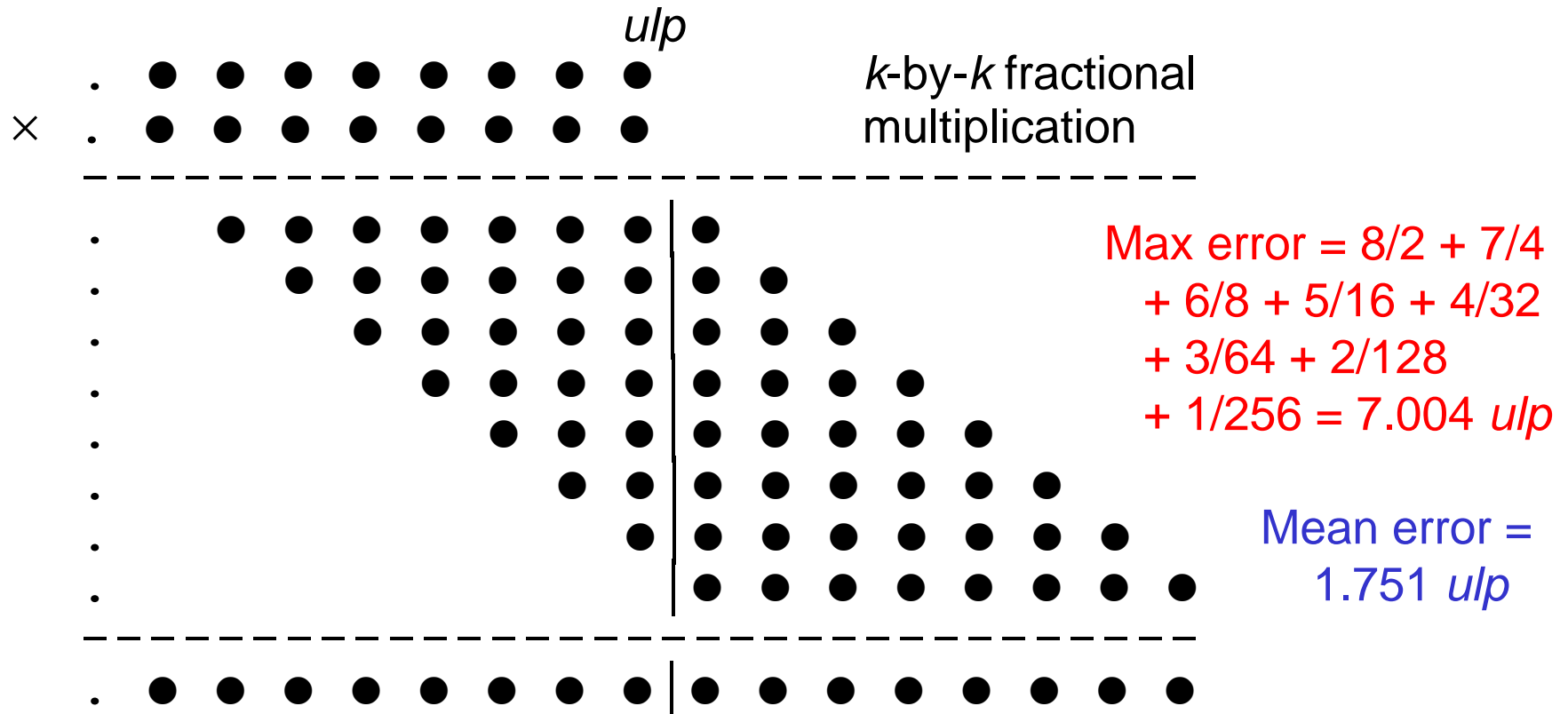
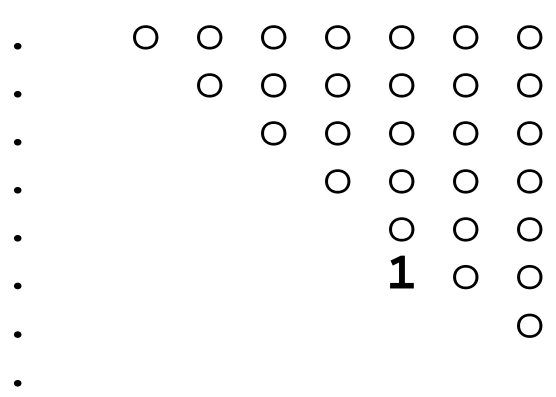


Fig. 11.10 The idea of a truncated multiplier with 8-bit fractional operands.

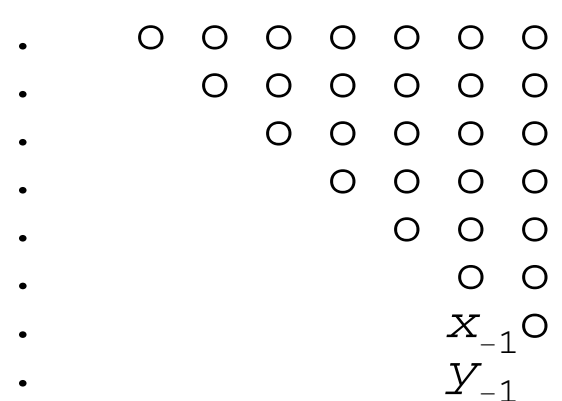
Truncated Multipliers with Error Compensation

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 \cong -3 ulp

Mean error = ? ulp

Max error = +? ulp

Max error \cong -? ulp

Mean error = ? ulp

11.5 Array Multipliers

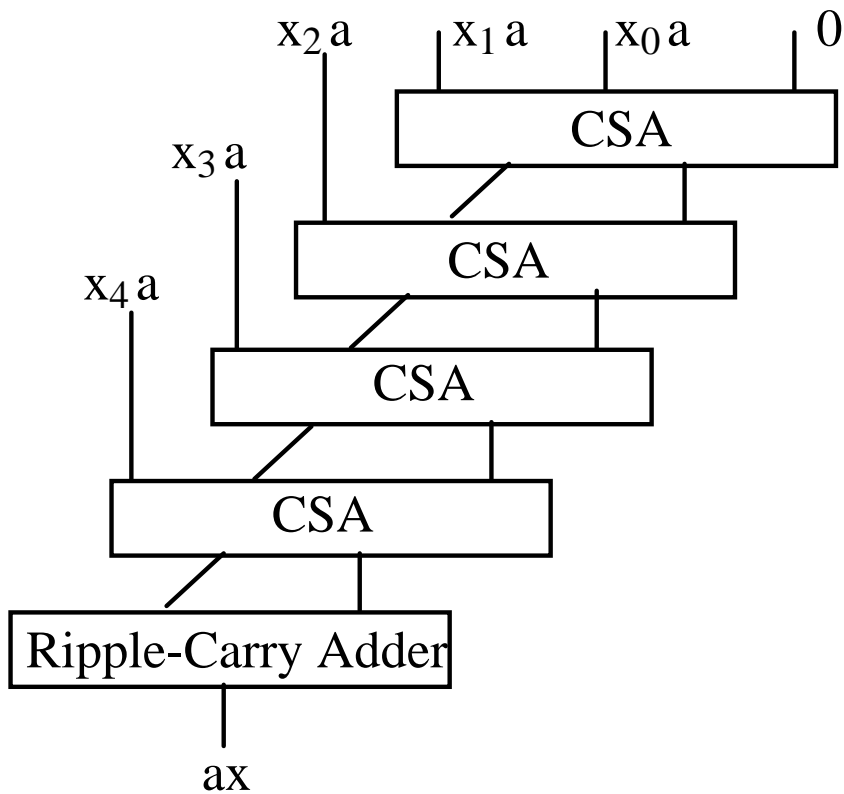


Fig. 11.11 A basic array multiplier uses a one-sided CSA tree and a ripple-carry adder.

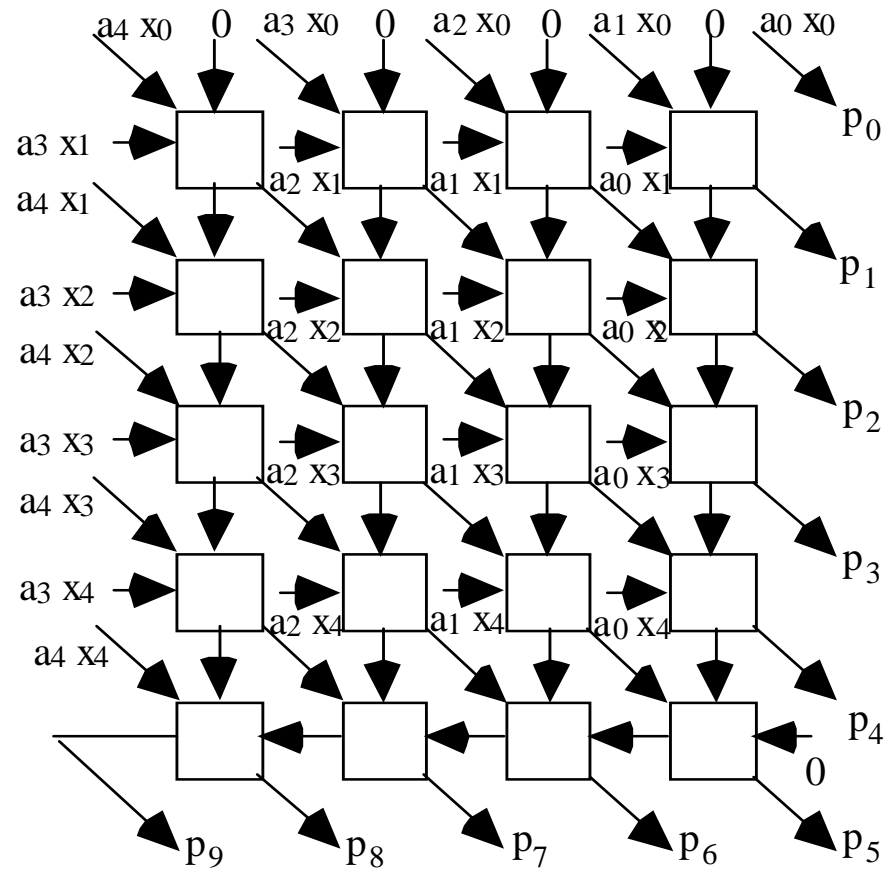
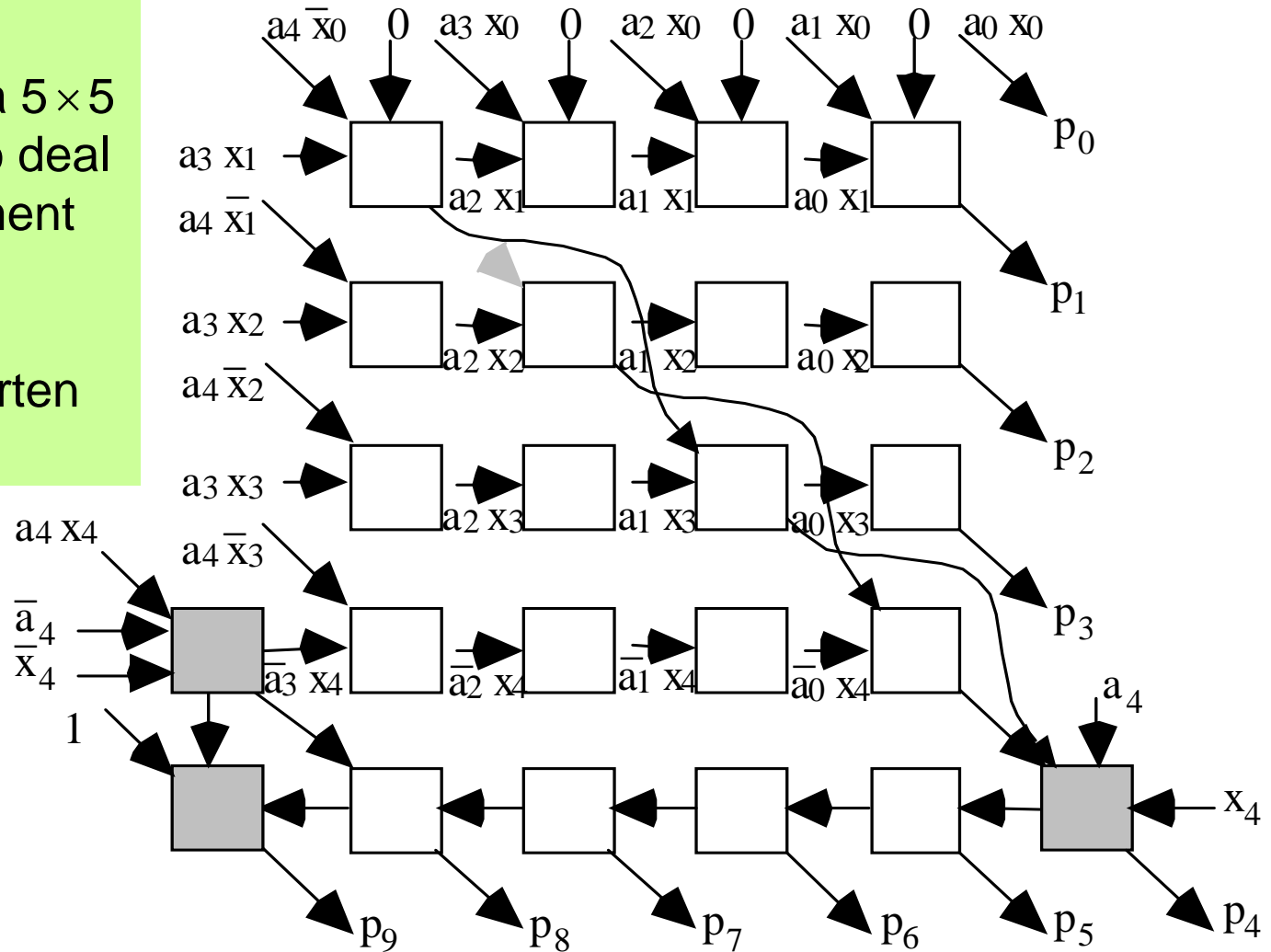


Fig. 11.12 Details of a 5×5 array multiplier using FA blocks.

Signed (2's-complement) Array Multiplier

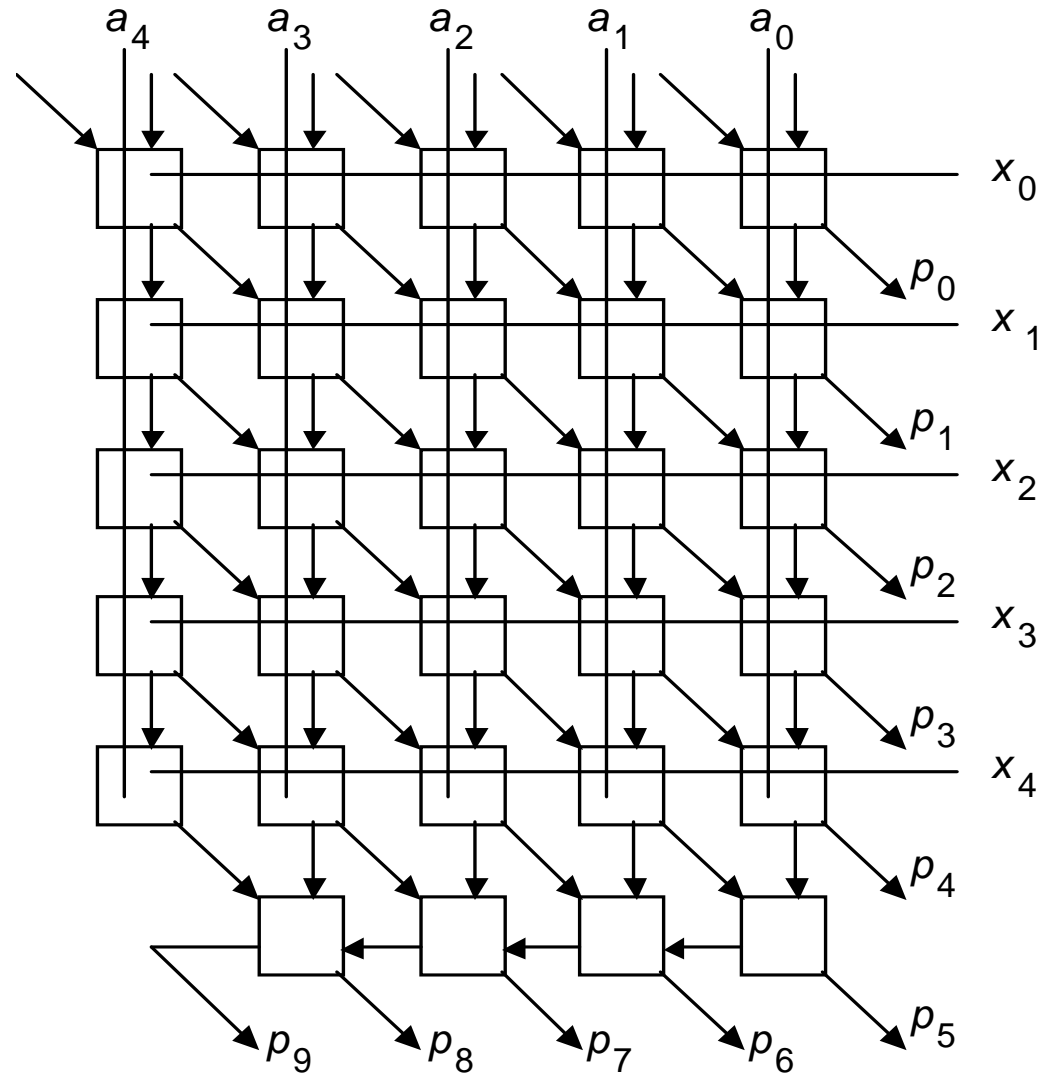
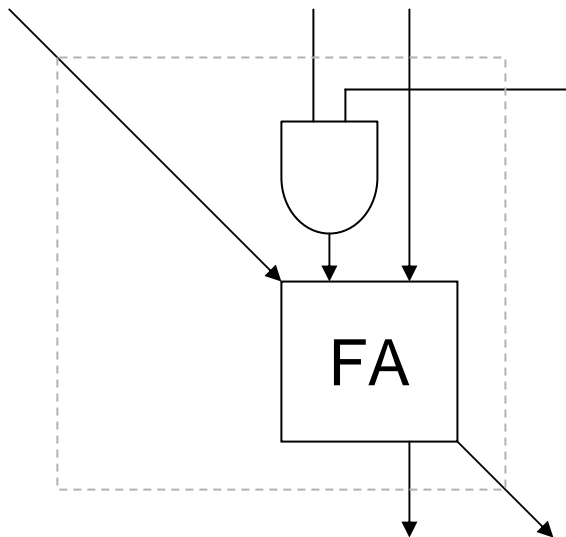
Fig. 11.13

Modifications in a 5×5 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

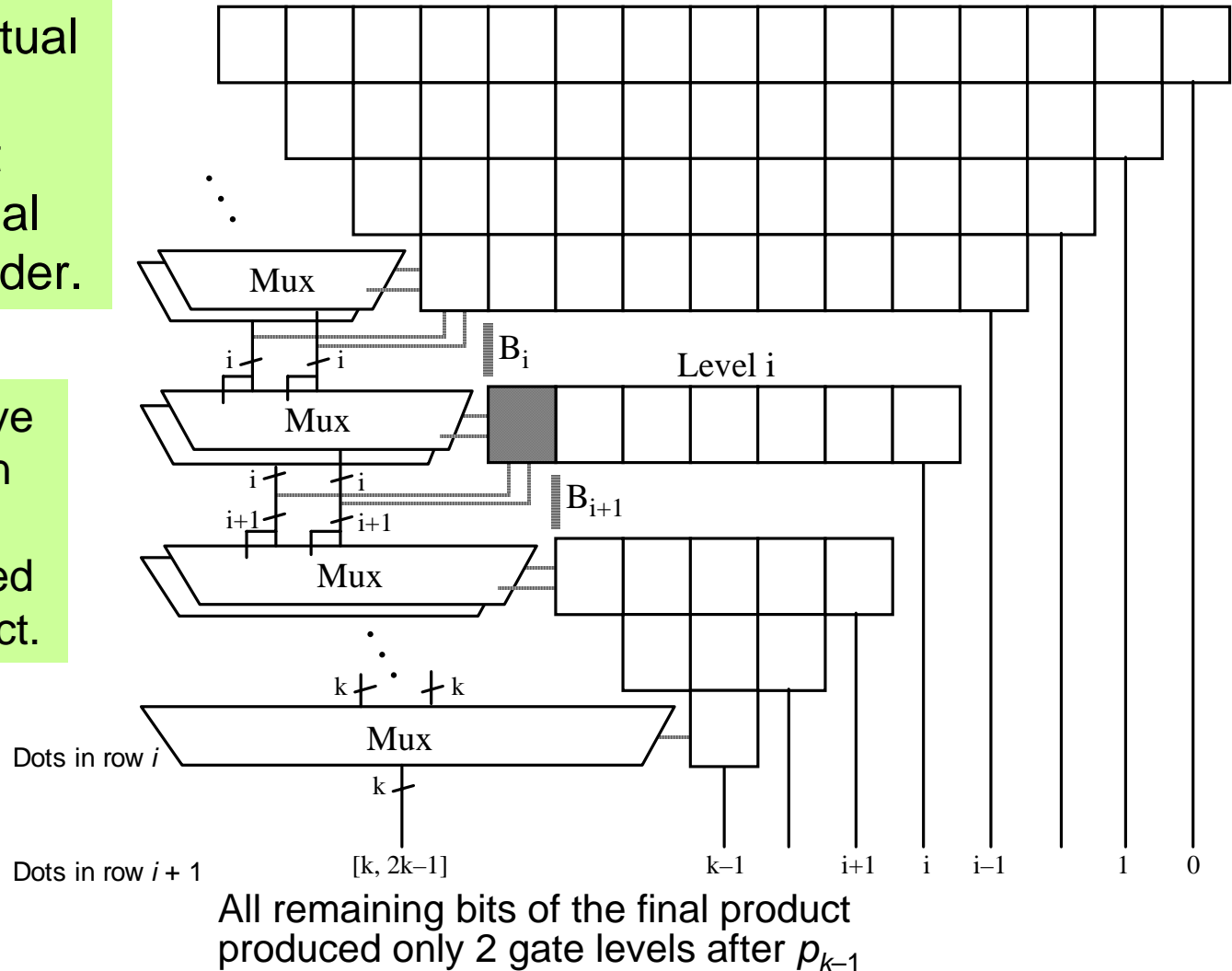
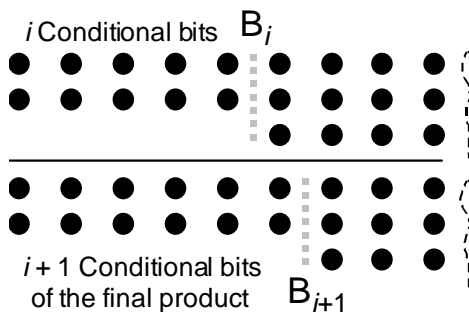
Fig. 11.14 Design of a 5×5 array multiplier with two additive inputs and full-adder blocks that include AND gates.



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.



11.6 Pipelined Tree and Array Multipliers

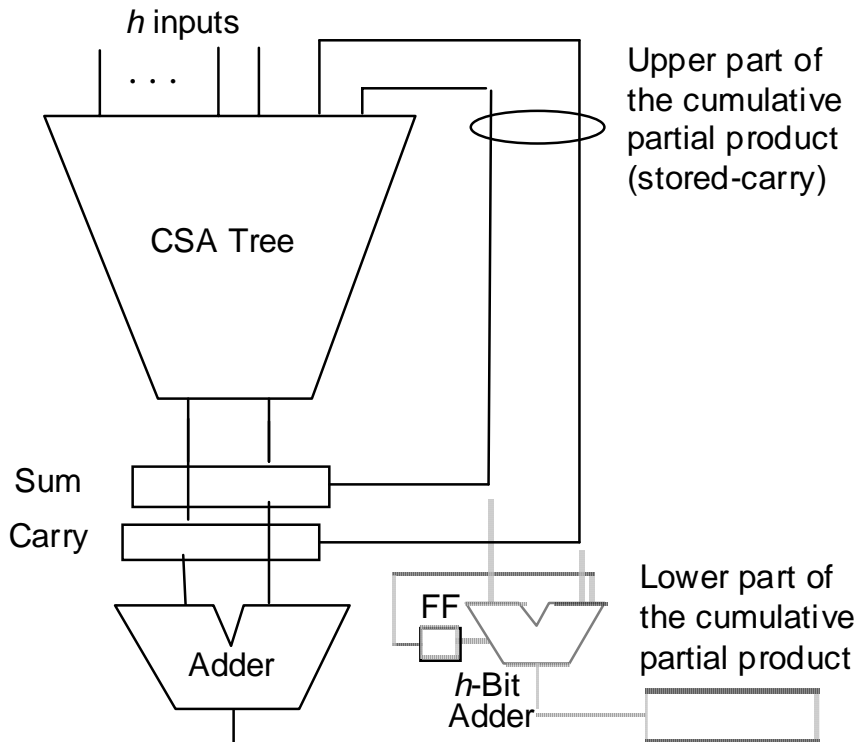


Fig. 11.9 General structure of a partial-tree multiplier.

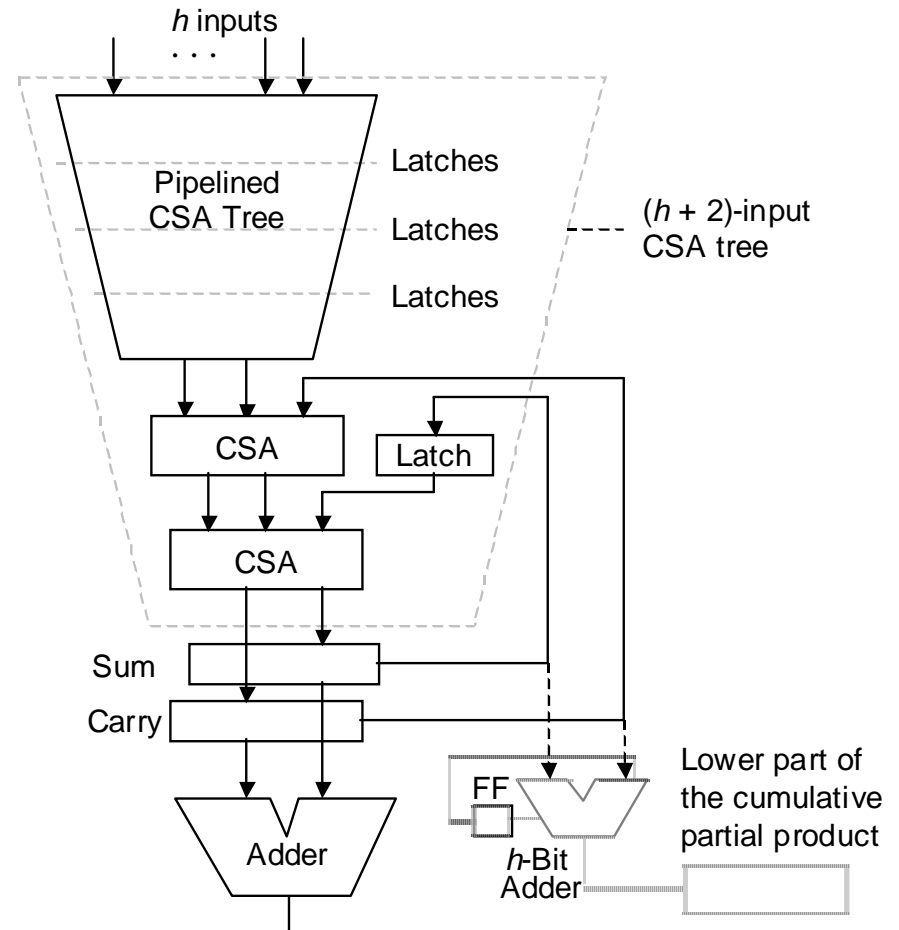


Fig. 11.17 Efficiently pipelined partial-tree multiplier.

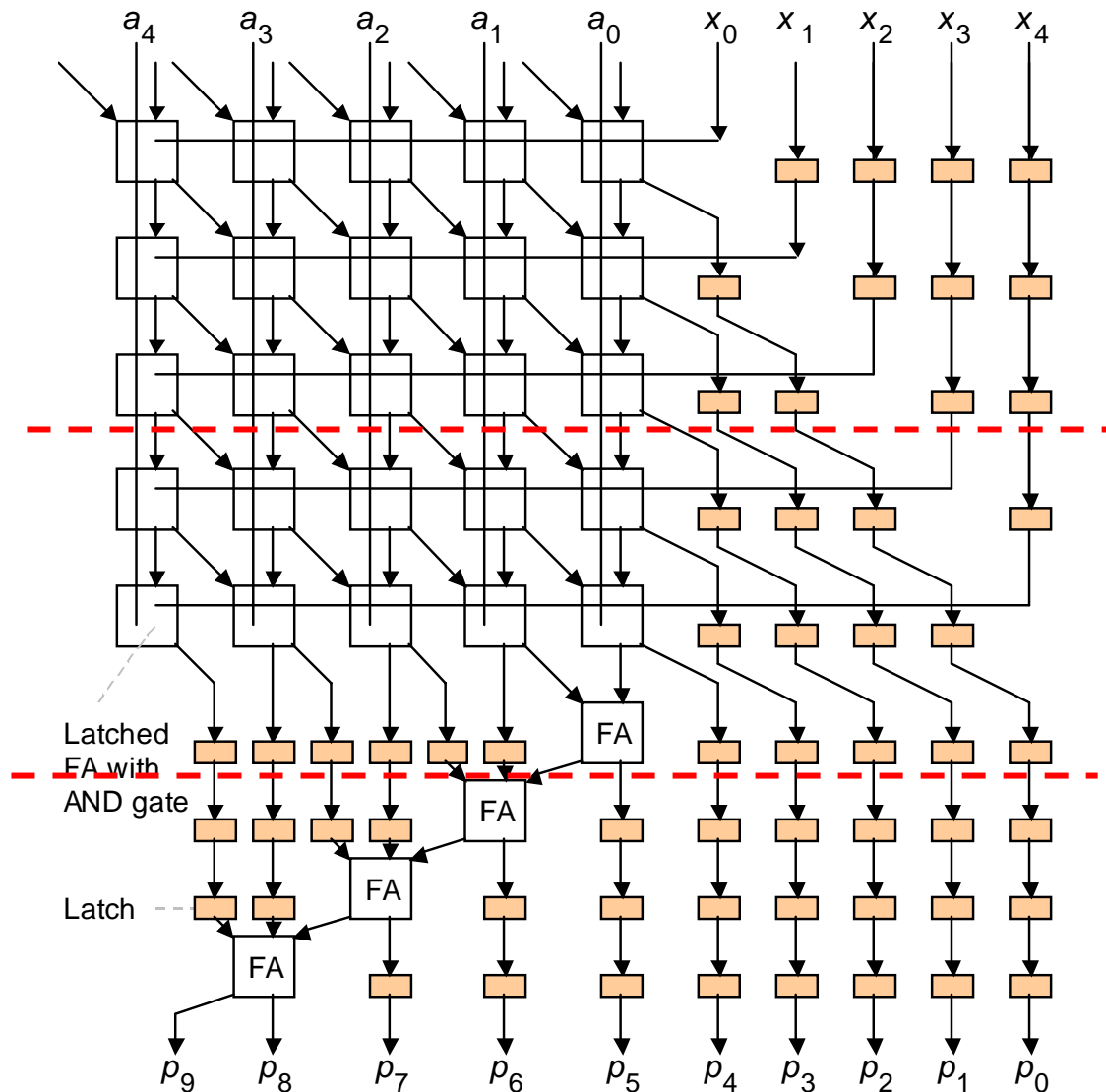
Pipelined Array Multipliers

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 5×5 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

Topics in This Chapter

12.1 Divide-and-Conquer Designs

12.2 Additive Multiply Modules

12.3 Bit-Serial Multipliers

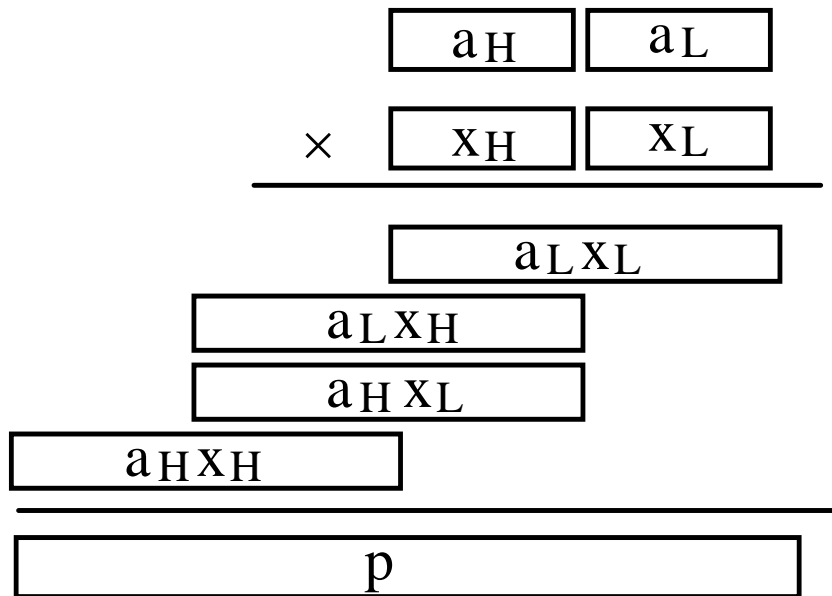
12.4 Modular Multipliers

12.5 The Special Case of Squaring

12.6 Combined Multiply-Add Units

12.1 Divide-and-Conquer Designs

Building wide multiplier from narrower ones



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

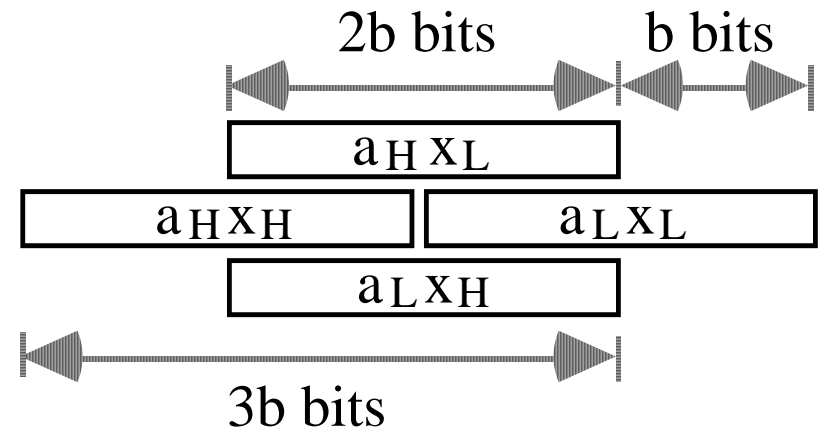


Fig. 12.1 Divide-and-conquer (recursive) strategy for synthesizing a $2b \times 2b$ multiplier from $b \times b$ multipliers.

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

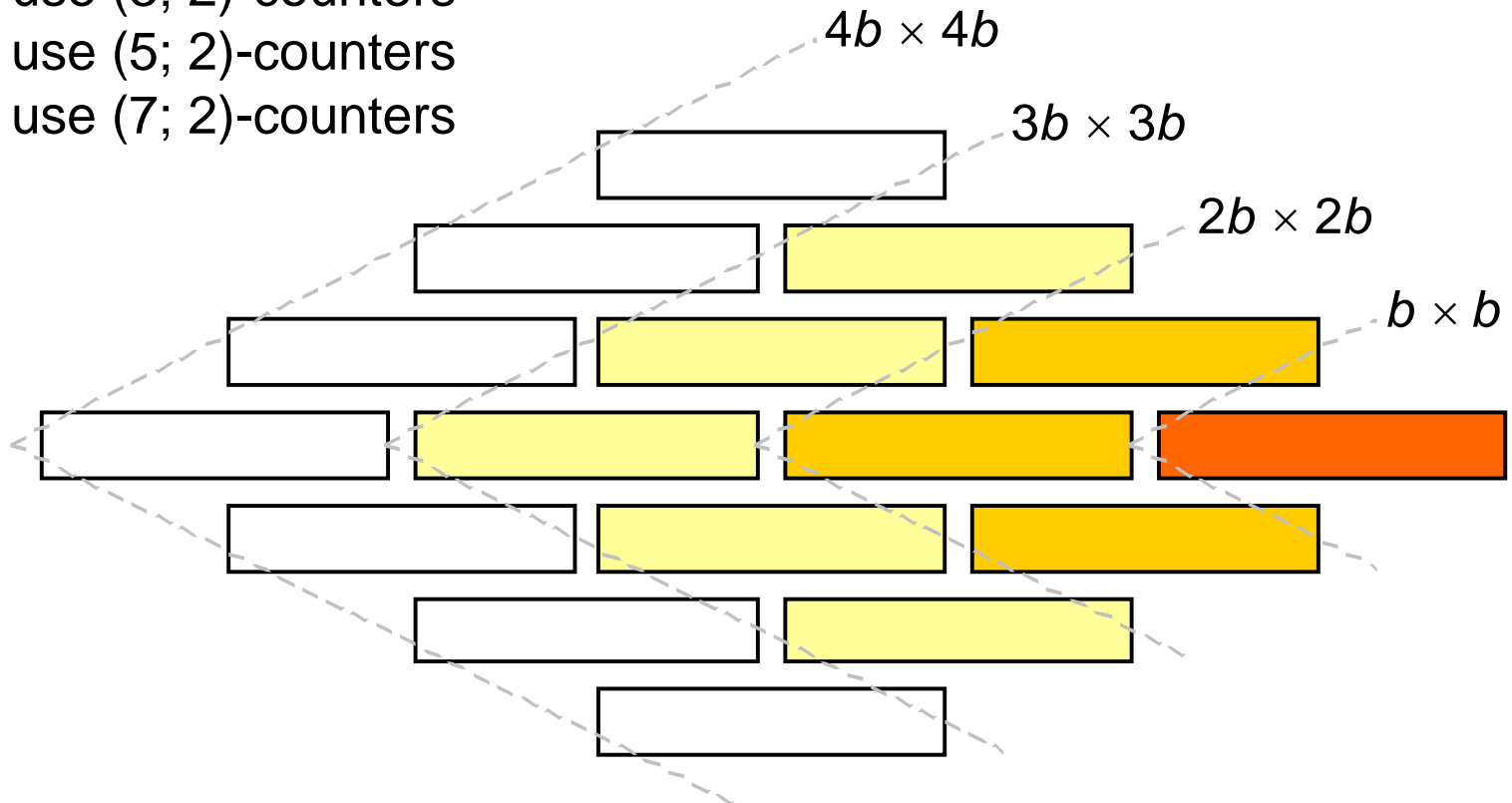
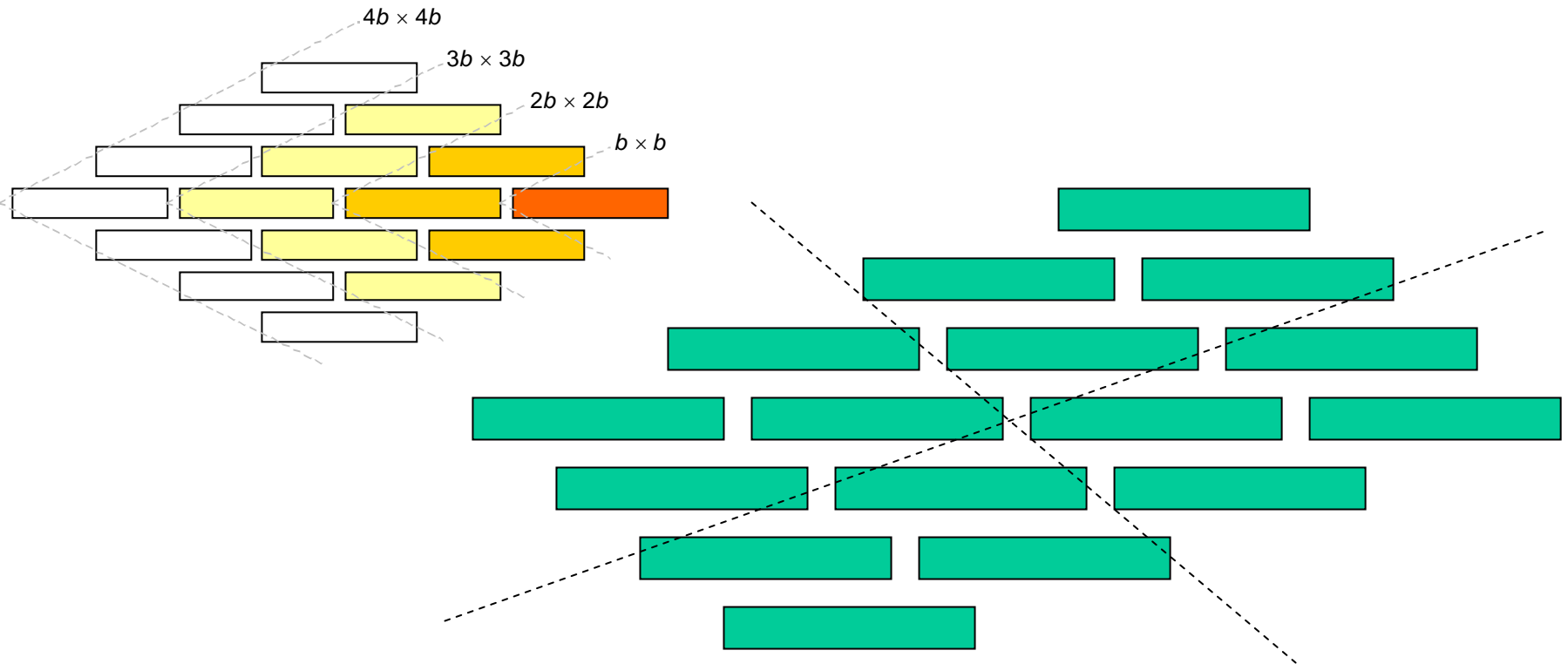


Fig. 12.2 Using $b \times b$ multipliers to synthesize $2b \times 2b$, $3b \times 3b$, and $4b \times 4b$ multipliers.

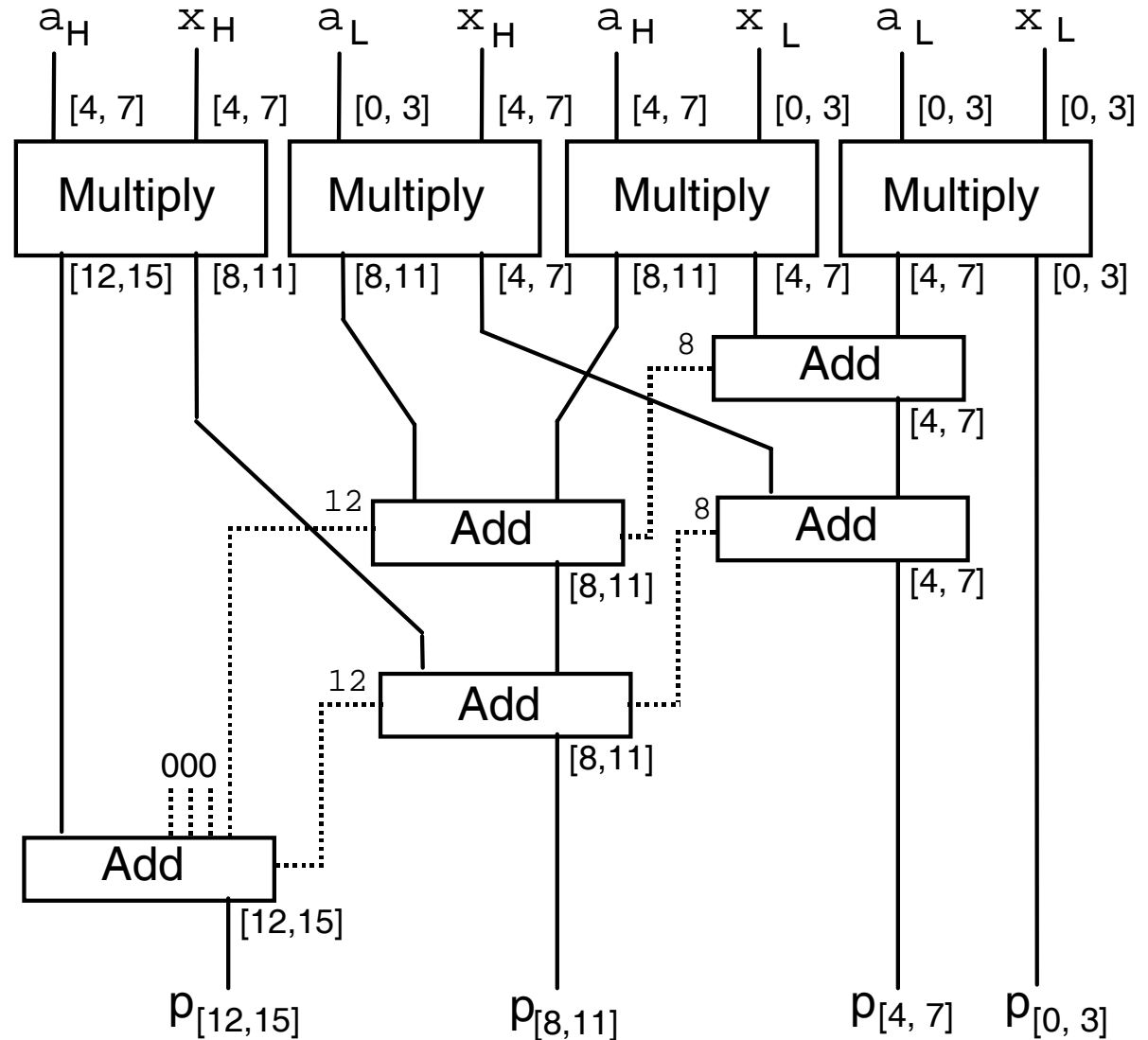
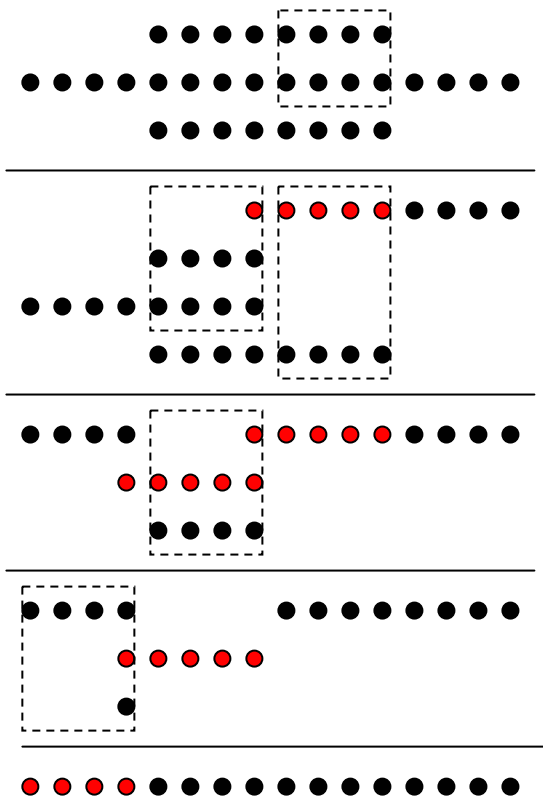
Using $b \times c$, rather than $b \times b$ Building Blocks



- $2b \times 2c$ 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 4×4 multipliers and 4-bit adders to synthesize an 8×8 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:

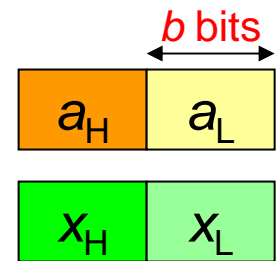
$$(2^b a_H + a_L) \times (2^b x_H + x_L) =$$

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

Mult 1

Mult 3

Mult 2



Benefit is quite significant for extremely wide operands

12.2 Additive Multiply Modules

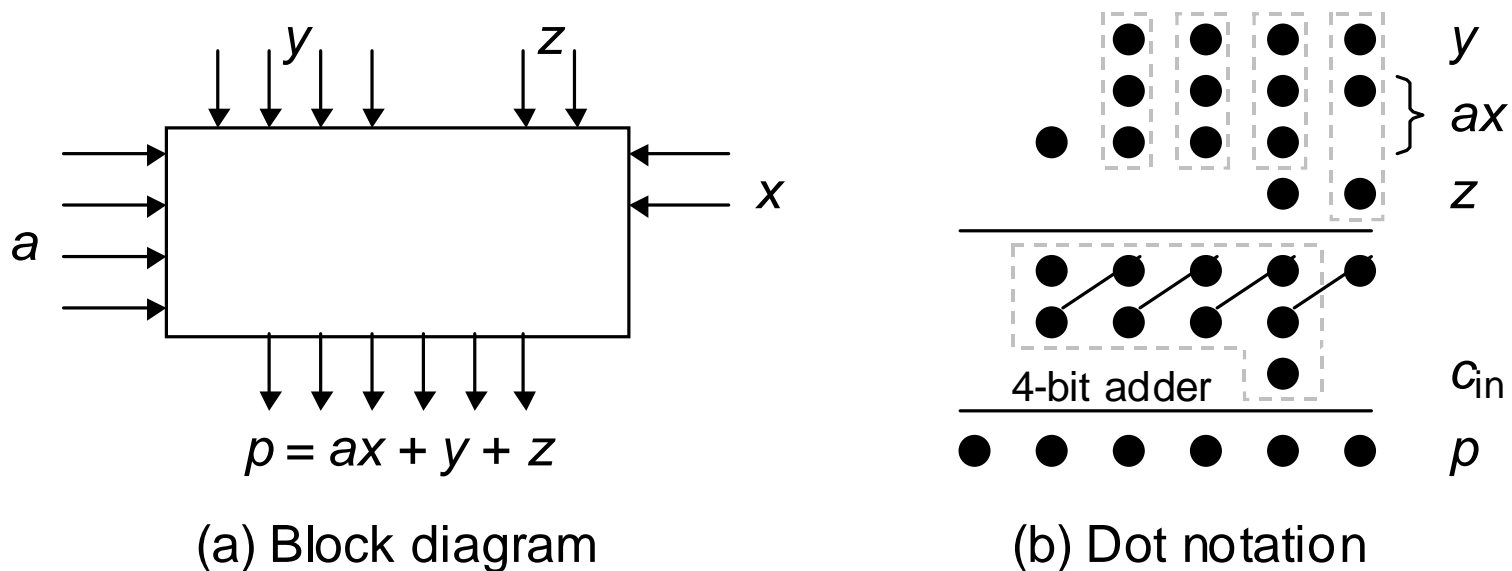
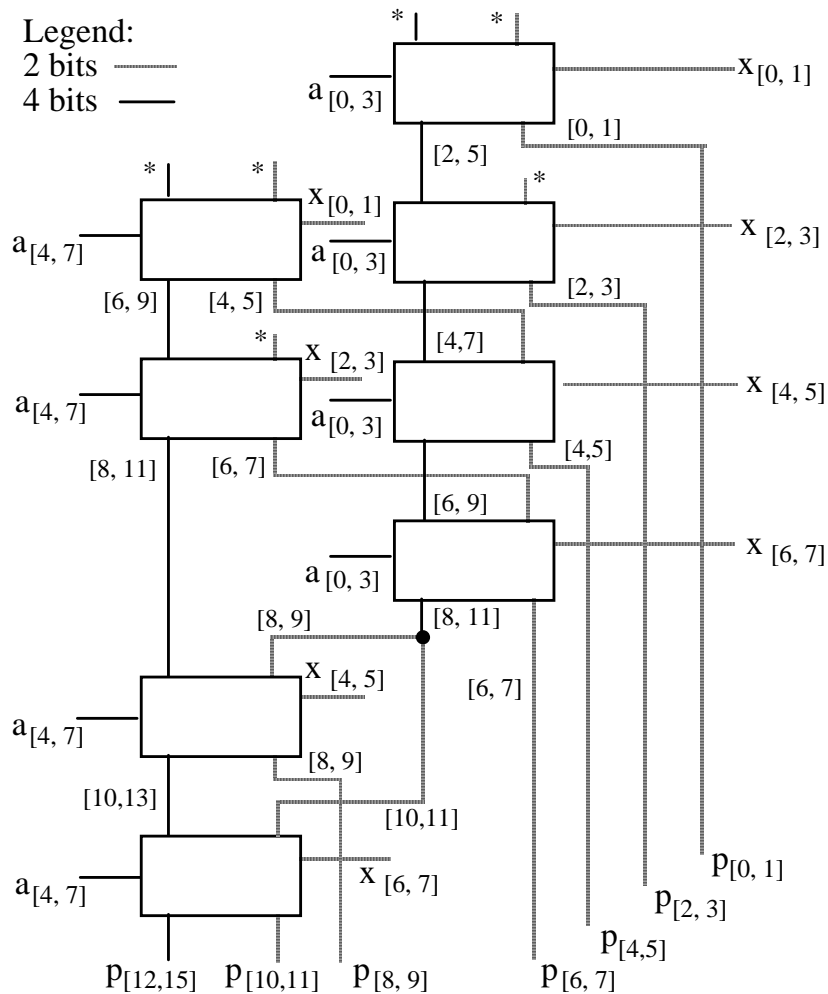


Fig. 12.4 Additive multiply module with 2×4 multiplier (ax) plus 4-bit and 2-bit additive inputs (y and z).

$b \times c$ AMM $\left\{ \begin{array}{l} b\text{-bit and } c\text{-bit multiplicative inputs} \\ b\text{-bit and } c\text{-bit additive inputs} \\ (b + c)\text{-bit output} \end{array} \right.$

$$(2^b - 1) \times (2^c - 1) + (2^b - 1) + (2^c - 1) = 2^{b+c} - 1$$

Multiplier Built of AMMs



Understanding
 an 8×8 multiplier
 built of 4×2
 AMMs using dot
 notation

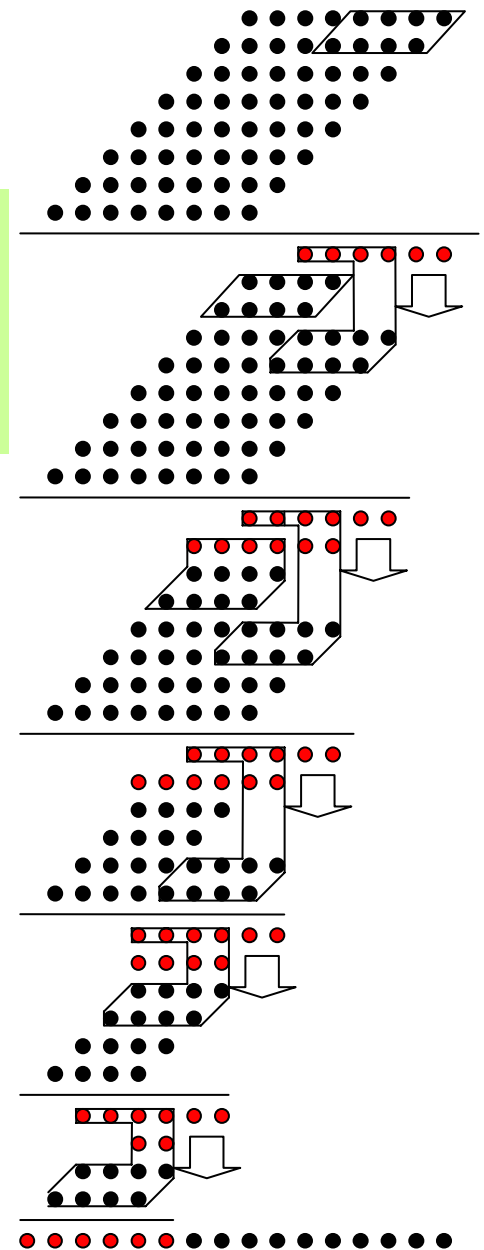
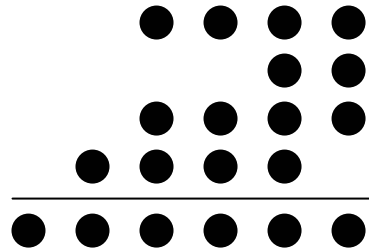
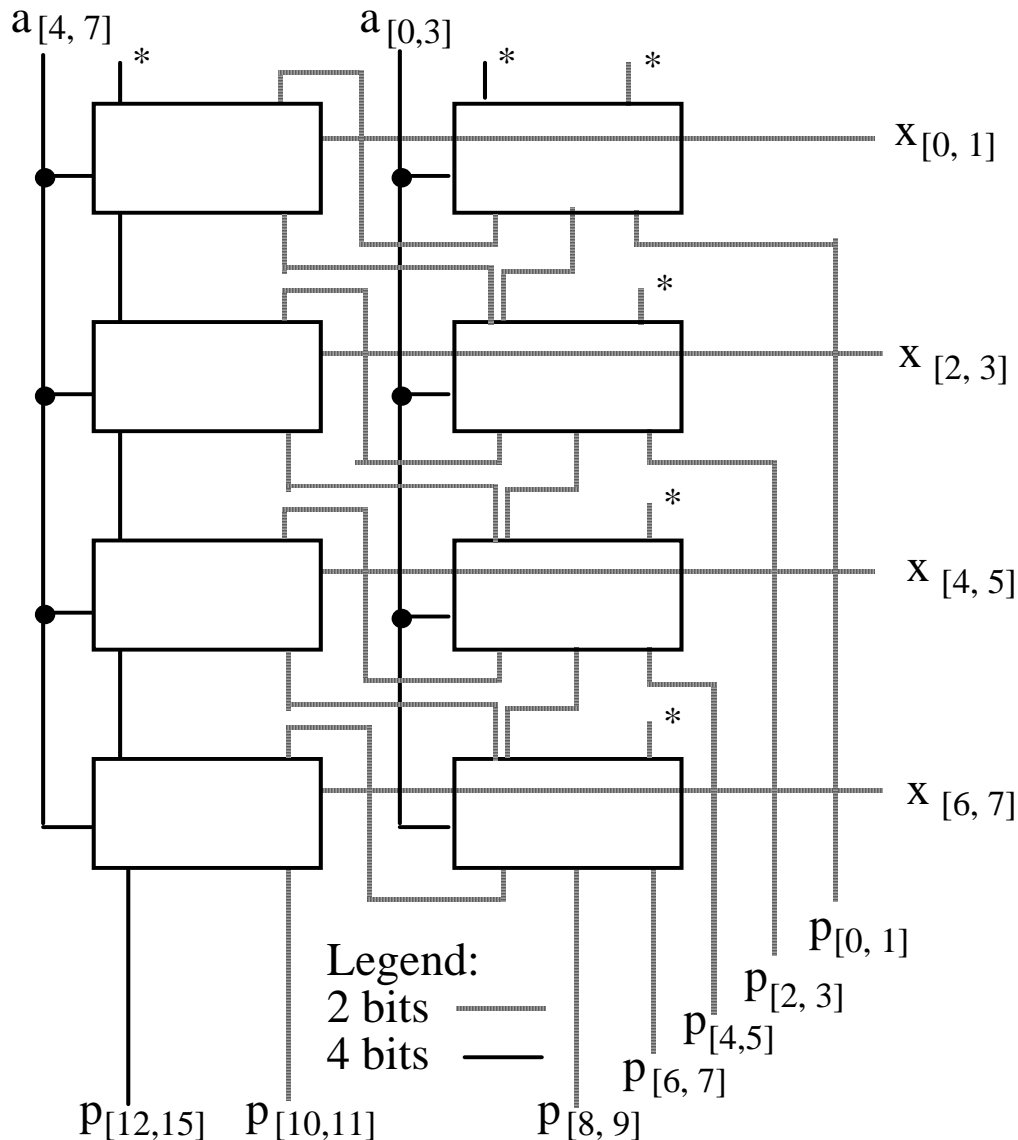


Fig. 12.5 An 8×8 multiplier built of 4×2 AMMs. Inputs marked with an asterisk carry 0s.

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

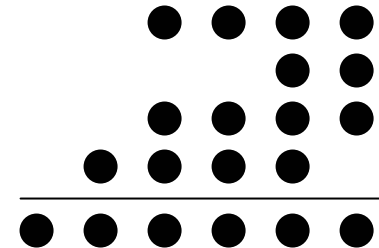
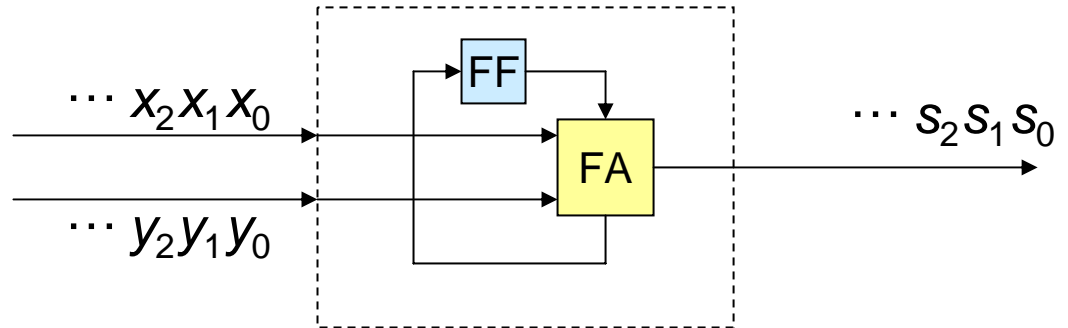


Fig. 12.6 Alternate 8×8 multiplier design based on 4×2 AMMs. Inputs marked with an asterisk carry 0s.

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?

Semisystolic Serial-Parallel Multiplier

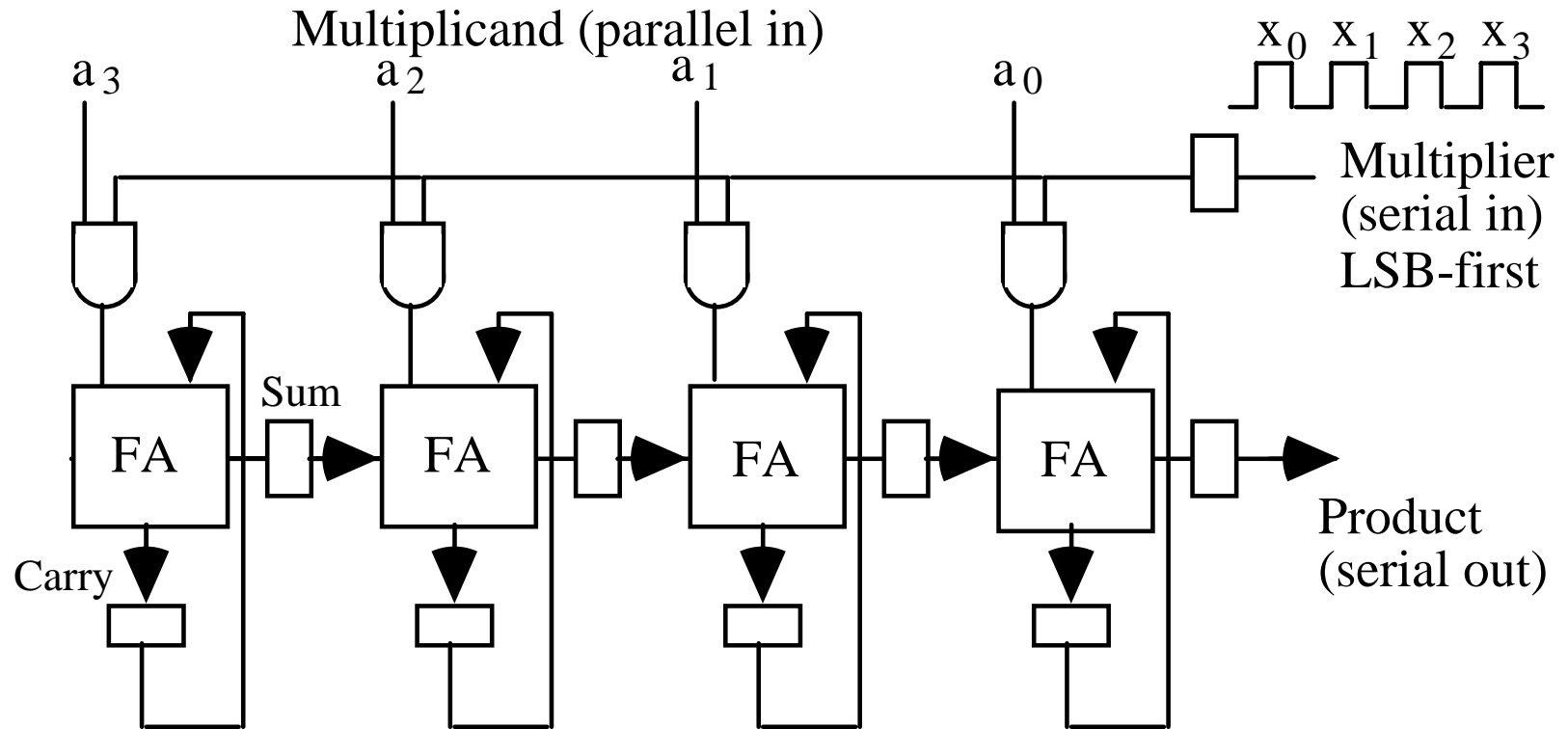


Fig. 12.7 Semi-systolic circuit for 4×4 multiplication in 8 clock cycles.

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

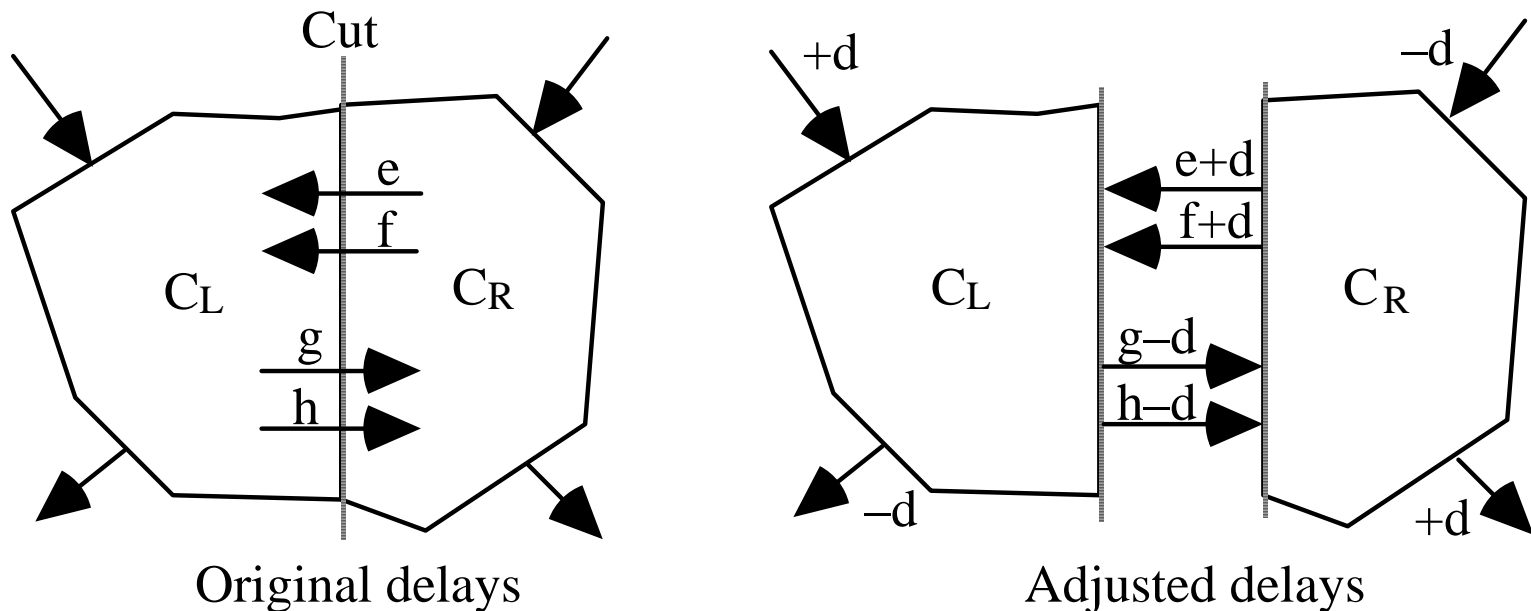
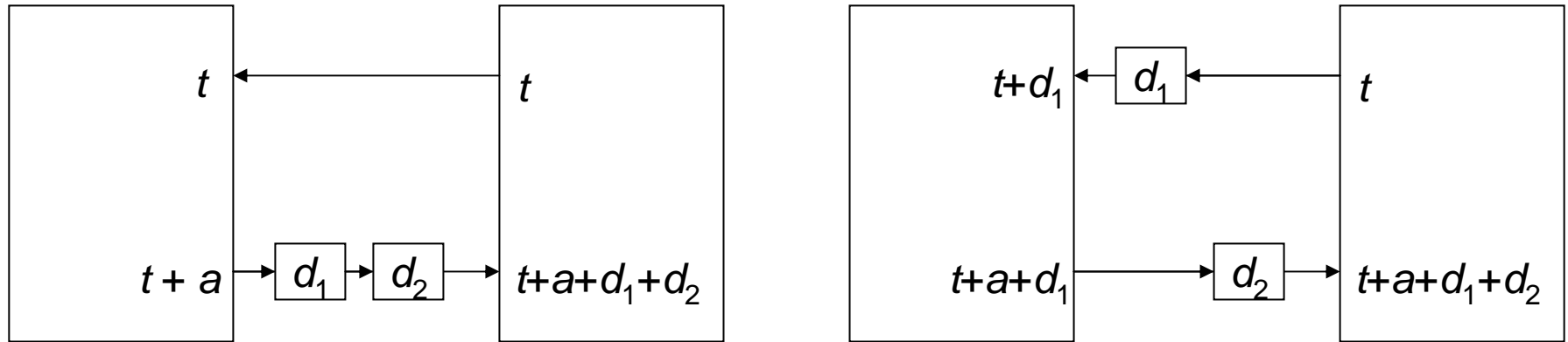


Fig. 12.8 Example of retiming by delaying the inputs to C_L and advancing the outputs from C_L by d units

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

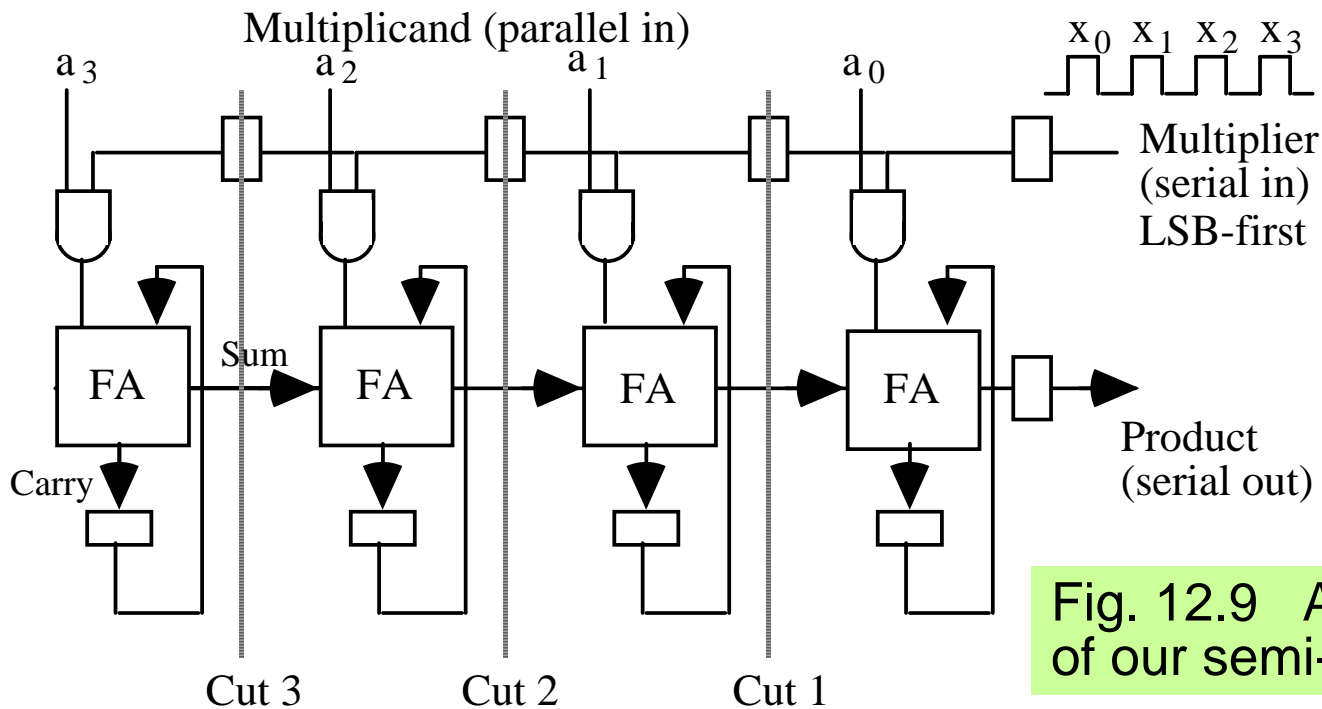
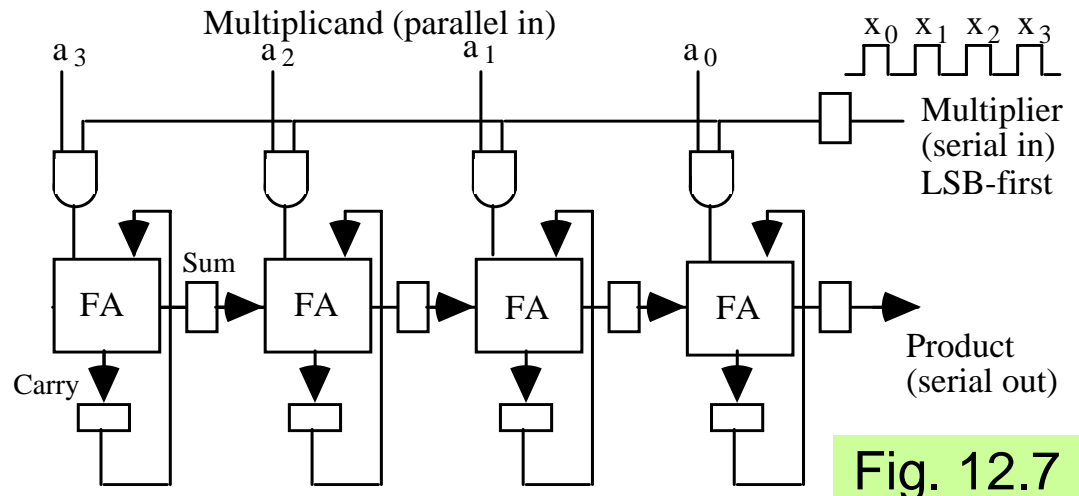
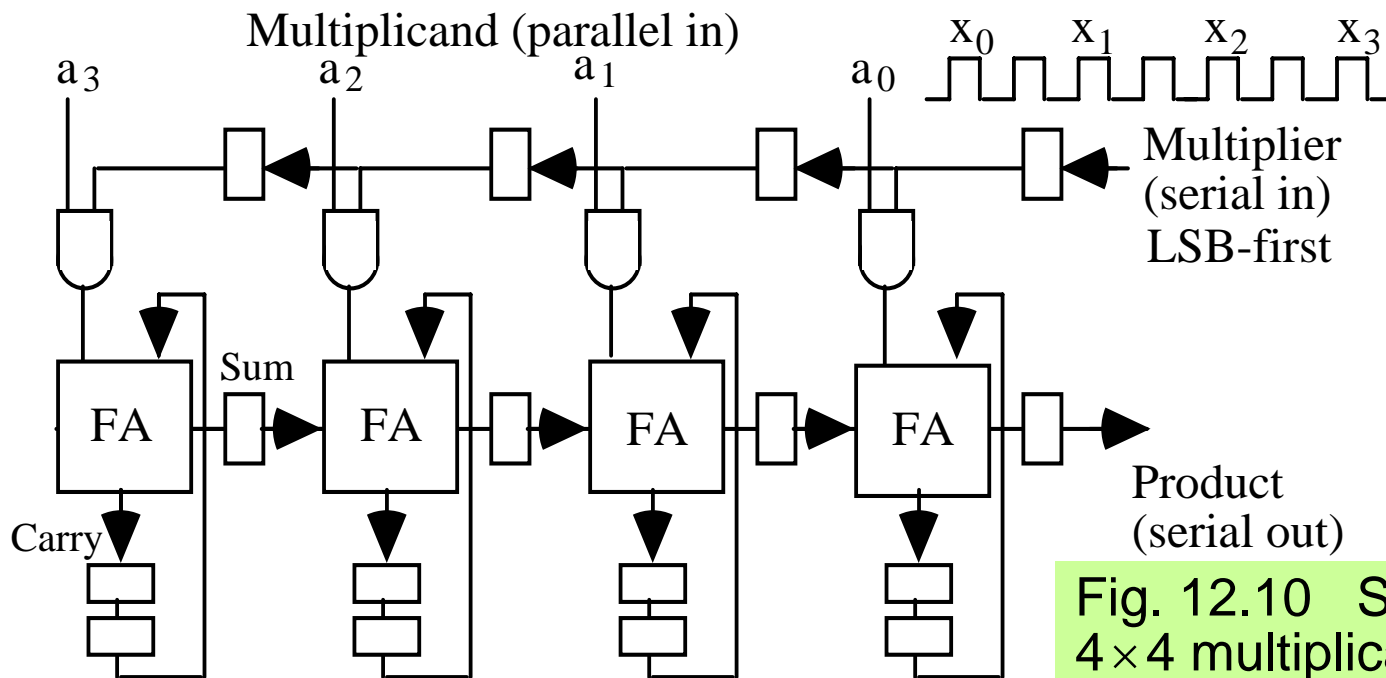
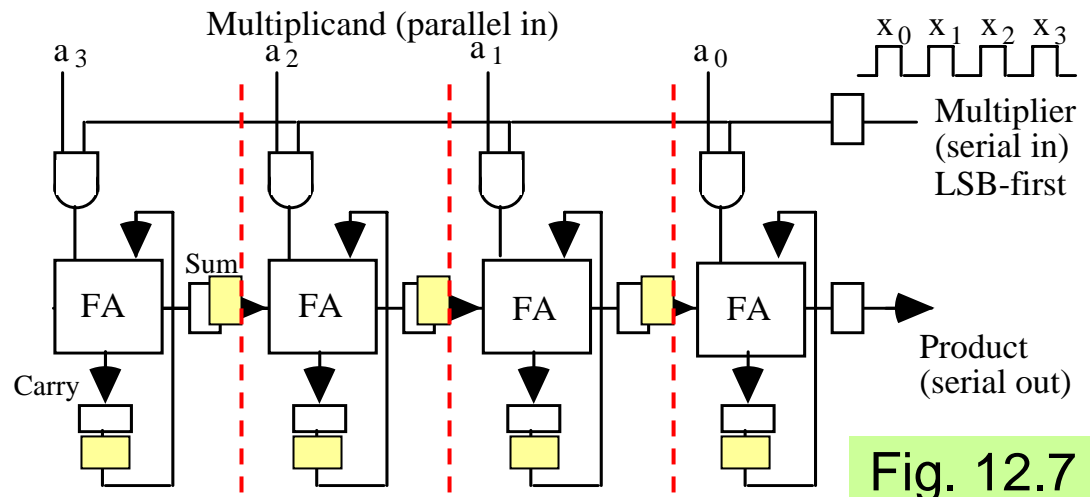


Fig. 12.9 A retimed version of our semi-systolic multiplier.

Deriving a Fully Systolic Multiplier



A Direct Design for a Bit-Serial Multiplier

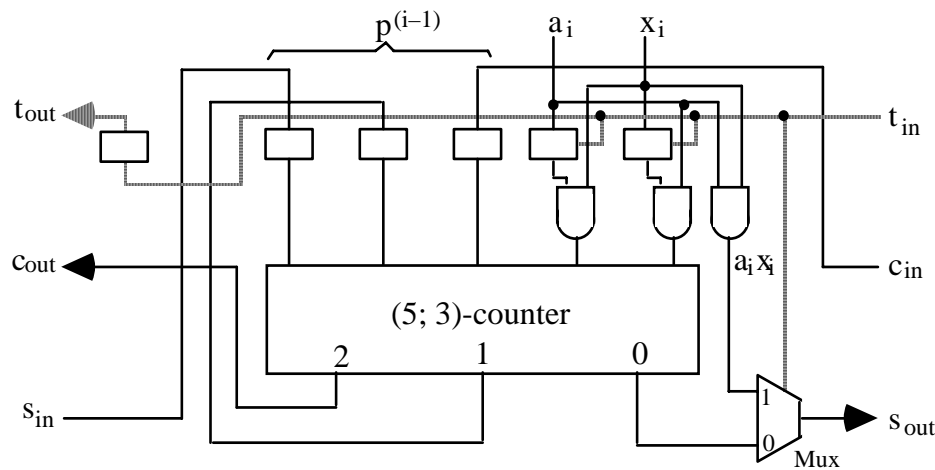


Fig. 12.11 Building block for a latency-free bit-serial multiplier.

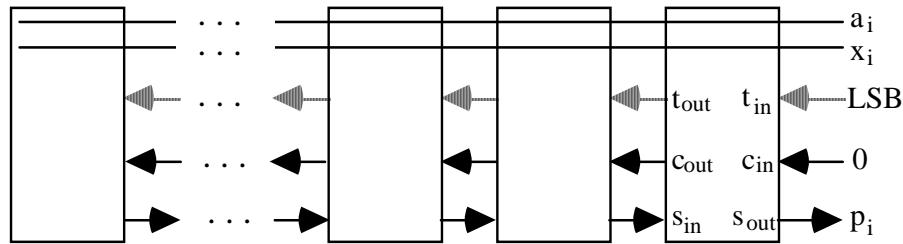
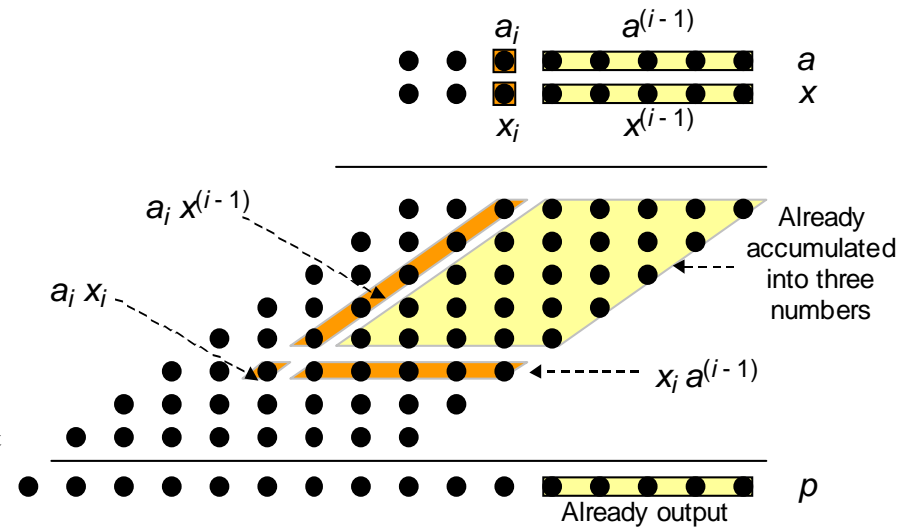
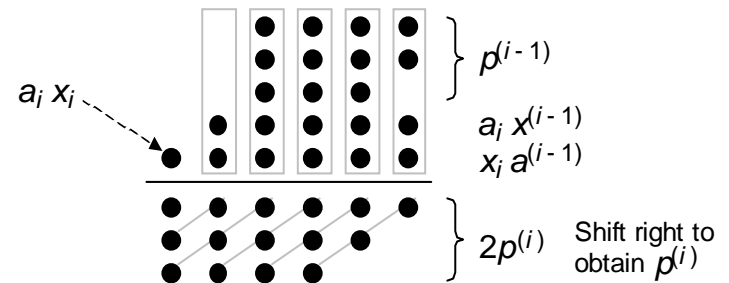


Fig. 12.12 The cellular structure of the bit-serial multiplier based on the cell in Fig. 12.11.



(a) Structure of the bit-matrix



(b) Reduction after each input bit

Fig. 12.13 Bit-serial multiplier design in dot notation.

12.4 Modular Multipliers

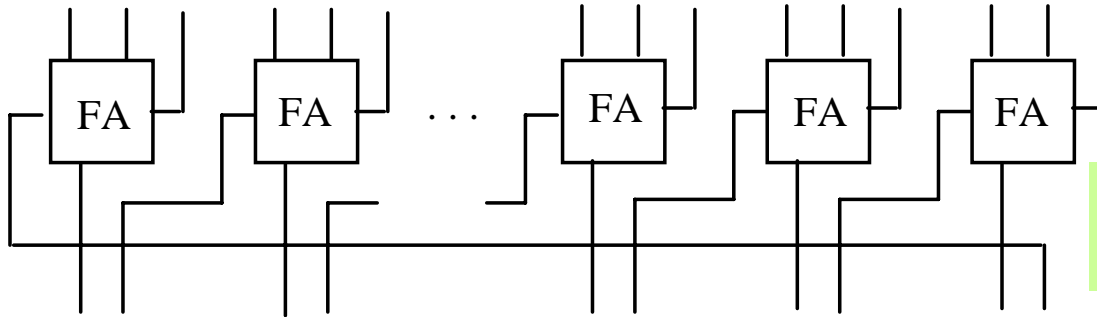


Fig. 12.14 Modulo- $(2^b - 1)$ carry-save adder.

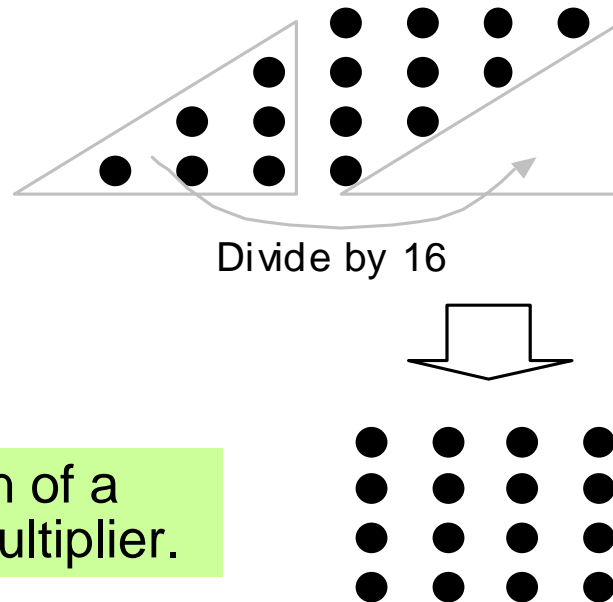
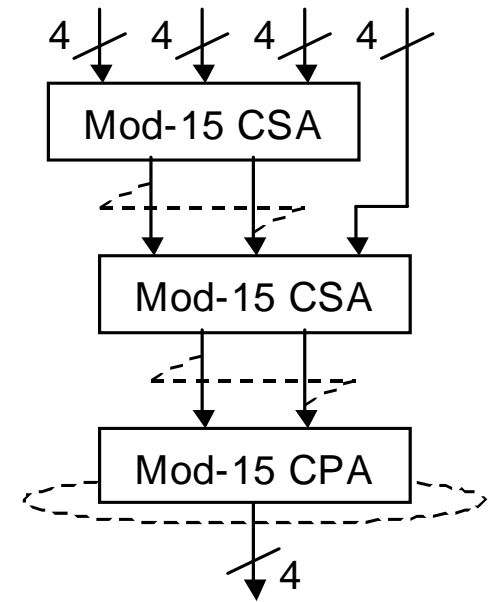


Fig. 12.15 Design of a 4×4 modulo-15 multiplier.



Other Examples of Modular Multiplication

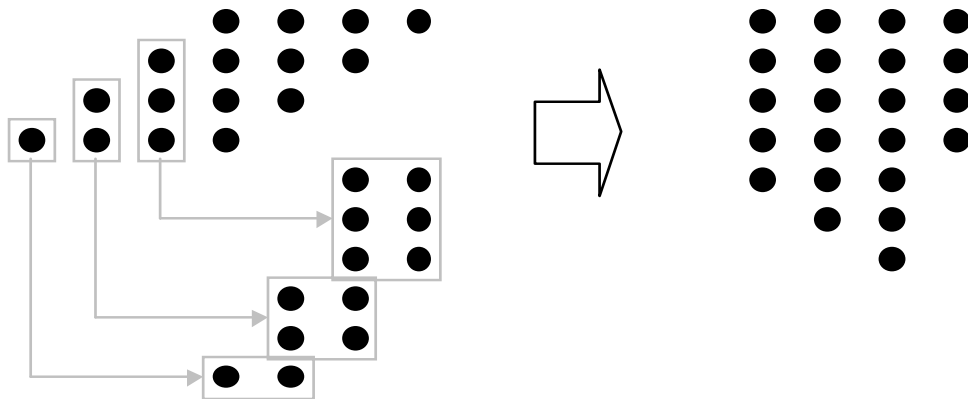


Fig. 12.16 One way to design of a 4×4 modulo-13 multiplier.

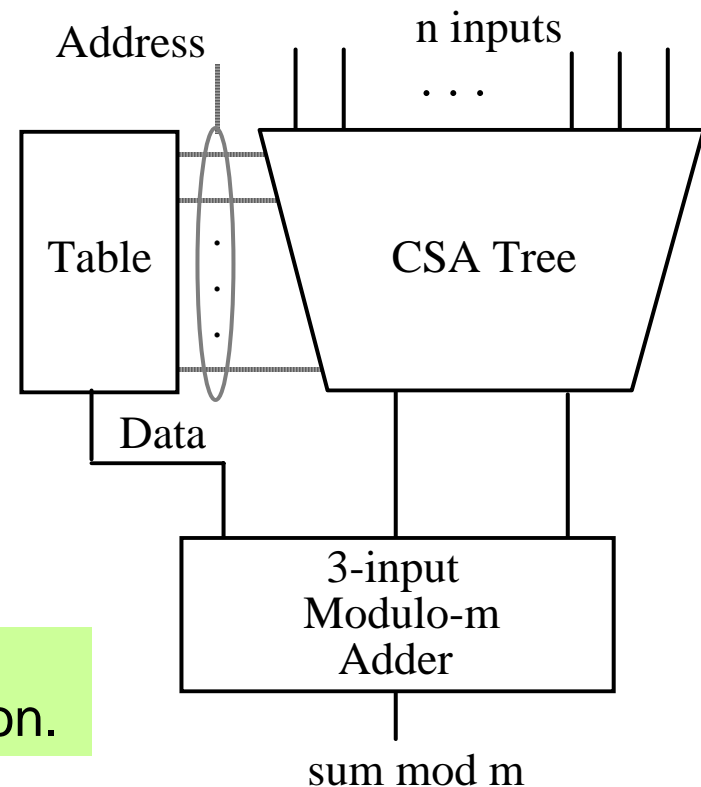


Fig. 12.17 A method for modular multioperand addition.

12.5 The Special Case of Squaring

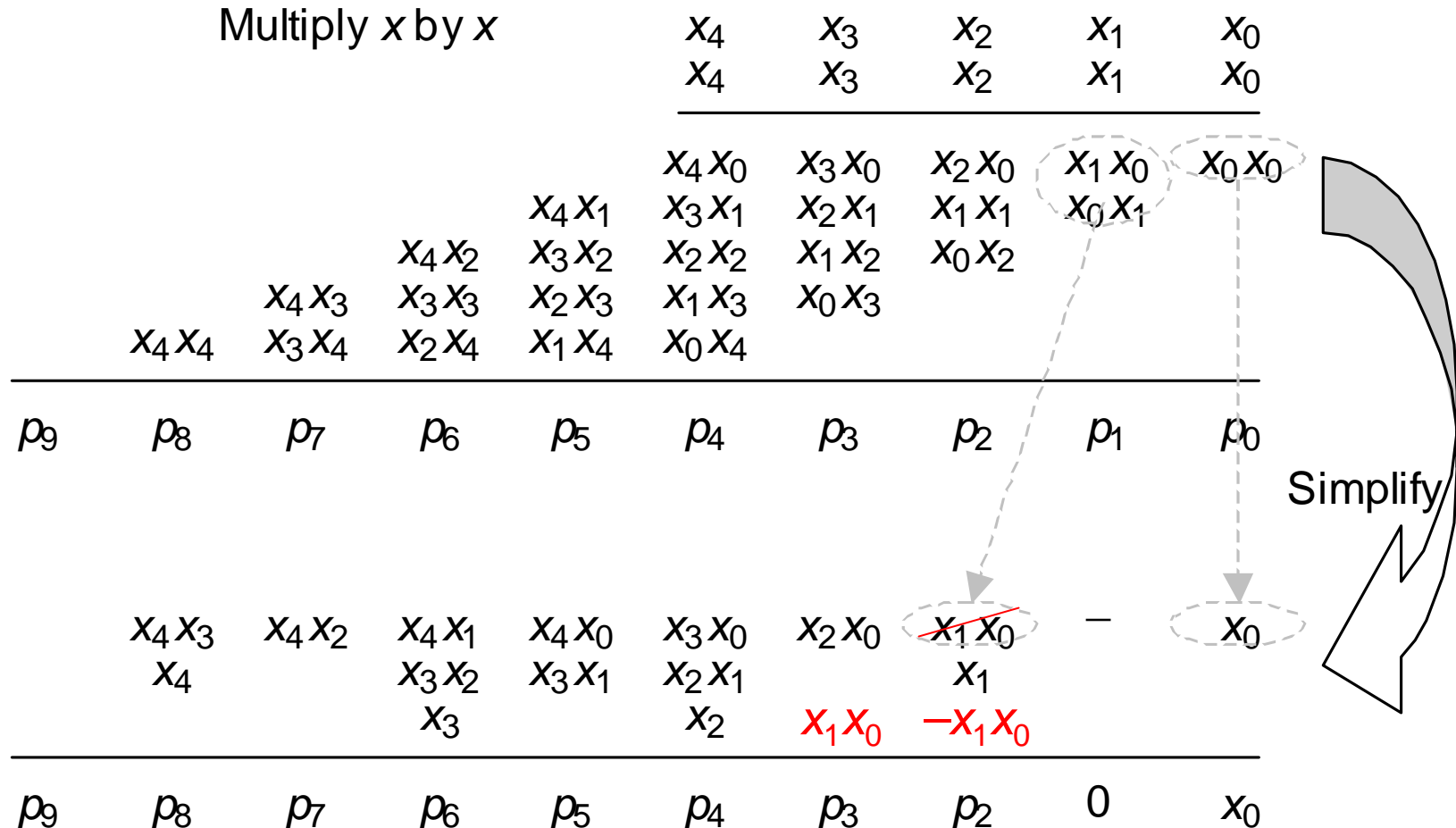
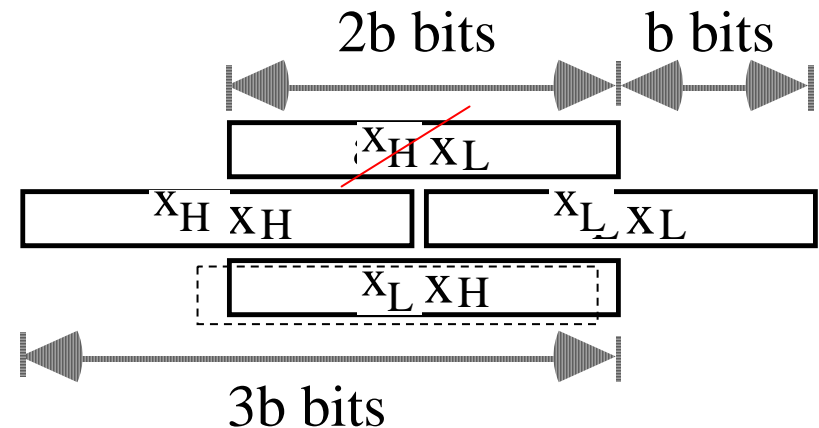
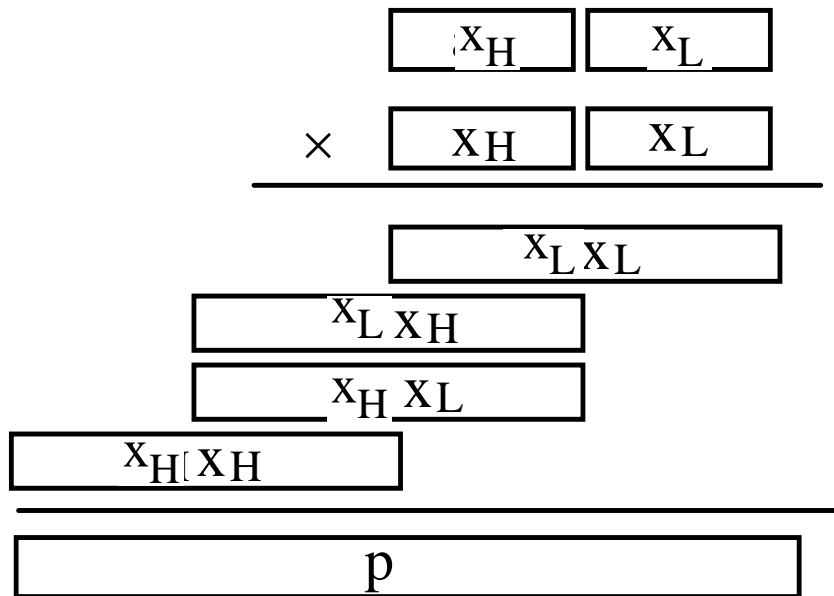


Fig. 12.18 Design of a 5-bit squarer.

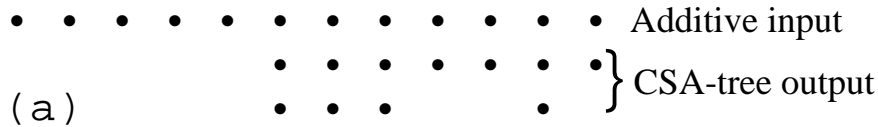
Divide-and-Conquer Squarers

Building wide squarers from narrower ones

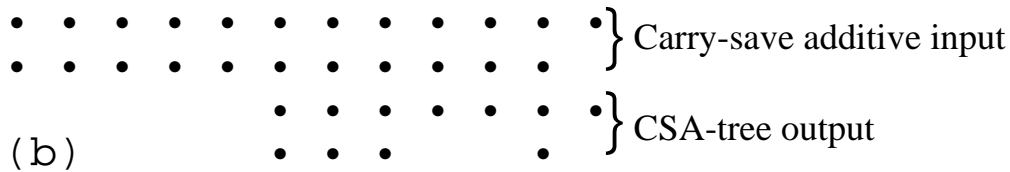


Divide-and-conquer (recursive) strategy for synthesizing a $2b \times 2b$ squarer from $b \times b$ squarers and multiplier.

12.6 Combined Multiply-Add Units



Multiply-add
versus
multiply-accumulate



Multiply-accumulate
units often have wider
additive inputs

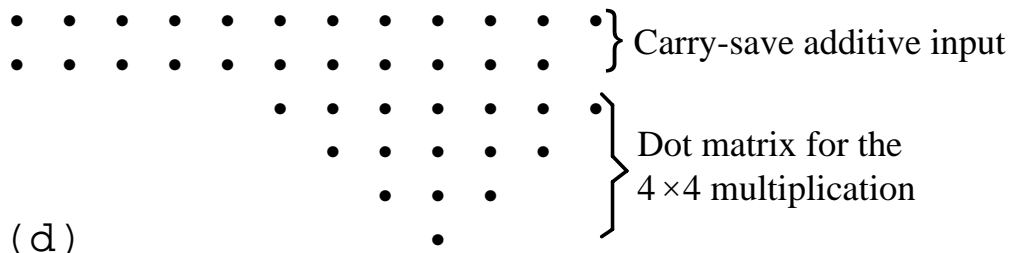
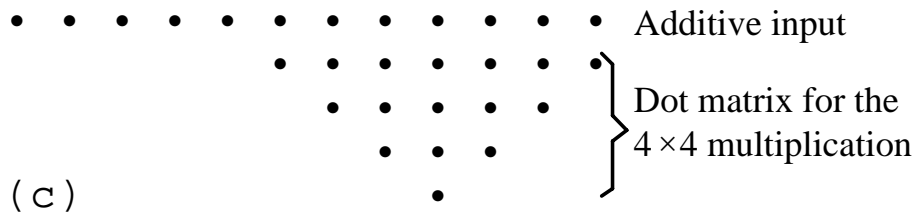


Fig. 12.19 Dot-notation representations of various methods for performing a multiply-add operation in hardware.