Lecture 9 – Combinational logic circuits 2

Dr. Aftab M. Hussain,

Assistant Professor, PATRIoT Lab, CVEST

Chapter 4

Mid Sem Q1

Q1. In ancient times, people used parrots to make predictions. Let us consider a parrot that we are using to predict the result of a badminton match between player 1 and player 2. Once we know the outcome of the match, we make a logic function (F) that outputs “1” when the parrot was right and “0” when the parrot was wrong. Make a truth-table for this function [2]. Assuming all the possible outcomes in the truth-table are equally likely, what is the probability that our parrot predicted correctly [2]? Now, let us say we are unhappy with the probability of correct prediction, and want to increase our chances of success. For that, we now employ three parrots and create a function (G) that outputs “1” when one or more of them was right, and “0” only when all of them were wrong simultaneously. Make a truth-table for this function [4]. Make a K-map for this function [2]. How many prime implicants are there [2]? How many of them are essential [2]? Obtain the simplest possible algebraic expression for this and make the corresponding logic circuit [6]

- We go from logic problem -> function I/O -> truth-table -> K-map -> expression -> circuit

Mid Sem Q1

- We go from logic problem -> function I/O -> truth-table -> K-map -> expression -> circuit
- Output of the function is given as F/G. What are the inputs?
- Match result and parrot's prediction
- The truth-table will be:

R	P	F
P1	P1	1
P1	P2	0
P2	P1	0
P2	P2	1

- We can take P1 as 0 and P2 as 1 to make it resemble a normal truth-table

Mid Sem Q1

- For three parrots:
- “Correctness” of the parrot is a derived variable from the result of the match and the prediction of the parrot

R	P1	P2	P3	G
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Mid Sem Q1

- The K-map for function G
- No clusters of 16 or 8
- Clusters of 4?
- Total 12: 4 vertical/horizontal, 8 squares
- No essential prime implicants!
- The function can be written as:

$$G = wz + xy' + w'x' + yz'$$

Or

$$G = (w + x' + y' + z')(w' + x + y + z)$$

$wx \backslash yz$	00	01	11	10
00	m_0 1	m_1 1	m_3 1	m_2 1
01	m_4 1	m_5 1	m_7 0	m_6 1
11	m_{12} 1	m_{13} 1	m_{15} 1	m_{14} 1
10	m_8 0	m_9 1	m_{11} 1	m_{10} 1

Mid Sem Q2

Q2. Let us define a set B = {1, p, q, pq}, where p and q are distinct prime natural numbers. Let us define two binary operators on the elements of this set as Least Common Multiple denoted by (a # b), and Greatest Common Divisor or Greatest Common Factor denoted by ($a*b$). Are the operations # and * closed on the set B [8]? Do p and q need to be prime for the closure to work [2]? Do the operations # and * have an identity element, if so, what are they [10]?

- We can make a table with all the inputs and see what the outputs are
- p and q should be co-prime for the closure to work
- Identity in this case is 1 and pq for LCM and GCD
- In general case, *it is important to define which set you are working on – because operators are defined on sets*
- Identity for LCM = 1. Identity for GCD on Integer set is 0; on natural numbers is infinity/not defined

Mid Sem Q3

Q3. Is the base-19 number $(306050780)_{19}$ divisible by the decimal number $(360)_{10}$ [6]. If not, what is the remainder [2]?

- Assume r = 19
- Then, $(306050780)_{19} = 3r^8 + 6r^6 + 5r^4 + 7r^2 + 8r$
- We need to check whether this is divisible by $360 \ = r^2 - 1$
- We write: $(306050780)_{19} = 3r^8 - 3 + 6r^6 - 6 + 5r^4 - 5 + 7r^2 - 7 + 8r + 3 + 6 + 5 + 7$
- We know that $r^{2n} - 1$ has $r^2 - 1$ as a factor, the remaining is the remainder = $(173)_{10}$ = $(92)_{19}$

Mid Sem Q4

Q4. Perform the following conversions:

$$(93.25)_{10} = (?)_8$$

$$(101)_{10} = (?)_2$$

$$(CAD.004)_{16} = (?)_8$$

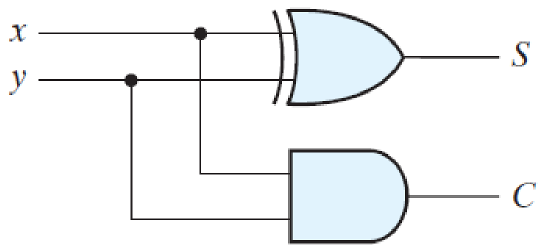
$$(93.25)_{10} = (135.2)_8$$

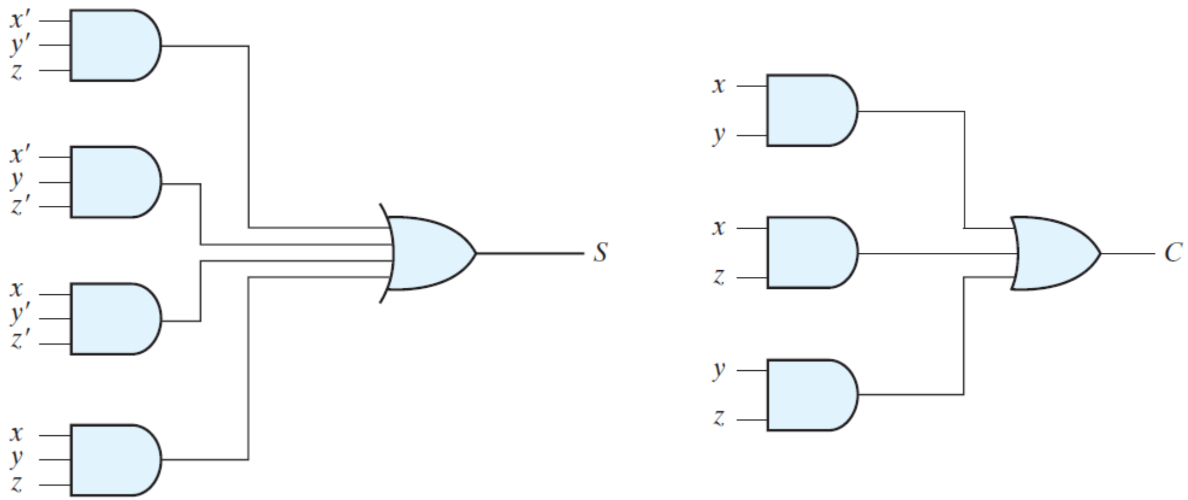
$$(101)_{10} = (1100101)_2$$

$$(CAD.004)_{16} = (6255.0004)_8$$

Binary adder

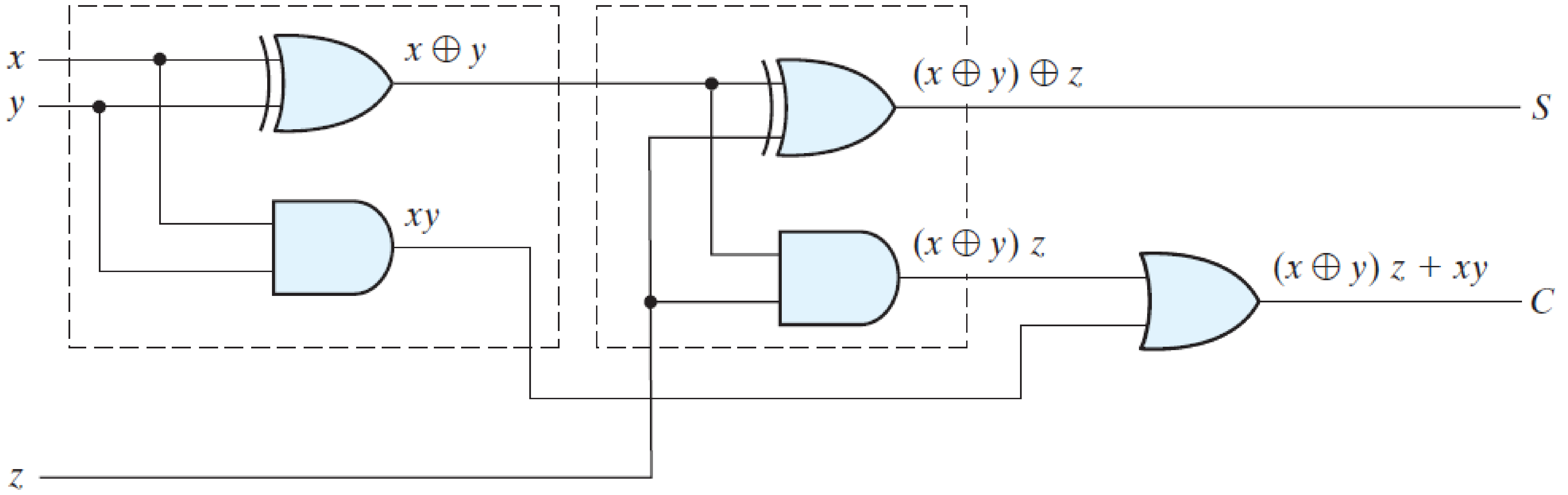
- Digital computers perform a variety of information-processing tasks
- Among the functions encountered are the various arithmetic operations
- The most basic arithmetic operation is the addition of two binary digits
- A combinational circuit that performs the addition of two bits is called a *half adder*.
- One that performs the addition of three bits (two significant bits and a previous carry) is a *full adder*





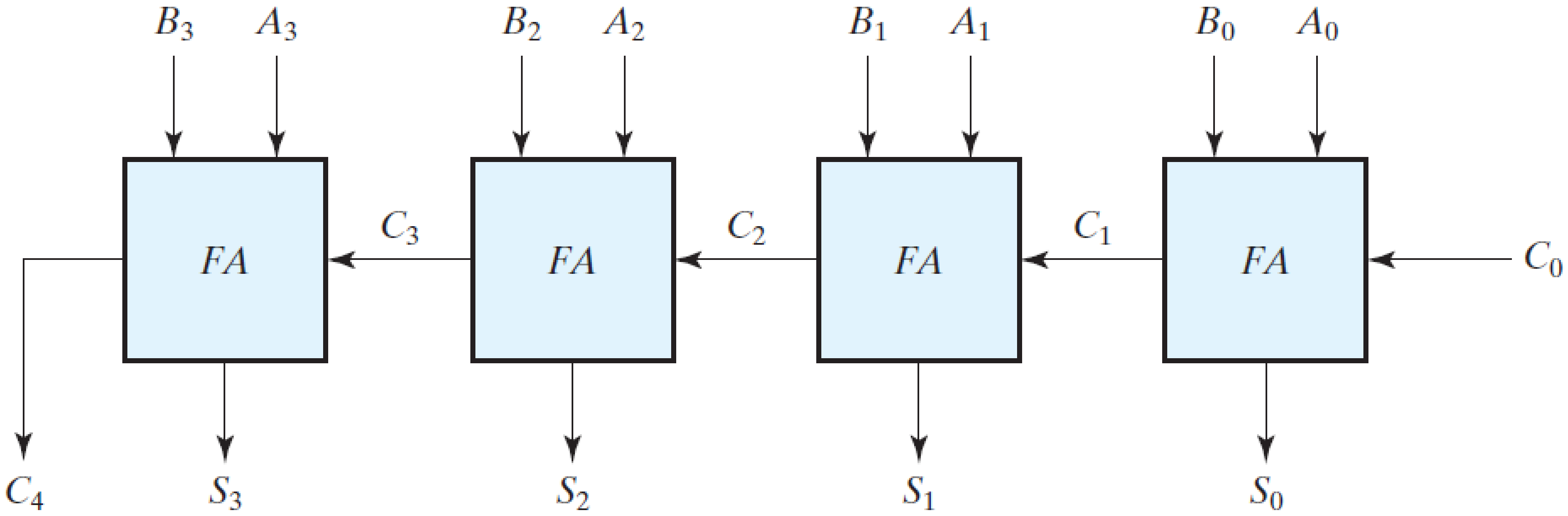
Binary adder

- We can use two half adders to create a full adder



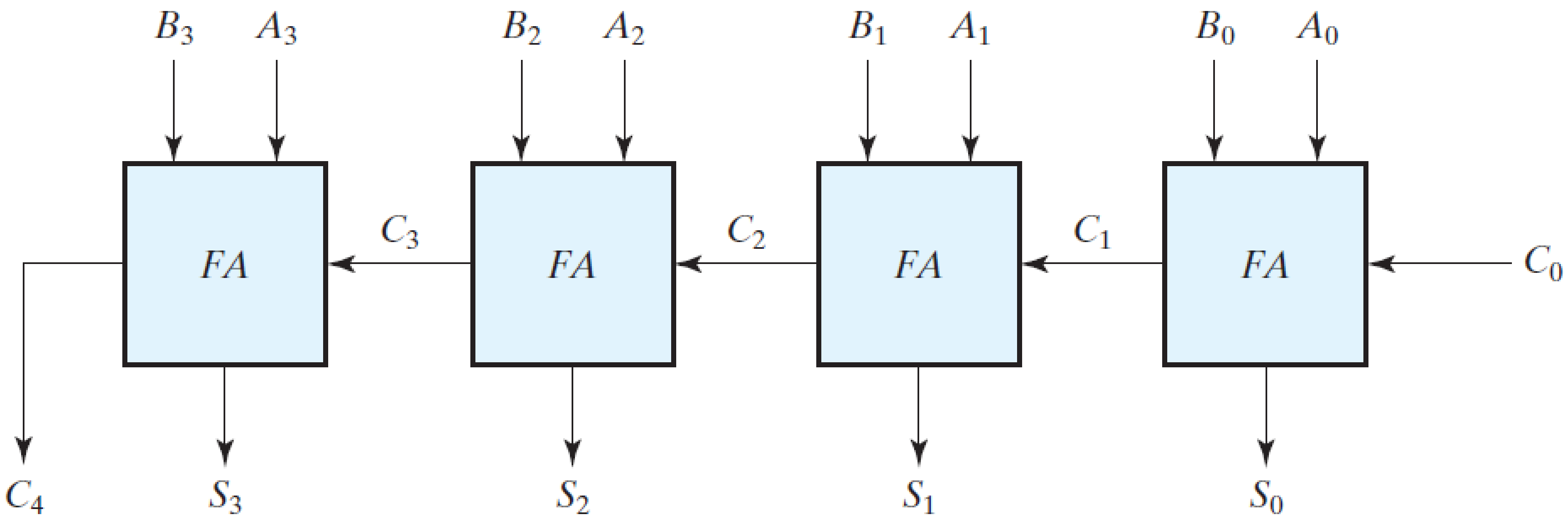
n-bit binary adder

- Addition of n-bit numbers requires a chain of n full adders or a chain of one-half adder and n-1 full adders
- Consider a four bit adder. The augend bits of A and the addend bits of B are designated by subscript numbers from right to left, with subscript 0 denoting the least significant bit
- The carries are connected in a chain through the full adders. The input carry to the adder is C_0, and it ripples through the full adders to the output carry C_4.
- The S outputs generate the required sum bits



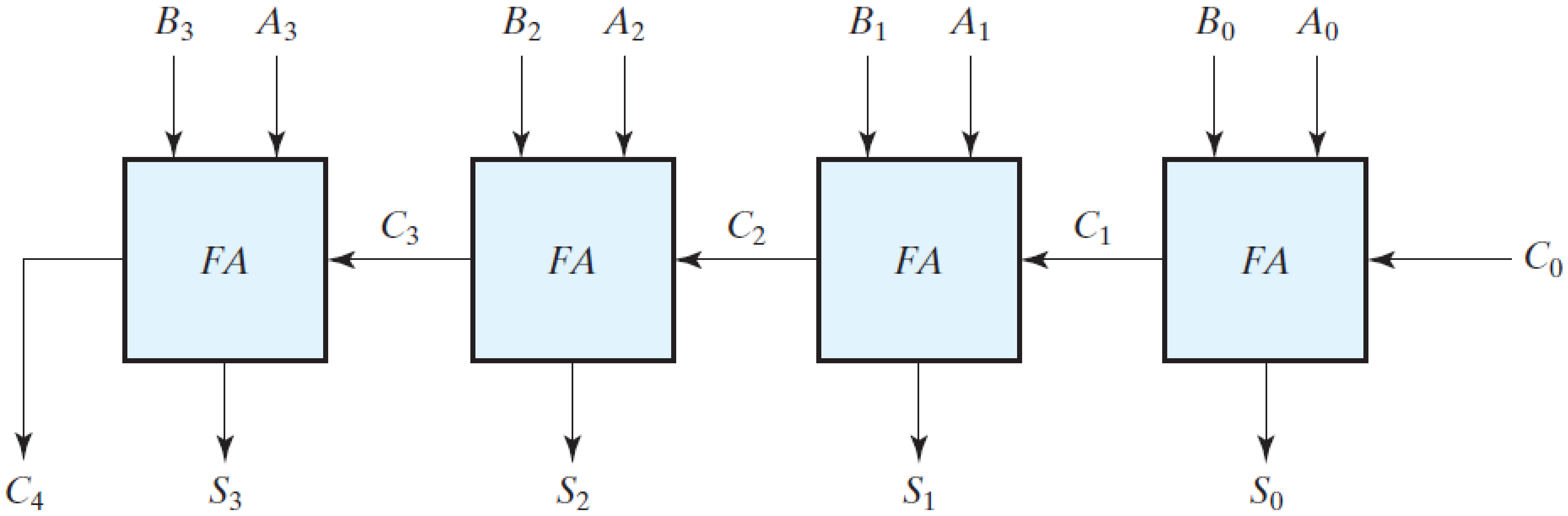
n-bit binary adder

- Can we make this circuit through the normal route?
- Note that the classical method would require a truth table (and K-map) with $2^9 = 512$ entries, since there are nine inputs to the circuit
- By using an iterative method of cascading a standard function, it is possible to obtain a simple and straightforward implementation



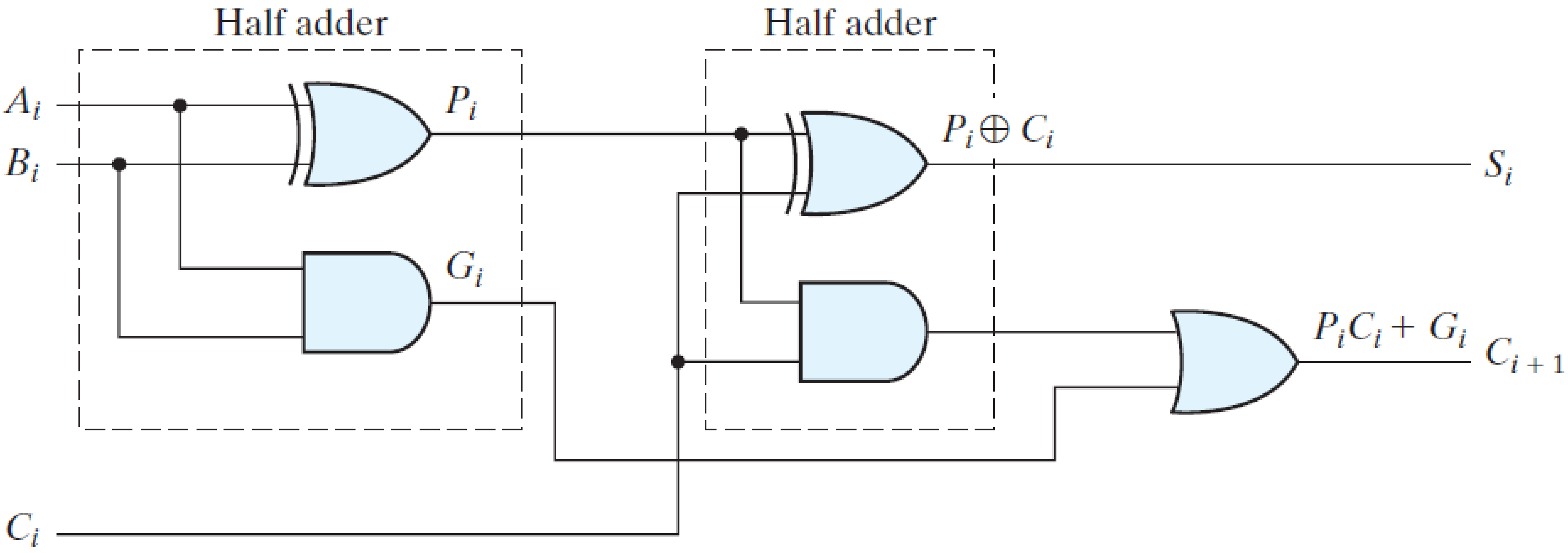
Carry propagation problem

- The addition of two binary numbers in parallel implies that all the bits of the augend and addend are available for computation at the same time
- As in any combinational circuit, the signal must propagate through the gates before the correct output sum is available in the output terminals
- The total propagation time is equal to the propagation delay of a typical gate, times the number of gate levels in the circuit
- The longest propagation delay time in an adder is the time it takes the carry to propagate through the full adders
- Since each bit of the sum output depends on the value of the input carry, the value of S_i at any given stage in the adder will be in its steady-state final value only after the input carry to that stage has been propagated



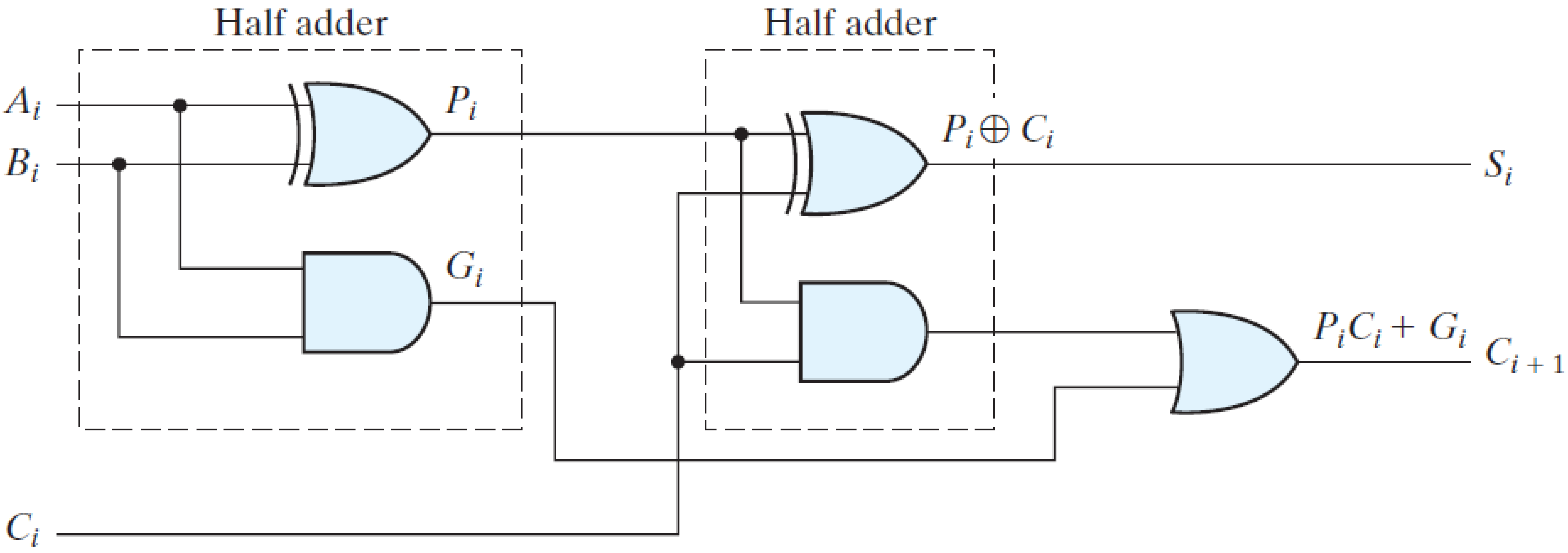
Carry propagation problem

- The number of gate levels for the carry propagation can be found from the circuit of the full adder
- The signals at P_i and G_i settle to their steady-state values after they propagate through their respective gates
- These two signals are common to all half adders and depend on only the input augend and addend bits
- The signal from the input carry C_i to the output carry C_{i+1} propagates through an AND gate and an OR gate, which constitute two gate levels
- If there are four full adders in the adder, the output carry C_4 would have 2 * 4 = 8 gate levels from C_0 to C_4
- For an n -bit adder, there are $2n$ gate levels for the carry to propagate from input to output



Carry propagation problem

- There are several techniques for reducing the carry propagation time in a parallel adder
- An obvious solution to this problem is to actually make the 2^n truth-table, K-map and get a two level implementation (either SoP or PoS)
- The most widely used technique employs the principle of *carry lookahead logic*

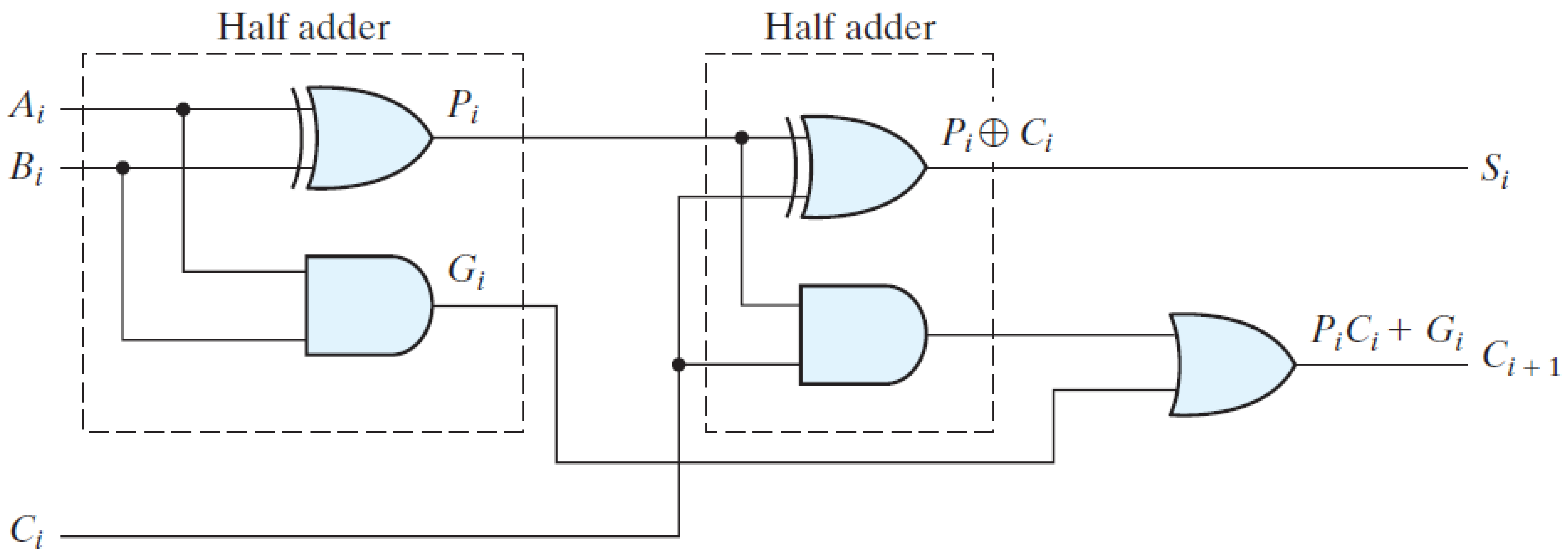


Carry propagation problem

- With the definition of P and G, we can write:

$$S_i = P_i + C_i \text{ and } C_{i+1} = G_i + P_i C_i$$

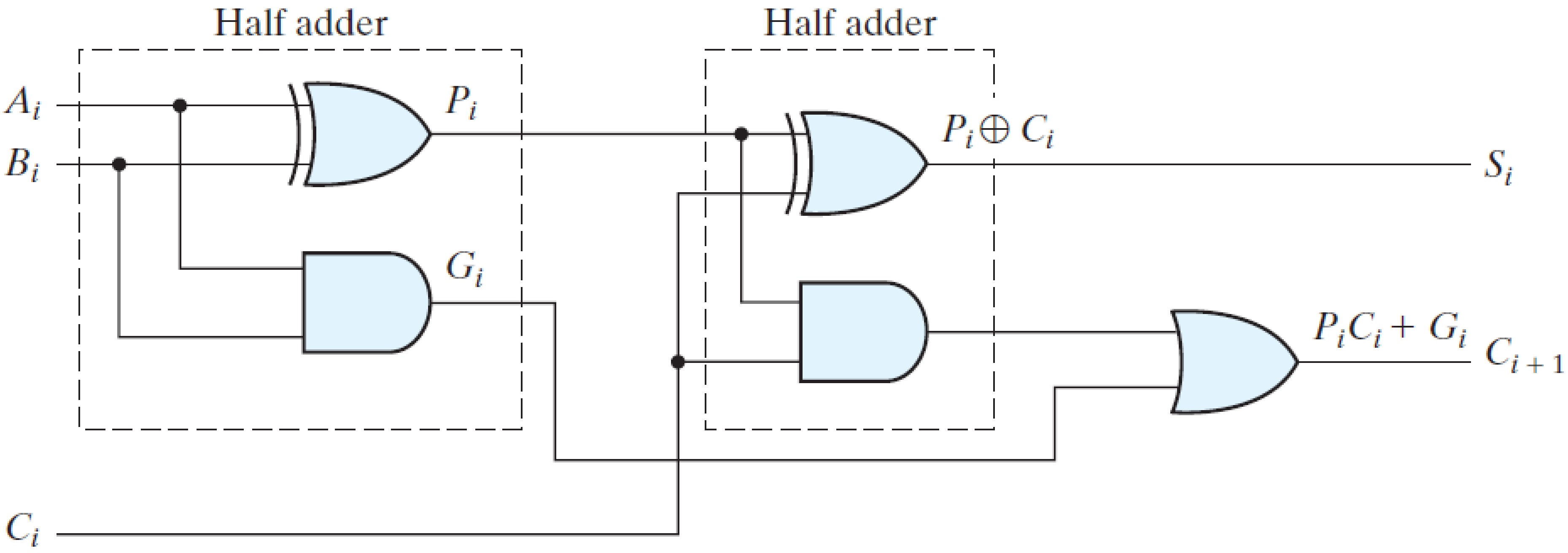
- G_i is called a *carry generate*, and it produces a carry of 1 when both A_i and B_i are 1, regardless of the input carry C_i
- P_i is called a *carry propagate*, because it determines whether a carry into stage i will propagate into stage $i + 1$ (i.e., whether an assertion of C_i will propagate to an assertion of C_{i+1})



Carry propagation problem

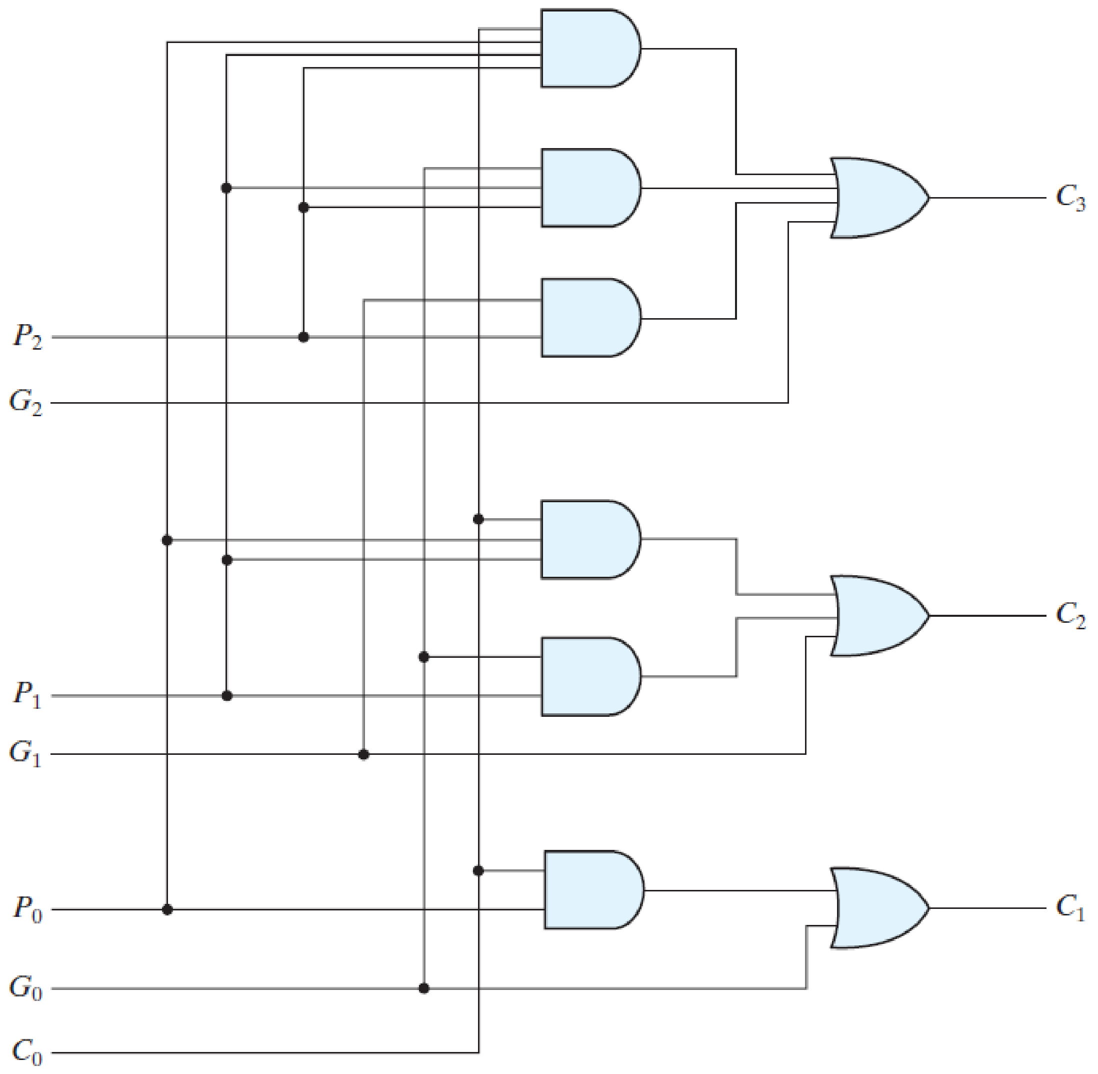
- We now write the Boolean functions for the carry outputs of each stage and substitute the value of each C_i from the previous equations:

$$C_0 = input\ carry$$
$$C_1 = G_0 + P_0C_0$$
$$C_2 = G_1 + P_1C_1 = G_1 + P_1G_0 + P_1P_0C_0$$
$$C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$$



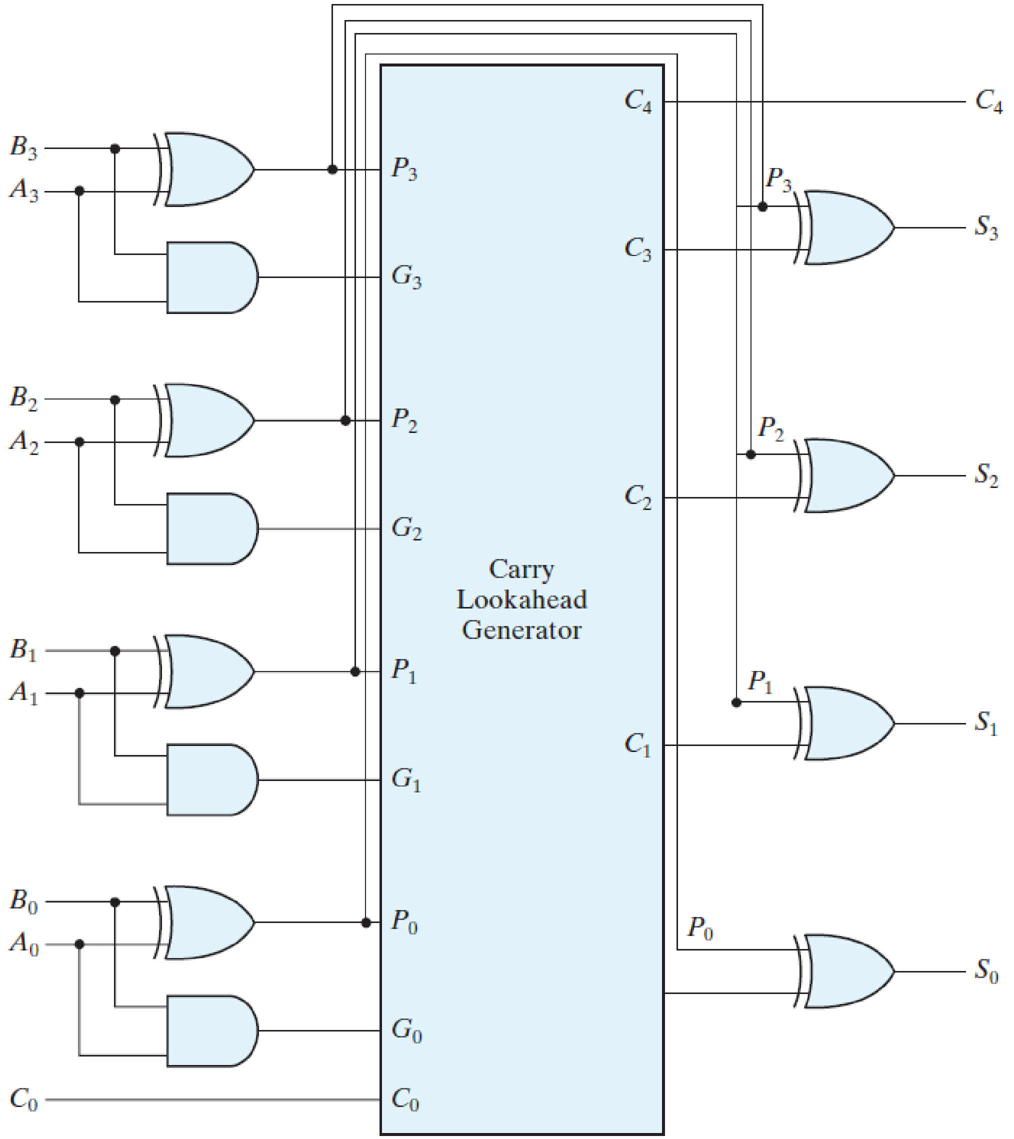
Carry propagation problem

- Since the Boolean function for each output carry is expressed in sum-of-products form only dependent on P and G, each function can be implemented with one level of AND gates followed by an OR gate (or by a two-level NAND)
- Note that this circuit can add in less time because C_3 does not have to wait for C_2 and C_1 to propagate; in fact, C_3 is propagated at the same time as C_1 and C_2
- This gain in speed of operation is achieved at the expense of additional complexity (hardware)



Carry propagation problem

- We can make the four bit adder as shown
- Each sum output requires two XOR gates
- The output of the first XOR gate generates the P_i variable, and the AND gate generates the G_i variable
- The carries are propagated through the carry lookahead generator and applied as inputs to the second XOR gate
- All output carries are generated after a delay through only two levels of gates
- Thus, outputs S_1 through S_3 have equal propagation delay times



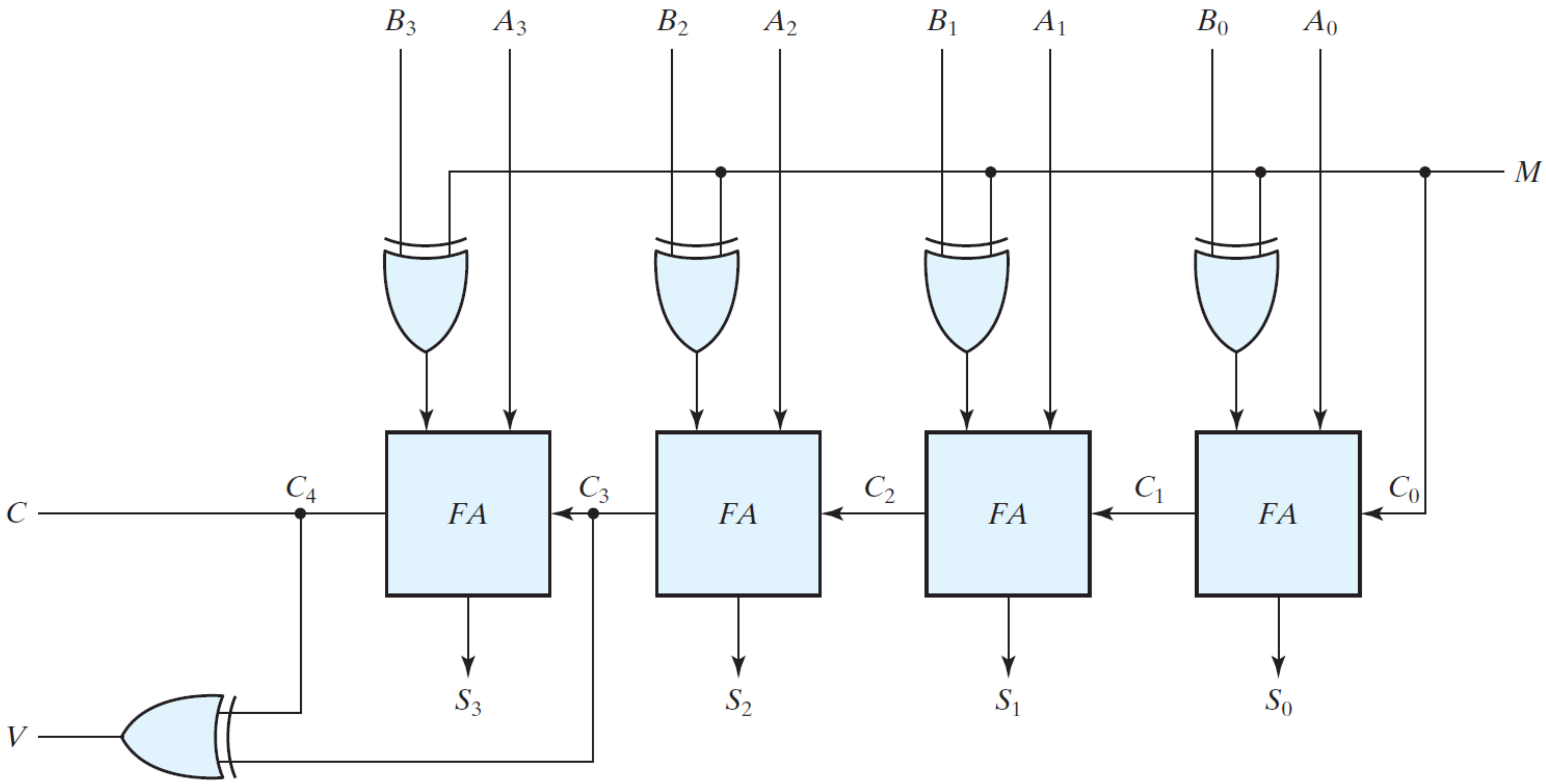
Binary subtractor

- The subtraction of unsigned binary numbers can be done most conveniently by means of complements
- Remember that the subtraction *A* - *B* can be done by taking the 2's complement of *B* and adding it to *A*
- The 2's complement can be obtained by taking the 1's complement and adding 1 to the least significant pair of bits
- The 1's complement can be implemented with inverters, and a 1 can be added to the sum through the input carry
- The circuit for subtracting *A* - *B* consists of an adder with inverters placed between each data input *B* and the corresponding input of the full adder
- The input carry C_0 must be equal to 1 when subtraction is performed
- The operation thus performed becomes *A,* plus the 1's complement of *B* , plus 1. This is equal to *A* plus the 2's complement of *B*
- For unsigned numbers, that gives *A* - *B* if $A \geq B$ or the 2's complement of *B* - *A* if $A < B$

Binary adder-subtractor

- Here is some magic: The addition and subtraction operations can be combined into one circuit with one common binary adder by including an XOR gate with each full adder
- The mode input M controls the operation
- When $M = 0$, the circuit is an adder, and when $M = 1$, the circuit becomes a subtractor
- Each XOR gate receives input M and one of the inputs of B
- When $M = 0$, we have the output as B. The full adders receive the value of B , the input carry is 0, and the circuit performs $A + B$
- When $M = 1$, we have the output as B' and $C_0 = 1$
- Thus, the B inputs are all complemented and a 1 is added through the input carry
- The circuit performs the operation A plus the 2's complement of B
- The exclusive-OR with output V is for detecting an overflow

Binary adder-subtractor



The overflow bit

- When two numbers with *n* digits each are added and the sum is a number occupying *n* + 1 digits, we say that an overflow occurred
- This is true for binary or decimal numbers, signed or unsigned
- Overflow is a problem in digital computers because the number of bits that hold the number is finite and a result that contains *n* + 1 bits cannot be accommodated by an *n*-bit word
- For this reason, many computers detect the occurrence of an overflow, and when it occurs
- The detection of an overflow after the addition of two binary numbers depends on whether the numbers are considered to be signed or unsigned
- When two unsigned numbers are added, an overflow is detected from the end carry out of the most significant position
- In the case of signed numbers, two details are important: the leftmost bit always represents the sign, and negative numbers are in 2’s-complement form
- When two signed numbers are added, the sign bit is treated as part of the number and the end carry does not indicate an overflow

The overflow bit

- An overflow cannot occur after an addition if one number is positive and the other is negative, since adding a positive number to a negative number produces a result whose magnitude is smaller than the larger of the two original numbers
- An overflow may occur if the two numbers added are both positive or both negative
- Consider the following example: Two signed binary numbers, +70 and +80, are stored in two eight-bit registers
- The range of numbers that each register can accommodate is from binary +127 to binary -128
- Since the sum of the two numbers is +150, it exceeds the capacity of an eight-bit register. This is also true for -70 and -80

carries:	0 1	
$+70$	0 1000110	
$+80$	0 1010000	
$+150$	1 0010110	

carries:	1 0
-70	1 0111010
-80	1 0110000
-150	0 1101010

The overflow bit

- In case of (+70+80), the eight-bit result that should have been positive has a negative sign bit (i.e., the eighth bit) and the eight-bit result that should have been negative has a positive sign bit
- If, however, the carry out of the sign bit position is taken as the sign bit of the result, then the nine-bit answer so obtained will be correct
- But since the answer cannot be accommodated within eight bits, we say that an overflow has occurred

carries:	0	1
+70	0	1000110
+80	0	1010000
+150	1	0010110

carries:	1	0
−70	1	0111010
−80	1	0110000
−150	0	1101010

The overflow bit

- An overflow condition can be detected by observing the carry into the sign bit position and the carry out of the sign bit position
- If these two carries are not equal, an overflow has occurred
- This is indicated in the examples in which the two carries are explicitly shown
- If the two carries are applied to an XOR gate, an overflow is detected when the output of the gate is equal to 1

carries:	0 1
+70	0 1000110
+80	0 1010000
+150	1 0010110

carries:	1 0
−70	1 0111010
−80	1 0110000
−150	0 1101010

The overflow bit

- If the two binary numbers are considered to be unsigned, then the C_4 bit detects a carry after addition or a borrow after subtraction
- If the numbers are considered to be signed, then the V bit detects an overflow
- If $V = 0$ after an addition or subtraction, then no overflow occurred and the n -bit result is correct
- If $V = 1$, then the result of the operation contains $n + 1$ bits, but only the rightmost n bits of the number fit in the space available, so an overflow has occurred
- The $(n + 1)^{th}$ bit is the actual sign and has been shifted out of position

$$\begin{array}{rr} \text{carries:} & 0\ \ 1 \\ +70 & 0\ \ 1000110 \\ +80 & 0\ \ 1010000 \\ \hline +150 & 1\ \ 0010110 \end{array}$$

$$\begin{array}{rr} \text{carries:} & 1\ \ 0 \\ -70 & 1\ \ 0111010 \\ -80 & 1\ \ 0110000 \\ \hline -150 & 0\ \ 1101010 \end{array}$$