Addition Circuits

Shmuel Wimer
Bar Ilan University, Engineering Faculty
Technion, EE Faculty

Full Adders

Table 10.2 Truth table for full adder

A	B	C	G	P	K	C_{out}	S
0	0	0	0	0	1	0	0
		1				0	1
0	1	0	0	1	0	0	1
		1				1	0
1	0	0	0	1	0	0	1
		1				1	0
1	1	0	1	0	0	1	0
		1				1	1

$$S = A \oplus B \oplus C \qquad C_{\text{out}} = A \bullet B + (A \oplus B)C$$

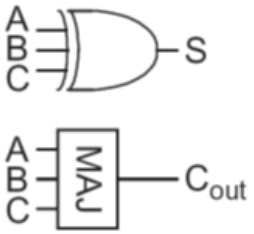
$G = A \bullet B$: C_{out} generation regardless of C_{in}

$P = A \oplus B$: C_{in} propagates to C_{out}

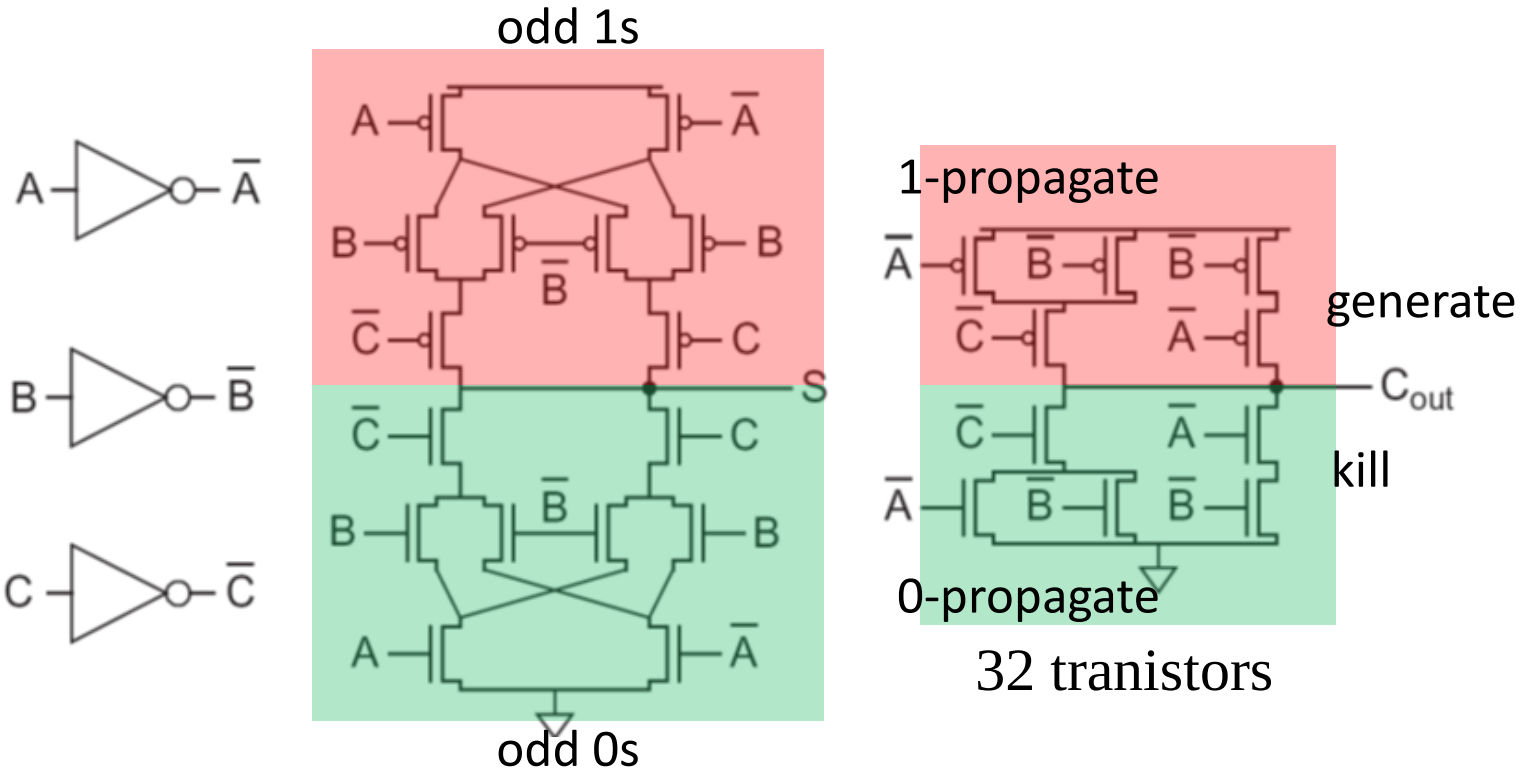
$K = \overline{A} \bullet \overline{B}$: C_{in} is killed

$$S = A \oplus B \oplus C = P \oplus C$$

$$C_{out} = AB + AC + BC = \mathrm{MAJ}(A, B, C)$$



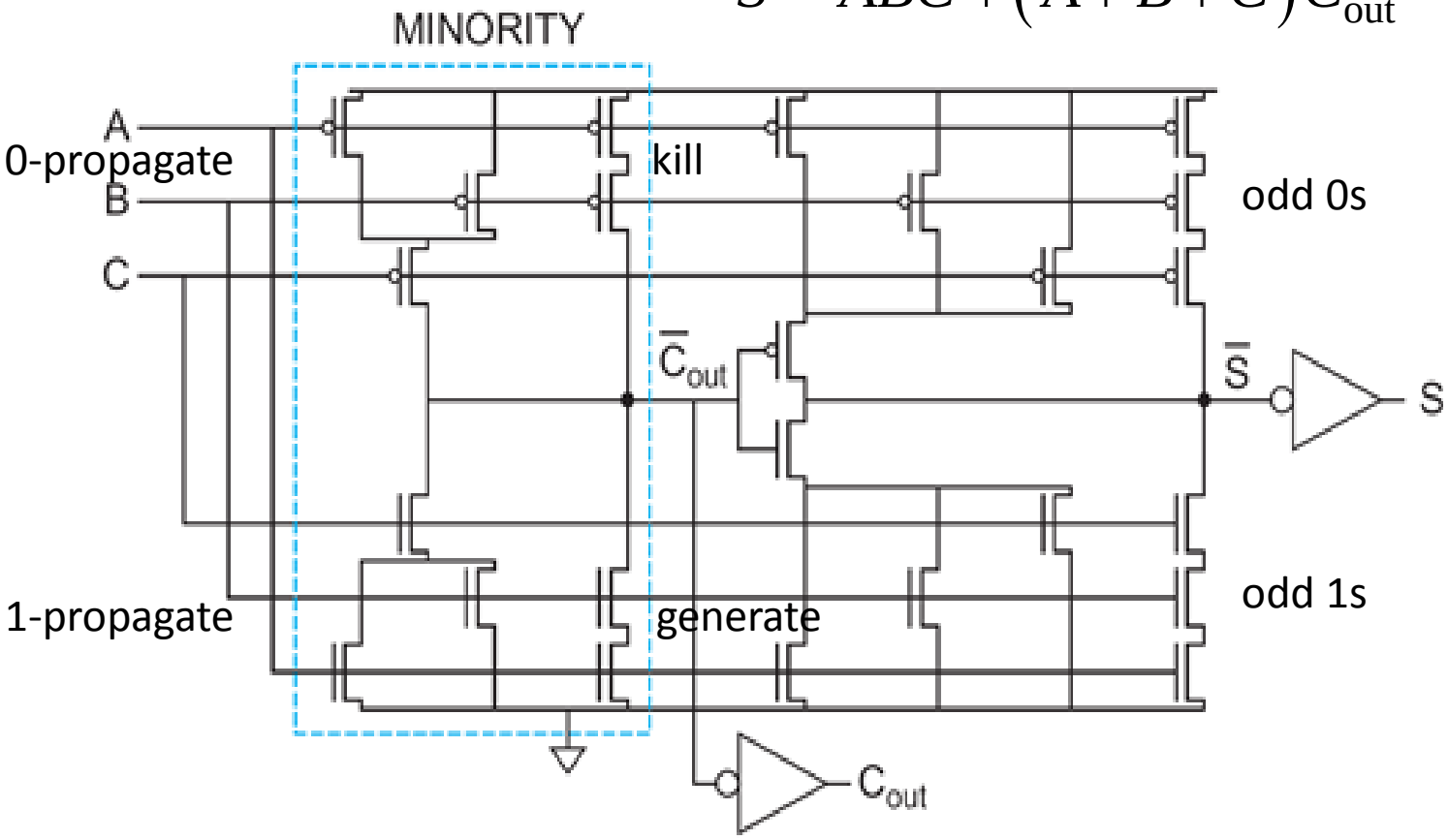
Design I: Mirror CMOS logic



N and P networks are identical rather than complementary!

Design II: *S* is factored to reuse *C*out

$$S = ABC + (A + B + C)\overline{C}_{out}$$

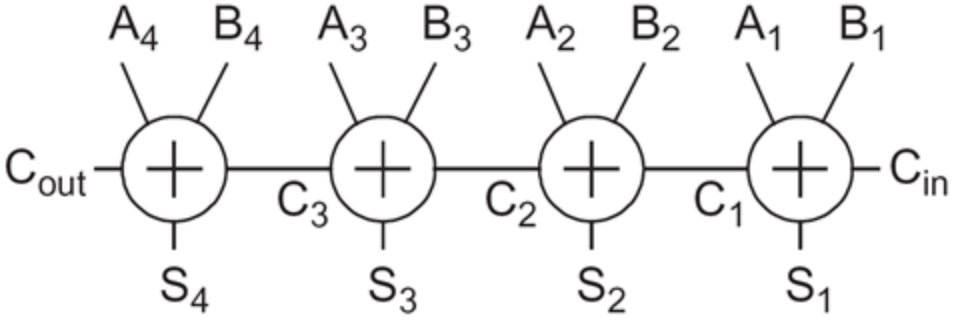


Uses only 28 transistors. Can be reduced to 24 transistors.

S has larger delay but it is not on the critical path

The transistors connected to C_{in} are closest to the output of the carry (and sum) circuits. (why?)

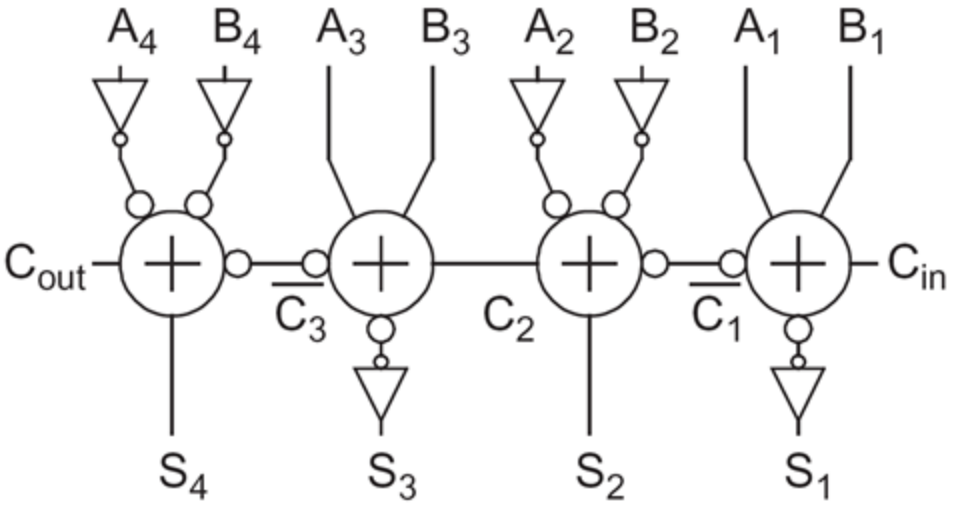
Only the transistors of the carry are optimized for speed. (why?)



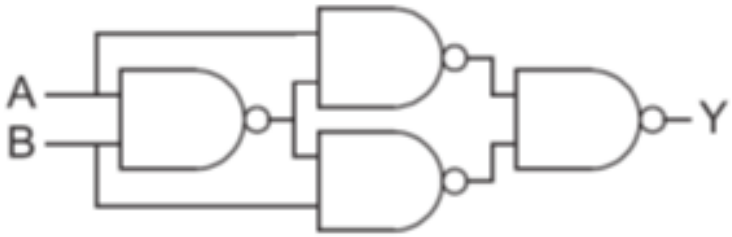
Ripple-Carry Addition

Carry computation is the critical path

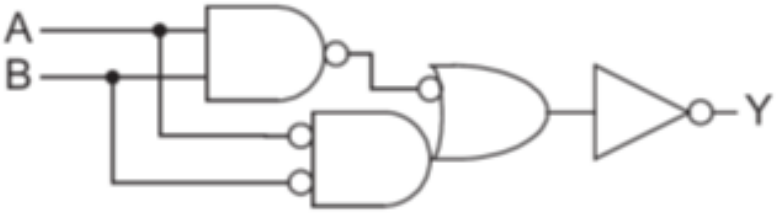
Carry propagation delay is reduced by using inverting adders where every other stage is working on complementary data.



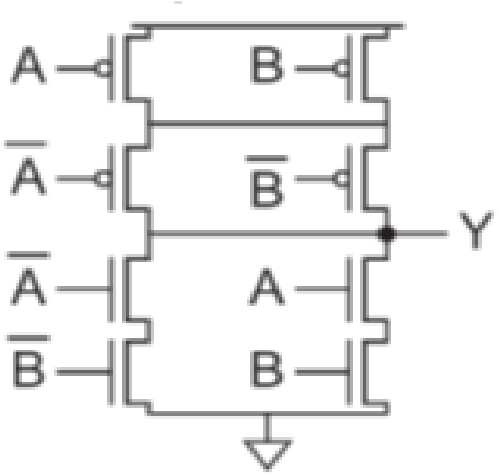
XOR / XNOR Circuits



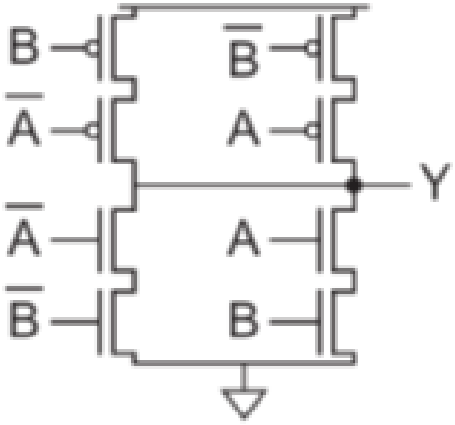
Straight-forward,
16 transistors



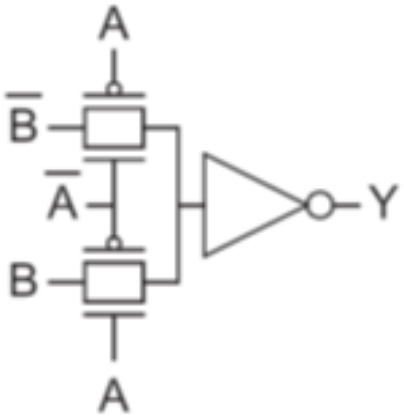
14 transistors



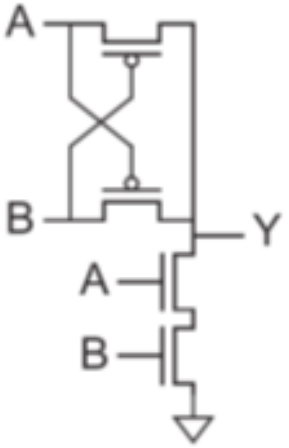
Complementary CMOS,
12 transistors



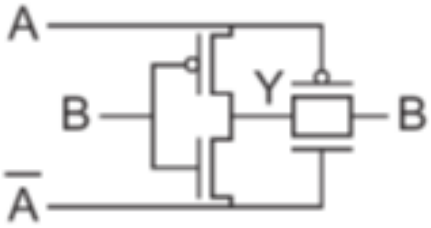
More efficient, less contacts, smaller layout, commonly used in STD cell Lib.



Transmission gate design, 10 transistor



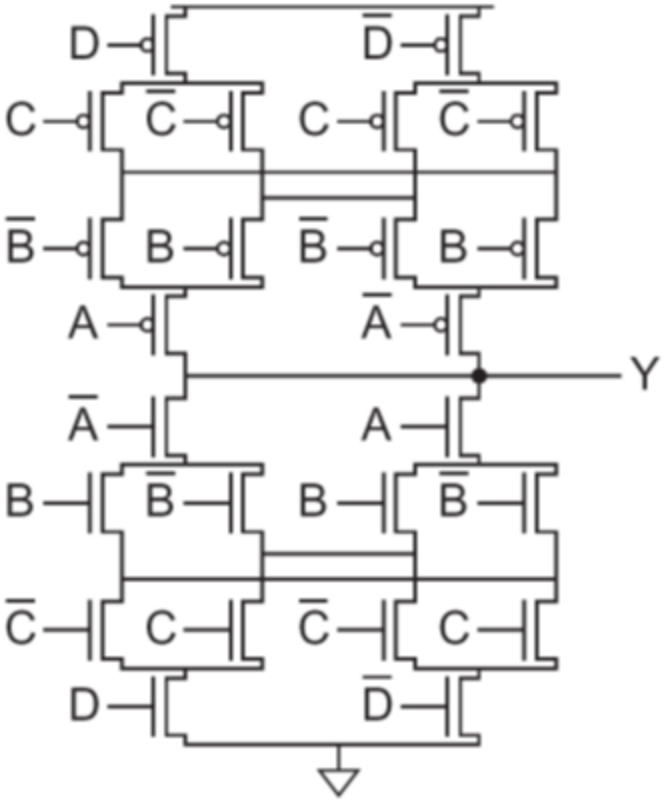
Only 4 transistors, fast, but doesn't swing rail-to-rail.



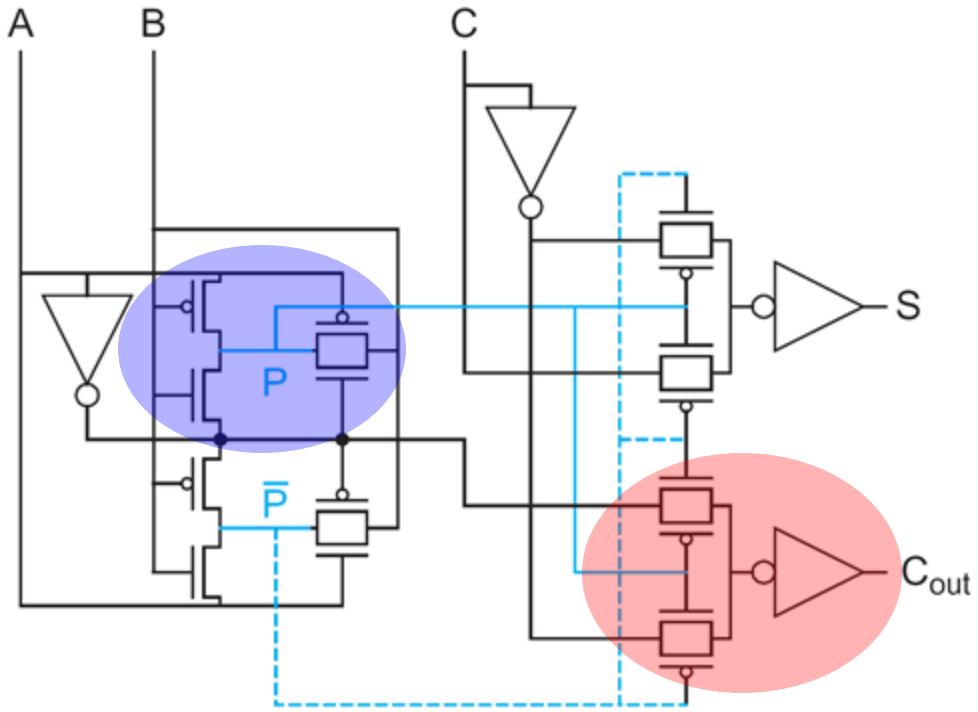
Only 6 transistors, but non restoring.

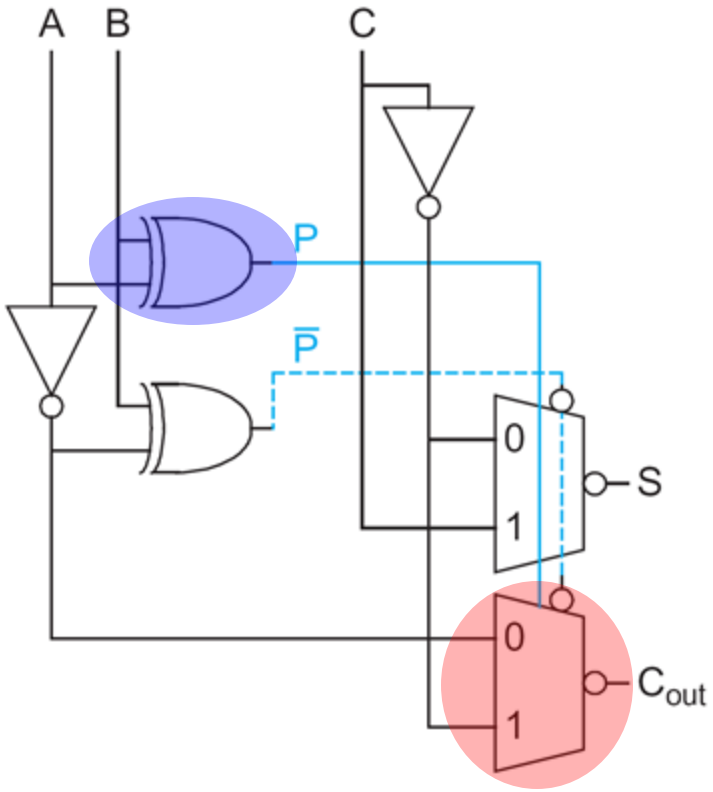
$$A = 0 \Rightarrow Y = B, \quad A = 1 \Rightarrow Y = \bar{B}$$

4-way



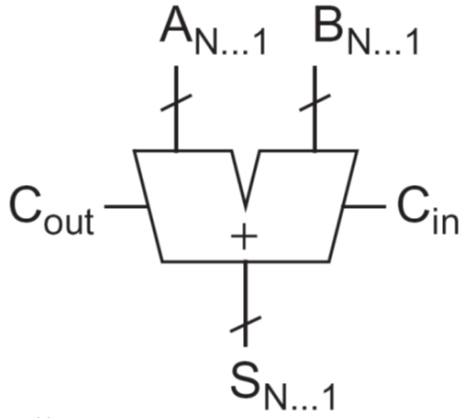
Full-adder using XOR and MUX $S = A \oplus B \oplus C = P \oplus C$

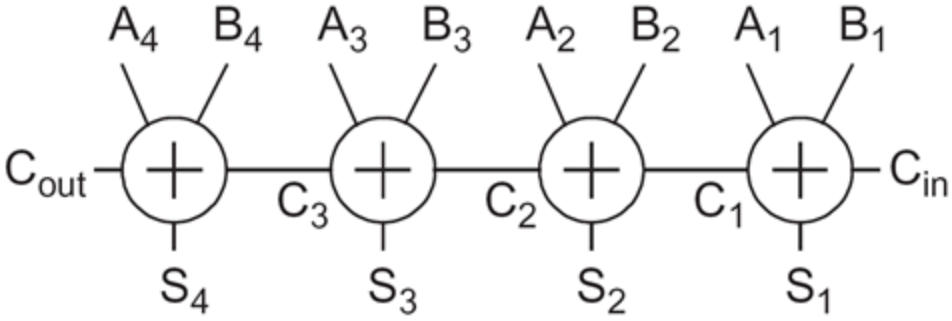




24 transistors and buffered outputs.
C_{out} and S have same delay.

Carry-Propagate Addition





Ripple-Carry Addition

Carry computation is the critical path in addition

Recall: $G = A \bullet B$: C_{out} generation regardless of C_{in}

$P = A \oplus B$: C_{in} propagates to C_{out}

Generate and Propagate signals are a key for fast addition

Generalize to describe whether the group of bits $i \geq k \geq j$ generates or propagates a carry.

$$G_{i:j} = G_{i:k} + P_{i:k} \bullet G_{k-1:j} \qquad P_{i:j} = P_{i:k} \bullet P_{k-1:j}$$

with the base case

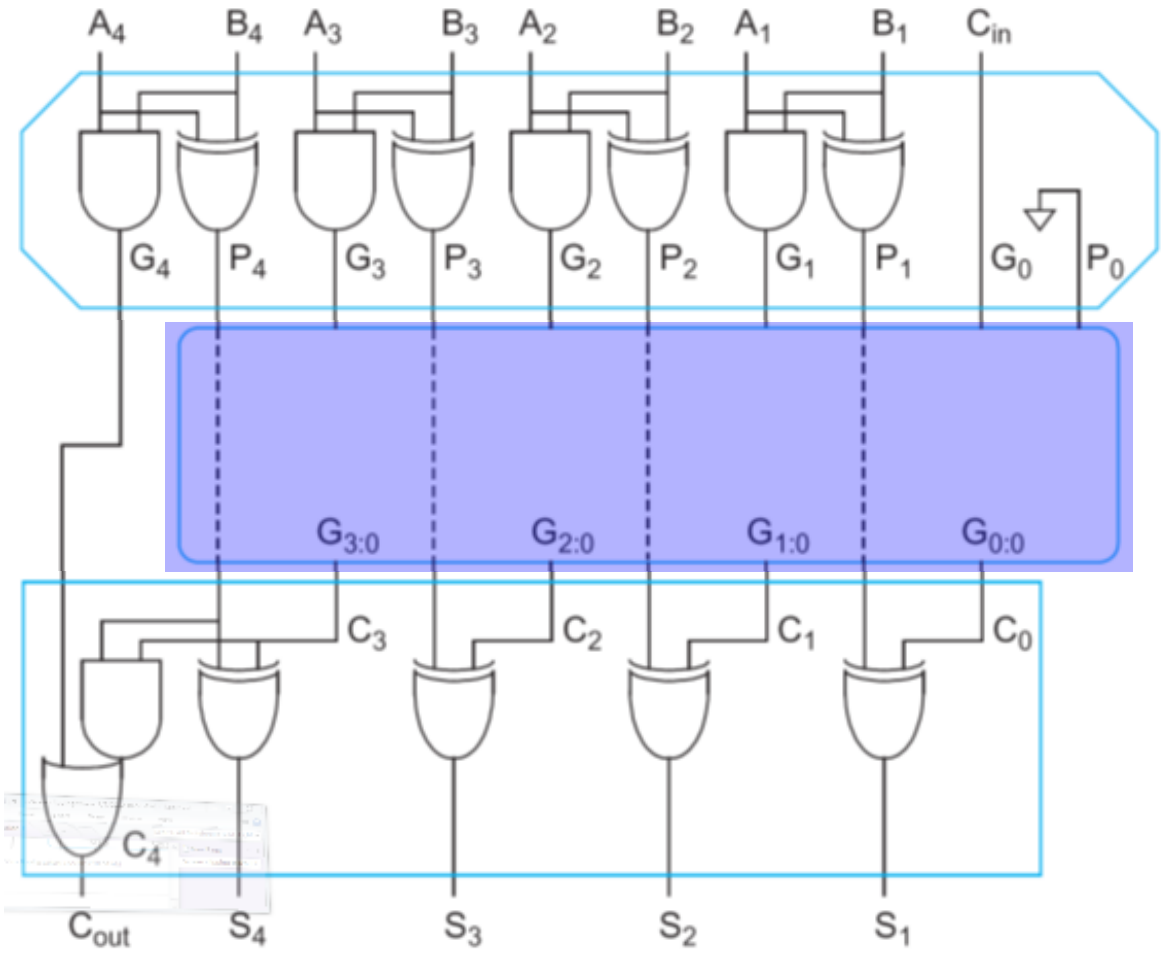
$$G_{i:i} \triangleq G_i = A_i \bullet B_i \qquad P_{i:i} \triangleq P_i = A_i \oplus B_i$$

Define $G_{0:0} = C_{\text{in}} \quad P_{0:0} = 0$

Recall: $S = P \oplus C$

The sum for bit *i* can be computed by: $S_i = P_i \oplus G_{i-1:0}$

Addition is reduced into 3-step computation process

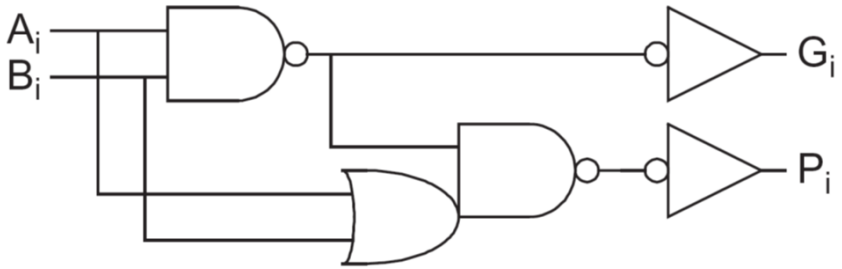


bitwise propagate and generate logic

group propagate and generate logic

sum logic

Addition acceleration is obtained by smart PG grouping



shared bitwise propagate-generate (PG) logic

$$G_{i:j} = G_{i:k} + P_{i:k} \bullet G_{k-1:j} \qquad P_{i:j} = P_{i:k} \bullet P_{k-1:j} \qquad i \geq k \geq j$$

A combined pair of smaller groups is called **valency-2** group PG logic.

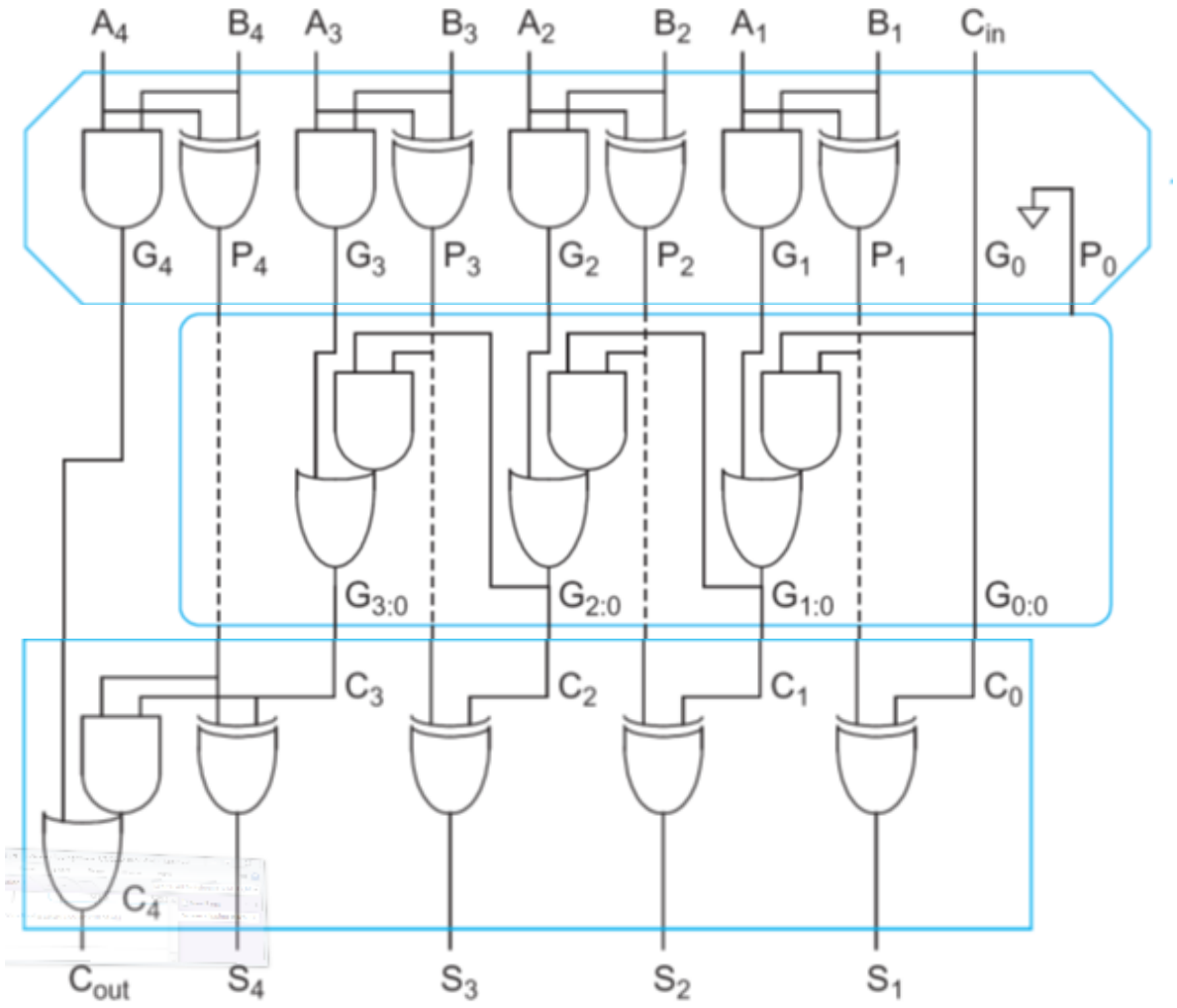
To use fewer stages for carry propagation, higher valency comprising more complex gates is possible, e.g. **valency-4**:

$$P_{i:j} = P_{i:k} \bullet P_{k-1:l} \bullet P_{l-1:m} \bullet P_{m-1:j} \qquad i \geq k > l > m > j$$

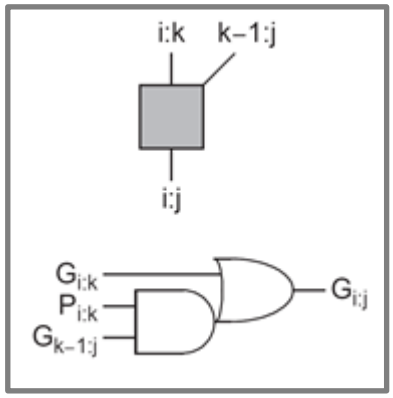
$$G_{i:j} = G_{i:k} + P_{i:k}\left(G_{k-1:l} + P_{k-1:l}\left(G_{l-1:m} + P_{l-1:m} \bullet G_{m-1:j}\right)\right)$$

PG Carry-Ripple Addition

$$C_i = A_i B_i + (A_i + B_i) C_{i-1} = A_i B_i + (A_i \oplus B_i) C_{i-1} = G_i + P_i C_{i-1}$$



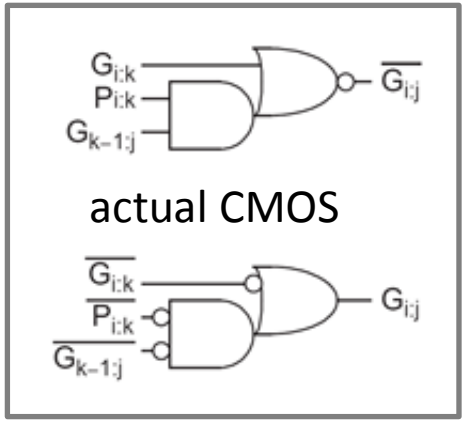
Adder architecture diagram

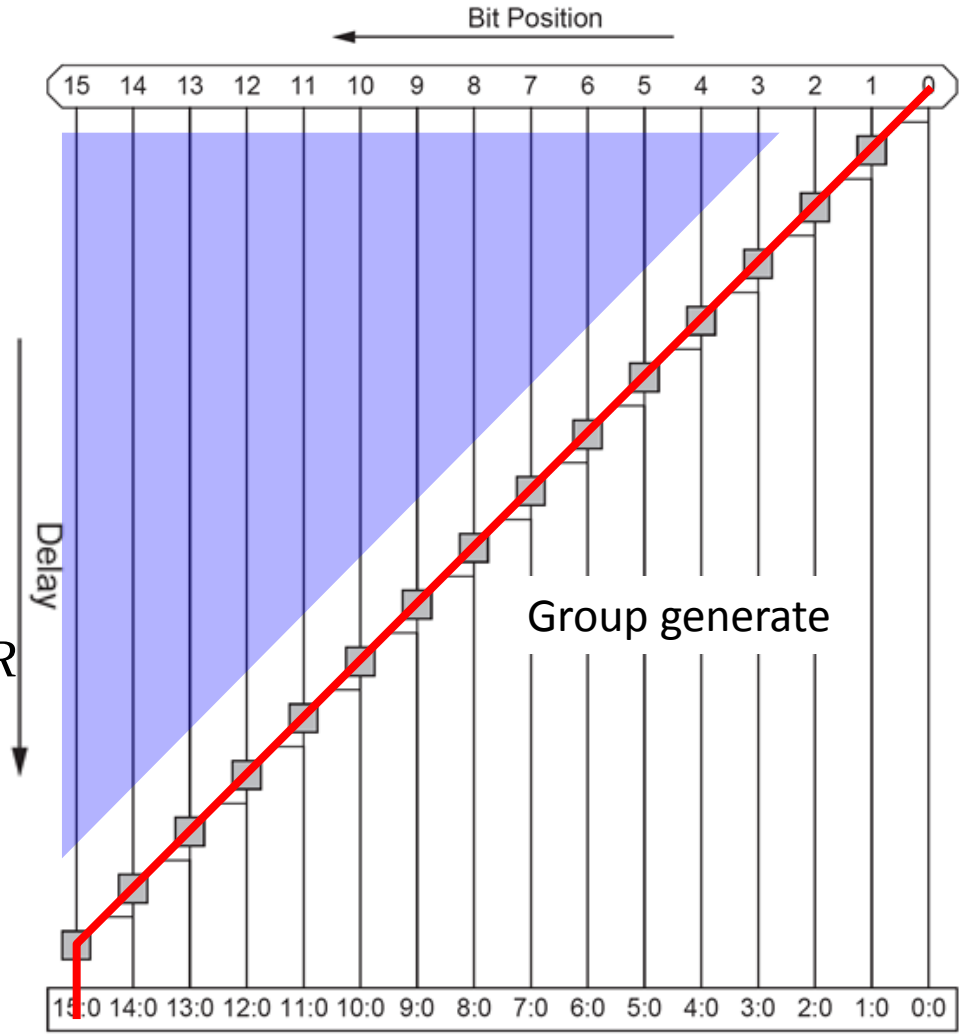


PG logic

$$C_{\text{out}} \equiv G_{N:0} = G_N + P_N G_{N-1:0}$$

$$t_{\text{ripple}} = t_{pg} + (N-1)t_{AO} + t_{XOR}$$

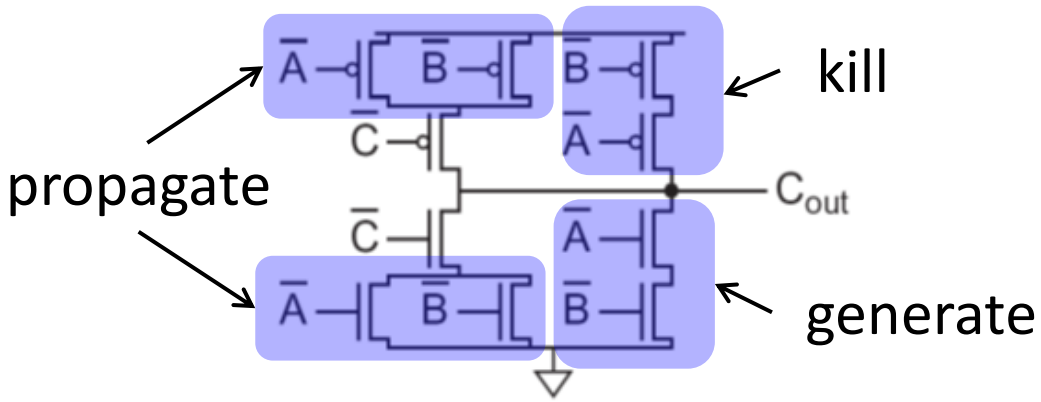




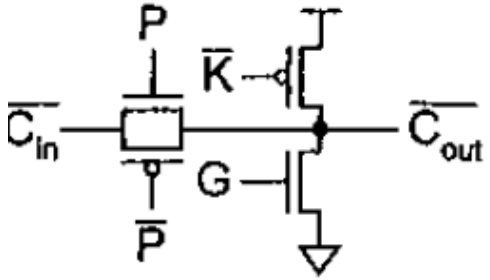
sum XOR

Carry Chain Adder

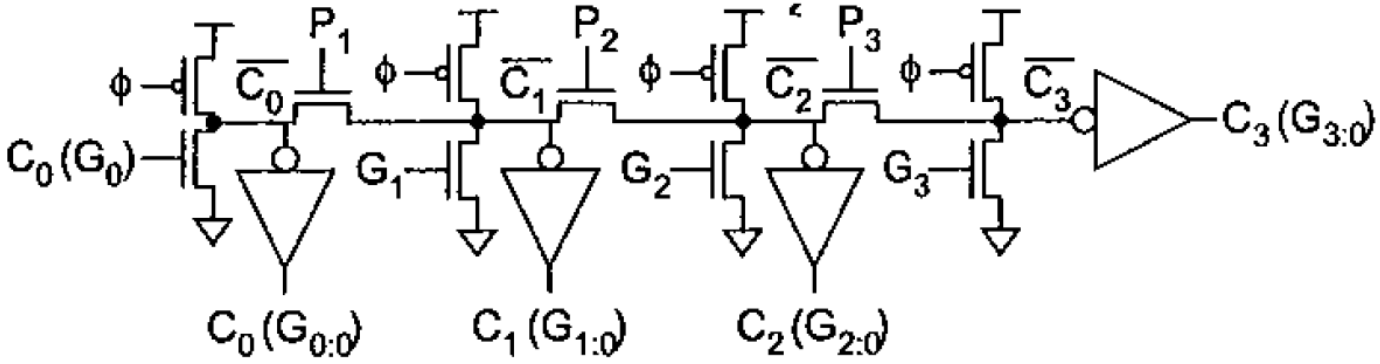
majority (carry) CMOS logic



pass transistor



Manchester valeny-4 carry chain adder (dynamic logic)

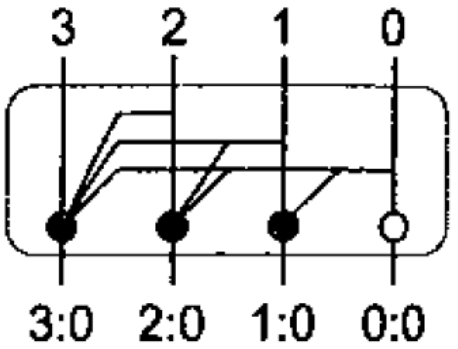


$$C_0 \equiv G_{0:0}$$

$$C_1 \equiv G_{1:0} = G_1 + P_1 C_0$$

$$C_2 \equiv G_{2:0} = G_2 + P_2 \left(G_1 + P_1 C_0 \right)$$

$$C_3 \equiv G_{3:0} = G_3 + P_3 \left(G_2 + P_2 \left(G_1 + P_1 C_0 \right) \right)$$

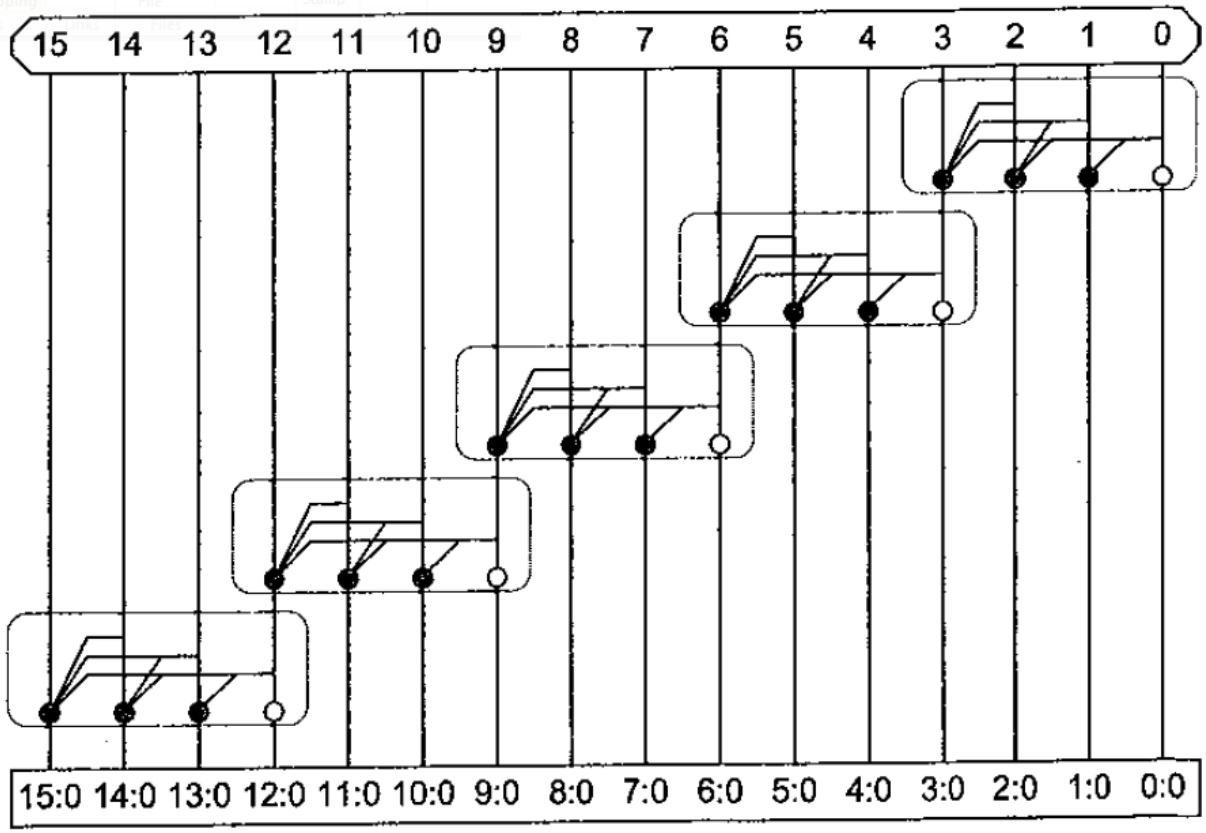


C_3 is calculated in "one" time unit but we must wait for carry to ripple through group to be ready.

How delay grows with chain length? quadratic!

Chain should be broken and buffered. Common length is 3-4.

Manchester carry chain adder using valency-4 stages



Similar to ripple carry adder but uses N/3 stages.

Involves a series propagate transistor per bit.

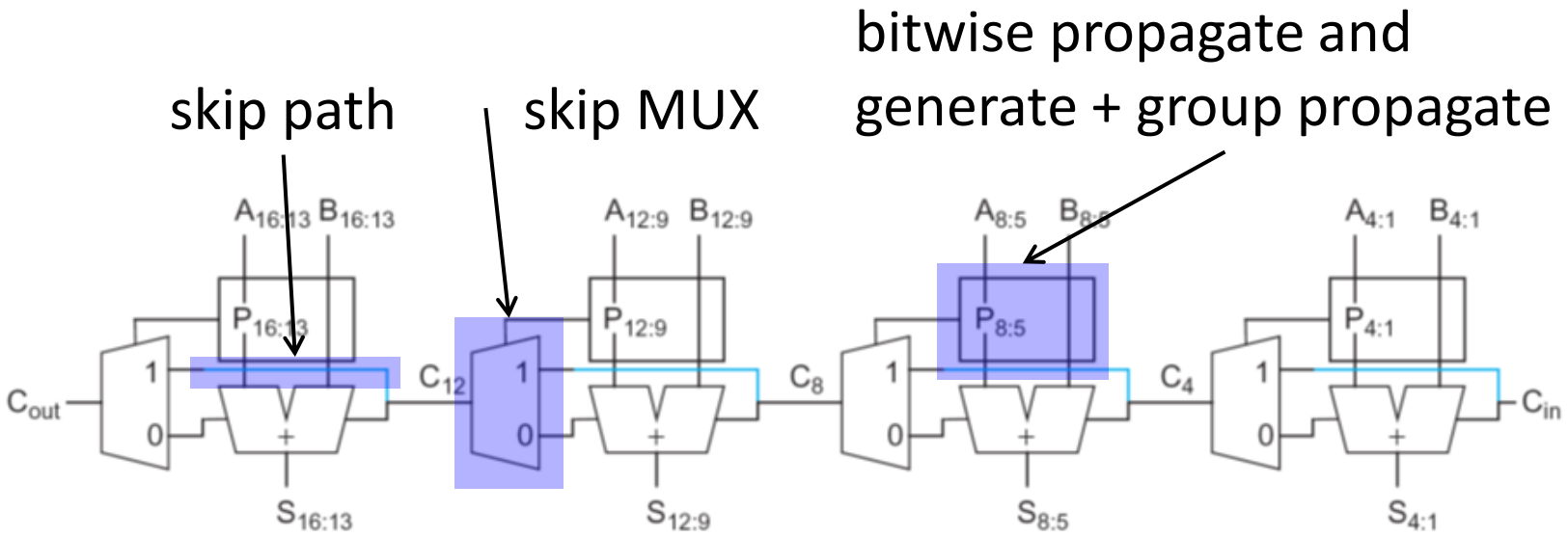
Faster than AND-OR or majority gate per bit in carry ripple.

Carry Skip Adder

Assume that the propagate computed for a group *i:j* is 1.

$$P_{i:j} = \prod_{k=j}^{i} A_k \oplus B_k = 1$$

Consequently, the carry-out of group *i:j* is the same as the carry-in and carry computation can be skipped.

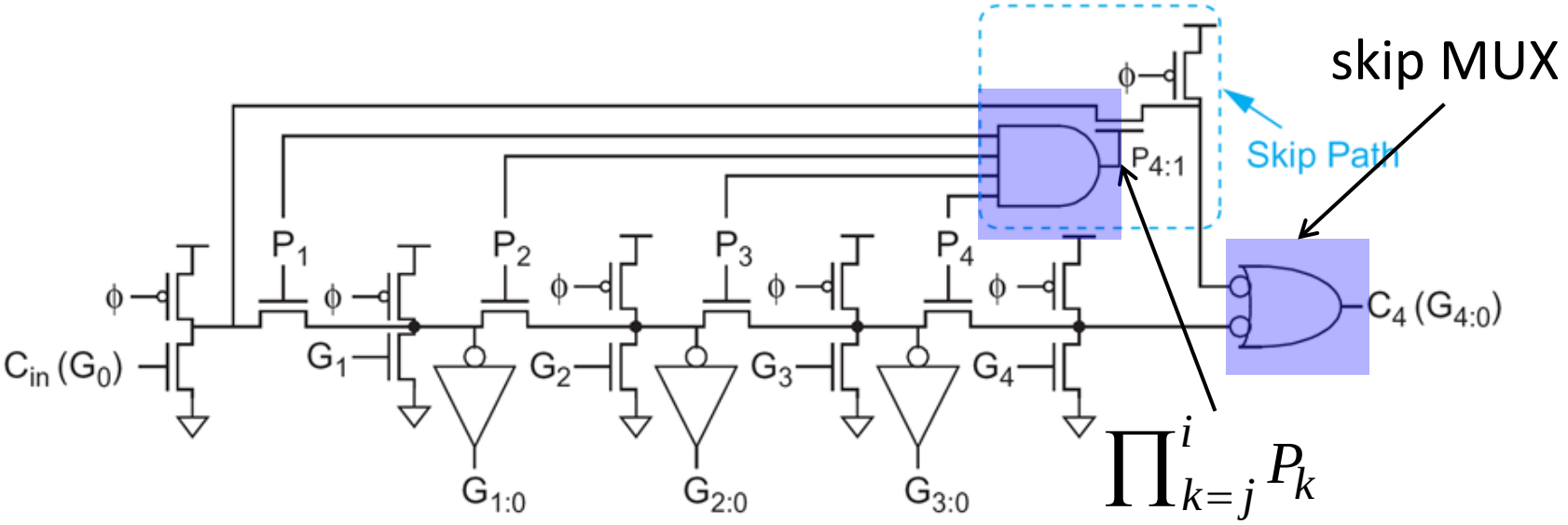


For group propagate 0 the carry generated within group is taken.

This is a considerable acceleration compared to carry-ripple, while hardware overhead is small.

Was proposed in 19th century by Charles Babbage and used by mechanical calculators.

Carry-skip adder Manchester stage (dynamic logic)

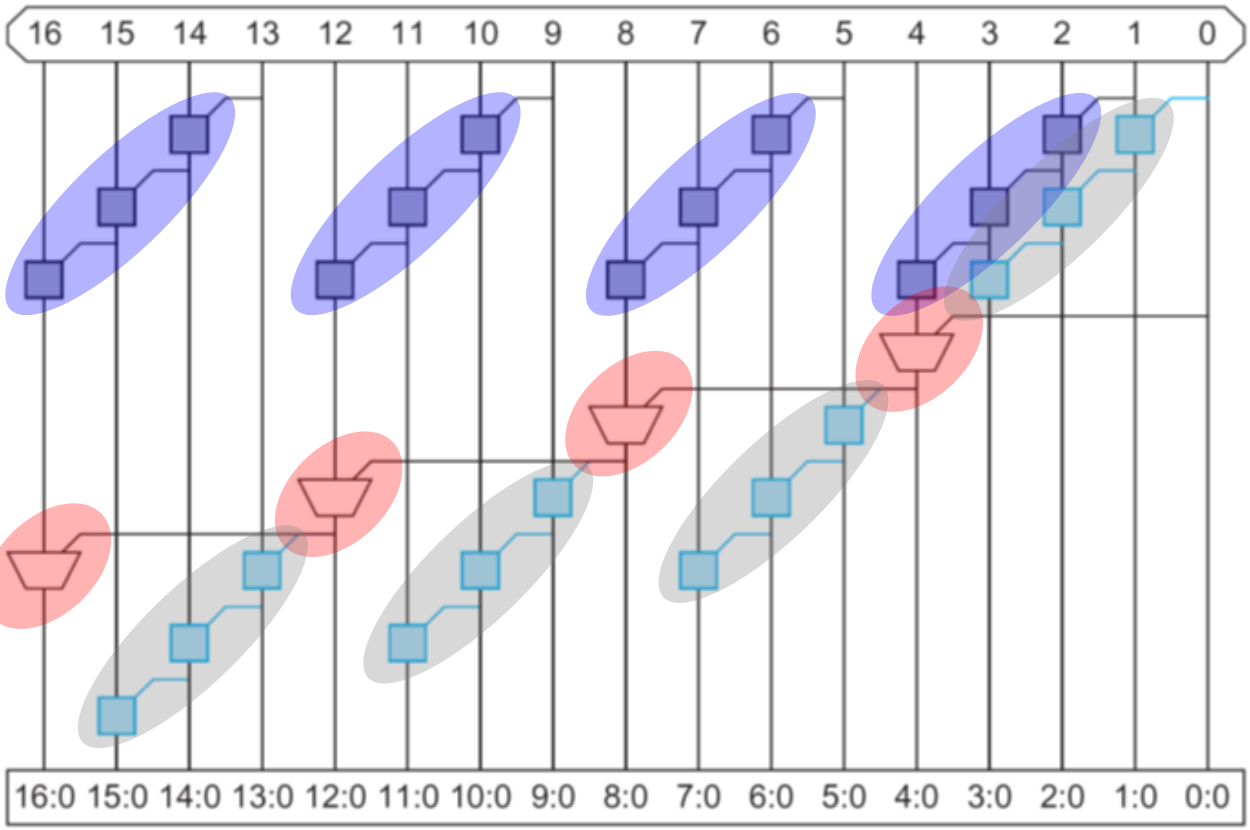


Propagation delay

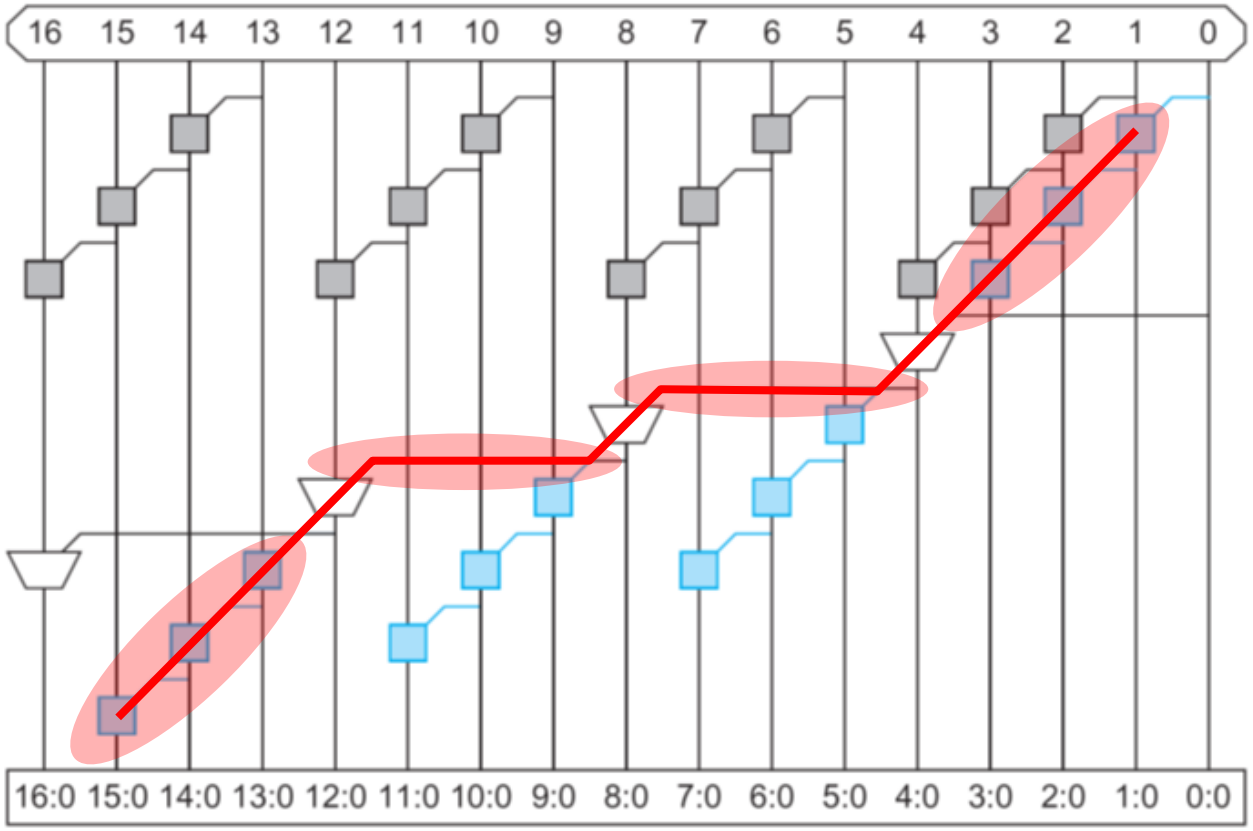
4-bit carry chain if each group generates a carry

carry skip chain

4-bit ripple chain if carry-in is by passed to chain



N-bit adder with *k* groups of *n* bits each (*N=kn*).



Delay of a chain is slower than skip propagation delay (AND, MUX).

First chain must compute sums and carry within n-1 delay units.

Carry propagates through *k-2* stages.

Last chain must compute sums within n-1 delay units.

$$T_{\text{carry-skip}} = \underbrace{(n-1)}_{\text{first chain}} + \underbrace{\left(\frac{N}{n}-2\right)}_{\text{skips}} + \underbrace{(n-1)}_{\text{last chain}} = 2n + \frac{N}{n} - 4$$

$$\frac{dT_{\text{carry-skip}}}{dn} = 2 - \frac{N}{n^2} \Rightarrow n^{\text{opt}} = \sqrt{\frac{N}{2}} \quad T^{\text{opt}}_{\text{carry-skip}} = 2\sqrt{2N} - 4$$

Example: consider 32-bit addition.

$T^{\text{opt}}_{\text{carry-skip}} = 2\sqrt{2 \bullet 32} - 4 = 12,$ compared to 32 units in carry-ripple adder.

Question: Can we further accelerate carry propagation?

Answer: Yes we can, block size may vary across adder.

Consider t ripple-carry adder groups $A_0, A_1, \ldots, A_{t-2}, A_{t-1}$. How should we distribute the N bits in those blocks?

Assume a skip chain of $A_1, \ldots, A_{t-2}$. Since skip is far faster than ripple carry, we wish to minimize the number b of bits in A_0 and A_{t-1}.

Bits are distribute as follows:

$$b,\ b+1,\ \ldots,\ b+t/2-1,\ b+t/2-1,\ \ldots,\ b+1,\ b$$

Summing over all blocks:

$$N = 2\sum_{i=0}^{t/2-1}(b+i) = t\left(b+\frac{t}{4}-\frac{1}{2}\right) \quad \Rightarrow \quad b = \frac{N}{t}-\frac{t}{4}+\frac{1}{2}$$

$$T_{\text{carry-skip}} = \underbrace{(b-1)}_{\text{first chain}} + \underbrace{(t-2)}_{\text{skips}} + \underbrace{(b-1)}_{\text{last chain}} = \frac{2N}{t} + \frac{t}{2} - 3$$

$$\frac{dT_{\text{carry-skip}}}{dt} = \frac{1}{2} - \frac{2N}{t^2} \quad \Rightarrow \begin{cases} t^{\text{opt}} = 2\sqrt{N} \\ T^{\text{opt}}_{\text{carry-skip}} = 2\sqrt{N} - 3 \end{cases}$$

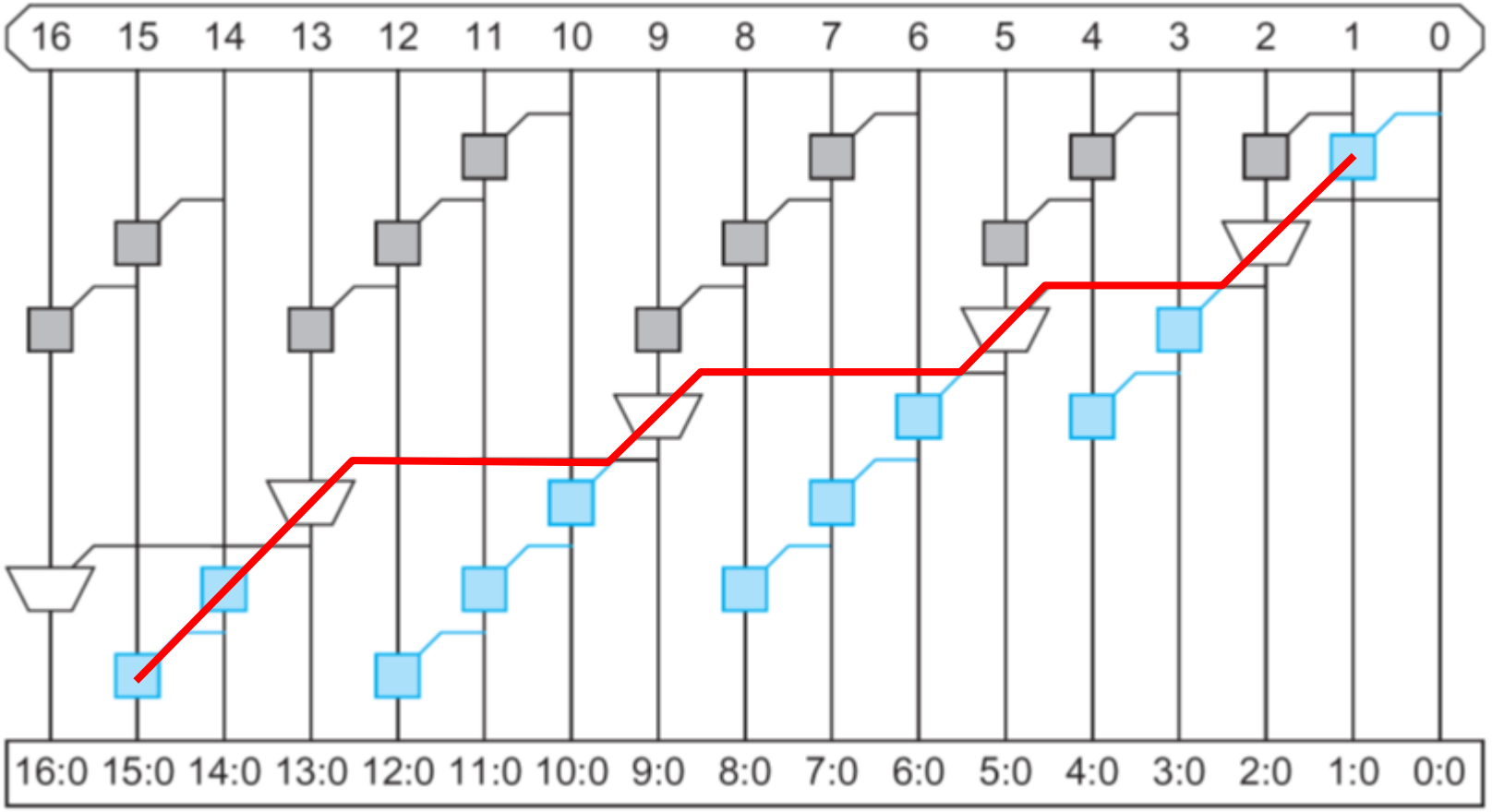
delay is decreased

$$\frac{T^{\text{opt}}_{\text{carry-skip}}(\text{var})}{T^{\text{opt}}_{\text{carry-skip}}(\text{fixed})} = \frac{2\sqrt{N} - 3}{2\sqrt{2N} - 4} \approx \frac{1}{\sqrt{2}}$$

number of blocks is increased

$$\frac{\#\text{blocks(var)}}{\#\text{blocks(fixed)}} = \frac{t^{\text{opt}}}{N/n^{\text{opt}}} = \frac{2\sqrt{N}}{\sqrt{2N}} = \sqrt{2}$$

Groups are of lengths [2, 3, 4, 4, 3] compared to [4, 4, 4, 4].



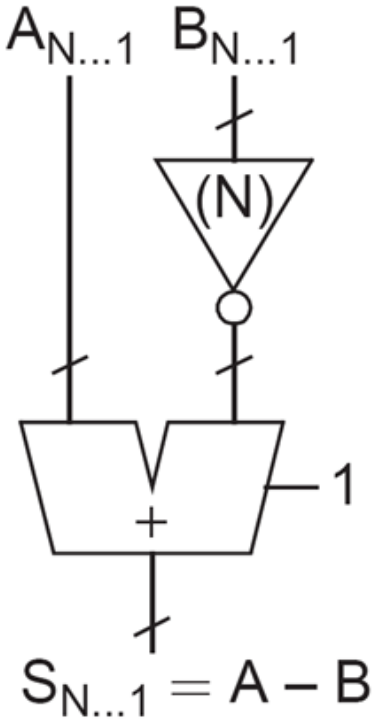
Saved 2 levels of logic on critical path compared to fixed.

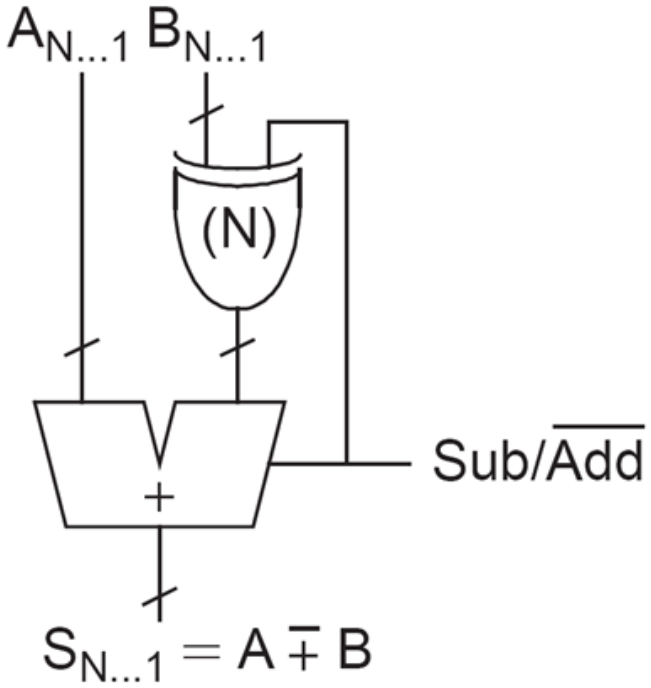
Subtraction

How to compute A-B?

Recall that in 2's complement $A - B = A + \bar{B} + 1$

We'd like to combine adder and subtracter in one circuit





Carry-Lookahead Adder

Carry-skip adder ripples the carry through the group, requiring waiting to determine whether the first group generates a carry.

Carry-lookahead (CLA) computes group generate signals as well as group propagate signals to avoid waiting for a ripple.

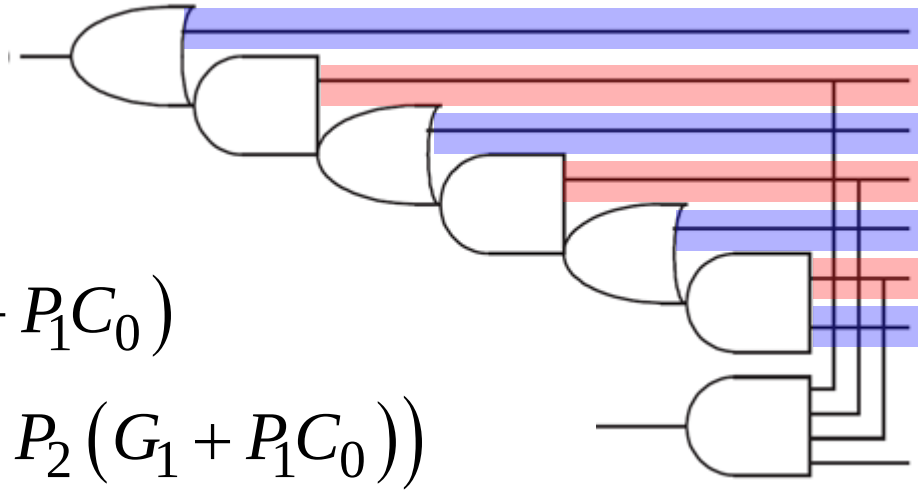
$$C_0 = G_{0:0}$$

$$C_1 = G_{1:0} = G_1 + P_1 C_0$$

$$C_2 = G_{2:0} = G_2 + P_2 \left(G_1 + P_1 C_0 \right)$$

$$C_3 = G_{3:0} = G_3 + P_3 \left(G_2 + P_2 \left(G_1 + P_1 C_0 \right) \right)$$

Valency-4 PG



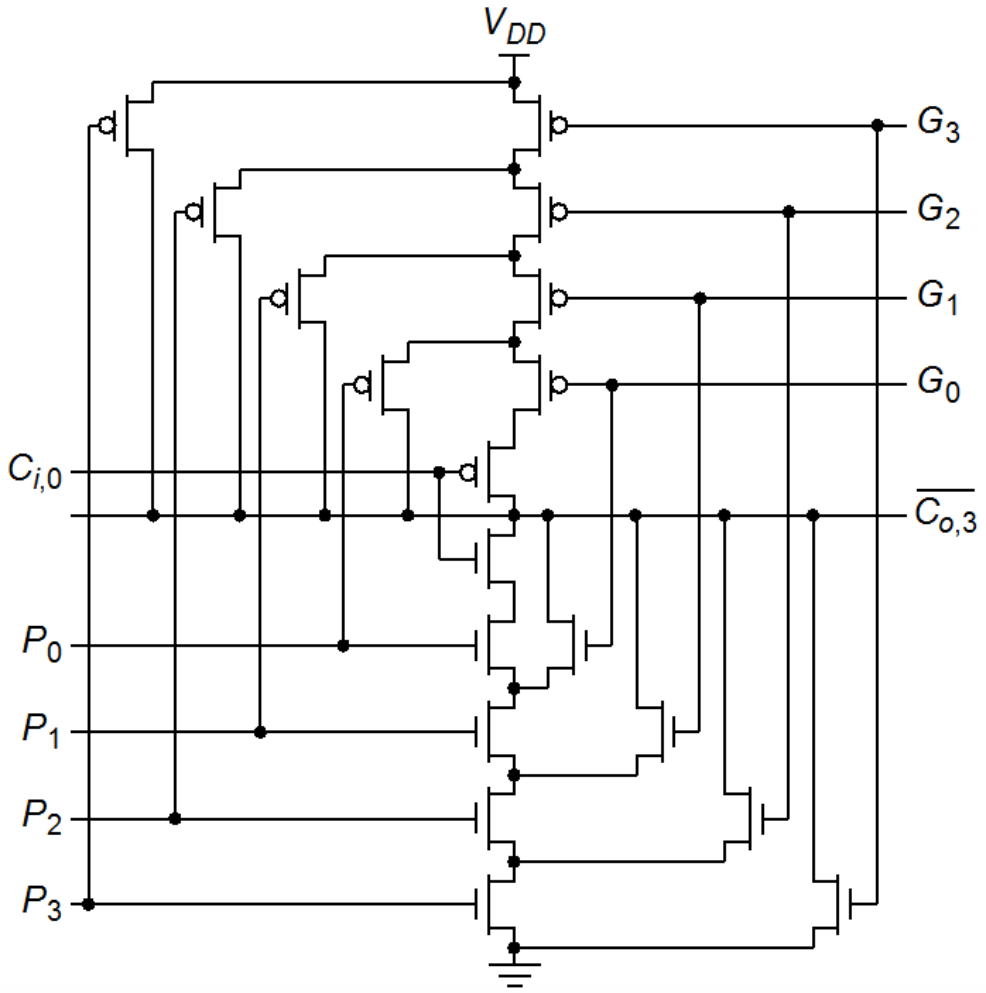
Carry-lookahead circuit with half devices compared to AOAO…

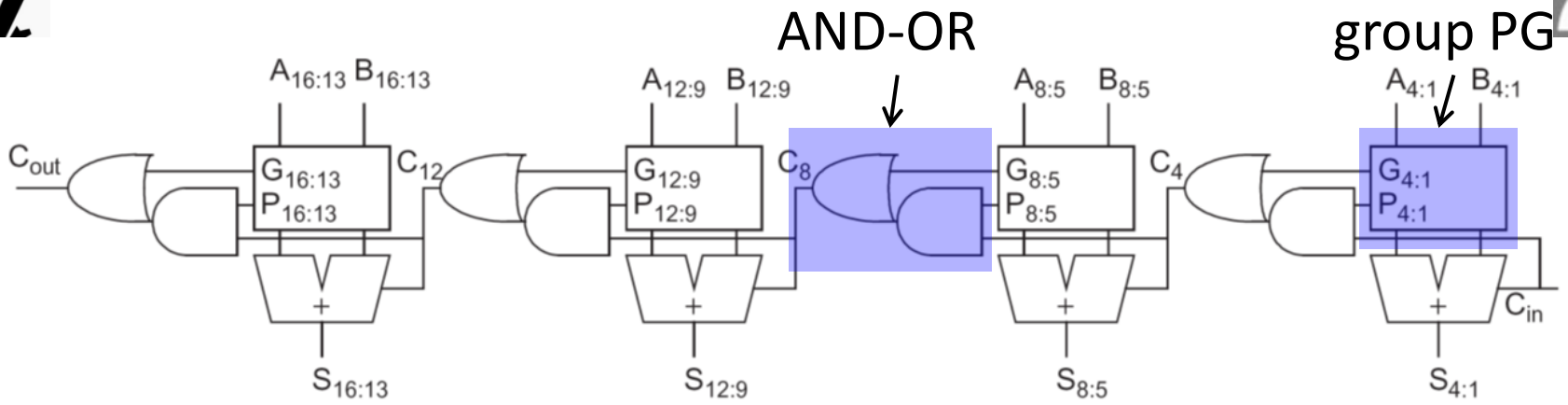
C_{out} is complementary

P and G signals connect and disconnect path to V_{dd} / V_{ss}

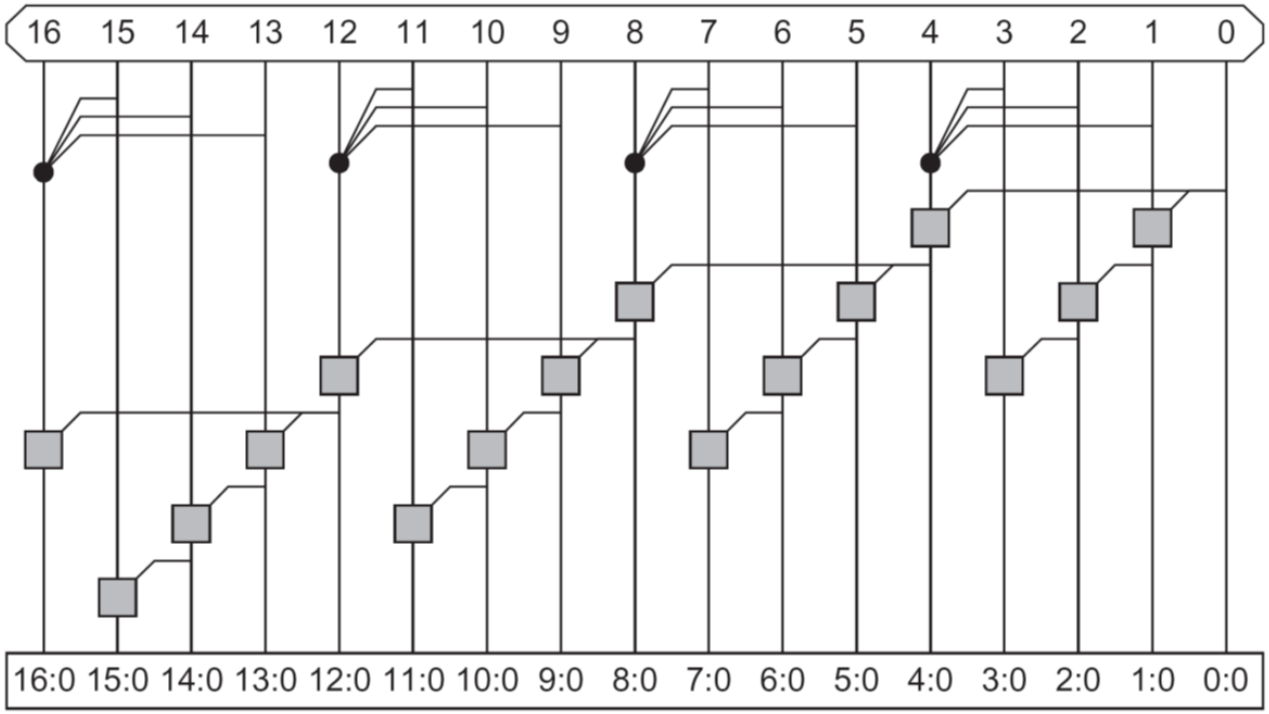
What happens when all P=1 and G=0?
Both paths to V_{dd} and V_{ss} are closed.

C_{in} then takes care

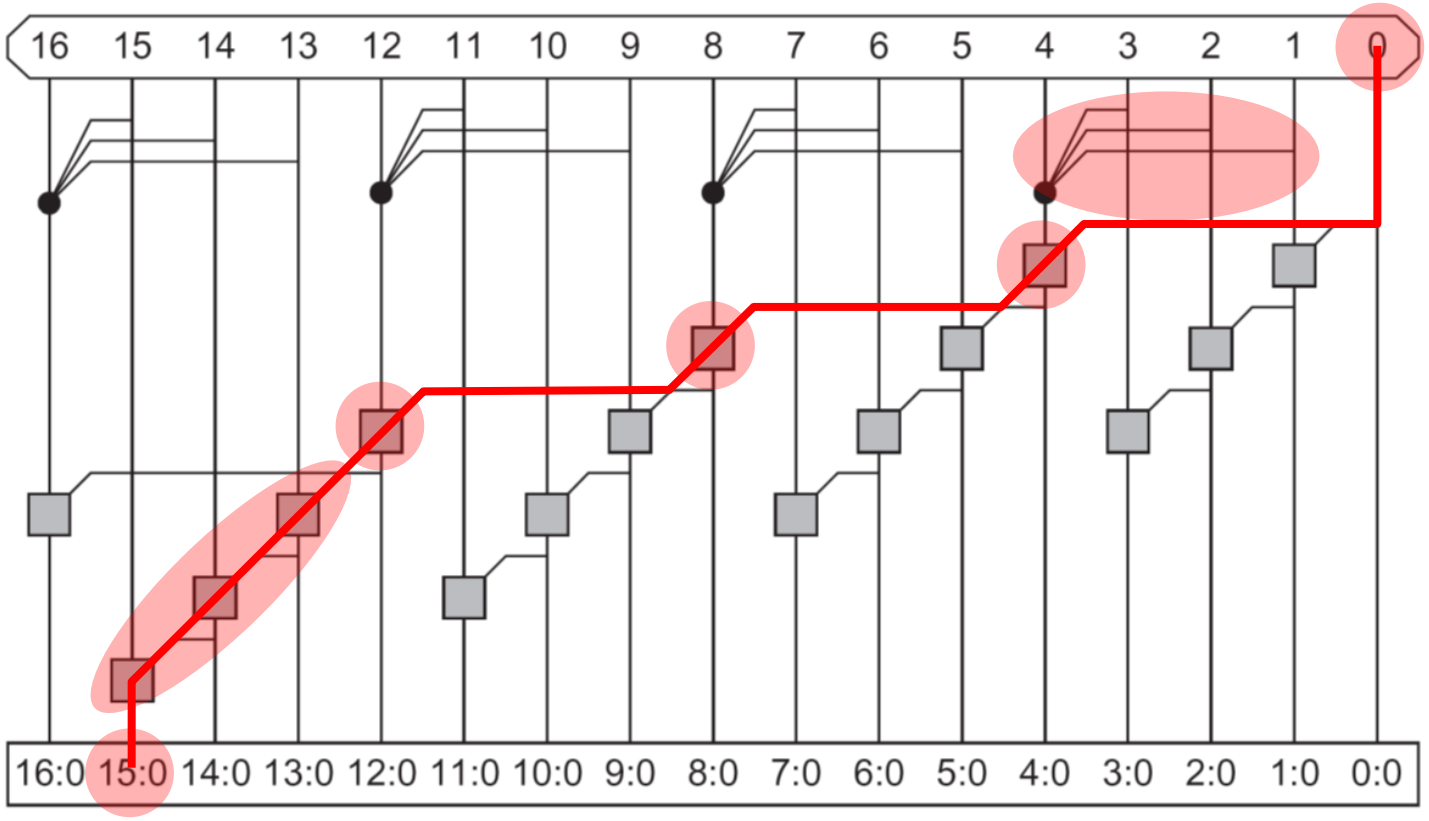




N-bit adder with *k* groups of *n* bits each (*N*=*kn*).



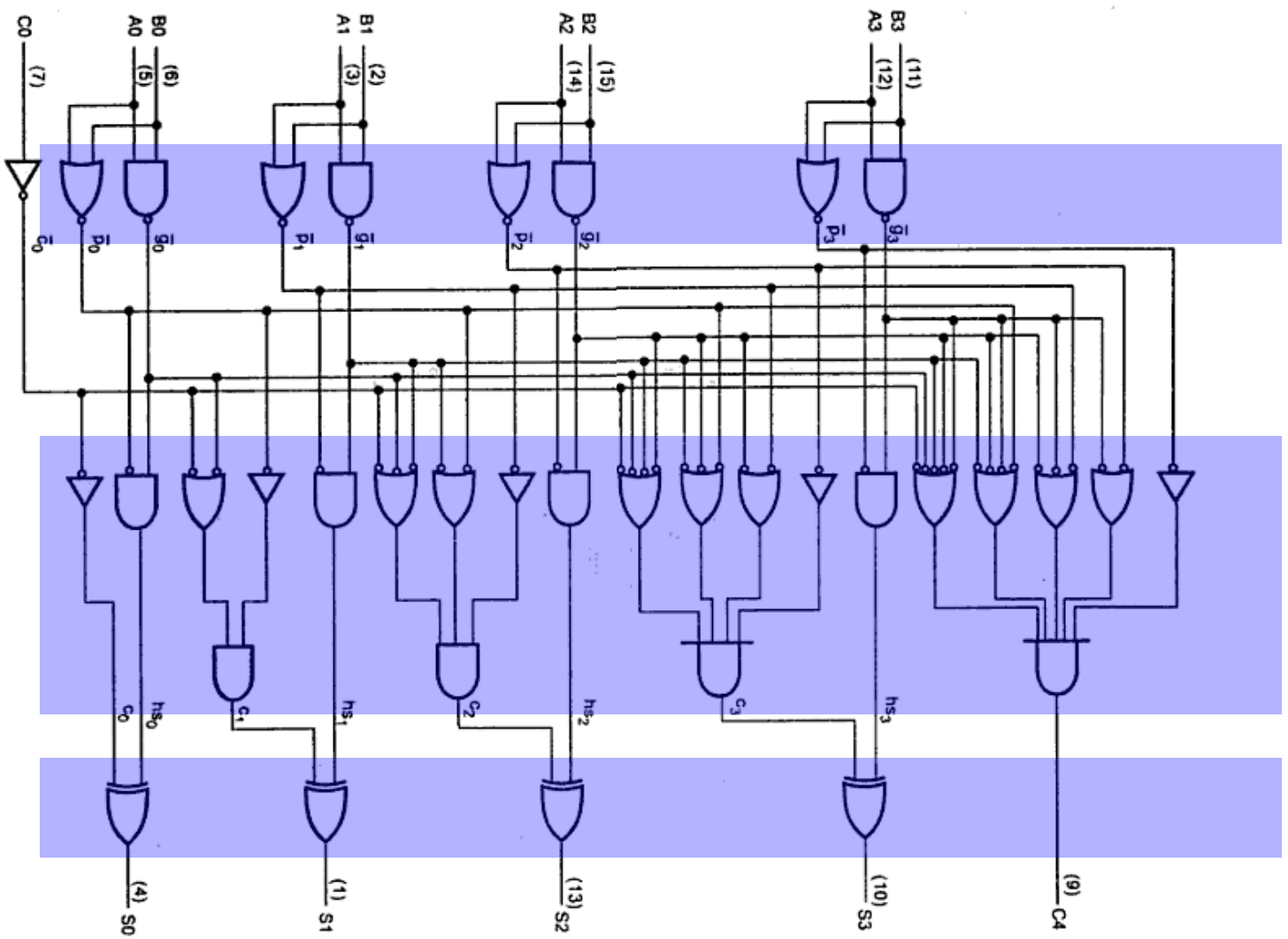
Propagation delay



$$T_{\mathrm{CLA}} = t_{\mathrm{PG}} + t_{\mathrm{PG}(n)} + t_{\mathrm{AO}}\left[(k-1)+(n-1)\right] + t_{\mathrm{XOR}}$$

No better than variable-length carry skip, but requires more HW due to PG generation per group.

Commercial MSI 4-bit CLA adder



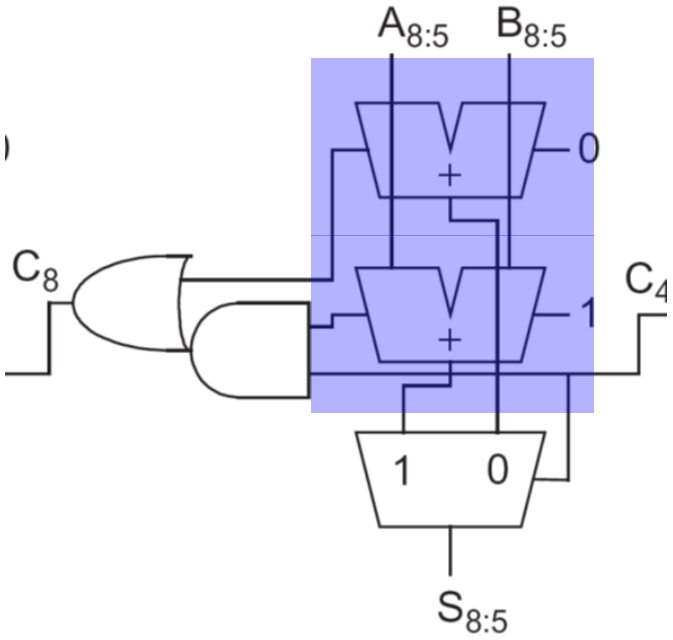
Carry-Select Adder

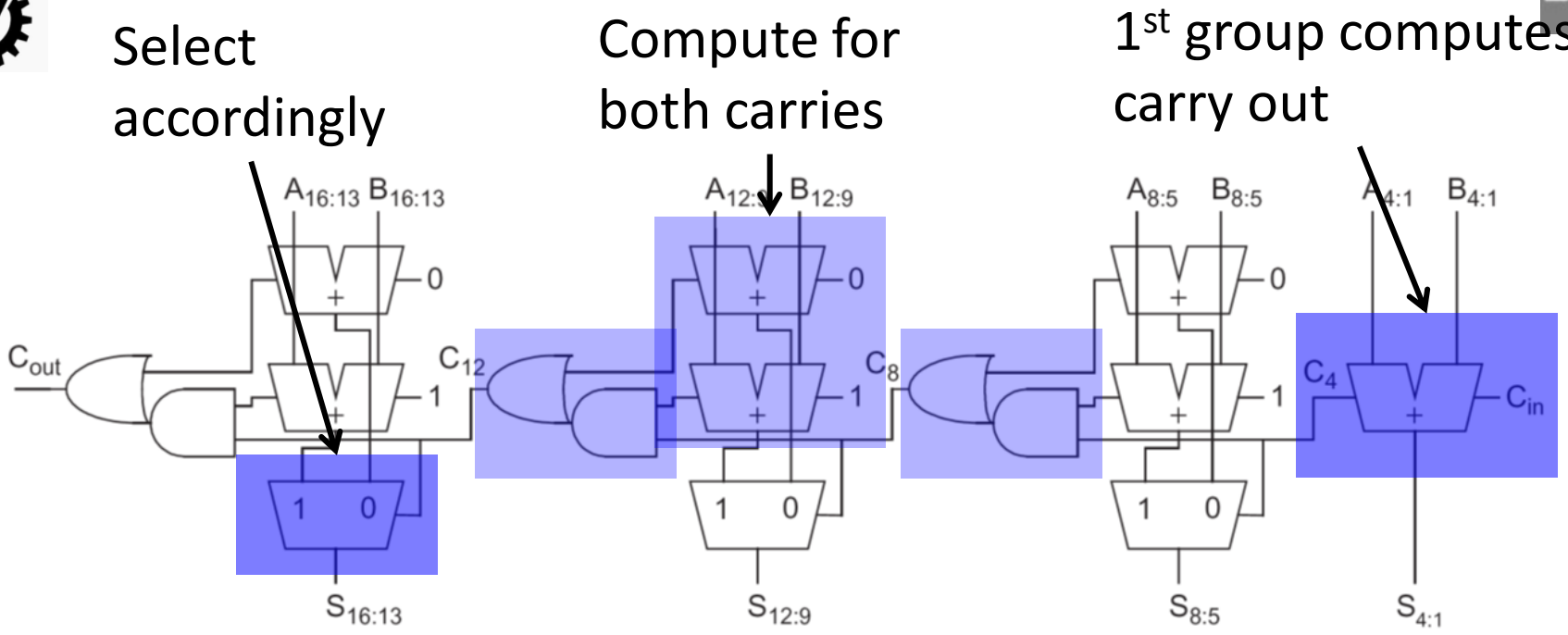
The critical paths in carry-skip and carry-lookahead involves carry calculation into each *n*-bit group and then using it for the sums within the group.

It is possible to pre compute the outputs for both 0/1 carry inputs and then select accordingly.

If C_4=0, top adder applies for C_8.

If C_4=1 bottom adder applies for C_8. Notice that $c_{out}(c_{in}=1) >= c_{out}(c_{in}=0)$.





Propagation delay

$$T_{\text{select}} = t_{\text{PG}} + t_{\text{AO}}\left[n + (k-2)\right] + t_{\text{MUX}}$$

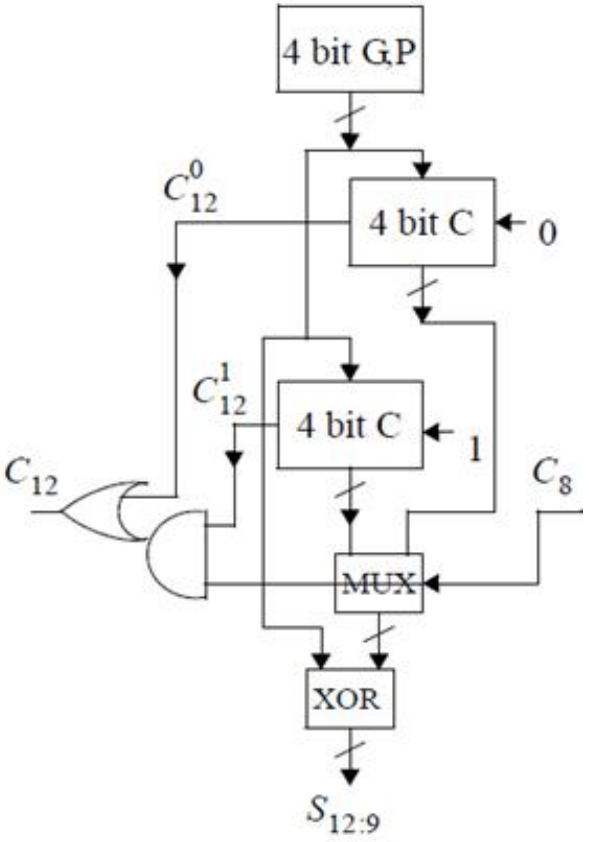
Simultaneous PG for all bits

n-bits of first group adder

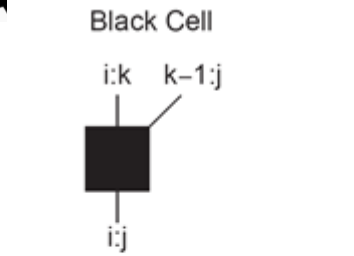
Carry-Increment Adder

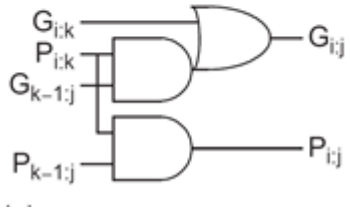
Carry-select adder is fast but the amount of circuits is about twice compared to others. This is both power and area penalty.

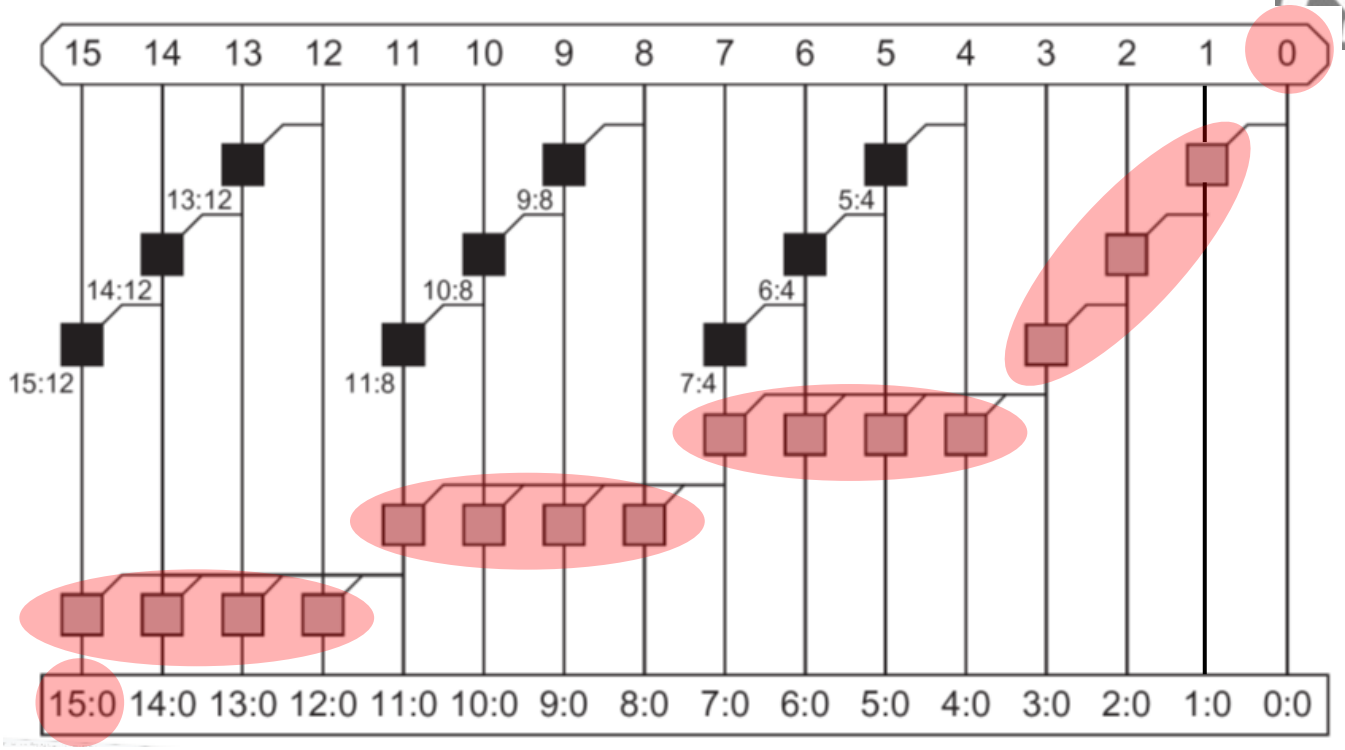
The PG and XOR circuits are similar in 0 and 1 adders, independent , hence MUX can be used to select the proper input of XOR.



This is called ***carry-increment*** adder.







$$T_{\text{increment}} = t_{\text{PG}} + t_{\text{AO}}\left[(n-1)+(k-1)\right] + t_{\text{XOR}}$$

Acceleration is possible by variable group size

$$T_{\text{increment}} = t_{\text{PG}} + \sqrt{2N}\,t_{\text{AO}} + t_{\text{XOR}}$$

Tree Adders

In wide adders the delay of the carry passing through stages becomes dominant.

The delay can be reduced by looking ahead across lookahead blocks.

The square root delay can be improved to logarithmic delay by constructing multilevel lookahead structures.

There are many ways to build lookahead trees, offering tradeoffs between number of circuits, fan-out and amount of interconnects. Those are translated into area and power.

Such adders are known as ***lookahead*** adders, ***logarithmic*** adders or ***parallel-prefix*** adders.

Brent-Kung tree

Compute prefixes for 2-bit groups. Then prefixes for 4-bit groups. Then 8-bit and 16-bit groups. Prefixes fan back down to compute carry-in to each bit. $2(log_2 N)$ - 1 levels (area), fan-out 2.

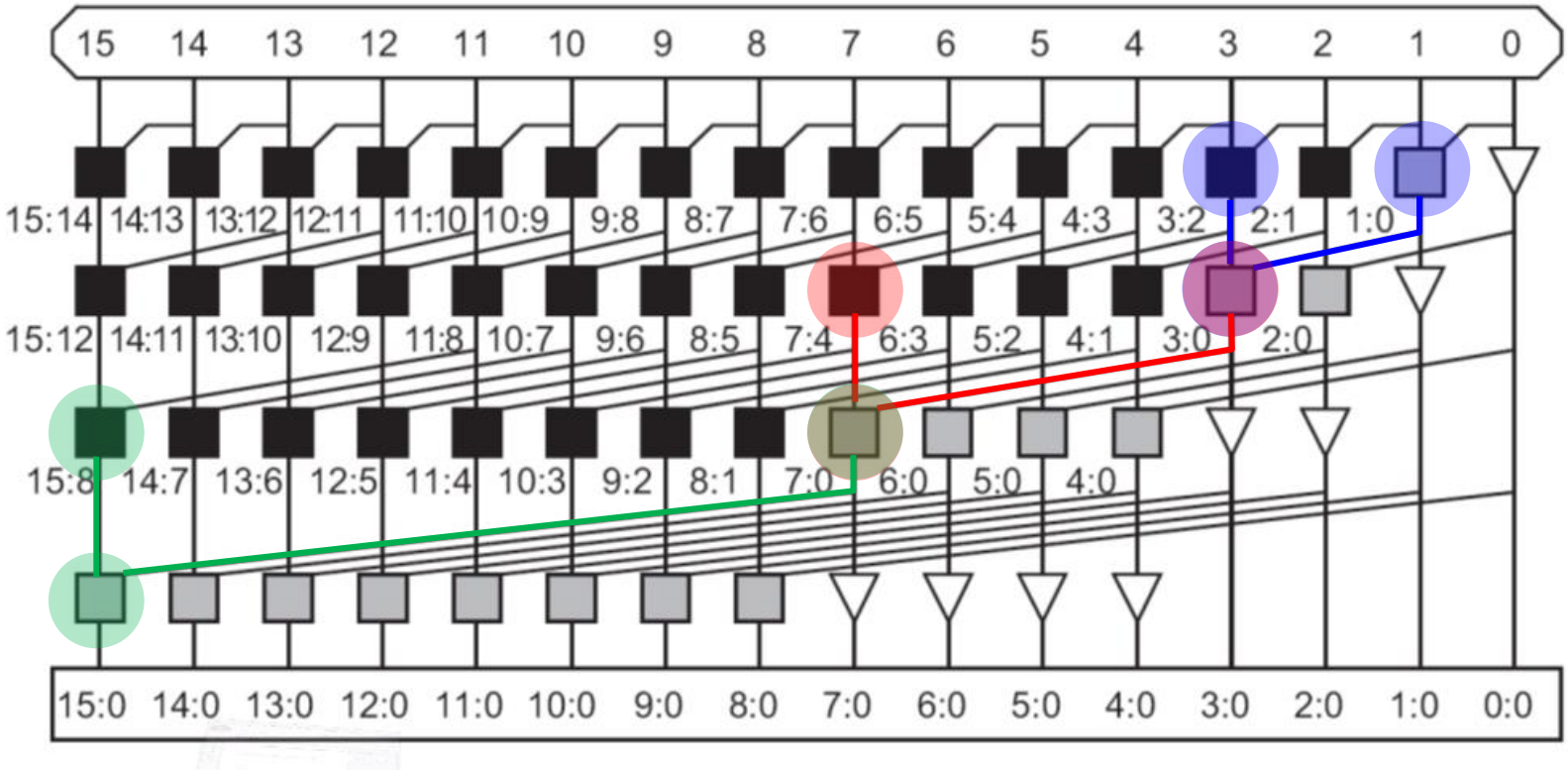
Sklansky tree

Intermediate prefixes can be computed along with those of large groups.

Delay reduced to $log_2 N$. Fan out is doubled at each row.

Transistor sizing and buffering is required (area, power).

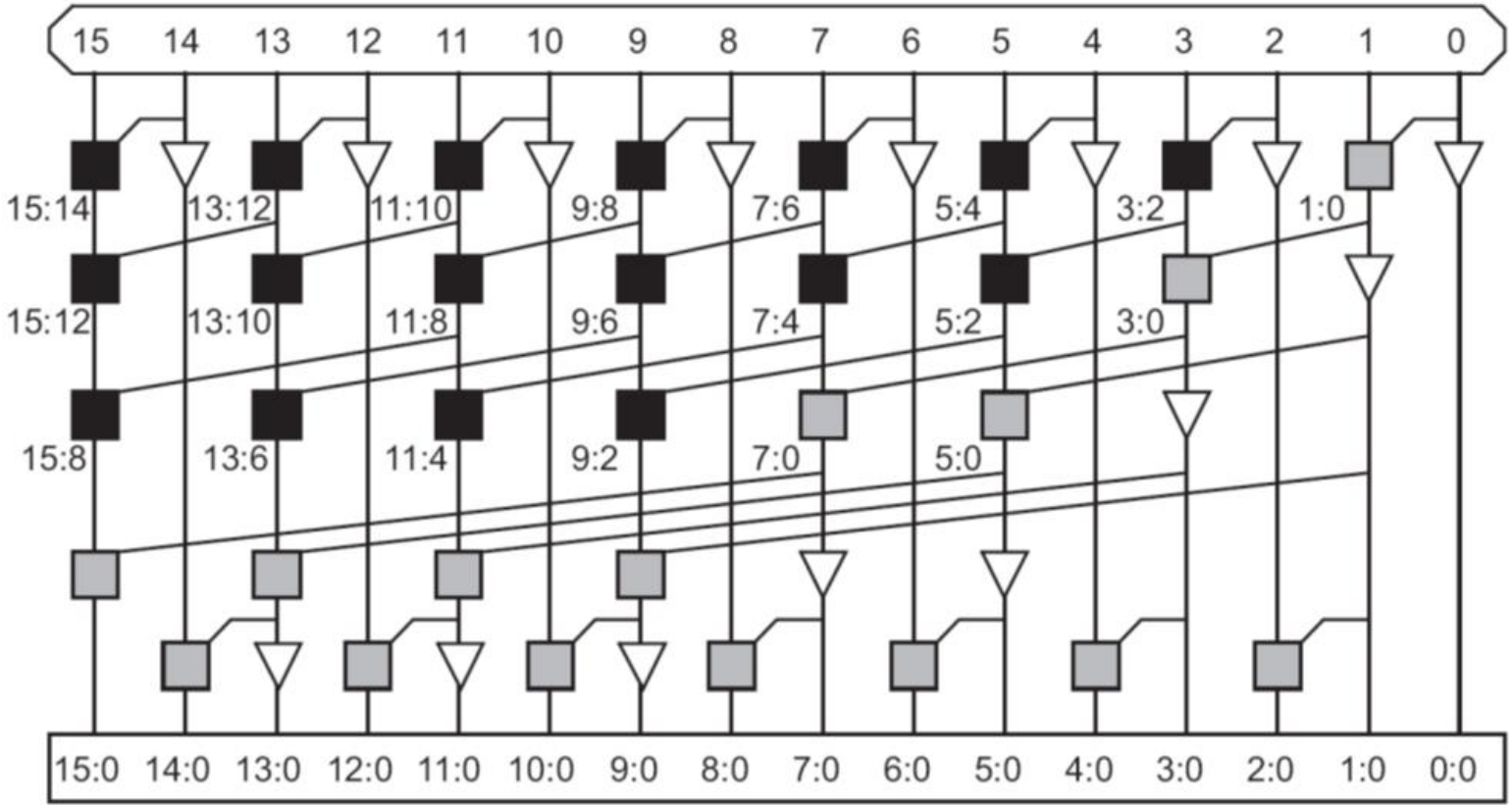
Kogge-Stone tree



Achieves *log₂N stages*. Fan out is 2.

Wire length grows is quadratic with *N*. It significantly increases area, buffers, power.

Han-Carlson tree



Use Kogge-Stone on odd bits, cutting hardware by factor 2.

Use one more stage to ripple into even bits.

Comparison of Adder Architectures

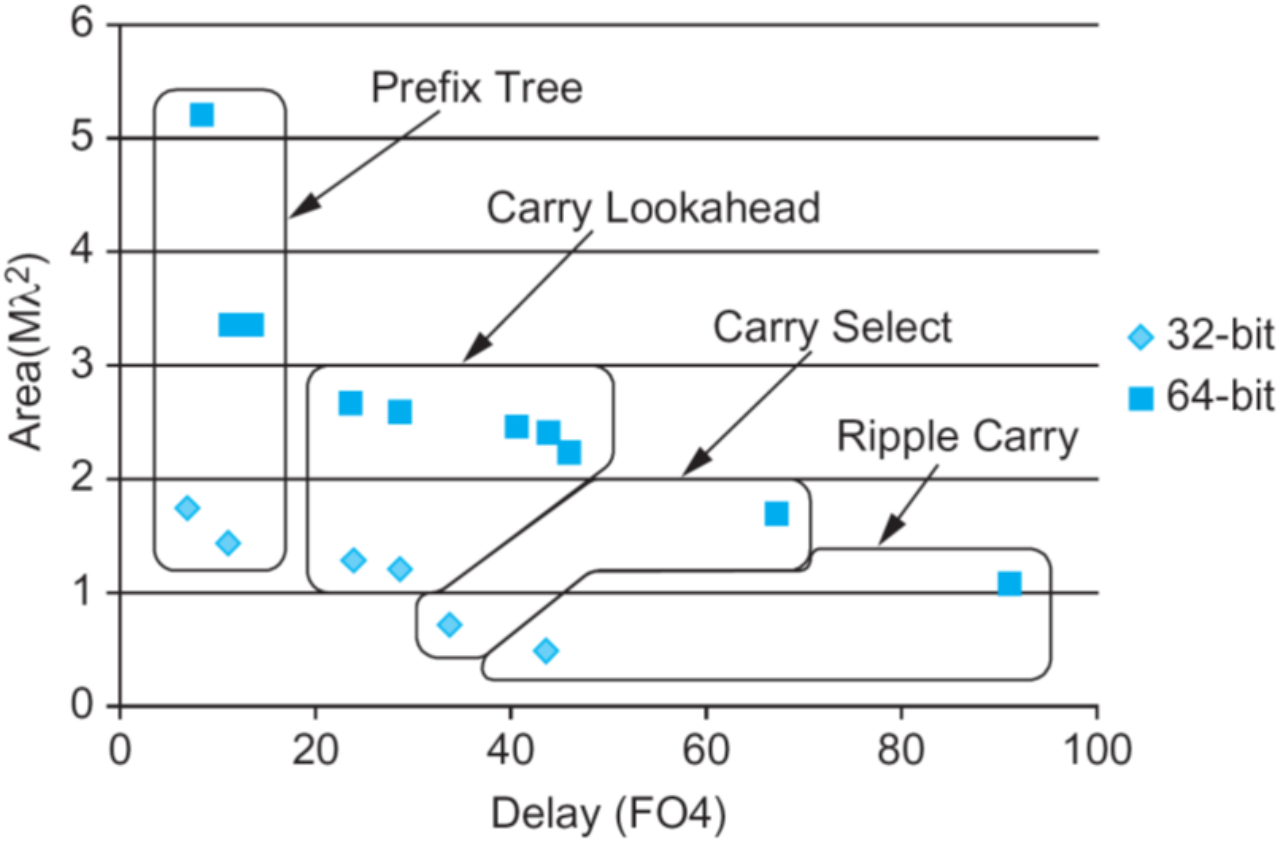
Architecture	Logic Levels	Max Fan-out	# Wiring Tracks	# Cells
Ripple-Carry	$N-1$	1	1	$N-1$
Carry-Skip (*n=4*)	$N/4+5$	2	1	$1.25N$
Carry-Increment (*n=4*)	$N/4+2$	4	1	$2N$
Carry-Increment (var.)	$\sqrt{2N}$	$\sqrt{2N}$	1	$2N$
Brent-Kung	$2\log_2 N-1$	2	1	$2N$
Sklansky	$\log_2 N$	$N/2+1$	1	$0.5N\log_2 N$
Kogge-Stone	$\log_2 N$	2	$N/2$	$N\log_2 N$
Han-Carlson	$\log_2 N+1$	2	$N/4$	$0.5N\log_2 N$

PG and XOR logic is not counted.

Ripple-carry should be used when they meet timing constraints (small area and power).

For 64 bits and up tree adders are distinctly faster.

Logic synthesizers automatically map the "+" operator into appropriate adder to meet timing constraints while minimizing area and power (aka **design ware**).



Carry Probabilities

What is the average length of a carry in addition?

Carry generation probability: $1/4$

Carry kill probability: $1/4$

Carry propagation probability: $1/2$

Given carry generated at bit i

The probability that it

- will propagate up to and including bit $j-1$
- and stops at bit j $(j > i)$ is:

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

For a k-bit adder, the expected length of a carry generated at bit i is:

$$\sum_{j=i+1}^{k-1} \overbrace{(j-i)2^{-(j-i)}}^{\text{stops}} + \overbrace{(k-i)2^{-(k-1-i)}}^{\text{does not stop}}$$

$$= \sum_{l=1}^{k-1-i} l2^{-l} + (k-i)2^{-(k-1-i)}$$

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

by induction $\sum_{l=1}^{p} l2^{-l} = 2-(p+2)2^{-p}$

Consequently, for long adders ($i \ll k$) the avarage length of carry propagation is nearly 2.

The short length of average carry propagation indicates that the average worst-case may also be short.

A usual design of a *k*-bit adder is targeting the worst-case where the carry is propagating along the entire bits, regardless of adder architecture.

Burks, Goldstine and von Neumann [1946] noticed that the average worst-case carry propagation length is **$\log_2 k$**.

Let $\eta_k(h)$ be the probability that the longest carry chain in a k-bit adder is h or more.

The probability that the longest carry chain is exactly h is therefore $\eta_k(h) - \eta_k(h+1)$.

The longest carry chain is $\geq h$ in two exclusive ways:

(a) The $k-1$ LSBs have a carry chain $\geq h$.

(b) The $k-1$ LSBs have no such a carry chain, but the h MSBs do have.

Thus, we have

$$\eta_k(h) = \underbrace{\eta_{k-1}(h)}_{\text{case (a)}} + \underbrace{\left[1-\eta_{k-1}(h)\right]}_{\text{not case (a)}} \times \overbrace{\frac{1}{2}\times 2^{-h}}^{\text{case (b)}} \leq \eta_{k-1}(h) + 2^{-(h+1)} .$$

carry generated and propagated along h bits

Therefore, $\eta_k(h) - \eta_{k-1}(h) \leq 2^{-(h+1)}$. Assuming $\eta_i(h) = 0$ for $i < h$,

$$\eta_k(h) = \overbrace{\sum_{i=h}^{k}\left[\eta_i(h) - \eta_{i-1}(h)\right]}^{\text{telescopic sum}} \leq (k-h+1)2^{-(h+1)} \leq k2^{-(h+1)} .$$

The expected length λ of the longest carry chain

$$\lambda = \sum_{h=1}^{k} h\left[\eta_k(h) - \eta_k(h+1)\right] = \left[\eta_k(1) - \eta_k(2)\right] + 2\left[\eta_k(2) - \eta_k(3)\right]$$

$$+ \cdots + k\left[\eta_k(k) - 0\right] = \sum_{h=1}^{k} \eta_k(h) .$$

$$\lambda = \sum_{h=1}^{k} \eta_k(h) = \sum_{h=1}^{\lfloor \log_2 k \rfloor - 1} \eta_k(h) + \sum_{h=\lfloor \log_2 k \rfloor}^{k} \eta_k(h)$$

$$\leq \sum_{h=1}^{\lfloor \log_2 k \rfloor - 1} 1 + \sum_{h=\lfloor \log_2 k \rfloor}^{k} k 2^{-(h+1)} < \left(\lfloor \log_2 k \rfloor - 1\right) + k 2^{-\lfloor \log_2 k \rfloor}.$$

Let $\lfloor \log_2 k \rfloor = \log_2 k - \varepsilon, \ 0 \leq \varepsilon < 1.$

Noting that $2^{-\log_2 k} = \frac{1}{k}$, and $2^{\varepsilon} < 1 + \varepsilon$, we get

$$\lambda < \left(\lfloor \log_2 k \rfloor - 1\right) + k 2^{-(\log_2 k - \varepsilon)} =$$

$$\left(\log_2 k - \varepsilon - 1\right) + 2^{\varepsilon} < \log_2 k.$$

Carry-Completion Detection

Worst-case carry propagation of length ***k*** almost never materializes.

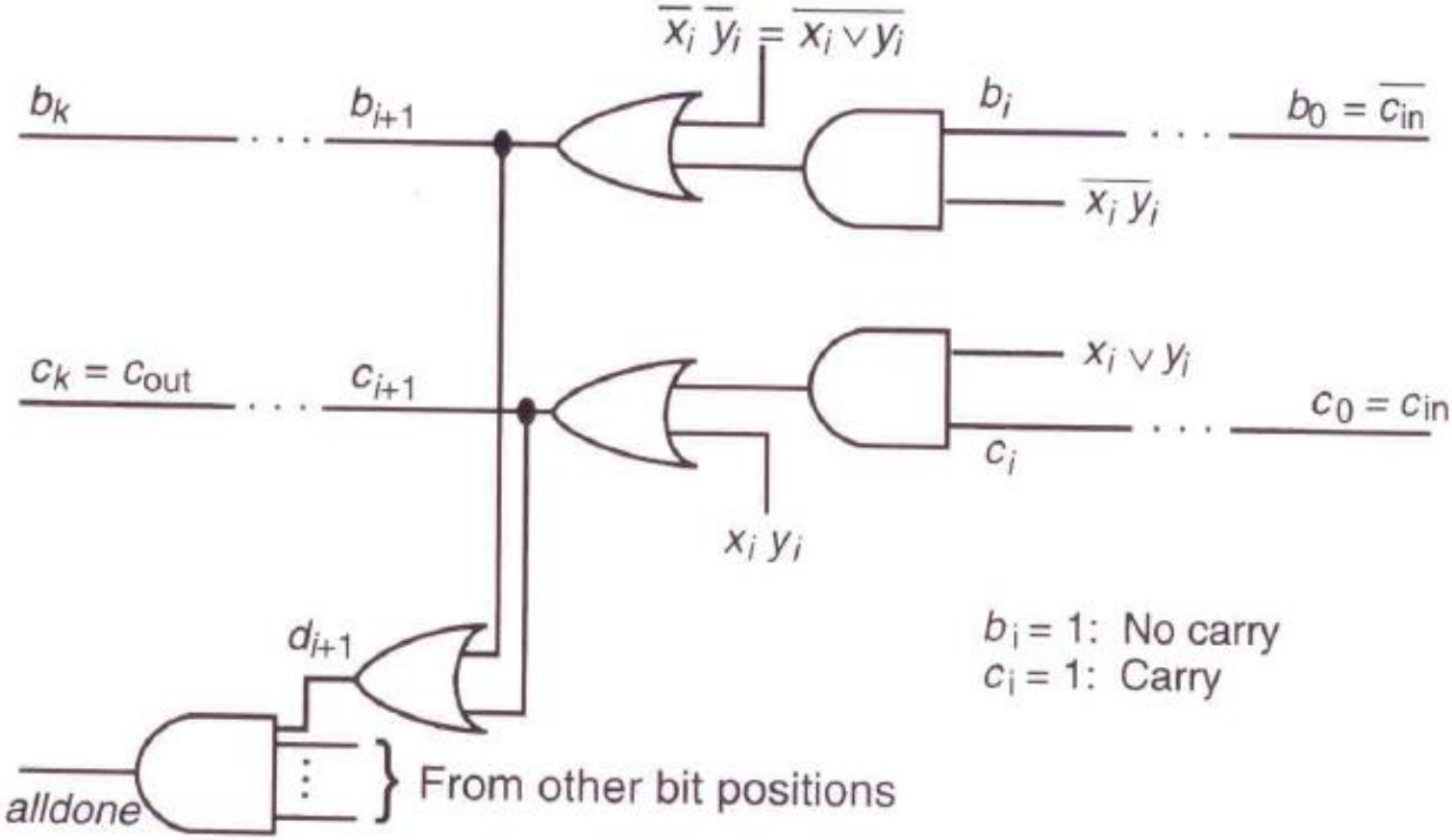
A carry-completion detection adder performs addition in average $\mathbf{O(\log_2 k)}$ time.

A carry 0 is also explicitly represented and allowed to propagate between stages. The carry into stage *i* is represented by the two-rail code:

$$(b_i, c_i) = \begin{cases} (0,0) & \text{Carry not yet known} \\ (0,1) & \text{Carry known to be 1} \\ (1,0) & \text{Carry known to be 0} \end{cases}$$

$(0,0)$ carry unknown,

$(0,1)$ carry known to be 1, $(1,0)$ carry known to be 0.



Two 1s generate a carry of 1 propagating towards MSB.

Two 0s generate a carry of 0 propagating towards MSB.

Initially, all carries are (0,0), namely, unknown.

The carry $(\overline{c}_{\text{in}}, c_{\text{in}})$ is injected into the LSB.

When every carry assumes one of the values (0,1) or (1,0) carry propagation is complete.

The local "done" signals $d_i = b_i \vee c_i$ are ANDed to form the global *alldone* signal, indicating carry propagation completion.

Excluding initialization and carry-completion detection times, the latency of k-bit carry-completion adder ranges from 1 to $\mathbf{2}k\mathbf{+1}$ gate delays, with $\mathbf{2log_2}k\mathbf{+1}$ average gate delays.

Behrooz Parhami, Computer Arithmetic, Oxford, 2010, page 100:

"Because the latency of the carry-completion adder is data-dependent, the design of Fig. 5.9 is suitable for use in asynchronous systems. Most modern computers, however, use synchronous logic and thus cannot take advantage of the high average speed of a carry-completion adder."