09 Logic Blocks

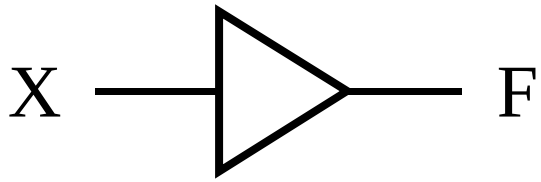
Arithmetic

©Department of Computer Science
University of Victoria

Buffer

- A buffer is a gate with the function F = X:

- As a Boolean function, a buffer is the same as a connection!
- Why use it?
  - A buffer is an electronic amplifier used to improve circuit voltage levels and increase the speed of circuit operation.



Hi-Impedance Outputs

- Logic gates introduced thus far
  - output values only 0 or 1
  - no outputs connected together
  - transmit signals in *only one* direction

- *Three-state logic* adds a third logic value
  Hi-Impedance (Hi-Z)

thus 3 outputs: 0, 1, and Hi-Z

Hi-Impedance Outputs (continued)

- What is a Hi-Z value?
  - The Hi-Z value behaves as an open circuit
    - I.e. the output appears to be disconnected
  - It is as if a switch between the internal circuitry and the output has been opened

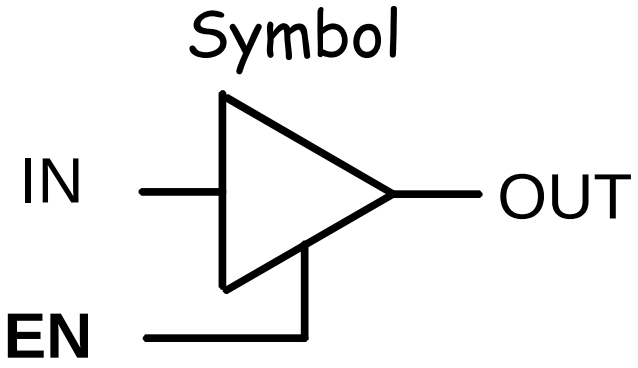
- Hi-Z may appear on the output of any gate, but we restrict gates to:
  - a 3-state buffer

  with one data input and one control input

The 3-State Buffer

- For the symbol and truth table, IN is the *data input*, and EN the *control input*
- For EN = 0,
  ➔ output = Hi-Z
- For EN = 1,
  ➔output value = input value

- Variations:
  - IN or EN can be inverted by addition of "bubbles" to signals



Truth Table

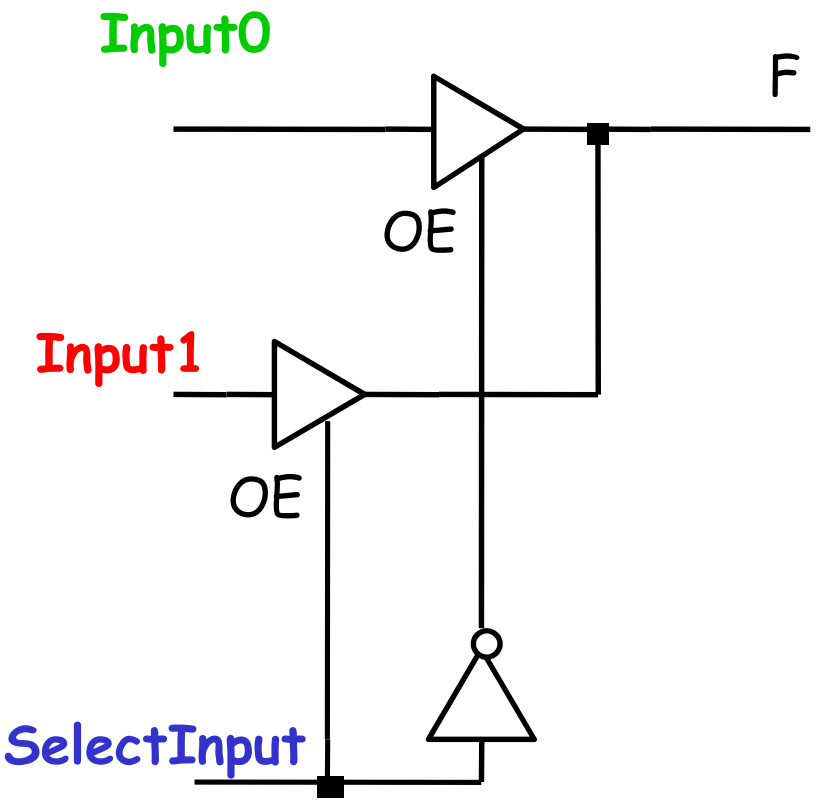
EN	IN	OUT
0	X	Hi-Z
1	0	0
1	1	1

Using tri-state gates to implement an economical multiplexer

When **SelectInput** is high
➔ F = Input1

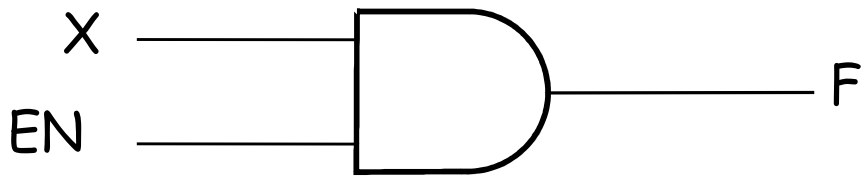
When **SelectInput** is low
➔ f = Input0

This is really a 2:1 Mux



Enabling Function

- *Enabling* permits an input signal to pass through to an output
- *Disabling* blocks an input signal from passing through to an output, replacing it with a fixed value
- The value on the output when it is disabled can be Hi-Z, 0 , or 1



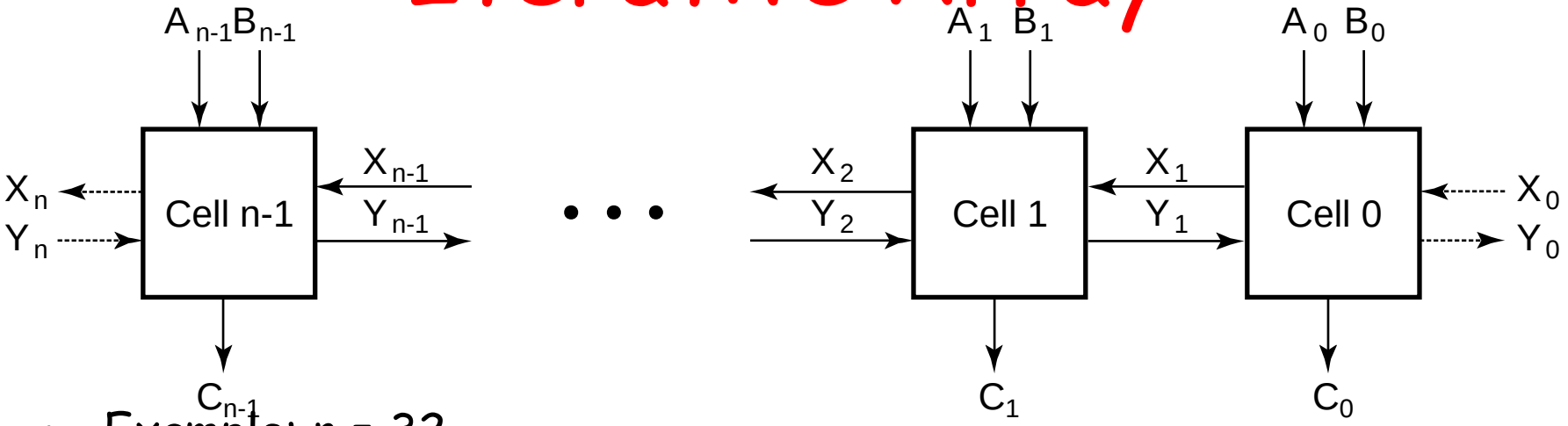
Arithmetic Overview

- Iterative combinational circuits
- Binary adders
  - Half and full adders
  - Ripple carry and carry lookahead adders
- Binary subtraction
- Binary adder-subtractors
  - Signed binary numbers
  - Signed binary addition and subtraction
  - Overflow
- Binary multiplication
- Other arithmetic functions
  - Design by contraction

Iterative Combinational Circuits

- Arithmetic functions
  - Operate on binary vectors
  - Use the same subfunction in each bit position
- Can design functional block for subfunction and repeat to obtain functional block for overall function
- *Cell* - subfunction block
- *Iterative array* - a array of interconnected cells
- An iterative array can be in a single dimension (1D) or multiple dimensions

Block Diagram of a 1D Iterative Array



- Example: n = 32
  - Number of inputs = ?
  - Truth table rows = ?
  - Equations with up to ? input variables
  - Equations with huge number of terms
  - Design impractical!
- Iterative array takes advantage of the regularity to make design feasible

Functional Blocks: Addition

- Binary addition used frequently
- Addition Development:
  - *Half-Adder* (HA), a 2-input bit-wise addition functional block,
  - *Full-Adder* (FA), a 3-input bit-wise addition functional block,
  - *Ripple Carry Adder*, an iterative array to perform binary addition, and
  - *Carry-Look-Ahead Adder* (CLA), a hierarchical structure to improve performance.

Functional Block: Half-Adder

- A 2-input, 1-bit width binary adder that performs the following computations:

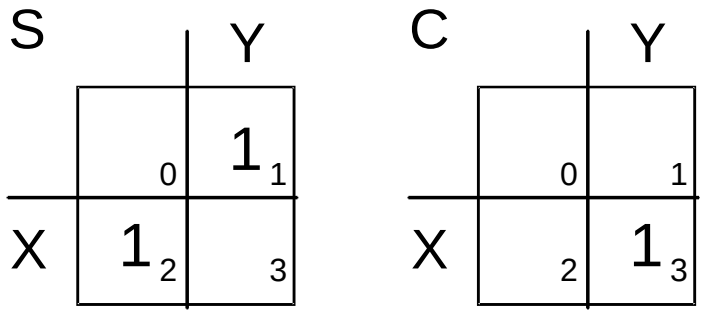
X	0	0	1	1
+ Y	+ 0	+ 1	+ 0	+ 1
C S	0 0	0 1	0 1	1 0

- A half adder adds two bits to produce a two-bit sum
- The sum is expressed as a sum bit , S and a carry bit, C
- The half adder can be specified as a truth table for S and C $\Rightarrow$

X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Logic Simplification: Half-Adder

- The K-Map for S, C is:
- This is a pretty trivial map! By inspection:



$$S = X \cdot \overline{Y} + \overline{X} \cdot Y = X \oplus Y$$

$$S = (X + Y) \cdot \overline{(X} + \overline{Y)}$$

- and

$$C = X \cdot Y$$

$$C = \overline{(\overline{(X \cdot Y)})}$$

- These equations lead to several implementations.

Five Implementations: Half-Adder

- We can derive following sets of equations for a half-adder:

(a) $S = X \cdot \overline{Y} + \overline{X} \cdot Y$
$C = X \cdot Y$

(b) $S = (X + Y) \cdot (\overline{X} + \overline{Y})$
$C = X \cdot Y$

(c) $S = \overline{(C + \overline{X} \cdot \overline{Y})}$
$C = X \cdot Y$

(d) $S = (X + Y) \cdot \overline{C}$
$\overline{C} = (\overline{X} + \overline{Y})$

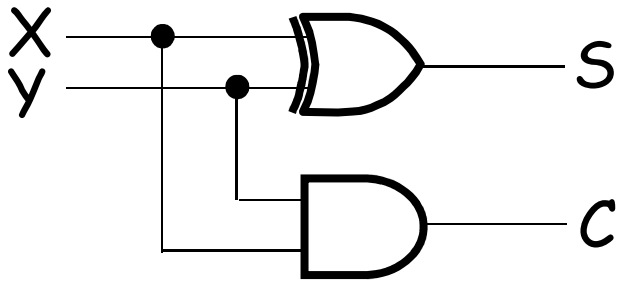
(e) $S = X \oplus Y$
$C = X \cdot Y$

- (a), (b), and (e) are SOP, POS, and XOR implementations for S.
- In (c), the C function is used as a term in the AND-NOR implementation of S, and in (d), the $\overline{C}$ function is used in a POS term for S.

Implementations: Half-Adder

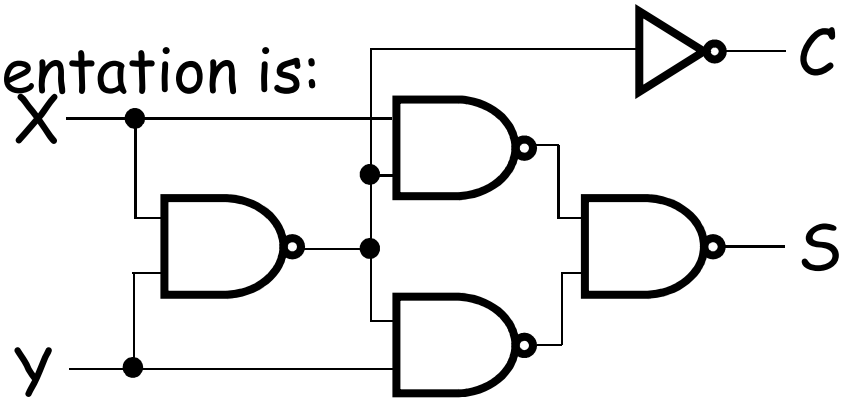
- The most common half adder implementation is: (e)

$$S = X \oplus Y$$
$$C = X \cdot Y$$



- A NAND only implementation is:

$$S = (\overline{X + Y}) \cdot C$$
$$C = (\overline{(X \cdot Y)})$$



Functional Block: Full-Adder

- A full adder is similar to a half adder, but includes a carry-in bit from lower stages. Like the half-adder, it computes a sum bit, S and a carry bit, C.
  - For a carry-in (Z) of 0, it is the same as the half-adder:

Z	0	0	0	0
X	0	0	1	1
+ Y	+ 0	+ 1	+ 0	+ 1
C S	00	0 1	0 1	1 0

  - For a carry- in (Z) of 1:

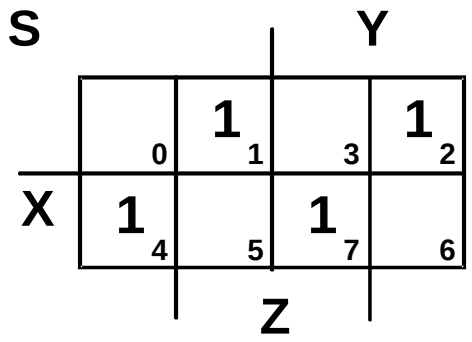
Z	1	1	1	1
X	0	0	1	1
+ Y	+ 0	+ 1	+ 0	+ 1
C S	0 1	1 0	1 0	1 1

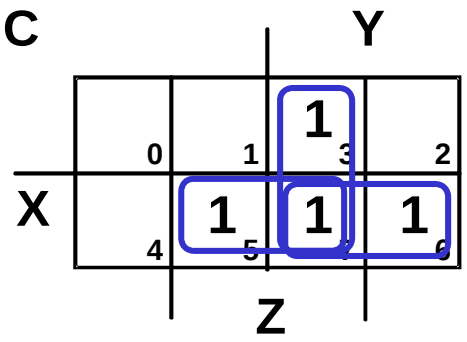
Logic Optimization: Full-Adder

- Full-Adder Truth Table:

X	Y	Z	C	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 K-Map:





Equations: Full-Adder

- From the K-Map, we get:

$$S = X\bar{Y}\bar{Z} + \bar{X}Y\bar{Z} + \bar{X}\bar{Y}Z + XYZ$$
$$C = XY + XZ + YZ$$

- The S function is the three-bit XOR function (Odd Function):

$$S = X \oplus Y \oplus Z$$

- The Carry bit C is 1 if both X and Y are 1 (the sum is 2), or if the sum is 1 and a carry-in (Z) occurs. Thus C can be re-written as:

$$C = XY + (X \oplus Y)Z$$

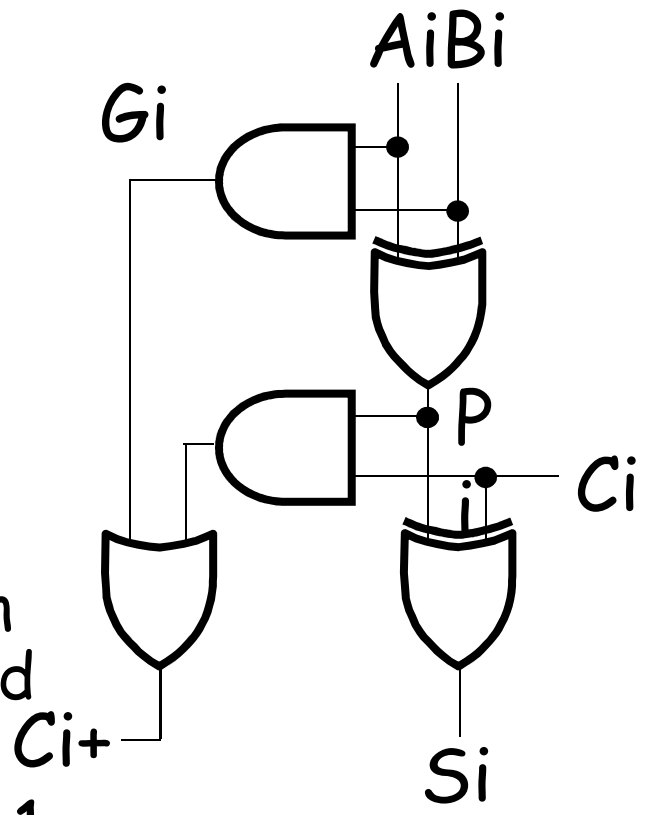
- The term X·Y is *carry generate*.
- The term X⊕Y is *carry propagate*.

Implementation: Full Adder

- Full Adder Schematic
- Here X, Y, and Z, and C (from the previous pages) are A, B, C_i and C_o, respectively. Also,

  G = generate and

  P = propagate.
- Note: This is really a combination of a 3-bit odd function (for S)) and Carry logic (for C_o):



(G = Generate) OR (P =Propagate AND C_i = Carry In)

$Co = G + P \cdot Ci$

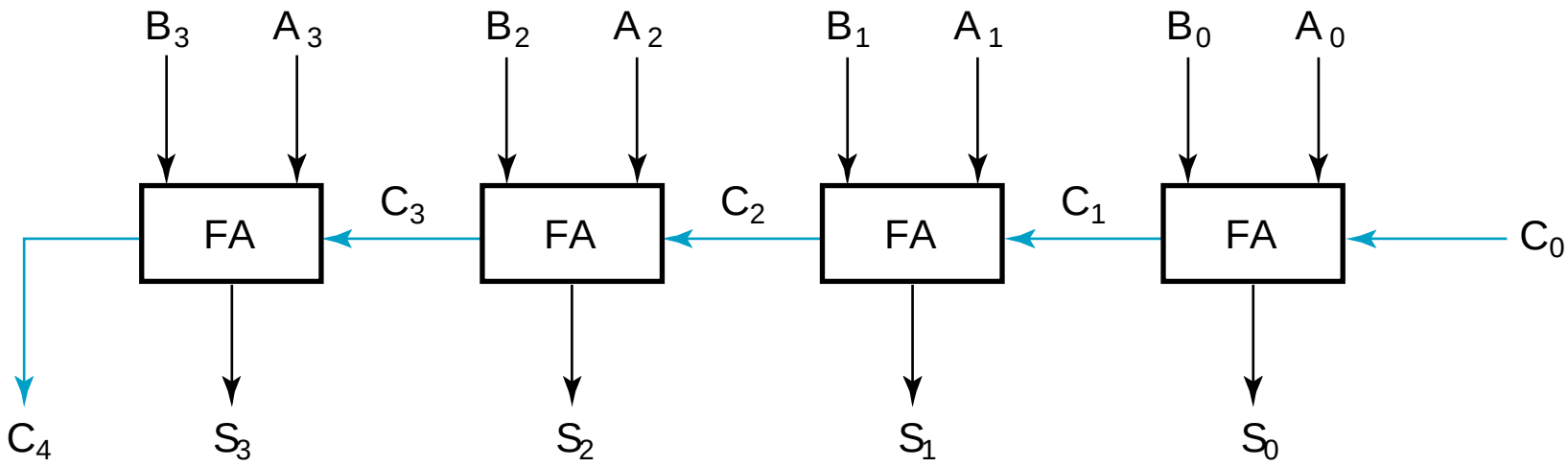
Binary Adders

- To add multiple operands, we "bundle" logical signals together into vectors and use functional blocks that operate on the vectors
- Example: 4-bit ripple carry adder: Adds input vectors A(3:0) and B(3:0) to get a sum vector S(3:0)
- Note: carry out of cell i becomes carry in of cell i + 1

Description	Subscript 3 2 1 0	Name
Carry In	0 1 1 0	Ci
Augend	1 0 1 1	Ai
Addend	0 0 1 1	Bi
Sum	1 1 1 0	Si
Carry out	0 0 1 1	Ci+1

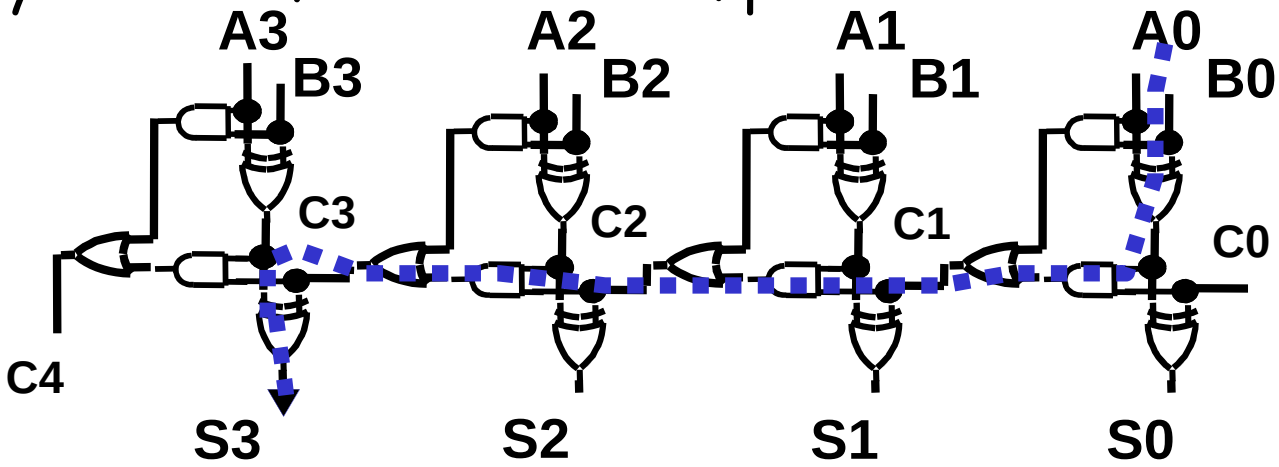
4-bit Ripple-Carry Binary Adder

- A four-bit Ripple Carry Adder made from four 1-bit Full Adders:



Carry Propagation & Delay

- One problem with the addition of binary numbers is the length of time to propagate the ripple carry from the least significant bit to the most significant bit.
- The gate-level propagation path for a 4-bit ripple carry adder of the last example:



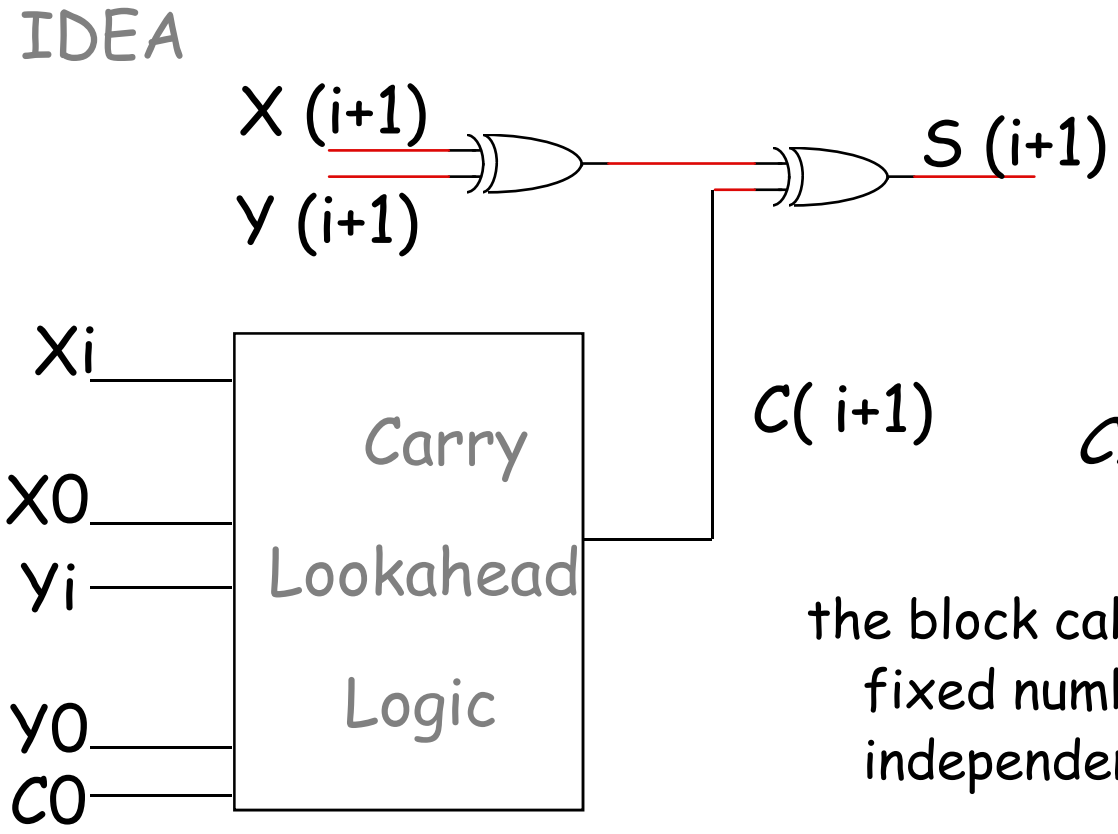
- Note: The "long path" is from A_0 or B_0 though the circuit to S_3.

Carry Lookahead Adders

- **optimize design by looking at equations for the Ci (carry ins) in terms of all the bit inputs**

  so C(i+1) = function(X0,.., Xi, Y0,..., Yi, C0)

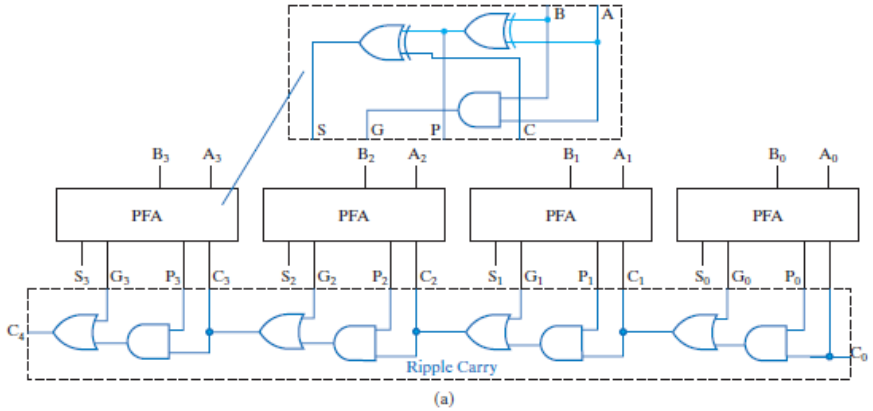
  rather than C(i+1) = XiYi + Ci (Xi + Yi)



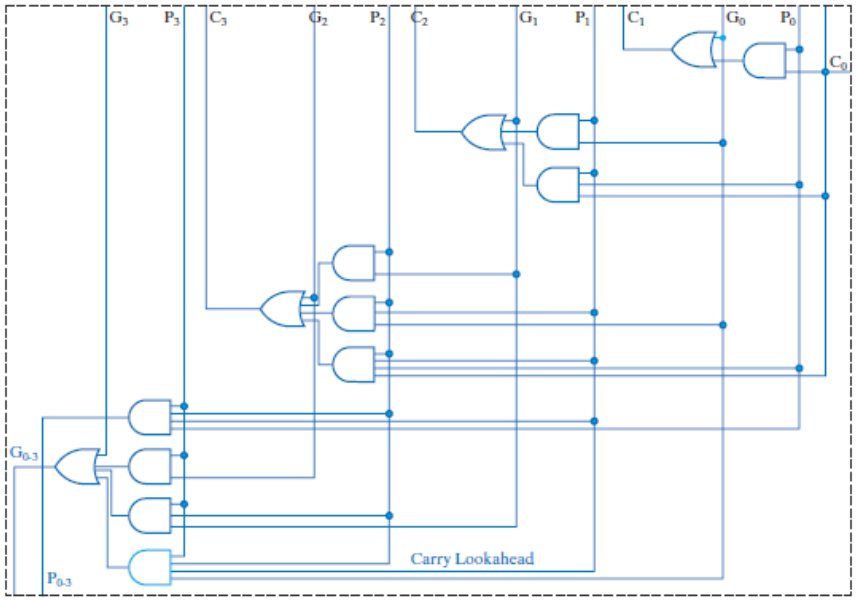
CAN WE DO IT ???

the block calculates C(i+1) in a fixed number of logic levels independent of n

Carry Look Ahead



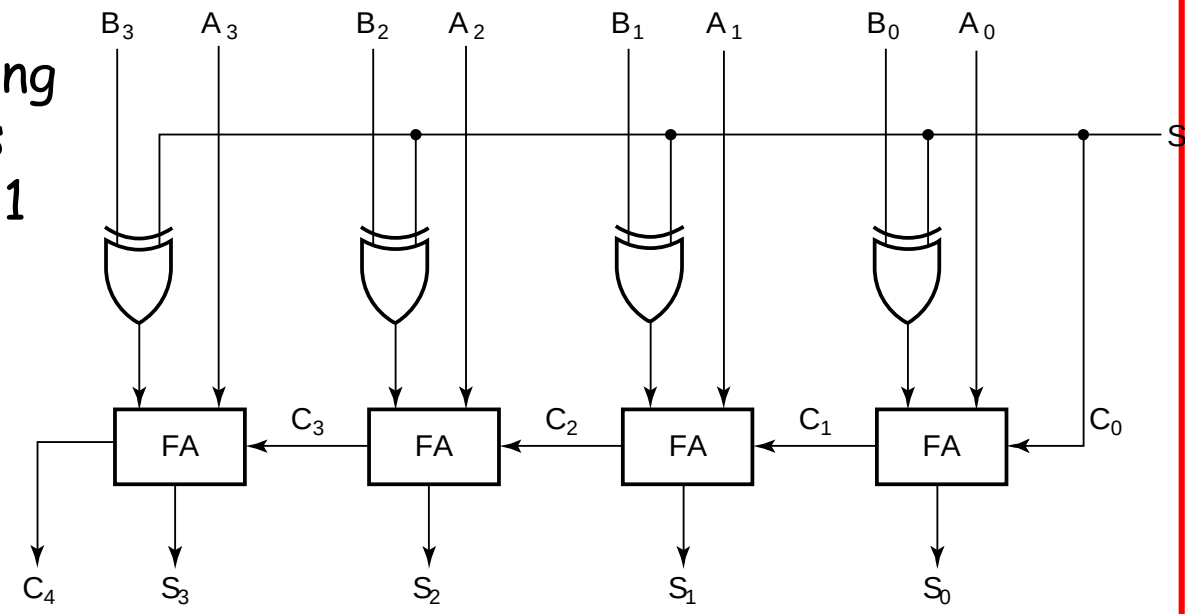
(a)



(b)

2's Complement Adder/Subtractor

- Subtraction can be done by addition of the 2's Complement.
  1. Complement each bit (1's Complement.)
  2. Add 1 to the result.
- The circuit shown computes A + B and A – B:
- For S = 1, subtract, the 2's complement of B is formed by using XORs to form the 1's comp and adding the 1 applied to C_0.
- For S = 0, add, B is passed through unchanged



Arithmetic Logic Unit

447-453

Sample ALU:

input words A, B and output word F

control lines M, S1, S0 and carry-in C0

M = 0, Logical Bitwise Operations

S1	S0	Function	Comment
0	0	Fi = Ai	Input Ai transferred to output
0	1	Fi = not Ai	Complement of Ai transferred to outp
1	0	Fi = Ai xor Bi	Compute XOR of Ai, Bi
1	1	Fi = Ai xnor Bi	Compute XNOR of Ai, Bi

M = 1, C0 = 0, Arithmetic Operations

0	0	F = A	Input A passed to output
0	1	F = not A	Complement of A passed to output
1	0	F = A plus B	Sum of A and B
1	1	F = (not A) plus B	Sum of B and complement of A

M = 1, C0 = 1, Arithmetic Operations

0	0	F = A plus 1	Increment A
0	1	F = (not A) plus 1	Twos complement of A
1	0	F = A plus B plus 1	Increment sum of A and B
1	1	F = (not A) plus B plus 1	B minus A