

Dr Milunka Damnjanović, red.prof,
Projektovanje VLSI

Sabirači

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

Basic Addition and Counting

Carry-Lookahead Adders

Variations in Fast Adder

Multioperand Addition

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

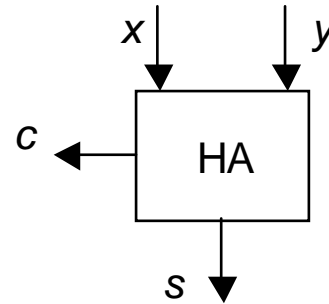
Basic Addition and Counting: Topics

Topics in This Chapter

1. Bit-Serial and Ripple-Carry Adders
2. Conditions and Exceptions
3. Analysis of Carry Propagation
4. Carry Completion Detection
5. Addition of a Constant
6. Manchester Carry Chains and Adders

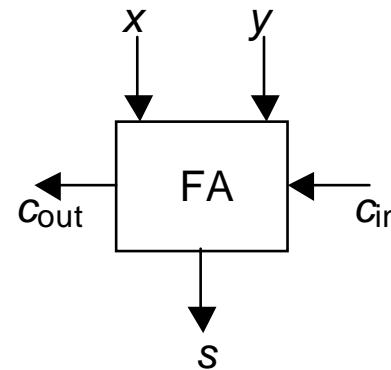
1 Bit-Serial and Ripple-Carry Adders

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



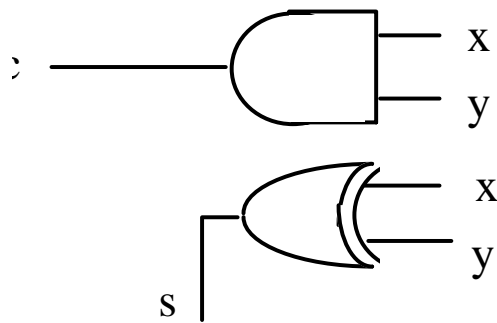
Half-adder (HA): Truth table and block diagram

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

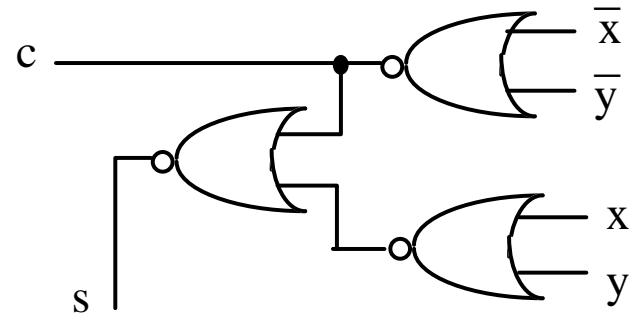


Full-adder (FA): Truth table and block diagram

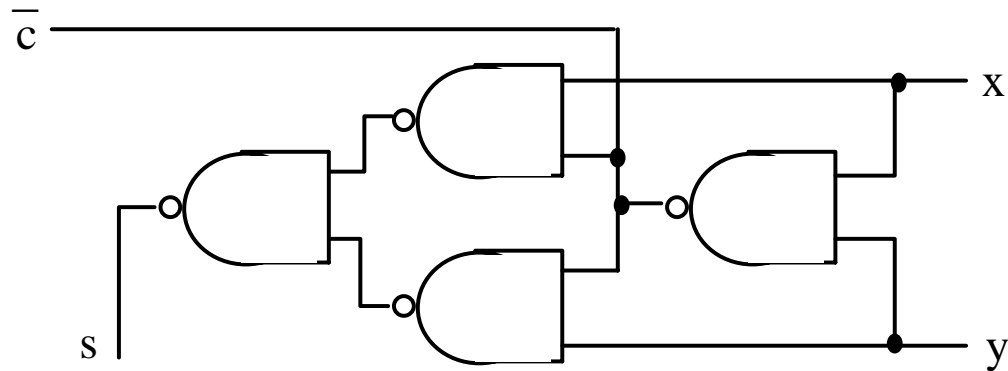
Half-Adder Implementations



(a) AND/XOR half-adder.

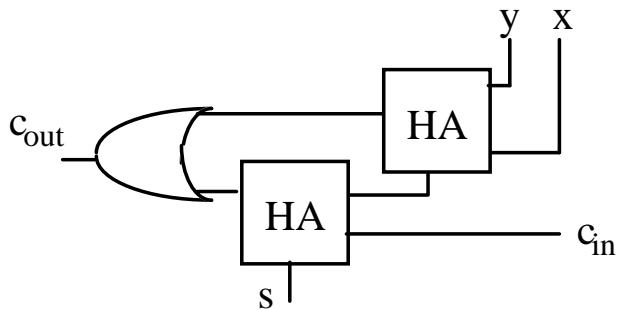


(b) NOR-gate half-adder.

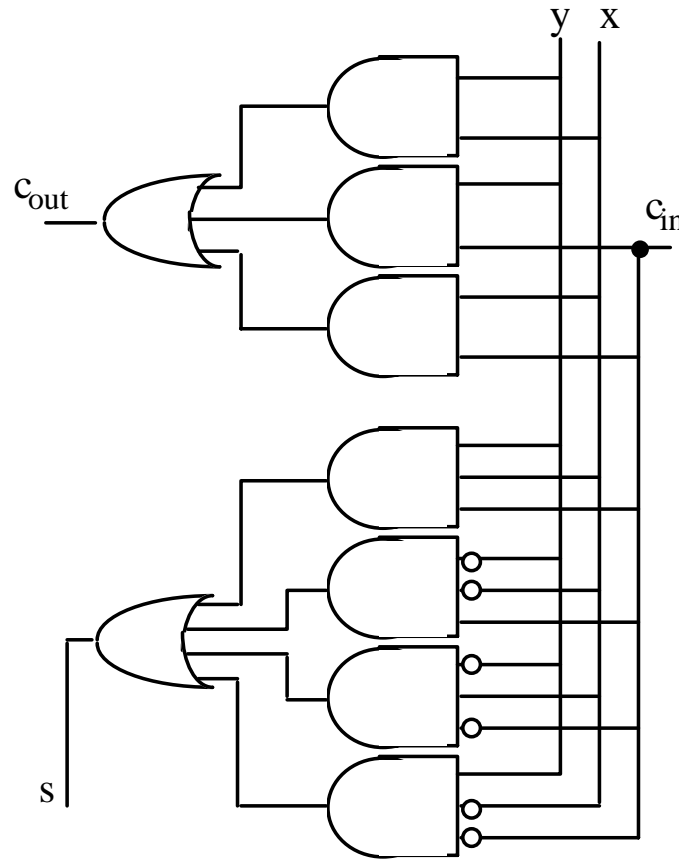


(c) NAND-gate half-adder with complemented carry.

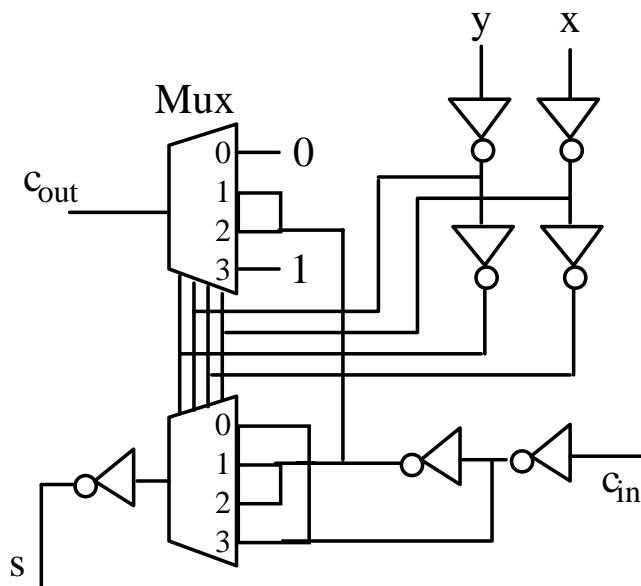
Full-Adder Implementations



(a) Built of half-adders.



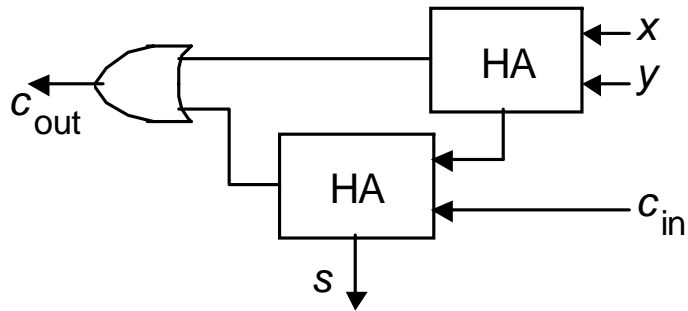
(b) Built as an AND-OR circuit.



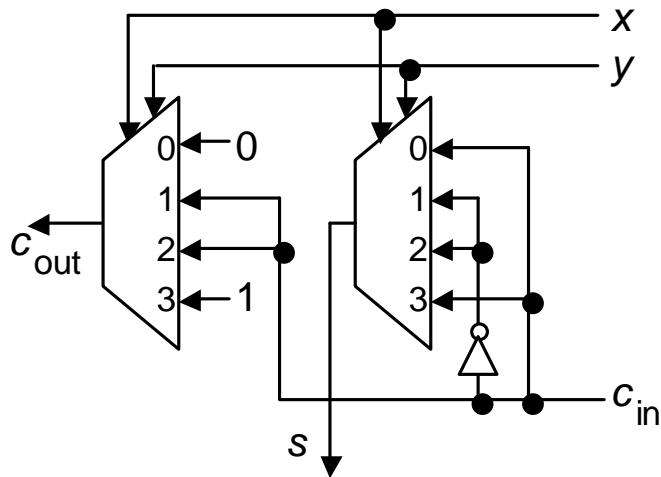
(c) Suitable for CMOS realization.

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

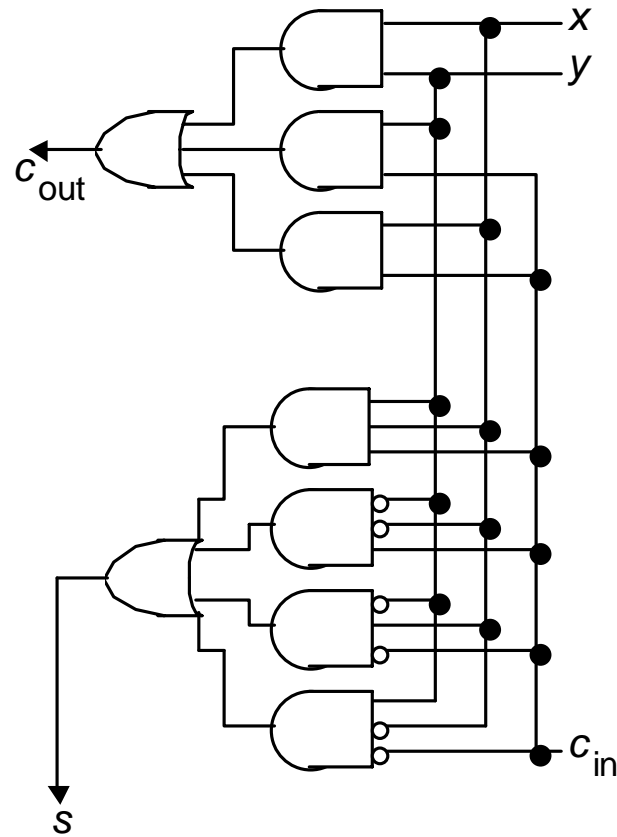
Full-Adder Implementations



(a) FA built of two HAs



(b) CMOS mux-based FA



(c) Two-level AND-OR FA

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

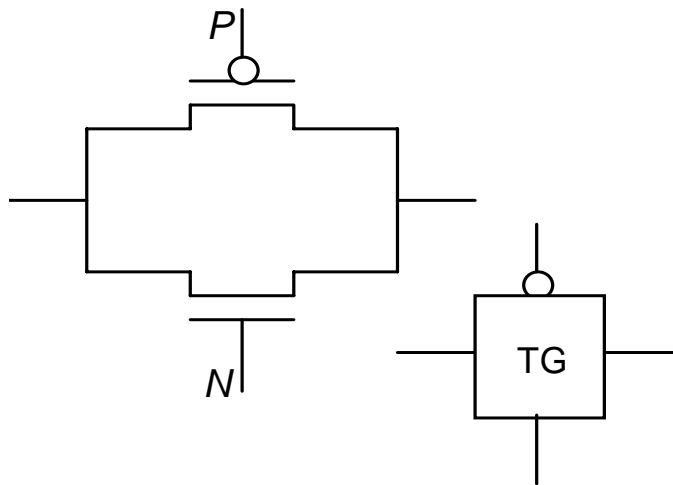
Some Full-Adder Details

Logic equations for a full-adder:

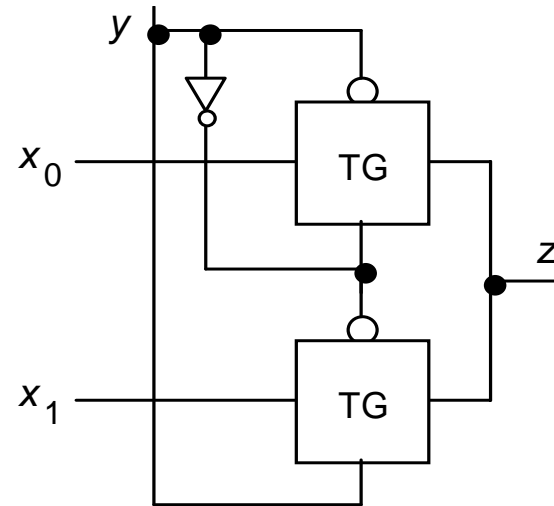
$$s = x \oplus y \oplus c_{in} \quad (\text{odd parity function})$$

$$= xyc_{in} \vee x'y'c_{in} \vee x'yc_{in}' \vee xy'c_{in}'$$

$$c_{out} = xy \vee xc_{in} \vee yc_{in} \quad (\text{majority function})$$



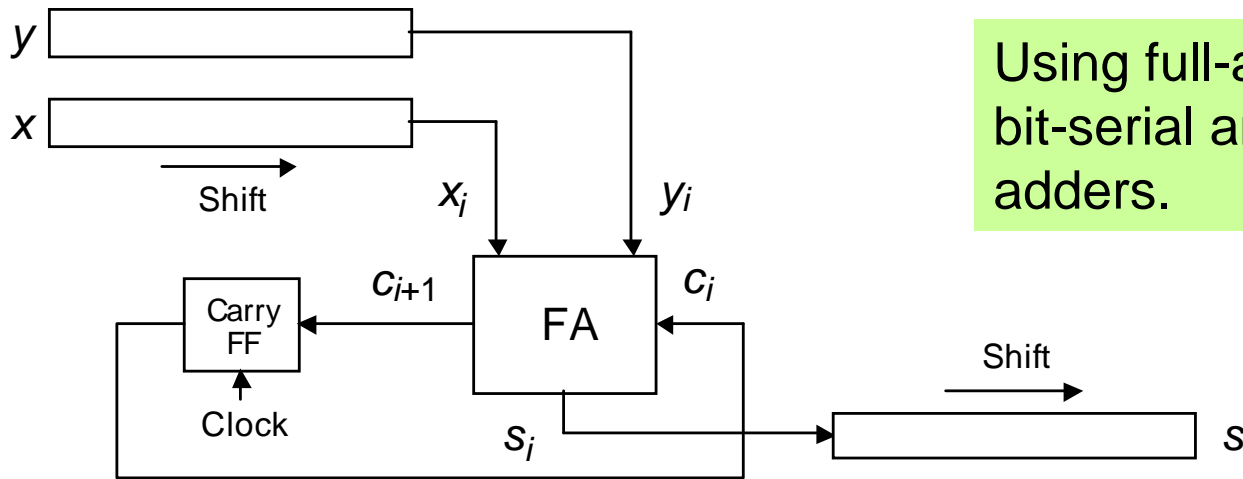
(a) CMOS transmission gate:
circuit and symbol



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

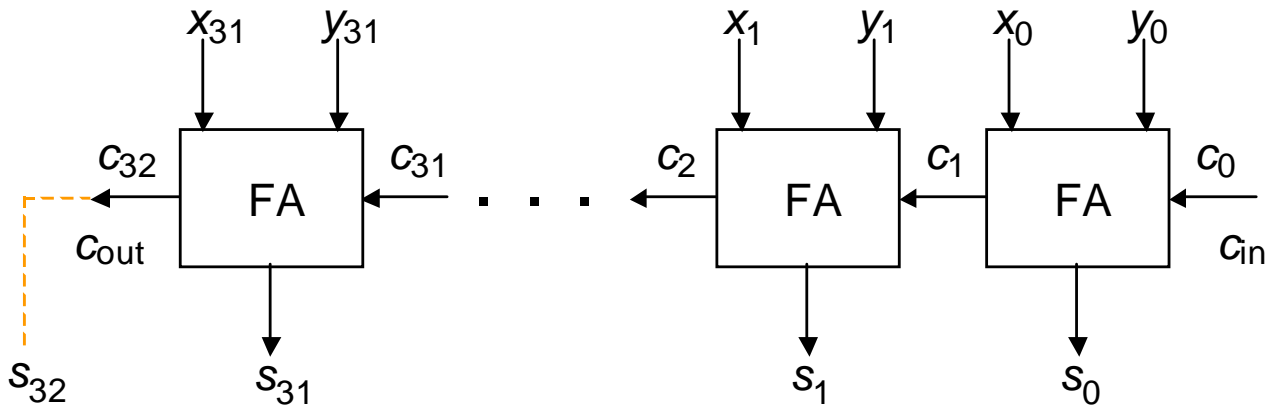
CMOS transmission gate and its use in a 2-to-1 mux.

Simple Adders Built of Full-Adders



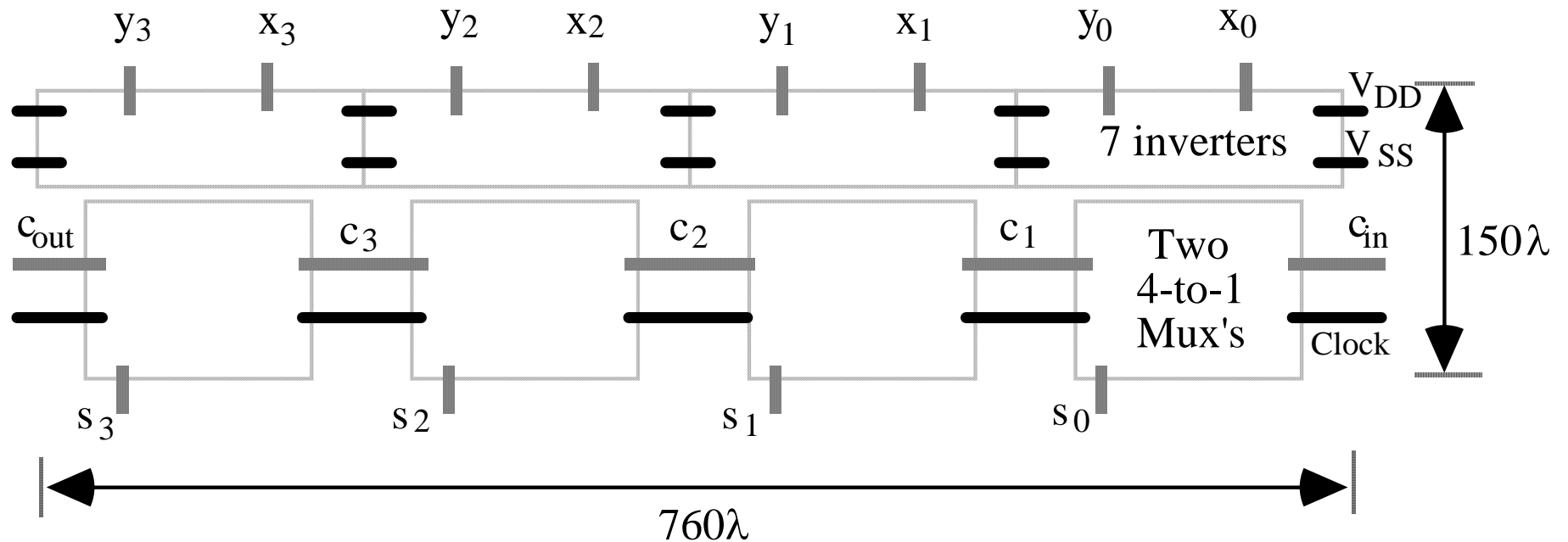
Using full-adders in building bit-serial and ripple-carry adders.

(a) Bit-serial adder.



(b) Ripple-carry adder.

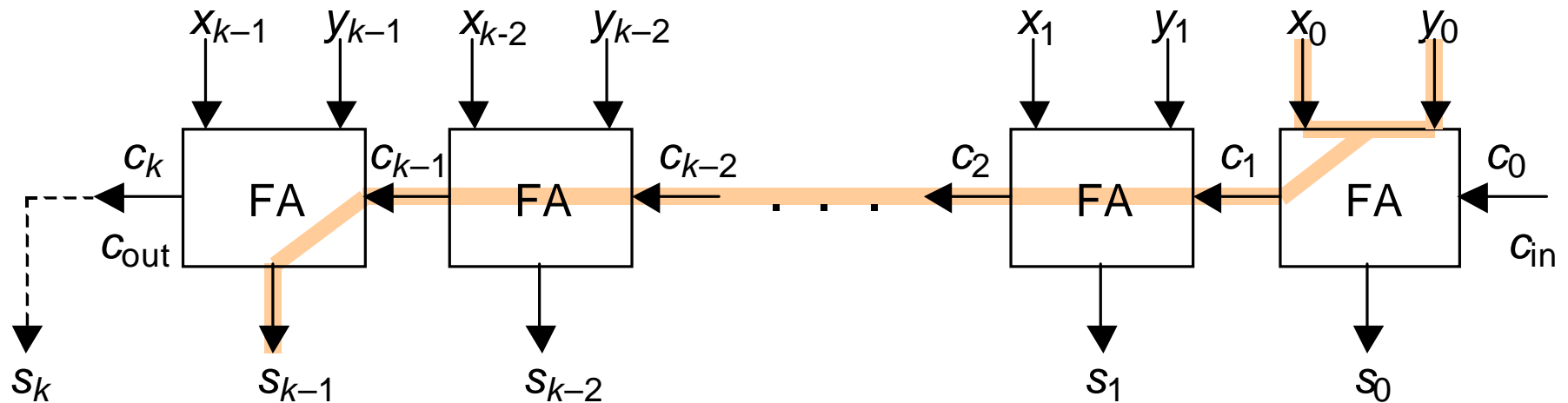
VLSI Layout of a Ripple-Carry Adder



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

Critical Path Through a Ripple-Carry Adder

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



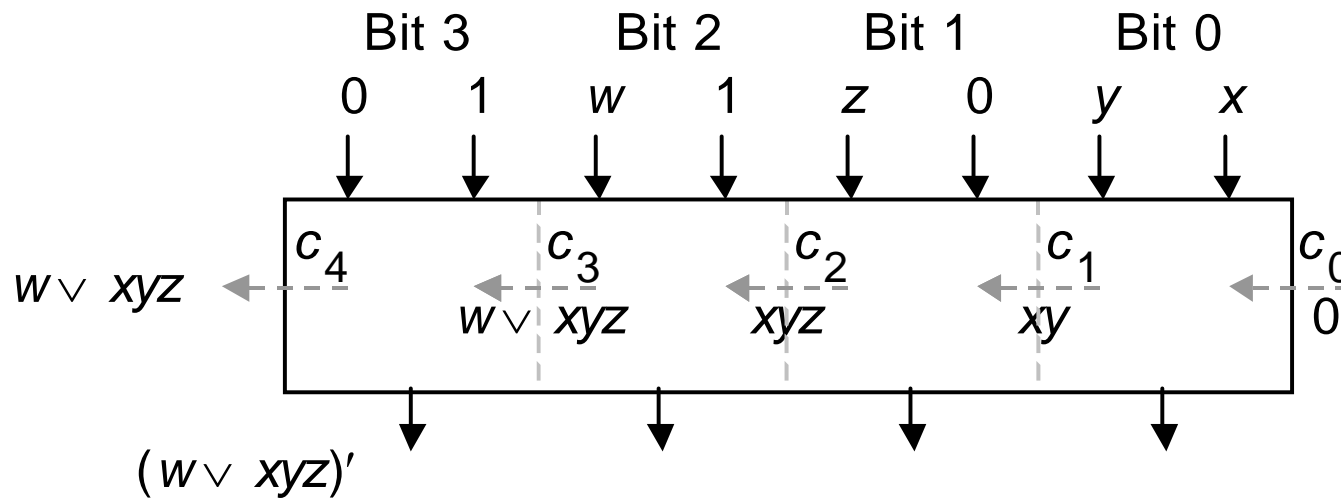
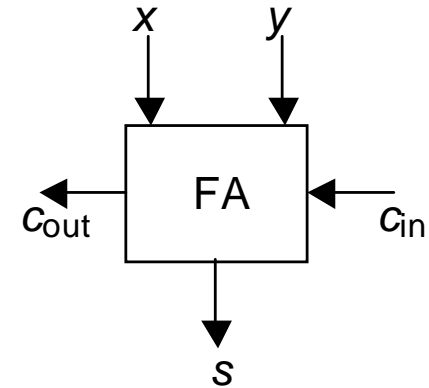
Critical path in a k -bit ripple-carry adder.

Binary Adders as Versatile Building Blocks

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

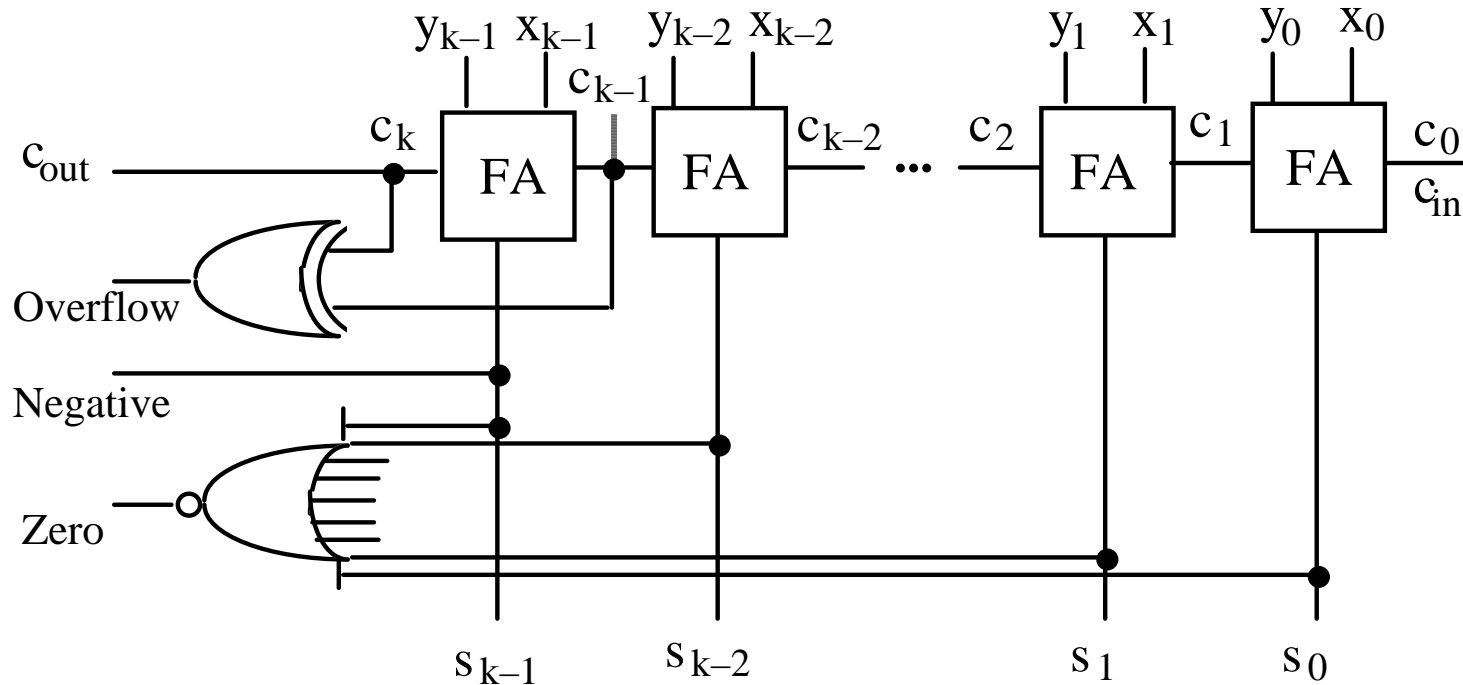
Set one input to 1: $c_{out} = \text{OR of other inputs}$

Set one input to 0
and another to 1: $s = \text{NOT of third input}$



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

2 Conditions and Exceptions



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

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

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

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:

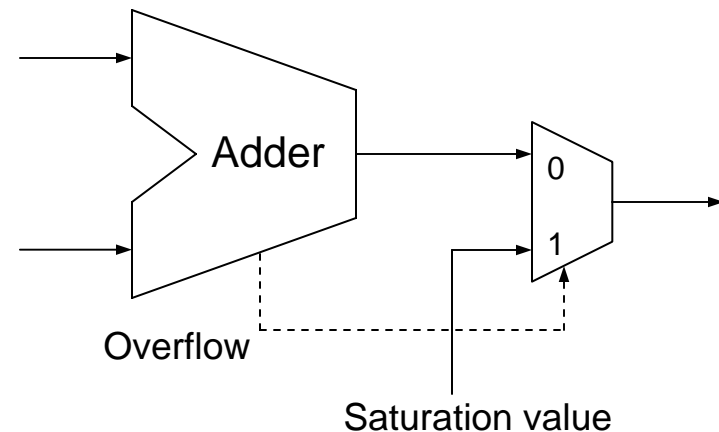
$120 + 26 \rightarrow 18$ (wraparound); $120 +_{\text{sat}} 26 \rightarrow 127$ (saturating)

Saturating arithmetic is desirable in many DSP applications

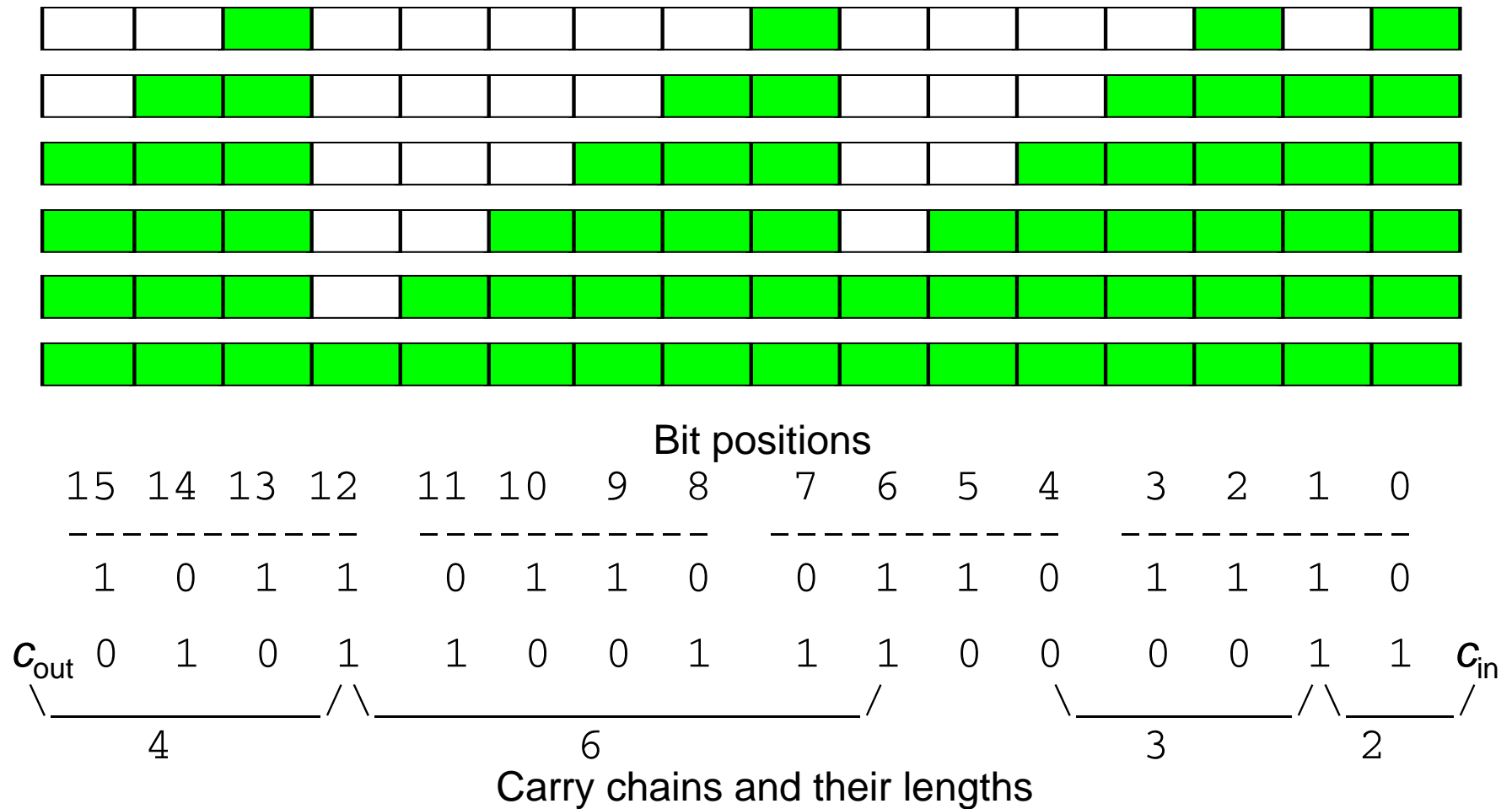
Designing saturating adders

Unsigned (quite easy)

Signed (only slightly harder)



3 Analysis of Carry Propagation



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 = $\frac{1}{4}$ (both 1s)

Probability of carry annihilation = $\frac{1}{4}$ (both 0s)

Probability of carry propagation = $\frac{1}{2}$ (different)

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

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

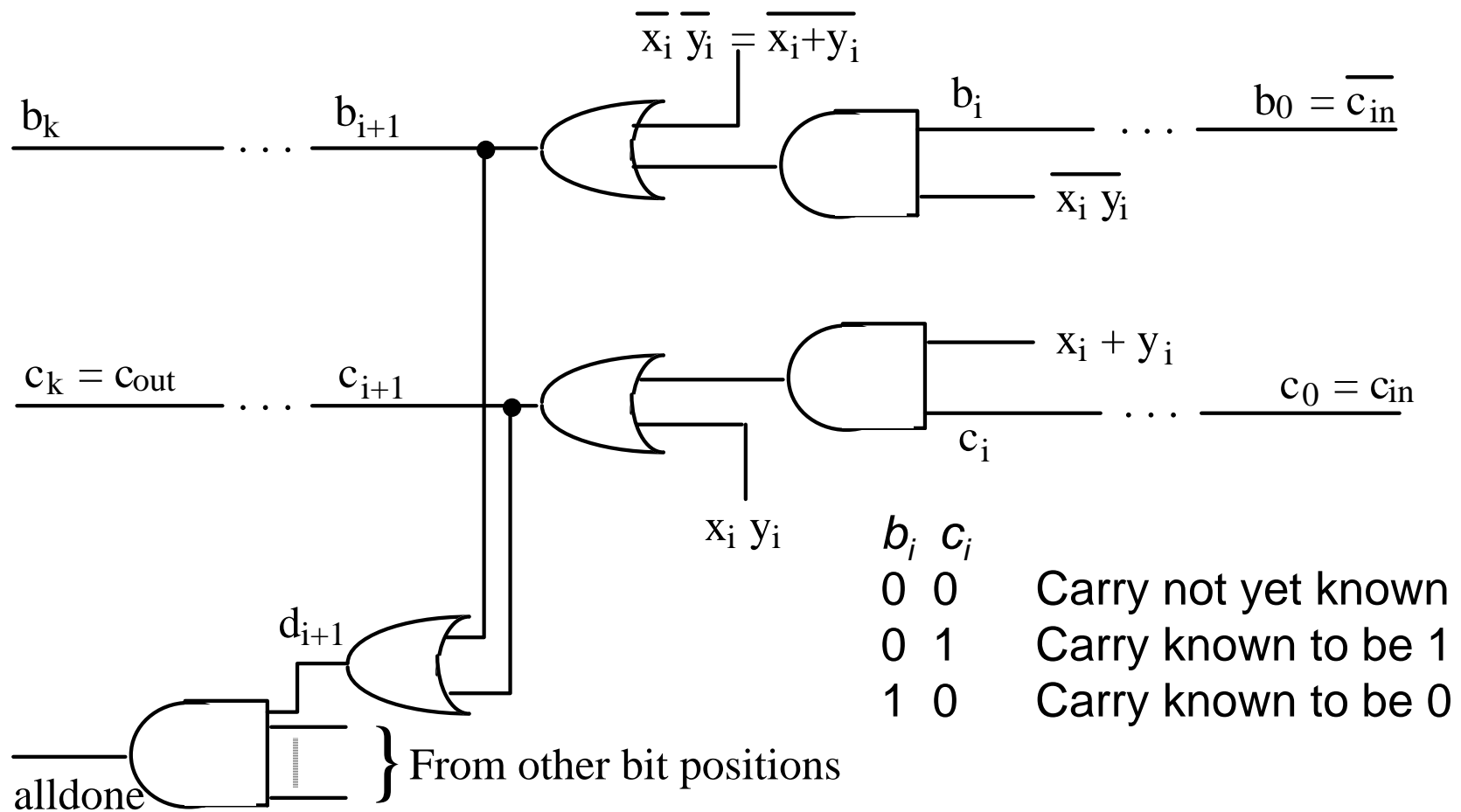
Expected length of the carry chain that starts at position i

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

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

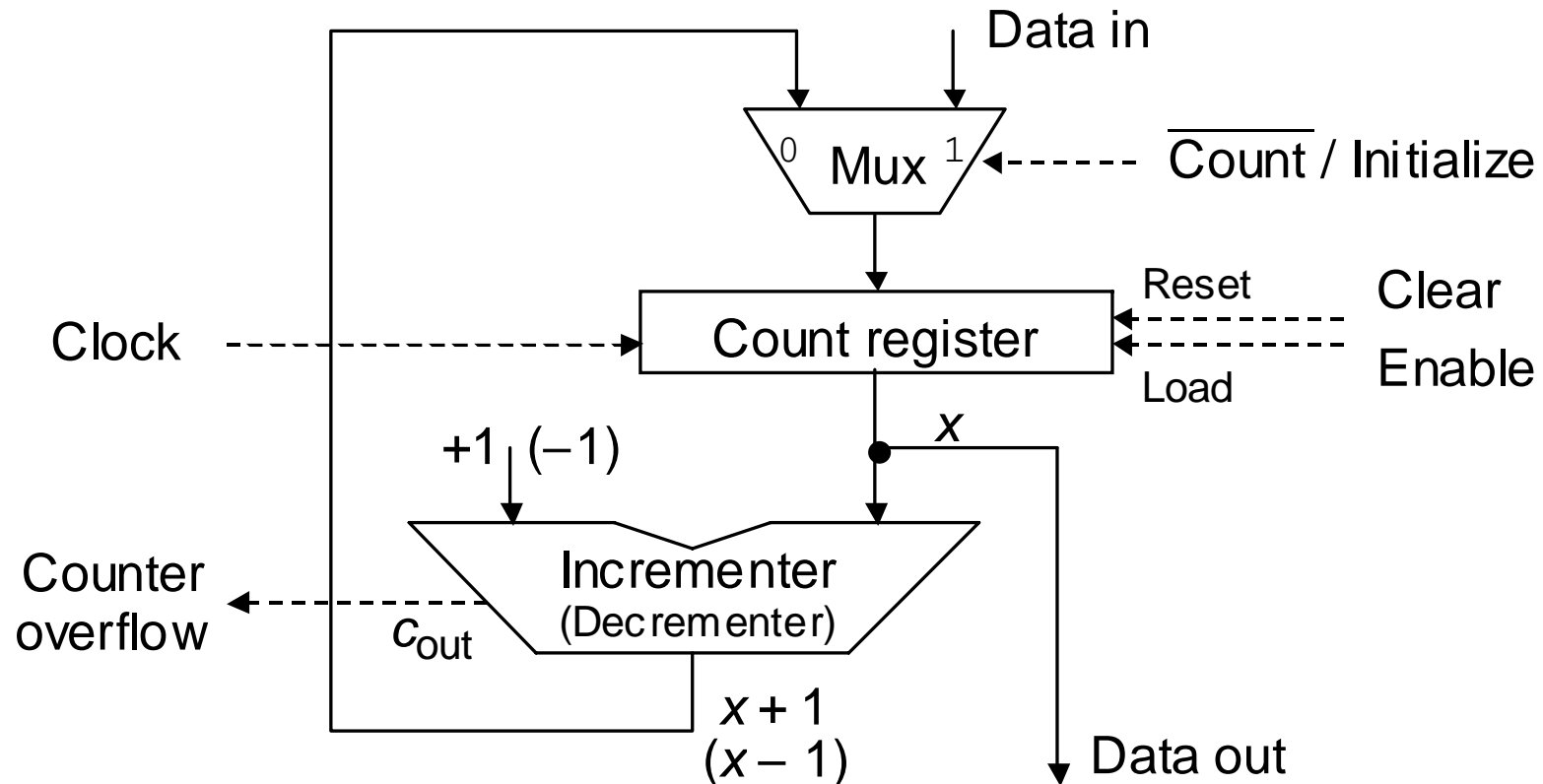
Analogy: Expected number when rolling one die is 3.5; if one rolls many dice, the expected value of the largest number shown grows

4 Carry Completion Detection



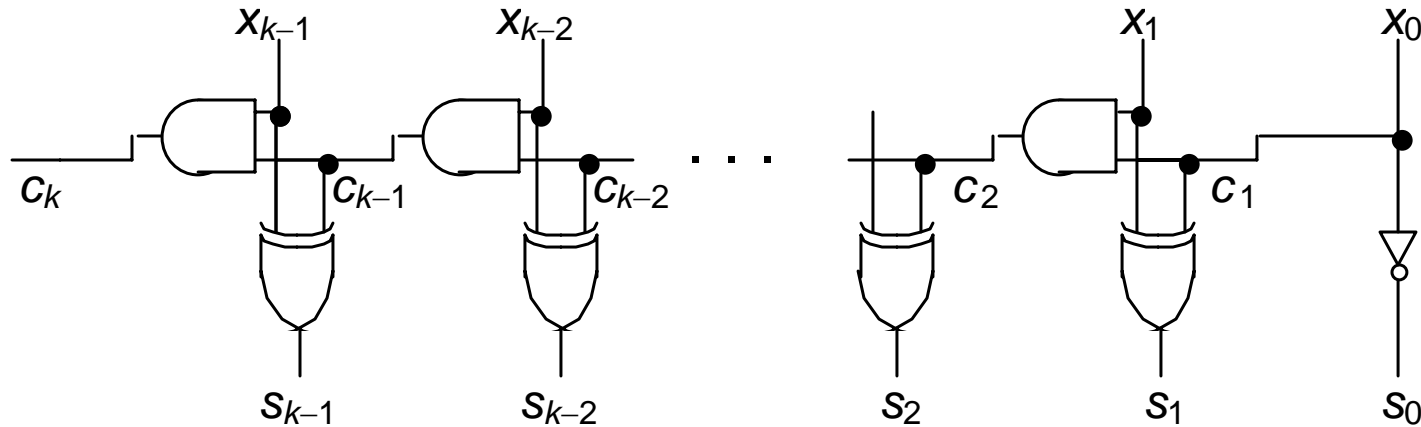
Example addition and its carry propagation chains.

5 Addition of a Constant: Counters

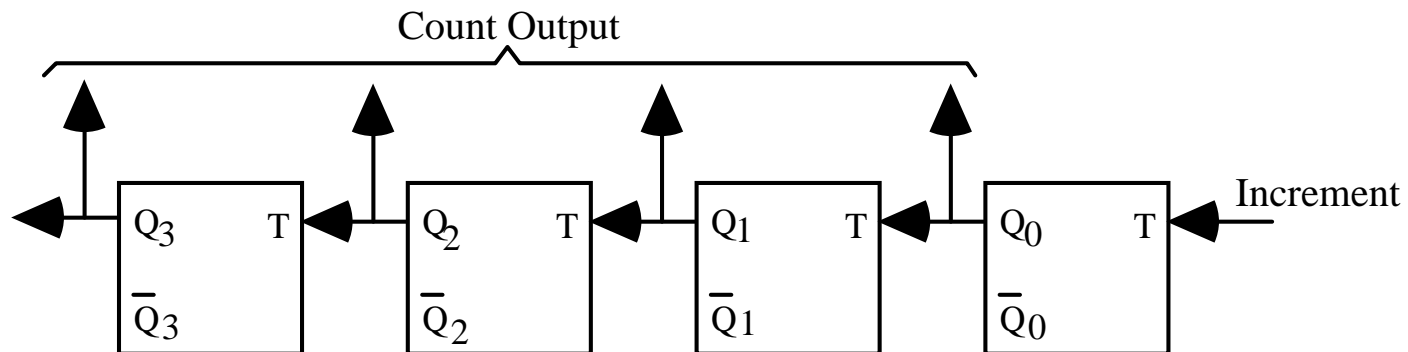


An up (down) counter built of a register, an incrementer (decrementer), and a multiplexer.

Implementing a Simple Up Counter



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

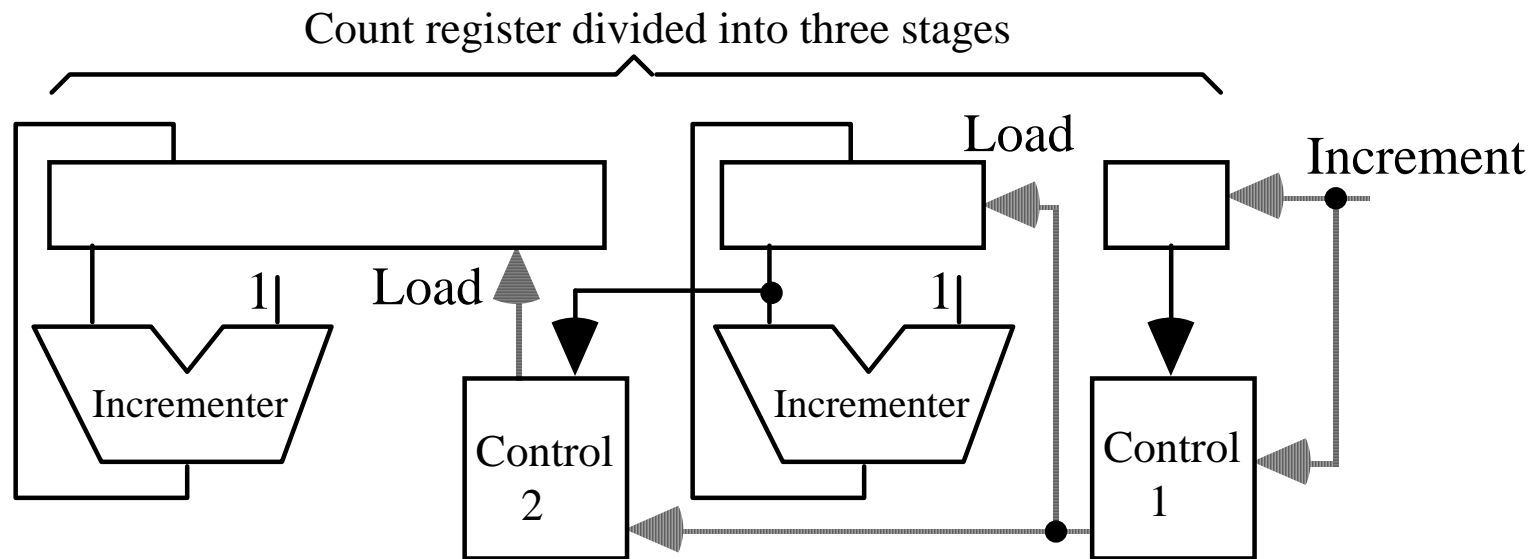


Four-bit asynchronous up counter built only of negative-edge-triggered T flip-flops.

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



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

6 Manchester Carry Chains and Adders

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

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

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:

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

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

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

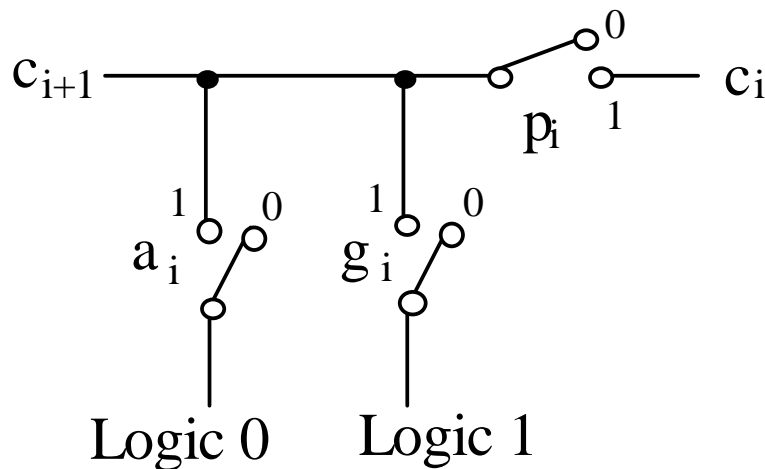
Using these signals, the *carry recurrence* is written as

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

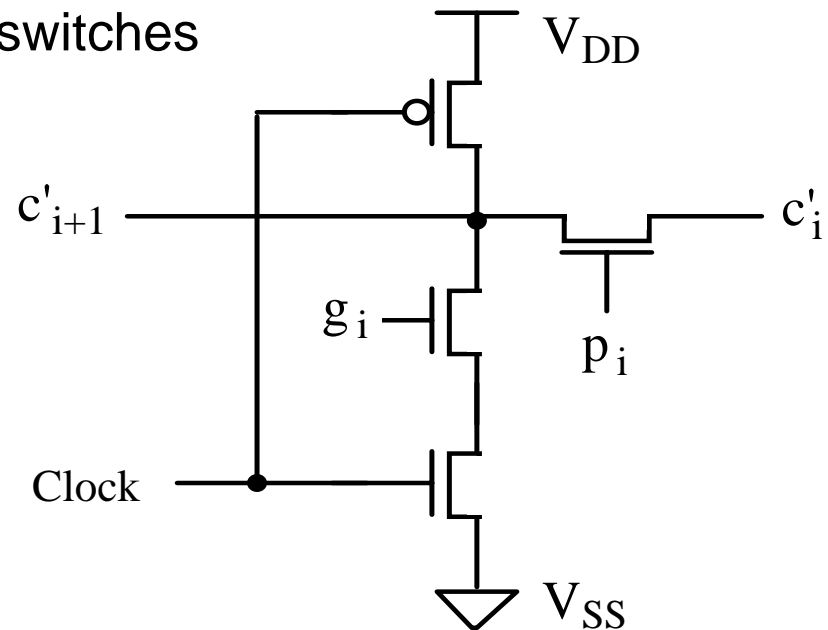
Manchester Carry Network

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



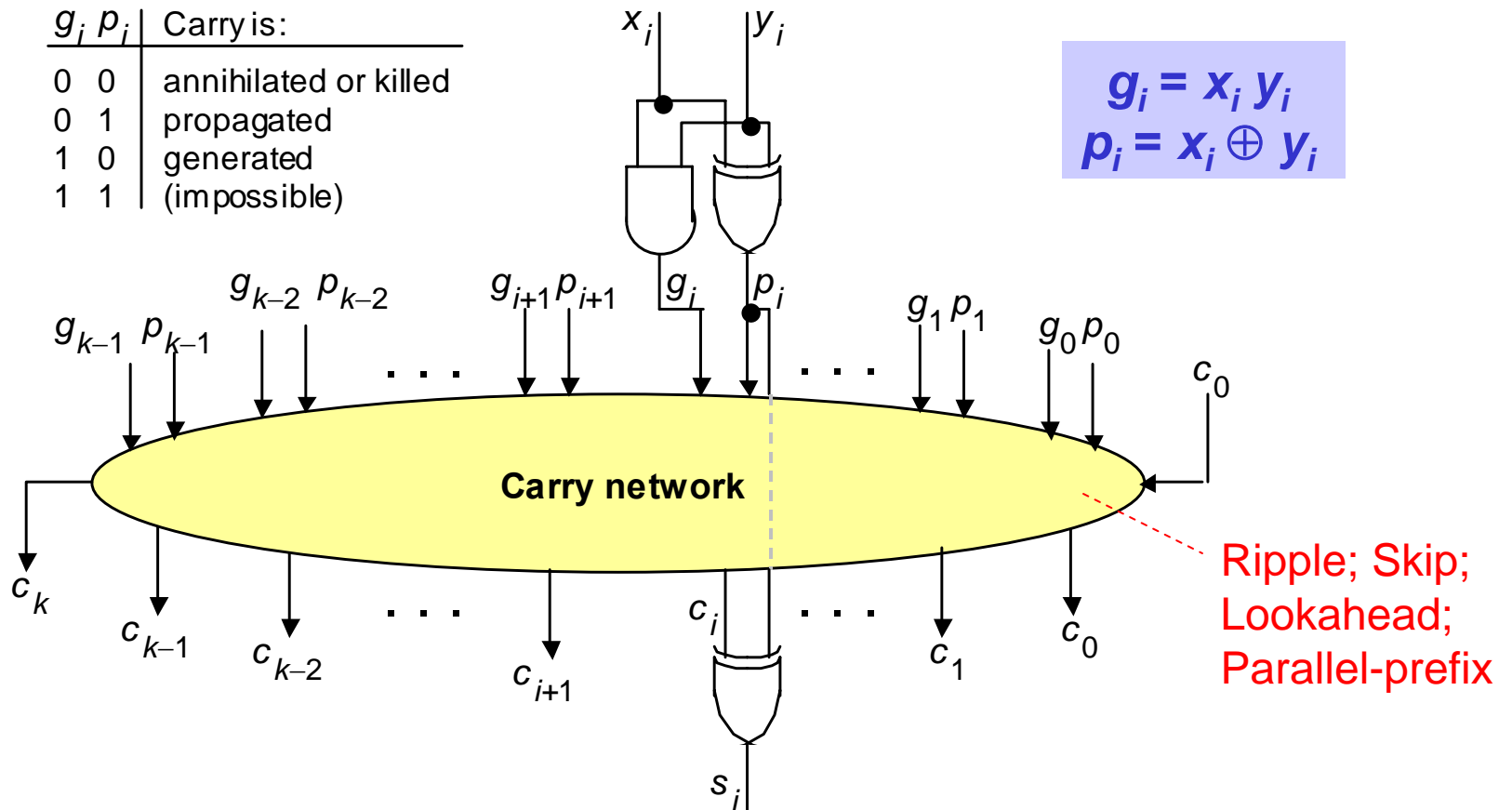
(a) Conceptual representation



(b) Possible CMOS realization.

Fig. 5.13 One stage in a Manchester carry chain.

Carry Network is the Essence of a Fast Adder



(From arch text) The main part of an adder is the carry network. The rest is just a set of gates to produce the g and p signals and the sum bits.

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

1. Unrolling the Carry Recurrence
2. Carry-Lookahead Adder Design
3. Ling Adder and Related Designs
4. Carry Determination as Prefix Computation
5. Alternative Parallel Prefix Networks
6. VLSI Implementation Aspects

1 Unrolling the Carry Recurrence

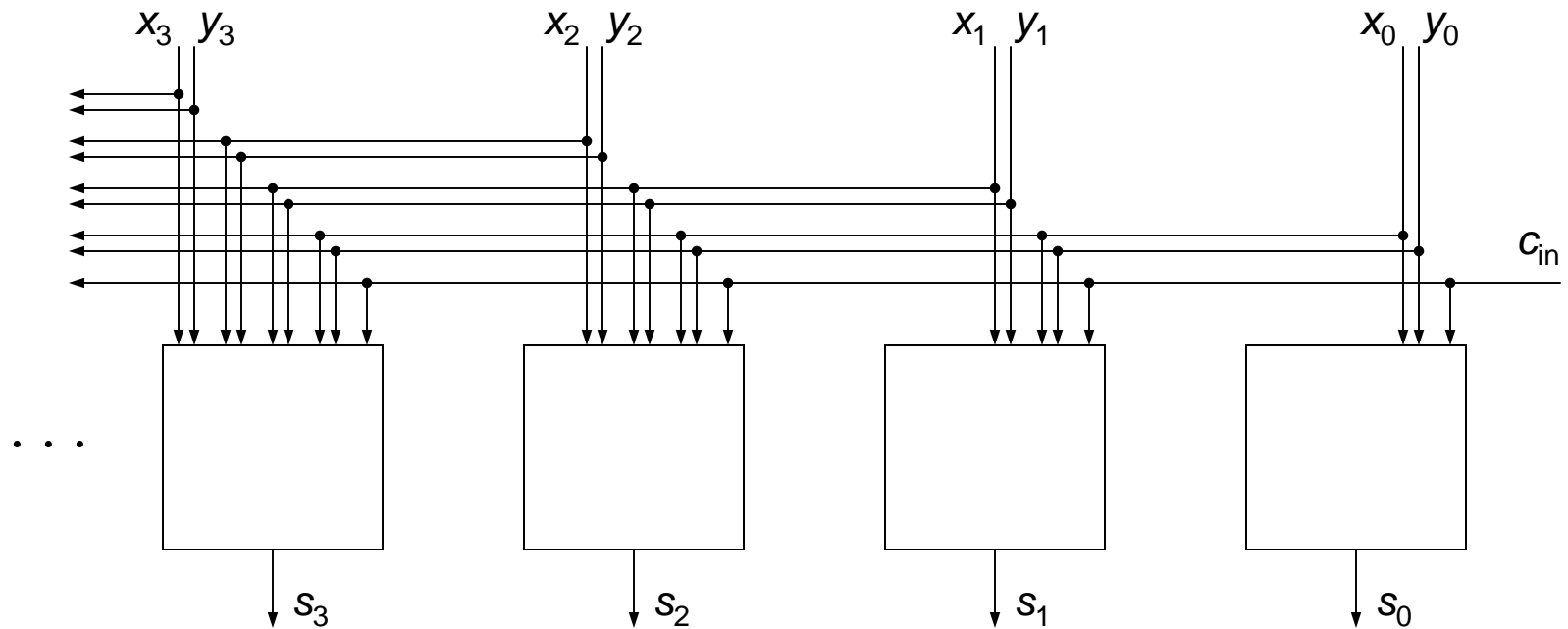
Recall the *generate*, *propagate*, *annihilate* (*absorb*), and *transfer* signals:

<u>Signal</u>	<u>Radix r</u>	<u>Binary</u>
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

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

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

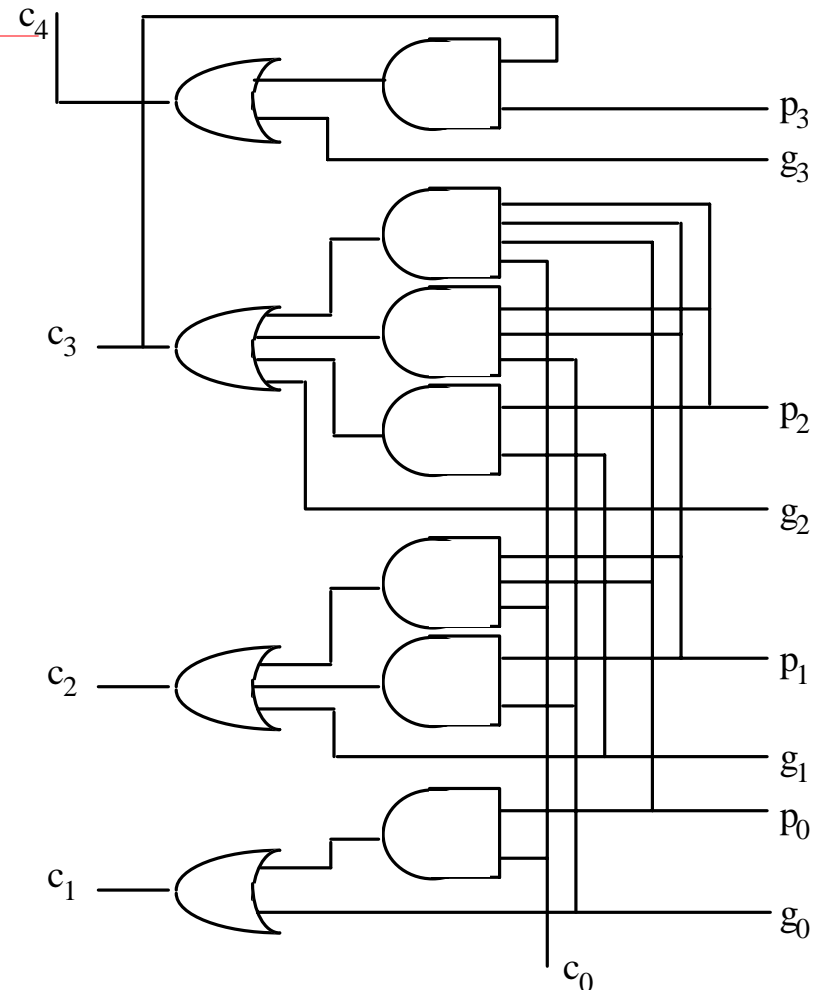
Full carry lookahead is quite practical for a 4-bit adder

$$c_1 = g_0 \vee c_0 p_0$$

$$c_2 = g_1 \vee g_0 p_1 \vee c_0 p_0 p_1$$

$$c_3 = g_2 \vee g_1 p_2 \vee g_0 p_1 p_2 \vee c_0 p_0 p_1 p_2$$

$$c_4 = g_3 \vee g_2 p_3 \vee g_1 p_2 p_3 \vee g_0 p_1 p_2 p_3 \vee c_0 p_0 p_1 p_2 p_3$$



Four-bit carry network with full lookahead.

Carry Lookahead Beyond 4 Bits

Consider a 32-bit adder

$$c_1 = g_0 \vee c_0 p_0$$

$$c_2 = g_1 \vee g_0 p_1 \vee c_0 p_0 p_1$$

$$c_3 = g_2 \vee g_1 p_2 \vee g_0 p_1 p_2 \vee c_0 p_0 p_1 p_2$$

⋮
⋮
⋮

$$c_{31} = g_{30} \vee g_{29} p_{30} \vee g_{28} p_{29} p_{30} \vee g_{27} p_{28} p_{29} p_{30} \vee \dots \vee c_0 p_0 p_1 p_2 p_3 \dots p_{29} p_{30}$$

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

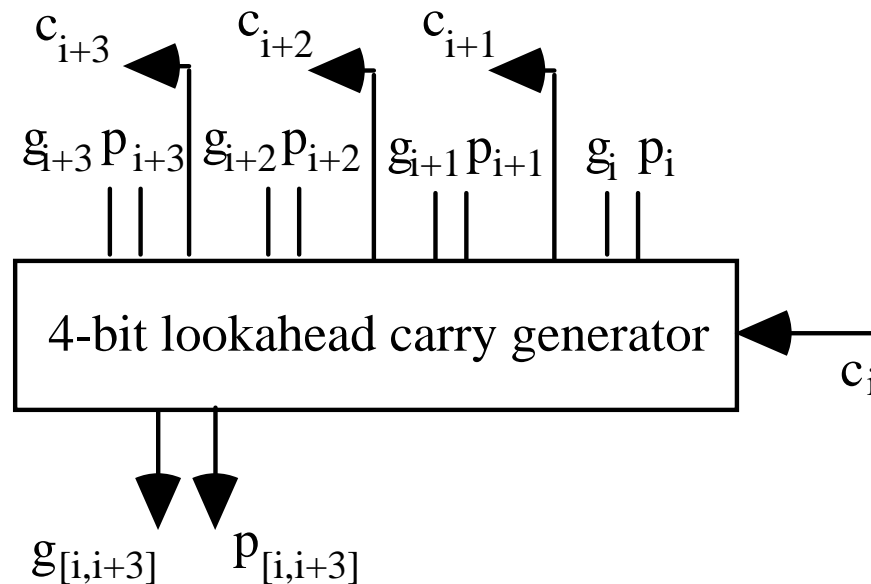
c_{16}	c_{15}	c_{14}	c_{13}	c_{12}	c_{11}	c_{10}	c_9	c_8	c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0
c_{out}				?				?				?				c_{in}

2 Carry-Lookahead Adder Design

Block *generate* and *propagate* signals

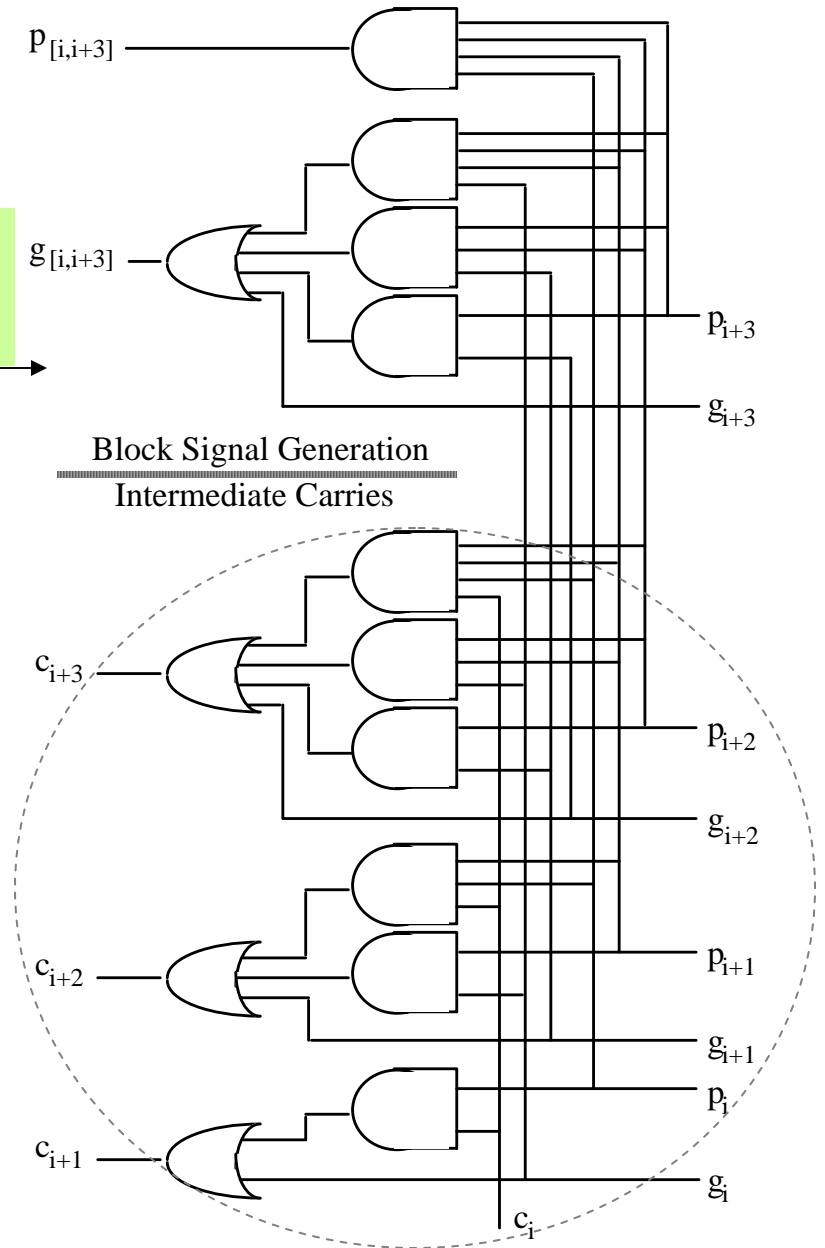
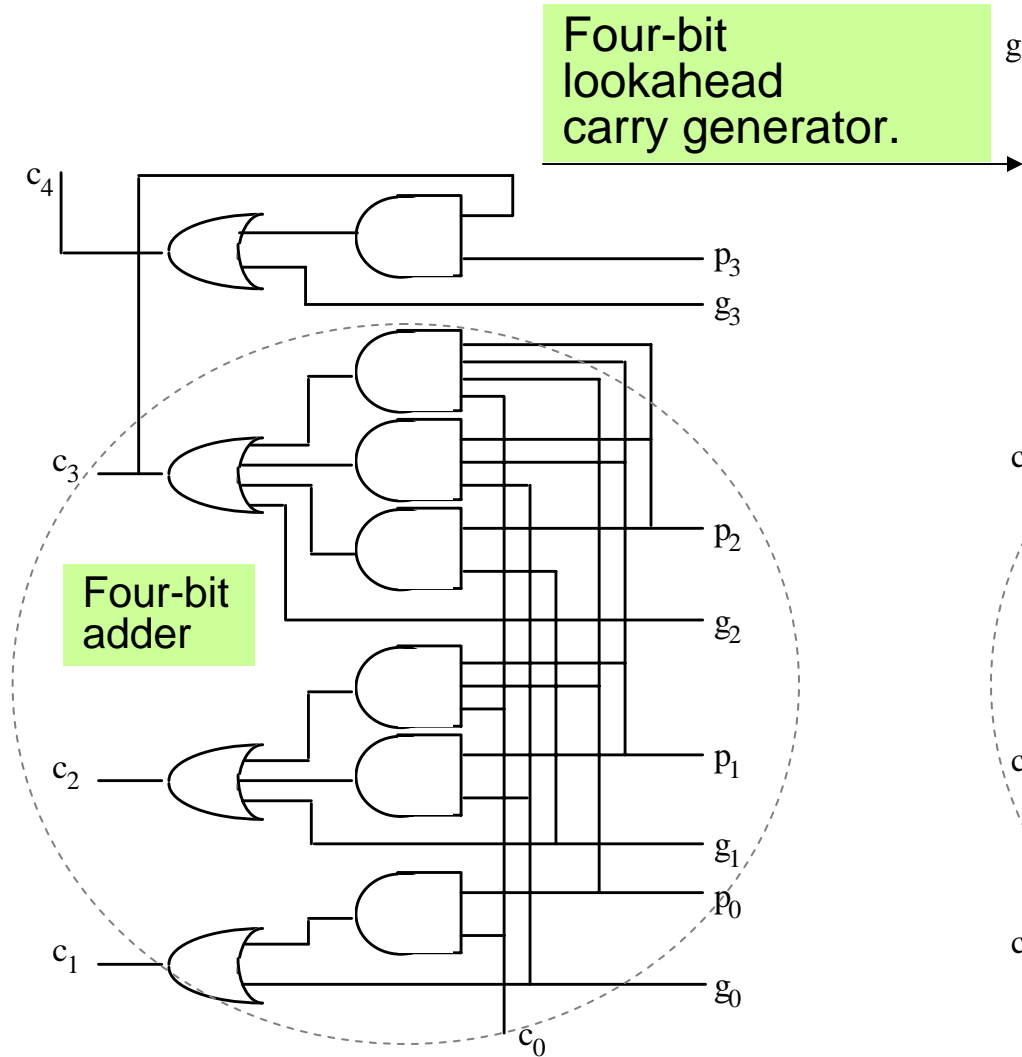
$$g_{[i,i+3]} = g_{i+3} \vee g_{i+2} p_{i+3} \vee g_{i+1} p_{i+2} p_{i+3} \vee g_i p_{i+1} p_{i+2} p_{i+3}$$

$$p_{[i,i+3]} = p_i p_{i+1} p_{i+2} p_{i+3}$$

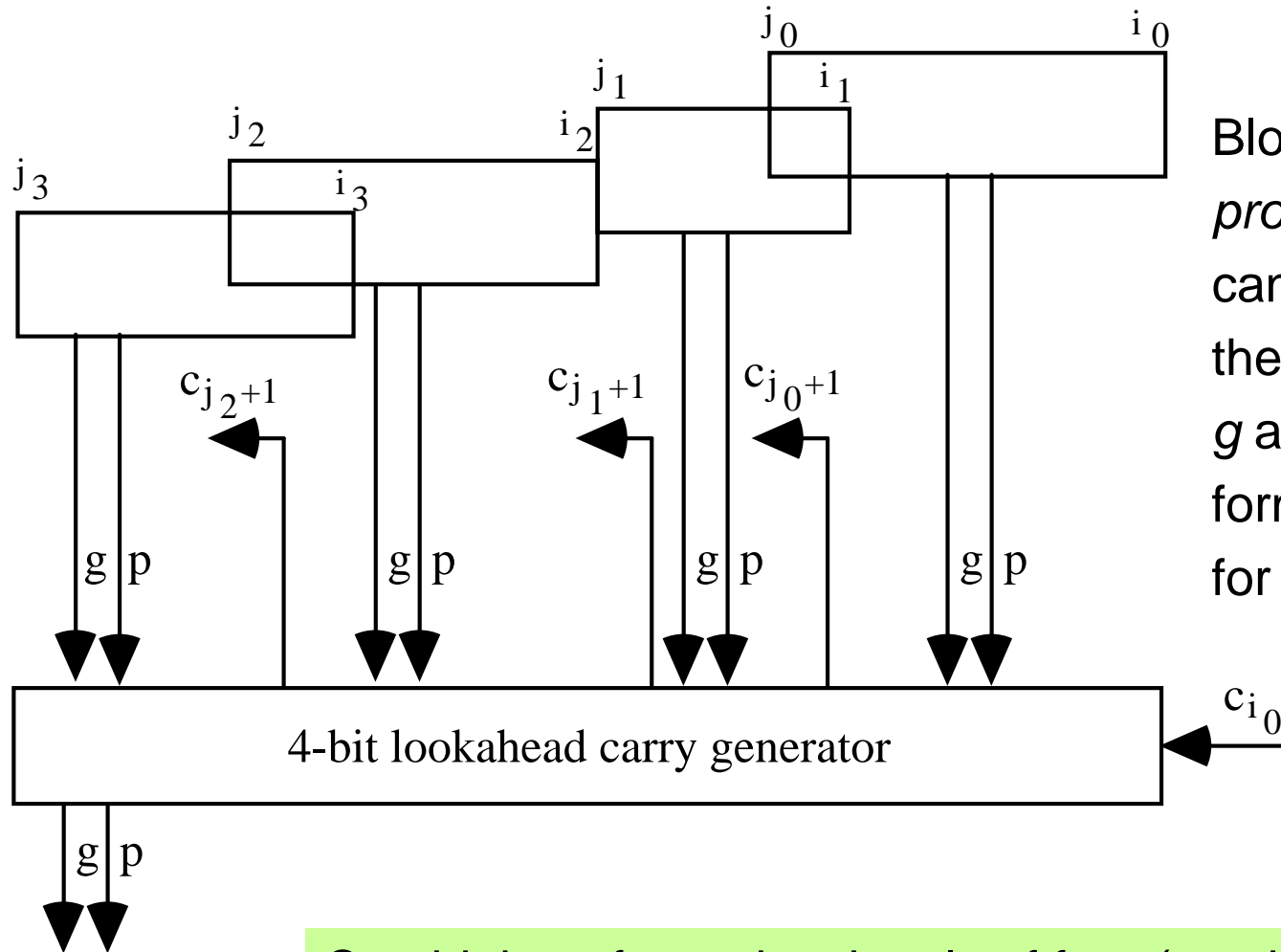


Schematic diagram of a 4-bit lookahead carry generator.

A Building Block for Carry-Lookahead Addition



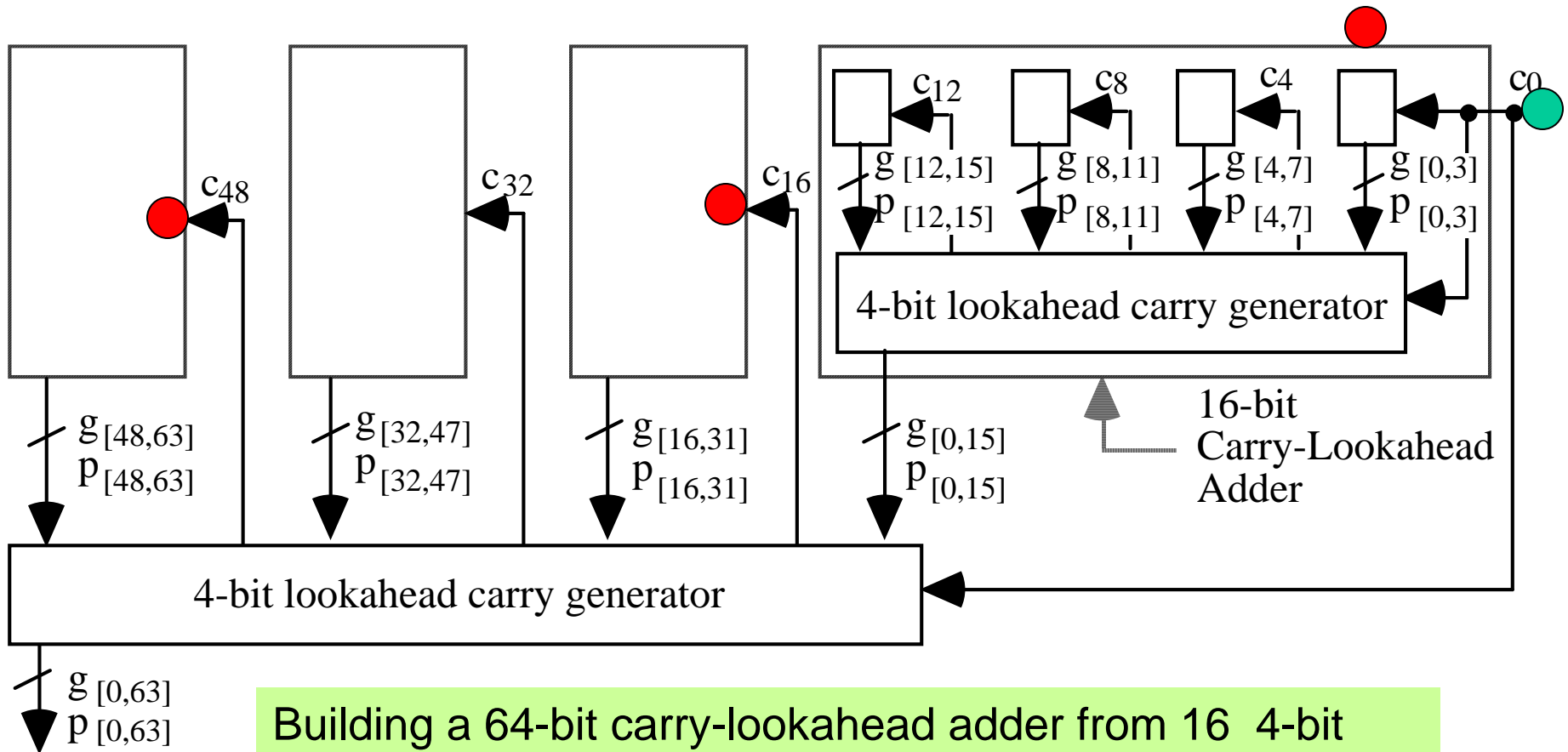
Combining Block g and p Signals



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

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



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

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 $h_i = c_i + c_{i-1}$ instead of c_i

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

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

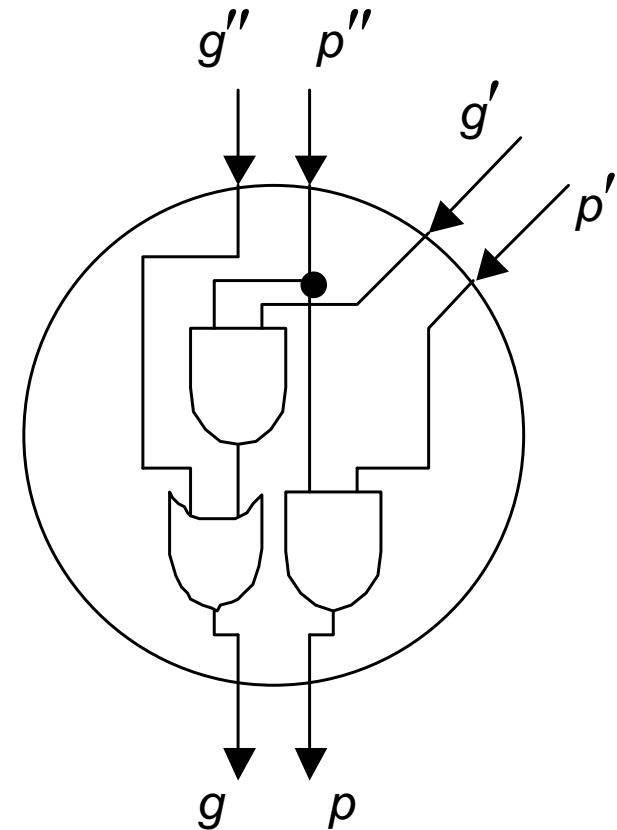
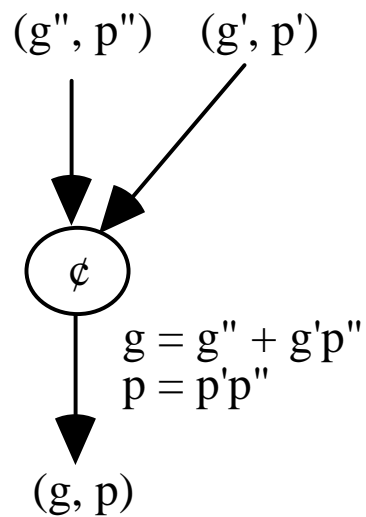
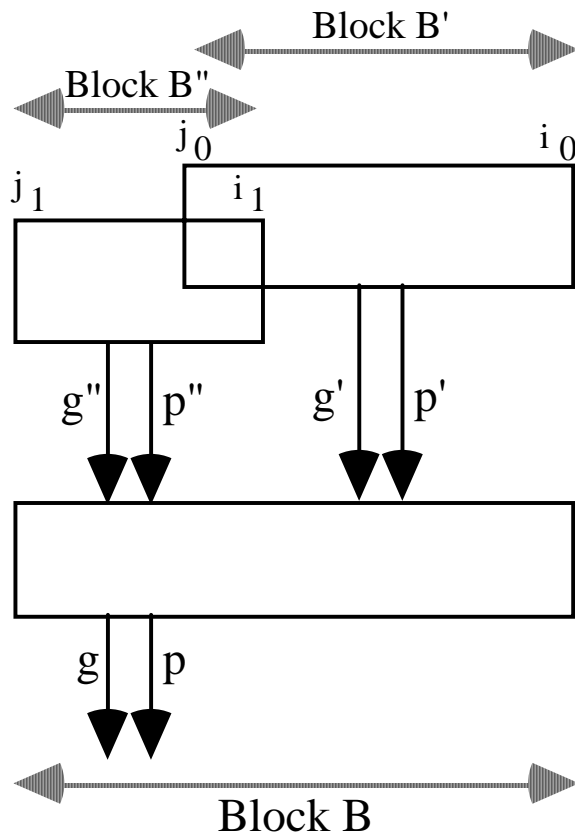
The advantage of h_i over c_i is even greater with wired-OR:

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 to $s_i = p_i \oplus c_i$

$$s_i = (t_i \oplus h_{i+1}) \vee h_i g_i t_{i-1}$$

4 Carry Determination as Prefix Computation



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:

$$\begin{array}{ccccccc}
 \text{Given} & (g_0, p_0) & & (g_1, p_1) & \dots & (g_{k-2}, p_{k-2}) & & (g_{k-1}, p_{k-1}) \\
 \text{Find} & (g_{[0,0]}, p_{[0,0]}) & & (g_{[0,1]}, p_{[0,1]}) & \dots & (g_{[0,k-2]}, p_{[0,k-2]}) & & (g_{[0,k-1]}, p_{[0,k-1]}) \\
 & \downarrow & & \downarrow & & \downarrow & & \downarrow \\
 & c_1 & & c_2 & \dots & c_{k-1} & & c_k
 \end{array}$$

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

$$\begin{array}{c}
 (g_0, p_0) \phi (g_1, p_1) \phi \dots \phi (g_{k-2}, p_{k-2}) \phi (g_{k-1}, p_{k-1}) \\
 \xrightarrow{\hspace{10em}} \\
 \xrightarrow{\hspace{10em}}
 \end{array}$$

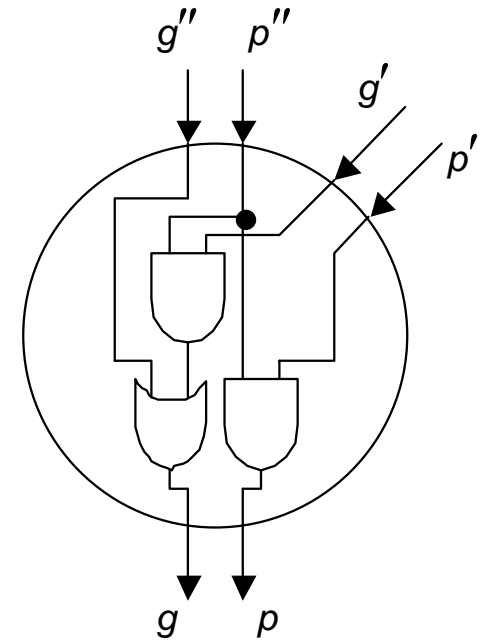
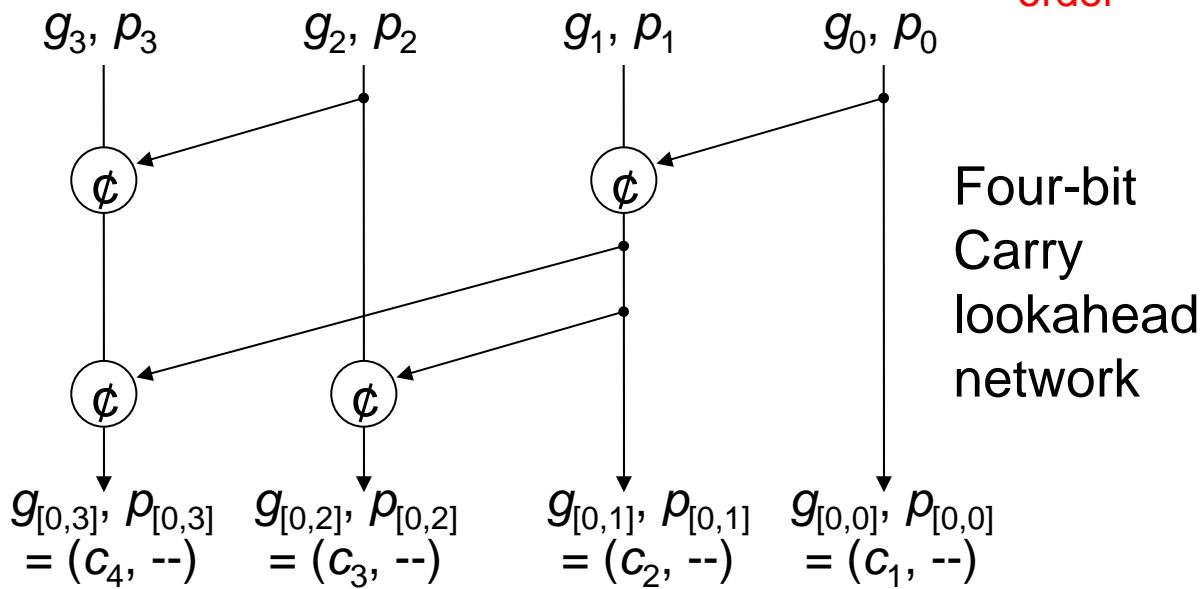
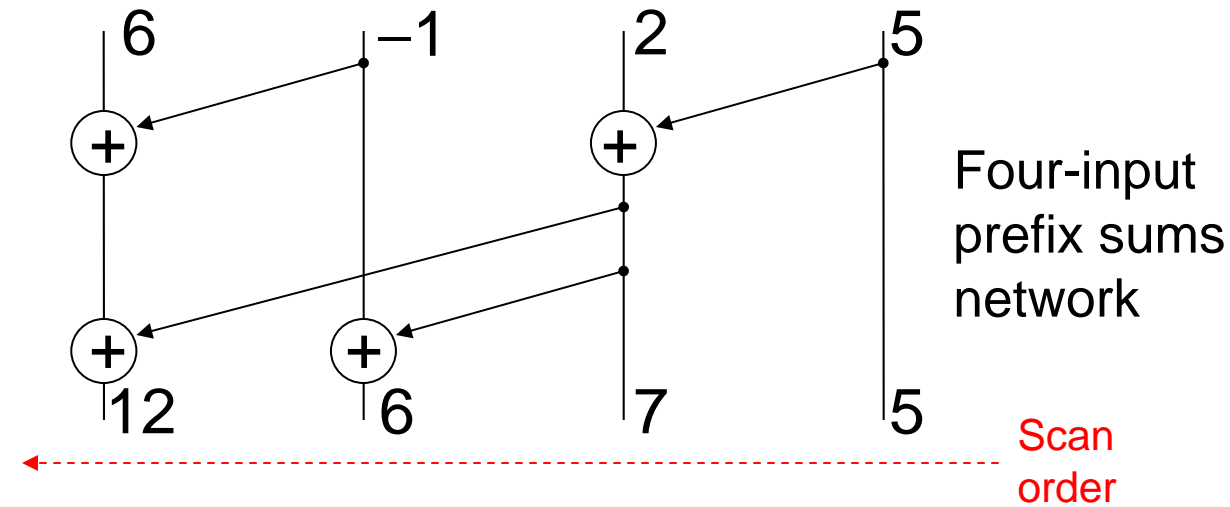
The carry operator ϕ is associative, but not commutative

$$[(g_1, p_1) \phi (g_2, p_2)] \phi (g_3, p_3) = (g_1, p_1) \phi [(g_2, p_2) \phi (g_3, p_3)]$$

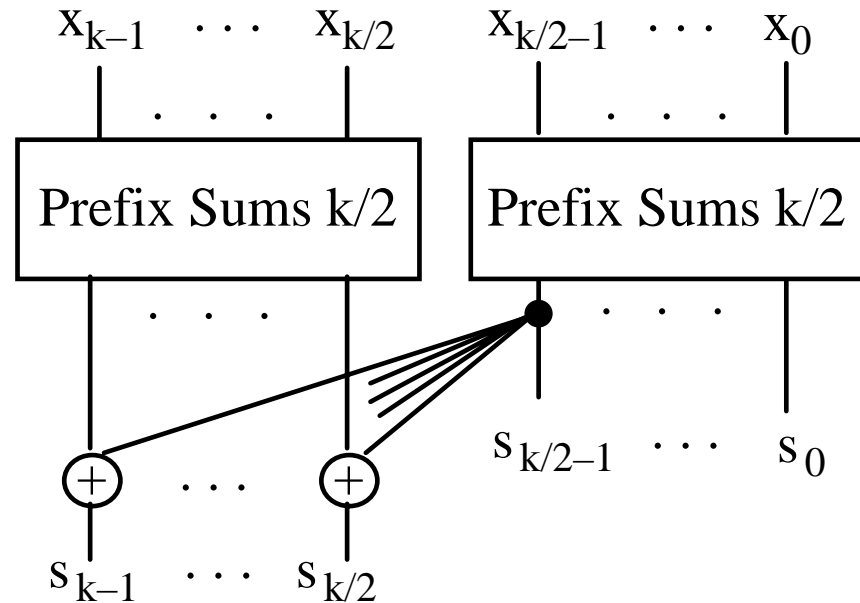
Prefix sums analogy:

$$\begin{array}{ccccccc}
 \text{Given} & x_0 & & x_1 & & x_2 & \dots & & x_{k-1} \\
 \text{Find} & x_0 & & x_0+x_1 & & x_0+x_1+x_2 & \dots & & x_0+x_1+\dots+x_{k-1}
 \end{array}$$

Example Prefix-Based Carry Network



5 Alternative Parallel Prefix Networks



Parallel prefix sums network built of two $k/2$ -input networks and $k/2$ adders. (Ladner-Fischer)

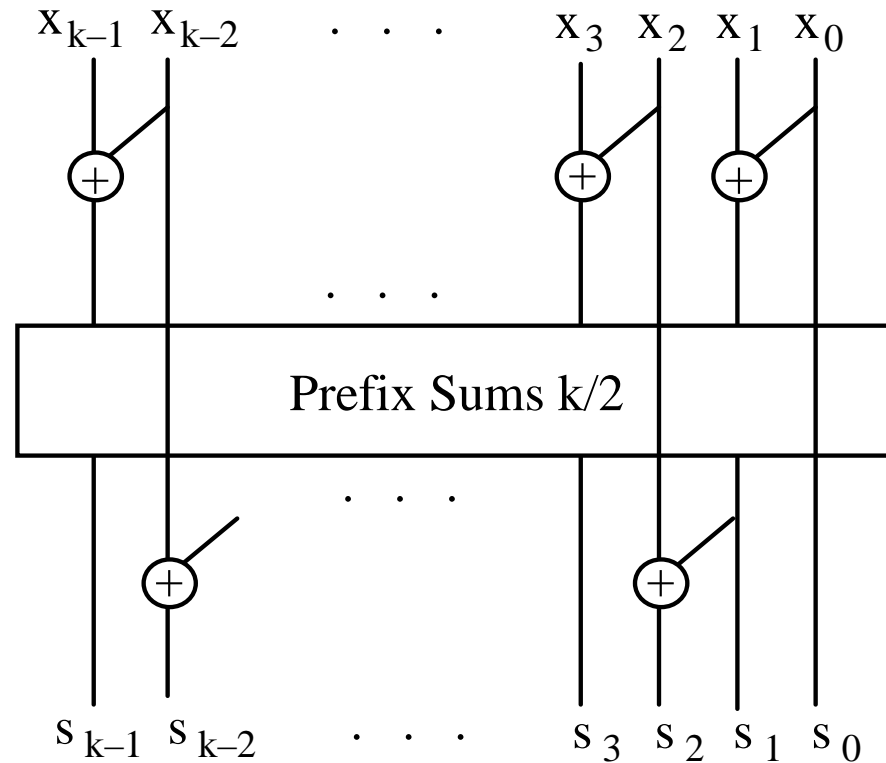
Delay recurrence

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

Cost recurrence

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

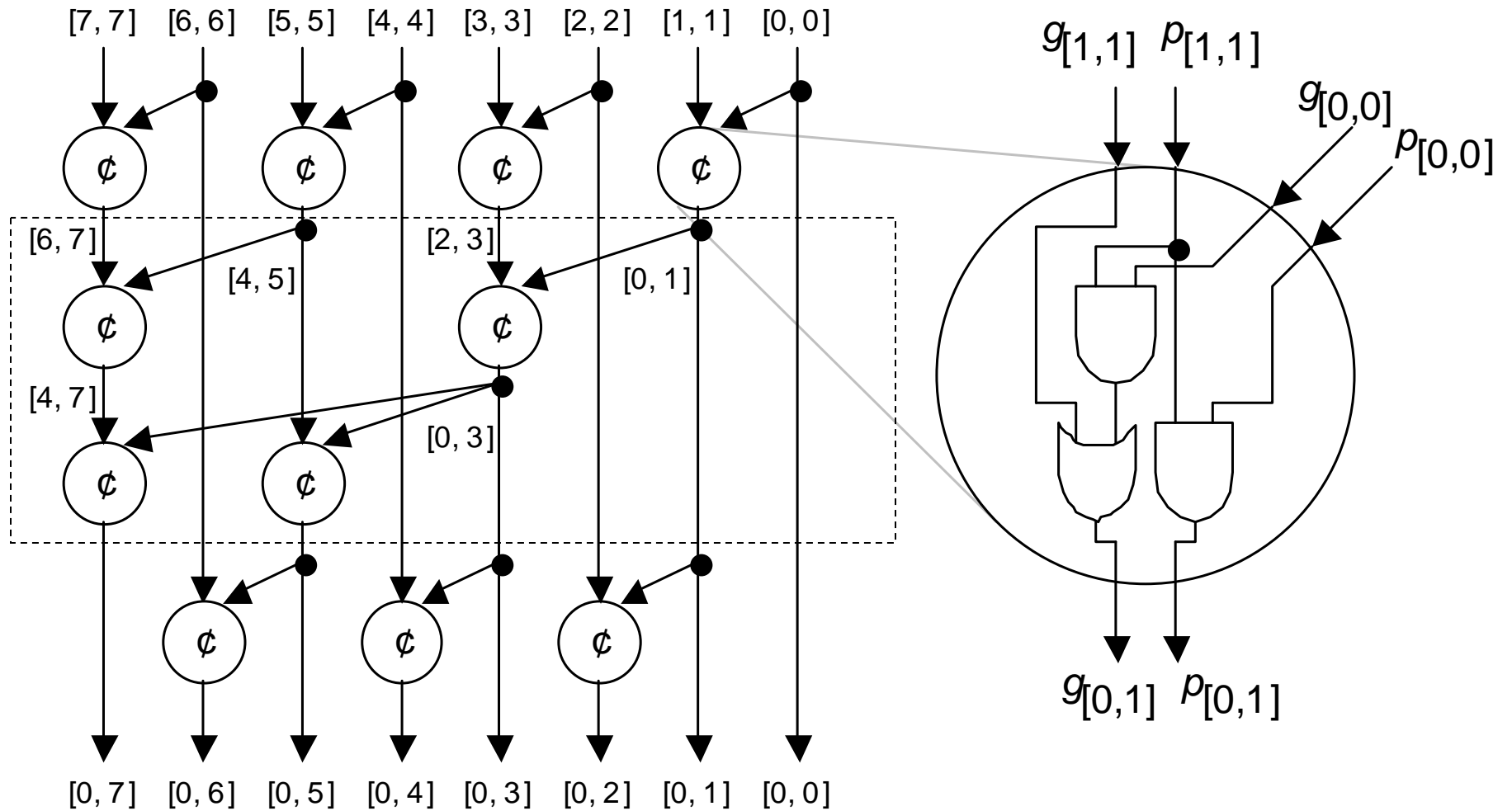
The Brent-Kung Recursive Construction



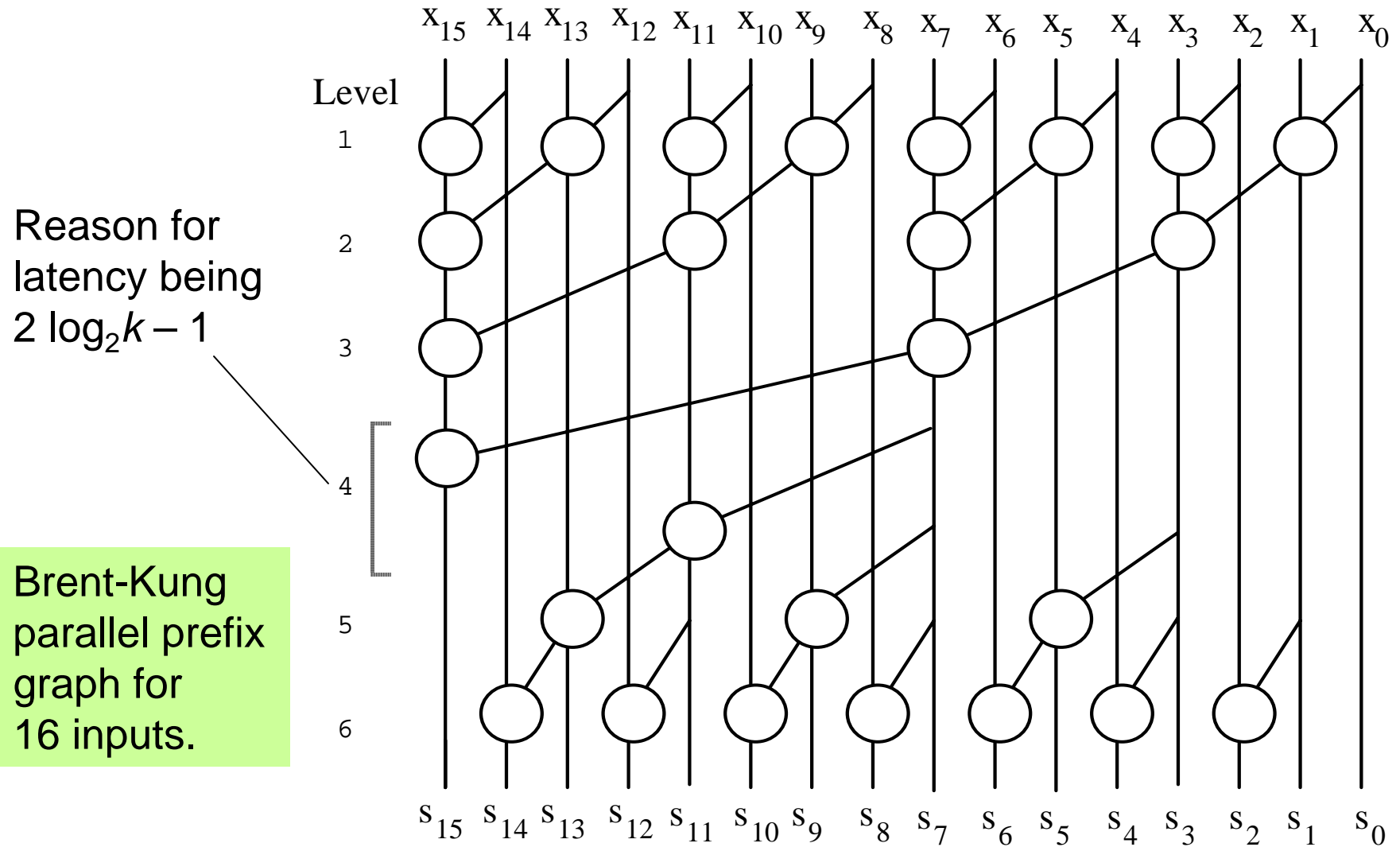
Parallel prefix sums network built of one $k/2$ -input network and $k - 1$ adders.

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)



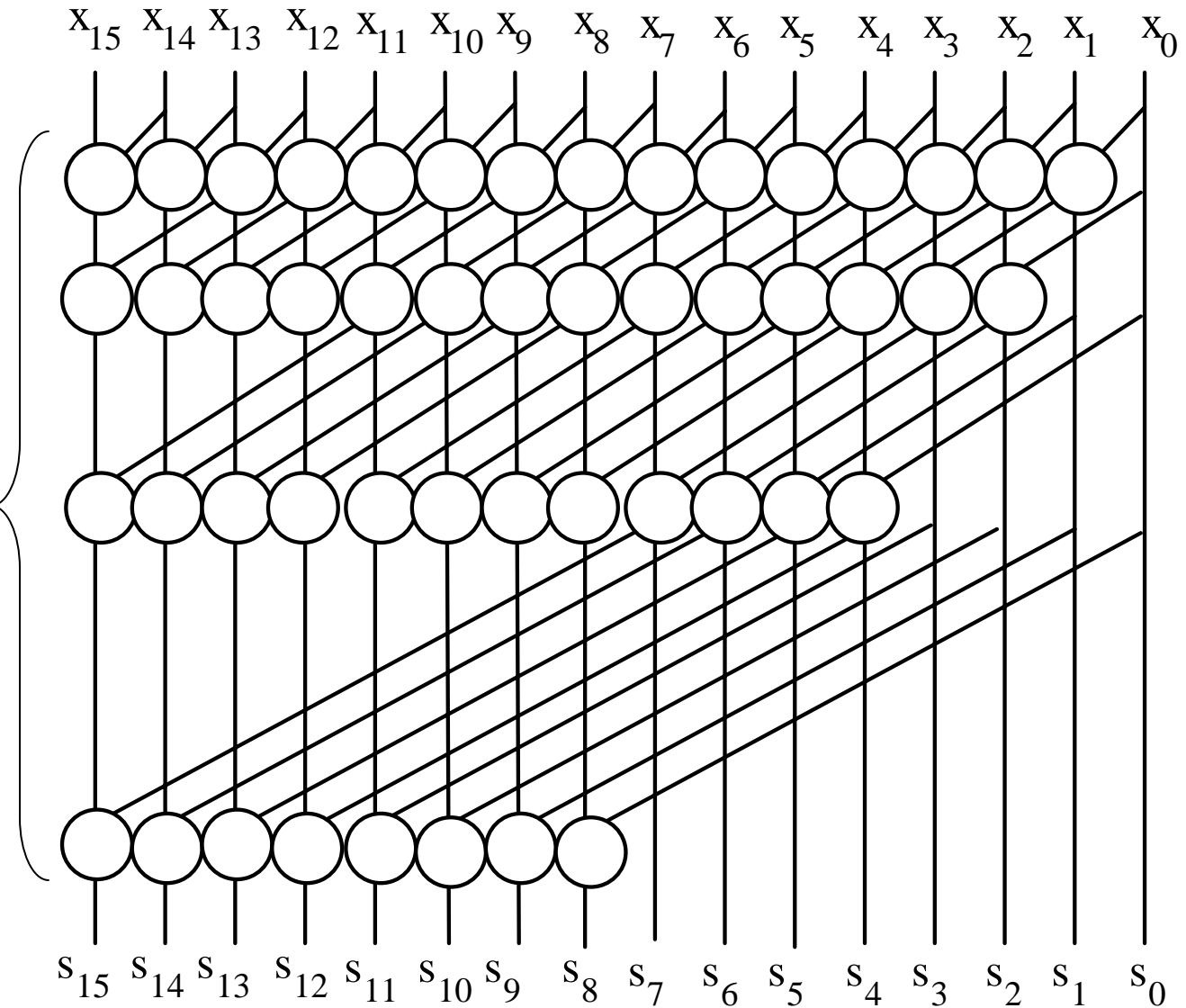
Kogge-Stone Carry Network (16-Bit Adder)

Cost formula

$$\begin{aligned}
 C(k) &= (k-1) \\
 &+ (k-2) \\
 &+ (k-4) + \dots \\
 &+ (k - k/2) \\
 &= k \log_2 k - k + 1
 \end{aligned}$$

$\log_2 k$ levels
(minimum possible)

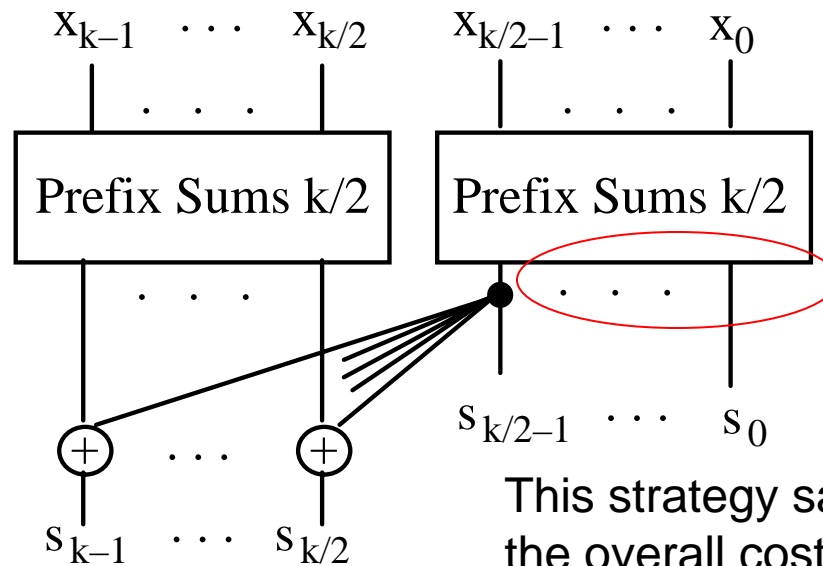
Kogge-Stone parallel prefix graph for 16 inputs.



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

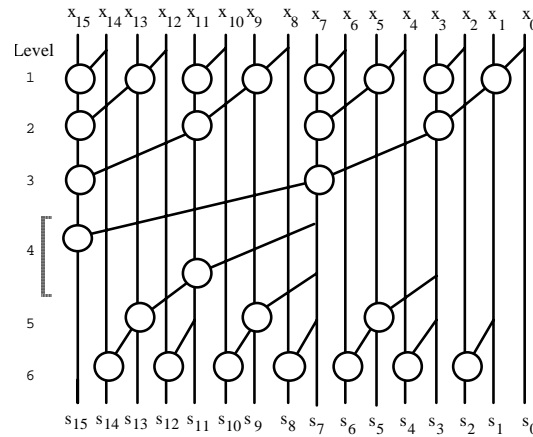


These outputs can
be produced one
time unit later without
increasing the overall
latency

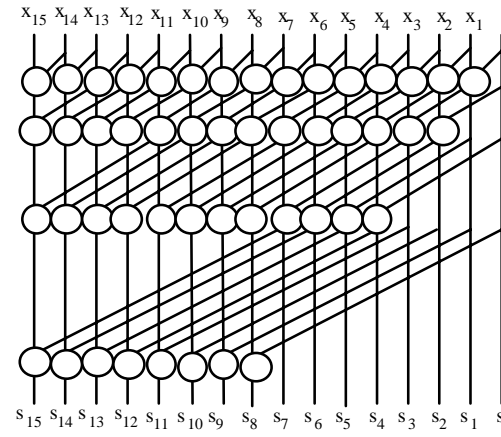
This strategy saves enough to make
the overall cost linear (best possible)

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

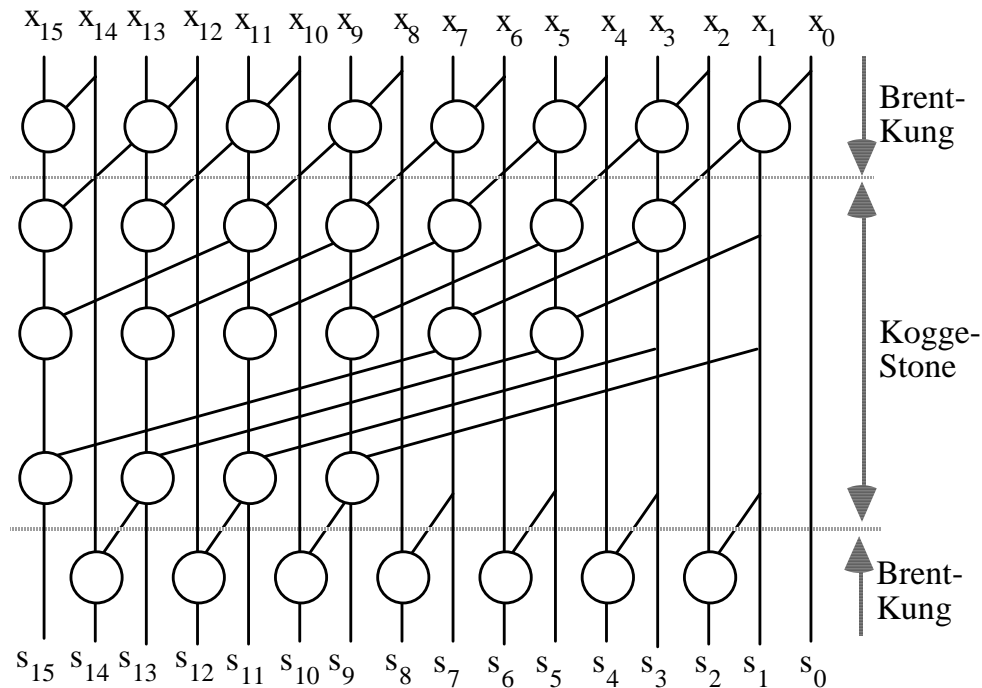
Brent-Kung:
6 levels
26 cells



Kogge-Stone:
4 levels
49 cells



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



Hybrid:
5 levels
32 cells

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

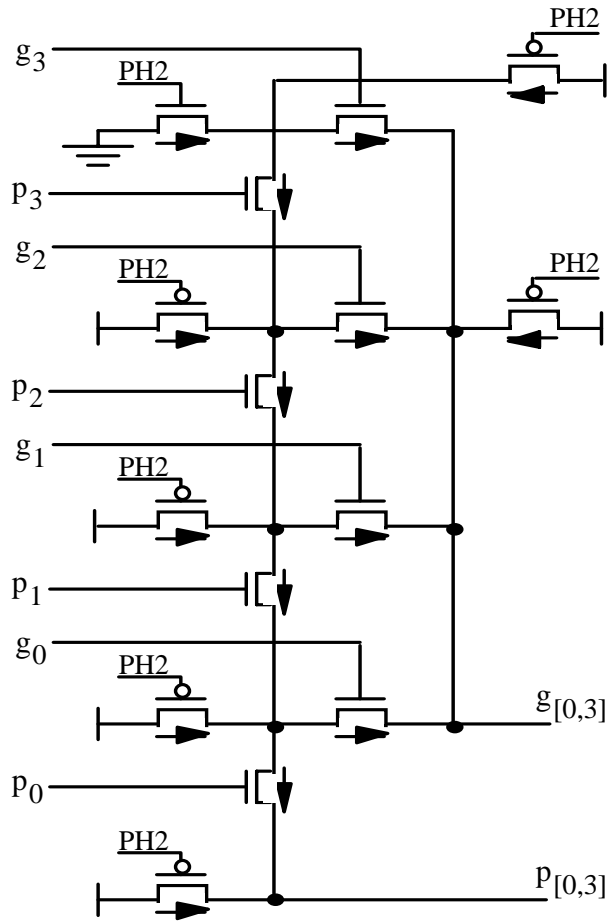
In radix-256, 64-bit addition, only these carries are needed:

C_{56} C_{48} C_{40} C_{32} C_{24} C_{16} C_8

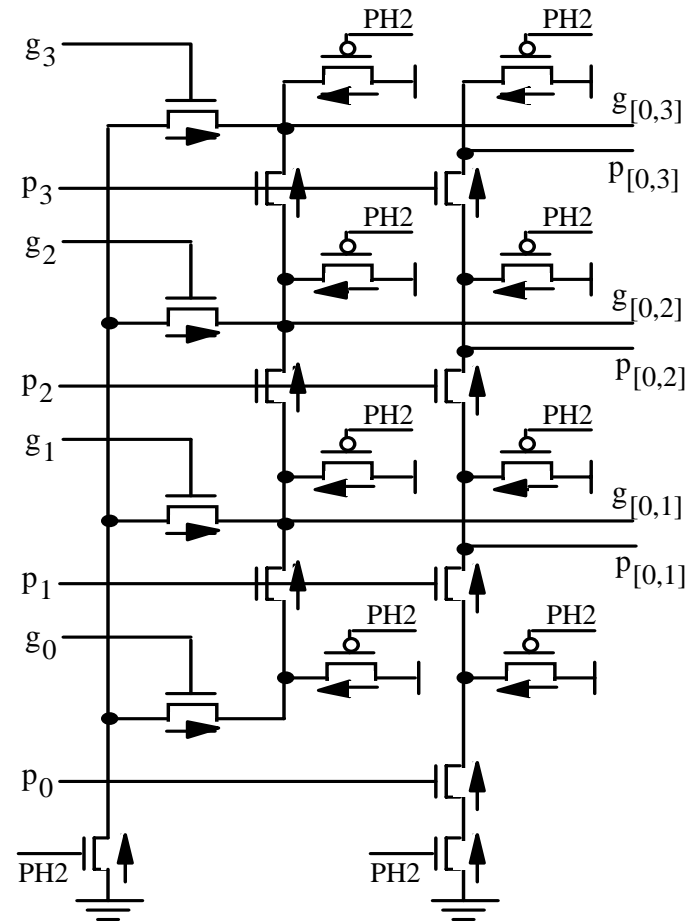
First, 4-bit Manchester carry chains (MCCs) of Fig. 6.12a are used to derive g and p signals for 4-bit blocks

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



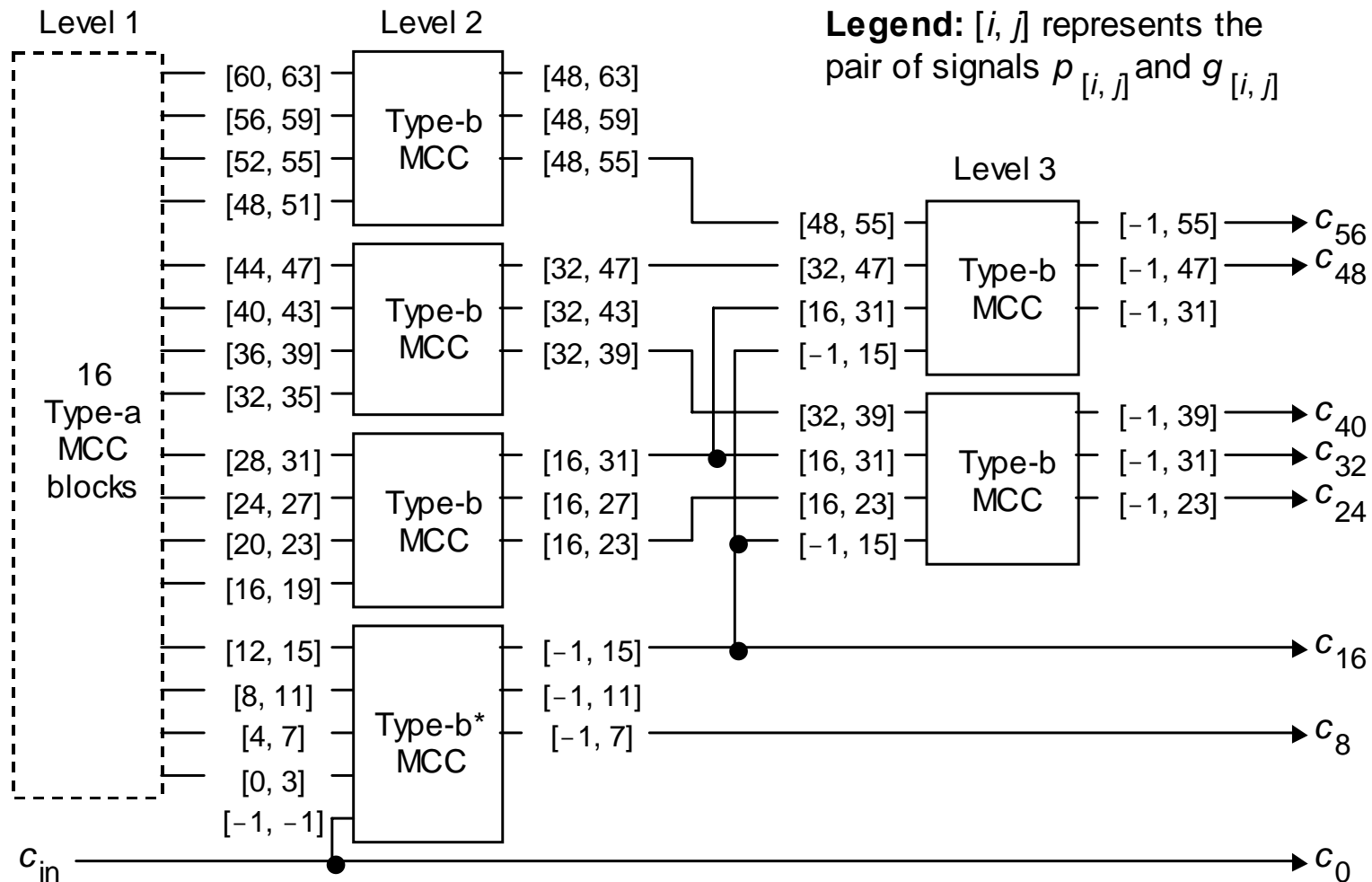
(a)



(b)

Example four-bit Manchester carry chain designs in CMOS technology [Lync92].

Carry Network for 64-Bit Adder



Spanning-tree carry-lookahead network [Lync92]. The 16 MCCs at level 1, that produce generate and propagate signals for 4-bit blocks, are not shown.

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)

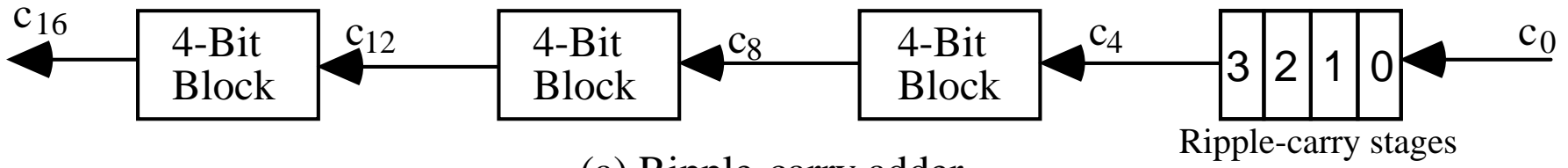
Knowledge of timing allows optimizations

Variations in Fast Adders: Topics

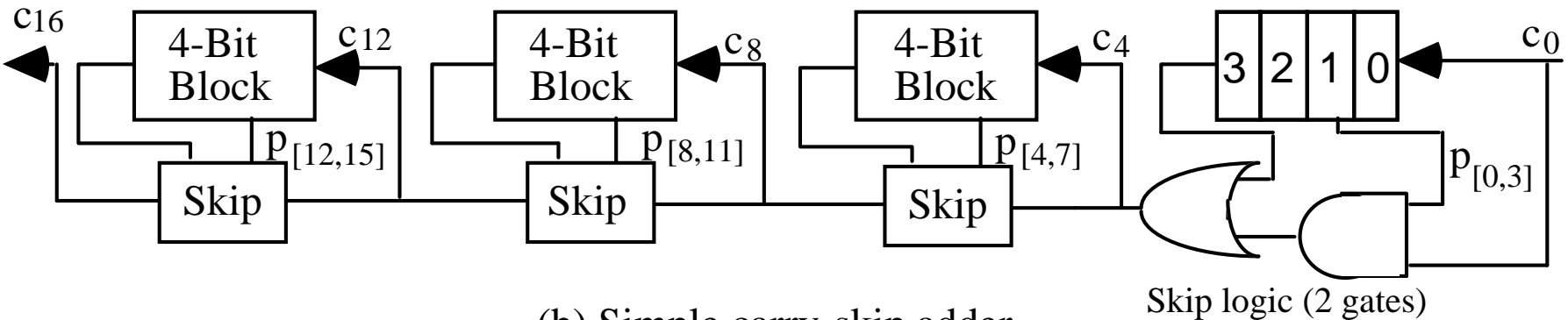
Topics in This Chapter

1. Simple Carry-Skip Adders
2. Multilevel Carry-Skip Adders
3. Carry-Select Adders
4. Conditional-Sum Adder
5. Hybrid Adder Designs
6. Optimizations in Fast Adders

1 Simple Carry-Skip Adders



(a) Ripple-carry adder.



(b) Simple carry-skip adder.

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

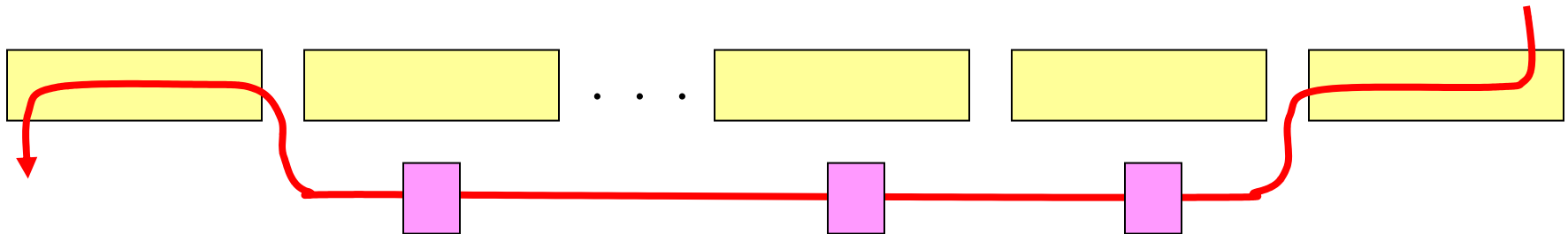
Carry-Skip Adder with Fixed Block Size

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

$$T_{\text{fixed-skip-add}} = \underbrace{(b-1)}_{\text{in block 0}} + \underbrace{0.5}_{\text{OR gate}} + \underbrace{(k/b-2)}_{\text{skips}} + \underbrace{(b-1)}_{\text{in last block}}$$
$$\cong 2b + k/b - 3.5 \text{ stages}$$

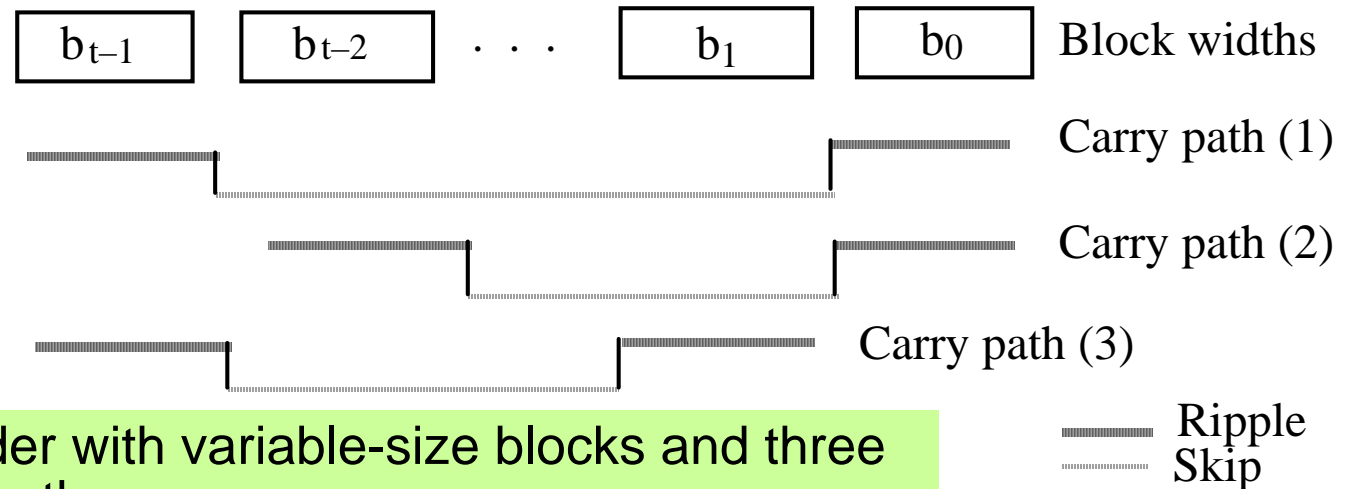
$$dT/db = 2 - k/b^2 = 0 \quad \Rightarrow \quad b^{\text{opt}} = \sqrt{k/2}$$

$$T^{\text{opt}} = 2\sqrt{2k} - 3.5$$



Example: $k = 32$, $b^{\text{opt}} = 4$, $T^{\text{opt}} = 12.5$ stages
(contrast with 32 stages for a ripple-carry adder)

Carry-Skip Adder with Variable-Width Blocks



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

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

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

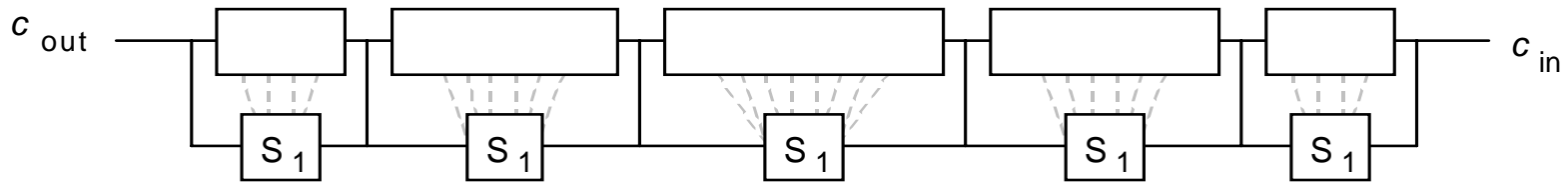
$$b = k/t - t/4 + 1/2$$

$$T_{\text{var-skip-add}} = 2(b - 1) + 0.5 + t - 2 = 2k/t + t/2 - 2.5$$

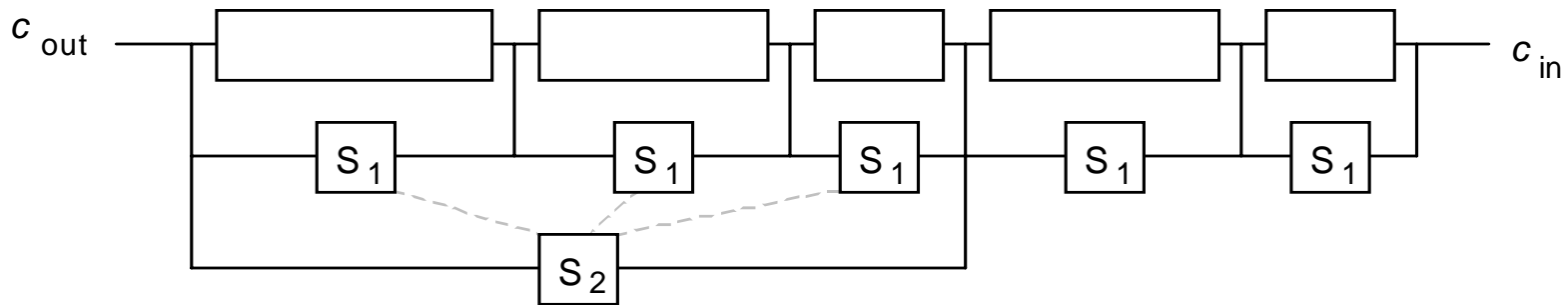
$$dT/db = -2k/t^2 + 1/2 = 0 \quad \Rightarrow \quad t^{\text{opt}} = 2\sqrt{k}$$

$$T^{\text{opt}} = 2\sqrt{k} - 2.5 \quad (\text{a factor of } \sqrt{2} \text{ smaller than for fixed-block})$$

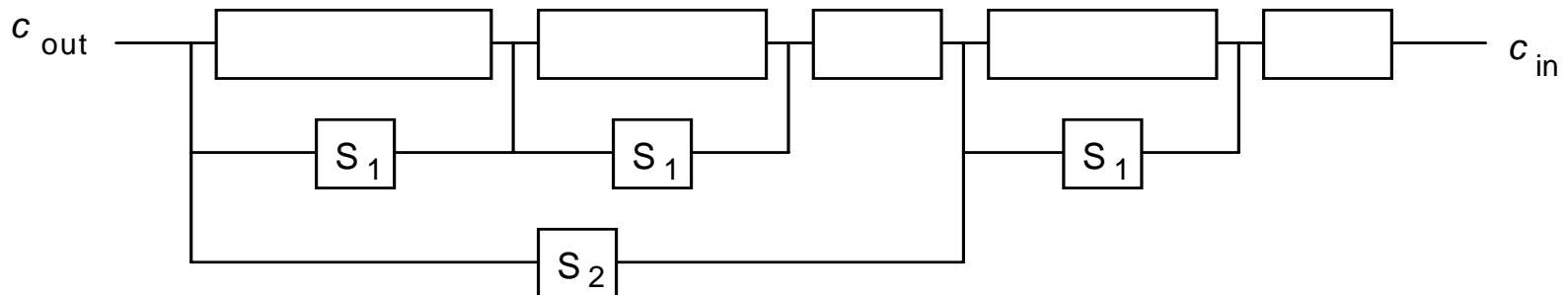
2 Multilevel Carry-Skip Adders



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



Example of a two-level carry-skip adder.



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

Designing a Single-Level Carry-Skip Adder

Example 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

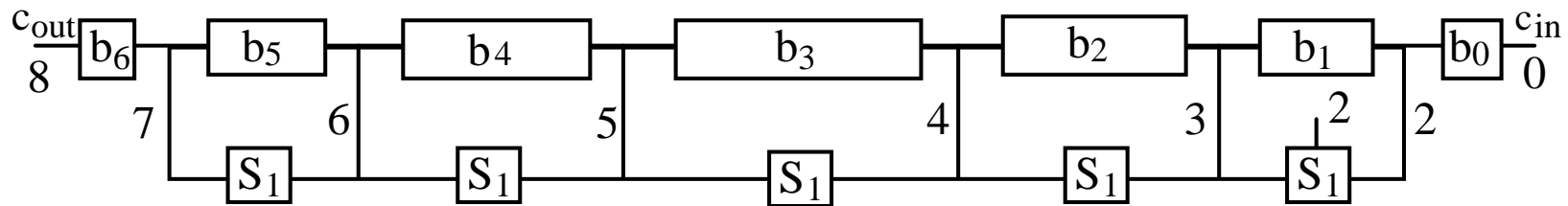


Fig. 7.6 Timing constraints of a single-level carry-skip adder with a delay of 8 units.

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

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

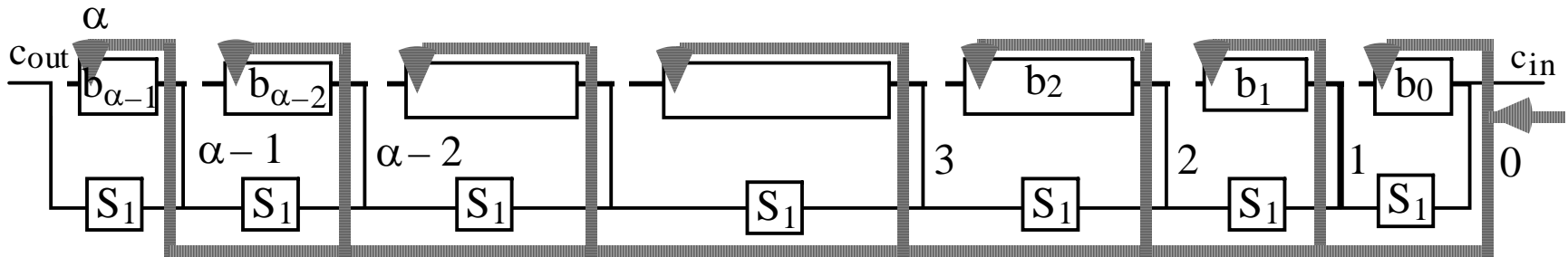
1	2	3	...	$T/2$	$T/2$...	4	3	1
1	2	3	...	$(T+1)/2$...	4	3	1	

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

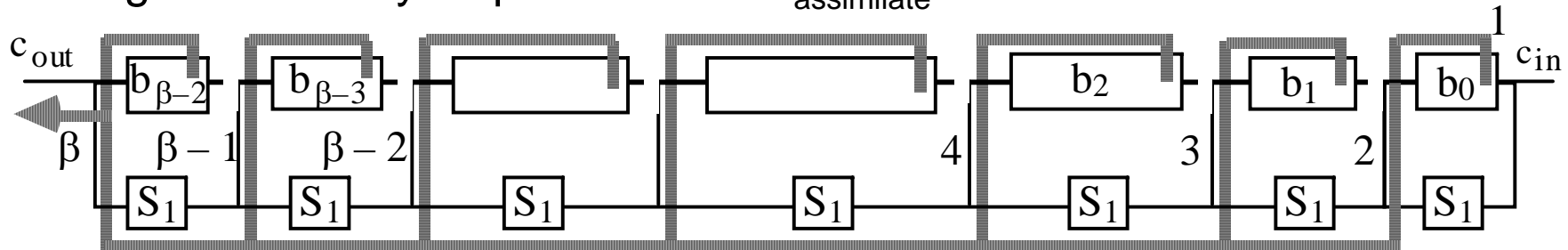
Elaboration on Two-Level Carry-Skip Adder

Example 2

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



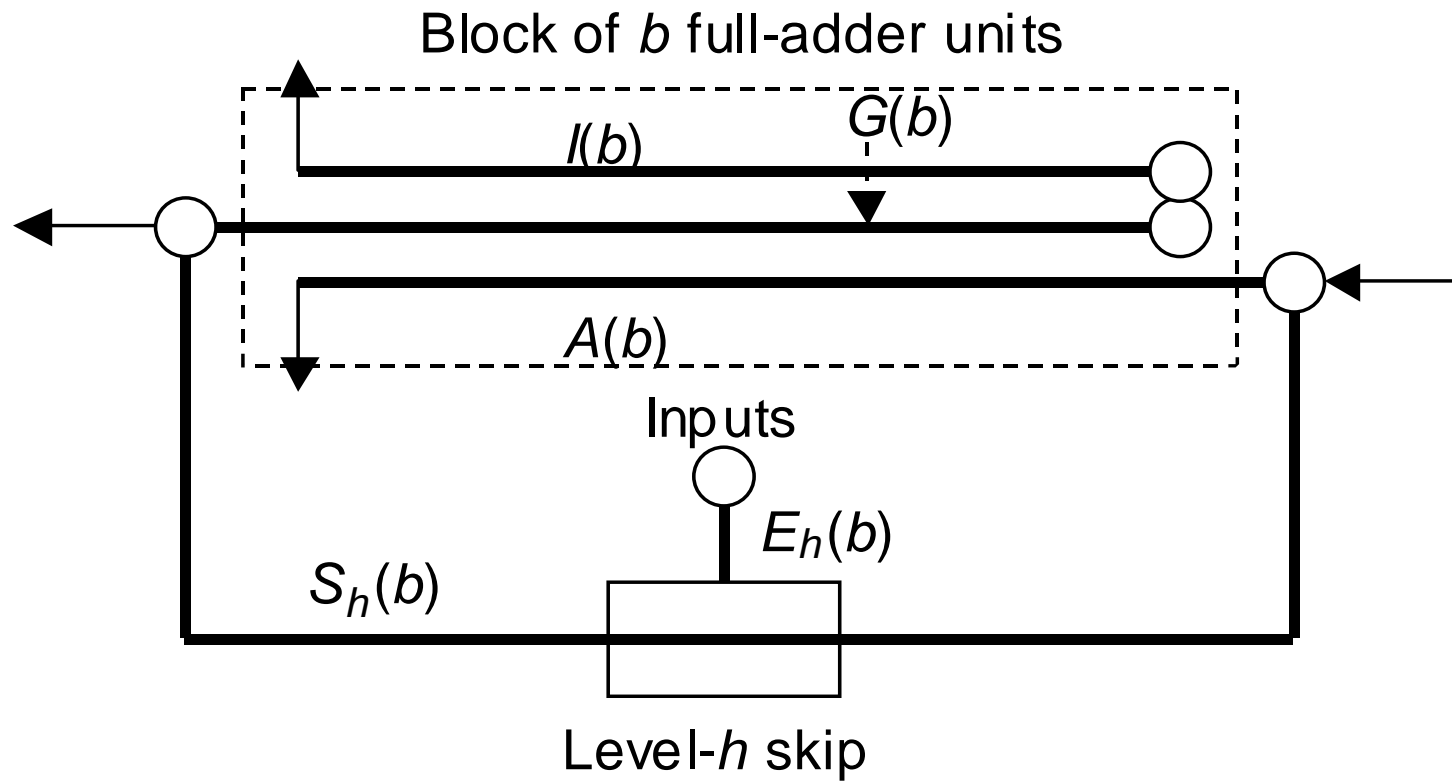
Single-level carry-skip adder with $T_{\text{assimilate}} = \alpha$



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

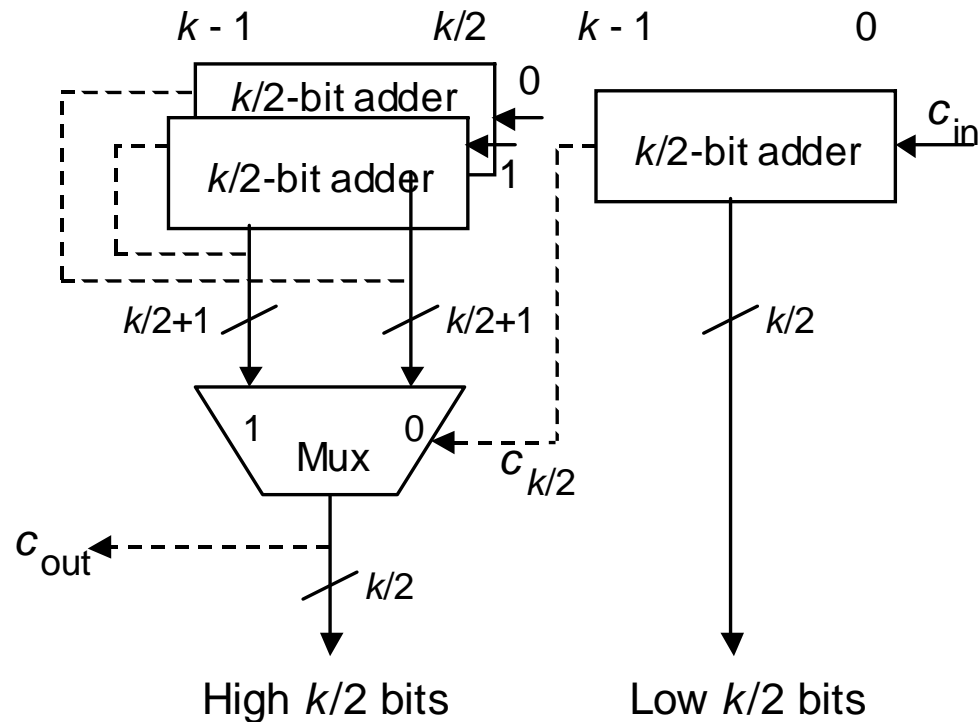
Width of the i th level-1 block in the level-2 block characterized by $\{\beta, \alpha\}$ is $b_i = \min(\beta - \gamma + i + 1, \alpha - i)$; the total block width is then $\sum_{i=0}^{\gamma-1} b_i$

Carry-Skip Adder Optimization Scheme



Generalized delay model for carry-skip adders.

3 Carry-Select Adders

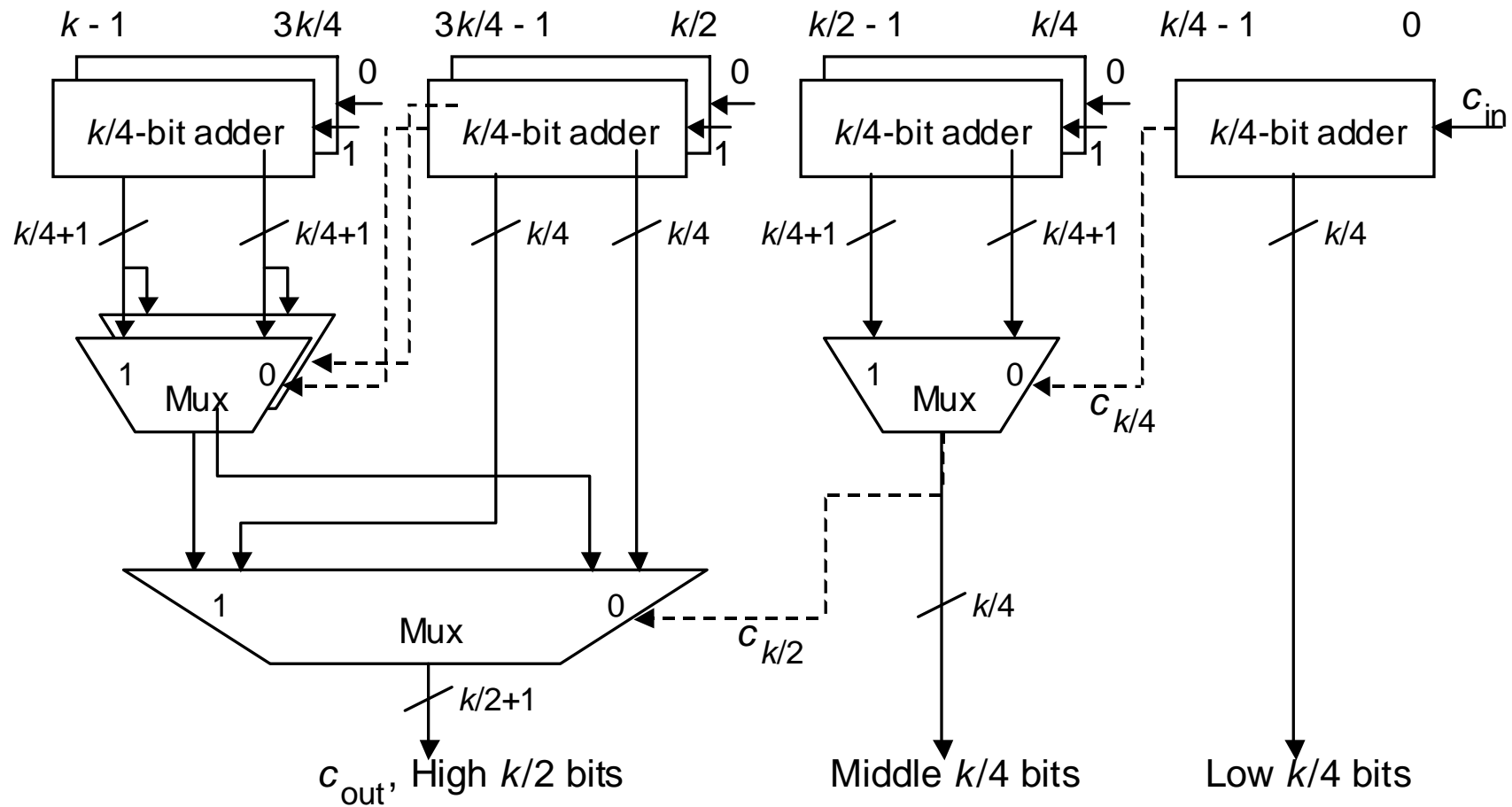


Carry-select adder for k -bit numbers built from three $k/2$ -bit adders.

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

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

Multilevel Carry-Select Adders



Two-level carry-select adder built of $k/4$ -bit adders.

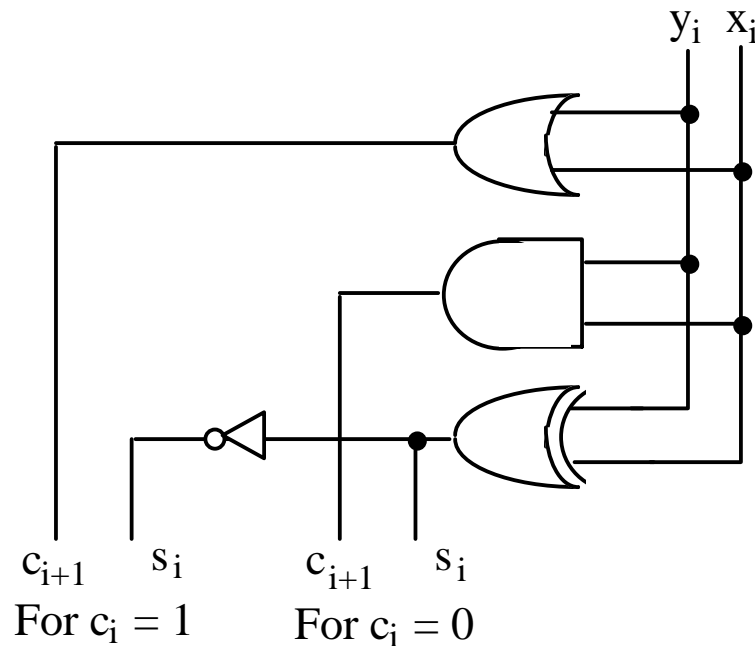
4 Conditional-Sum Adder

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.

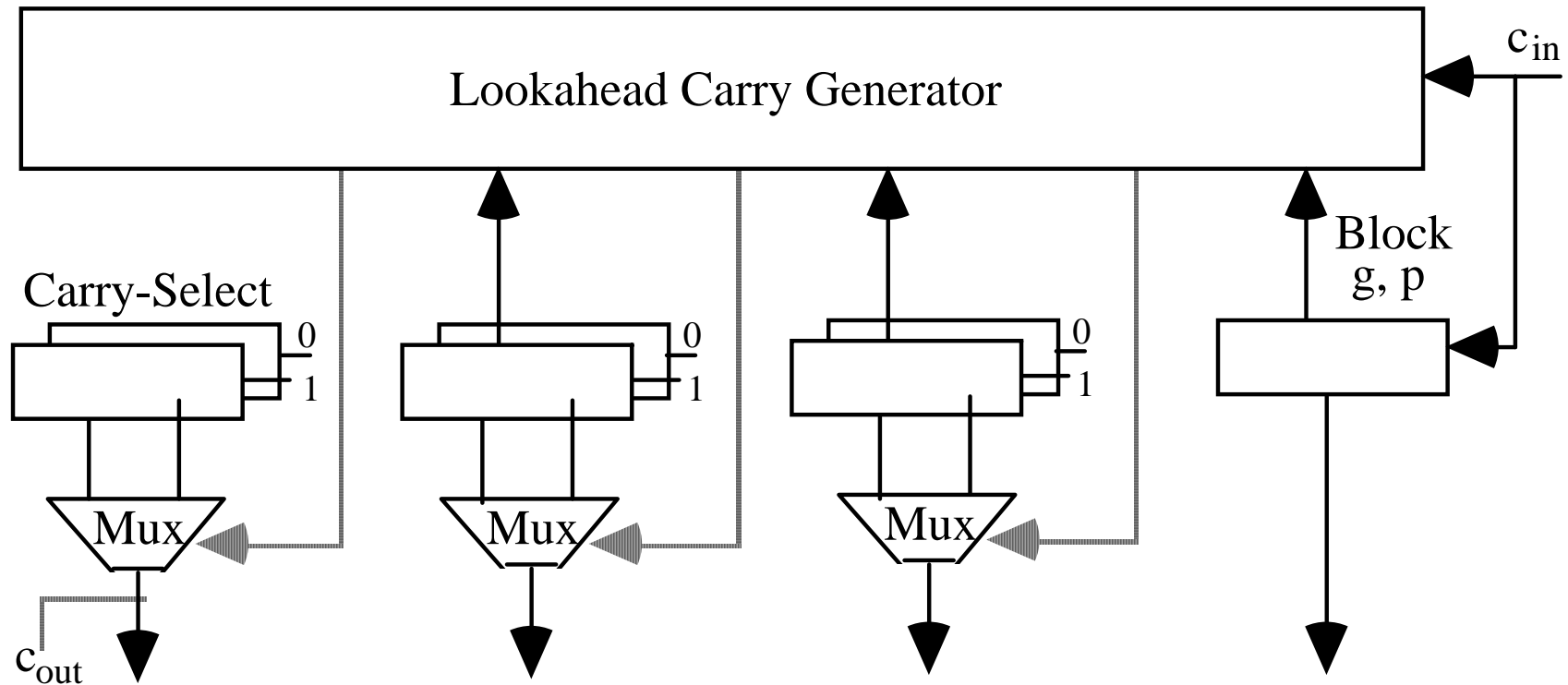
Conditional-Sum Addition Example

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	0	1	0	1	0	1	0
		y	0	1	0	0	1	0	1	1	0	1	0	1	1	1	0	1	0
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	0	1
		c	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0
1	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	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	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	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	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	1	s																	
		c																	

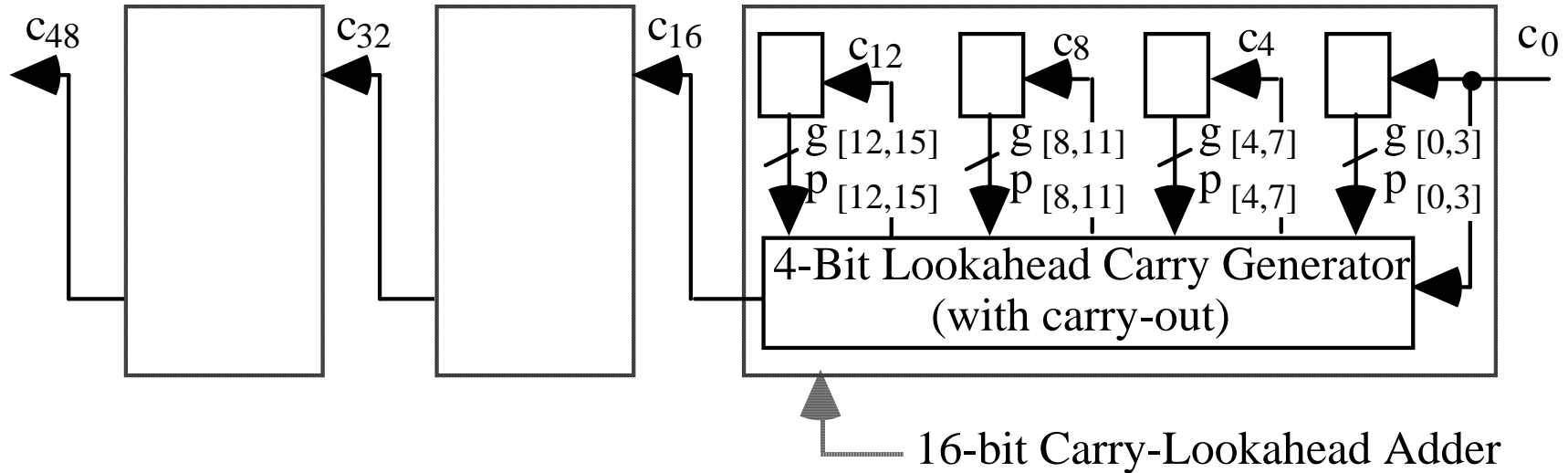
5 Hybrid Adder Designs

The most popular hybrid addition scheme:



A hybrid carry-lookahead/carry-select adder.

Any Two Addition Schemes Can Be Combined



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

Other possibilities:

- hybrid carry-select/ripple-carry
- hybrid ripple-carry/carry-select

...

6 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

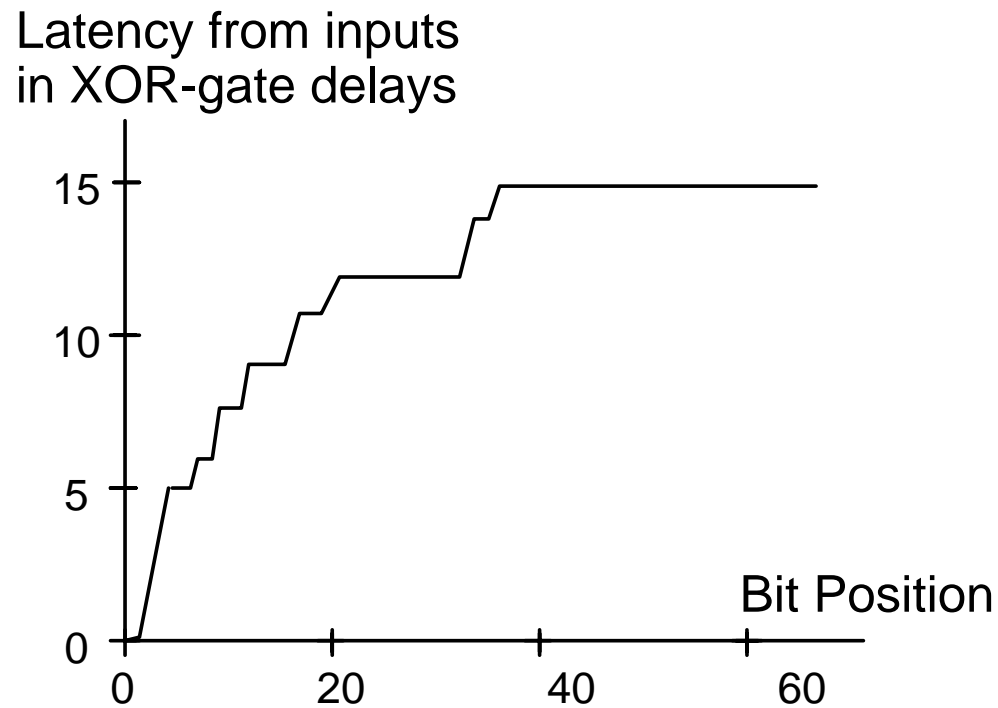
Variable-block carry-lookahead adder

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



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

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 \rightarrow New total
Deferred carry assimilation
Wallace/Dadda trees and parallel counters

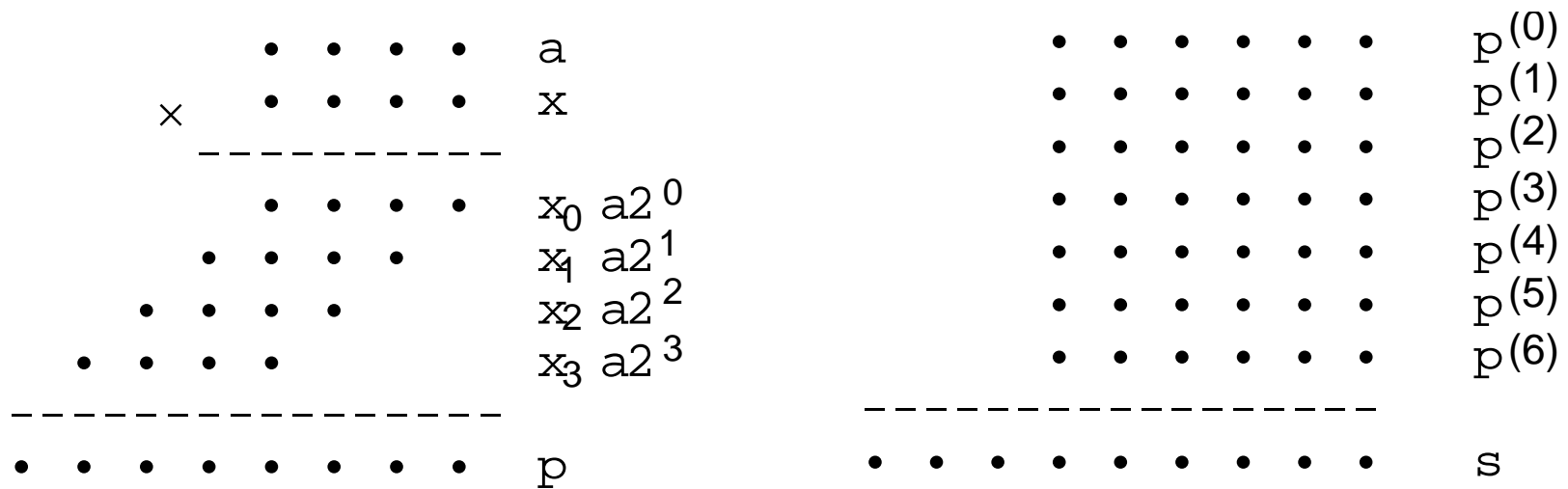
Multioperand Addition: Topics

Topics in This Chapter

1. Using Two-Operand Adders
2. Carry-Save Adders
3. Wallace and Dadda Trees
4. Parallel Counters
5. Generalized Parallel Counters
6. Adding Multiple Signed Numbers

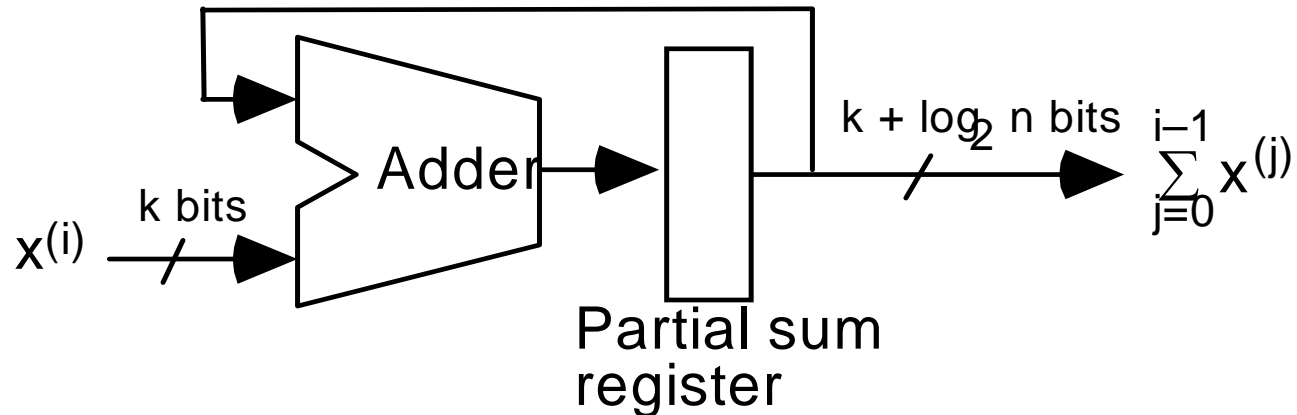
1 Using Two-Operand Adders

Some applications of multioperand addition



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

Serial Implementation with One Adder



Serial implementation of multi-operand addition with a single 2-operand adder.

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

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?

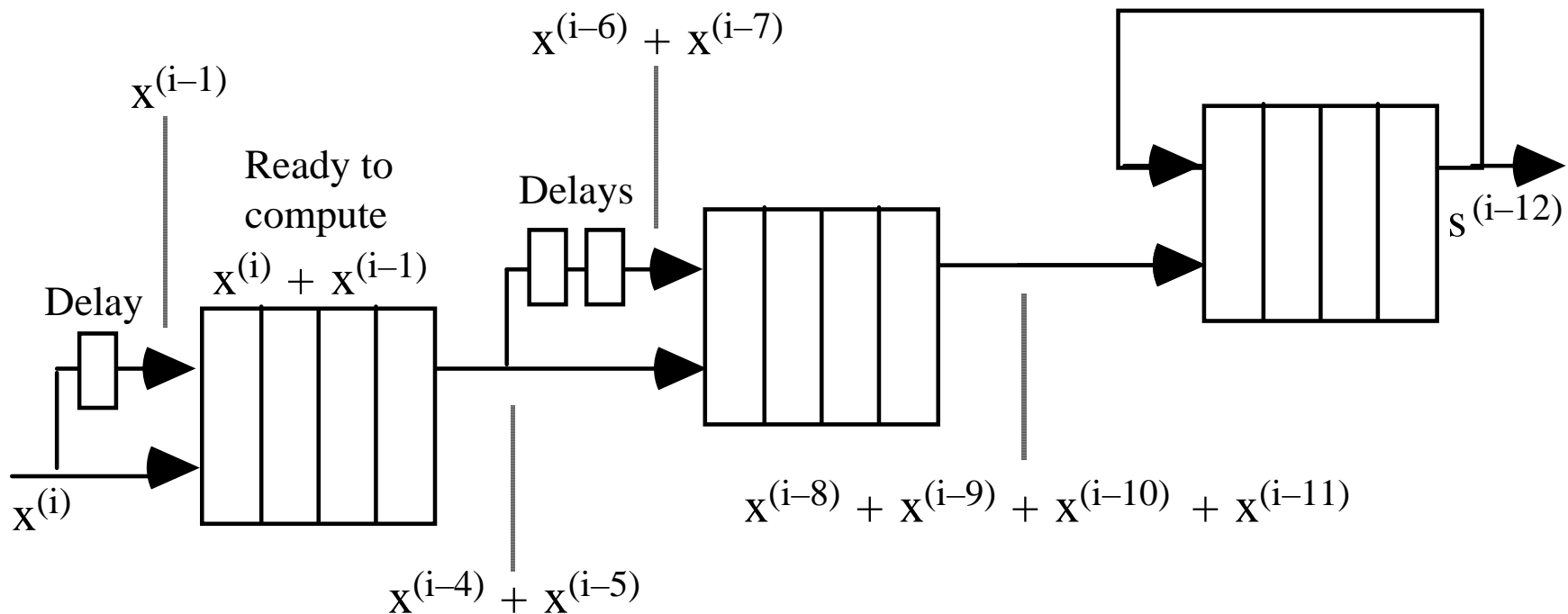
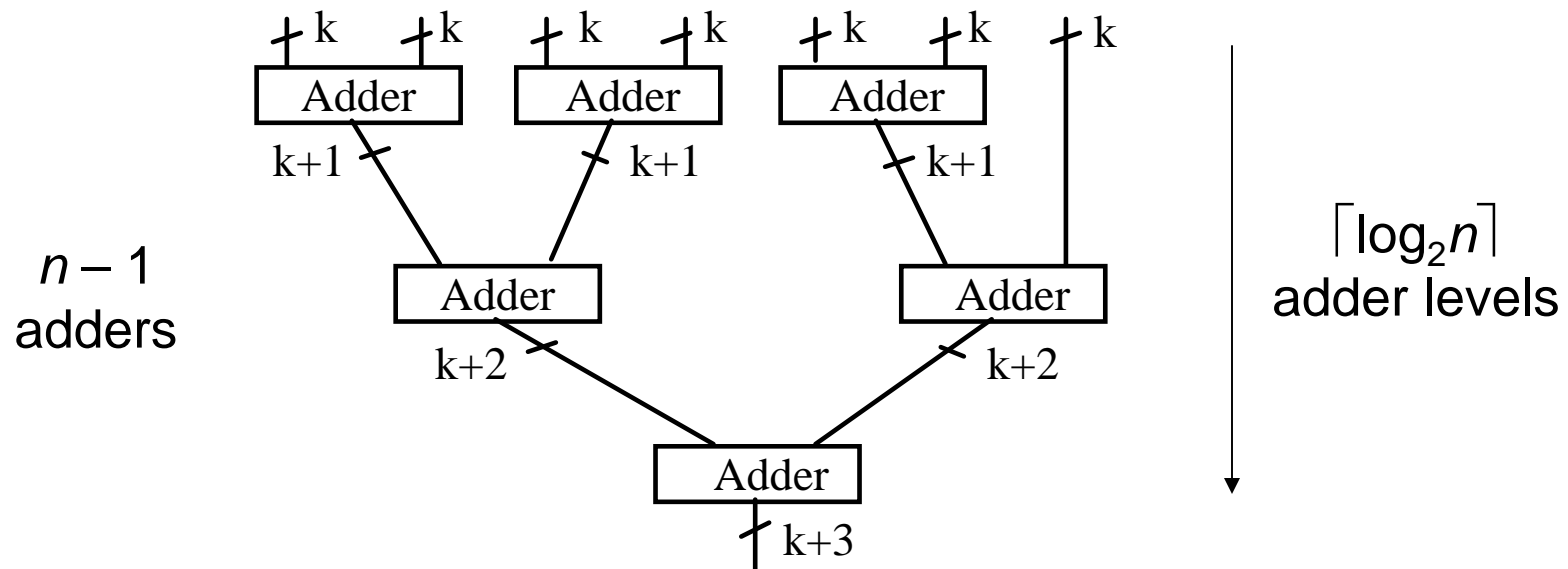


Fig. 8.1 Serial multi-operand addition when each adder is a 4-stage pipeline.

Parallel Implementation as Tree of Adders

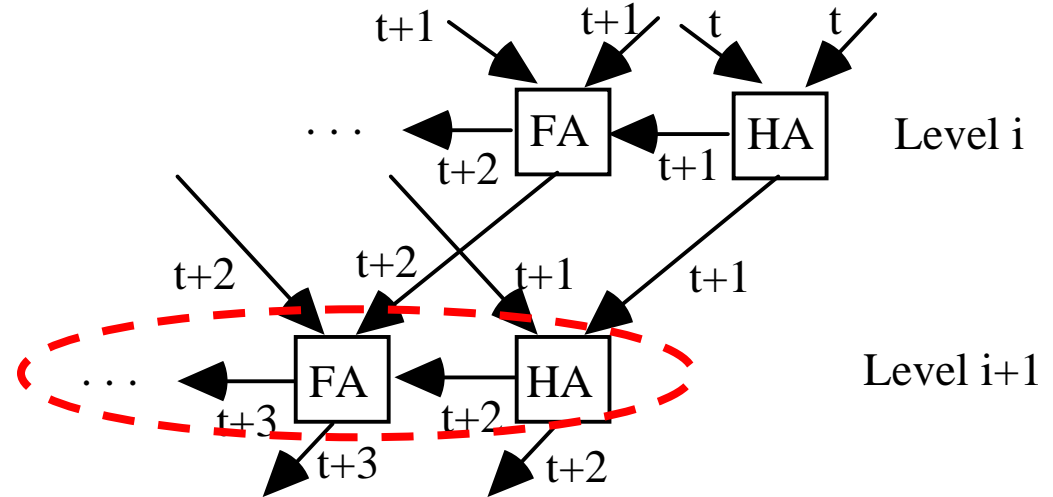
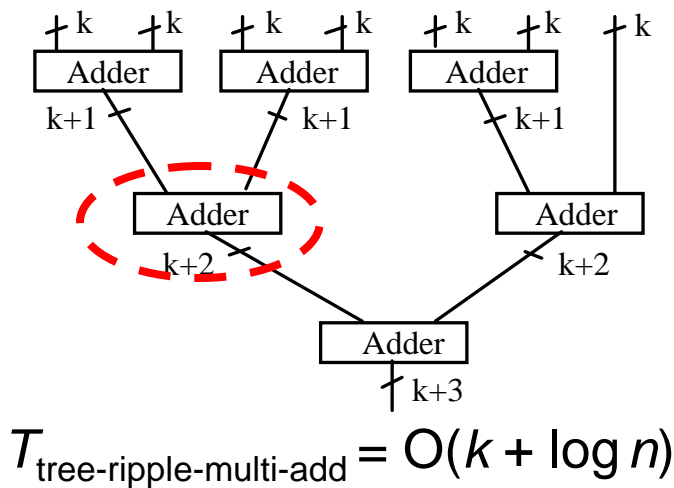


Adding 7 numbers in a binary tree of adders.

$$\begin{aligned}
 T_{\text{tree-fast-multi-add}} &= O(\log k + \log(k + 1) + \dots + \log(k + \lceil \log_2 n \rceil - 1)) \\
 &= O(\log n \log k + \log n \log \log n)
 \end{aligned}$$

$$T_{\text{tree-ripple-multi-add}} = O(k + \log n) \quad \text{[Justified on the next slide]}$$

Elaboration on Tree of Ripple-Carry Adders



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

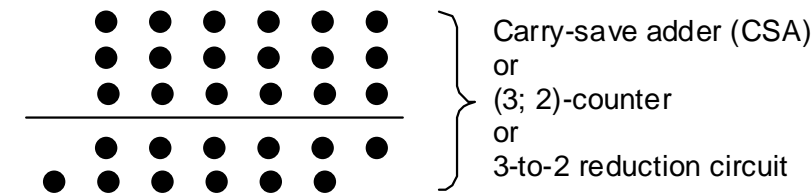
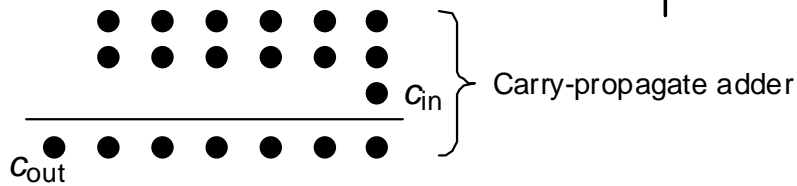
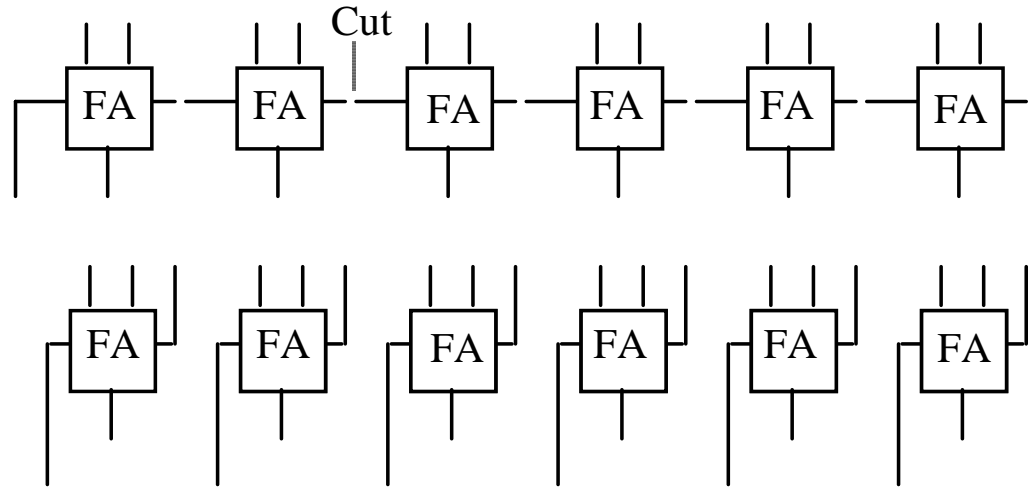
The absolute best latency that we can hope for is $O(\log k + \log n)$

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

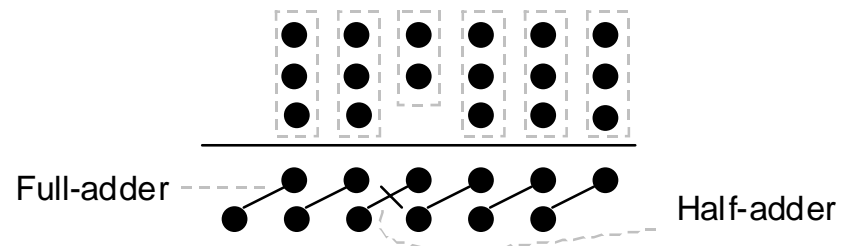
We will see shortly that carry-save adders achieve this optimum time

2 Carry-Save Adders

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



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



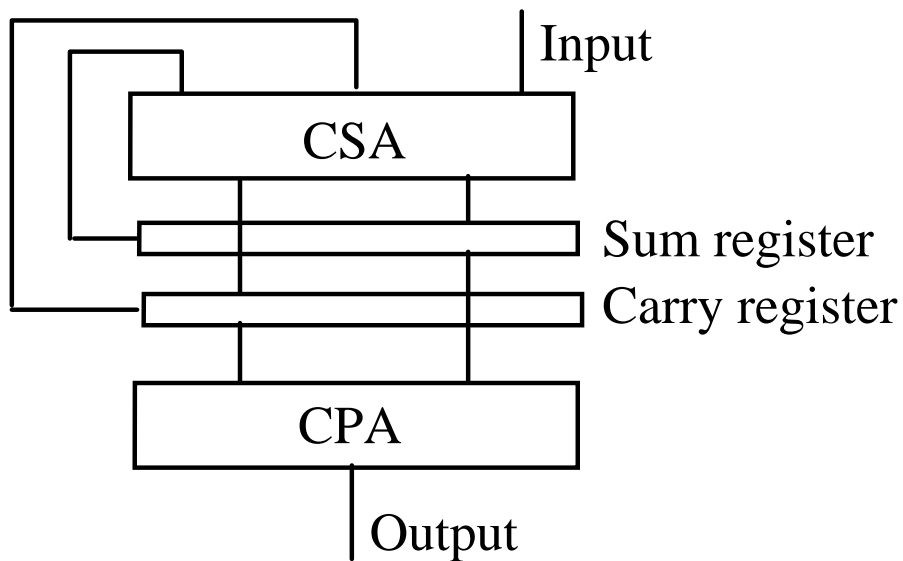
Specifying full- and half-adder blocks, with their inputs and outputs, in dot notation.

Multioperand Addition Using Carry-Save Adders

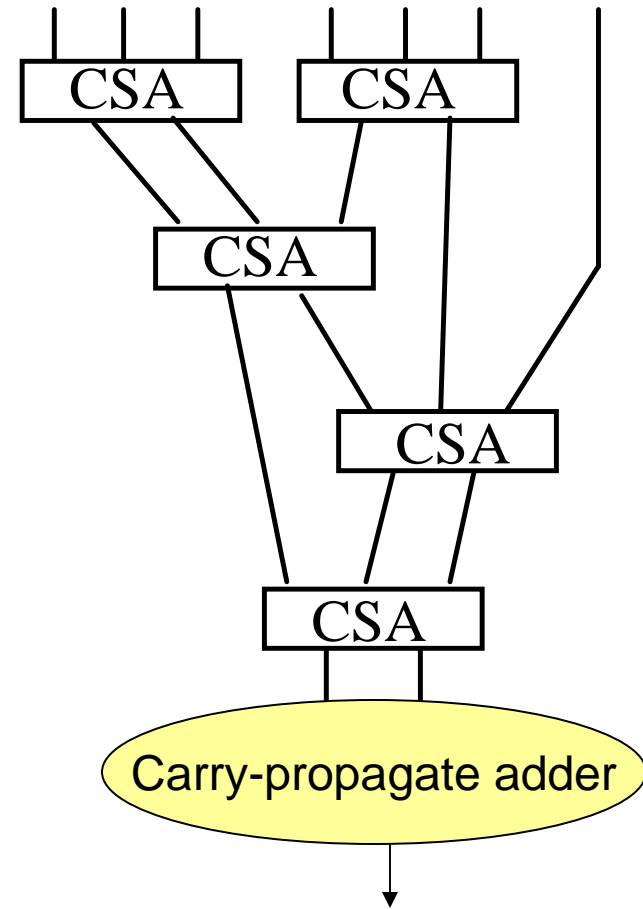
$$T_{\text{carry-save-multi-add}} = O(\text{tree height} + T_{\text{CPA}})$$

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

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

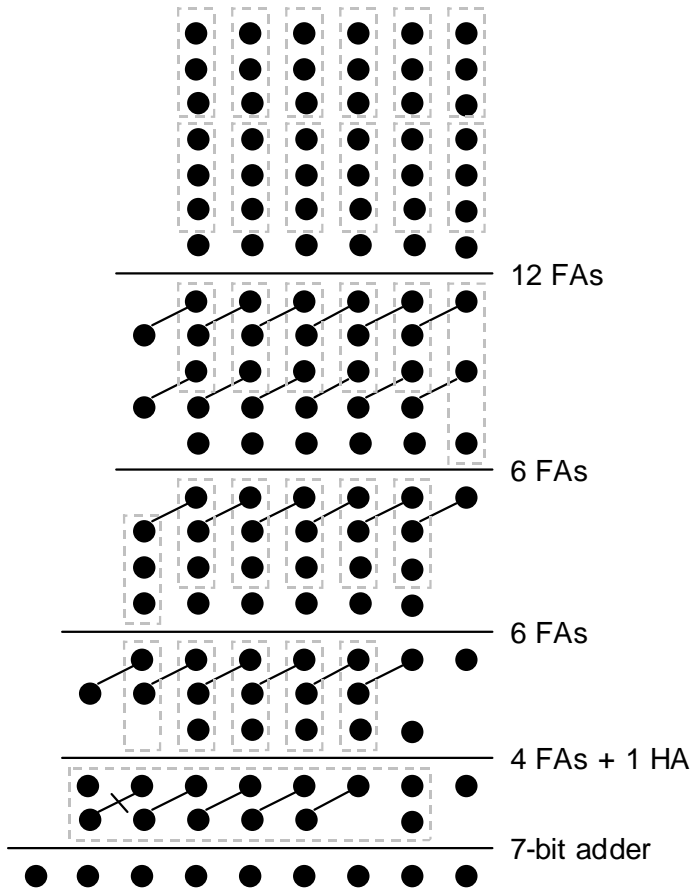


Serial carry-save addition using a single CSA.



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

Addition of seven 6-bit numbers in dot notation.

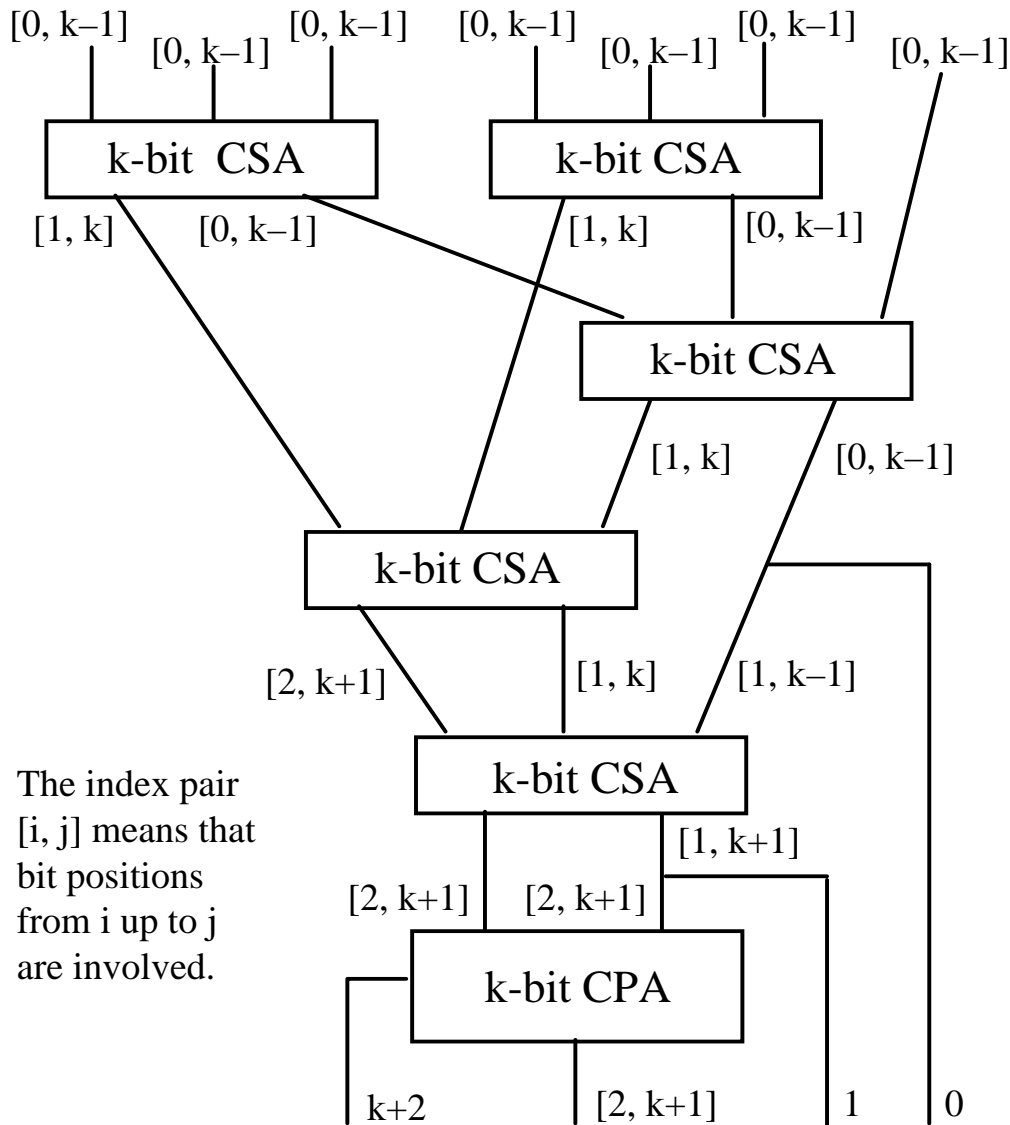
8	7	6	5	4	3	2	1	0	Bit position
			7	7	7	7	7	7	$6 \times 2 = 12$ FAs
		2	5	5	5	5	5	3	6 FAs
		3	4	4	4	4	4	1	6 FAs
	1	2	3	3	3	3	2	1	4 FAs + 1 HA
	2	2	2	2	2	1	2	1	7-bit adder
--Carry-propagate adder--									
1	1	1	1	1	1	1	1	1	

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

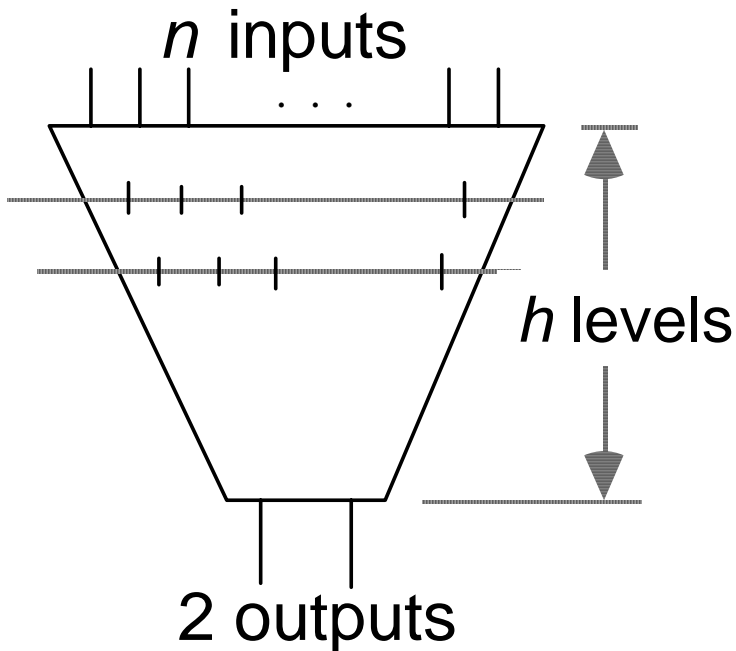


The index pair $[i, j]$ means that bit positions from i up to j are involved.

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

3 Wallace and Dadda Trees



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

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

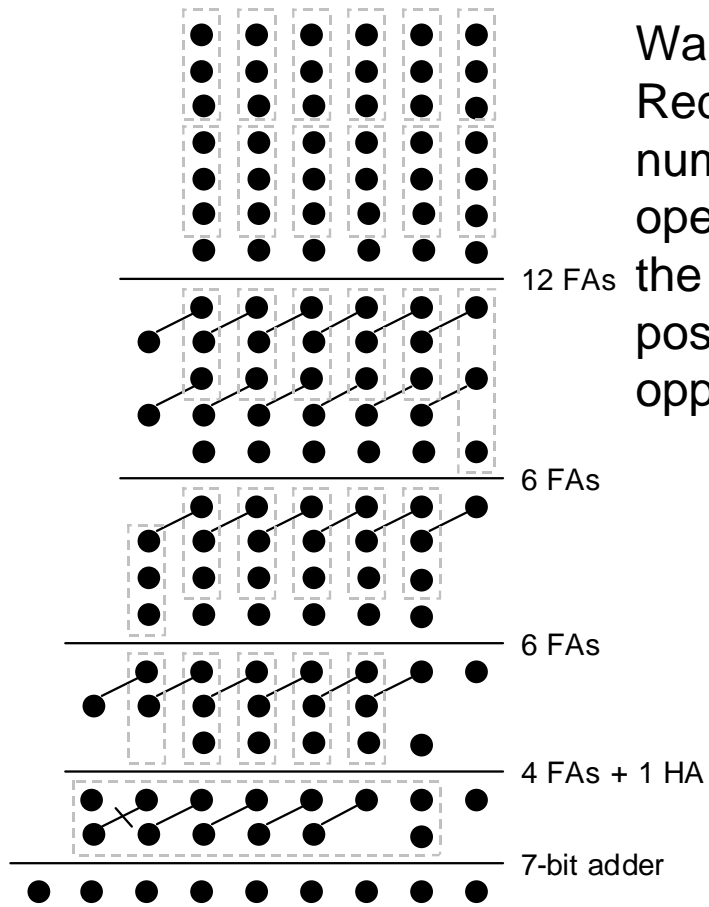
$$2 \times 1.5^{h-1} < n(h) \leq 2 \times 1.5^h$$

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

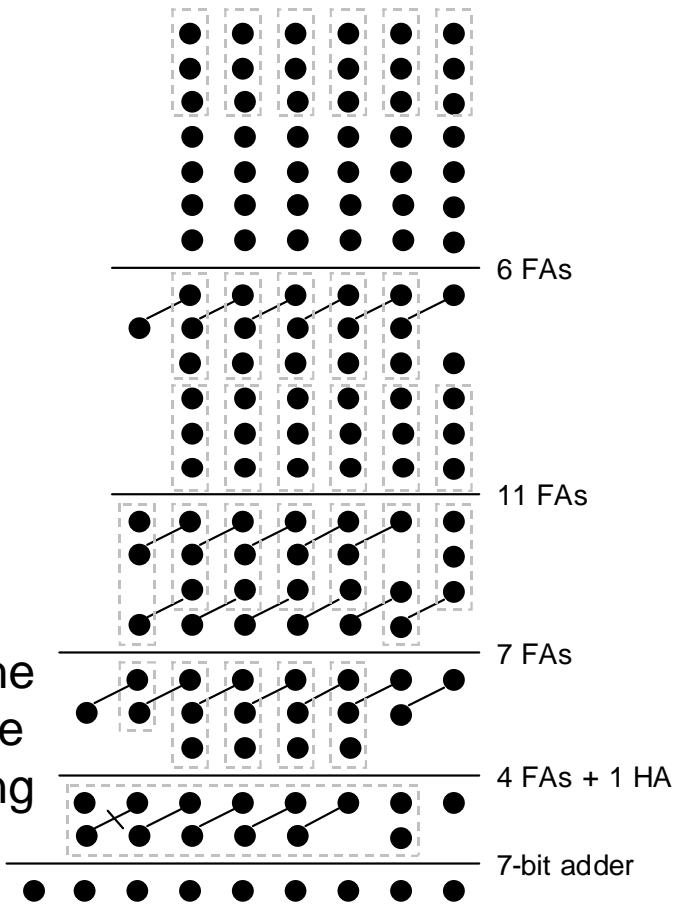


Total cost = 7-bit adder + 28 FAs + 1 HA

Addition of seven 6-bit numbers in dot notation.

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

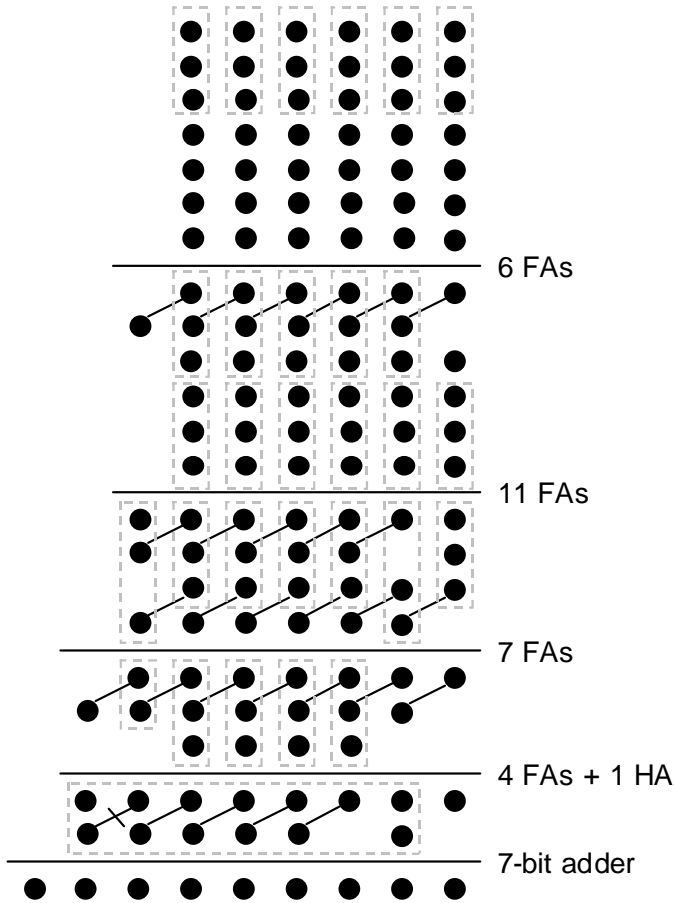
Dadda tree:
Postpone the reduction to the extent possible without causing added delay



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

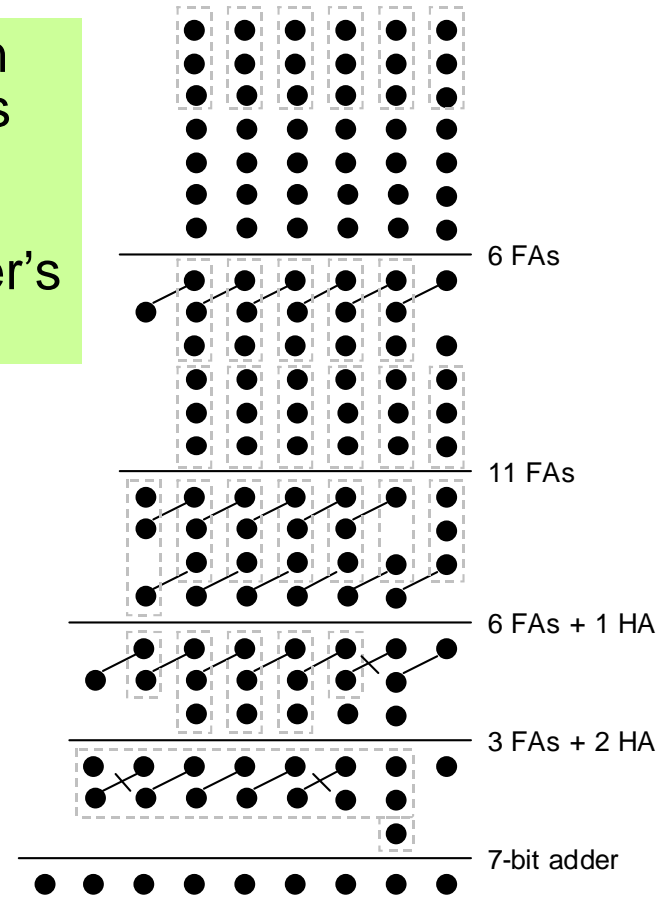
Adding seven 6-bit numbers using Dadda's strategy.

A Small Optimization in Reduction Trees



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

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

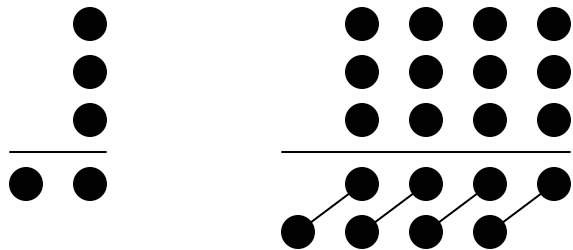


Total cost = 7-bit adder + 26 FAs + 3 HA

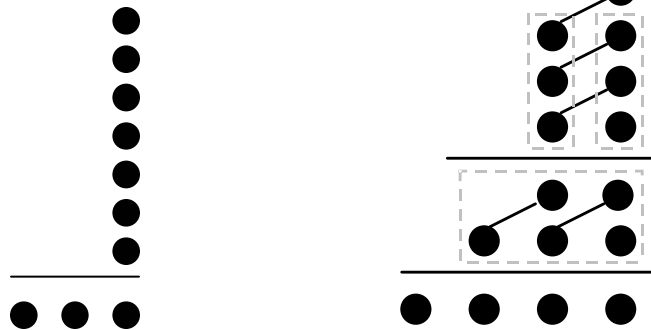
Adding seven 6-bit numbers using Dadda's strategy.

4 Parallel Counters

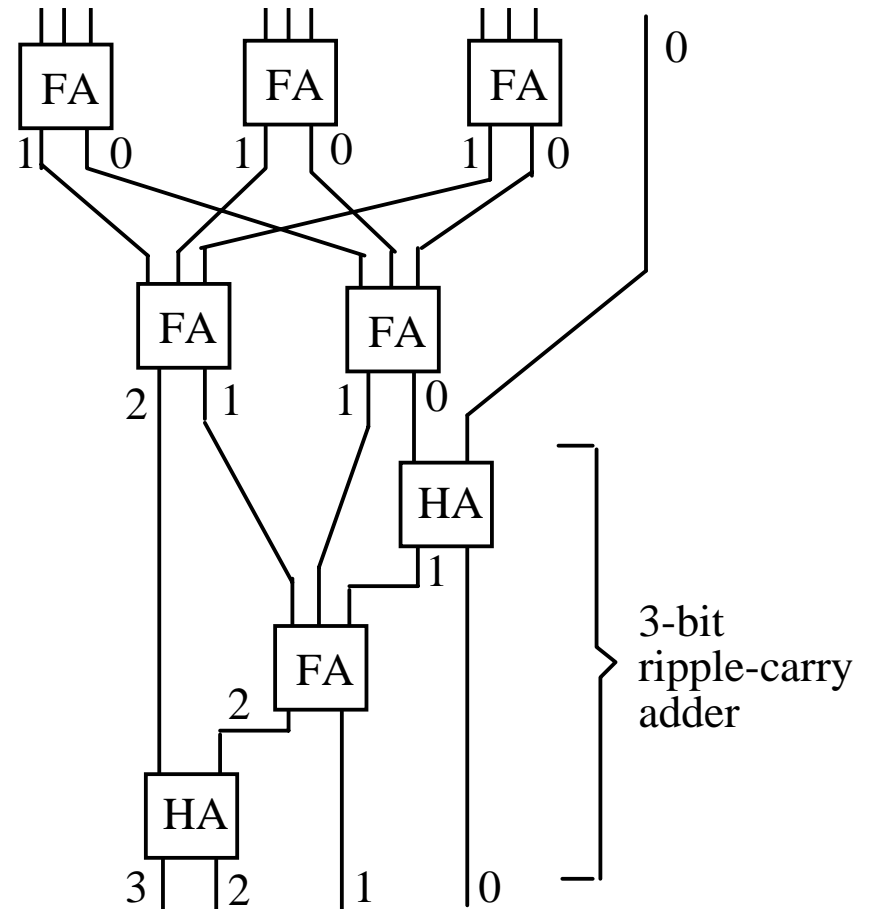
1-bit full-adder = (3; 2)-counter



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

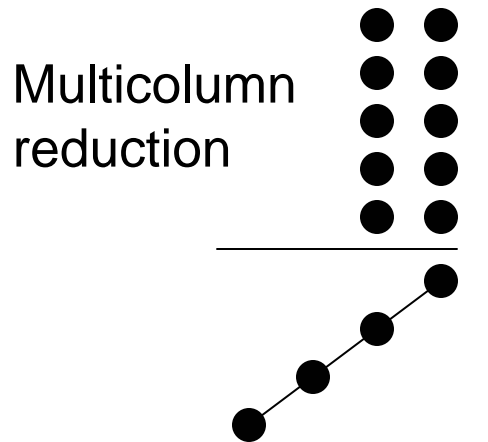


Circuit reducing n bits to their $\lceil \log_2(n+1) \rceil$ -bit sum = $(n; \lceil \log_2(n+1) \rceil)$ -counter

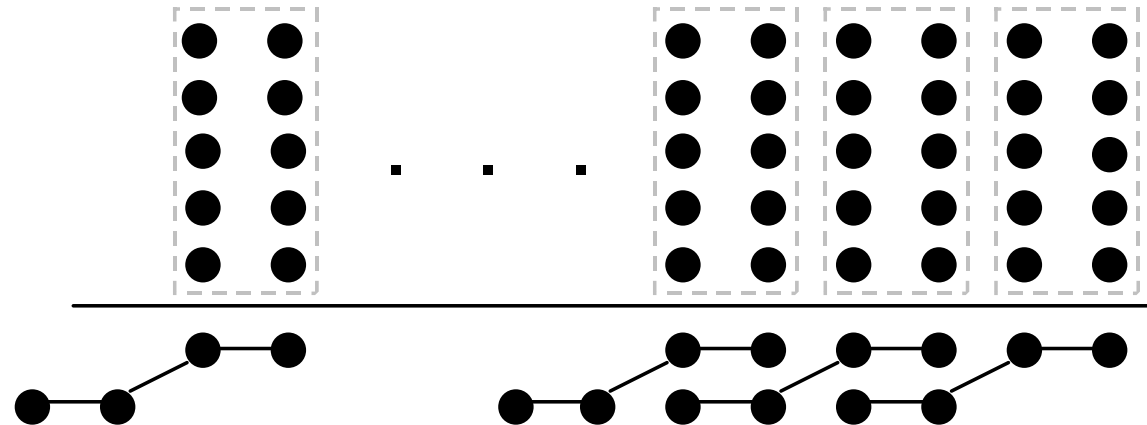


A 10-input parallel counter also known as a (10; 4)-counter.

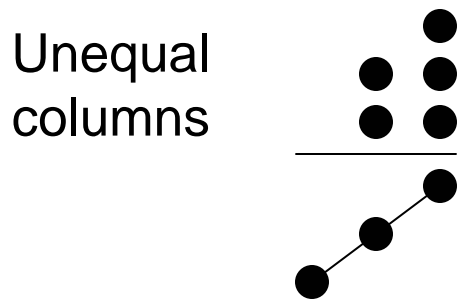
5 Generalized Parallel Counters



(5, 5; 4)-counter



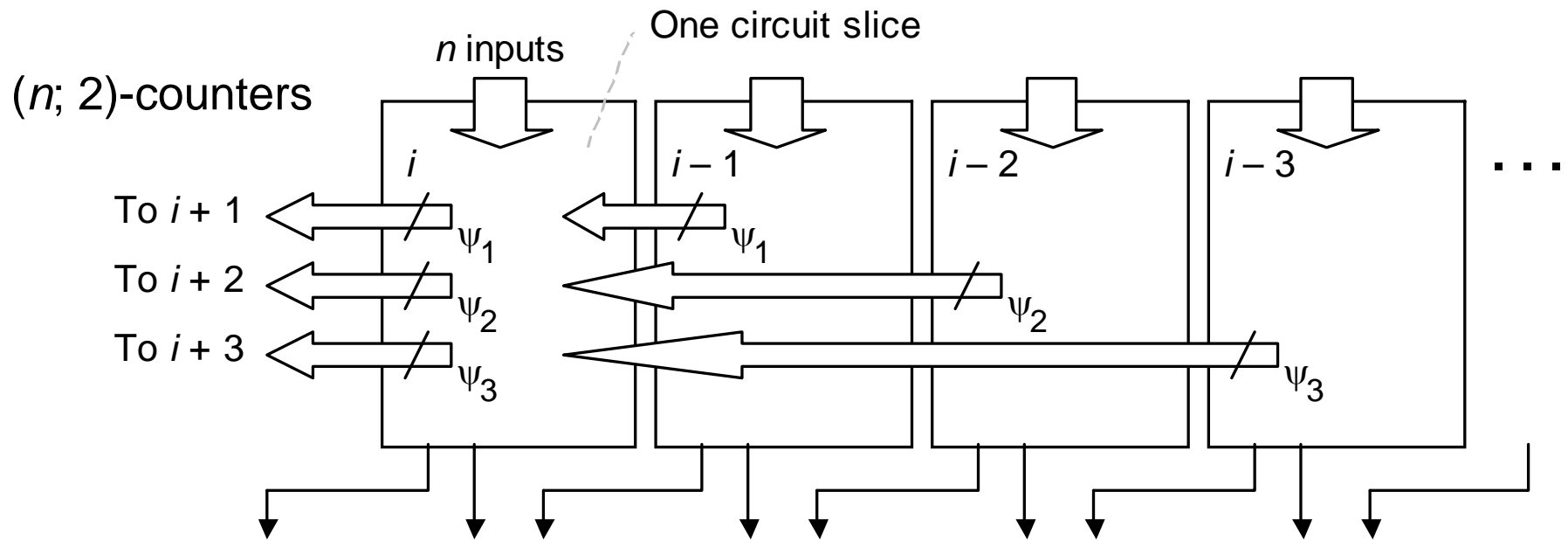
Dot notation for a (5, 5; 4)-counter and the use of such counters for reducing five numbers to two numbers.



(2, 3; 3)-counter

Gen. parallel counter = Parallel compressor

A General Strategy for Column Compression



$$n + \psi_1 + \psi_2 + \psi_3 + \dots \leq 3 + 2\psi_1 + 4\psi_2 + 8\psi_3 + \dots$$

$$n - 3 \leq \psi_1 + 3\psi_2 + 7\psi_3 + \dots$$

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

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

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

6 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					

$$-b = (1 - b) + 1 - 2$$

(b) Using negatively weighted bits

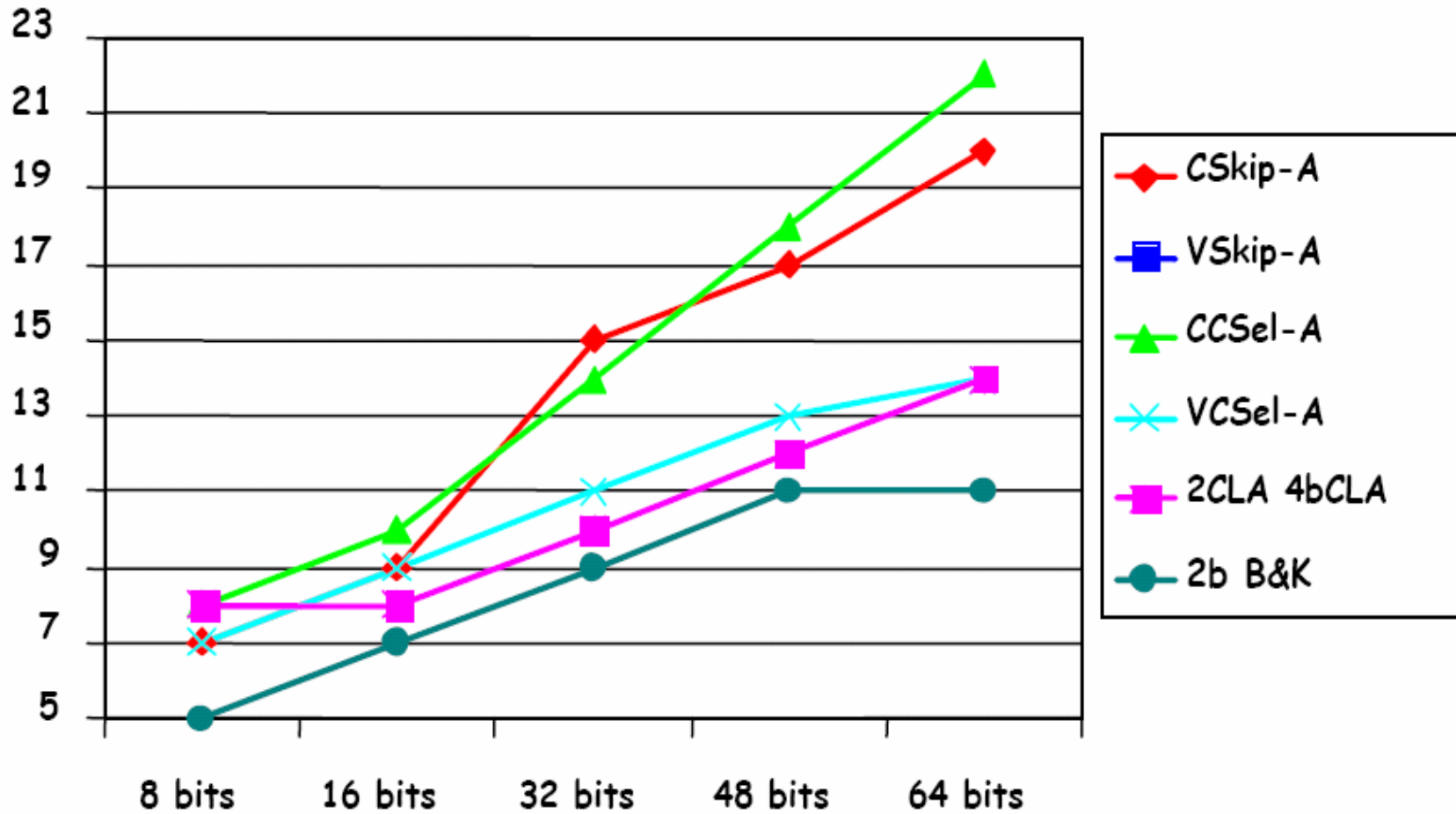
Adding three 2's-complement numbers.

Comparisons

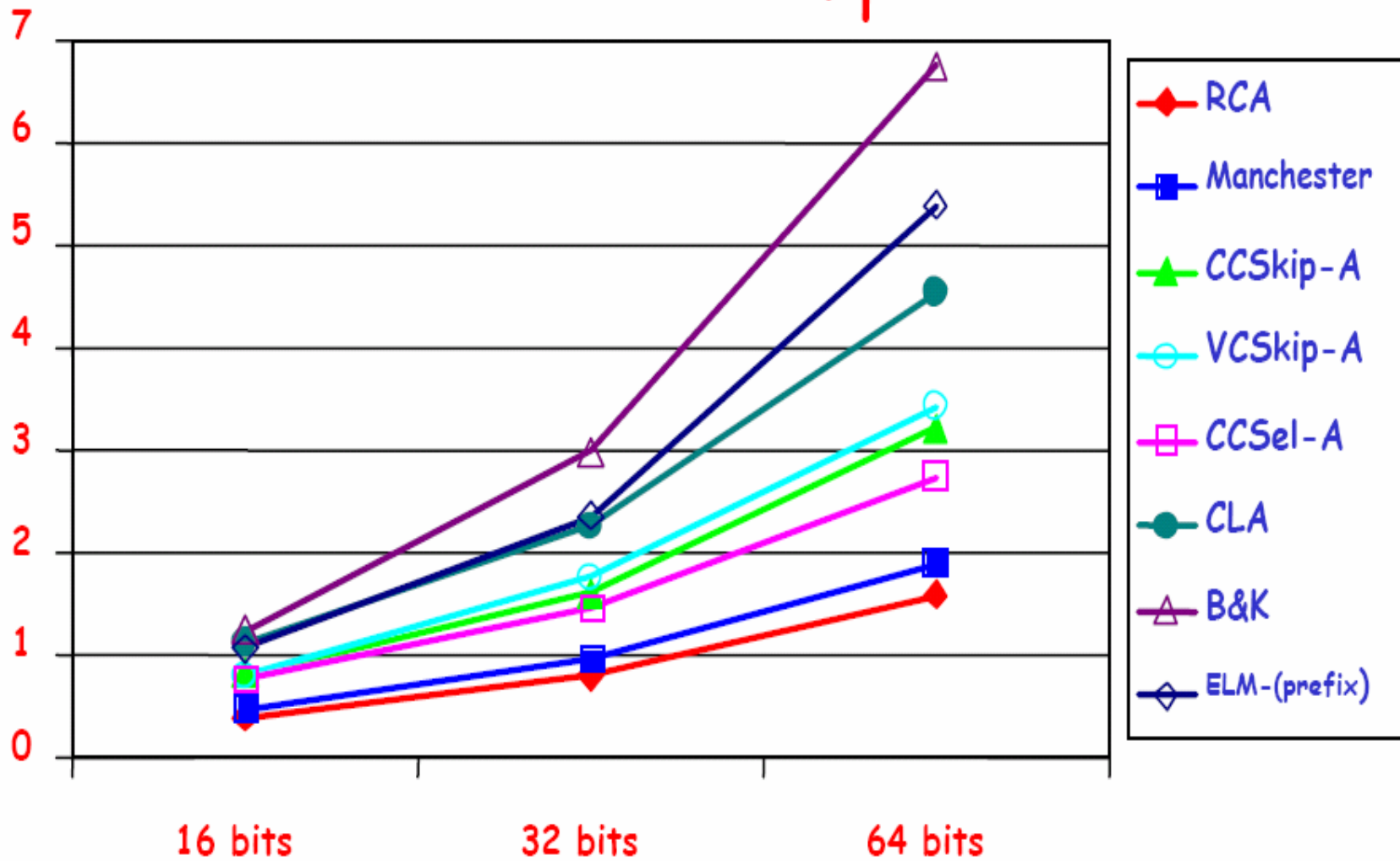
Adders:

- Ripple carry adder
- Carry Look ahead adder
- Carry Select adder
- Carry Skip adder
- Carry Save adder
- Manchester Carry Chains
- Variable Skip adder
- Brent-Kung adder
- Kogge-Stone adder
- Sklansky adder
- ELM adder and many more...

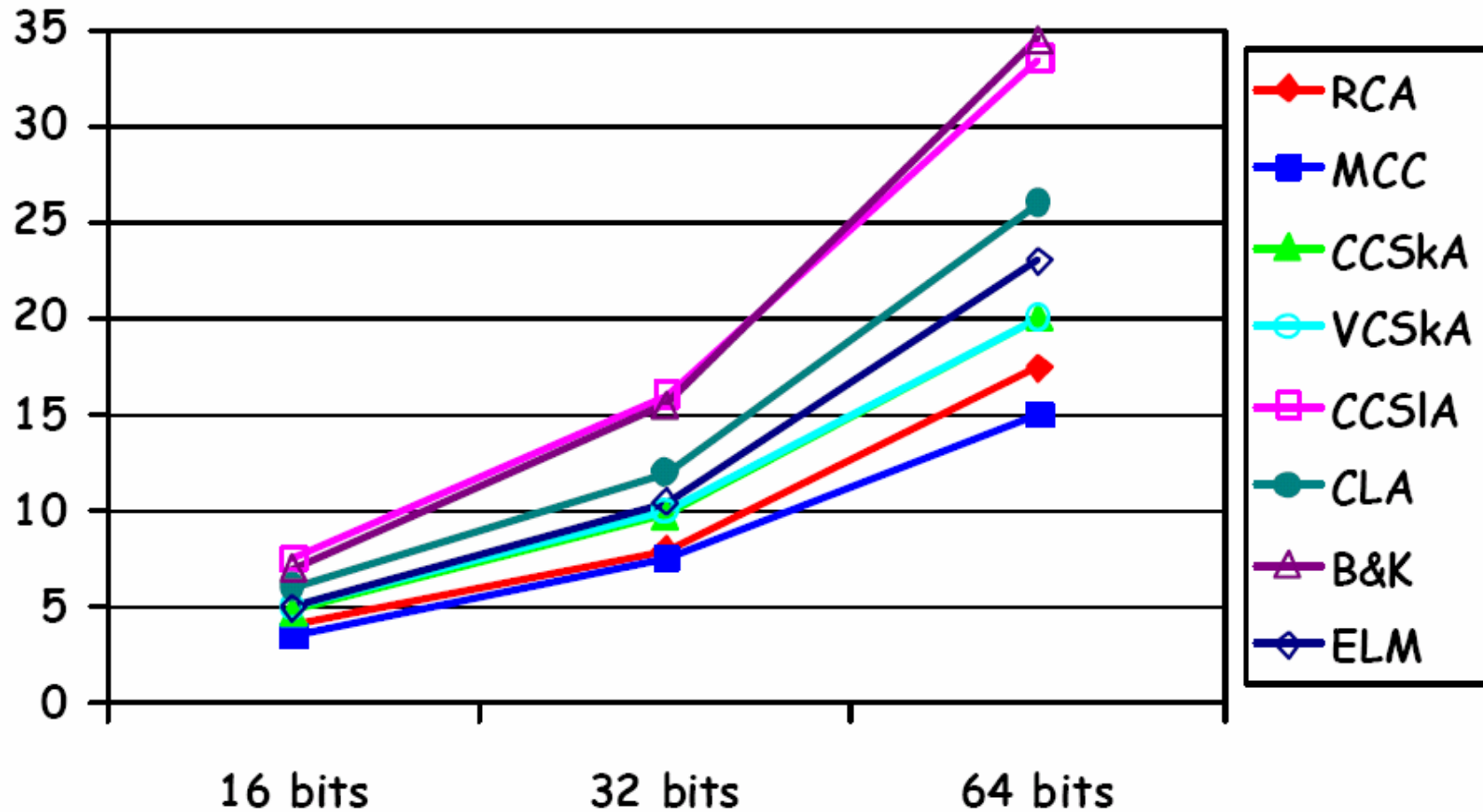
Adder Delay Comparisons



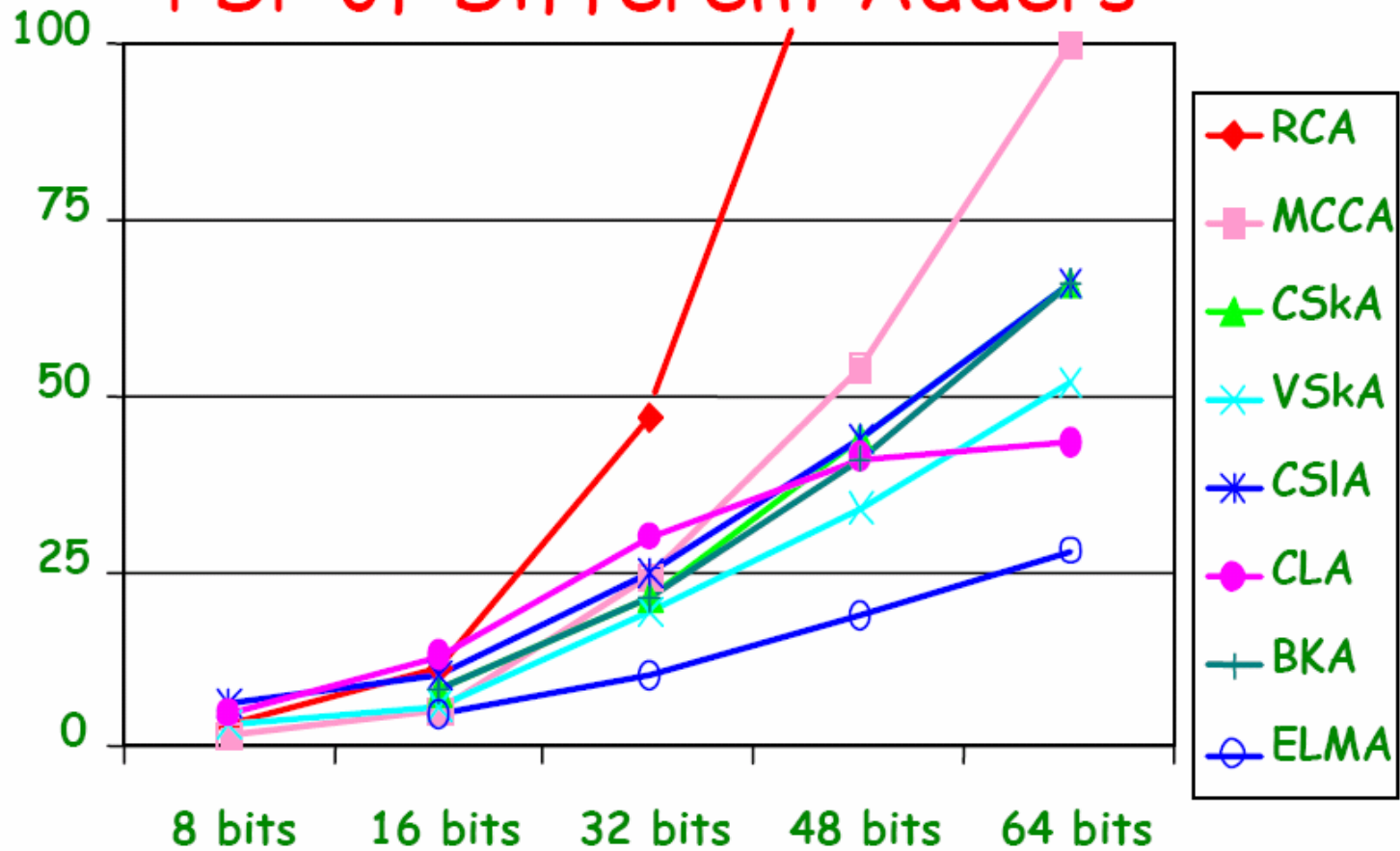
Adder Area Comparisons



Adder Average Power Comparisons



PDP of Different Adders



PDP: power delay product

How to do fast Arithmetic?

