EDA322
Digital Design

Lecture 2:
Logic Minimization, Adders

Ioannis Sourdis

Lecture 2 outline

- Boolean logic refresher
- Logic minimization
  - Karnaugh
- Adders
  - Ripple-carry,
  - Carry-select
  - Carry-Lookahead

Boolean Logic refresher

Boolean Algebra

1) $X \cdot 0 = 0$
2) $X \cdot 1 = X$
3) $X \cdot X = X$
4) $X \cdot \overline{X} = 0$
5) $X + 0 = X$
6) $X + 1 = 1$
7) $X + X = X$
8) $X + \overline{X} = 1$
9) $\overline{\overline{X}} = X$

10A) $X \cdot Y = Y \cdot X$
10B) $X + Y = Y + X$

Commutative Law

11A) $X(YZ) = (XY)Z$
11B) $X + (Y + Z) = (X + Y) + Z$

Associative Law

12A) $X(Y + Z) = XY + XZ$
12B) $X + (YZ) = (X + Y)(X + Z)$
12C) $(X + Y)(W + Z) = XW + XZ + YW + YZ$

Distributive Law

13A) $X + \overline{X}Y = X + Y$
13B) $\overline{X} + XY = \overline{X} + Y$
13C) $X + \overline{X}\,\overline{Y} = X + \overline{Y}$
13D) $\overline{X} + X\overline{Y} = \overline{X} + \overline{Y}$

Consensus Theorem

14A) $\overline{XY} = \overline{X} + \overline{Y}$
14B) $\overline{X + Y} = \overline{X}\ \overline{Y}$

DeMorgan's Theorem

Book Notations

- For AND logic function:
  - Book uses $\wedge$ or &
    - e.g., : $(a \wedge b)$ (a & b)
  - We are used to •
    - e.g., : (a•b)
- For OR logic function:
  - Book uses $\vee$ or |
    - e.g., : $(a \vee b)$ (a | b)
  - We are used to +
    - e.g., : (a+b)
- For NOT logic function:
  - Book uses $\neg$ or ~
    - e.g., : $(\neg a)$ (~a)
  - We are used to ’ or –
    - e.g., : (a’) ($\bar{a}$)
- For XOR logic function:
  - Book and we use $\oplus$
    - e.g., : $(a \oplus b)$

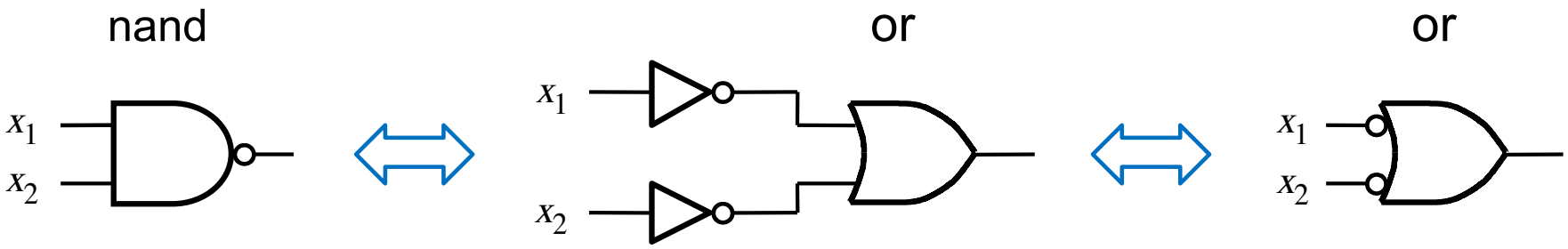
Algebraic Forms of Logic Functions

- ***Literal***: A variable, complemented or uncomplemented.
- ***Product***: Literals ANDed together.
- ***Sum***: Literals ORed together.

- ***SOP (Sum of Products)***:
- ORing product terms
- $f(A, B, C) = ABC + A'C + B'C$

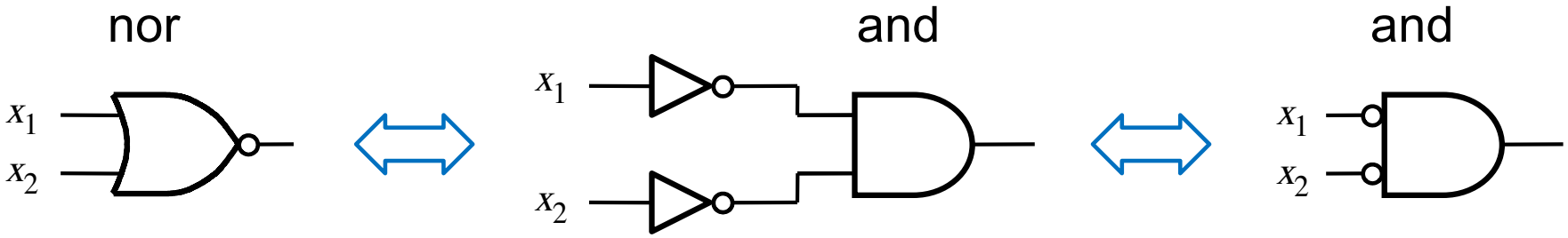
- ***POS (Product of Sums)***
- ANDing sum terms
- $f(A, B, C) = (A' + B' + C')(A + C')(B + C')$

de Morgan theorem



(a) $\overline{x_1 x_2} = \bar{x}_1 + \bar{x}_2$

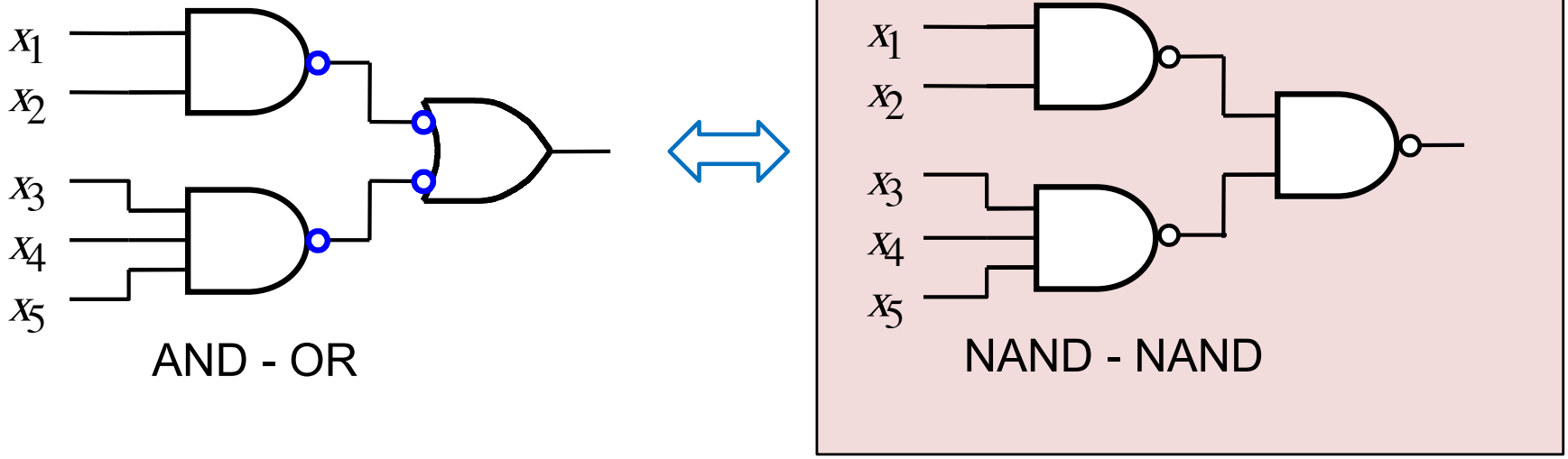
The inversion of a product is the sum of the inverted variable



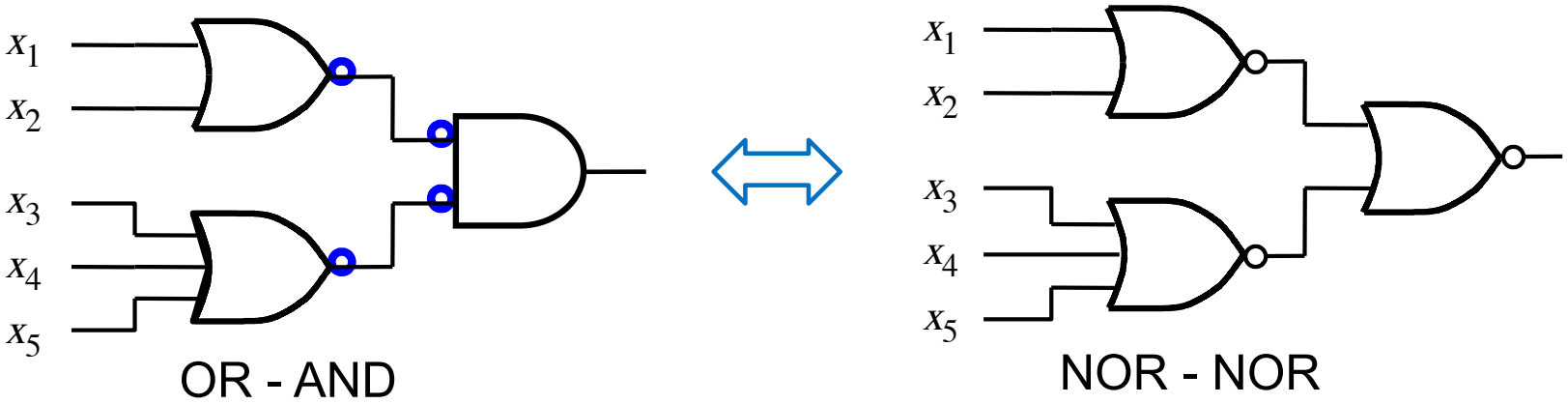
(b) $\overline{x_1 + x_2} = \bar{x}_1 \bar{x}_2$

The inversion of a sum is the product of the inverted variable

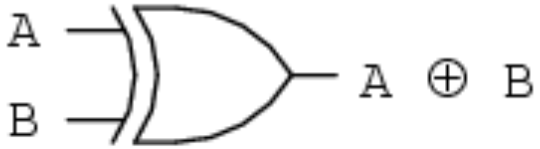
SoP


x_1
x_2
x_3
x_4
x_5
AND - OR
x_1
x_2
x_3
x_4
x_5
NAND - NAND

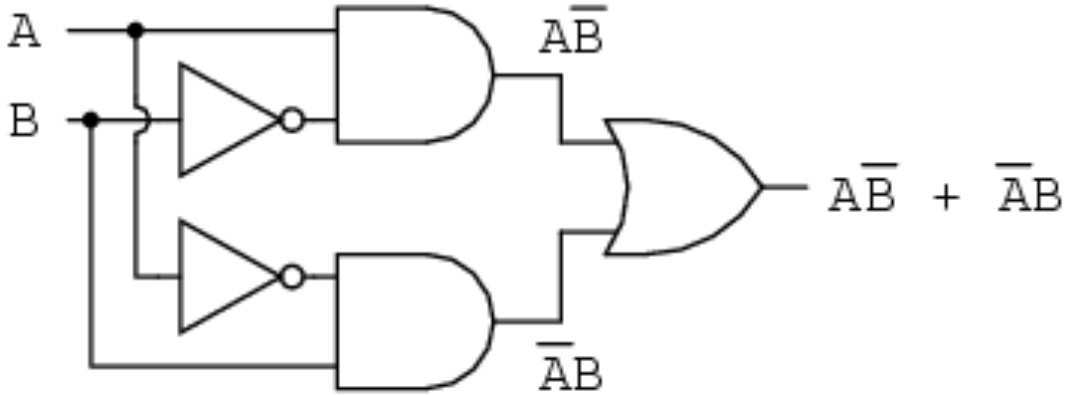
PoS


x_1
x_2
x_3
x_4
x_5
OR - AND
x_1
x_2
x_3
x_4
x_5
NOR - NOR

The XOR gate



. . . is equivalent to . . .



$$A \oplus B = A\overline{B} + \overline{A}B$$

The XOR gate

Commutative Law

$$A \oplus B = B \oplus A$$

Associative Law

$$(A \oplus B) \oplus C = A \oplus (B \oplus C) = A \oplus B \oplus C$$

Distributive Law

$$(AB) \oplus (AC) = A(B \oplus C)$$

Algebraic Forms of Logic Functions

- A ***minterm*** is a product term in which all the variables appear exactly once either complemented or uncomplemented.
- ***Canonical Sum of Products*** (***canonical SOP***):
  - Represented as a sum of minterms only.
  - ***Example***: $f_1(A,B,C) = A'BC' + ABC' + A'BC + ABC$
  - Minterms of three variables:

Minterm	Minterm Code	Minterm Number
$A'B'C'$	000	m_0
$A'B'C$	001	m_1
$A'BC'$	010	m_2
$A'BC$	011	m_3
$AB'C'$	100	m_4
$AB'C$	101	m_5
ABC'	110	m_6
ABC	111	m_7

Algebraic Forms of Logic Functions

- A ***maxterm*** is a sum term in which all the variables appear exactly once either complemented or uncomplemented.
- ***Canonical Product of Sums*** (***canonical POS***):
  - Represented as a product of maxterms only.
  - ***Example***: $f_2(A,B,C) = (A+B+C)(A+B+C')(A'+B+C)(A'+B+C')$
- Maxterms of three variables:

Maxterm	Maxterm Code	Maxterm Number
$A+B+C$	000	M_0
$A+B+C'$	001	M_1
$A+B'+C$	010	M_2
$A+B'+C'$	011	M_3
$A'+B+C$	100	M_4
$A'+B+C'$	101	M_5
$A'+B'+C$	110	M_6
$A'+B'+C'$	111	M_7

Canonical (Normal) Forms

SoP canonical form

F= Σ(1,3,5,6,7)

F = *001* *011* *101* *110* *111*

F = A′B′C + A′BC + AB′C + ABC′ + ABC

A	B	C	F	F′	#
0	0	0	0	1	**0**
0	0	1	1	0	**1**
0	1	0	0	1	**2**
0	1	1	1	0	**3**
1	0	0	0	1	**4**
1	0	1	1	0	**5**
1	1	0	1	0	**6**
1	1	1	1	0	**7**

F= Π(0,2,4)

PoS canonical form

F = (A + B + C) (A + B′ + C) (A′ + B + C)

F = *000* *010* *100*

F′ = A′B′C′ + A′BC′ + AB′C′

canonical form ≠ minimal form

Conversion between canonical forms

Relation between maxterms and minterms of a function

$$F(x,y,z) = \Sigma(1,3,5,6,7) = \Pi(0,2,4)$$

… and for the inverted function:

$$F'(x,y,z) = \Pi(1,3,5,6,7) = \Sigma(0,2,4)$$

Shannon's expansion theorem

- ***Shannon's expansion theorem***

  (a). $f(x_1, x_2, \ldots, x_n) = x_1 * f(1, x_2, \ldots, x_n) + x_1' * f(0, x_2, \ldots, x_n)$

  (b). $f(x_1, x_2, \ldots, x_n) = [x_1 + f(0, x_2, \ldots, x_n)]\ [x_1' + f(1, x_2, \ldots, x_n)]$

Example:

$F(x_1, x_2, x_3) = x_1x_2 + x_1x_3 + x_2x_3$

$= x_1'F(0, x_2, x_3) + x_1F(1, x_2, x_3)$

$= x_1' (0x_2 + 0x_3 + x_2x_3) + x_1(1x_2 + 1x_3 + x_2x_3)$

$= x_1' (x_2x_3) + x_1(x_2 + x_3 + x_2x_3)$

$= x_1' (x_2x_3) + x_1(x_2 + x_3(1 + x_2))$

$= x_1' (x_2x_3) + x_1(x_2 + x_3)$

Derivation of Canonical Forms

- Derive canonical PoS or SoP using Boolean algebra.
- ***Shannon's expansion theorem***

  (a). $f(x_1, x_2, \ldots, x_n) = x_1 * f(1, x_2, \ldots, x_n) + x_1' * f(0, x_2, \ldots, x_n)$

  (b). $f(x_1, x_2, \ldots, x_n) = [x_1 + f(0, x_2, \ldots, x_n)]\ [x_1' + f(1, x_2, \ldots, x_n)]$

Convert a function to SoP (Two alternatives, second is simpler!)

1. Apply Shannon Theorem until you get sum of minterms
2. Get the function to a SoP form, as follows:
   - if a product of literals is a minterm, keep it
   - for every variable x_i that doesn't exist in a product we add $(x_i + x_i')$, e.g.:
     - $x_1 * x_2 = x_1 * x_2 * (x_3 + x_3') = x_1 * x_2 * x_3 + x_1 * x_2 * x_3'$
   - Continue and delete redundant products

Derivation of Canonical Forms

- Derive canonical PoS or SoP using Boolean algebra.
- ***Shannon's expansion theorem***

  (a). $f(x_1, x_2, \ldots, x_n) = x_1 * f(1, x_2, \ldots, x_n) + x_1' * f(0, x_2, \ldots, x_n)$

  (b). $f(x_1, x_2, \ldots, x_n) = [x_1 + f(0, x_2, \ldots, x_n)]\,[x_1' + f(1, x_2, \ldots, x_n)]$

- ***Example***: $f(A,B,C) = AB + AC' + A'C$
  - $f(A,B,C) = AB + AC' + A'C = A\,f(1,B,C) + A'\,f(0,B,C)$

    $= A(1 \times B + 1 \times C' + 1' \times C) + A'(0 \times B + 0 \times C' + 0' \times C) = A(B + C') + A'C$
  - $f(A,B,C) = A(B + C') + A'C = B[A(1+C') + A'C] + B'[A(0 + C') + A'C]$

    $= B[A + A'C] + B'[AC' + A'C] = AB + A'BC + AB'C' + A'B'C$
  - $f(A,B,C) = AB + A'BC + AB'C' + A'B'C$

    $= C[AB + A'B \times 1 + AB' \times 1' + A'B' \times 1] + C'[AB + A'B \times 0 + AB' \times 0' + A'B' \times 0]$

    $= ABC + A'BC + A'B'C + ABC' + AB'C'$

Derivation of Canonical Forms

- Derive canonical PoS or SoP using Boolean algebra.
- ***Shannon's expansion theorem***

  (a). $f(x_1, x_2, \ldots, x_n) = x_1 * f(1, x_2, \ldots, x_n) + x_1' * f(0, x_2, \ldots, x_n)$

  (b). $f(x_1, x_2, \ldots, x_n) = [x_1 + f(0, x_2, \ldots, x_n)]\ [x_1' + f(1, x_2, \ldots, x_n)]$

- ***Example***: $f(A,B,C) = AB + AC' + A'C$

  $f(A,B,C) = AB + AC' + A'C =$

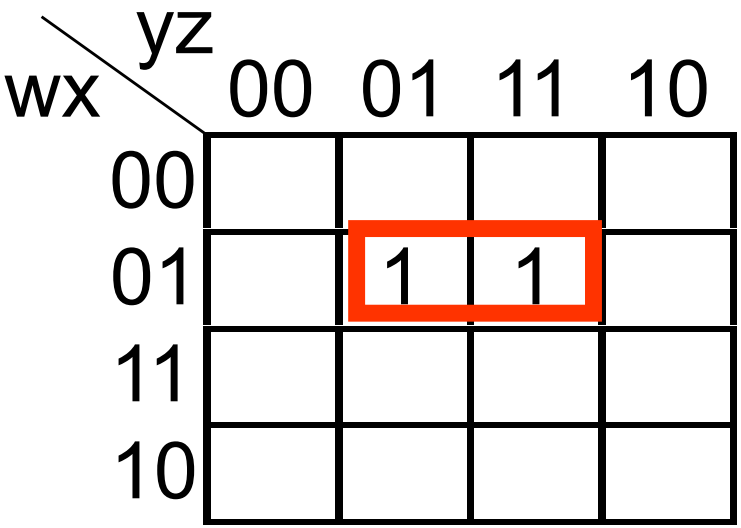
  $= AB(C+C') + AC'(B+B') + A'C(B+B')$

  $= ABC + ABC' + ABC' + AB'C' + A'BC + A'B'C$

  $= ABC + ABC' + AB'C' + A'BC + A'B'C$

Logic Simplification

Karnaugh-diagram

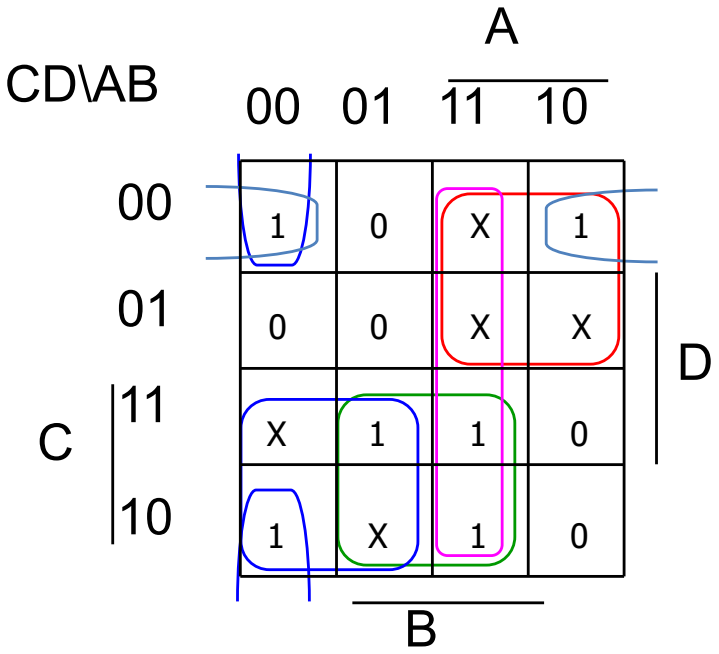


W'XY'Z + W'XYZ =
(W'XZ) (Y'+Y) = W'XZ

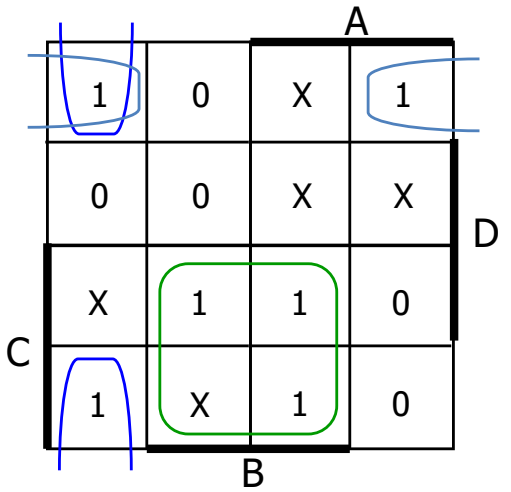
Note: (Y'+Y)=1

Karnaugh of 4 variables

Minimize function F = Σ m(0, 2, 7, 8, 14, 15) + d(3, 6, 9, 12, 13)

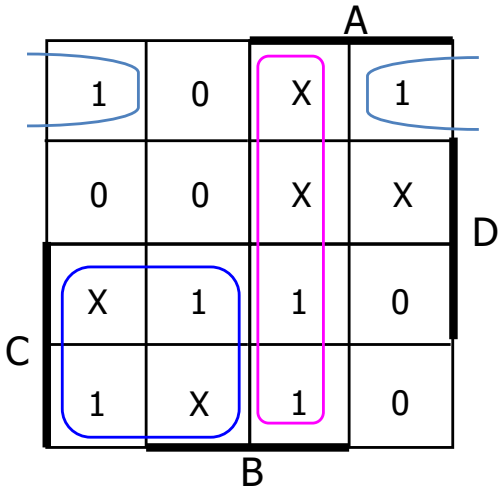


F = ~~AC' +~~ ~~A'C +~~ **BC +** ~~AB +~~ **A'B'D' + B'C'D'**

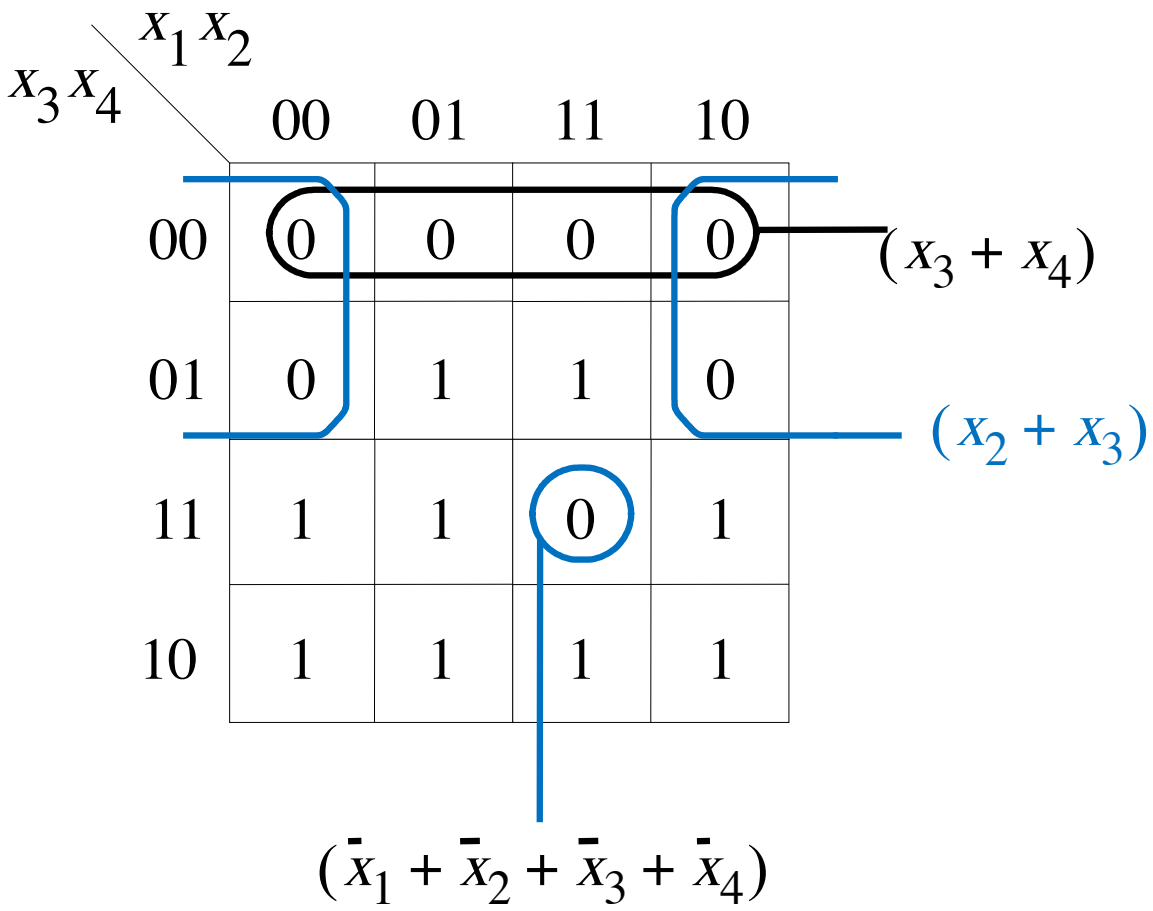


F = BC + A'B'D' + B'C'D'

F = A'C + AB + B'C'D'



Product of Sums (maxterms)

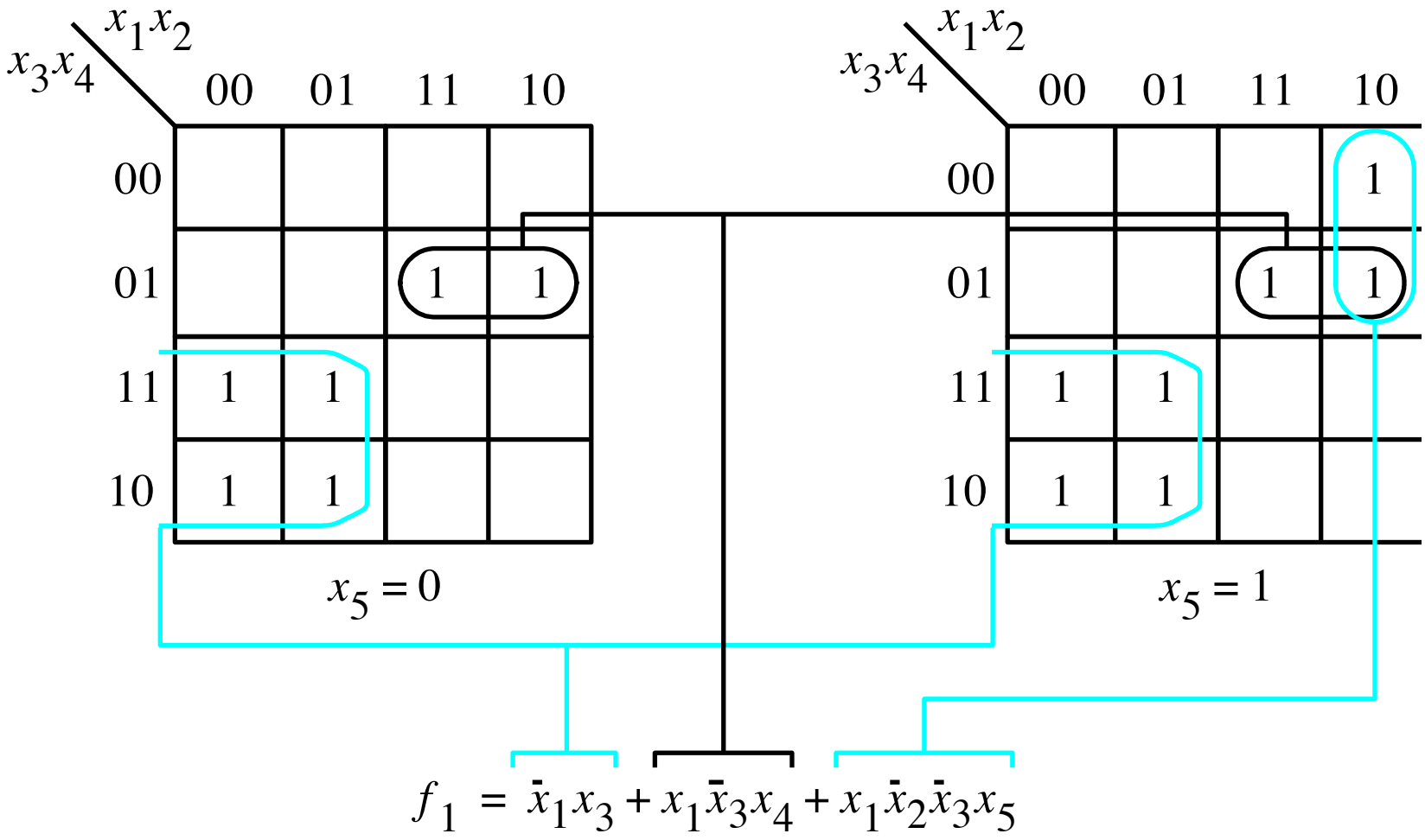


$$\text{POS } f(x_1,\ldots,x_4) = \Pi\, M(0, 1, 4, 8, 9, 12, 15).$$

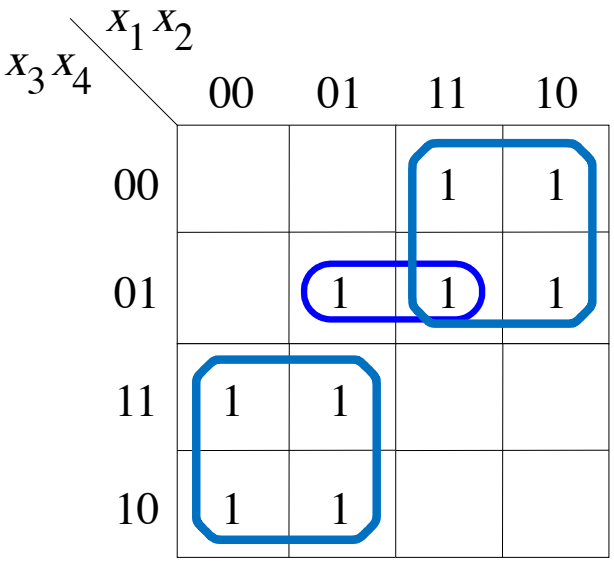
$$f = (x_3+x_4)(x_2+x_3)(\bar{x}_1+\bar{x}_2+\bar{x}_3+\bar{x}_4)$$

How would one choose between PoS and SoP?

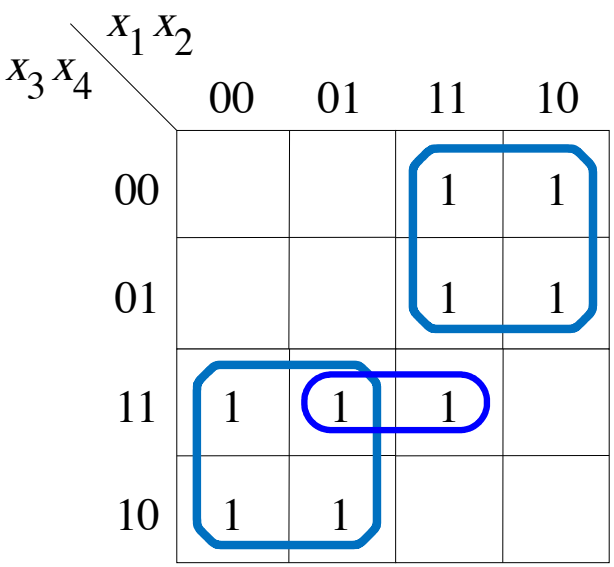
Karnaugh of 5 variables



Minimizing multiple functions

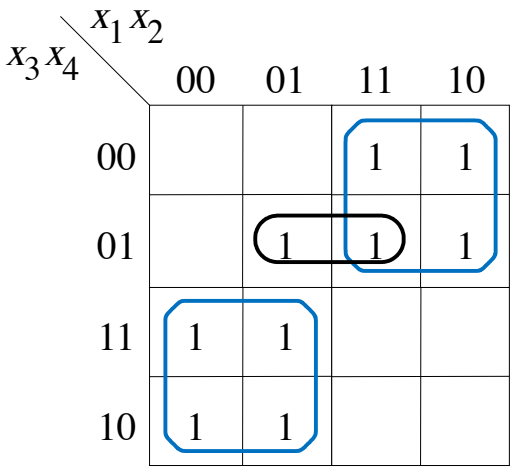


(a) Function f_1

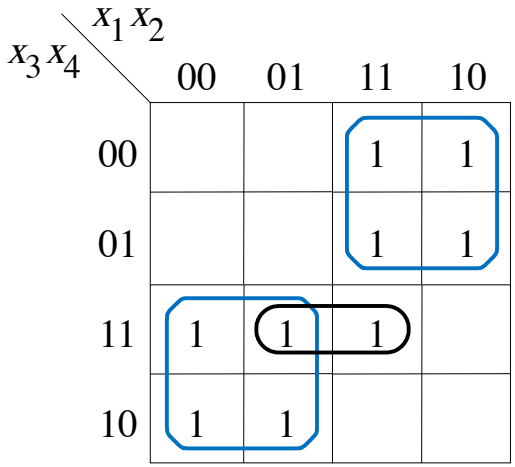


(b) Function f_2

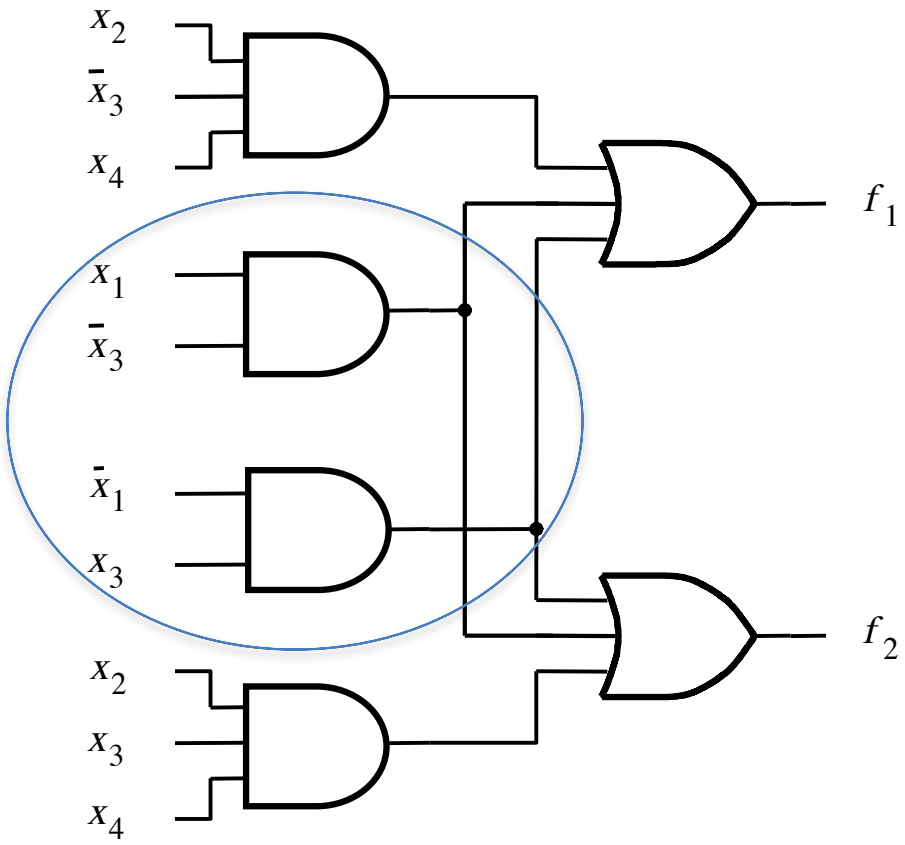
Minimizing multiple functions



(a) Function f_1



(b) Function f_2

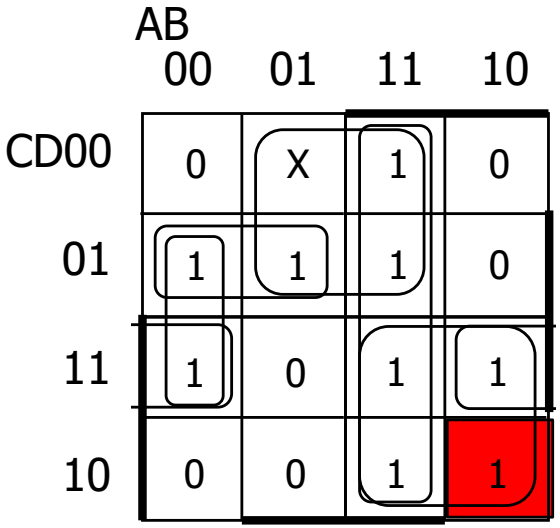


(c) Combined circuit for f_1 and f_2

Definition of terms

- A function *F* **covers** a function *g*, if it takes the value ’1’ when function *g* does.
- **Implicant**: a product of literals of a function *f* for which *f* gets the value ’1’.
- **Prime implicant:** an implicant that cannot be covered by a more general (more reduced - meaning with fewer literals) implicant.
- **Essential prime implicant:** a prime implicant of function *f* that includes a minterm, not included by any other prime implicant of the function.

Examples to illustrate terms



6 **prime implicants**:

A'B'D, BC', AC, A'C'D, AB, B'CD

essential

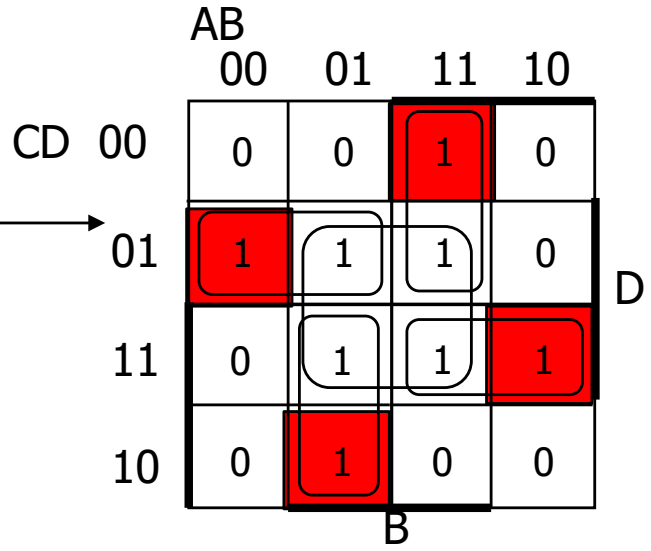
minimum cover: AC + BC' + A'B'D

5 **prime implicants**:

BD, ABC', ACD, A'BC, A'C'D

Essential prime implicants

minimum cover: 4 essential implicants



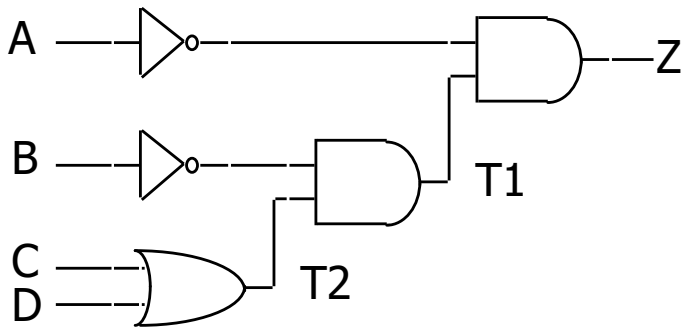
From Boolean expressions to logic gates

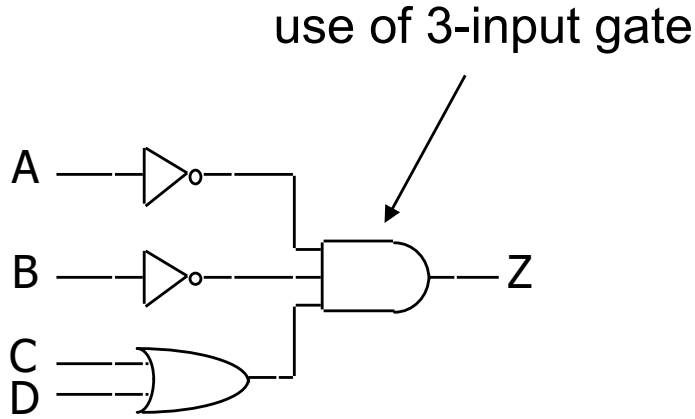
- More than one way to map expressions to gates
  - e.g., $F = A' \bullet B' \bullet (C + D) = (A' \bullet (B' \bullet (C + D)))$

$$\overline{\overline{T2}}$$
$$\overline{T1}$$

VHDL:

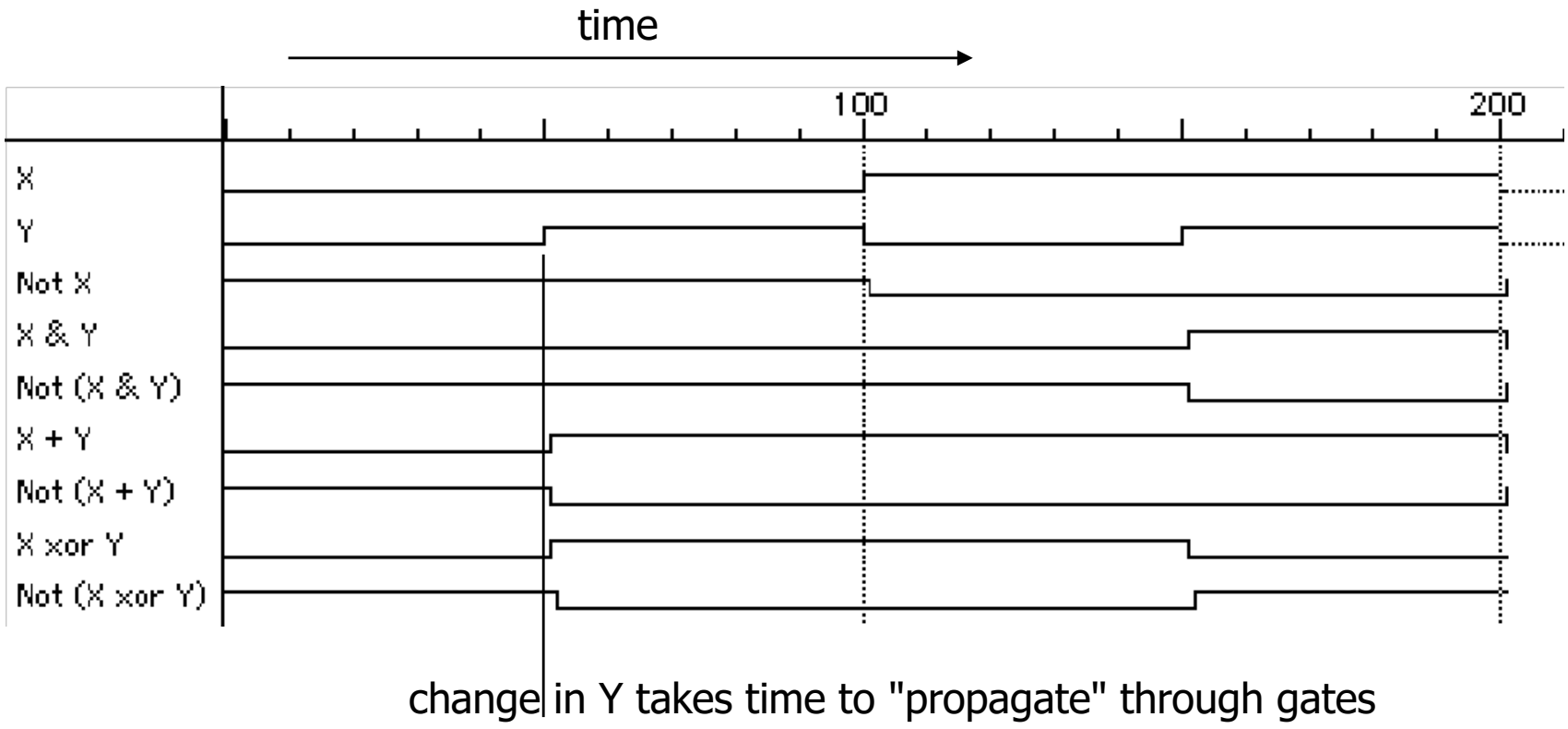
```
F<= not A and (not B and ( C or D )));
```





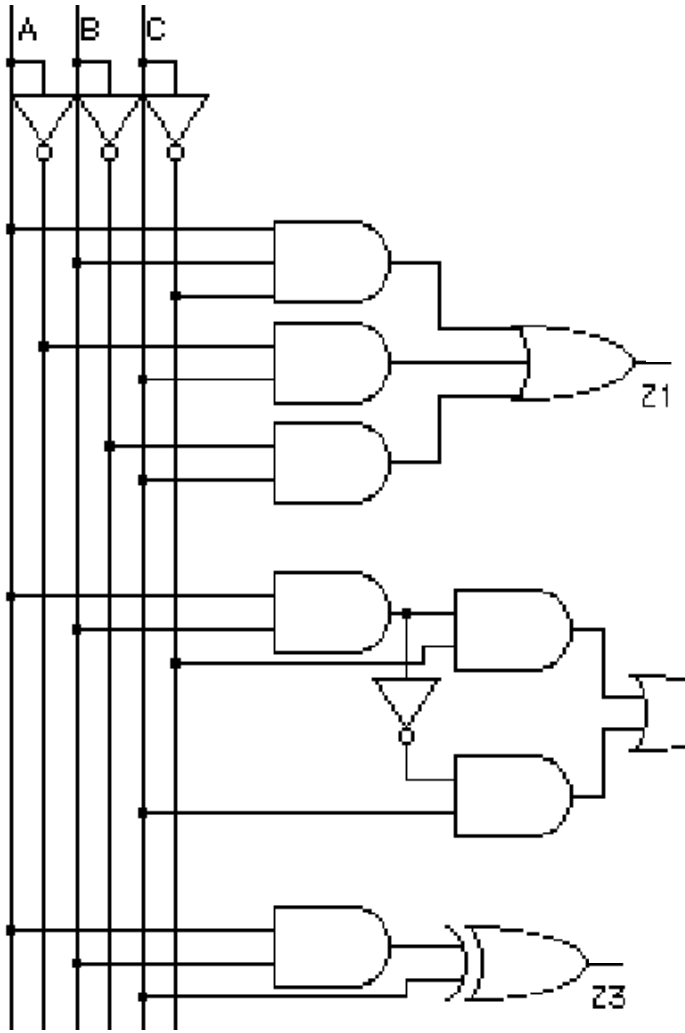
Waveform view of logic functions

- Note how edges don't line up exactly
- It takes time for a gate to switch its output!



Choosing different implementations of a function

A	B	C	Z
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

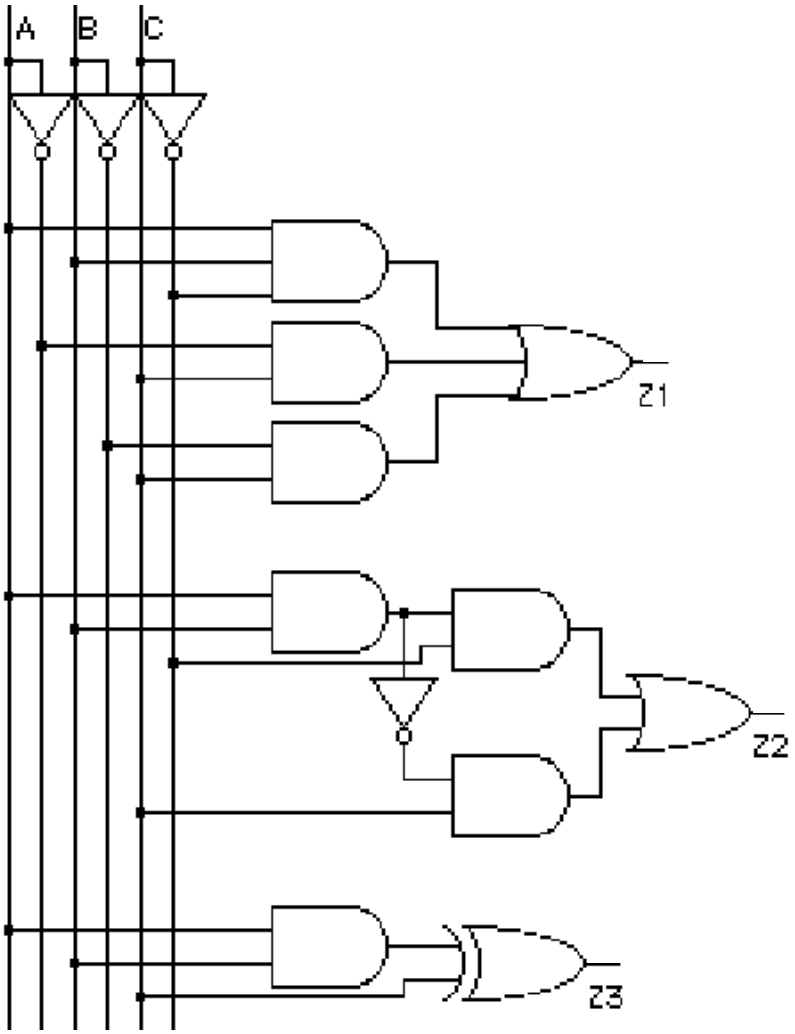


two-level realization
(we don't count NOT gates)

multi-level realization
(gates with fewer inputs)

XOR gate (easier to draw
but costlier to build)

What is the difference?



- Delay
  - Number of gates in a row
- Area
  - fewer gates in total or
  - Smaller circuit
- Power?

- Tradeoffs:
  - More Area for lower delay
  - Higher Delay for lower power

Quiz 2-1

http://m.socrative.com/student/#joinRoom

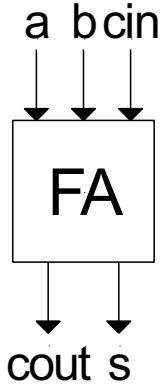
room number: **713113**

- Q1: which is the number of minterm AB'C
- Q2: which is the number of maxterm A+B'+C
- Q3: What are the maxterms of F(x0,x1,x2)=Σ(0,1,2,3,4)
- Q4: What are the minterms of F' if F(x0,x1,x2)=Σ(0,1,2,3,4)
- Q5: Expand F(x0,x1,x2)= x0*x1 to its canonical SoP

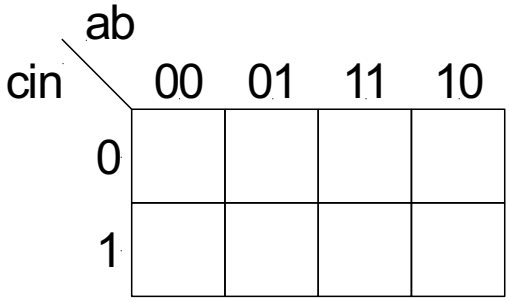
Adders

Adders

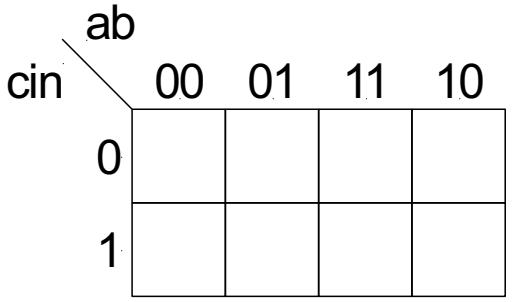
Full-adder cell (FA) revisited:



a b cin	cout	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

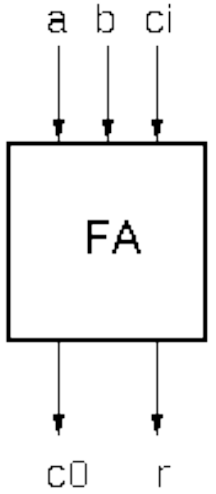


cout



s

Boolean functions for the full adder



R = AB'Cin’ + A'BCin’ + ABCin + A'B'Cin

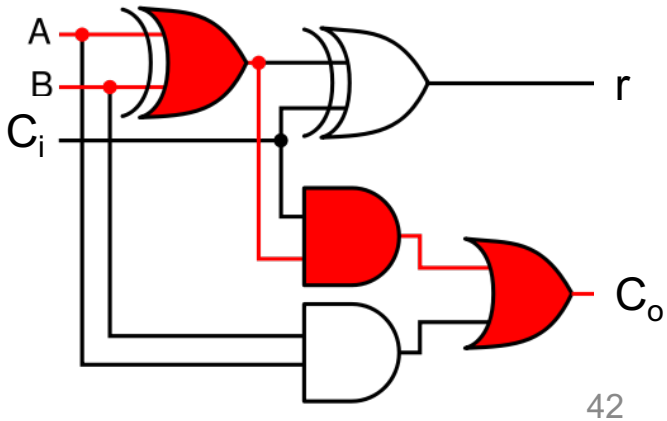
= Cin'(AB'+A'B) + Cin(AB+A'B')

=Cin'(AB'+A'B) + Cin (AB'+A'B)’

= Cin'(A xor B) + Cin (A xor B)’

=Cin xor (A xor B)

Critical path:
The longest path from an input to an output



Cout=AB'Cin + A'Bcin + AB =

=Cin(AB'+A'B) + AB

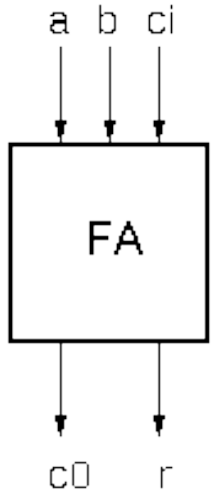
=Cin(A xor B) + AB

Ripple-Carry Adder

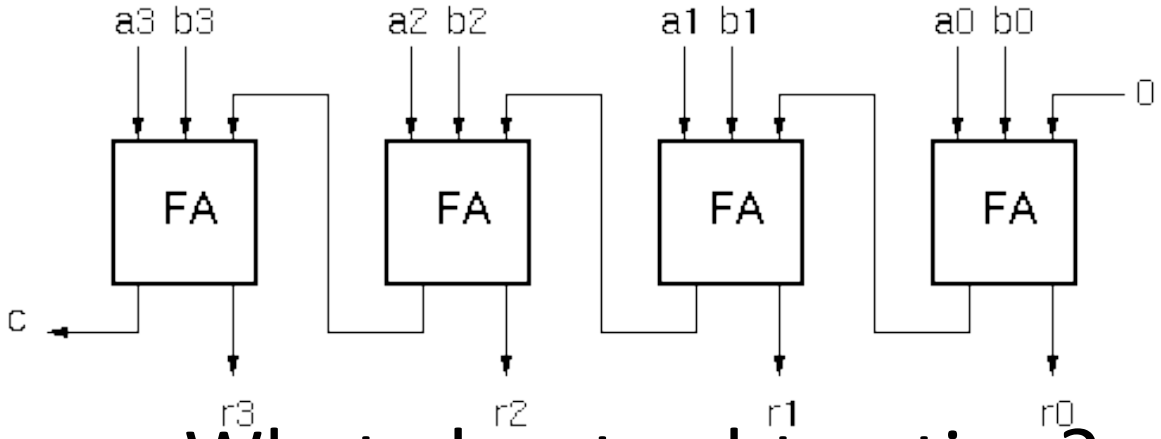
- Each cell:

  $r_i = a_i$ XOR b_i XOR c_{in}

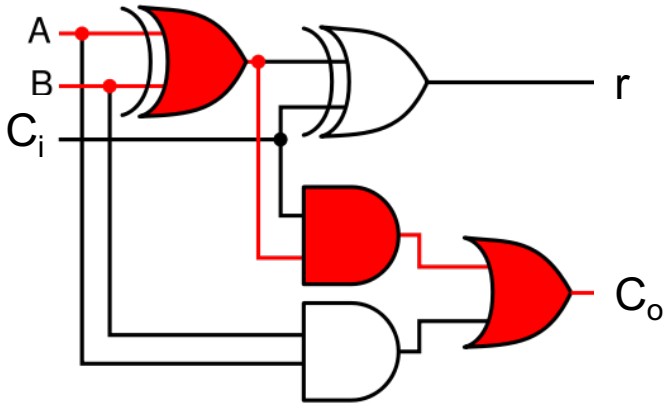
  $c_{out} = c_{in}(a_i$ XOR $b_i) + a_i b_i$



- 4-bit adder:



"Full adder cell"



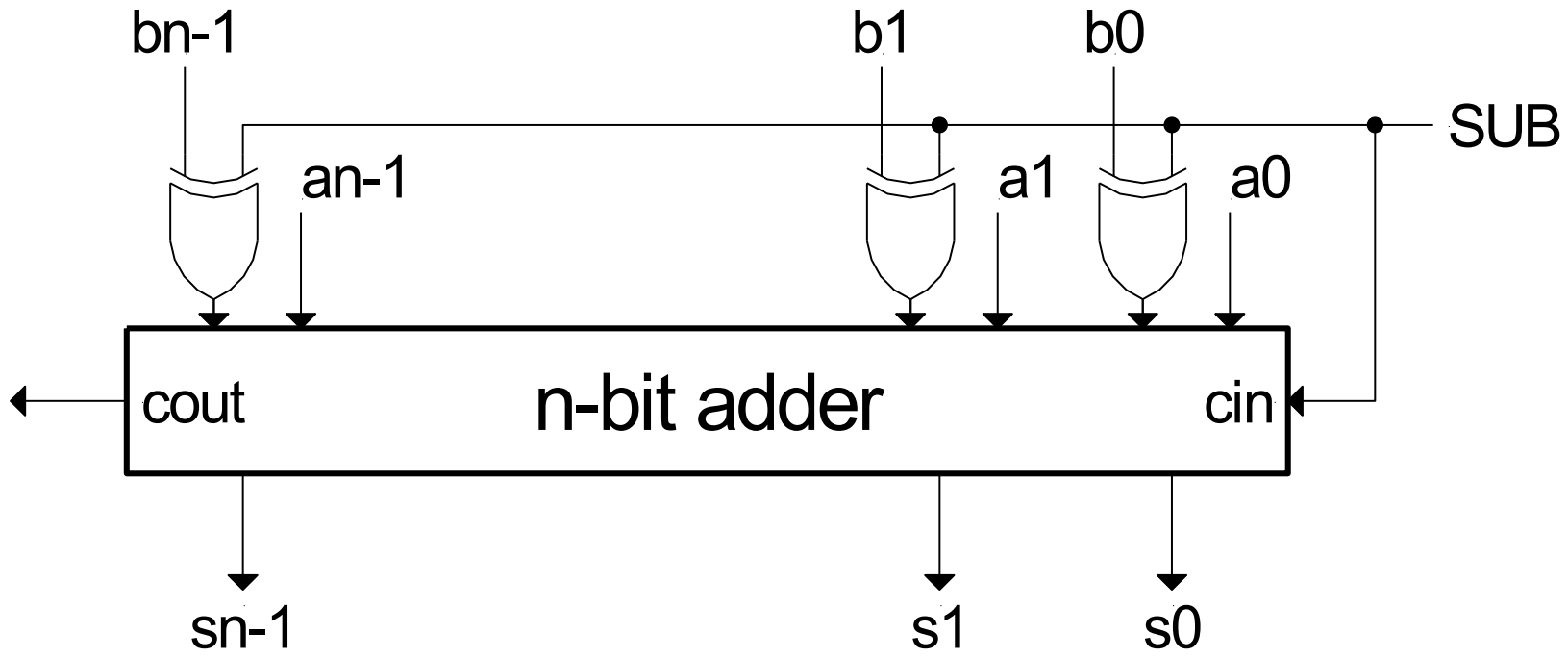
- What about subtraction?

Subtractors

A - B = A + (-B)

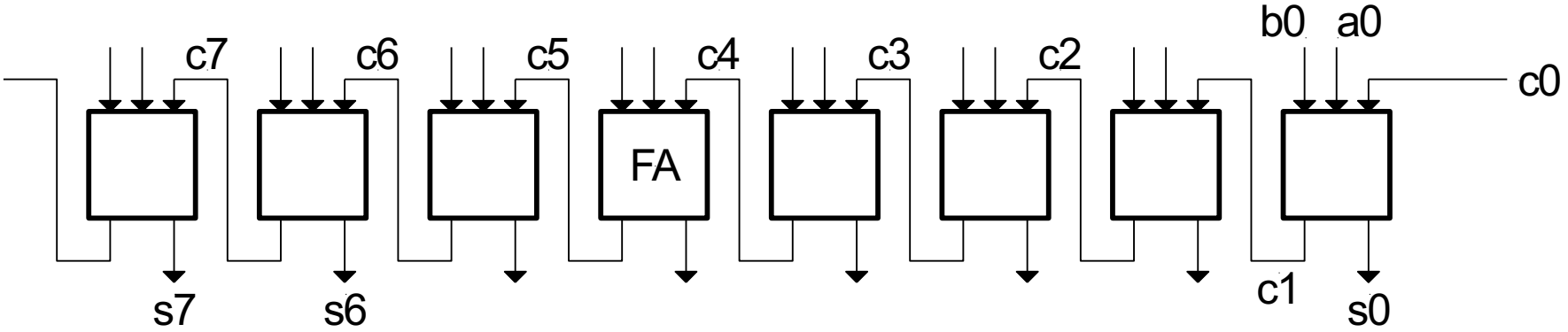
How do we form -B?

1. complement B
2. add 1



Adders (cont.)

Ripple Adder



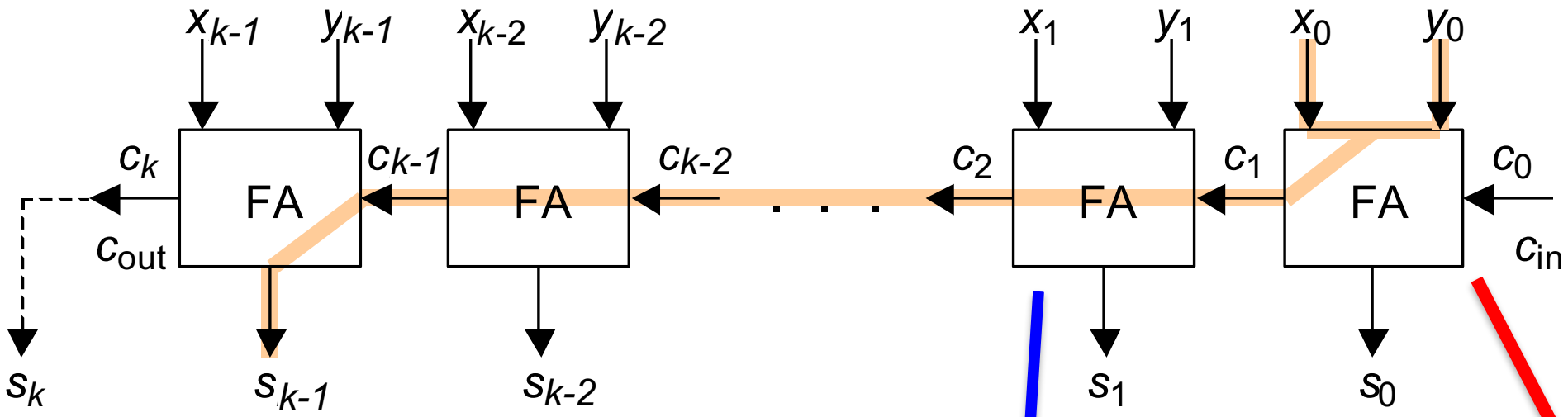
Ripple adder is inherently slow because, in general s7 must wait for c7 which must wait for c6 …

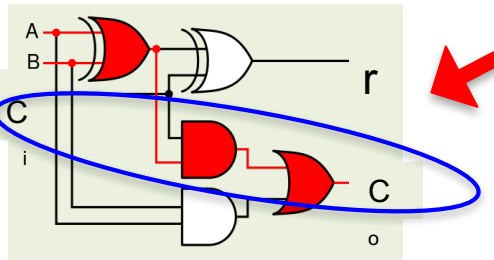
T is O(n), ***Cost*** is O(n)

What's the Critical path here?

Critical Path Through a Ripple-Carry Adder

$$T_{\text{ripple-add}} = T_{FA}(x,y \rightarrow c_{out}) + (k-2) \times T_{FA}(c_{in} \rightarrow \text{cout}) + T_{FA}(c_{in} \rightarrow s)$$

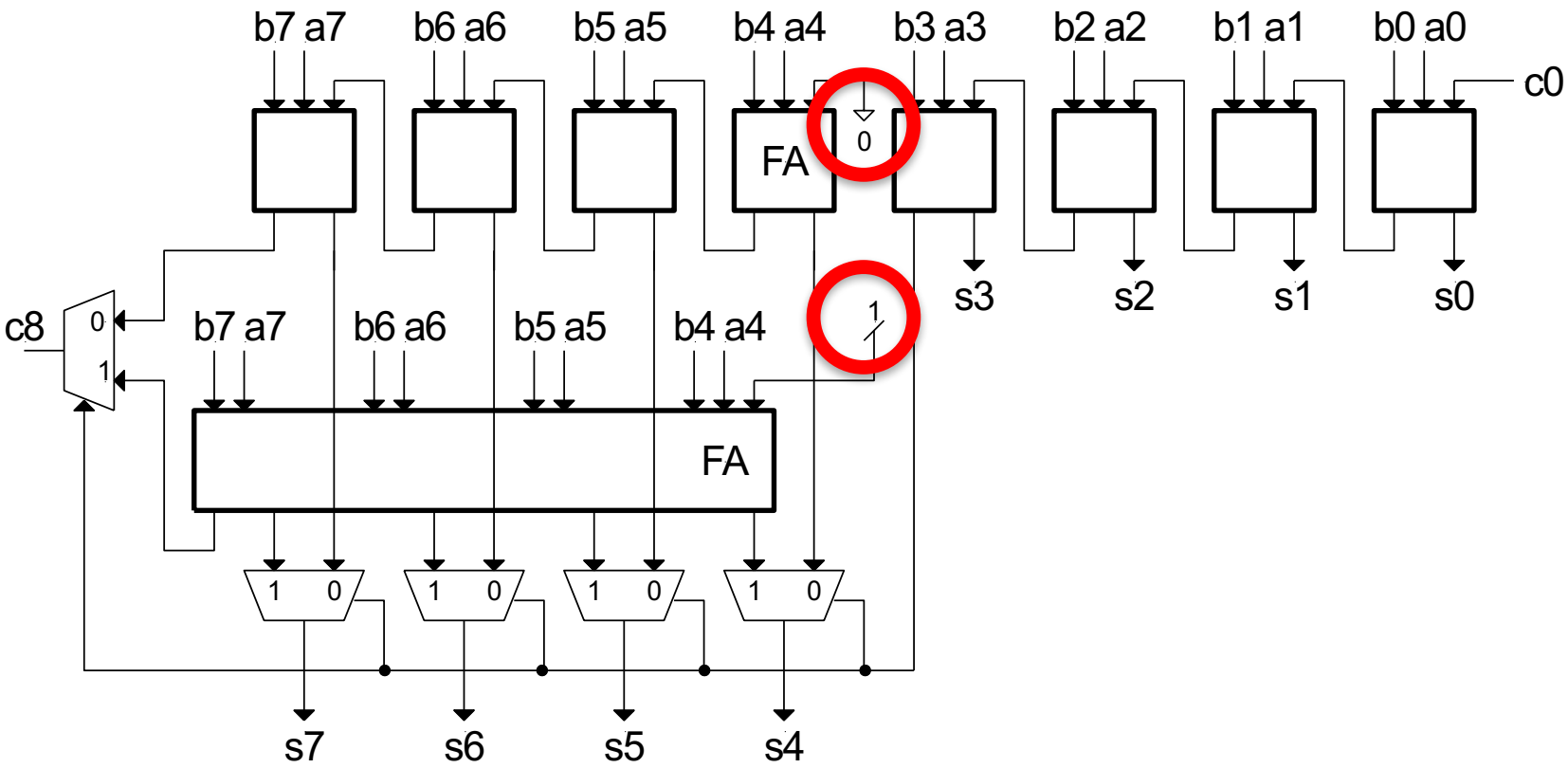




How do we make it faster, perhaps with more cost?

Mechanical adder: https://youtu.be/GcDshWmhF4A

Carry Select Adder

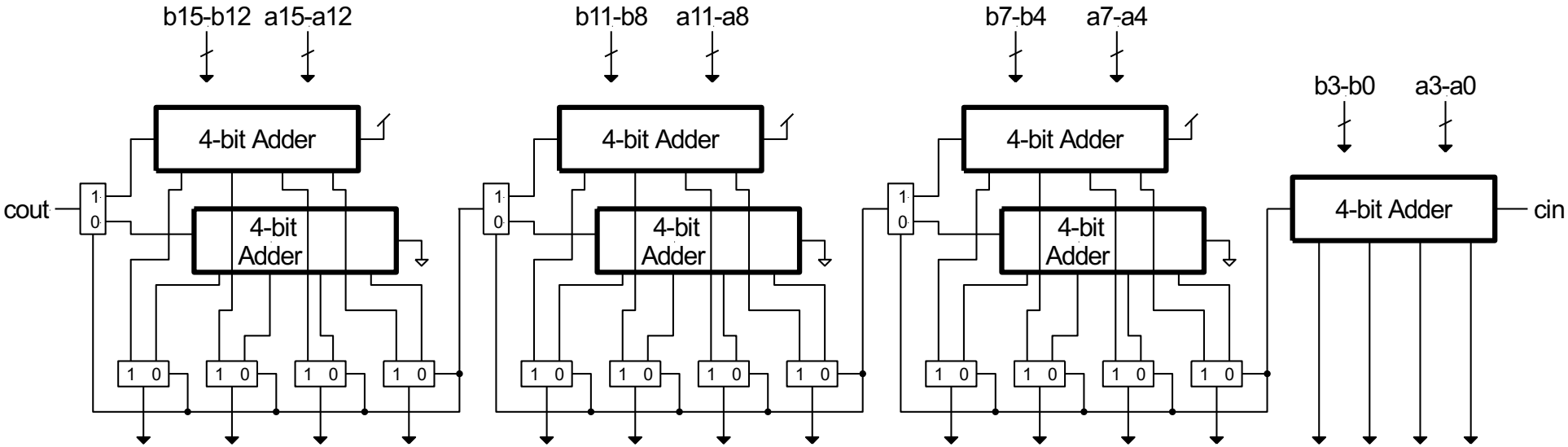


$$T = T_{ripple\_adder} / 2 + T_{MUX}$$

$$COST = 1.5 * COST_{ripple\_adder} + (n+1) * COST_{MUX}$$

Carry Select Adder

- Extending Carry-select to multiple blocks

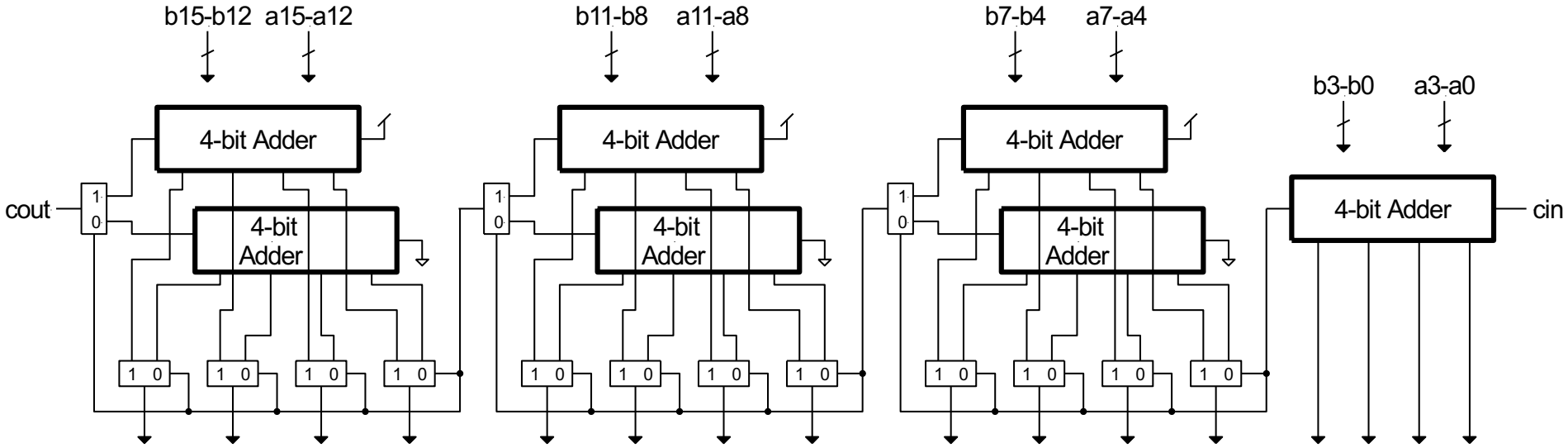


- What is the optimal # of blocks and # of bits/block?
  - If # blocks too large delay dominated by total mux delay
  - If # blocks too small delay dominated by adder delay

$$\sqrt{N} \text{ stages of } \sqrt{N} \text{ bits}$$

T is $O(\sqrt{N})$,
Cost ≈2*ripple + muxes

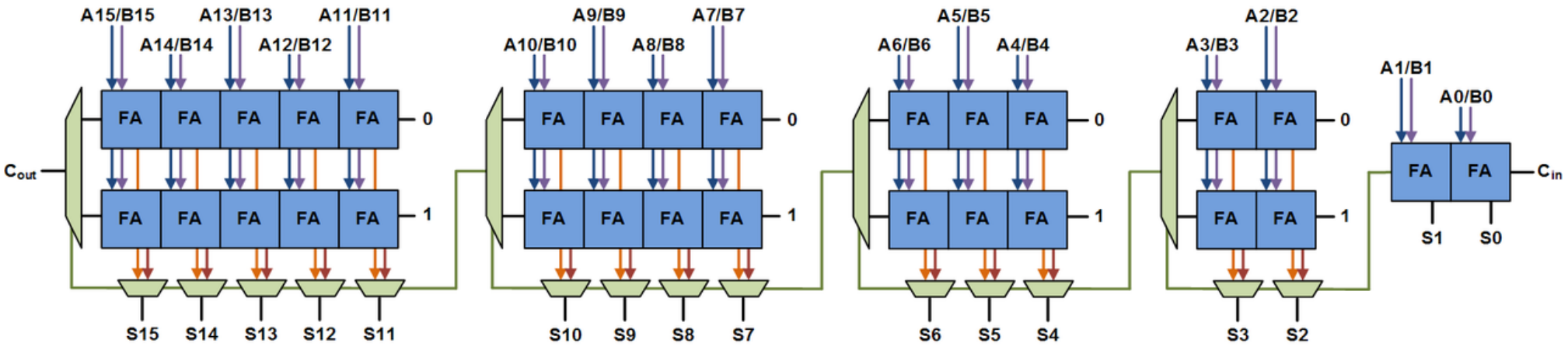
Carry Select Adder



- $T_{total} = (2*sqrt(N)-1)\ T_{FA}$
  - assuming $T_{FA} = T_{MUX}$
- For ripple adder $T_{total} = N\ T_{FA}$

- Is sqrt(N) really the optimum?
  - From right to left increase size of each block to better match delays
  - E.g.: 16-bit adder, use block sizes [5 4 3 2 2]
  - E.g.: 64-bit adder, use block sizes [13 12 11 10 9 8 7]

Carry Select Adder variable size blocks



16-bit adder Carry select adder with variable block sizes [5 4 3 2 2]

Carry Look-ahead Adders

- In general, for n-bit addition best we can achieve is

  delay O(log(n))

- How do we arrange this? (think trees)
- First, reformulate basic adder stage:

a b c_i	c_{i+1}	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

p

g

carry "propagate"

$p_i = a_i \oplus b_i$

Exactly one of the 2 inputs is 1, then propagate the carry you received from the previous stage

carry "generate"

$g_i = a_i\, b_i$

If both inputs are 1, then no matter what carry-in you received, generate a carry

$c_{i+1} = g_i + p_i c_i$

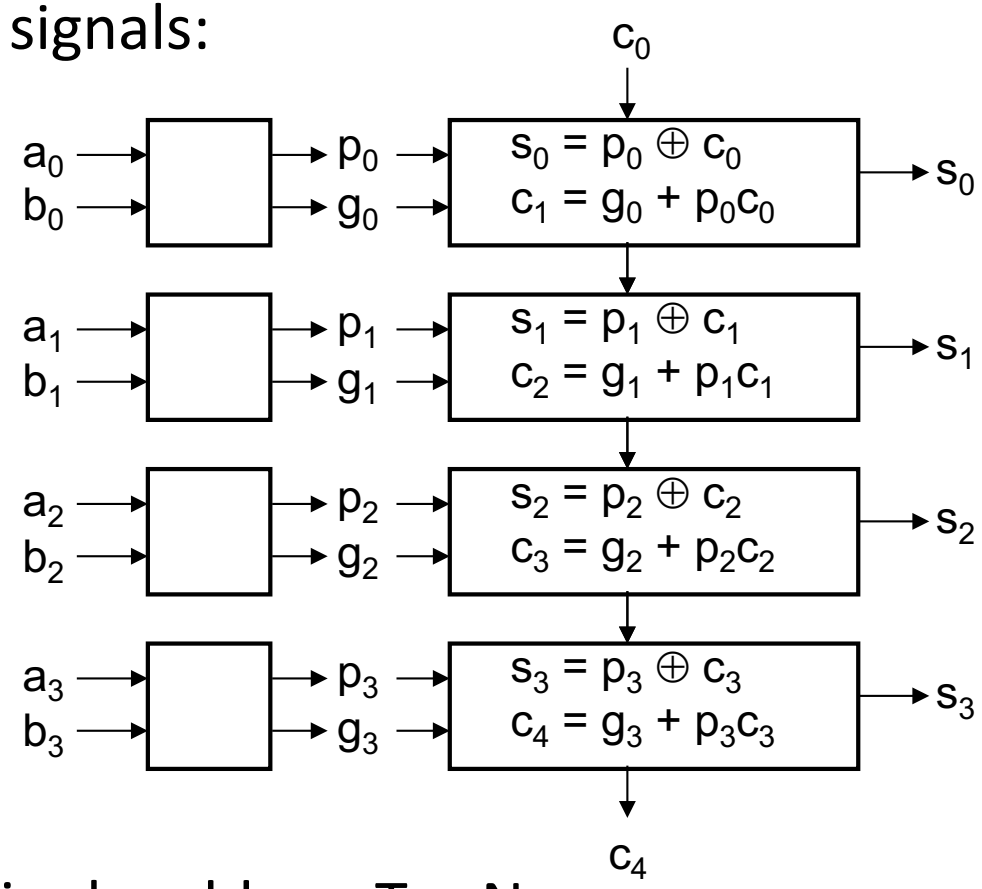
Carry-out is one if you generate a carry or you have a carry-in to propagate

$s_i = p_i \oplus c_i$

Result is 1 either when exactly 1 of the Inputs (a, b) is 1 (carry-propagate) (exclusive) or have a carry-in

Carry Look-ahead Adders

- Ripple adder using p and g signals:



- So far, no advantage over ripple adder: $T \alpha N$

Carry Look-ahead Adders

- Expand carries:

c_0

$c_1 = g_0 + p_0 c_0$

$c_2 = g_1 + p_1 c_1 = g_1 + p_1 g_0 + p_1 p_0 c_0$

$c_3 = g_2 + p_2 c_2 = g_2 + p_2 g_1 + p_1 p_2 g_0 + p_2 p_1 p_0 c_0$

$c_4 = g_3 + p_3 c_3 = g_3 + p_3 g_2 + p_3 p_2 g_1 + \ldots$

.

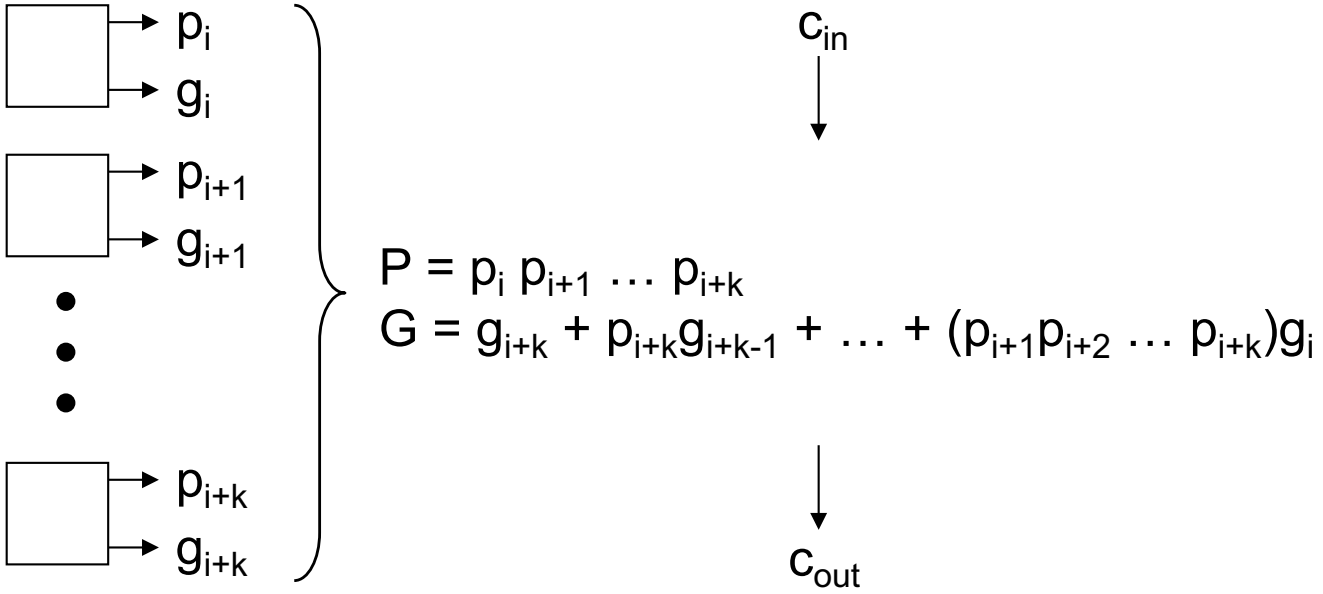
.

.

- Why not implement these equations directly to avoid ripple delay?
  - Lots of gates. Redundancies (full tree for each).
  - Gate with high # of inputs.
- Let's reorganize the equations.

Carry Look-ahead Adders

- "Group" propagate and generate signals:

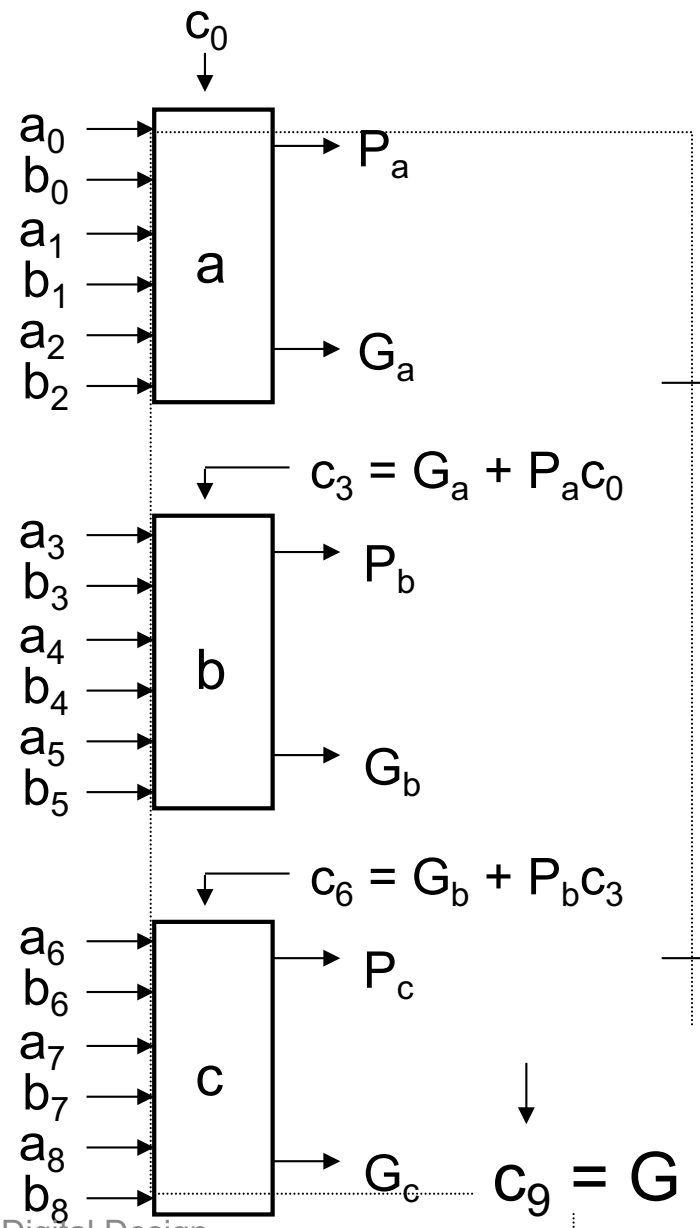


- P true if the group as a whole propagates a carry to c_{out}
- G true if the group as a whole generates a carry

$$C_{out} = G + PC_{in}$$

- Group P and G can be generated hierarchically.

Carry Look-ahead Adders

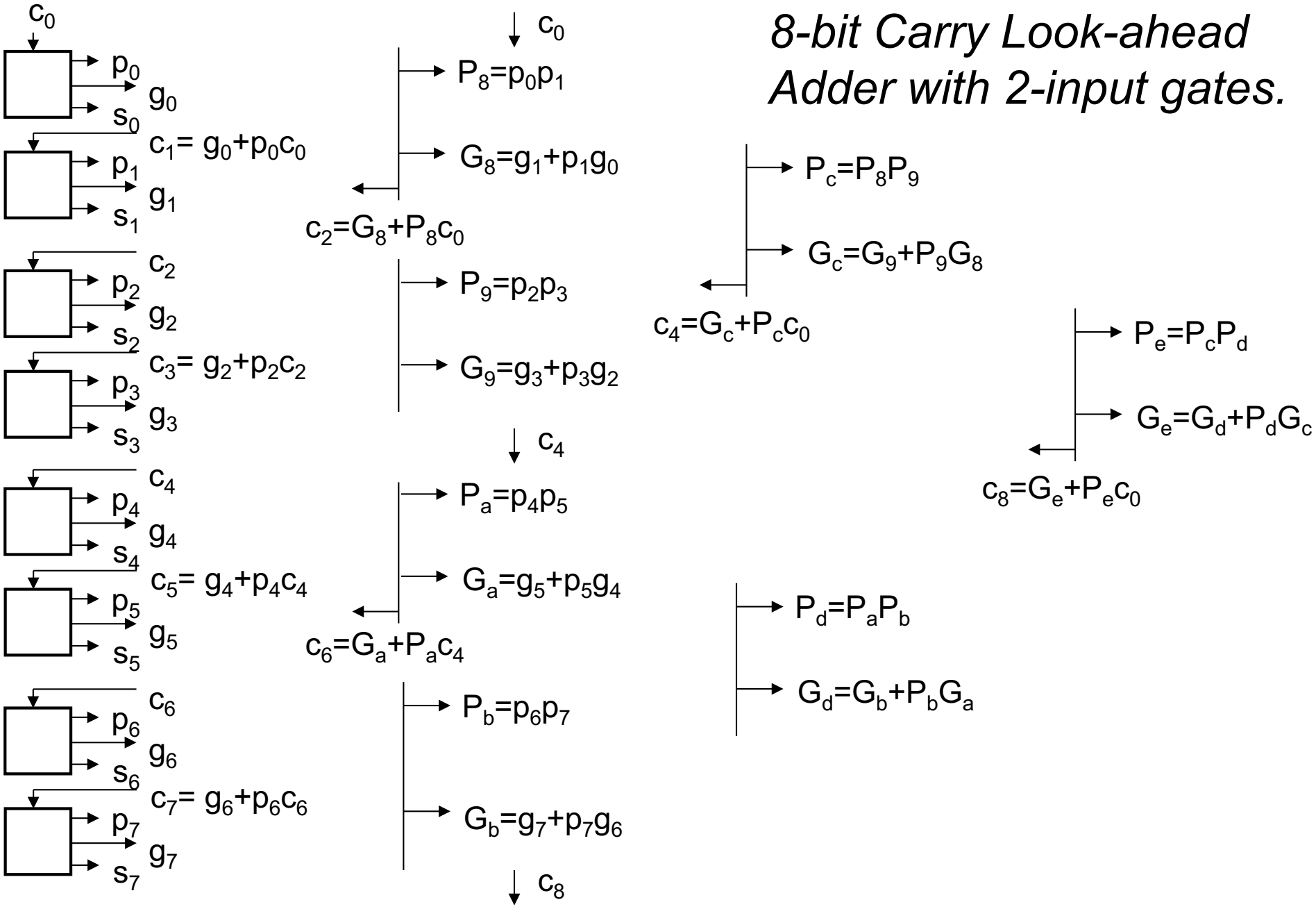


9-bit Example of hierarchically generated P and G signals:

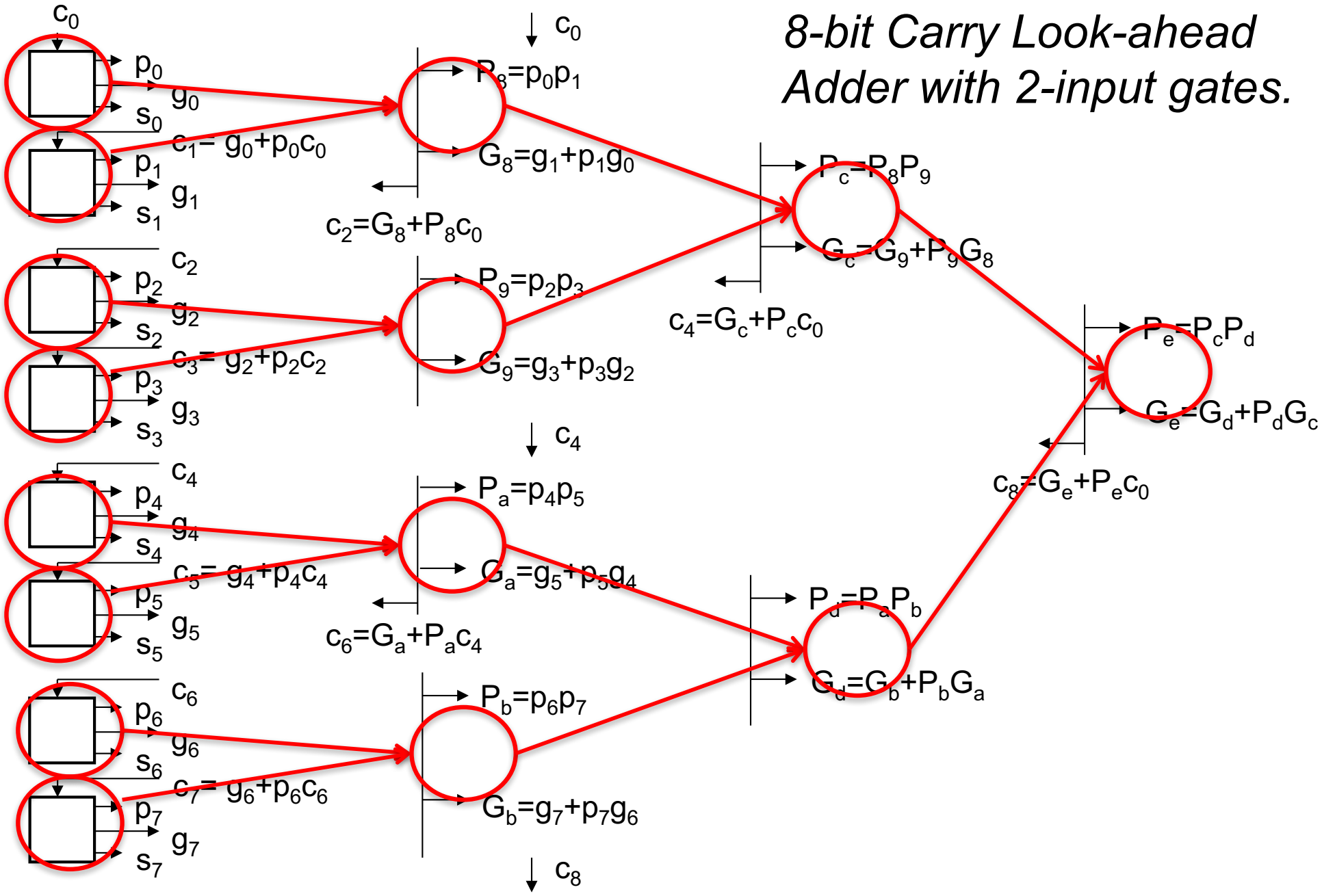
$$P = P_a P_b P_c$$

$$G = G_c + P_c G_b + P_b P_c G_a$$

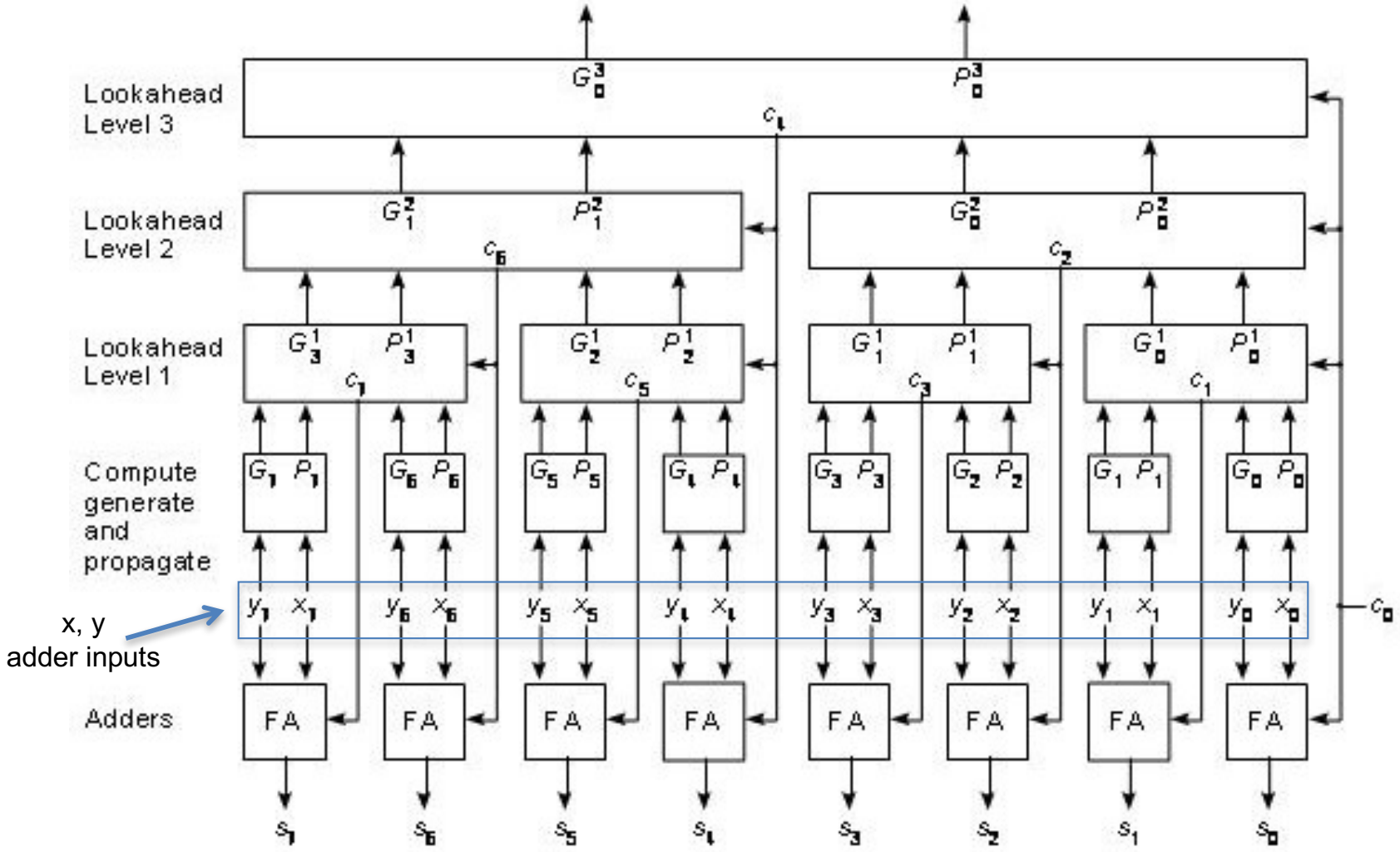
8-bit Carry Look-ahead Adder with 2-input gates.



8-bit Carry Look-ahead Adder with 2-input gates.



Carry Look-ahead adder



Quiz 2-2

http://m.socrative.com/student/#joinRoom

room number: **713113**

- Q1: a 8-bit ripple carry adder compared to a 4-bit ripple carry adder has:
  - 50% more area
  - 2 times more delay
- Q2: when adding 111 and 101 in a ripple carry adder, how many carries are generated?
- Q3: What is the delay and area of a 8-bit carry select adder of 2 blocks (each block 4-bits). Consider that:
  - FA delay=1, MUX delay= 1
  - FA area = 1 and MUX area = 1
- Q4: What is the delay of a carry-lookahead adder:
  - Linear, O(n)?
  - $O(\sqrt{\mathrm{N}})$?
  - *O(logN)*?

Summary of Lecture 2

- Boolean algebra basics
  - Axioms, theorems,
  - PoS, SoP, minterms, maxterms
- Logic minimization:
  - Karnaugh
    - Multiple variables
    - Multiple functions
- Adders
  - Ripple carry, Carry select, Carry lookahead

- Book sections (complimentary to the slide lectures):
  - 3.1, 3.2, 3.4, 3.5, 6.1-6.3, 6.5-6.9, 10.2, 10.3, 12.1
- Next Lecture 3 (Thursday):
  - Combinational logic (VHDL)