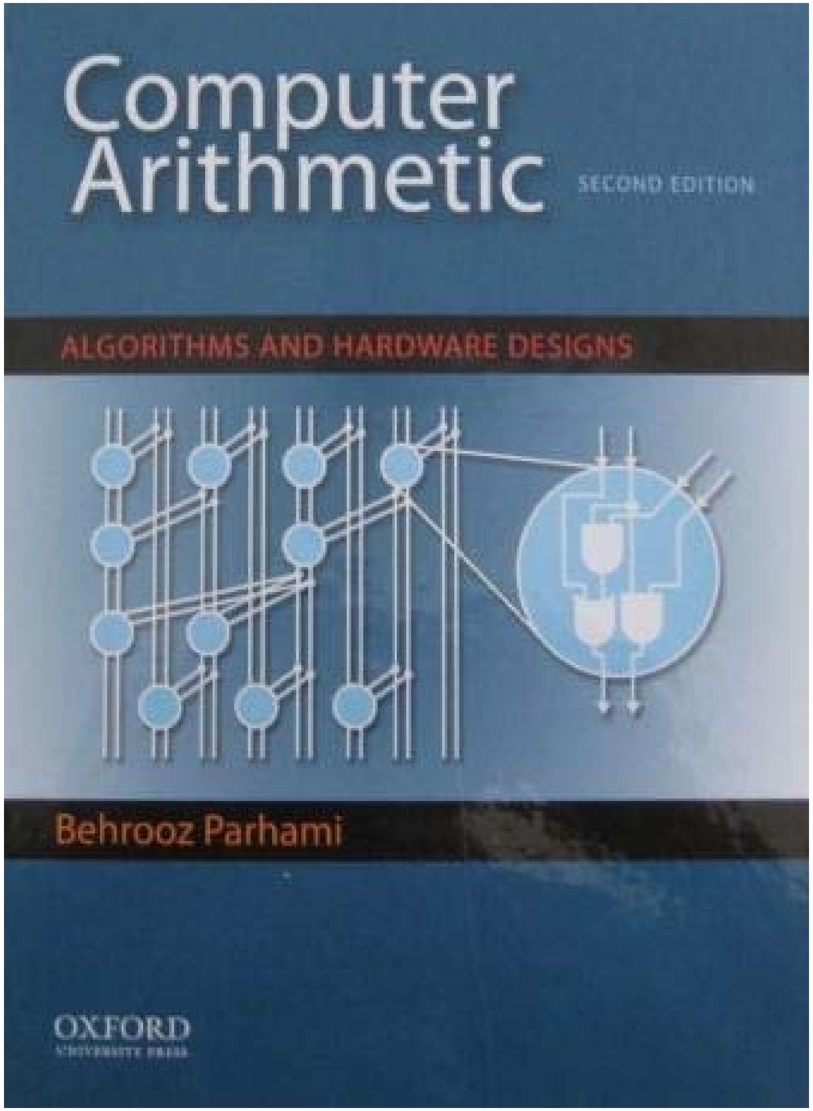


Part II
Addition / Subtraction

	Parts	Chapters
	I. Number Representation	1. Numbers and Arithmetic 2. Representing Signed Numbers 3. Redundant Number Systems 4. Residue Number Systems
Elementary Operations	II. Addition / Subtraction	5. Basic Addition and Counting 6. Carry-Lookahead Adders 7. Variations in Fast Adders 8. Multioperand Addition
	III. Multiplication	9. Basic Multiplication Schemes 10. High-Radix Multipliers 11. Tree and Array Multipliers 12. Variations in Multipliers
	IV. Division	13. Basic Division Schemes 14. High-Radix Dividers 15. Variations in Dividers 16. Division by Convergence
	V. Real Arithmetic	17. Floating-Point Reperesentations 18. Floating-Point Operations 19. Errors and Error Control 20. Precise and Certifiable Arithmetic
	VI. Function Evaluation	21. Square-Rooting Methods 22. The CORDIC Algorithms 23. Variations in Function Evaluation 24. Arithmetic by Table Lookup
	VII. Implementation Topics	25. High-Throughput Arithmetic 26. Low-Power Arithmetic 27. Fault-Tolerant Arithmetic 28. Reconfigurable Arithmetic

Appendix: Past, Present, and Future

About This Presentation

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

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

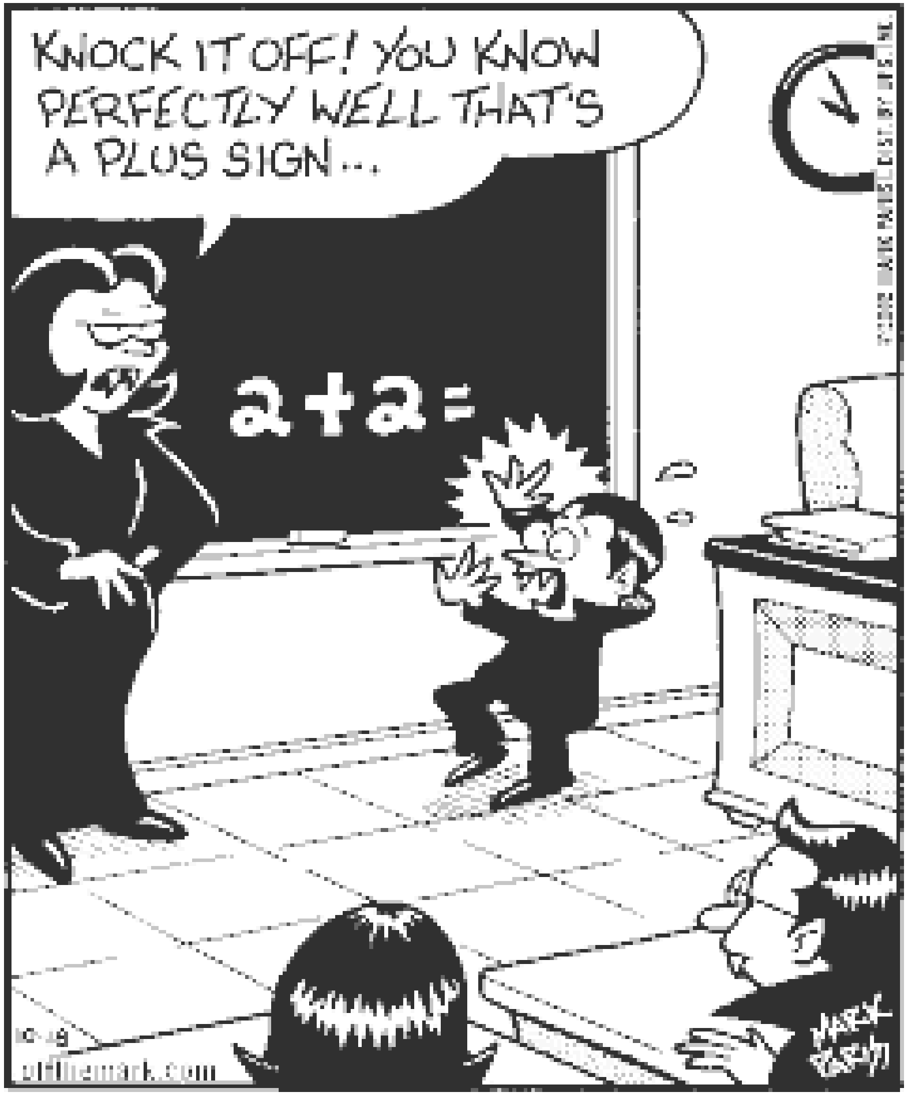
II Addition / Subtraction

Review addition schemes and various speedup methods

- Addition is a key op (in itself, and as a building block)
- Subtraction = negation + addition
- Carry propagation speedup: lookahead, skip, select, …
- Two-operand versus multioperand addition

Topics in This Part	
Chapter 5	Basic Addition and Counting
Chapter 6	Carry-Lookahead Adders
Chapter 7	Variations in Fast Adder
Chapter 8	Multioperand Addition

"You can't add apples
and oranges, son; only the
government can do that."



5 Basic Addition and Counting

Chapter Goals

Study the design of ripple-carry adders, discuss why their latency is unacceptable, and set the foundation for faster adders

Chapter Highlights

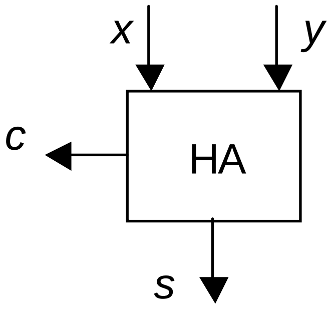
Full adders are versatile building blocks
Longest carry chain on average: $\log_2 k$ bits
Fast asynchronous adders are simple
Counting is relatively easy to speed up
Key part of a fast adder is its carry network

Basic Addition and Counting: Topics

Topics in This Chapter
5.1 Bit-Serial and Ripple-Carry Adders
5.2 Conditions and Exceptions
5.3 Analysis of Carry Propagation
5.4 Carry Completion Detection
5.5 Addition of a Constant
5.6 Manchester Carry Chains and Adders

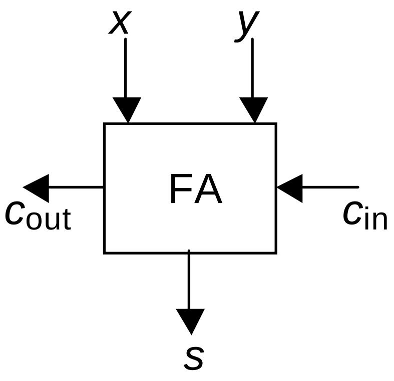
5.1 Bit-Serial and Ripple-Carry Adders

Inputs		Outputs	
x	y	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



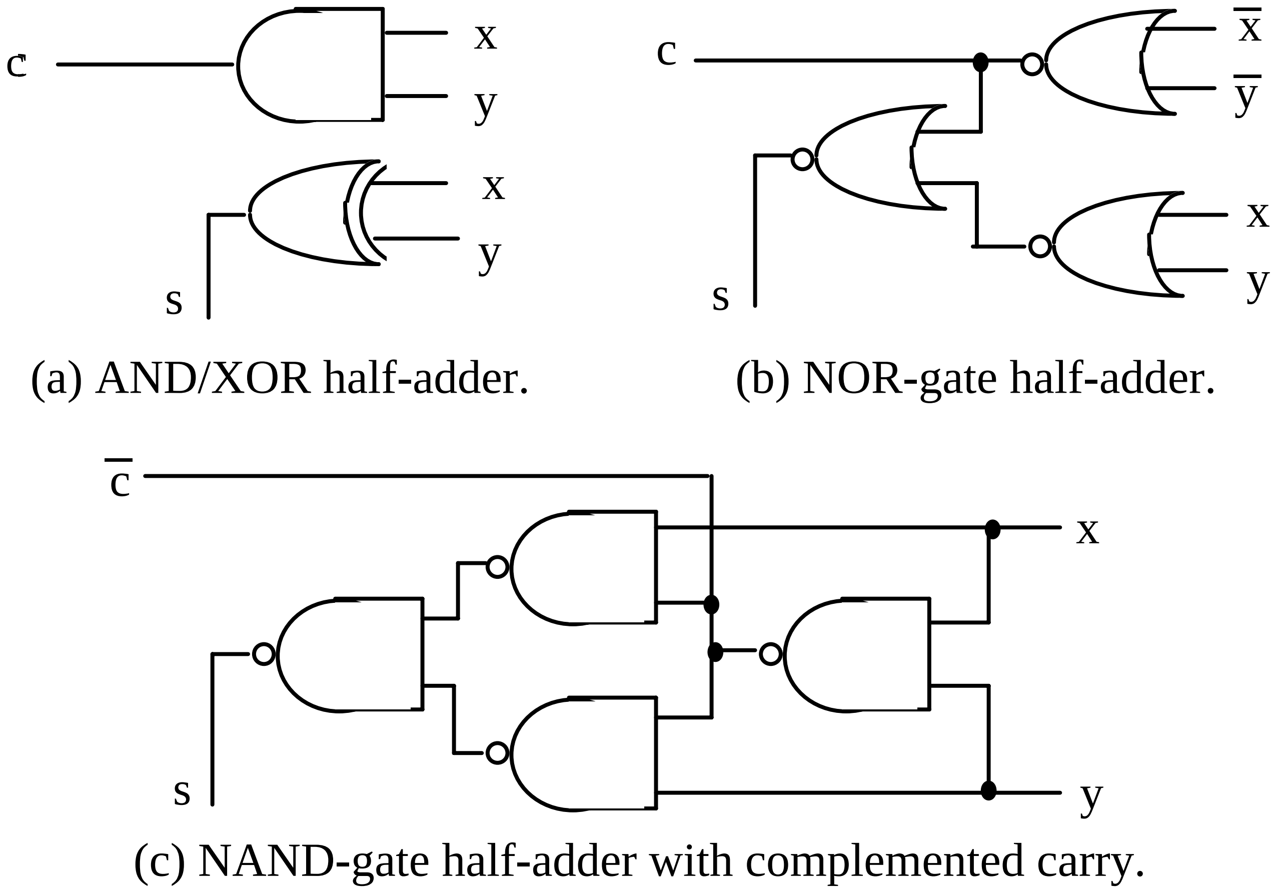
Half-adder (HA): Truth table and block diagram

Inputs			Outputs	
x	y	c_{in}	c_{out}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



Full-adder (FA): Truth table and block diagram

Half-Adder Implementations



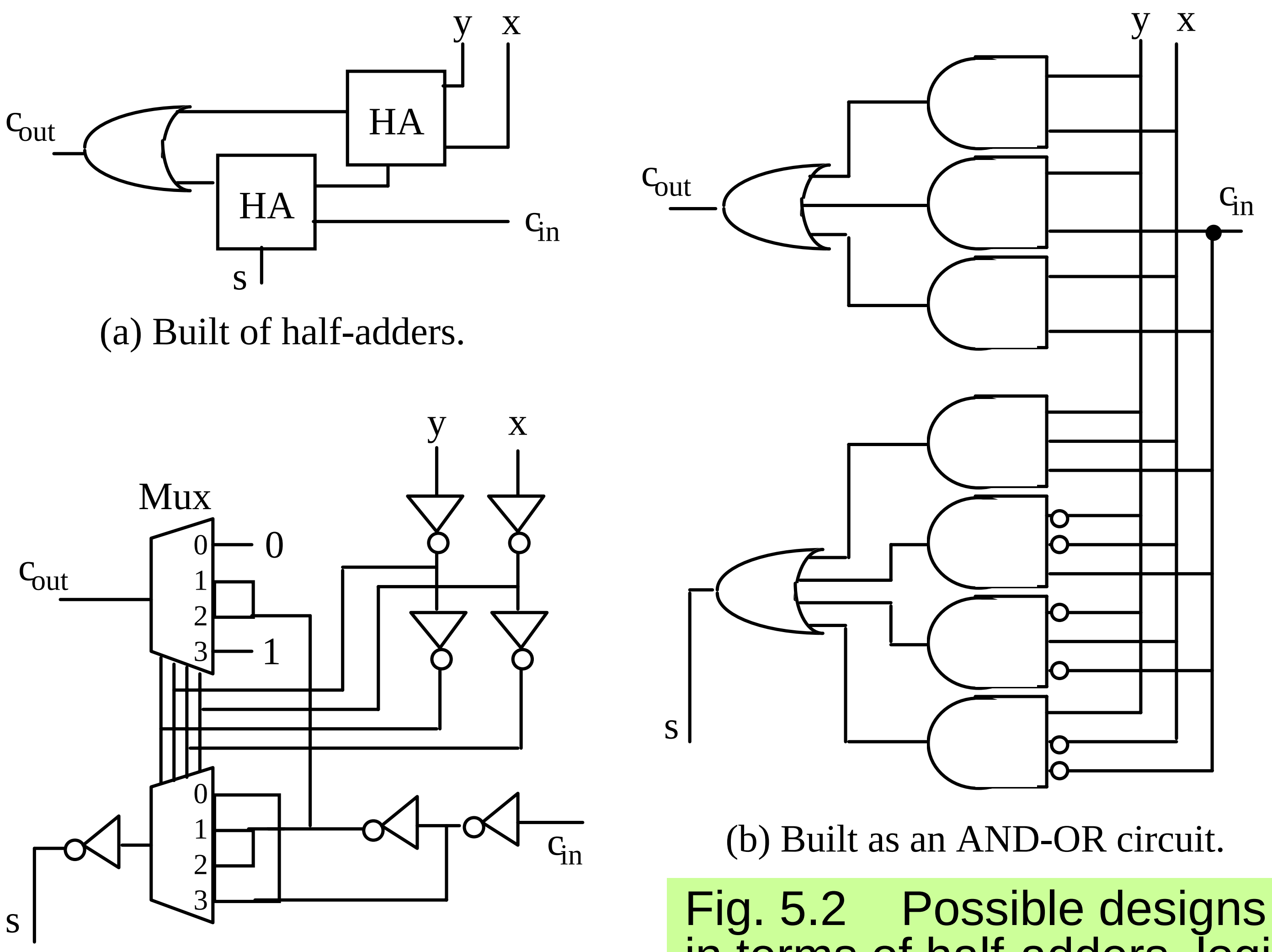
(a) AND/XOR half-adder.

(b) NOR-gate half-adder.

(c) NAND-gate half-adder with complemented carry.

Fig. 5.1 Three implementations of a half-adder.

Full-Adder Implementations



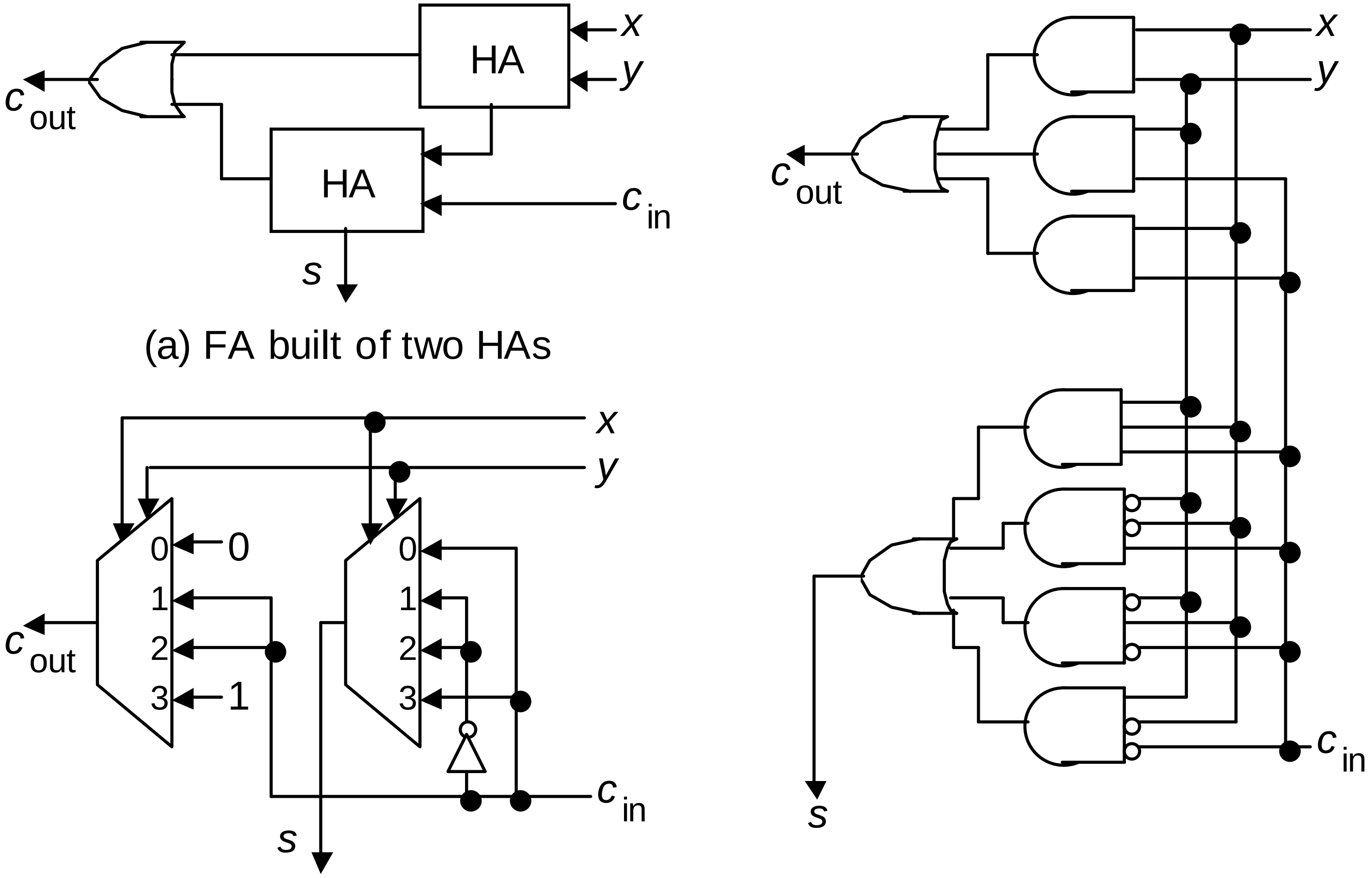
(a) Built of half-adders.

(b) Built as an AND-OR circuit.

(c) Suitable for CMOS realization.

Fig. 5.2 Possible designs for a full-adder in terms of half-adders, logic gates, and CMOS transmission gates.

Full-Adder Implementations



(a) FA built of two HAs

(b) CMOS mux-based FA

(c) Two-level AND-OR FA

Fig. 5.2 (alternate version) Possible designs for a full-adder in terms of half-adders, logic gates, and CMOS transmission gates.

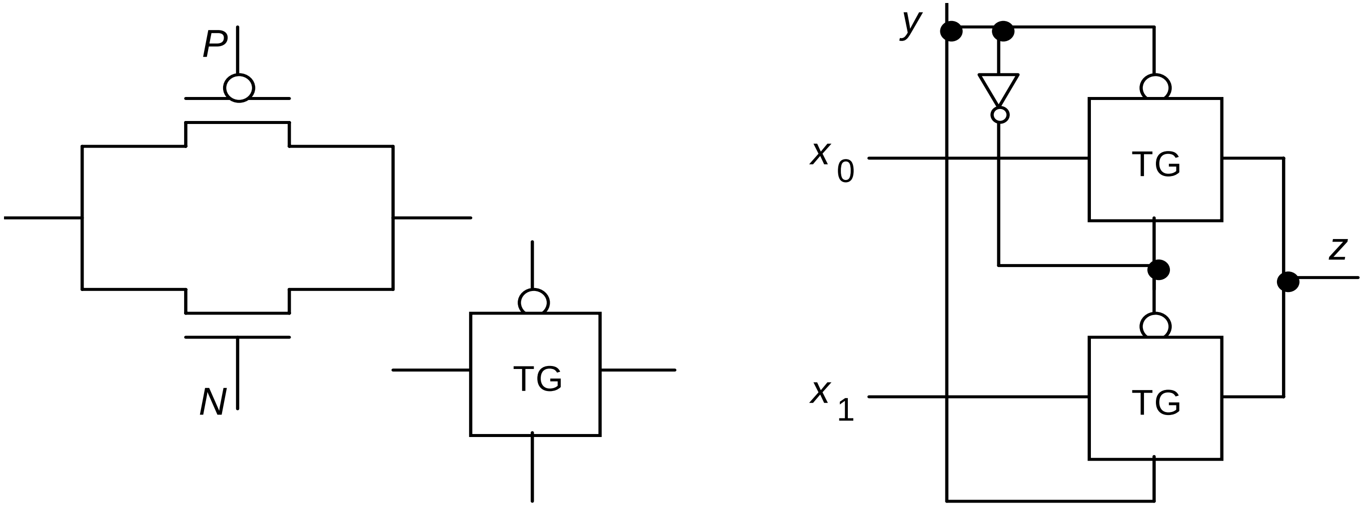
Some Full-Adder Details

Logic equations for a full-adder:

$$s = x \oplus y \oplus c_{\text{in}} \qquad \text{(odd parity function)}$$

$$= x y c_{\text{in}} \vee x' y' c_{\text{in}} \vee x' y c_{\text{in}}' \vee x y' c_{\text{in}}'$$

$$c_{\text{out}} = x y \vee x c_{\text{in}} \vee y c_{\text{in}} \qquad \text{(majority function)}$$



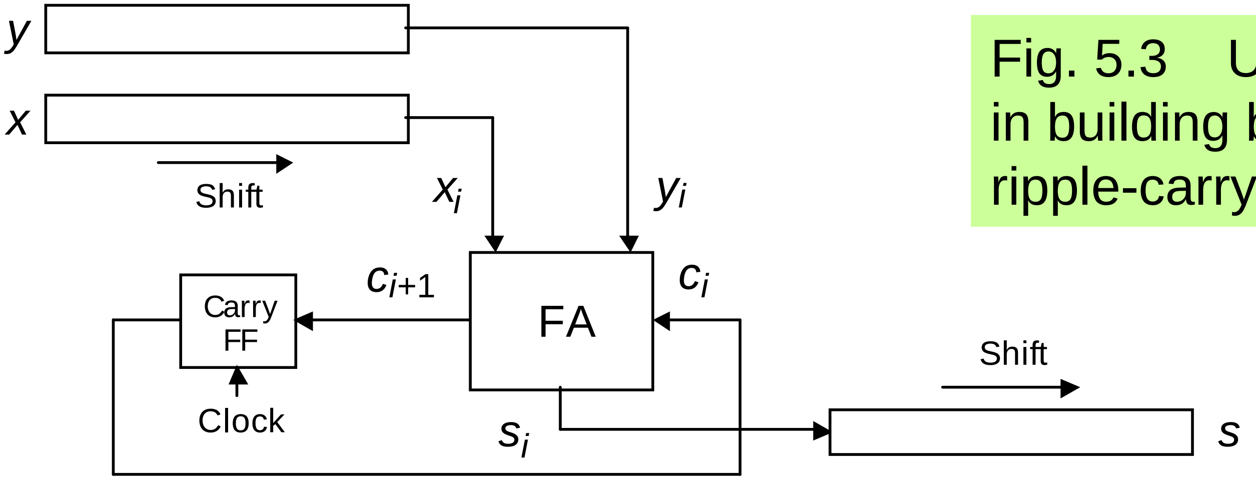
(a) CMOS transmission gate: circuit and symbol

(b) Two-input mux built of two transmission gates

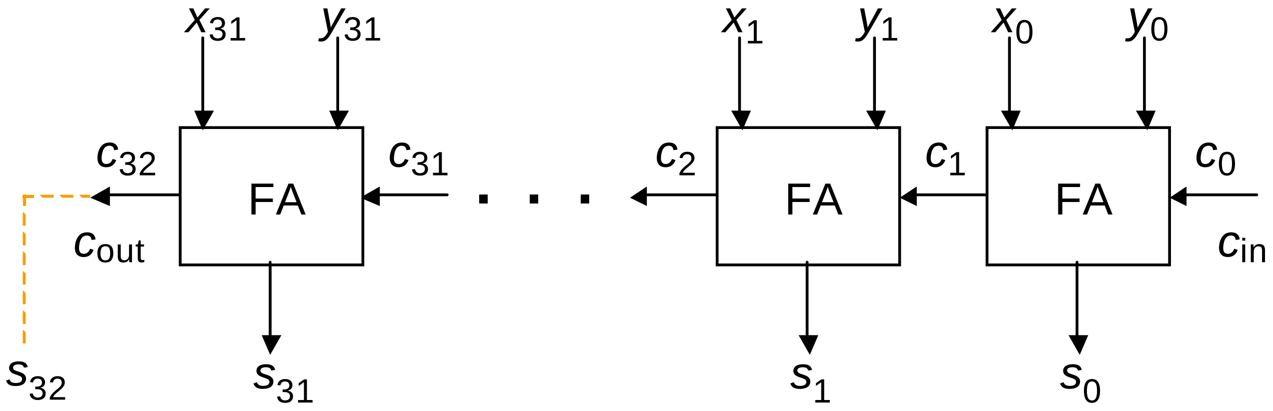
CMOS transmission gate and its use in a 2-to-1 mux.

Simple Adders Built of Full-Adders

Fig. 5.3 Using full-adders in building bit-serial and ripple-carry adders.



(a) Bit-serial adder.



(b) Ripple-carry adder.

VLSI Layout of a Ripple-Carry Adder

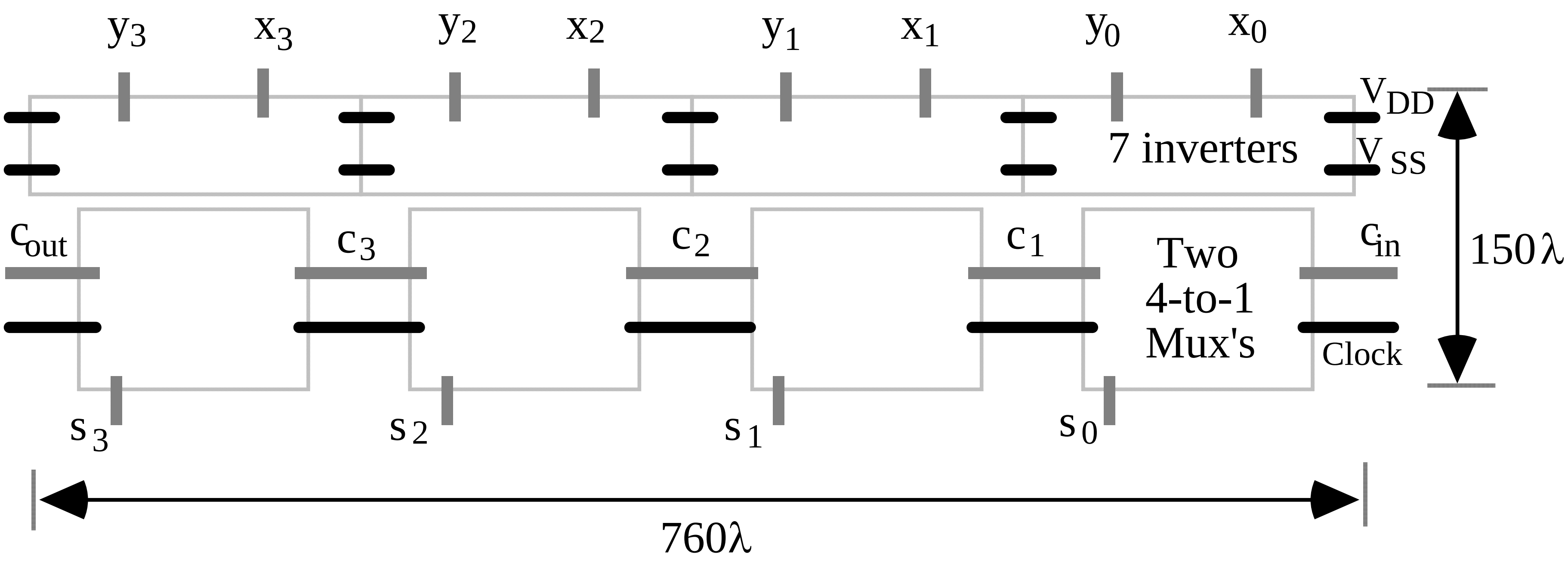


Fig. 5.4 The layout of a 4-bit ripple-carry adder in CMOS implementation [Puck94].

Critical Path Through a Ripple-Carry Adder

$$T_{\text{ripple-add}} = T_{\text{FA}}(x,y \to c_{\text{out}}) + (k-2) \times T_{\text{FA}}(c_{\text{in}} \to \text{cout}) + T_{\text{FA}}(c_{\text{in}} \to s)$$

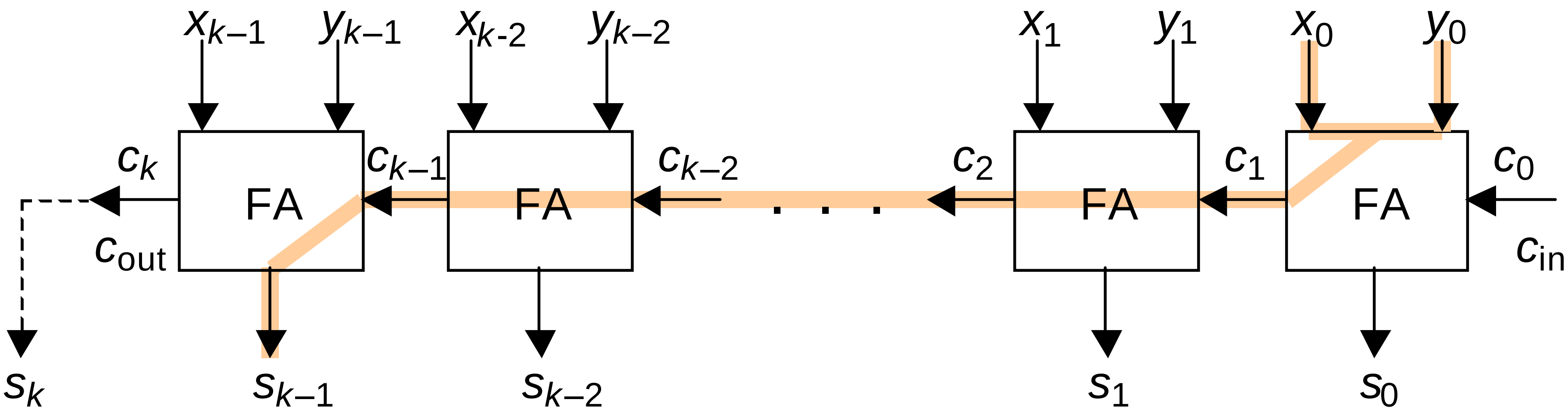


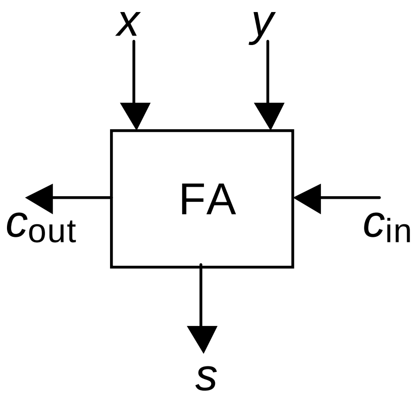
Fig. 5.5 Critical path in a *k*-bit ripple-carry adder.

Binary Adders as Versatile Building Blocks

Set one input to 0: c_{out} = AND of other inputs

Set one input to 1: c_{out} = OR of other inputs

Set one input to 0
and another to 1: s = NOT of third input



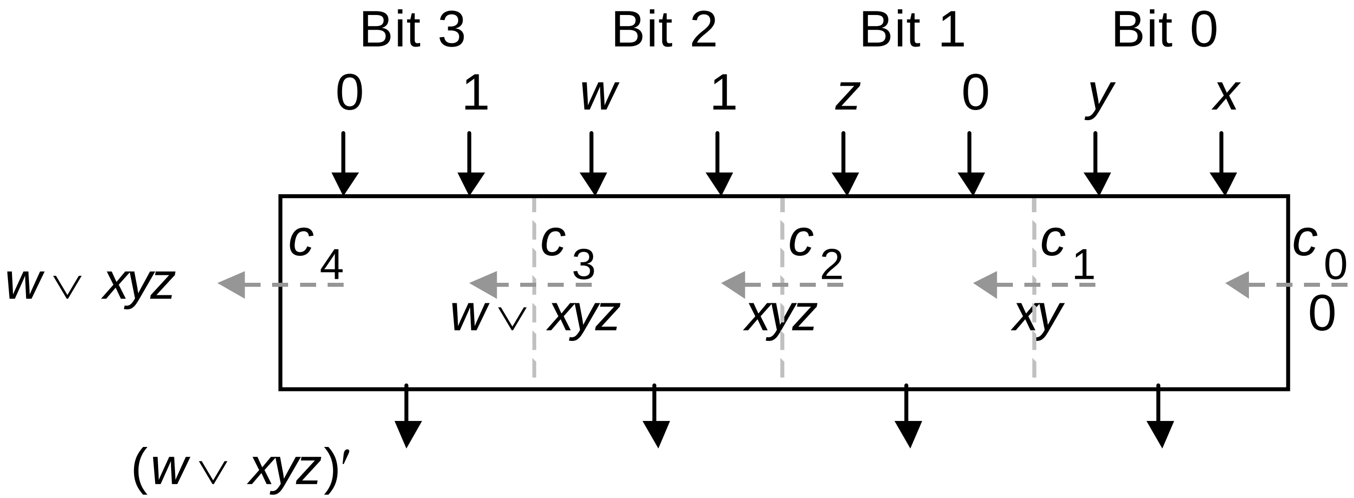


Fig. 5.6 Four-bit binary adder used to realize the logic function $f = w + xyz$ and its complement.

5.2 Conditions and Exceptions

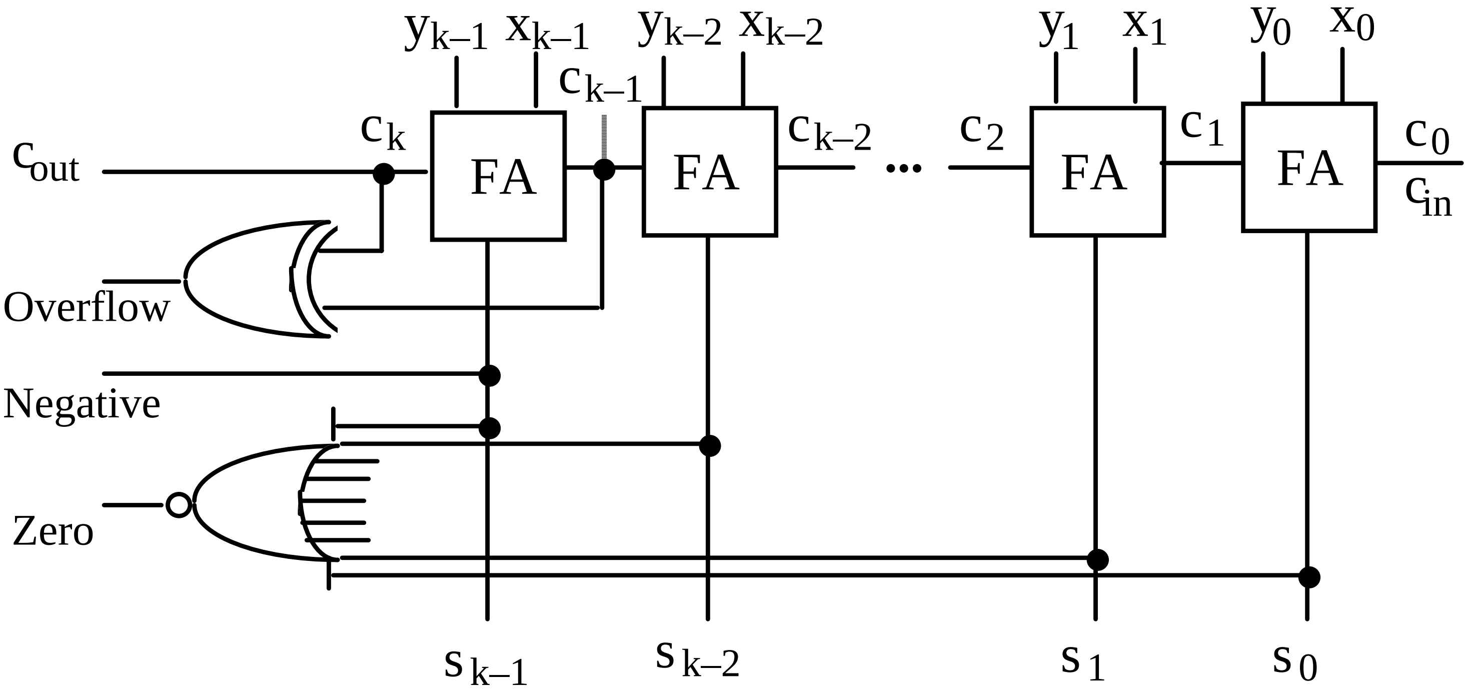


Fig. 5.7 Two's-complement adder with provisions for detecting conditions and exceptions.

$$\text{overflow}_{\text{2's-compl}} = x_{k-1}\, y_{k-1}\, s_{k-1}' \vee x_{k-1}'\, y_{k-1}'\, s_{k-1}$$

$$\text{overflow}_{\text{2's-compl}} = c_k \oplus c_{k-1} = c_k\, c_{k-1}' \vee c_k'\, c_{k-1}$$

Saturating Adders

Saturating (saturation) arithmetic:

When a result’s magnitude is too large, do not wrap around; rather, provide the most positive or the most negative value that is representable in the number format

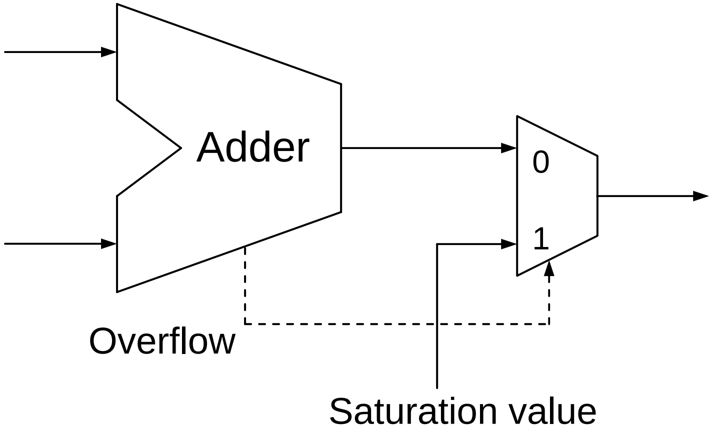
Example – In 8-bit 2’s-complement format, we have:
120 + 26 → 18 (wraparound);   120 $+_{\text{sat}}$ 26 → 127 (saturating)

Saturating arithmetic in desirable in many DSP applications

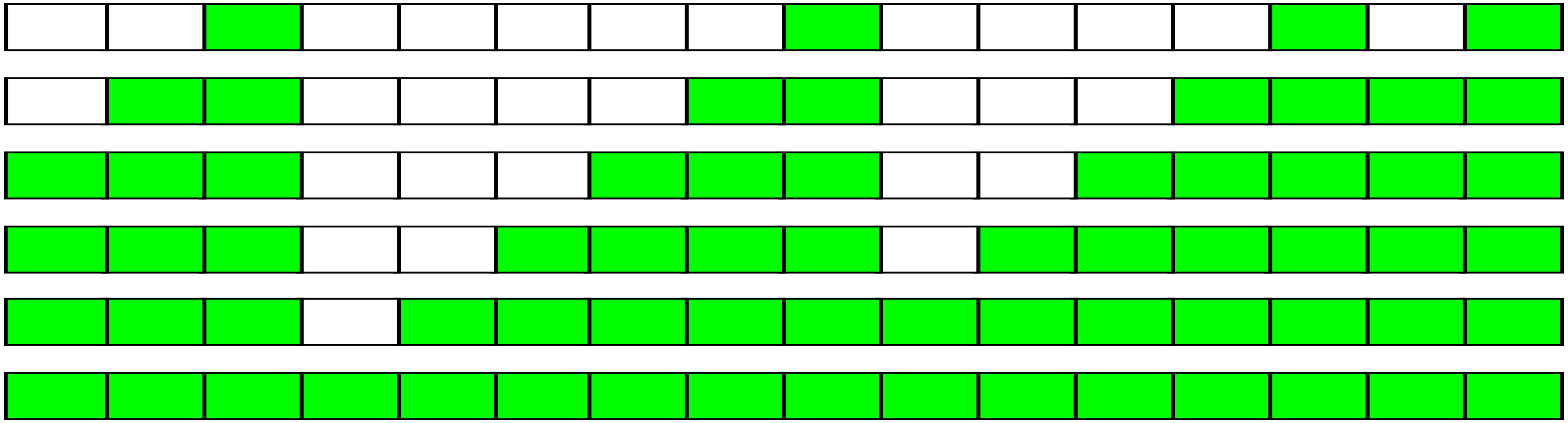
Designing saturating adders

Unsigned (quite easy)

Signed (only slightly harder)



5.3 Analysis of Carry Propagation



Bit positions

```
      15  14  13  12   11  10   9   8    7   6   5   4    3   2   1   0
      -----------      -----------      -----------      -----------
       1   0   1   1    0   1   1   0    0   1   1   0    1   1   1   0

c_out  0   1   0   1    1   0   0   1    1   1   0   0    0   0   1   1  c_in
    \_____________/\___________________________/     \___________/\_____/
           4                     6                          3        2
```

Carry chains and their lengths

Fig. 5.8 Example addition and its carry propagation chains.

Using Probability to Analyze Carry Propagation

Given binary numbers with random bits, for each position *i* we have

Probability of carry generation	=	¼	(both 1s)
Probability of carry annihilation	=	¼	(both 0s)
Probability of carry propagation	=	½	(different)

Probability that carry generated at position *i* propagates through position *j* – 1 and stops at position *j* (*j* > *i*)

$$2^{-(j-1-i)} \times 1/2 = 2^{-(j-i)}$$

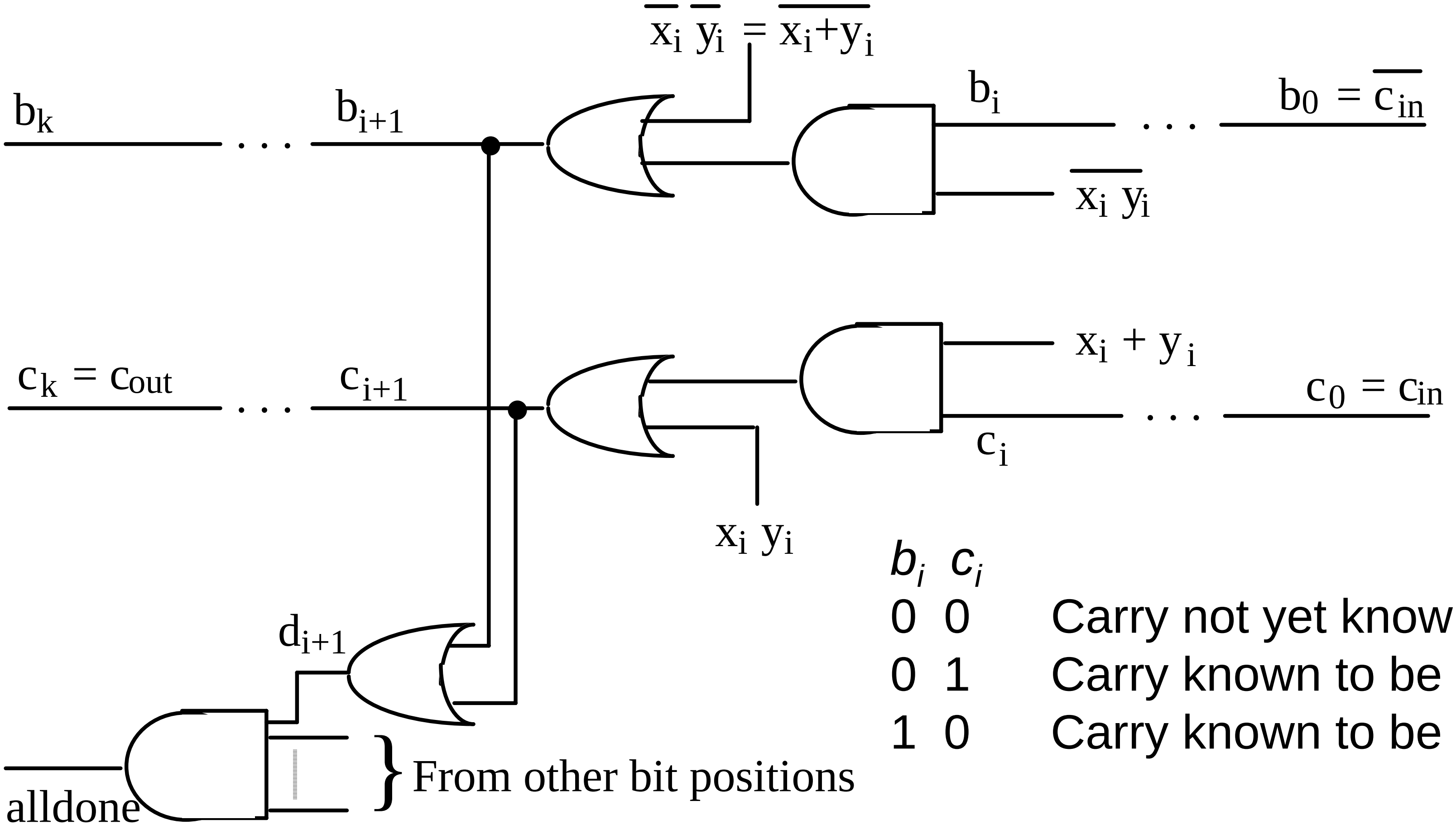
Expected length of the carry chain that starts at position *i*

$$2 - 2^{-(k-i-1)}$$

Average length of the longest carry chain in *k*-bit addition is strictly less than $\log_2 k$; it is $\log_2(1.25k)$ per experimental results

Analogy: Expected number when rolling one die is 3.5; if one rolls many dice, the expected value of the largest number shown grows

5.4 Carry Completion Detection



b_i	c_i	
0	0	Carry not yet known
0	1	Carry known to be 1
1	0	Carry known to be 0

Fig. 5.9 The carry network of an adder with two-rail carries and carry completion detection logic.

5.5 Addition of a Constant: Counters

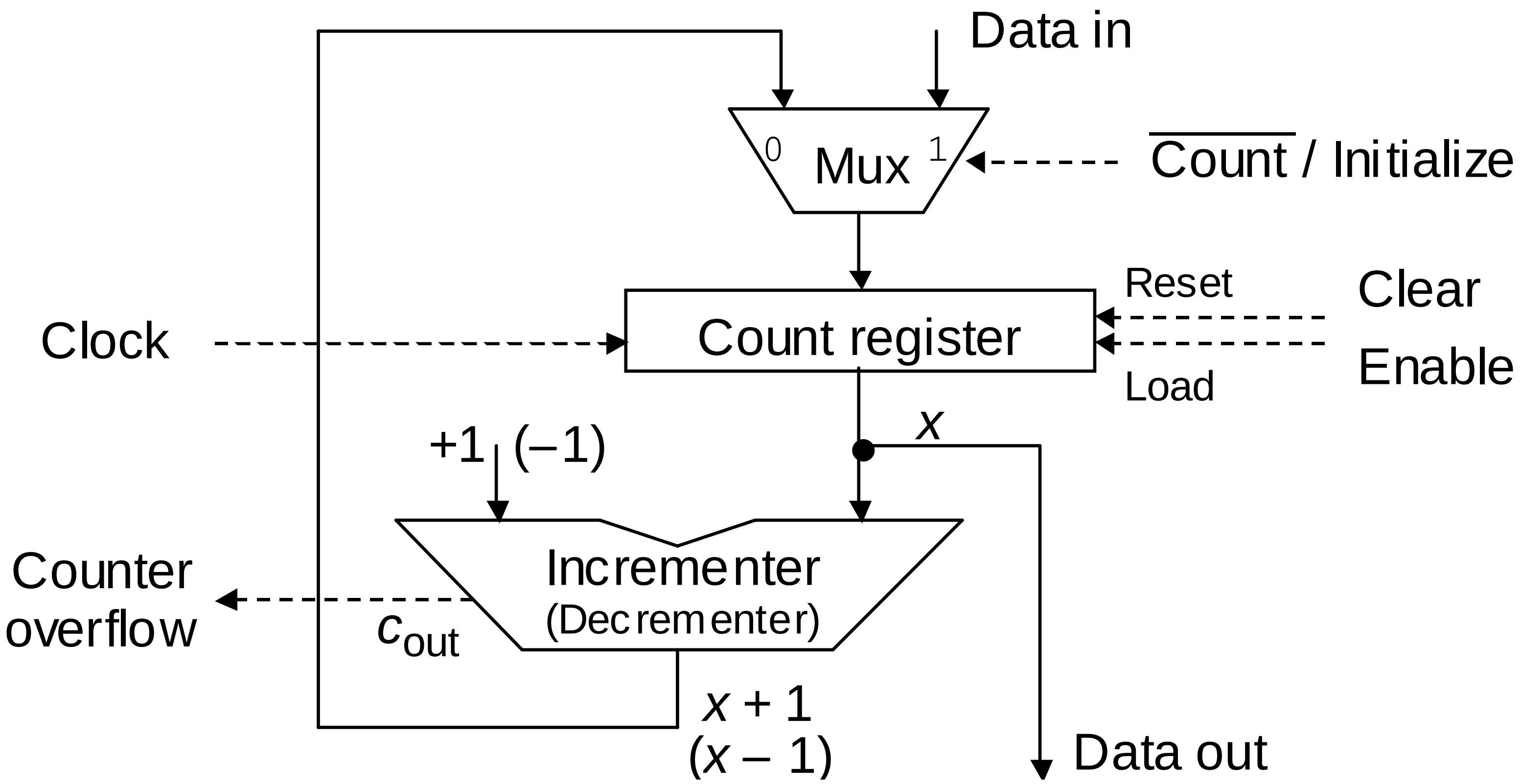
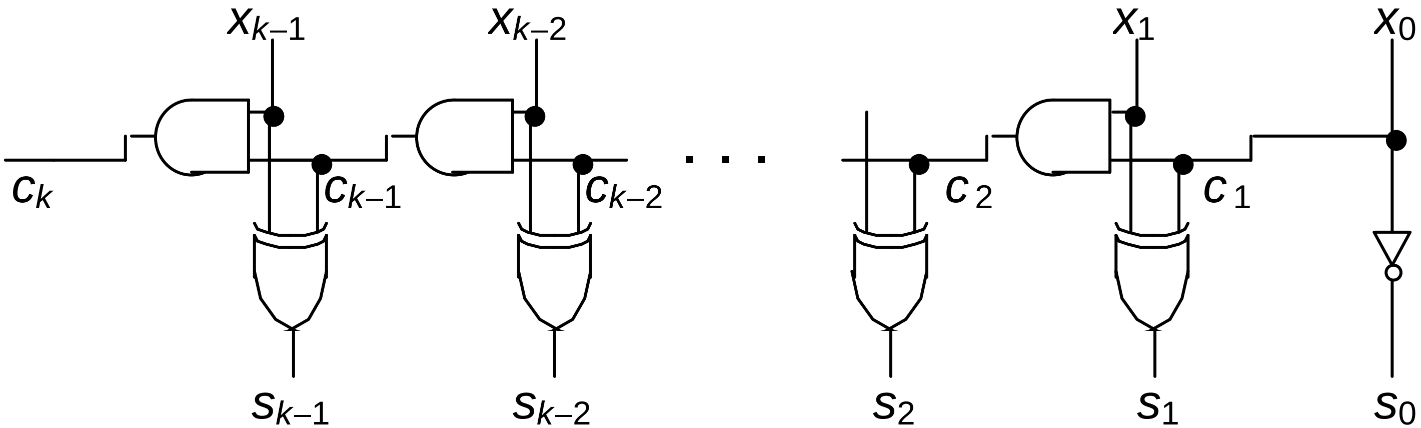


Fig. 5.10 An up (down) counter built of a register, an incrementer (decrementer), and a multiplexer.

Implementing a Simple Up Counter



(Fm arch text) Ripple-carry incrementer for use in an up counter.

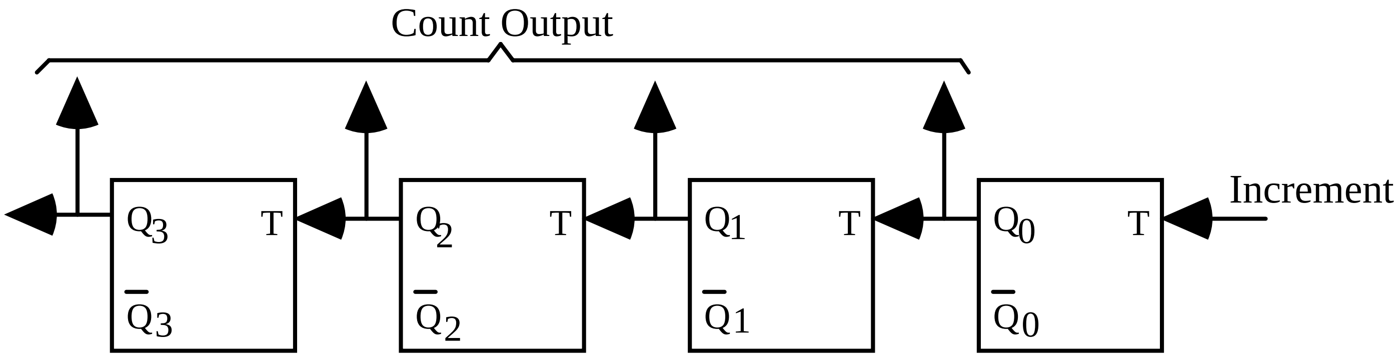


Fig. 5.11   Four-bit asynchronous up counter built only of negative-edge-triggered T flip-flops.

Faster and Constant-Time Counters

Any fast adder design can be specialized and optimized to yield a fast counter (carry-lookahead, carry-skip, etc.)

One can use redundant representation to build a constant-time counter, but a conversion penalty must be paid during read-out

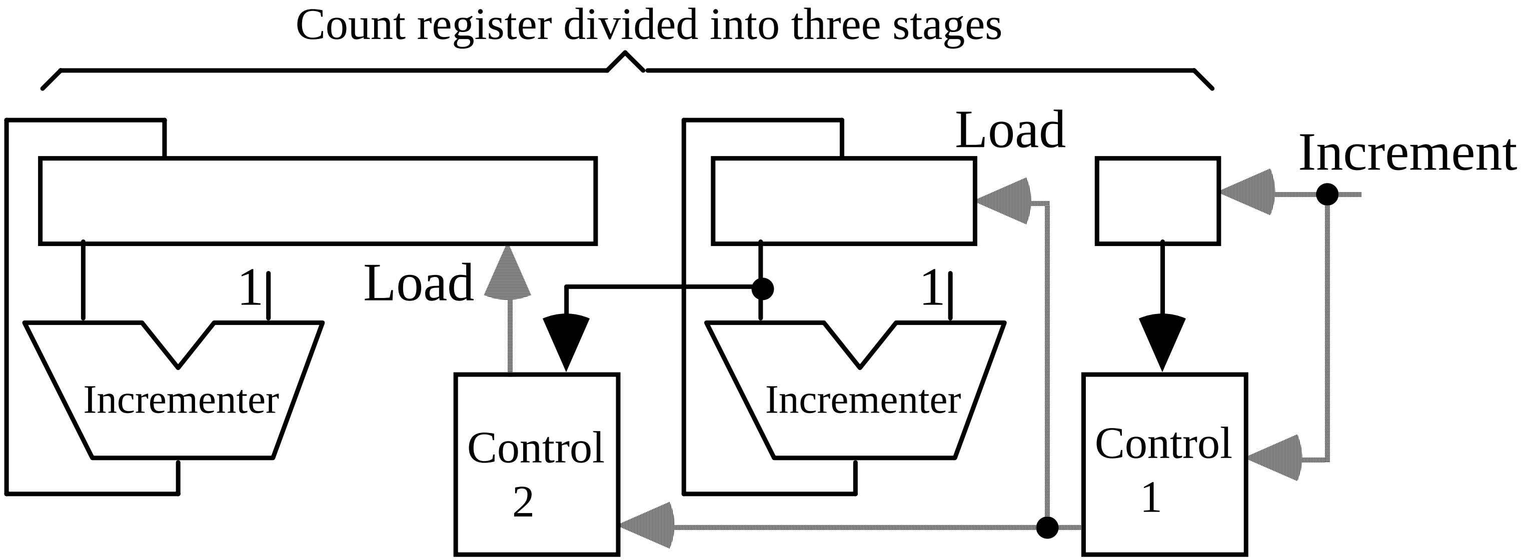


Fig. 5.12 Fast (constant-time) three-stage up counter.

5.6 Manchester Carry Chains and Adders

Sum digit in radix *r* $\quad s_i = (x_i + y_i + c_i) \bmod r$

Special case of radix 2 $\quad s_i = x_i \oplus y_i \oplus c_i$

Computing the carries c_i is thus our central problem
For this, the actual operand digits are not important
What matters is whether in a given position a carry is

generated, *propagated*, or *annihilated* (*absorbed*)

For binary addition:

$$g_i = x_i y_i \qquad p_i = x_i \oplus y_i \qquad a_i = x_i' y_i' = (x_i \vee y_i)'$$

It is also helpful to define a *transfer* signal:

$$t_i = g_i \vee p_i = a_i' = x_i \vee y_i$$

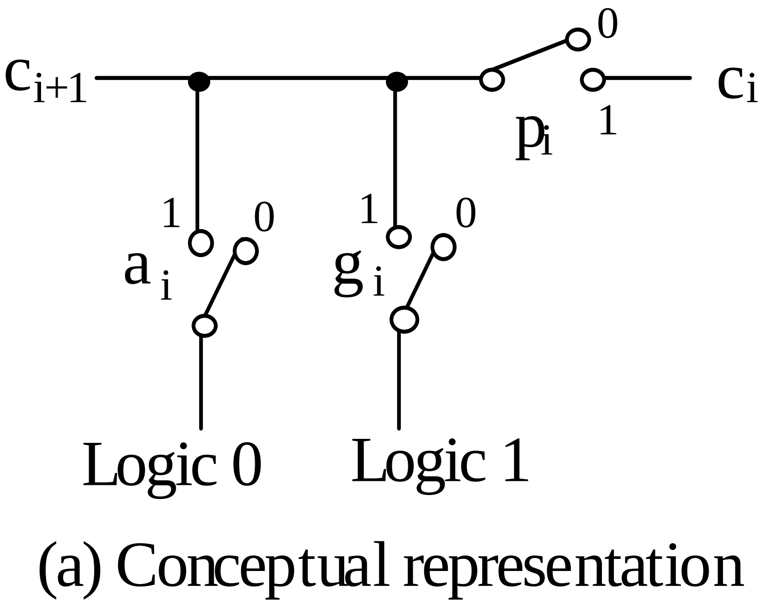
Using these signals, the *carry recurrence* is written as

$$c_{i+1} = g_i \vee c_i p_i = g_i \vee c_i g_i \vee c_i p_i = g_i \vee c_i t_i$$

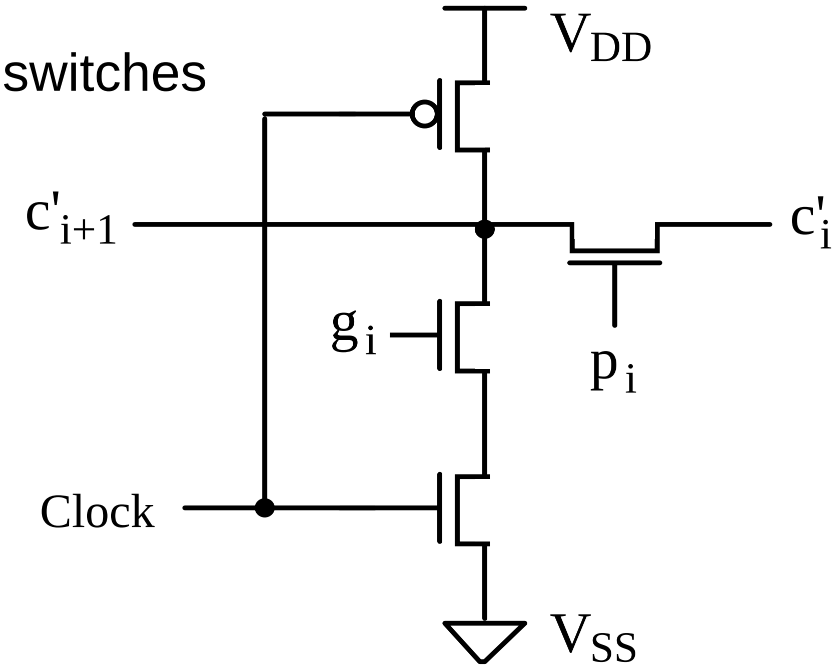
Manchester Carry Network

The worst-case delay of a Manchester carry chain has three components:

1. Latency of forming the switch control signals
2. Set-up time for switches
3. Signal propagation delay through *k* switches



(a) Conceptual representation

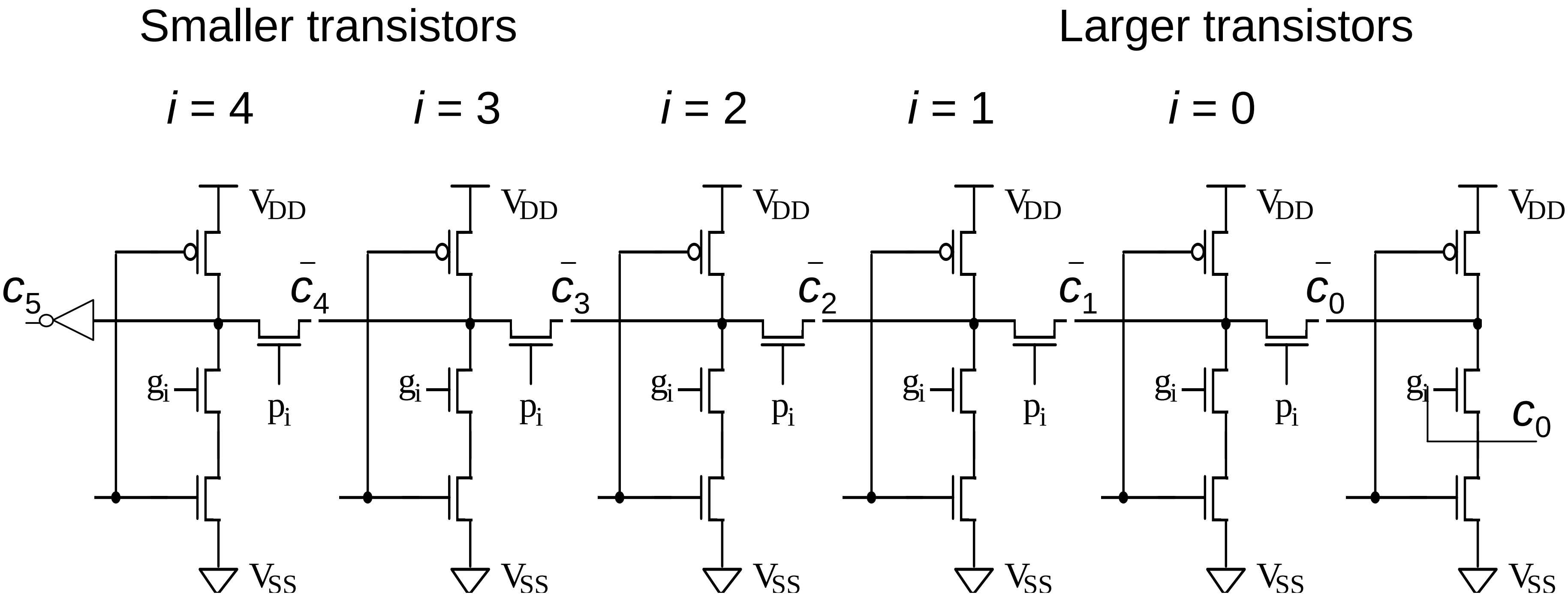


(b) Possible CMOS realization.

Fig. 5.13 One stage in a Manchester carry chain.

Details of a 5-Bit Manchester Carry Network

The transistors must be sized appropriately for maximum speed

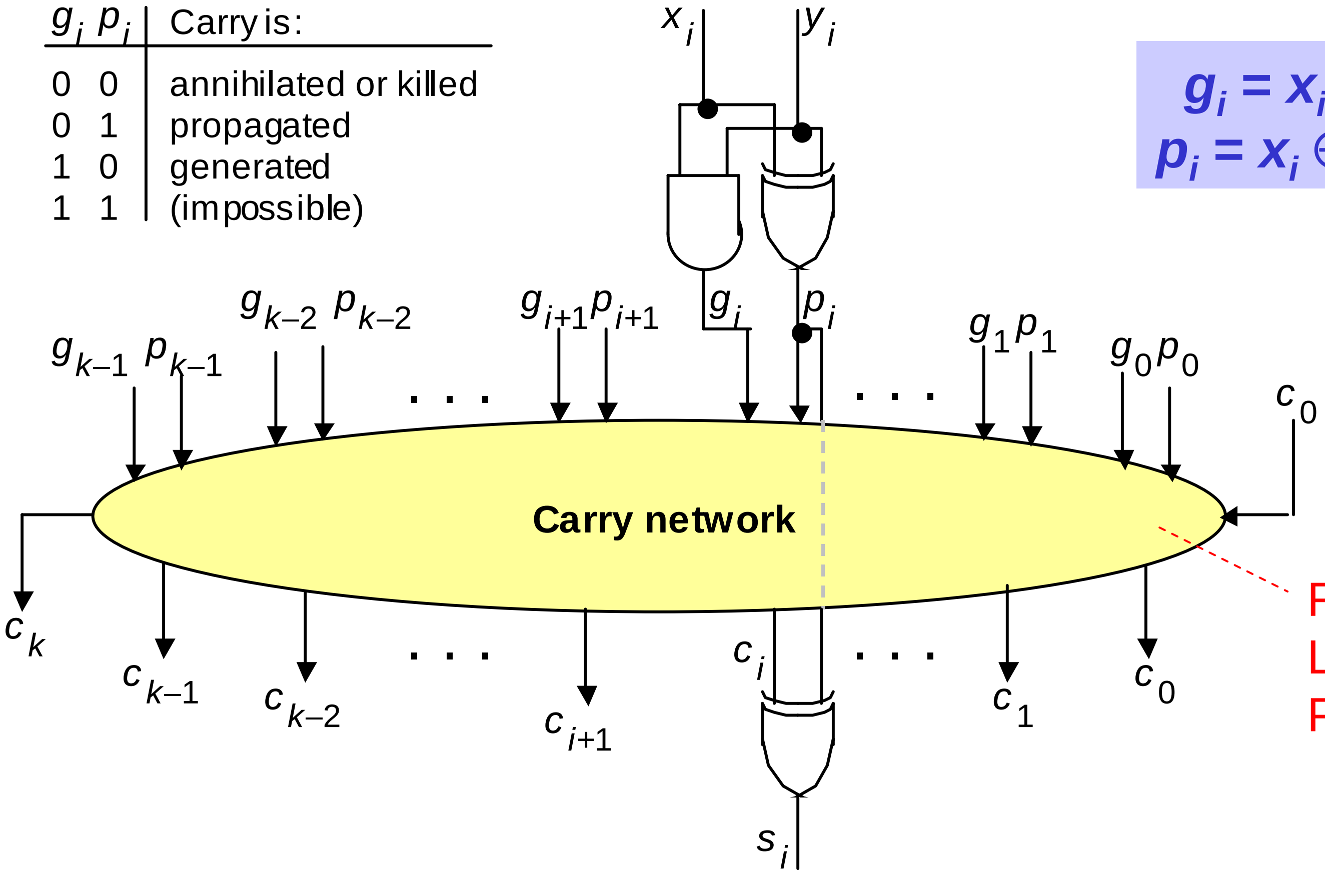


Carry chain of a 5-bit Manchester adder.

Carry Network is the Essence of a Fast Adder

g_i	p_i	Carry is:
0	0	annihilated or killed
0	1	propagated
1	0	generated
1	1	(impossible)

$$g_i = x_i\, y_i$$
$$p_i = x_i \oplus y_i$$



Ripple; Skip; Lookahead; Parallel-prefix

Fig. 5.14 Generic structure of a binary adder, highlighting its carry network.

Ripple-Carry Adder Revisited

The carry recurrence: $c_{i+1} = g_i \vee p_i c_i$

Latency of *k*-bit adder is roughly 2*k* gate delays:

1 gate delay for production of *p* and *g* signals, plus
2(*k* – 1) gate delays for carry propagation, plus
1 XOR gate delay for generation of the sum bits

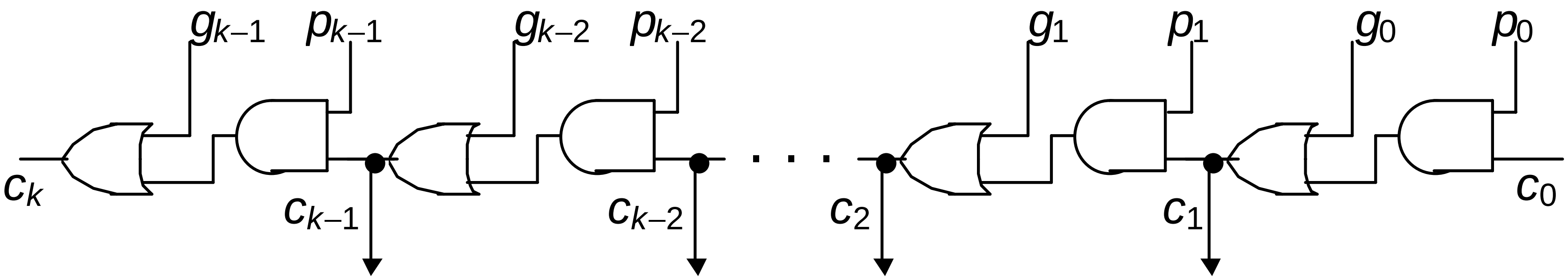


Fig. 5.15 Alternate view of a ripple-carry network in connection with the generic adder structure shown in Fig. 5.14.

The Complete Design of a Ripple-Carry Adder

g_i	p_i	Carry is:
0	0	annihilated or killed
0	1	propagated
1	0	generated
1	1	(impossible)

$$g_i = x_i\, y_i$$
$$p_i = x_i \oplus y_i$$

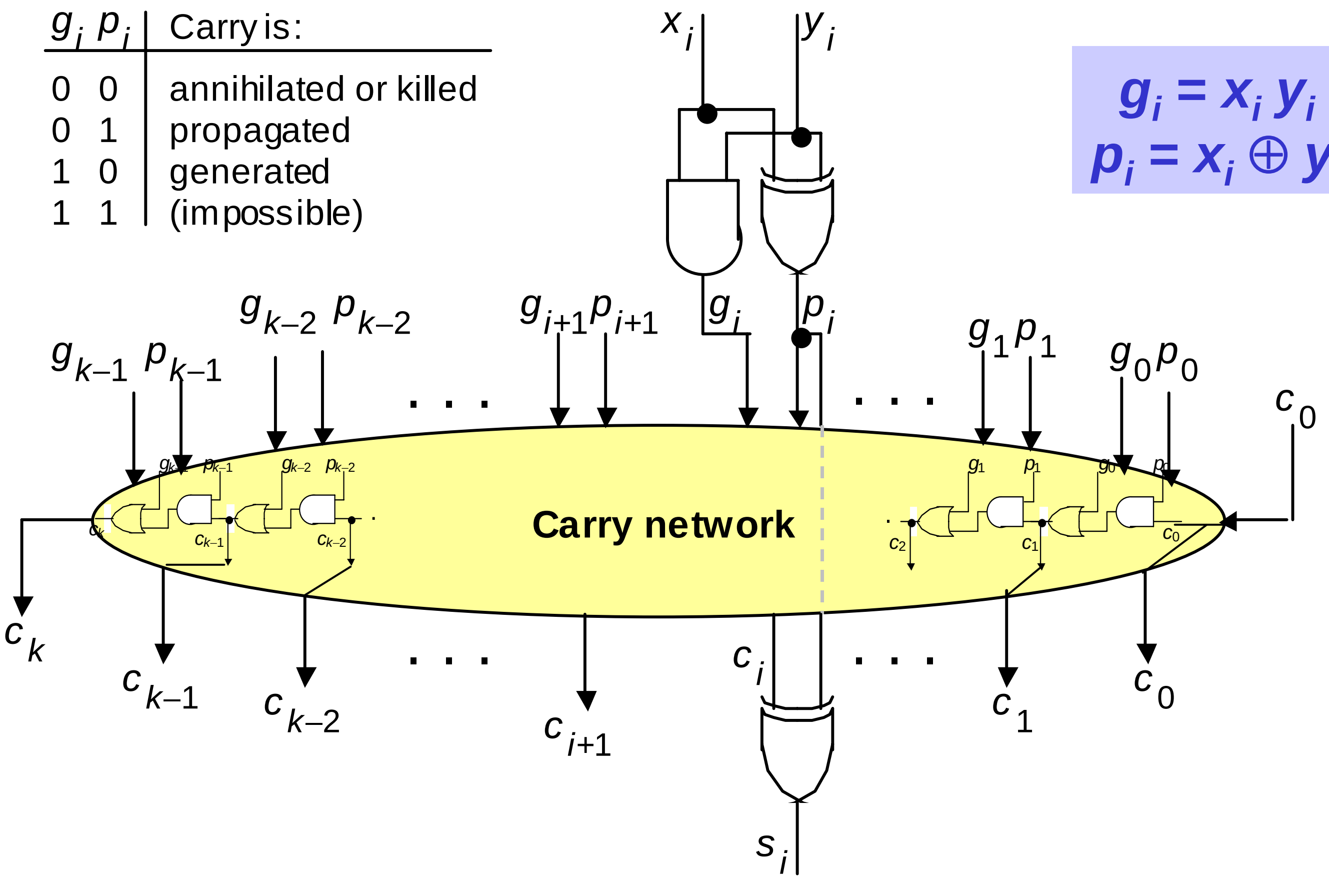


Fig. 5.15 (ripple-carry network) superimposed on Fig. 5.14 (generic adder).

6 Carry-Lookahead Adders

Chapter Goals

Understand the carry-lookahead method
and its many variations
used in the design of fast adders

Chapter Highlights

Single- and multilevel carry lookahead
Various designs for log-time adders
Relating the carry determination problem
to parallel prefix computation
Implementing fast adders in VLSI

Carry-Lookahead Adders: Topics

Topics in This Chapter
6.1 Unrolling the Carry Recurrence
6.2 Carry-Lookahead Adder Design
6.3 Ling Adder and Related Designs
6.4 Carry Determination as Prefix Computation
6.5 Alternative Parallel Prefix Networks
6.6 VLSI Implementation Aspects

6.1 Unrolling the Carry Recurrence

Recall the *generate*, *propagate*, *annihilate* (*absorb*), and *transfer* signals:

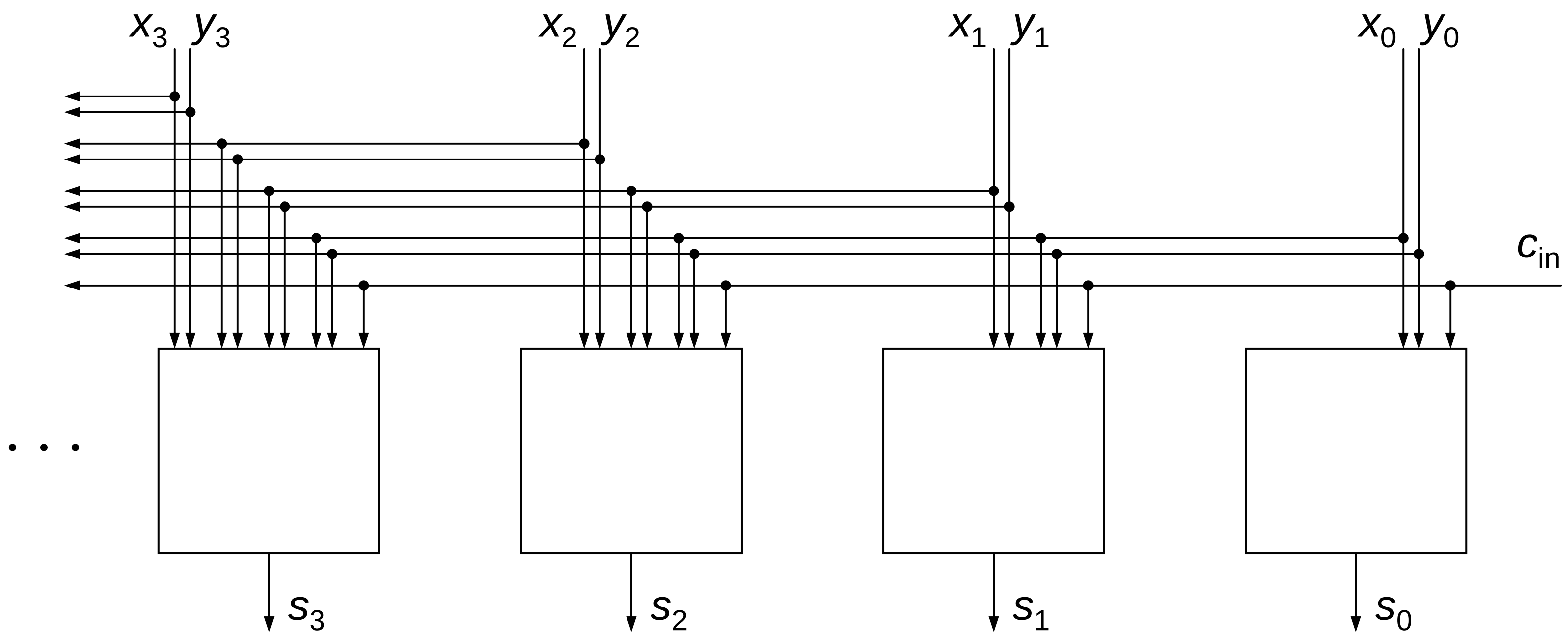
Signal	Radix *r*	Binary
g_i	is 1 iff $x_i + y_i \geq r$	$x_i\, y_i$
p_i	is 1 iff $x_i + y_i = r - 1$	$x_i \oplus y_i$
a_i	is 1 iff $x_i + y_i < r - 1$	$x_i' y_i' = (x_i \vee y_i)'$
t_i	is 1 iff $x_i + y_i \geq r - 1$	$x_i \vee y_i$
s_i	$(x_i + y_i + c_i) \bmod r$	$x_i \oplus y_i \oplus c_i$

The carry recurrence can be unrolled to obtain each carry signal directly from inputs, rather than through propagation

$$\begin{aligned}
c_i &= g_{i-1} \vee c_{i-1} p_{i-1} \\
&= g_{i-1} \vee (g_{i-2} \vee c_{i-2} p_{i-2})\, p_{i-1} \\
&= g_{i-1} \vee g_{i-2} p_{i-1} \vee c_{i-2} p_{i-2} p_{i-1} \\
&= g_{i-1} \vee g_{i-2} p_{i-1} \vee g_{i-3} p_{i-2} p_{i-1} \vee c_{i-3} p_{i-3} p_{i-2} p_{i-1} \\
&= g_{i-1} \vee g_{i-2} p_{i-1} \vee g_{i-3} p_{i-2} p_{i-1} \vee g_{i-4} p_{i-3} p_{i-2} p_{i-1} \vee c_{i-4} p_{i-4} p_{i-3} p_{i-2} p_{i-1} \\
&= \ldots
\end{aligned}$$

Note:
Addition symbol vs logical OR

Full Carry Lookahead

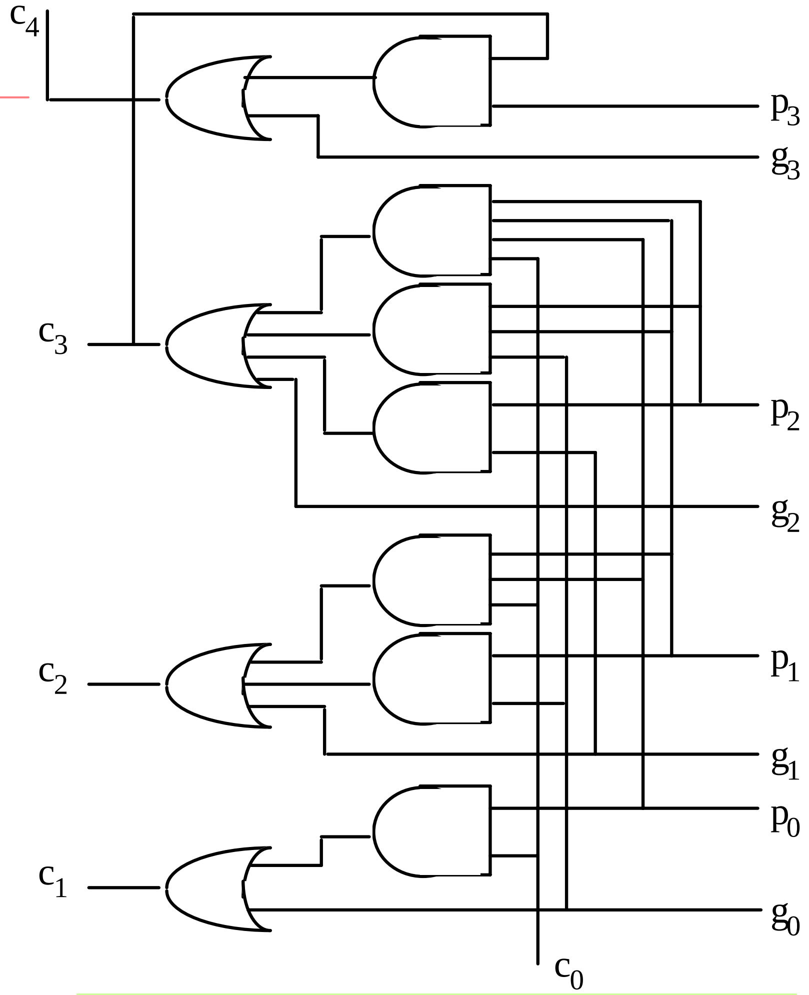


Theoretically, it is possible to derive each sum digit directly from the inputs that affect it

Carry-lookahead adder design is simply a way of reducing the complexity of this ideal, but impractical, arrangement by hardware sharing among the various lookahead circuits

Four-Bit Carry-Lookahead Adder

Complexity reduced by deriving the carry-out indirectly



Full carry lookahead is quite practical for a 4-bit adder

$$
\begin{aligned}
c_1 &= g_0 \vee c_0 p_0 \\
c_2 &= g_1 \vee g_0 p_1 \vee c_0 p_0 p_1 \\
c_3 &= g_2 \vee g_1 p_2 \vee g_0 p_1 p_2 \vee c_0 p_0 p_1 p_2 \\
c_4 &= g_3 \vee g_2 p_3 \vee g_1 p_2 p_3 \vee g_0 p_1 p_2 p_3 \\
&\qquad \vee c_0 p_0 p_1 p_2 p_3
\end{aligned}
$$

Fig. 6.1 Four-bit carry network with full lookahead.

Carry Lookahead Beyond 4 Bits

Consider a 32-bit adder

$$
\begin{aligned}
c_1 &= g_0 \vee c_0 p_0 \\
c_2 &= g_1 \vee g_0 p_1 \vee c_0 p_0 p_1 \\
c_3 &= g_2 \vee g_1 p_2 \vee g_0 p_1 p_2 \vee c_0 p_0 p_1 p_2 \\
&\ \vdots \\
c_{31} &= g_{30} \vee g_{29} p_{30} \vee g_{28} p_{29} p_{30} \vee g_{27} p_{28} p_{29} p_{30} \vee \ . \ . \ . \ \vee c_0 p_0 p_1 p_2 p_3 \ldots p_{29} p_{30}
\end{aligned}
$$

32-input AND

32-input OR

High fan-ins necessitate tree-structured circuits

Two Solutions to the Fan-in Problem

High-radix addition (i.e., radix 2^h)

Increases the latency for generating *g* and *p* signals and sum digits, but simplifies the carry network (optimal radix?)

Multilevel lookahead

Example: 16-bit addition

Radix-16 (four digits)

Two-level carry lookahead (four 4-bit blocks)

Either way, the carries c_4, c_8, and c_{12} are determined first

c_{16}	c_{15}	c_{14}	c_{13}	c_{12}	c_{11}	c_{10}	c_9	c_8	c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0
c_{out}				?				?				?				c_{in}

6.2 Carry-Lookahead Adder Design

Block *generate* and *propagate* signals

$$g_{[i,i+3]} = g_{i+3} \vee g_{i+2}p_{i+3} \vee g_{i+1}p_{i+2}p_{i+3} \vee g_i\, p_{i+1}p_{i+2}p_{i+3}$$
$$p_{[i,i+3]} = p_i\, p_{i+1}p_{i+2}p_{i+3}$$

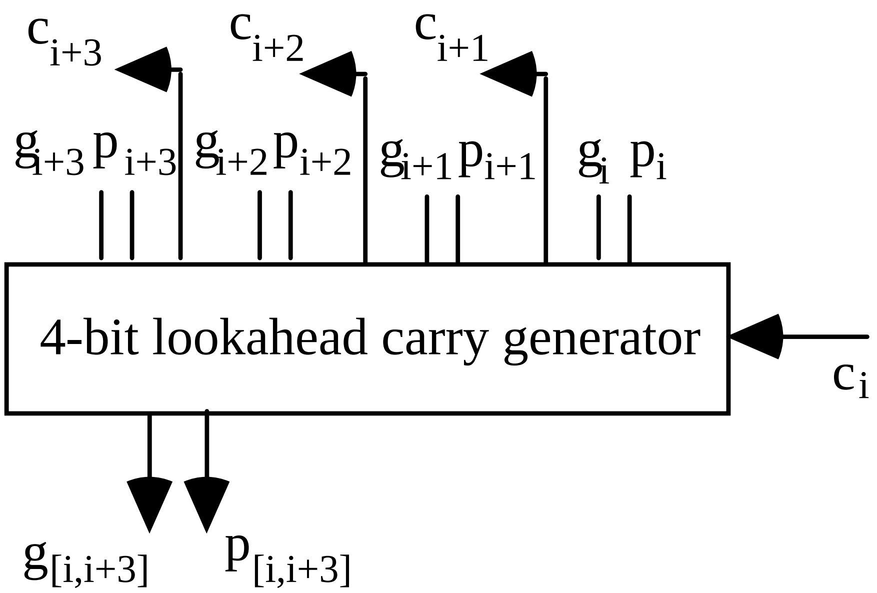


Fig. 6.2b Schematic diagram of a 4-bit lookahead carry generator.

A Building Block for Carry-Lookahead Addition

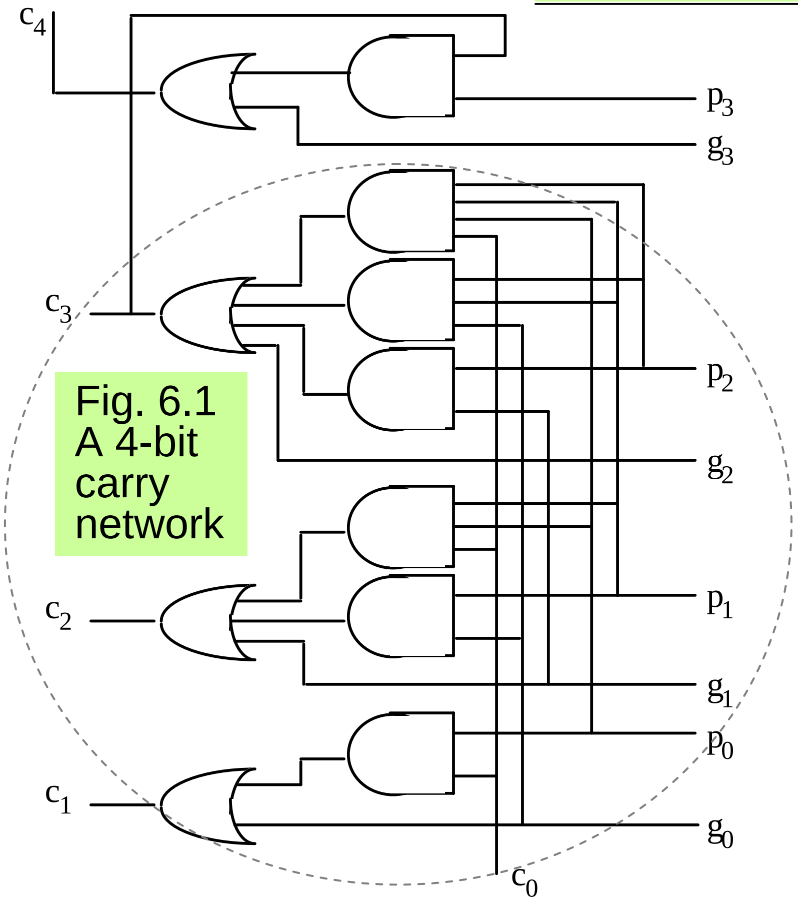
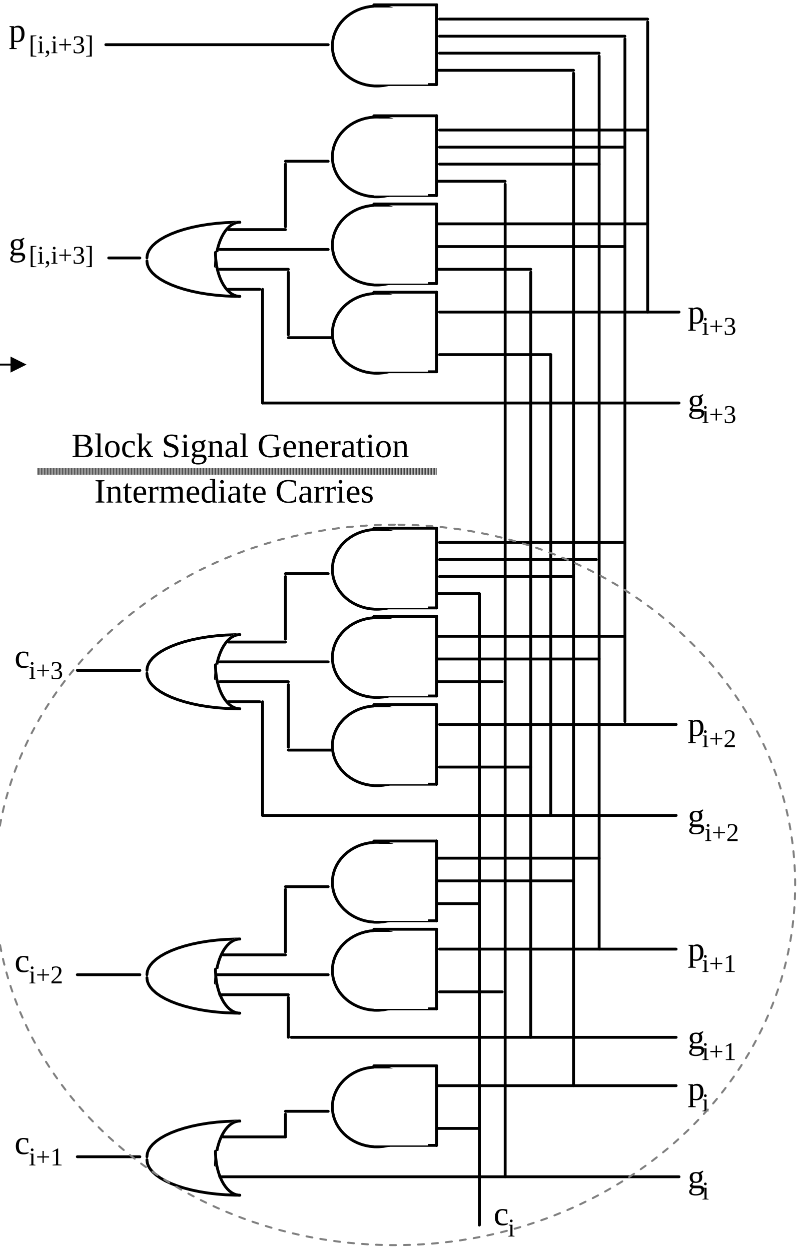
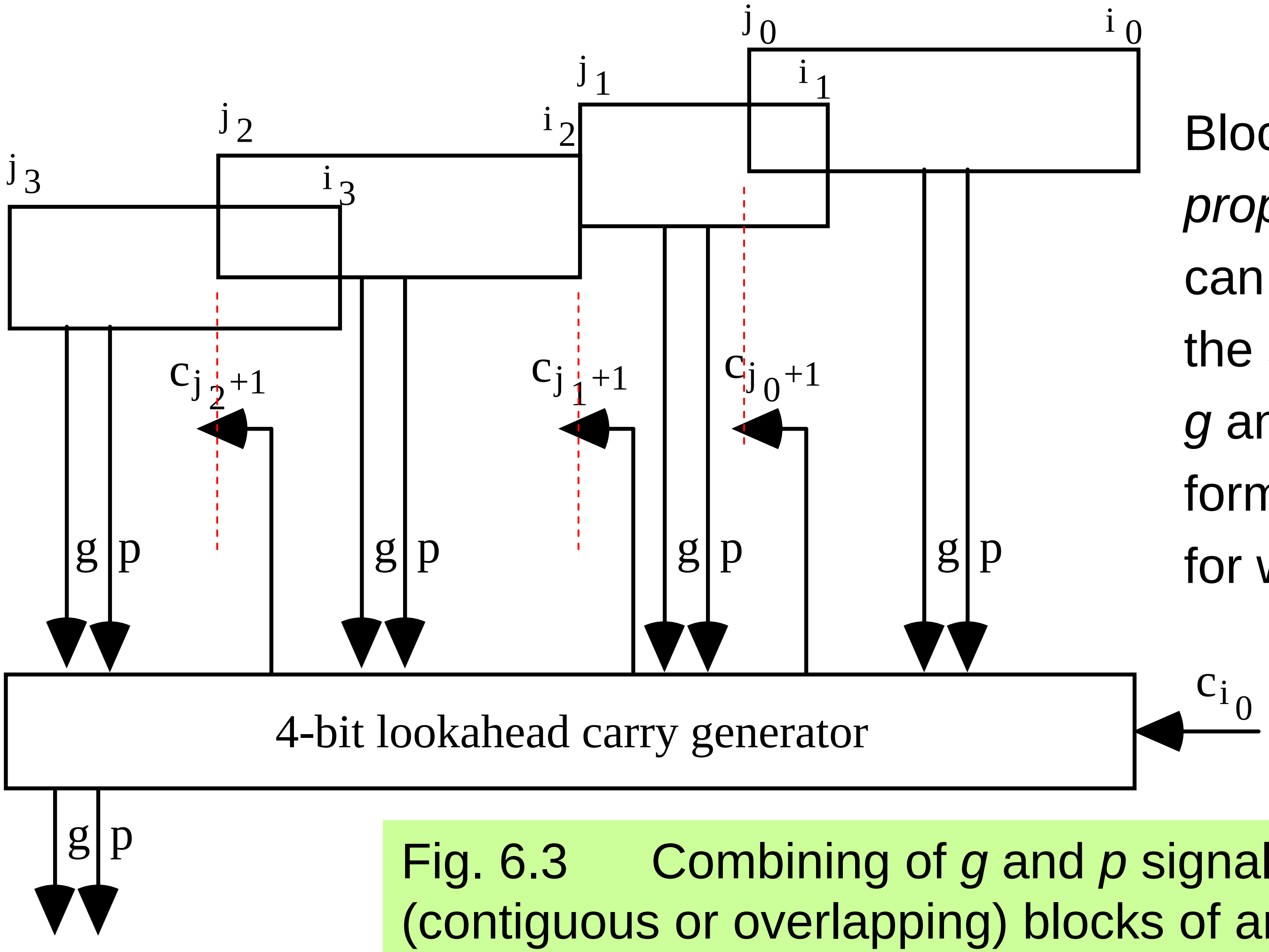


Fig. 6.1
A 4-bit
carry
network

Fig. 6.2a A 4-bit
lookahead
carry generator



Combining Block *g* and *p* Signals



Block *generate* and *propagate* signals can be combined in the same way as bit *g* and *p* signals to form *g* and *p* signals for wider blocks

Fig. 6.3 Combining of *g* and *p* signals of four (contiguous or overlapping) blocks of arbitrary widths into the *g* and *p* signals for the overall block $[i_0, j_3]$.

A Two-Level Carry-Lookahead Adder

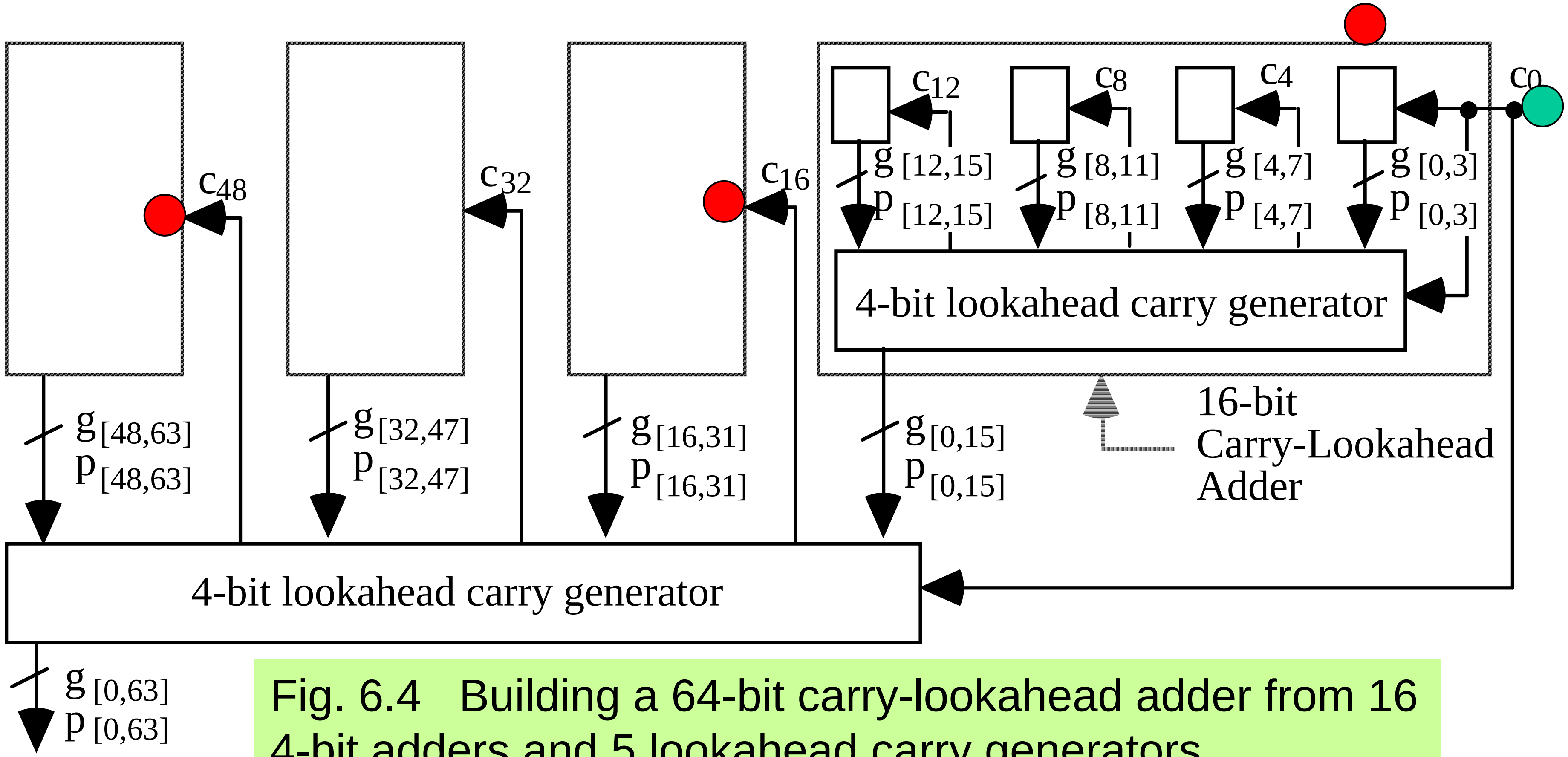


Fig. 6.4 Building a 64-bit carry-lookahead adder from 16 4-bit adders and 5 lookahead carry generators.

Carry-out: $c_{\text{out}} = g_{[0,k-1]} \vee c_0\, p_{[0,k-1]} = x_{k-1}y_{k-1} \vee s_{k-1}'\,(x_{k-1} \vee y_{k-1})$

Latency of a Multilevel Carry-Lookahead Adder

Latency through the 16-bit CLA adder consists of finding:

g and *p* for individual bit positions	1 gate level
g and *p* signals for 4-bit blocks	2 gate levels
Block carry-in signals c_4, c_8, and c_{12}	2 gate levels
Internal carries within 4-bit blocks	2 gate levels
Sum bits	2 gate levels

Total latency for the 16-bit adder 9 gate levels

(compare to 32 gate levels for a 16-bit ripple-carry adder)

Each additional lookahead level adds 4 gate levels of latency

Latency for *k*-bit CLA adder: $T_{\text{lookahead-add}} = 4 \log_4 k + 1$ gate levels

6.3 Ling Adder and Related Designs

Consider the carry recurrence and its unrolling by 4 steps:

$$\begin{aligned} c_i &= g_{i-1} \vee c_{i-1} t_{i-1} \\ &= g_{i-1} \vee g_{i-2} t_{i-1} \vee g_{i-3} t_{i-2} t_{i-1} \vee g_{i-4} t_{i-3} t_{i-2} t_{i-1} \vee c_{i-4} t_{i-4} t_{i-3} t_{i-2} t_{i-1} \end{aligned}$$

Ling’s modification: Propagate $h_i = c_i \vee c_{i-1}$ instead of c_i

$$\begin{aligned} h_i &= g_{i-1} \vee h_{i-1} t_{i-2} \\ &= g_{i-1} \vee g_{i-2} \vee g_{i-3} t_{i-2} \vee g_{i-4} t_{i-3} t_{i-2} \vee h_{i-4} t_{i-4} t_{i-3} t_{i-2} \end{aligned}$$

Propagate harry, not carry!

CLA:	5 gates	max 5 inputs	19 gate inputs
Ling:	4 gates	max 5 inputs	14 gate inputs

The advantage of h_i over c_i is even greater with wired-OR:

CLA:	4 gates	max 5 inputs	14 gate inputs
Ling:	3 gates	max 4 inputs	9 gate inputs

Once h_i is known, however, the sum is obtained by a slightly more complex expression compared with $s_i = p_i \oplus c_i$

$$s_i = p_i \oplus h_i t_{i-1}$$

6.4 Carry Determination as Prefix Computation

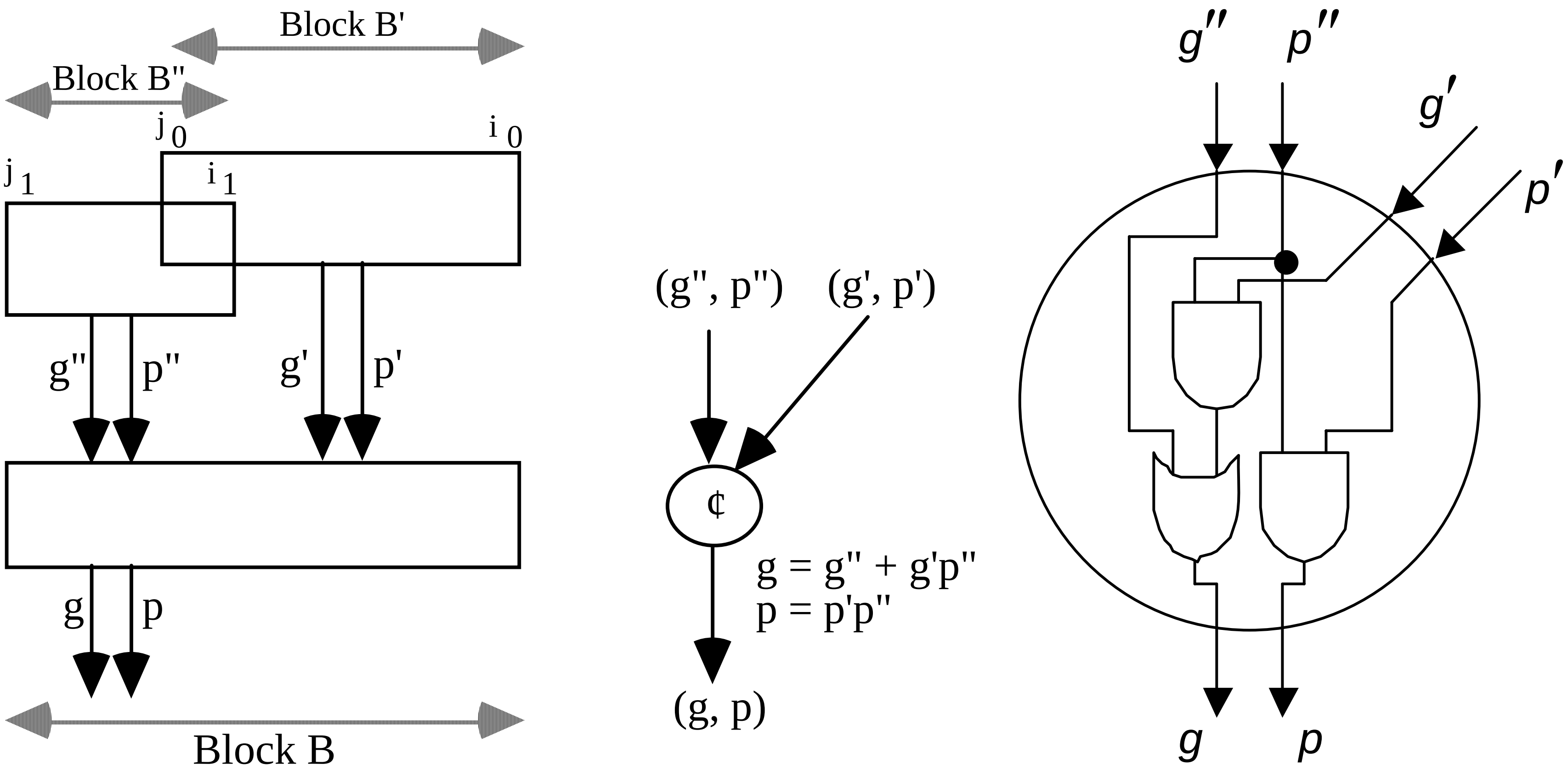


Fig. 6.5 Combining of *g* and *p* signals of two (contiguous or overlapping) blocks B' and B" of arbitrary widths into the *g* and *p* signals for block B.

Formulating the Prefix Computation Problem

The problem of carry determination can be formulated as:

Given	(g_0, p_0)	(g_1, p_1)	. . .	(g_{k-2}, p_{k-2})	(g_{k-1}, p_{k-1})
Find	$(g_{[0,0]}, p_{[0,0]})$	$(g_{[0,1]}, p_{[0,1]})$	. . .	$(g_{[0,k-2]}, p_{[0,k-2]})$	$(g_{[0,k-1]}, p_{[0,k-1]})$
	$\downarrow$	$\downarrow$		$\downarrow$	$\downarrow$
	c_1	c_2	. . .	c_{k-1}	c_k

Carry-in can be viewed as an extra (–1) position: $(g_{-1}, p_{-1}) = (c_{in}, 0)$

The desired pairs are found by evaluating all prefixes of

$$(g_0, p_0) \text{ ¢ } (g_1, p_1) \text{ ¢ } \;.\;.\;.\; \text{ ¢ } (g_{k-2}, p_{k-2}) \text{ ¢ } (g_{k-1}, p_{k-1})$$

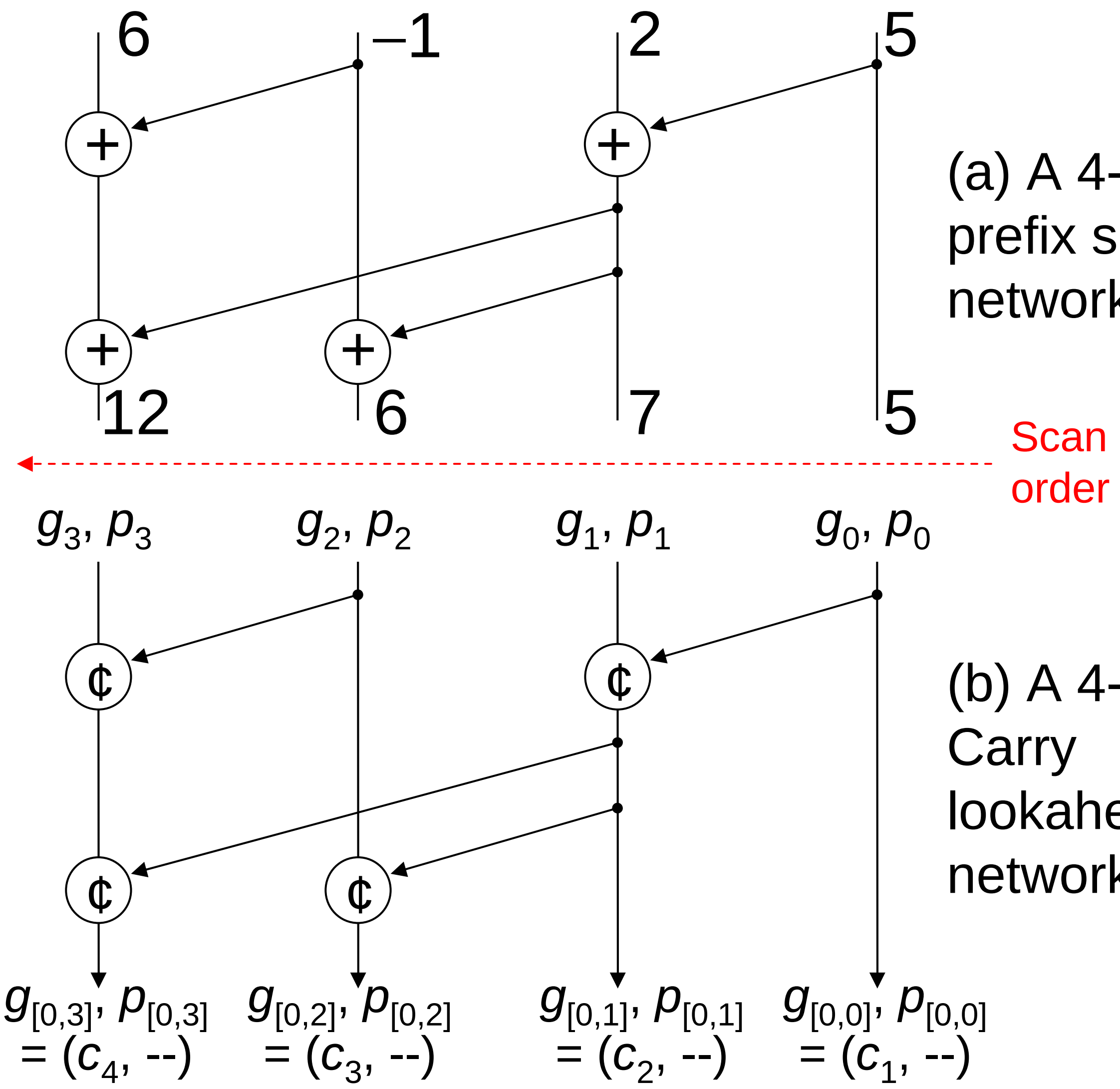
The carry operator ¢ is associative, but not commutative

$$[(g_1, p_1) \text{ ¢ } (g_2, p_2)] \text{ ¢ } (g_3, p_3) = (g_1, p_1) \text{ ¢ } [(g_2, p_2) \text{ ¢ } (g_3, p_3)]$$

Prefix sums analogy:

Given	x_0	x_1	x_2	. . .	x_{k-1}
Find	x_0	x_0+x_1	$x_0+x_1+x_2$	. . .	$x_0+x_1+\ldots+x_{k-1}$

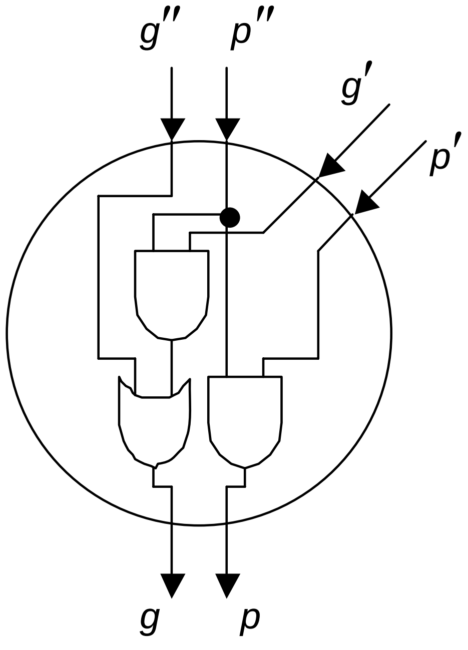
Example Prefix-Based Carry Network


6
–1
2
5
+
+
+
+
12
6
7
5
Scan order
g_3, p_3
g_2, p_2
g_1, p_1
g_0, p_0
¢
¢
¢
¢
$g_{[0,3]}, p_{[0,3]}$ = $(c_4$, --)
$g_{[0,2]}, p_{[0,2]}$ = $(c_3$, --)
$g_{[0,1]}, p_{[0,1]}$ = $(c_2$, --)
$g_{[0,0]}, p_{[0,0]}$ = $(c_1$, --)

(a) A 4-input prefix sums network

(b) A 4-bit Carry lookahead network

Fig. 6.6 Four-input parallel prefix sums network and its corresponding carry network.



6.5 Alternative Parallel Prefix Networks

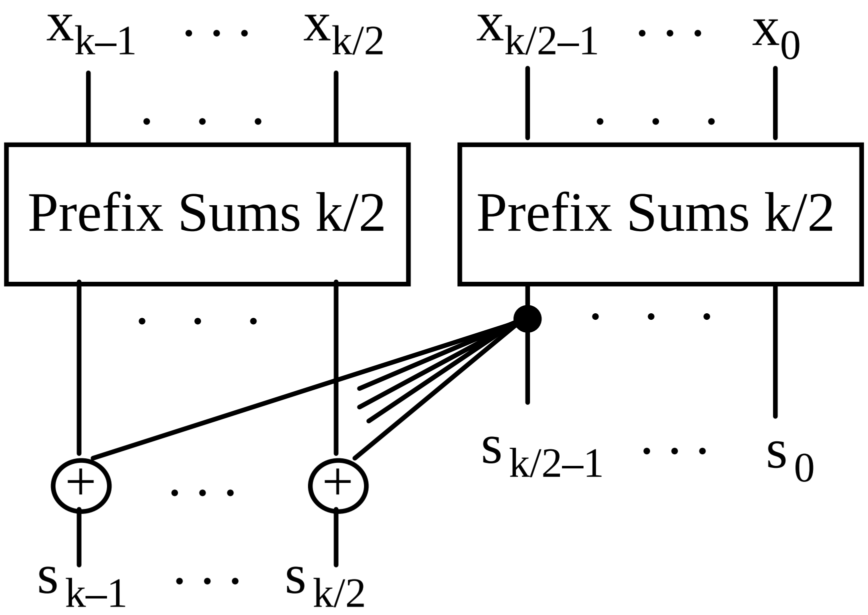


Fig. 6.7 Ladner-Fischer parallel prefix sums network built of two *k*/2-input networks and *k*/2 adders.

Delay recurrence $D(k) = D(k/2) + 1 = \log_2 k$

Cost recurrence $C(k) = 2C(k/2) + k/2 = (k/2) \log_2 k$

The Brent-Kung Recursive Construction

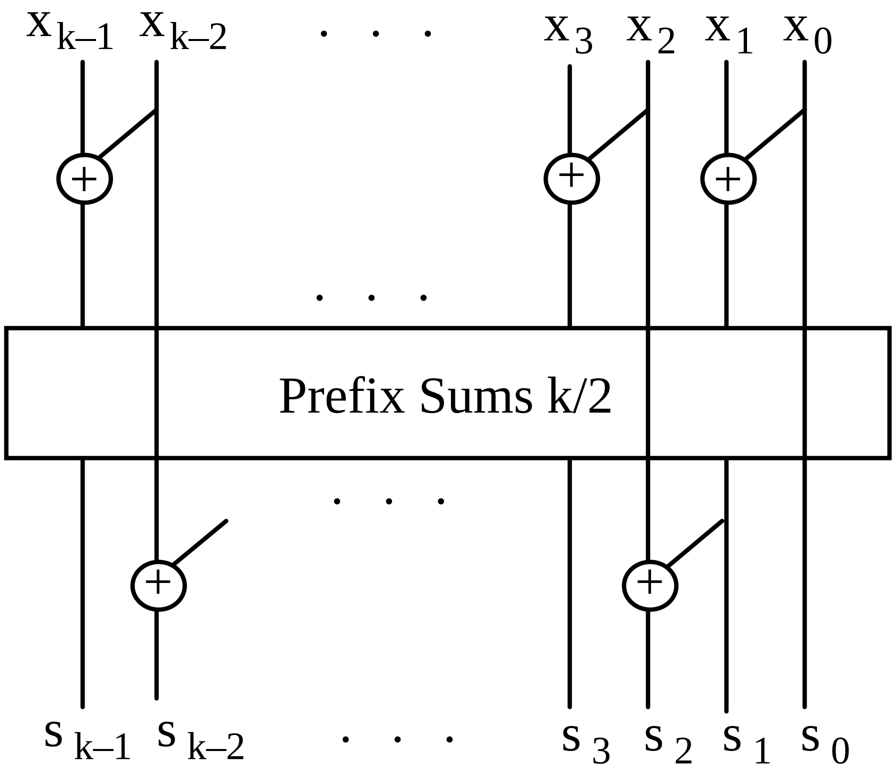
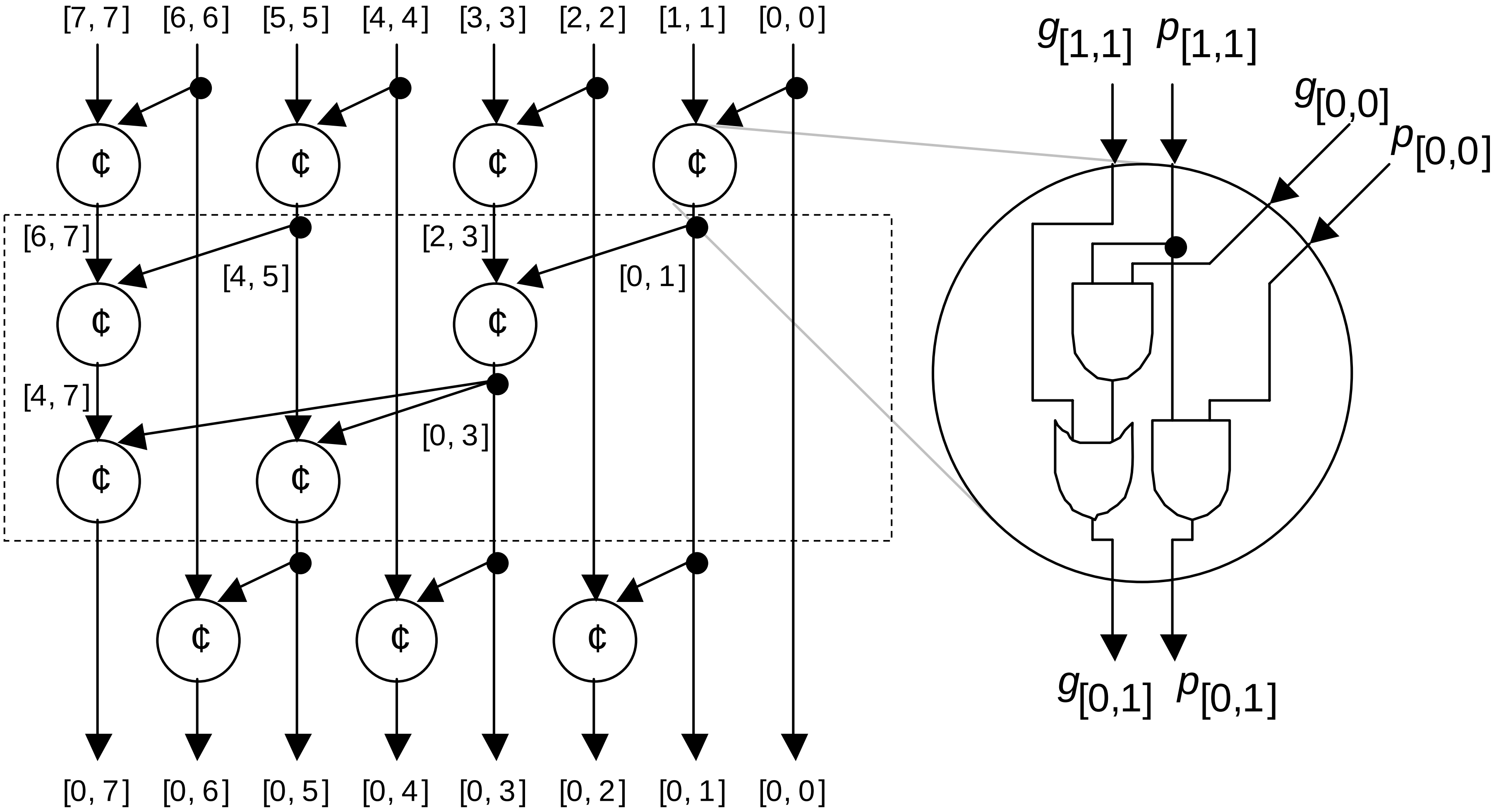

x_{k-1} x_{k-2} . . . x_3 x_2 x_1 x_0
+
. . .
Prefix Sums k/2
s_{k-1} s_{k-2} . . . s_3 s_2 s_1 s_0

Fig. 6.8 Parallel prefix sums network built of one *k*/2-input network and *k* – 1 adders.

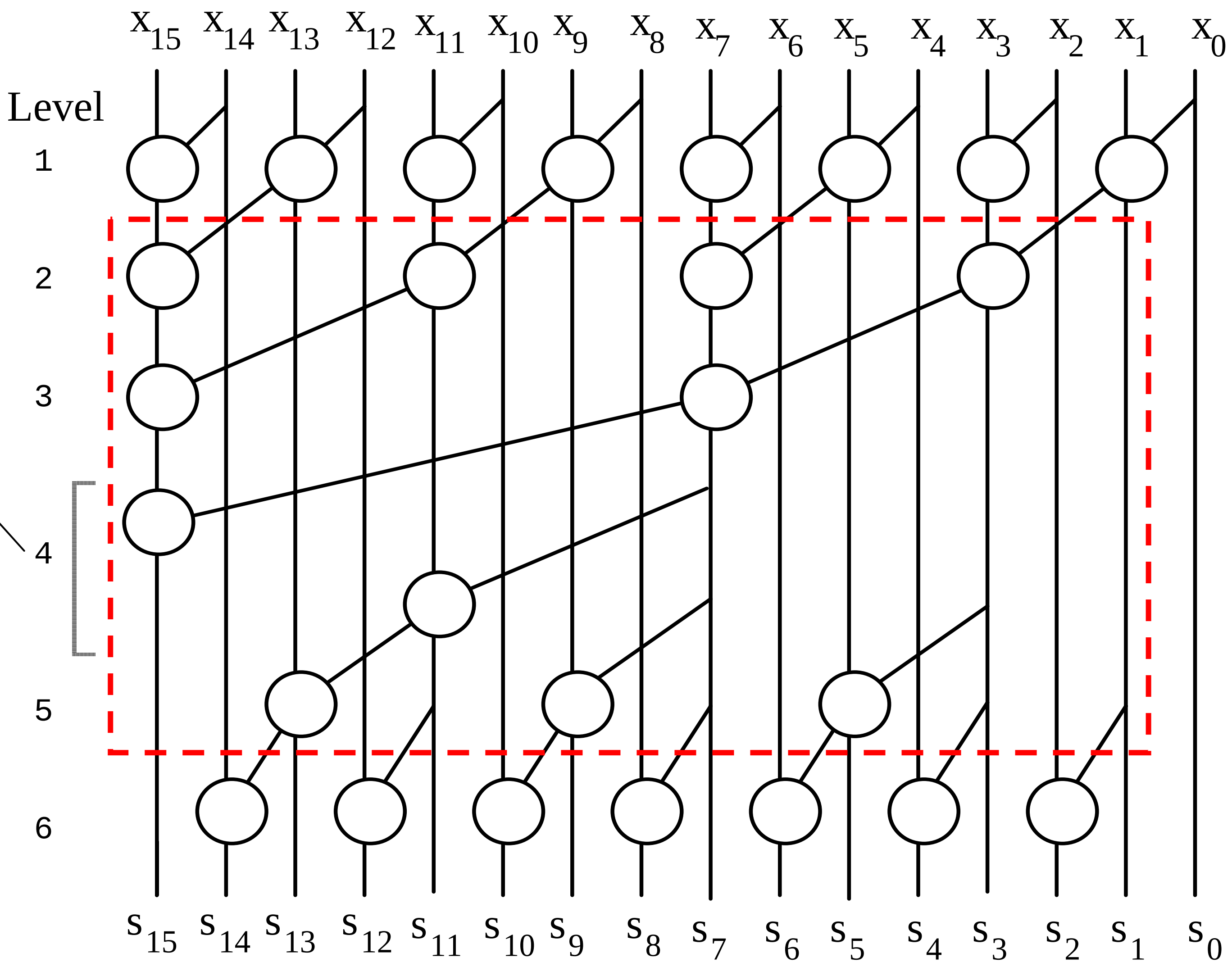
Delay recurrence $D(k) = D(k/2) + 2 = 2 \log_2 k - 1$ (–2 really)

Cost recurrence $C(k) = C(k/2) + k - 1 = 2k - 2 - \log_2 k$

Brent-Kung Carry Network (8-Bit Adder)



Brent-Kung Carry Network (16-Bit Adder)


x_{15} x_{14} x_{13} x_{12} x_{11} x_{10} x_9 x_8 x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0
Level
1
2
3
4
5
6
s_{15} s_{14} s_{13} s_{12} s_{11} s_{10} s_9 s_8 s_7 s_6 s_5 s_4 s_3 s_2 s_1 s_0

Reason for latency being $2 \log_2 k - 2$

Fig. 6.9 Brent-Kung parallel prefix graph for 16 inputs.

Kogge-Stone Carry Network (16-Bit Adder)

Cost formula
$C(k) = (k-1)$
$+ (k-2)$
$+ (k-4) + \ldots$
$+ (k-k/2)$
$= k\log_2 k - k + 1$

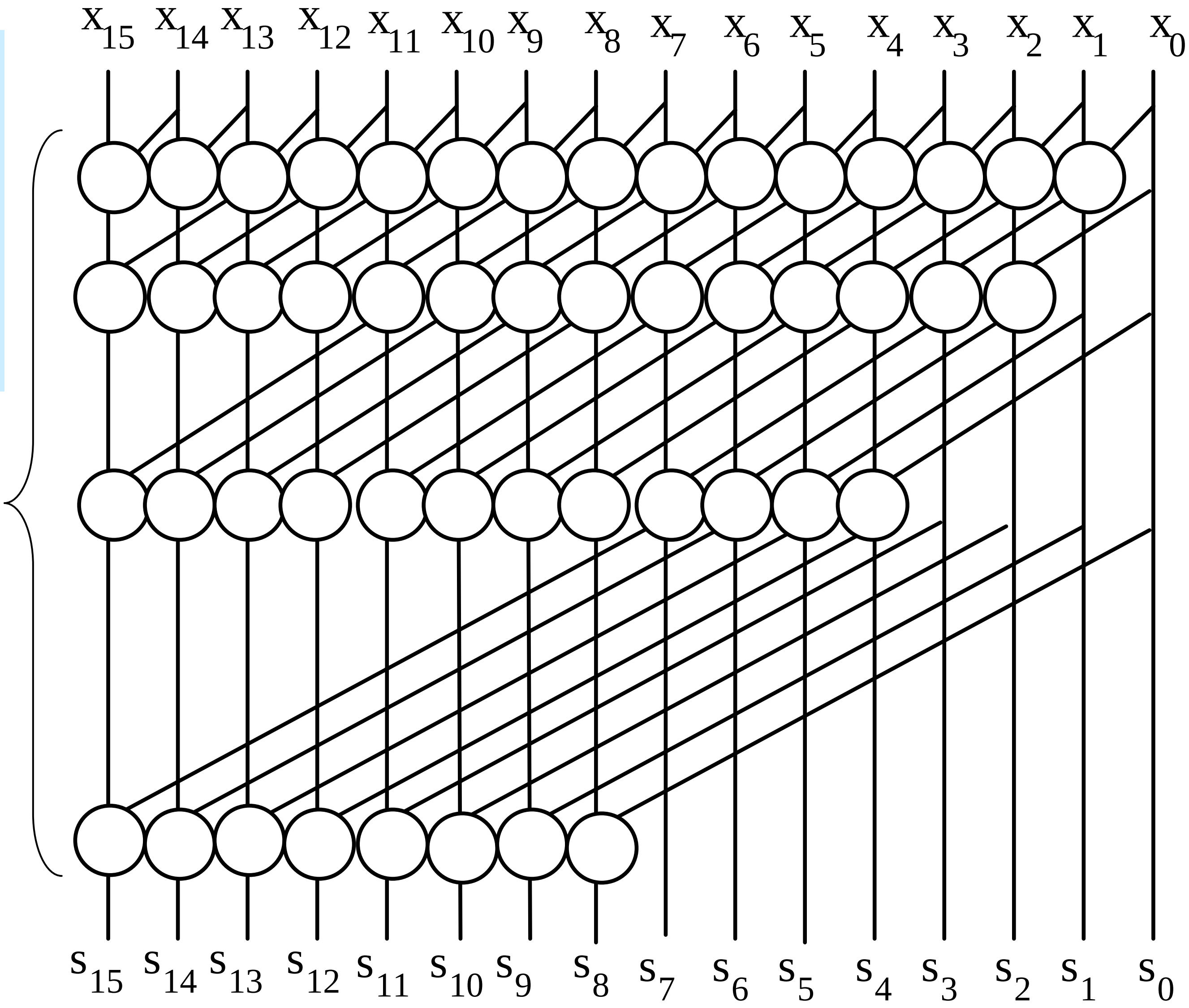
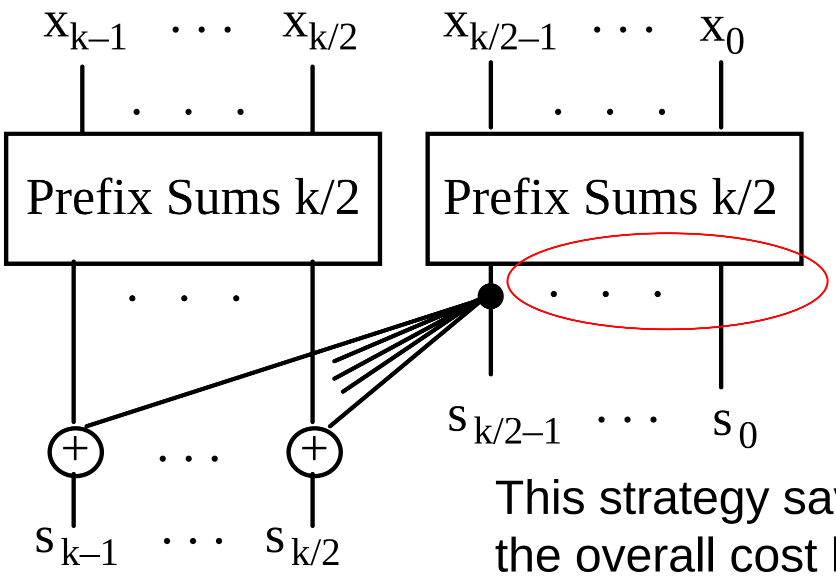

x_{15} x_{14} x_{13} x_{12} x_{11} x_{10} x_9 x_8 x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0
$\log_2 k$ levels (minimum possible)
s_{15} s_{14} s_{13} s_{12} s_{11} s_{10} s_9 s_8 s_7 s_6 s_5 s_4 s_3 s_2 s_1 s_0

Fig. 6.10 Kogge-Stone parallel prefix graph for 16 inputs.

Speed-Cost Tradeoffs in Carry Networks

Method	Delay	Cost
Ladner-Fischer	$\log_2 k$	$(k/2) \log_2 k$
Kogge-Stone	$\log_2 k$	$k \log_2 k - k + 1$
Brent-Kung	$2 \log_2 k - 2$	$2k - 2 - \log_2 k$

Improving the Ladner/Fischer design

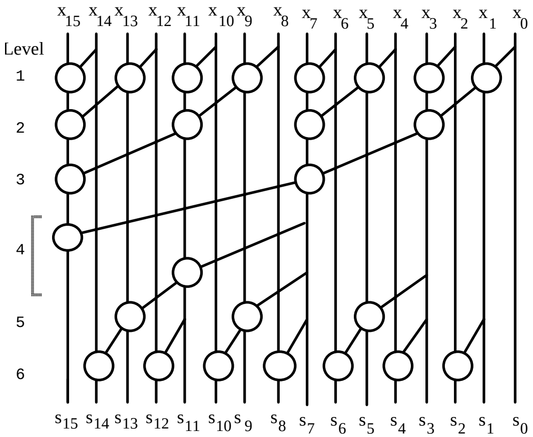


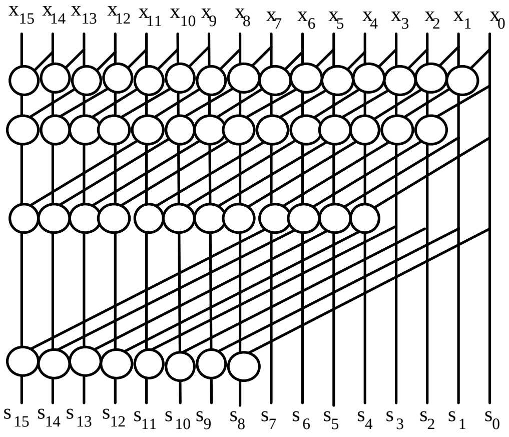
These outputs can be produced one time unit later without increasing the overall latency

This strategy saves enough to make the overall cost linear (best possible)

Hybrid B-K/K-S Carry Network (16-Bit Adder)

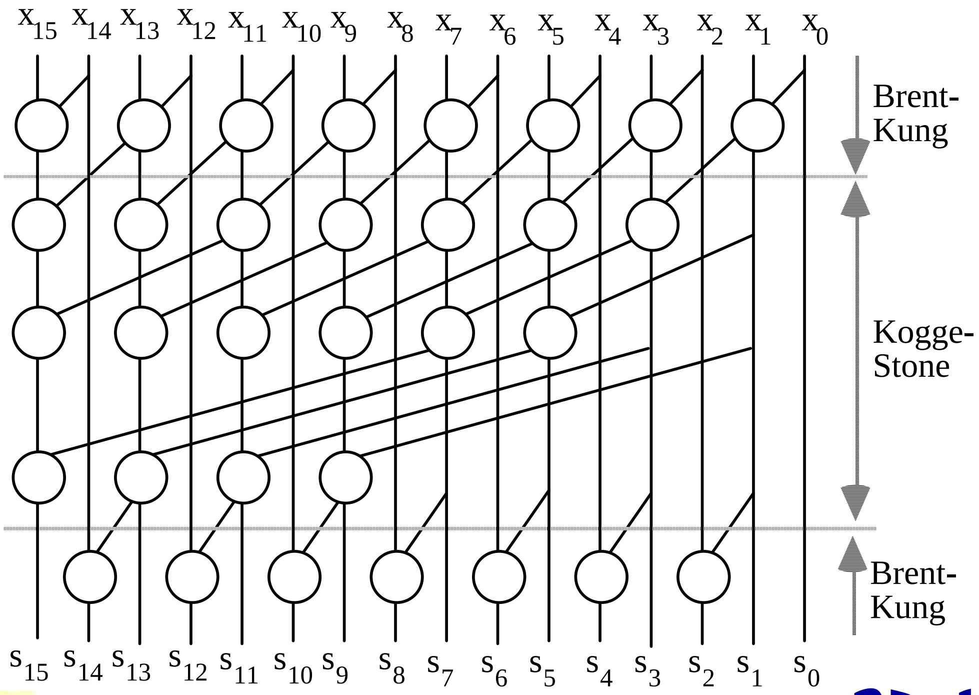
Brent-Kung:
6 levels
26 cells


x15 x14 x13 x12 x11 x10 x9 x8 x7 x6 x5 x4 x3 x2 x1 x0
Level
1
2
3
4
5
6
s15 s14 s13 s12 s11 s10 s9 s8 s7 s6 s5 s4 s3 s2 s1 s0


x15 x14 x13 x12 x11 x10 x9 x8 x7 x6 x5 x4 x3 x2 x1 x0
s15 s14 s13 s12 s11 s10 s9 s8 s7 s6 s5 s4 s3 s2 s1 s0

Kogge-Stone:
4 levels
49 cells

Fig. 6.11
A Hybrid Brent-Kung/ Kogge-Stone parallel prefix graph for 16 inputs.


x15 x14 x13 x12 x11 x10 x9 x8 x7 x6 x5 x4 x3 x2 x1 x0
Brent-
Kung
Kogge-
Stone
Brent-
Kung
s15 s14 s13 s12 s11 s10 s9 s8 s7 s6 s5 s4 s3 s2 s1 s0

Hybrid:
5 levels
32 cells

6.6 VLSI Implementation Aspects

Example: Radix-256 addition of 56-bit numbers
as implemented in the AMD Am29050 CMOS micro

Our description is based on the 64-bit version of the adder

In radix-256, 64-bit addition, only these carries are needed:

c_{56} c_{48} c_{40} c_{32} c_{24} c_{16} c_8

First, 4-bit Manchester carry chains (MCCs) of Fig. 6.12a are used to derive *g* and *p* signals for 4-bit blocks

Next, the *g* and *p* signals for 4-bit blocks are combined to form the desired carries, using the MCCs in Fig. 6.12b

Four-Bit Manchester Carry Chains

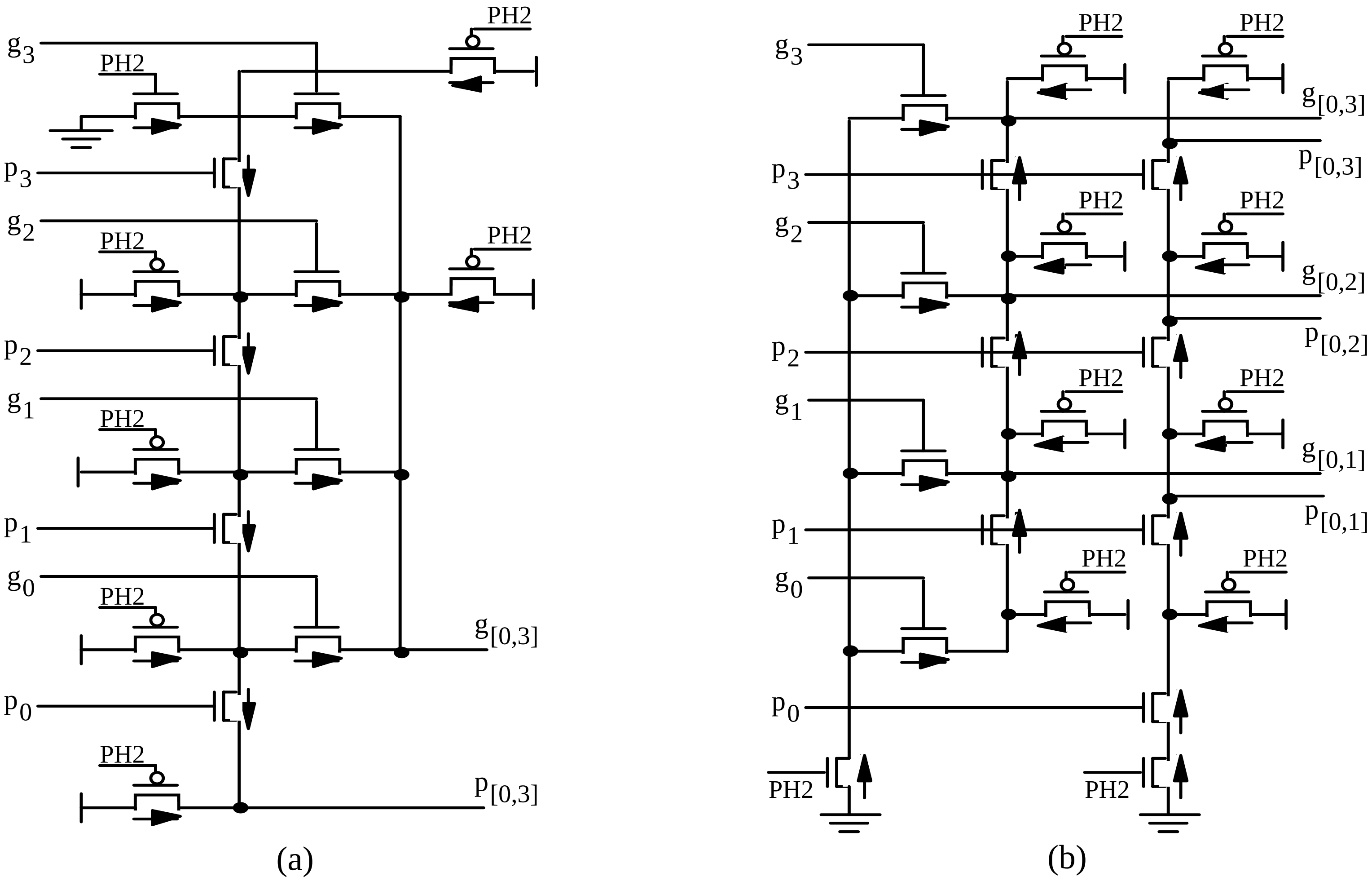


Fig. 6.12 Example 4-bit Manchester carry chain designs in CMOS technology [Lync92].

Carry Network for 64-Bit Adder

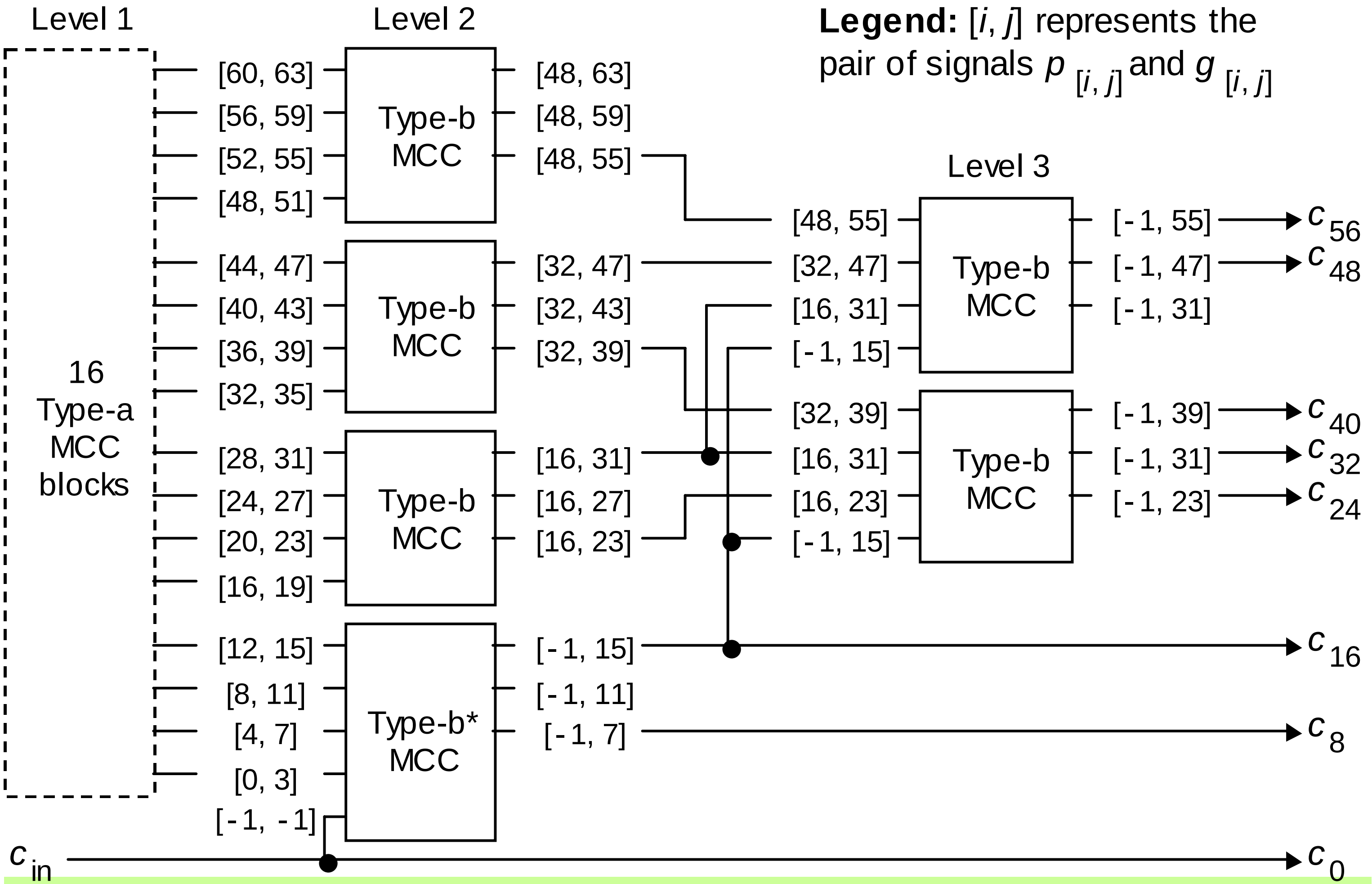


Fig. 6.13 Spanning-tree carry-lookahead network [Lync92]. Type-a and Type-b MCCs refer to the circuits of Figs. 6.12a and 6.12b, respectively.

7 Variations in Fast Adders

Chapter Goals

Study alternatives to the carry-lookahead method for designing fast adders

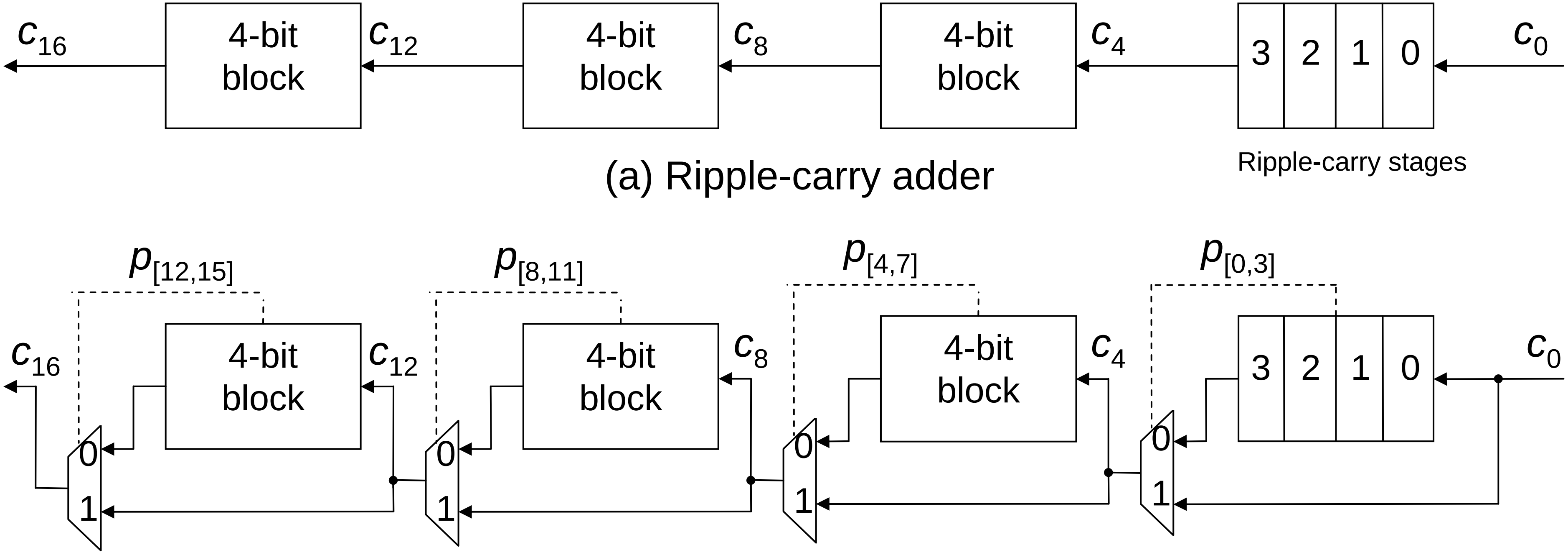
Chapter Highlights

Many methods besides CLA are available
(both competing and complementary)
Best design is technology-dependent
(often hybrid rather than pure)
Knowledge of timing allows optimizations

Variations in Fast Adders: Topics

Topics in This Chapter
7.1 Simple Carry-Skip Adders
7.2 Multilevel Carry-Skip Adders
7.3 Carry-Select Adders
7.4 Conditional-Sum Adder
7.5 Hybrid Designs and Optimizations
7.6 Modular Two-Operand Adders

7.1 Simple Carry-Skip Adders

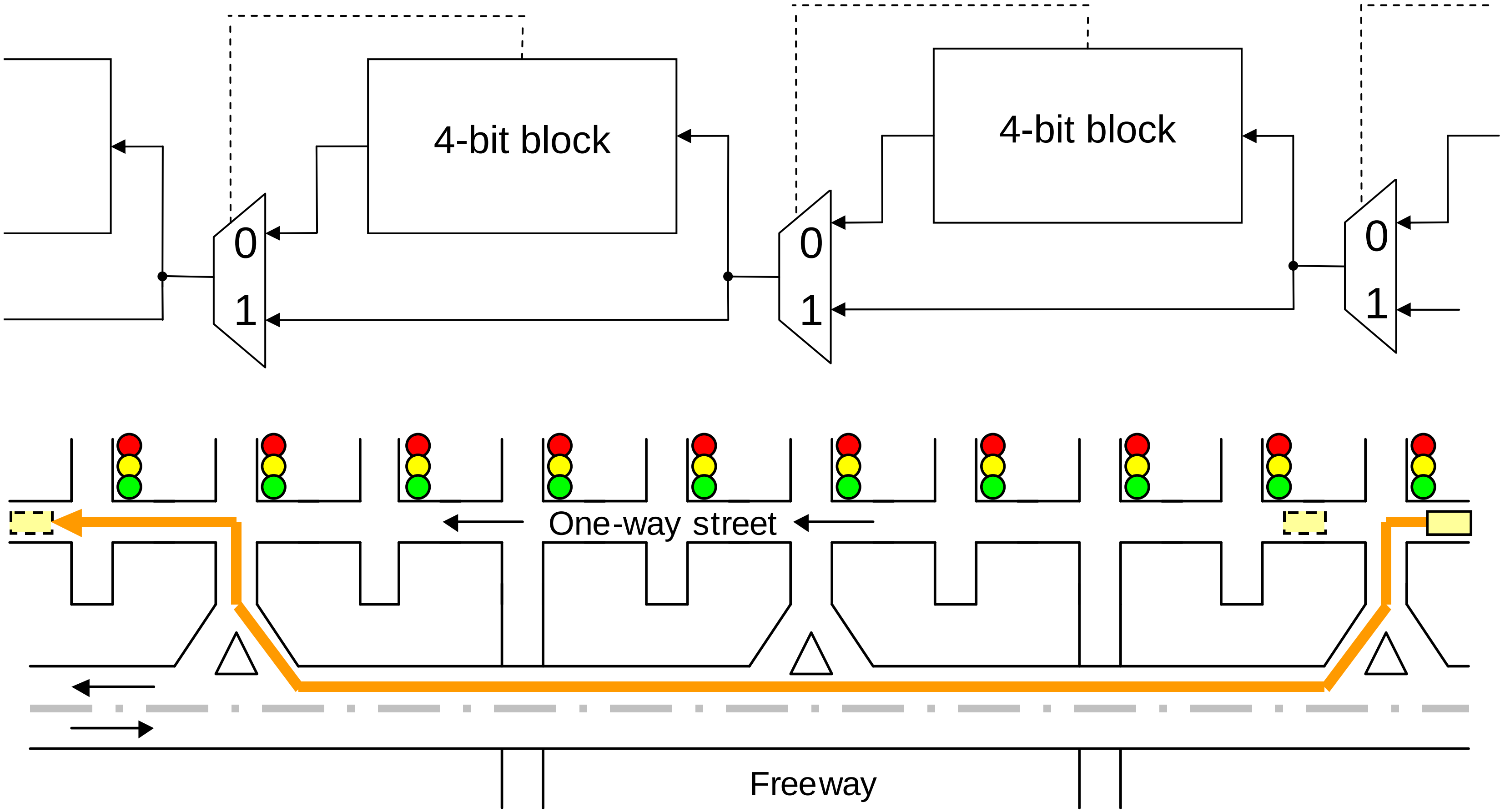


(a) Ripple-carry adder

(b) Simple carry-skip adder

Fig. 7.1 Converting a 16-bit ripple-carry adder into a simple carry-skip adder with 4-bit skip blocks.

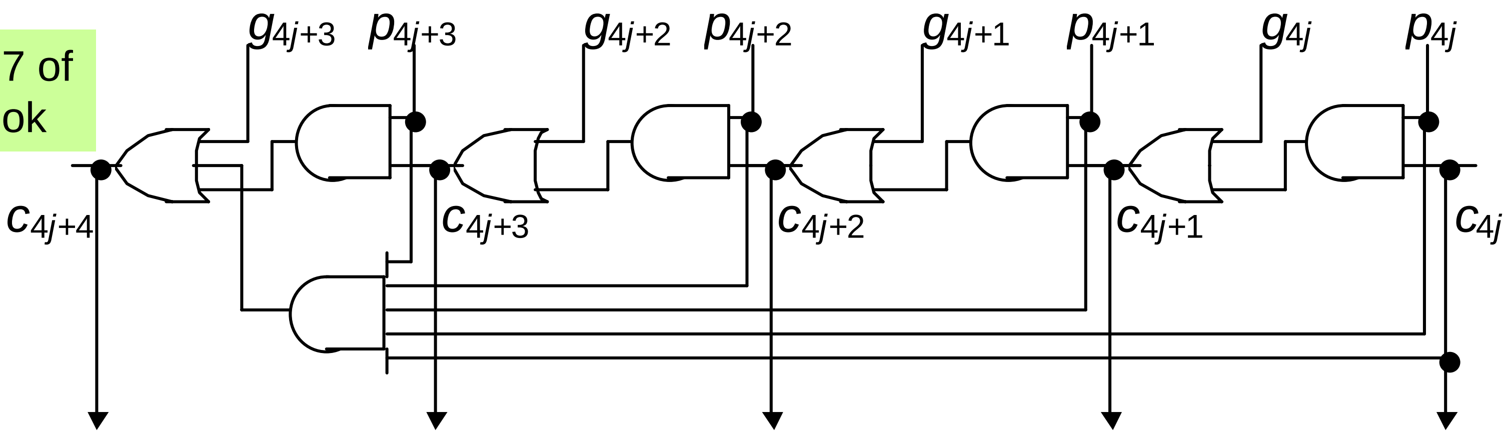
Another View of Carry-Skip Addition

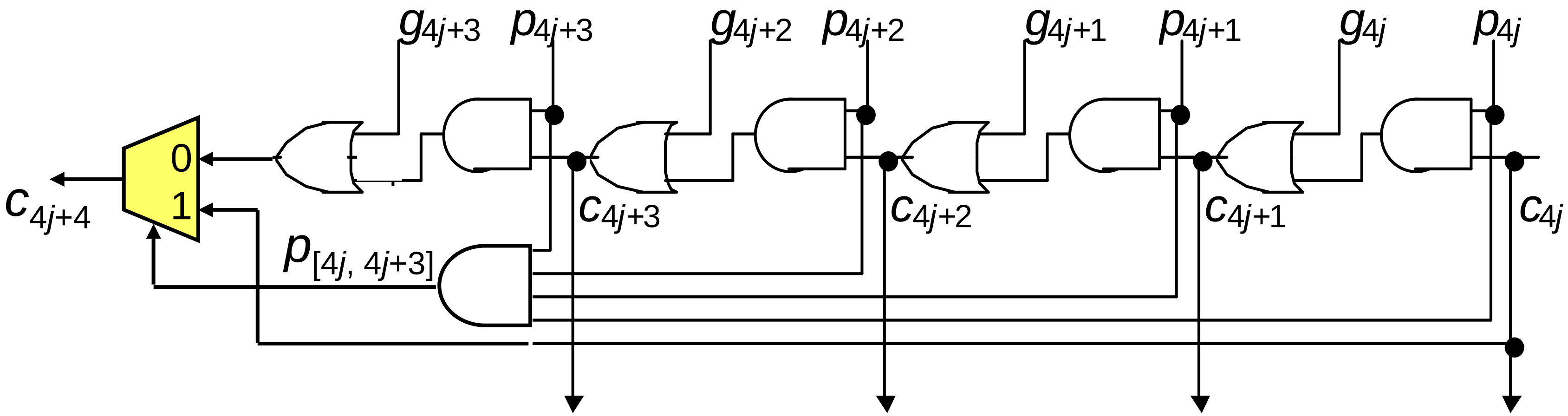


Street/freeway analogy for carry-skip adder.

Skip Carry Logic with OR Gate vs. Mux

Fig. 10.7 of arch book





The carry-skip adder with “OR combining” works fine if we begin with a clean slate, where all signals are 0s at the outset; otherwise, it will run into problems, which do not exist in mux-based version

Carry-Skip Adder with Fixed Block Size

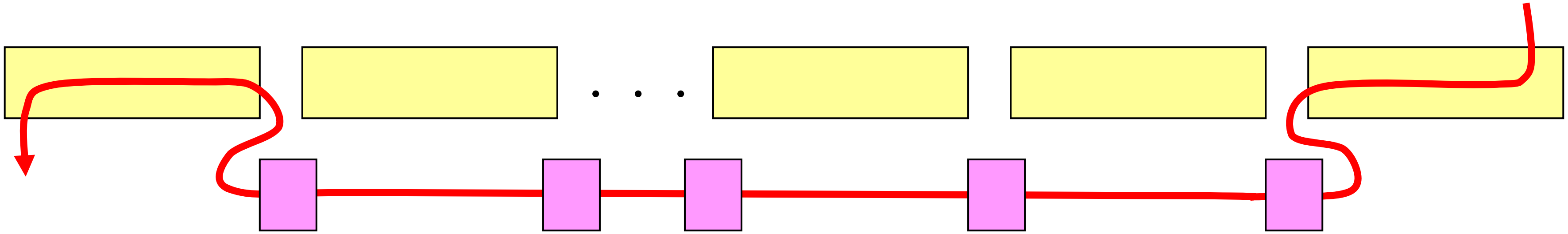
Block width b; k/b blocks to form a k-bit adder (assume b divides k)

$$T_{\text{fixed-skip-add}} = \underbrace{(b-1)}_{\text{in block 0}} + \underbrace{(k/b-1)}_{\text{skips}} + \underbrace{(b-1)}_{\text{in last block}}$$

$$\cong 2b + k/b - 3 \text{ stages}$$

$$dT/db = 2 - k/b^2 = 0 \quad \Rightarrow \quad b^{\text{opt}} = \sqrt{k/2}$$

$$T^{\text{opt}} = 2\sqrt{2k} - 3$$



Example: $k = 32$, $b^{\text{opt}} = 4$, $T^{\text{opt}} = 13$ stages

(contrast with 32 stages for a ripple-carry adder)

Carry-Skip Adder with Variable-Width Blocks

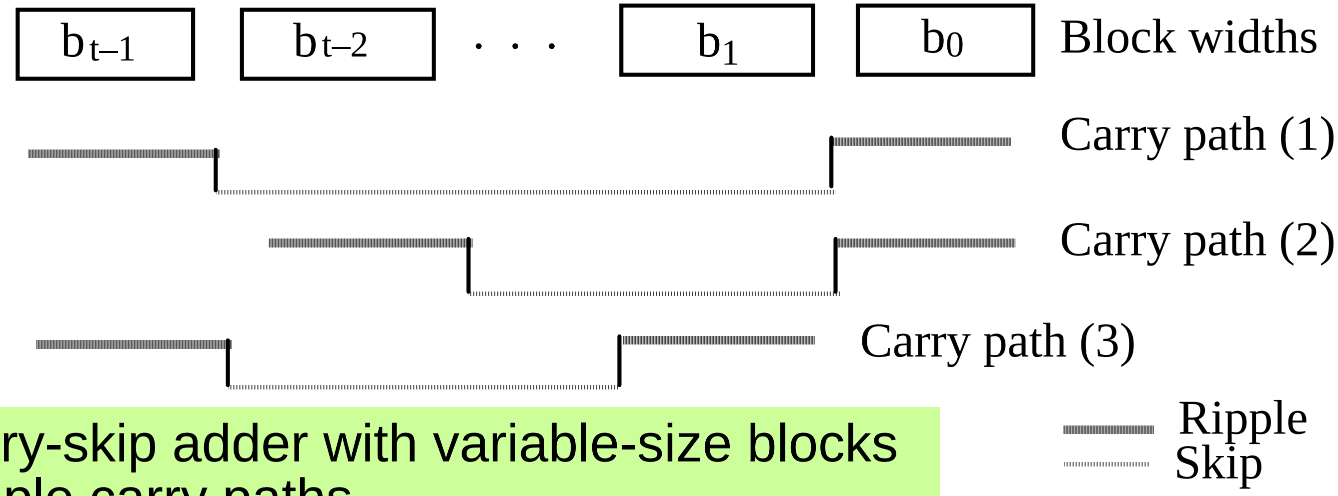


Fig. 7.2 Carry-skip adder with variable-size blocks and three sample carry paths.

The total number of bits in the *t* blocks is *k*:

$$2[b + (b + 1) + \ . \ . \ . + (b + t/2 - 1)] = t(b + t/4 - 1/2) = k$$

$$b = k/t - t/4 + 1/2$$

$$T_{\text{var-skip-add}} = 2(b - 1) + t - 1 = 2k/t + t/2 - 2$$

$$dT/db = -2k/t^2 + 1/2 = 0 \qquad \Rightarrow \qquad t^{\text{opt}} = 2\sqrt{k}$$

$$T^{\text{opt}} = 2\sqrt{k} - 2$$ (a factor of $\sqrt{2}$ smaller than for fixed-block)

7.2 Multilevel Carry-Skip Adders

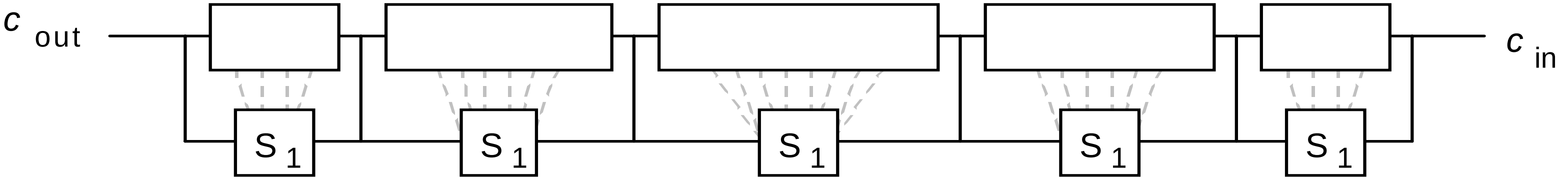


Fig. 7.3 Schematic diagram of a one-level carry-skip adder.

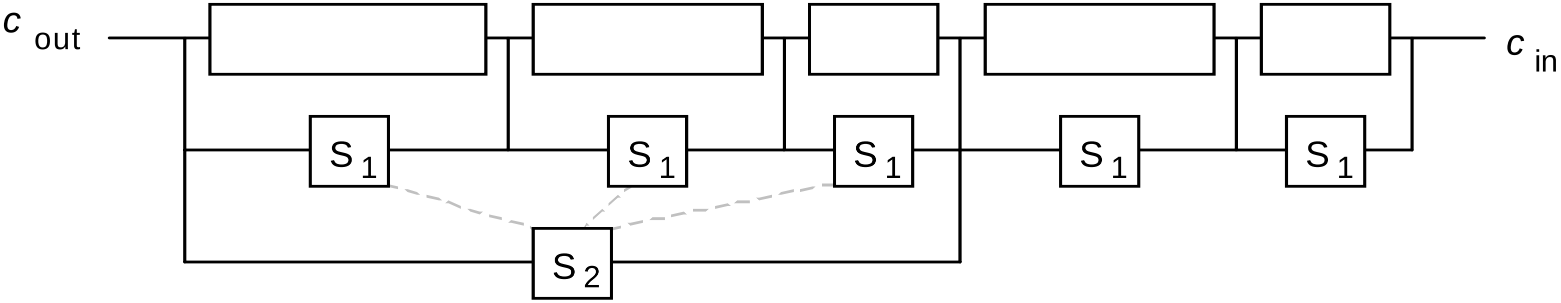


Fig. 7.4 Example of a two-level carry-skip adder.

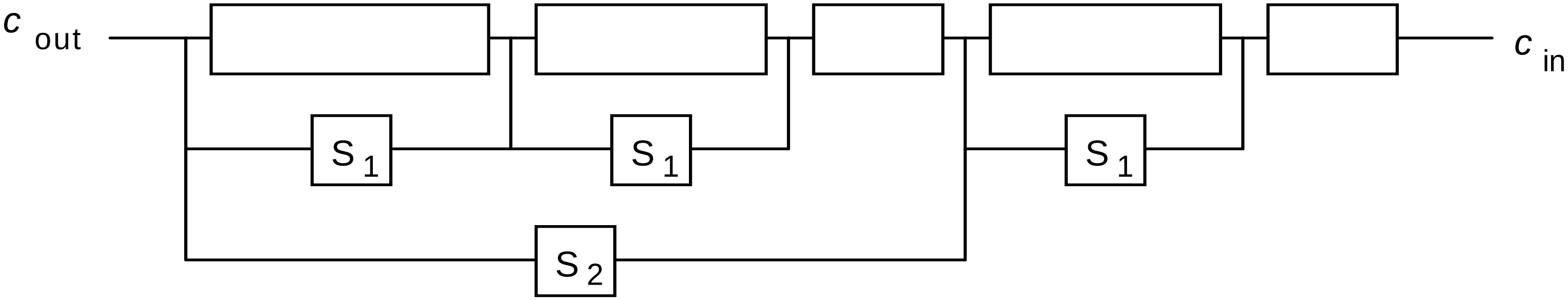


Fig. 7.5 Two-level carry-skip adder optimized by removing the short-block skip circuits.

Designing a Single-Level Carry-Skip Adder

Example 7.1

Each of the following takes one unit of time: generation of g_i and p_i, generation of level-i skip signal from level-(i–1) skip signals, ripple, skip, and formation of sum bit once the incoming carry is known

Build the widest possible one-level carry-skip adder with total delay of 8

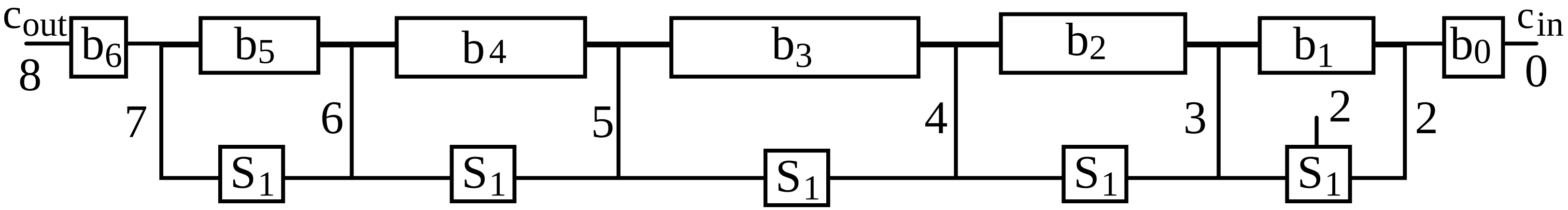


Fig. 7.6 Timing constraints of a single-level carry-skip adder with a delay of 8 units.

Max adder width = 18
(1 + 2 + 3 + 4 + 4 + 3 + 1)

Generalization of Example 7.1 for total time T (even or odd)

1 2 3 . . . $T/2$ $T/2$ . . . 4 3 1

1 2 3 . . . $(T + 1)/2$ . . . 4 3 1

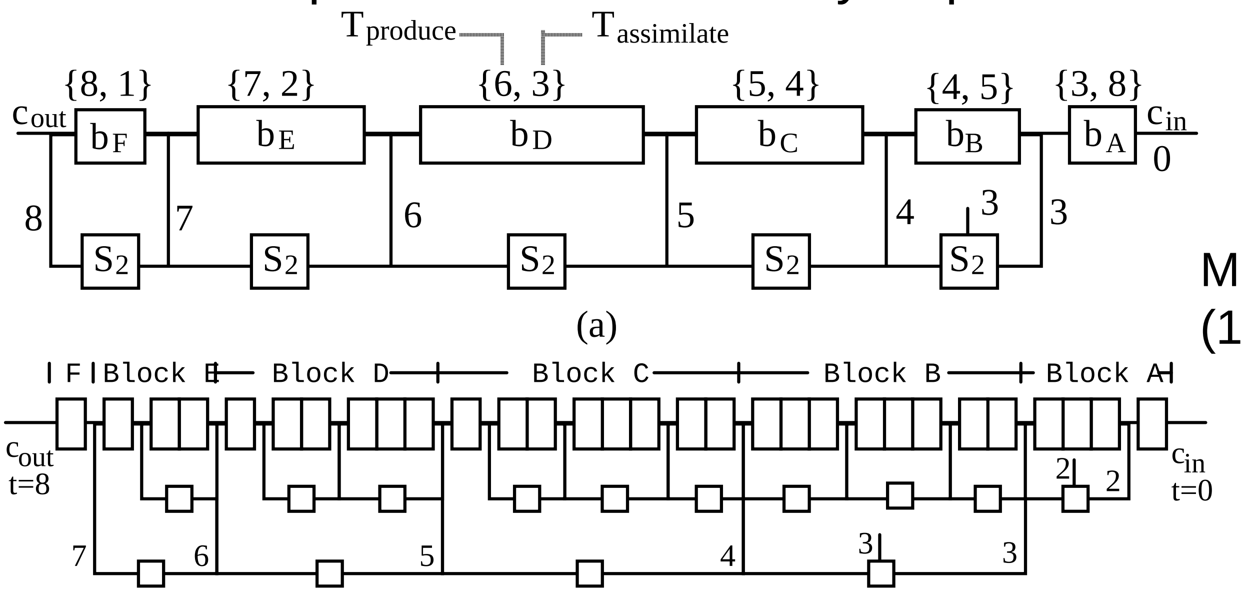
Thus, for any T, the total width is $\lfloor (T + 1)^2/4 \rfloor - 2$

Designing a Two-Level Carry-Skip Adder

Example 7.2

Each of the following takes one unit of time: generation of g_i and p_i, generation of level-i skip signal from level-(i–1) skip signals, ripple, skip, and formation of sum bit once the incoming carry is known

Build the widest possible two-level carry-skip adder with total delay of 8



(a) Initial timing constraints

Max adder width = 30
(1 + 3 + 6 + 8 + 8 + 4)

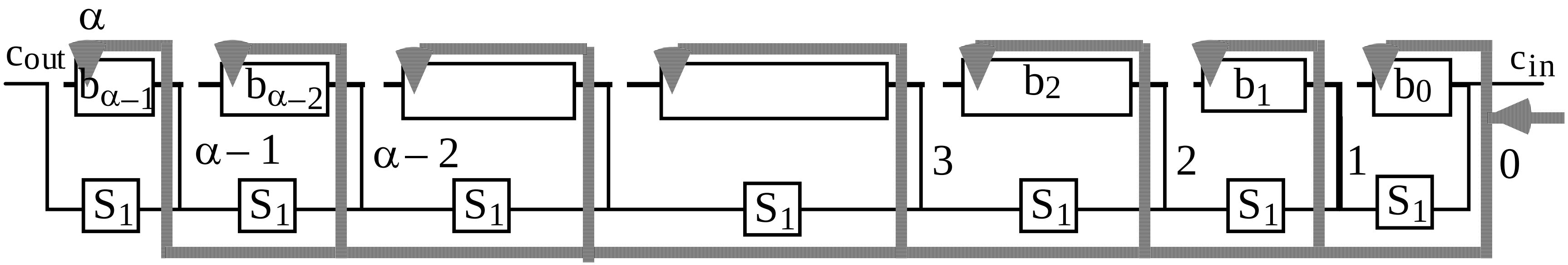
(b) Final design

Fig. 7.7 Two-level carry-skip adder with a delay of 8 units.

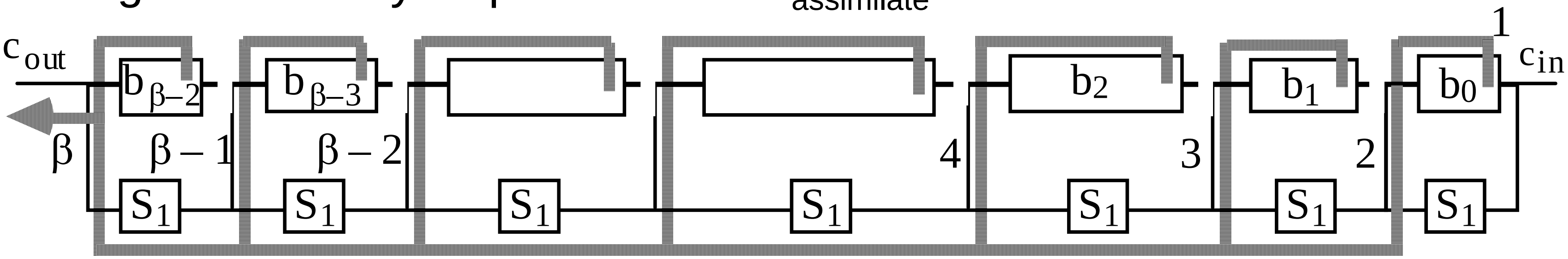
Elaboration on Two-Level Carry-Skip Adder

Example 7.2

Given the delay pair {β, α} for a level-2 block in Fig. 7.7a, the number of level-1 blocks that can be accommodated is $\gamma = min(\beta - 1, \alpha)$



Single-level carry-skip adder with $T_{assimilate} = \alpha$



Single-level carry-skip adder with $T_{produce} = \beta$

Width of the *i*th level-1 block in the level-2 block characterized by {β, α} is $b_i = min(\beta - \gamma + i + 1, \alpha - i)$; the total block width is then $\sum_{i=0 \text{ to } \gamma-1} b_i$

Carry-Skip Adder Optimization Scheme

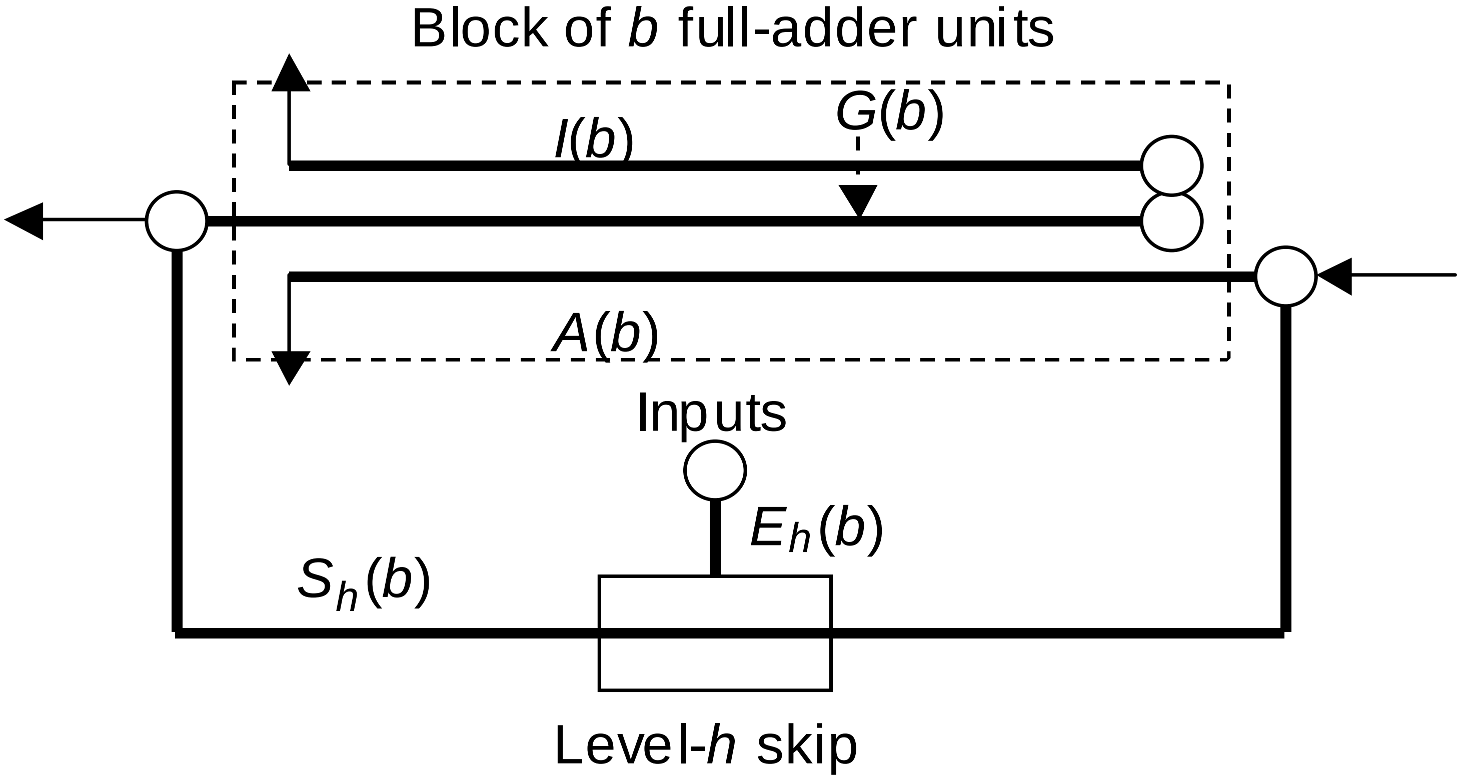


Fig. 7.8 Generalized delay model for carry-skip adders.

7.3 Carry-Select Adders

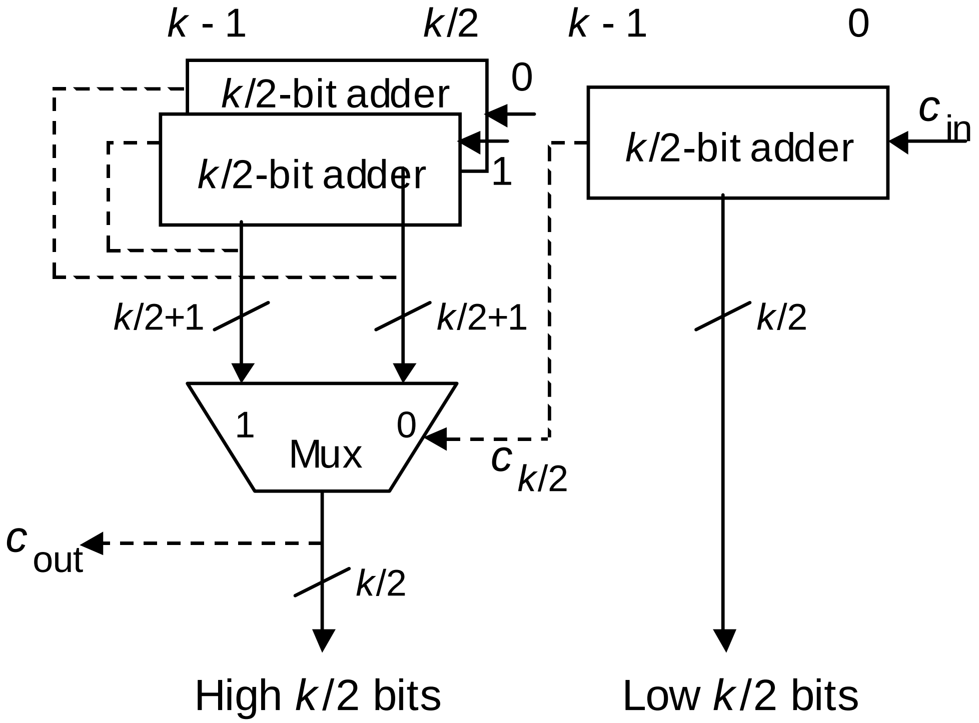


Fig. 7.9 Carry-select adder for *k*-bit numbers built from three *k*/2-bit adders.

$$C_{\text{select-add}}(k) = 3C_{\text{add}}(k/2) + k/2 + 1$$

$$T_{\text{select-add}}(k) = T_{\text{add}}(k/2) + 1$$

Multilevel Carry-Select Adders

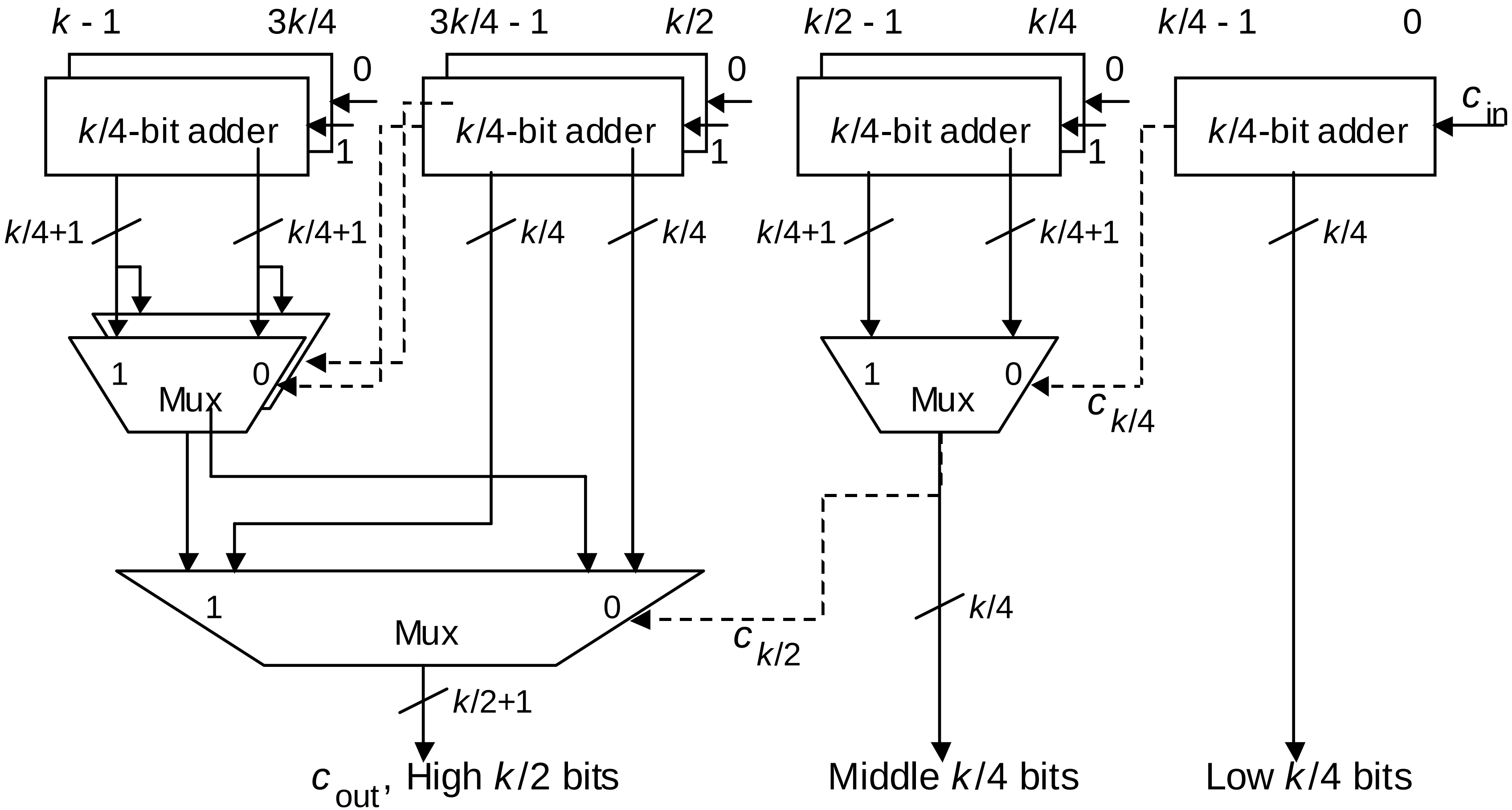


Fig. 7.10 Two-level carry-select adder built of *k*/4-bit adders.

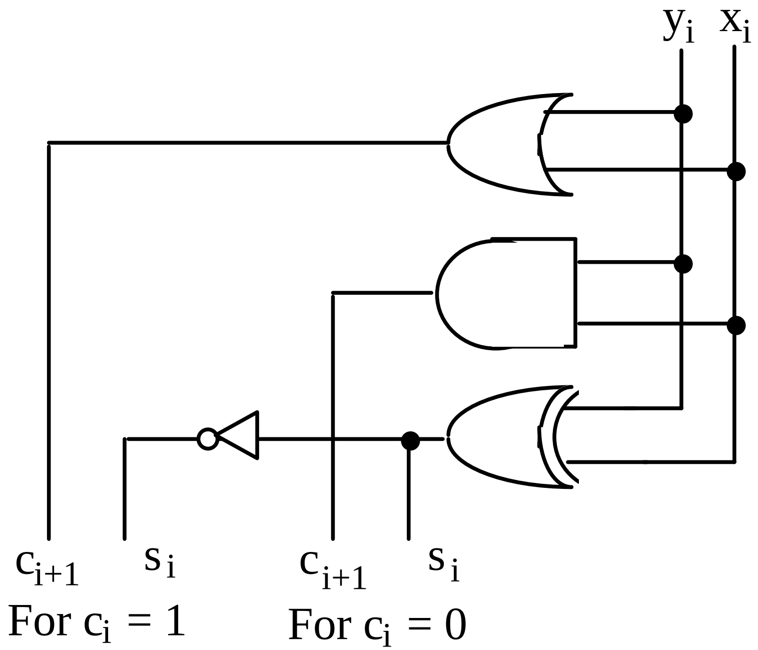
7.4 Conditional-Sum Adder

Multilevel carry-select idea carried out to the extreme (to 1-bit blocks.

$$C(k) \cong 2C(k/2) + k + 2 \cong k(\log_2 k + 2) + k\,C(1)$$

$$T(k) = T(k/2) + 1 = \log_2 k + T(1)$$

where $C(1)$ and $T(1)$ are the cost and delay of the circuit of Fig. 7.11 for deriving the sum and carry bits with a carry-in of 0 and 1



$k + 2$ is an upper bound on number of single-bit 2-to-1 multiplexers needed for combining two $k/2$-bit adders into a k-bit adder

Fig. 7.11 Top-level block for one bit position of a conditional-sum adder.

Conditional-Sum Addition Example

Table 7.2

Conditional-sum addition of two 16-bit numbers. The width of the block for which the sum and carry bits are known doubles with each additional level, leading to an addition time that grows as the logarithm of the word width *k*.

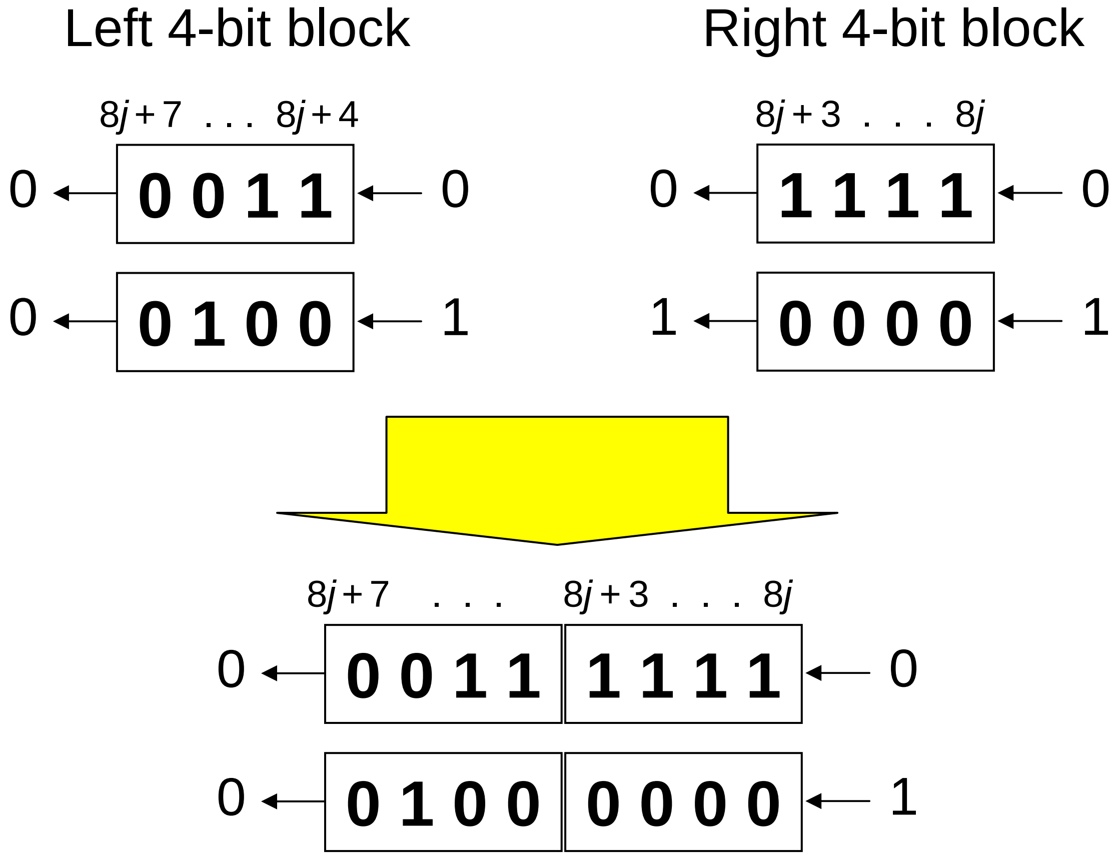
		x	0	0	1	0	0	1	1	0	1	1	1	0	1	0	1	0	
		y	0	1	0	0	1	0	1	1	0	1	0	1	1	1	0	1	
Block width	Block carry-in	Block sum and block carry-out																	c_{in}
			15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	s	0	1	1	0	1	1	0	1	1	0	1	1	0	1	1	1	
		c	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0
	1	s	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0		
		c	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1		
2	0	s	0	1	1	0	1	1	0	1	0	0	1	1	0	1	1	1	
		c	0		0		0		1		1		0		1		0		
	1	s	1	0	1	1	0	0	1	0	0	1	0	0	1	0			
		c	0		0		1		1		1		1		1				
4	0	s	0	1	1	0	0	0	0	1	0	0	1	1	0	1	1	1	
		c	0				1				1				1				
	1	s	0	1	1	1	0	0	1	0	0	1	0	0					
		c	0				1				1								
8	0	s	0	1	1	1	0	0	0	1	0	1	0	0	0	1	1	1	
		c	0								1								
	1	s	0	1	1	1	0	0	1	0									
		c	0																
16	0	s	0	1	1	1	0	0	1	0	0	1	0	0	0	1	1	1	
		c	0																
	1	s																	
		c																	

c_{out}

Elaboration on Conditional-Sum Addition

Two adjacent 4-bit blocks, forming an 8-bit block

Two versions of sum bits and carry-out in 4-bit blocks



Left 4-bit block

$8j + 7 \; . . . \; 8j + 4$

0 ← **0 0 1 1** ← 0

0 ← **0 1 0 0** ← 1

Right 4-bit block

$8j + 3 \; . \; . \; . \; 8j$

0 ← **1 1 1 1** ← 0

1 ← **0 0 0 0** ← 1

$8j + 7 \; . \; . \; . \; 8j + 3 \; . \; . \; . \; 8j$

0 ← **0 0 1 1 1 1 1 1** ← 0

0 ← **0 1 0 0 0 0 0 0** ← 1

Two versions of sum bits and carry-out in 8-bit block

7.5 Hybrid Designs and Optimizations

The most popular hybrid addition scheme:

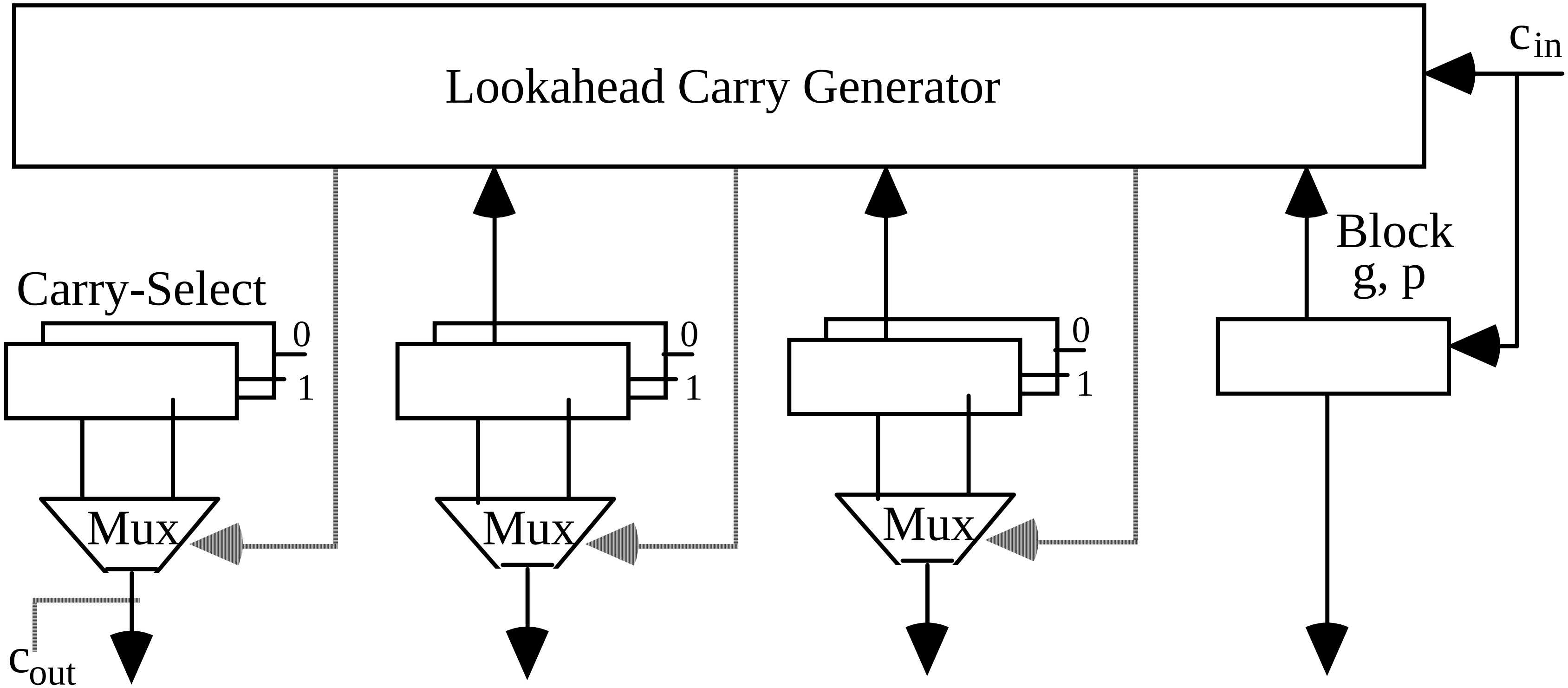
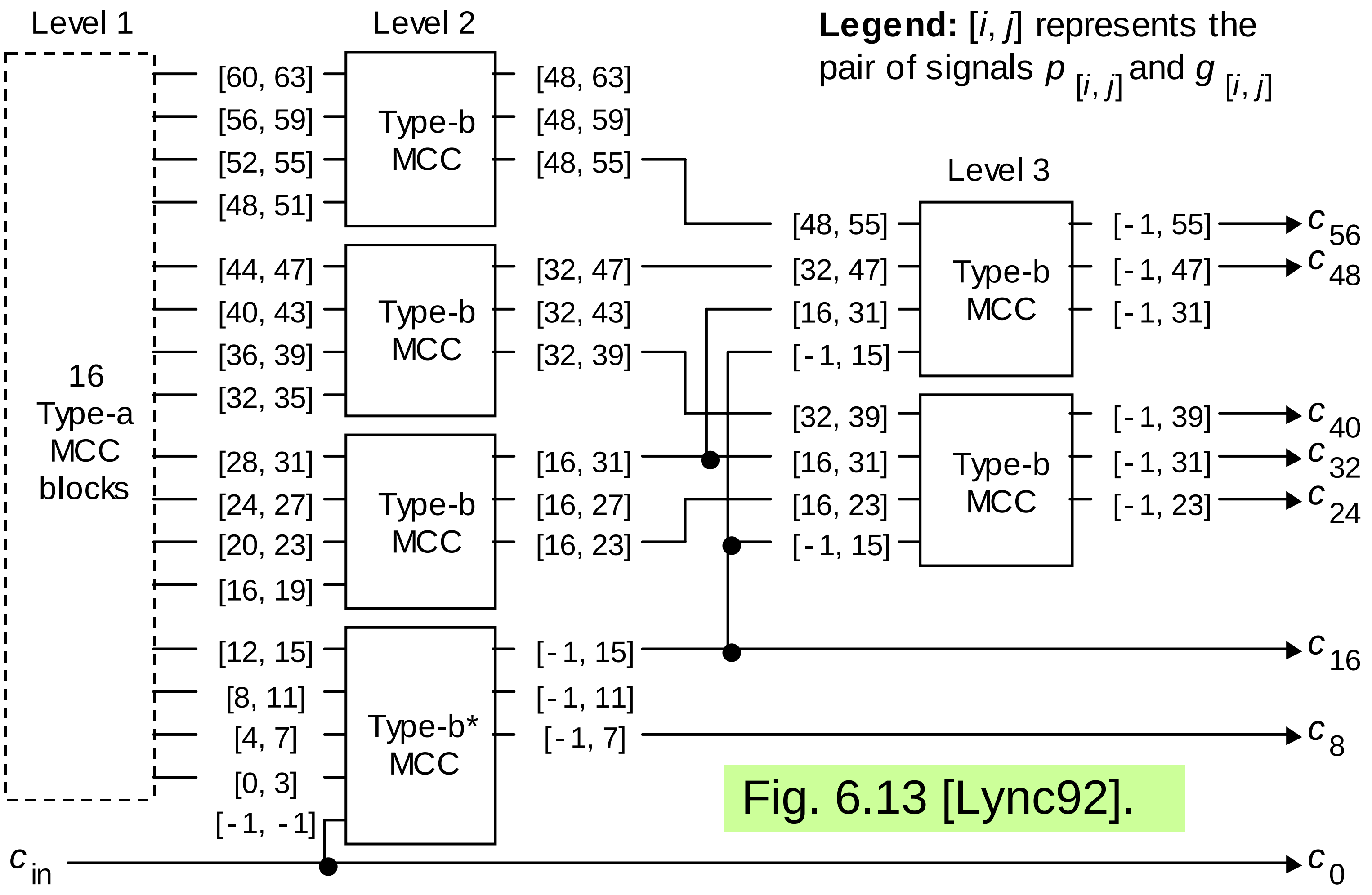


Fig. 7.12 A hybrid carry-lookahead/carry-select adder.

Details of a 64-Bit Hybrid CLA/Select Adder



Legend: [*i*, *j*] represents the pair of signals $p_{[i, j]}$ and $g_{[i, j]}$

Fig. 6.13 [Lync92].

Each of the carries c_{8j}, produced by the tree network above is used to select one of the two versions of the sum in positions 8*j* to 8*j* + 7

Any Two Addition Schemes Can Be Combined

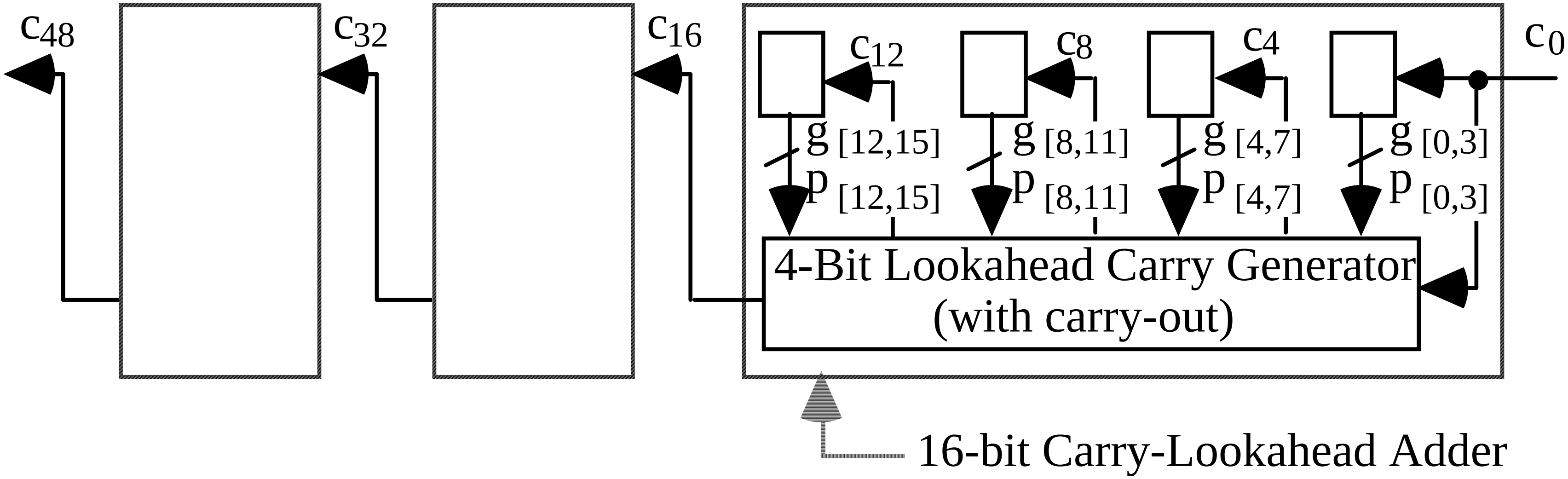


Fig. 7.13 Example 48-bit adder with hybrid ripple-carry/carry-lookahead design.

Other possibilities: hybrid carry-select/ripple-carry
hybrid ripple-carry/carry-select
. . .

Optimizations in Fast Adders

What looks best at the block diagram or gate level may not be best when a circuit-level design is generated (effects of wire length, signal loading, . . .)

Modern practice: Optimization at the transistor level

Variable-block carry-lookahead adder

Optimizations for average or peak power consumption

Timing-based optimizations (next slide)

Optimizations Based on Signal Timing

So far, we have assumed that all input bits are presented at the same time and all output bits are also needed simultaneously

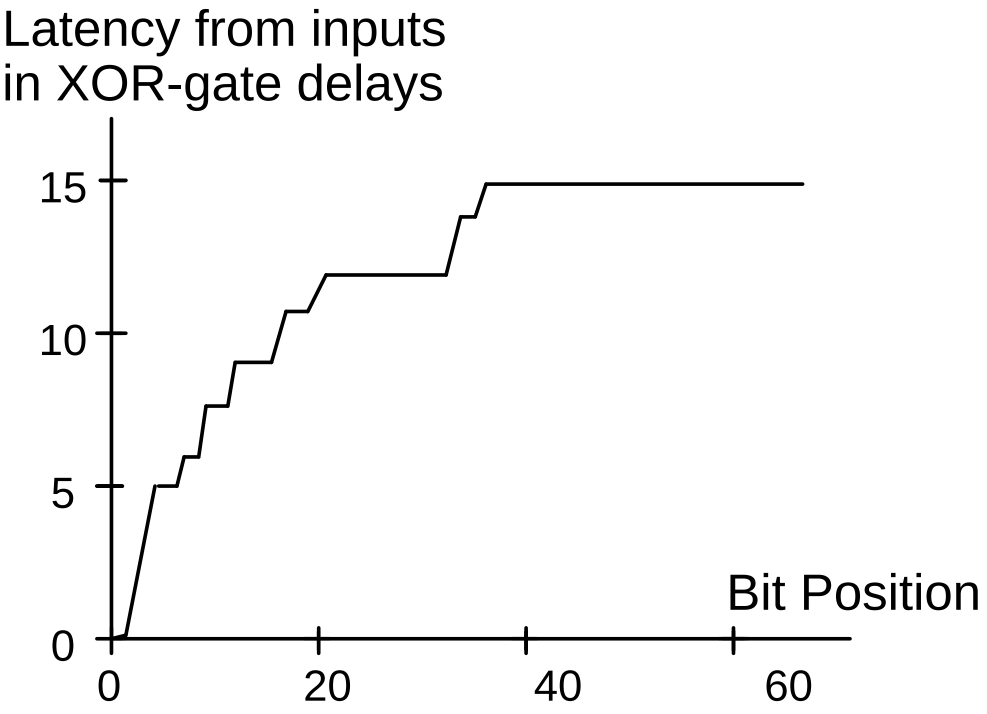
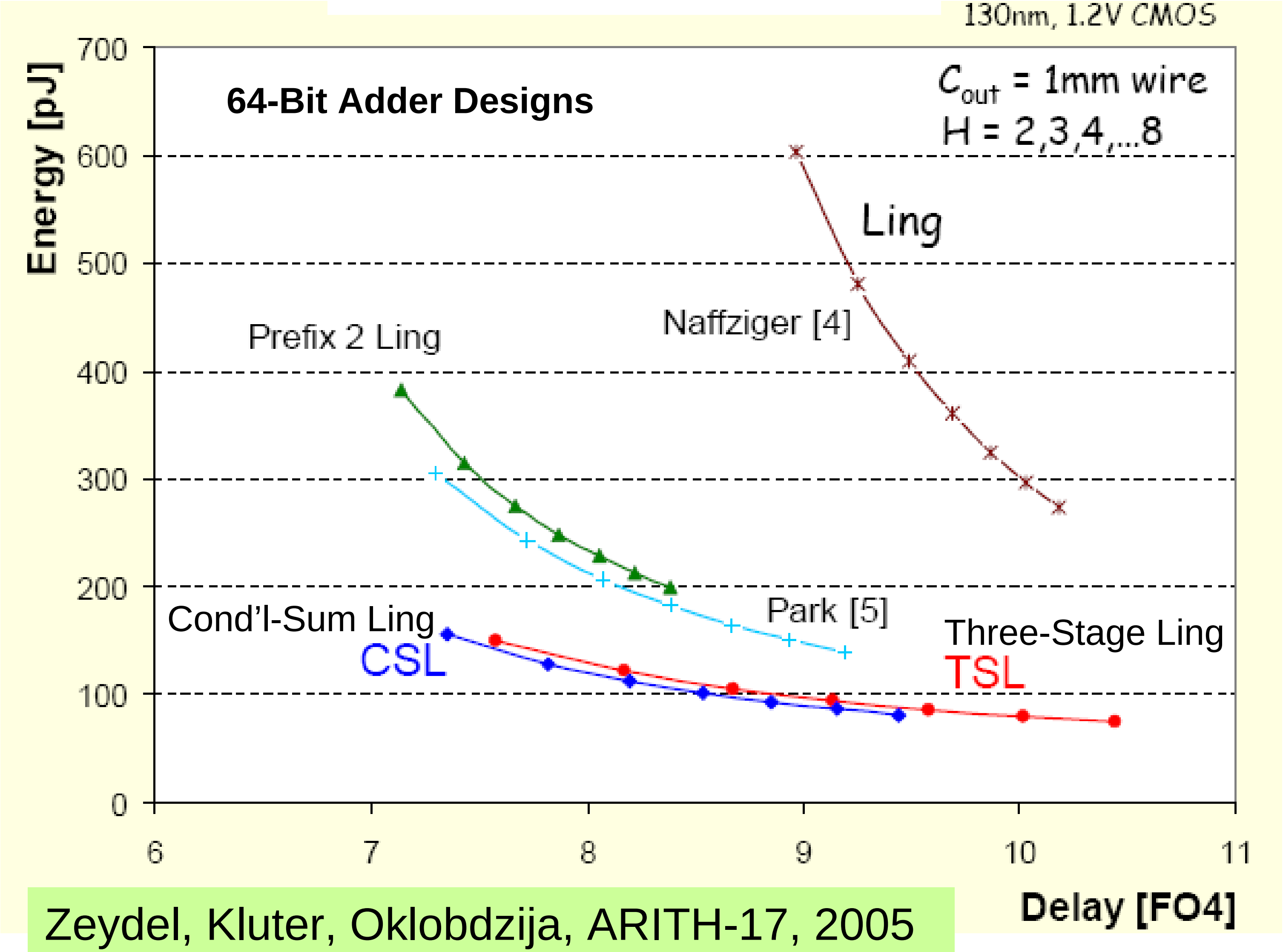


Fig. 7.14 Example arrival times for operand bits in the final fast adder of a tree multiplier [Oklo96].

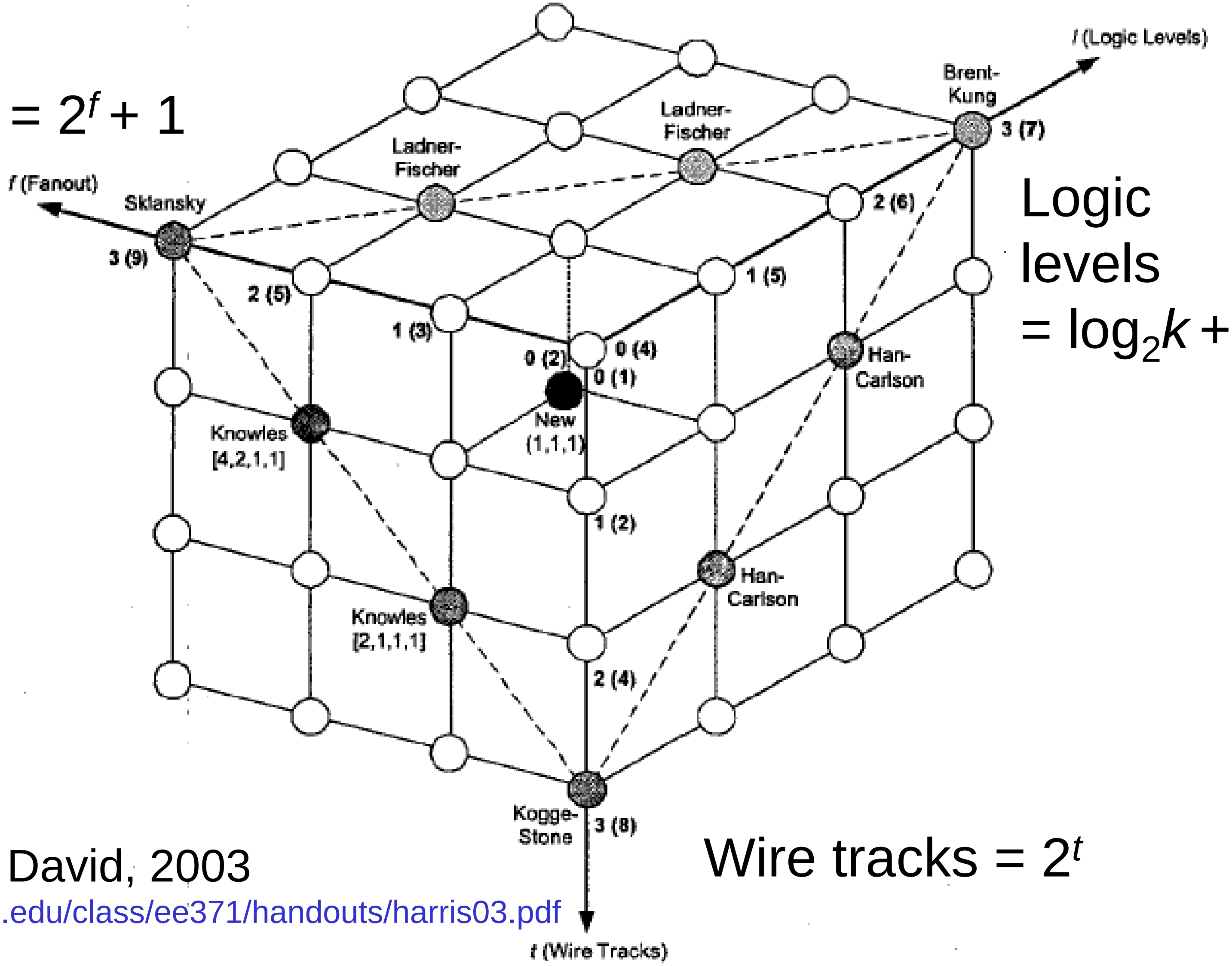
Modern Low-Power Adders Implemented in CMOS



Zeydel, Kluter, Oklobdzija, ARITH-17, 2005

Taxonomy of Parallel Prefix Networks

Fanout = $2^f + 1$



Logic levels = $\log_2 k + l$

Wire tracks = 2^t

From: Harris, David, 2003

http://www.stanford.edu/class/ee371/handouts/harris03.pdf

7.6 Modular Two-Operand Adders

mod-2^k: Ignore carry out of position $k - 1$

mod-$(2^k - 1)$: Use end-around carry because $2^k = (2^k - 1) + 1$

mod-$(2^k + 1)$: Residue representation needs $k + 1$ bits

Number	Std. binary	Diminished-1
0	0 0 . . . 0 0 0	1 x . . . x x x
1	0 0 . . . 0 0 1	0 0 . . . 0 0 0
2	0 0 . . . 0 1 0	0 0 . . . 0 0 1
.	.	.
.	.	.
.	.	.
$2^k–1$	0 1 . . . 1 1 1	0 1 . . . 1 1 0
2^k	1 0 . . . 0 0 0	0 1 . . . 1 1 1

$x + y \geq 2^k + 1$ iff

$(x–1) + (y–1) + 1 \geq 2^k$

$(x + y) - 1 =$

$(x - 1) + (y - 1) + 1$

$xy - 1 =$

$(x–1)(y–1)+(x–1)+(y–1)$

General Modular Adders

$(x + y) \bmod m$

if $x + y \geq m$
then $x + y - m$
else $x + y$

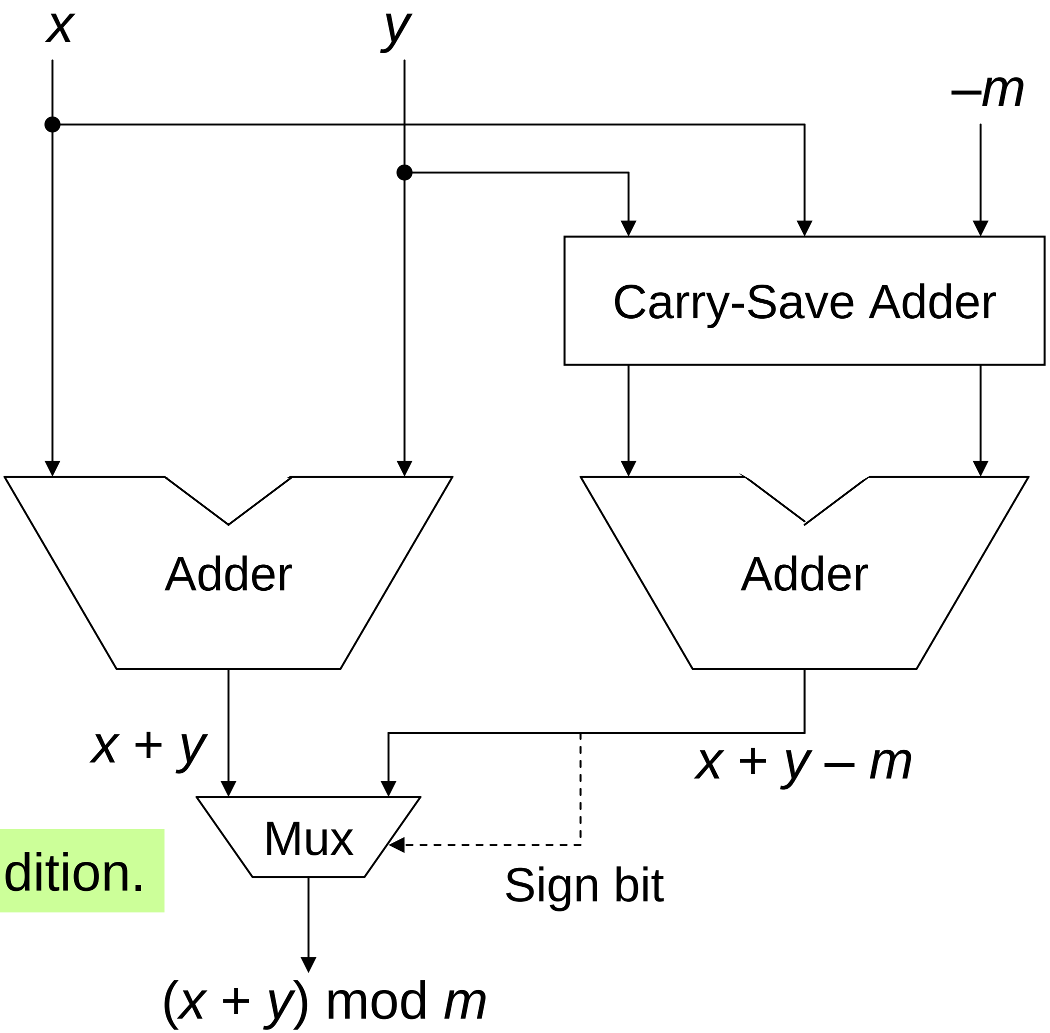


Fig. 7.15 Fast modular addition.

8 Multioperand Addition

Chapter Goals

Learn methods for speeding up the addition of several numbers (needed for multiplication or inner-product)

Chapter Highlights

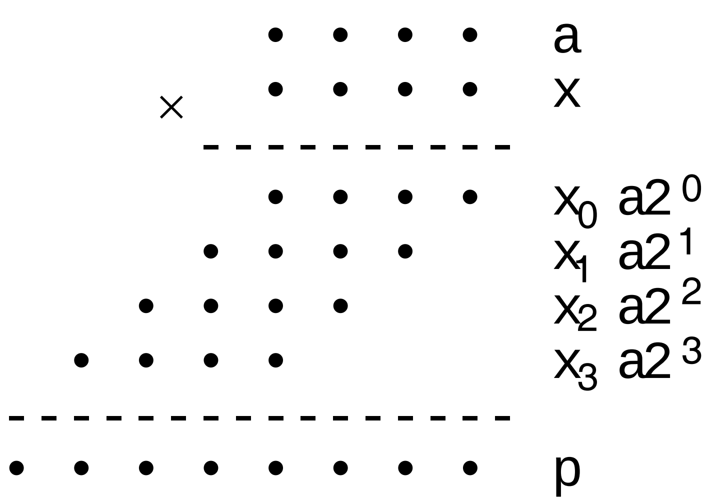
Running total kept in redundant form
Current total + Next number → New total
Deferred carry assimilation
Wallace/Dadda trees, parallel counters
Modular multioperand addition

Multioperand Addition: Topics

Topics in This Chapter
8.1 Using Two-Operand Adders
8.2 Carry-Save Adders
8.3 Wallace and Dadda Trees
8.4 Parallel Counters and Compressors
8.5 Adding Multiple Signed Numbers
8.6 Modular Multioperand Adders

8.1 Using Two-Operand Adders

Some applications of multioperand addition



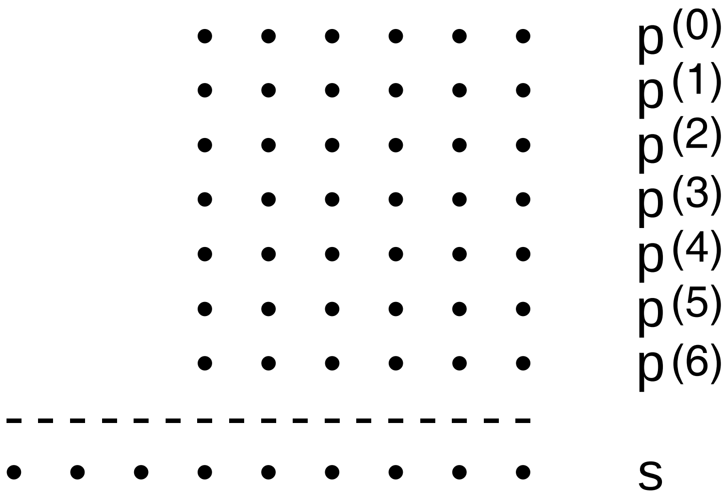


Fig. 8.1 Multioperand addition problems for multiplication or inner-product computation in dot notation.

Serial Implementation with One Adder

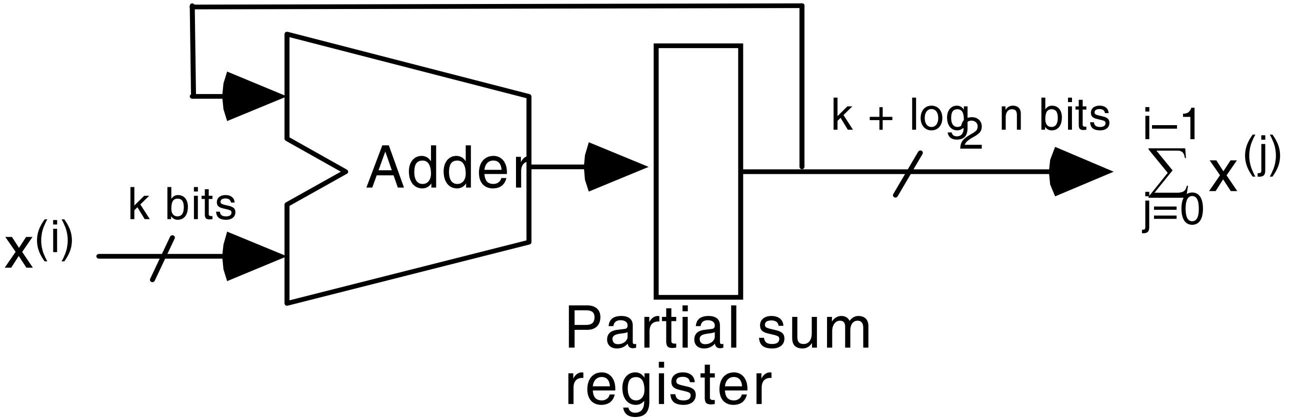


Fig. 8.2 Serial implementation of multioperand addition with a single 2-operand adder.

$$T_{\text{serial-multi-add}} = O(n \log(k + \log n))$$
$$= O(n \log k + n \log\log n)$$

Therefore, addition time grows superlinearly with *n* when *k* is fixed and logarithmically with *k* for a given *n*

Pipelined Implementation for Higher Throughput

Problem to think about: Ignoring start-up and other overheads, this scheme achieves a speedup of 4 with 3 adders. How is this possible?

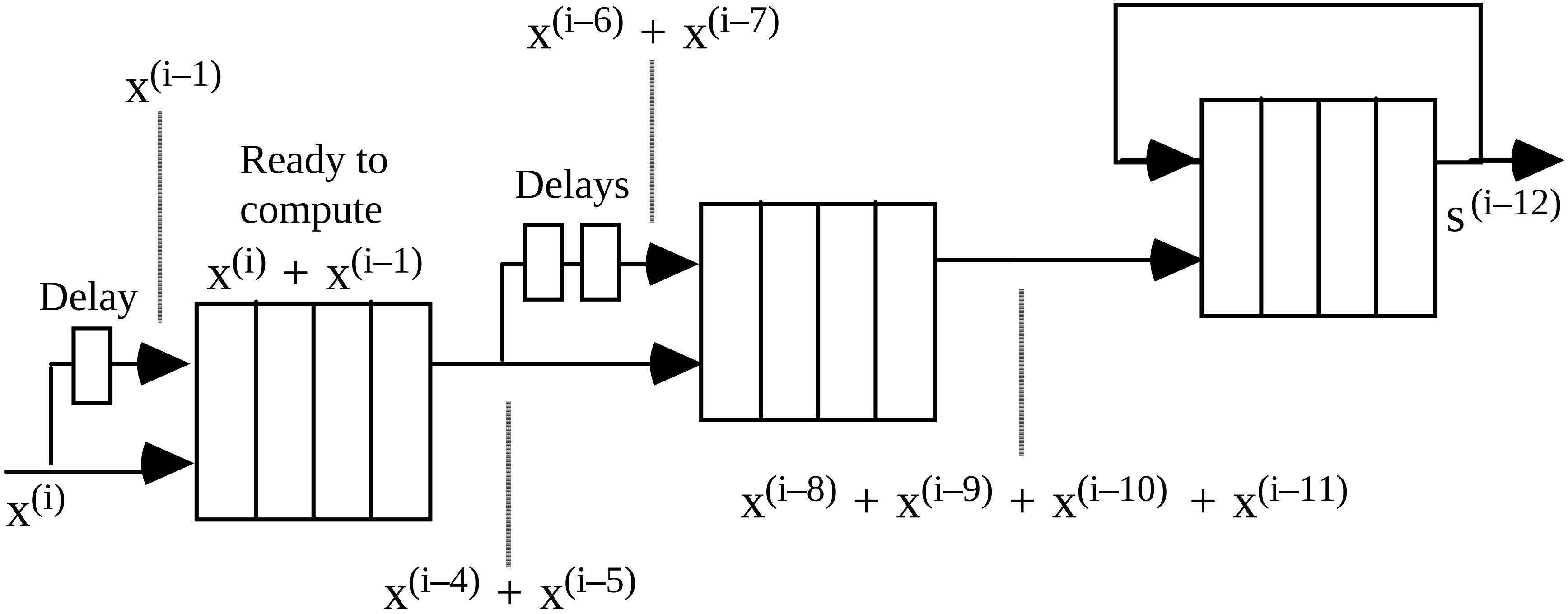


Fig. 8.3 Serial multioperand addition when each adder is a 4-stage pipeline.

Parallel Implementation as Tree of Adders

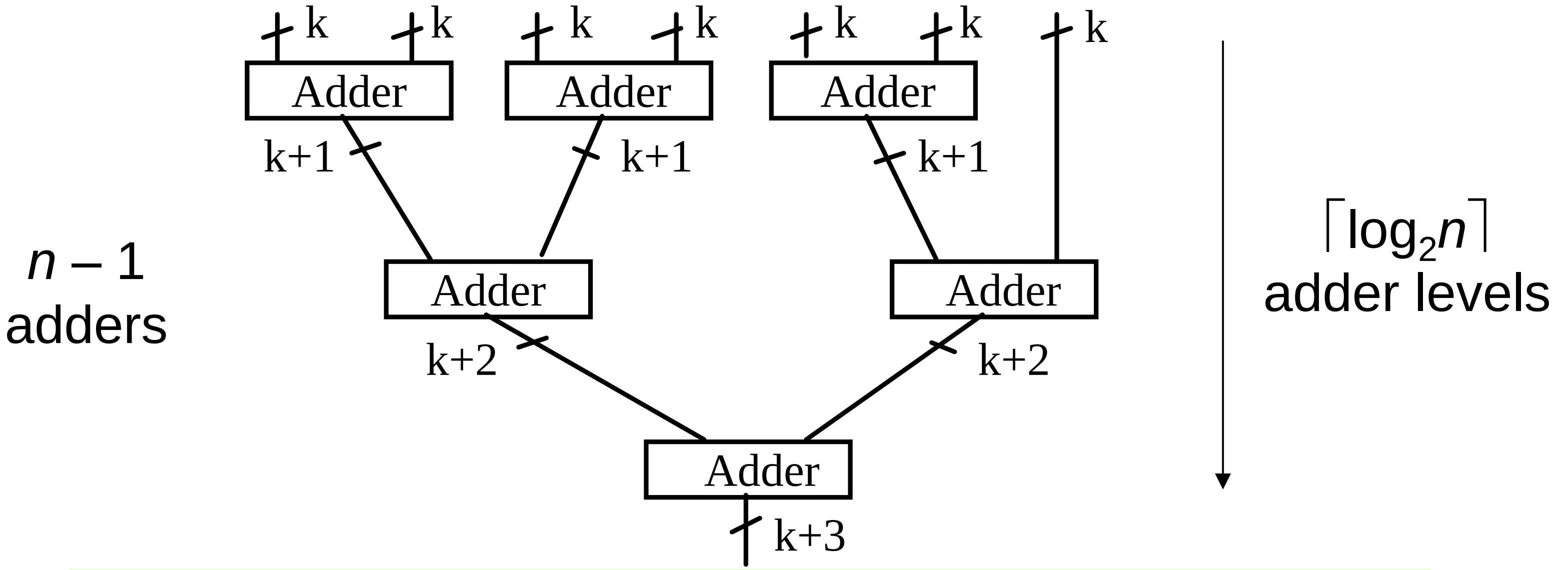


Fig. 8.4 Adding 7 numbers in a binary tree of adders.

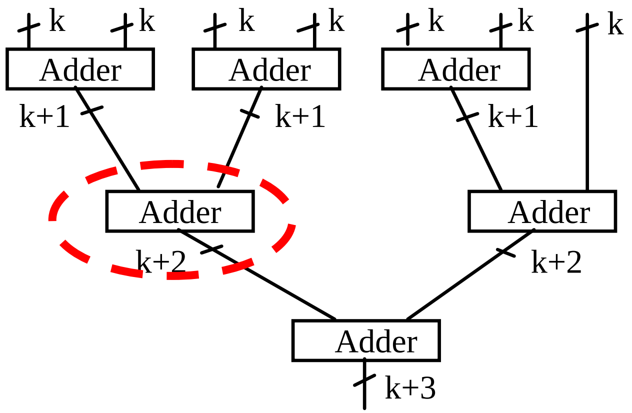
$$T_{\text{tree-fast-multi-add}} = O(\log k + \log(k + 1) + \ldots + \log(k + \lceil \log_2 n \rceil - 1))$$

$$= O(\log n \log k + \log n \log \log n)$$

$$T_{\text{tree-ripple-multi-add}} = O(k + \log n)$$

[Justified on the next slide]

Elaboration on Tree of Ripple-Carry Adders



$T_{\text{tree-ripple-multi-add}} = \mathrm{O}(k + \log n)$

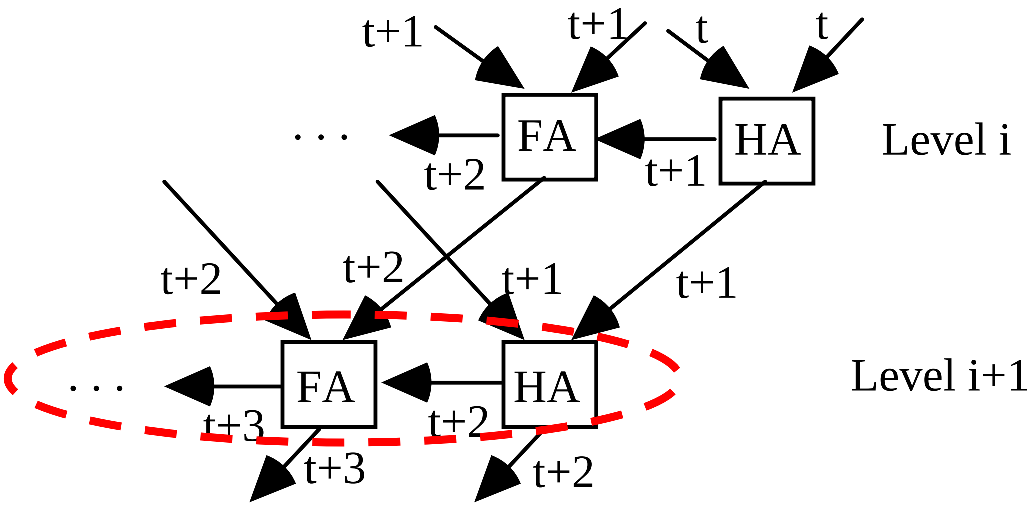


Fig. 8.5 Ripple-carry adders at levels *i* and *i* + 1 in the tree of adders used for multi-operand addition.

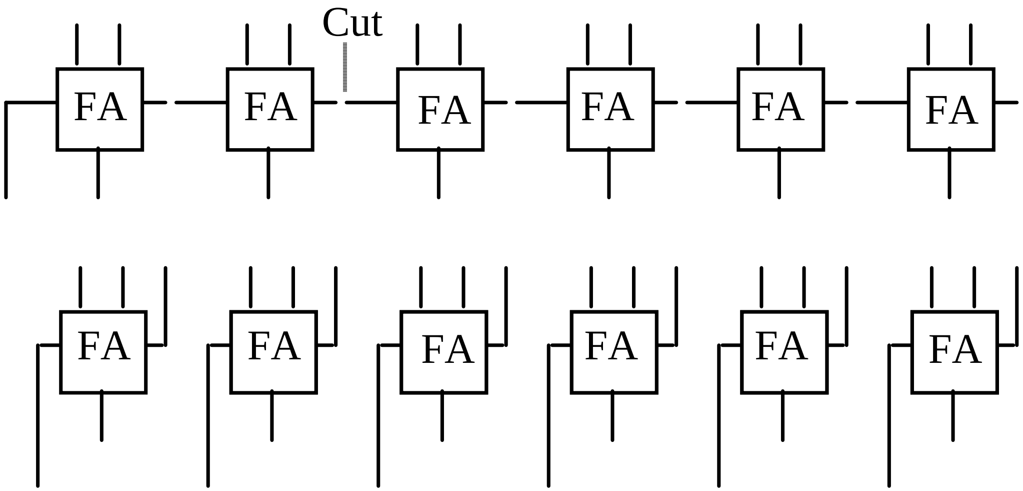
The absolute best latency that we can hope for is O(log *k* + log *n*)

There are *kn* data bits to process and using any set of computation elements with constant fan-in, this requires O(log(*kn*)) time

We will see shortly that carry-save adders achieve this optimum time

8.2 Carry-Save Adders

Fig. 8.6 A ripple-carry adder turns into a carry-save adder if the carries are saved (stored) rather than propagated.



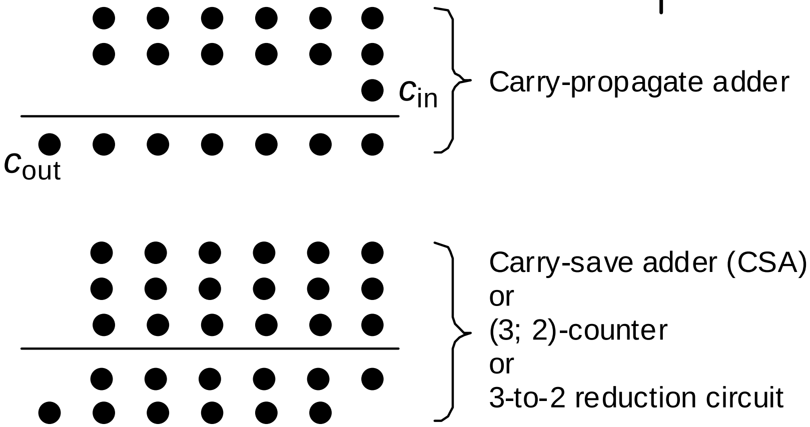


Fig. 8.7 Carry-propagate adder (CPA) and carry-save adder (CSA) functions in dot notation.

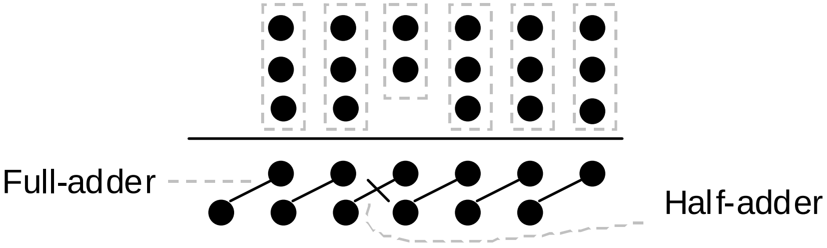


Fig. 8.8 Specifying full- and half-adder blocks, with their inputs and outputs, in dot notation.

Multioperand Addition Using Carry-Save Adders

$$T_{\text{carry-save-multi-add}} = O(\text{tree height} + T_{\text{CPA}})$$
$$= O(\log n + \log k)$$

$$C_{\text{carry-save-multi-add}} = (n - 2)C_{\text{CSA}} + C_{\text{CPA}}$$

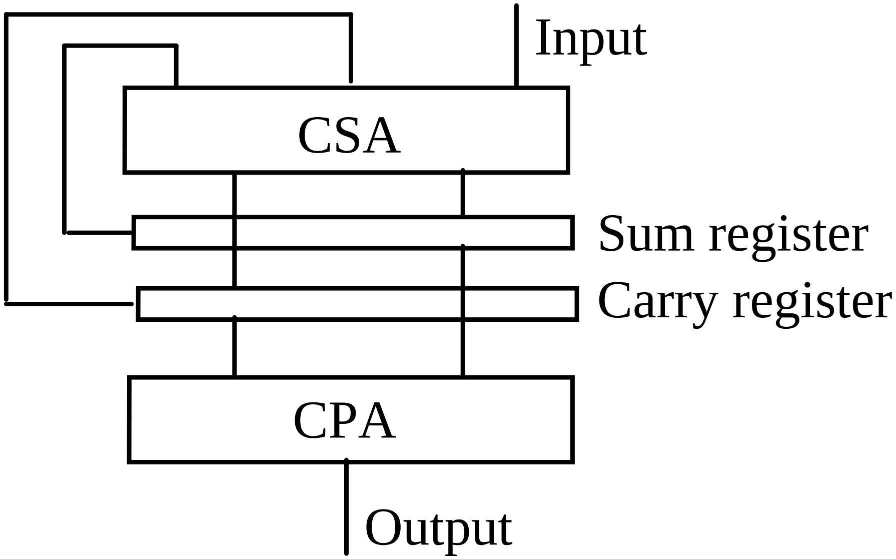


Fig. 8.13 Serial carry-save addition using a single CSA.

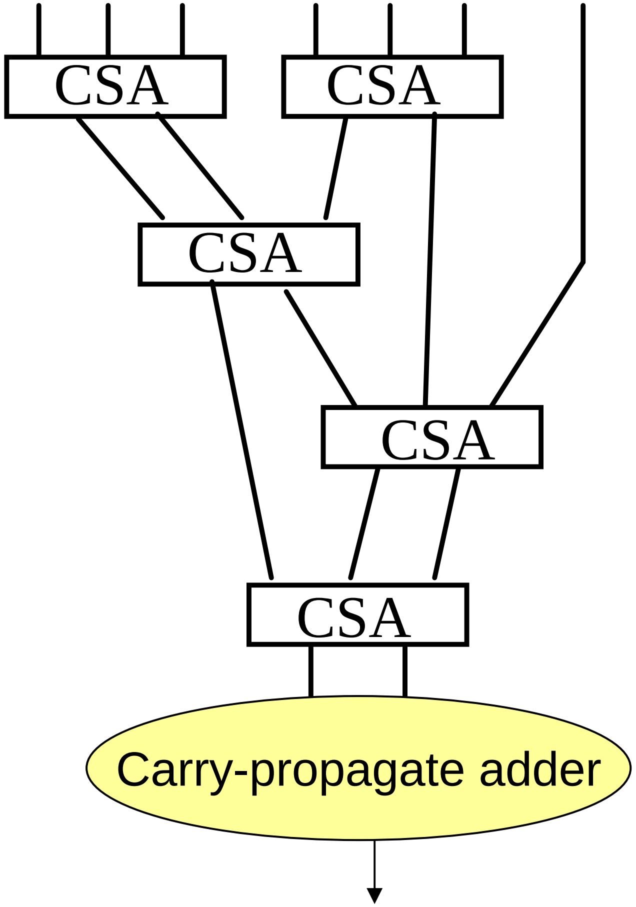
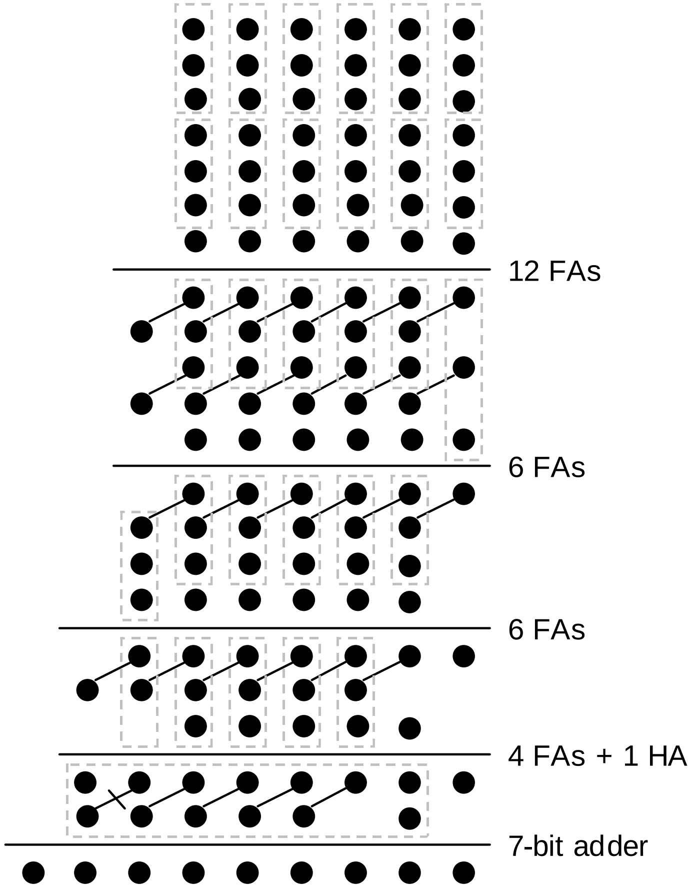


Fig. 8.9 Tree of carry-save adders reducing seven numbers to two.

Example Reduction by a CSA Tree



Total cost = 7-bit adder + 28 FAs + 1 HA

Fig. 8.10 Addition of seven 6-bit numbers in dot notation.

8	7	6	5	4	3	2	1	0	Bit position
			7	7	7	7	7	7	6×2 = 12 FAs
		2	5	5	5	5	5	3	6 FAs
		3	4	4	4	4	4	1	6 FAs
	1	2	3	3	3	3	2	1	4 FAs + 1 HA
	2	2	2	2	2	1	2	1	7-bit adder
--Carry-propagate adder--									
1	1	1	1	1	1	1	1	1	

Fig. 8.11 Representing a seven-operand addition in tabular form.

A full-adder compacts 3 dots into 2 (compression ratio of 1.5)

A half-adder rearranges 2 dots (no compression, but still useful)

Width of Adders in a CSA Tree

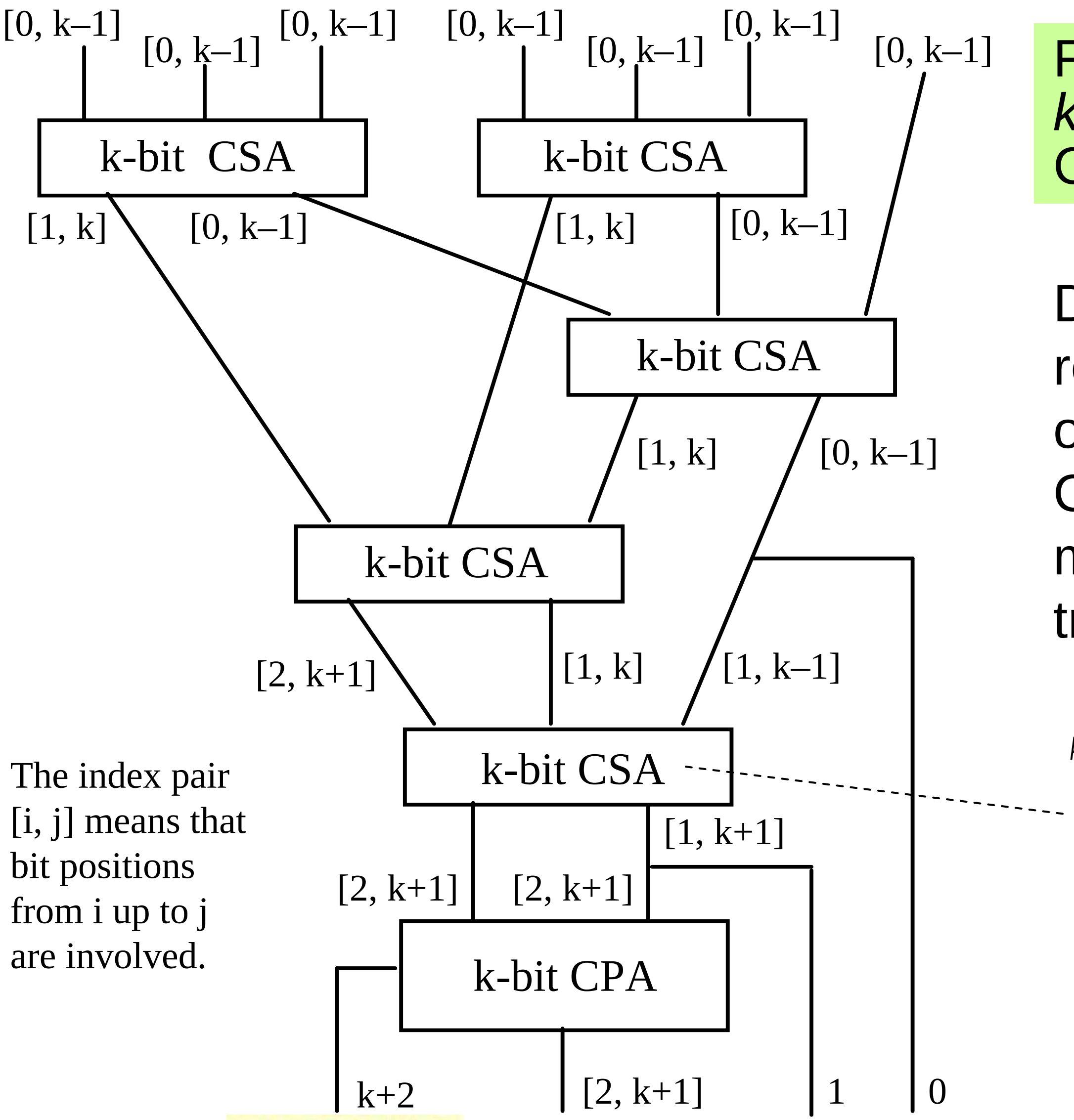
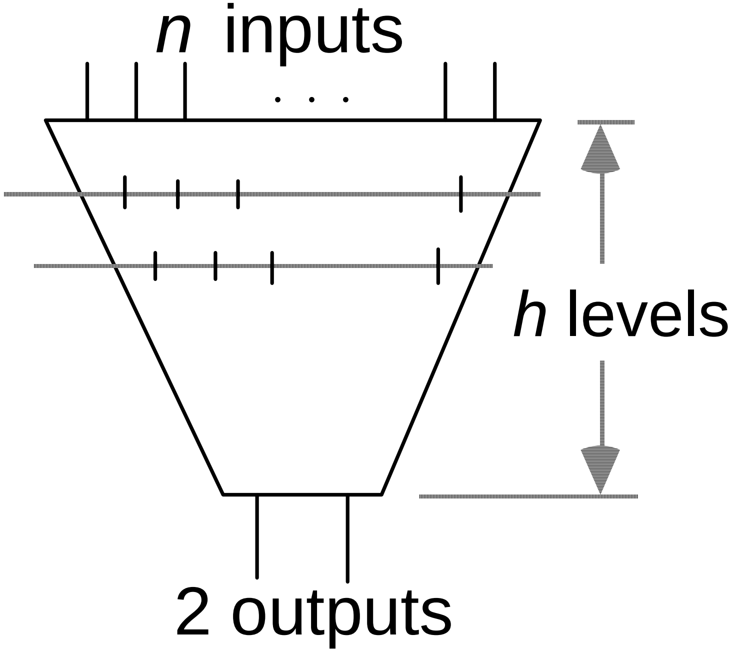


Fig. 8.12 Adding seven *k*-bit numbers and the CSA/CPA widths required.

Due to the gradual retirement (dropping out) of some of the result bits, CSA widths do not vary much as we go down the tree levels

The index pair [i, j] means that bit positions from i up to j are involved.

8.3 Wallace and Dadda Trees



$h(n) = 1 + h(\lceil 2n/3 \rceil)$

$n(h) = \lfloor 3n(h-1)/2 \rfloor$

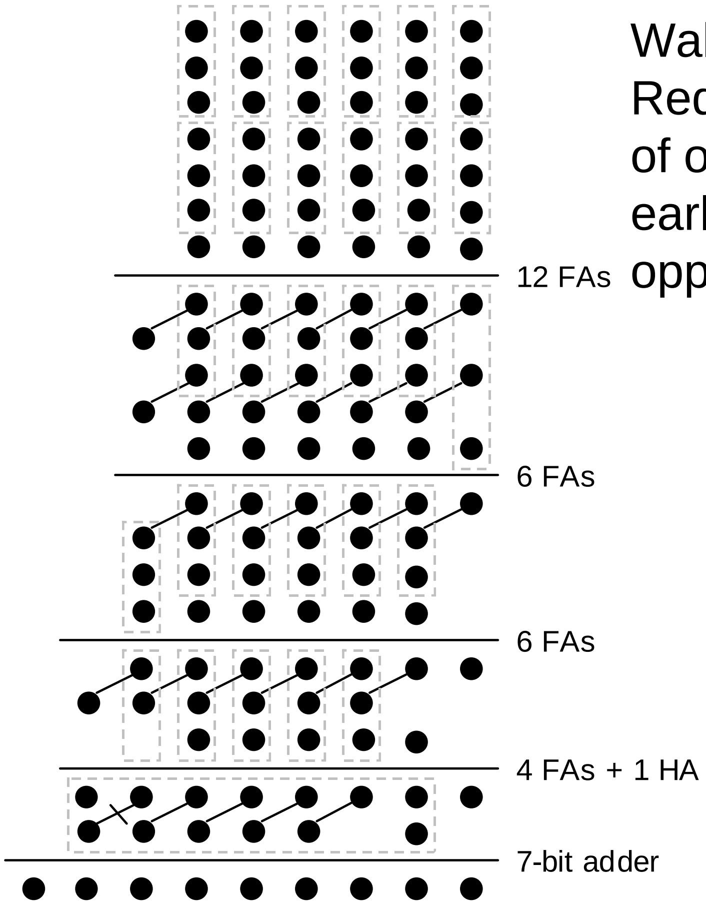
$2 \times 1.5^{h-1} < n(h) \leq 2 \times 1.5^{h}$

Table 8.1 The maximum number *n*(*h*) of inputs for an *h*-level CSA tree

h	*n*(*h*)	*h*	*n*(*h*)	*h*	*n*(*h*)
0	2	7	28	14	474
1	3	8	42	15	711
2	4	9	63	16	1066
3	6	10	94	17	1599
4	9	11	141	18	2398
5	13	12	211	19	3597
6	19	13	316	20	5395

n(*h*): Maximum number of inputs for *h* levels

Example Wallace and Dadda Reduction Trees

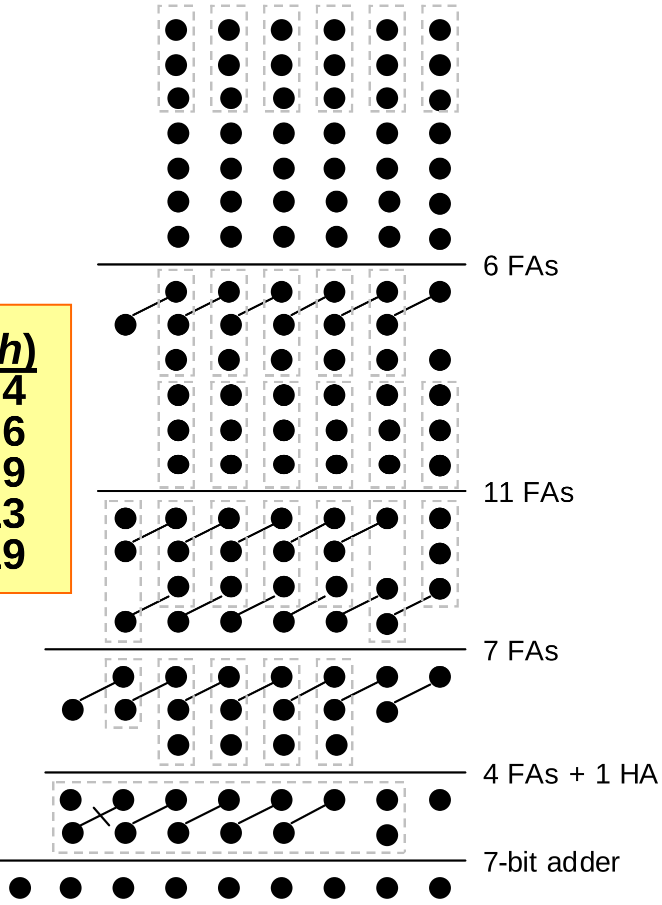


Total cost = 7-bit adder + 28 FAs + 1 HA

Wallace tree:
Reduce the number of operands at the earliest possible opportunity

h	*n*(*h*)
2	4
3	6
4	9
5	13
6	19

Dadda tree:
Postpone the reduction to the extent possible without causing added delay

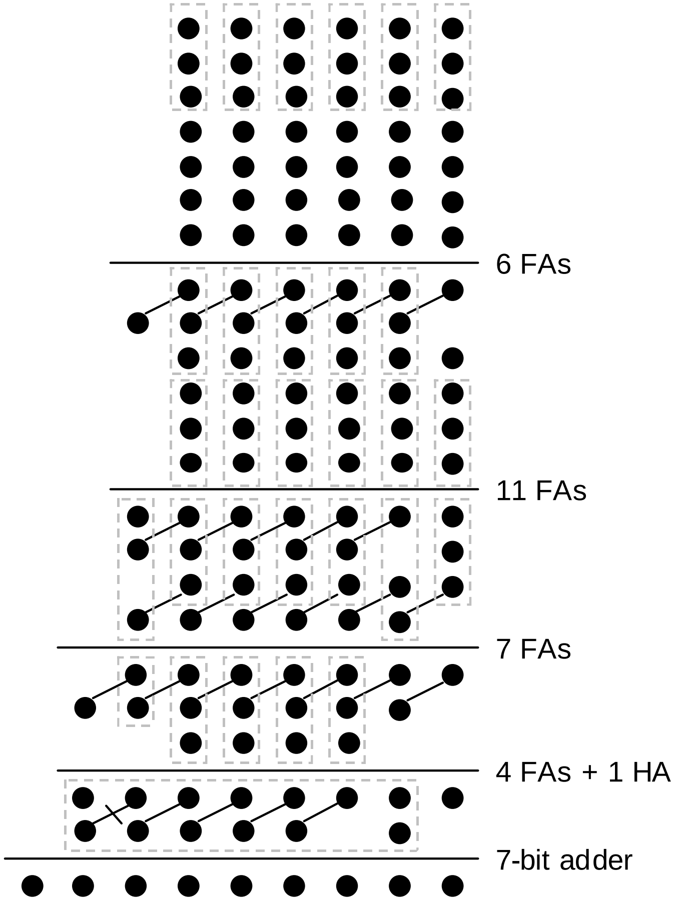


Total cost = 7-bit adder + 28 FAs + 1 HA

Fig. 8.10 Addition of seven 6-bit numbers in dot notation.

Fig. 8.14 Adding seven 6-bit numbers using Dadda's strategy.

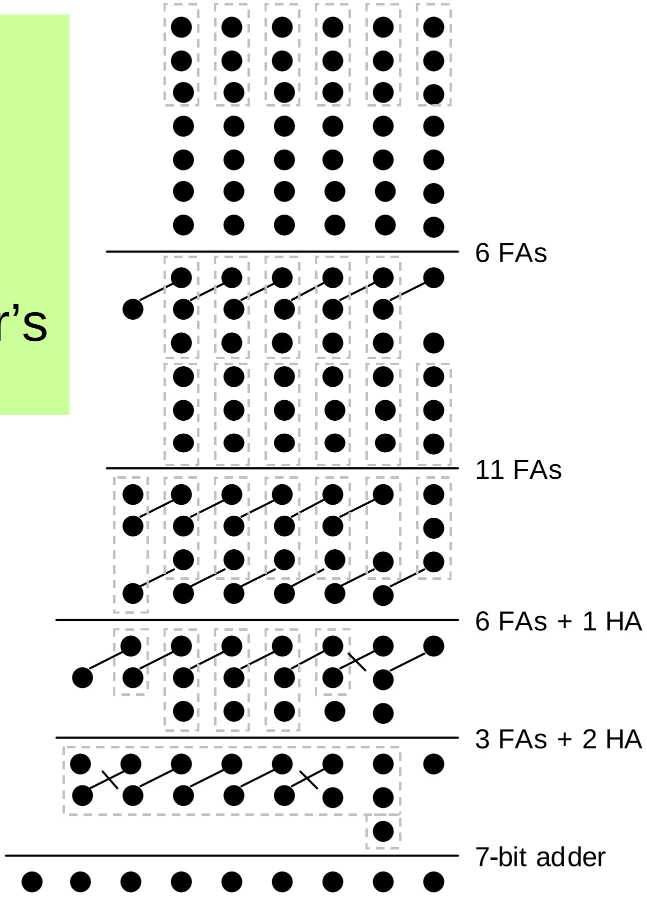
A Small Optimization in Reduction Trees



Total cost = 7-bit adder + 28 FAs + 1 HA

Fig. 8.14 Adding seven 6-bit numbers using Dadda's strategy.

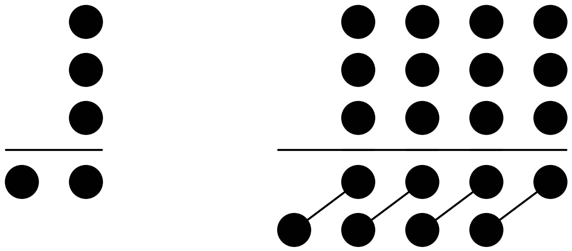
Fig. 8.15
Adding seven 6-bit numbers by taking advantage of the final adder's carry-in.



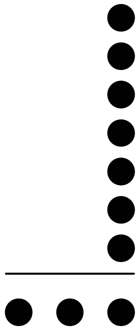
Total cost = 7-bit adder + 26 FAs + 3 HA

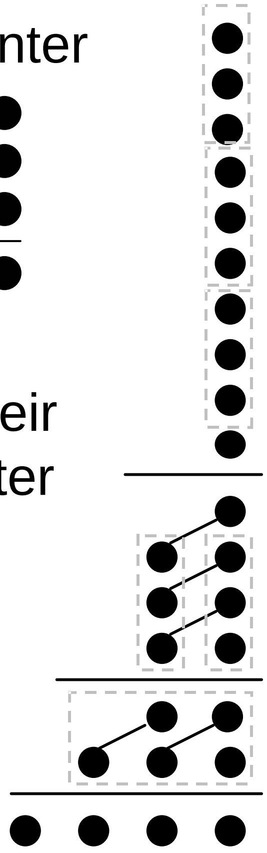
8.4 Parallel Counters and Compressors

1-bit full-adder = (3; 2)-counter



Circuit reducing 7 bits to their 3-bit sum = (7; 3)-counter





Circuit reducing *n* bits to their $\lceil \log_2(n + 1) \rceil$-bit sum = $(n; \lceil \log_2(n+1) \rceil)$-counter

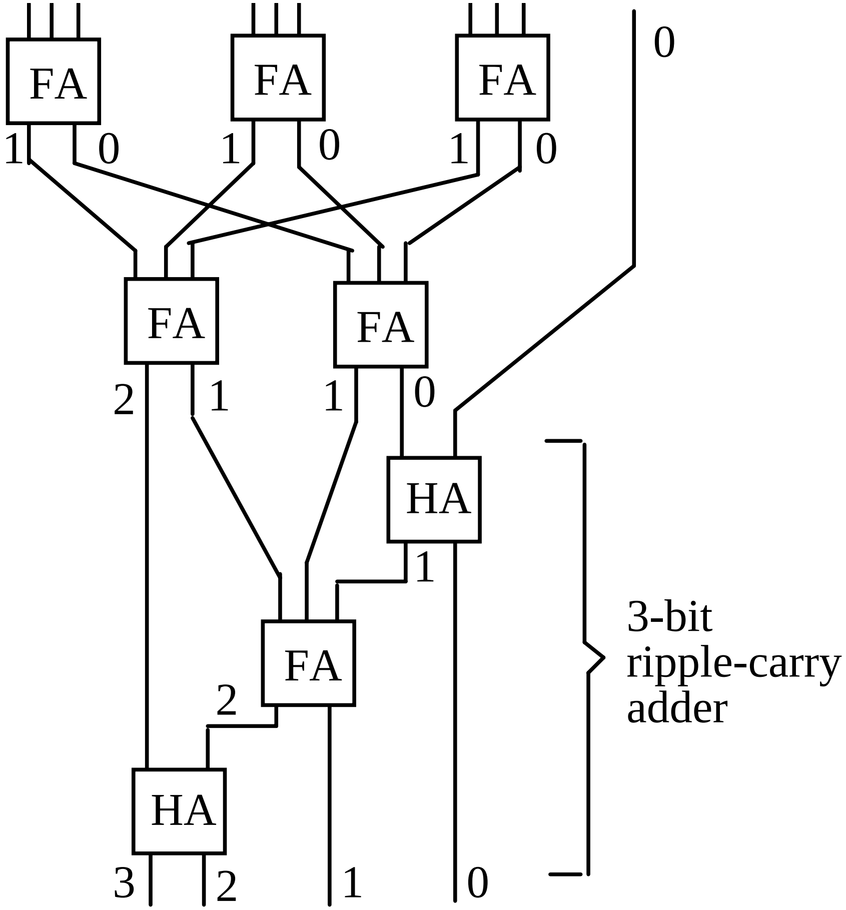
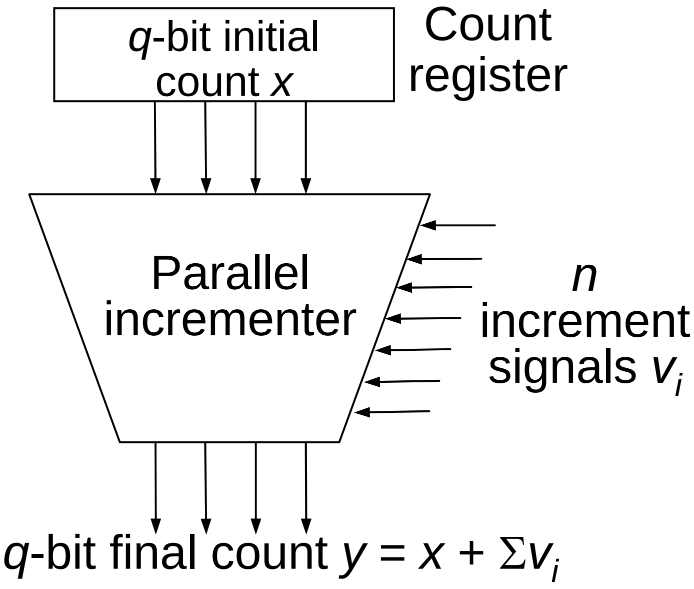
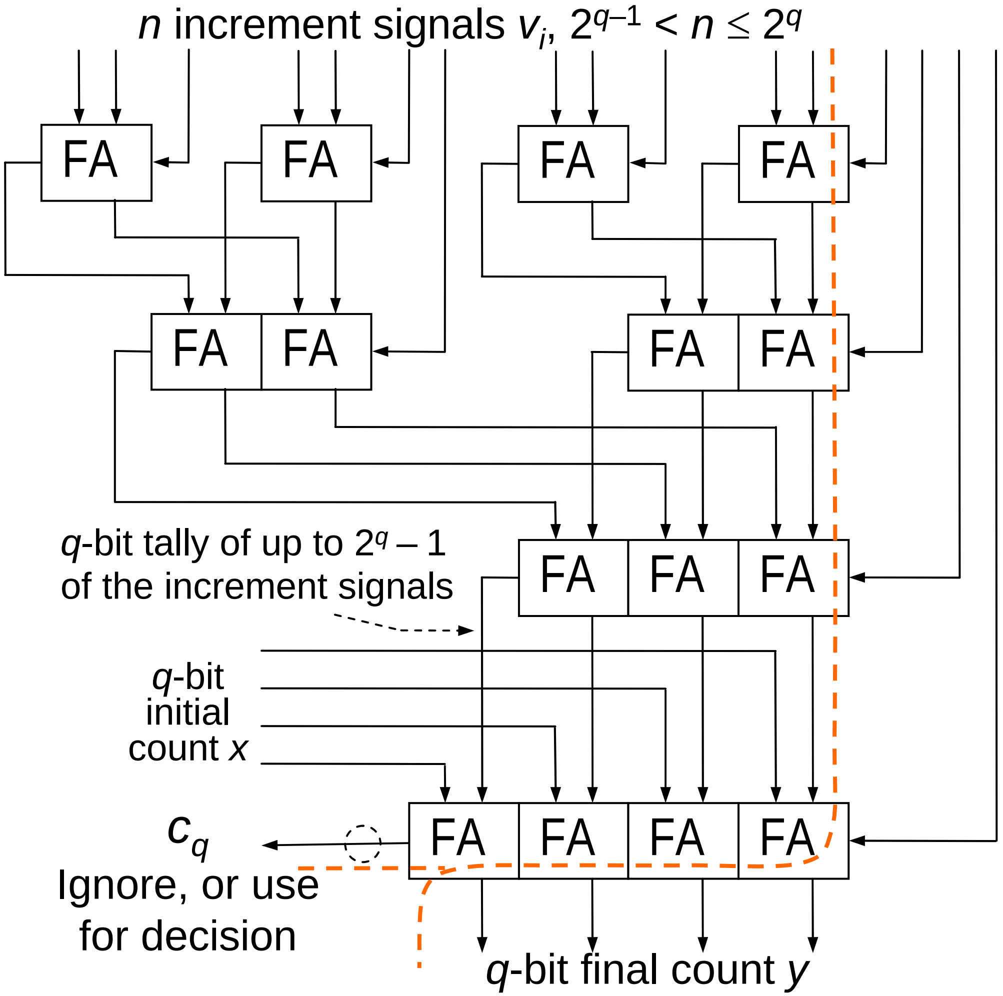


Fig. 8.16 A 10-input parallel counter also known as a (10; 4)-counter.

Accumulative Parallel Counters

True generalization of sequential counters

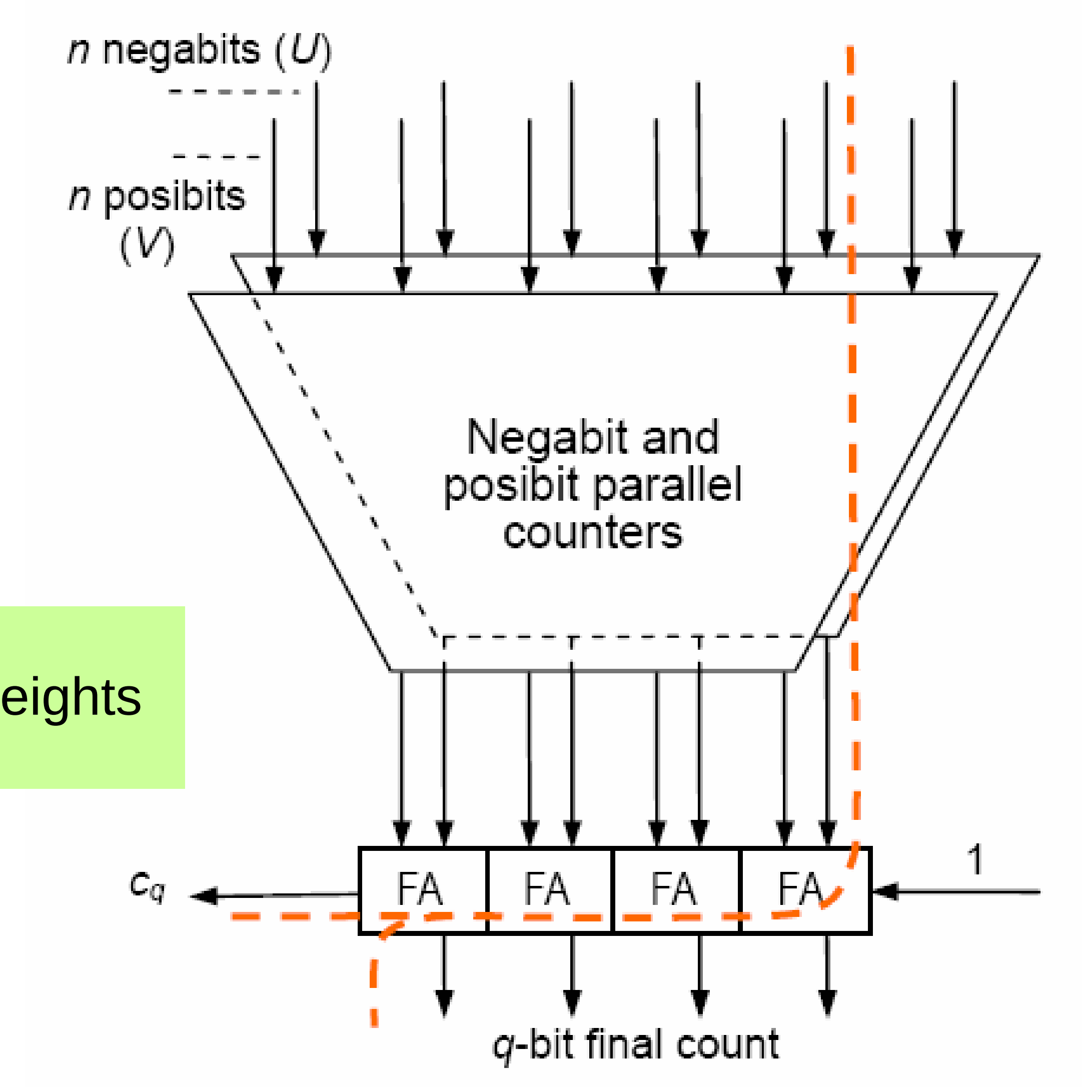

q-bit initial count x
Count register
Parallel incrementer
n increment signals v_i
q-bit final count $y = x + \Sigma v_i$


n increment signals v_i, $2^{q-1} < n \le 2^q$
FA
q-bit tally of up to $2^q - 1$ of the increment signals
q-bit initial count x
c_q
Ignore, or use for decision
q-bit final count y

Possible application:
Compare Hamming weight
of a vector to a constant

Up/Down Parallel Counters

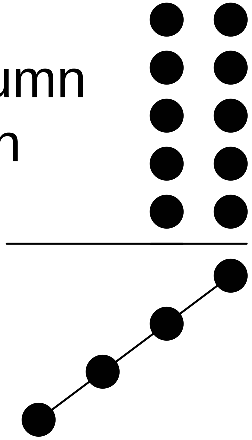
Generalization of up/down counters



Possible application: Compare Hamming weights of two input vectors

8.5 Generalized Parallel Counters

Multicolumn reduction



(5, 5; 4)-counter

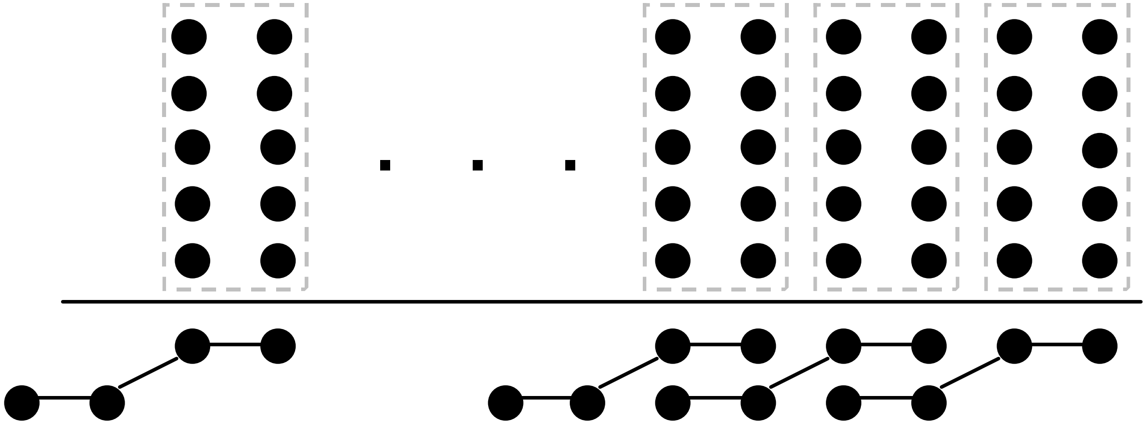
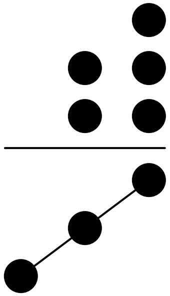


Fig. 8.17 Dot notation for a (5, 5; 4)-counter and the use of such counters for reducing five numbers to two numbers.

Unequal columns



(2, 3; 3)-counter

Gen. parallel counter = Parallel compressor

A General Strategy for Column Compression

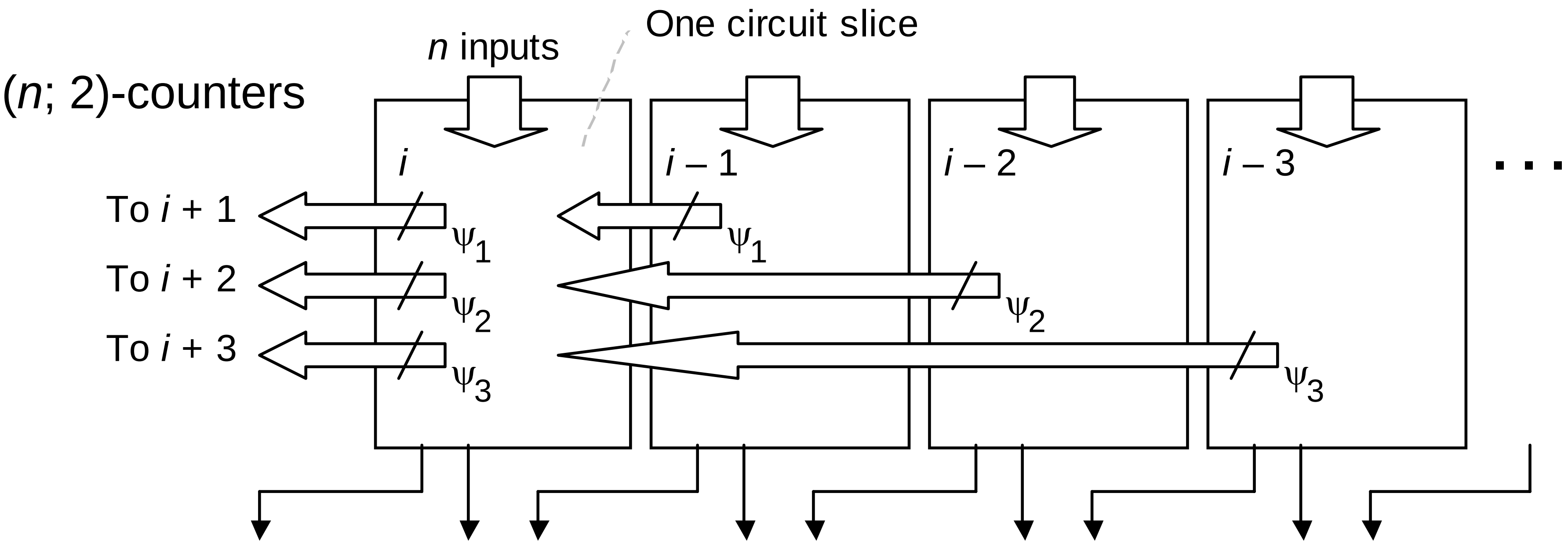


Fig. 8.18 Schematic diagram of an (*n*; 2)-counter built of identical circuit slices

$$n + \psi_1 + \psi_2 + \psi_3 + \ldots \le 3 + 2\psi_1 + 4\psi_2 + 8\psi_3 + \ldots$$

$$n - 3 \le \psi_1 + 3\psi_2 + 7\psi_3 + \ldots$$

Example: Design a bit-slice of an (11; 2)-counter

Solution: Let's limit transfers to two stages. Then, $8 \le \psi_1 + 3\psi_2$

Possible choices include $\psi_1 = 5$, $\psi_2 = 1$ or $\psi_1 = \psi_2 = 2$

8.5 Adding Multiple 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}	. . .

(a) Using sign extension

---------- Extended positions ----------					Sign	Magnitude positions ---------			
1	1	1	1	0	x_{k-1}'	x_{k-2}	x_{k-3}	x_{k-4}	. . .
					y_{k-1}'	y_{k-2}	y_{k-3}	y_{k-4}	. . .
					z_{k-1}'	z_{k-2}	z_{k-3}	z_{k-4}	. . .
					1				

$$\boldsymbol{-b = (1 - b) + 1 - 2}$$

(b) Using negatively weighted bits

Fig. 8.19 Adding three 2's-complement numbers.

8.6 Modular Multioperand Adders

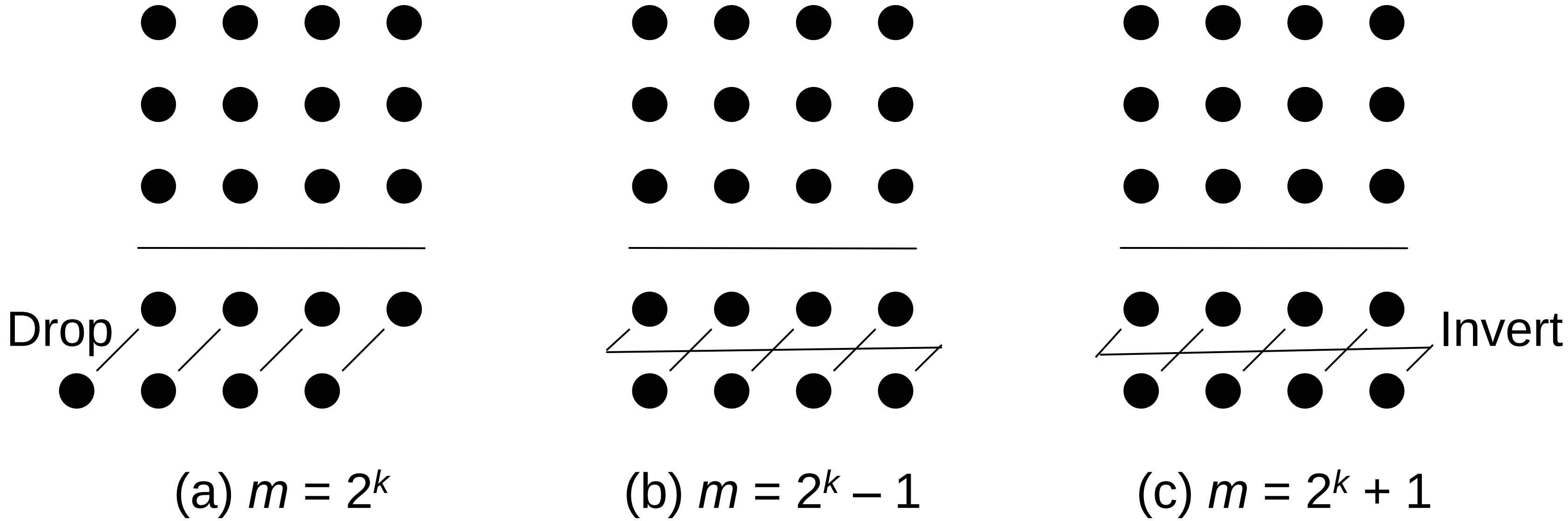
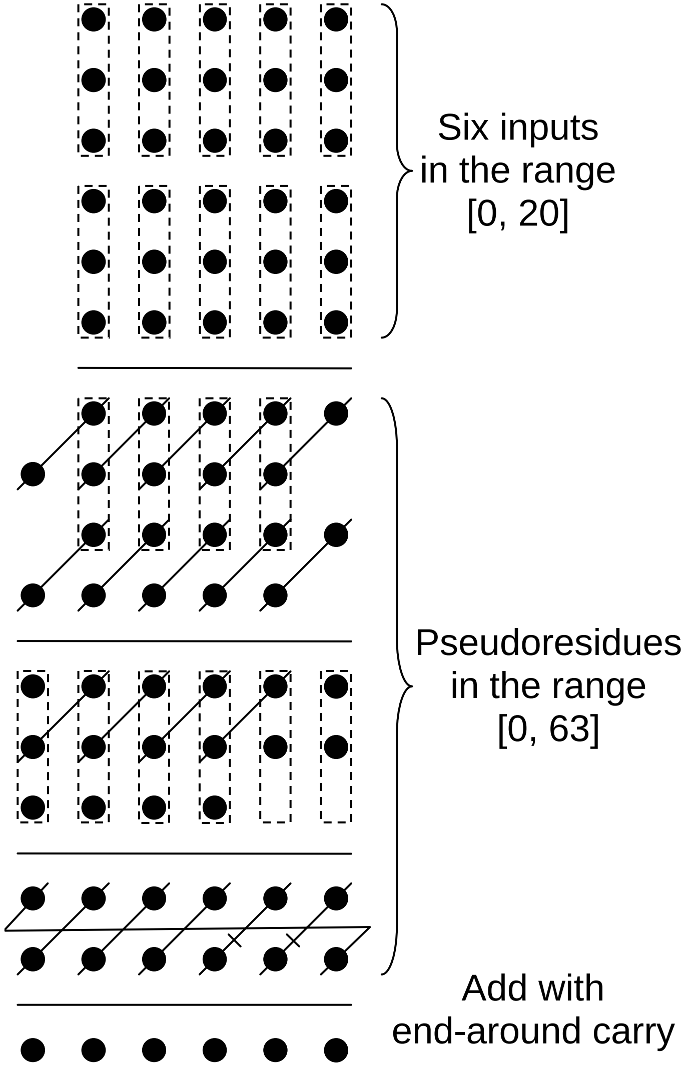


Fig. 8.20 Modular carry-save addition with special moduli.

Modular Reduction with Pseudoresidues


Six inputs in the range [0, 20]
Pseudoresidues in the range [0, 63]
Add with end-around carry

Final pseudoresidue (to be reduced)

Fig. 8.21 Modulo-21 reduction of 6 numbers taking advantage of the fact that 64 = 1 mod 21 and using 6-bit pseudoresidues.