# Digital Integrated Circuits
(83-313)

# Lecture 6:
# Arithmetic Circuits

Semester B, 2016-17

Lecturer: Dr. Adam Teman

TAs: Itamar Levi, Robert Giterman

Emerging Nanoscaled Integrated Circuits and Systems Labs

7 May 2017

Tradition of Excellence

Bar-Ilan University
אוניברסיטת בר-אילן

## Lecture Content

DataPaths

3 Emerging Nanoscaled Integrated Circuits and Systems Labs

Bar-Ilan University אוניברסיטת בר-אילן

Adders

6 Emerging Nanoscaled Integrated Circuits and Systems Labs

Bar-Ilan University אוניברסיטת בר-אילן

Multipliers

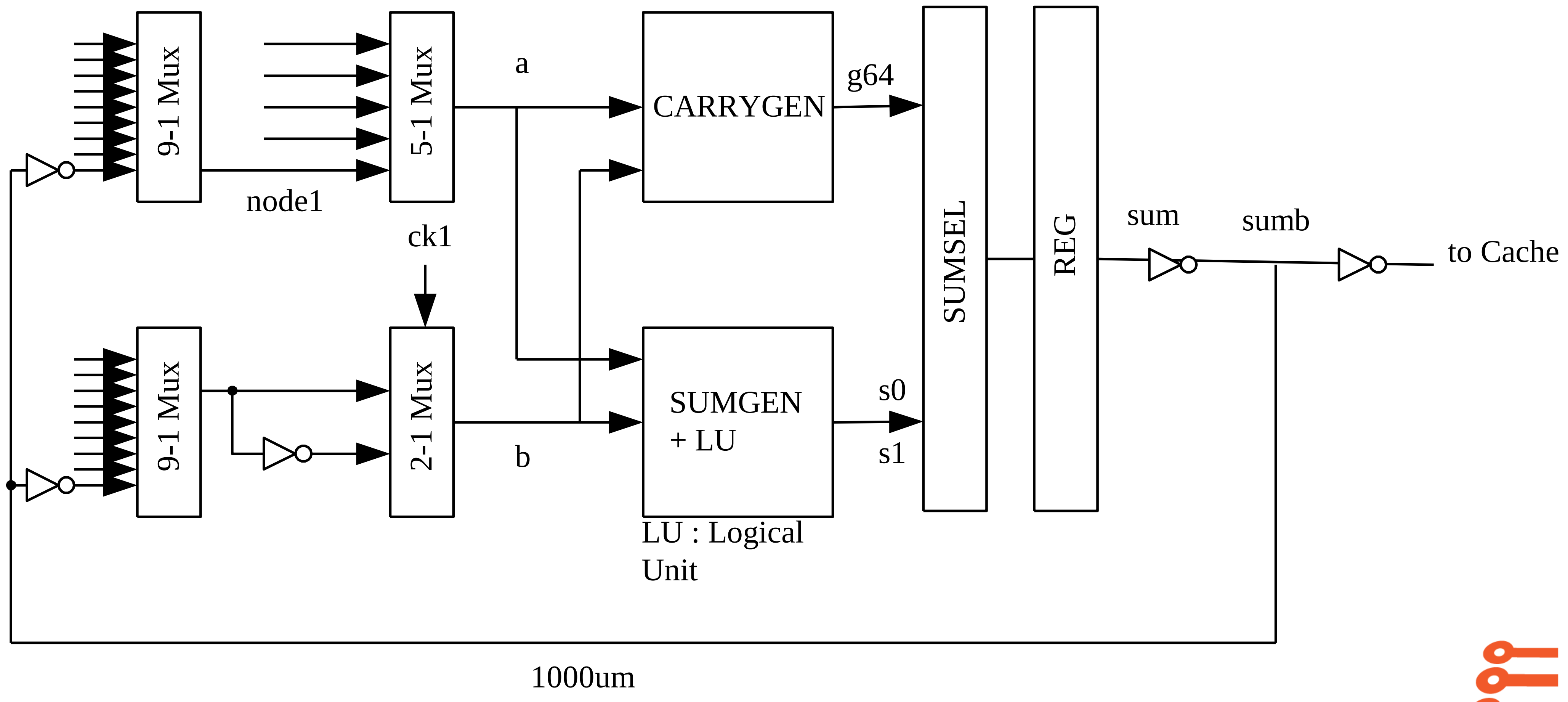
23 Emerging Nanoscaled Integrated Circuits and Systems Labs

Bar-Ilan University אוניברסיטת בר-אילן

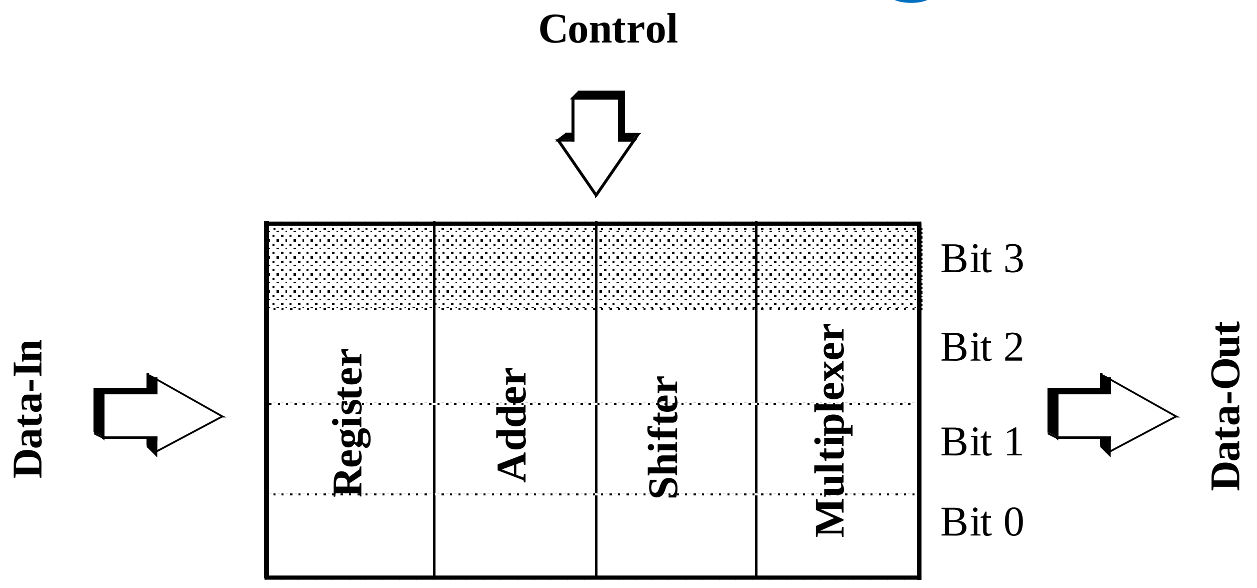
# DataPaths

# Intel Microprocessor

- Itanium has 6 integer execution units like this

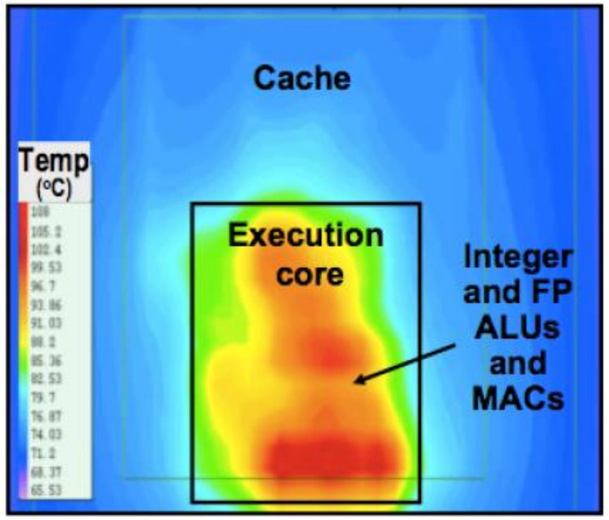


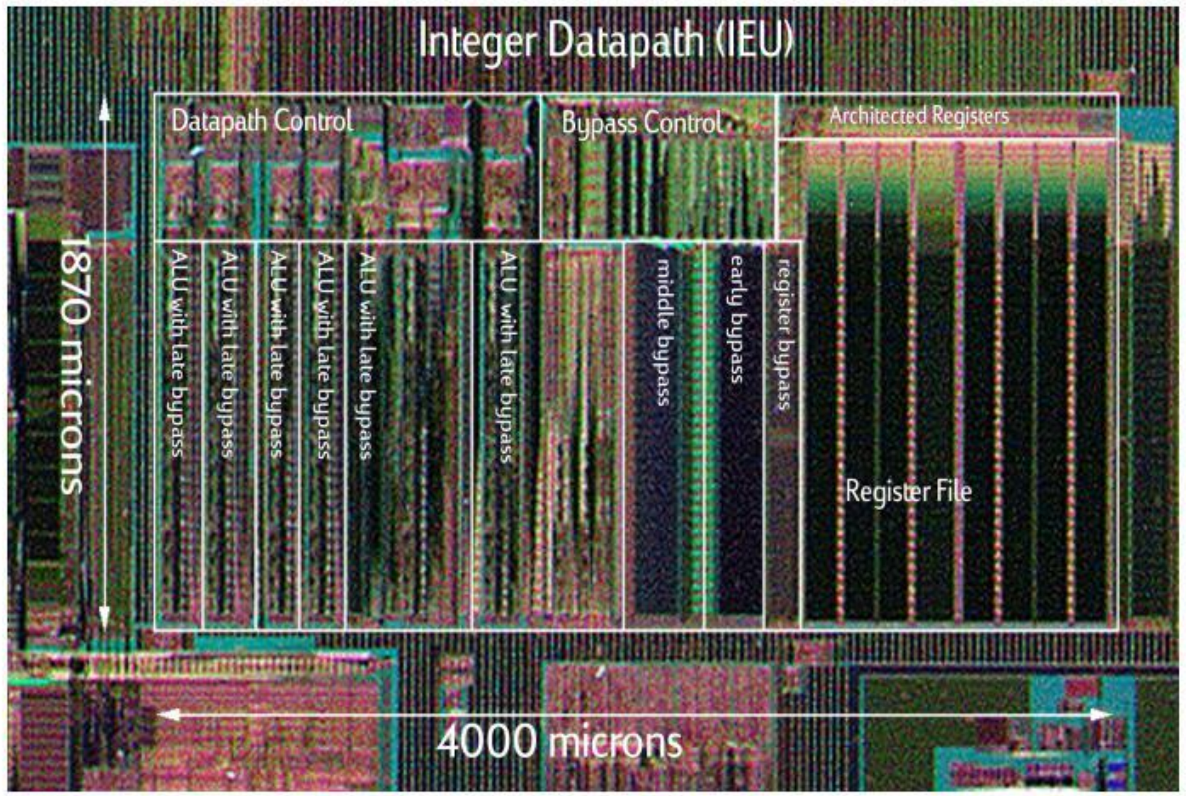
# Bit-Sliced Design



Tile identical Processor Elements

Processor thermal map





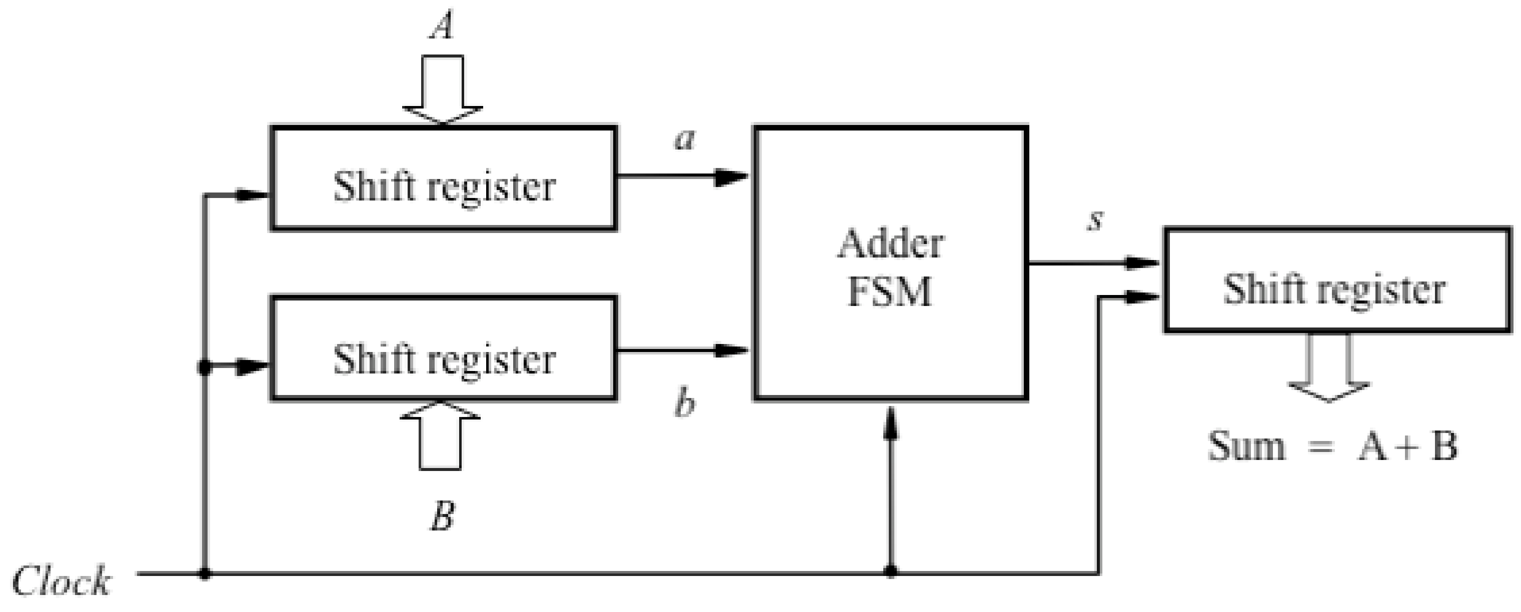
Fetzer, Orton, ISSCC'02

Design for energy efficiency!

## Adders

## Serial Adder Concept

- **At time $i$, read $a_i$ and $b_i$. Produce $s_i$ and $c_i+1$**
- **Internal state stores $c_i$. Carry bit $c_0$ is set as $c_{in}$**

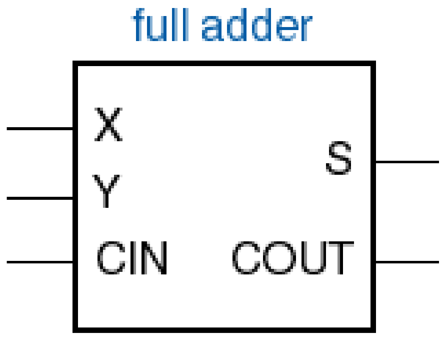


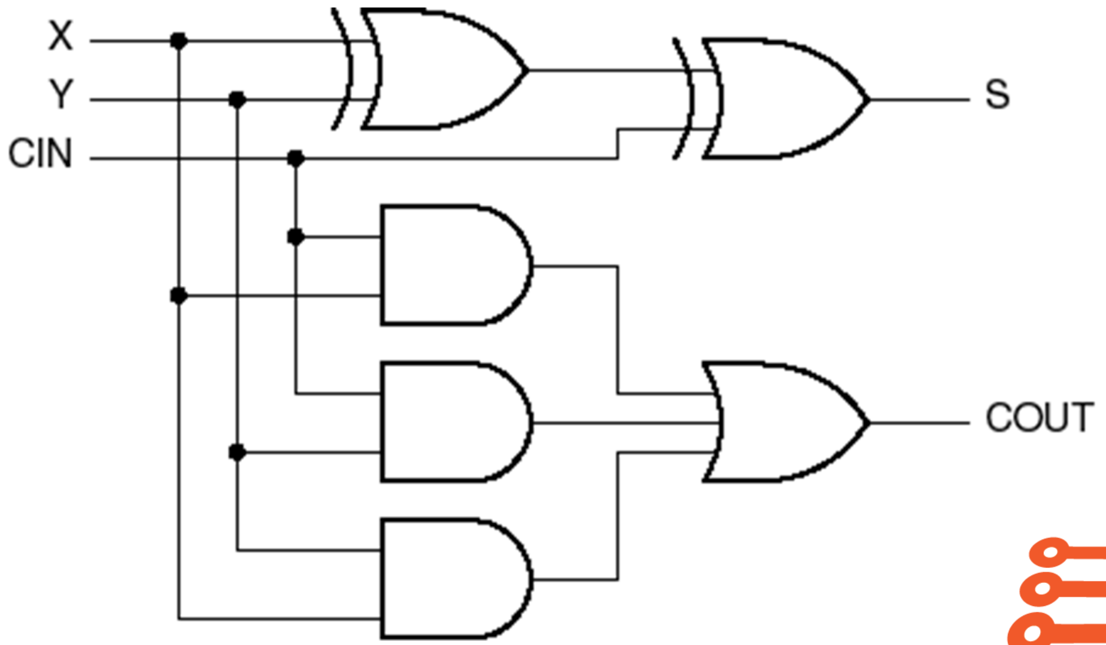
# Basic Addition Unit – Full Adder

| X | Y | Cin | S | Cout |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

$$S = x \oplus y \oplus C_{in}$$
$$\Rightarrow S = P \oplus C_{in}$$
$$C_{out} = xy + xC_{in} + yC_{in}$$
$$\Rightarrow C_{out} = G + P \cdot C_{in}$$

$$\text{Kill} = \overline{x} \cdot \overline{y}$$
$$\text{Generate} = x \cdot y$$
$$\text{Propagate} = x \oplus y$$
$$\approx x + y$$

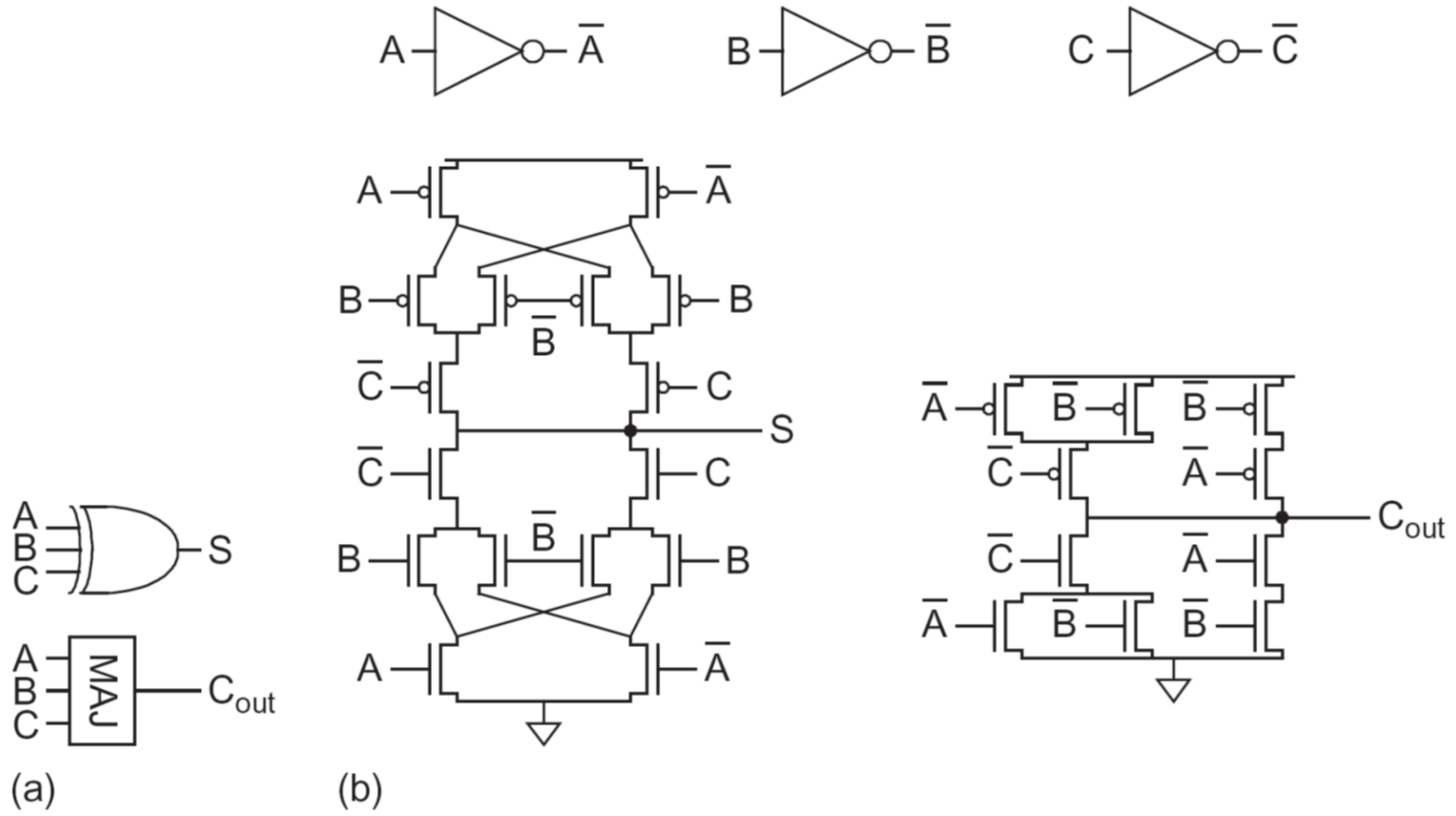




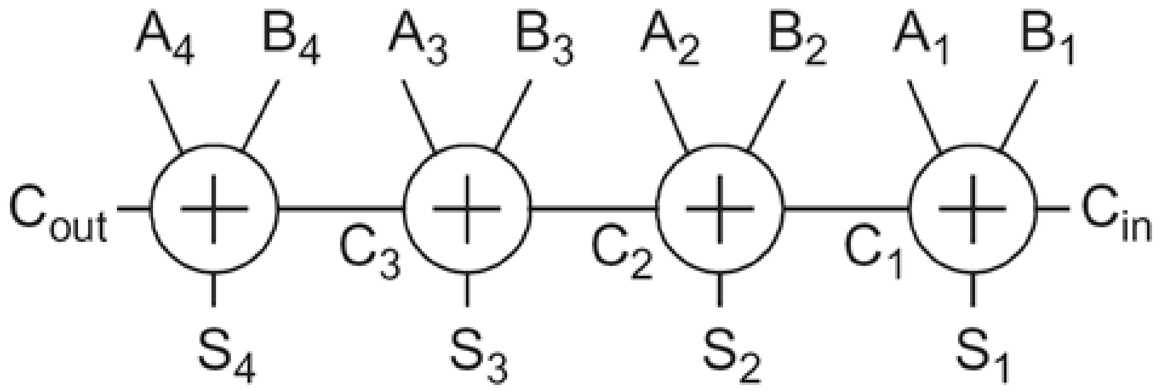
# Full-Adder Implementation

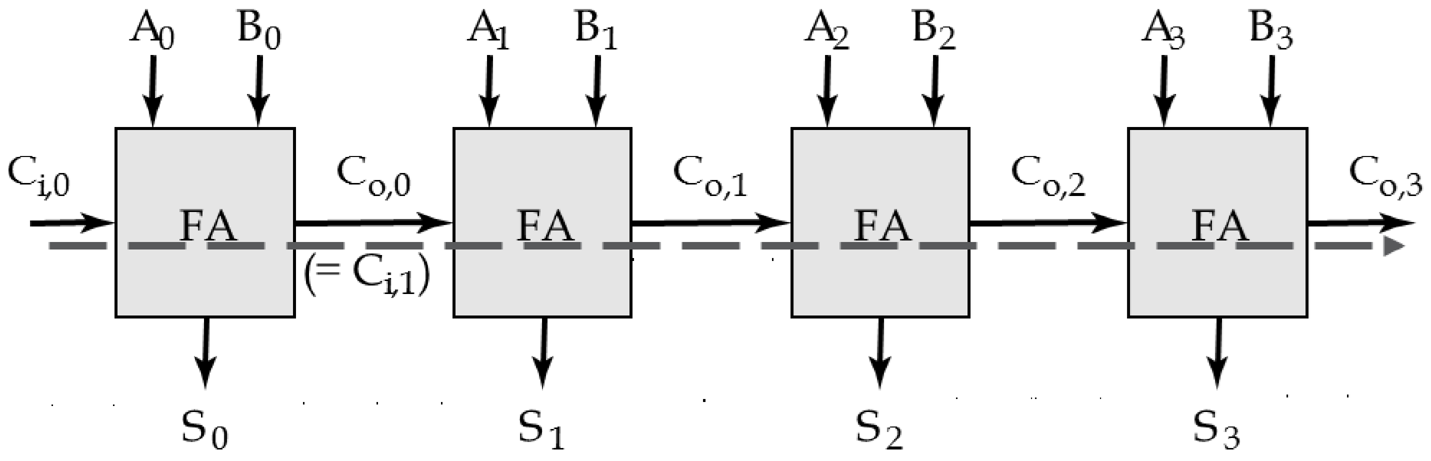
- **A full-adder is therefore a majority gate and a 3-input XOR:**

***Total: 32 Transistors***



# Ripple Carry Adder





- So it is clear, the $C_{\text{out}}$ output of the Full Adder is on the critical path.
- Can we exploit this to improve the design?

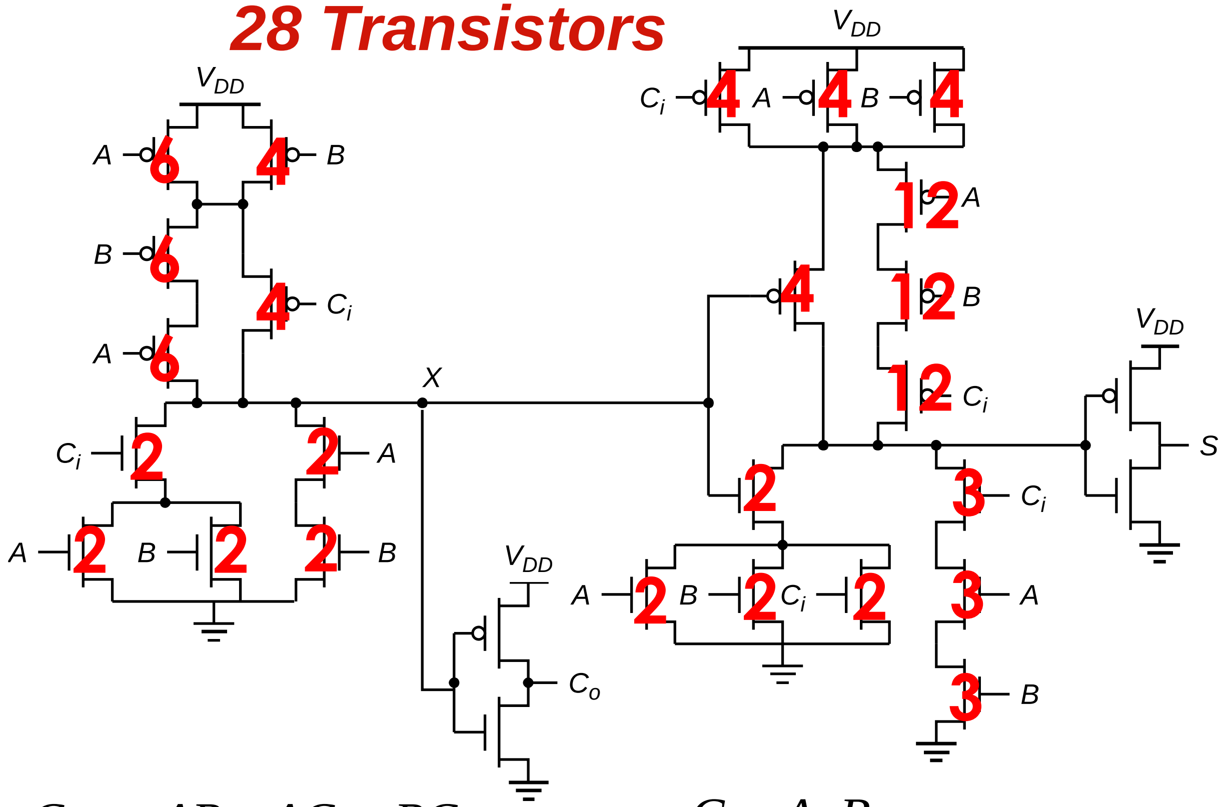
$$S = A \oplus B \oplus C_{\text{in}} =$$

$$= ABC_{\text{in}} + \left(A + B + C_{\text{in}}\right)\bar{C}_{\text{out}}$$

$t_{adder} = (N\text{-}1)t_{carry} + t_{sum}$ $\Rightarrow$ $t_d = O(N)$

# Full-Adder Implementation

*28 Transistors*



$$C_{out} = AB + AC_i + BC_i$$

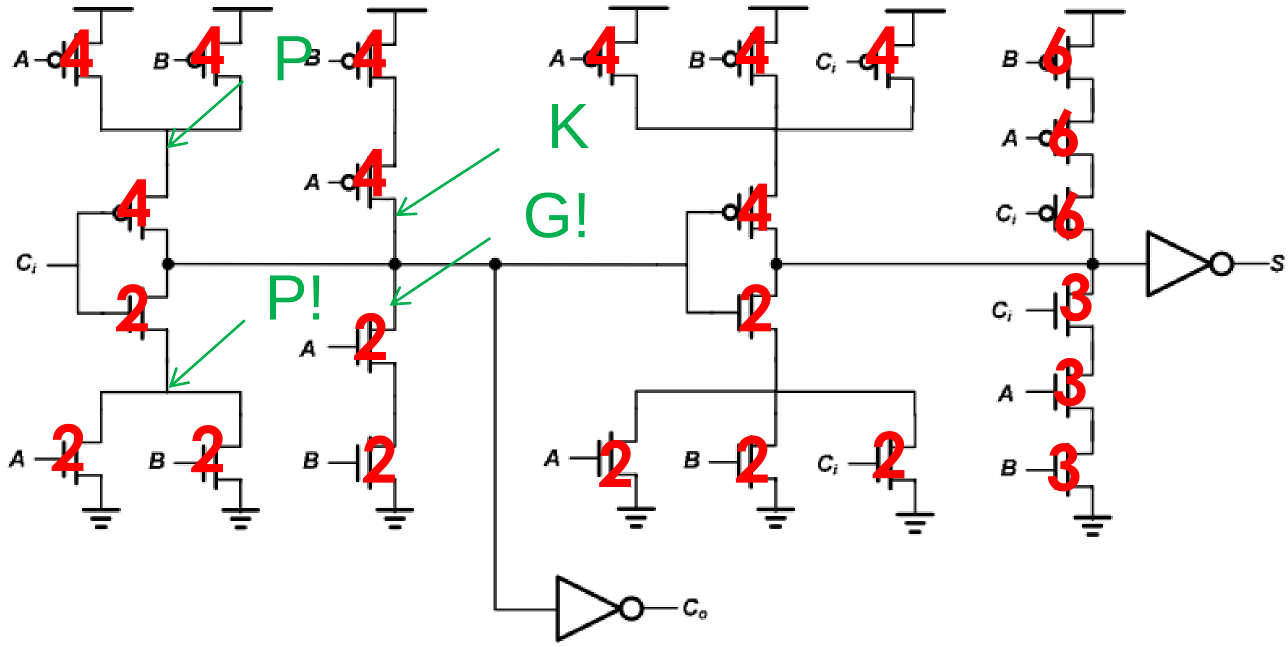
$$S = ABC_{in} + \left(A + B + C_{in}\right)\bar{C}_{out}$$

$$G = A \cdot B$$

$$P = A \oplus B$$

$$LE_{C_i} = \frac{2+4+2+3+12+4}{3} = 9$$

*24 Transistors*

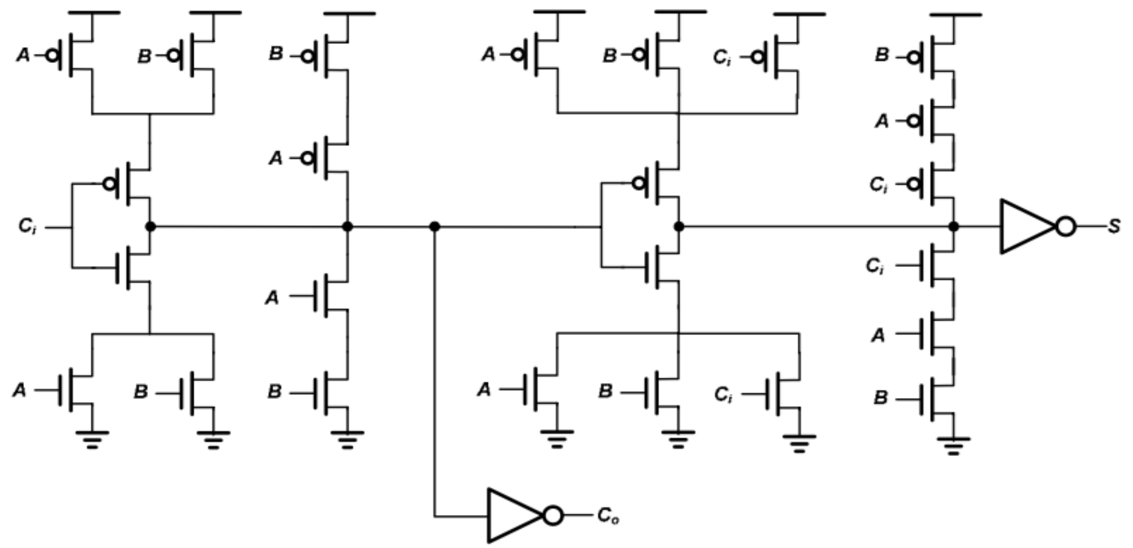


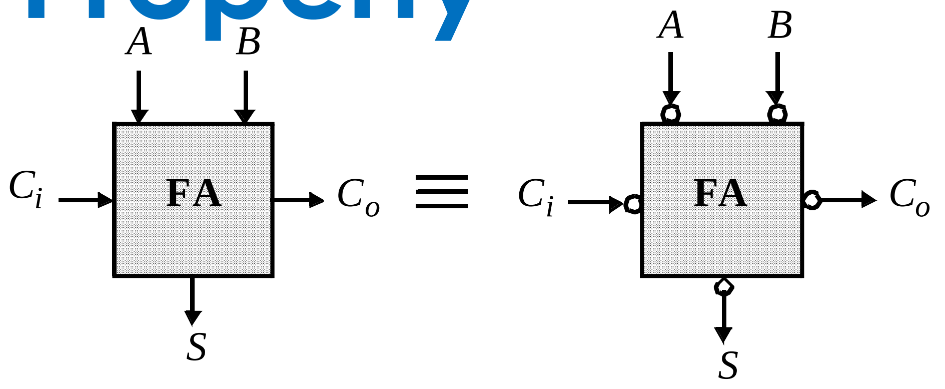
$$LE_{C_i} = \frac{2+4+2+4+6+3}{3} = 7$$

…BUT ~64 stages to propagate

i.e. $PE_{opt} = 4^{64}$

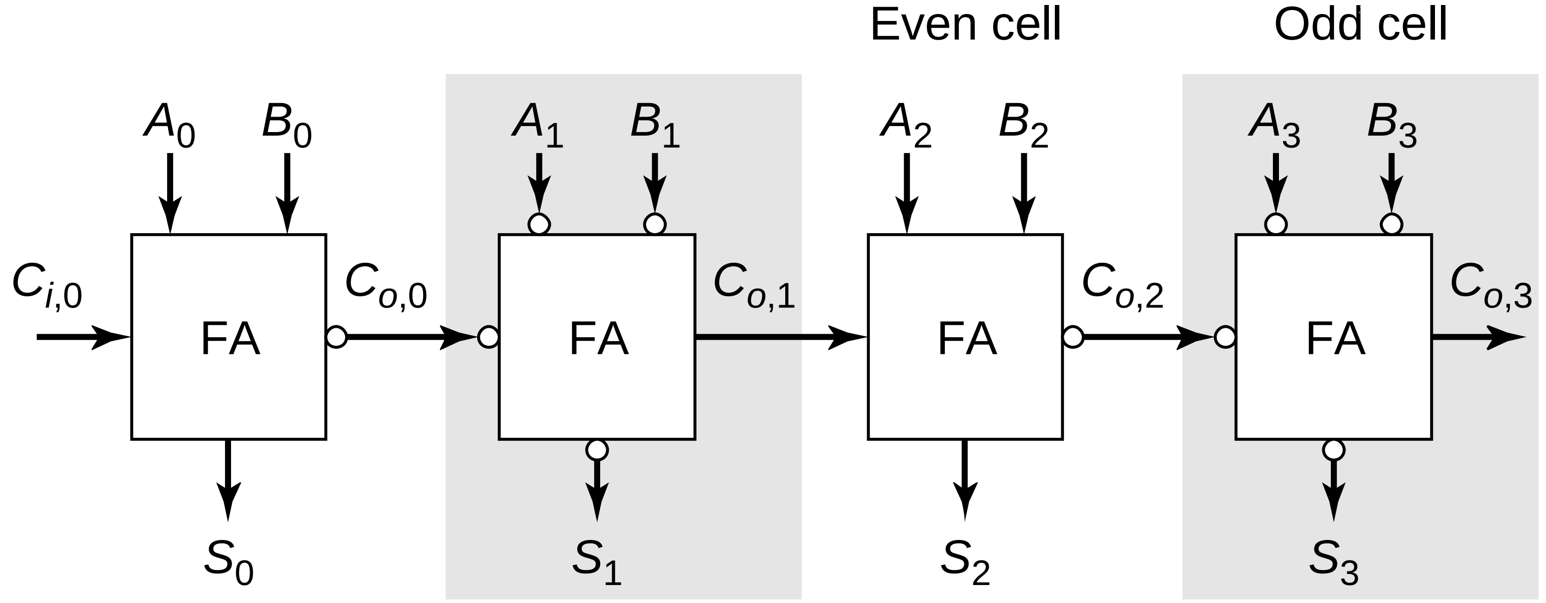
# Exploiting the Inversion Property





$$\bar{S}(A,B,C_i) = S(\bar{A},\bar{B},\bar{C}_i)$$

$$\bar{C}_o(A,B,C_i) = C_o(\bar{A},\bar{B},\bar{C}_i)$$



We saved the inverter, so $PE_{opt}=4^{32}$

# Sizing the Mirror Adder

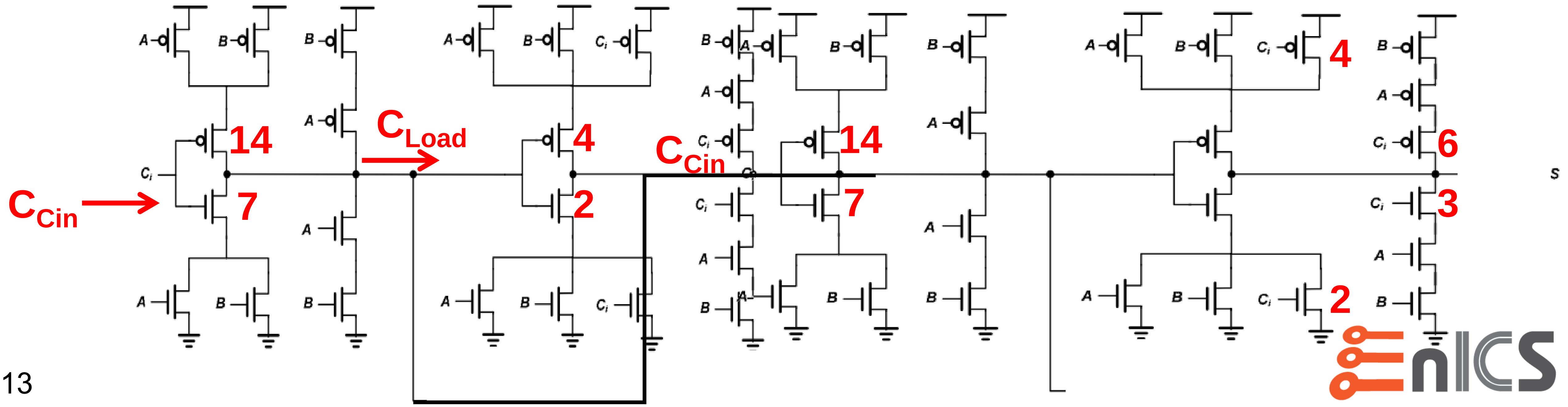
$$LE_{Cin} = \frac{4+2}{3} = 2$$

- **Problem: How can we make a high speed bitslice layout?**
  - If we upsize each stage according to Logical Effort, we will have non-identical bitslices.
  - Such upsizing will result in huge gates.

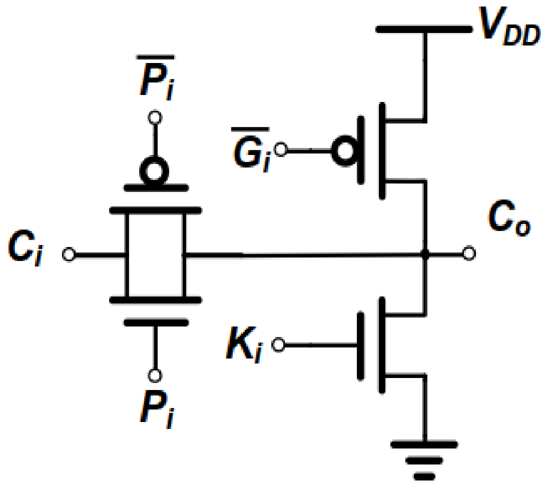
$$EF_{FA,Cin} = \frac{LE_{Cin} \cdot C_{L,Cout}}{C_{Cin}} \Rightarrow \left.\frac{C_{L,Cout}}{C_{Cin}}\right|_{EF=4} = 2$$

$$C_{L,Cout} = 6 + C_{Cin} + 6 + 9 \Rightarrow C_{Cin} = 21$$

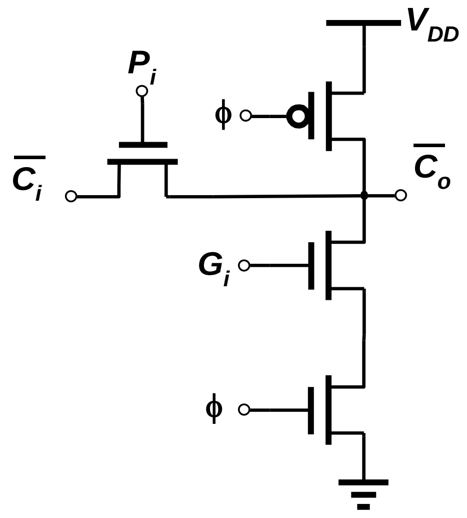
- **Why not design the adder to inherently achieve optimal Electrical Effort** ($EF_{opt}=4$)**?**
  - Assume everything not on the carry path can be sized like a minimum inverter!



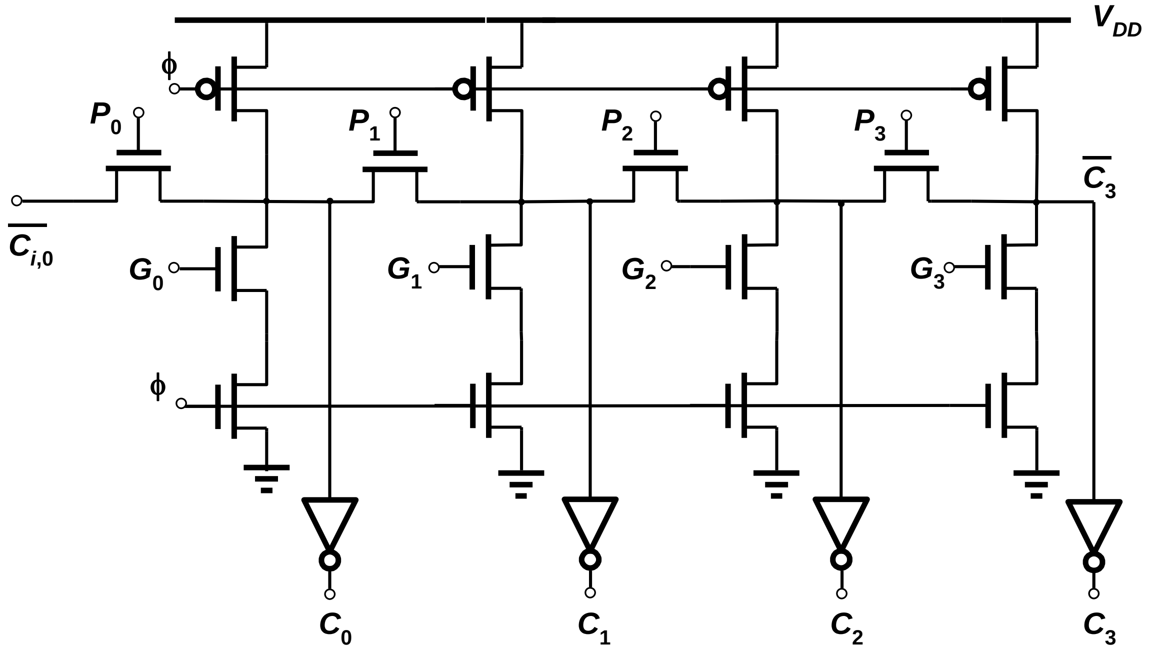
# Manchester Carry-Chain Adder



Static Circuits



Dynamic Circuit

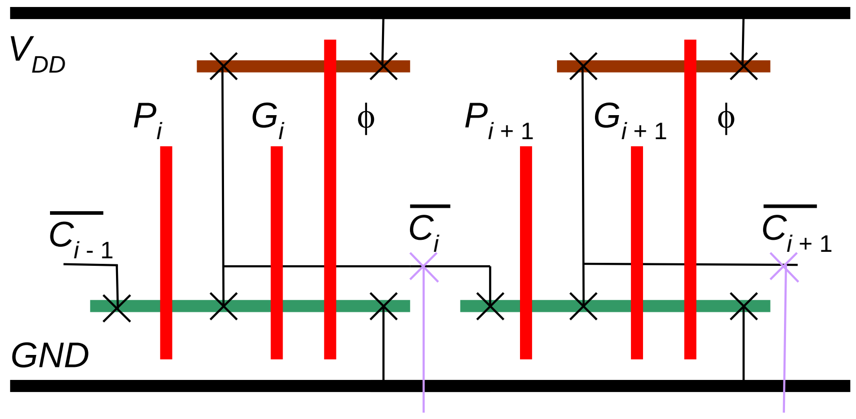


$$t_P = 0.69 \sum_{i=1}^{N} C_i \cdot \left( \sum_{j=1}^{i} R_j \right)$$

$$= 0.69 \frac{N(N+1)}{2} RC$$

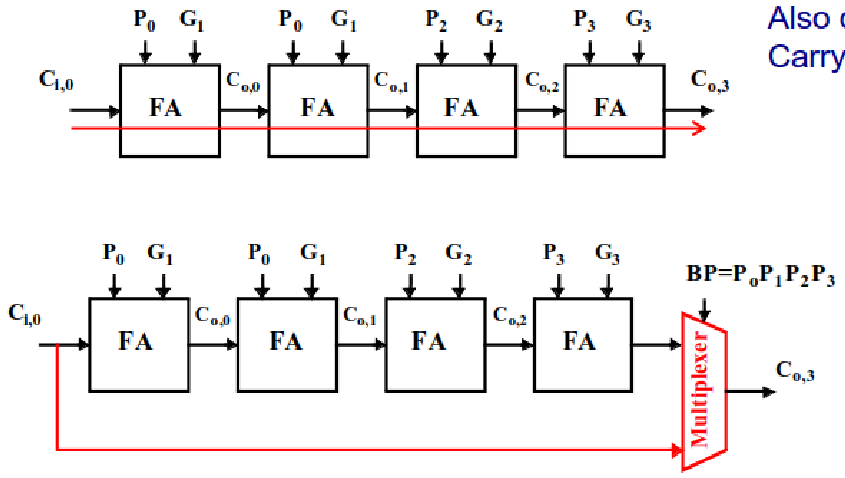
where $R_j = R, \quad C_i = C$

Propagate/Generate Row



Inverter/Sum Row

# Carry-Skip (Carry Bypass) Adder

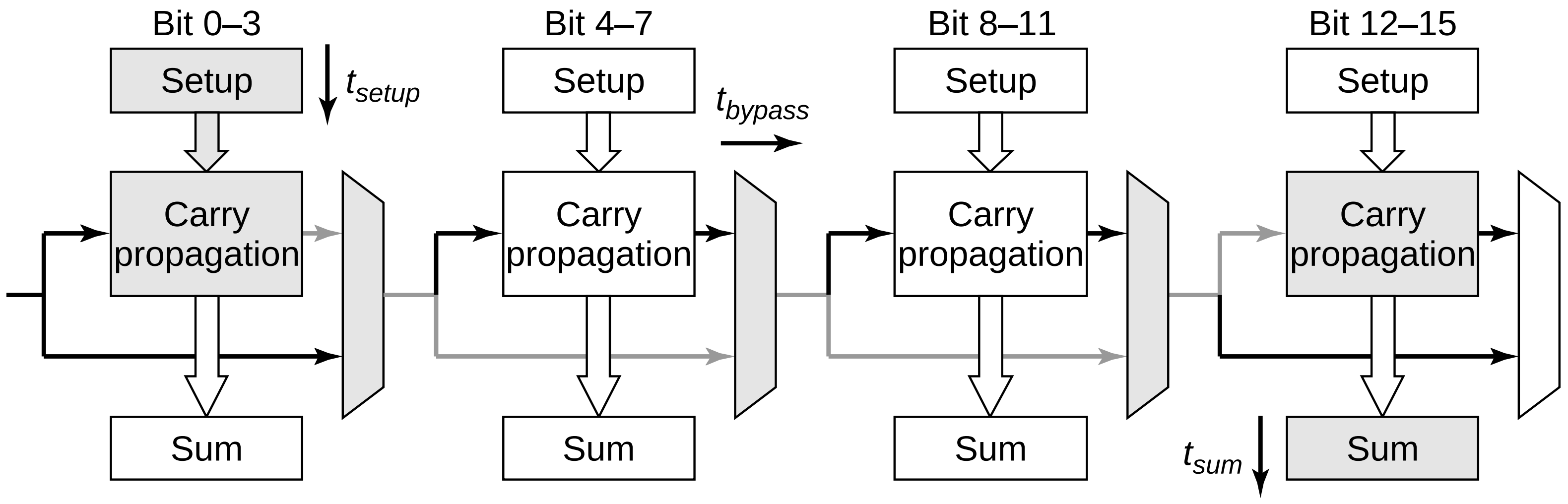


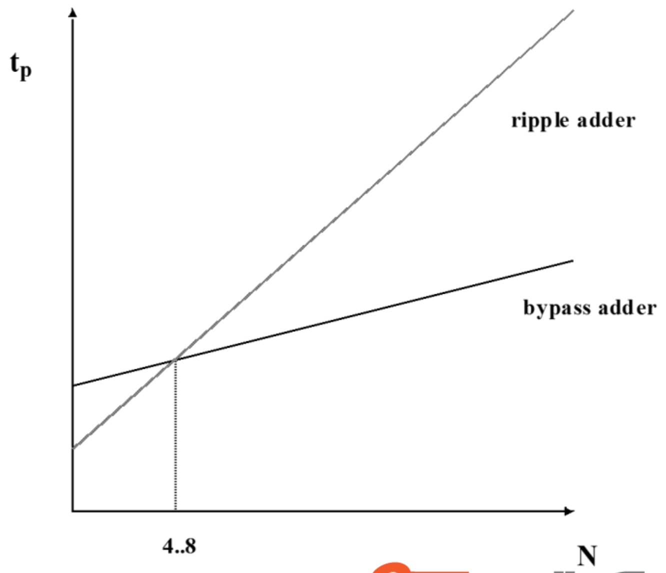
Also called Carry-Skip

*M Sections of (N/M) Bits Each*

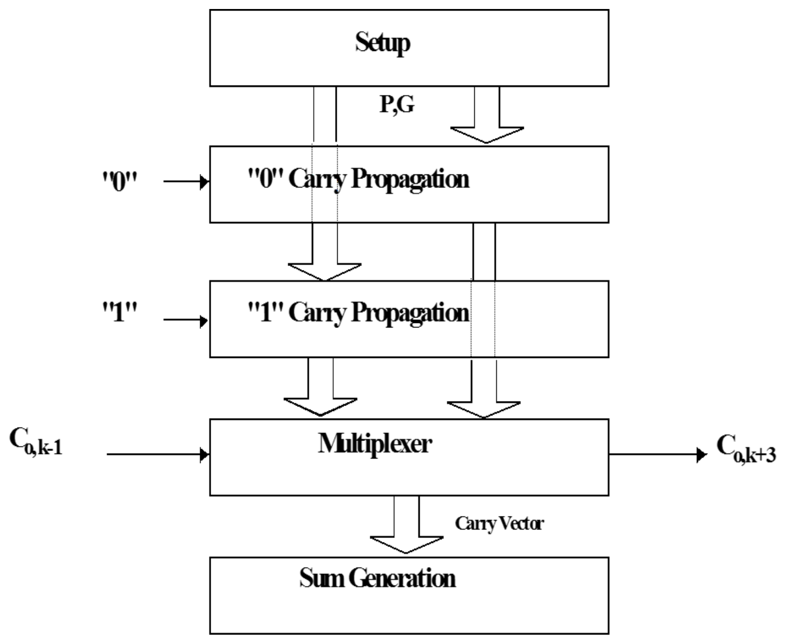
$$t_{adder} = t_{setup} + \left(\frac{N}{M} - 1\right) t_{carry} + (M-1) t_{bypass} + t_{sum}$$

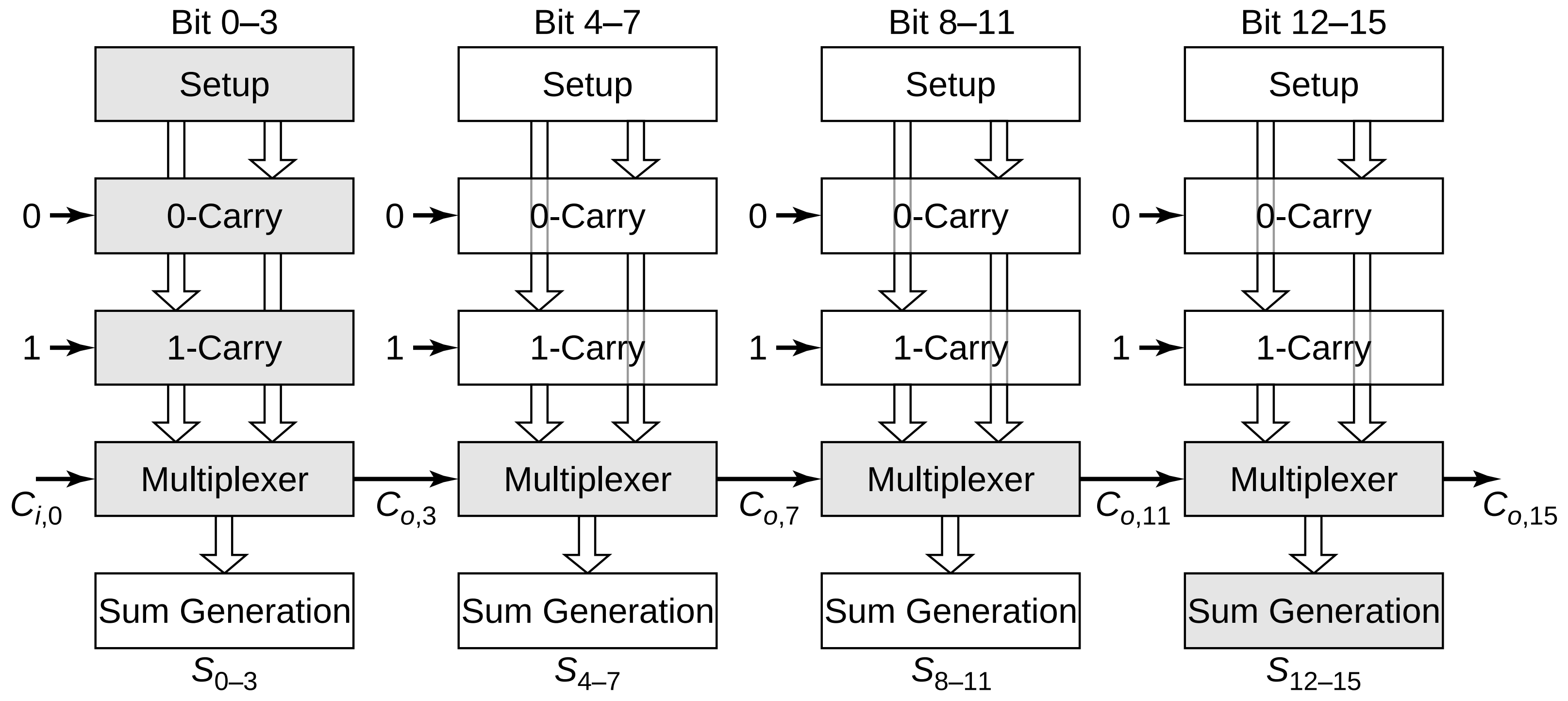
Idea: If (P0 and P1 and P2 and P3 = 1) then $C_{o3} = C_0$, else "kill" or "generate".





# Carry-Select Adder



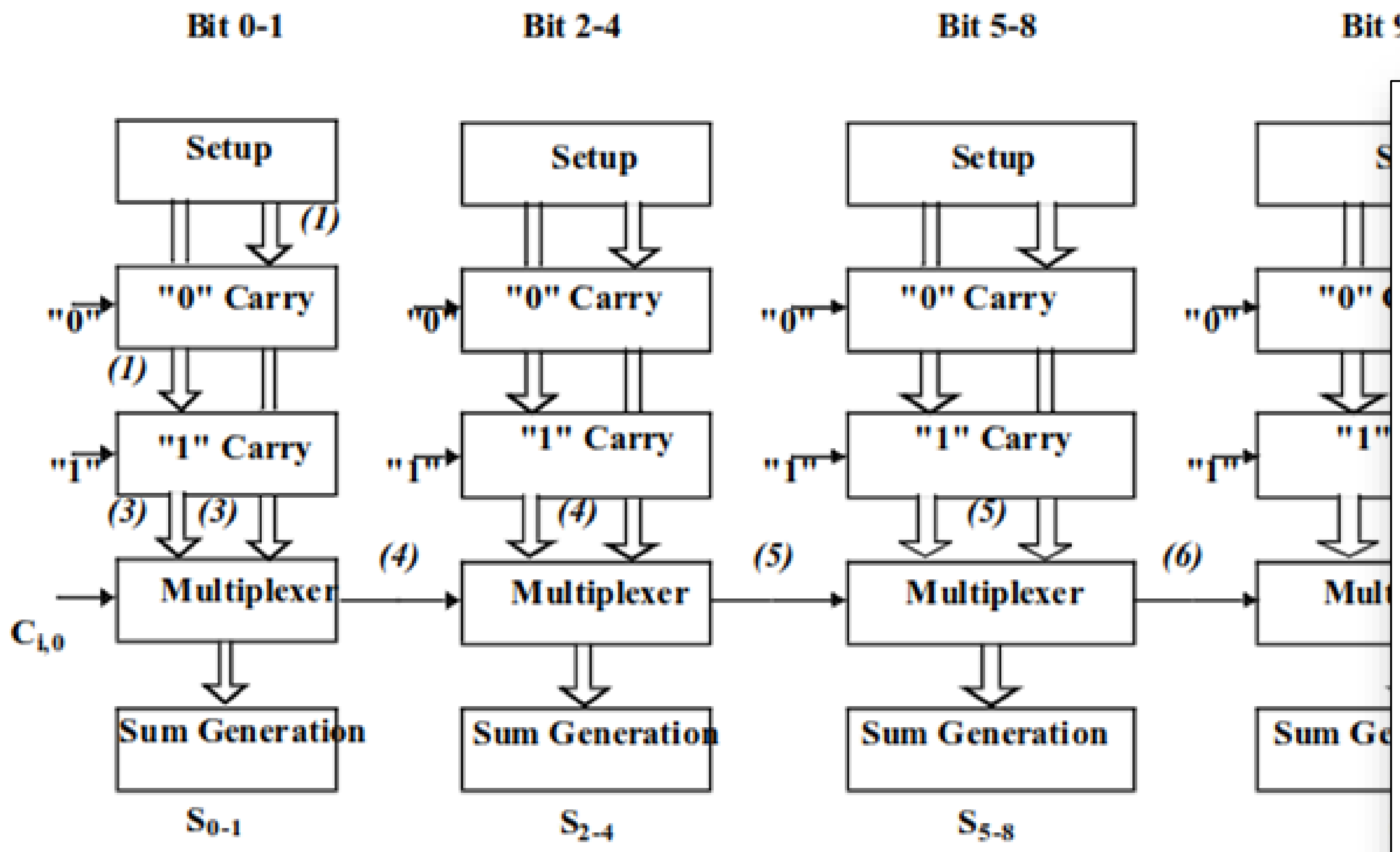


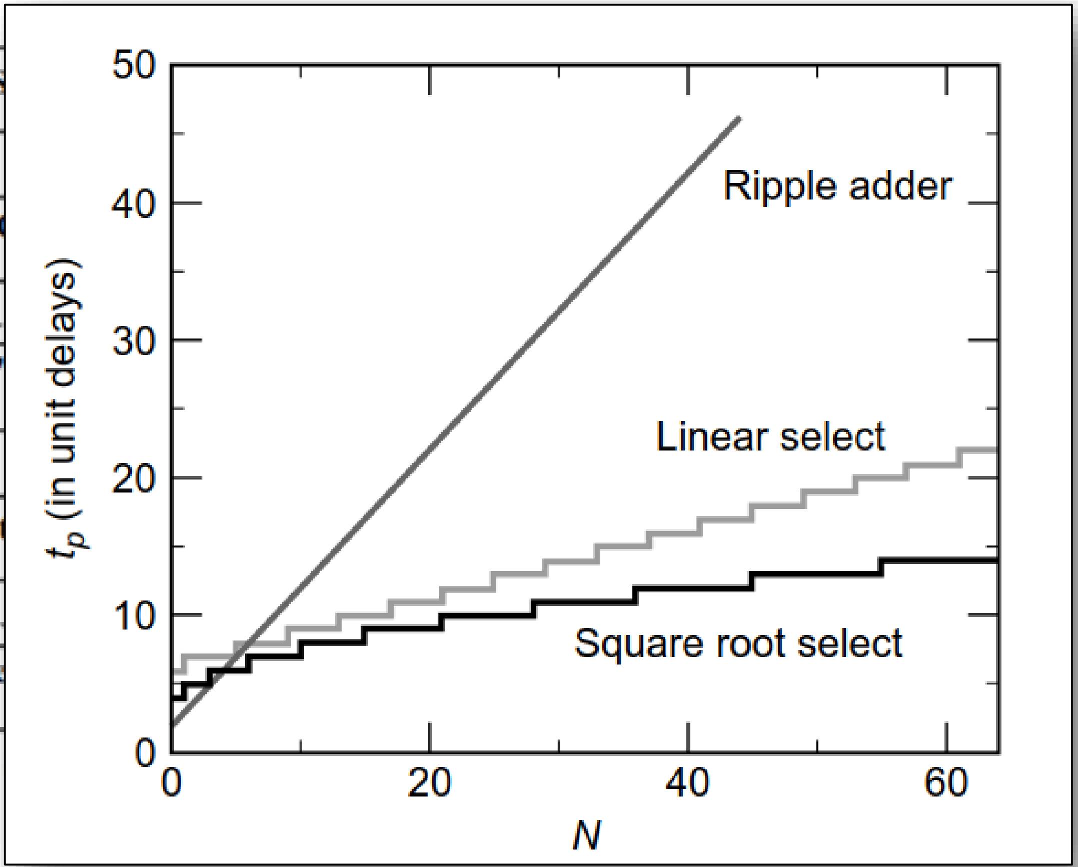
Let’s guess the answer for each value of the carry.

*N*-bit input with *M* CSA blocks

$$t_{adder} = t_{setup} + \frac{N}{M} t_{carry} + M \cdot t_{mux} + t_{sum}$$

# Square Root Carry Select





$$t_{adder} = t_{setup} + Mt_{carry} + \sqrt{2N}t_{mux} + t_{sum}$$

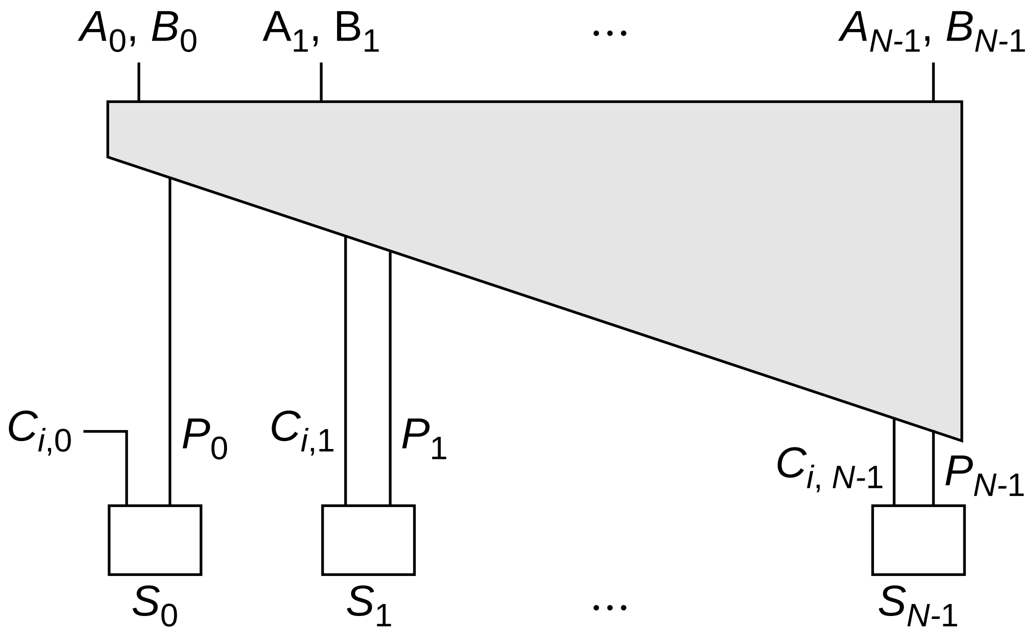
# Carry Lookahead Adder – Basic Idea

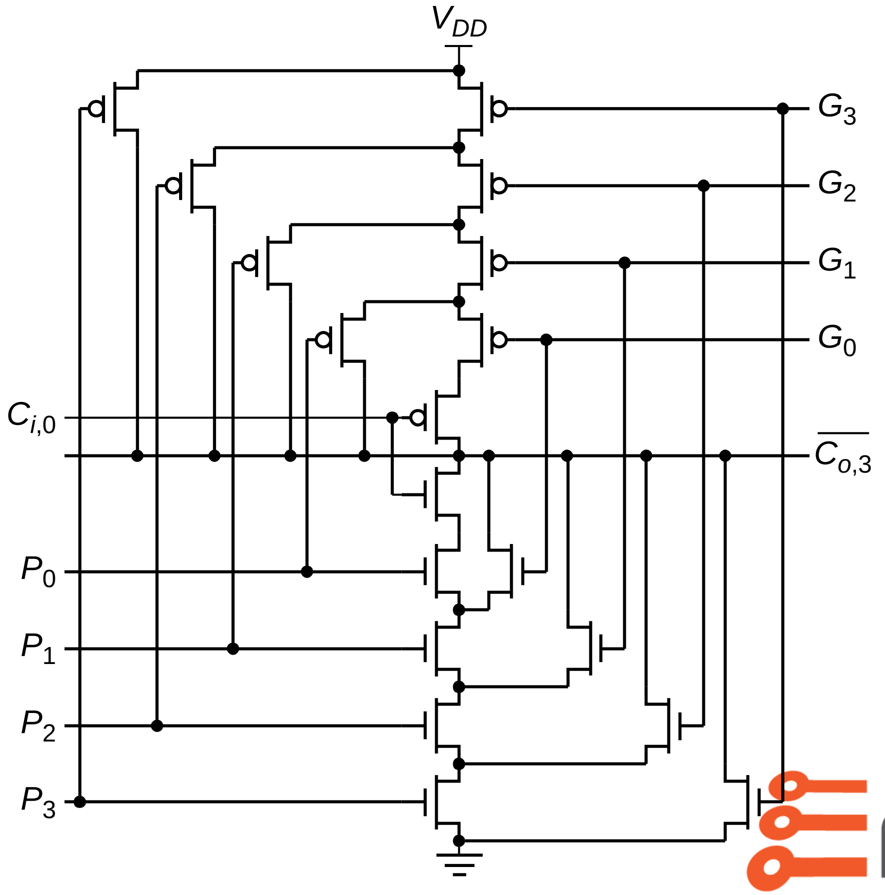
- **Problem – $C_{o,k}$ takes approximately $k$ gate delays to ripple.**
- **Question – can we calculate the carry without any ripple?**

$$C_{O,k} = f(A_k, B_k, C_{O,k-1}) = G_k + P_k \cdot C_{O,k-1}$$

$$C_{O,k} = G_k + P_k \cdot (G_{k-1} + P_{k-1} \cdot C_{O,k-2})$$

$$C_{O,k} = G_k + P_k \cdot (G_{k-1} + P_{k-1} \cdot (\ldots + P_1(G_0 + P_0 C_{i,0})))$$





# Logarithmic CLA (Tree Adder)

- **Can we reduce the complexity of calculating $P_i$, $G_i$ ?**

$$P_{1:0} = P_1 \cdot P_0 \quad G_{1:0} = G_1 + P_1 \cdot G_0$$

$$\Rightarrow C_{out,1} = G_{1:0} + P_{1:0} C_{in,0}$$

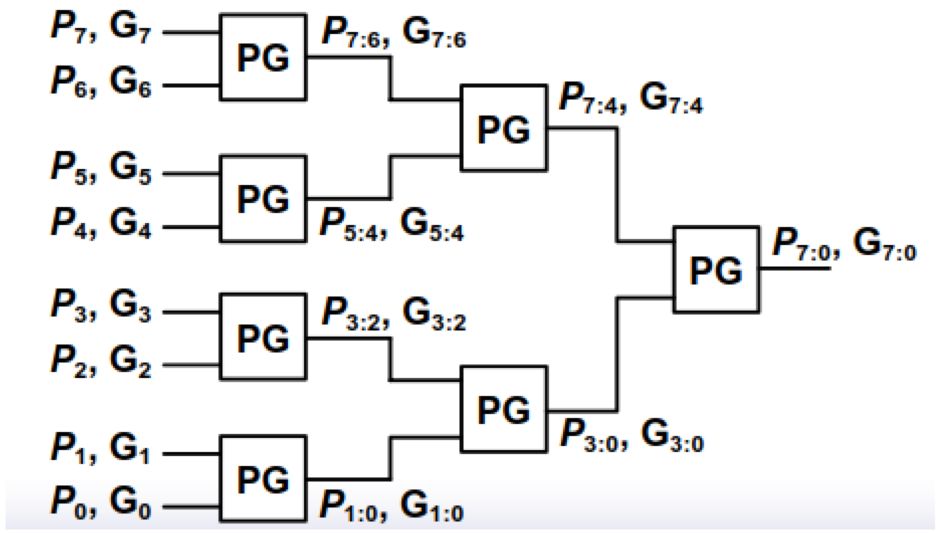
$$P_{3:2} = P_3 \cdot P_2 \quad G_{3:2} = G_3 + P_3 \cdot G_2$$

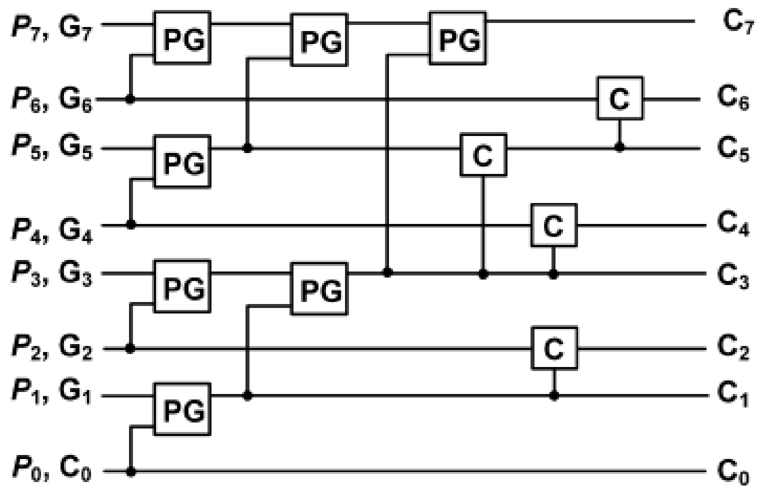
$$\Rightarrow C_{out,3} = G_{3:2} + P_{3:2} C_{in,2}$$

$$\Downarrow$$

$$P_{3:0} = P_{3:2} \cdot P_{1:0} \quad G_{3:0} = G_{3:2} + P_{3:2} \cdot G_{1:0}$$

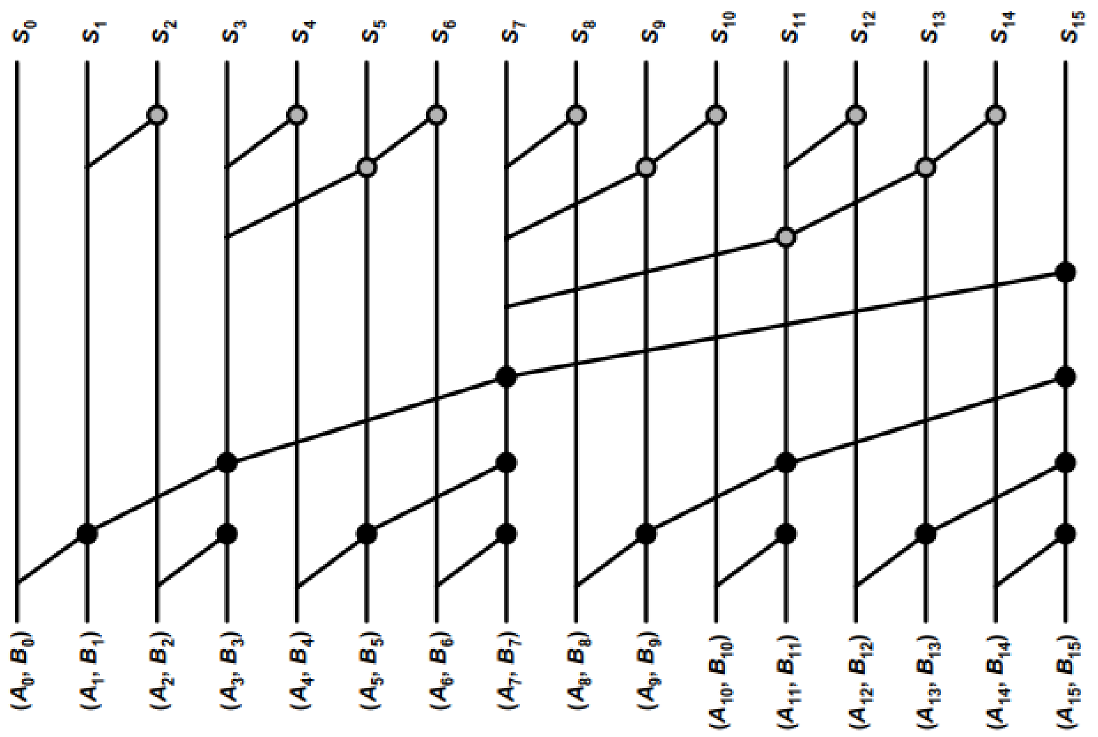
$$\Rightarrow C_{out,3} = G_{3:0} + P_{3:0} C_{in,0}$$





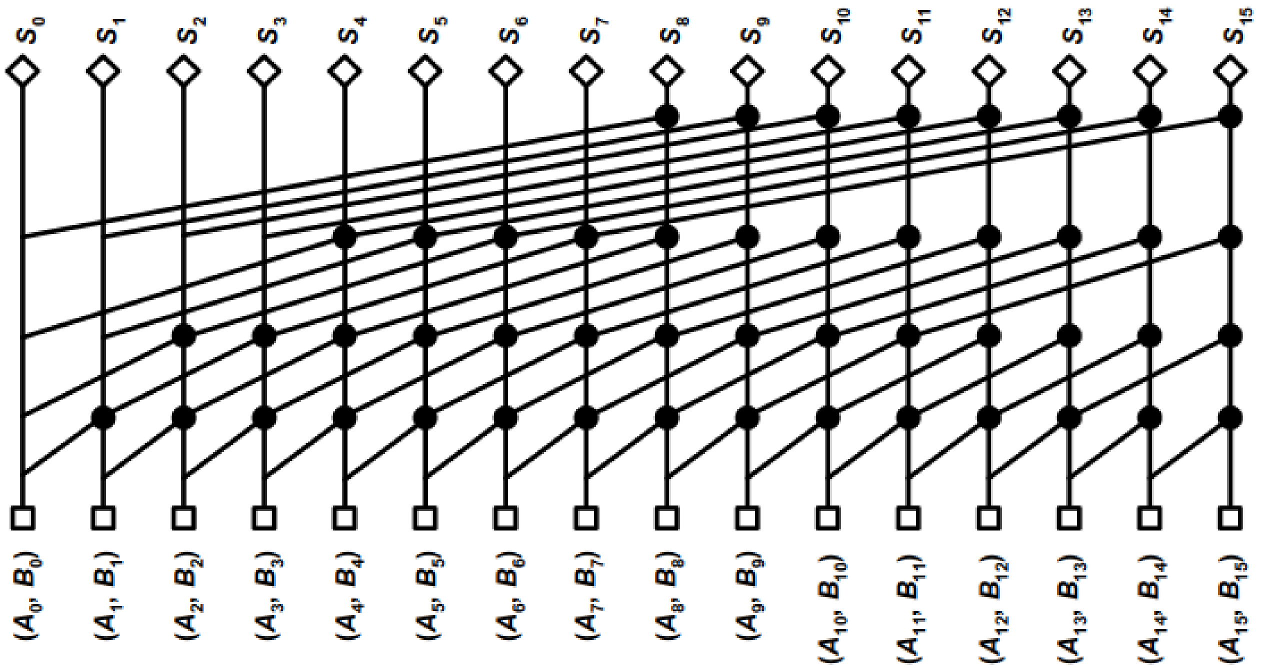
# Logarithmic CLA (Tree Adder)

- **Many ways to construct these CLA or tree adders, based on:**
  - Radix: How many bits combined in each gate
  - Tree Depth: How many stages of logic to the final carry ($>=\log_{\text{radix}}N$)
  - Fanout: Maximal logic branching in tree



$s_0$ $s_1$ $s_2$ $s_3$ $s_4$ $s_5$ $s_6$ $s_7$ $s_8$ $s_9$ $s_{10}$ $s_{11}$ $s_{12}$ $s_{13}$ $s_{14}$ $s_{15}$
$(A_0, B_0)$ $(A_1, B_1)$ $(A_2, B_2)$ $(A_3, B_3)$ $(A_4, B_4)$ $(A_5, B_5)$ $(A_6, B_6)$ $(A_7, B_7)$ $(A_8, B_8)$ $(A_9, B_9)$ $(A_{10}, B_{10})$ $(A_{11}, B_{11})$ $(A_{12}, B_{12})$ $(A_{13}, B_{13})$ $(A_{14}, B_{14})$ $(A_{15}, B_{15})$

Brent-Kung Tree



$s_0$ $s_1$ $s_2$ $s_3$ $s_4$ $s_5$ $s_6$ $s_7$ $s_8$ $s_9$ $s_{10}$ $s_{11}$ $s_{12}$ $s_{13}$ $s_{14}$ $s_{15}$
$(A_0, B_0)$ $(A_1, B_1)$ $(A_2, B_2)$ $(A_3, B_3)$ $(A_4, B_4)$ $(A_5, B_5)$ $(A_6, B_6)$ $(A_7, B_7)$ $(A_8, B_8)$ $(A_9, B_9)$ $(A_{10}, B_{10})$ $(A_{11}, B_{11})$ $(A_{12}, B_{12})$ $(A_{13}, B_{13})$ $(A_{14}, B_{14})$ $(A_{15}, B_{15})$

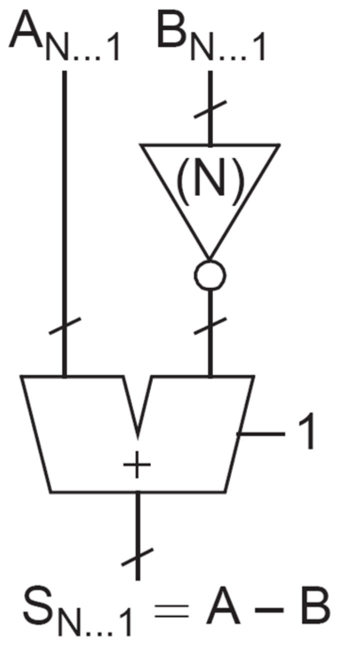
16-bit radix-2 Kogge-Stone tree

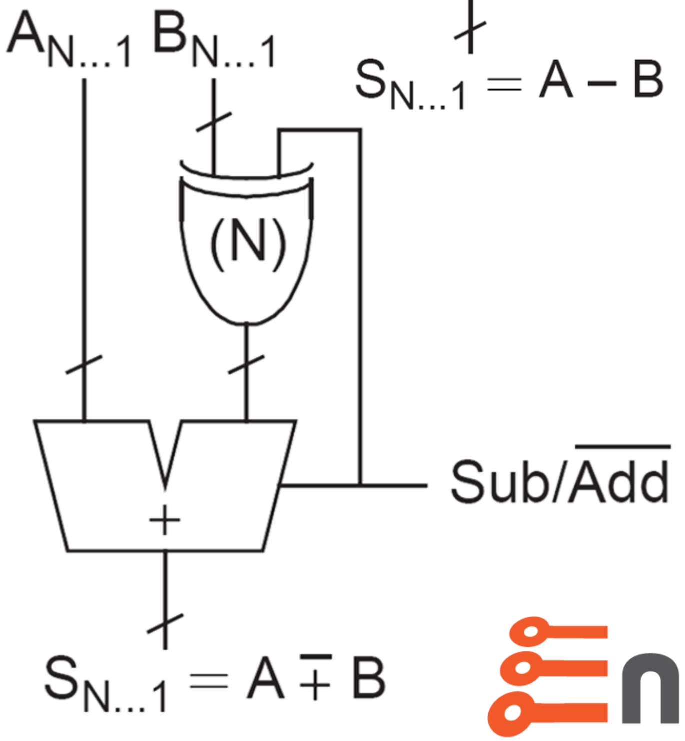
# Subtraction

- **To subtract two's complement, just remember that:**

$$-x = \overline{x} + 1 \Rightarrow A - B = A + \overline{B} + 1$$

- **So to subtract:**
  - Invert one of the operands.
  - Add a carry in to the first bit.
- **Therefore, to provide an adder/subtractor:**
  - Add an XOR gate to the B-input
  - Use the sub/add selector to the XOR and carry in.





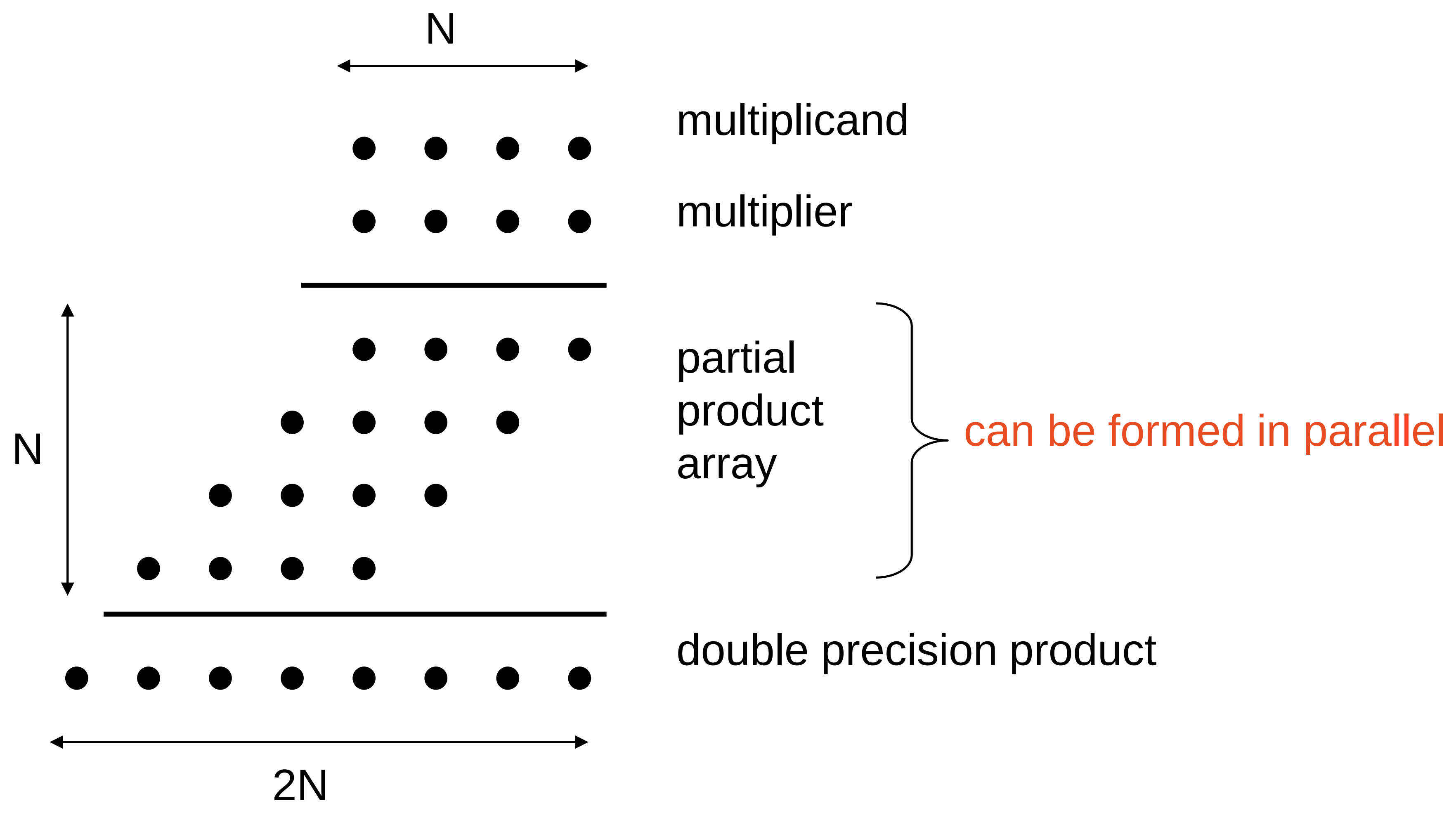
## Multipliers

## Grade School Multiplication

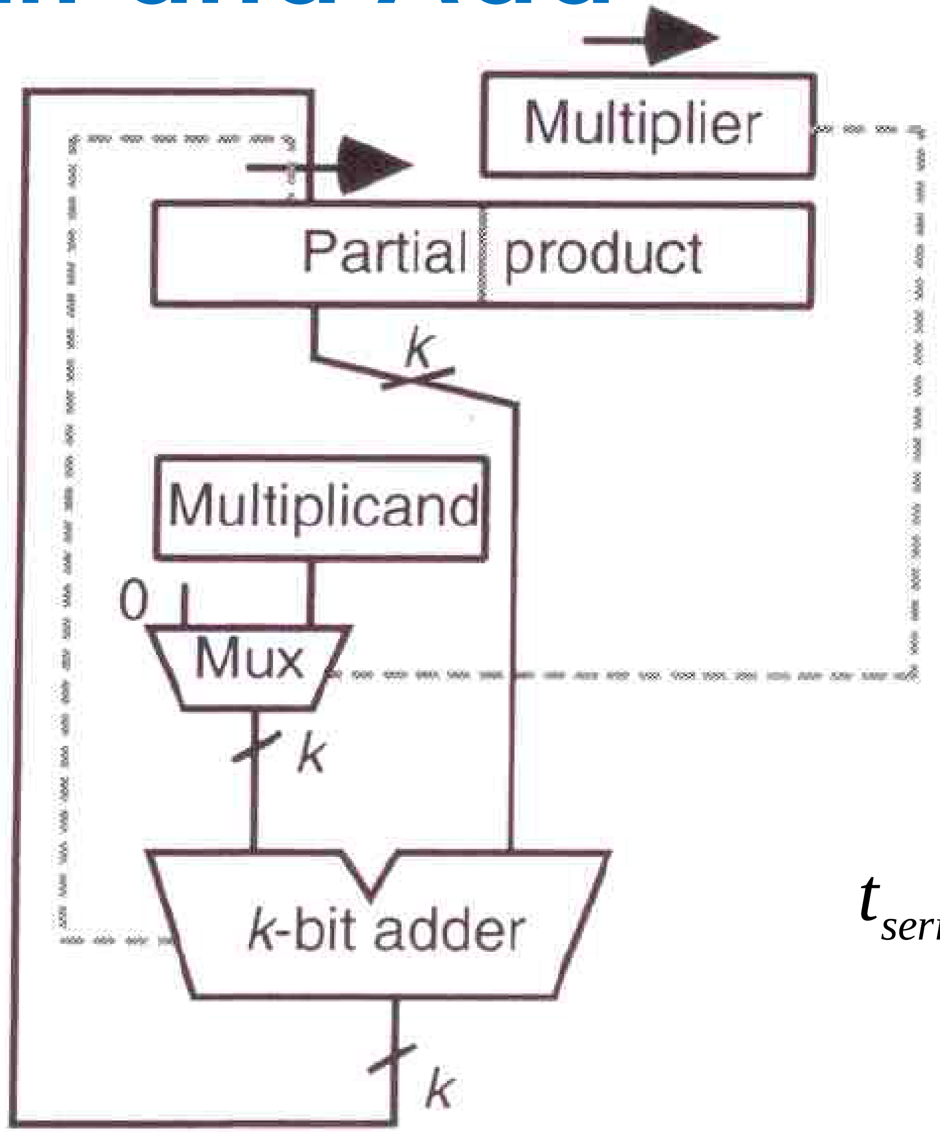
# Multiplication using serial addition

$$
\begin{array}{c|ccccccccc|l}
 & & & & 1 & 0 & 1 & 0 & 1 & 0 & \text{Multiplicand} \\
\text{x} & & & & & & 1 & 0 & 1 & 1 & \text{Multiplier} \\
\hline
 & & & & 1 & 0 & 1 & 0 & 1 & 0 & \\
 & & & 1 & 0 & 1 & 0 & 1 & 0 & & \\
 & & 0 & 0 & 0 & 0 & 0 & 0 & & & \text{Partial products} \\
+ & 1 & 0 & 1 & 0 & 1 & 0 & & & & \\
\hline
 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & \text{Result}
\end{array}
$$

# Binary Multiplication

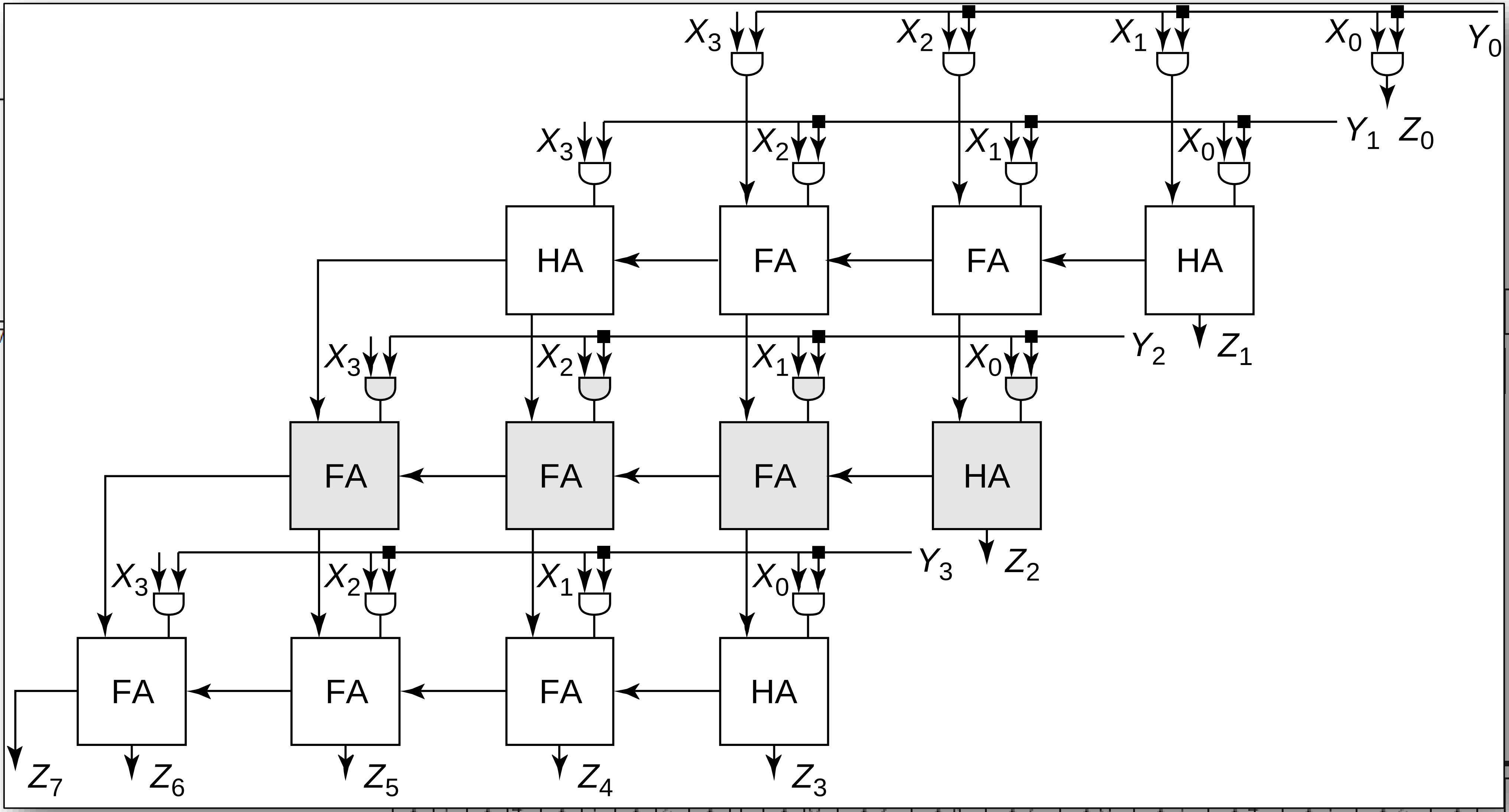


## Serial Shift and Add

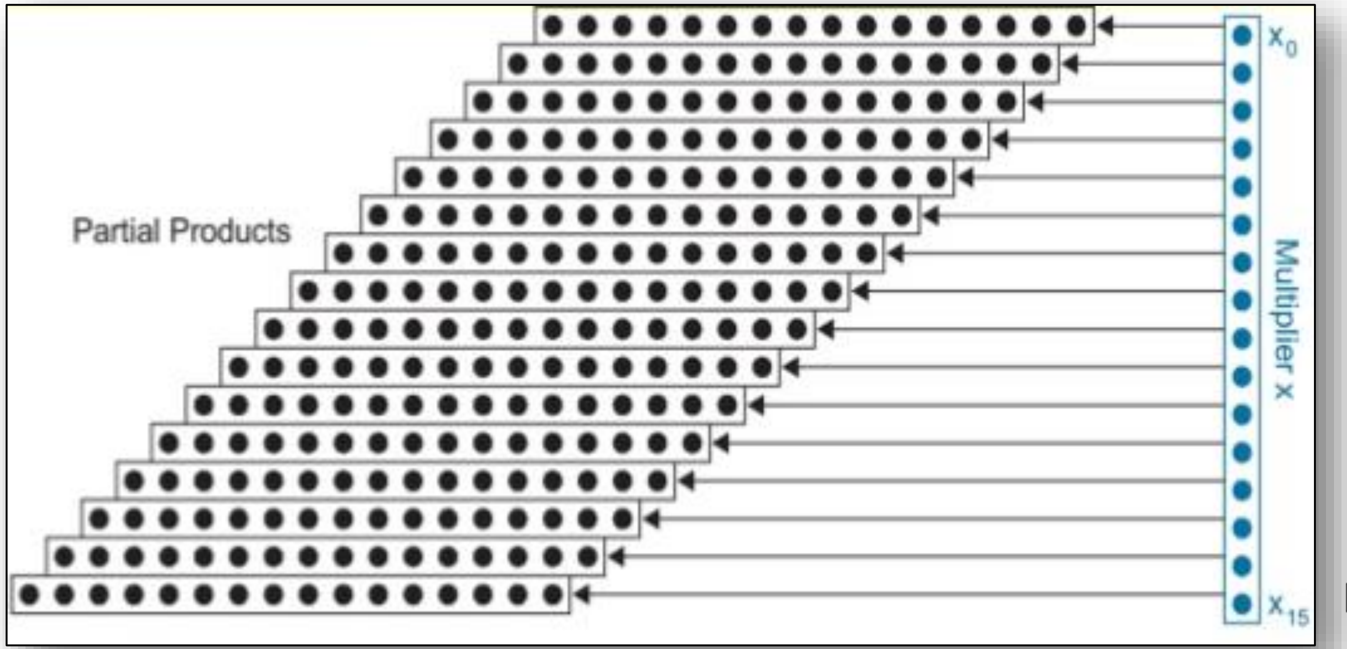


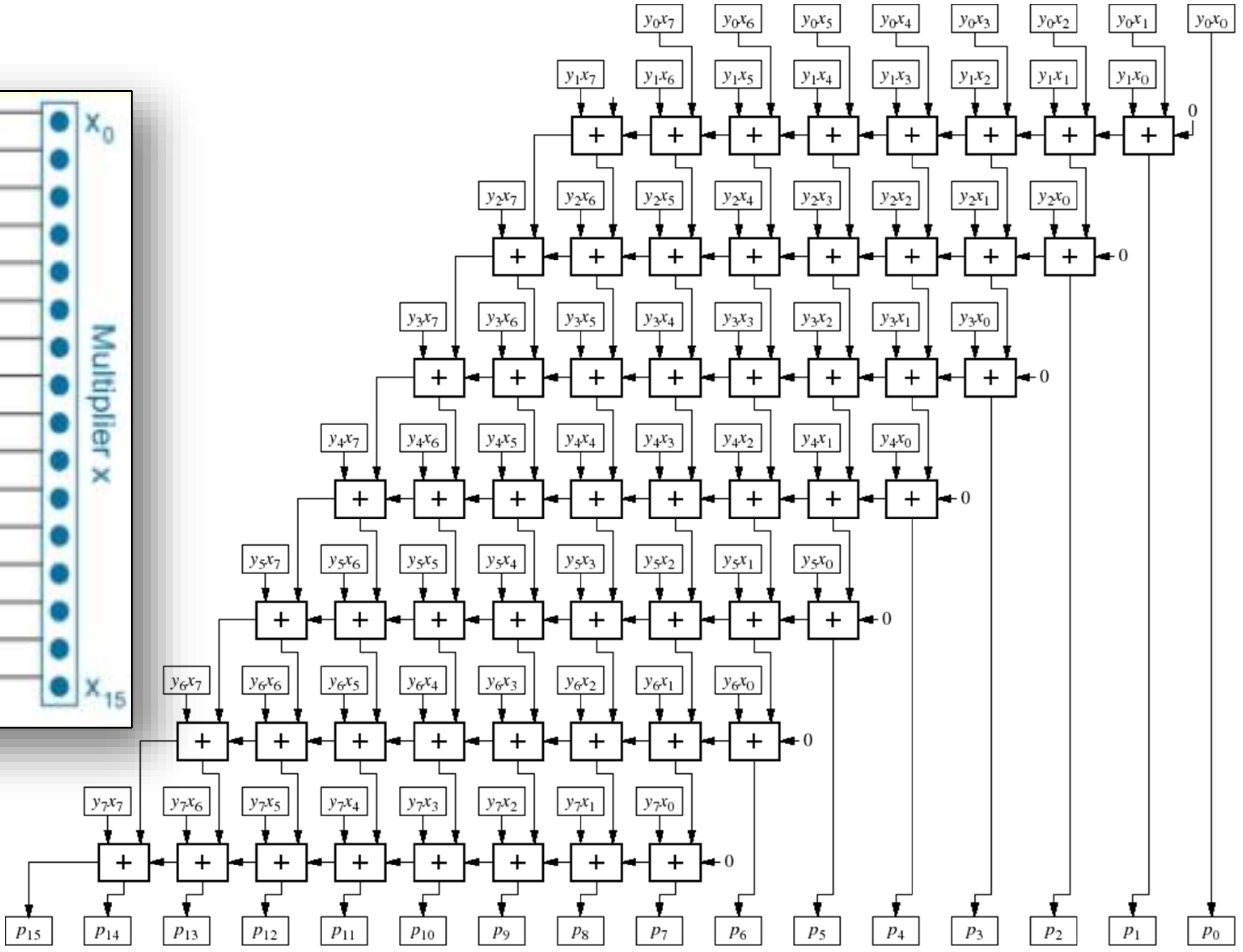
$$t_{serial} = O\left(N \cdot t_{adder}\right) = O\left(N^2\right)\Big|_{\text{for RCA}}$$

## Array Multiplier



## Many Critical Paths

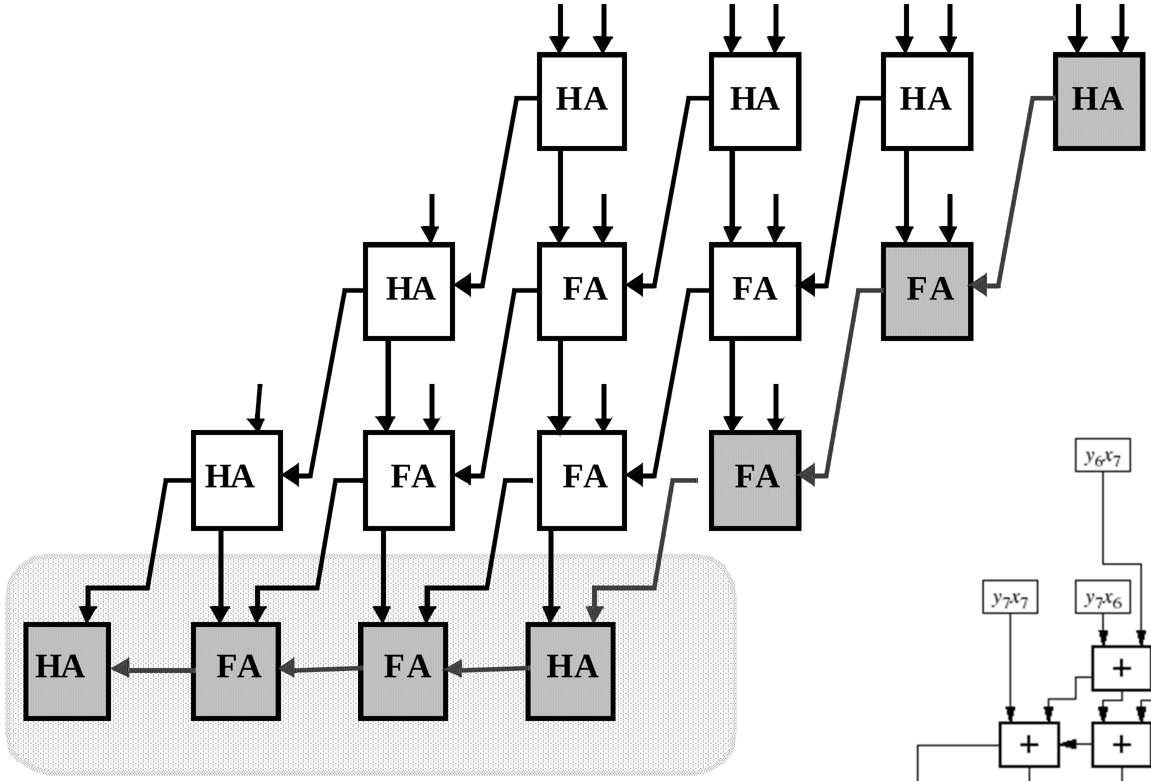




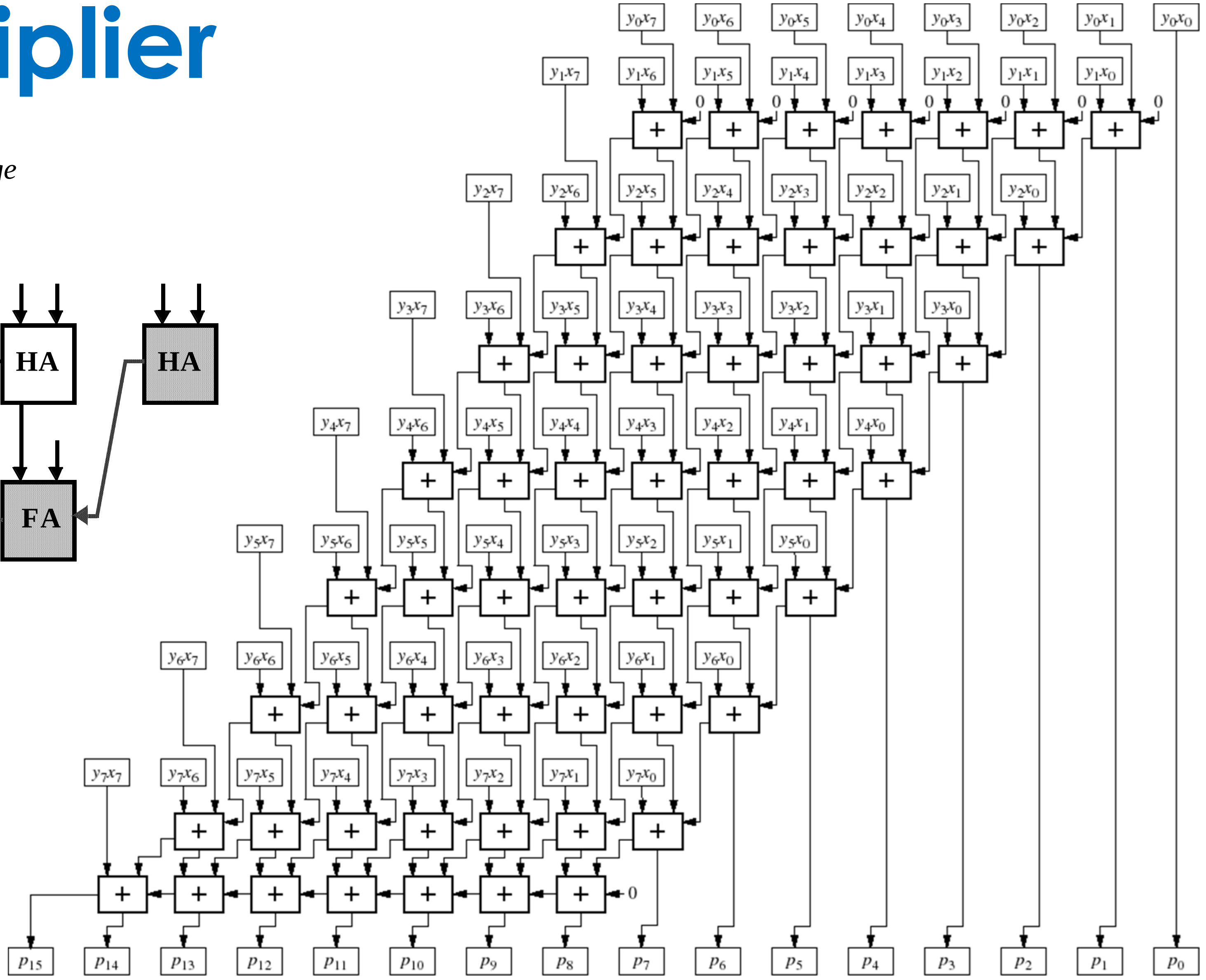
$$t_{mult} \approx t_{AND} + \left[(M-1)+(N-2)\right]t_{carry} + (N-1)t_{sum}$$

# Carry-Save Multiplier

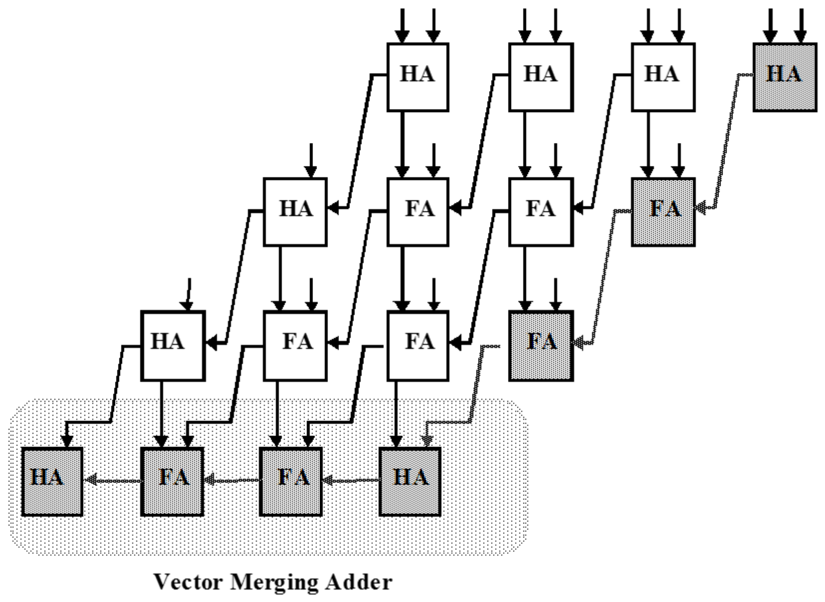
$$t_{mult} = t_{AND} + (N-1)t_{carry} + t_{merge}$$

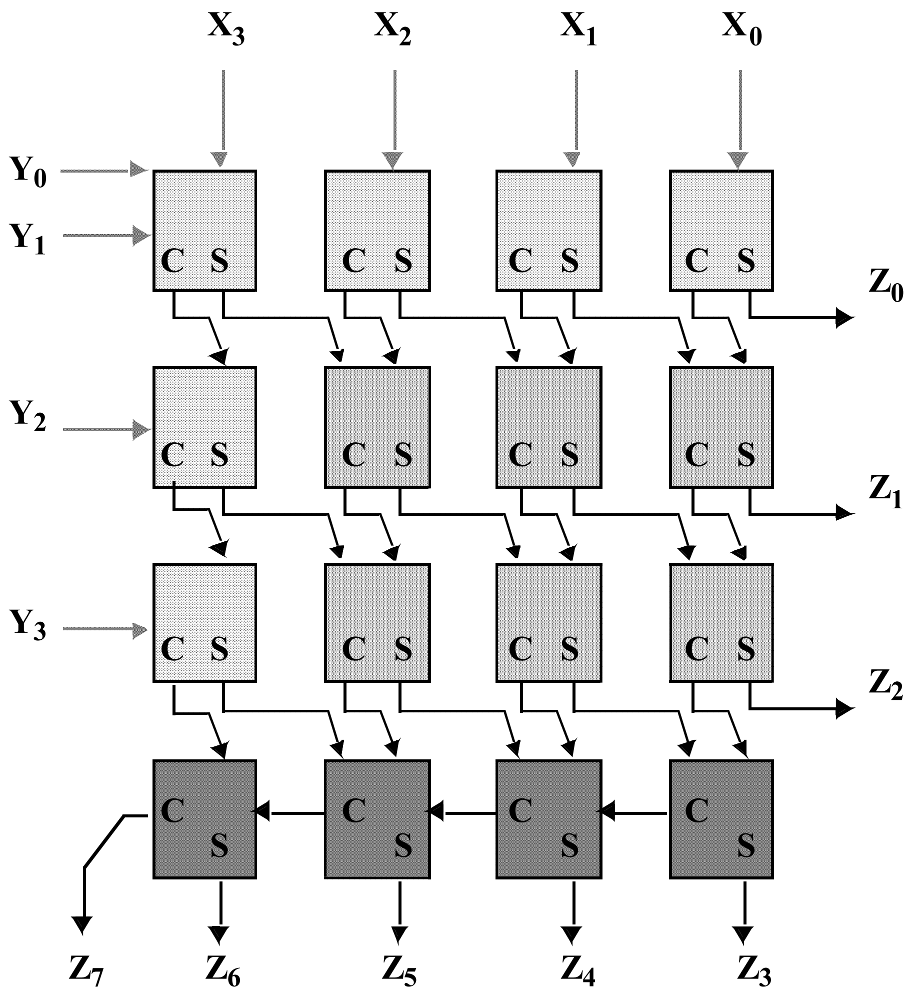


Vector Merging Adder



# Multiplier Floorplan





▨ HA Multiplier Cell

▨ FA Multiplier Cell

■ Vector Merging Cell

X and Y signals are broadcasted through the complete array. ( $\longrightarrow$ )

# Booth Recoding

- **Multiplying by ‘0’ is redundant.**
- **Can we reduce the number of partial products?**

$$\sum_{i=0}^{n-1} 2^i = 2^n - 1$$

- **Based on the observation that**
  - We can turn sequences of 1’s into sequences of 0’s. For example: 0111=1000-0001

$$\begin{array}{rr} 1000 & (8) \\ -0001 & (1) \\ \hline 0111 & (7) \end{array} \qquad \begin{array}{rr} 01000000 & (64) \\ -00001000 & (8) \\ \hline 00111000 & (56) \end{array}$$

- **So we can introduce a ‘-1’ bit and recode the multiplier:**
  - For example, the number 56

$$\begin{array}{cccccccc} 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\ \hline 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & +1 & 0 & 0 & -1 & 0 & 0 & 0 \\ \hline \end{array}$$

$$\begin{aligned} 2^5 + 2^4 + 2^3 &= 32 + 16 + 8 \\ &= 56 \\ 2^6 - 2^3 &= 64 - 8 \\ &= 56 \end{aligned}$$

# Radix-2 Booth Recoding

- **Parse multiplier from left to right**
  - For each change from 0 to 1, encode a ‘1’
  - For each change from 1 to 0, encode a ‘-1’
  - For bit 0, assume bit *i*=-1 is a 0
- **Example: 0011 0111 0011 = Ox373**

$$\begin{array}{cccc|cccc|cccc} 0 & 1 & 0 & -1 & 1 & 0 & 0 & -1 & 0 & 1 & 0 & -1 \end{array}$$

$$\begin{array}{cccc|cccc|cccc} 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ -0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{array}
\qquad
\begin{array}{r} \text{Ox } 484 \\ -\text{Ox } 111 \\ \hline \text{Ox } 373 \end{array}$$

## Modified (Radix-4) Booth Recoding

- **Radix-2 Booth Recoding doesn't work for parallel hardware implementations:**
  - A worst case (010101010101010) doesn't reduce the number of partial products.
  - Variable length recoders (according to the length of '1' strings) cannot be implemented efficiently.
- **Instead, just assume a constant length recoder.**
  - First apply standard booth recoding.
  - Next encode each pair of bits:

1. Within a sequence:

$$\begin{array}{cc} 0 & 0 \\ \hline & 0 \end{array}$$

2. Begin of a 1's-sequence:

$$\begin{array}{cc} 0 & +1 \\ \hline & +1 \end{array} \qquad \begin{array}{cc} +1 & -1 \\ \hline & +1 \end{array} \qquad \begin{array}{cc} +1 & 0 \\ \hline & +2 \end{array}$$

3. End of a 1's-sequence:

$$\begin{array}{cc} 0 & -1 \\ \hline & -1 \end{array} \qquad \begin{array}{cc} -1 & +1 \\ \hline & -1 \end{array} \qquad \begin{array}{cc} -1 & 0 \\ \hline & -2 \end{array}$$

- **This can be summarized in a truth table:**

| Partial Product Selection Table | |
|---|---|
| **Multiplier Bits** | **Recorded Bits** |
| 000 | 0 |
| 001 | + Multiplicand |
| 010 | + Multiplicand |
| 011 | +2 × Multiplicand |
| 100 | -2 × multiplicand |
| 101 | - Multiplicand |
| 110 | - Multiplicand |
| 111 | 0 |

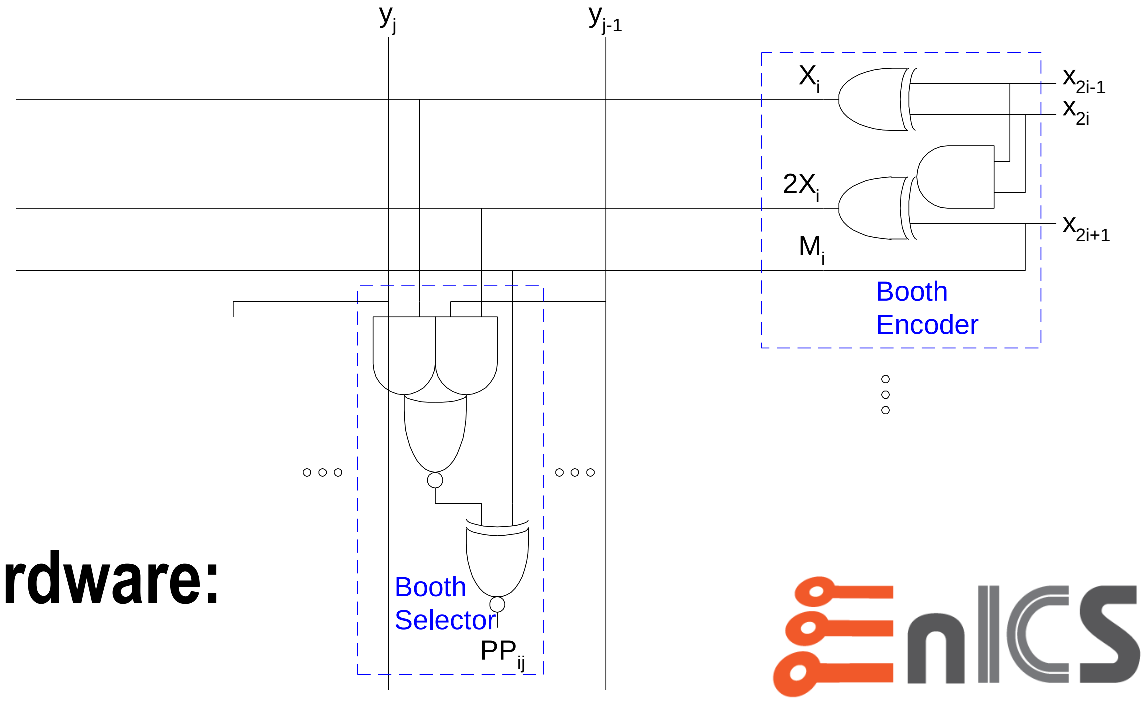
# Modified (Radix-4) Booth Recoding

- **For example, let's take our previous example:**
  - 0011 0111 0011 = 01 0-1 10 0-1 01 0-1
  - This comes out: 1 -1 2 -1 1 -1.
- **We could have done this by using the table:**

| Partial Product Selection Table | |
|---|---|
| **Multiplier Bits** | **Recorded Bits** |
| 000 | 0 |
| 001 | + Multiplicand |
| 010 | + Multiplicand |
| 011 | +2 × Multiplicand |
| 100 | -2 × multiplicand |
| 101 | - Multiplicand |
| 110 | - Multiplicand |
| 111 | 0 |

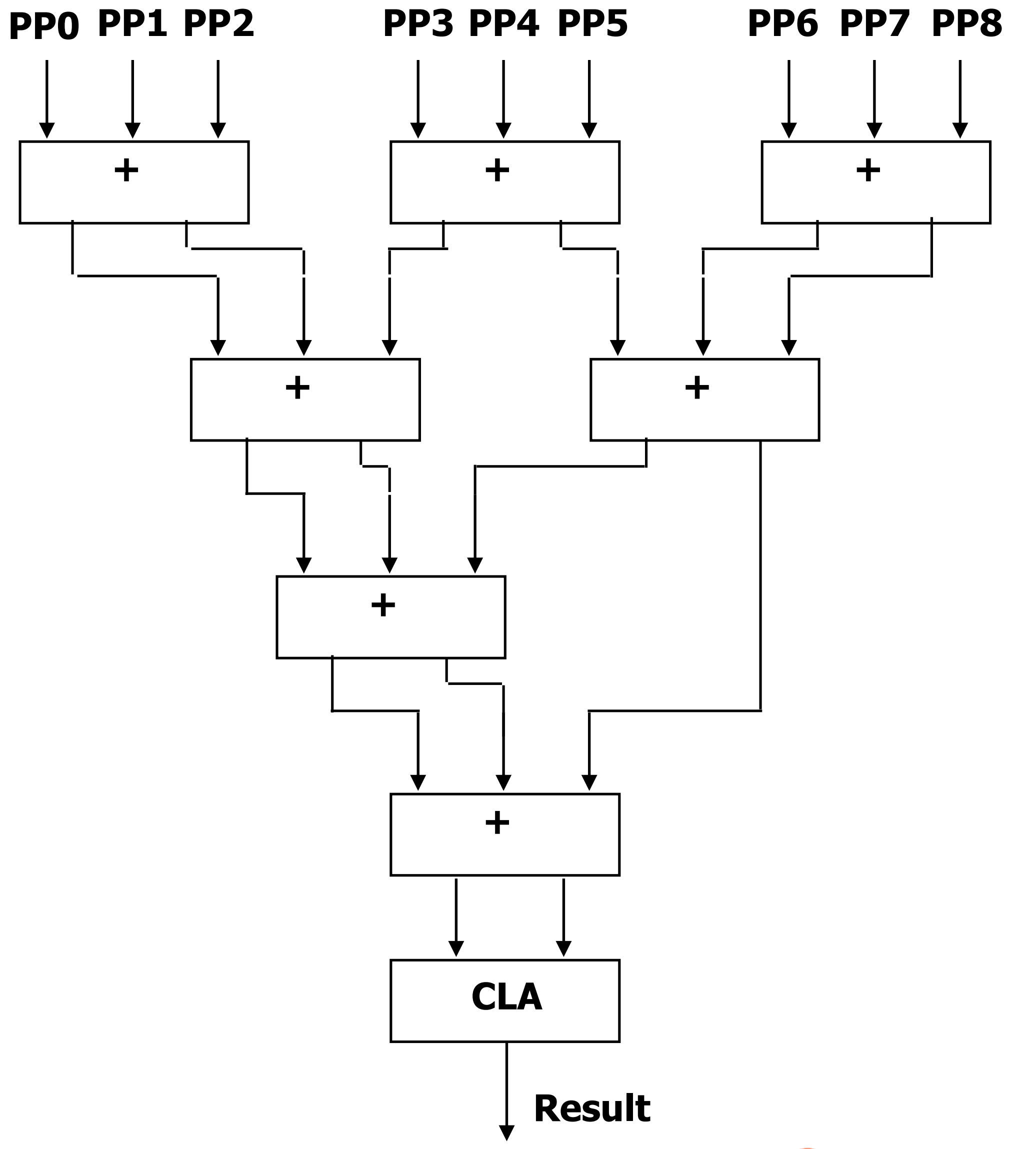
0 0 1 1 0 1 1 1 0 0 1 1

- **To implement this we need pretty simple hardware:**

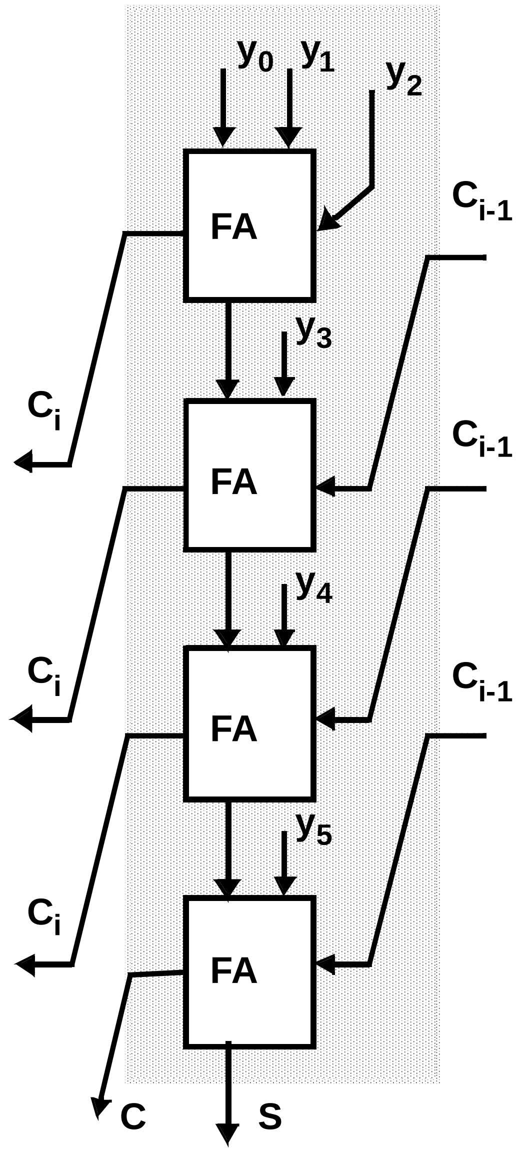


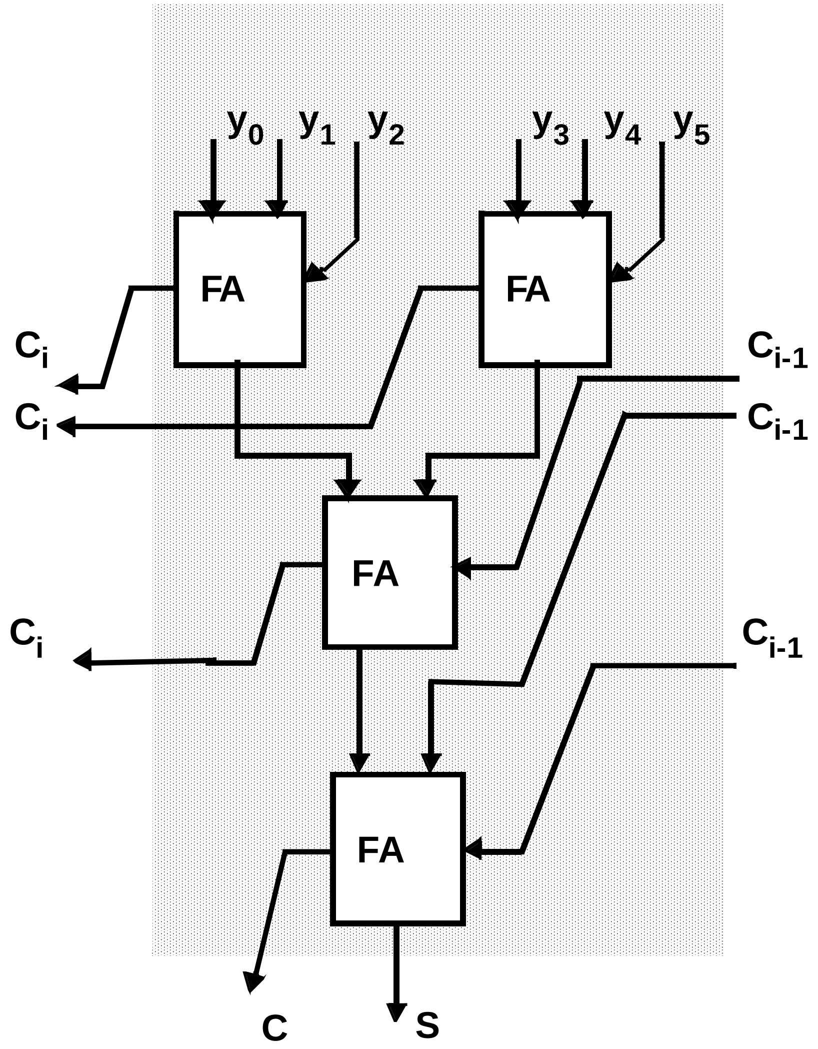
## Tree Multipliers

- Can we further reduce the multiplier delay by employing logarithmic (tree) structures?

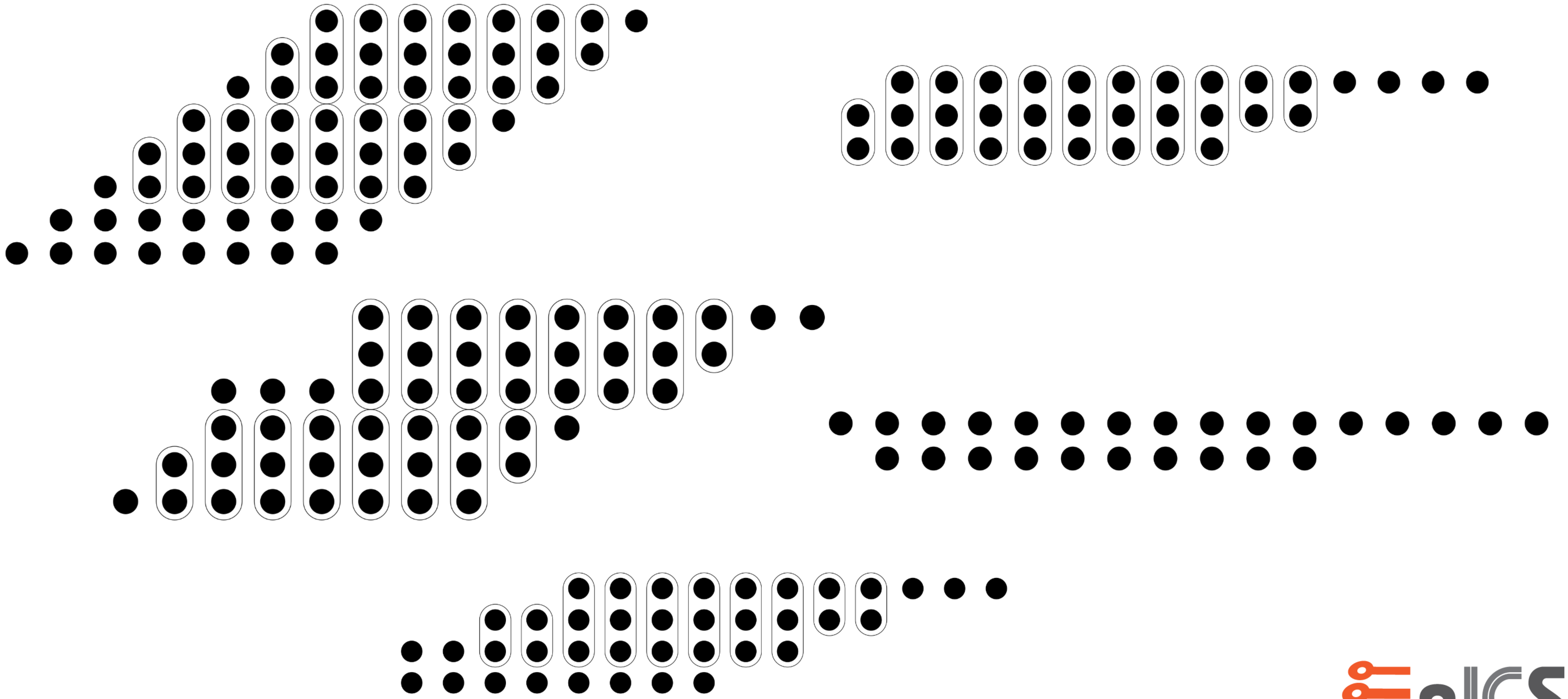


# Wallace-Tree Multiplier

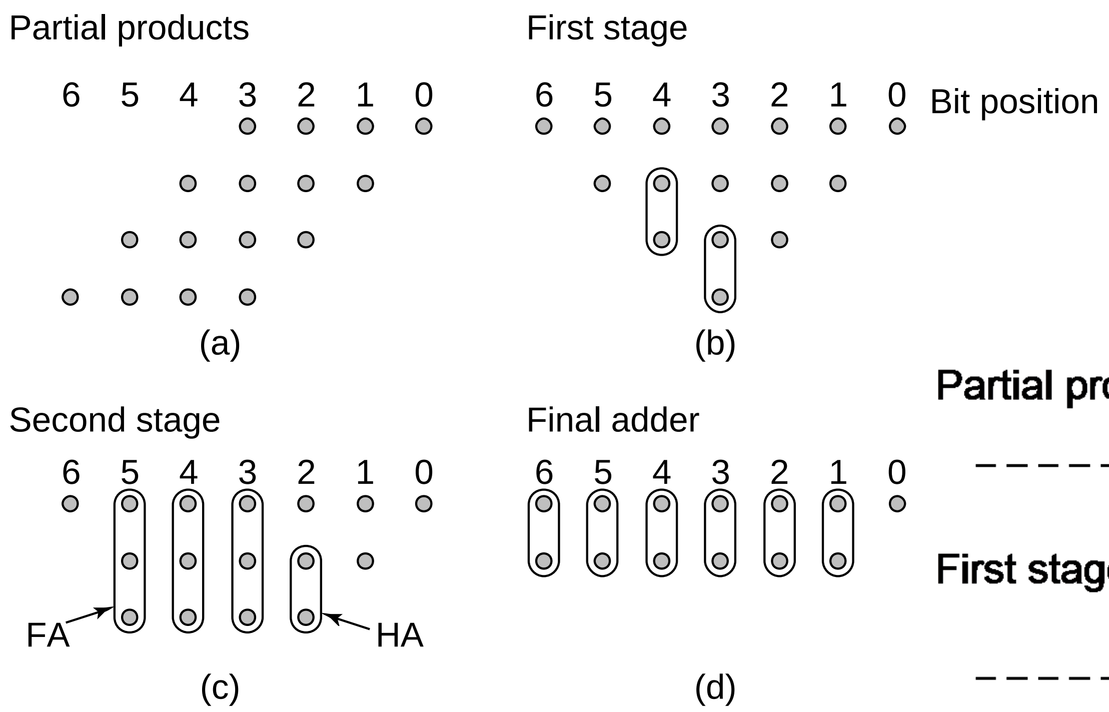


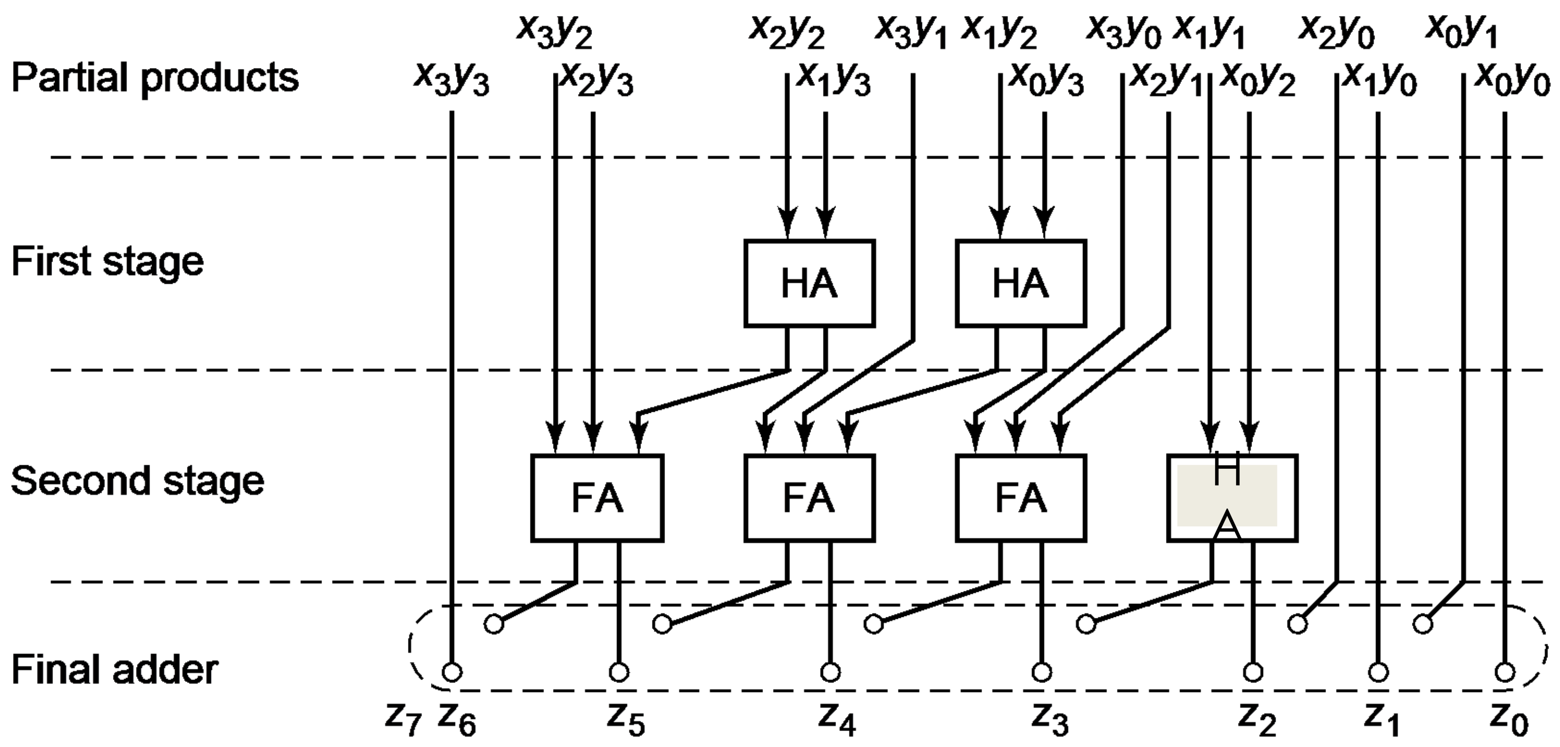


# Wallace-Tree Multiplier



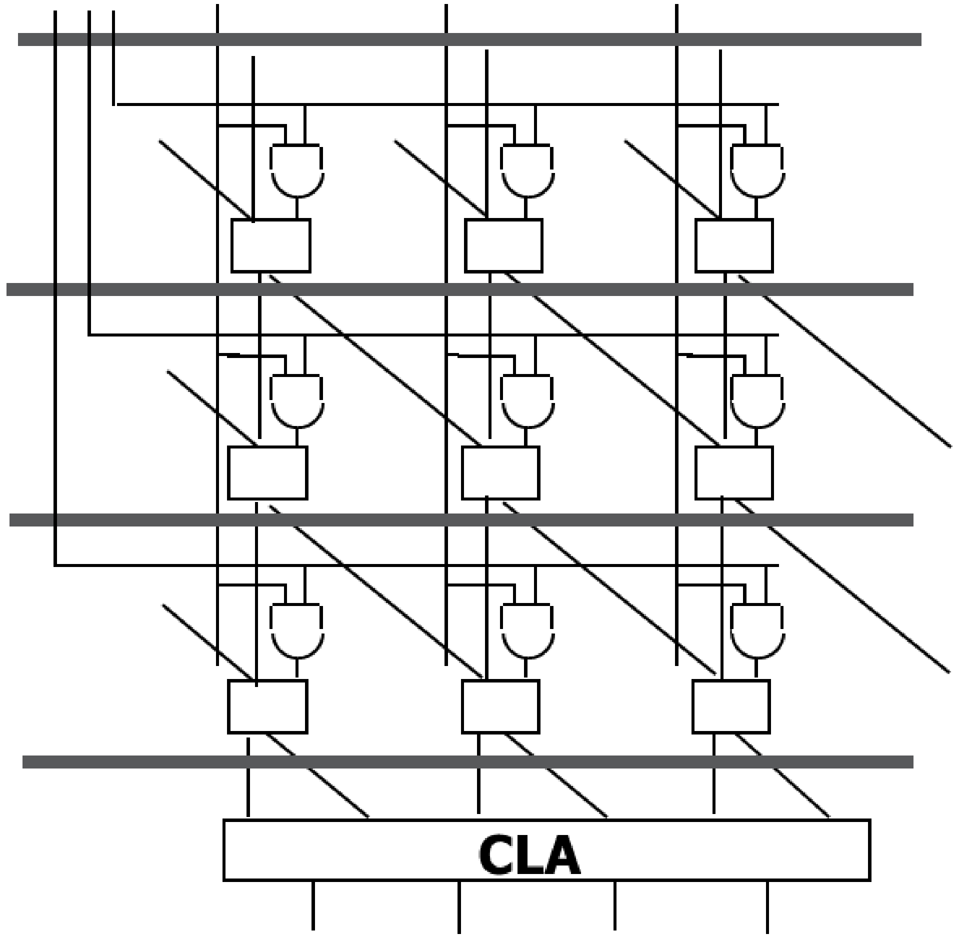
# Wallace-Tree Multiplier

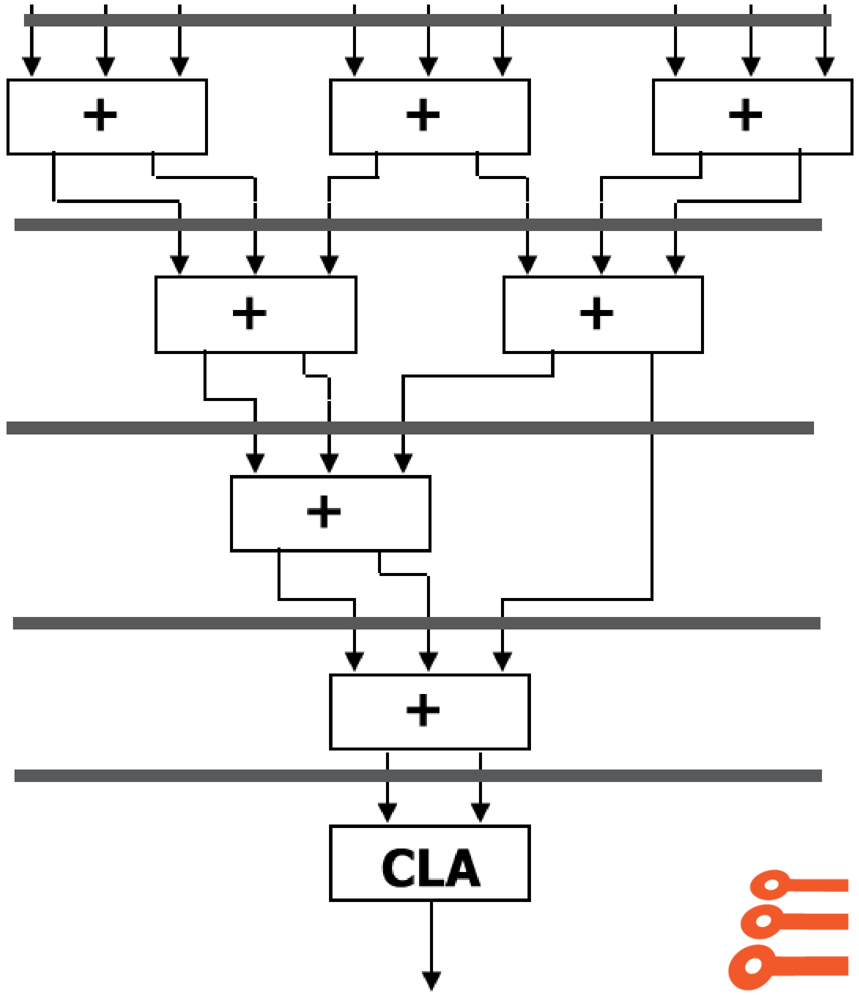




# Pipelining Multipliers

- Pipelining can be applied to most multiplier structures:





## Further Reading

- Rabaey, et al. “Digital Integrated Circuits” (2nd Edition)
- Elad Alon, Berkeley ee141 (online)
- Weste, Harris, “CMOS VLSI Design (4th Edition)”