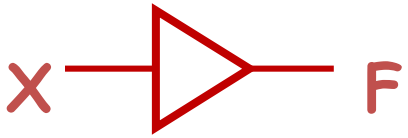
Other Gate Types

- ✓ Why?
  - Implementation **feasibility** and **low cost**
  - Power in implementing Boolean functions
  - Convenient conceptual representation
- ✓ Gate classifications
  - **Primitive gate** - a gate that can be described using a single primitive operation type (AND or OR) plus an optional inversion(s).
  - **Complex gate** - a gate that requires more than one primitive operation type for its description

Buffer

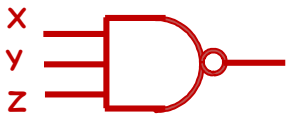
- ✓ A **buffer** is a gate with the function F = X



- ✓ In terms of Boolean function, a buffer is the same as a connection!
- ✓ A buffer is **an electronic amplifier** used to improve circuit voltage levels and increase the speed of circuit operation.

NAND Gate

- The basic NAND gate has the following symbol, illustrated for three inputs:



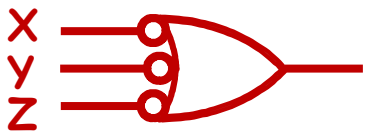
$$F(X, Y, Z) = \overline{X \; Y \; Z}$$

AND-Invert (NAND)

- NAND represents **Not AND**, i. e., the AND function followed by a NOT.
- The symbol shown is an AND-Invert. The small circle (o, "bubble") represents the invert function.

NAND Gates

- ✓ Applying DeMorgan's Law gives Invert-OR (NAND)

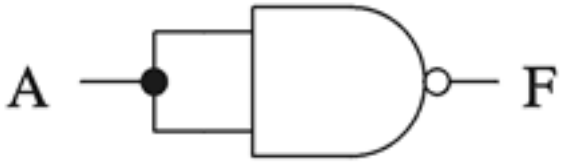


$$F(X, Y, Z) = \overline{X} + \overline{Y} + \overline{Z}$$

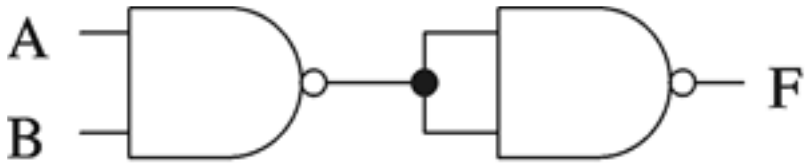
- ✓ This NAND symbol is called Invert-OR, since inputs are inverted and then ORed together.
- ✓ AND-Invert and Invert-OR both represent the **NAND gate**.

NAND Gates

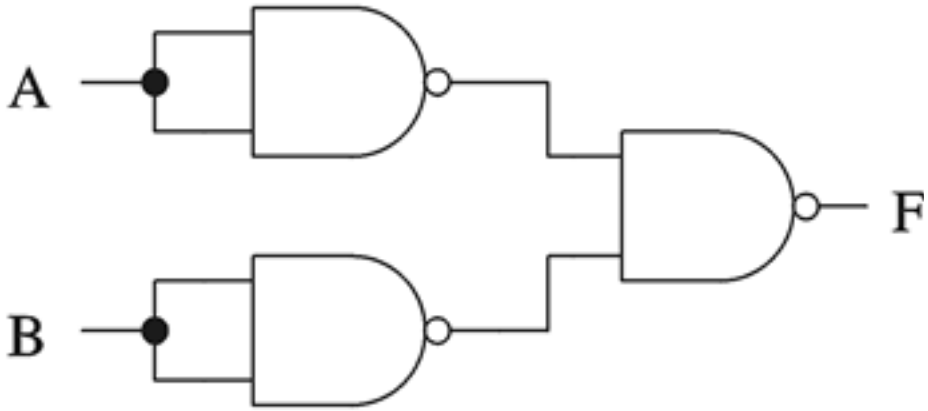
- ✓ The NAND gate is the natural implementation for CMOS technology in terms of chip area and speed.
- ✓ Universal gate: a gate type that can implement any Boolean function, NAND is an universal gate.



NOT



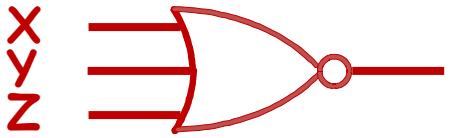
AND



OR

NOR Gate

- The basic NOR gate has the following symbol, illustrated for three inputs:



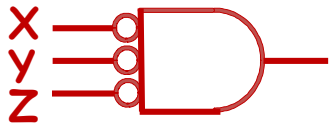
$$F(X, Y, Z) = \overline{X + Y + Z}$$

OR-Invert (NOR)

- NOR represents **NOT-OR**, i. e., the OR function followed by a NOT.
- The symbol shown is an OR-Invert. The small circle (o, "bubble") represents the invert function.

NOR Gate

- ✓ Applying DeMorgan's Law gives Invert-AND (NOR)

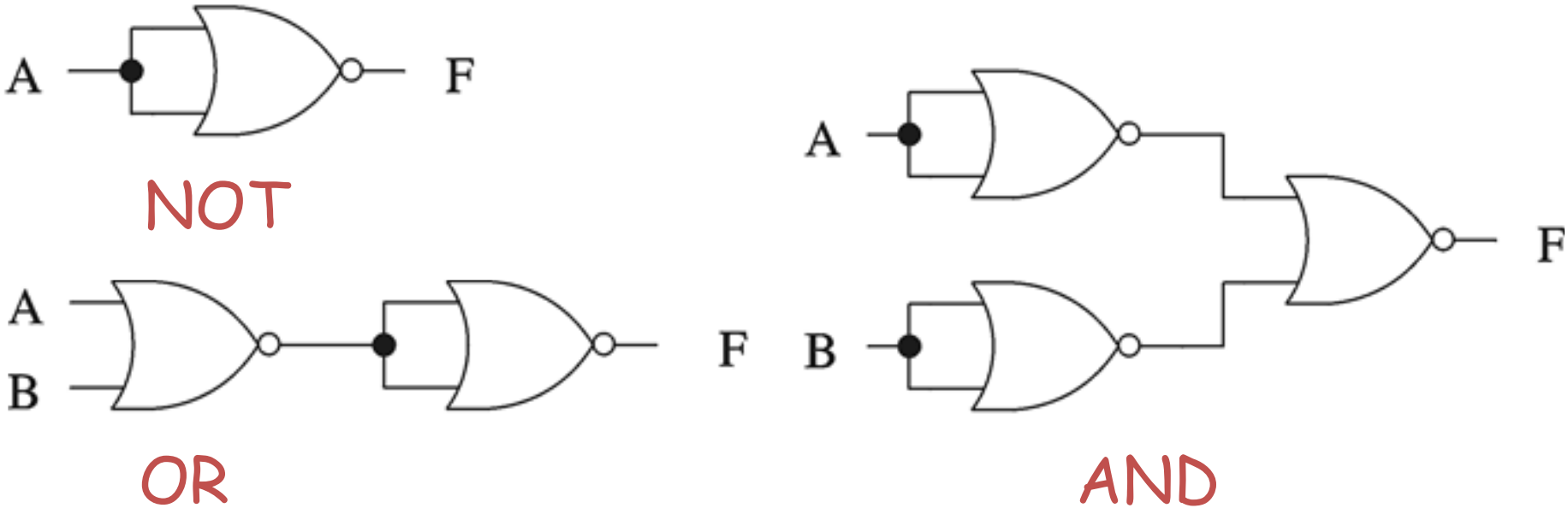


$$F(X, Y, Z) = \overline{X} \cdot \overline{Y} \cdot \overline{Z}$$

- ✓ This NOR symbol is called Invert-AND, since inputs are inverted and then ANDed together.
- ✓ OR-Invert and Invert-AND both represent the **NOR gate.**

NOR Gate (continued)

- ✓ The NOR gate is a natural implementation for some technologies other than CMOS in terms of chip area and speed.
- ✓ The NOR gate is an universal gate



Exclusive OR/Exclusive NOR

- ✓ The *eXclusive OR* (*XOR*) function is an important Boolean function used extensively in logic circuits.
- ✓ The XOR function may be;
  - implemented directly as an electronic circuit
  - implemented by interconnecting other gate types

- ✓ The *eXclusive NOR* (XNOR) function is the complement of the XOR function
- ✓ By our definition, XOR and XNOR gates are complex gates.

Exclusive OR/ Exclusive NOR

- ✓ Uses for the XOR and XNORs gate include:
  - Adders/subtractors/multipliers
  - Counters/incrementers/decrementers
  - Parity generators/checkers
- ✓ Definitions
  - The XOR function is: $X \oplus Y = X\overline{Y} + \overline{X}Y$
  - The eXclusive NOR (XNOR) function, otherwise known as *equivalence* is: $\overline{X \oplus Y} = XY + \overline{X}\,\overline{Y}$
- ✓ Strictly speaking, XOR and XNOR gates do no exist for more that two inputs. Instead, they are replaced by **odd** and **even** functions.

Truth Tables for XOR/XNOR

- ✓ Operator Rules:

X	Y	$X \oplus Y$
0	0	0
0	1	1
1	0	1
1	1	0

XOR

X	Y	$\overline{X \oplus Y}$ or $X \equiv Y$
0	0	1
0	1	0
1	0	0
1	1	1

XNOR

- ✓ The XOR function means:

  X OR Y, but NOT BOTH

- ✓ The XNOR function means:

  X AND Y are equal, BOTH 0 OR 1

XOR/XNOR

- The XOR function can be extended to 3 or more variables. For more than 2 variables, it is called an **odd function** or modulo 2 sum (*Mod 2 sum*), not an XOR:

$$X \oplus Y \oplus Z = \overline{X}\,\overline{Y}\,Z + \overline{X}\,Y\,\overline{Z} + X\,\overline{Y}\,\overline{Z} + X\,Y\,Z$$

- The complement of the odd function is the even function.
- The XOR identities:

$$X \oplus 0 = X \qquad X \oplus 1 = \overline{X}$$

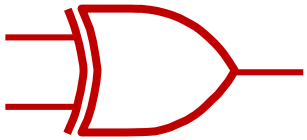
$$X \oplus X = 0 \qquad X \oplus \overline{X} = 1$$

$$X \oplus Y = Y \oplus X \qquad \overline{X \oplus Y} = Y \oplus X$$

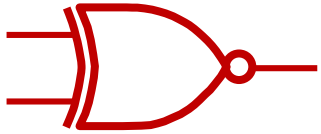
$$(X \oplus Y) \oplus Z = X \oplus (Y \oplus Z) = X \oplus Y \oplus Z$$

Symbols For XOR and XNOR

- ✓ XOR symbol:



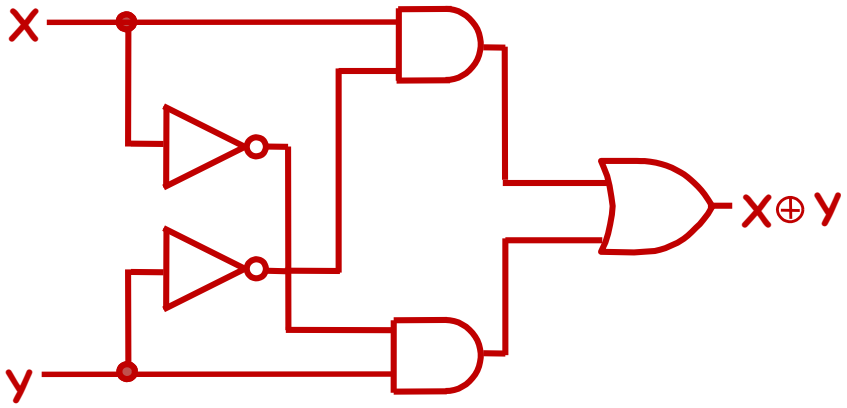
- ✓ XNOR symbol:



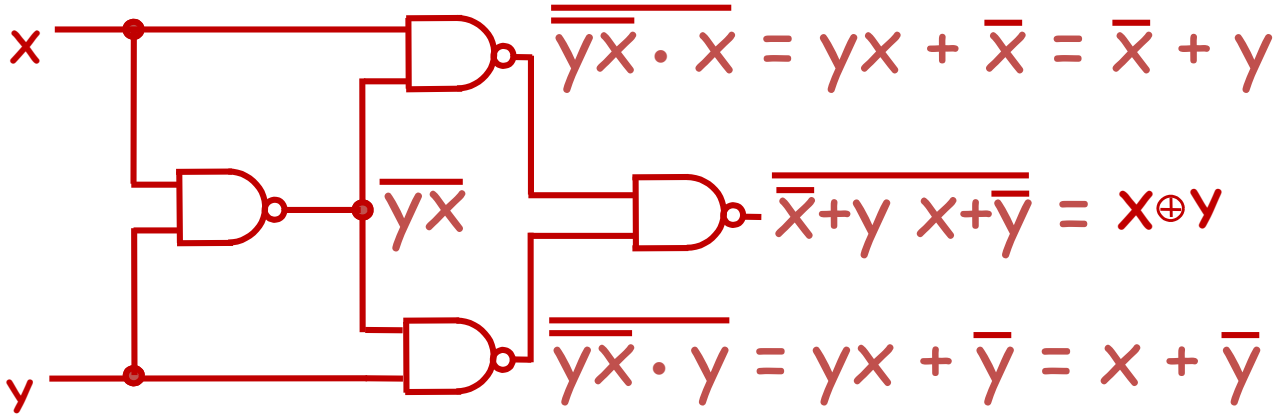
- ✓ Shaped symbols exist only for two inputs

XOR Implementations

- ✓ The simple SOP implementation uses the following structure:



- ✓ A NAND-only implementation is:

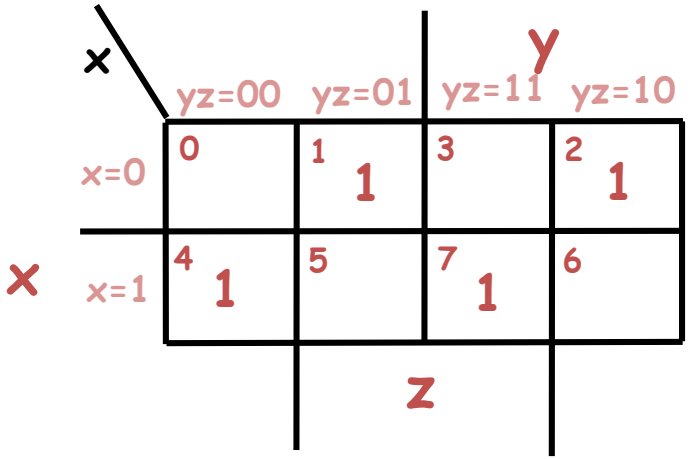


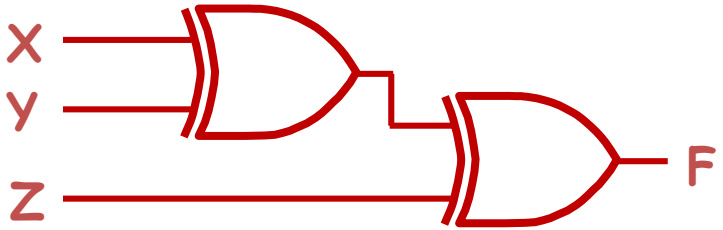
Odd and Even Functions

- ✓ The odd and even functions on a K-map form checkerboard patterns.
- ✓ SOP for odd and even functions are consist of disjoint of sets 2^{n-1} minterms that are prime implicants.
- ✓ Implementation of odd and even functions for greater than four variables as a two-level circuit is tough, so we use "trees".

Example: Odd Function Implementation

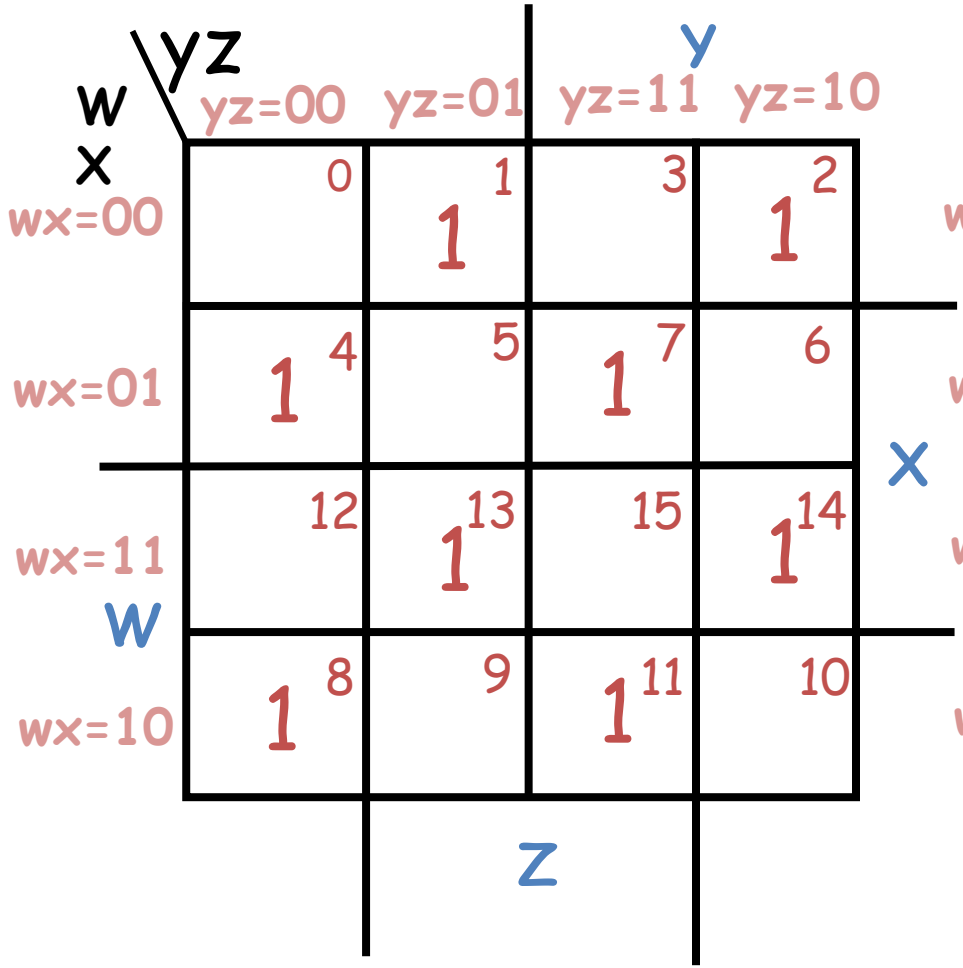
- ✓ Design a 3-input odd function $F = X \oplus Y \oplus Z$ with 2-input XOR gates
- ✓ Factoring, $F = (X \oplus Y) \oplus Z$
- ✓ The circuit:



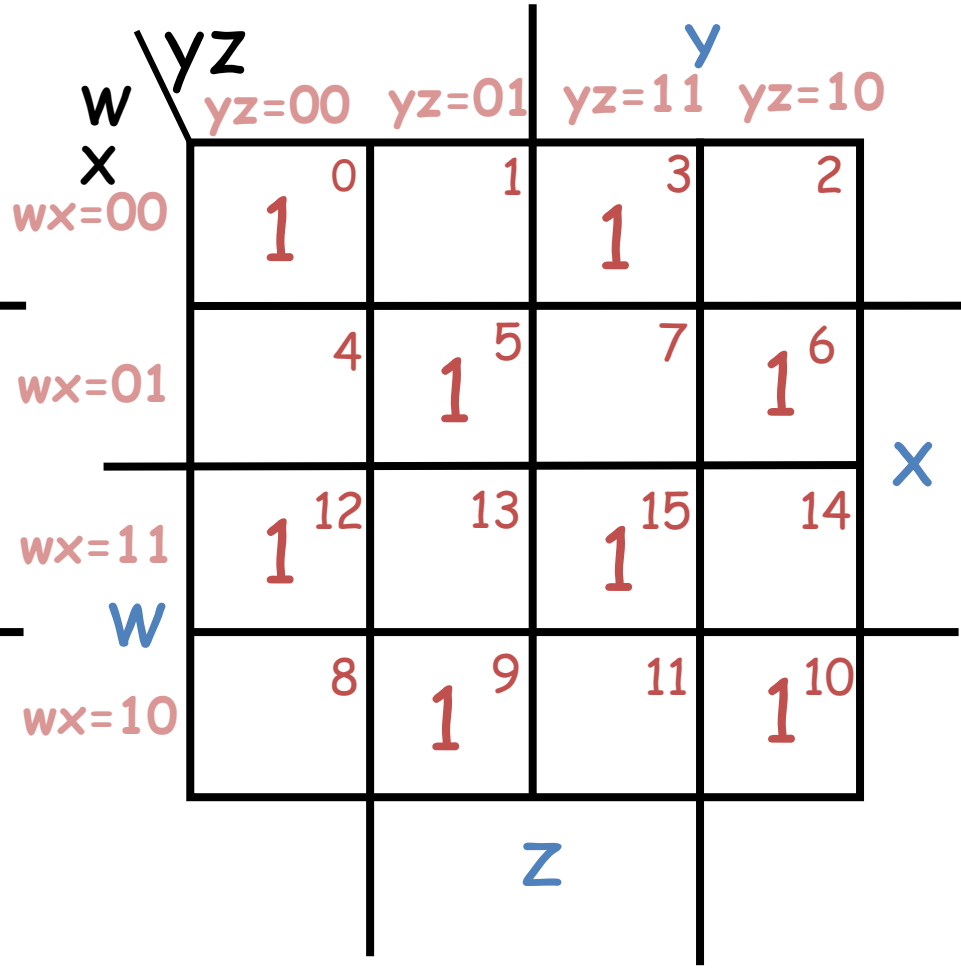


Odd and Even Functions

✓ Odd function

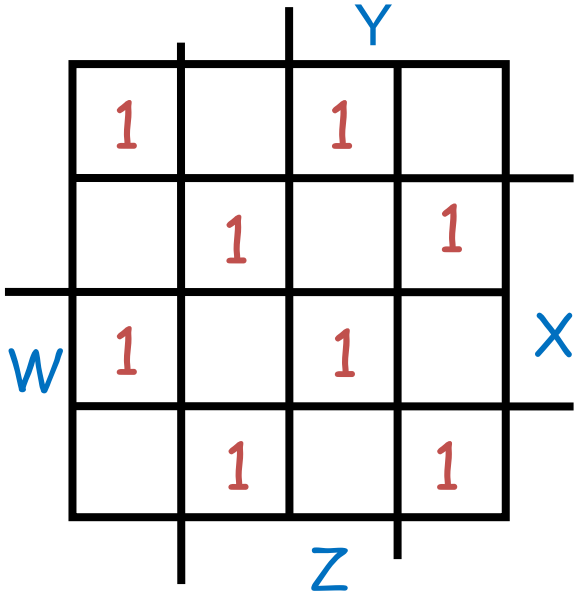


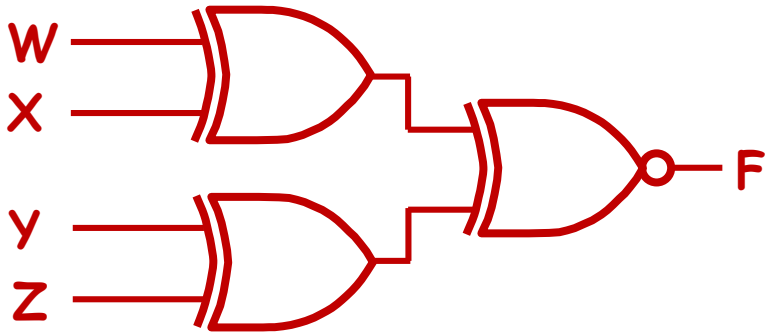
Even function



Example: Even Function Implementation

- ✓ Design a 4-input even function $F = \overline{W \oplus X \oplus Y \oplus Z}$ with 2-input XOR and XNOR gates
- ✓ Factoring, $F = \overline{(W \oplus X) \oplus (Y \oplus Z)}$
- ✓ The circuit:

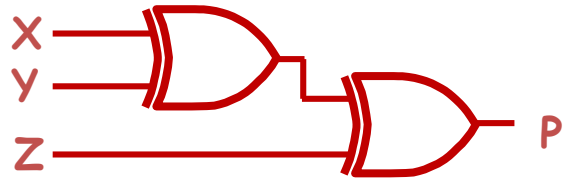




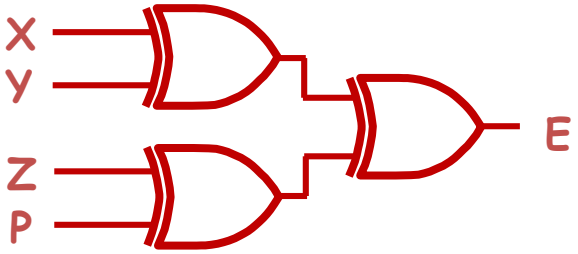
Parity Generators and Checkers

- ✓ Parity bit added to n-bit code to produce an n + 1 bit code:
  - Add odd parity bit to generate code words with even parity
  - Add even parity bit to generate code words with odd parity
  - Use odd parity circuit to check error words with even parity
  - Use even parity circuit to check error words with odd parity

- ✓ Example: n = 3. Generate even parity code words of length four with odd parity generator:



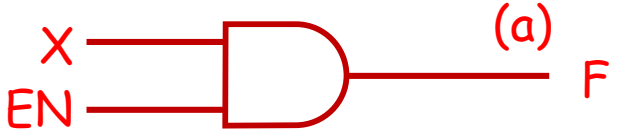
- ✓ Check even parity code words of length four with odd parity checker:
- ✓ Operation: (X,Y,Z) = (0,0,1) gives (X,Y,Z,P) = (0,0,1,1) and E = 0.
  If Y changes from 0 to 1 between generator and checker (X,Y,Z, P) = (0,1,1,1) , then E = 1 indicates an error.



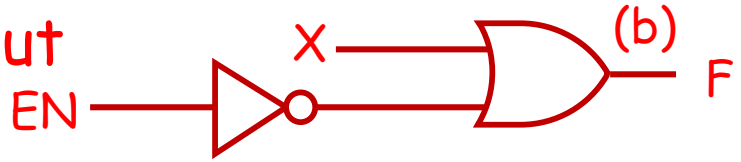
Enabling Function

- ✓ 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
- ✓ When disabled EN=0

0 output



1 output

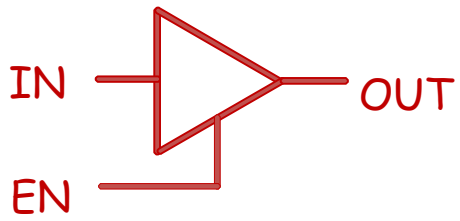


- ✓ When enabled EN = 1 F = X
- ✓ The value on the output when it is disable can be 0, 1, or Hi-Z

The 3-State Buffer

- For the symbol and truth table, IN is the data input, and EN, the control input.
- For EN = 0, regardless of the value on IN (denoted by X), the output value is Hi-Z.
- For EN = 1, the output value follows the input value.
- High impedance (or Hi-Z): In that state the output is disconnected which is equal to open circuit. In the other words in that state circuit has no logic significant.

Symbol



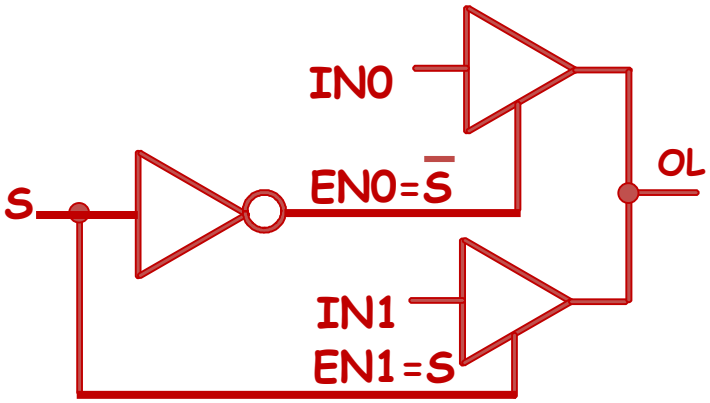
Truth Table

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

3-State Logic Circuit

- ✓ Data Selection Function: If S = 0, OL = IN0, else OL = IN1
- ✓ Performing data selection with 3-state buffers:

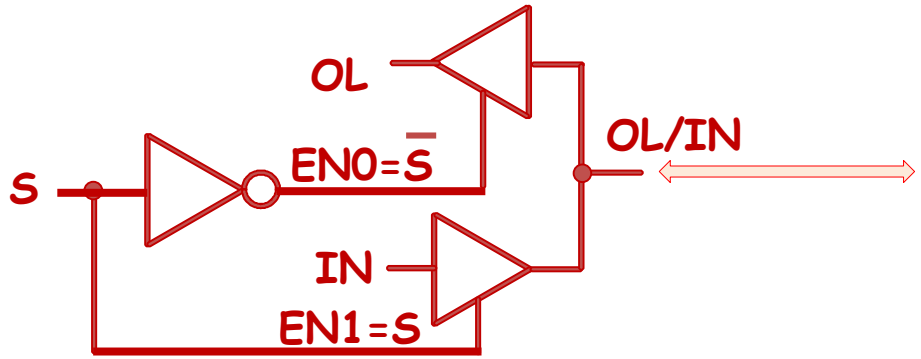
EN0 $\overline{S}$	IN0	EN1 S	IN1	OL
0	X	1	0	0
0	X	1	1	1
1	0	0	X	0
1	1	0	X	1



- ✓ Since EN0 = $\overline{S}$ and EN1 = S, one of the two buffer outputs is always Hi-Z and never occurs EN0 = EN1 = 1.

More Complex Gates

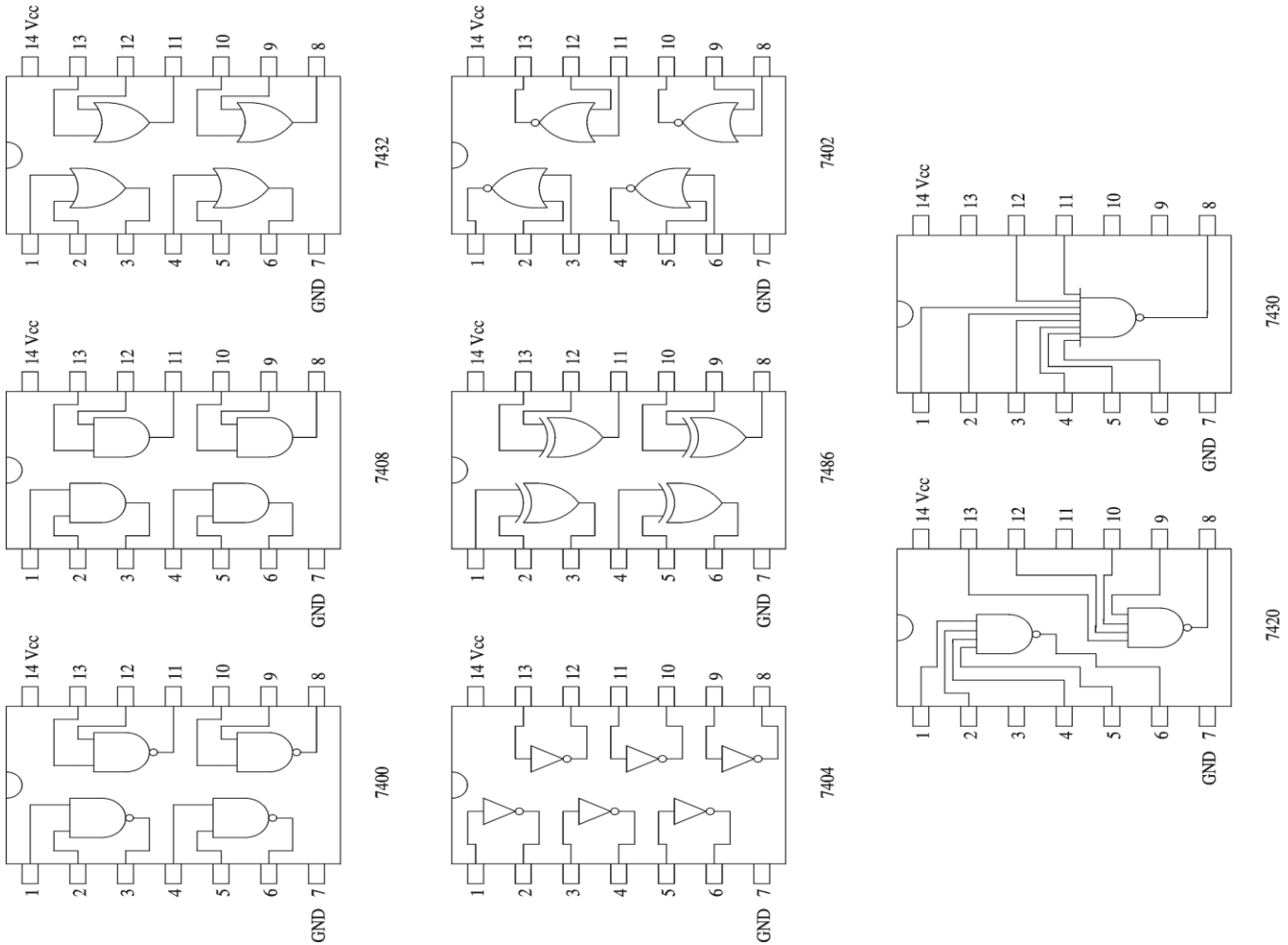
- ✓ Bi-diretional line:
  - S=0: Left ⊂ Right;
  - S=1: Left ⊃ Right, or viceversa?



More Complex Gates (continued)

- ✓ The remaining complex gates are SOP or POS structures with and without an output inverter.
- ✓ The names are derived using:
  - A - AND
  - O - OR
  - I - Inverter
  - Numbers of inputs on first-level "gates" or directly to second-level "gates"
- ✓ Example AOI: AND-OR-Invert consists of a single gate with AND functions driving an OR function which is inverted.
- ✓ Example: 2-1 AO has two 2-input ANDs driving an OR
- ✓ These gate types are used because:
  - the number of transistors needed is fewer than required by connecting together primitive gates
  - potentially, the circuit delay is smaller, increasing the circuit operating speed

Commercial Logic Chips

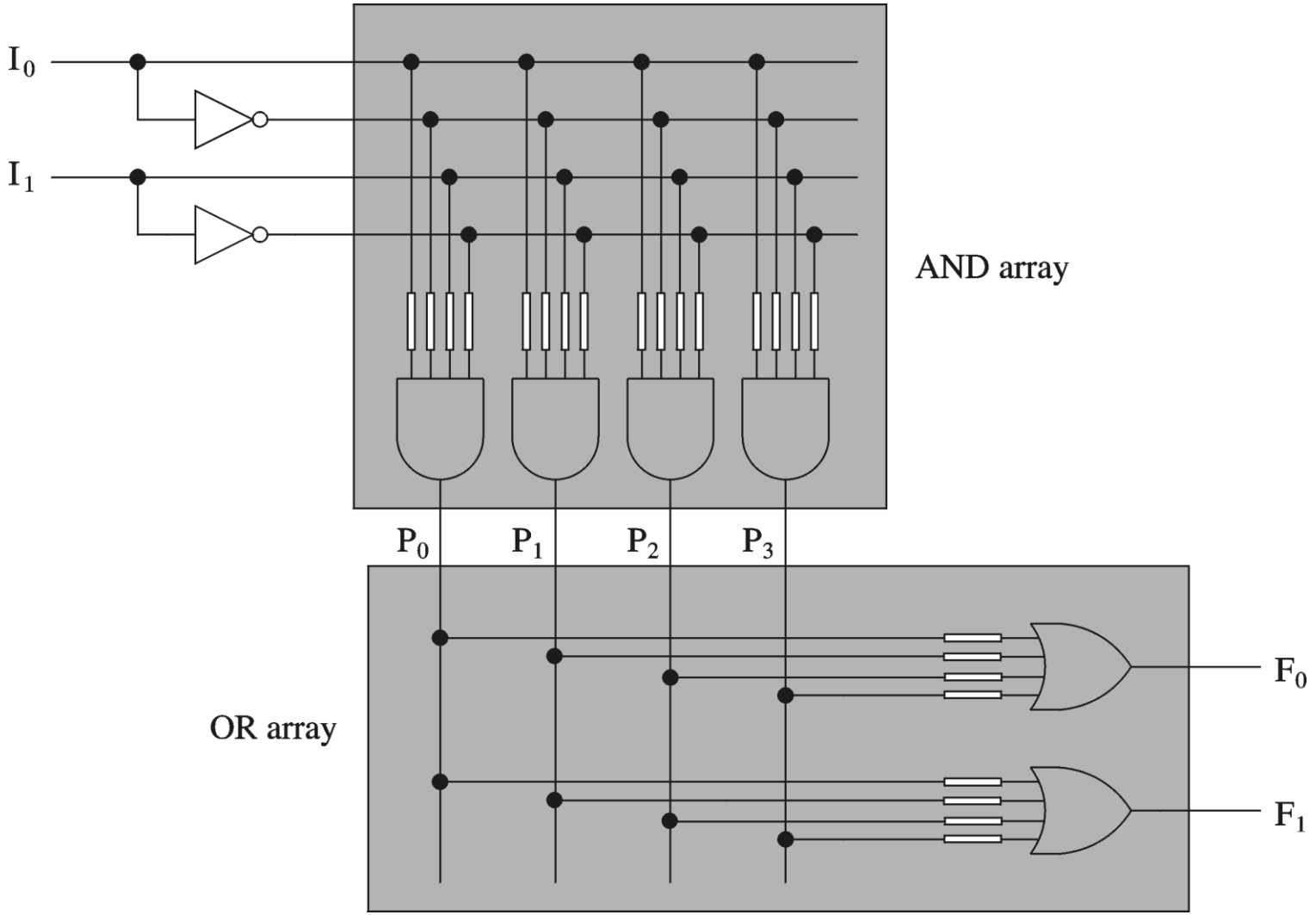


PLA: Programmable Logic Arrays

- ✓ PLAs
  - Implement sum-of-product expressions
    - 2 levels, no possibility to simplify the logical expressions
  - Take *N* inputs and produce *M* outputs
    - Each input represents a logical variable
    - Each output represents a logical function output
  - Internally uses
    - An AND array
      - Each AND gate receives 2*N* inputs: *N* inputs and their complements
    - An OR array

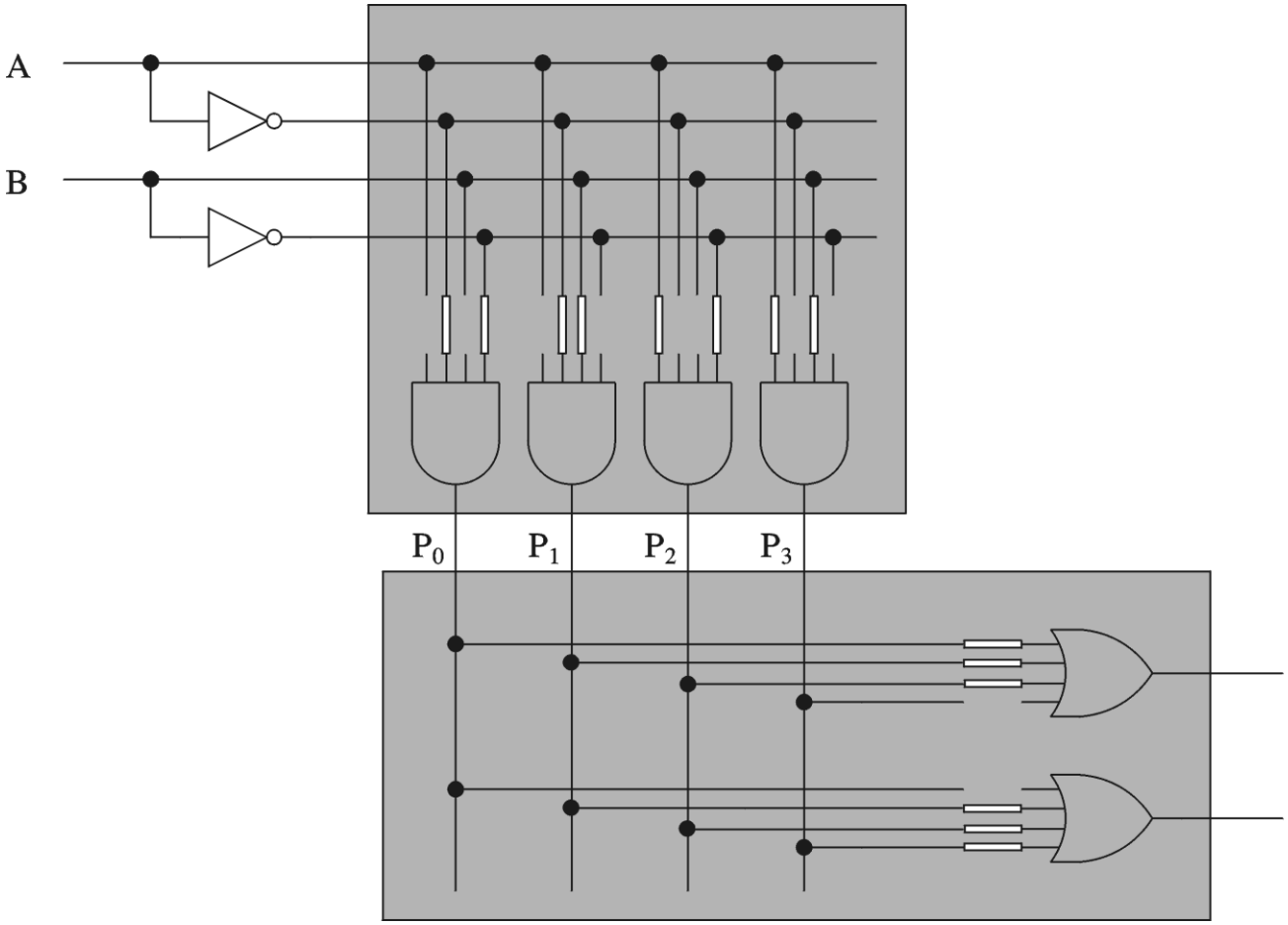
Programmable Logic Arrays (cont.)

- ✓ A blank PLA with 2 inputs and 2 outputs



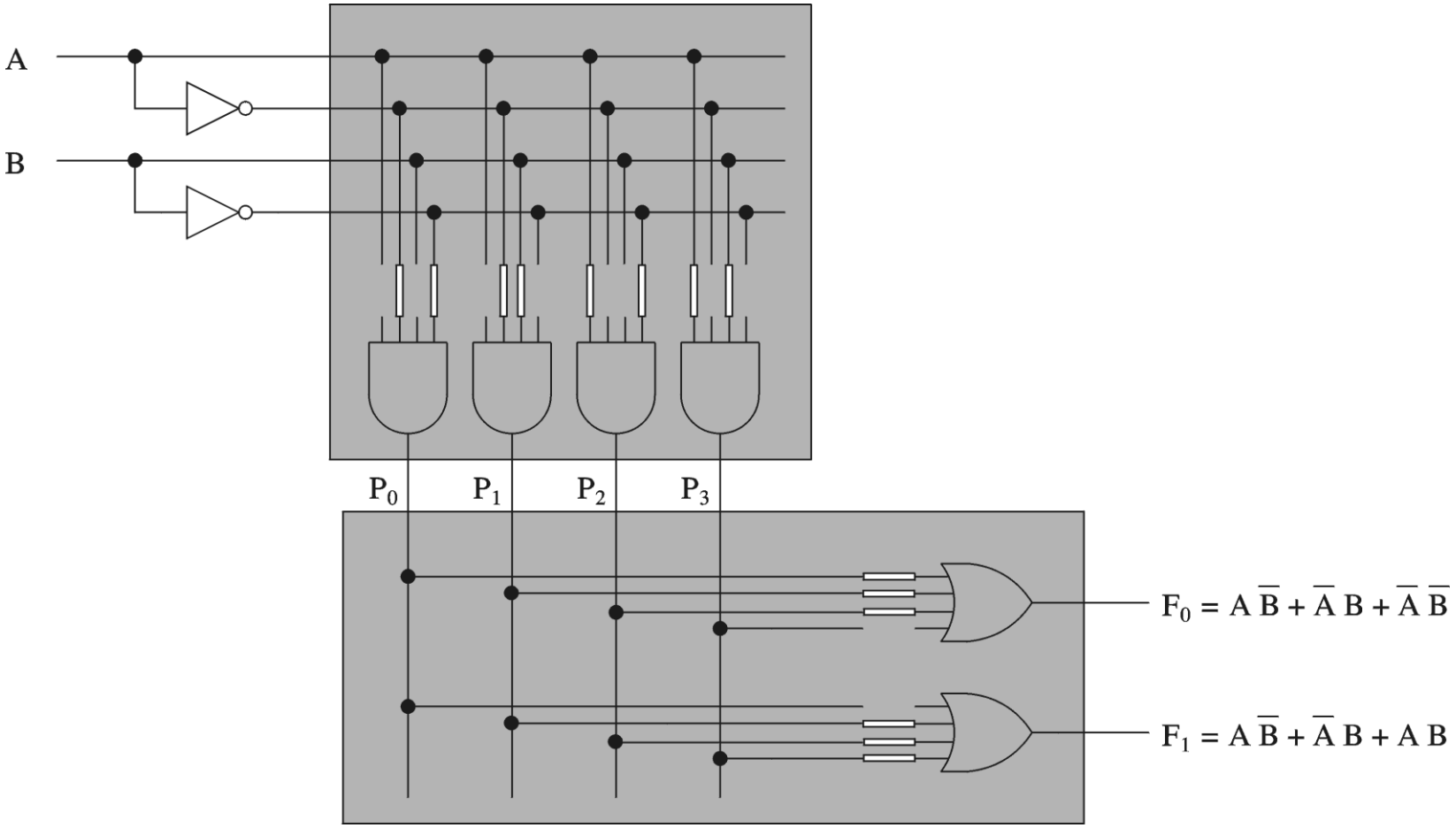
Programmable Logic Arrays (cont.)

- ✓ Implementation examples



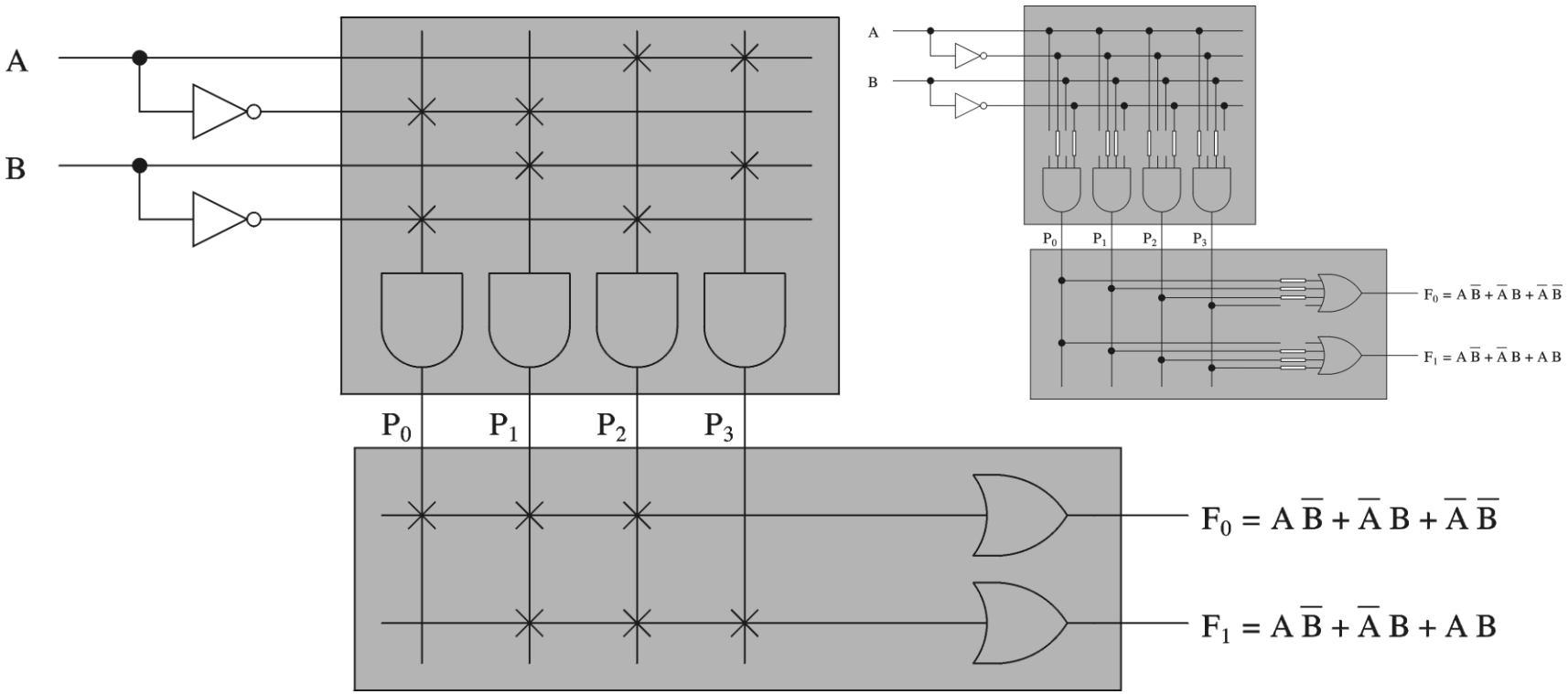
Programmable Logic Arrays (cont.)

- ✓ Implementation examples



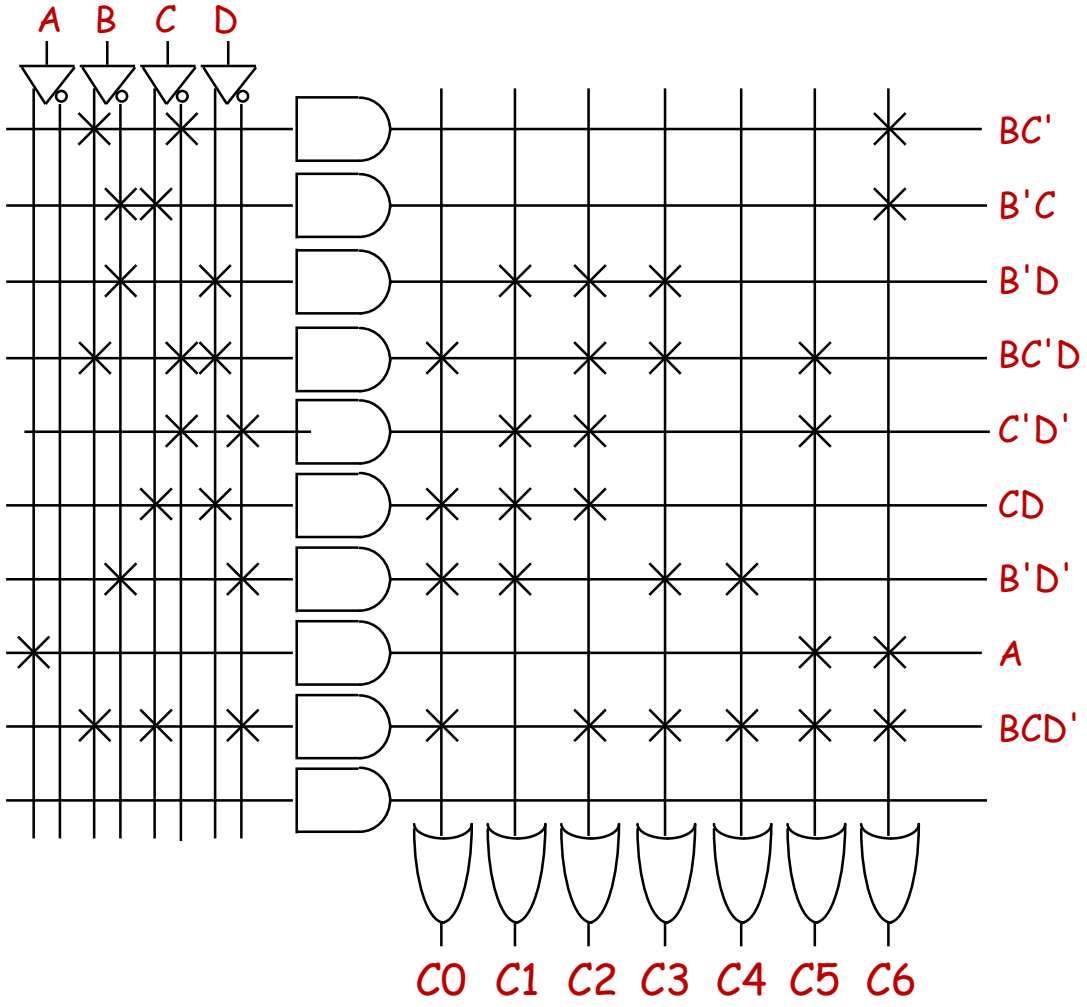
Programmable Logic Arrays (cont.)

- ✓ Simplified notation



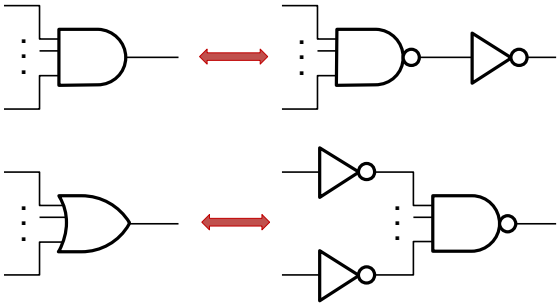
PLA implementation
BCD to 7-segment display controller

C0 = B C' D + C D + B' D' + B C D'
C1 = B' D + C' D' + C D + B' D'
C2 = B' D + B C' D + C' D' + C D + B C D'
C3 = B C' D + B' D + B' D' + B C D'
C4 = B' D' + B C D'
C5 = B C' D + C' D' + A + B C D'
C6 = B' C + B C' + B C D' + A



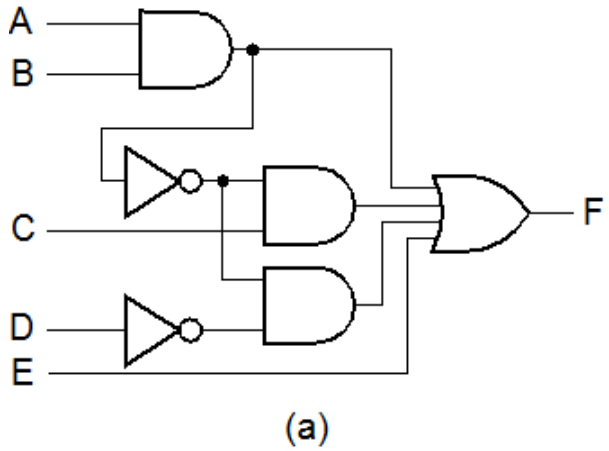
NAND Mapping Algorithm

1. Replace ANDs and ORs:

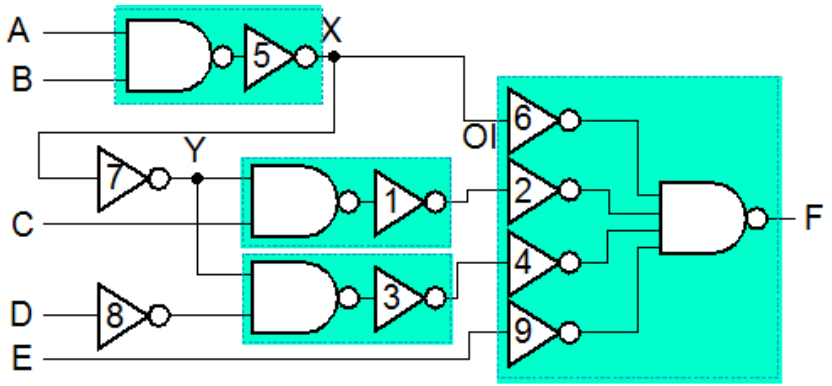


2. Note that:

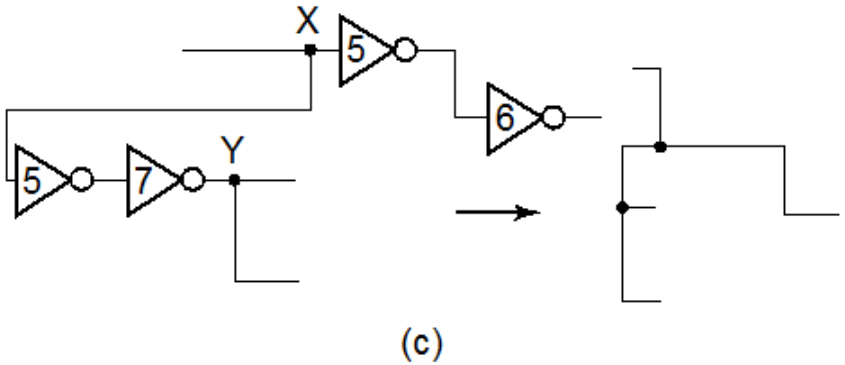
NAND Mapping Example



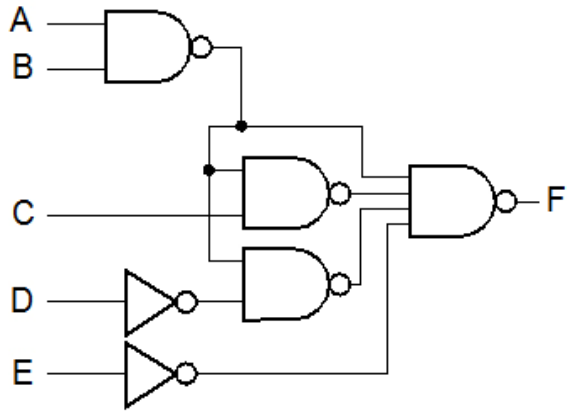
(a)



(b)



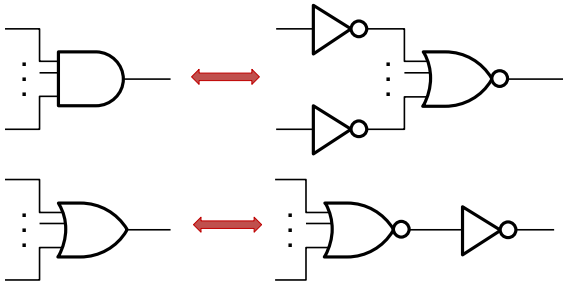
(c)



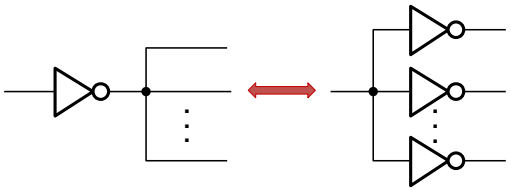
(d)

NOR Mapping Algorithm

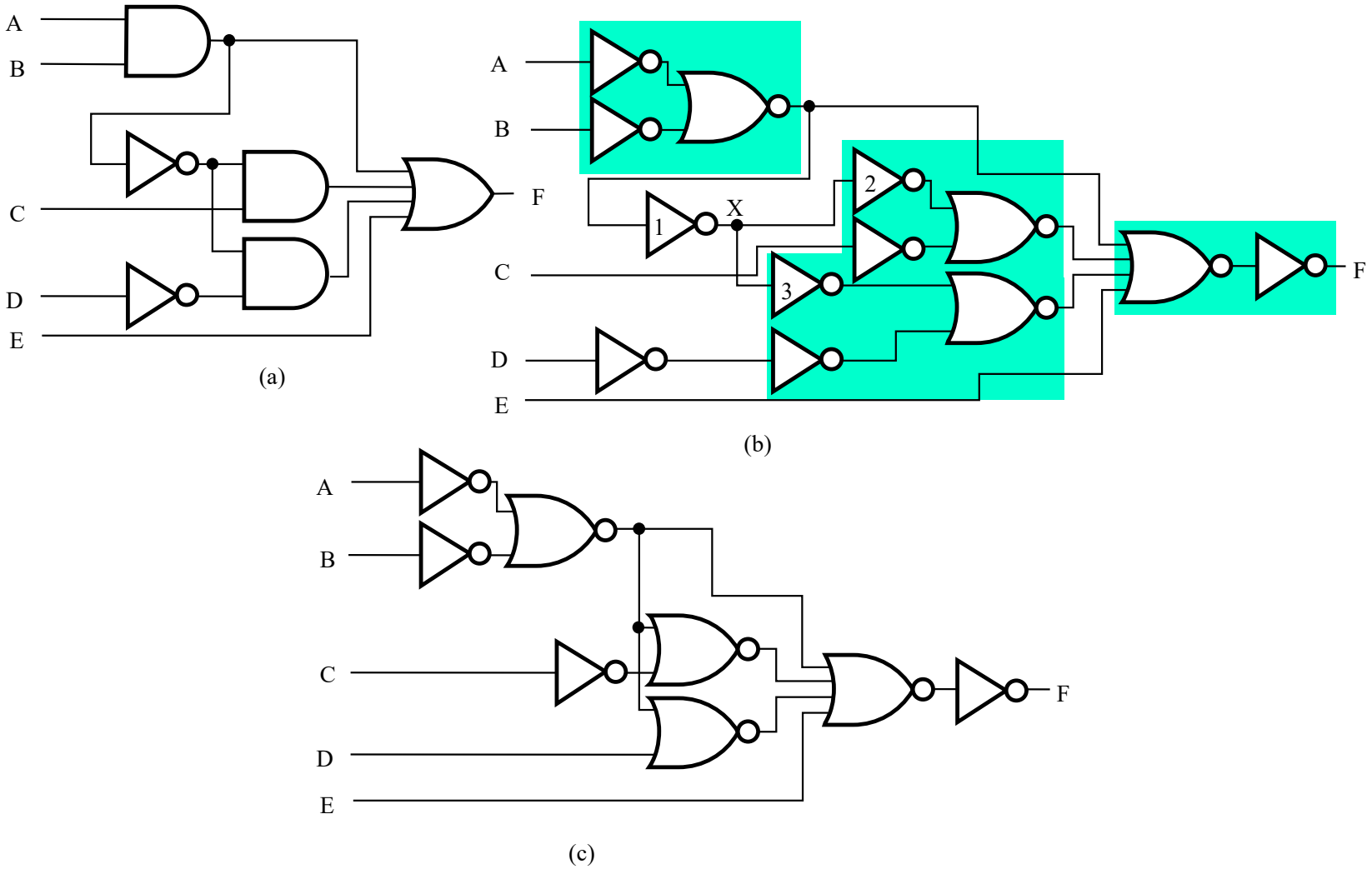
1. Replace ANDs and ORs:



2. Note that:



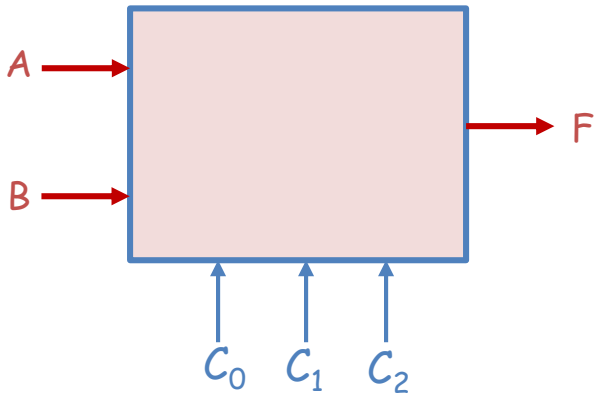
NOR Mapping Example



Logical Function Unit

- ✓ Multi-purpose Function Block
  - 3 control inputs to specify operation to perform on operands
  - 2 data inputs for operands
  - 1 output of the same bit-width as operands

3 control inputs: C_0, C_1, C_2
2 data inputs: A, B
1 output: F



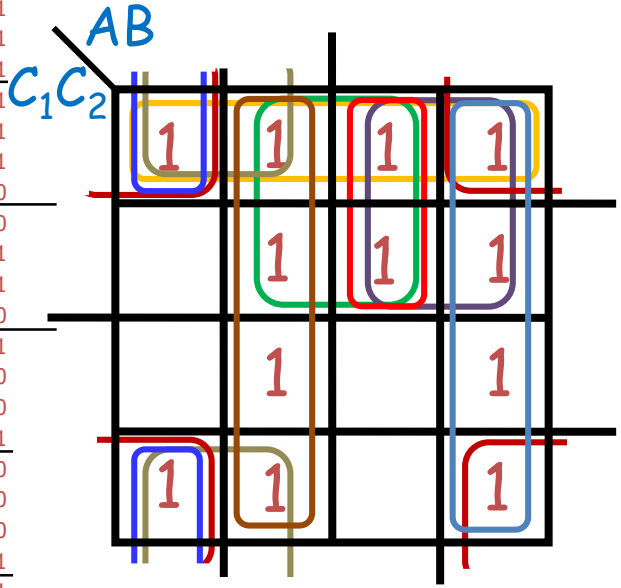
C_0	C_1	C_2	Function	Comments
0	0	0	1	always 1
0	0	1	A + B	logical OR
0	1	0	(A • B)'	logical NAND
0	1	1	A xor B	logical xor
1	0	0	A xnor B	logical xnor
1	0	1	A • B	logical AND
1	1	0	(A + B)'	logical NOR
1	1	1	0	always 0

Formalize the Problem

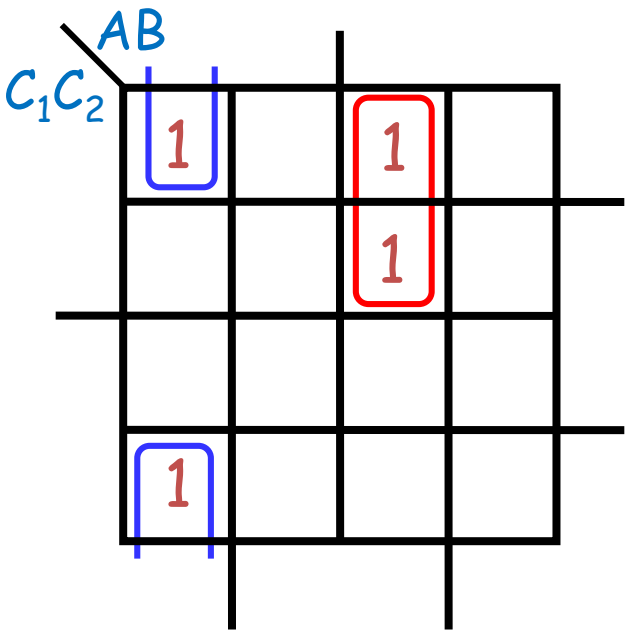
C_0	C_1	C_2	Function	Comments
0	0	0	1	always 1
0	0	1	A + B	logical OR
0	1	0	(A + B)'	logical NAND
0	1	1	A xor B	logical xor
1	0	0	A xnor B	logical xnor
1	0	1	A • B	logical AND
1	1	0	(A • B)'	logical NOR
1	1	1	0	always 0

C_0	C_1	C_2	A	B	F
0	0	0	0	0	1
0	0	0	0	1	1
0	0	0	1	0	1
0	0	0	1	1	1
0	0	1	0	0	0
0	0	1	0	1	1
0	0	1	1	0	1
0	0	1	1	1	1
0	1	0	0	0	1
0	1	0	0	1	1
0	1	0	1	0	1
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	0	1	1
0	1	1	1	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	0	0	1	0
1	0	0	1	0	0
1	0	0	1	1	1
1	0	1	0	0	0
1	0	1	0	1	0
1	0	1	1	0	0
1	0	1	1	1	1
1	1	0	0	0	1
1	1	0	0	1	0
1	1	0	1	0	0
1	1	0	1	1	0
1	1	1	0	0	0
1	1	1	0	1	0
1	1	1	1	0	0
1	1	1	1	1	0

$C_0 = 0$



$C_0 = 1$



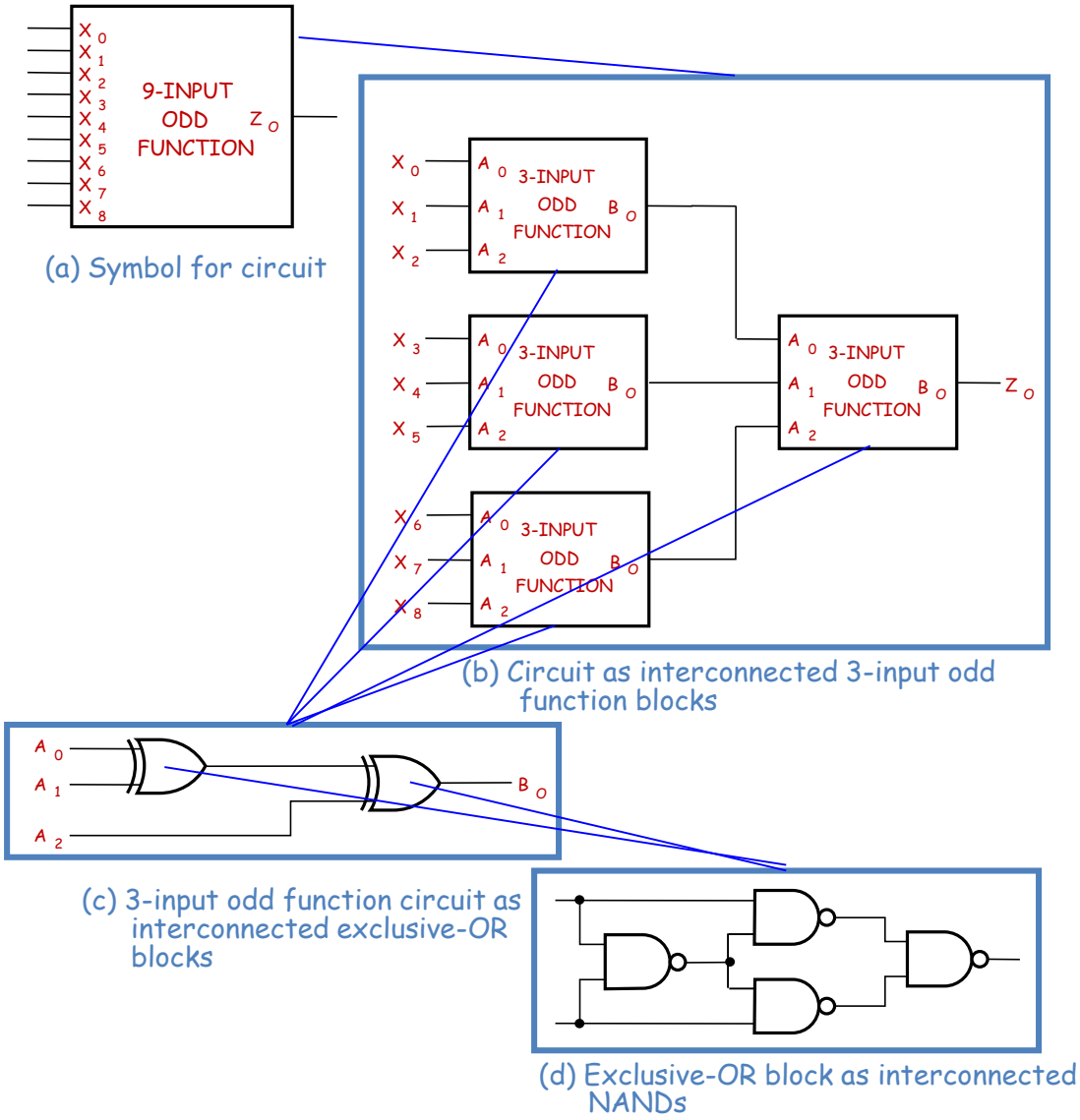
$$\sum_m = C_0'A'B + C_0'AB' + C_2'A'B' + C_1'AB$$

$$\sum_c = \sum_m + C_0'C_1'C_2' + C_0'C_1'B + C_0'C_1'A + C_0'C_2'A' + C_0'C_2'B'$$

Beginning Hierarchical Design

- ✓ To control the complexity of the function mapping inputs to outputs:
  - Decompose the function into smaller pieces called blocks
  - Decompose each block's function into smaller blocks, repeating as necessary until all blocks are small enough
  - Any block not decomposed is called a primitive block
  - The collection of all blocks including the decomposed ones is a hierarchy
- ✓ Example: 9-input odd function
  - Top Level: 9 inputs, one output
    - 2nd Level: Four 3-bit odd parity trees in two levels
      - 3rd Level: Two 2-bit exclusive-OR functions
        - Primitives: Four 2-input NAND gates
  - Design requires 4 X 2 X 4 = 32 2-input NAND gates

Hierarchy for Parity Tree Example



Reusable Functions

- ✓ Whenever possible, we try to decompose a complex design into common, reusable function blocks
- ✓ These blocks are
  - verified and well-documented
  - placed in libraries for future use

Functions and Functional Blocks

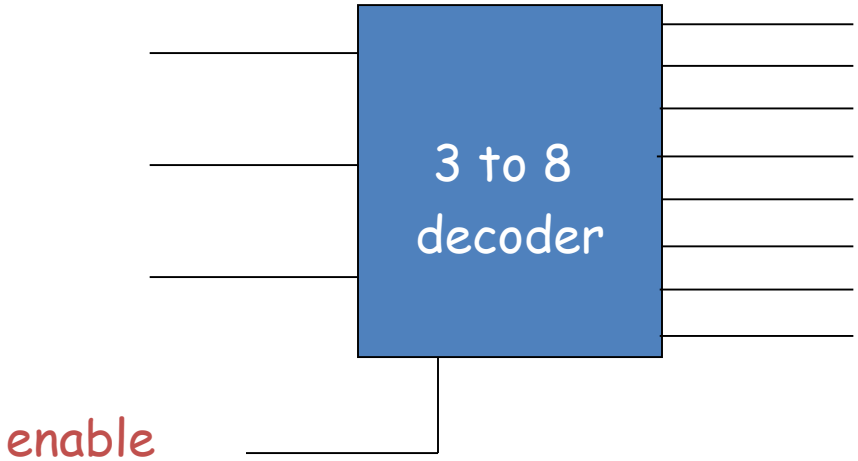
- ✓ The functions considered are those found to be very useful in design
- ✓ Corresponding to each of the functions is a combinational circuit implementation called a functional block.
- ✓ In the past, functional blocks were packaged as small-scale-integrated (SSI), medium-scale integrated (MSI), and large-scale-integrated (LSI) circuits.
- ✓ Today, they are often simply implemented within a very-large-scale-integrated (VLSI) circuit.

Top-Down versus Bottom-Up

- ✓ A top-down design proceeds from an abstract, high-level specification to a more and more detailed design by decomposition and successive refinement
- ✓ A bottom-up design starts with detailed primitive blocks and combines them into larger and more complex functional blocks
- ✓ Design usually proceeds top-down to known building blocks ranging from complete CPUs to primitive logic gates or electronic components.

Decoder

- ✓ Is a combinational circuit that converts binary information from n input lines to a maximum of 2^n unique output lines. For example if the number of input is n=3 the number of output lines can be $m=2^3$. It is also known as 1 of 8 because one output line is selected out of 8 available lines:



Decoding

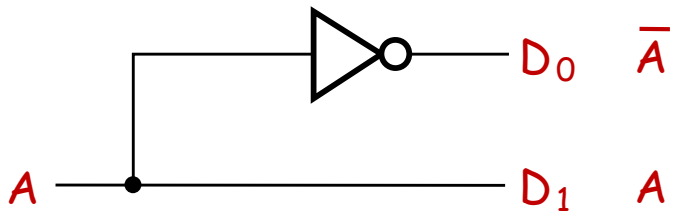
- ✓ Decoding - the conversion of an *n*-bit input code to an *m*-bit output code with $n \leq m \leq 2^n$ such that each valid code word produces a unique output code
- ✓ Functional blocks for decoding are
  - called *n*-to-*m* line decoders, where $m \leq 2^n$, and
  - generate 2^n (or fewer) minterms for the *n* input variables

Decoder Examples

✓ 1-to-2-Line Decoder

A	D_0	D_1
0	1	0
1	0	1

(a)

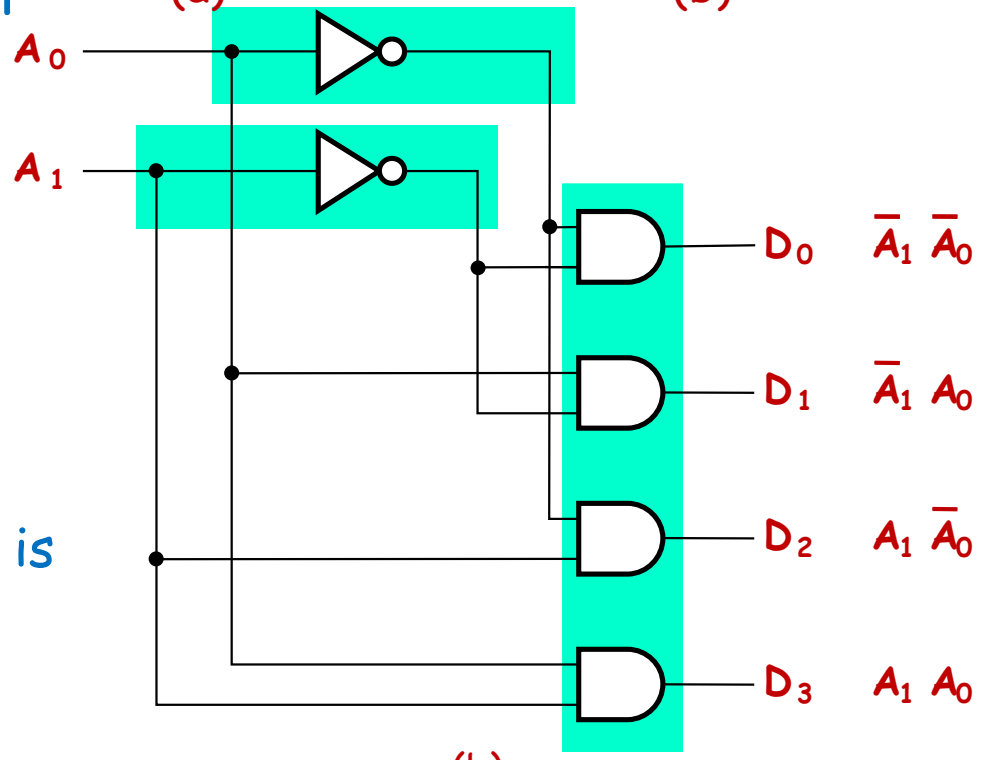


(b)

✓ 2-to-4-Line Decoder

A_1	A_0	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

(a)



(b)

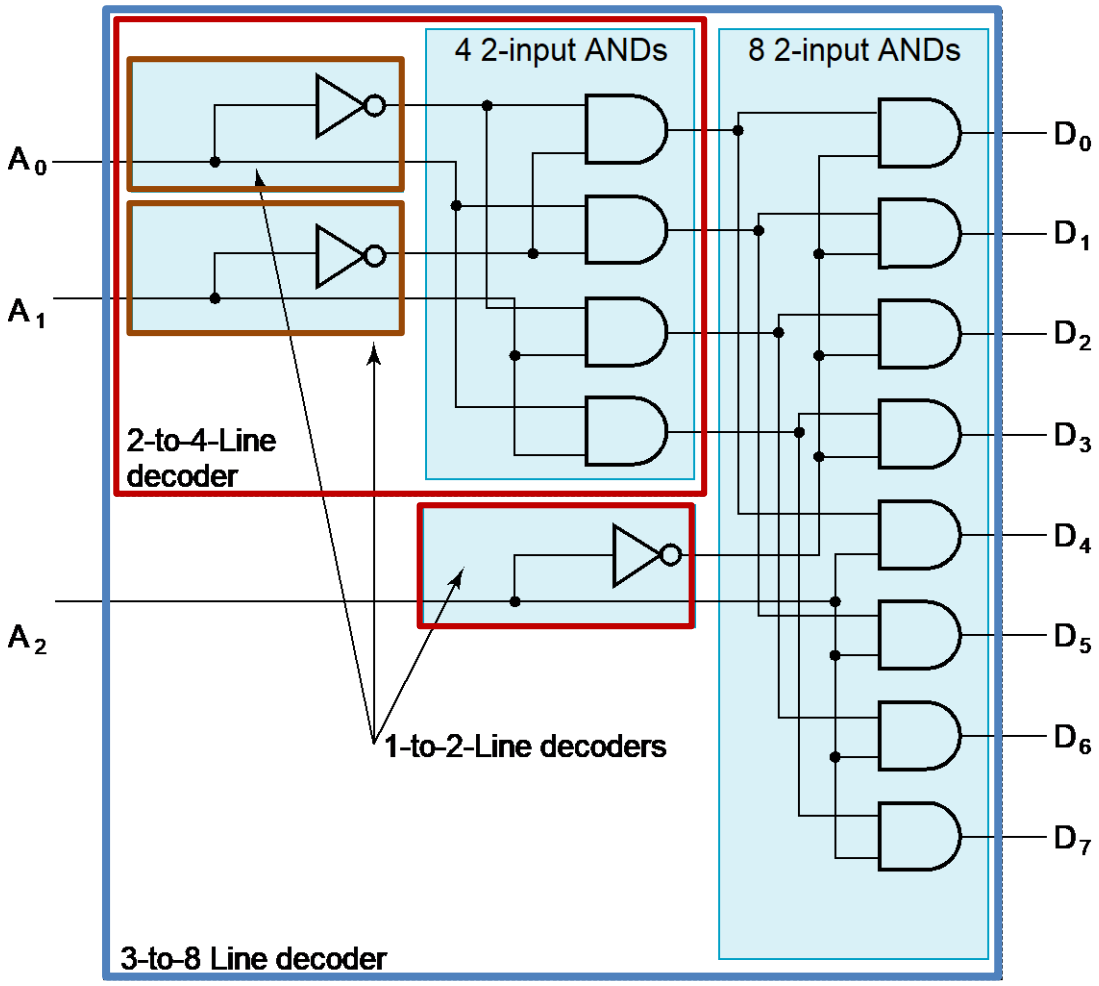
Note that the 2-to-4-line is made up of:

- 2 1-to-2 line decoders
- 4 AND gates.

Decoder Expansion - Example 1

- ✓ 3-to-8-line decoder
  - Number of output ANDs = 8
  - Number of inputs to decoder = 3
    - Closest possible split to equal
      - 2-to-4-line decoder
      - 1-to-2-line decoder
        - 2-to-4-line decoder
          - Number of output ANDs = 4
          - Number of inputs to decoder = 2
            - Closest possible split to equal
              - Two 1-to-2-line decoders

Decoder Expansion - Example 1



Decoder Expansion - Example 2

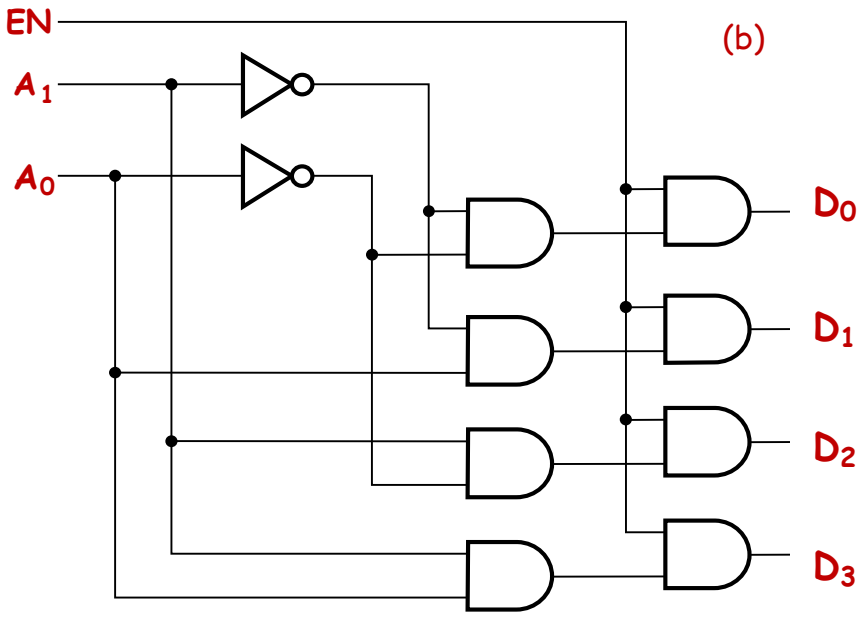
- ✓ 7-to-128-line decoder
  - Number of output ANDs = 128
  - Number of inputs to decoder = 7
    - Closest possible split to equal
    - 4-to-16-line decoder
      - Number of output ANDs = 16
      - Number of inputs to decoder = 4
        - Closest possible split to equal
        - 2 2-to-4-line decoders
    - 3-to-8-line decoder
      - Number of output ANDs = 8
      - Number of inputs to decoder = 3
        - Closest possible split to equal
        - 2-to-4-line decoder
        - 1-to-2-line decoder

Decoder with Enable

- ✓ In general, attach *m*-enabling circuits to the outputs
- ✓ See truth table below for function
  - Note use of X's to denote both 0 and 1 (don't care)
  - Combination containing two X's represent four binary combinations (compact table)
- ✓ Alternatively, can be viewed as distributing value of signal EN to 1 of 4 outputs, in this case, called a **demultiplexer**

EN	A_1	A_0	D_0	D_1	D_2	D_3
0	X	X	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

(a)



(b)

General Combinatorial Circuit implementation by a Decoder and OR Gates

- ✓ Implement *m* functions of *n* variables with:
  - Sum-of-minterms expressions (SOP)
    - One *n*-to-2^n-line decoder
    - *m* OR gates, one for each output
- ✓ Approach
    - Find the minterms for each output function
    - OR the minterms together

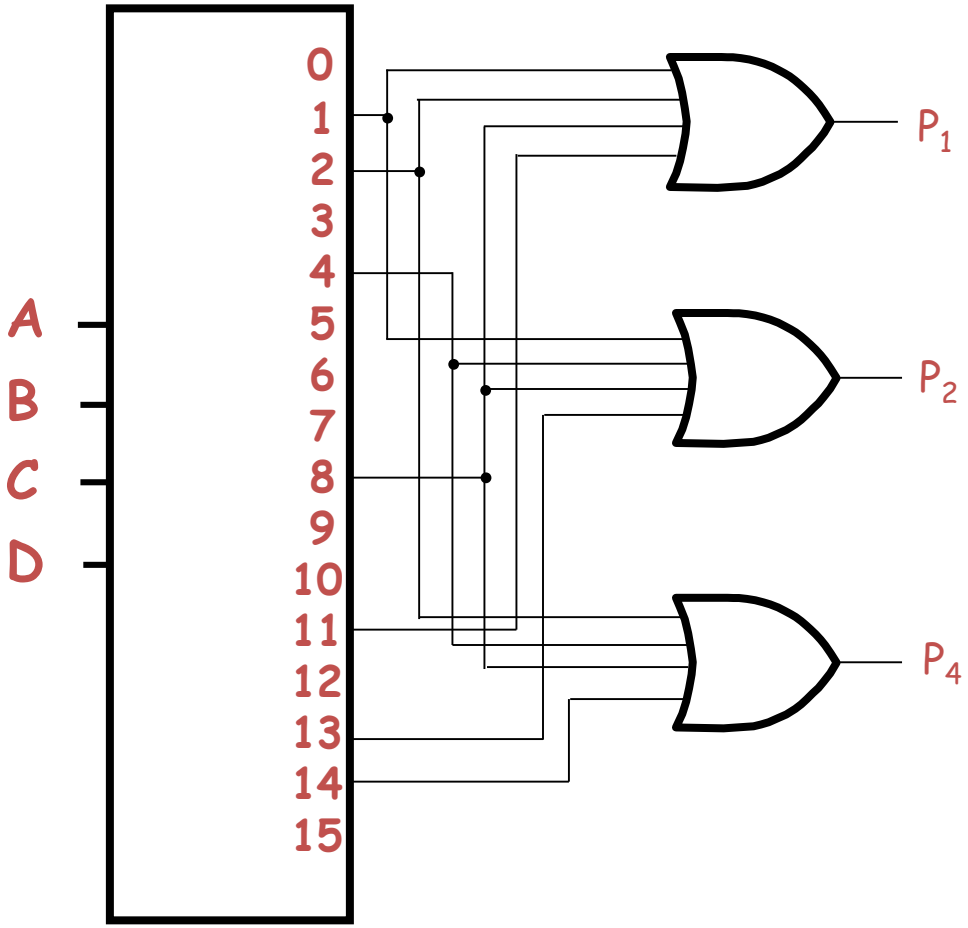
Decoder and OR Gates Example

- ✓ Implement the following set of functions of four input A, B, C, D:

$P_1 = \Sigma_m(1,2,8,11)$

$P_2 = \Sigma_m(1,4,8,13)$

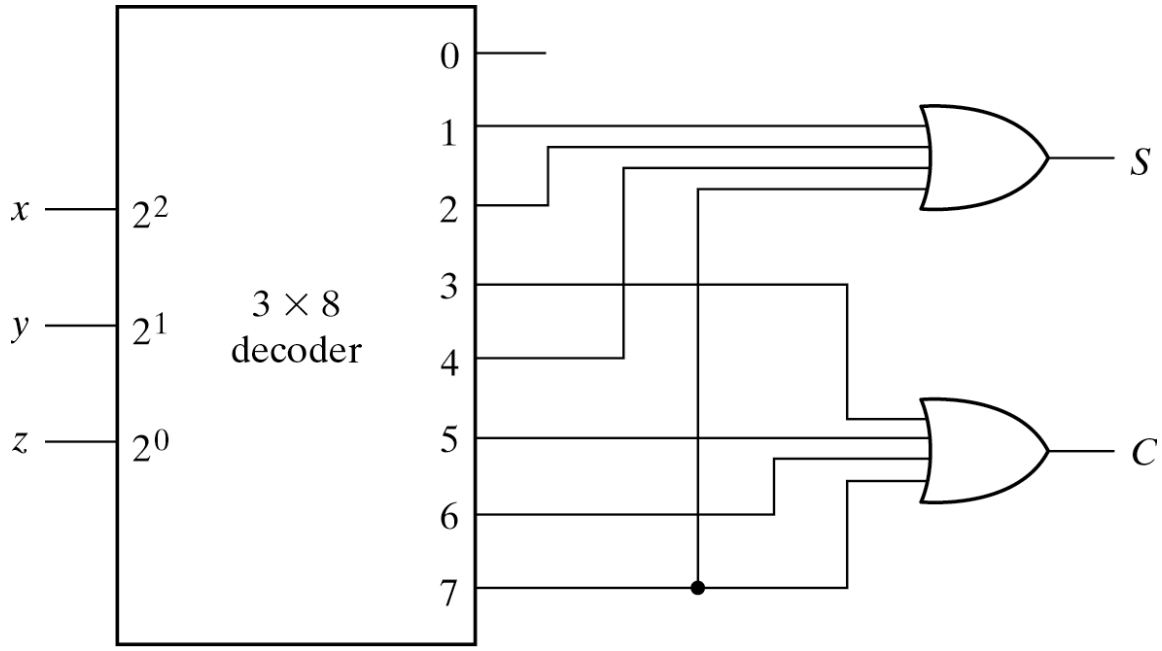
$P_4 = \Sigma_m(2,4,8,14)$



Major application of Decoder

- ✓ Decoder is use to implement any combinational circuits (f^n)
- ✓ For example the truth table for full adder is:

  $s(x,y,z)=\sum(1,2,4,7)$ and $C(x,y,z)=\sum(3,5,6,7)$.
- ✓ The implementation with decoder is:



Encoding

- ✓ Encoding - the opposite of decoding - the conversion of an m-bit input code to a n-bit output code with $n \leq m \leq 2^n$ such that each valid code word produces a unique output code
- ✓ An encoder has 2^n (or fewer) input lines and n output lines which generate the binary code corresponding to the input values
- ✓ Typically, an encoder converts a code containing exactly one bit that is 1 to a binary code corresponding to the position in which the 1 appears.

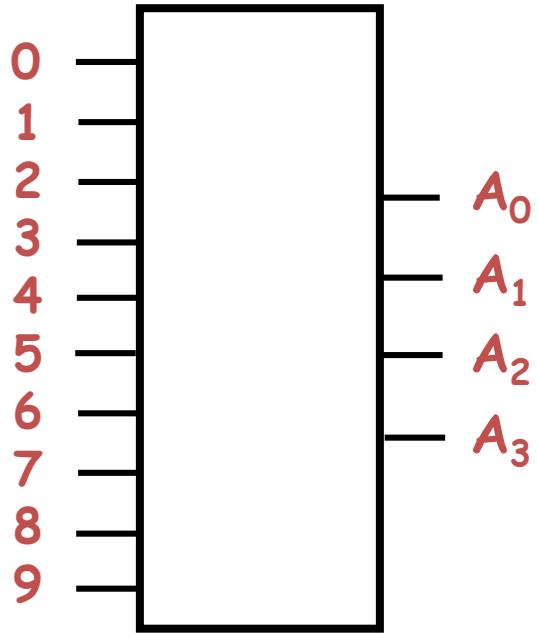
Encoder Example

- ✓ A decimal-to-BCD encoder
  - Inputs: 10 bits corresponding to decimal digits 0 through 9, (D_0, …, D_9)
  - Outputs: 4 bits with BCD codes
  - Function: If input bit D_i is a 1, then the output (A_3, A_2, A_1, A_0) is the BCD code for i
- ✓ The truth table could be formed, but alternatively, the equations for each of the four outputs can be obtained directly.

Encoder Example

- ✓ Input D_i is a term in equation A_j if bit A_j is 1 in the binary value for i.
- ✓ Equations:

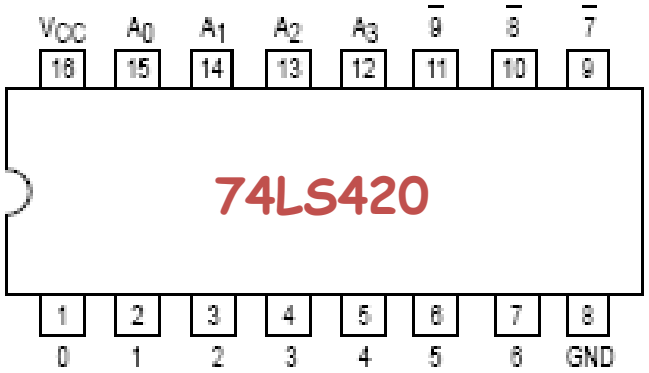
$$A_3 = D_8 + D_9$$
$$A_2 = D_4 + D_5 + D_6 + D_7$$
$$A_1 = D_2 + D_3 + D_6 + D_7$$
$$A_0 = D_1 + D_3 + D_5 + D_7 + D_9$$



- ✓ $F_1 = D_6 + D_7$ can be extracted from A_2 and A_1.

Combinational Circuits

One of Ten Decoder



TRUTH TABLE

A_0	A_1	A_2	A_3	0	1	2	3	4	5	6	7	8	9
L	L	L	L	L	H	H	H	H	H	H	H	H	H
H	L	L	L	H	L	H	H	H	H	H	H	H	H
L	H	L	L	H	H	L	H	H	H	H	H	H	H
H	H	L	L	H	H	H	L	H	H	H	H	H	H
L	L	H	L	H	H	H	H	L	H	H	H	H	H
H	L	H	L	H	H	H	H	H	L	H	H	H	H
L	H	H	L	H	H	H	H	H	H	L	H	H	H
H	H	H	L	H	H	H	H	H	H	H	L	H	H
L	L	L	H	H	H	H	H	H	H	H	H	L	H
H	L	L	H	H	H	H	H	H	H	H	H	H	L
L	H	L	H	H	H	H	H	H	H	H	H	H	H
H	H	L	H	H	H	H	H	H	H	H	H	H	H
L	L	H	H	H	H	H	H	H	H	H	H	H	H
H	L	H	H	H	H	H	H	H	H	H	H	H	H
L	H	H	H	H	H	H	H	H	H	H	H	H	H
H	H	H	H	H	H	H	H	H	H	H	H	H	H

H = HIGH Voltage Level
L = LOW Voltage Level

n-bit encoder with priority

- ✓ If one of the input lines is active the encoder produces the binary code corresponding to that line
- ✓ If more than one of the input lines is activated all the output is undefined. We can consider don't care for these situations but in many cases we consider this problem by using priority encoder.

Priority Encoder Example

- Priority encoder with 5 inputs (D_4, D_3, D_2, D_1, D_0) - highest priority to most significant 1 present - Code outputs A2, A1, A0 and V where V indicates at least one 1 present.

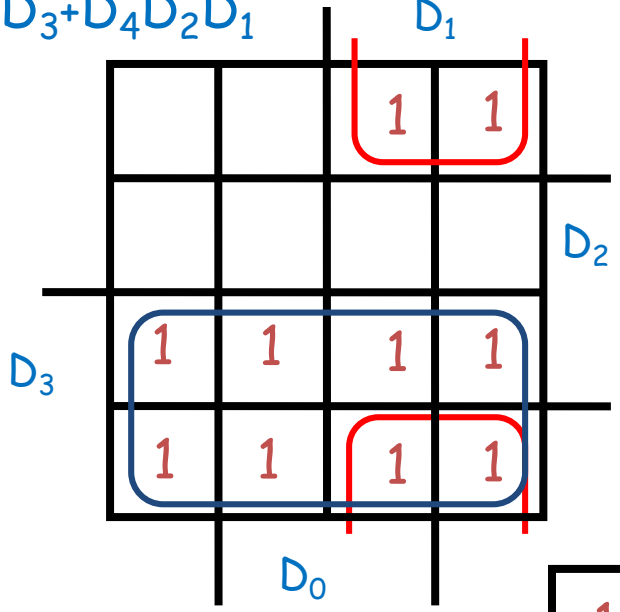
No. of Minterms	Inputs					Outputs			
	D4	D3	D2	D1	D0	A2	A1	A0	V
	0	0	0	0	0	X	X	X	0
1	0	0	0	0	1	0	0	0	1
2	0	0	0	1	X	0	0	1	1
4	0	0	1	X	X	0	1	0	1
8	0	1	X	X	X	0	1	1	1
16	1	X	X	X	X	1	0	0	1

- Xs in input part of table represent 0 or 1 (don't care); thus table entries correspond to product terms instead of minterms. The column on the left shows that all 32 minterms are present in the product terms in the table

Priority Encoder Example (continued)

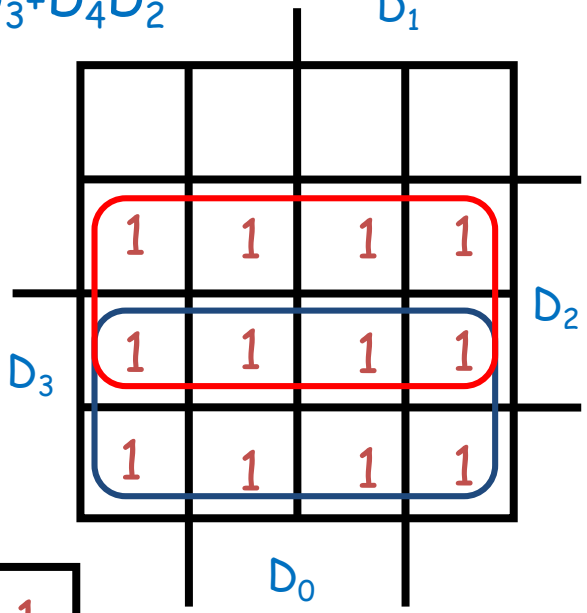
$D_4 = 0$

$A_0 = \bar{D}_4 D_3 + \bar{D}_4 \bar{D}_2 D_1$



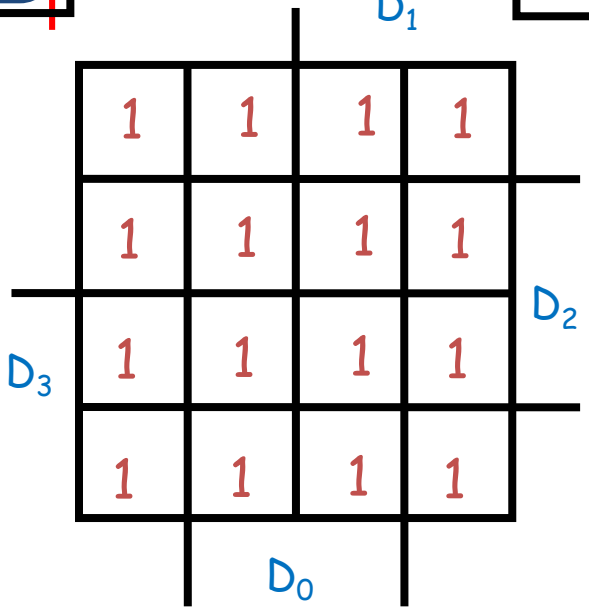
$D_4 = 0$

$A_1 = \bar{D}_4 D_3 + \bar{D}_4 D_2$



$D_4 = 1$

$A_2 = D_4$



No. of Minterms	Inputs					Outputs			
	D4	D3	D2	D1	D0	A2	A1	A0	V
	0	0	0	0	0	X	X	X	0
1	0	0	0	0	1	0	0	0	1
2	0	0	0	1	X	0	0	1	1
4	0	0	1	X	X	0	1	0	1
8	0	1	X	X	X	0	1	1	1
16	1	X	X	X	X	1	0	0	1

Priority Encoder Example (continued)

- ✓ Could use a K-map to get equations, but can be read directly from table and manually optimized if careful:

$A_2 = D_4$

$A_1 = \overline{D}_4 D_3 + \overline{D}_4 D_2 = \overline{D}_4 F_1,\ F_1 = (D_3 + D_2)$

$A_0 = \overline{D}_4 D_3 + \overline{D}_4 \overline{D}_2 D_1$

$V = D_4 + F_1 + D_1 + D_0$

Multiplexer

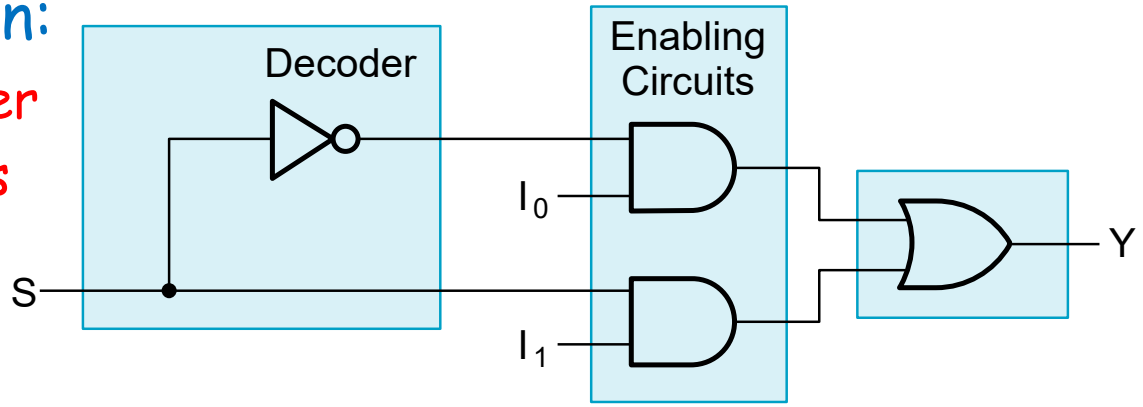
- ✓ It is a combinational circuit that selects binary information from one of the input lines and directs it to a single output line
- ✓ For example for 2-to-1 multiplexer if selection S is zero then I_0 has the path to output and if S is one I_1 has the path to output
- ✓ A typical multiplexer has *n* control inputs (S_{n-1}, … S_0) called selection inputs, 2^n information inputs (I_{2^n-1}, … I_0), and one output Y
- ✓ A multiplexer can be designed to have *m* information inputs with $m < 2^n$ as well as *n* selection inputs

2-to-1-Line Multiplexer

- ✓ Since $2 = 2^1$, $n = 1$
- ✓ The single selection variable S has two values:
  - S = 0 selects input I_0
  - S = 1 selects input I_1
- ✓ The equation:

$$Y = \overline{S} I_0 + S I_1$$

- ✓ Circuit composition:
  - 1-to-2-line Decoder
  - 2 Enabling circuits
  - 2-input OR gate

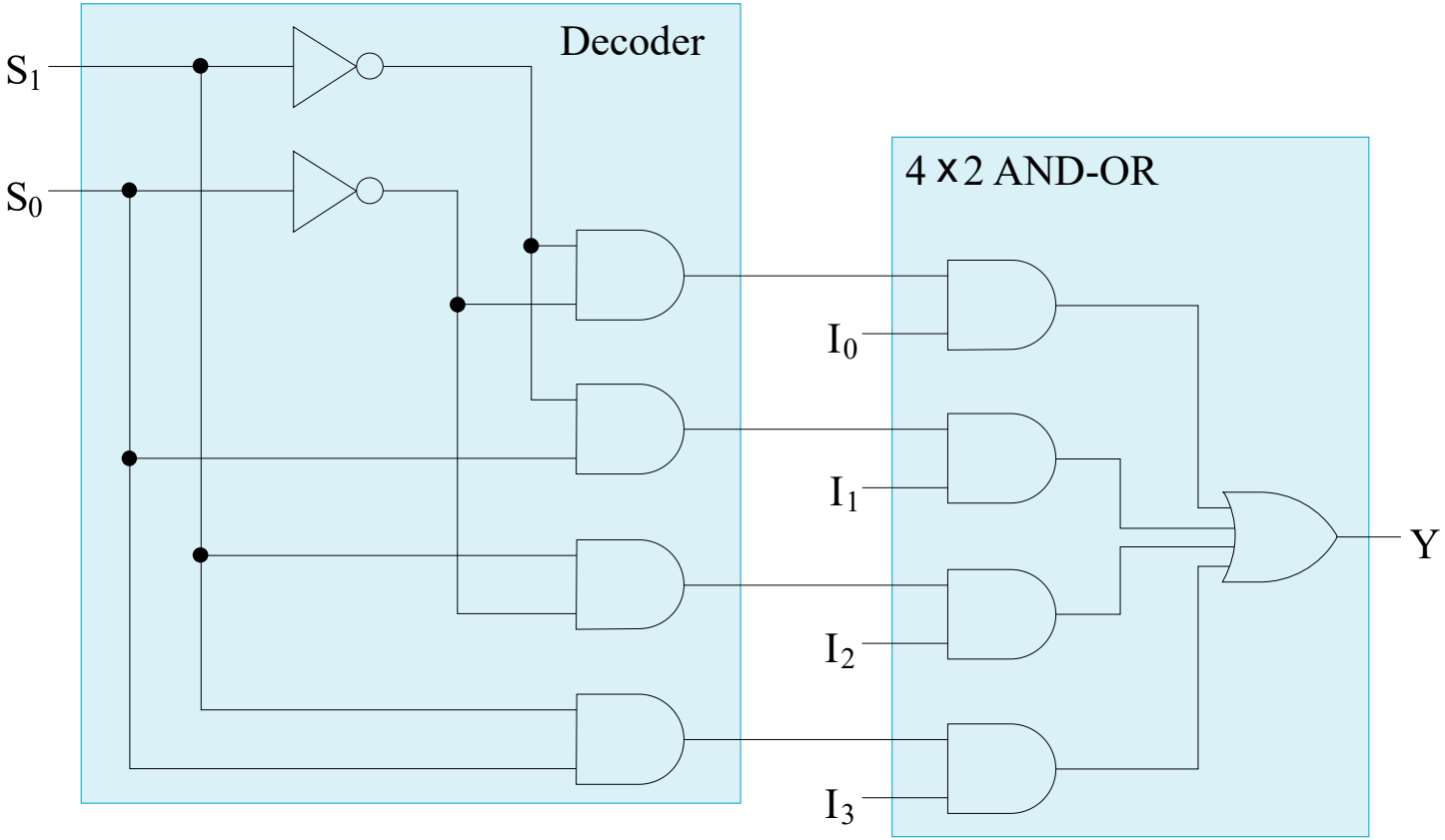


2-to-1-Line Multiplexer (continued)

- ✓ To obtain a basis for multiplexer expansion, we combine the Enabling circuits and OR gate into an AND-OR circuit:
  - 1-to-2-line decoder
  - 2×2 AND-OR
- ✓ In general, for an 2^n-to-1-line multiplexer:
  - n-to-2^n-line decoder
  - $2^n \times 2$ AND-OR

Example: 4-to-1-line Multiplexer

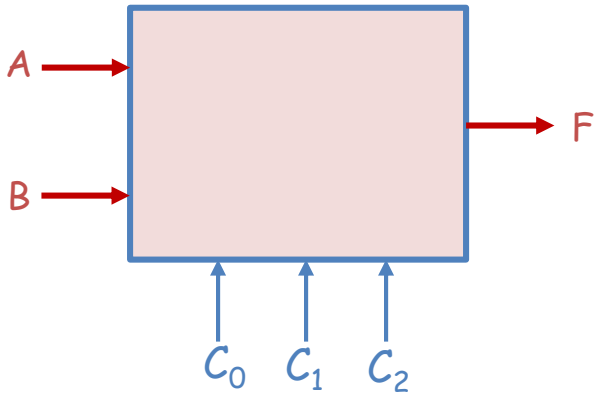
- ✓ 2-to-2^2-line decoder
- ✓ $2^2 \times 2$ AND-OR



Logical Function Unit: with a Multiplexer block

- ✓ Example: Multi-purpose Function Block
  - 3 control inputs to specify operation to perform on operands
  - 2 data inputs for operands
  - 1 output of the same bit-width as operands

3 control inputs: C_0, C_1, C_2
2 data inputs: A, B
1 output: F

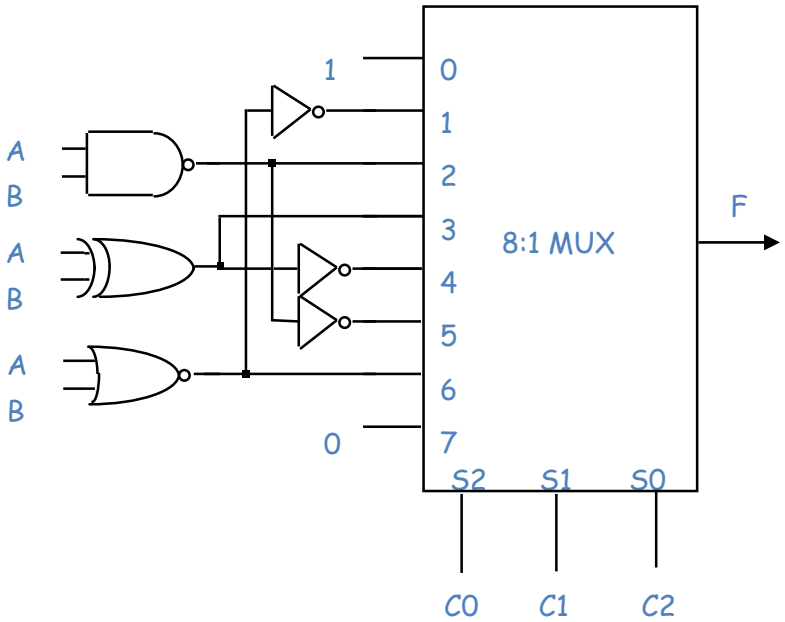


C_0	C_1	C_2	Function	Comments
0	0	0	1	always 1
0	0	1	A + B	logical OR
0	1	0	(A • B)'	logical NAND
0	1	1	A xor B	logical xor
1	0	0	A xnor B	logical xnor
1	0	1	A • B	logical AND
1	1	0	(A + B)'	logical NOR
1	1	1	0	always 0

Multiplexer implementation

C_0	C_1	C_2	A	B	F
0	0	0	0	0	1
0	0	0	0	1	1
0	0	0	1	0	1
0	0	0	1	1	1
0	0	1	0	0	0
0	0	1	0	1	1
0	0	1	1	0	1
0	0	1	1	1	1
0	1	0	0	0	1
0	1	0	0	1	1
0	1	0	1	0	1
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	0	1	1
0	1	1	1	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	0	0	1	0
1	0	0	1	0	0
1	0	0	1	1	1
1	0	1	0	0	0
1	0	1	0	1	0
1	0	1	1	0	0
1	0	1	1	1	1
1	1	0	0	0	1
1	1	0	0	1	0
1	1	0	1	0	0
1	1	0	1	1	0
1	1	1	0	0	0
1	1	1	0	1	0
1	1	1	1	0	0
1	1	1	1	1	0

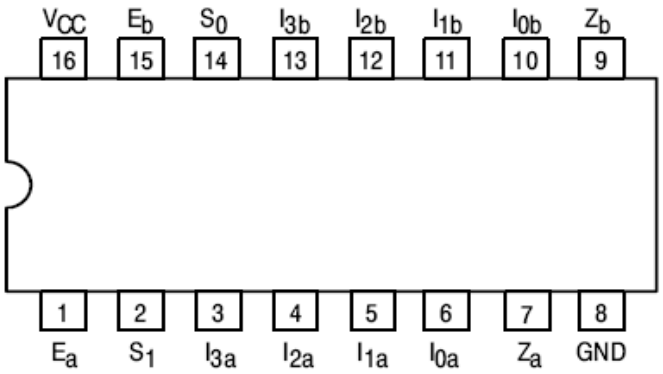
Alternative implementation to the 5-variable K-map: by discrete gates and multiplexer implementation



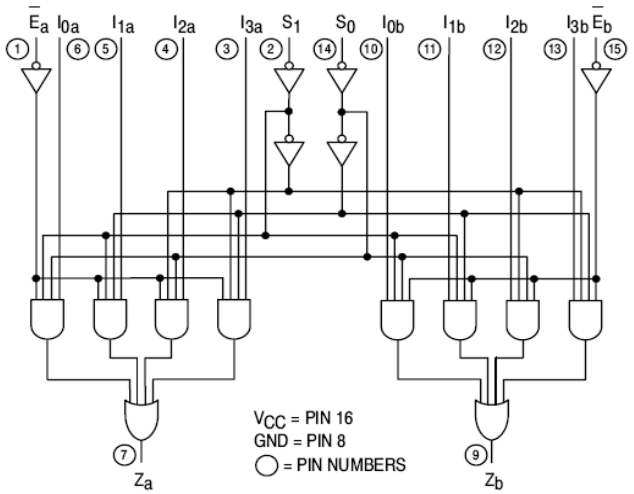
Multiplexer

SN74LS153

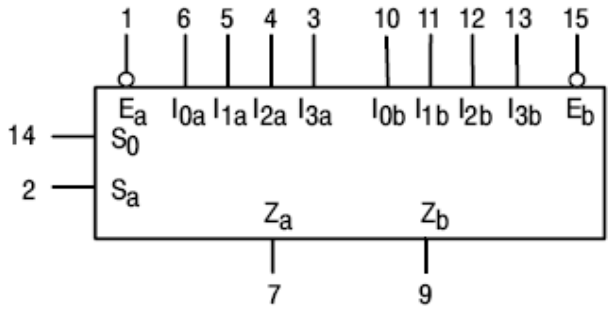
CONNECTION DIAGRAM DIP (TOP VIEW)



LOGIC DIAGRAM



LOGIC SYMBOL



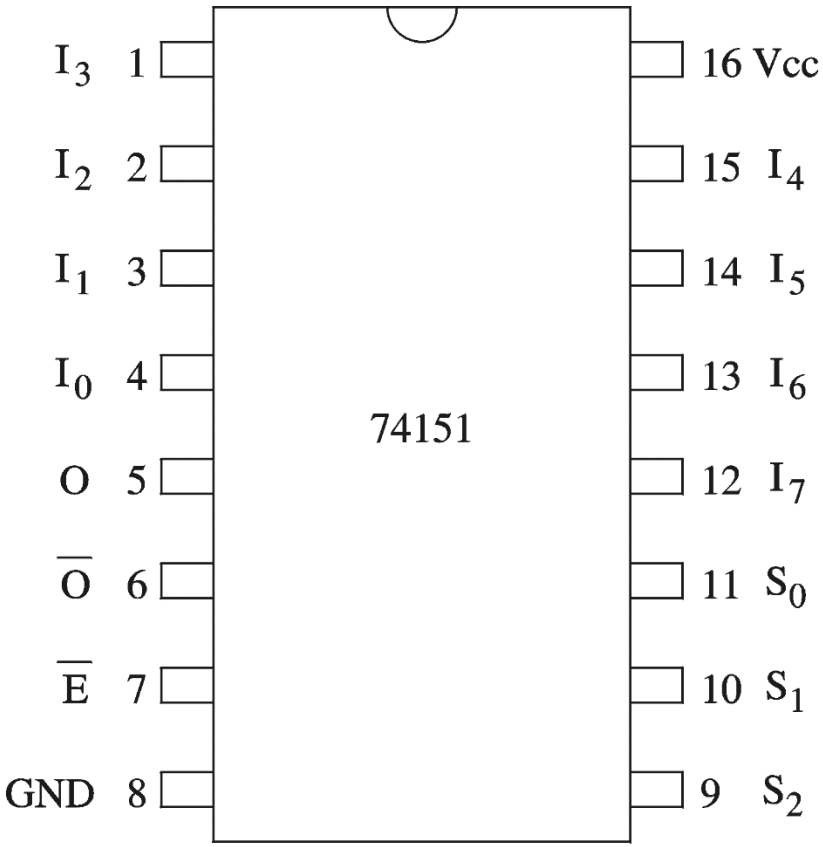
TRUTH TABLE

SELECT INPUTS		INPUTS (a or b)					OUTPUT
S_0	S_1	$\overline{E}$	I_0	I_1	I_2	I_3	Z
X	X	H	X	X	X	X	L
L	L	L	L	X	X	X	L
L	L	L	H	X	X	X	H
H	L	L	X	L	X	X	L
H	L	L	X	H	X	X	H
L	H	L	X	X	L	X	L
L	H	L	X	X	H	X	H
H	H	L	X	X	X	L	L
H	H	L	X	X	X	H	H

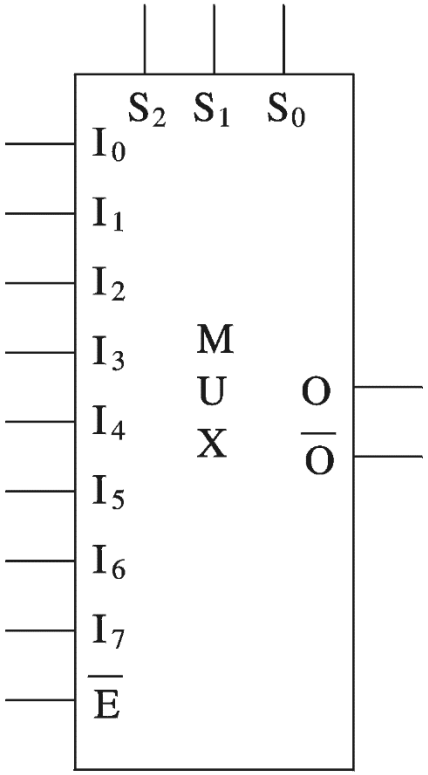
H = HIGH Voltage Level
L = LOW Voltage Level
X = Don't Care

V_{CC} = PIN 16
GND = PIN 8

Multiplexers (8-to-1 MUX)



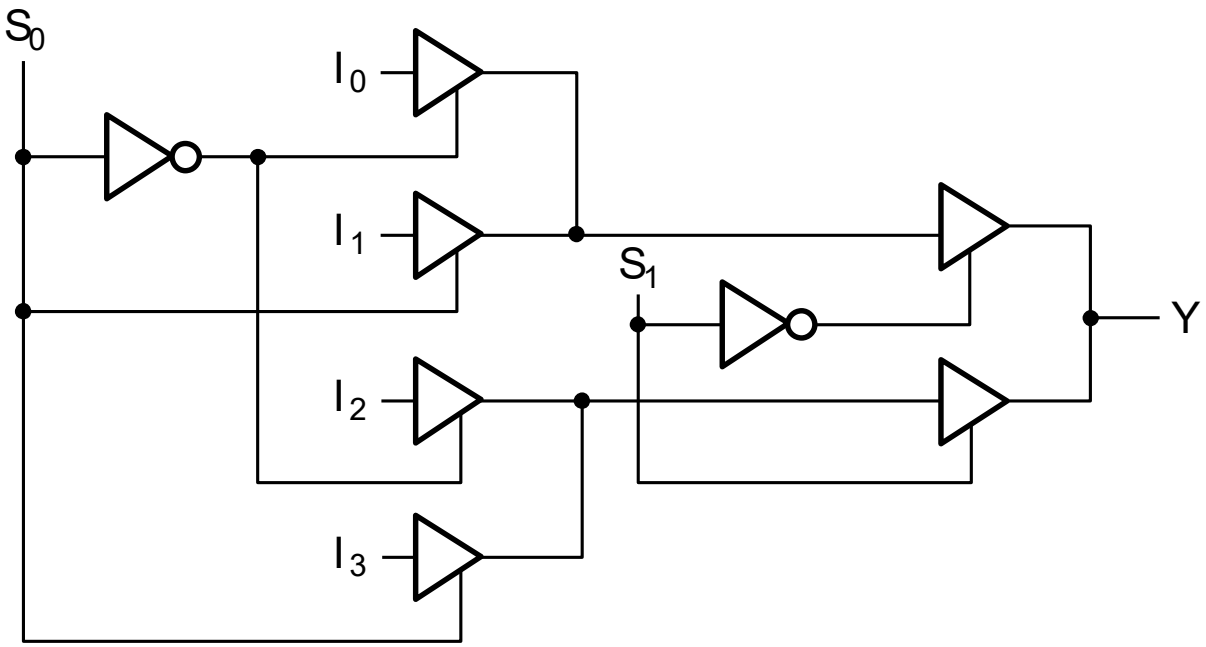
(a) Connection diagram



(b) Logic symbol

Other Multiplexer Implementations

- ✓ Three-state logic in place of AND-OR

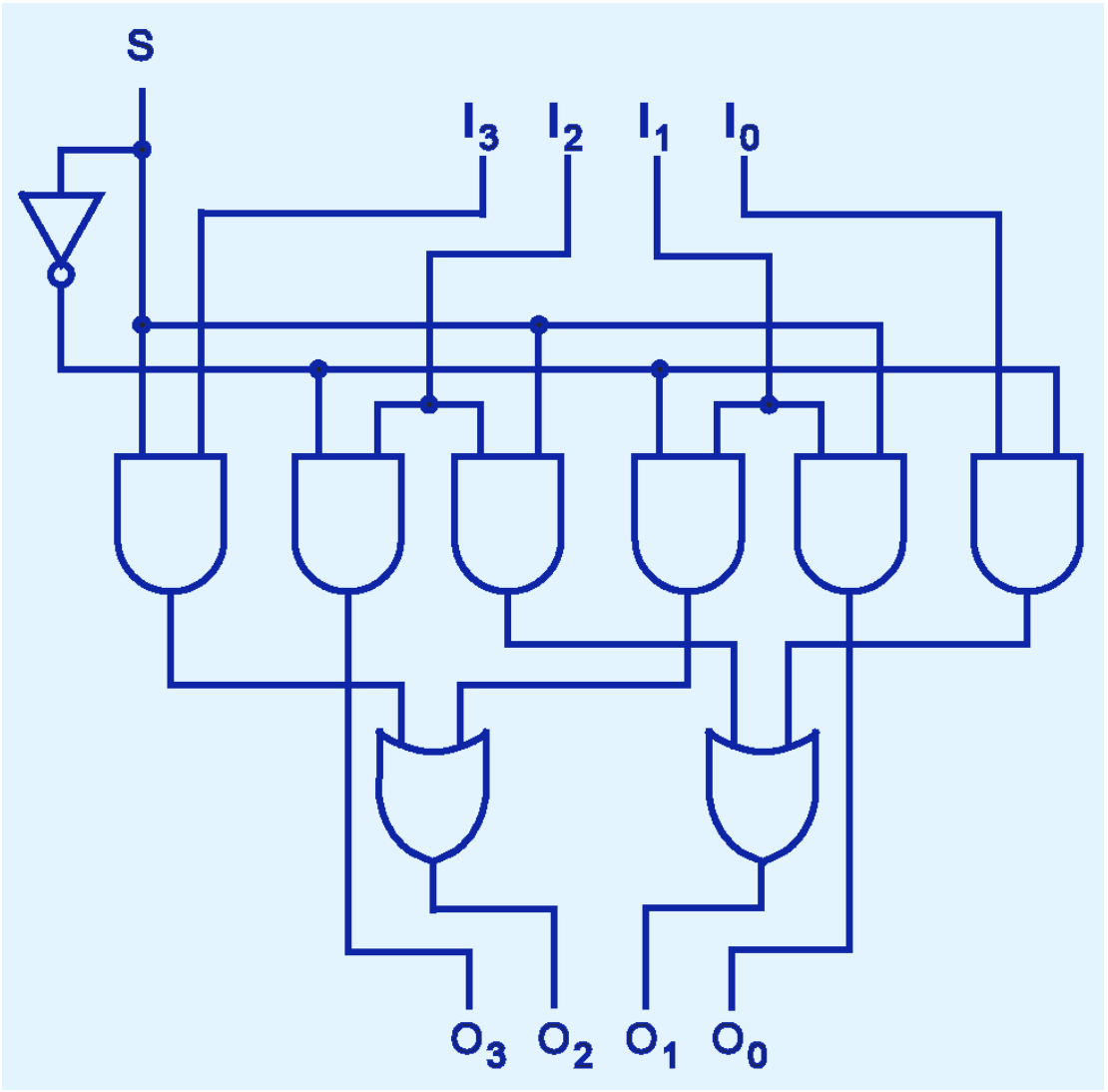


(b)

- ✓ Gate input cost = 14 compared to 22 for gate implementation

Shift Register

- ✓ This shifter moves the bits of a nibble one position to the left or right?



If S = 0, in which direction do the input bits shift?

Shift Register

MODE SELECT — TRUTH TABLE

OPERATING MODE	INPUTS			OUTPUTS	
	$\overline{MR}$	A	B	Q_0	Q_1–Q_7
Reset (Clear)	L	X	X	L	L – L
Shift	H	l	l	L	q_0 – q_6
	H	l	h	L	q_0 – q_6
	H	h	l	L	q_0 – q_6
	H	h	h	H	q_0 – q_6