集積回路設計

14. トピック : 算術演算回路とセルライブラリ合成

一色 剛

工学院情報通信系

isshiki@ict.e.titech.ac.jp

集積回路設計期末試験

■ 日時・場所

- 6月7日（金） 13:20～14:50、S321講義室
- 出題範囲：第1回～第13回（本日講義分は除く）
- ノート・講義資料・その他：持ち込み不可

VLSIの設計方式 (1)

■ フルカスタム方式

- 論理設計、回路設計、レイアウト設計を全て手動設計
- 回路面積・動作速度・消費電力を最適化できる
- 開発コスト（設計コスト）は高いが、量産効果による製品単価の大幅削減が可能（汎用CPU、メモリ）
- アナログ回路は通常フルカスタム方式で設計される

■ スタンダードセル方式

- 論理設計、回路設計、レイアウト設計を全て自動化
- 論理セル : 数十～数百種のレイアウト済み設計物
- 集積度・性能はフルカスタムに劣るが、開発コストを大幅削減可能

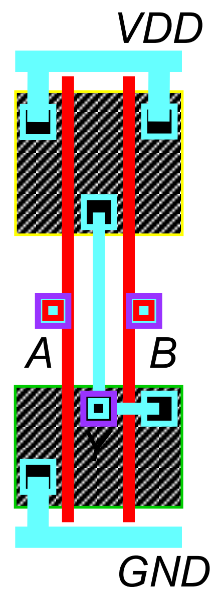
Technology Library (1)

A) Standard-cell library

- Cell types :
  - Primitive/complex gates : NAND, NOR, INV, AOI, MUX, tristate-buffer, half-adder, full-adder, carry-generator, ...
  - Storage elements : DFF, LATCH
- Models :
  - Functional (technology-independent)
  - Delay, layout (technology-dependent)

Load :
  A : 0.025pF
  B : 0.024pF
Internal delay :
  A=>Y(rise) : 0.055 ns
  A=>Y(fall) : 0.051 ns
  B=>Y(rise) : 0.073 ns
  B=>Y(fall) : 0.060 ns
Output transition delay :
  Y(rise) : 1.532 ns/pF
  Y(fall) : 1.153 ns/pF

delay



layout

Technology Library (2)

B) Memory module library/generator

- Input parameters
  - capacity, word size
  - access timing (synchronous / asynchronous)
  - # of ports (ex. 1 R/W, 1R + 1W, 2R + 1W)
- Layout data : predesigned or auto-generated

C) Arithmetic module library/generator

- Input parameters
  - Functionality (add, subtract, multiply, divide, ...)
  - Word size
  - Speed
- "Hard" module : layout data
- "Soft" module : synthesizable RTL or gate-level netlist

Technology Library (3)

D) Complex function modules

(IP : intellectual property)

- Complex functions take long time to design → design reuse
- Types of IPs : CPU-core, interface, customized datapath
- "Hard" IP and "Soft" IP
- IP designers and IP users are different people (different organization in most cases)
  - Need very good documentation of IP. (Difficulty of understanding how the IP works and how to use the IP is the major problem in IP-based designs)

Hard Module vs. Soft Module

- Hard module (layout data)
  - Optimize at transistor-level and layout
  - Smaller, faster, power efficient
  - Technology-dependent
    - → need to redesign for each different technology
    - → expensive
- Soft module (gate-level / RTL netlist)
  - Optimize at architecture level
  - Technology-independent
    - → can take advantage of the latest process technologies
    - → easier to incorporate into user's designs

➢ *Soft module is becoming more popular than hard modules*

➢ *Need good cell library !*

Cell Synthesis

- Purpose :
  - Standard-cell library compilation
  - Transister-level custom module synthesis
  - Because of the fast change in process technology, libraries have to be designed quickly → more tasks are being automated
- Input : logic function, layout design rule (technology)
- Output : cell layout
- Process flow :

  Logic function
  ↓
  CMOS circuit topology
  ↓
  CMOS placement & routing (abstract / stick diagram)
  ↓
  CMOS layout

CMOS Circuit Topology Enumeration (1)

Ex. 2-input XOR : $Y = A \oplus B$

A) Decompose to negative-unate functions

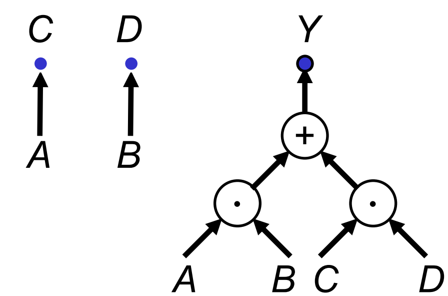
i. $Y = A \cdot (\sim B) + (\sim A) \cdot B$

$= \sim((\sim A) \cdot (\sim B) + A \cdot B)$

$C = \sim A$

$D = \sim B$

$Y = \sim(C \cdot D + A \cdot B)$

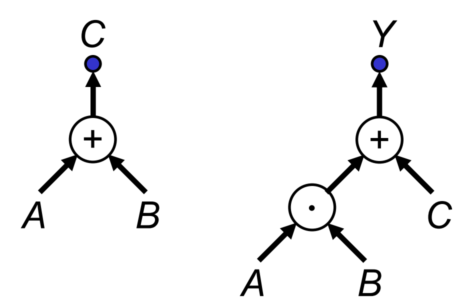


ii. $Y = \sim((\sim A) \cdot (\sim B) + A \cdot B)$

$= \sim(\sim(A + B) + A \cdot B)$

$C = \sim(A + B)$

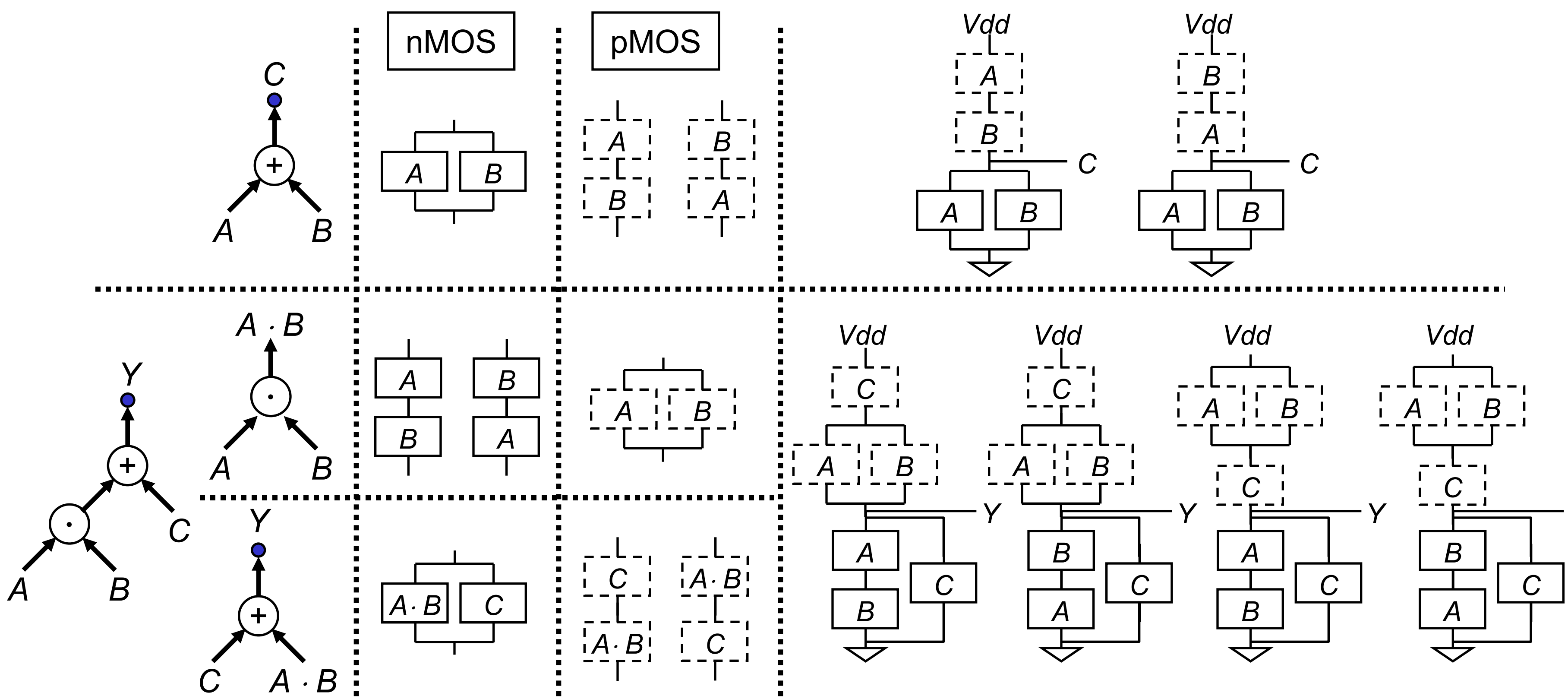
$Y = \sim(C + A \cdot B)$



CMOS Circuit Topology Enumeration (2)

B) Construct circuit topology

- OR : nMOS = parallel, pMOS = serial
- AND : nMOS = serial, pMOS = parallel

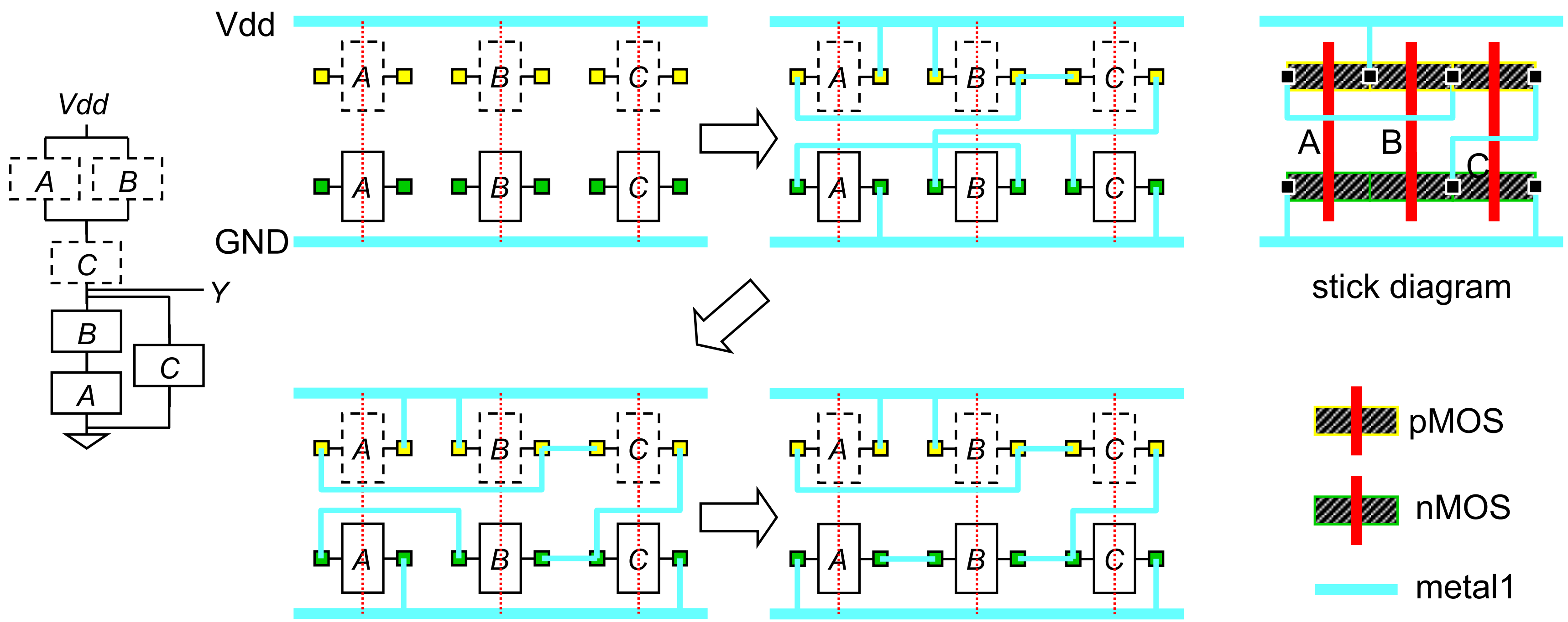


CMOS Placement and Routing

A) Transistor placement (gate ordering)
- place the pMOS and nMOS with the same gate input at the same vertical line
- Vdd : top-horizontal, GND : horizontal-bottom

B) Permutate transistor pin (drain/source) direction

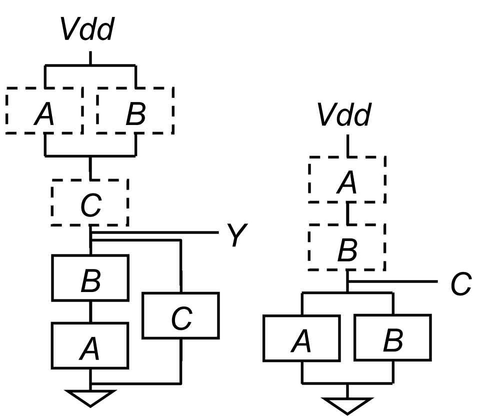
C) Cost function : area (drain/source sharing), # of routing tracks , # of routing layers (*routing cost is not easy to evaluate before actual layout*)

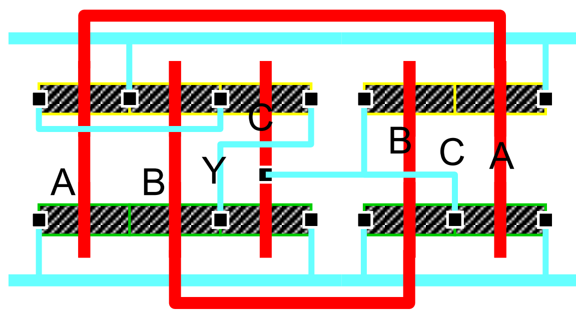


CMOS Layout

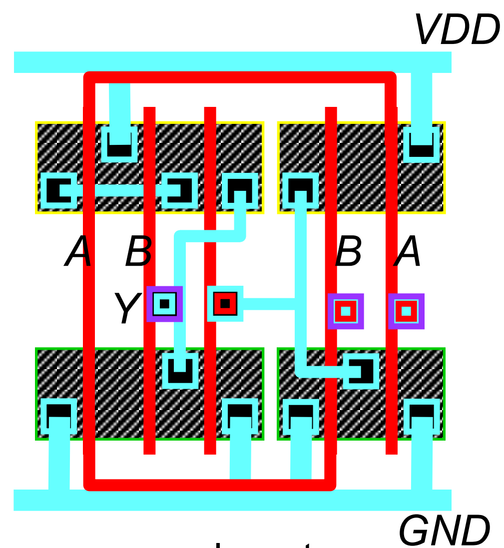
A) Generate layout data from stick diagram

- Technology-dependent design rule :
  - Minimum width
  - Minimum spacing





stick diagram



layout

Arithmetic Logic Synthesis

- Circuit implementation techniques for arithmetic logic modules (adder/subtractor, multiplier, divider, square root, trigonometric functions, etc.) have been extensively studied.
  - Transistor-level structures
  - Gate-level structures
  - RTL structures
- Logic optimization algorithms do not work well with arithmetic logic

→ Logic synthesis tools generate predesigned logic structures for a limited types of arithmetic logic (adder, subtractor, multiplier)

→ Some standard cell libraries include cells dedicated for arithmetic logic (such as carry computation)

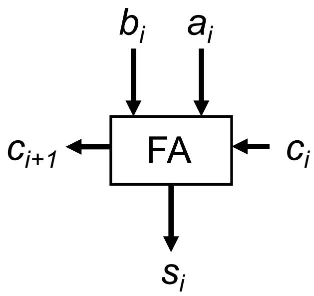
Adders

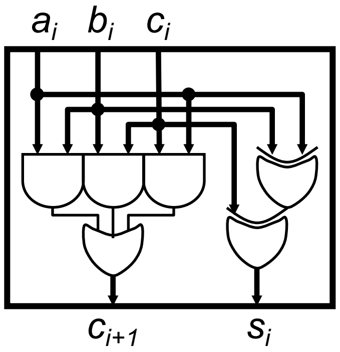
- Addition is the most basic arithmetic operation
- Variety of adder structures
  - ✓ Ripple-carry adder
  - ✓ Carry-lookahead adder
  - ✓ Carry-skip adder
  - ✓ Carry-select adder
  - ✓ Carry-save adder (for intermediate accumulation)
- Tradeoff in circuit area, speed (power)

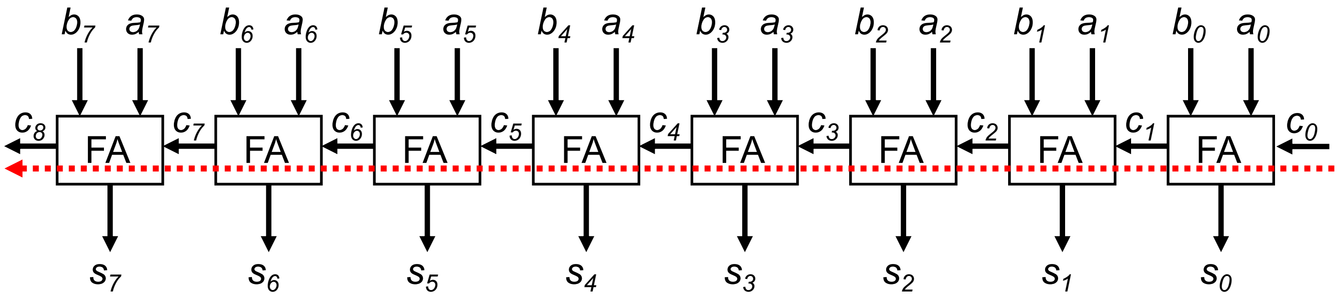
Ripple-Carry Adder

– $c_{i+1} = a_i \cdot b_i + a_i \cdot c_i + b_i \cdot c_i$

– $s_i = a_i \oplus b_i \oplus c_i$







- Small area
- Slow because of carry propagation

Full Adder with Carry-Generate and Carry-Propagate (1)

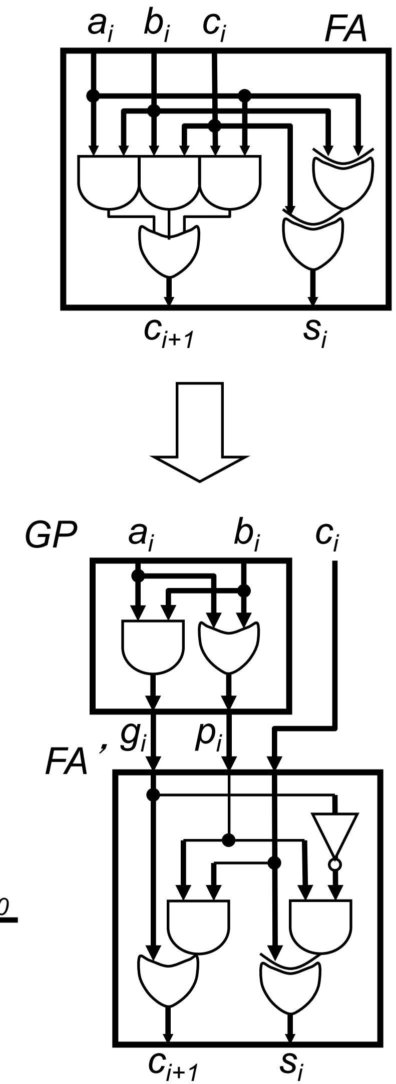
- *Generate* : $g_i = a_i \cdot b_i$
- *Propagate* : $p_i = a_i + b_i$

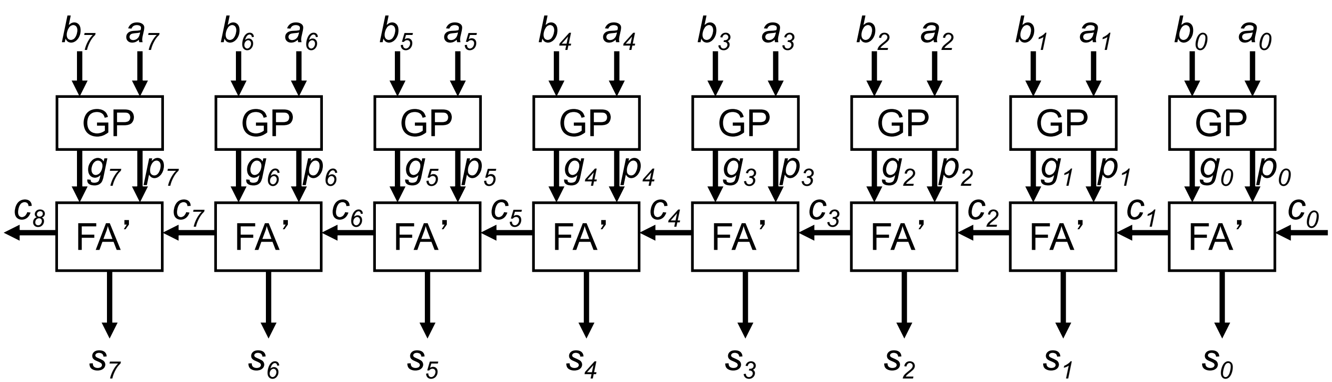
– $c_{i+1} = a_i \cdot b_i + a_i \cdot c_i + b_i \cdot c_i$

$\quad = a_i \cdot b_i + (a_i + b_i) \cdot c_i = g_i + p_i \cdot c_i$

– $s_i = a_i \oplus b_i \oplus c_i$

$\quad = (\bar{a}_i \cdot b_i + a_i \cdot \bar{b}_i) \oplus c_i = (p_i \cdot \bar{g}_i) \oplus c_i$





Full Adder with Carry-Generate and Carry-Propagate (2)

- *Generate :* $g_i = a_i \cdot b_i$
- *Propagate :* $p_i = a_i \oplus b_i$

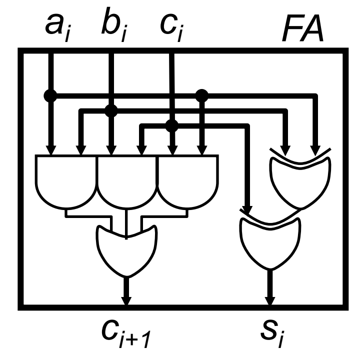
– $c_{i+1} = a_i \cdot b_i + a_i \cdot c_i + b_i \cdot c_i$

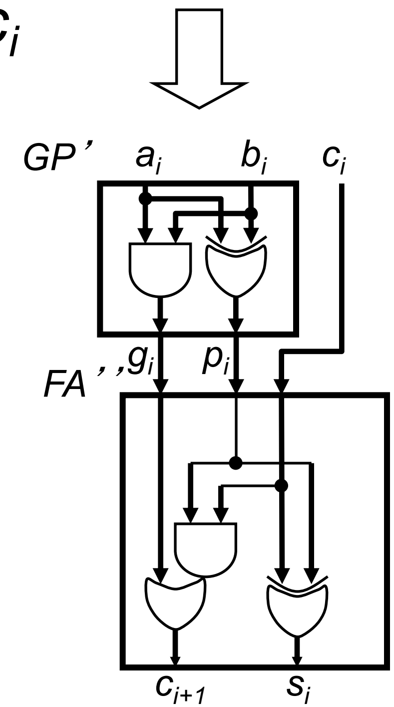
$$= a_i \cdot b_i + a_i \cdot (b_i + \overline{b}_i) \cdot c_i + b_i \cdot (a_i + \overline{a}_i) \cdot c_i$$

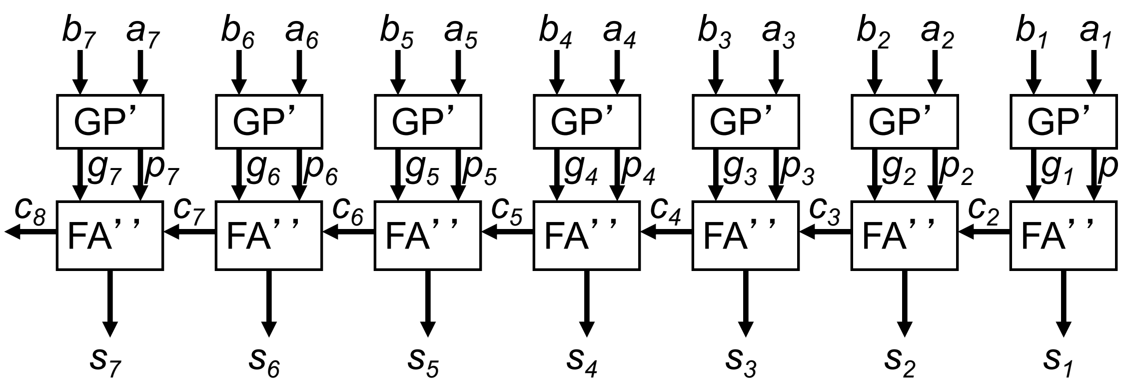
$$= a_i \cdot b_i + a_i \cdot b_i \cdot c_i + (a_i \cdot \overline{b}_i + \overline{a}_i \cdot b_i) \cdot c_i$$

$$= a_i \cdot b_i + (a_i \oplus b_i) \cdot c_i = g_i + p_i \cdot c_i$$

– $s_i = a_i \oplus b_i \oplus c_i = p_i \oplus c_i$

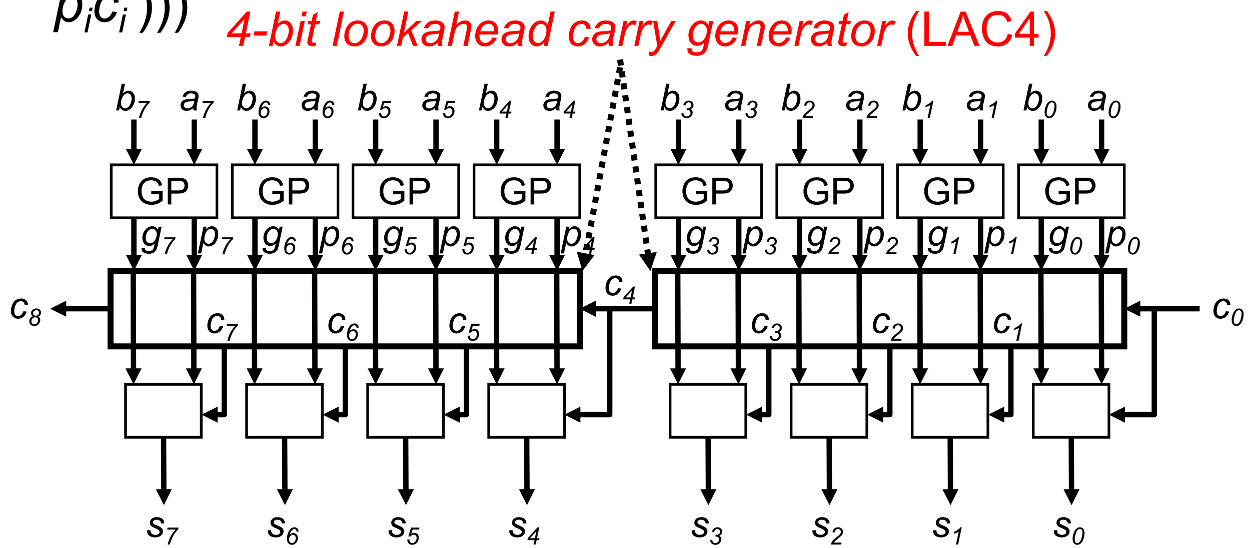






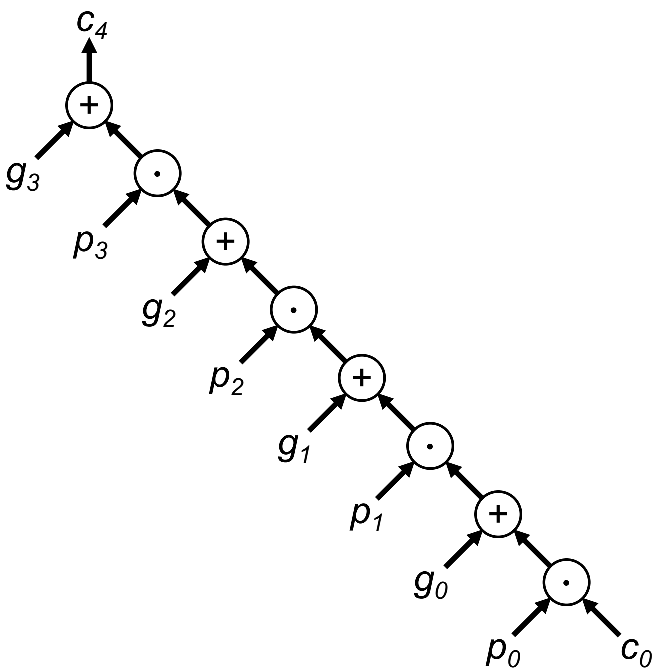
Carry-Lookahead Adder

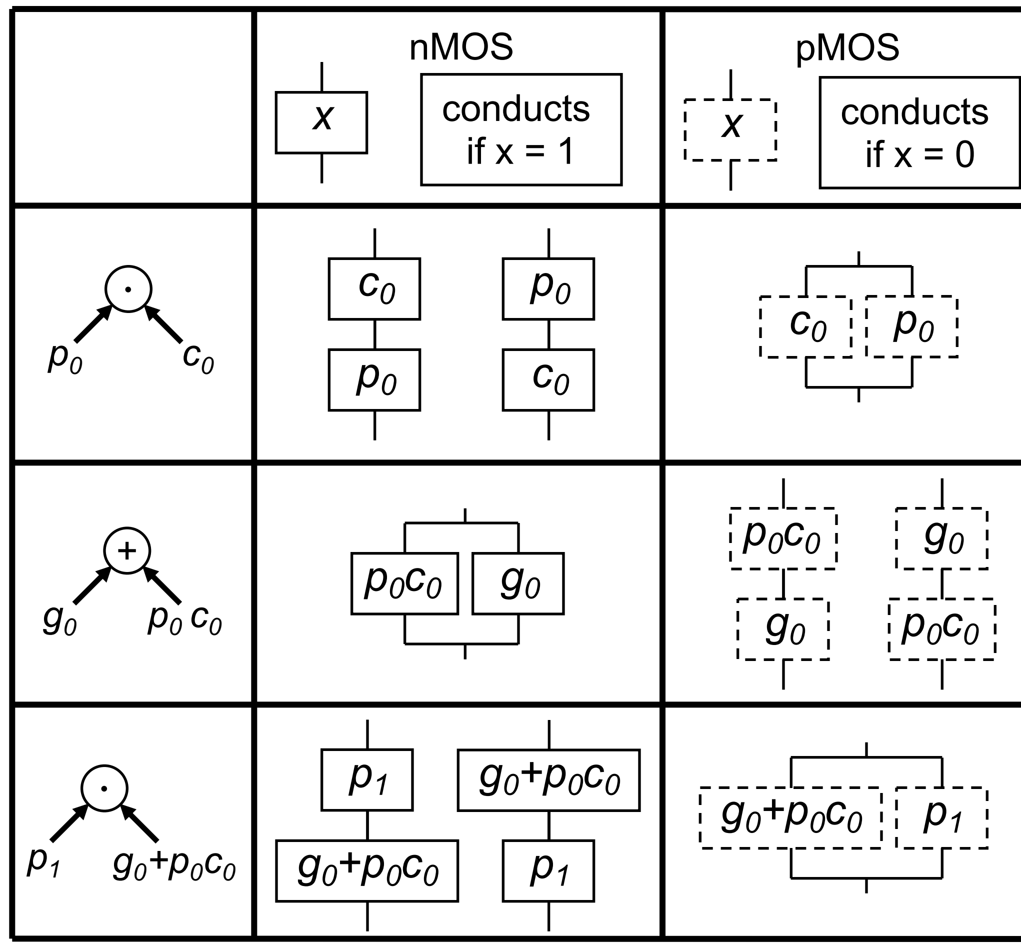
- Unfolding carry propagation
  → 4-bit lookahead carry generator (LAC4)
  - $c_{i+1} = g_i + p_i\, c_i$
  - $c_{i+2} = g_{i+1} + p_{i+1}\, c_{i+1} = g_{i+1} + p_{i+1}(g_i + p_i c_i)$
  - $c_{i+3} = g_{i+2} + p_{i+2}\, c_{i+2} = g_{i+2} + p_{i+2}(g_{i+1} + p_{i+1}(g_i + p_i c_i))$
  - $c_{i+4} = g_{i+3} + p_{i+3}\, c_{i+3} = g_{i+3} + p_{i+3}(g_{i+2} + p_{i+2}(g_{i+1} + p_{i+1}(g_i + p_i c_i)))$



CMOS Implementation of Lookahead Carry Generator (1)

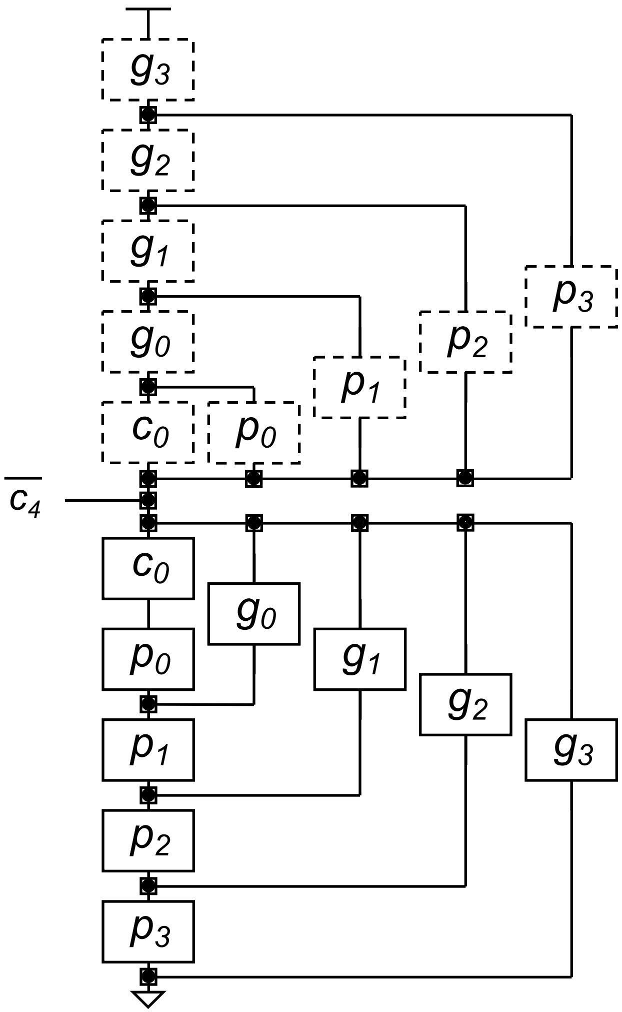
- $c_4 = g_3 + p_3(g_2 + p_2(g_1 + p_1(g_0 + p_0c_0)))$

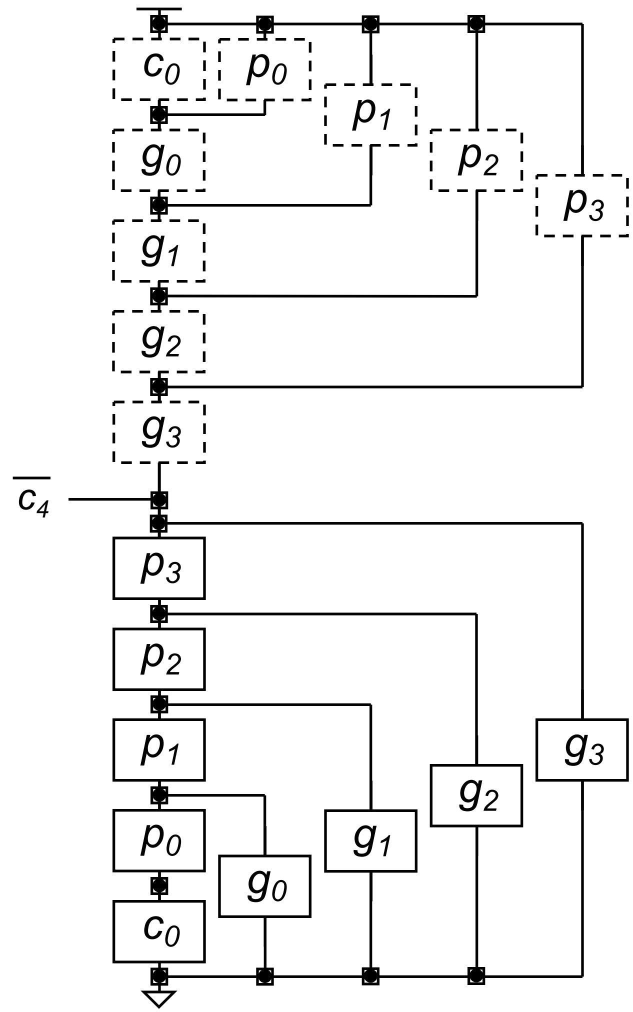




CMOS Implementation of Lookahead Carry Generator (2)

- $c_4 = g_3 + p_3(g_2 + p_2(g_1 + p_1(g_0 + p_0c_0)))$

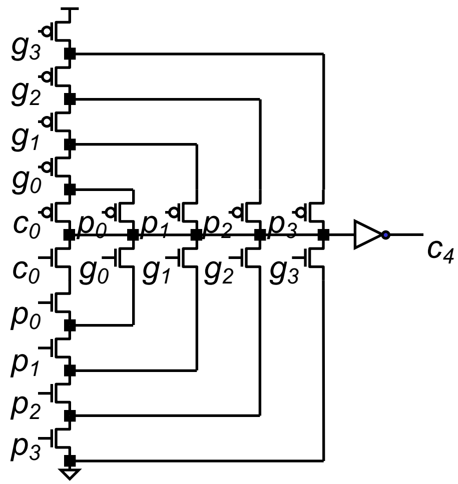


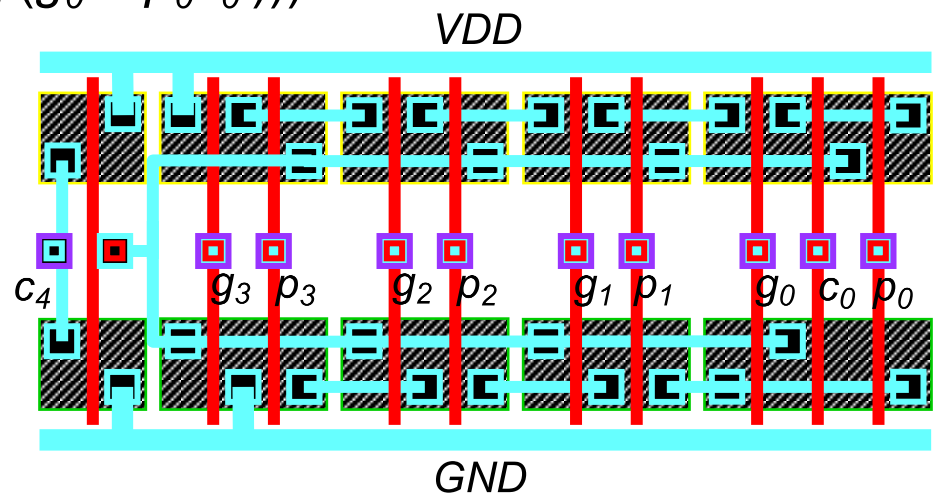


There are a total of 2^{10} combinations of network topologies

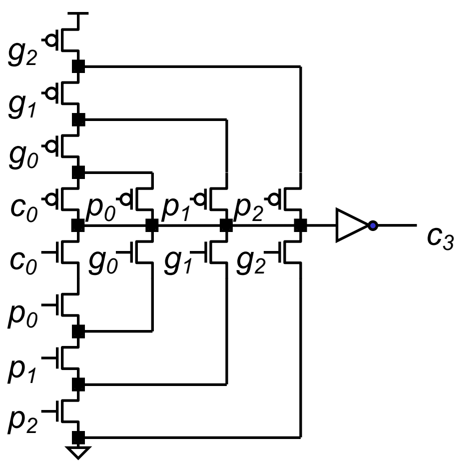
CMOS Implementation of Lookahead Carry Generator (3)

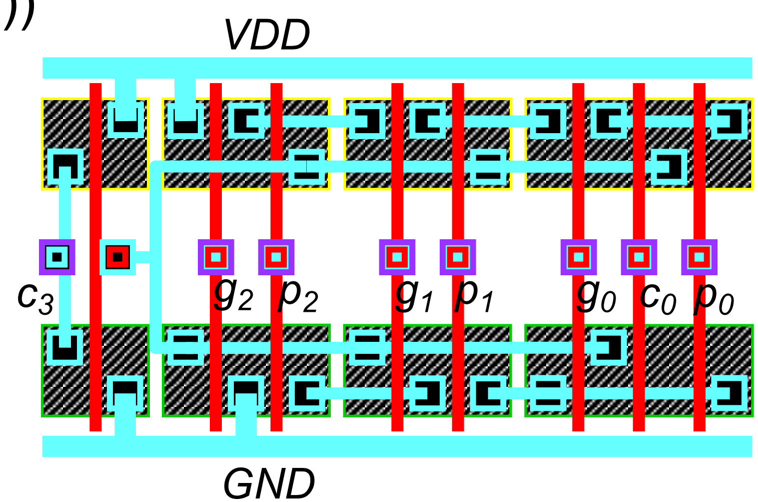
- $c_4 = g_3 + p_3(g_2 + p_2(g_1 + p_1(g_0 + p_0c_0)))$





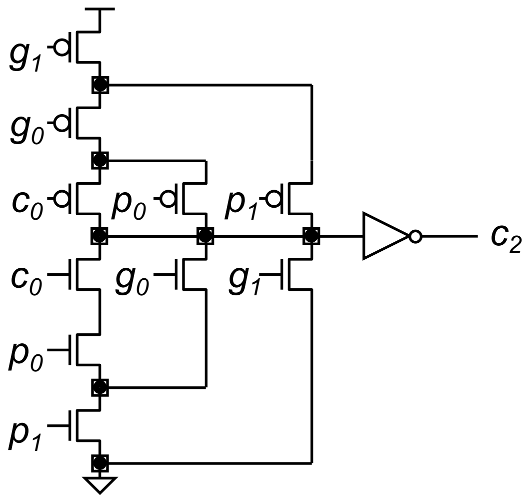
- $c_3 = g_2 + p_2(g_1 + p_1(g_0 + p_0c_0))$

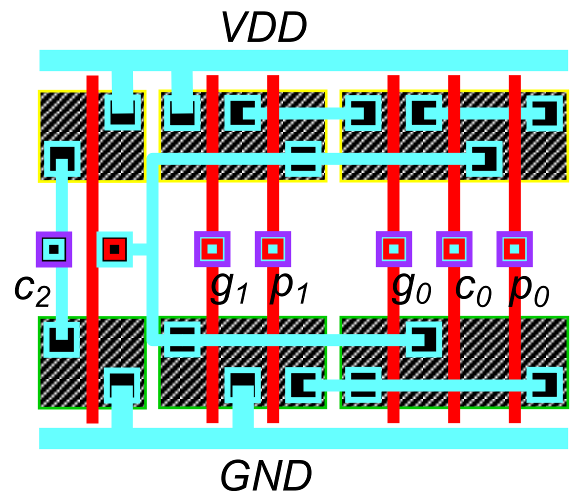




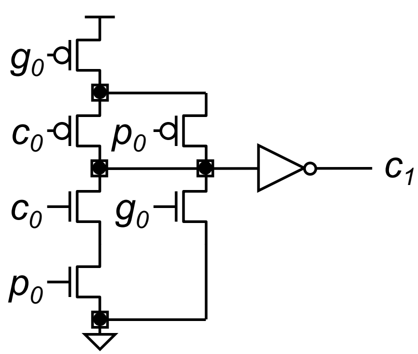
CMOS Implementation of Lookahead Carry Generator (4)

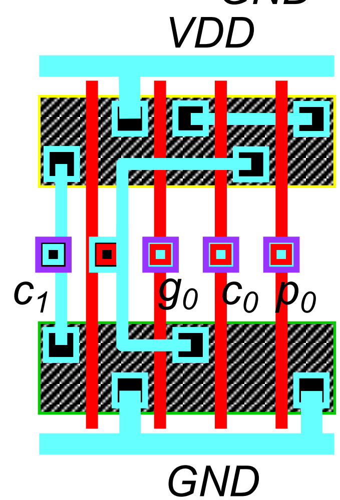
- $c_1 = g_0 + p_0 c_0$





- $c_2 = g_1 + p_1 (g_0 + p_0 c_0)$





Hierarchical Carry-Lookahead (1)

- 4-bit generate-propagate :

✓ $c_{i+4} = g_{i+3} + p_{i+3}(g_{i+2} + p_{i+2}(g_{i+1} + p_{i+1}(g_i + p_i c_i)))$

$= \underbrace{g_{i+3} + p_{i+3}(g_{i+2} + p_{i+2}(g_{i+1} + p_{i+1}g_i))}_{g'_i} + (\underbrace{p_{i+3}p_{i+2}p_{i+1}p_i}_{p'_i})c_i$

$= g'_i + p'_i c_i$

✓ $c_{i+8} = g'_{i+4} + p'_{i+4}c_{i+4} = g'_{i+4} + p'_{i+4}(g'_i + p'_i c_i)$

✓ $c_{i+12} = g'_{i+8} + p'_{i+8}c_{i+8} = g'_{i+8} + p'_{i+8}(g'_{i+4} + p'_{i+4}(g'_i + p'_i c_i))$

✓ $c_{i+16} = g'_{i+12} + p'_{i+12}c_{i+12} =$

$= g'_{i+12} + p'_{i+12}(g'_{i+8} + p'_{i+8}(g'_{i+4} + p'_{i+4}(g'_i + p'_i c_i)))$

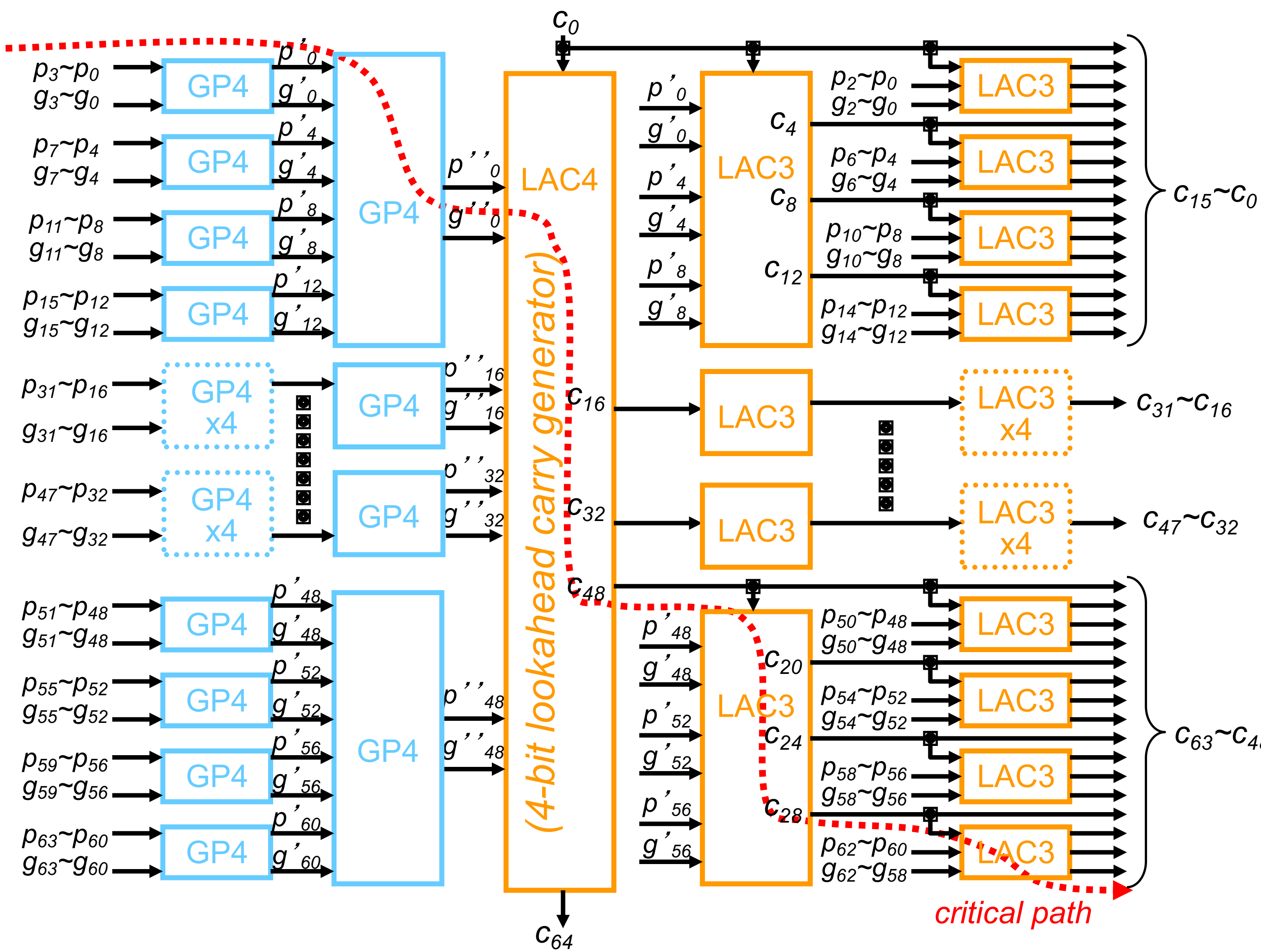
- 16-bit (4^2-bit) generate-propagate :

✓ $c_{i+16} = \underbrace{g'_{i+12} + p'_{i+12}(g'_{i+8} + p'_{i+8}(g'_{i+4} + p'_{i+4}g'_i))}_{g''_i}$

$+ (\underbrace{p'_{i+12}p'_{i+8}p'_{i+4}p'_i}_{p''_i})c_i$

$= g''_i + p''_i c_i$

Hierarchical Carry-Lookahead (2)



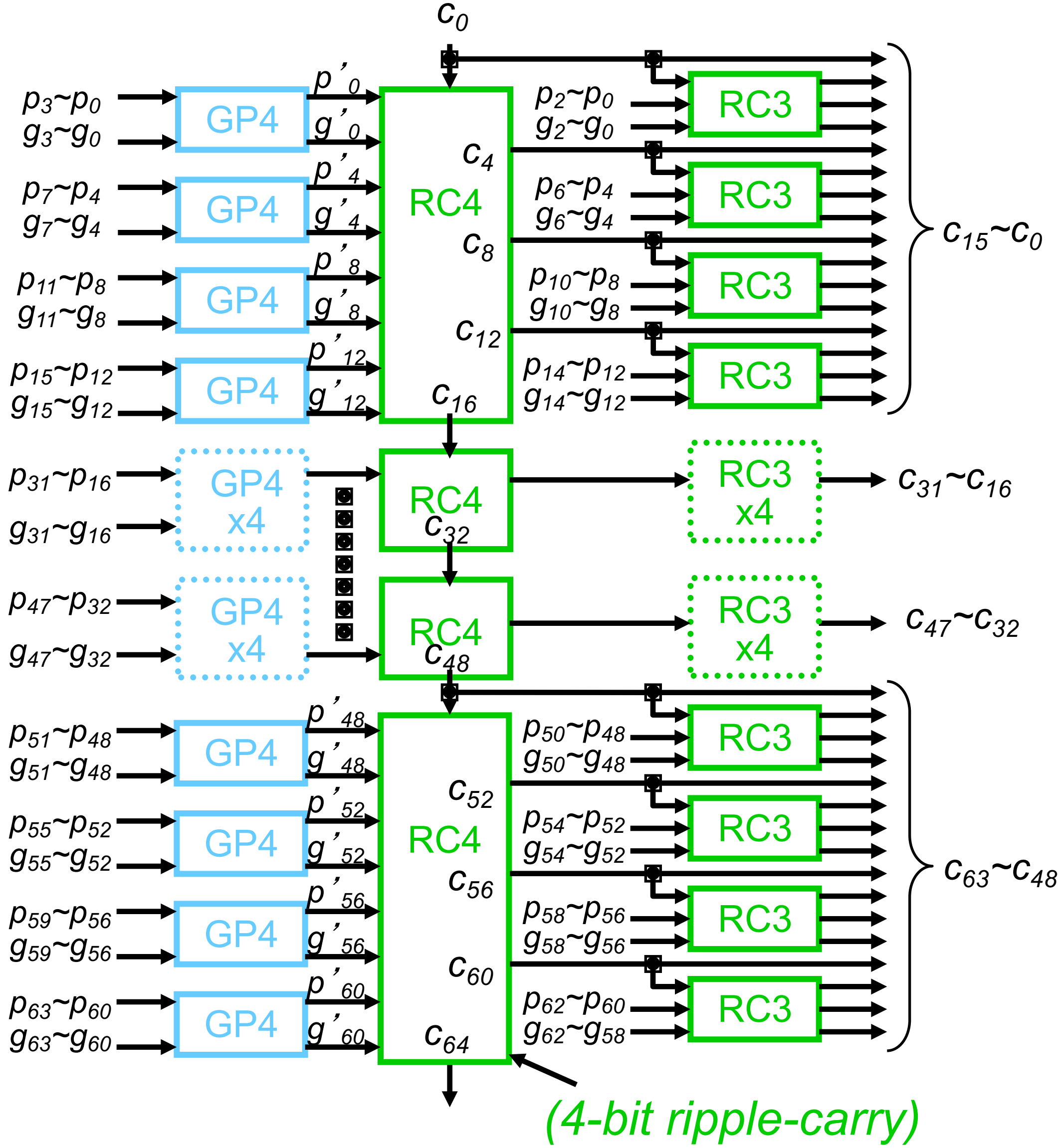
Hierarchical Carry-Lookahead (3)

- Delay estimation (assume the below normalized gate delays):
  - $D_{g1} = 1.0 : g_i = a_i \cdot b_i$
  - $D_{p1} = 2.0 : p_i = a_i \oplus b_i$
  - $D_s = 2.0 : s_i = p_i \oplus c_i$
  - $D_{c1} = 1.0 : c_{i+1} = g_i + p_i\, c_i$
  - $D_{c2} = 1.5 : c_{i+2} = g_{i+1} + p_{i+1}(g_i + p_i c_i)$
  - $D_{c3} = 2.0 : c_{i+3} = g_{i+2} + p_{i+2}(g_{i+1} + p_{i+1}(g_i + p_i c_i))$
  - $D_{c4} = 2.5 : c_{i+4} = g_{i+3} + p_{i+3}(g_{i+2} + p_{i+2}(g_{i+1} + p_{i+1}(g_i + p_i c_i)))$
  - $D_{g4} = 2.0 : g'_i = g_{i+3} + p_{i+3}(g_{i+2} + p_{i+2}(g_{i+1} + p_{i+1} g_i))$
  - $D_{p4} = 2.0 : p'_i = p_{i+3} p_{i+2} p_{i+1} p_i$
  - ➢ *64-bit ripple-carry adder :*
  - $D(a_0 \rightarrow c_{64}) = \max\{D_{g1}, D_{p1}\} + 64^* D_{c1} = 2.0 + 64.0 = 66.0$
  - $D(a_0 \rightarrow s_{63}) = \max\{D_{g1}, D_{p1}\} + 63^* D_{c1} + D_s = 2.0 + 63.0 + 2.0 = 67.0$
  - ➢ *64-bit carry-lookahead adder using 4^2-bit carry-lookahead:*
  - $D(a_0 \rightarrow c_{64}) = \max\{D_{g1}, D_{p1}\} + 2^*\max\{D_{g4}, D_{p4}\} + D_{c4} = 2.0 + 4.0 + 2.5 = 8.5$
  - $D(a_0 \rightarrow s_{63}) = \max\{D_{g1}, D_{p1}\} + 2^*\max\{D_{g4}, D_{p4}\} + 3^* D_{c3} + D_s$
    $= 2.0 + 4.0 + 6.0 + 2.0 = 14.0$

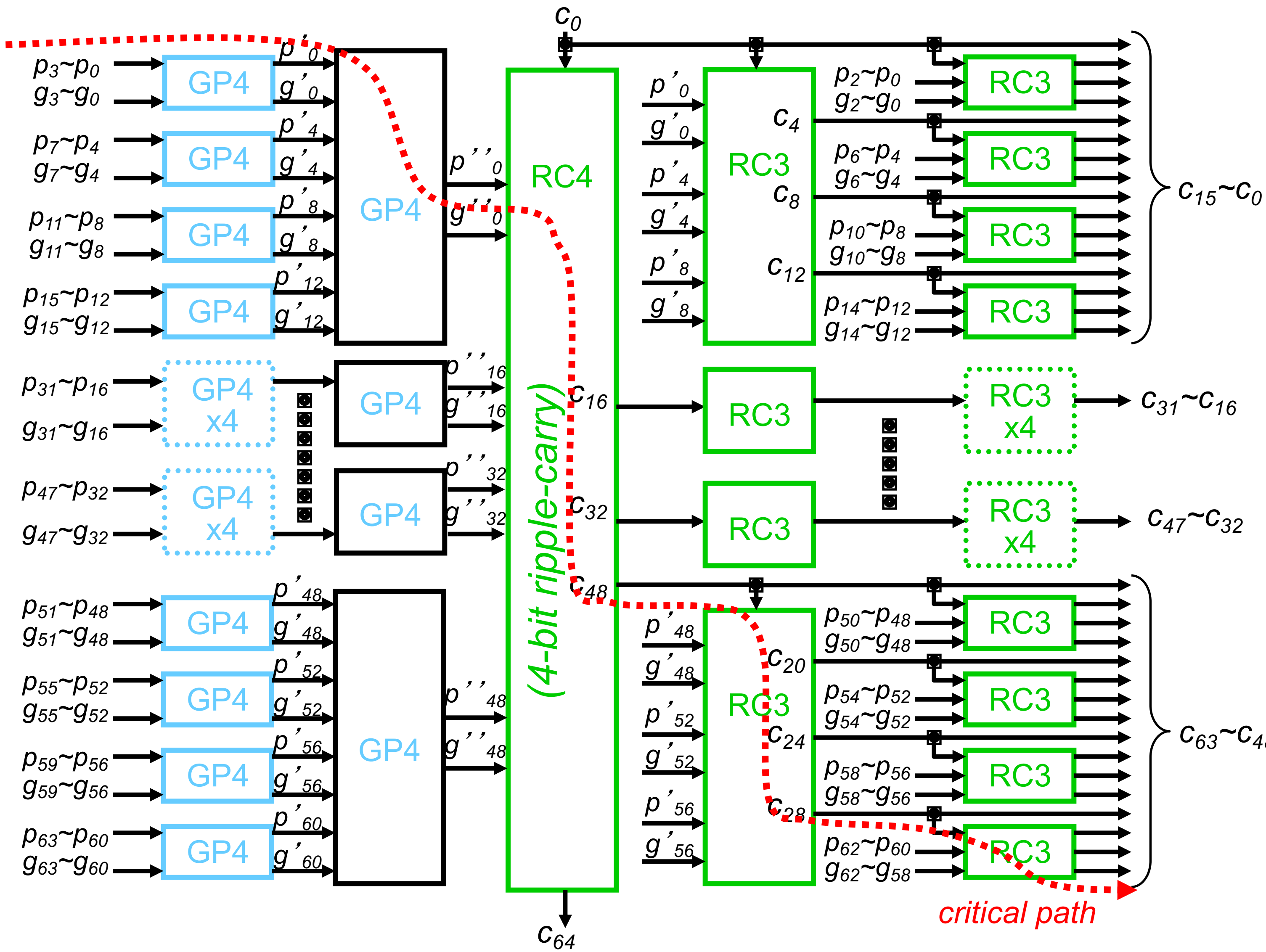
Hierarchical Carry-Lookahead (4)

- Area estimation (assume the following normalized areas):
  - $A_{g1} = 1.0 : g_i = a_i \cdot b_i$
  - $A_{p1} = 1.75 : p_i = a_i \oplus b_i$
  - $A_s = 1.75 : s_i = p_i \oplus c_i$
  - $A_{c1} = 1.25 : c_{i+1} = g_i + p_i c_i$
  - $A_{c2} = 2.00 : c_{i+2} = g_{i+1} + p_{i+1}(g_i + p_i c_i)$
  - $A_{c3} = 2.75 : c_{i+3} = g_{i+2} + p_{i+2}(g_{i+1} + p_{i+1}(g_i + p_i c_i))$
  - $A_{c4} = 3.50 : c_{i+4} = g_{i+3} + p_{i+3}(g_{i+2} + p_{i+2}(g_{i+1} + p_{i+1}(g_i + p_i c_i)))$
  - $A_{g4} = 2.75 : g'_i = g_{i+3} + p_{i+3}(g_{i+2} + p_{i+2}(g_{i+1} + p_{i+1} g_i))$
  - $A_{p4} = 1.5 : p'_i = p_{i+3} p_{i+2} p_{i+1} p_i$
  - ➢ *64-bit ripple-carry adder :*
  - $A = 64 * (A_{PG} + A_s + A_{c1}) = 64 * (2.75 + 1.75 + 1.25) = 368$
  - ➢ *64-bit carry-lookahead adder using 4^2-bit carry-lookahead:*
  - $A = 64 * (A_{PG} + A_s) + (16 + 4) * A_{PG4} + A_{LCA4} + (16 + 4) * A_{LCA3}$
    $= 64 * (2.75 + 1.75) + (16 + 4) * 4.25 + 9.5 + (16 + 4) * 6.0 = 502.5$

4-bit Generate-Propagate + Ripple-Carry Adder

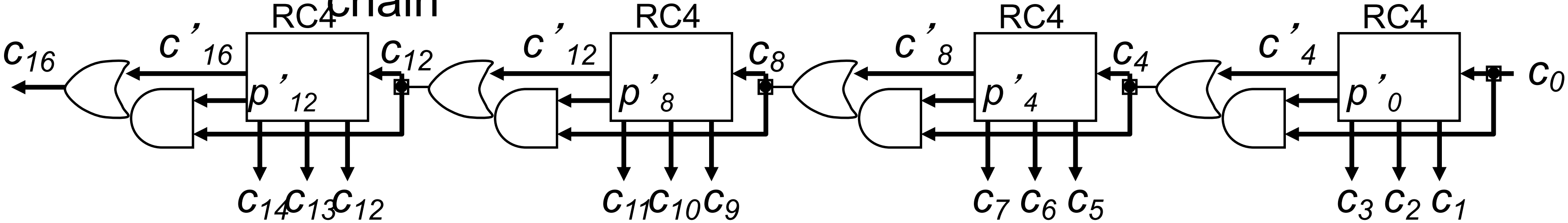


4^2-bit Generate-Propagate + Ripple-Carry Adder



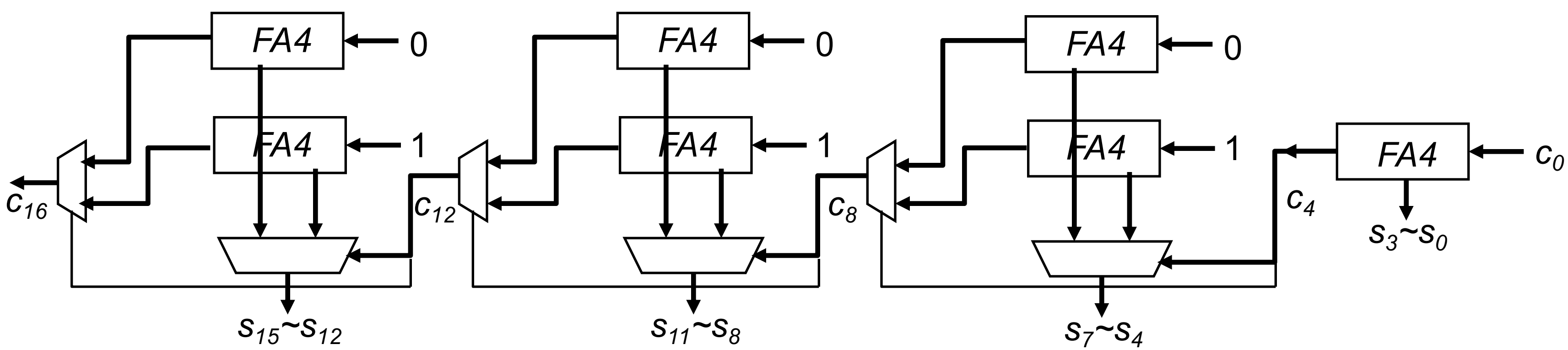
Carry-Skip (Carry-Bypass) Adder

- "propagate" → carry-skip condition
  - $c_{i+1} = g_i + p_i\, c_i$
  - $c_{i+2} = g_{i+1} + p_{i+1}\, c_{i+1}$
  - $c_{i+3} = g_{i+2} + p_{i+2}\, c_{i+2}$
  - $c'_{i+4} = g_{i+3} + p_{i+3}\, c_{i+3}$
  - $p'_i = p_{i+3} p_{i+2} p_{i+1} p_i$
  - $c_{i+4} = c'_{i+4} + p'_i c_i$
    - Carry input with value '1' will instantly propagate to the carry output if $p'_i = 1$, bypassing the ripple-carry chain



Carry-Select Adder

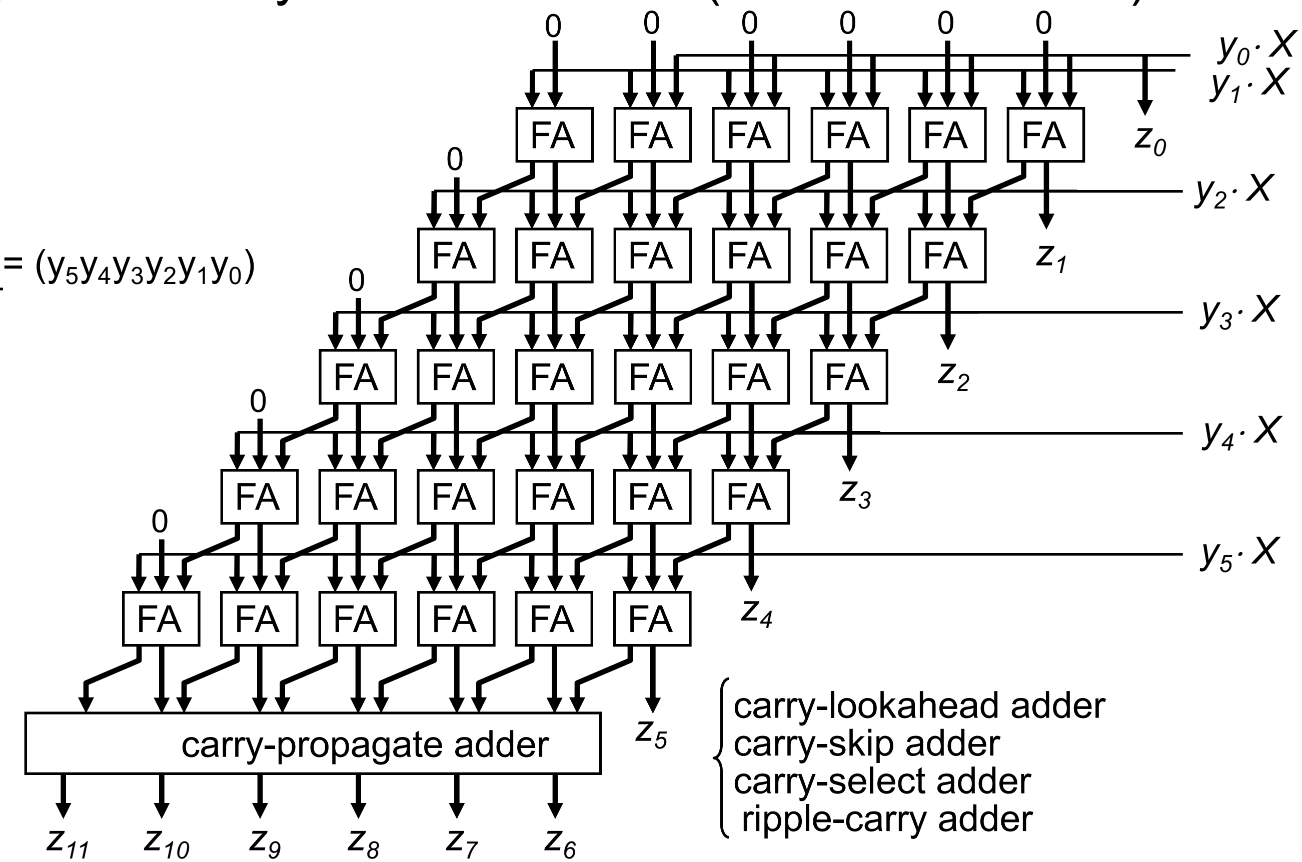
- Calculate the sum values and carry values in parallel on the assumptions of carry input values (0 and 1). Select the two values by the carry input value
- Some optimization on conditional adders is possible (but not much)



Carry-Save Adder

- Used in multiplier for accumulating a large number of partial products.
- Propagate the carry to the next level (not to the next bit)

```
    011010 X
  x 010011 Y = (y5y4y3y2y1y0)
  ------------
    011010
   011010
  000000
 000000
011010
000000
------------
000111101110 Z
```



High-Level Synthesis and Its Design Environment

- High-level synthesis converts the design described at algorithm-level to RTL

→ Needs a rich module library consisting of high-level circuit blocks (adders, multipliers, ALUs, decoders, RAM, ROM, etc.) as well as high quality standard cell library

→ Numerous circuit implementation techniques for arithmetic logic are captured into these libraries, and high-level synthesis provides a path for utilizing these design resource efficiently and intelligently.