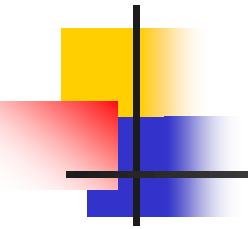
# Chapter 8
# Datapath Subsystems

闕志達
台灣大學電機系



# Datapath

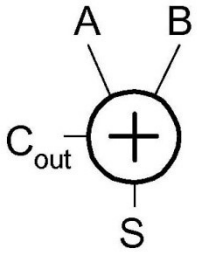
- Digital systems consist mainly of
  - Datapath operators
  - Memory elements
  - Control structures
  - Special cells – I/O, power distribution, clock generation/distribution
- Datapath operations are arithmetic and logic operations on the digital signals:
  - Adders
  - Comparators
  - Shifters
  - Counters
  - Multipliers

# Single-Bit Addition

## Half Adder

$$S = A \oplus B$$

$$C_{out} = A \bullet B$$

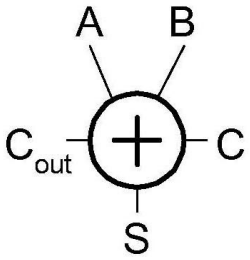



| A | B | $C_{out}$ | S |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

## Full Adder

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

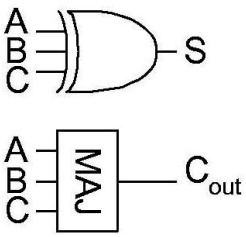
$$C_{out} = MAJ(A,B,C)$$

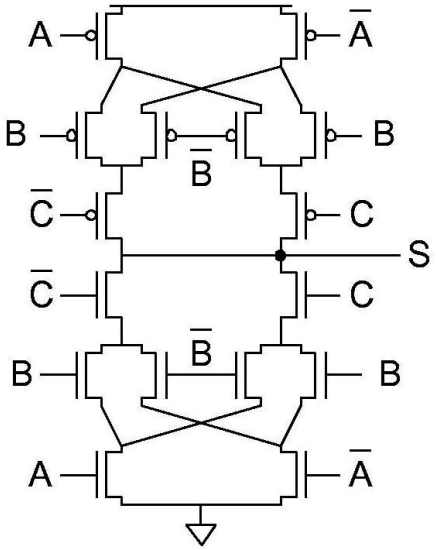


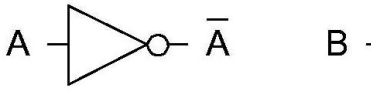

| A | B | C | $C_{out}$ | S |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

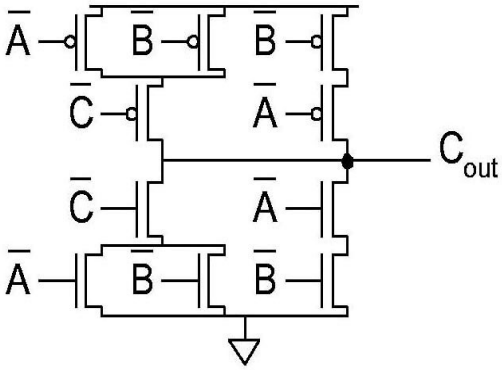
# Full Adder

- $S = A \oplus B \oplus C$
- $C_{out} = MAJ(A, B, C) = AB + BC + CA = AB + C(A+B)$



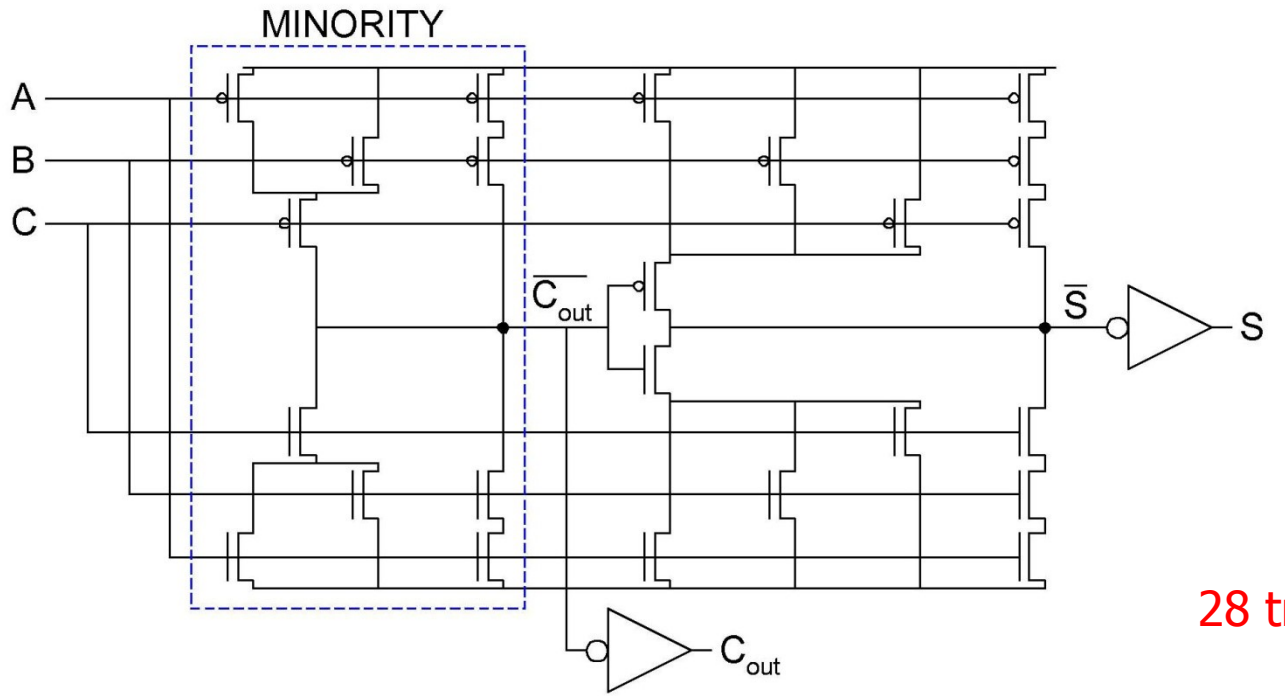




32 transistors

# Full Adder II

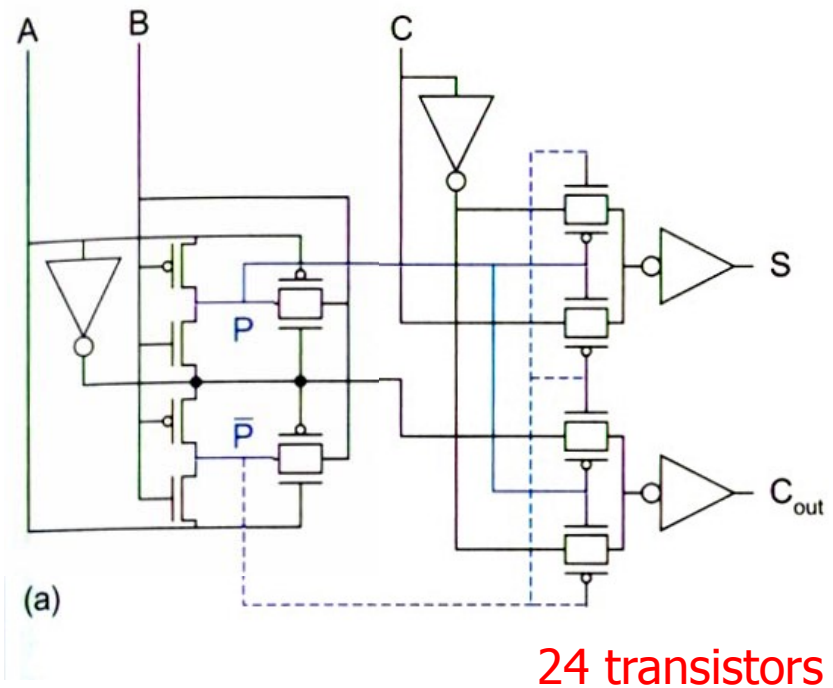
- Factor S in terms of $C_{out}$
- $S = ABC + (A + B + C)(\sim C_{out})$
- Critical path is usually C to $C_{out}$ in ripple adder

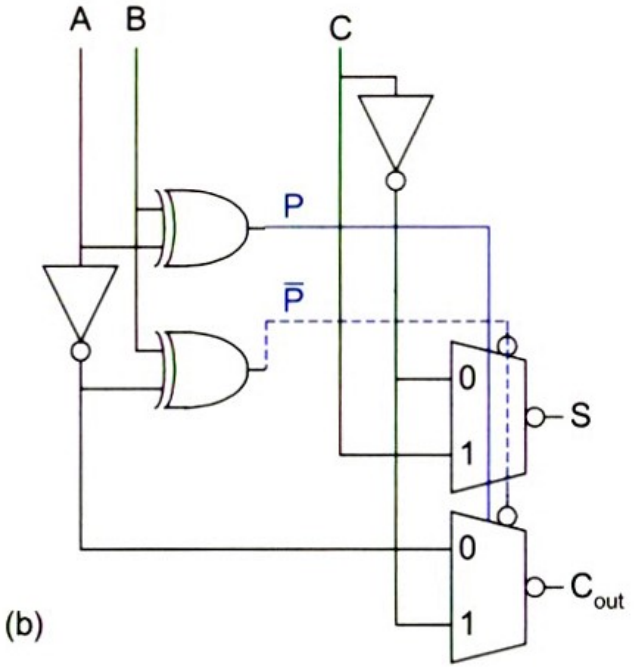



28 transistors

# Transmission-Gate Full Adder

- A six-transistor transmission-gate XOR gate.
- This adder has equal SUM and CARRY delay times. The number of transistors can be further reduced if pass transistors instead of transmission gates are used and the output buffers removed.

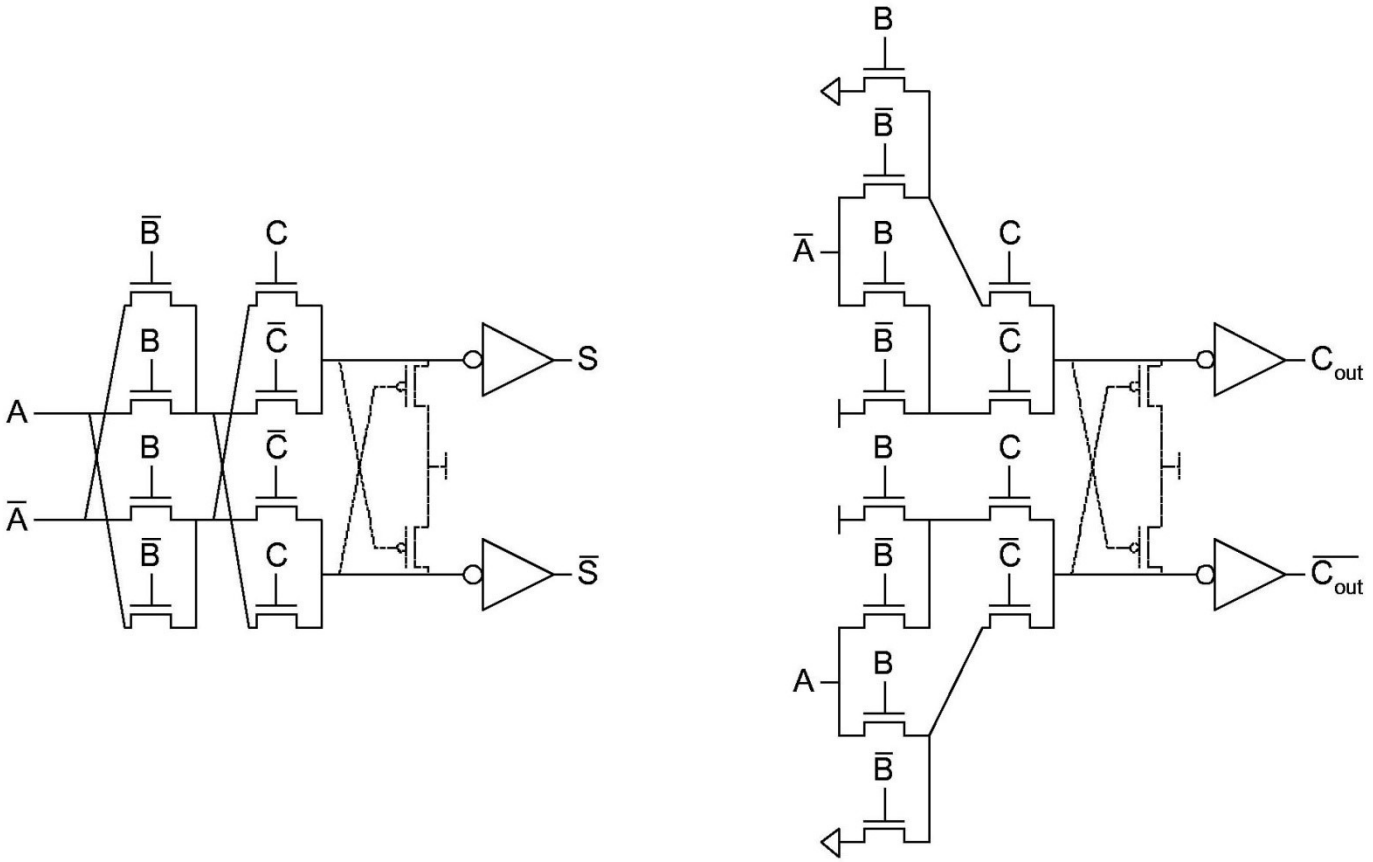



24 transistors

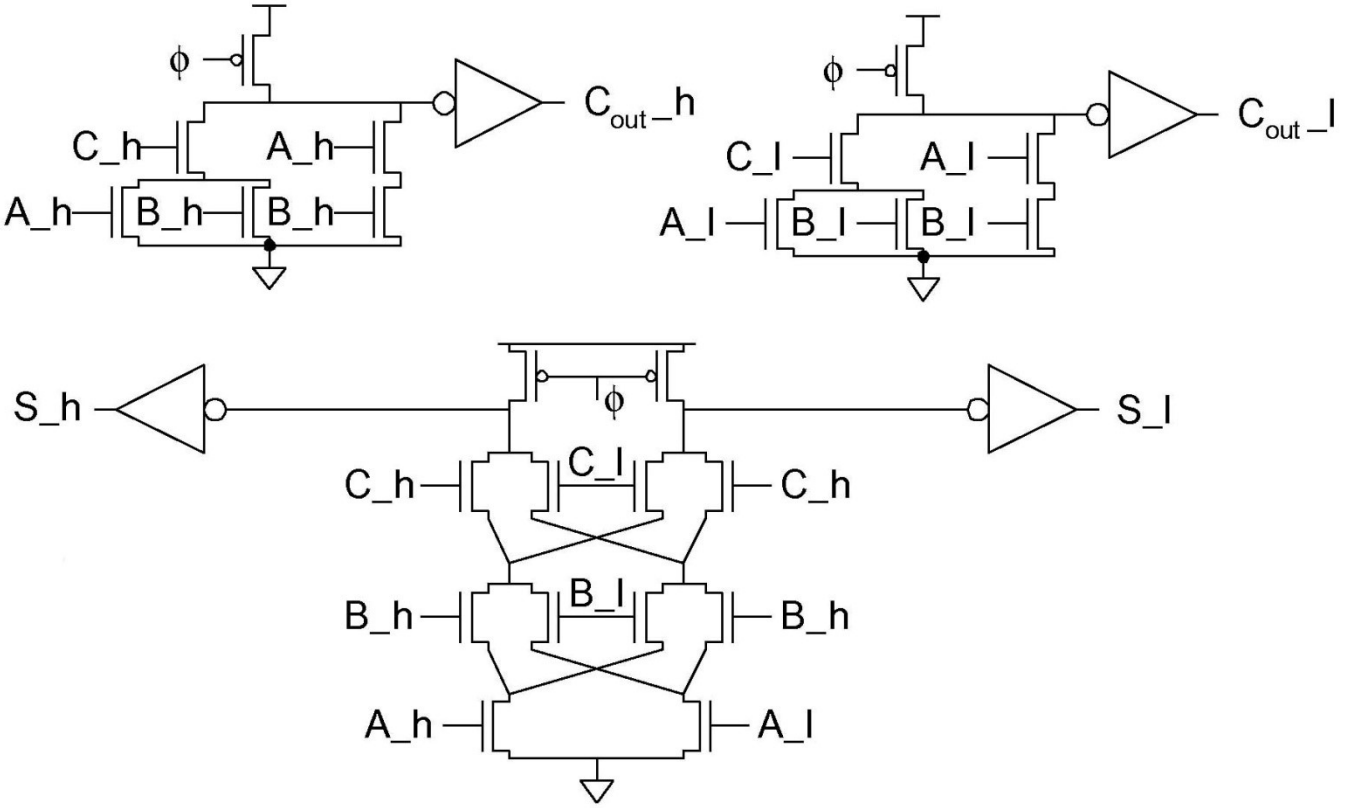
# CPL Full Adder

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



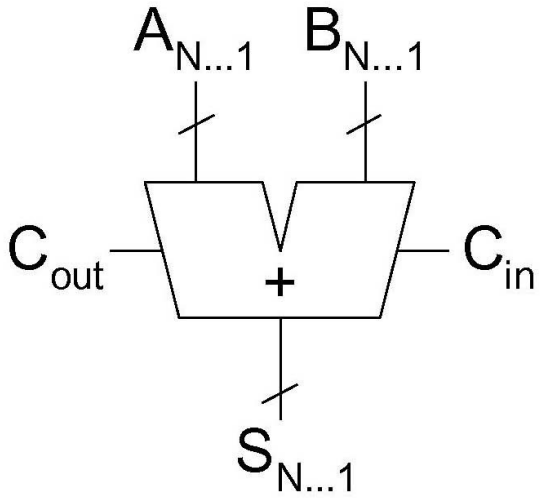
# Dual-Rail Domino Full Adder

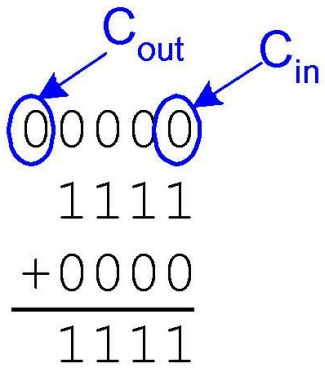
- Dual-rail domino
  - Very fast, but large and power hungry
  - Used in very fast multipliers

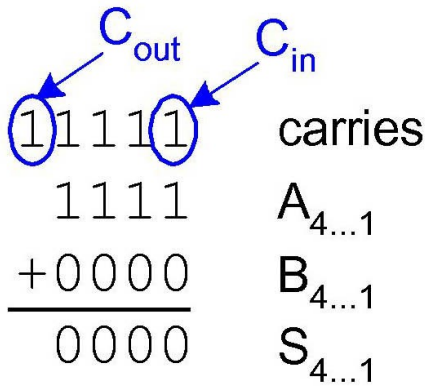


# N-bit Adders

- Each sum bit depends on all previous carry signals
- Simplest design is to cascade N FAs, but very slow!
- Can we compute all these carries quickly?

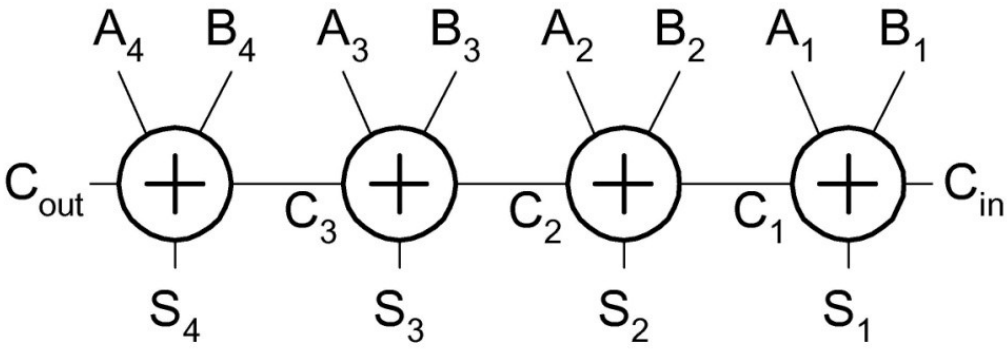


# Carry-Ripple Adder

- Simplest design: cascade full adders
  - Critical path goes from Cin to Cout
  - Design full adder to have fast carry delay



# Propagate/Generate/Kill

- For a full adder, define three cases that can happen to carry (Cout).
- Generate: Cout = 1 independent of C
  - $G = A \bullet B$
- Propagate: Cout = C
  - $P = A \oplus B$
- Kill: Cout = 0 independent of C
  - $K = \sim A \bullet \sim B$

- Note that
  - $Cout = A \bullet B + (A+B) \bullet C = G + P \bullet C$
  - $S = A \oplus B \oplus C = P \oplus C$

# Group Generate/Propagate

- Often need to compute G and P for a group of bits, not for only one bit.
- Generate and propagate for groups spanning i:j

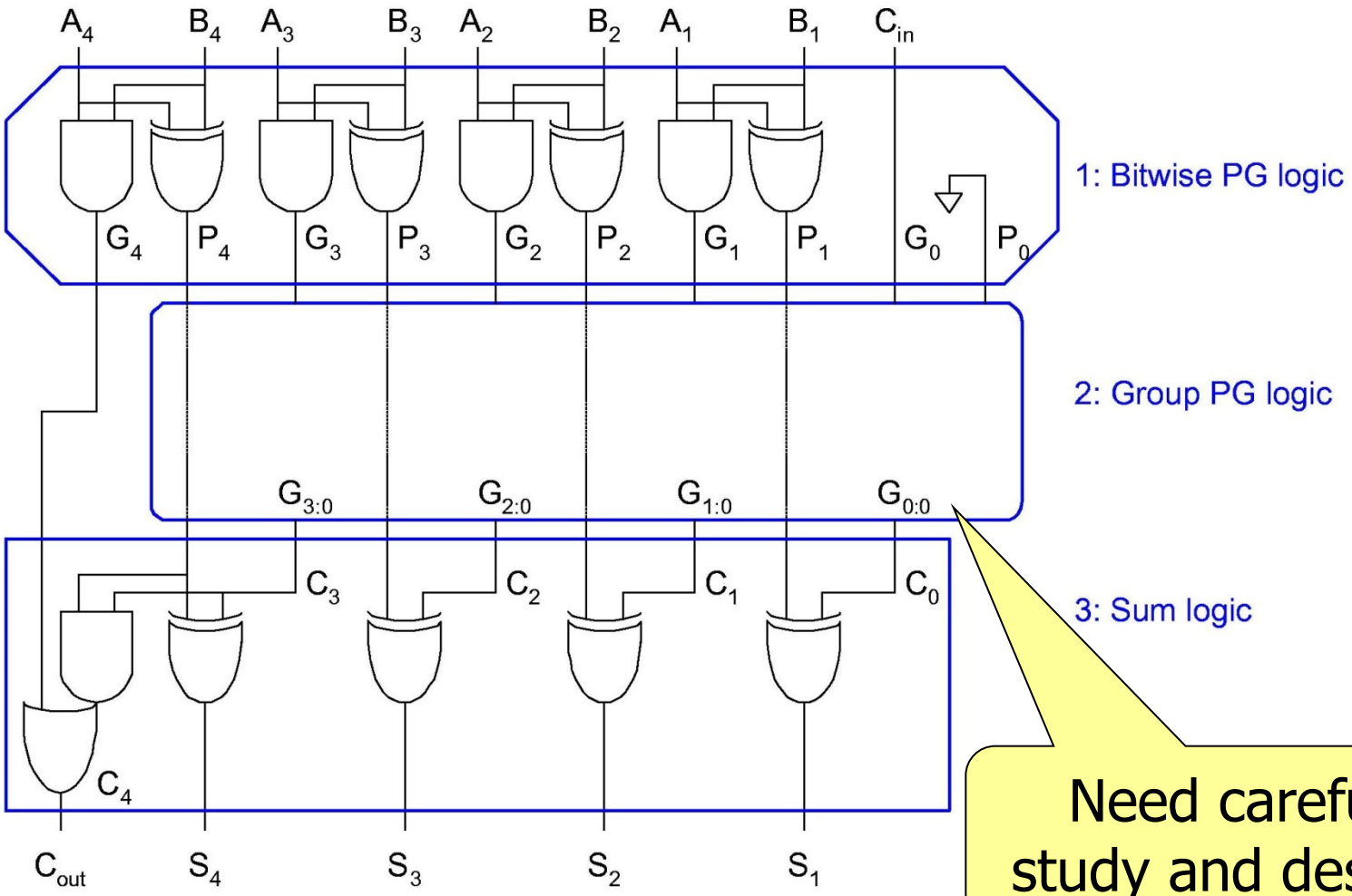
$$G_{i:j} = G_{i:k} + P_{i:k} \bullet G_{k-1:j}$$
$$P_{i:j} = P_{i:k} \bullet P_{k-1:j}$$

- Single-bit case

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

- Note that $G_{i-1:0} = C_{i-1}$ so $S_i = P_i \oplus G_{i-1:0}$

# Group PG Logic in N-bit Adder

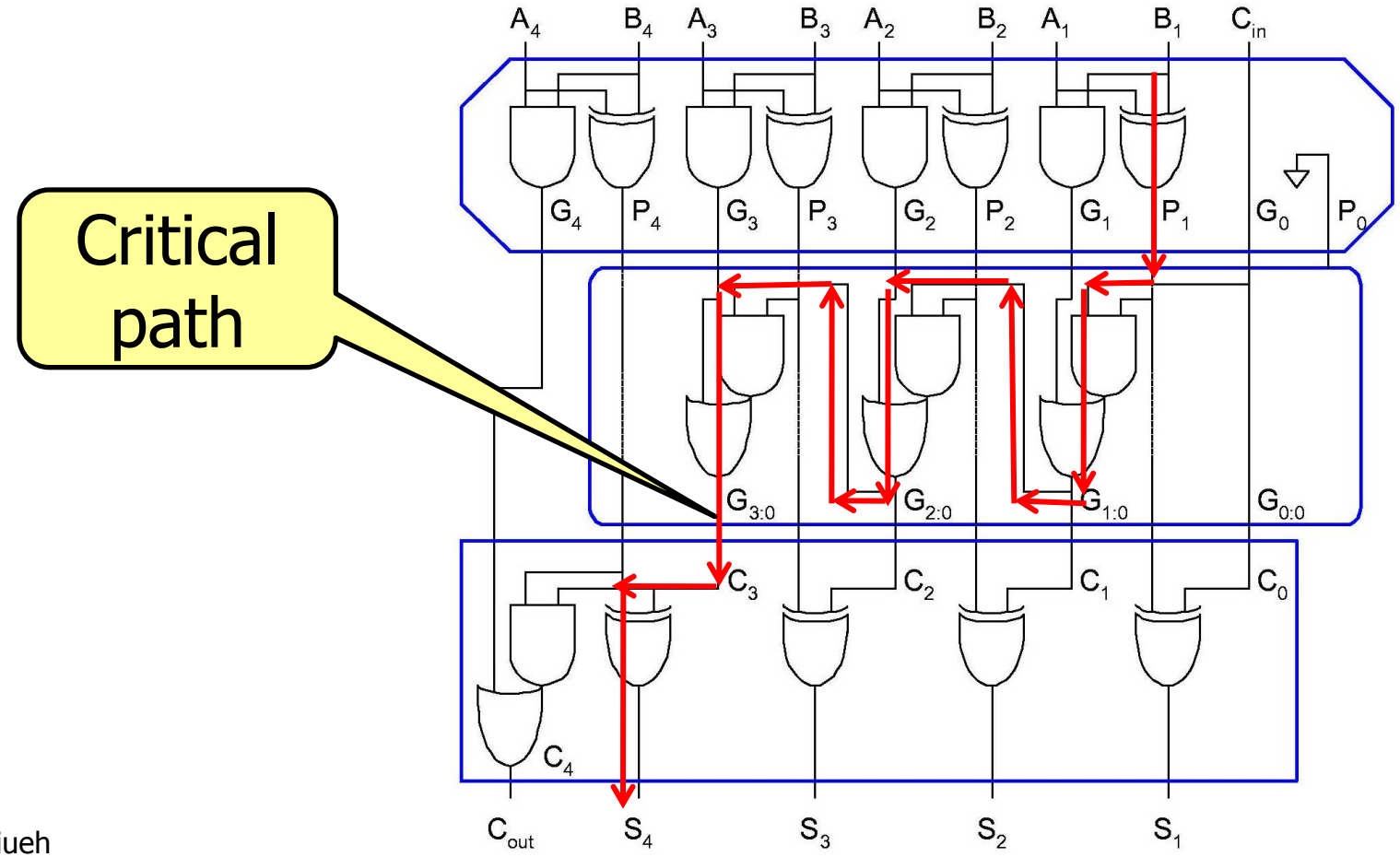


# Group PG Logic Design

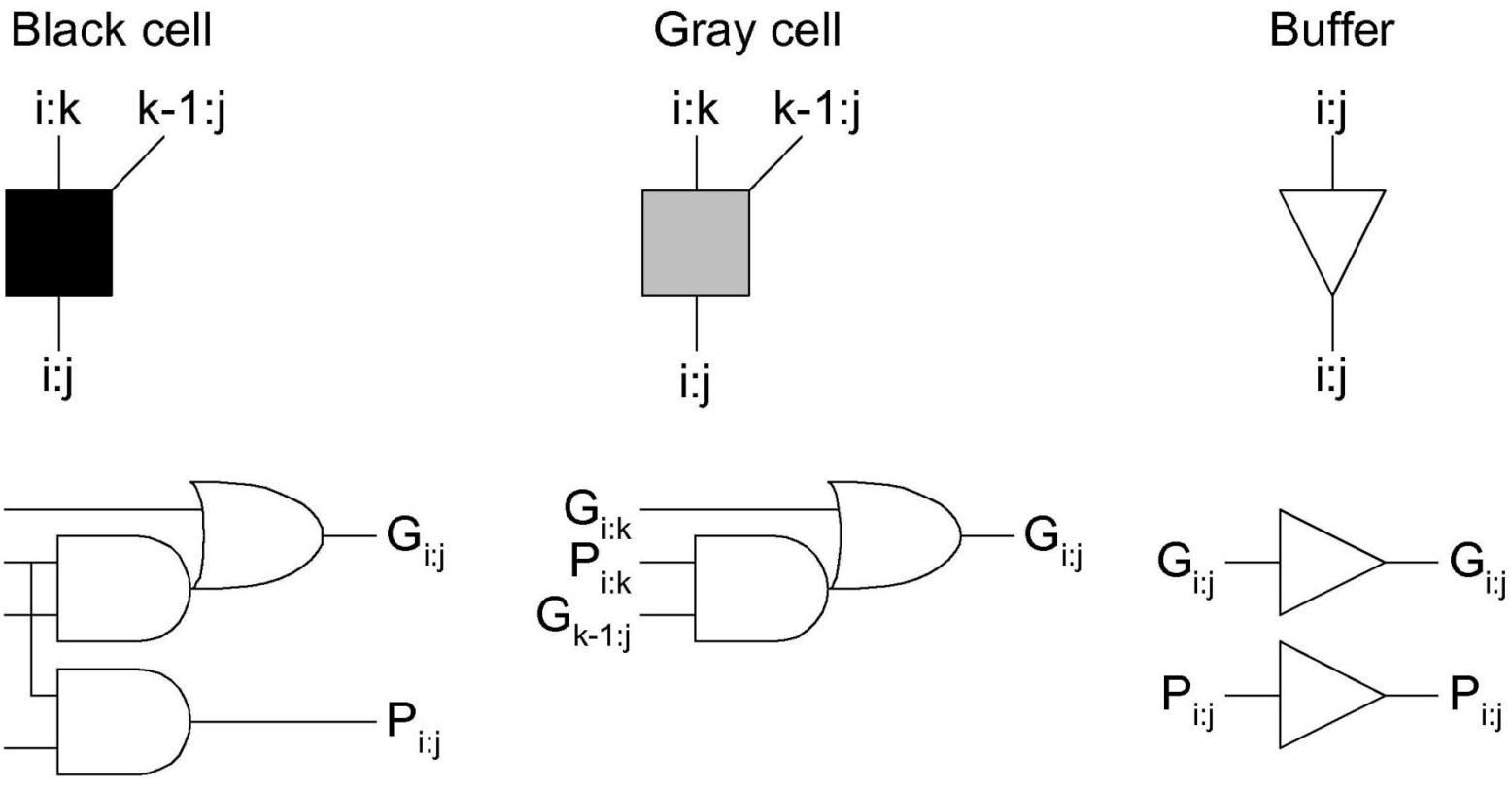
- In the following we will introduce many designs for group PG logic.
- They start from the simplest to the more advanced.
- The goal is to minimize the propagation delay for generating the answer for the $N$-bit adder, i.e. $N$ sum bits and one $Cout$ bit at the MSB.

# Carry-Ripple PG Logic

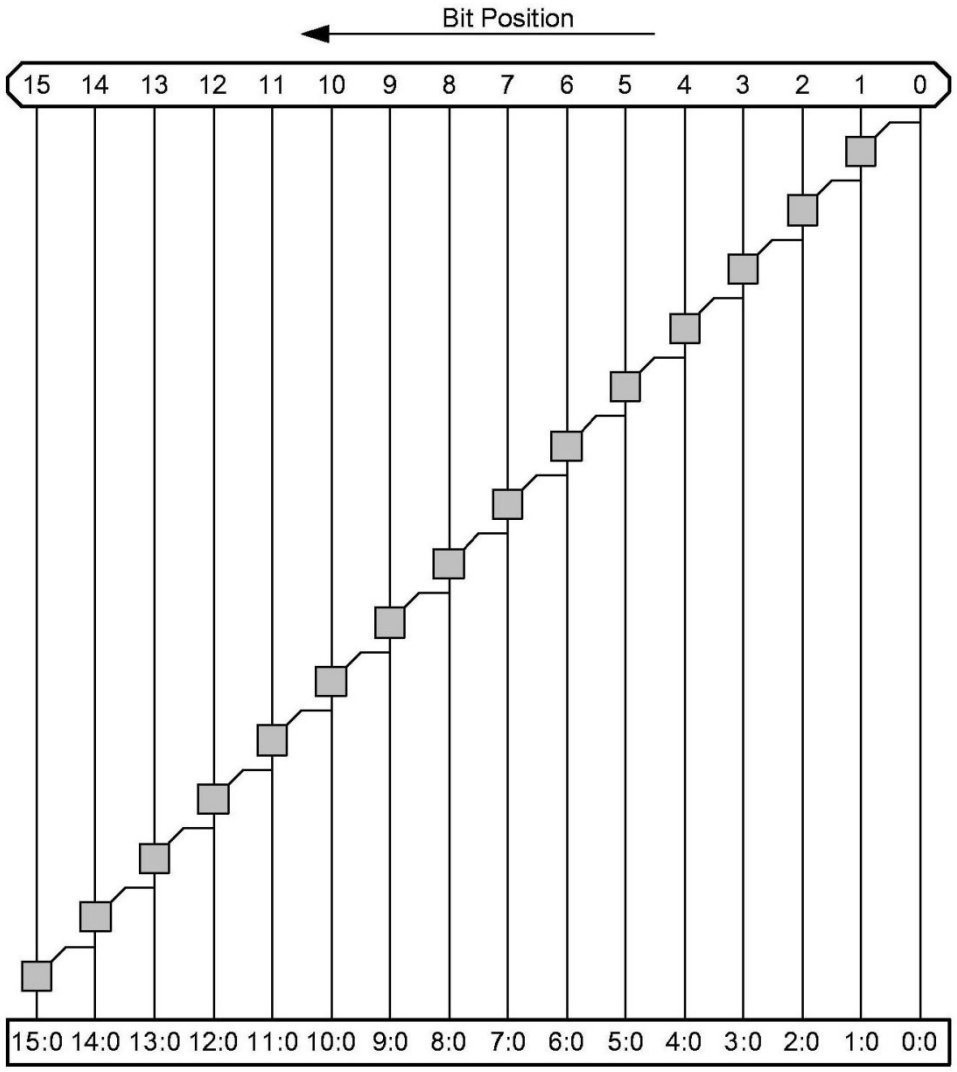
- Simplest Design. Note that $C_i = G_i + P_i\, C_{i\text{-}1}$
- Use $G_{i:0} = G_i + P_i\, G_{i\text{-}1:0}$



# PG Diagram Notation



# PG Diagram – Carry Ripple Adder




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

$t_{pg}$: Single bit PG logic delay

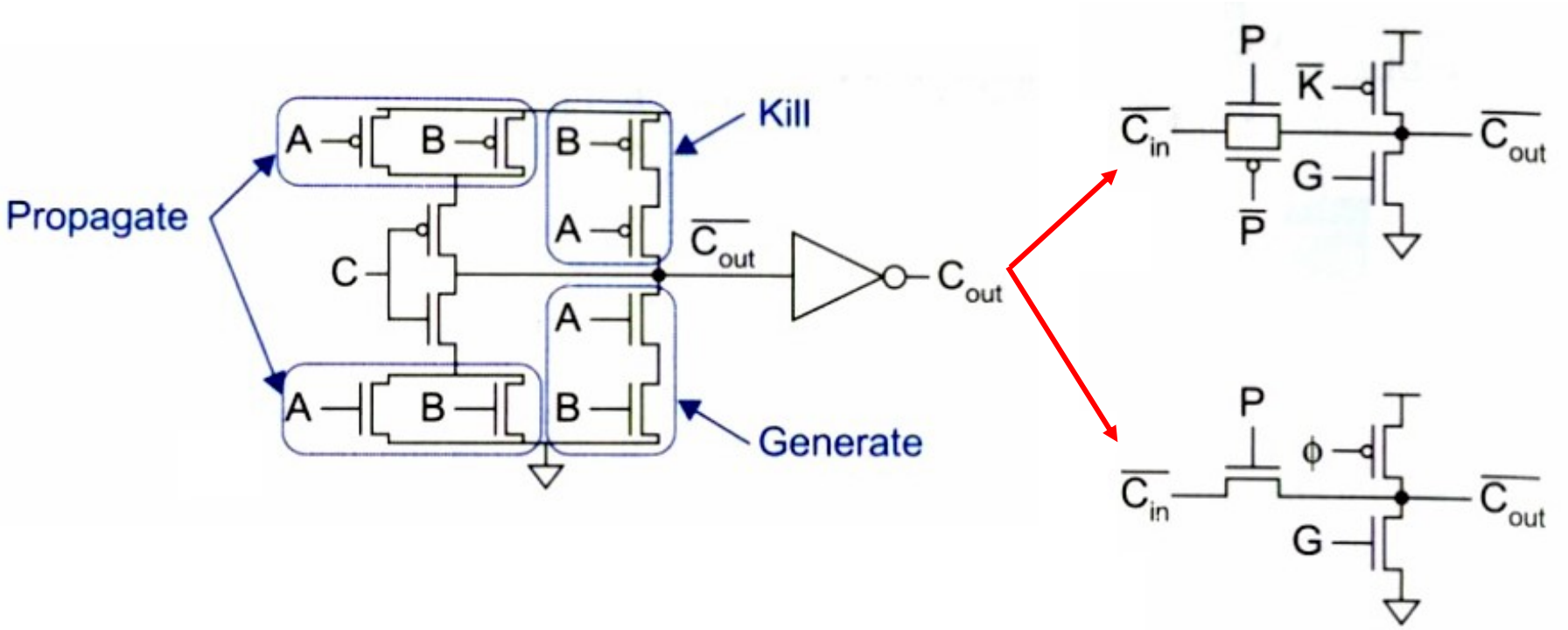
$t_{AO}$: AND-OR gate delay (gray cell)

$t_{XOR}$: XOR gate delay (generate $S_N$)

Note all PG diagrams depict **worst-case** scenarios to estimate **worst-case delay**.

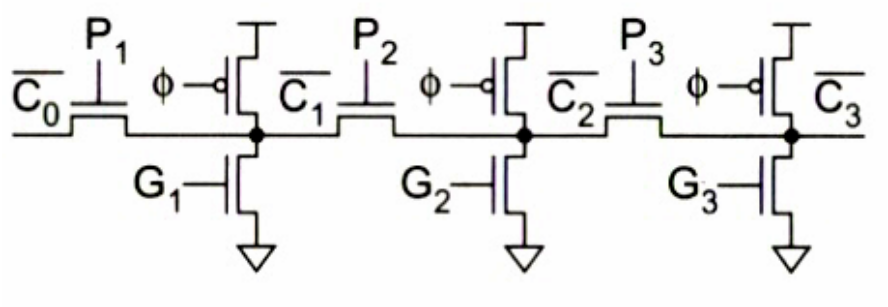
# Carry Chain

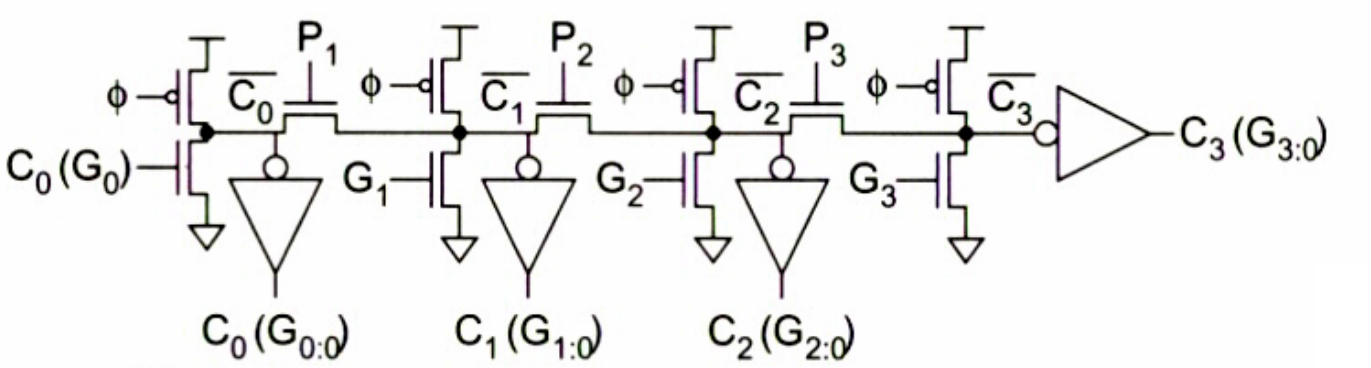
- The static logic gate that computes carry out (Cout) signal
- It can be implemented as a switch network.



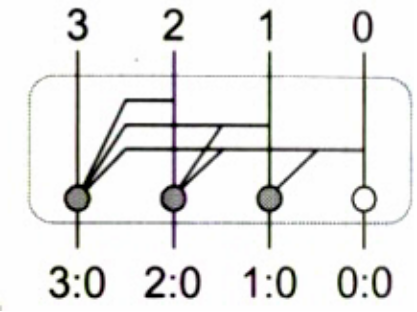
# Manchester Carry Chain

- A chain of 3-4 bits
- Critical path is now a series of pass transistors, better than a series of AND-OR gates in carry-ripple adder.



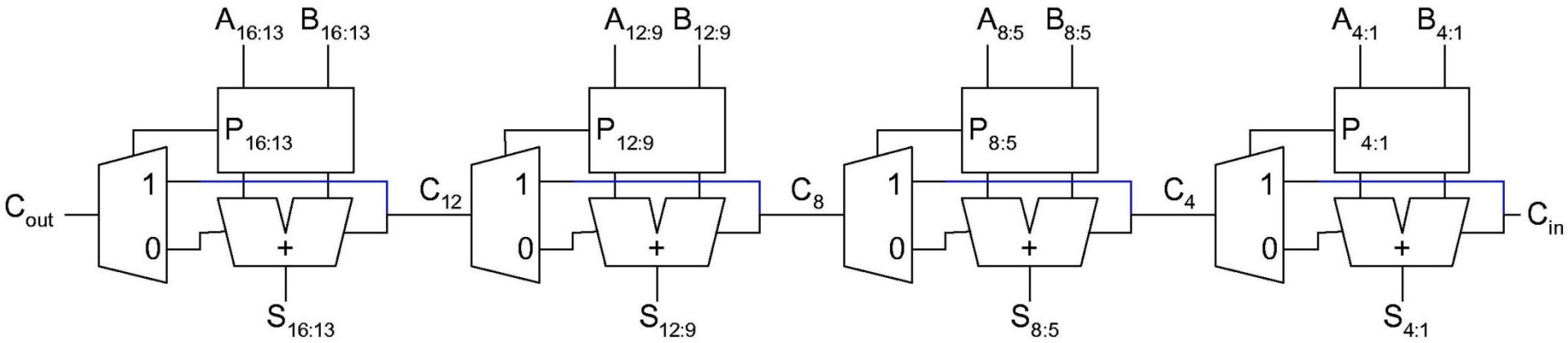




Symbol



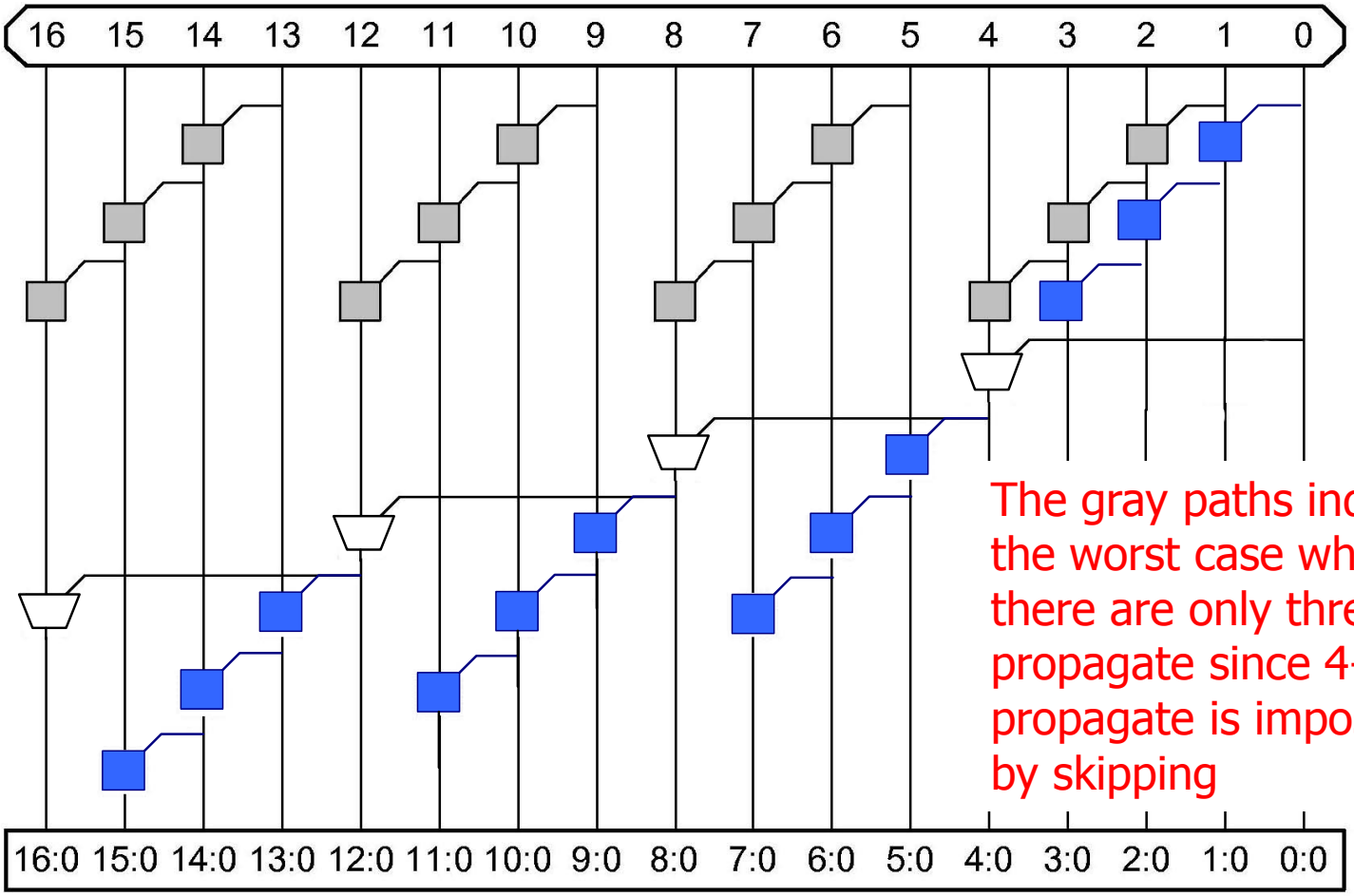
# Carry-Skip Adder

- Carry-ripple (even Manchester carry chain) is slow through all $N$ stages if $N$ is large.
- Carry-skip allows carry to skip over groups of $n$ bits
  - Decision based on $n$-bit propagate signal
- Shorter groups at the beginning and longer groups in the end can shorten the critical path.

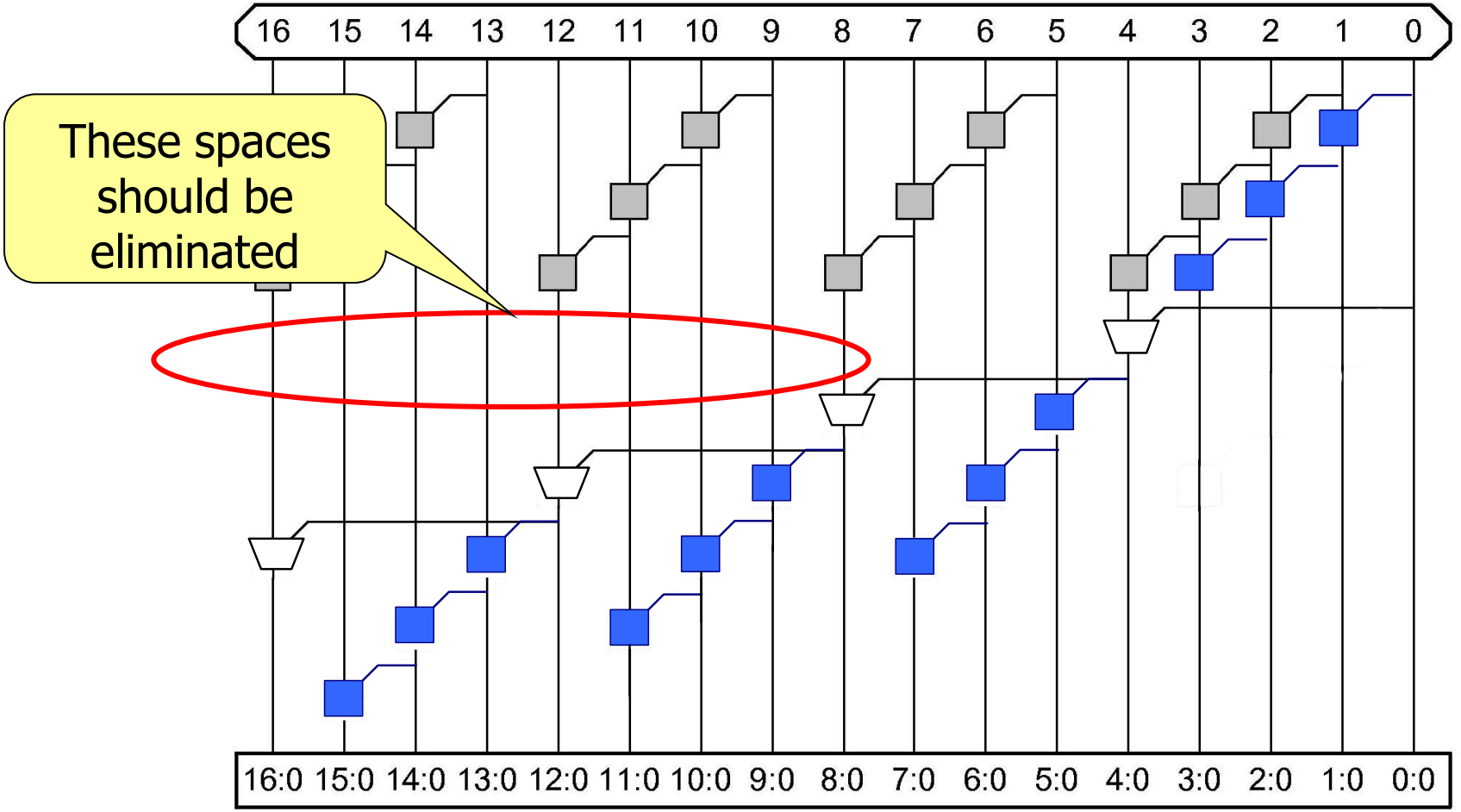


# Carry-Skip PG Diagram

- For $k$ $n$-bit groups ($N = nk$) $t_{\text{skip}} = t_{pg} + 2(n-1)t_{AO} + (k-1)t_{\text{mux}} + t_{\text{xor}}$

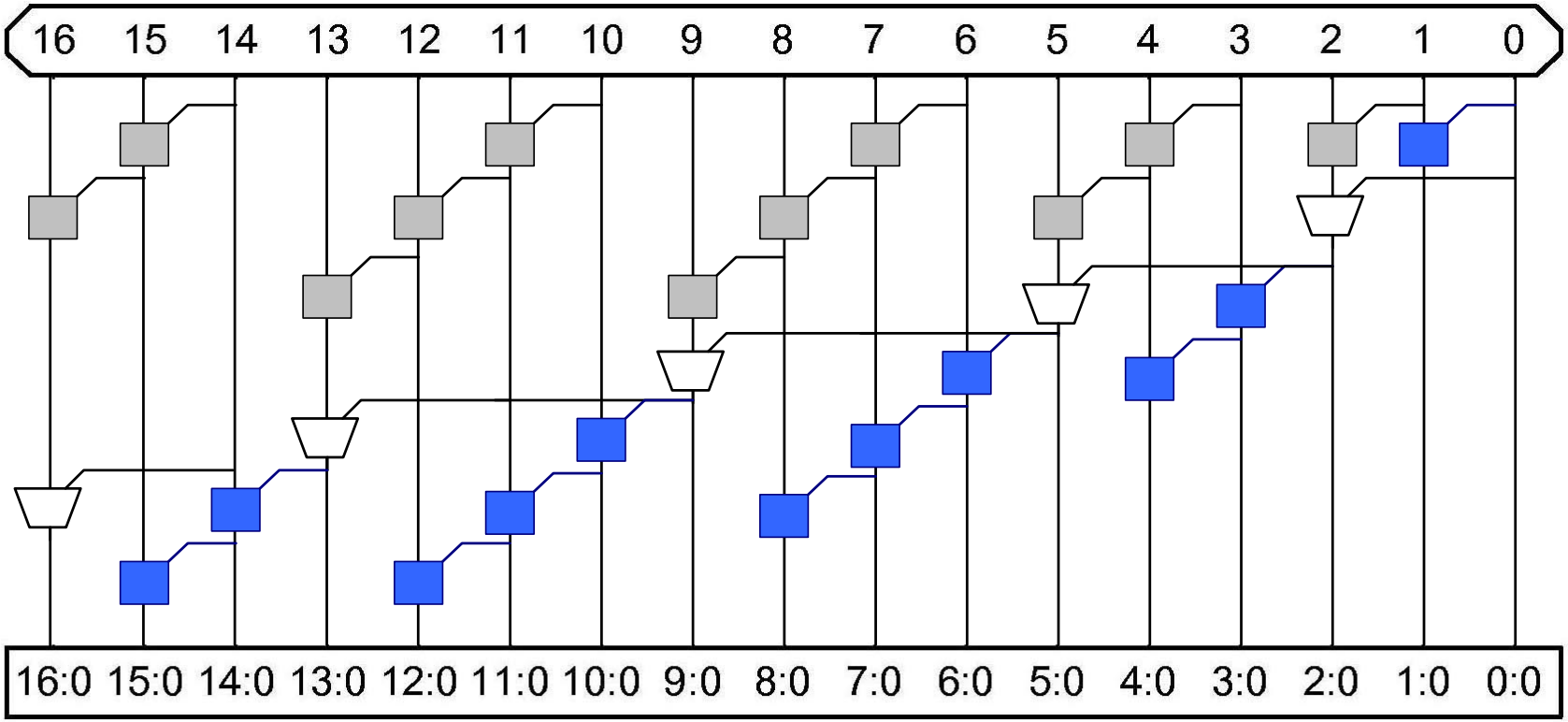


# Where to improve?



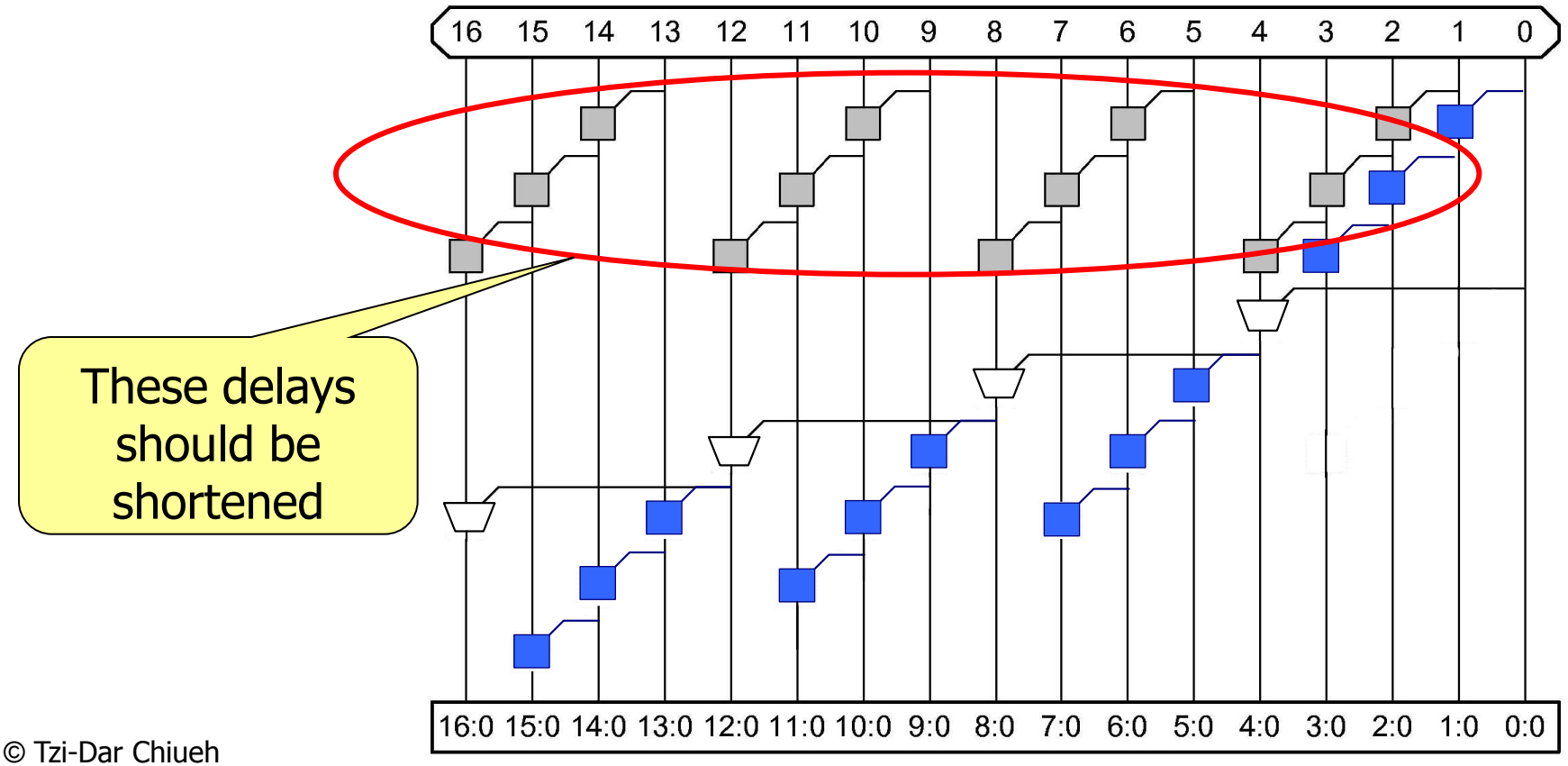
# Variable-Size Carry Skip Adder

- For longer adders, can use shorter groups at the beginning and longer groups in the end can shorten the critical path.



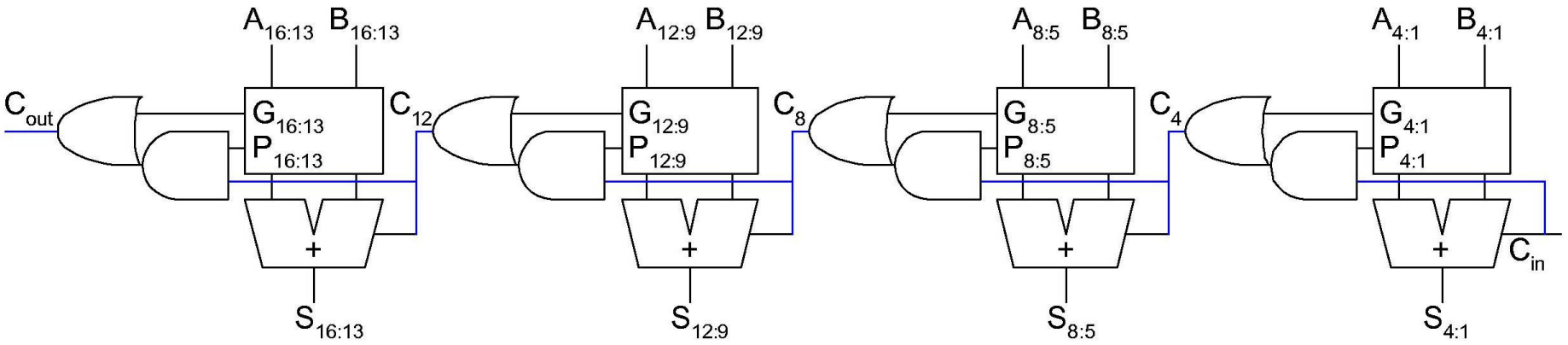
# Carry-Lookahead Adder

- In addition to group propagate in the carry-skip adder, the carry-lookahead adder uses special circuit to compute group generate, avoiding using ripple for group generate.

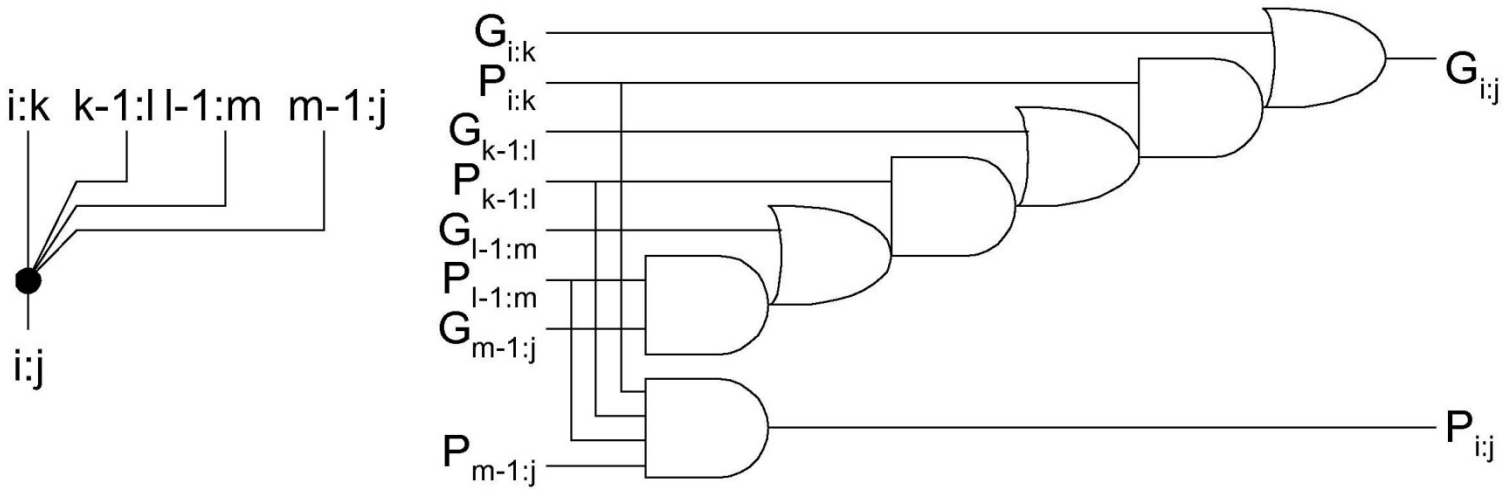


# Carry-Lookahead Adder

- Uses higher-valency cells with more than two inputs.
- Can be slower than the variable group size carry-skip adder.

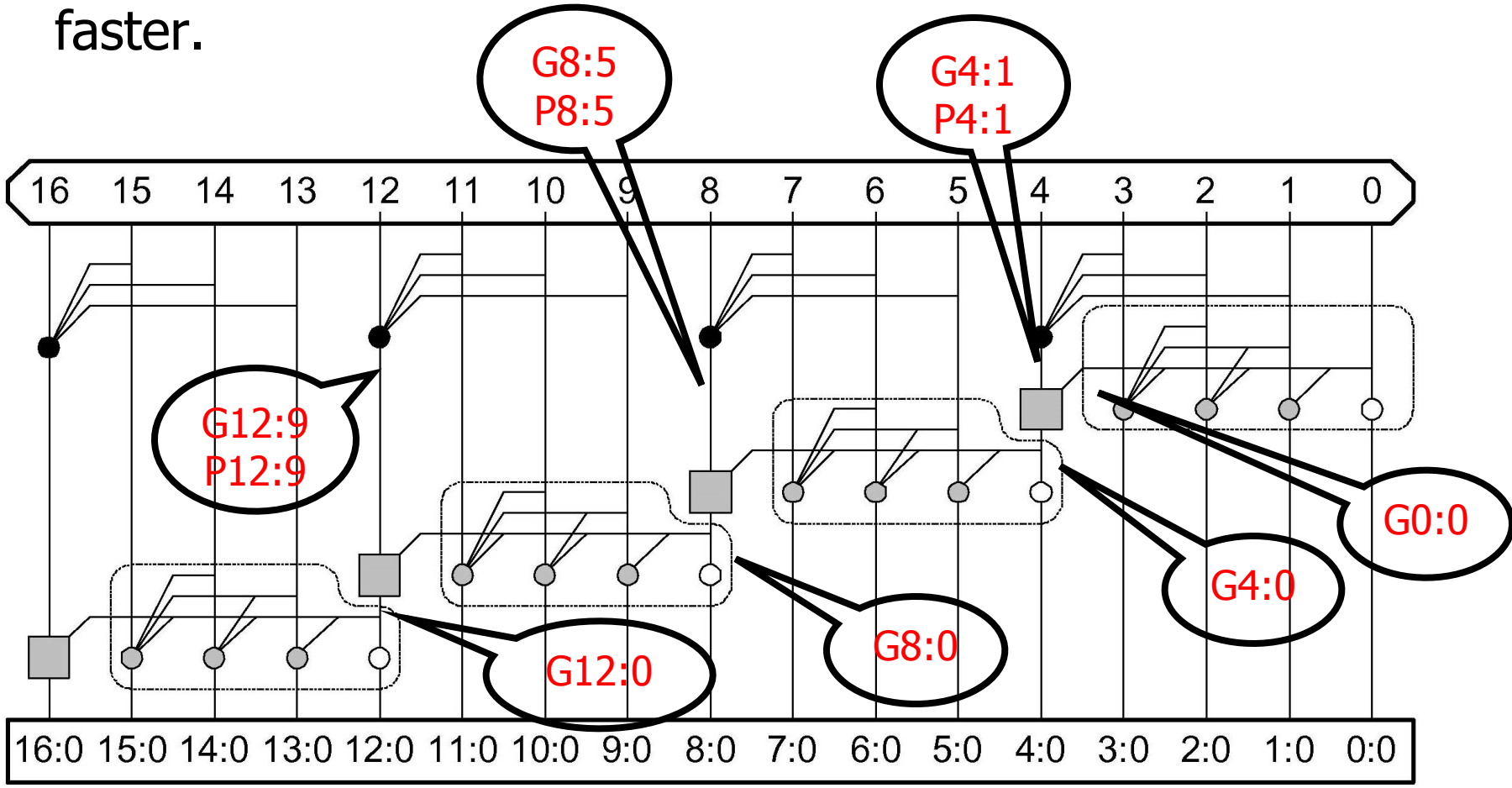


# High Valence Cell



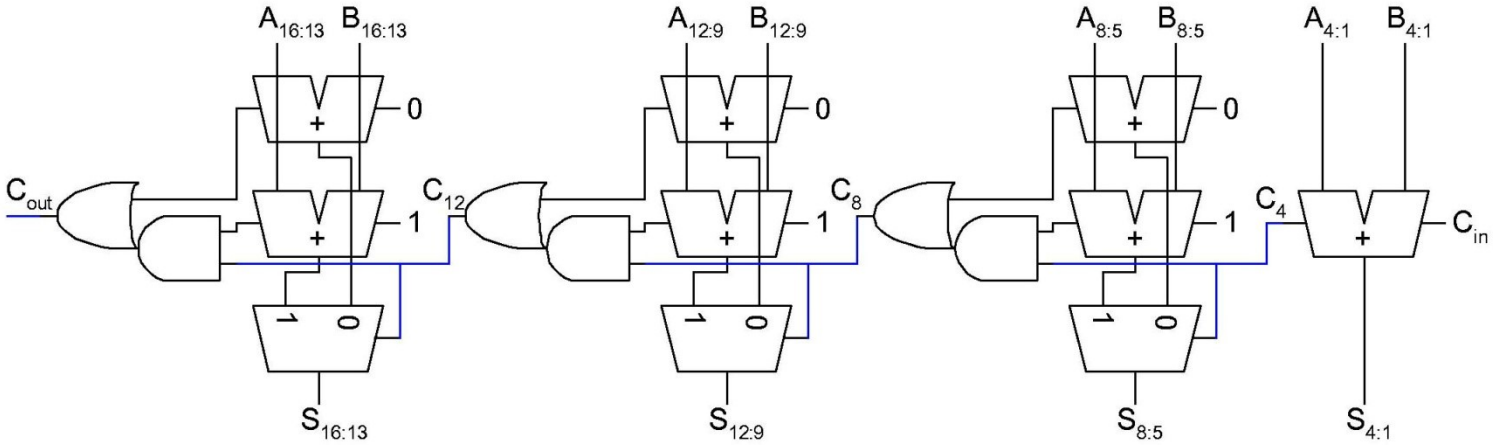
# CLA PG Diagram

- Using carry chain to compute 4-bit adder can make CLA faster.

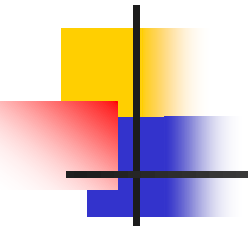


# Carry Select Adder

- Trick for critical paths dependent on late input X
  - Precompute two possible outputs for X = 0, 1
  - Select proper output when X arrives
- Carry-select adder precomputes n-bit sums for both possible carries into n-bit group




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



# Tree Adders

- Instead of partition into fixed-size group of size n bits, can partition into N/2, then recursively partition into N/4, then N/8, etc, to form a binary tree.
- Can have as low as O(logN) delay.
- Many different tree adders. For detail, see textbook.

# Summary on Adders

- Adder architectures offer area / power / delay tradeoffs.
- Choose the best one for your application.

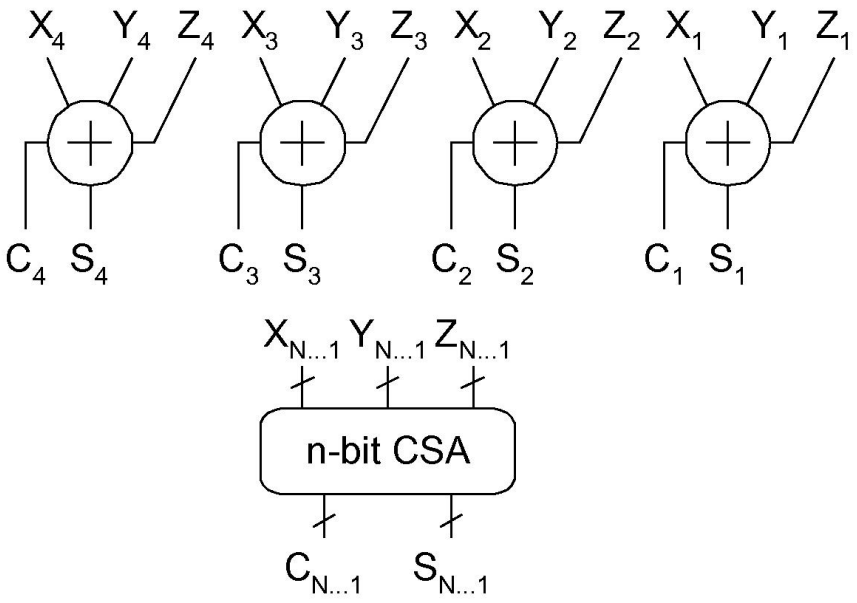
| Architecture | Logic levels | Cells |
|---|---|---|
| Carry Propagate | N-1 | N |
| Carry Skip, n=4 | N/4+5 | 1.25N |
| Carry Select, n=4 | N/4+2 | 2N |

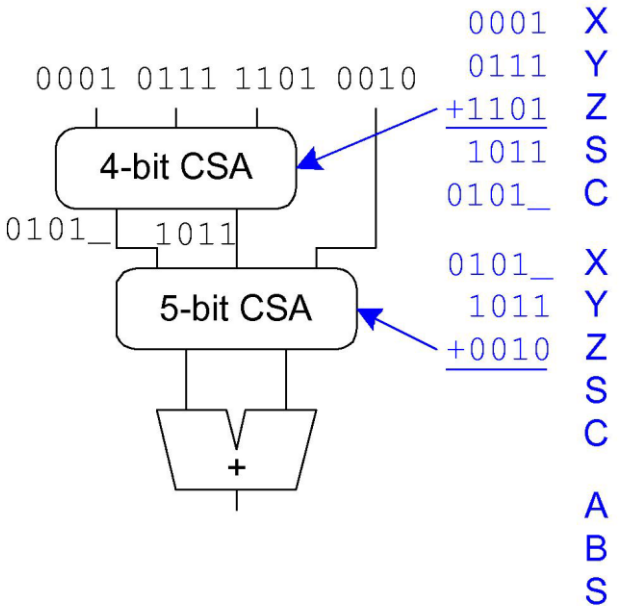
Logic level is the number of cascaded AND-OR gates in the critical path.
Cell is the number of gray and black cells used.

Tree-type adders can have only $\log_2 N$ levels

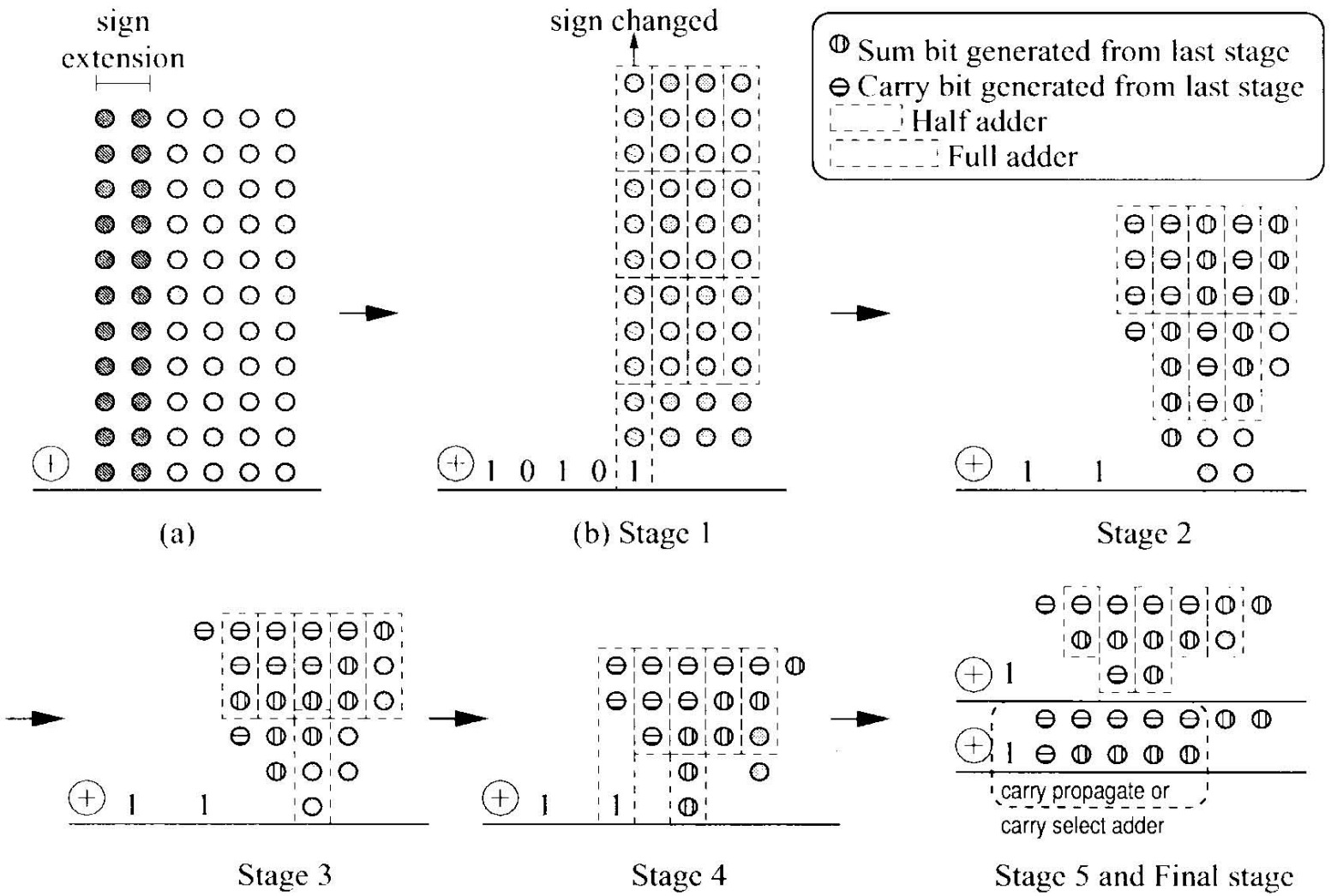
# Multiple-Input Addition

- Full adder can be looked upon as a 3:2 compressor and the carry can be saved and passed to the next stage rather than propagate in this stage.
- **Carry Save Adder (CSA) : many important applications**



# Multiple-Input Addition Example



# Subtraction

- Subtraction using two's complement

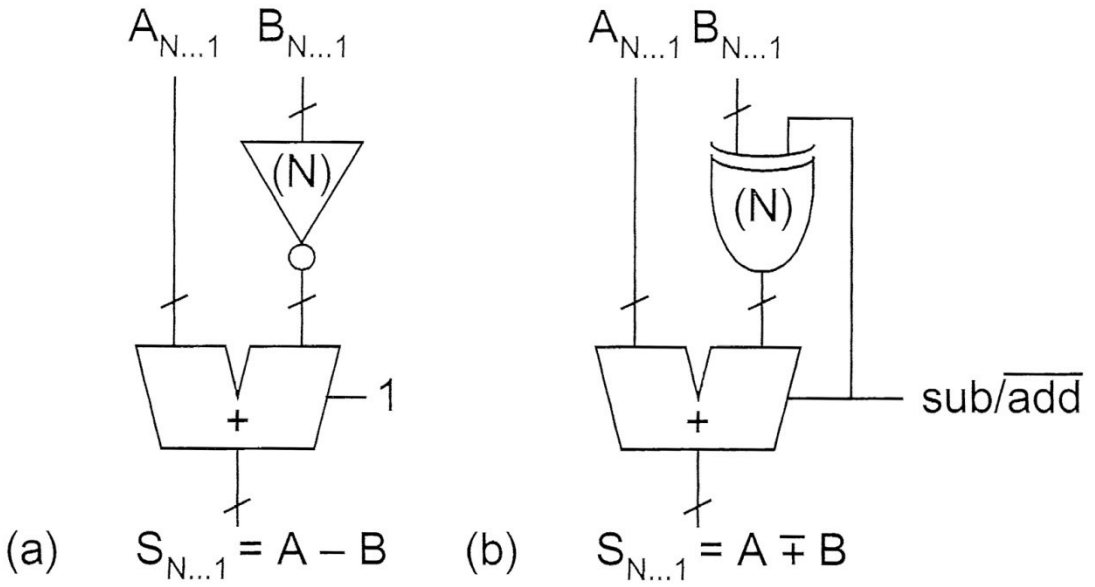
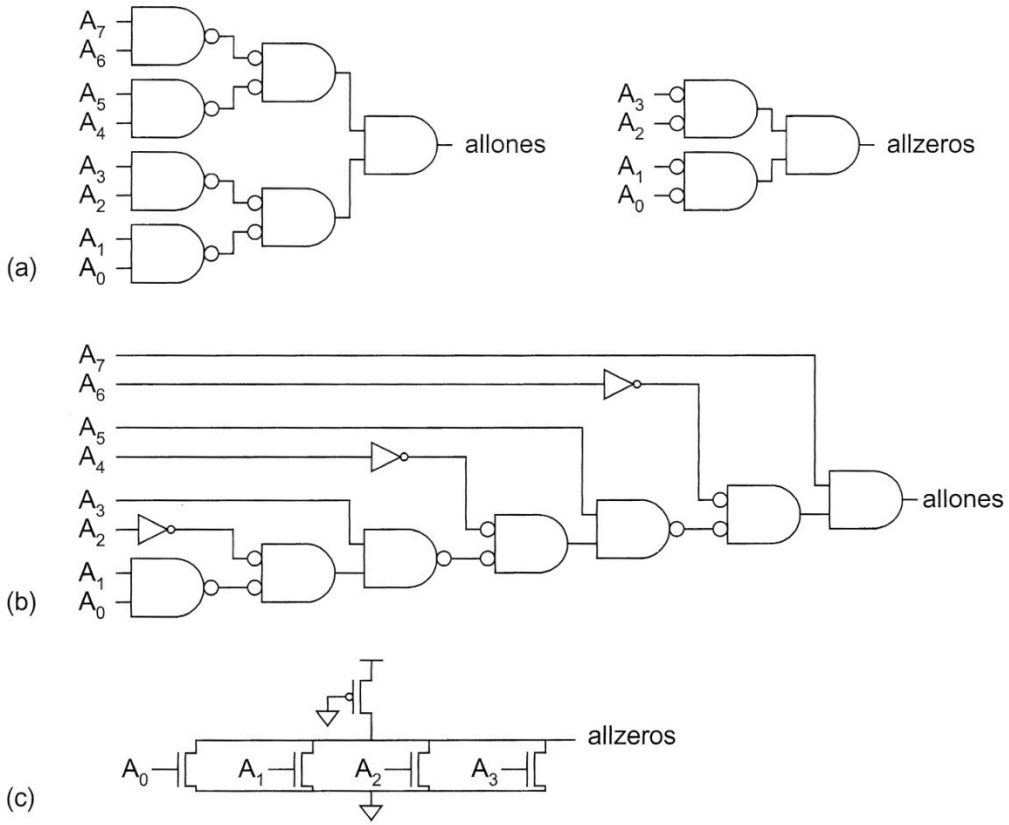



**FIG 10.48** Subtracters

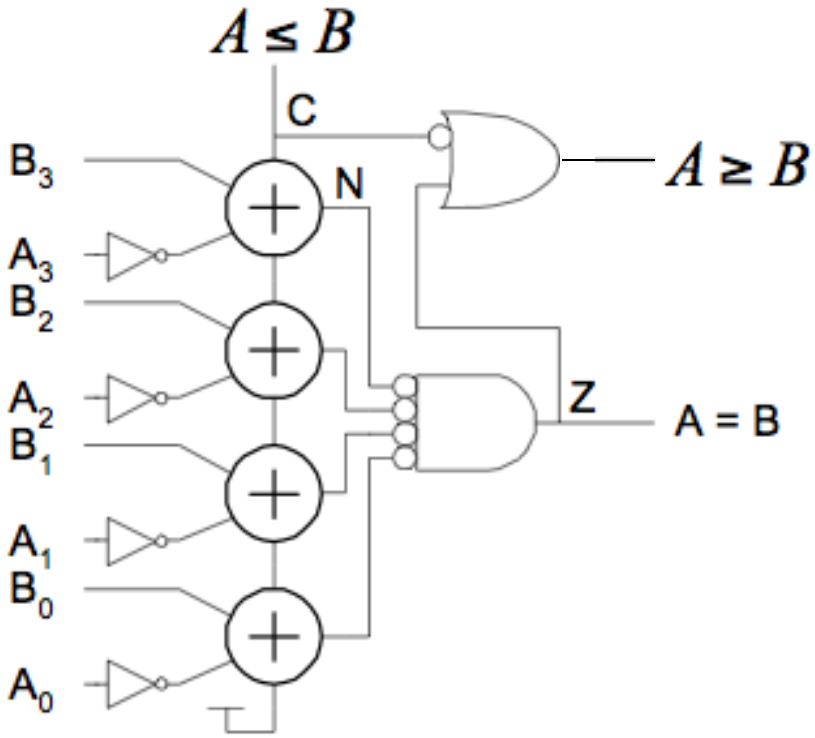
# One/Zero Detectors

- Tree-type zero/one detector; sequential-structure zero/one detector can be used when the inputs arrive at different times. NOR-type pseudo-NMOS or dynamic logic gates can be fast.



# Unsigned Magnitude Comparator

- Compute B-A and look at the sign of the result
- $B\text{-}A = B + \overline{A} + 1$
- For unsigned numbers, carry out is sign bit



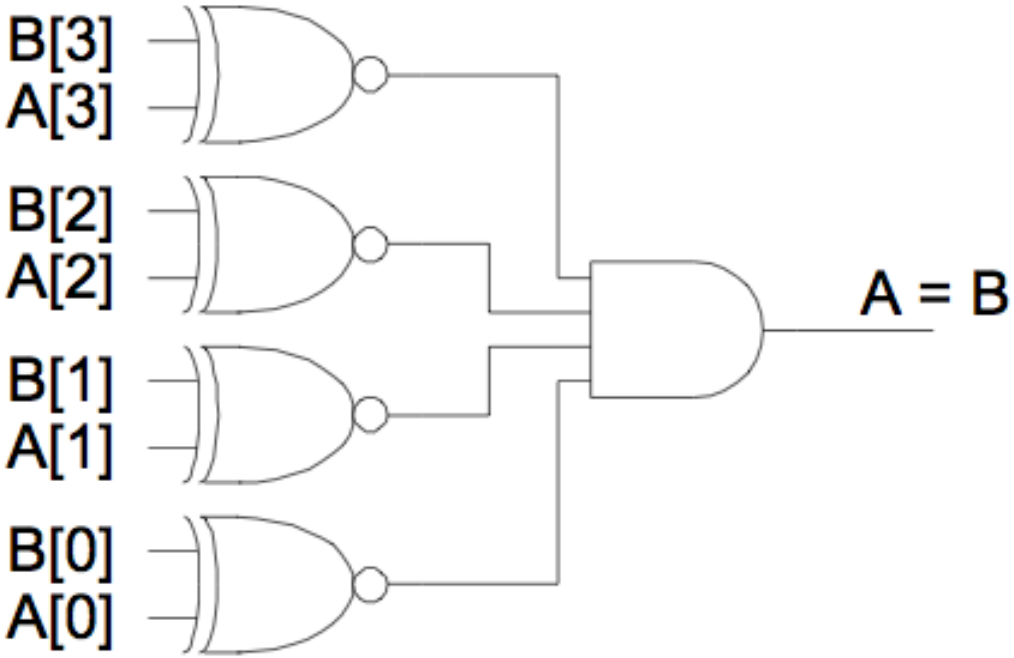
# Signed Comparator

- For signed numbers, comparison is harder
- – C: carry out
- – Z: zero (all bits of B-A are 0)
- – N: negative (MSB of result)
- – V: overflow (inputs had different signs, output sign ≠ B's sign)
- The actual sign of B-A is S= N⊕V because overflow flips the sign.

| Relation | Unsigned Comparison | Signed Comparison |
|---|---|---|
| $A = B$ | $Z$ | $Z$ |
| $A \neq B$ | $\overline{Z}$ | $\overline{Z}$ |
| $A < B$ | $C \cdot \overline{Z}$ | $\overline{S} \cdot \overline{Z}$ |
| $A > B$ | $\overline{C}$ | $S$ |
| $A \leq B$ | $C$ | $\overline{S}$ |
| $A \geq B$ | $\overline{C} + Z$ | $S + Z$ |

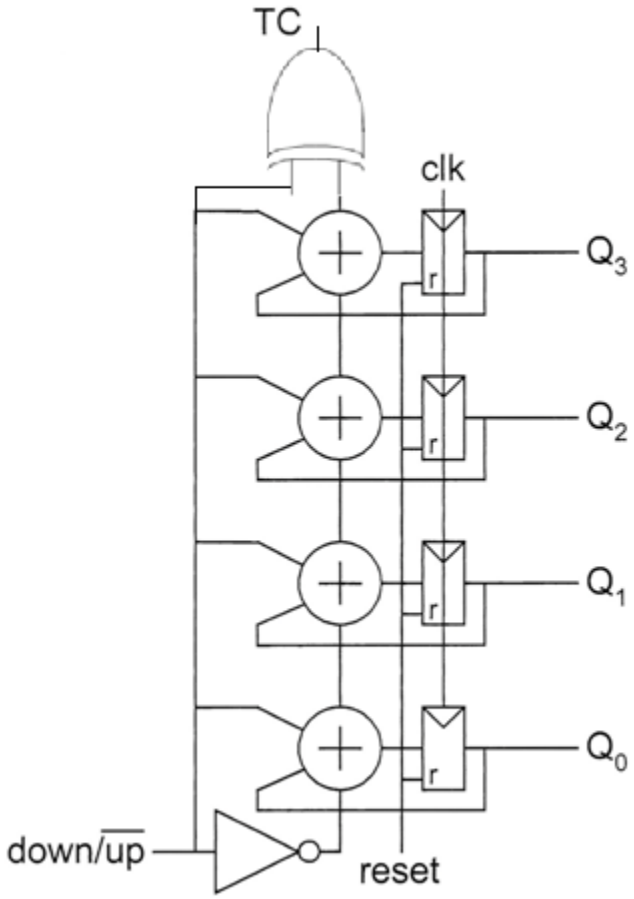
# Equality Comparator

- Check if each pair of bits are equal (XNOR, a.k.a. equality gate)
- Could be done with a pseudo-NMOS or dynamic gate in which the XOR outputs were combined with a wired-OR.

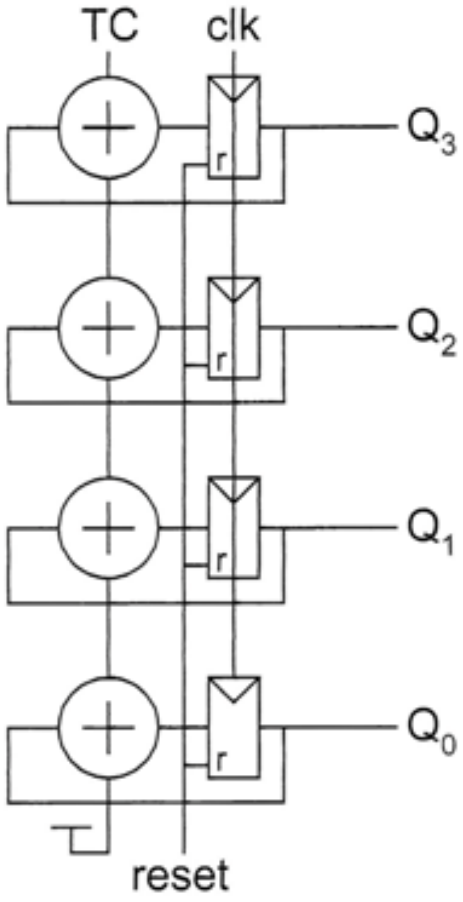


# Synchronous Binary Counters

Synchronous up/down counter




Synchronous incrementer

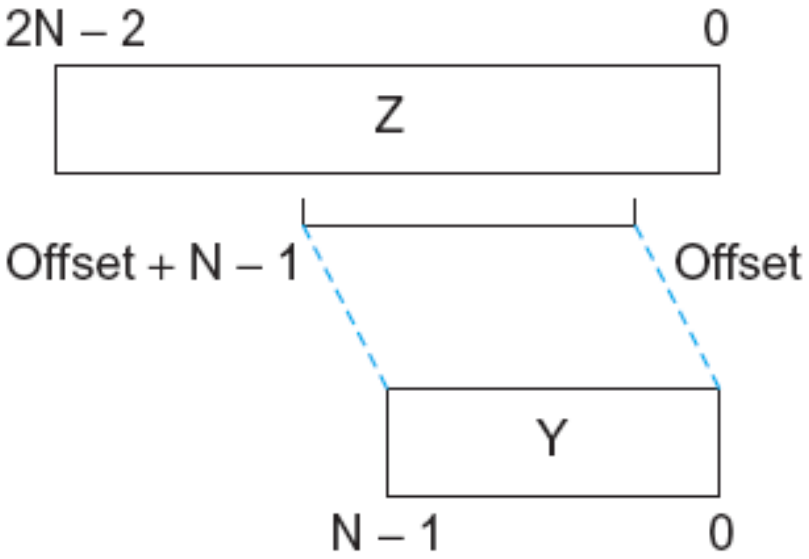


# Shifters

- Logical shift
  - Shift the number to the left or right and fills empty spots with 0's
  - Specify by << or >> in Verilog
  - Ex: 1101, LSR 1=0110, LSL 1=1010
- Arithmetic shift
  - Same as logical shifter but on right shifts fills the MSBs with copies of the sign bit
  - Specify by <<< or >>> in Verilog
  - Ex: 1101 ASR 1=1110, ASL 1=1010
- Rotate
  - Rotates numbers in a circle such that empty spots are filled with bits shifted off the other end
  - Ex: 1101 ROR 1=1110, ROL 1=1011

# Funnel Shifter

- A funnel shifter can do all six types of shifts
- Selects N-bit field Y from (2N-1)-bit input
  - shift by k bits ($0 \leq k < N$)



# Funnel Shifter Operation

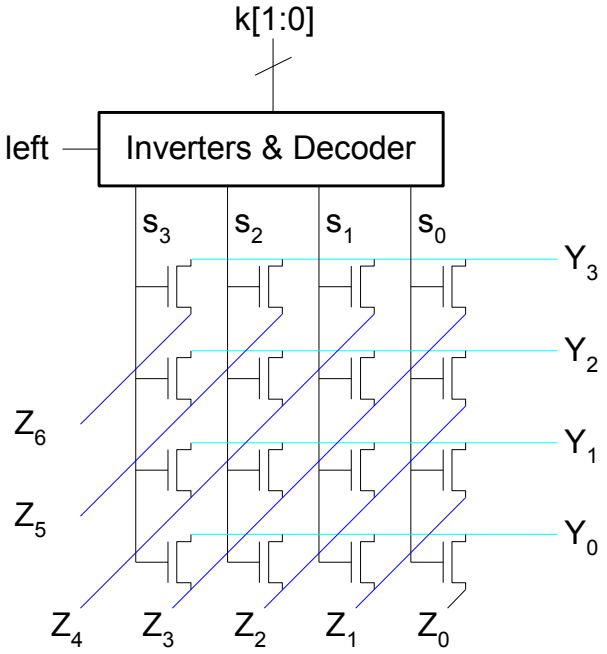
- Computing 1's complement of k
- The left rotate k is identical to right rotate N-k and N-k = $\overline{k}$+1, but when loading we already preshift to right by 1, so offset by $\overline{k}$ is enough.

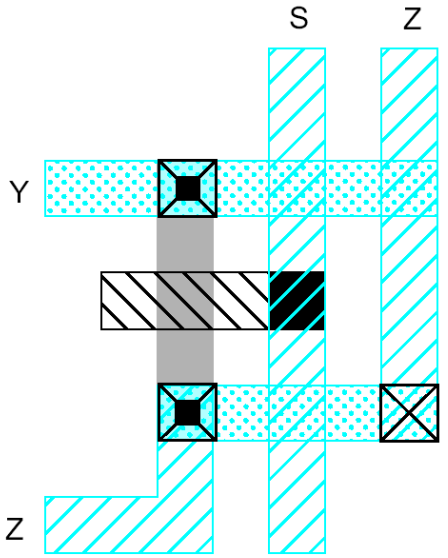
Error in textbook

| Shift Type | $Z_{2N-2:N}$ | $Z_{N-1}$ | $Z_{N-2:0}$ | Offset |
|---|---|---|---|---|
| Rotate Right | $A_{N-2:0}$ | $A_{N-1}$ | $A_{N-2:0}$ | $k$ |
| Logical Right | 0 | $A_{N-1}$ | $A_{N-2:0}$ | $k$ |
| Arithmetic Right | sign | $A_{N-1}$ | $A_{N-2:0}$ | $k$ |
| Rotate Left | $A_{N-1:1}$ | $A_0$ | $A_{N-1:1}$ | $\overline{k}$ |
| Logical/Arithmetic Left | $A_{N-1:1}$ | $A_0$ | 0 | $\overline{k}$ |

# Funnel Shifter Design 1

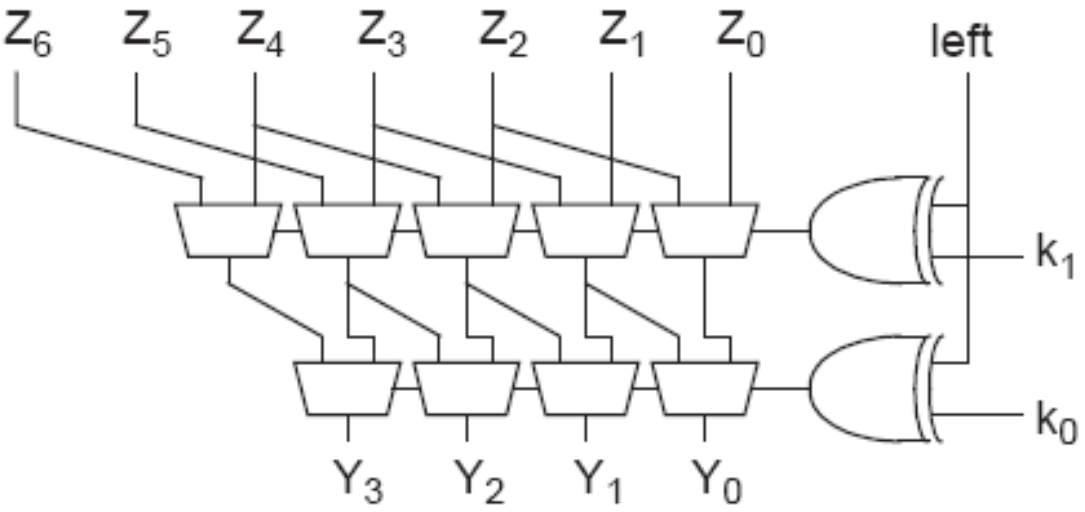
- N N-input multiplexers
  - Use 1-of-N hot select signals for shift amount
  - nMOS pass transistor design ($V_t$ drops!)



# Funnel Shifter Design 2

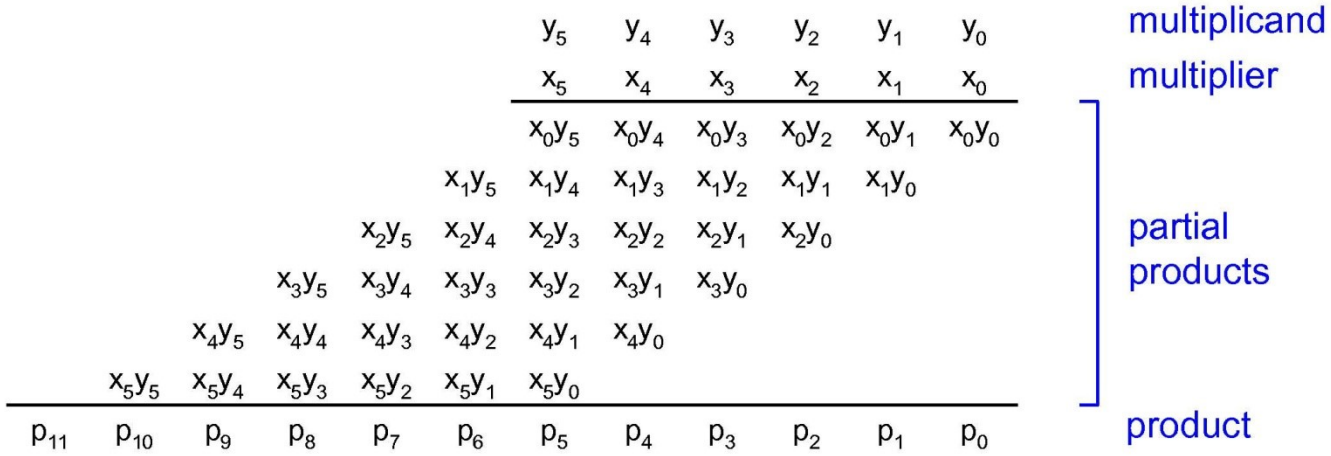
- Log N stages of 2-input MUXes
  - No select decoding needed



# Multiplication

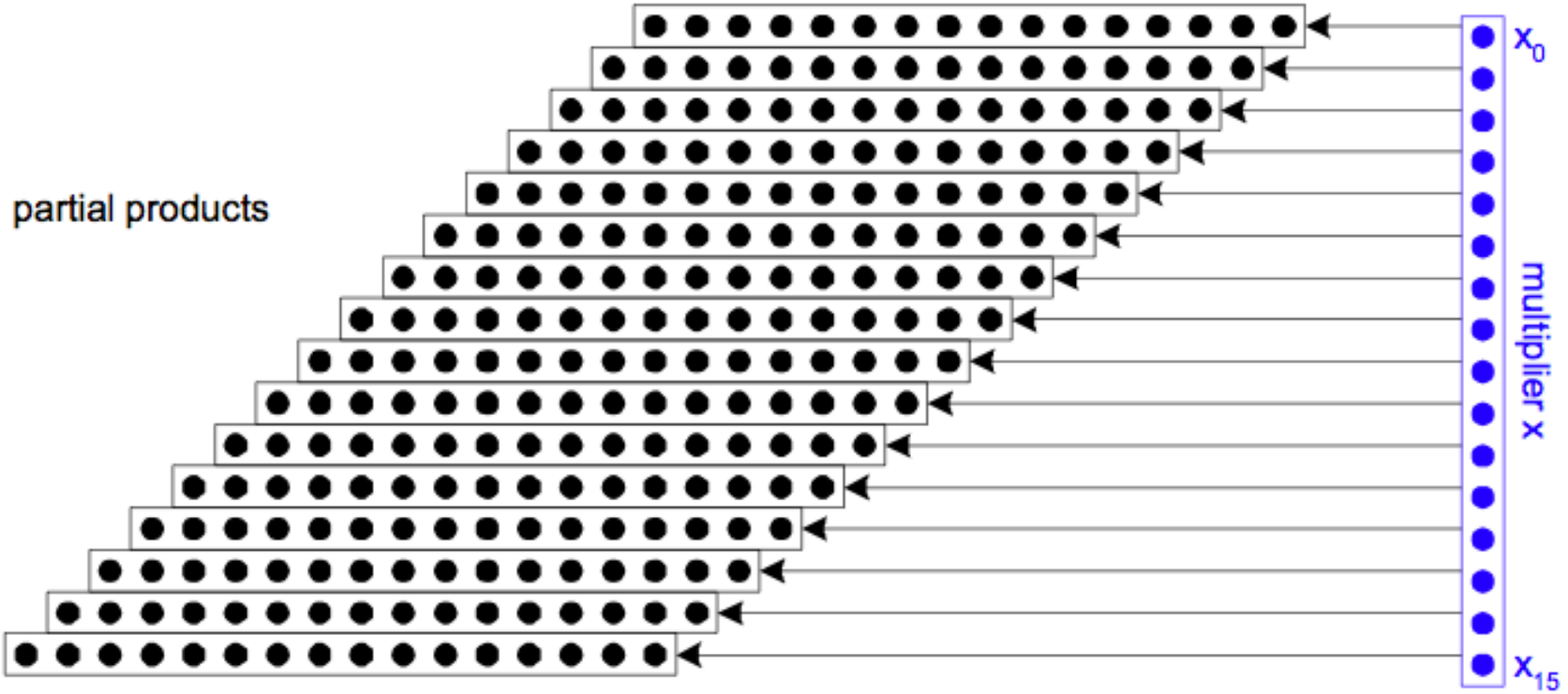
- Multiplicand: $Y = (y_{M-1}, y_{M-2}, \ldots, y_1, y_0)$
- Multiplier: $X = (x_{N-1}, x_{N-2}, \ldots, x_1, x_0)$
- Product:

$$P = \left( \sum_{j=0}^{M-1} y_j 2^j \right) \left( \sum_{i=0}^{N-1} x_i 2^i \right) = \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} x_i y_j 2^{i+j}$$

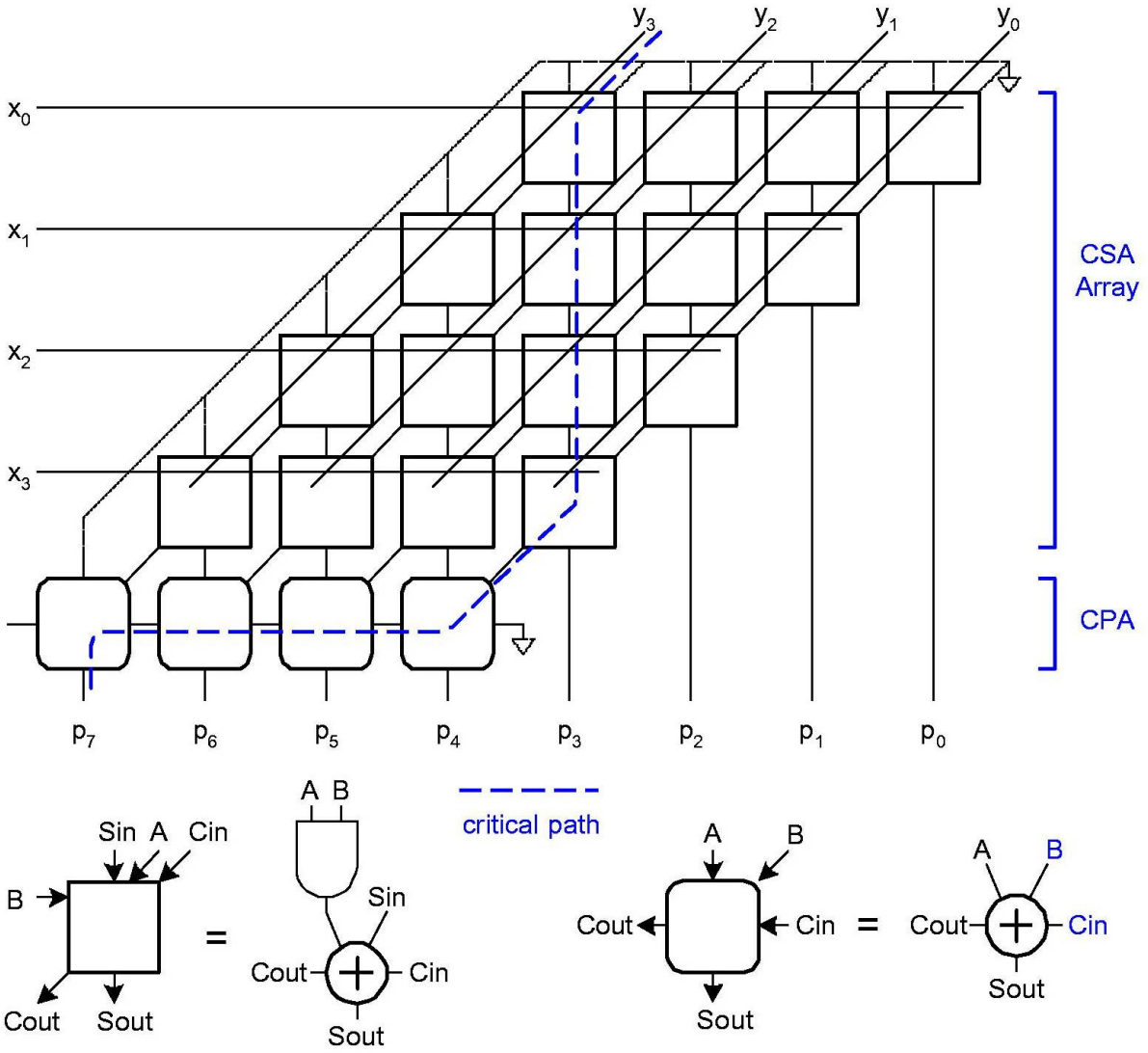


# Dot Diagram

- Each dot represents a bit

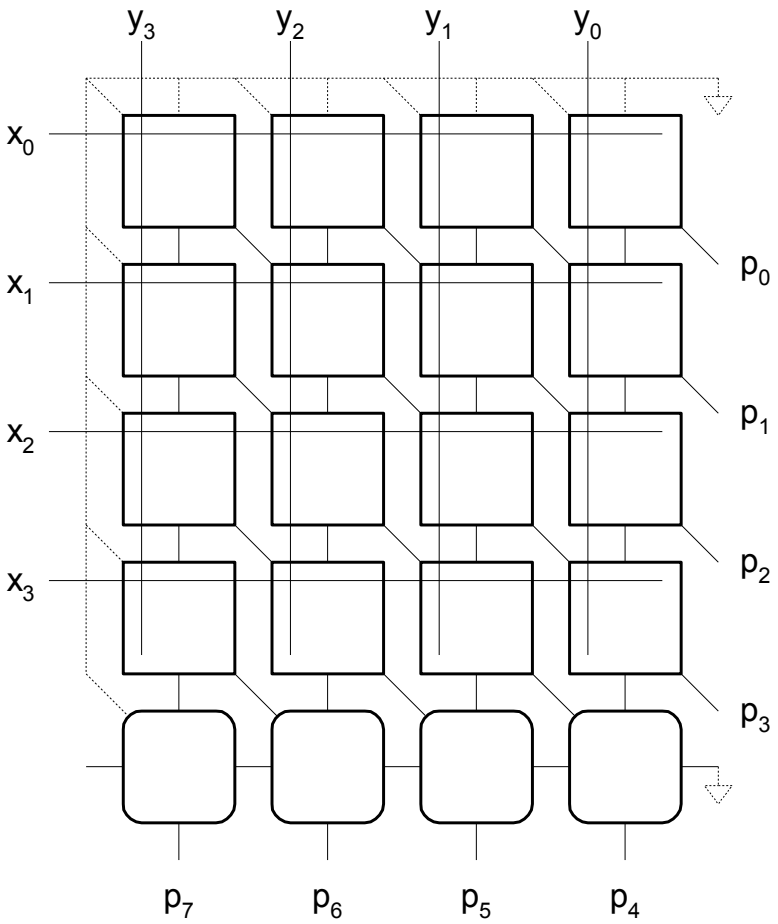


# Unsigned Array Multiplication



# Rectangular Array

- Squash array to fit rectangular floorplan



# Two's Complement Multiplication

- A binary two's-complement number is formulated as

$$X = -X_{N-1}2^{N-1} + X_{N-2}2^{N-2} + X_{N-3}2^{N-3} \cdots + X_0 2^0$$
$$= -X_{N-1}2^{N-1} + \sum_{i=0}^{N-2} X_i 2^i$$

- Similarly for $Y$ and the product of $X$ and $Y$ is

$$XY = (-X_{N-1}2^{N-1} + \sum_{i=0}^{N-2} X_i 2^i)(-Y_{N-1}2^{N-1} + \sum_{j=0}^{N-2} Y_j 2^j)$$

$$= X_{N-1}Y_{N-1}2^{2N-2} + \sum_{i=0}^{N-2}\sum_{j=0}^{N-2} X_i Y_j 2^{i+j} - Y_{N-1}\sum_{i=0}^{N-2} X_i 2^{N+i-1} - X_{N-1}\sum_{j=0}^{N-2} Y_j 2^{N+j-1}$$

- The last two terms can both be expressed as

$$-\sum_{i=0}^{N-2} Y_{N-1}X_i 2^{N+i-1} = -2^{2N-2} + (\sum_{i=0}^{N-2}(1 - Y_{N-1}X_i)2^{N+i-1}) + 2^{N-1}$$
$$= -2^{2N-2} + (\sum_{i=0}^{N-2}\overline{Y_{N-1}X_i}\,2^{N+i-1}) + 2^{N-1}$$

# Two's Complement Multiplication

- Thus

$$XY = -2^{2N-1} + 2^N + X_{N-1}Y_{N-1}2^{2N-2} + \left(\sum_{i=0}^{N-2} \overline{Y_{N-1}X_i}\, 2^{N+i-1}\right) +$$
$$\left(\sum_{j=0}^{N-2} \overline{X_{N-1}Y_j}\, 2^{N+i-1}\right) + \sum_{i=0}^{N-2}\sum_{j=0}^{N-2} X_i Y_j 2^{i+j}$$
$$= 2^{2N-1} + 2^N + X_{N-1}Y_{N-1}2^{2N-2} + \left(\sum_{i=0}^{N-2} \overline{Y_{N-1}X_i}\, 2^{N+i-1}\right) +$$
$$\left(\sum_{j=0}^{N-2} \overline{X_{N-1}Y_j}\, 2^{N+i-1}\right) + \sum_{i=0}^{N-2}\sum_{j=0}^{N-2} X_i Y_j 2^{i+j}$$

- Example

| | | | | | |
|---|---|---|---|---|---|
| | | x3 | x2 | x1 | x0 |
| | | y3 | y2 | y1 | y0 |
| | | --- | --- | --- | --- |
| | 1 | $\overline{x3y0}$ | x2y0 | x1y0 | x0y0 |
| | $\overline{x3y1}$ | x2y1 | x1y1 | x0y1 | |
| $\overline{x3y2}$ | x2y2 | x1y2 | x0y2 | | |
| 1 x3y3 | $\overline{x2y3}$ | $\overline{x1y3}$ | $\overline{x0y3}$ | | |

---

# Two's Complement Array Multiplication

- Modified Baugh-Wooly multiplier

| $p_{11}$ | $p_{10}$ | $p_9$ | $p_8$ | $p_7$ | $p_6$ | $p_5$ | $p_4$ | $p_3$ | $p_2$ | $p_1$ | $p_0$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
| | | | | | | $x_5$ | $x_4$ | $x_3$ | $x_2$ | $x_1$ | $x_0$ |
| | | | | | 1 | $\overline{x_5y_0}$ | $x_0y_4$ | $x_0y_3$ | $x_0y_2$ | $x_0y_1$ | $x_0y_0$ |
| | | | | | $\overline{x_5y_1}$ | $x_1y_4$ | $x_1y_3$ | $x_1y_2$ | $x_1y_1$ | $x_1y_0$ | |
| | | | | $\overline{x_5y_2}$ | $x_2y_4$ | $x_2y_3$ | $x_2y_2$ | $x_2y_1$ | $x_2y_0$ | | |
| | | | $\overline{x_5y_3}$ | $x_3y_4$ | $x_3y_3$ | $x_3y_2$ | $x_3y_1$ | $x_3y_0$ | | | |
| | | $\overline{x_5y_4}$ | $x_4y_4$ | $x_4y_3$ | $x_4y_2$ | $x_4y_1$ | $x_4y_0$ | | | | |
| 1 | $x_5y_5$ | $\overline{x_4y_5}$ | $\overline{x_3y_5}$ | $\overline{x_2y_5}$ | $\overline{x_1y_5}$ | $\overline{x_0y_5}$ | | | | | |

# Booth Algorithm

- Array multiplier requires N partial products
- If we looked at groups of r bits, we could form N/r partial products, called radix-$2^r$ encoding
- Looking at two bits of the multiplier at each step and there are four cases in $Y * X$

| $X_i$ | $X_{i-1}$ | | | Operation |
|---|---|---|---|---|
| 0 | 0 | Middle of a run of 0s | 00001111$\overline{00}$00 | Do nothing |
| 0 | 1 | End of a run of 1s | 0000$\overline{1}$1110000 | Add $Y$ |
| 1 | 0 | start of a run of 1s | 00001111$\overline{0}$0000 | Subtract $Y$ |
| 1 | 1 | Middle of a run of 1s | 00001$\overline{11}$0000 | Do nothing |

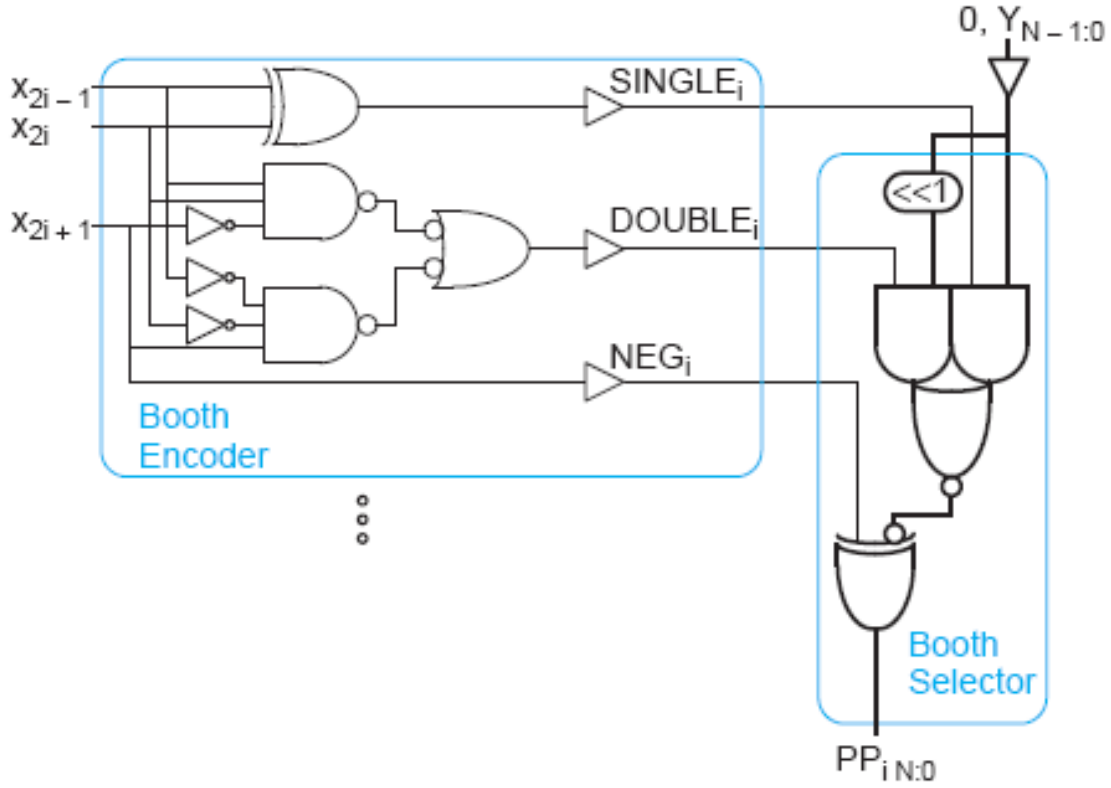
# Modified Booth Encoding

- Examine 3 bits at a time
  - Avoid generating 3Y by using negative partial products

| Inputs | | | Partial Product | Booth Selects | | |
|---|---|---|---|---|---|---|
| $x_{2i+1}$ | $x_{2i}$ | $x_{2i-1}$ | $PP_i$ | $X_i$ | $2X_i$ | $M_i$ |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | $Y$ | 1 | 0 | 0 |
| 0 | 1 | 0 | $Y$ | 1 | 0 | 0 |
| 0 | 1 | 1 | $2Y$ | 0 | 1 | 0 |
| 1 | 0 | 0 | $-2Y$ | 0 | 1 | 1 |
| 1 | 0 | 1 | $-Y$ | 1 | 0 | 1 |
| 1 | 1 | 0 | $-Y$ | 1 | 0 | 1 |
| 1 | 1 | 1 | $-0$ (= 0) | 0 | 0 | 1 |

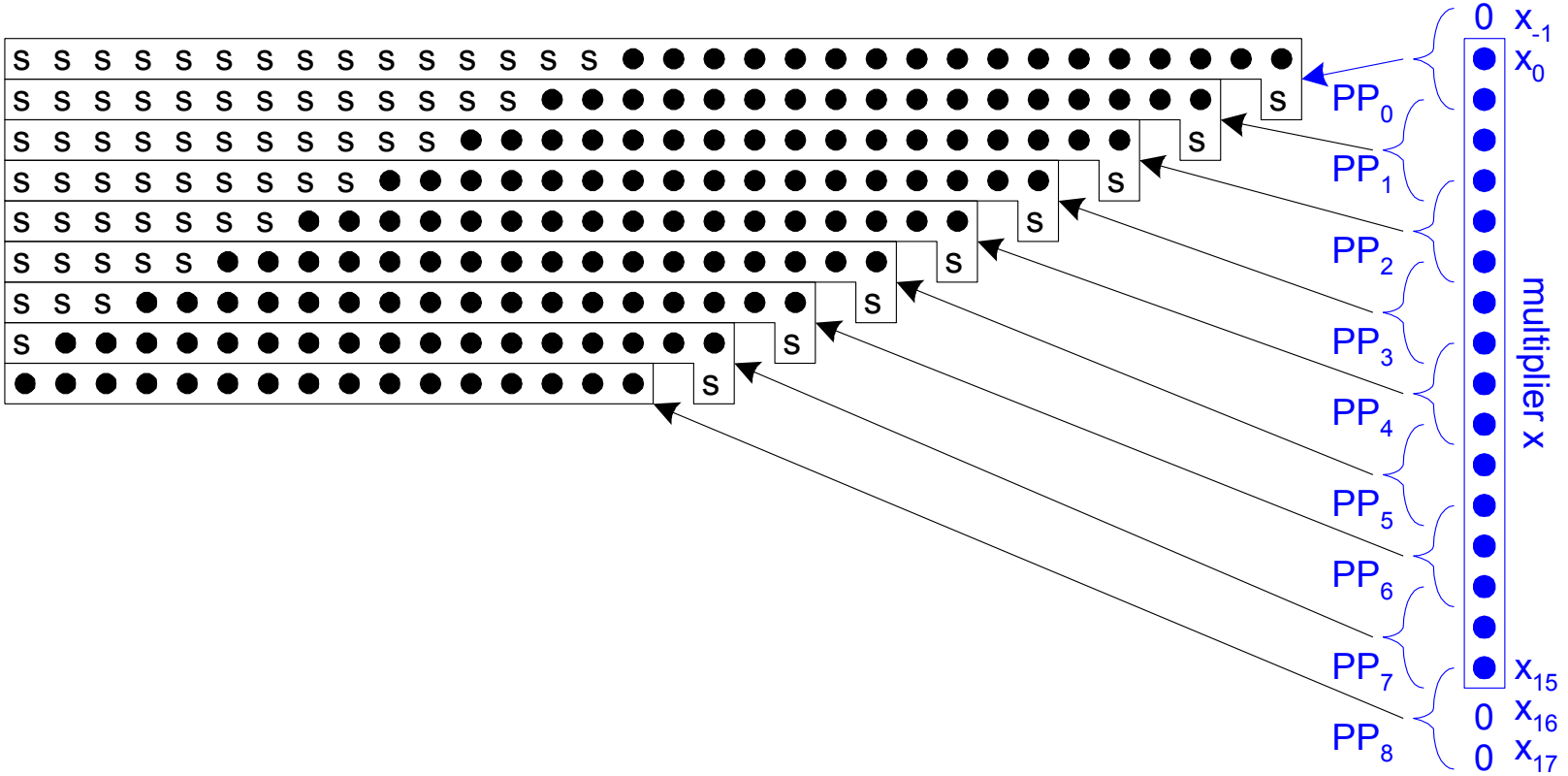
# Booth Encoding

- Radix-4 Booth encoder and selector



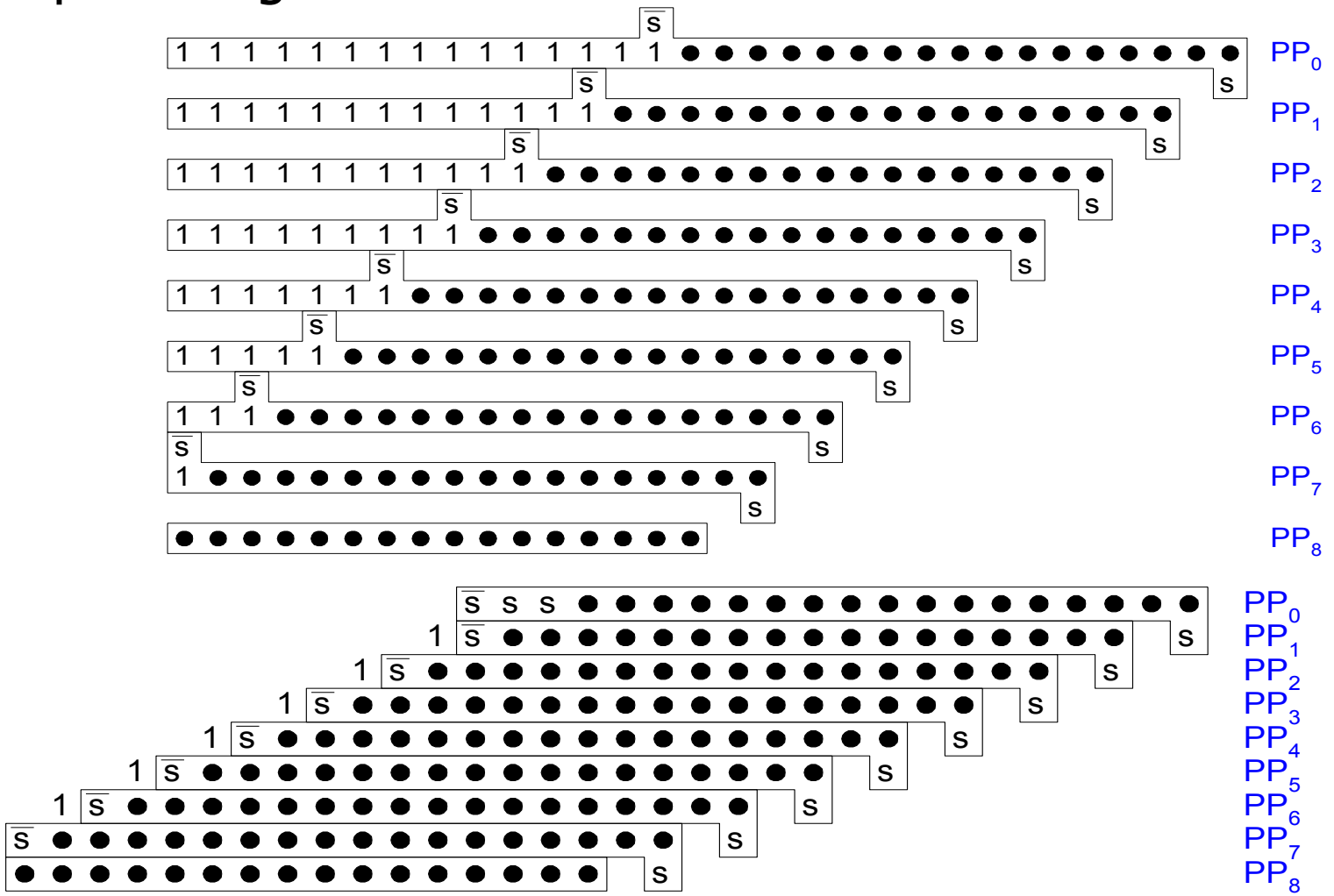
# Unsigned Multiplier Using Booth Encoding

- Sign extension



# Booth Encoding

- Simplified sign extension

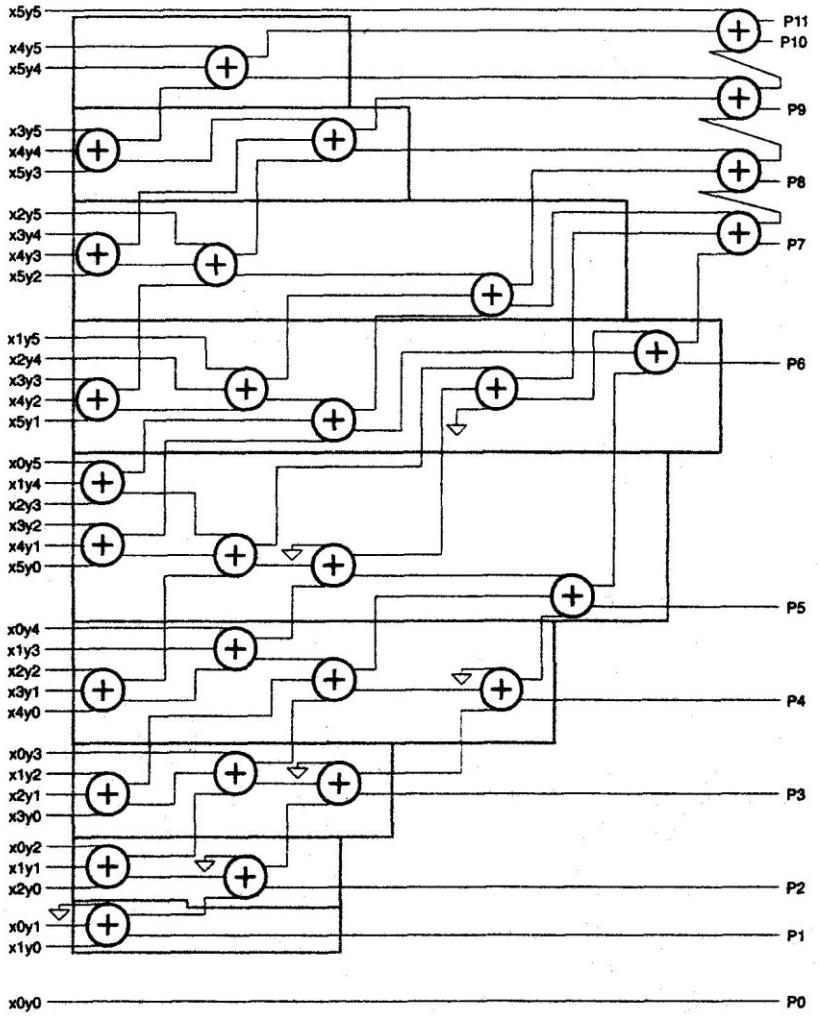


# Wallace Tree Multiplication

- A full adder can be looked upon as a one's counter and it provides a 3:2 compression in the number of bits.
- In a multiplication, the addition of the partial products in the same column can be thought of as counting the number of ones in these partial products.

# Wallace Tree Multiplier

- An *n* x *n* Wallace tree multiplier has approximately $\log_{3/2} n$ stages. So its delay is smaller than array multiplier and Booth recoded multiplier. However, the Wallace tree architecture can be combined with Booth encoding to yield a more efficient multiplier.
- Note a final CPA is necessary.



# Wallace Tree

- Other techniques, such as pipelining and higher-radix multiplication scheme can all be combined with Wallace tree architecture to form a super fast multiplier.