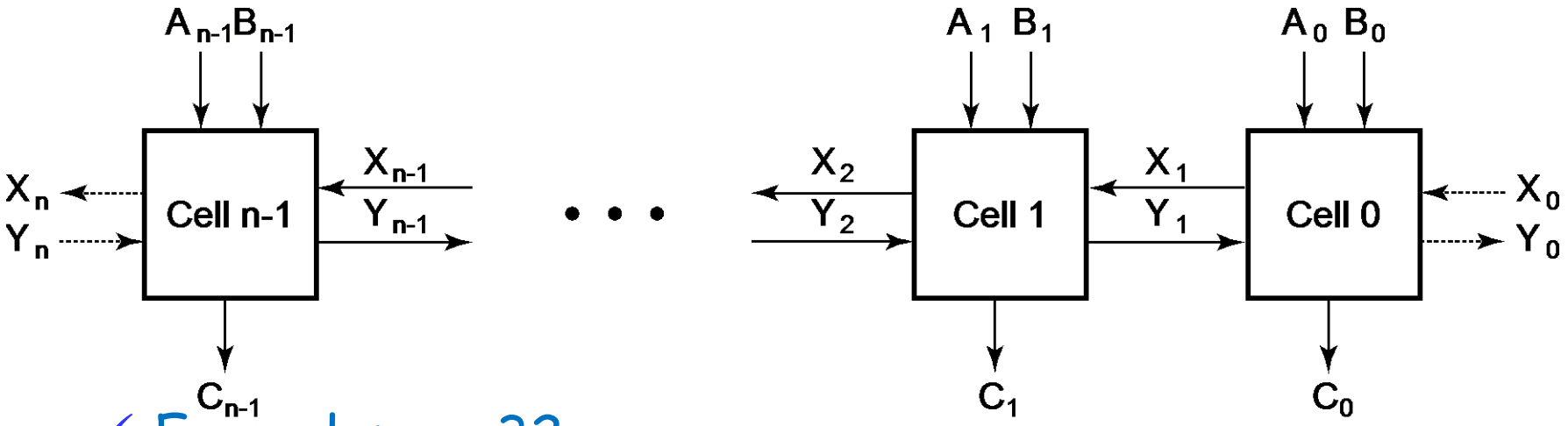
Iterative Combinational Circuits

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

Block Diagram of a 1D Iterative Array



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

Functional Blocks: Addition

- ✓ 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
  - **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:

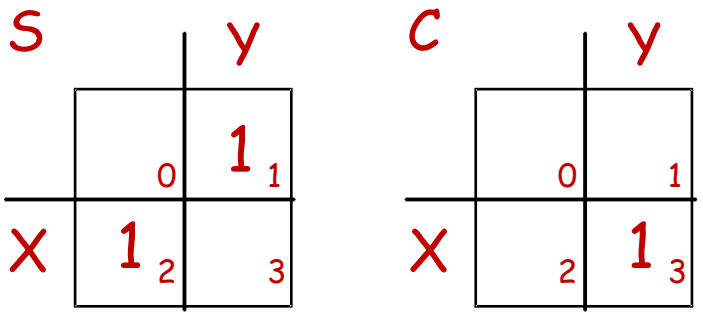
X	0	0	1	1
+ Y	+ 0	+ 1	+ 0	+ 1
C S	0 0	0 1	0 1	1 0

- ✓ 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 a truth table for S and C ⇒

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

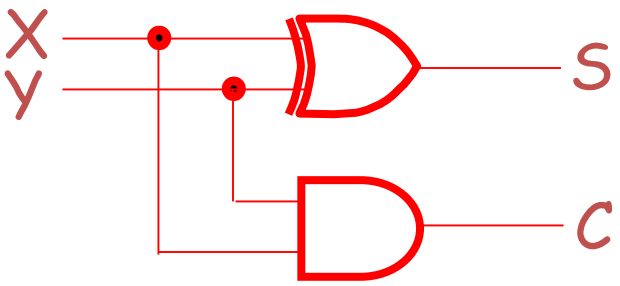
(e) $S = X \oplus Y$
$C = X \cdot Y$

- (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

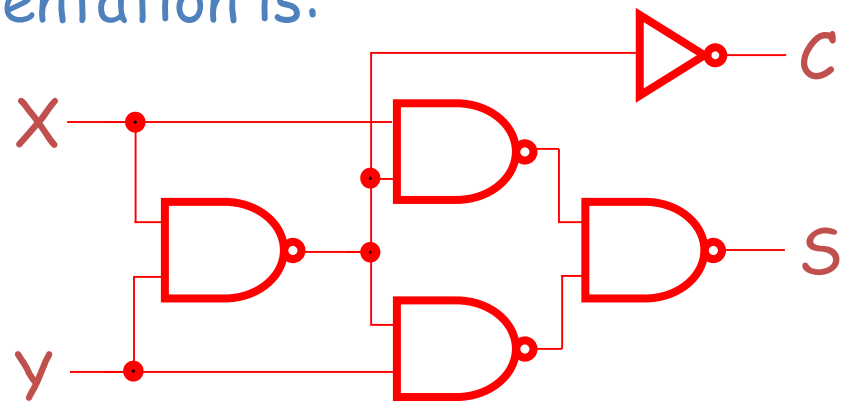
- ✓ The most common half adder implementation is: (e)

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



- ✓ A NAND only implementation is:

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



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	0 0	0 1	0 1	1 0

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

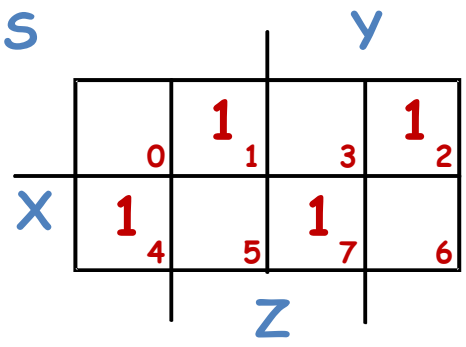
Z	1	1	1	1
X	0	0	1	1
+ Y	+ 0	+ 1	+ 0	+ 1
C S	0 1	1 0	1 0	1 1

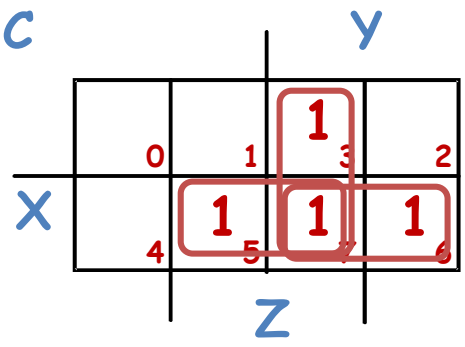
Logic Optimization: Full-Adder

- ✓ 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:

$$S = X\,\bar{Y}\,\bar{Z} + \bar{X}\,Y\,\bar{Z} + \bar{X}\,\bar{Y}\,Z + X\,Y\,Z$$

$$C = X\,Y + X\,Z + Y\,Z = X\,Y + (X + Y)\,Z$$

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

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

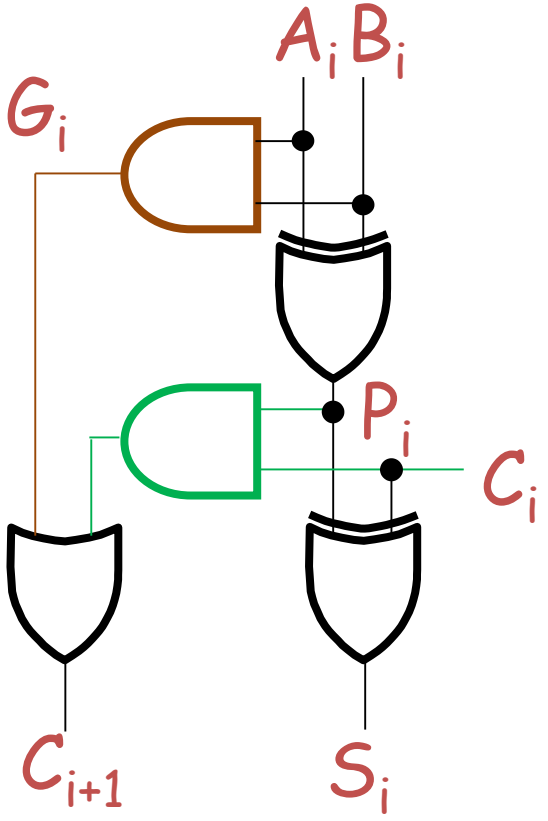
- The Carry bit C is 1 if both X and Y are 1 (the sum is 2), or if the sum is 1 and a carry-in (Z) occurs. Thus C can be re-written as:

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

- The term $X \cdot Y$ is *carry generate*.
- The term $X \oplus Y$ 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.



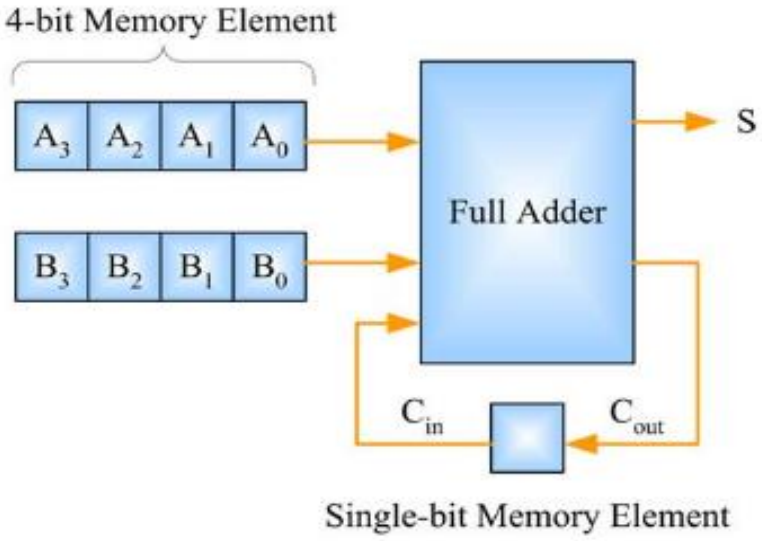
- ✓ Note: This is really a combination of a 3-bit odd function (for S) and Carry logic (for C_o):

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

$$C_{i+1} = G + P \cdot C_i$$

Sequential Adder

- ✓ 1-bit memory and 2 4-bit memory
- ✓ Only one full-adder!
- ✓ 4 clocks to get the output
- ✓ The 1-bit memory defines the circuit state (0 or 1)



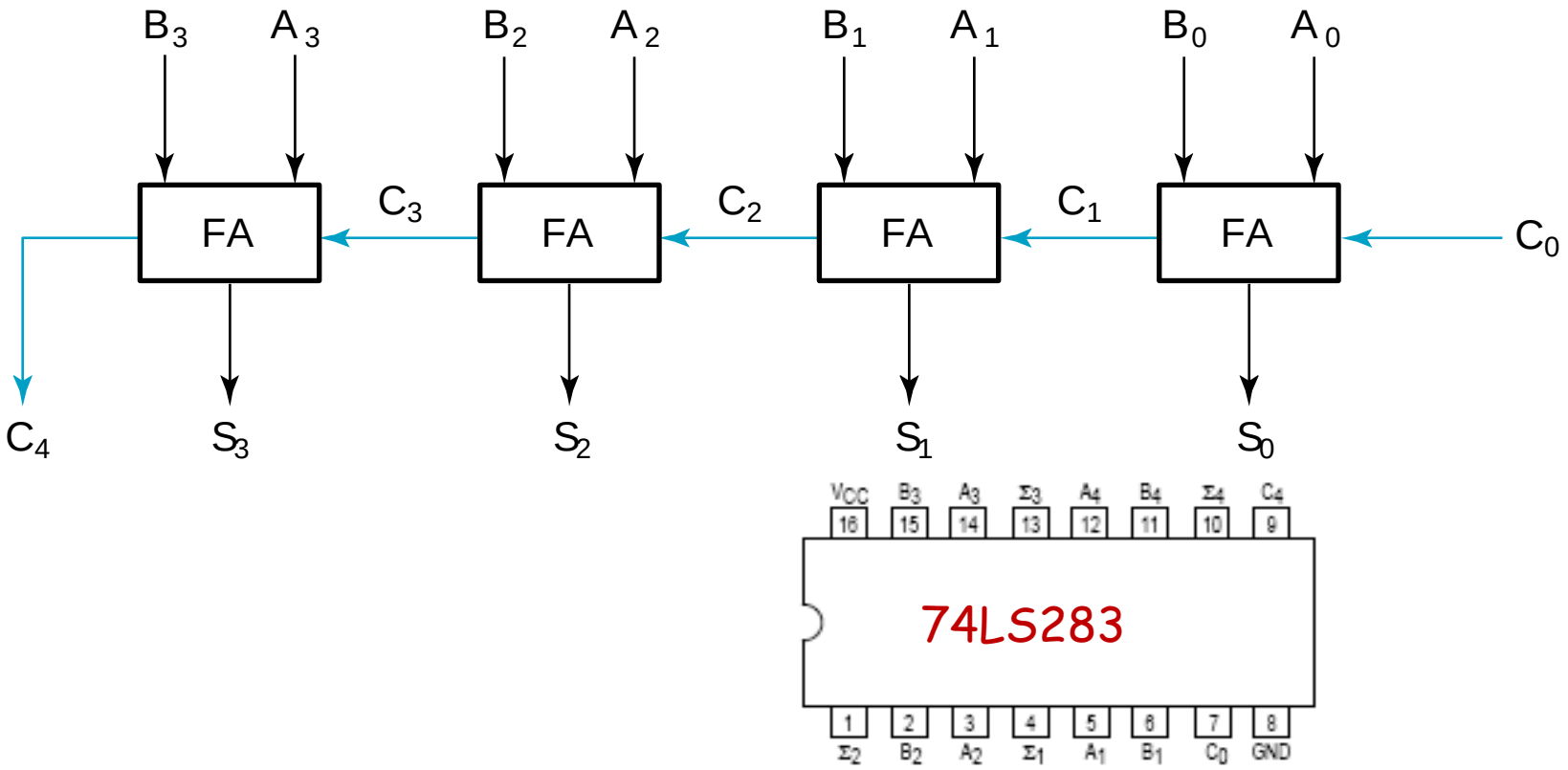
Binary Adders

- 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	C_i
Augend	1 0 1 1	A_i
Addend	0 0 1 1	B_i
Sum	1 1 1 0	S_i
Carry out	0 0 1 1	C_{i+1}

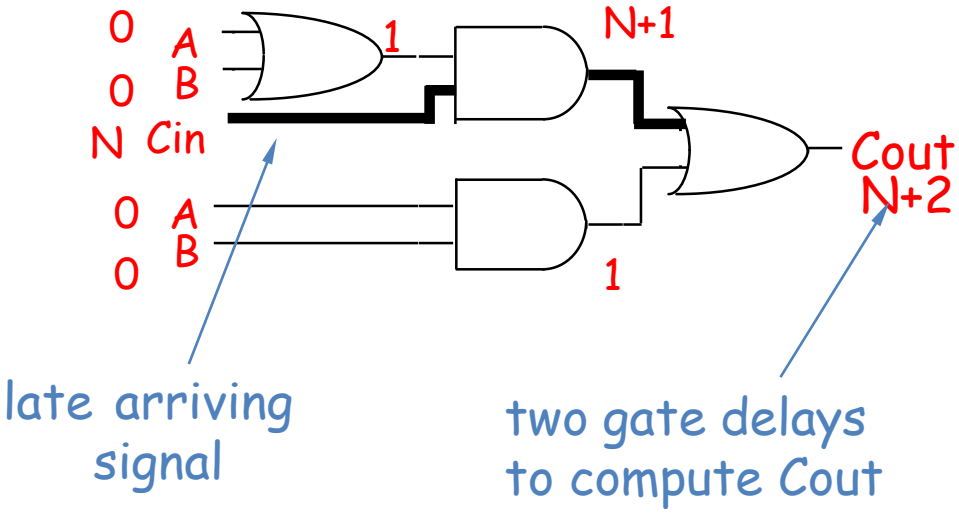
4-bit Ripple-Carry Binary Adder

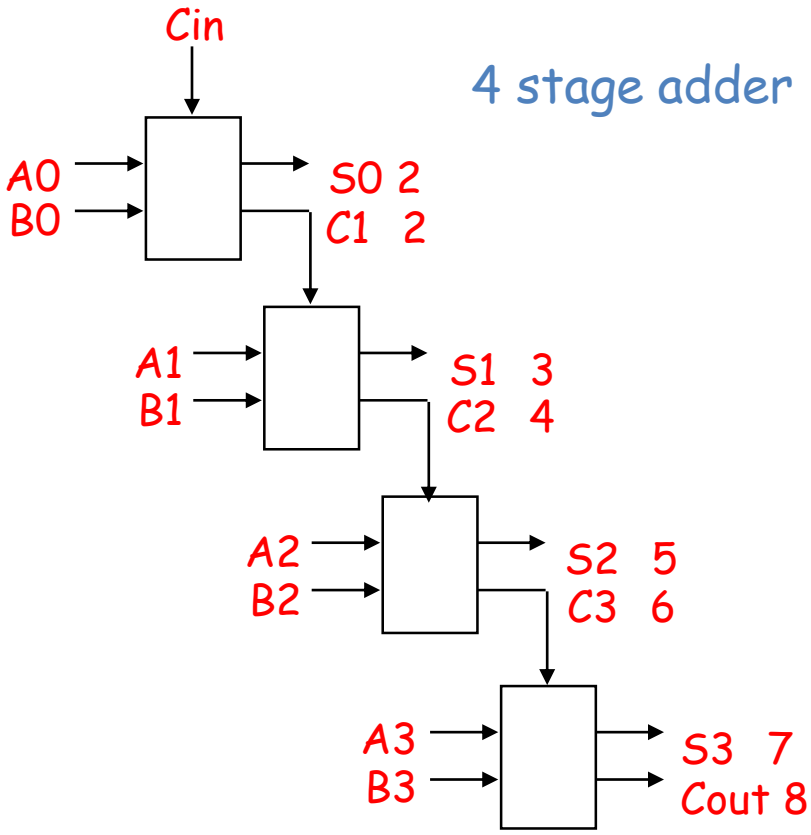
- ✓ A four-bit Ripple Carry Adder made from four 1-bit Full Adders:



Ripple-Carry Adders

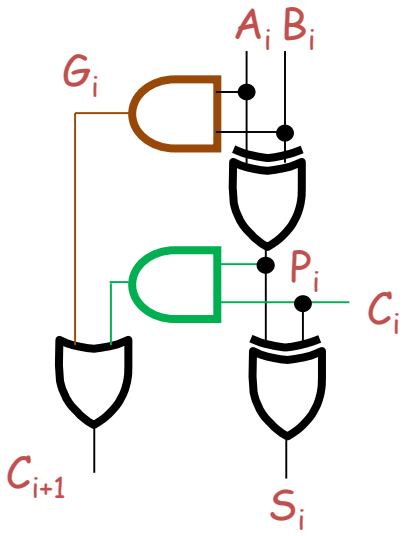
- ✓ Critical Delay
  - The propagation of carry from low to high order stages

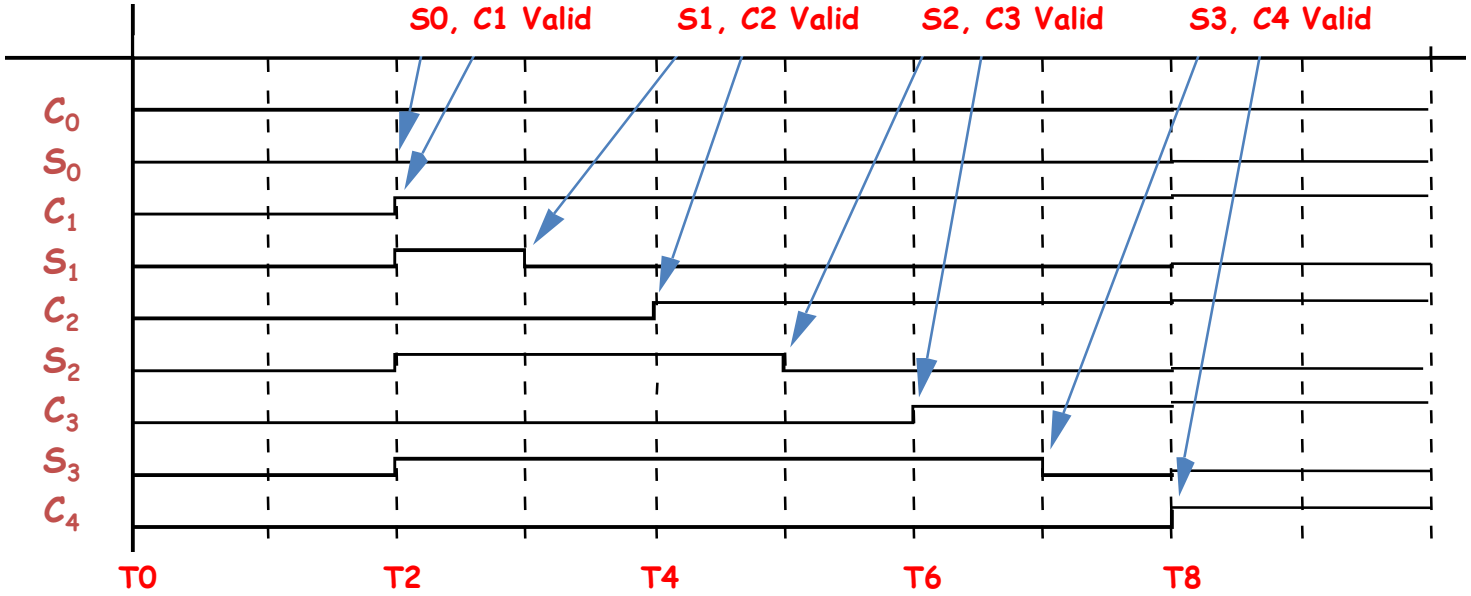




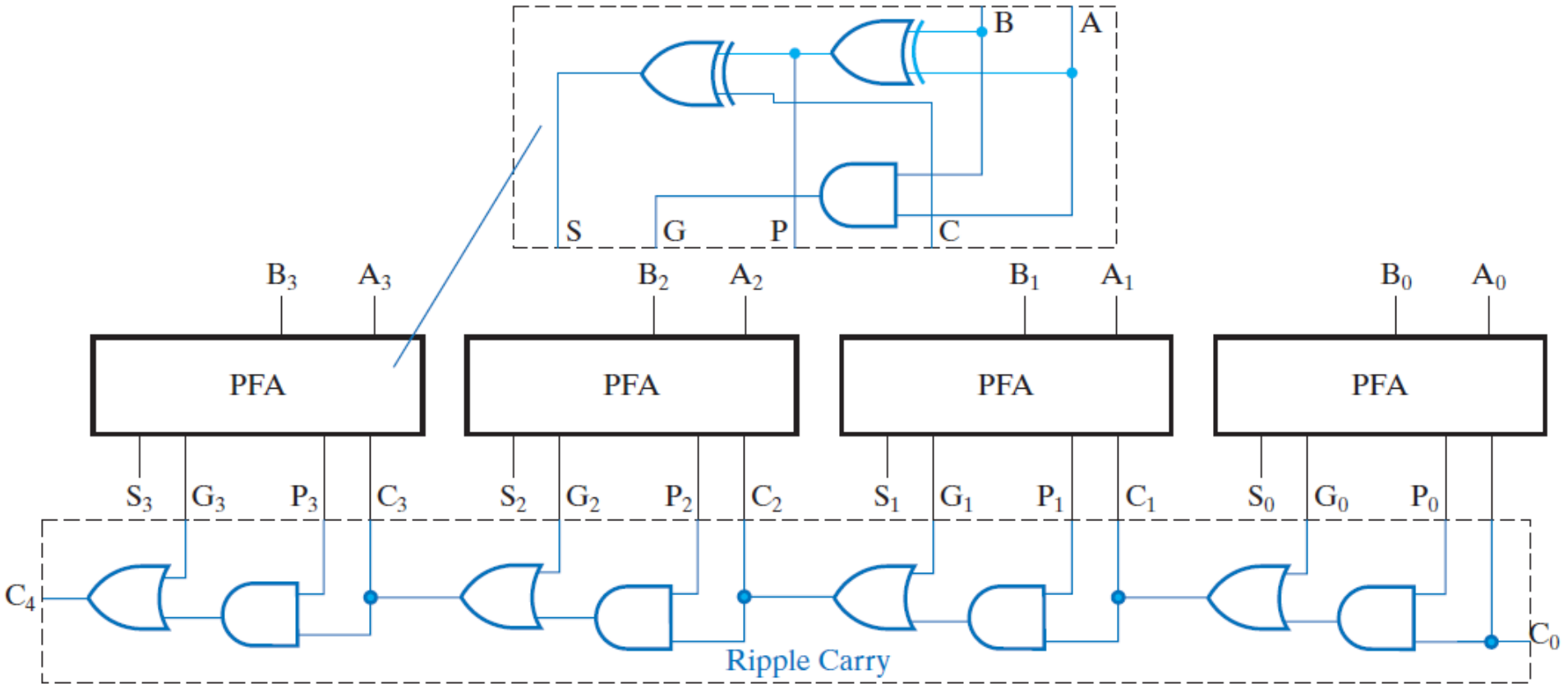
Ripple-Carry Adders (cont'd)

- ✓ Critical delay
  - The propagation of carry from low to high order stages
  - 1111 + 0001 is the worst case addition
  - Carry must propagate through all bits



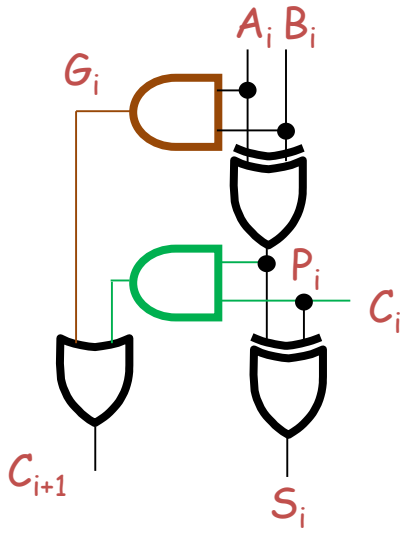


Ripple-Carry Adders (cont'd)



Carry-Lookahead Logic

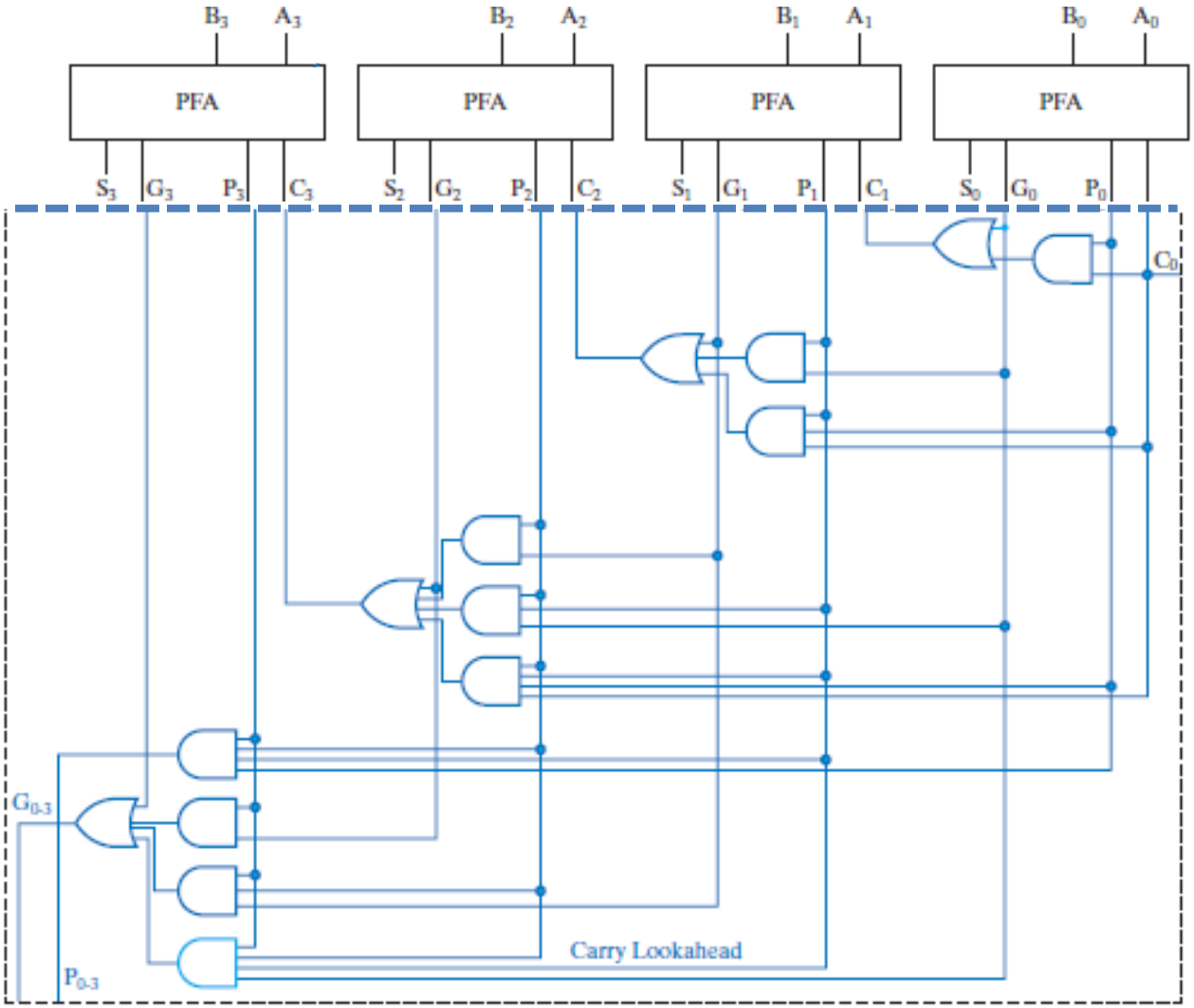
- ✓ Carry generate: Gi = Ai Bi
  - Must generate carry when A = B = 1
- ✓ Carry propagate: Pi = Ai xor Bi
  - Carry-in will equal carry-out here
- ✓ Sum and Cout can be re-expressed in terms of generate/propagate:
  - Si = Ai xor Bi xor Ci
    = Pi xor Ci
  - Ci+1 = Gi + Ci Pi



Carry-Lookahead Logic (cont'd)

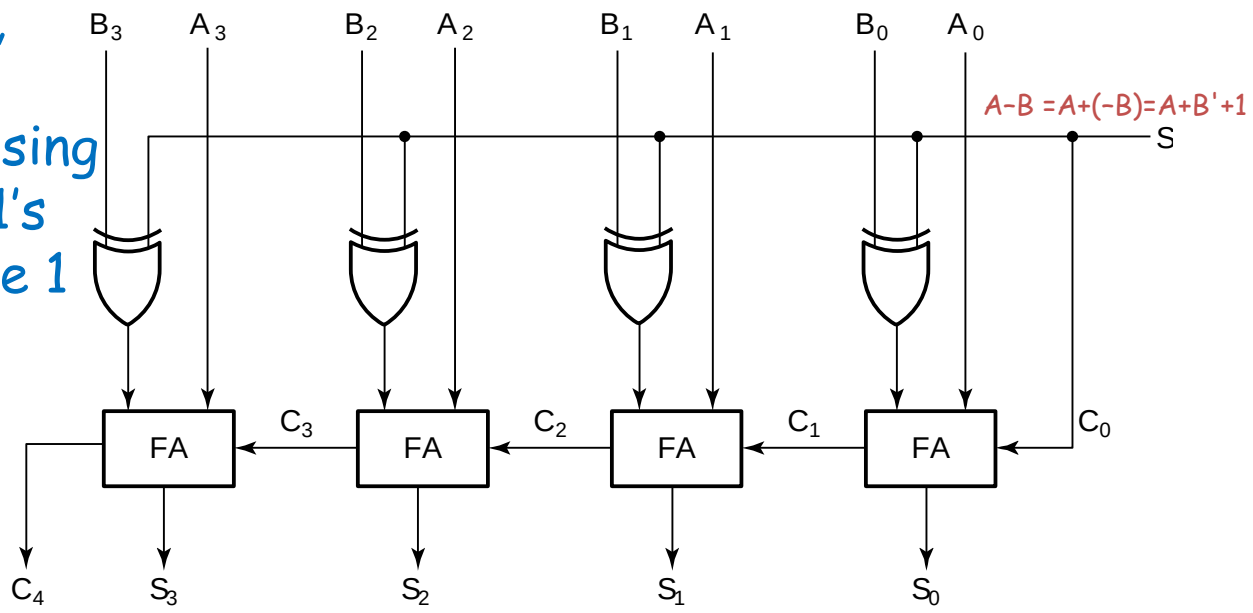
- ✓ Re-express the carry logic as follows:
  - $C_1 = G_0 + P_0C_0$
  - $C_2 = G_1 + P_1C1 = G_1 + P_1G_0 + P_1P_0C_0$
  - $C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$
  - $C_4 = G_3 + P_3C_3 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$
- ✓ Each of the carry equations can be implemented with two-level logic
  - All inputs are now directly derived from data inputs and not from intermediate carries
  - this allows computation of all sum outputs to proceed in parallel

Carry-Lookahead Implementation (cont'd)



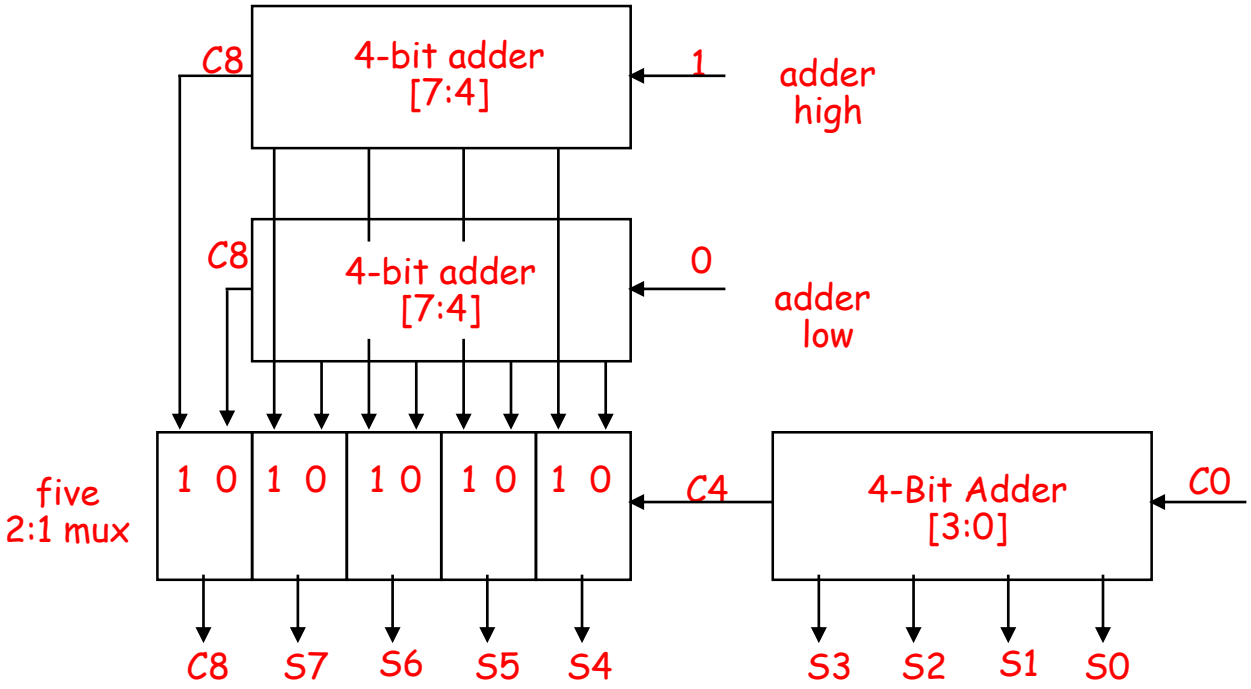
2's Complement Adder/Subtractor

- ✓ 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



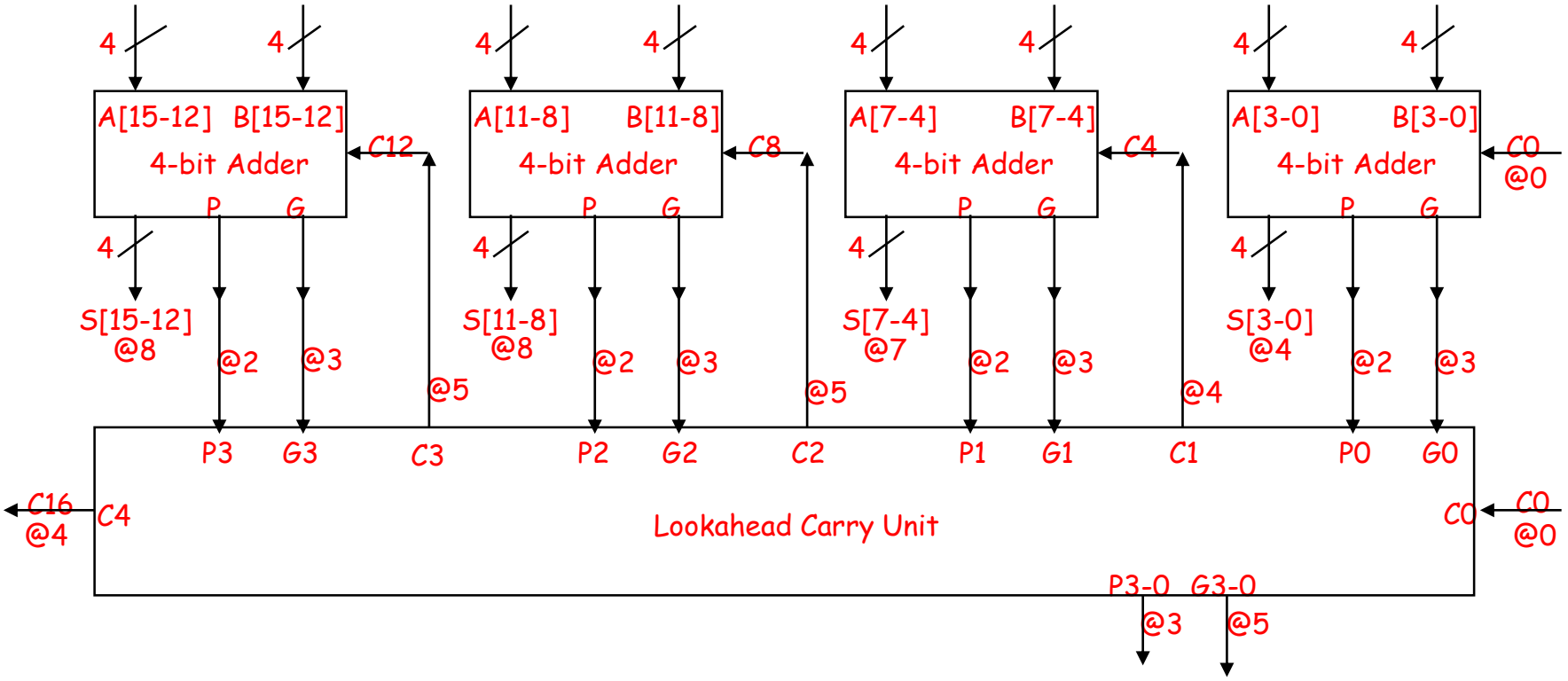
Carry-Select Adder

- ✓ Redundant hardware to make carry calculation go faster
  - Compute two high-order sums in parallel while waiting for carry-in
  - One assuming carry-in is 0 and another assuming carry-in is 1
  - Select correct result once carry-in is finally computed



Carry-Lookahead Adder with Cascaded Carry-Lookahead Logic

- ✓ Carry-lookahead adder
  - 4 four-bit adders with internal carry lookahead
  - Second level carry lookahead unit extends lookahead to 16 bits



Overflow Detection

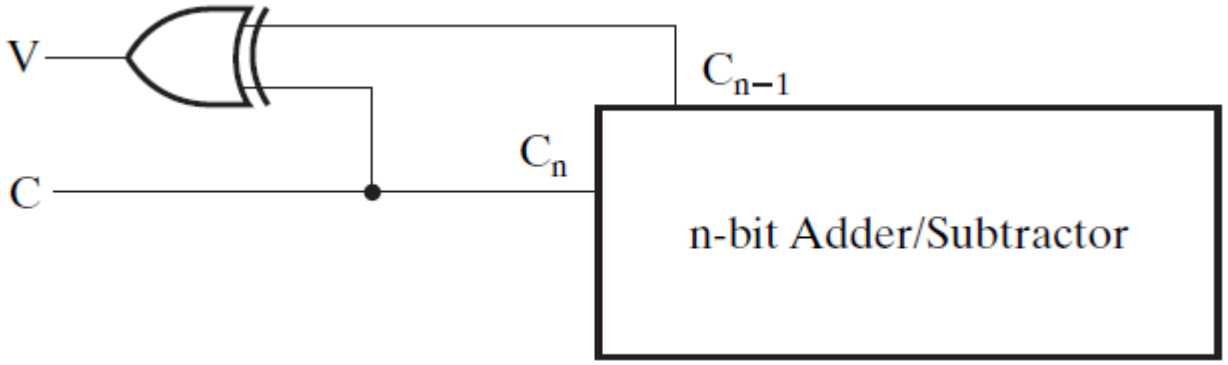
- ✓ Overflow occurs if $n + 1$ bits are required to contain the result from an n-bit addition or subtraction
- ✓ Overflow can occur for:
  - Addition of two operands with the same sign
  - Subtraction of operands with different signs
- ✓ Signed number overflow cases with correct result sign

$$\begin{array}{rrrr} 0 & 0 & 1 & 1 \\ +\ 0 & -1 & -0 & +1 \\ \hline 0 & 0 & 1 & 1 \end{array}$$

- ✓ Detection can be performed by examining the result signs which should match the signs of the top operand

Overflow Detection

- ✓ The simplest way to implement overflow is $V = C_n \oplus C_{n-1}$ where C_n and C_{n-1} are the carries in 2's complement respectively from e to the sign bit:
  - $C_n = 0$ and $C_{n-1} = 1$ means $A_n = B_n = 0$ and $S_n = 1$
  - $C_n = 1$ and $C_{n-1} = 0$ means $A_n = B_n = 1$ and $S_n = 0$



Other Arithmetic Functions

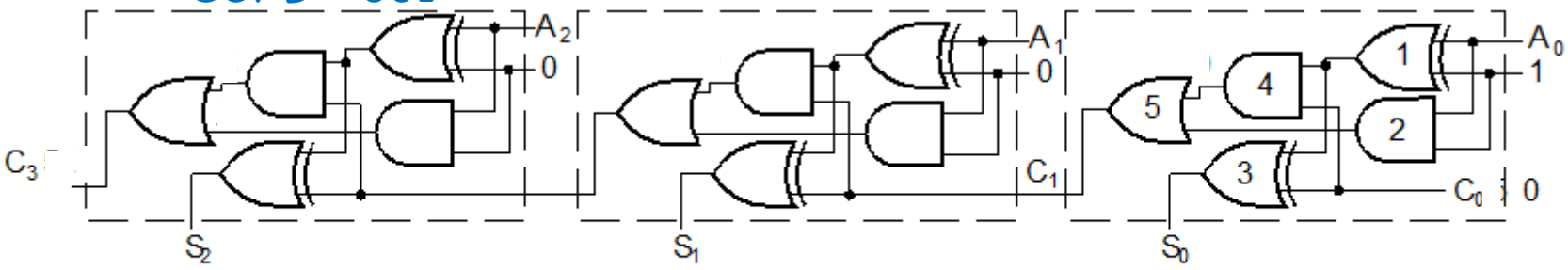
- ✓ Convenient to design the functional blocks by *contraction* - removal of redundancy from circuit to which input fixing has been applied
- ✓ Functions
  - Incrementing
  - Decrementing
  - Multiplication by Constant
  - Division by Constant
  - Zero Fill and Extension

Design by Contraction

- ✓ Contraction is a technique for simplifying the logic in a functional block to implement a different function
- ✓ The new function must be realizable from the original function by applying rudimentary functions to its inputs
  - Contraction is treated here only for application of 0s and 1s (not for X and $\overline{X}$)
  - After application of 0s and 1s, equations or the logic diagram are simplified.

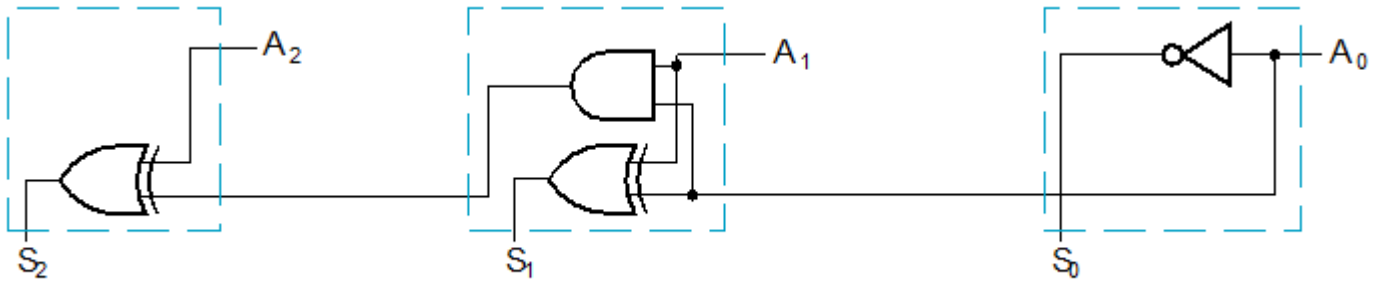
Design by Contraction Example: Incrementer

- ✓ Contraction of a ripple carry adder to incrementer for $n = 3$
  - Set B = 001



(a)

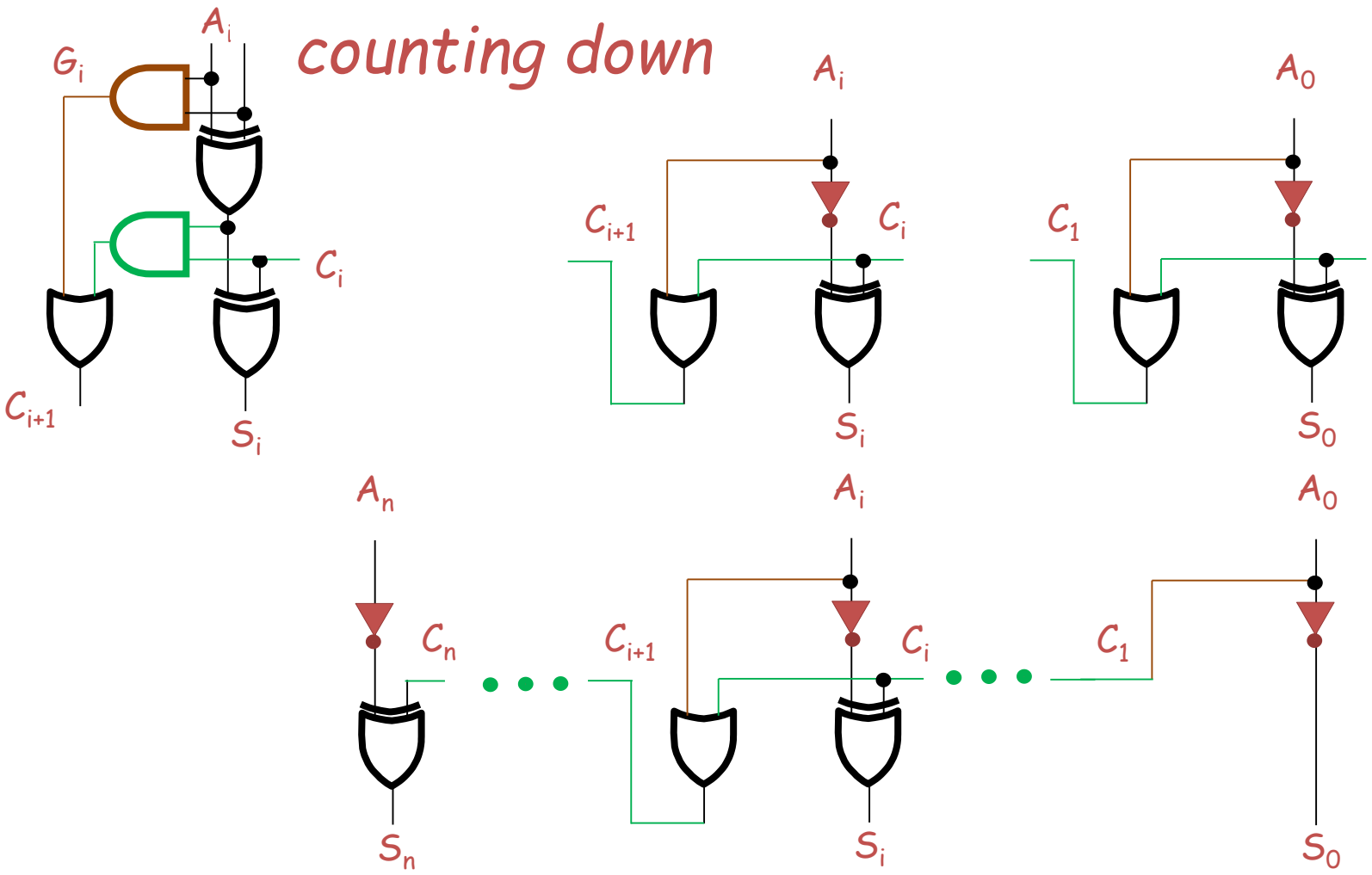
counting up



(b)

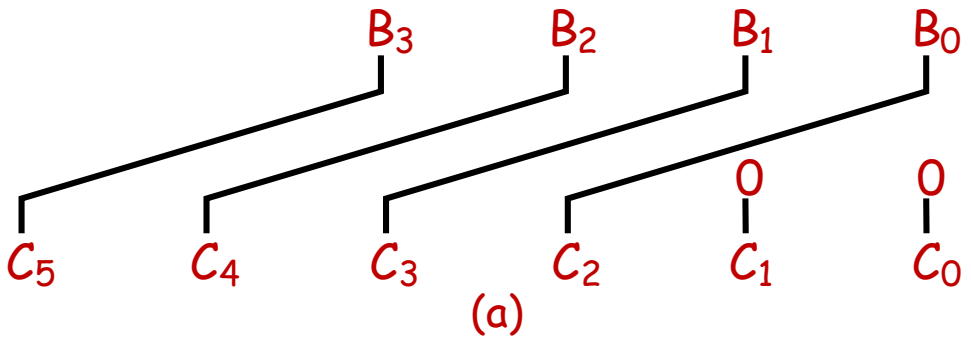
- The middle cell can be repeated to make an incrementer with $n > 3$.

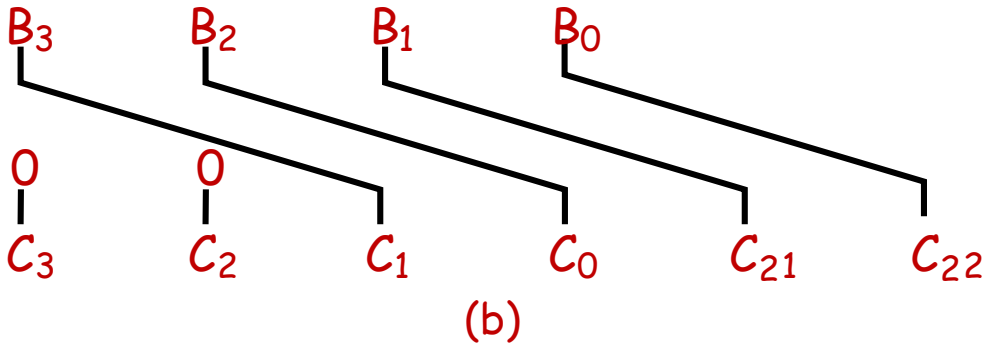
Decrementer $D=A-1_{10}$; $D=A+11\cdots111_2$



Multiplication/Division by 2^n

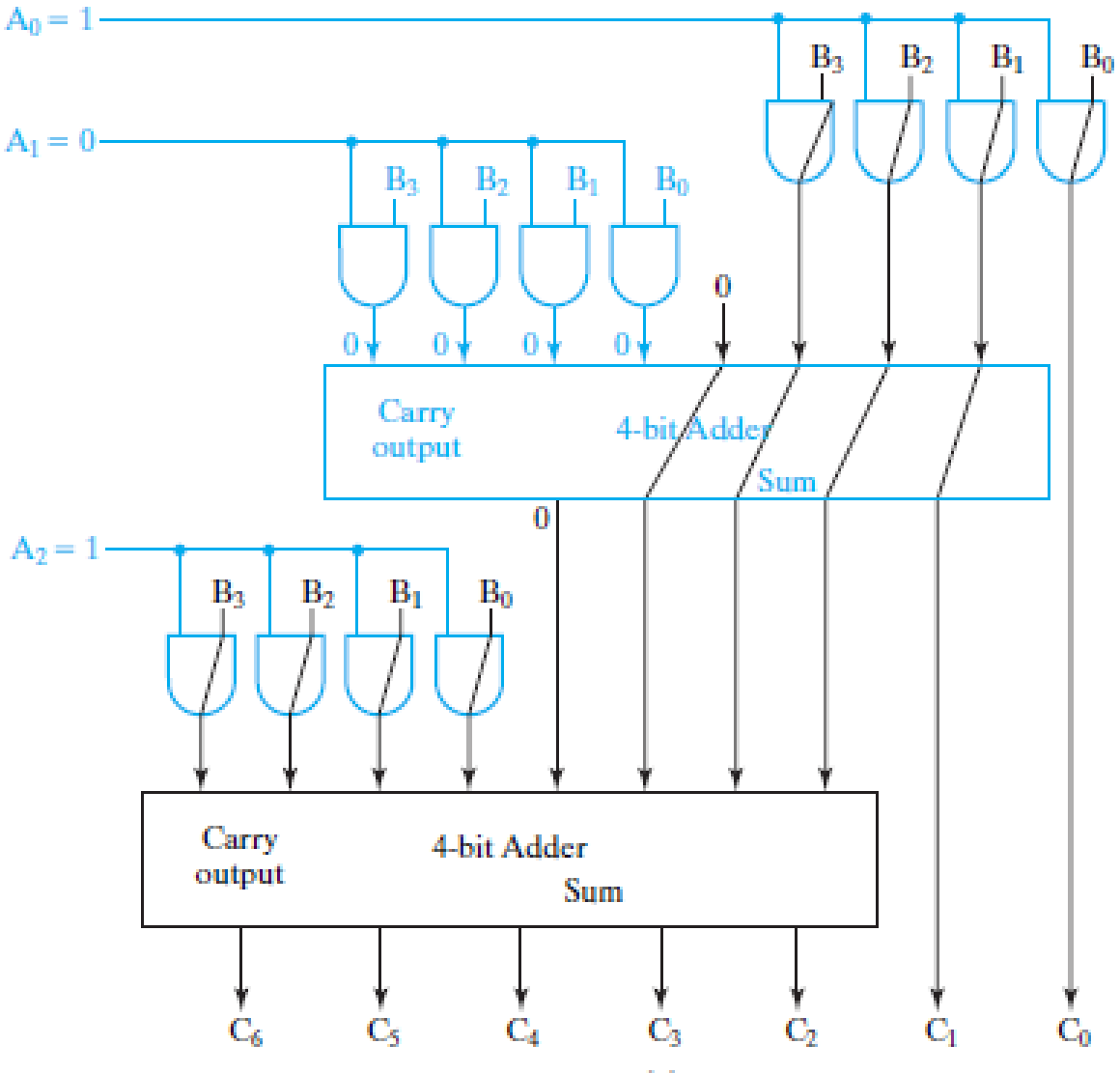
- ✓ (a) Multiplication by 100_2
  - Shift left by 2
- ✓ (b) Division by 100_2
  - Shift right by 2
  - Remainder preserved





Multiplication by a Constant

- ✓ Multiplication of B(3:0) by 101



Comparator

- ✓ Used to implement comparison operators (=, ≠)