Function with Don’t care inputs

- Don’t cares included while computing Prime implicants

-In the Selection of Essential Prime implicants don’t cares not used.

Simplify Using QM Method

$F(A,B,C,D) = \Sigma(6,7,14)$

$d(A,B,C,D) = \Sigma(0,8,15)$

EX-OR Function

$x \oplus y = xy' + x'y$

EX-NOR Function

$(x \oplus y)' = xy + x'y'$

Interesting XOR properties

- There are several fascinating properties of XOR that you can prove using Boolean algebra, starting from the definition $x \oplus y = x'y + xy'$

$x \oplus 0 = x$	$x \oplus 1 = x'$	
$x \oplus x = 0$	$x \oplus x' = 1$	
$x \oplus (y \oplus z) = (x \oplus y) \oplus z$		Associative
$x \oplus y = y \oplus x$		Commutative

A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

3-input EX-OR

$$Y = A \oplus B \oplus C$$
$$= A'B'C + A'BC' \quad AB'C' + ABC$$
$$= \Sigma\,(1,2,4,7)$$

- ODD Function

	1		1
1		1	

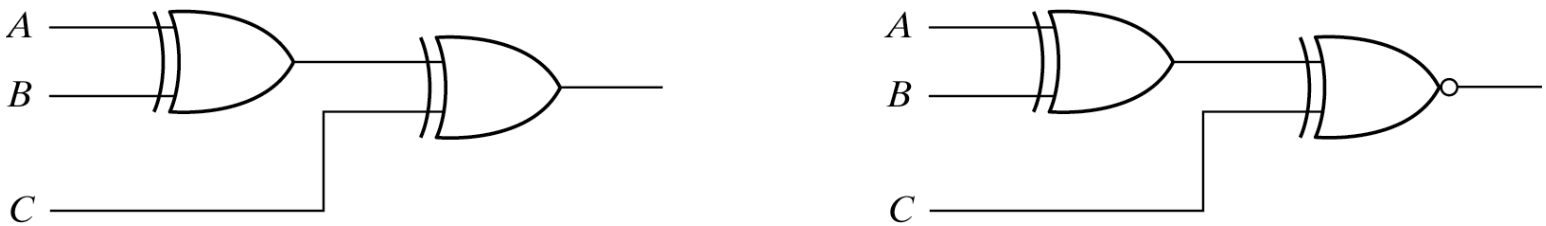
Odd Function

$F = A \oplus B \oplus C$

1		1	
	1		1

Even Function

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



(a) 3-input odd function

(b) 3-input even function

Fig. 3-34 Logic Diagram of Odd and Even Functions

	1		1
1		1	
	1		1
1		1	

$F = \mathbf{A \oplus B \oplus C \oplus D}$

- ODD FUNCTION

1		1	
	1		1
1		1	
	1		1

$F = \mathbf{(A \oplus B \oplus C \oplus D)'}$

- EVEN FUNCTION

Parity Generation and Checking:

Useful in error detection and correction

Parity bit- extra bit included with binary message

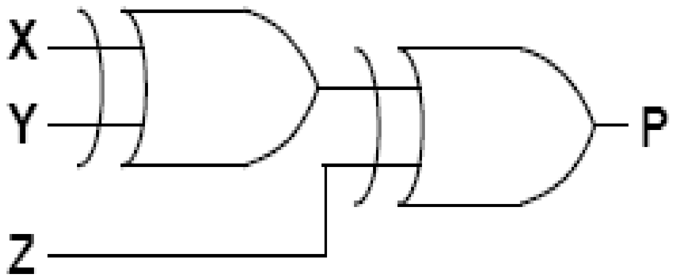
Parity Generator

Parity checker

- A parity generator/checker can detect a 1-bit error in a message.

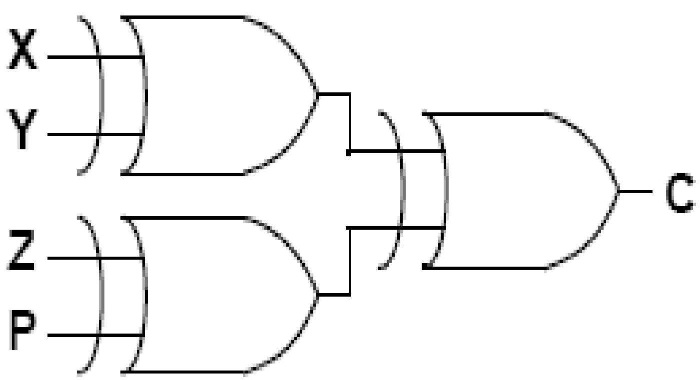
To generate an even parity bit

$$P = X \oplus Y \oplus Z$$



To check a even parity bit

$$C = X \oplus Y \oplus Z \oplus P$$



Message			Even	
X	**Y**	**Z**	Parity Bit, **P**	**C**
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	0
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	0

If no errors detected, **C = 0**

COMBINATIONAL LOGIC

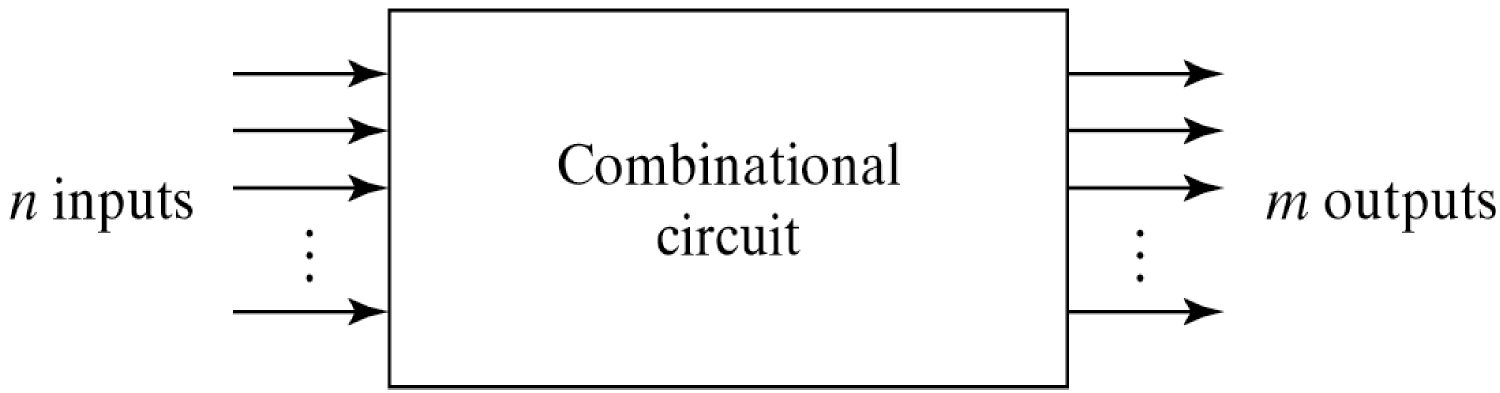


Fig. 4-1 Block Diagram of Combinational Circuit

BCD-to-Excess-3 Code Converter

- Design a circuit that converts a binary-coded-decimal (BCD) codeword to its corresponding excess-3 codeword.
- Excess-3 code: Given a decimal digit *n*, its corresponding excess-3 codeword $(n+3)_2$
  Example:
  $n=5 \rightarrow n+3=8 \rightarrow 1000_{excess\text{-}3}$
  $n=0 \rightarrow n+3=3 \rightarrow 0011_{excess\text{-}3}$
- We need 4 input variables (A,B,C,D) and 4 output functions W(A,B,C,D), X(A,B,C,D), Y(A,B,C,D), and Z(A,B,C,D).

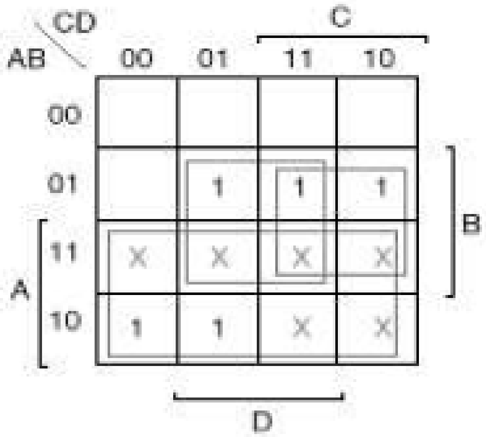
BCD-to-Excess-3 Converter (cont.)

- The truth table relating the input and output variables is shown below.
- Note that the outputs for inputs 1010 through 1111 are *don't cares* (not shown here).

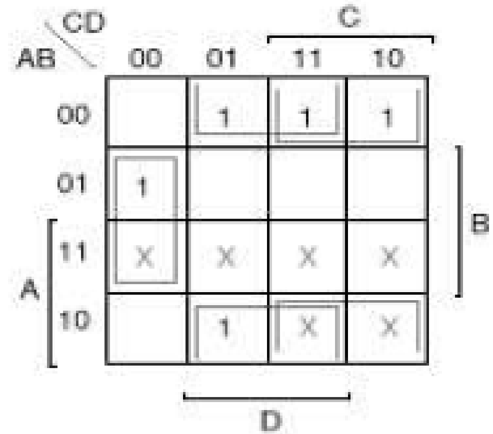
Decimal Digit	Input BCD				Output Excess-3			
	A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	1
9	1	0	0	1	1	1	0	0

Maps for BCD-to-Excess-3 Code Converter

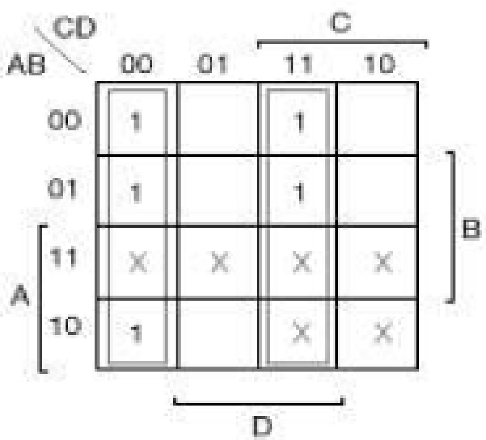
The K-maps for are constructed using the don't care terms



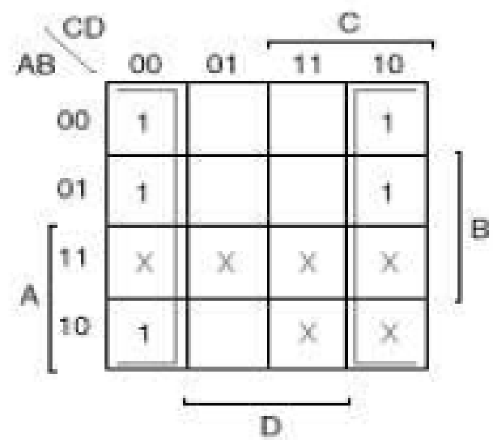
W = A + BC + BD



$X = \overline{B}C + \overline{B}D + B\overline{C}\overline{D}$

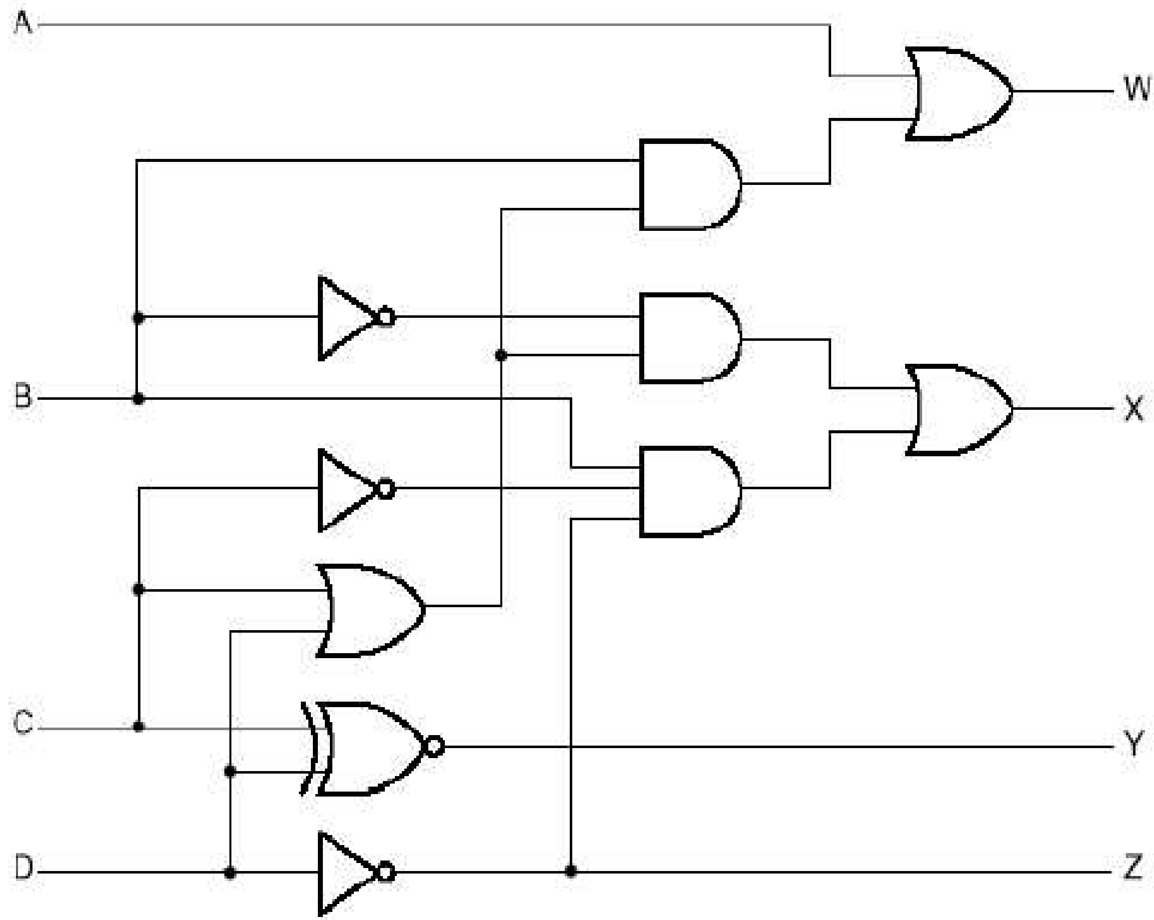


$Y = CD + \overline{C}\overline{D}$



$Z = \overline{D}$

BCD-to-Excess-3 Converter (cont.)



Binary addition by hand

- You can add two binary numbers one column at a time starting from the right, just like you add two decimal numbers.
- But remember it's binary. For example, 1 + 1 = 10 and you have to carry!

The initial carry in is implicitly 0

	1	1	1	0		Carry in
		1	0	1	1	Augend
+		1	1	1	0	Addend
	1	1	0	0	1	Sum

most significant bit (MSB)

least significant bit (LSB)

HALF ADDER

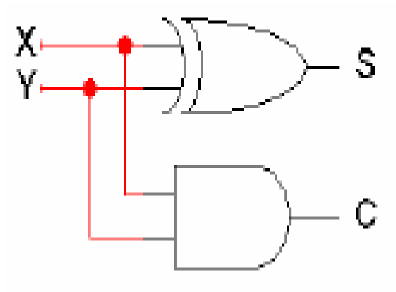
Adding two bits

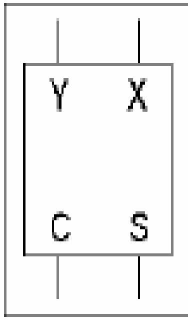
- We'll make a hardware adder based on our human addition algorithm.
- We start with a half adder, which adds two bits X and Y and produces a two-bit result: a sum S (the right bit) and a carry out C (the left bit).
- Here are truth tables, equations, circuit and block symbol.

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

$$C = XY$$

$$S = X'Y + XY' = X \oplus Y$$





FULL ADDER

Adding three bits

- But what we really need to do is add *three* bits: the augend and addend bits, *and* the carry in from the right.
- A full adder circuit takes three inputs X, Y and C_{in}, and produces a two-bit output consisting of a sum S and a carry out C_{out}.

	1	1	1	0	
		1	0	1	1
+		1	1	1	0
	1	1	0	0	1

X	Y	C_{in}	C_{out}	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 equations

- Using Boolean algebra, we can simplify S and C_{out} as shown here.

X	Y	C_{in}	C_{out}	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

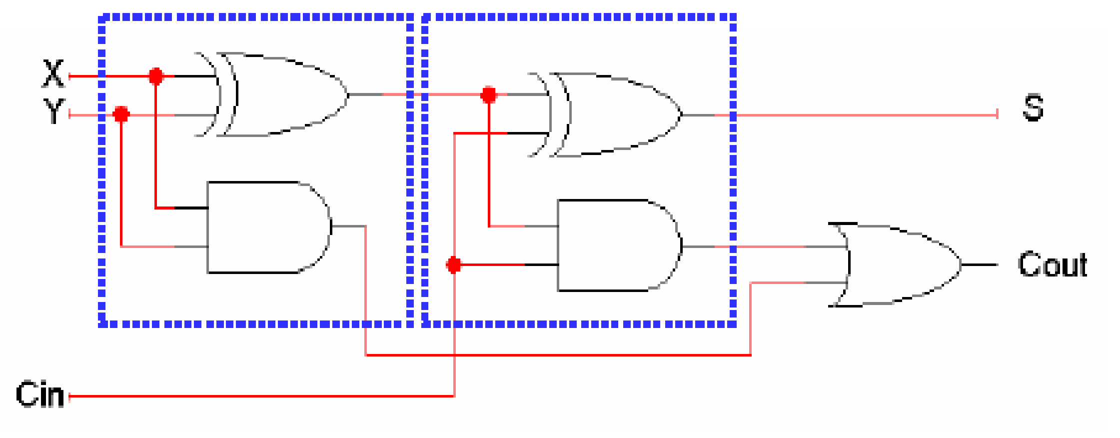
$$
\begin{aligned}
S &= \Sigma m(1,2,4,7) \\
&= X'Y'C_{in} + X'YC_{in}' + XY'C_{in}' + XYC_{in} \\
&= X'(Y'C_{in} + YC_{in}') + X(Y'C_{in}' + YC_{in}) \\
&= X'(Y \oplus C_{in}) + X(Y \oplus C_{in})' \\
&= X \oplus Y \oplus C_{in}
\end{aligned}
$$

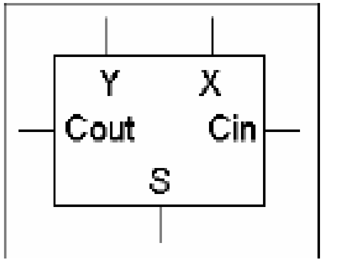
$$
\begin{aligned}
C_{out} &= \Sigma m(3,5,6,7) \\
&= X'YC_{in} + XY'C_{in} + XYC_{in}' + XYC_{in} \\
&= (X'Y + XY')C_{in} + XY(C_{in}' + C_{in}) \\
&= (X \oplus Y)C_{in} + XY
\end{aligned}
$$

Full adder circuit

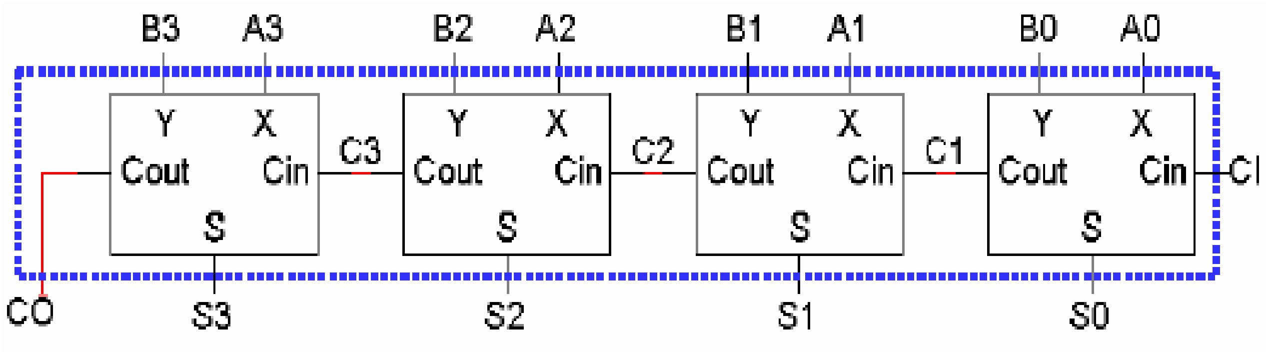
- We write the equations this way to highlight the hierarchical nature of adder circuits—you can build a full adder by combining two half adders!

$$S = X \oplus Y \oplus C_{in}$$
$$C_{out} = (X \oplus Y)\, C_{in} + XY$$

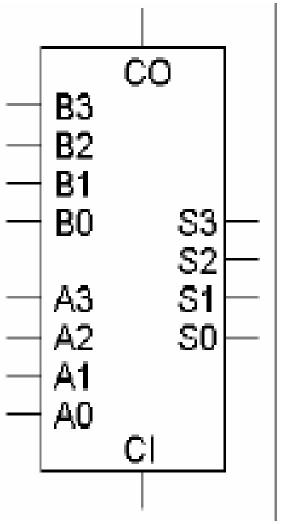




A four-bit adder

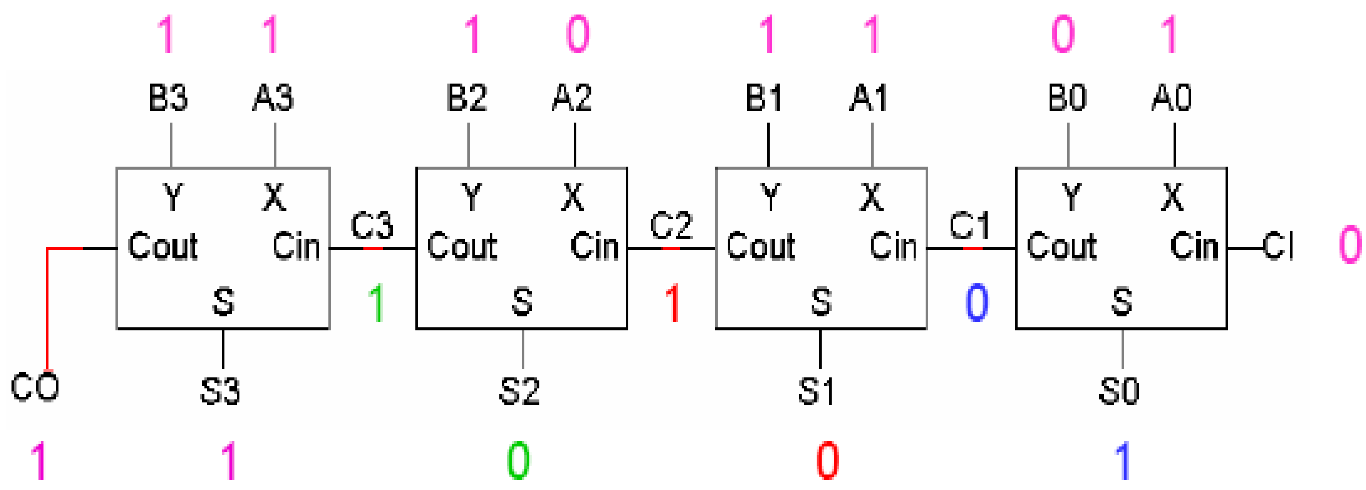


- Similarly, we can cascade four full adders to build a four-bit adder.
  - The inputs are two four-bit numbers (A3A2A1A0 and B3B2B1B0) and a carry in CI.
  - The two outputs are a four-bit sum S3S2S1S0 and the carry out CO.
- If you designed this adder without taking advantage of the hierarchical structure, you'd end up with a 512-row truth table with five outputs!



An example of 4-bit addition

- Let's put our initial example into this circuit, with A=1011 and B=1110.



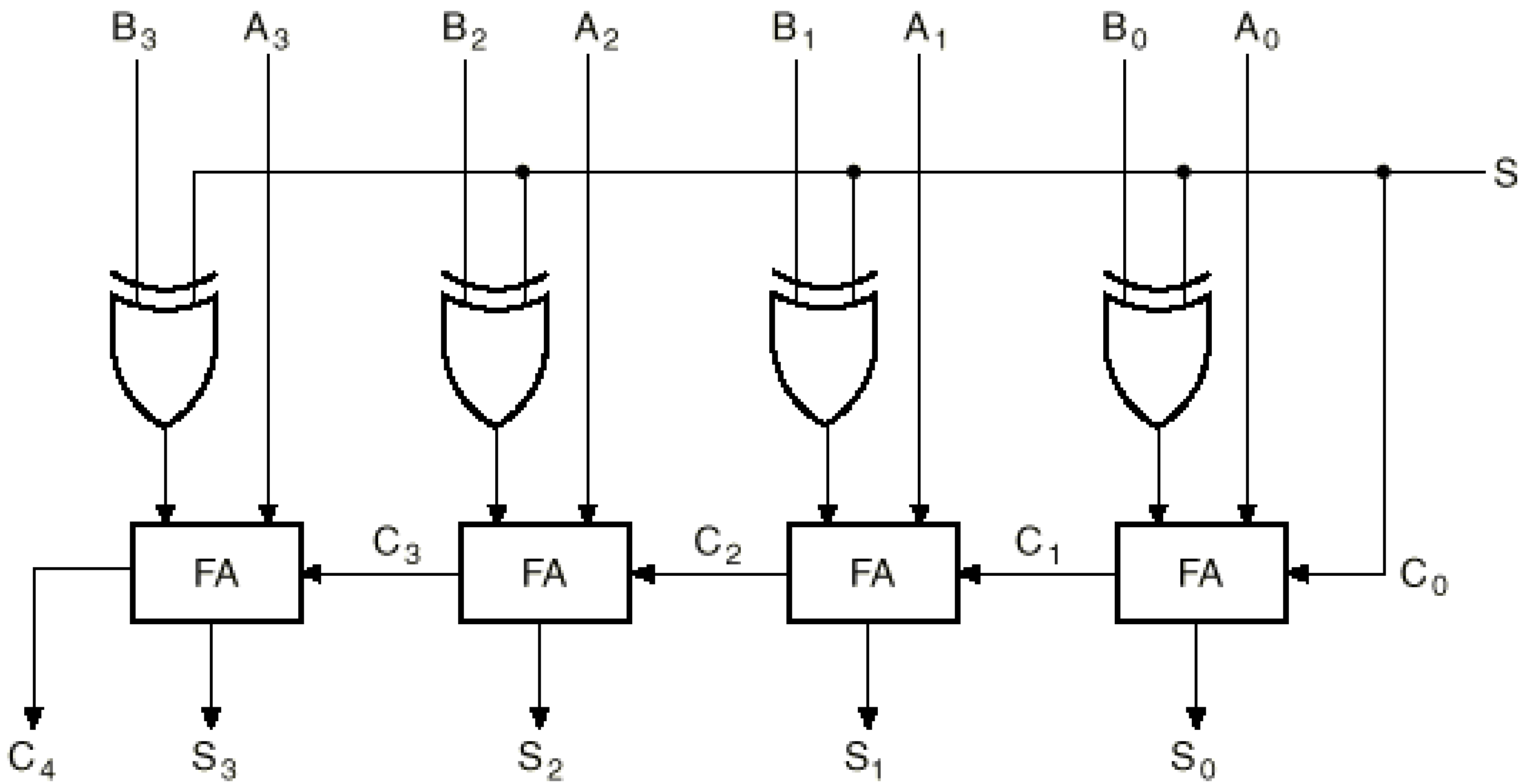
1. Fill in all the inputs, including CI=0
2. The circuit produces C1 and S0 (1 + 0 + 0 = 01)
3. Use C1 to find C2 and S1 (1 + 1 + 0 = 10)
4. Use C2 to compute C3 and S2 (0 + 1 + 1 = 10)
5. Use C3 to compute CO and S3 (1 + 1 + 1 = 11)

Binary Adder/Subtractors

- The subtraction A-B can be performed by taking the 2's complement of B and adding to A.
- The 2's complement of B can be obtained by complementing B and adding one to the result.

$$
\begin{aligned}
A\text{-}B &= A + 2C(B) \\
&= A + 1C(B) + 1 \\
&= A + B' + 1
\end{aligned}
$$

4-bit Binary Adder/Subtractor

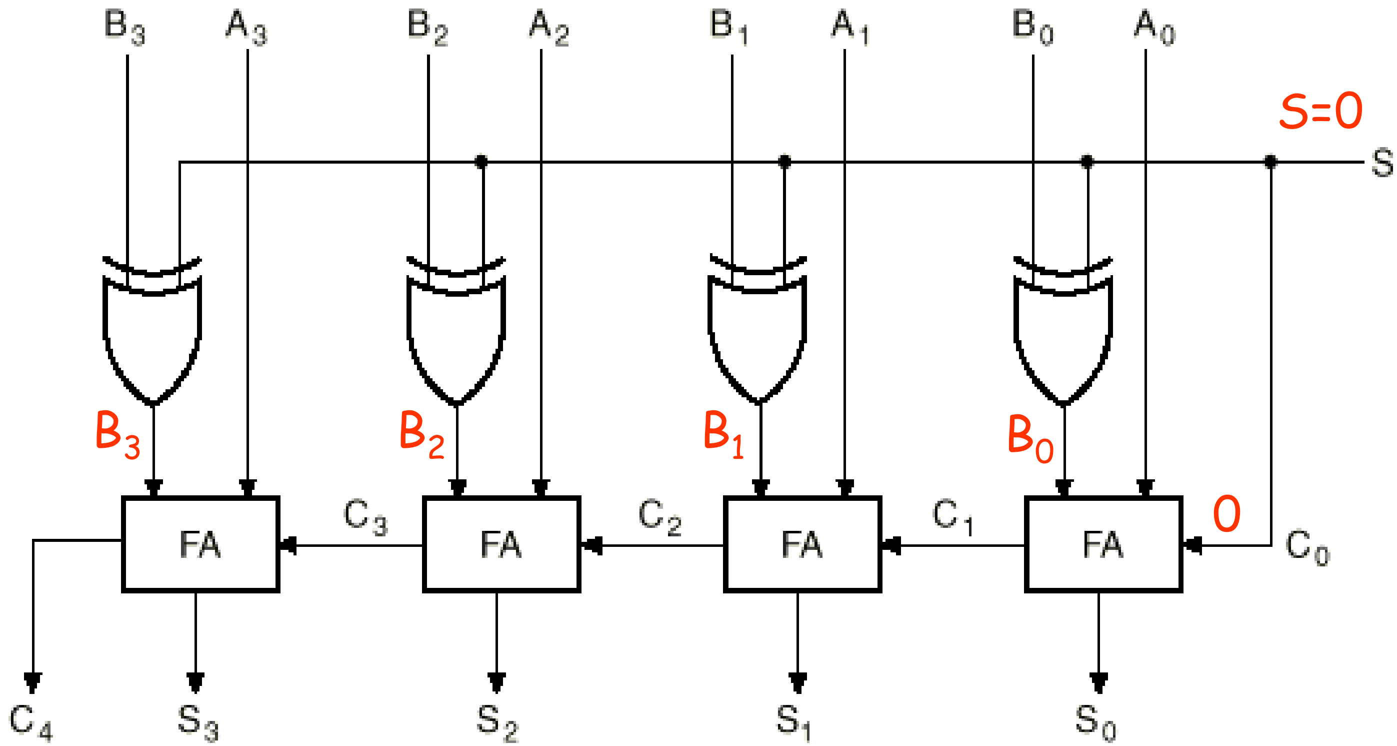


-XOR gates act as programmable inverters

4-bit Binary Adder/Subtractor (cont.)

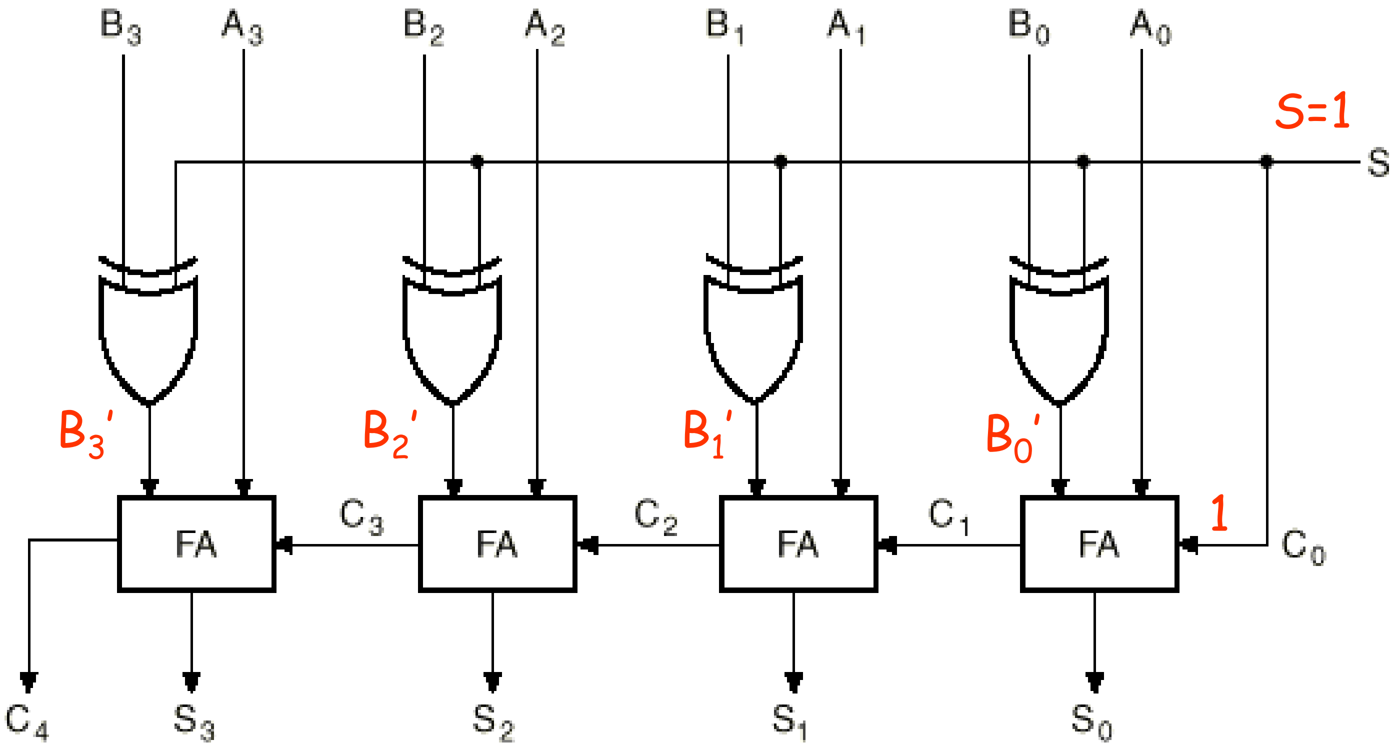
- **When *S*=0, the circuit performs *A* + *B*. The carry in is 0, and the XOR gates simply pass *B* untouched.**
- **When *S*=1, the carry into the least significant bit (LSB) is 1, and *B* is complemented (1's complement) prior to the addition; hence, the circuit adds to A the 1's complement of *B* plus 1 (from the carry into the LSB).**

4-bit Binary Adder/Subtractor (cont.)



S=0 selects addition

4-bit Binary Adder/Subtractor (cont.)



S=1 selects subtraction

Decimal Adders

-Is Binary sum less than or equal to 1001

-For binary sum more than 1001 add 0110

-Circuit needs modification

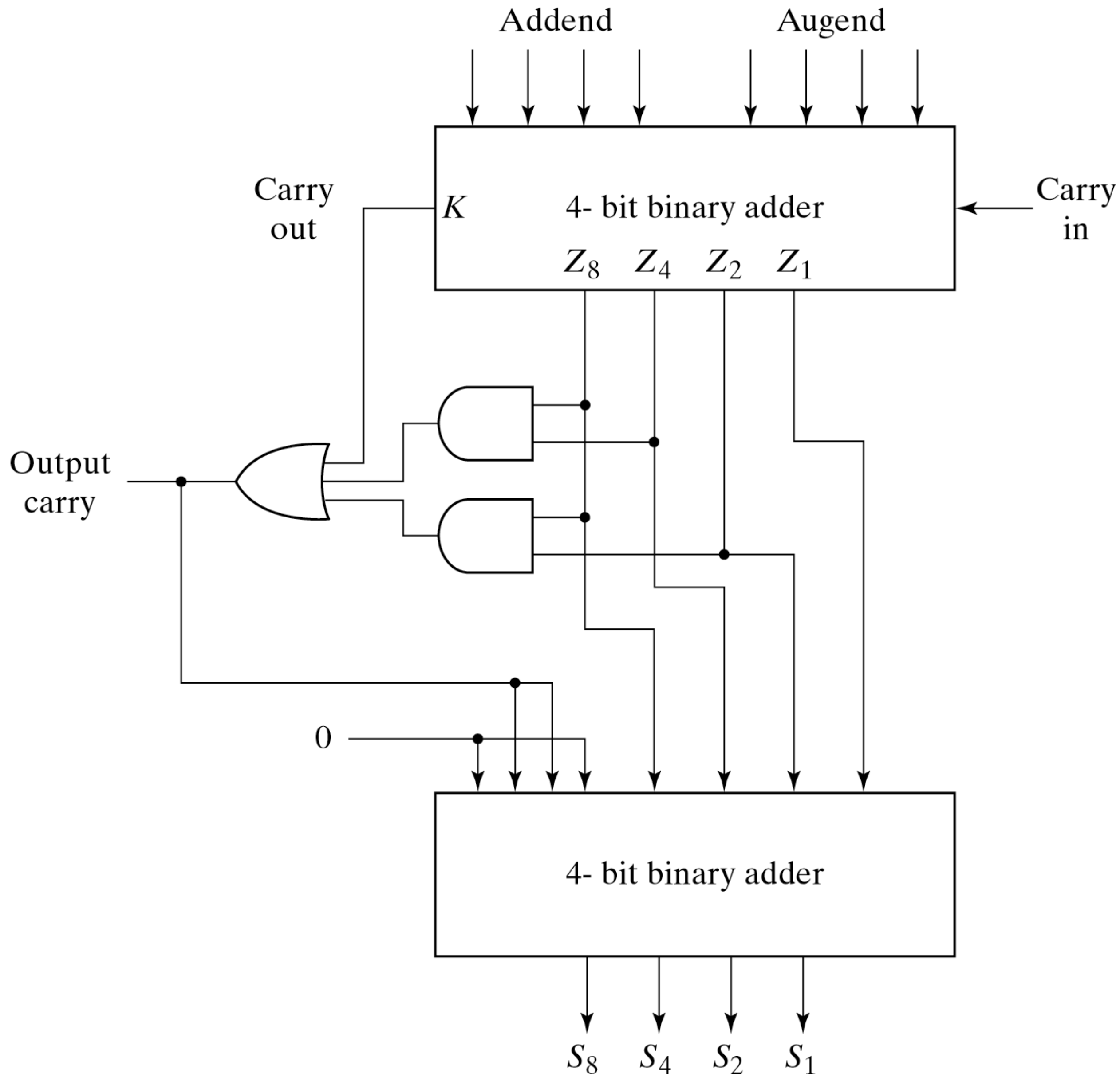
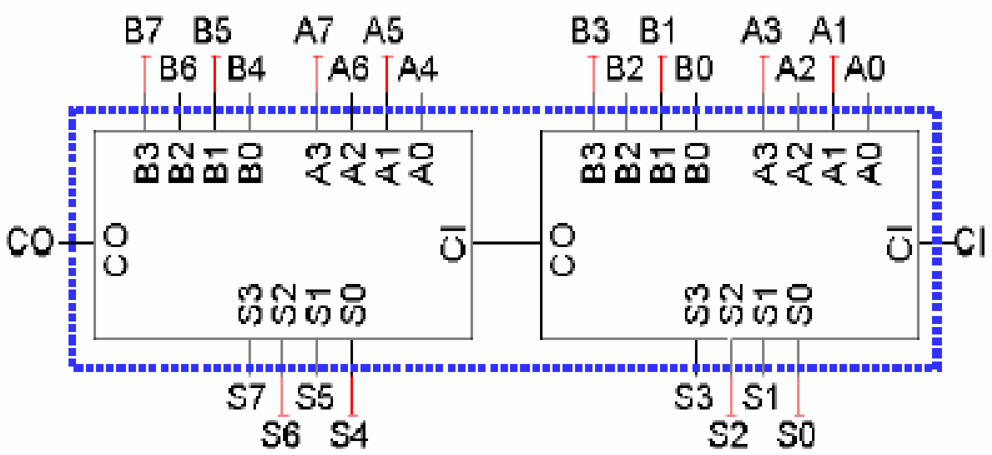


Fig. 4-14 Block Diagram of a BCD Adder

Hierarchical adder design

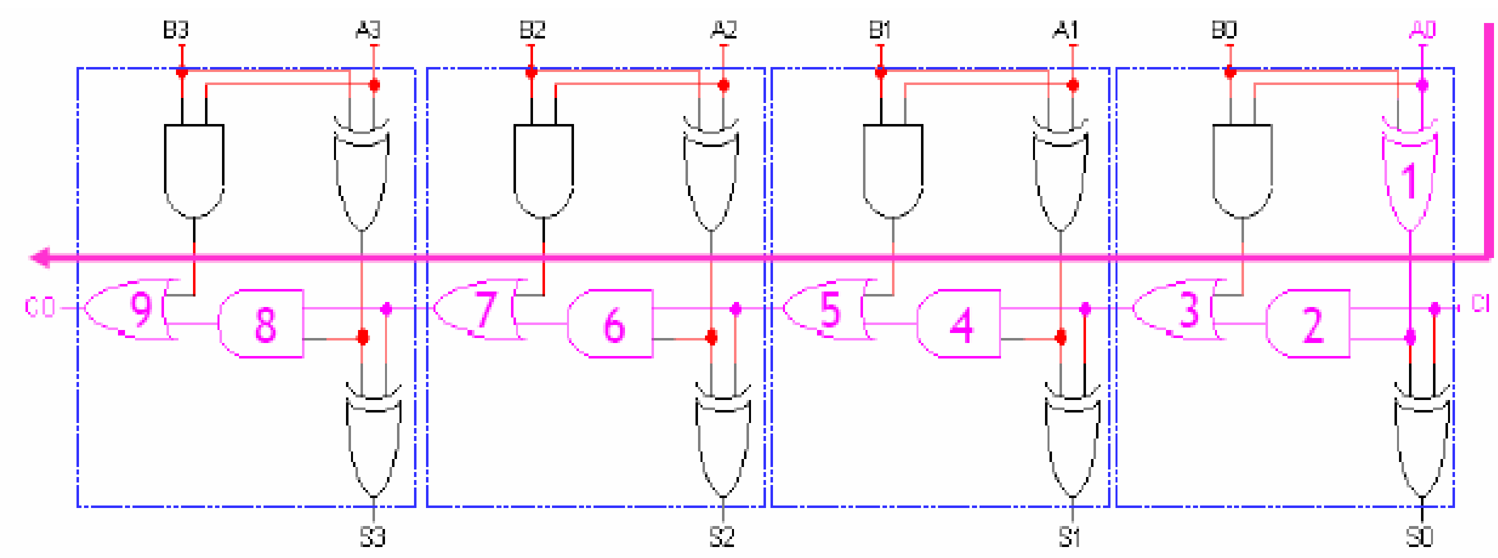
- When you add two 4-bit numbers the carry in is always 0, so why does the four-bit adder have a CI input?
- We can use CI to combine four-bit adders together to make even larger adders, just like we combined half adders and full adders earlier.
- Here is one way to build an eight-bit adder, for example.



- CI is also useful for subtraction,

Ripple carry delays

- The diagram below shows our four-bit adder completely drawn out.
- This is called a ripple carry adder, because the inputs A0, B0 and CI "ripple" leftwards until CO and S3 are produced.
- Ripple carry adders are slow!
  - There is a very long path from A0, B0 and CI to CO and S3.
  - For an *n*-bit ripple carry adder, the longest path has 2*n*+1 gates.
  - The longest path in a 64-bit adder would include 129 gates!



A faster way to compute carry outs

- Instead of waiting for the carry out from each previous stage, we can minimize the delay by computing it directly with a two-level circuit.
- First we'll define two functions.
  - The "generate" function G_i produces 1 when there *must* be a carry out from position i (i.e., when A_i and B_i are both 1).

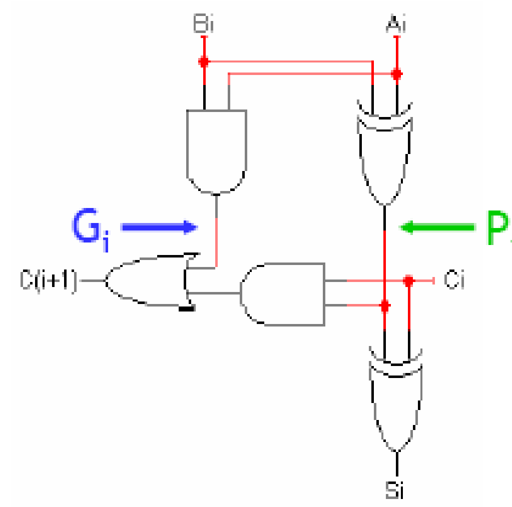
$$G_i = A_iB_i$$

  - The "propagate" function P_i is true when an incoming carry is propagated (i.e, when A_i=1 or B_i=1, but not both).

$$P_i = A_i \oplus B_i$$

- Then we can rewrite the carry out function.

$$C_{i+1} = G_i + P_iC_i$$



A_i	B_i	C_i	C_{i+1}
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

- Let's look at the carry out equations for specific bits, using the general equation from the previous page $C_{i+1} = G_i + P_iC_i$.

$$
\begin{aligned}
C_1 &= G_0 + P_0C_0 \\[1em]
C_2 &= G_1 + P_1C_1 \\
&= G_1 + P_1(G_0 + P_0C_0) \\
&= G_1 + P_1G_0 + P_1P_0C_0 \\[1em]
C_3 &= G_2 + P_2C_2 \\
&= G_2 + P_2(G_1 + P_1G_0 + P_1P_0C_0) \\
&= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0 \\[1em]
C_4 &= G_3 + P_3C_3 \\
&= G_3 + P_3(G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0) \\
&= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0
\end{aligned}
$$

- These expressions are all sums of products, so we can use them to make a circuit with only a two-level delay.

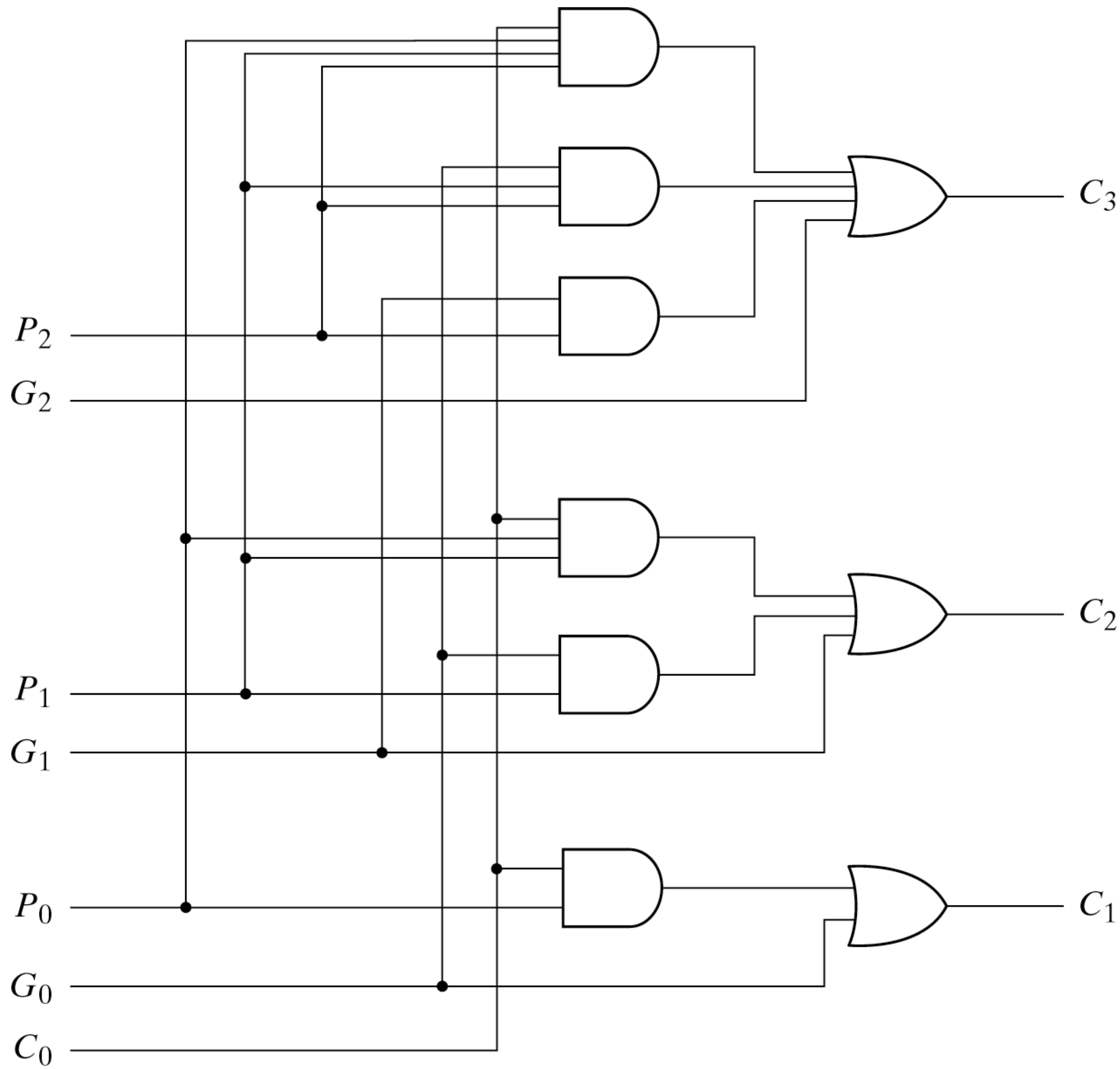


Fig. 4-11 Logic Diagram of Carry Lookahead Generator

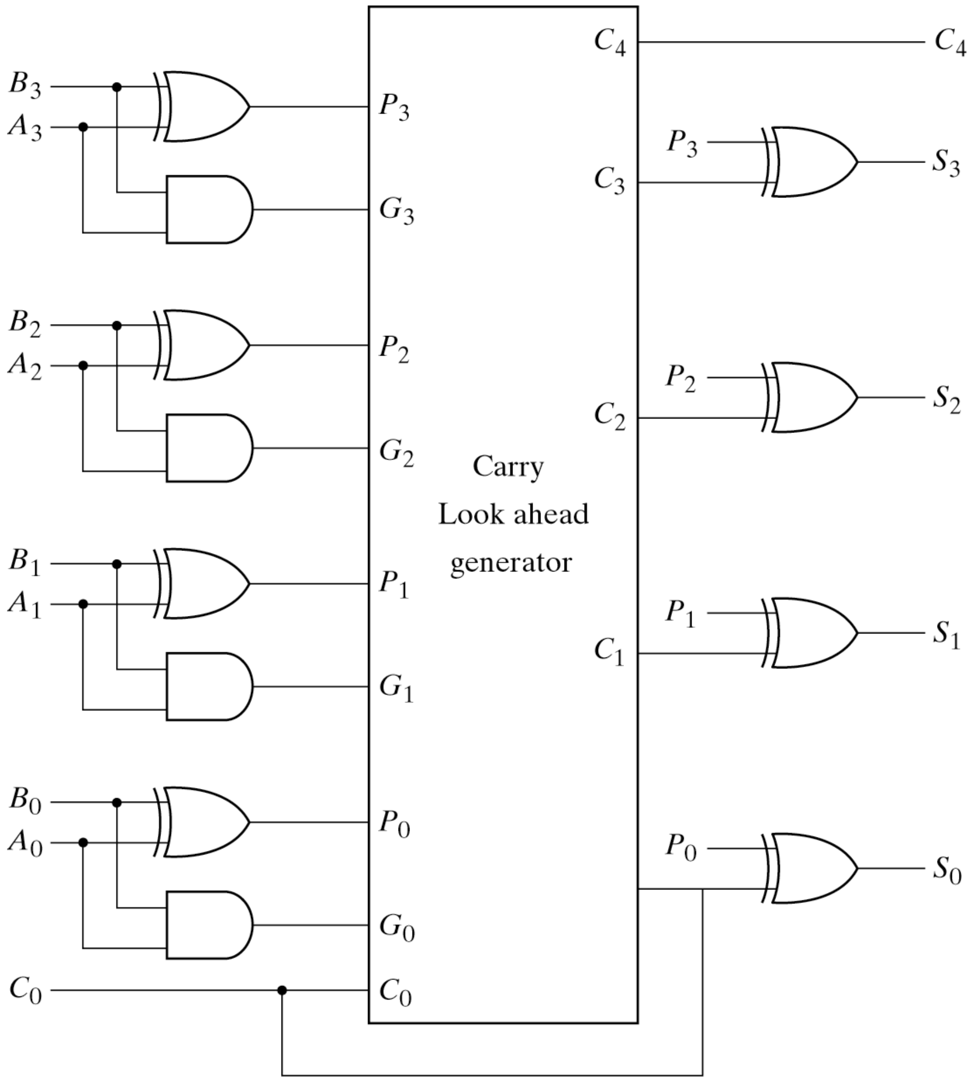
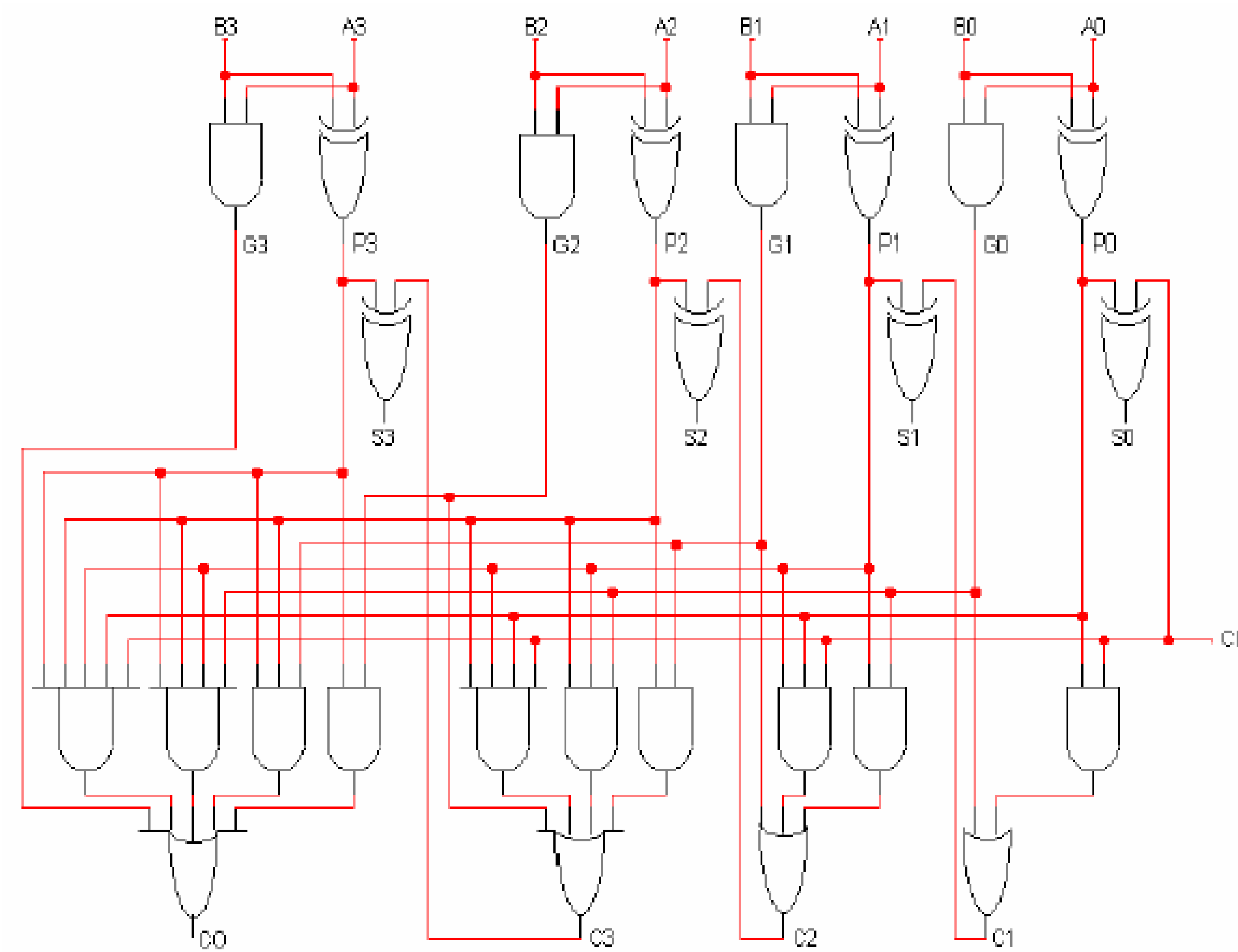


Fig. 4-12 4-Bit Adder with Carry Lookahead

A faster four-bit adder



Binary multiplication by hand

- Multiplication can't be that hard! It's just repeated addition, so if we have adders, we should be able to do multiplication also.
- Here's an example of binary multiplication

$$
\begin{array}{cccccccc}
 & & & & 1 & 1 & 0 & 1 \\
 & & & \times & 0 & 1 & 1 & 0 \\
\hline
 & & & & 0 & 0 & 0 & 0 \\
 & & & 1 & 1 & 0 & 1 & \\
 & & 1 & 1 & 0 & 1 & & \\
+ & 0 & 0 & 0 & 0 & & & \\
\hline
 & 1 & 0 & 0 & 1 & 1 & 1 & 0
\end{array}
$$

Binary multiplication

				1	1	0	1	Multiplicand
			×	0	1	1	0	Multiplier
				0	0	0	0	Partial products
			1	1	0	1		
		1	1	0	1			
+	0	0	0	0				
	1	0	0	1	1	1	0	Product

- Since we always multiply by either 0 or 1, the partial products are always either 0000 or the multiplicand (1101 in this example).
- There are four partial products which are added to form the result.
  - We can add them in pairs, using three adders.
  - The product can have up to 8 bits, but we can use four-bit adders if we stagger them leftwards, like the partial products themselves.

2×2 binary multiplication

- Here is an outline of multiplying the two-bit numbers A1A0 and B1B0, to produce the four-bit product P3-P0.

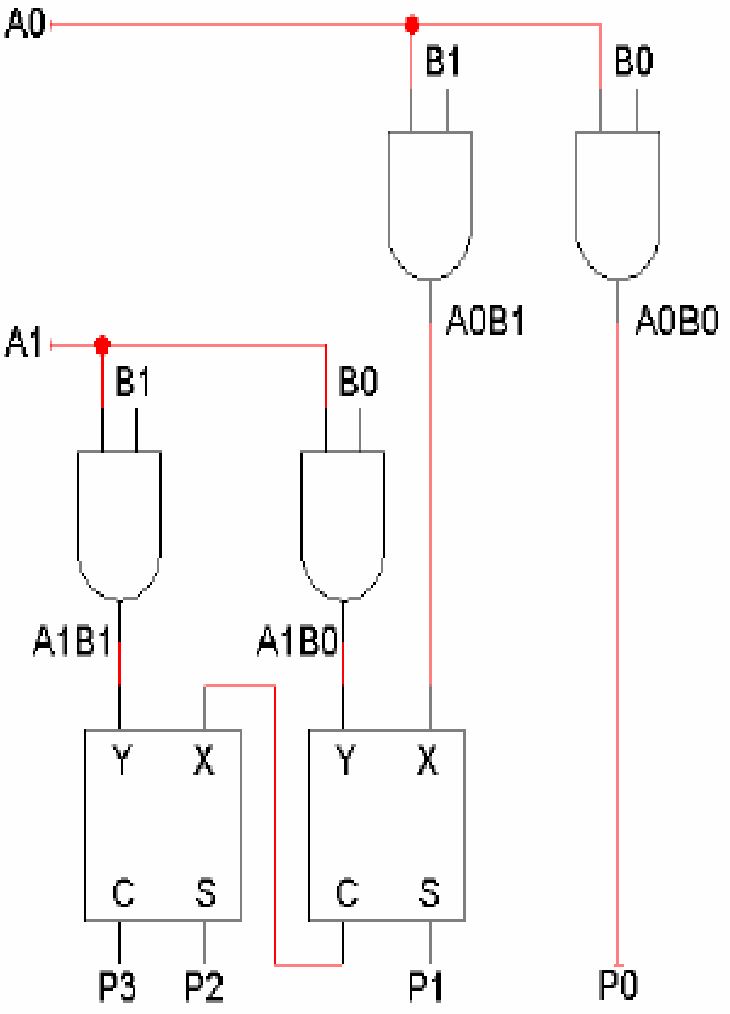
$$
\begin{array}{rrrrr}
 & & & B1 & B0 \\
 & & \times & A1 & A0 \\
\cline{3-5}
 & & & A0B1 & A0B0 \\
+ & & A1B1 & A1B0 & \\
\hline
 & P3 & P2 & P1 & P0
\end{array}
$$

- The bits of each partial product are computed by multiplying two bits of the input.
- Since two-bit multiplication is the same as the logical AND operation, we can use AND gates to generate the partial products.

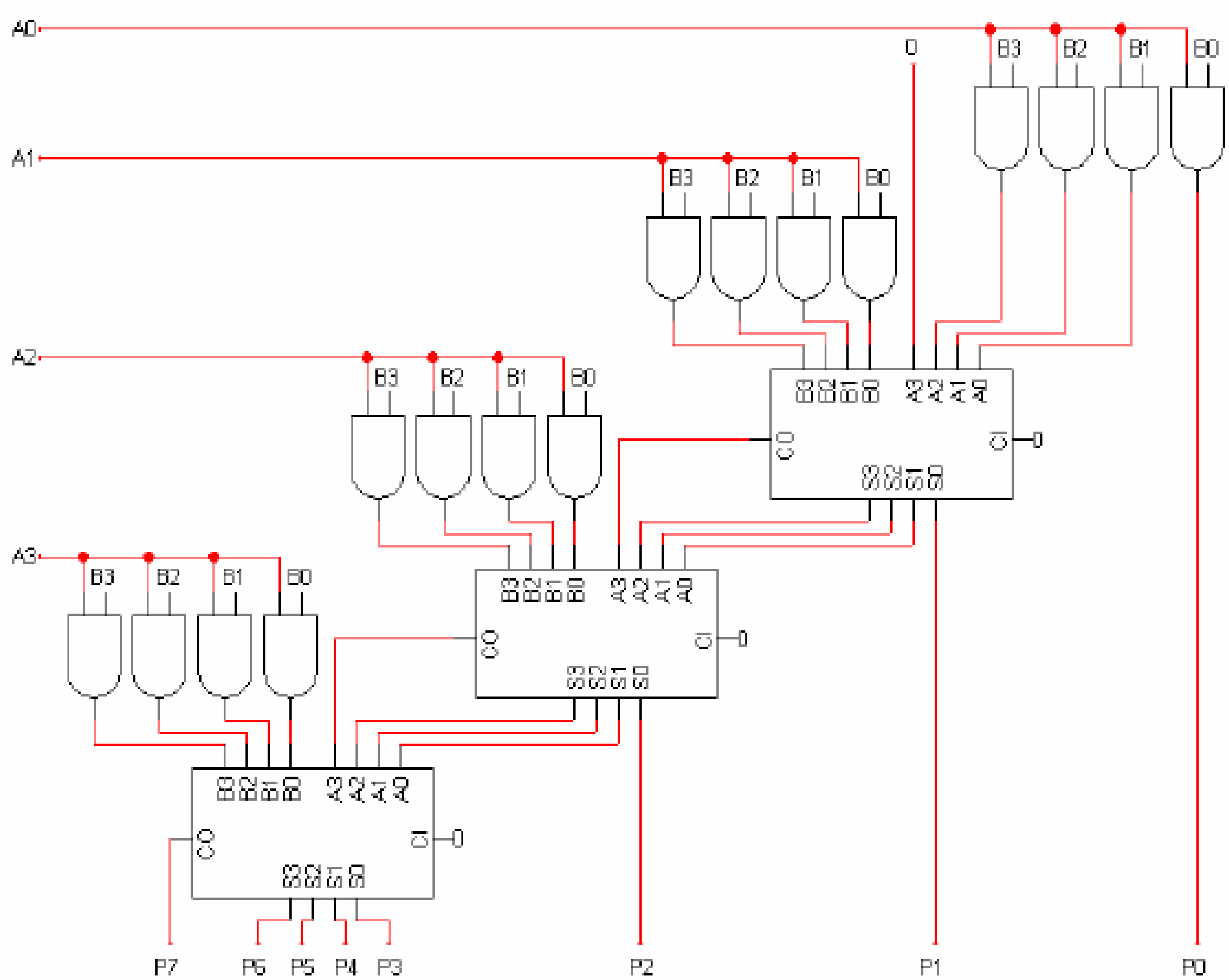
A	B	A×B	A•B
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	1

A 2×2 binary multiplier

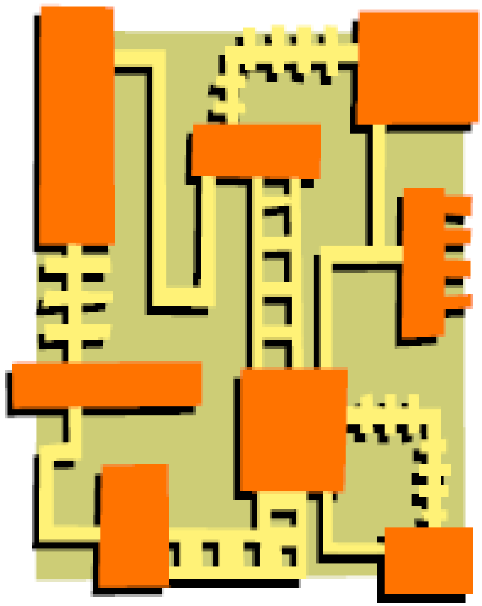
- Here is a circuit that multiplies the two-bit numbers A1A0 and B1B0, resulting in the four-bit product P3-P0.
- For a 2×2 multiplier we can just use two half adders to sum the partial products. In general, though, we'll need full adders.
- The diagram on the next page shows how this can be extended to a four-bit multiplier, taking inputs A3-A0 and B3-B0 and outputting the product P7-P0.



A 4×4 binary multiplier



Complexity of multiplication circuits



- In general, when multiplying an *m*-bit number by an *n*-bit number:
  - There will be *n* partial products, one for each bit of the multiplier.
  - This requires *n*-1 adders, each of which can add *m* bits.
- The circuit for 32-bit or 64-bit multiplication would be huge!