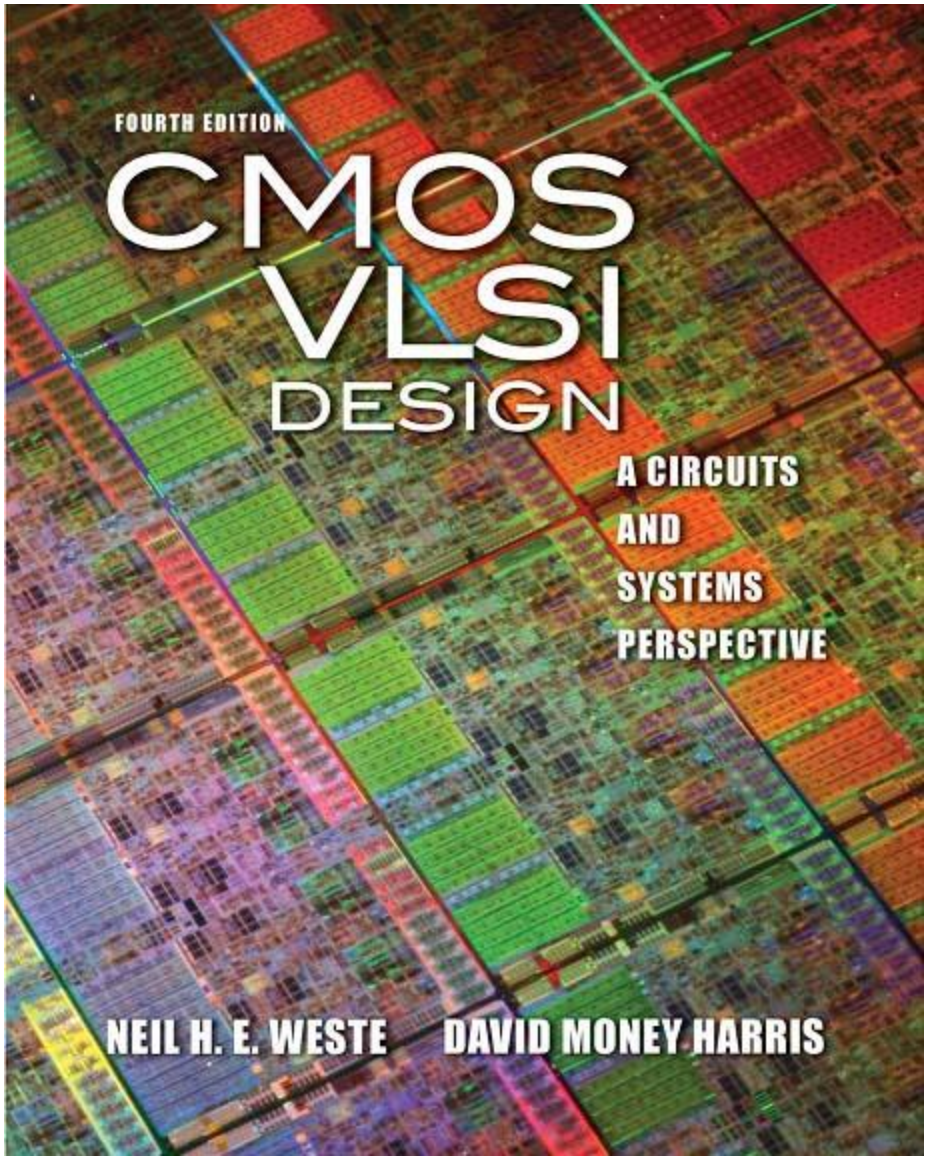
CPE 110408423
VLSI Design
Chapter 11: Datapath Subsystems

Bassam Jamil

[Computer Engineering Department,
Hashemite University]

Chip Functions:

- Datapath Operators
- Memory Elements
- Control Structure
- Special-Purpose Cells:
  - I/O
  - Power Distribution
  - Clock Generation and Distribution
  - Analog and RF



Lecture 1: Adders / Subtractors

Outline

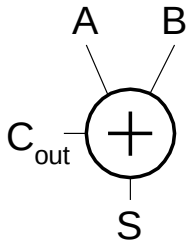
- Single-bit Addition
- Carry-Ripple Adder
- Propagate and Generate (P/G) logic
- Carry-Lookahead Adder
- Tree Adder

11.2.1 Single-Bit Addition

Half Adder

$S = A \oplus B$

$C_{out} = A.B$

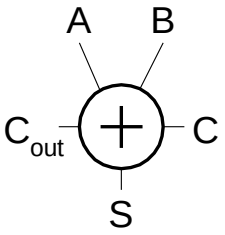


A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Full Adder

$S = A \oplus B \oplus C$

$C_{out} = MAJ(A,B,C)$



A	B	C	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

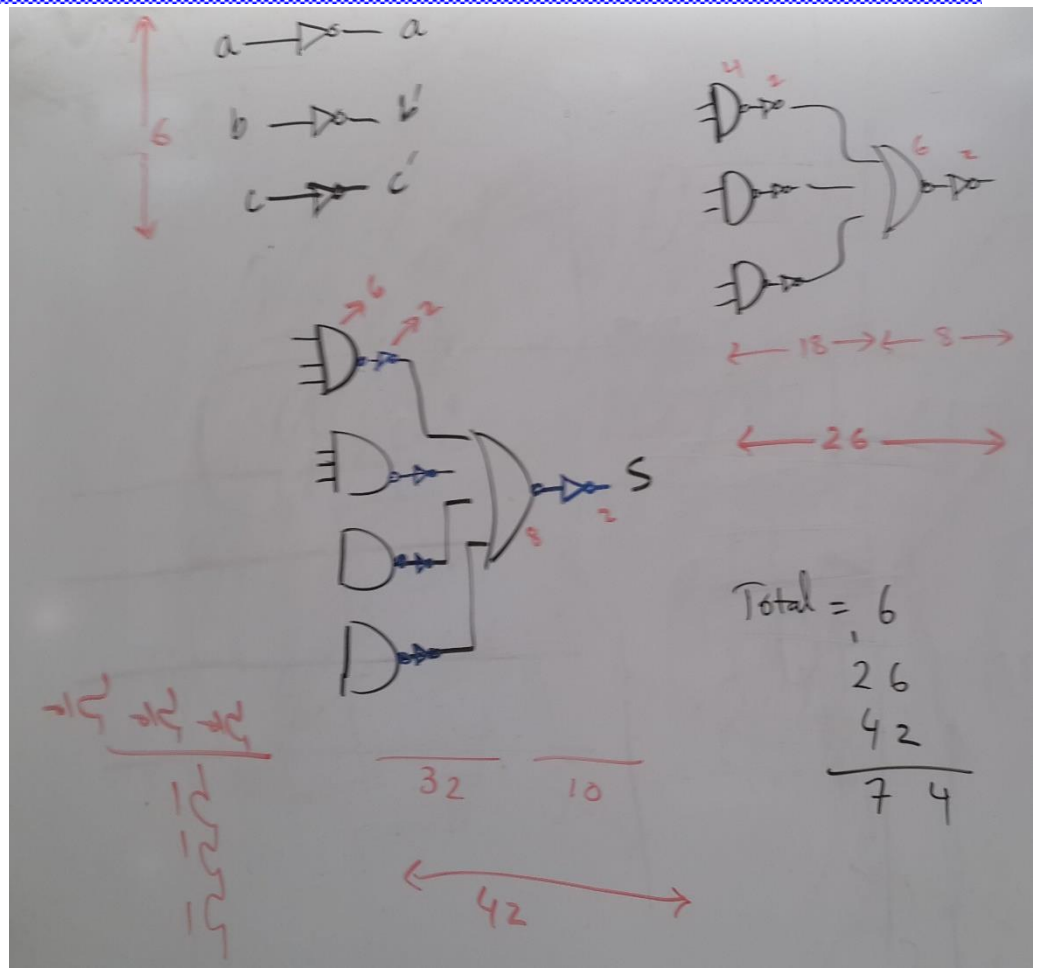
The carry gate is also called a *majority gate because it produces a 1 if at least two of* the three inputs are 1.

Cout = A B + B C + A C

= [A' B' + C' (A' + B')]'

Unoptimized FA implementation

Un-optimized FA Imp. requires 74 transistors.



PGK

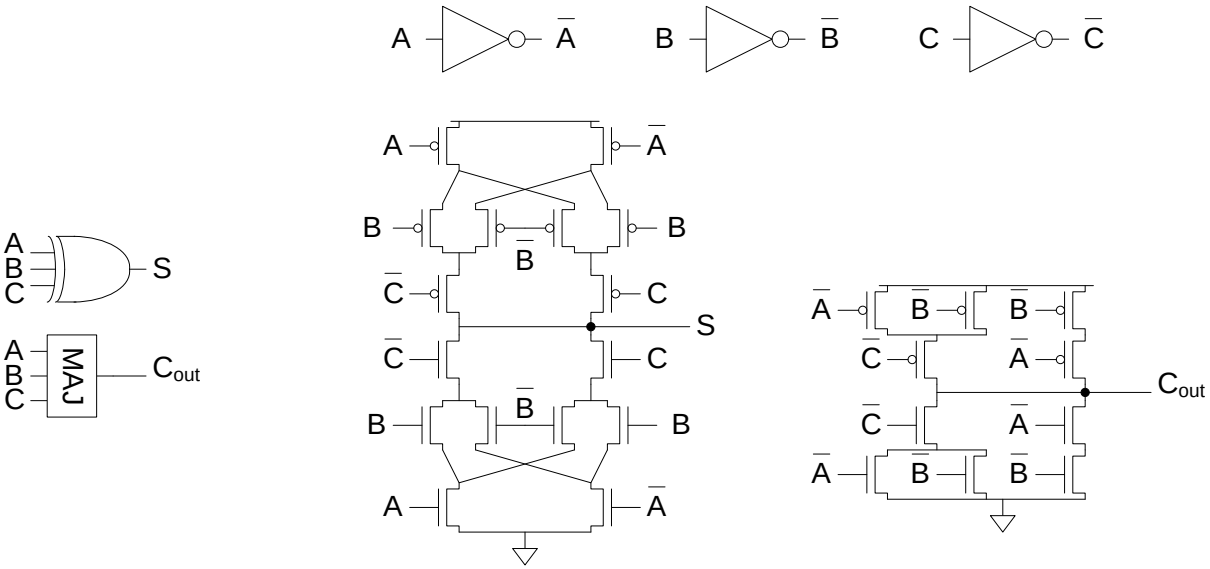
- For a full adder, define what happens to carries (in terms of A and B)
  - Generate: $C_{out} = 1$ independent of C
    - $G = A \bullet B$
  - Propagate: $C_{out} = C$
    - $P = A \oplus B$
  - Kill: $C_{out} = 0$ independent of C
    - $K = \sim A \bullet \sim B$

Full Adder Design I: 32 Transistors

- Brute force implementation from eqns:32 transistors

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

$$C_{out} = MAJ(A, B, C)$$



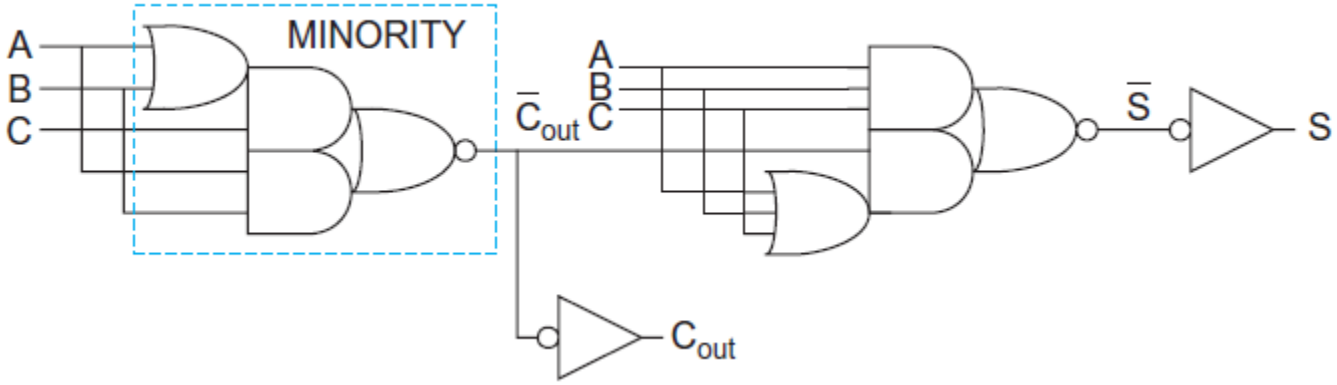
S = [A xor B xnor C]' Cout = [A' B' + C' (A' + B')]'

Full Adder Design II: 28 transistors

- Factor S in terms of C_{out}

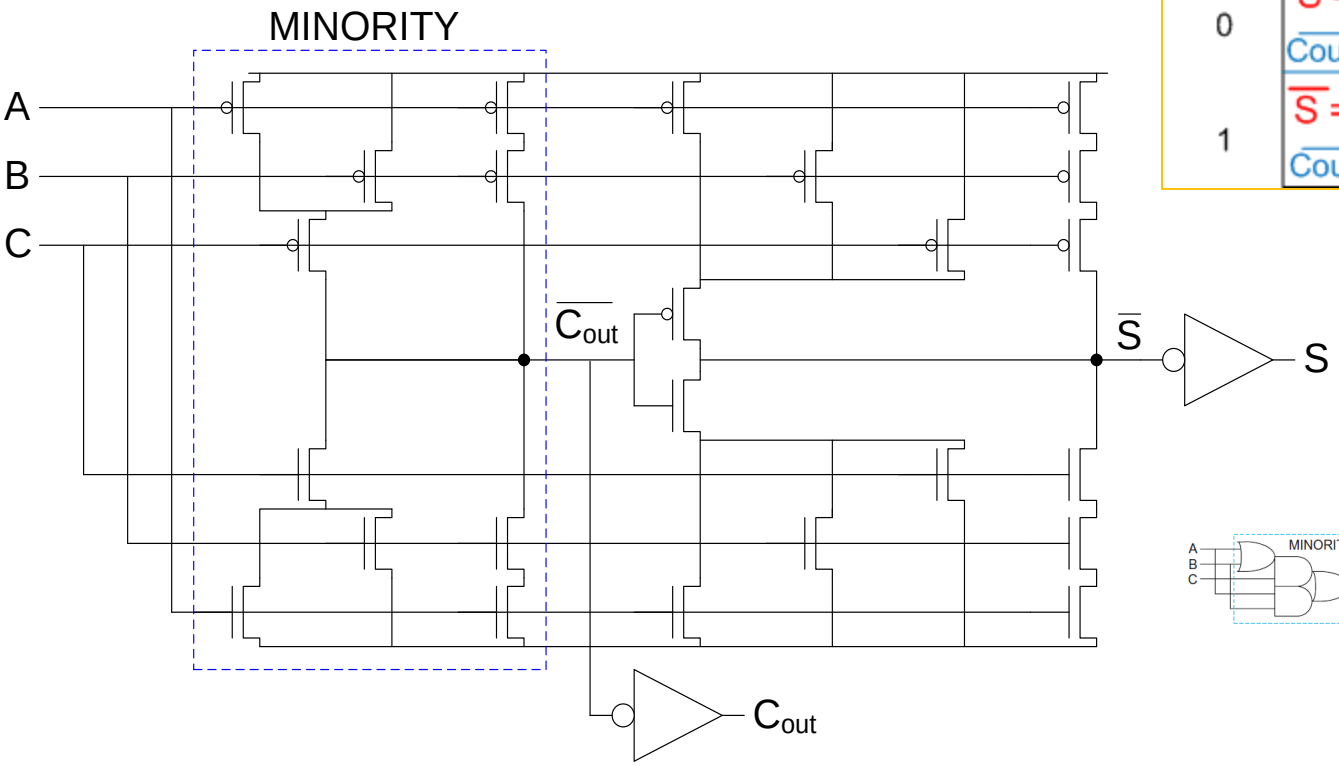
  $S = ABC + (A + B + C)(\sim C_{out})$

- Critical path is usually C to C_{out} in ripple adder, **28 transistors**, extra delay, good in CRA.

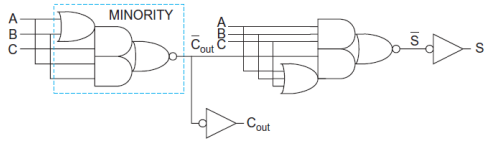


Full Adder Design II: 28 transistors

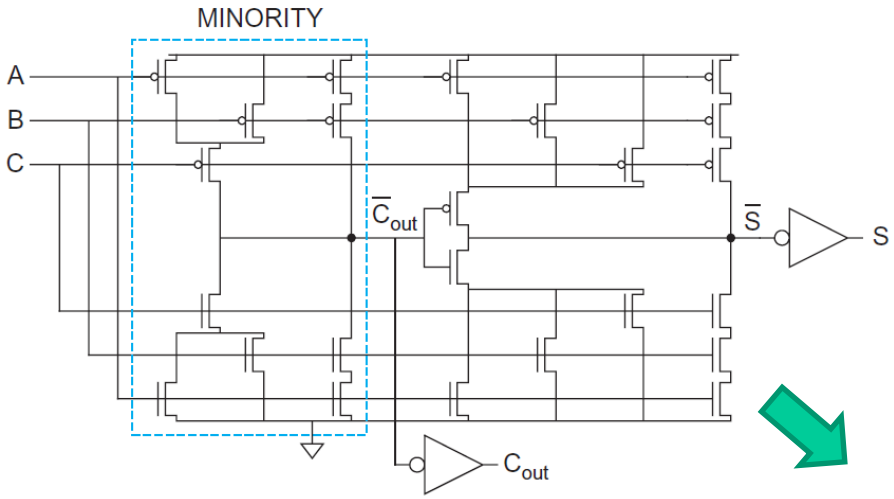
- Critical path is usually C to C_{out} in ripple adder
- Mirror adder: pMOS network is identical to nMOS network
  - Reduces number of transistors and simplify layout



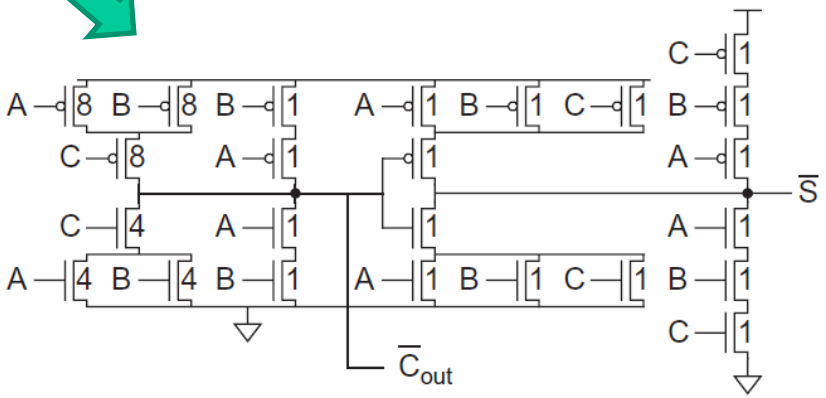
A \ B C	00	01	11	01
0	$\overline{S}=1$, $\overline{Cout}=1$	$\overline{S}=0$, $\overline{Cout}=1$	$\overline{S}=1$, $\overline{Cout}=0$	$\overline{S}=0$, $\overline{Cout}=1$
1	$\overline{S}=0$, $\overline{Cout}=1$	$\overline{S}=1$, $\overline{Cout}=0$	$\overline{S}=0$, $\overline{Cout}=0$	$\overline{S}=1$, $\overline{Cout}=0$



Full Adder Design II: 28 -> 24 trans.



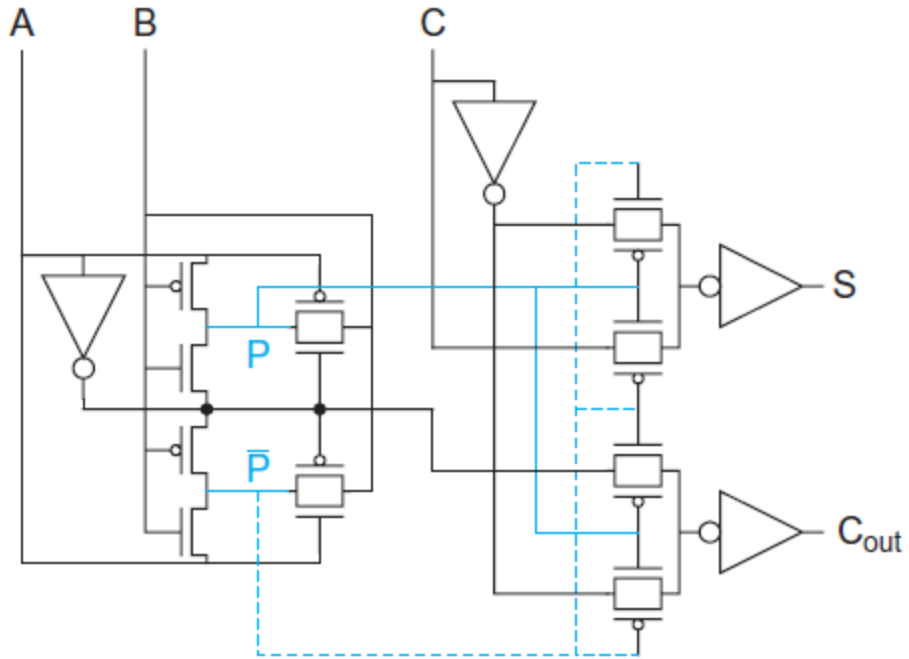
Hint: remove invertors, useful for some adder such as the one in Fig 11.11.

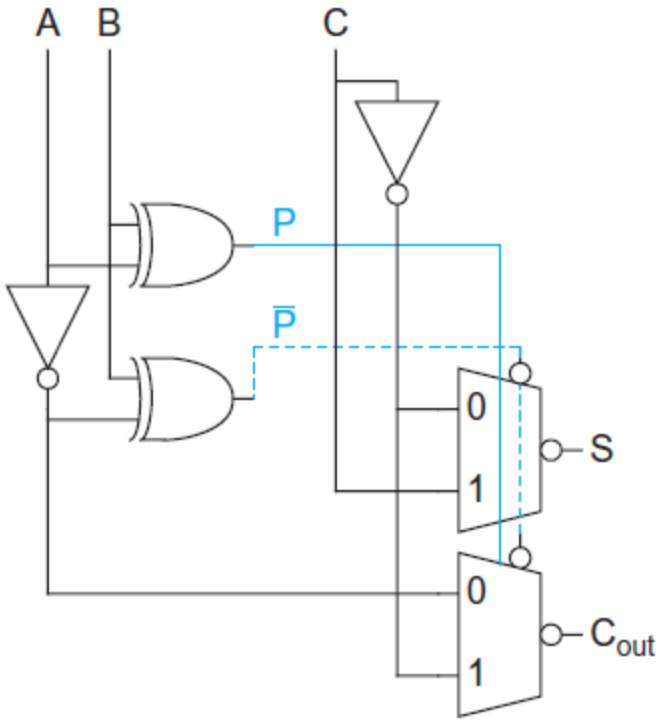


Full Adder Design III: 24 transistors

- Transmission gate to form multiplexers and XORs.
  - 24 transistors, equally delay (S and Cout).

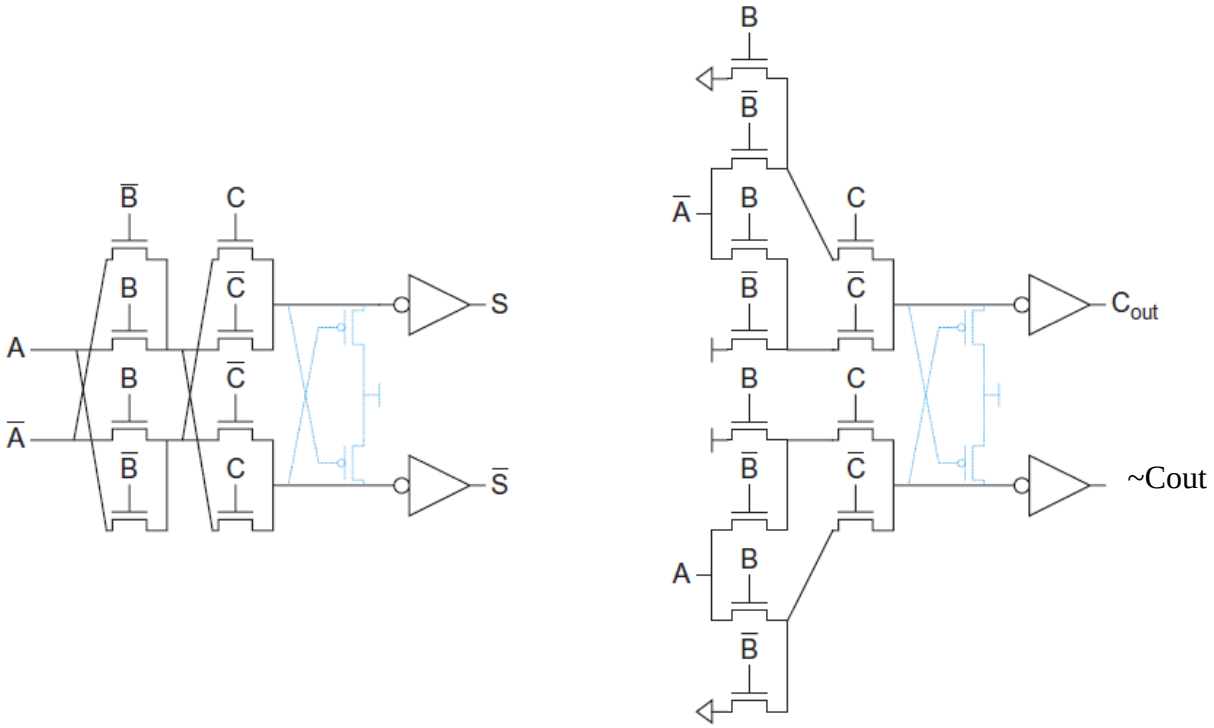
$$Cout = A \cdot P' + C \cdot P$$
$$S = C \cdot P' + C' \cdot P$$





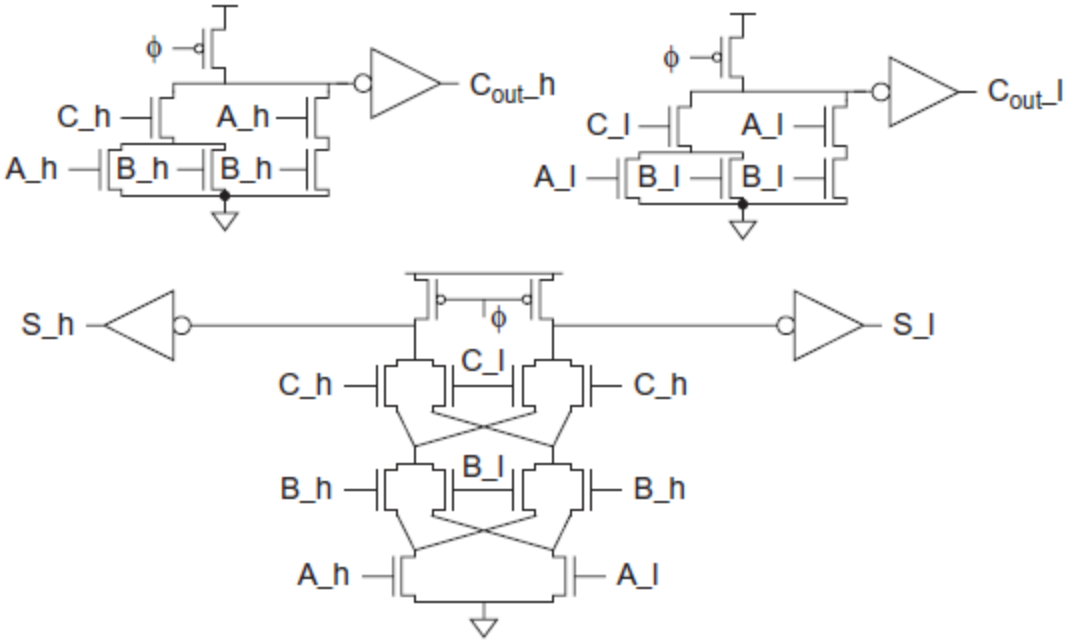
Full Adder Design IV: 40 trans.

- Complementary Pass Transistor Logic (CPL)
  - Slightly faster, similar in power, but more area



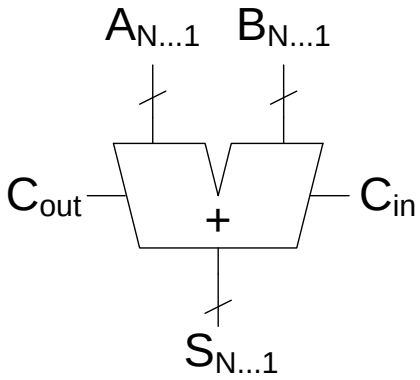
Full Adder Design V: 32 Trans.

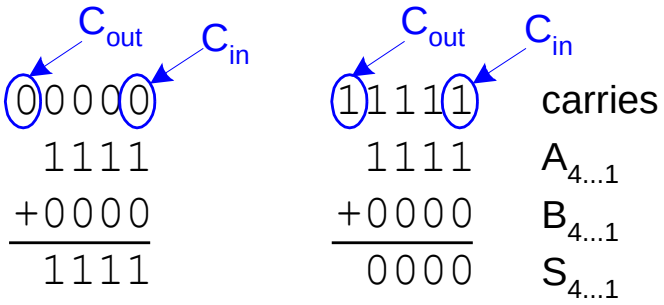
- Dual-rail domino (XOR/XNOR, MAJORITY/MINORITY)
  - Very fast, but large and power hungry
  - Used in very fast multipliers



11.2.2 Carry Propagate Adders

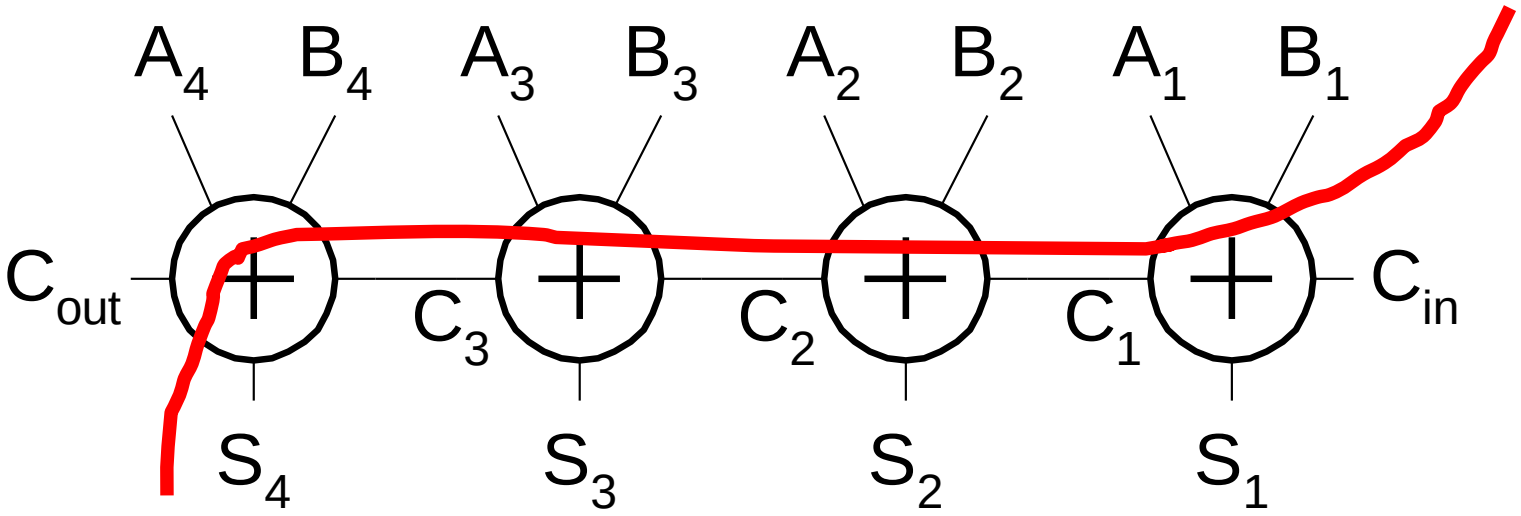
- N-bit adder called CPA
  - Each sum bit depends on all previous carries
  - How do we compute all these carries quickly?





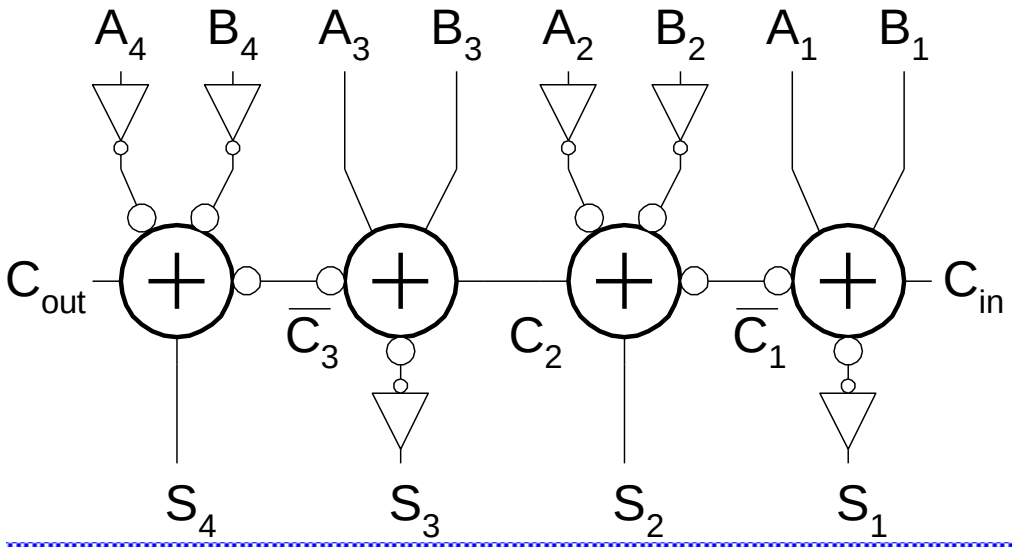
11.2.2.1 Carry-Ripple Adder

- Simplest design: cascade full adders
  - Critical path goes from C_{in} to C_{out}
  - Design full adder to have fast carry delay
- The delay of the adder is set by the time for the carries to ripple through the *N stages, so the* $t_{C->Cout}$ *delay should be* minimized.

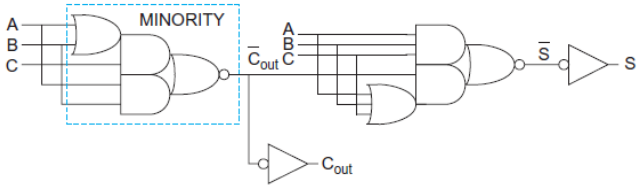


Omitting Inversions

- Critical path passes through majority gate.
- This delay can be reduced by omitting the inverters on the outputs, as shown below.
- Because addition is a *self-dual function* (i.e., the function of complementary inputs is the complement of the function), an inverting full adder receiving complementary inputs produces true outputs.



Recall:



Self-Dual Function

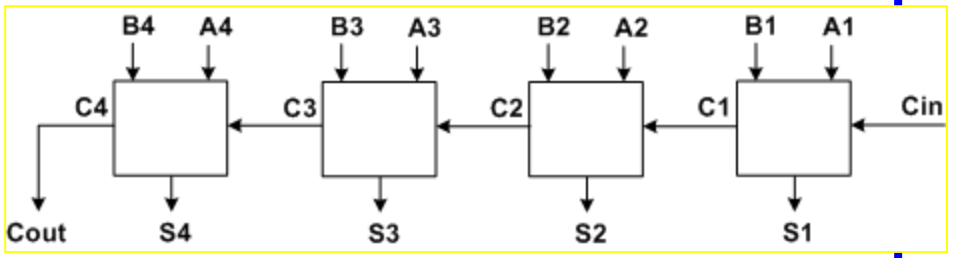
- The function of complementary inputs is the complement of the function),
- An inverting full adder receiving complementary inputs produces true outputs.

A	B	Cin	Cout	S	A'	B'	Cin'	Cout'	S'
0	0	0	0	0	1	1	1	1	1
0	0	1	0	1	1	1	0	1	0
0	1	0	0	1	1	0	1	1	0
0	1	1	1	0	1	0	0	0	1
1	0	0	0	1	0	1	1	1	0
1	0	1	1	0	0	1	0	0	1
1	1	0	1	0	0	0	1	0	1
1	1	1	1	1	0	0	0	0	0

1.2.2.2 Carry Generation and Propagation

- We introduce the group Propagate (P) and group generate (G).
- We can generalize G and P signals to describe whether a group spanning bits *i…j, inclusive, generate a carry (G=1) or propagate a carry (P=1).*
- A group of bits:
  - generates a carry if its carry-out is true independent of the carry-in;
  - propagates a carry if its carry-out is true when there is a carry-in.

Recall Ps&Gs: 4-Bit Adder Example



- $C_{out} = G_{4:0}$
- $C_{in} = C_0 = G_0 = G_{0:0}$
- $C_1 = G_{1:0} = G_1 + P_1 C_0$
- $C_2 = G_{2:0} = G_2 + P_2 C_1$
  $= G_2 + P_2 G_1 + P_2 P_1 C_0$
- $C_3 = G_{3:0} = G_3 + P_3 C_2$
  $= G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 C_0$
- $C_4 = G_{4:0} = G_4 + P_4 C_3$
  $= G_4 + P_4 G_3 + P_4 P_3 G_2 + P_4 P_3 P_2 G_1 + P_4 P_3 P_2 P_1 C_0$

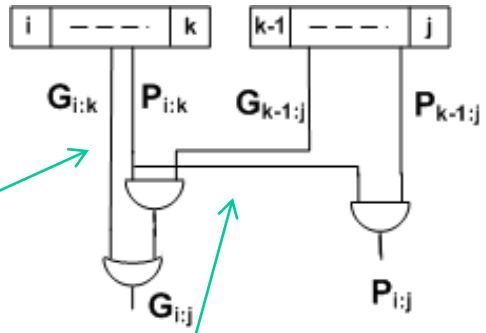
We need to simplify this MESS!!

Carry Generation and Propagation

- For group of signals spanning i..j, and *i≥k>j. The* Generate and propagate signals:

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

$$P_{i:j} = P_{i:k} \cdot P_{k-1:j}$$



  - A group generates a carry if
    - the upper (more significant) generates a carry **OR**
    - the lower portion generates and the upper portion propagates that carry.
  - The group propagates a carry if both the upper and lower portions propagate the carry

- Base case: group size = 1, which means i==j:

$$G_{i:i} \equiv G_i = A_i \cdot B_i$$
$$P_{i:i} \equiv P_i = A_i \oplus B_i$$

$$G_{0:0} \equiv G_0 = C_{in}$$
$$P_{0:0} \equiv P_0 = 0$$

- Sum/Carry: $S_i = P_i \oplus G_{i-1:0}$ $C_{i-1} = G_{i-1:0}$

Steps of Addition Using PG Logic

- Addition can be reduced to a three-step process:
  1. Computing bitwise generate and propagate signals using Eqs:

$$G_{i:i} \equiv G_i = A_i \cdot B_i \qquad G_{0:0} \equiv G_0 = C_{in}$$
$$P_{i:i} \equiv P_i = A_i \oplus B_i \qquad P_{0:0} \equiv P_0 = 0$$

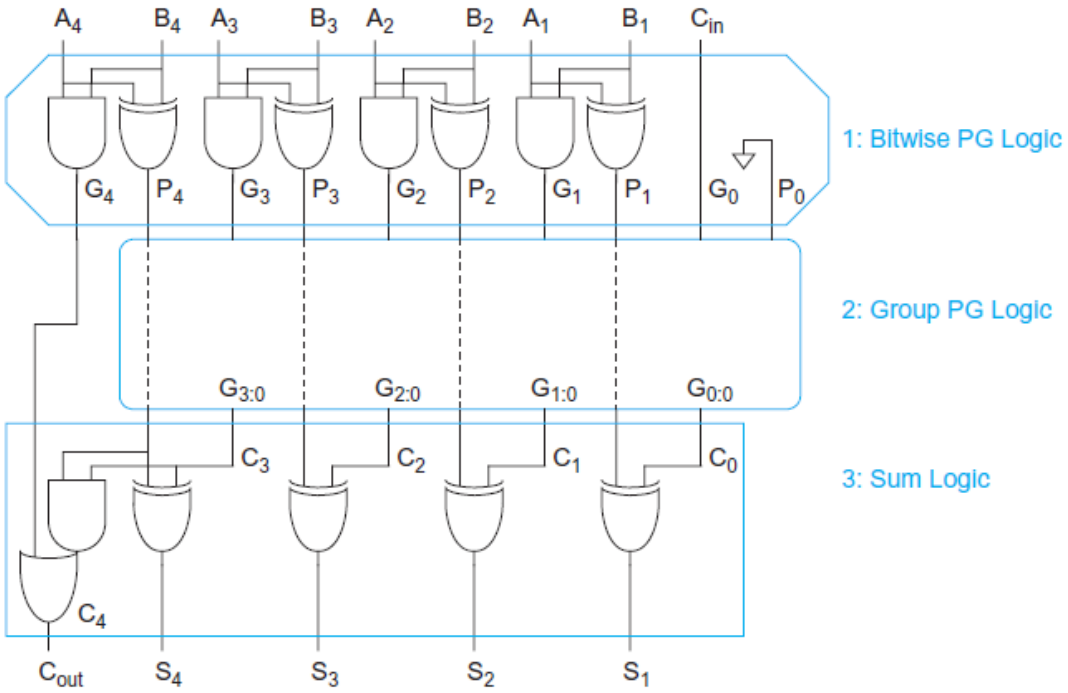
  2. Combining PG signals to determine group generates G_{i-1}*:0 for all* $N \geq i \geq 1$ *using* EQs:

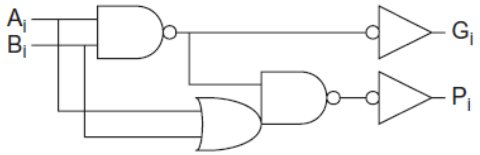
$$G_{i:j} = G_{i:k} + P_{i:k} \cdot G_{k-1:j}$$
$$P_{i:j} = P_{i:k} \cdot P_{k-1:j}$$

  3. Calculating the sums using EQ:

$$S_i = P_i \oplus G_{i-1:0}$$

Steps of Addition Using PG Logic



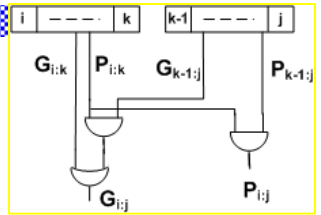


The first and third steps are routine, so most of the attention in the remainder of this section is devoted to alternatives for the group PG logic with different trade-offs between speed, area, and complexity. Some of the hardware can be shared in the bitwise PG logic, as shown

Higher Valency

- We discussed the equations for $G_{i:j}$ and $P_{i:j}$

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



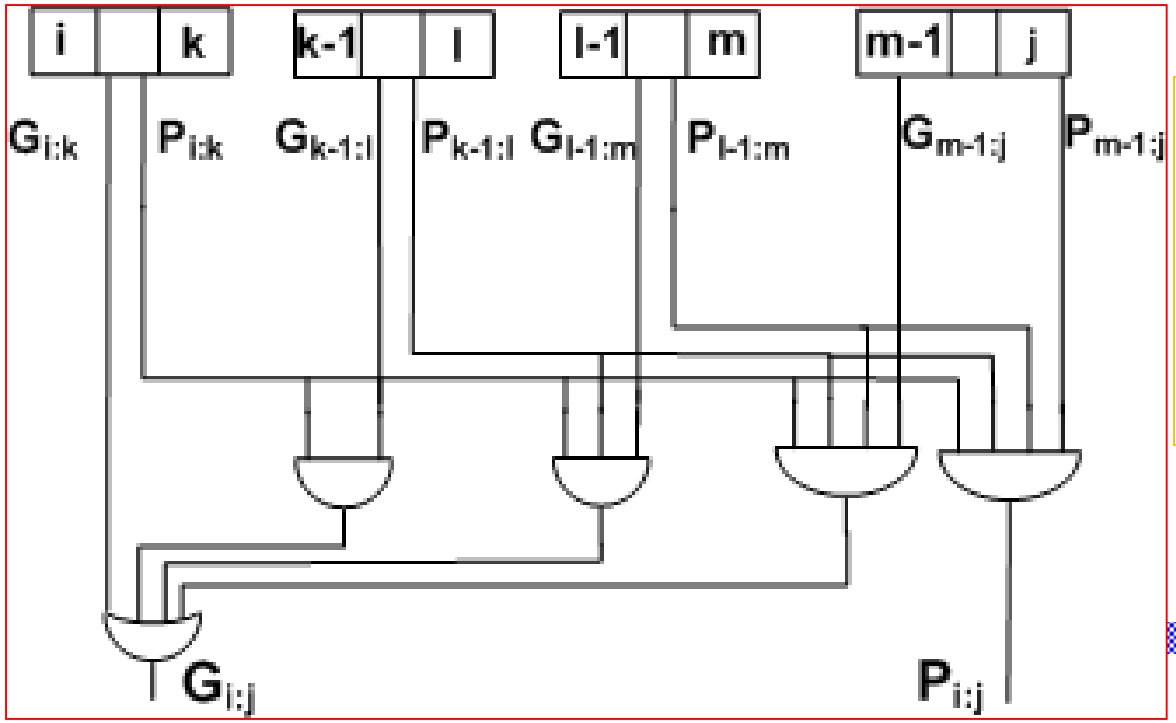
- The above equations define *valency-2 (also called radix-2) group PG logic because it combines* pairs of smaller groups.
- It is also possible to define higher-valency group logic to use fewer stages of more complex gates, as shown in the following equations:

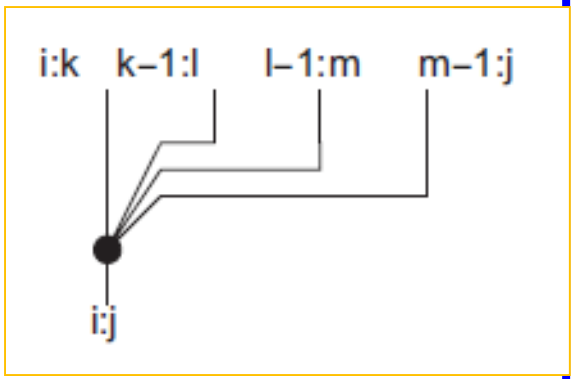
$$\left.\begin{aligned} G_{i:j} &= G_{i:k} + P_{i:k} \cdot G_{k-1:l} + P_{i:k} \cdot P_{k-1:l} \cdot G_{l-1:m} + P_{i:k} \cdot P_{k-1:l} \cdot P_{l-1:m} \cdot G_{m-1:j} \\ &= G_{i:k} + P_{i:k}\left(G_{k-1:l} + P_{k-1:l}\left(G_{l-1:m} + P_{l-1:m} G_{m-1:j}\right)\right) \\ P_{i:j} &= P_{i:k} \cdot P_{k-1:l} \cdot P_{l-1:m} \cdot P_{m-1:j} \end{aligned}\right\} \quad (i \geq k > l > m > j)$$

- For example, in valency-4 group logic, a group propagates the carry if all four portions propagate. A group generates a carry if the upper portion generates, the second portion generates and the upper propagates, the third generates and the upper two propagate, or the lower generates and the upper three propagate.

Valency-4

$$
\begin{aligned}
G_{i:j} &= G_{i:k} + P_{i:k} \cdot G_{k-1:l} + P_{i:k} \cdot P_{k-1:l} \cdot G_{l-1:m} + P_{i:k} \cdot P_{k-1:l} \cdot P_{l-1:m} \cdot G_{m-1:j} \\
&= G_{i:k} + P_{i:k}\left(G_{k-1:l} + P_{k-1:l}\left(G_{l-1:m} + P_{l-1:m} G_{m-1:j}\right)\right) \\
P_{i:j} &= P_{i:k} \cdot P_{k-1:l} \cdot P_{l-1:m} \cdot P_{m-1:j}
\end{aligned}
\qquad \left(i \geq k > l > m > j\right)
$$





11.2.2.3 PG Carry Ripple Addition

- The critical path of the carry-ripple adder passes from carry-in to carry-out along the carry chain majority gates.
- As the *P and G signals will have* already stabilized by the time the carry arrives, we can use them to simplify the majority function into an AND-OR gate.

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

- Because $C_i = G_{i:0}$, *carry-ripple addition can now be viewed as the extreme case of* group PG logic in which a 1-bit group is combined with an *i-bit group to form an (i+1)*-bit group.

$$G_{i:0} = G_i + P_i \cdot G_{i-1:0}$$

- In this extreme, the group propagate signals are never used and need not be computed.

4-bit Carry-Ripple Using PG logic

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

where t_{pg} *is the delay of the 1-bit propagate/generate gates,*
t_{AO} *is the delay of the ANDOR* gate in the gray cell,
and t_{xor} *is the delay of the final sum XOR.*

For N=4

$t_{ripple}=t_{pg}+ 3\times t_{AO}+t_{xor}$

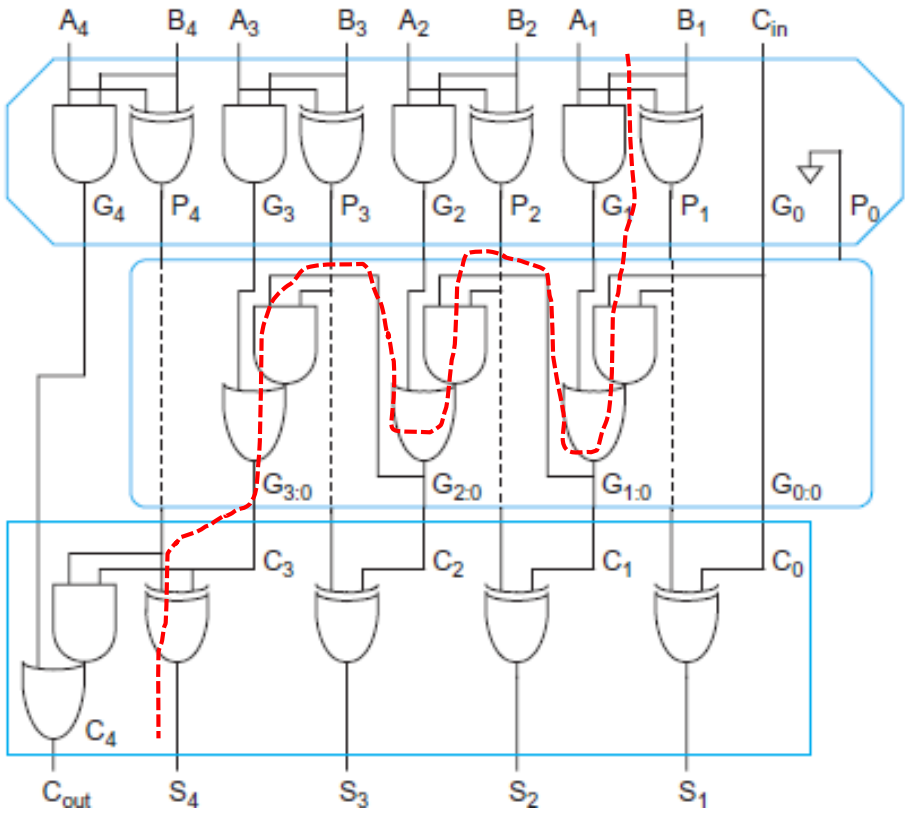
The ripple counter is a special case of CLA adder (covered next). The CLA adder delay:

$t_{cla}=t_{pg}+t_{pg(n)}+[(n-1)+(k-1)]t_{AO}+t_{xor}$

For ripple adder k=1,

*N=4, k=1, N= n*k, so: n=4*

$t_{pg(k=1)}$ *is not used and never computed.*

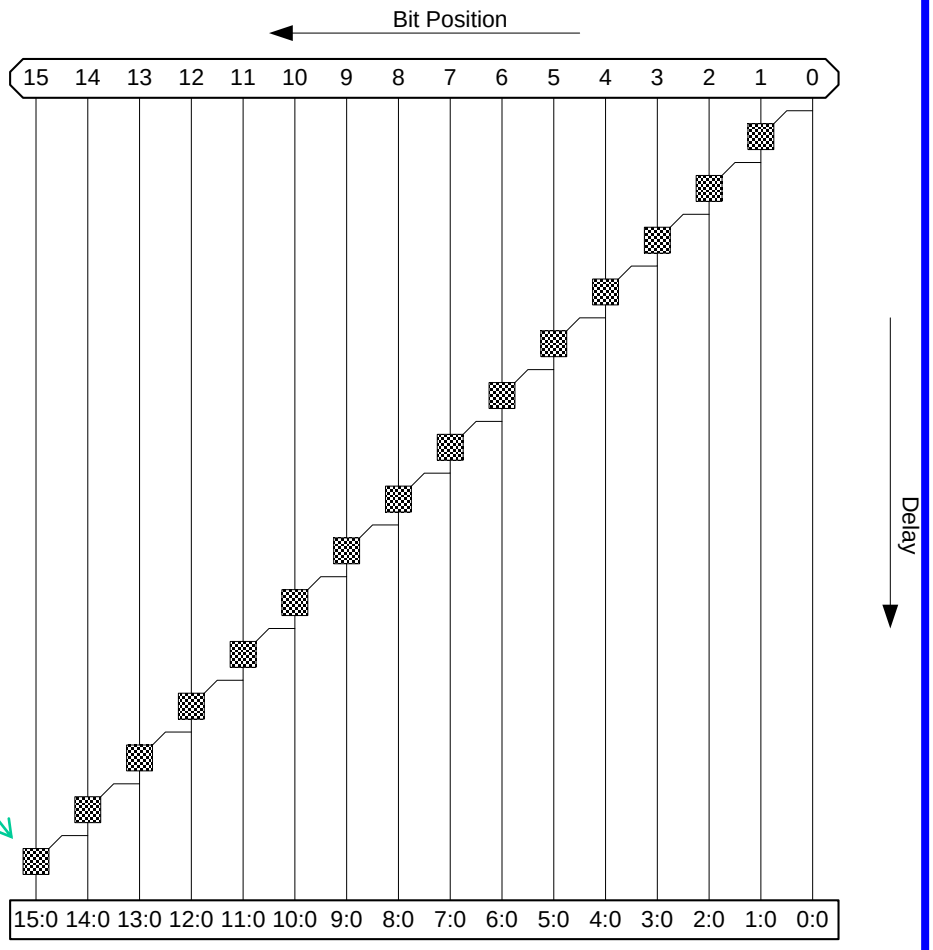


16-bit Carry-Ripple using PG Logic

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

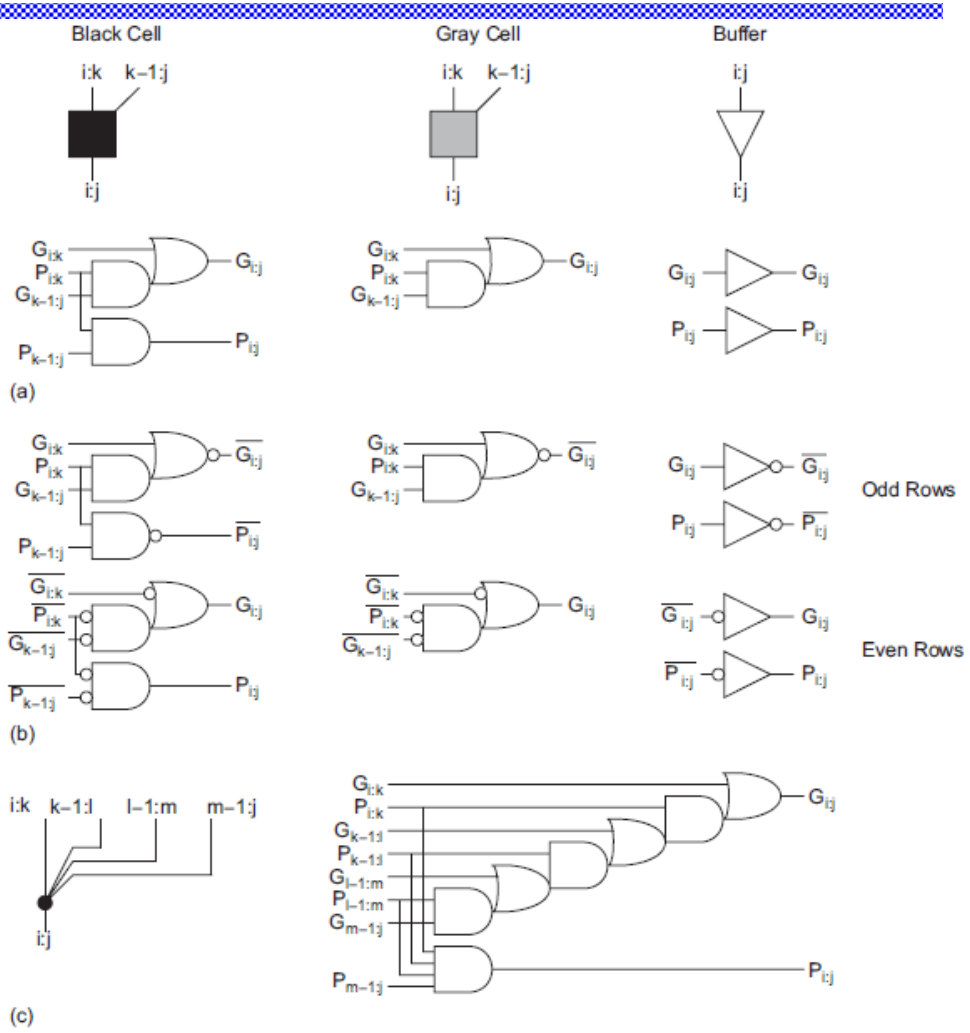
$$t_{ripple} = t_{pg} + 15 \times t_{AO} + t_{xor}$$

AND-OR gates in the PG network.



Groups PG Cells

- **Black cells** contain the group generate and propagate logic (an AND-OR gate and an AND gate) defined in EQ (11.4).
- **Gray cells** containing only the group generate logic are used at the final cell position in each column because only the group generate signal is required to compute the sums.
- **Buffers** can be used to minimize the load on critical paths.
- Each line represents a *bundle of the group generate* and propagate signals (propagate signals are omitted after gray cells).

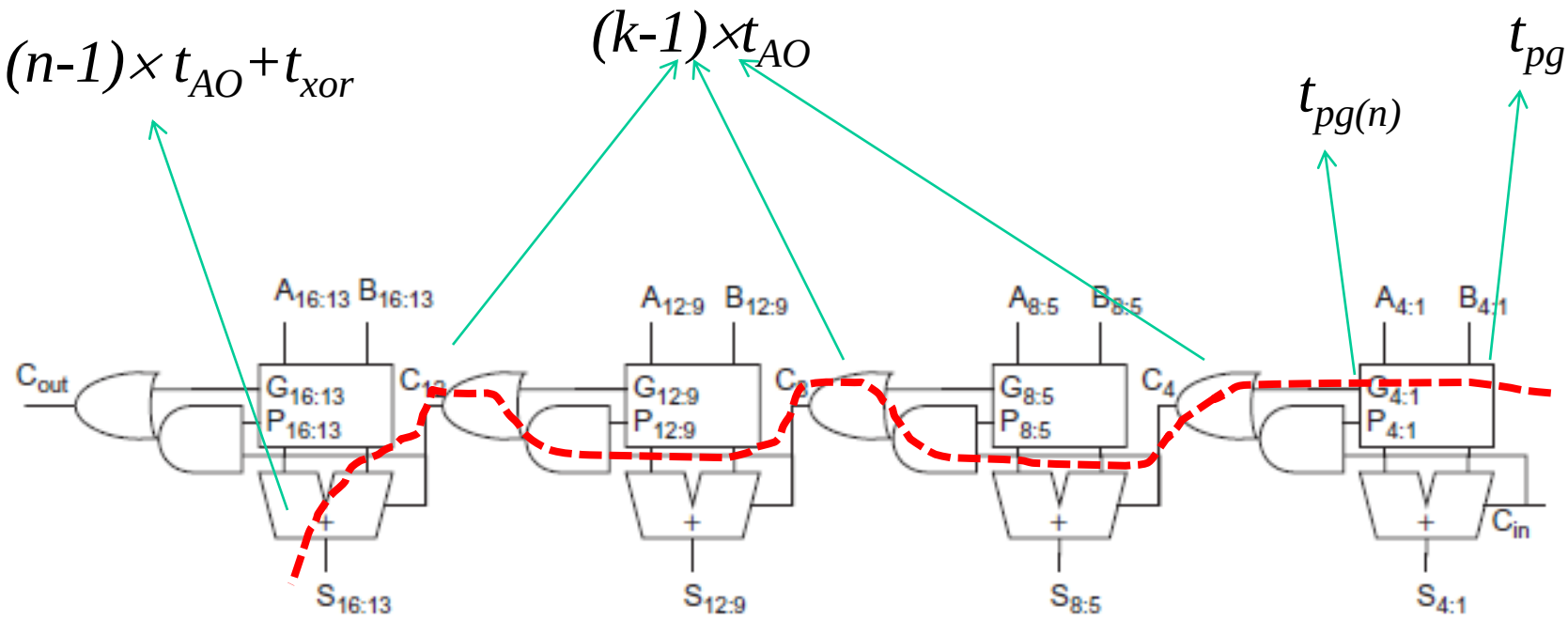


11.2.2.6 Carry-Lookahead Adder

- Computes group generate signals and group propagate signals to avoid waiting for a ripple
- Uses valency-4 black cells to compute 4-bit group PG signals.
- In general the delay for CLA using ***k*** *groups of* ***n*** *bits:*

  $t_{cla} = t_{pg} + t_{pg(n)} + [(n-1)+(k-1)]t_{AO} + t_{xor}$

  where $t_{pg(n)}$ *is the delay of the AND-OR-AND-OR-…-AND-OR gate computing the* valency-*n generate signal.*



CLA: PG logic

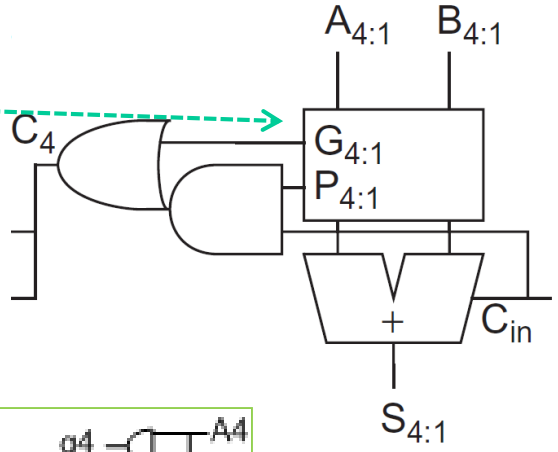
- The PG logic is shown below:

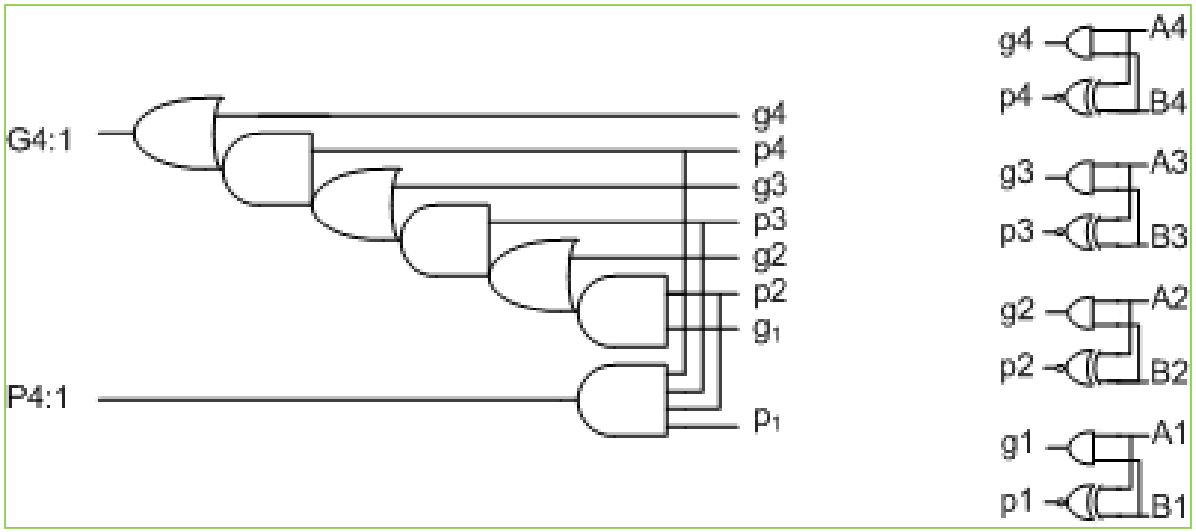
$$P_{4:1} = p4.\ p3\ .\ p2.\ p1$$

$$G_{4:1} = g4 + p4\ (\ g3 + p3\ (\ g2 + p2\ g1))$$

$$= g4 + p4\ .\ g3\ + p4\ p3\ g2$$

$$+ p4\ .\ p3\ .\ p2\ .\ g1$$





11.2.2.8 Tree Adders

- For wide adders (roughly, *N > 16 bits), the delay of carry-lookahead* adders becomes dominated by the delay of passing the carry through the lookahead stages. This delay can be reduced by looking ahead across the lookahead blocks.
- In general, you can construct a multilevel tree of look-ahead structures to achieve delay that grows with **log *N***.
- Such adders are variously referred to as *tree adders, logarithmic adders, multilevel-lookahead adders, parallel-prefix adders, or simply lookahead adders.*

11.2.2.8 Tree Adders

- There are many ways to build the lookahead tree that offer trade-offs among
  - the number of stages of logic,
  - the number of logic gates,
  - the maximum fanout on each gate,
  - the amount of wiring between stages.
- The delay of tree adders is approximately

$$t_{\text{tree}} \approx t_{pg} + \lceil \log_2 N \rceil t_{AO} + t_{\text{xor}}$$

Tree Adder Examples

- Reading the examples in the textbook.
  - Brent-Kung
  - Sklansky
  - Kogge-Stone
  - Han-Carlson
  - Knowles [2,1,1,1]
  - Ladner-Fischer