Chapter 4 – Arithmetic Functions

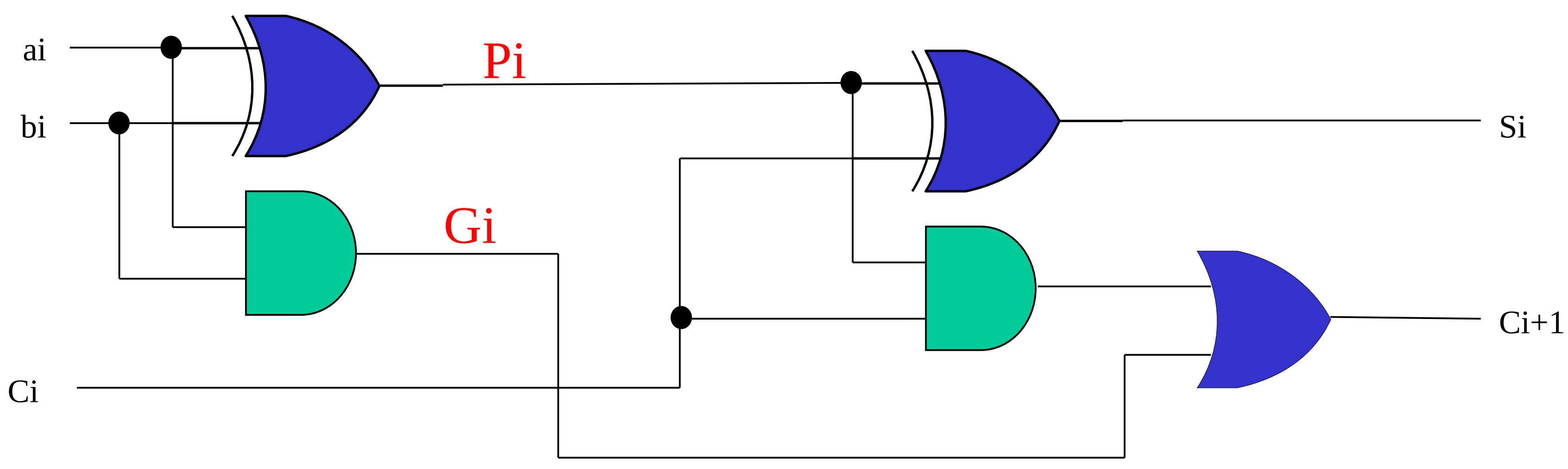
and Some Building Blocks

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, and**
  - ***Carry-Look-Ahead Adder*** **(CLA), a hierarchical structure to improve performance.**

Carry Propagation

- **What is the total delay of 4-bit ripple carry adder?**
  - **τ_{FA}: delay of a one full adder**
  - **Serial connected 4 full adders are used.**
  - **Total delay: $4\tau_{FA}$.**



$$4\tau_{FA} \approx 8\tau_{XOR}$$

Faster Adders

- **The carry propagation technique is a limiting factor in the speed with which two numbers are added.**
- **Two alternatives**
  - **use faster gates with reduced delays**
  - **Increase the circuit complexity (i.e. put more gates) in such a way that the carry delay time is reduced.**
- **An example for the latter type of solution is carry lookahead adders**
  - **Two binary variables:**
    1. $P_i = a_i \oplus b_i$ – **carry propagate**
    2. $G_i = a_i b_i$ – **carry generate**

Carry Lookahead Adders

- **Sum and carry can be expressed in terms of P_i and G_i:**
  - $S_i = P_i \oplus C_i$
  - $C_{i+1} = G_i + P_i C_i$
- **Why the names (carry propagate and generate)?**
  - **If $G_i = 1$ (both $a_i = b_i = 1$), then a "new" carry is generated**
  - **If $P_i = 1$ (either $a_i = 1$ or $b_i = 1$), then a carry coming from the previous lower bit position is propagated to the next higher bit position**

4-bit Carry Lookahead Adder

- **We can use the carry propagate and carry generate signals to compute carry bits used in addition operation**
  - C_0 = input
  - $C_1 = G_0 + P_0C_0$
  - $C_2 = G_1 + P_1C_1$

    $= G_1 + P_1(G_0 + P_0C_0) = G_1 + P_1G_0 + P_1P_0C_0$
  - $C_3 = G_2 + P_2C_2 = G_2 + P_2(G_1 + P_1G_0 + P_1P_0C_0)$

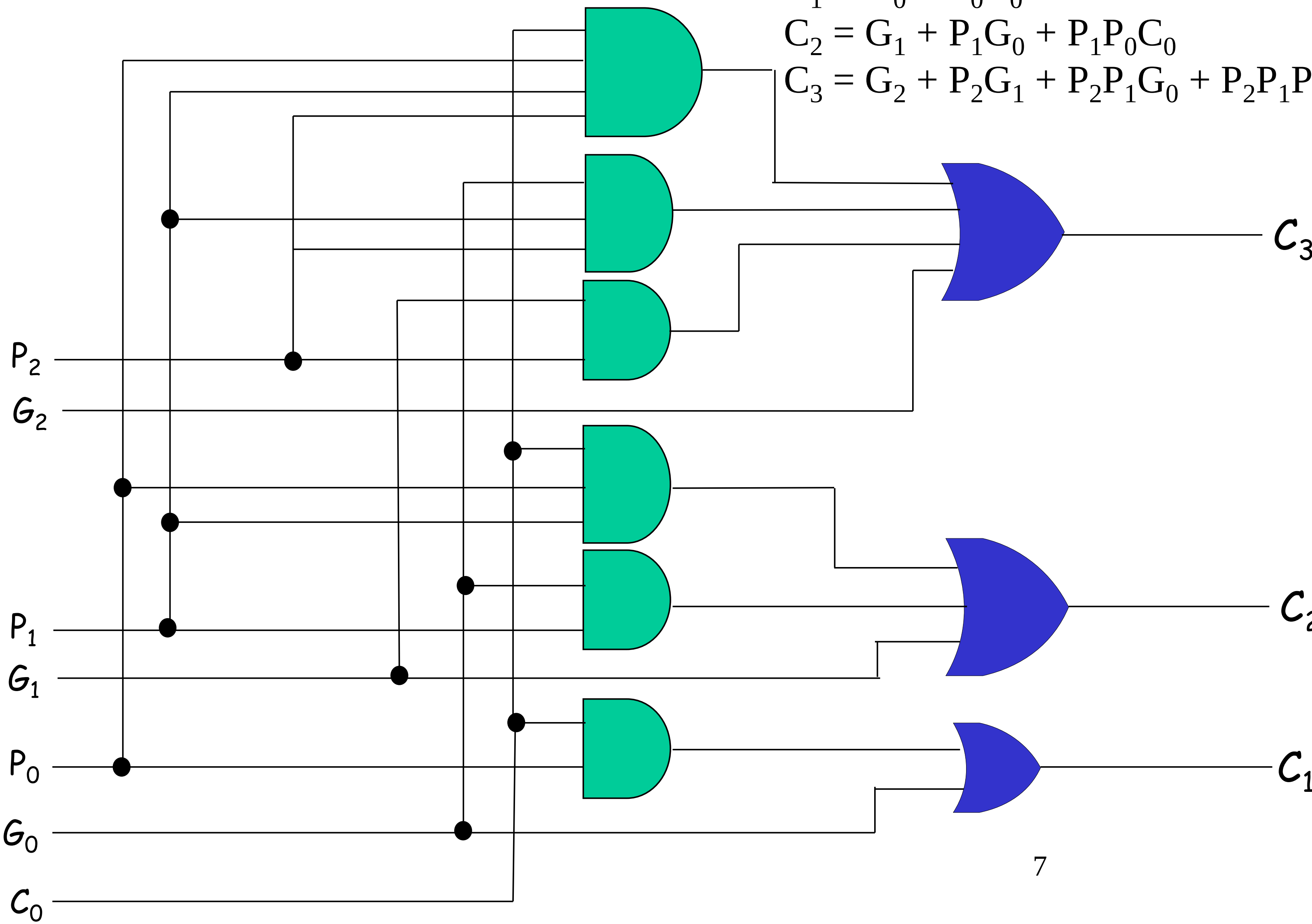
    $= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$
  - $P_0 = a_0 \oplus b_0$ and $G_0 = a_0b_0$
  - $P_1 = a_1 \oplus b_1$ and $G_1 = a_1b_1$
  - $P_2 = a_2 \oplus b_2$ and $G_2 = a_2b_2$
  - $P_3 = a_3 \oplus b_3$ and $G_3 = a_3b_3$

4-bit Carry Lookahead Circuit 1/3

$C_1 = G_0 + P_0C_0$

$C_2 = G_1 + P_1G_0 + P_1P_0C_0$

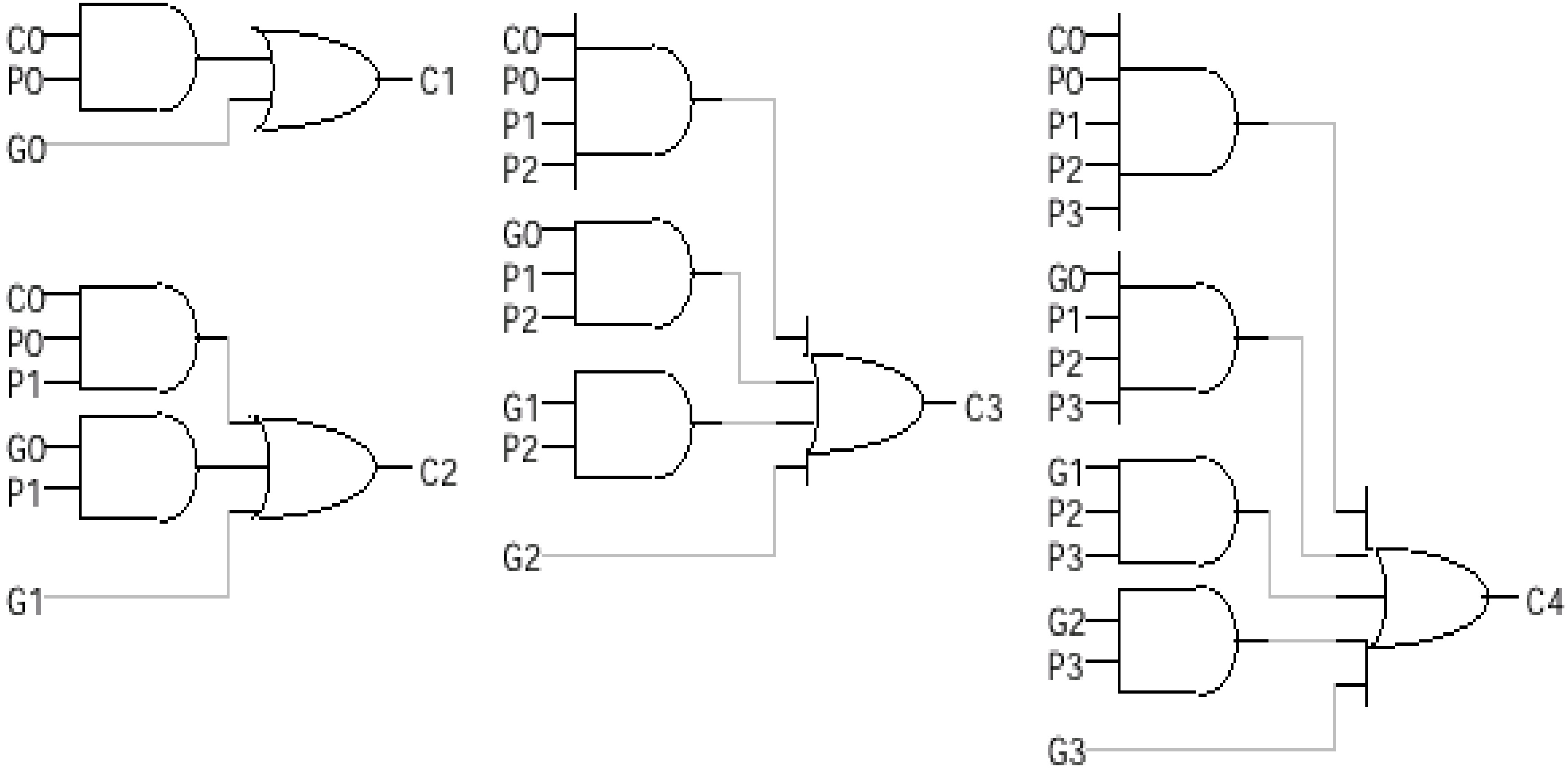
$C_3 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$



4-bit Carry Lookahead Circuit 2/3

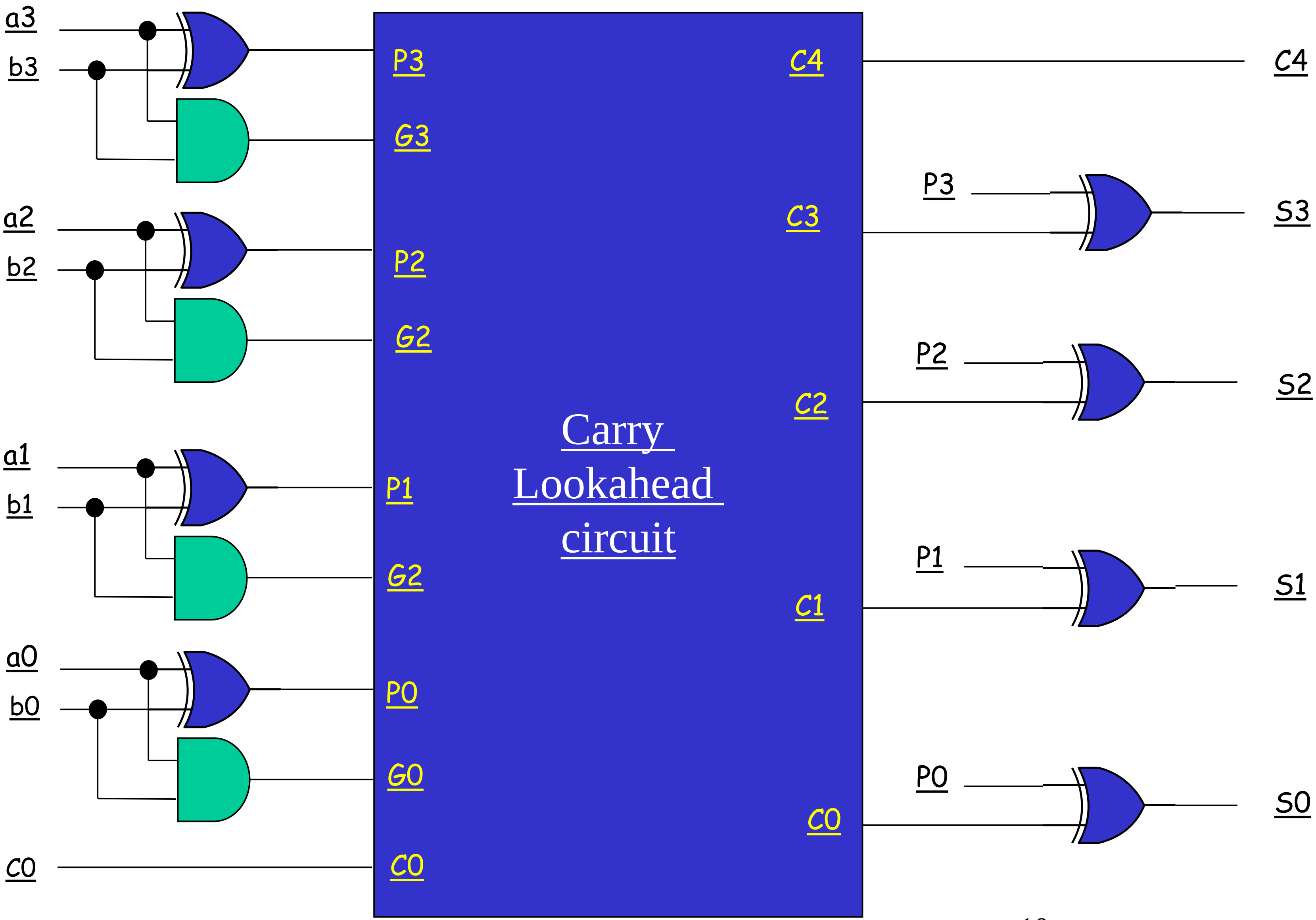
- **All three carries (C_1, C_2, C_3) can be realized as two-level implementation (i.e. AND-OR)**
- **C_3 does not have to wait for C_2 and C_1 to propagate**
- **C_3 has its own circuit**
- **The propagations happen concurrently**

4-bit Carry Lookahead Circuit 3/3

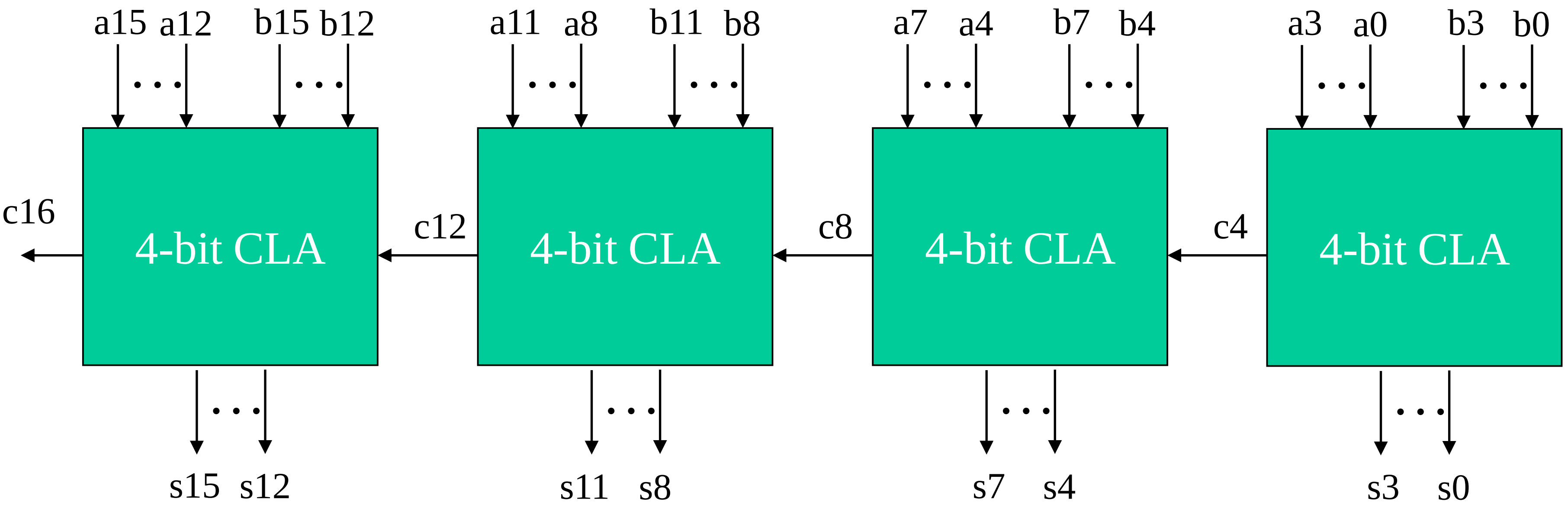


- **Two levels of logic**

4-bit Carry Lookahead Adder



Hybrid Approach for 16-bit Adder



Representation of Negative Numbers

- **In order to differ between positive and negative numbers the MSB is used.**
  - **If “0” positive**
  - **If “1” negative**
- **The positive numbers that can be shown by 8 bits are between 0000 0000 and 0111 1111, hence between 0 and + 127.**
- **2’s complement method is used for representation of negative numbers.**
  - **2’s complement of a positive number shows the negative of it.**
- **In order to find the 2’s complement of a number**
  - **1’s complement is found: 0s are changed to 1s, 1s are changed to 0s.**
  - **1 is added to 1’s complement of the number.**

Addition of Positive and Negative Numbers Represented By 2's Complement

Carry	0 0 1 0		0 1 0 0	
X	**1101**	**-3**	**0011**	**3**
Y	**+0001**	**+1**	**+0010**	**+2**
Sum	1 1 1 0	**-2**	0 1 0 1	**5**
	Negative		Positive	

Addition of Positive and Negative Numbers Represented By 2's Complement

Carry	11110		11100	
X	**1101**	**-3**	**0011**	**3**
Y	**+1111**	**-1**	**+1110**	**-2**
Sum	11100	**-4**	10001	**1**

negative

ignored

positive

ignored

Addition of Positive and Negative Numbers Represented By 2's Complement

Carry	**1000**		**0000**	
X	**0100**	**4**	**1010**	**-6**
Y	**+0101**	**+5**	**+1101**	**-3**
Sum	**1001**	**9**	**10111**	**-9**

Is the sum negative?

ignored

Is the sum positive?

- Overflow occured. The largest positive number that can be represented by 4-bits is +7. Larger numbers can not be reprsented by 4-bits.
- The smallest negative number that can be represented by 4-bits is -8. Smaller numbers can not be reprsented by 4-bits.
- The number of bits to be used in the representation of the numbers should be decided according to the boundaries of the inputs and the outputs of the operations.

Subtraction of Numbers With Sign Bit and 2's complement

X	3	0011		0011	3
Y	-1	-0001	2's complement	+1111	+(-1)
Difference	2			10010	2

ignored

positive

X	3	0011		0011	3
Y	-4	-0100	2's complement	+1100	+(-4)
Difference	-1			1111	-1

negative

X	3	0011		0011	3
Y	-(-1)	-1111	2's complement	+0001	+1
Difference	4			0100	4

positive

Subtraction of Numbers With Sign Bit and 2's complement

X	1	0001		0001	1
Y	-(-7)	-1001	2's complement	+0111	+7
Difference	8			1000	8
X	-5	1011		1011	-5
Y	-4	-0100	2's complement	+1100	+(-4)
Difference	-9			10111	-9

Is the result negative?

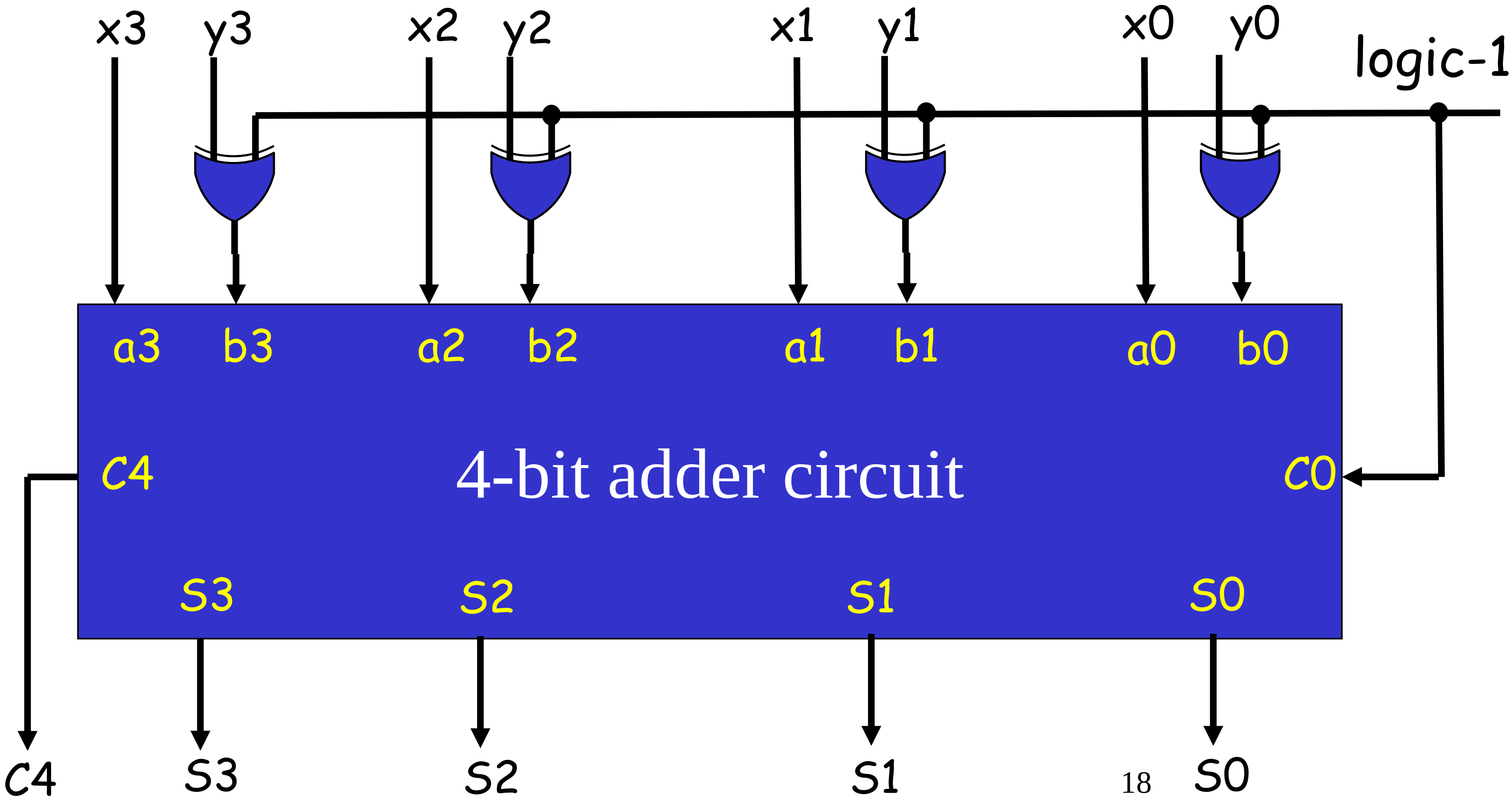
ignored

Is the result positive?

- Overflow occured. The largest positive number that can be represented by 4-bits is +7. Larger numbers can not be represented by 4-bits.
- The smallest negative number that can be represented by 4-bits is -8. Smaller numbers can not be reprsented by 4-bits.
- The number of bits to be used in the representation of the numbers should be decided according to the boundaries of the inputs and the outputs of the operations.

Subtractor

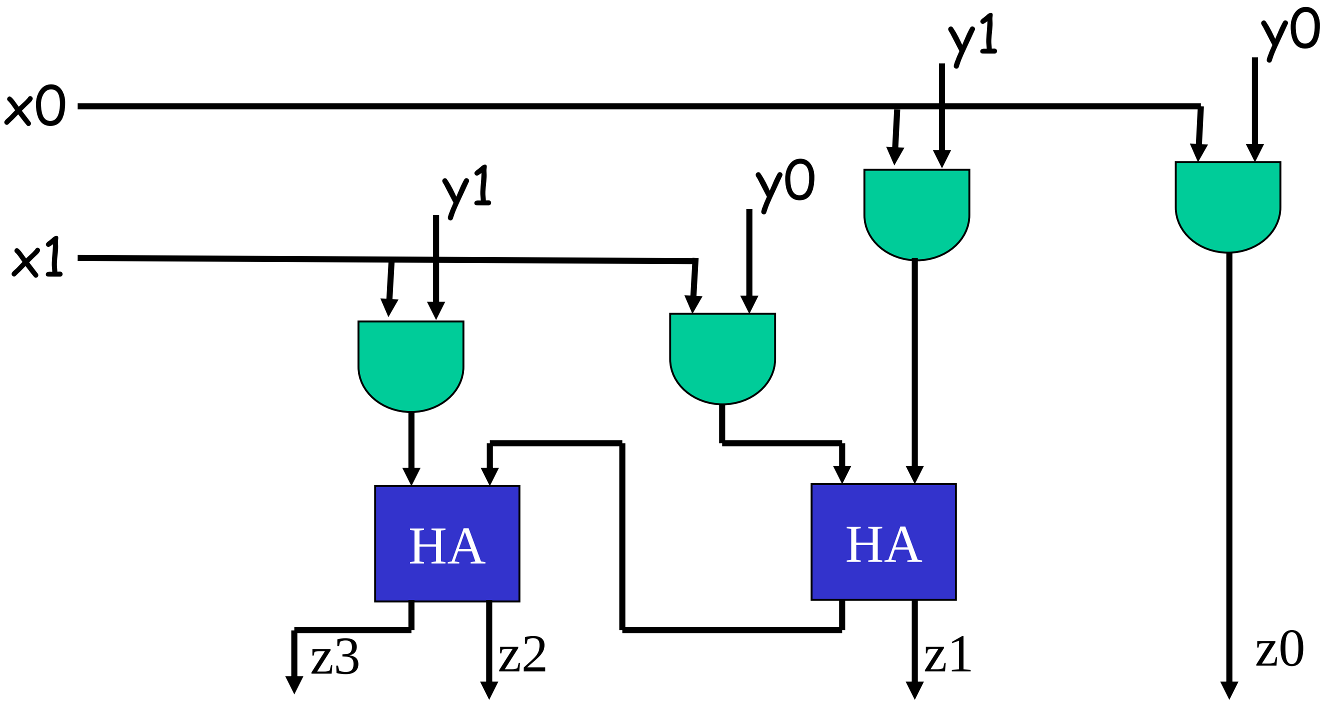
- **Recall how we do subtraction (2's complement)**
  - **X – Y = X + (2n – Y) = X + ~Y + 1**



Binary Multipliers

- **Two-bit multiplier**

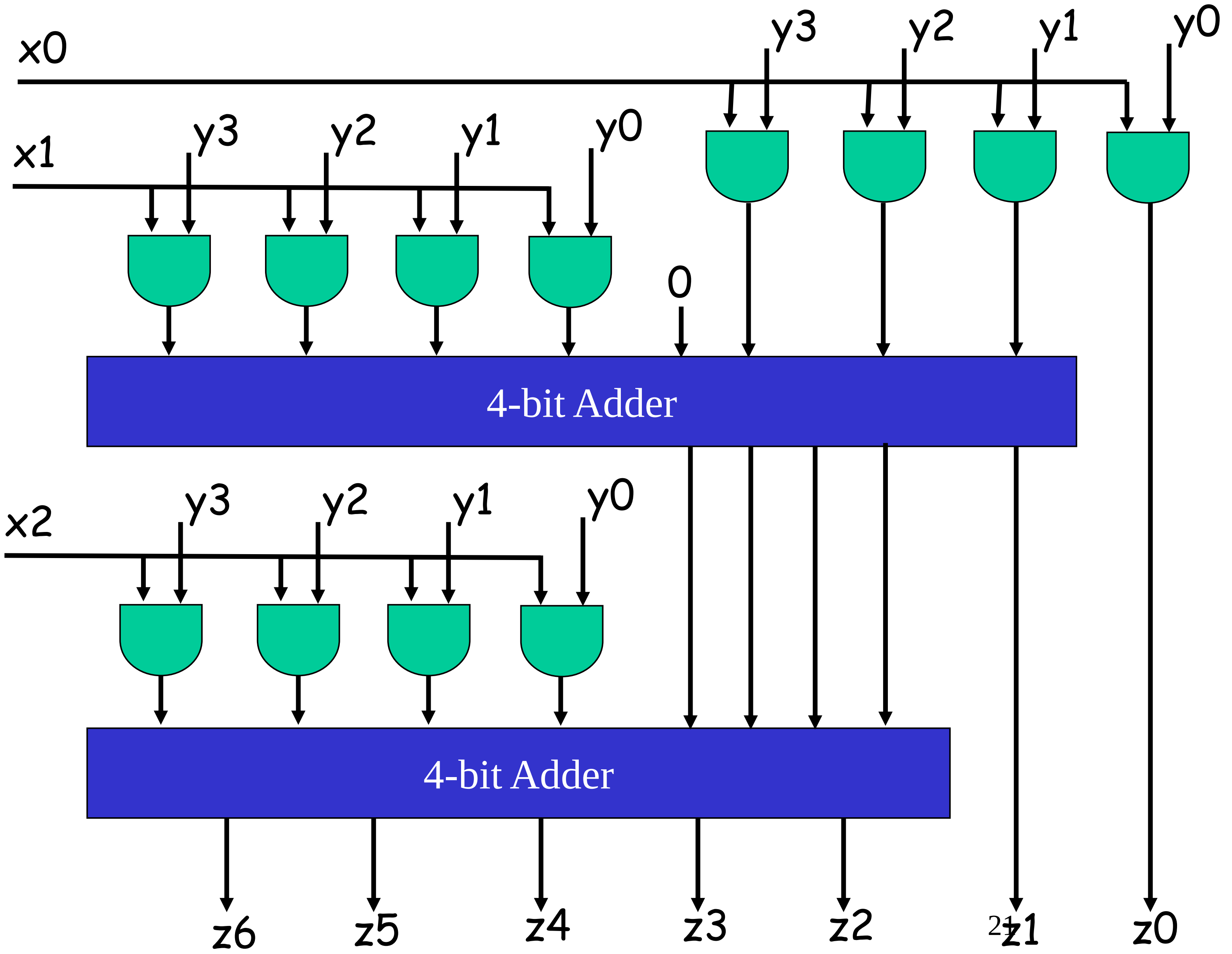
			y_1	y_0	Y
		$\times$	x_1	x_0	X
			$x_0\ y_1$	$x_0\ y_0$	
+		$x_1\ y_1$	$x_1\ y_0$		
	z_3	z_2	z_1	z_0	Z



(3x4)-bit Multiplier: Method

$$
\begin{array}{ccccccccc}
 & & & & y_3 & y_2 & y_1 & y_0 & Y \\
 & & & \times & & x_2 & x_1 & x_0 & X \\
\cline{4-9}
 & & & & x_0\,y_3 & x_0\,y_2 & x_0\,y_1 & x_0\,y_0 & \\
 & & & x_1\,y_3 & x_1\,y_2 & x_1\,y_1 & x_1\,y_0 & & \\
+ & & x_2\,y_3 & x_2\,y_2 & x_2\,y_1 & x_2\,y_0 & & & \\
\cline{1-8}
 & z_6 & z_5 & z_4 & z_3 & z_2 & z_1 & z_0 &
\end{array}
$$

4-bit Multiplier: Circuit



mxn-bit Multipliers

- **Generalization:**
- **multiplier: m-bit integer**
- **multiplicand: n-bit integers**
- **m×n AND gates**
- **(m-1) adders**
  - **each adder is n-bit**

Magnitude Comparator

- **Comparison of two integers: A and B.**
  - **$A > B \Rightarrow (1, 0, 0) = (x, y, z)$**
  - **$A = B \Rightarrow (0, 1, 0) = (x, y, z)$**
  - **$A < B \Rightarrow (0, 0, 1) = (x, y, z)$**
- **Example: 4-bit magnitude comparator**
  - **$A = (a_3, a_2, a_1, a_0)$ and $B = (b_3, b_2, b_1, b_0)$**

1. **(A = B) case**
   - **they are equal if and only if $a_i = b_i \quad 0 \leq i \leq 3$**
   - **$t_i = (a_i \oplus b_i)' \qquad 0 \leq i \leq 3$**
   - **$y = (A=B) = t_3\, t_2\, t_1\, t_0$**

4-bit Magnitude Comparator

2. **(A > B) and (A < B) cases**
   - **We compare the most significant bits of A and B first.**
     - **if (a_3 = 1 and b_3 = 0) ➔ A > B**
     - **else if (a_3 = 0 and b_3 = 1) ➔ A < B**
     - **else (i.e. $a_3 = b_3$) compare a_2 and b_2.**

$$x = (A>B) = a_3b_3'+t_3a_2b_2'+t_3t_2a_1b_1'+t_3t_2t_1a_0b_0'$$

$$z = (A<B) = a_3'b_3+t_3a_2'b_2+t_3t_2a_1'b_1+t_3t_2t_1a_0'b_0$$

$$y = (A=B) = t_3t_2t_1t_0$$

4-bit Magnitude Comparator: Circuit

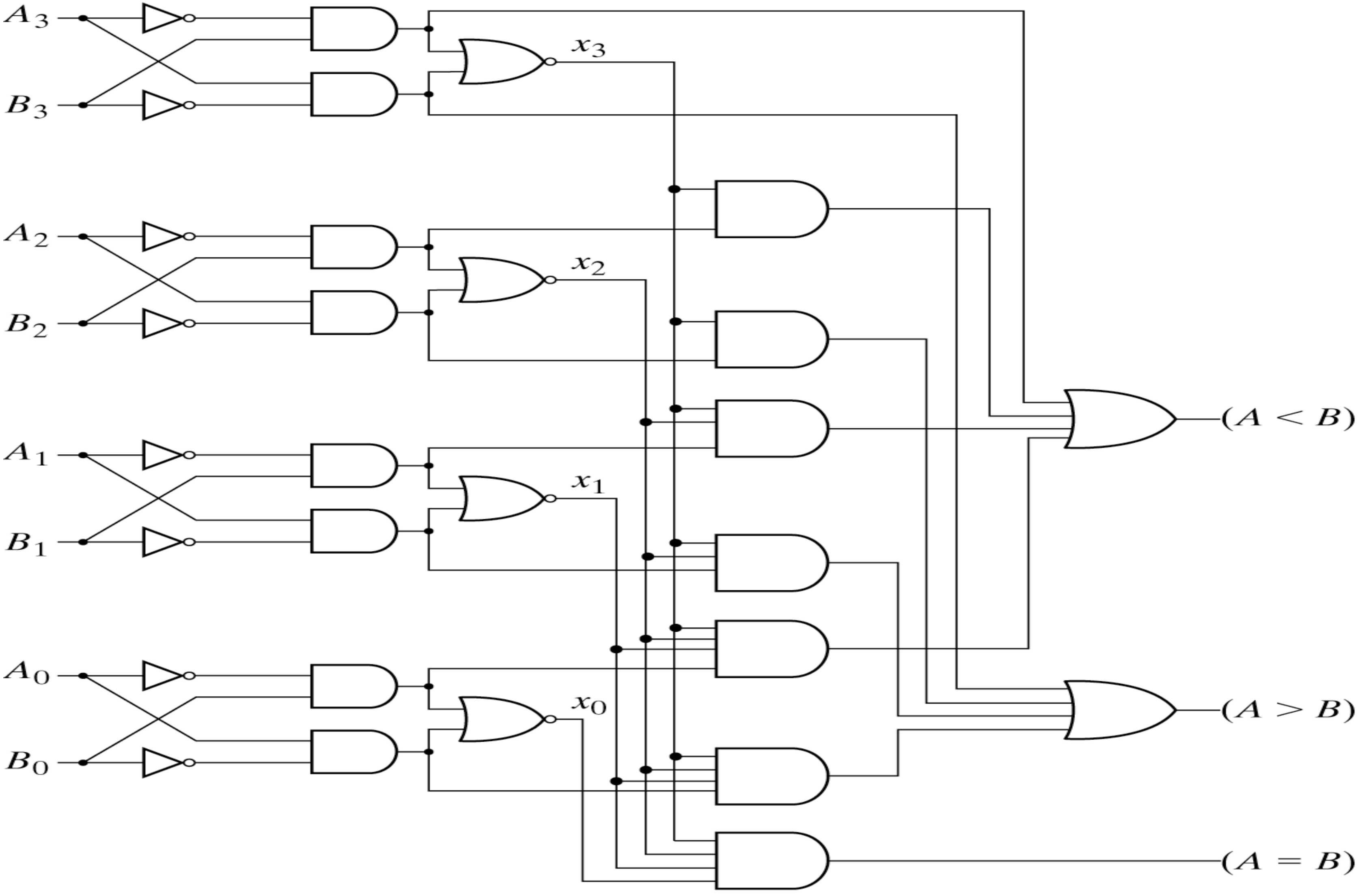
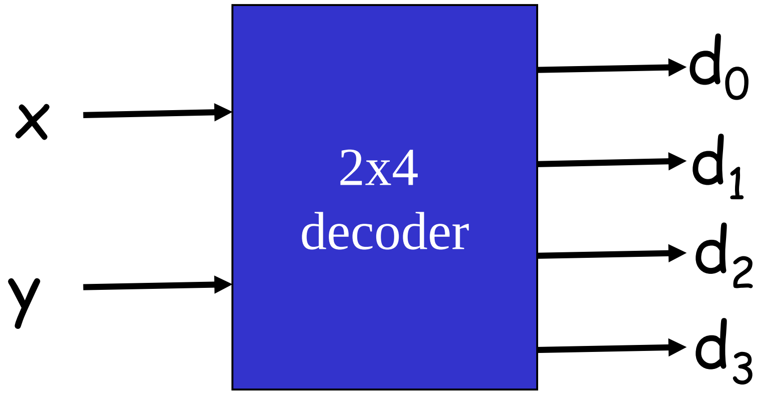


Fig. 4-17 4-Bit Magnitude Comparator

Decoders

- **A binary code of n bits**
  - **capable of representing 2^n distinct elements of coded information**
  - **A decoder is a combinational circuit that converts binary information from n binary inputs to a maximum of 2^n unique output lines**

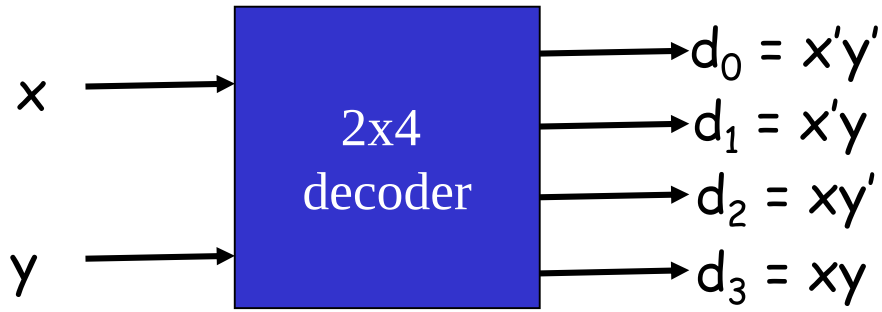


x	y	d_0	d_1	d_2	d_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

- $d_0 =$
- $d_1 =$
- $d_2 =$
- $d_3 =$

Decoder as a Building Block

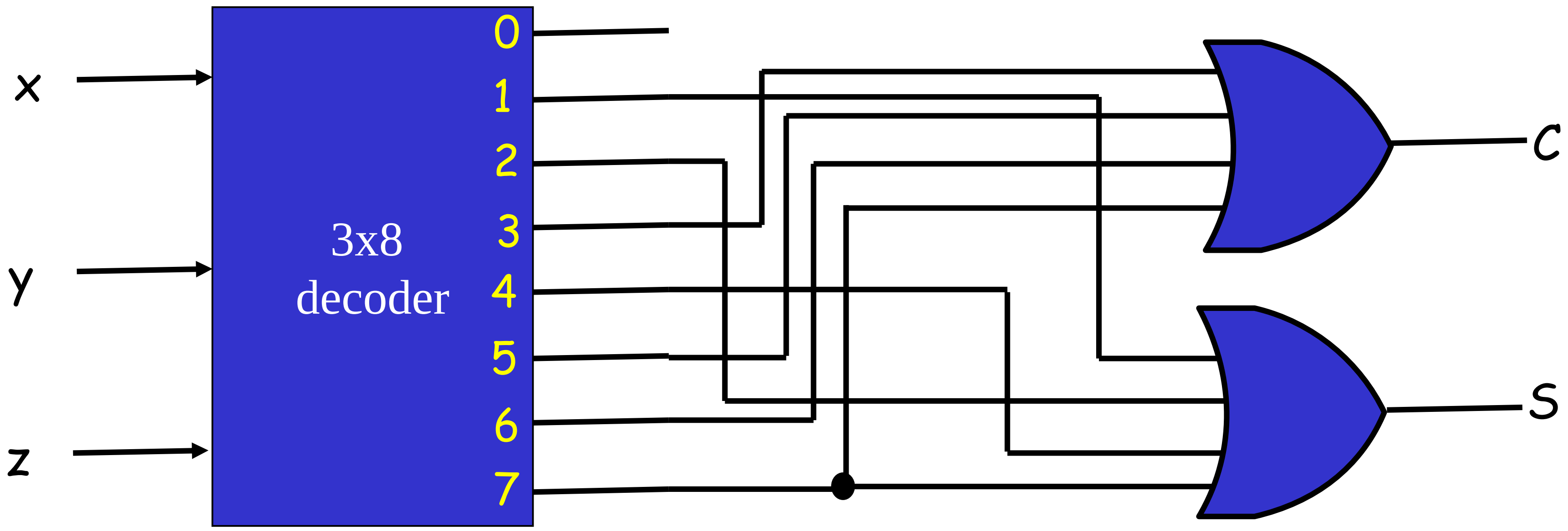
- **A decoder provides the 2^n minterms of n input variable**



- We can use a decoder and OR gates to realize any Boolean function expressed as sum of minterms
  - Any circuit with n inputs and m outputs can be realized using an n-to-2^n decoder and m OR gates.

Example: Decoder as a Building Block

- **Full adder**
  - **C = xy + xz + yz = Σ(3, 5, 6, 7)**
  - **S = x ⊕ y ⊕ z = Σ(1, 2, 4, 7)**



Encoders

- **An encoder is a combinational circuit that performs the inverse operation of a decoder**
  - **number of inputs: 2^n**
  - **number of outputs: n**
  - **the output lines generate the binary code corresponding to the input value**
- **Example: n = 2**

d_0	d_1	d_2	d_3	x	y
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1

Priority Encoder

- **Problem with a regular encoder:**
  - **only one input can be active at any given time**
  - **the output is undefined for the case when more than one input is active simultaneously.**
- **Priority encoder:**
  - **there is a priority among the inputs**

d_0	d_1	d_2	d_3	x	y	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

4-bit Priority Encoder

- **In the truth table**
  - **X for input variables represents both 0 and 1.**
  - **Good for condensing the truth table**
  - **Example: X100 → (0100, 1100)**
    - **This means d_1 has priority over d_0**
    - **d_3 has the highest priority**
    - **d_2 has the next**
    - **d_0 has the lowest priority**
  - **V = ?**

Maps for 4-bit Priority Encoder

d_0d_1 \ d_2d_3	00	01	11	10
00	X	1	1	1
01	0	1	1	1
11	0	1	1	1
10	0	1	1	1

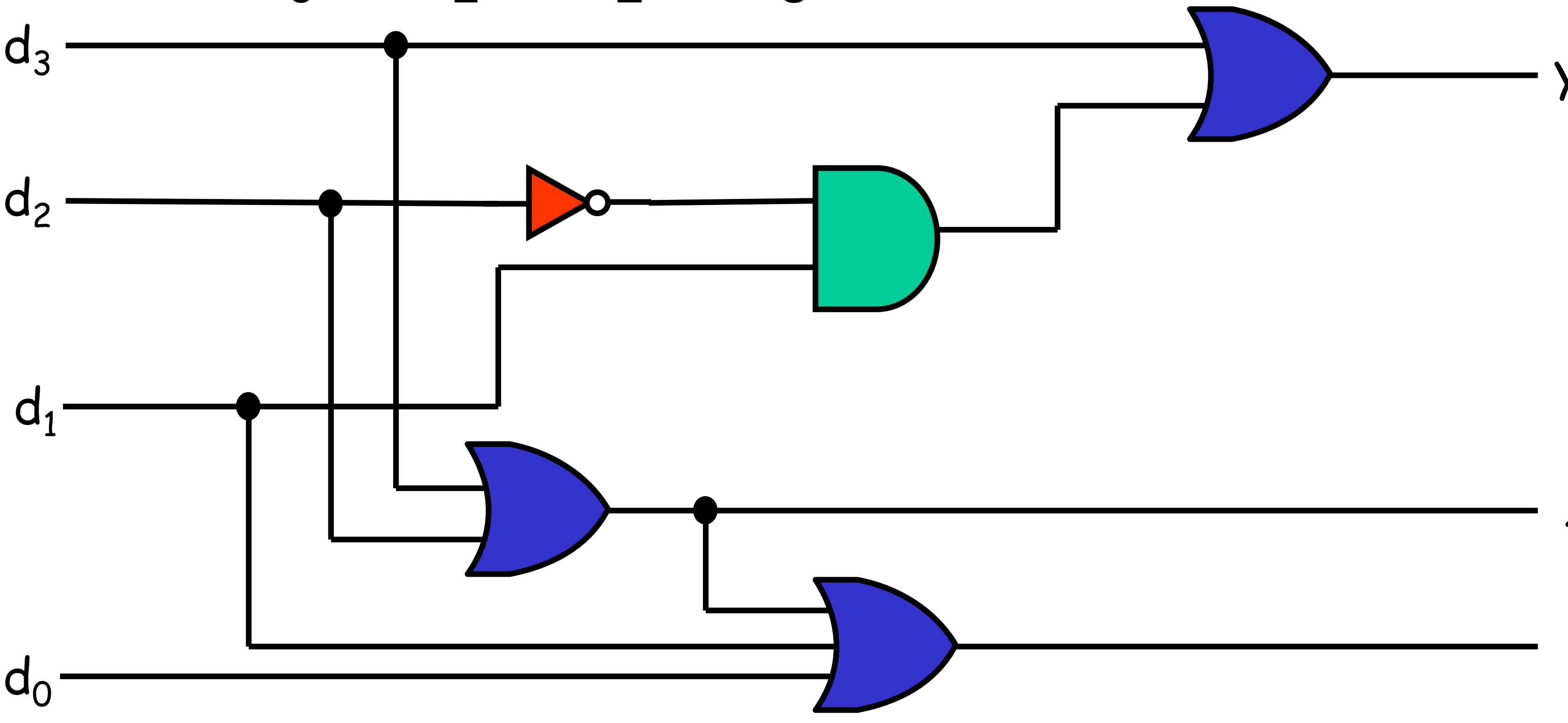
– x =

d_0d_1 \ d_2d_3	00	01	11	10
00	X	1	1	0
01	1	1	1	0
11	1	1	1	0
10	0	1	1	0

– y =

4-bit Priority Encoder: Circuit

- $x = d_2 + d_3$
- $y = d_1 d_2' + d_3$
- $V = d_0 + d_1 + d_2 + d_3$



Multiplexers

- **A combinational circuit**
  - **It selects binary information from one of the many input lines and directs it to a single output line.**
  - **Many inputs – m**
  - **One output line**
  - **selection lines n → n = ?**
- **Example: 2-to-1-line multiplexer**
  - **2 input lines I_0, I_1**
  - **1 output line Y**
  - **1 select line S**

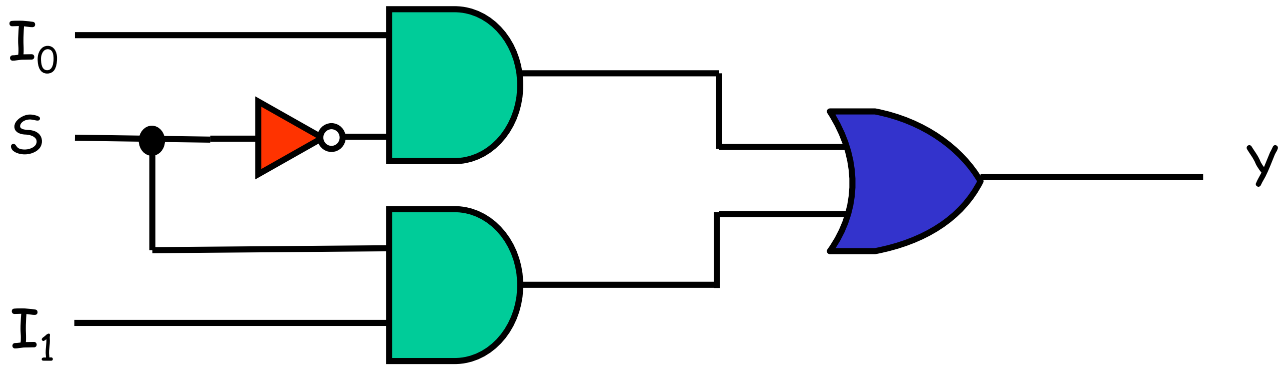
    Y = ?

S	Y
0	I_0
1	I_1

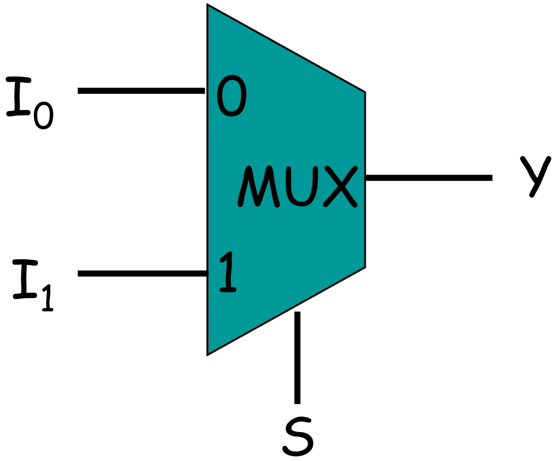
Function Table

2-to-1-Line Multiplexer

Y = ?



- **Special Symbol**



4-to-1-Line Multiplexer

- **4 input lines: I_0, I_1, I_2, I_3**
- **1 output line: Y**
- **2 select lines: S_1, S_0.**

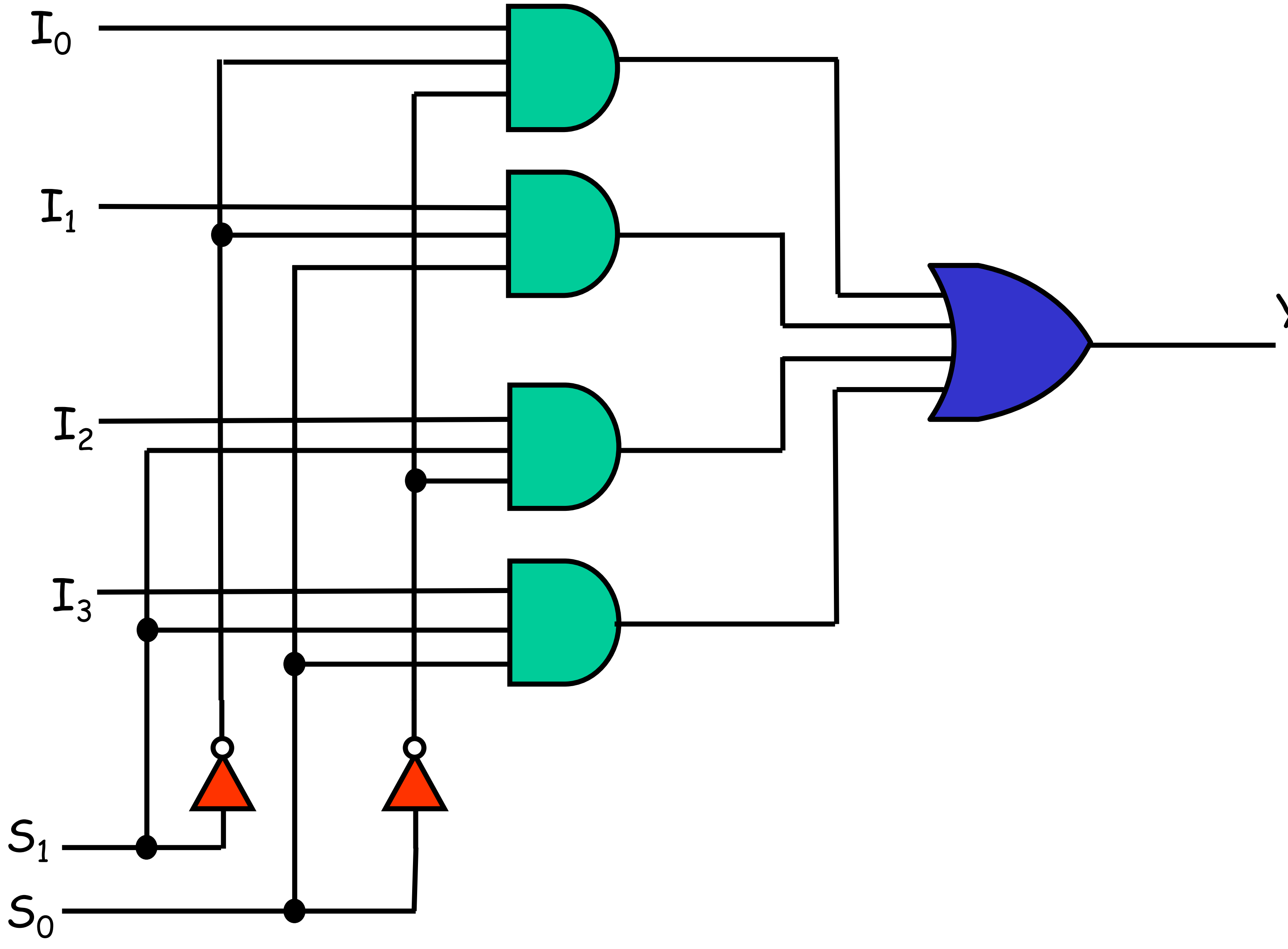
S_1	S_0	Y
0	0	?
0	1	?
1	0	?
1	1	?

Y = ?

Interpretation:

- In case $S_1 = 0$ and $S_0 = 0$, Y selects I_0
- In case $S_1 = 0$ and $S_0 = 1$, Y selects I_1
- In case $S_1 = 1$ and $S_0 = 0$, Y selects I_2
- In case $S_1 = 1$ and $S_0 = 1$, Y selects I_3

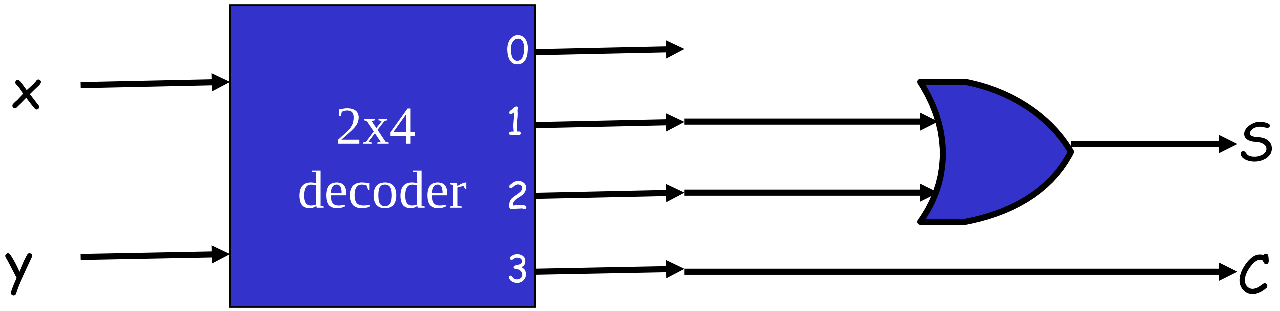
4-to-1-Line Multiplexer: Circuit



Design with Multiplexers 1/2

- **Reminder: design with decoders**

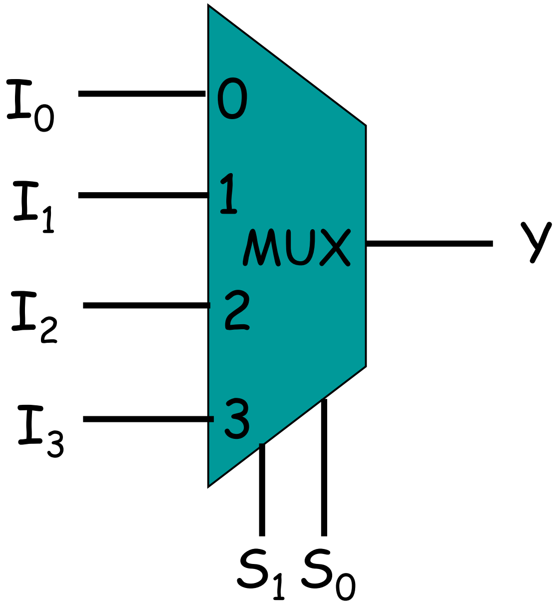
- Half adder
  - $C = xy = \Sigma$
  - $S = x \oplus y = x'y + xy' = \Sigma$



- A closer look will reveal that a multiplexer is nothing but a decoder with OR gates

Design with Multiplexers 2/2

- **4-to-1-line multiplexer**

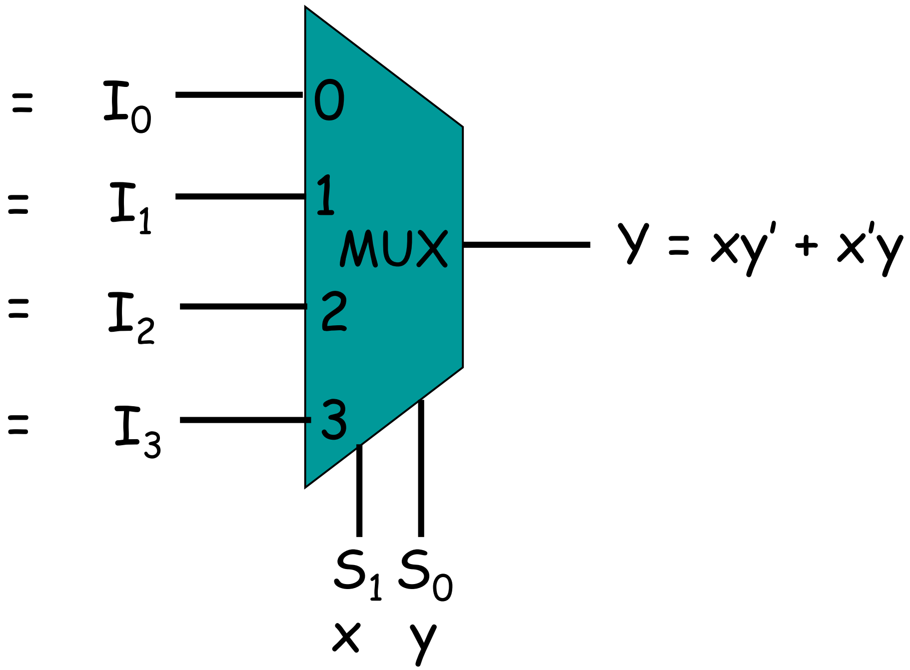


- $S_1 \rightarrow x$
- $S_0 \rightarrow y$
- $S_1'S_0' = x'y'$,
- $S_1'S_0 = x'y$,
- $S_1S_0' = xy'$,
- $S_1S_0 = xy$

- $Y = S_1'S_0' I_0 + S_1'S_0 I_1 + S_1S_0' I_2 + S_1S_0 I_3$.
- $Y = x'y' I_0 + x'y I_1 + xy' I_2 + xyI_3$
- $Y =$

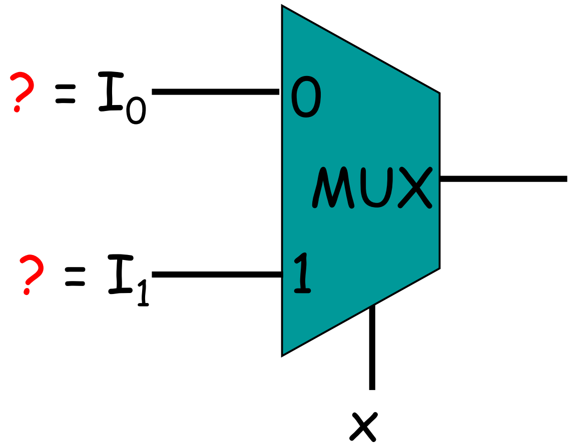
Example: Design with Multiplexers

- **Example: S = Σ(1, 2)**



Design with Multiplexers Efficiently

- **More efficient way to implement an n-variable Boolean function**
  1. **Use a multiplexer with n-1 selection inputs**
  2. **First (n-1) variables are connected to the selection inputs**
  3. **The remaining variable is connected to data inputs**
- **Example: $Y = \Sigma(1, 2)$**



$Y = S' I_0 + S I_1$

$Y = x' I_0 + x I_1$

Example: Design with Multiplexers

- **$F(x, y, z) = \Sigma(1, 2, 6, 7)$**
  - **$F = x'y'z + x'yz' + xyz' + xyz$**
  - **$Y = S_1'S_0' I_0 + S_1'S_0 I_1 + S_1S_0' I_2 + S_1S_0 I_3$**
  - **$I_0 = z, I_1 = z', I_2 = 0, I_3 = 1.$**

x	y	z	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

F =

F =

F =

F =

Example: Design with Multiplexers

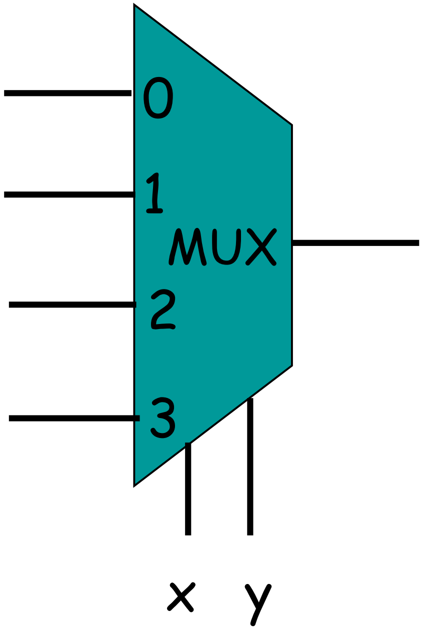
$F = x'y'z + x'yz' + xyz' + xyz$

$F = z$ when $x = 0$ and $y = 0$

$F = z'$ when $x = 0$ and $y = 1$

$F = 0$ when $x = 1$ and $y = 0$

$F = 1$ when $x = 1$ and $y = 1$



$F = x'y'z + x'yz' + xy$

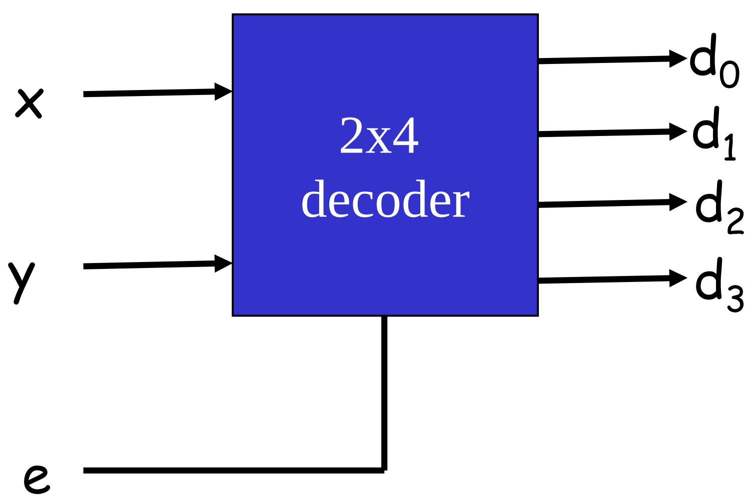
Design with Multiplexers

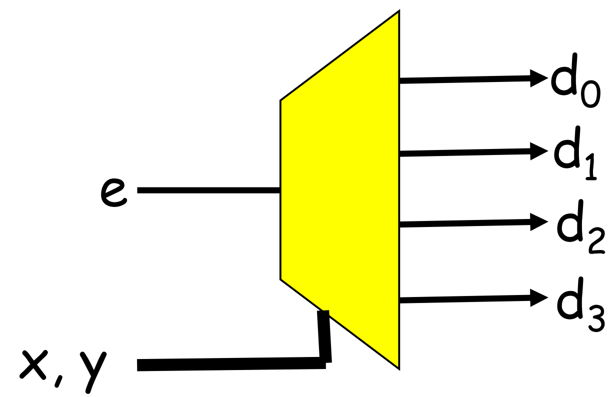
- **General procedure for n-variable Boolean function**
  - **$F(x_1, x_2, ..., x_n)$**

1. **The Boolean function is expressed in a truth table**
2. **The first (n-1) variables are applied to the selection inputs of the multiplexer ($x_1, x_2, ..., x_{n-1}$)**
3. **For each combination of these (n-1) variables, evaluate the value of the output as a function of the last variable, x_n.**
   - **$0, 1, x_n, x_n'$**
4. **These values are applied to the data inputs in the proper order.**

Decoder/Demultiplexer

- **A demultiplexer is a combinational circuit**
  - **it receives information from a single line and directs it one of 2^n output lines**
  - **It has n selection lines as to which output will get the input**





$d_0 = e$ when $x = 0$ and $y = 0$

$d_1 = e$ when $x = 0$ and $y = 1$

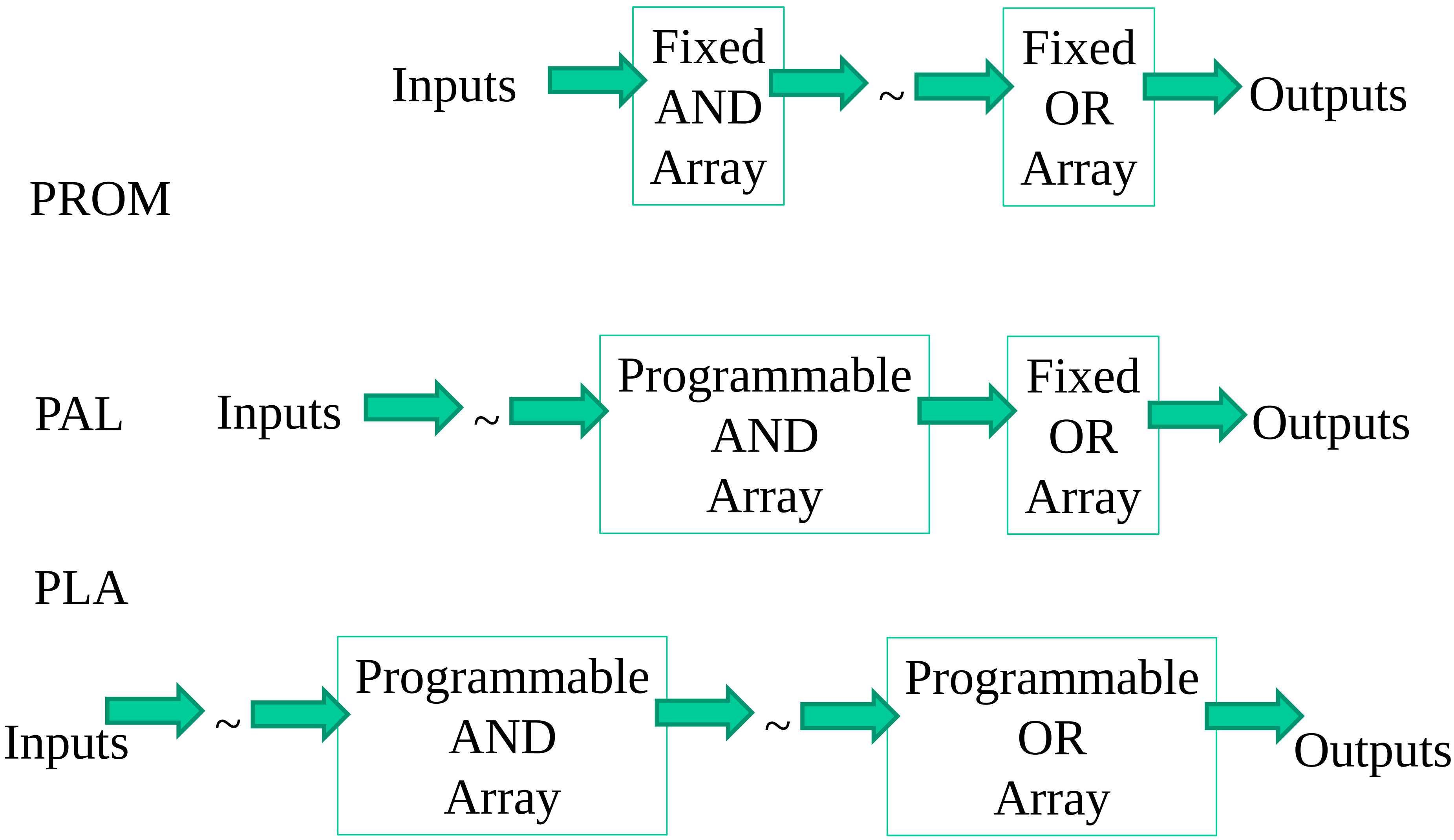
$d_2 = e$ when $x = 1$ and $y = 0$

$d_3 = e$ when $x = 1$ and $y = 1$

Programmable Logic Devices (PLD's)

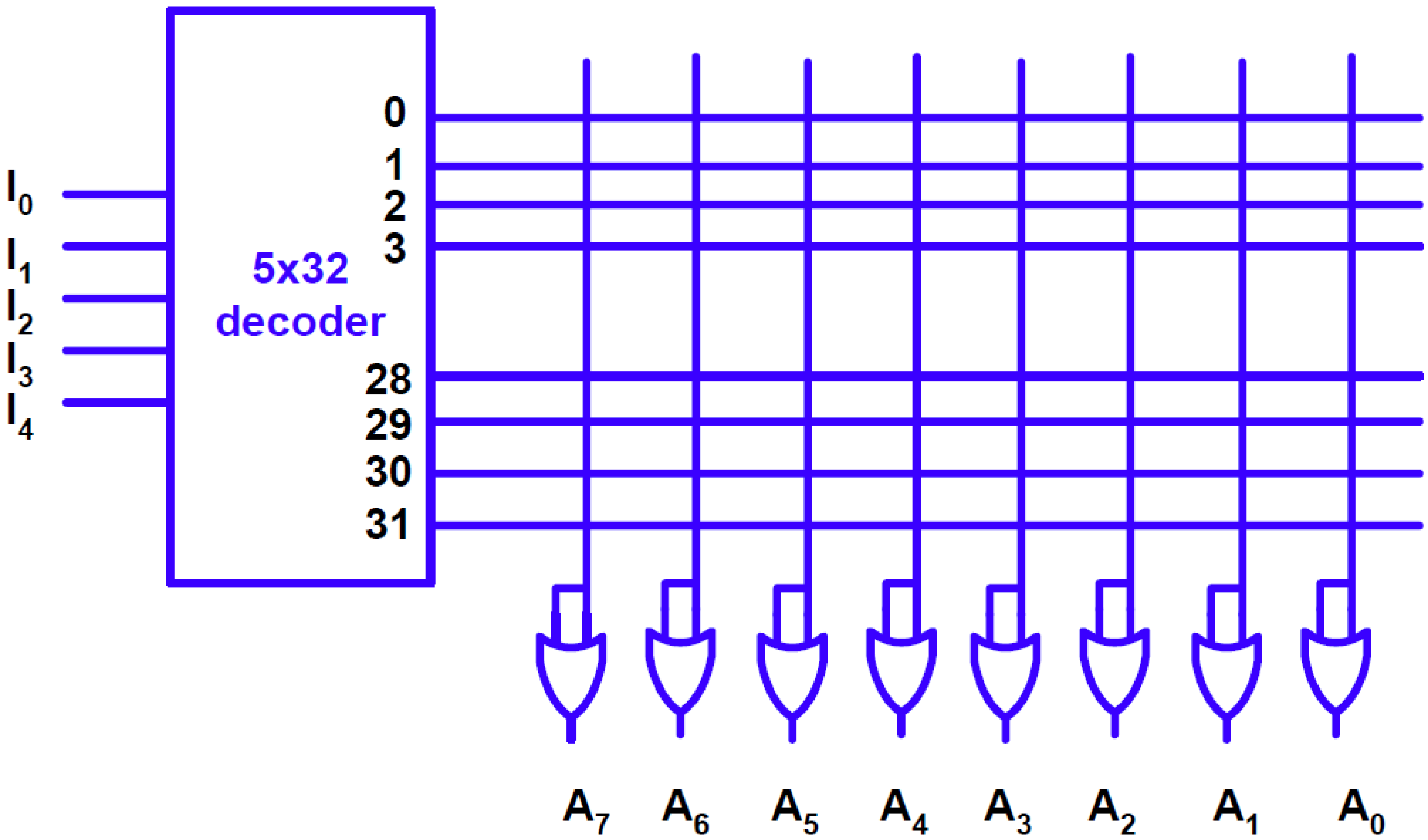
- **Programmable Logic Devices are formed by AND and OR arrays. The gate arrays are programmed by using switches in order implement a special Boolean function.**
- **We will discuss three PLDs in this course.**
  1. **Programmable Read Only Memory (PROM)**
  2. **Programmable Logic Array (PLA)**
  3. **Programmable Array Logic (PAL)**

Programmable Logic Devices



Read Only Memory (ROM)

- **ROM is a device which can store binary information and keep it even when the power is cut.**
- **ROM contains a decoder and a fixed OR array.**



Architecture of a 32x8-bit ROM

Combinational Circuit Design by Using ROM

- **It is direct implementation of a Boolean function.**
  - **There is no need to optimize the Boolean function. It produces all the minterms.**
- **Reprogramme gives the chance to implement different Boolean funcions on the same device.**

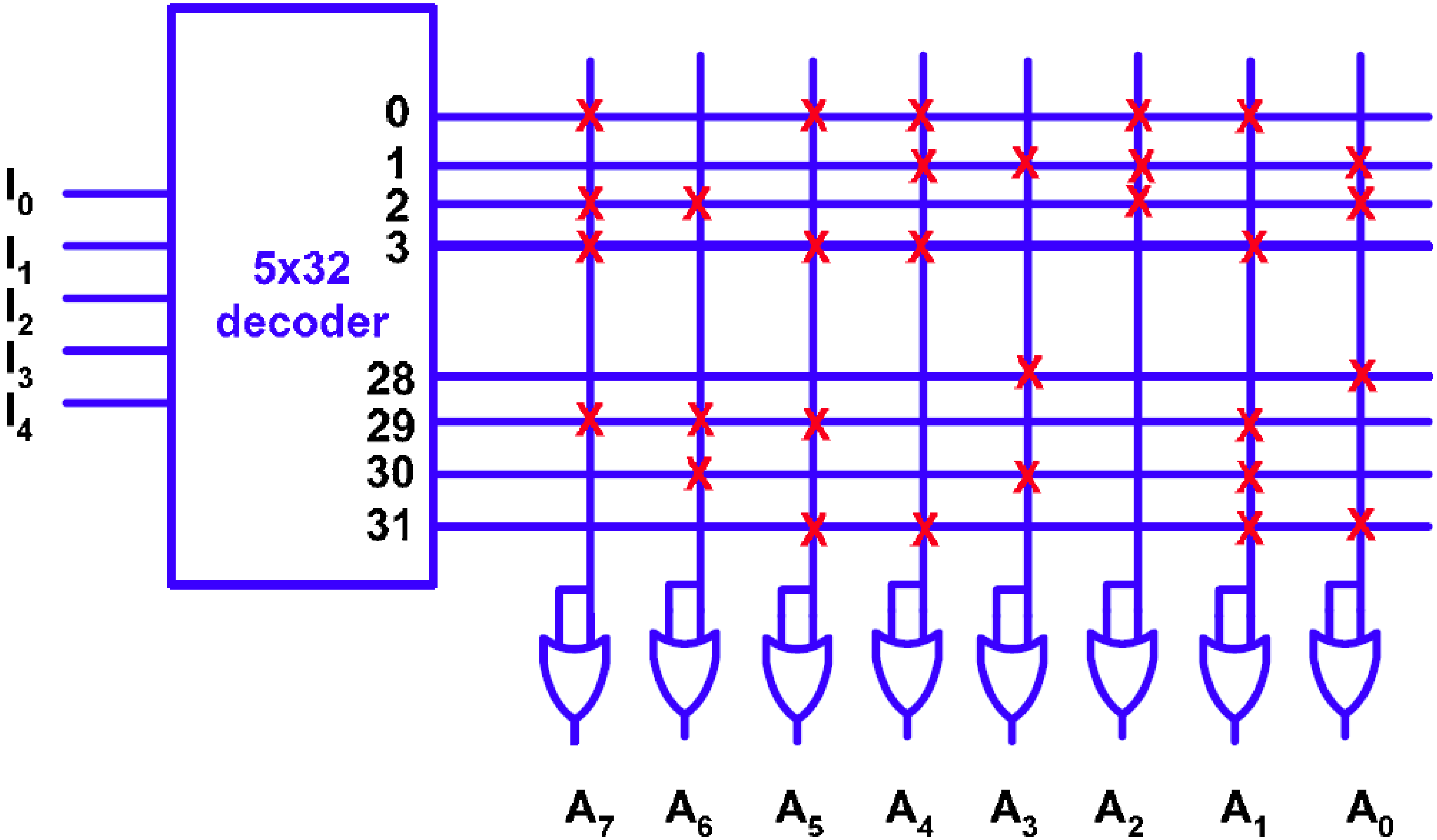
Design with ROM

- **The truth table of the Boolean function shows the positions of the switches that are closed.**

Inputs					**Outputs**							
I_4	I_3	I_2	I_1	I_0	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
0	0	0	0	0	1	0	1	1	0	1	1	0
0	0	0	0	1	0	0	0	1	1	1	0	1
0	0	0	1	0	1	1	0	0	0	1	0	1
0	0	0	1	1	1	0	1	1	0	0	1	0
…					…							
1	1	1	0	0	0	0	0	0	1	0	0	1
1	1	1	0	1	1	1	1	0	0	0	1	0
1	1	1	1	0	0	1	0	0	1	0	1	0
1	1	1	1	1	0	0	1	1	0	0	1	1

Design with ROM

- X shows that there is connection. Hence X shows logic-1.
- If there is no X, then there is no connection. Hence absense of X means logic-0.



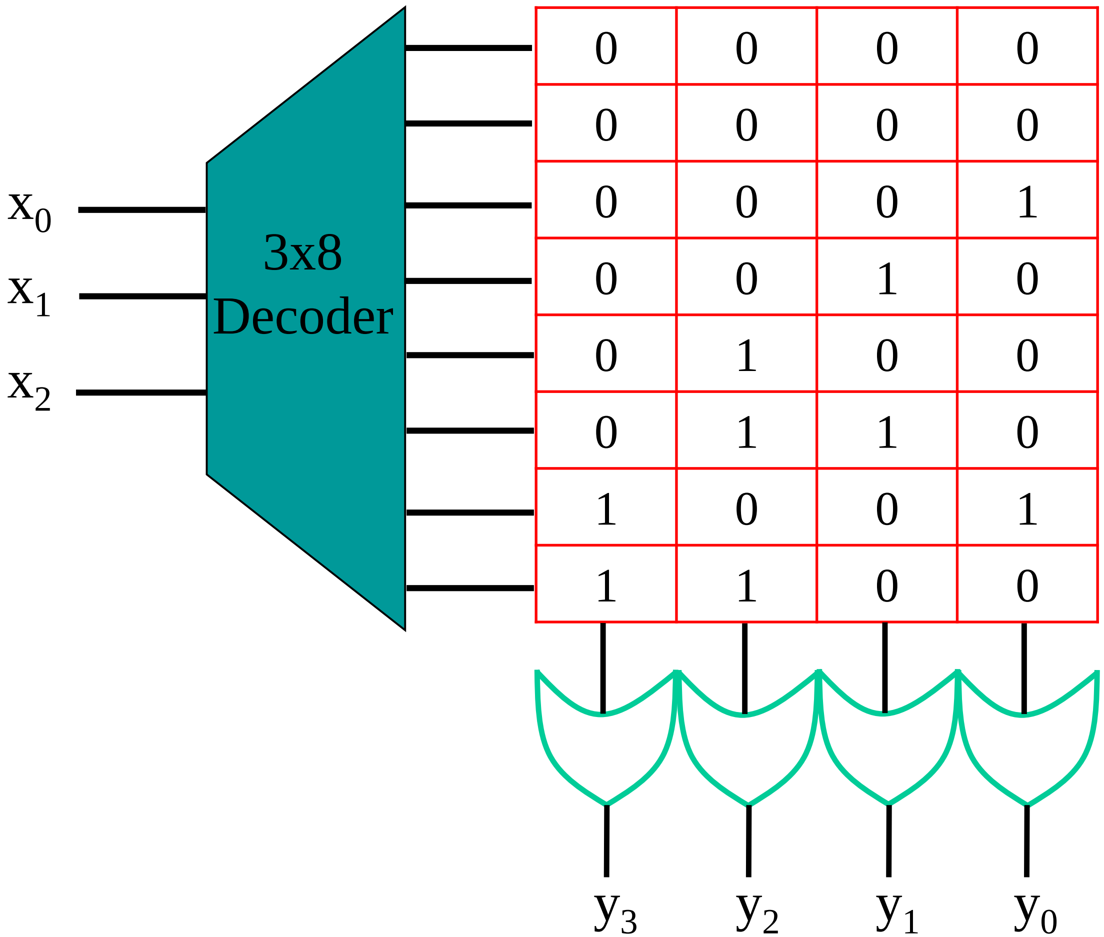
Example

- **Design a circuit which calculates the square of the 3-bit input with ROM.**
- **We have to find the input and output bit length.**
  - **The input bit length is 3. The output bit length is 6. 72 = 49 = 1100012.**
- **We have to produce the truth table.**
  - **Doğruluk Tablosu:**

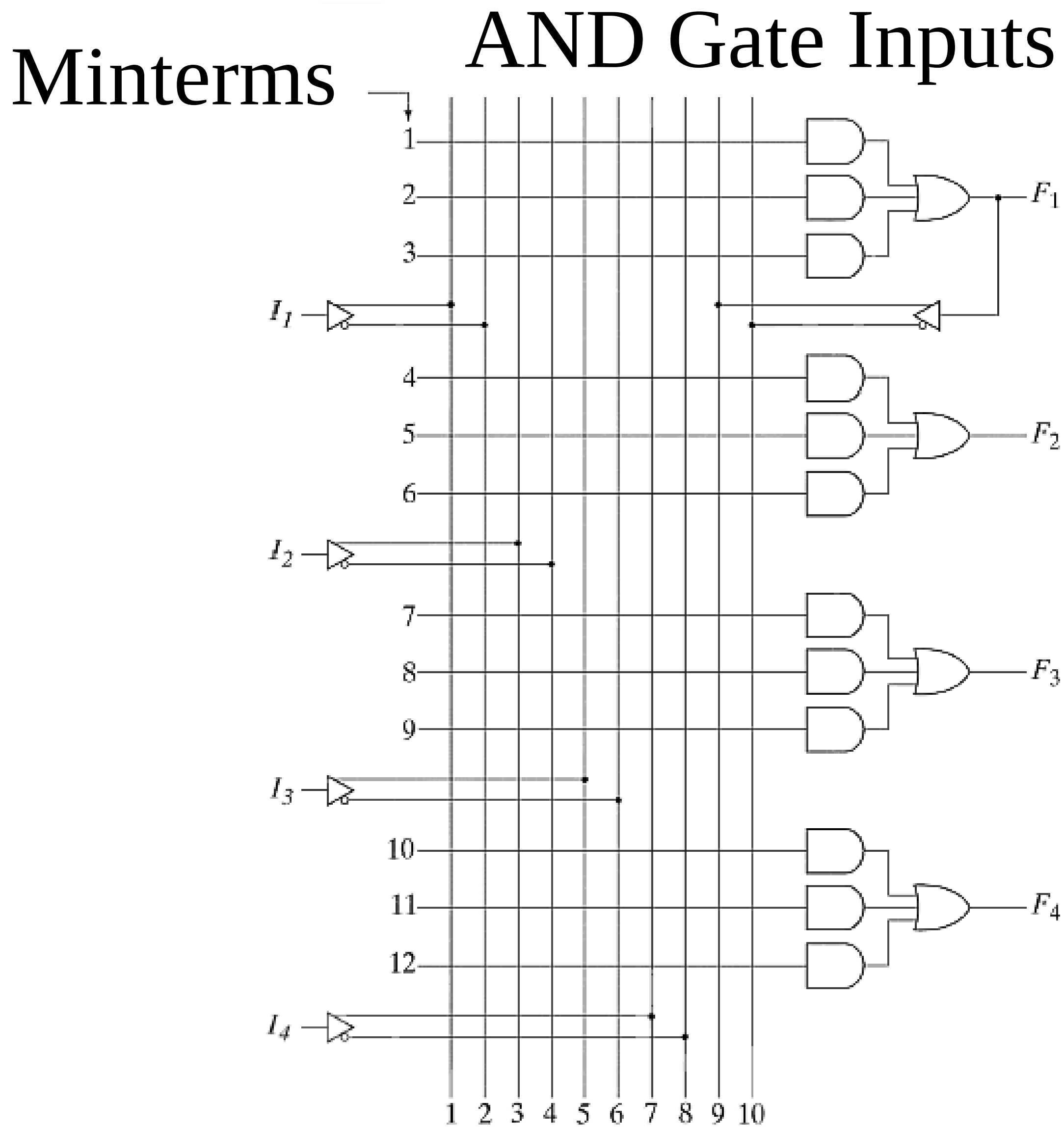
x_2	x_1	x_0	y_5	y_4	y_3	y_2	y_1	y_0
0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	1
0	1	0	0	0	0	1	0	0
0	1	1	0	0	1	0	0	1
1	0	0	0	1	0	0	0	0
1	0	1	0	1	1	0	0	1
1	1	0	1	0	0	1	0	0
1	1	1	1	1	0	0	0	1

Example

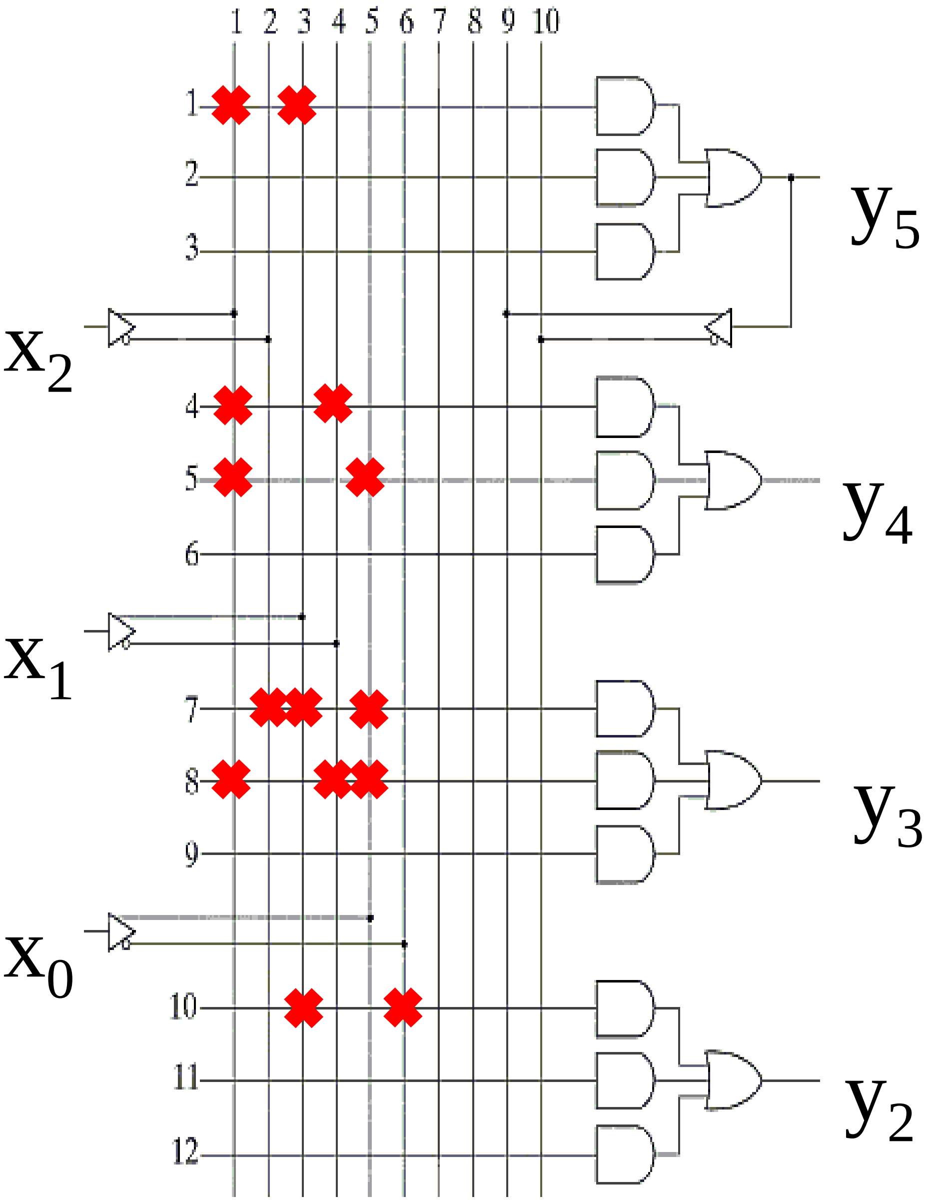
- **We decide that $y_0 = x_0$ and $y_1 = 0$ from the truth table.**
- **We need a 8×4 ROM.**



Programmable Array Logic (PAL)



Design with PAL



$y_5 = x_2x_1$

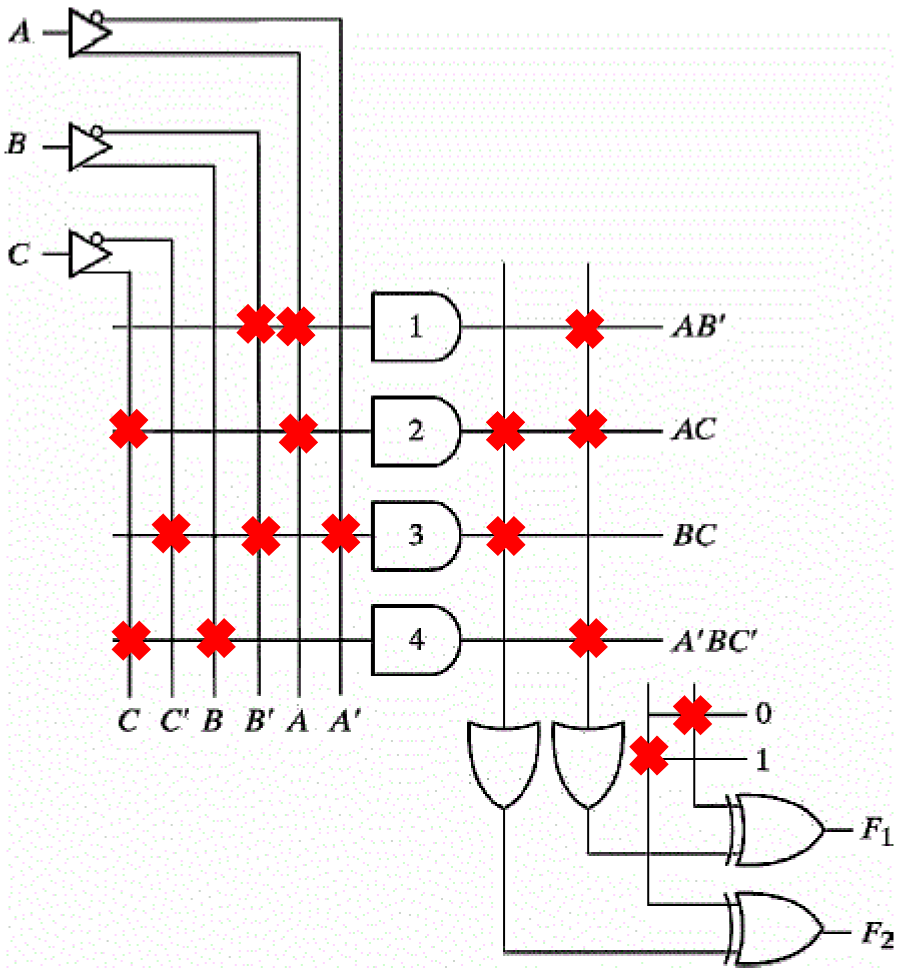
$x_0 \backslash x_2x_1$	00	01	11	10
0				1
1			1	1

$y_4 = x_2x_1' + x_2x_0$

$y_3 = x_2'x_1x_0 + x_2x_1'x_0$

$y_2 = x_1x_0'$

Programmable Logic Array (PLA)



F1 = AB' + AC + A'BC'

F2 = (AC + BC)'