Logic and Computer Design Fundamentals

Chapter 3 – Combinational Logic Design

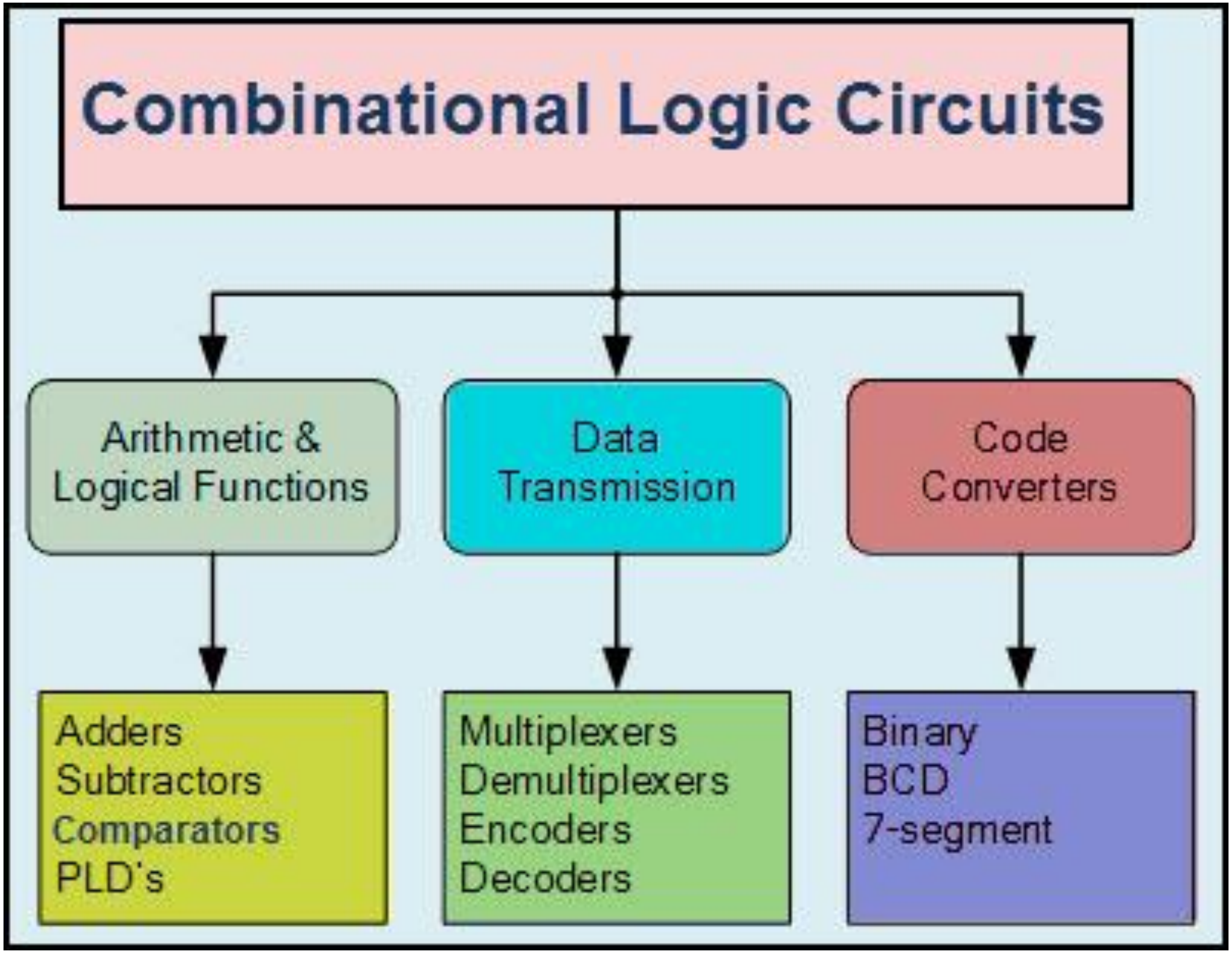
Part 2 – Combinational Functional Blocks

Ming Cai
cm@zju.edu.cn
College of Computer Science and Technology
Zhejiang University

Overview

- Part 2 – Combinational Logic
  - Functions and functional blocks
  - Rudimentary logic functions
  - Decoding using Decoders
    - Implementing Combinational Functions with Decoders
  - Encoding using Encoders
  - Selecting using Multiplexers
    - Implementing Combinational Functions with Multiplexers

Classification of Combinational Logic



Functions and Functional Blocks

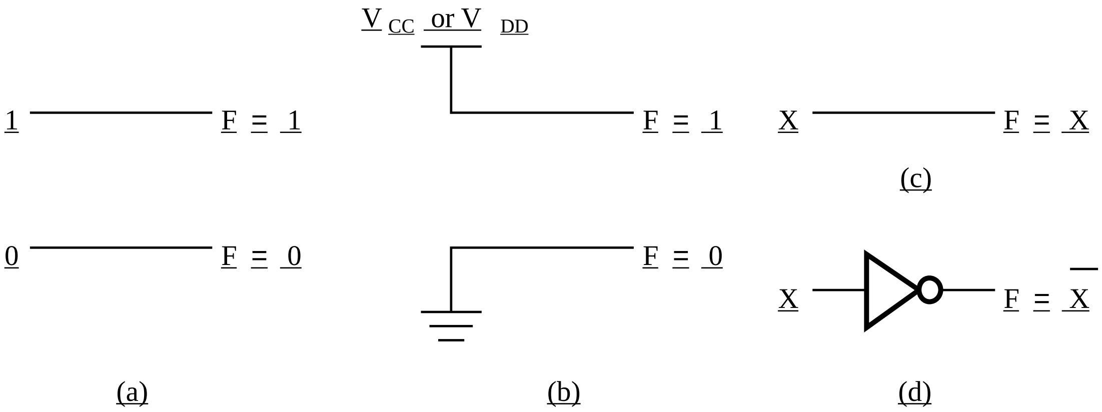
- The functions considered are those found to be very useful in design.
- Corresponding to each of the functions is a combinational circuit implementation called a ***functional block***.
- In the past, functional blocks were packaged as small-scale-integrated (SSI), medium-scale integrated (MSI), and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

Rudimentary Logic Functions

- **Four elementary combinational logic functions**
  - **Value-Fixing**: $F = 0$ **or** $F = 1$, *no Boolean operator*
  - **Transferring**: $F = X$, *no Boolean operator*
  - **Inverting**: $F = \overline{X}$, *involves one logic gate*
  - **Enabling**: $F = X \cdot EN$ *or* $F = X + \overline{EN}$, *involves one or two logic gates*
  - **The first three are functions of a single variable X**

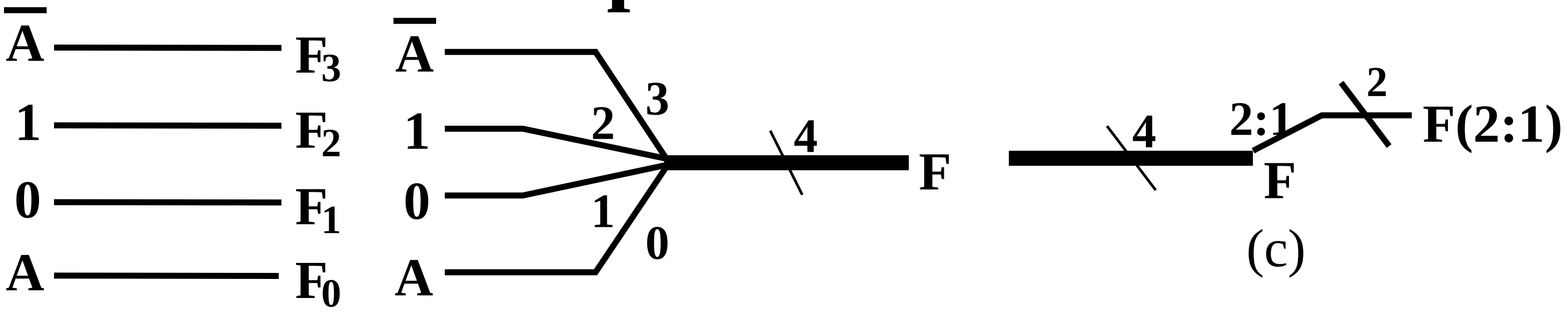
Table Functions of one variable

X	F =0	F =X	F = $\overline{X}$	F=1
0	0	0	1	1
1	0	1	0	1

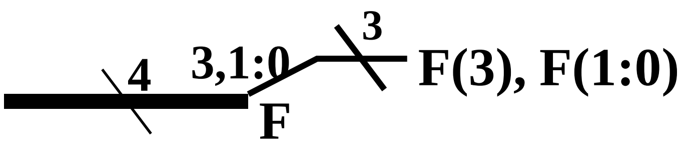


Multiple-bit Rudimentary Functions

- **Multi-bit Examples:**



(a) (b)

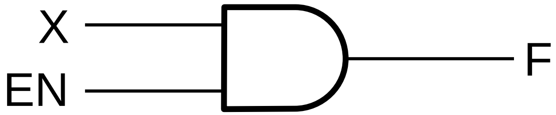


(d)

- **A wide line is used to represent a *bus* which is a vector signal**
- **In (b) of the example, F = (F_3, F_2, F_1, F_0) is a bus.**
- **The bus can be split into individual bits as shown in (b)**
- **Sets of bits can be split from the bus as shown in (c) for bits 2 and 1 of F.**
- **The sets of bits need not be continuous as shown in (d) for bits 3, 1, and 0 of F.**

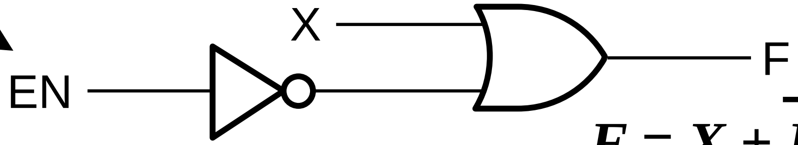
Enabling Function

- ***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**
- **The value on the output when it is disable can be 0, 1, or Hi-Z (as for three-state buffers and transmission gates)**
- **When disabled, 0 output**
- **When disabled, 1 output**



(a)

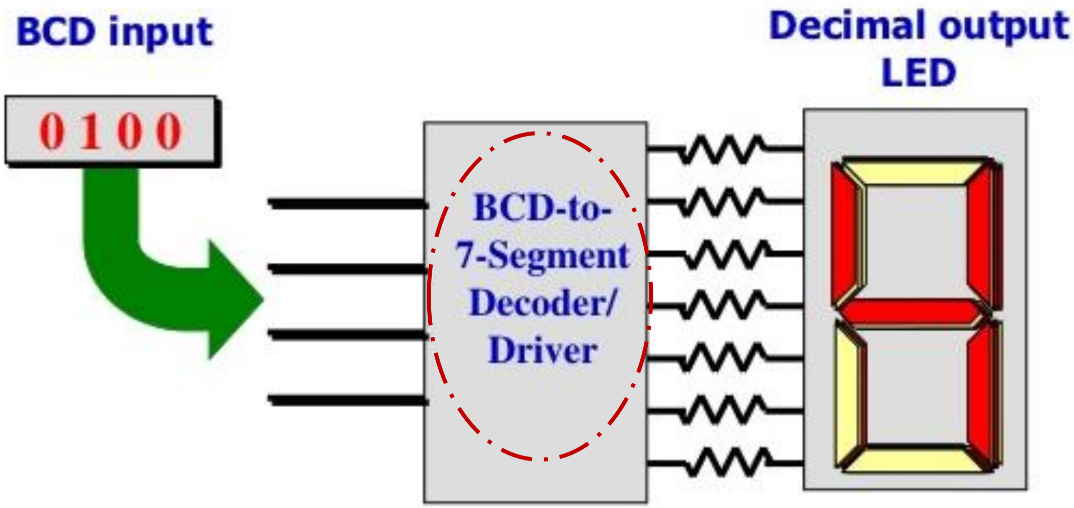
$F = X \cdot EN$



$F = X + \overline{EN}$

(b)

Decoding

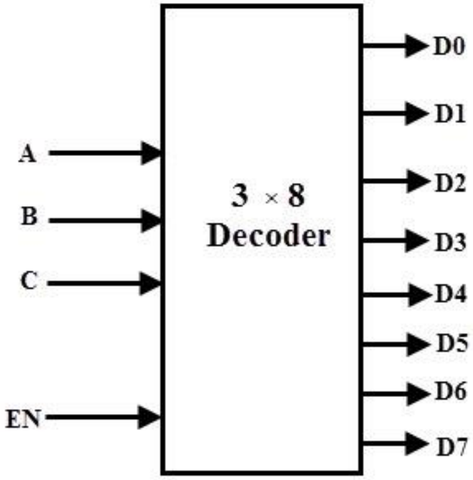


BCD to 7-segment decoder

- **Decoding - the conversion of an *n*-bit input code to an *m*-bit output code with n ≤ m ≤ 2*ⁿ* such that each valid code word produces a unique output code**
- **Circuits that perform decoding are called *decoders***

Types of decoder

- Types of decoder
  - Variable Decoder (minterm detector)
  - Display Decoder
  - Code Translation Decoder



3-to-8 binary decoder

- Commonly used decoders:
  - decoder with 2 input and 4 output, 74LS139 (2-to-4-Line Decoder)
  - decoder with 3 input and 8 output, 74LS138 (3-to-8-Line Decoder)
  - decoder with 4 input and 16 output, MC14514(4-to-16-Line Decoder)

Decoder

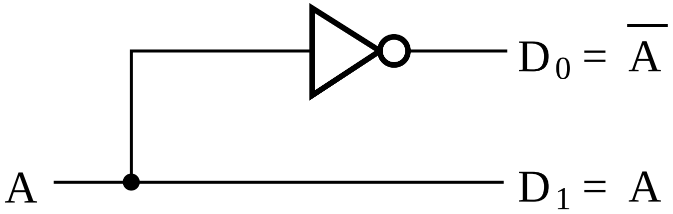
A	B	C	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Decoder Examples

- **1-to-2-Line Decoder**

A	D_0	D_1
0	1	0
1	0	1

(a)

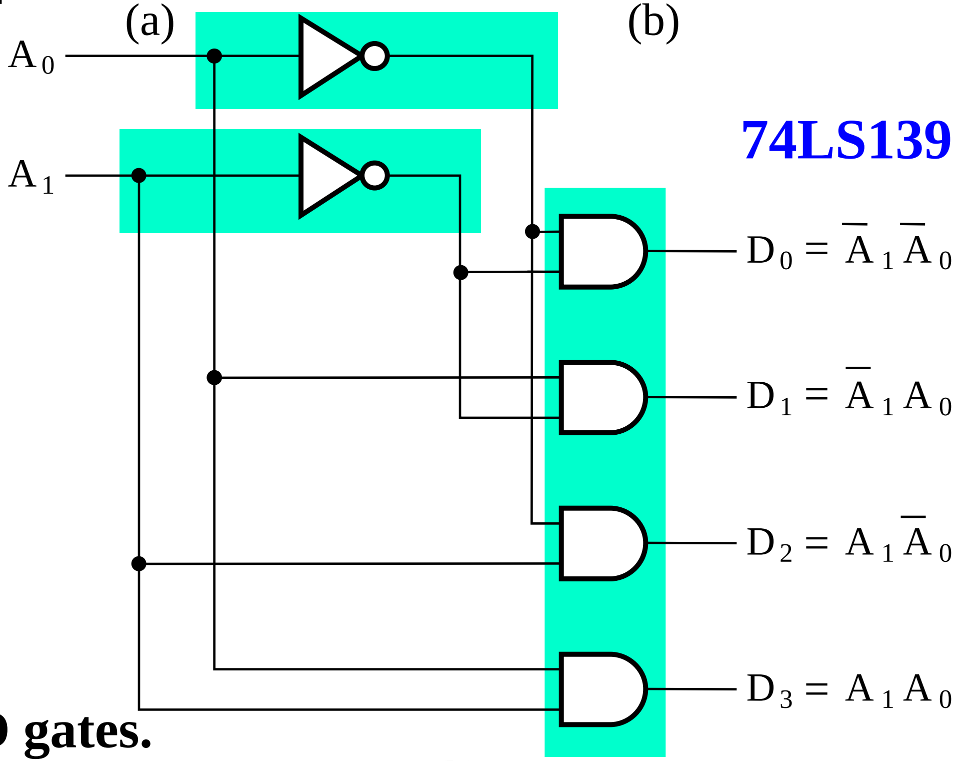


(b)

- **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)



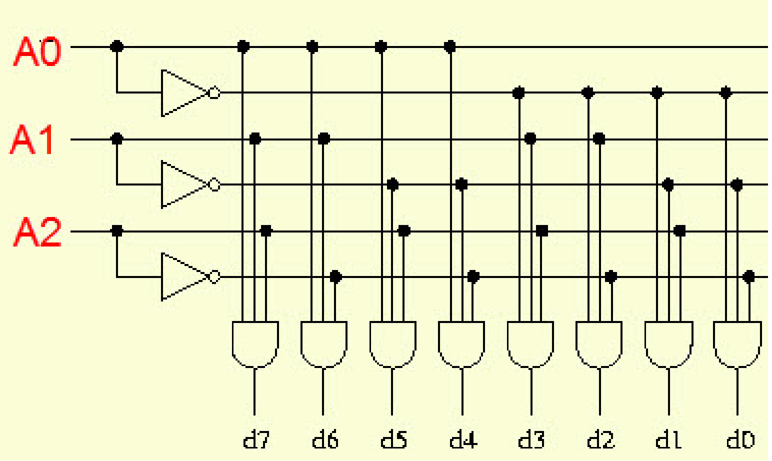
A_0
A_1
$D_0 = \overline{A}_1\overline{A}_0$
$D_1 = \overline{A}_1 A_0$
$D_2 = A_1\overline{A}_0$
$D_3 = A_1 A_0$

74LS139

(b)

- **Note that the 2-4-line made up of 2 1-to-2-line decoders and 4 AND gates.**

Decoder Expansion



$$D_0 = \overline{A_2}\,\overline{A_1}\,\overline{A_0} \qquad D_4 = A_2\,\overline{A_1}\,\overline{A_0}$$

$$D_1 = \overline{A_2}\,\overline{A_1}\,A_0 \qquad D_5 = A_2\,\overline{A_1}\,A_0$$

$$D_2 = \overline{A_2}\,A_1\,\overline{A_0} \qquad D_6 = A_2\,A_1\,\overline{A_0}$$

$$D_3 = \overline{A_2}\,A_1\,A_0 \qquad D_7 = A_2\,A_1\,A_0$$

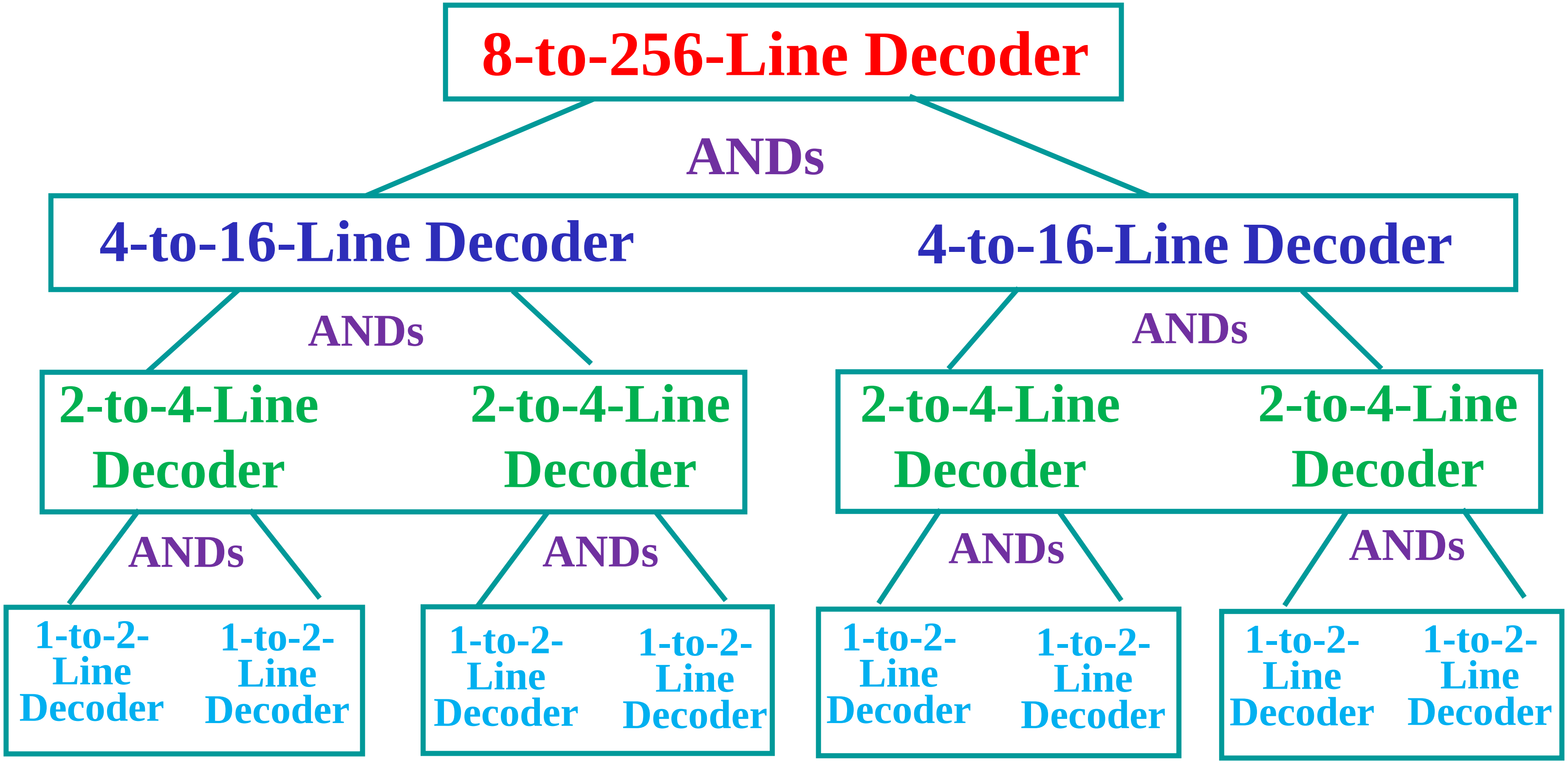
3-to-8-Line Decoder

GN = 3+3×8 = 27

- **Large decoders can be constructed by implementing each minterm function using a single AND gate with more inputs.**
- **Unfortunately, as decoders become larger, this approach gives a high fan-in and gate-input cost.**
- **We give a procedure that uses design hierarchy and collections of AND gates to construct any decoder with a lower fan-in and gate-input cost.**

Decoder Expansion (continued)

- This procedure applies divide-and-conquer strategy to build a large decoder.



Decoder Expansion (continued)

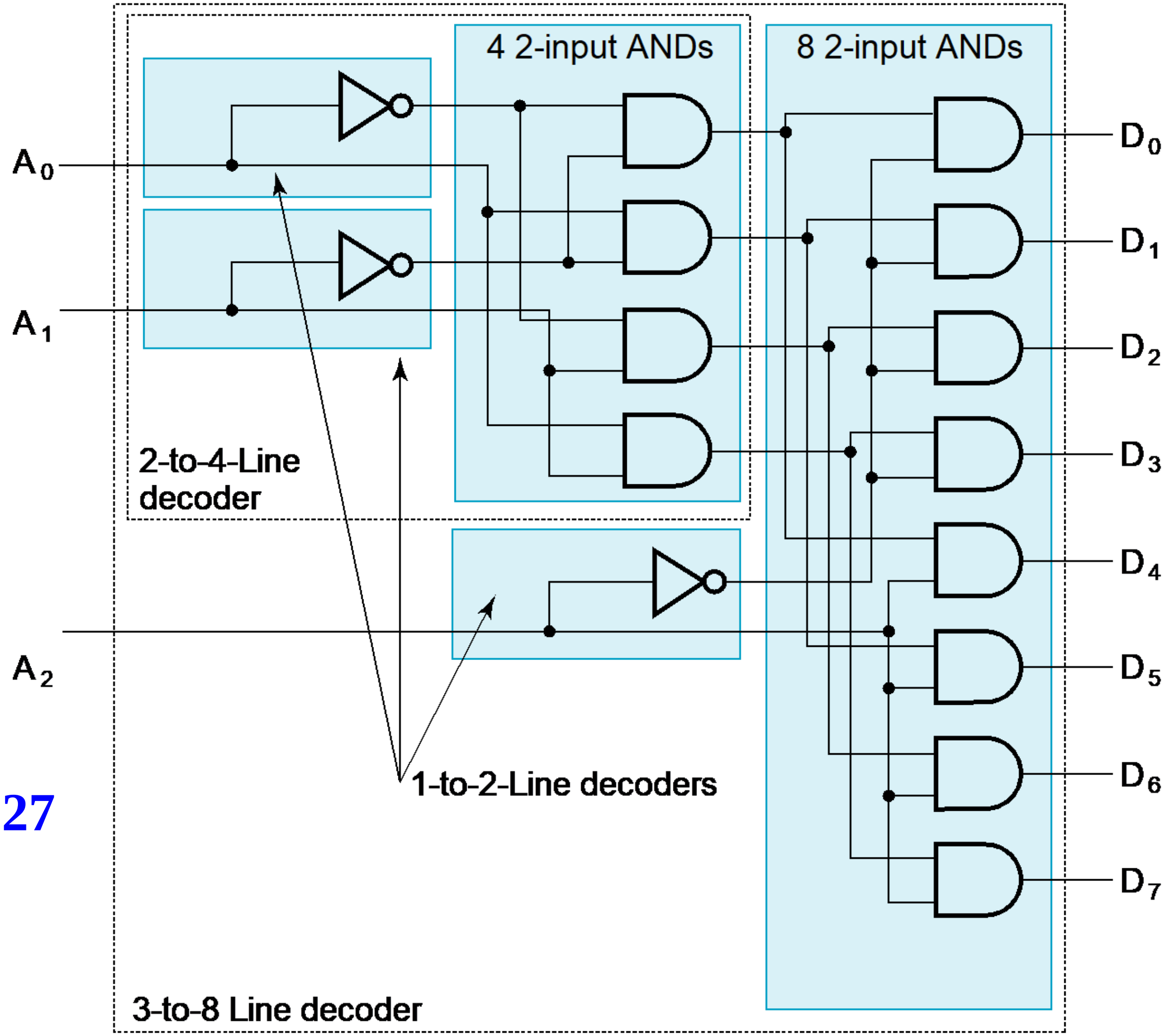
- **This procedure builds a multi-level decoder backward from the outputs:**
  1. **The output AND gates are driven by two decoders with their numbers of inputs either equal or differing by 1.**
  2. **These decoders are then designed using the same procedure until 2-to-1-line decoders are reached.**
- **The procedure can be modified to apply to decoders with the number of outputs $\neq 2^n$**

Decoder Expansion - Example 1

- **3-to-8-Line Decoder**
  - **Number of output ANDs = 8**
  - **Hierarchically, divide the input signals equally**
    - **2-to-4-Line decoder**
    - **1-to-2-Line decoder**
- **2-to-4-Line Decoder**
  - **Number of output ANDs = 4**
  - **Divide the input signals equally**
    - **Two 1-to-2-Line decoder**

Decoder Expansion - Example 1

- Result

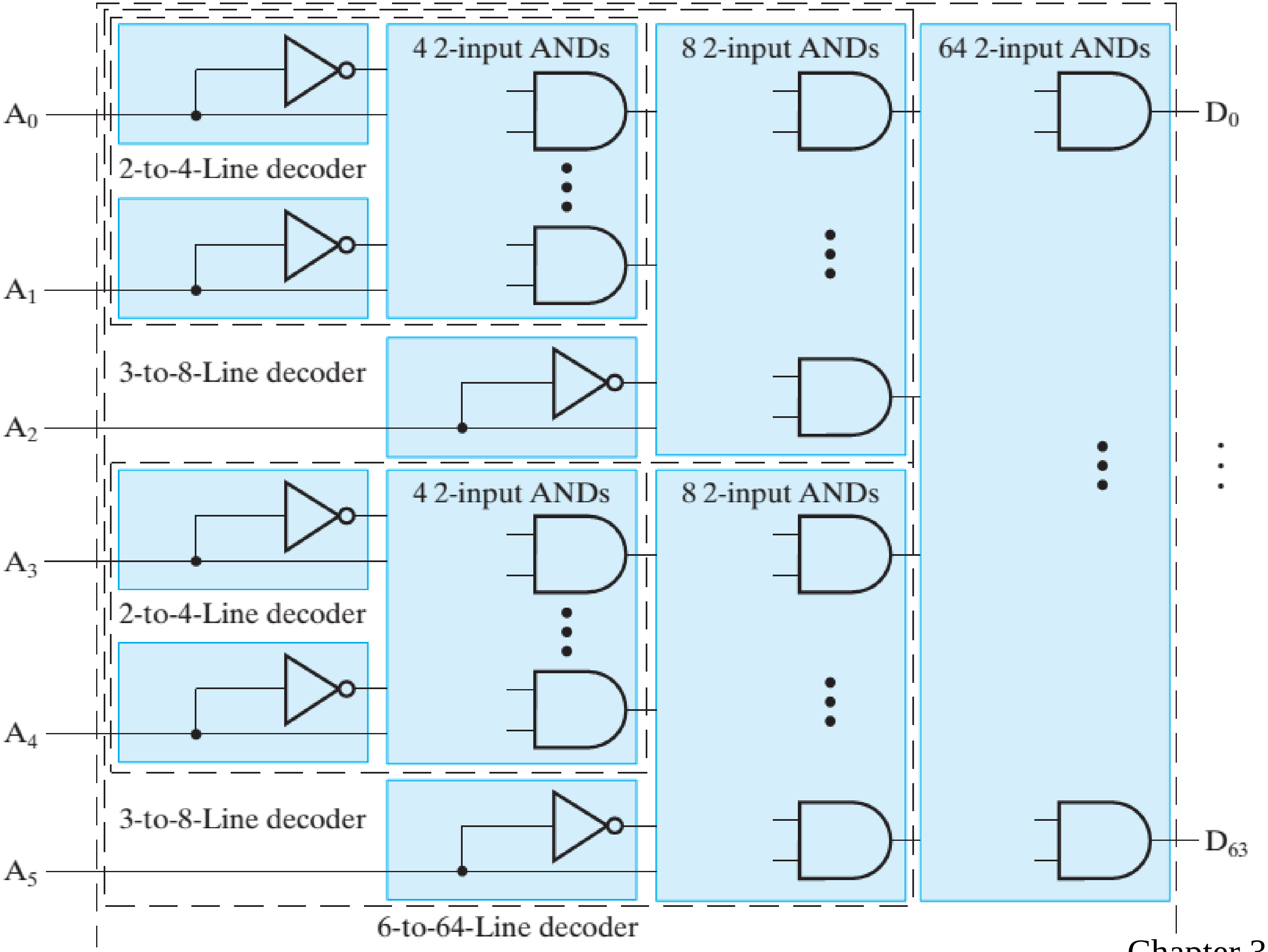


GN = 3+2 ×8+2 ×4=27

Decoder Expansion - Example 2

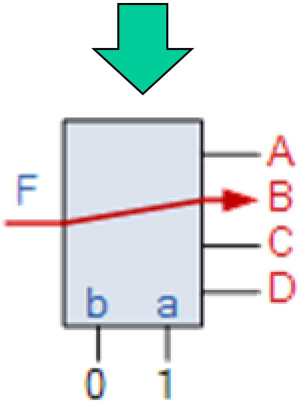
- **6-to-64-line decoder**
  - **Number of inputs to decoders driving output ANDs = 6**
  - **Number of output ANDs = 64**
  - **Closest possible split to equal**
    - **3-to-8-line decoder**
    - **3-to-8-line decoder**
  - **Complete using known 3-to-8 line and 2-to-4 line decoders**
- **For gate input cost**
  - **GN = 6+6 ×64 = 390** **One-level method**
  - **GN = 6+2 ×64+2 ×2 ×8+2 ×2 ×4 = 182** **Three-level method**

6-to-64-line decoder



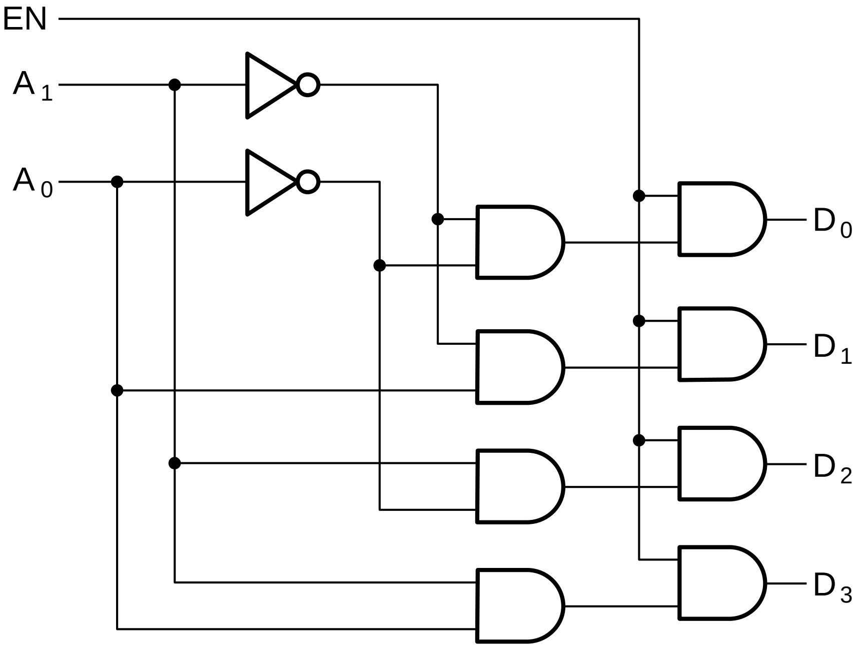
Decoder with Enable

- 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
  - Combination containing two X’s represent four binary combinations
- Alternatively, can be viewed as distributing value of signal EN to 1 of 4 outputs
- In this case, called a *demultiplexer*



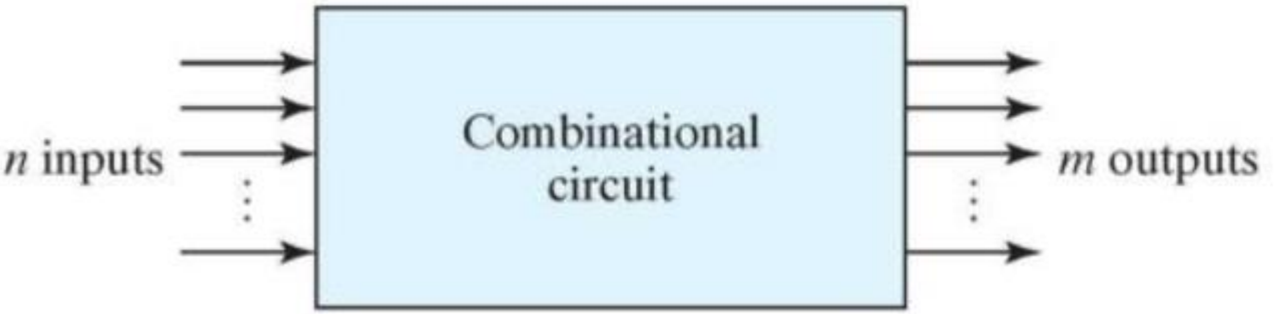
EN	A_1	A_0	D_0	D_1	D_2	D_3
0	X	X	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

(a)



(b)

Combinational Logic Implementation - Decoder and OR Gates



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

Decoder and OR Gates Example

- **Implement a binary Adder**
- **Finding sum of minterms expressions**

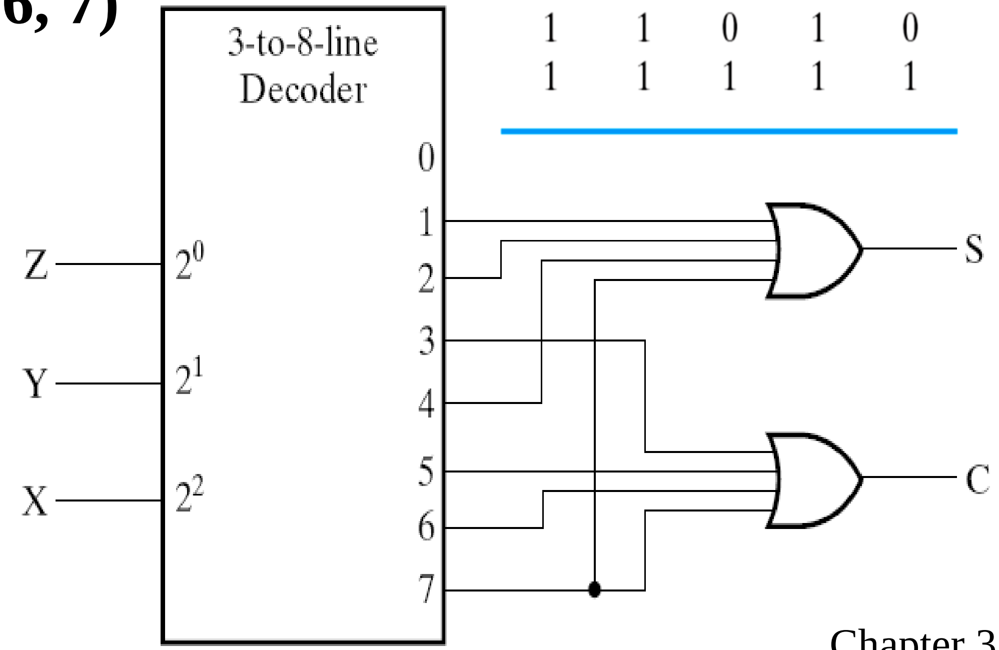
$S(X, Y, Z) = \Sigma_m(1, 2, 4, 7)$

$C(X, Y, Z) = \Sigma_m(3, 5, 6, 7)$

Truth Table

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

Find circuit



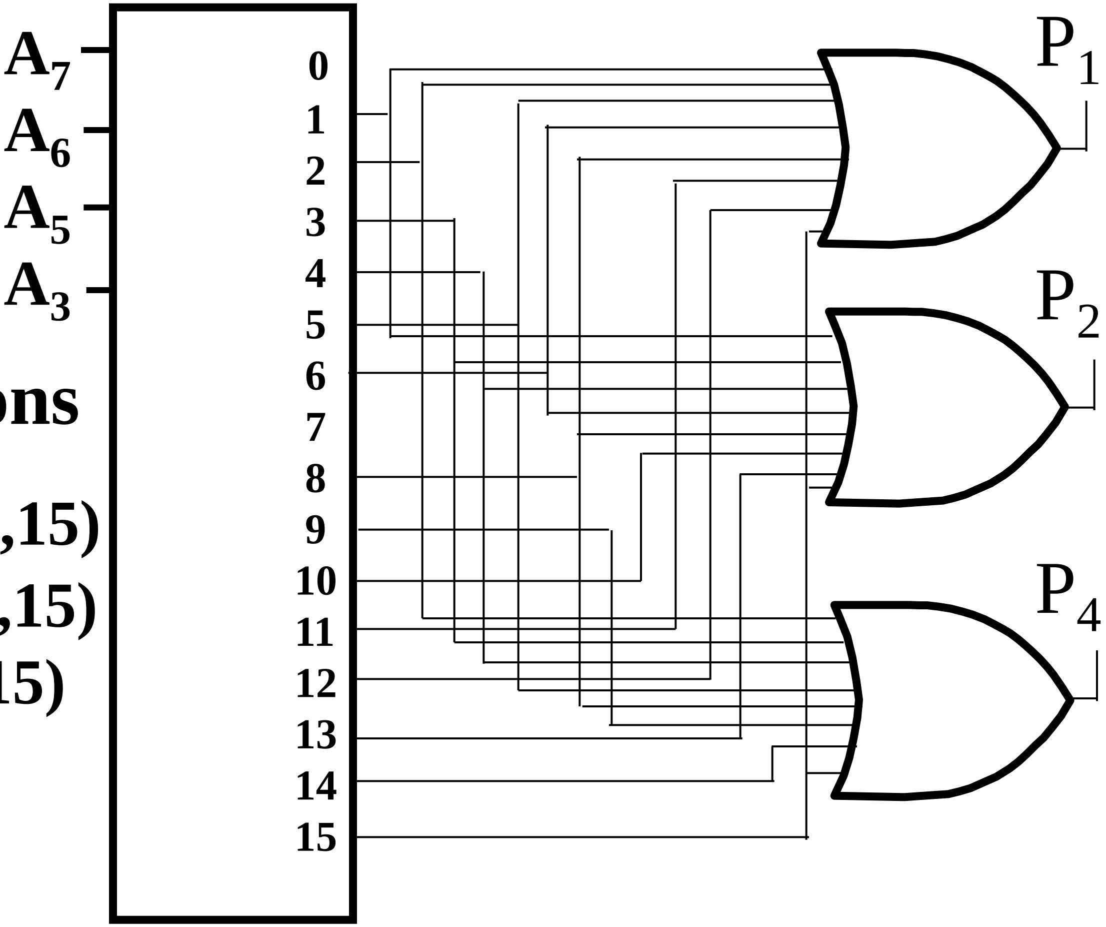
Decoder and OR Gates Example

- **Implement the following set of odd parity functions of (A_7, A_6, A_5, A_3)**
  $P_1 = A_7 \oplus A_5 \oplus A_3$
  $P_2 = A_7 \oplus A_6 \oplus A_3$
  $P_4 = A_7 \oplus A_6 \oplus A_5$

- **Finding sum of minterms expressions**

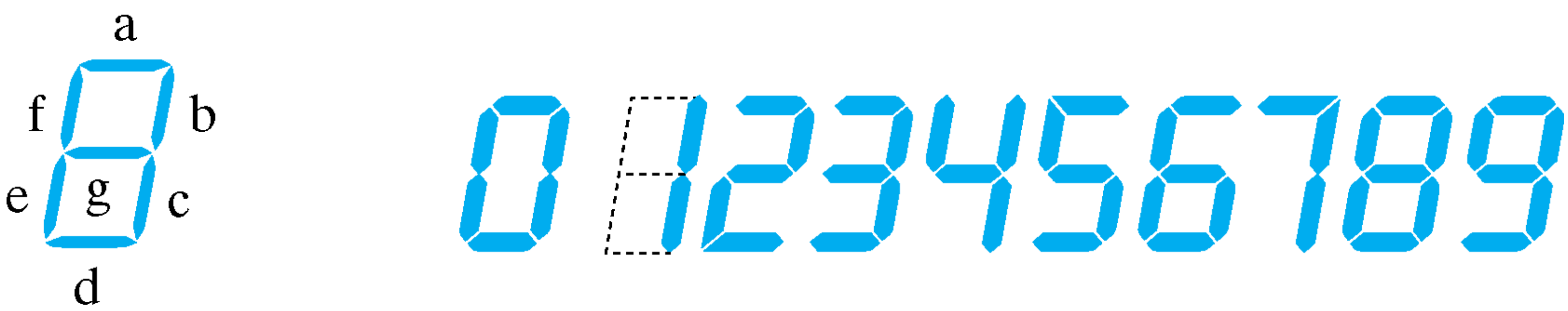
  $P_1 = \Sigma_m(1,2,5,6,8,11,12,15)$
  $P_2 = \Sigma_m(1,3,4,6,8,10,13,15)$
  $P_4 = \Sigma_m(2,3,4,5,8,9,14,15)$

- **Find circuit**
- **Is this a good idea?**



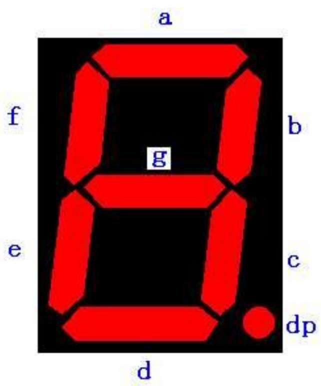
BCD-to-Segment Decoder

- **Seven-Segment Displayer**

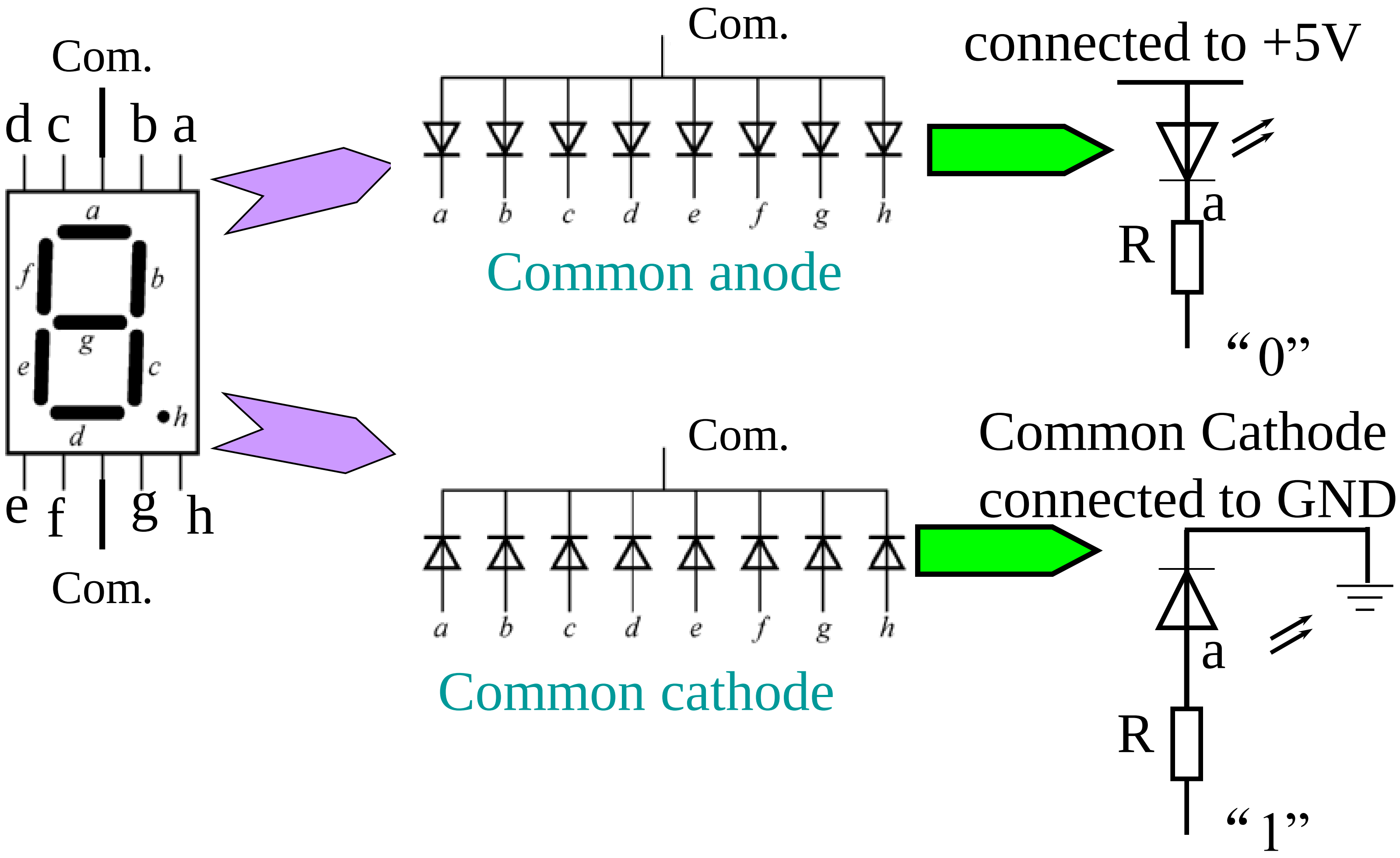


(a) Segment designation

(b) Numeric designation for display



Seven-Segment Displayer



Common anode

Common cathode

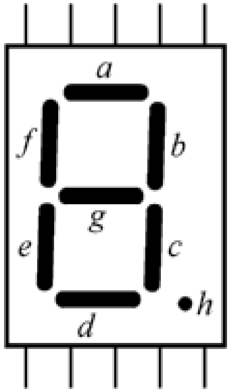
Common anode connected to +5V

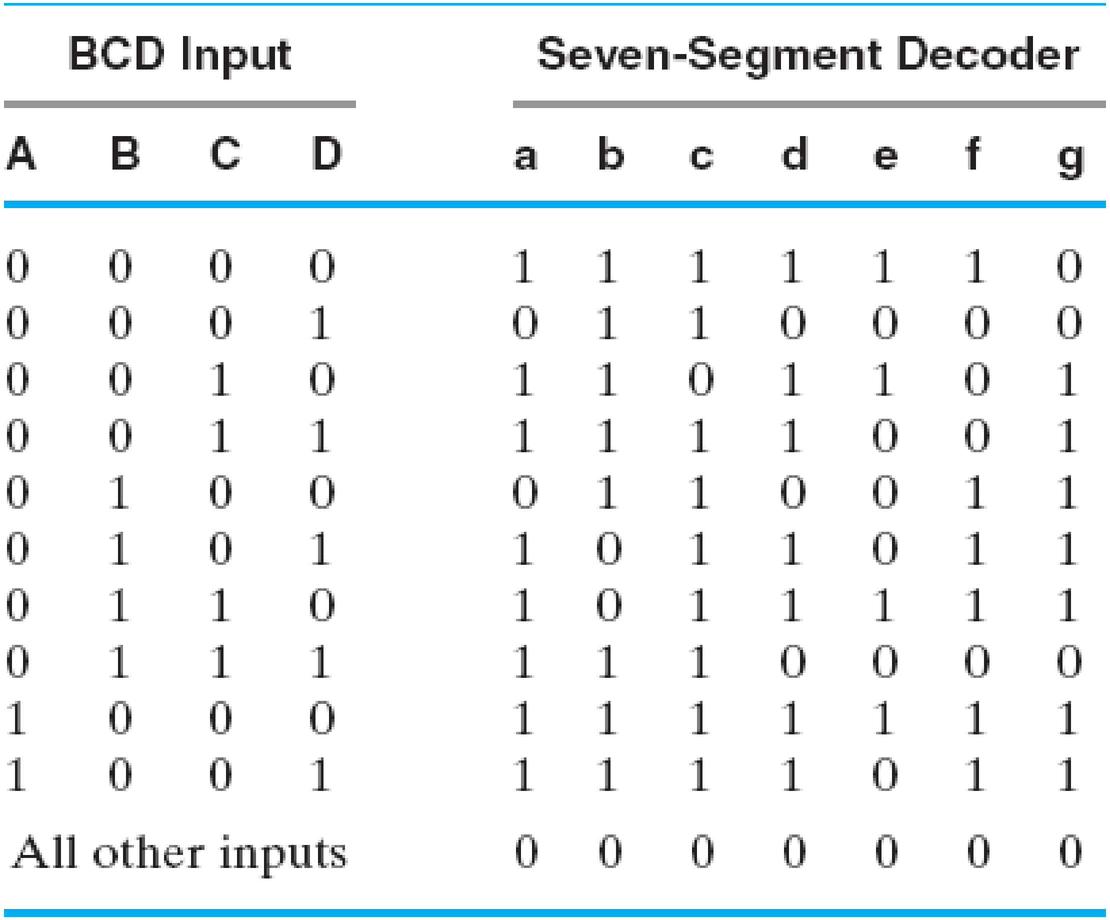
Common Cathode connected to GND

BCD-to-Segment Decoder (Cont.)

- **Truth Table for BCD-to-Seven-Segment Decoder**

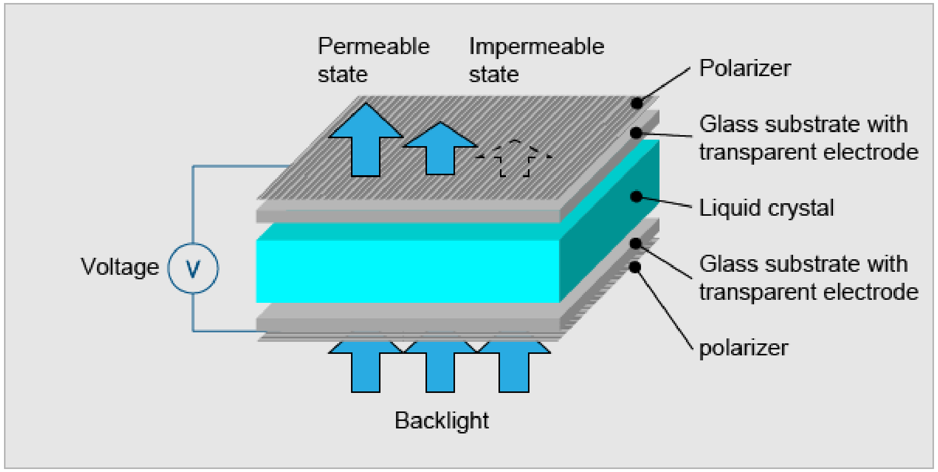
Common cathode





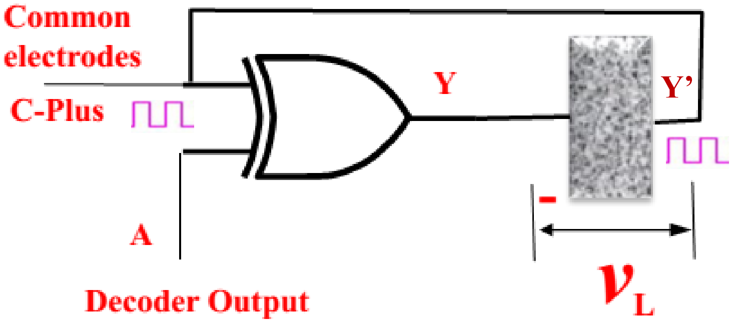
BCD Input				Seven-Segment Decoder						
A	B	C	D	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
All other inputs				0	0	0	0	0	0	0

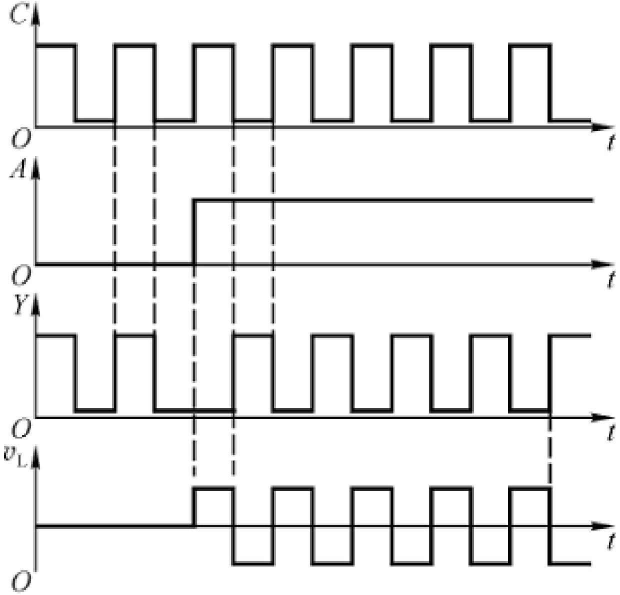
LCD Driving Principle



Permeable state
Impermeable state
Polarizer
Glass substrate with transparent electrode
Liquid crystal
Voltage
V
Glass substrate with transparent electrode
polarizer
Backlight

LCD working principle

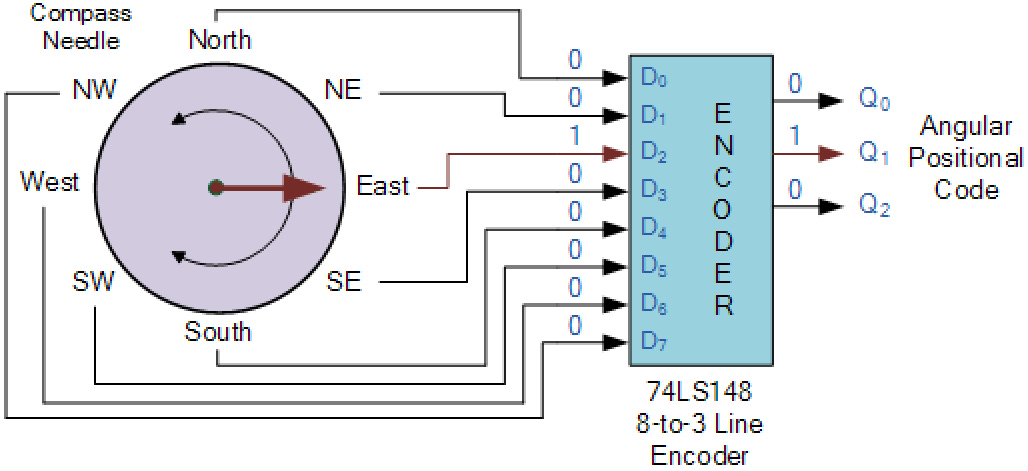




LCD driving principle

$$Y = CP \oplus A$$

Encoding

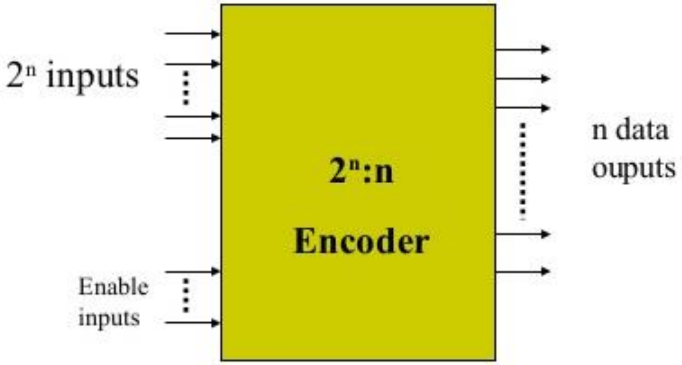


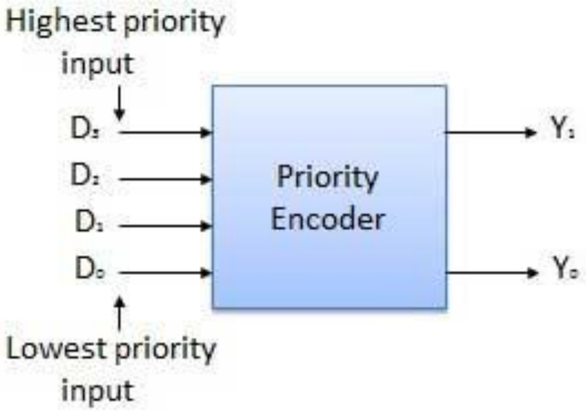
- **Encoding**
  - **the opposite of decoding - the conversion of an *m*-bit input code to a *n*-bit output code with $n \leq m \leq 2^n$ such that each valid code word produces a unique output code**
- **Circuits that perform encoding are called *encoders***
- **An encoder has 2^n (or fewer) input lines and *n* output lines which generate the binary code corresponding to the input values**

Types of Encoder

- 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.

- Types of encoder:
  - Instruction Encoder
  - Decimal-to-BCD Encoder
  - Priority Encoder (widely used in computer priority interrupt system and keyboard coding system)
  - Cypher Encoder



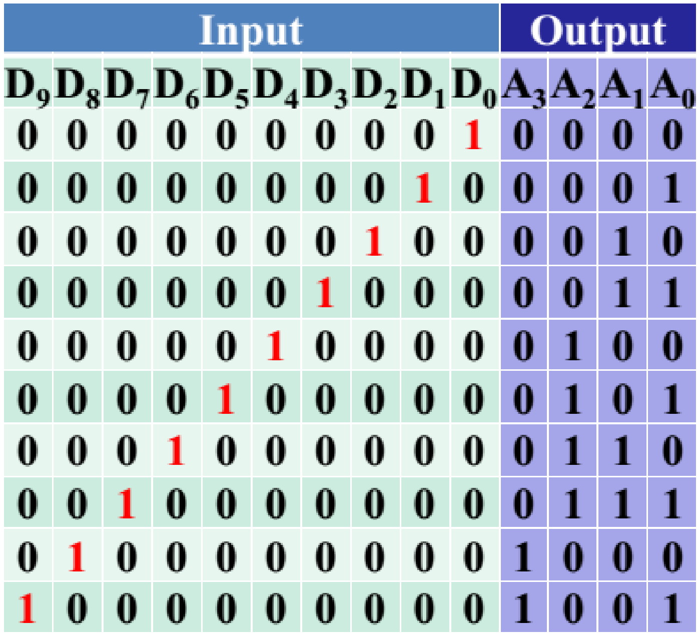


Encoder Example

- **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 (continued)

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



Input										Output			
D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_3	A_2	A_1	A_0
0	0	0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	0	1
0	0	0	0	0	0	0	1	0	0	0	0	1	0
0	0	0	0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	0	1	0	0	0	0	0	1	0	0
0	0	0	0	1	0	0	0	0	0	0	1	0	1
0	0	0	1	0	0	0	0	0	0	0	1	1	0
0	0	1	0	0	0	0	0	0	0	0	1	1	1
0	1	0	0	0	0	0	0	0	0	1	0	0	0
1	0	0	0	0	0	0	0	0	0	1	0	0	1

- Equations:

$$A_3 = D_8 + D_9$$

$$A_2 = D_4 + D_5 + D_6 + D_7$$

$$A_1 = D_2 + D_3 + D_6 + D_7$$

$$A_0 = D_1 + D_3 + D_5 + D_7 + D_9$$

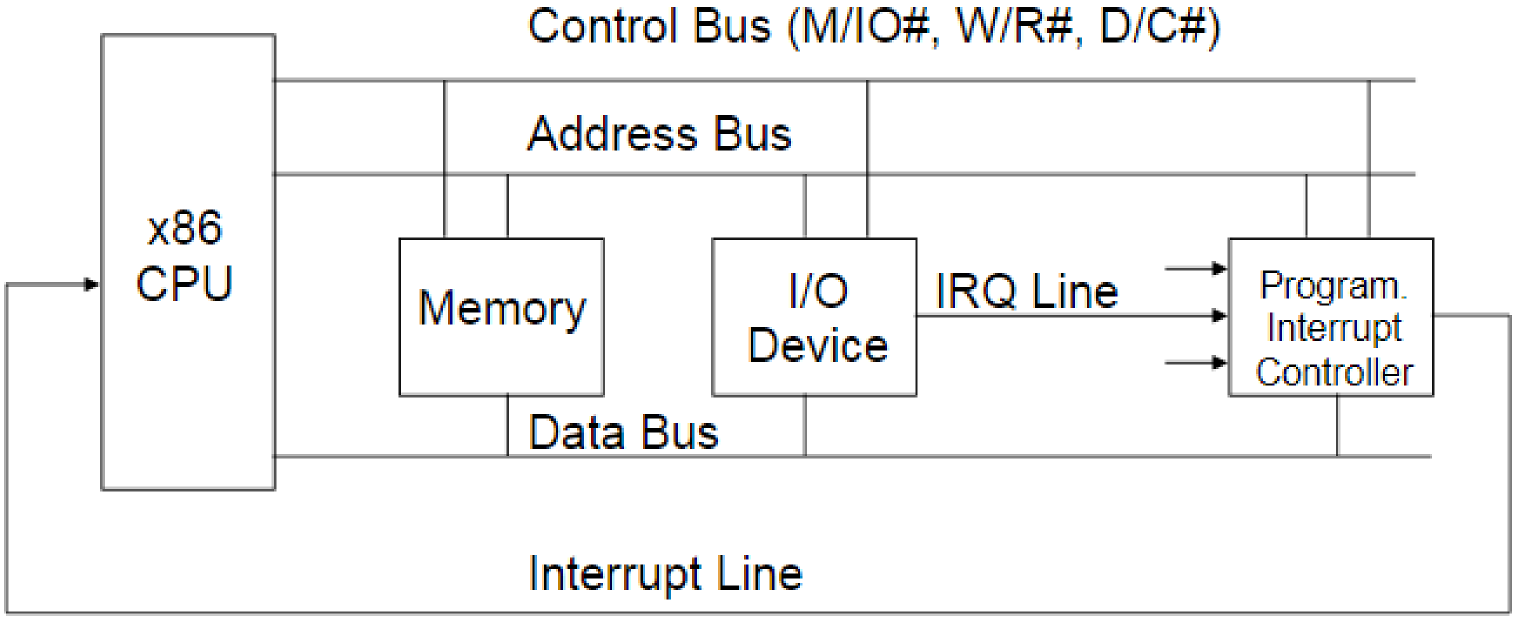
- $F_1 = D_6 + D_7$ can be extracted from A_2 and A_1 Is there any cost saving?

Priority Encoder

- **If more than one input value is 1, then the encoder just designed does not work.**
- **One encoder that can accept all possible combinations of input values and produce a meaningful result is a *priority encoder*.**
- **Among the 1s that appear, it selects the most significant input position (or the least significant input position) containing a 1 and responds with the corresponding binary code for that position.**

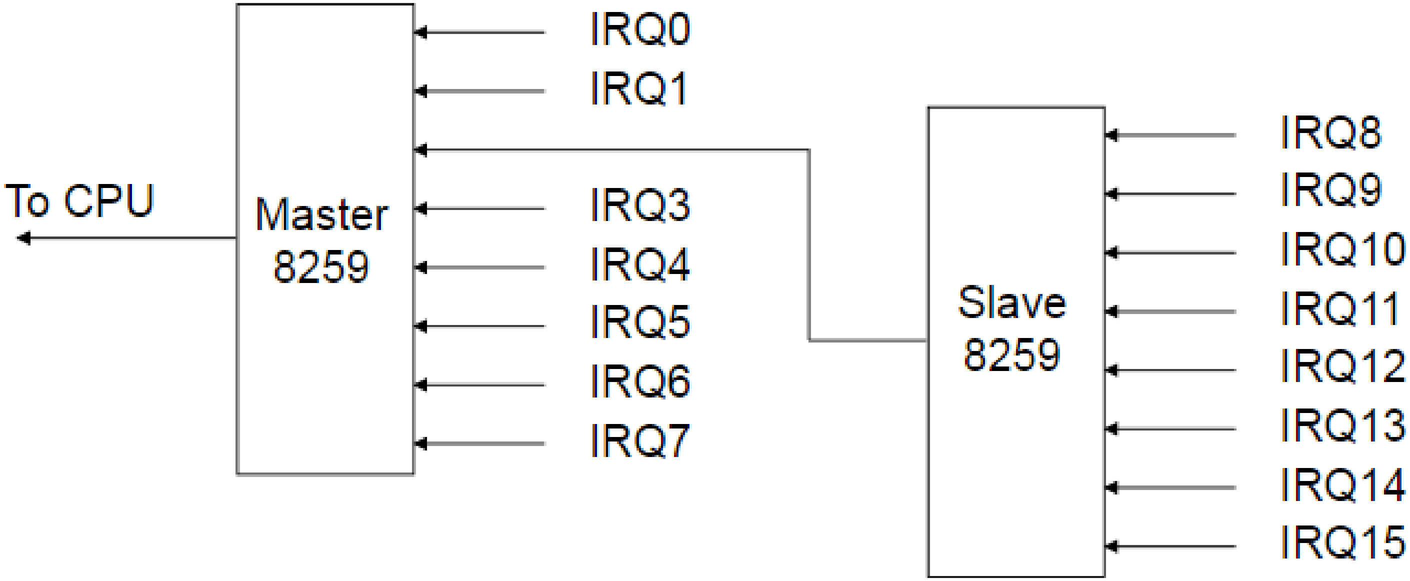
Priority Encoder Example 1

- Adding interrupt to the hardware
  - IRQ line from I/O device to Programmable Interrupt Controller (PIC)
  - Interrupt line from PIC to CPU



Priority Encoder Example 1 (continued)

- **Programmable Interrupt Controller (8259A chip)**
  - **Support eight interrupt request (IRQ) lines**
  - **Two chips used in PC, called "master" and "slave"**
  - **Priority: highest to lowest order is IRQ(0-1, 8-15, 3-7)**
  - **Asserts INTR to CPU, responds to resulting INTA with interrupt type code**



Priority Encoder Example 2

- 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 Min-terms/Row	Inputs					Outputs			
	D4	D3	D2	D1	D0	A2	A1	A0	V
0	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; 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 2 (continued)

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

$$A_2 = D_4$$

$$A_1 = \overline{D_4}D_3 + \overline{D_4}\,\overline{D_3}D_2 = \overline{D_4}F_1 \qquad F_1 = (D_3 + D_2)$$

$$A_0 = \overline{D_4}D_3 + \overline{D_4}\,\overline{D_3}\,\overline{D_2}D_1 = \overline{D_4}(D_3 + \overline{D_2}D1)$$

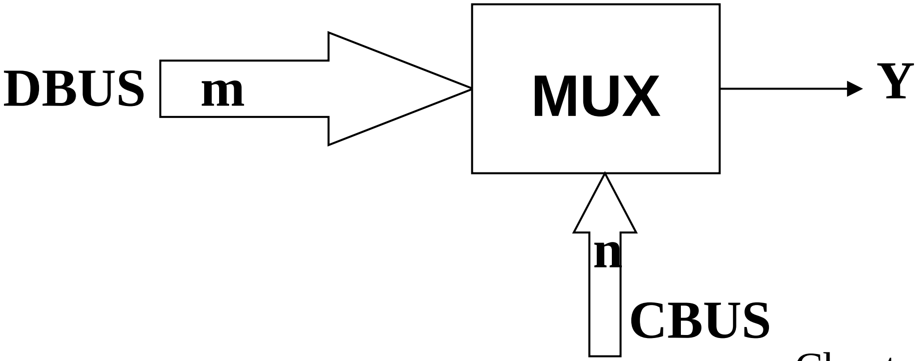
$$V = D_4 + F_1 + D_1 + D_0$$

Selecting

- Selecting of data or information is a critical function in digital systems and computers
- Circuits that perform selecting have:
  - A set of information inputs from which the selection is made
  - A set of control lines for making the selection
  - A single output
- Logic circuits that perform selecting are called ***multiplexers***
- Selecting can also be done by three-state logic or transmission gates

Multiplexers

- **A multiplexer selects information from an input line and directs the information to an output line**
- **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**

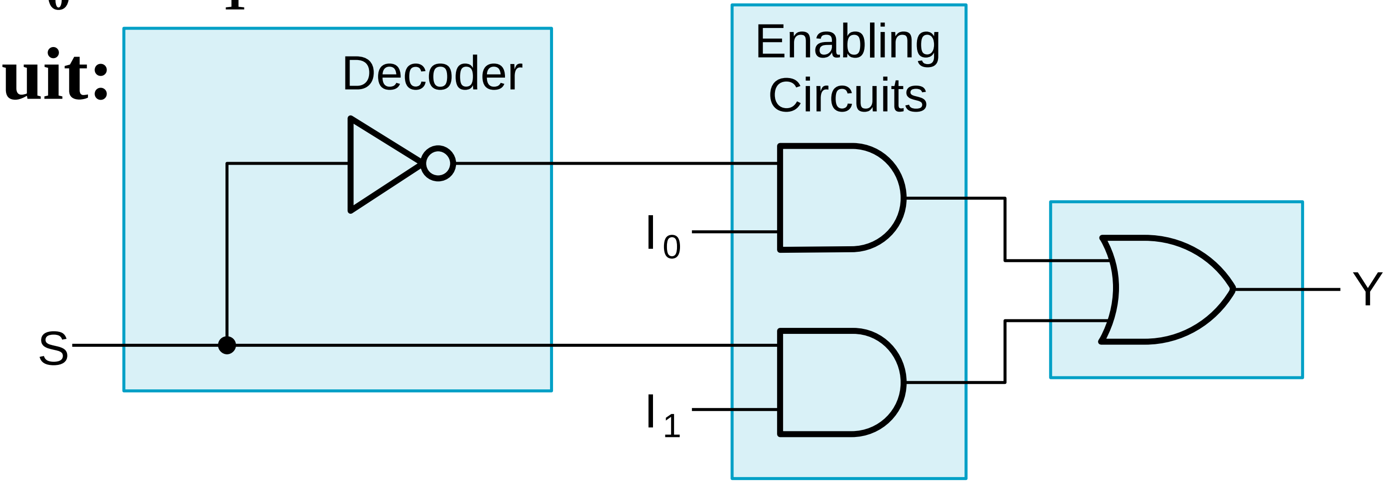


2-to-1-Line Multiplexer

- 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:

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

- The circuit:

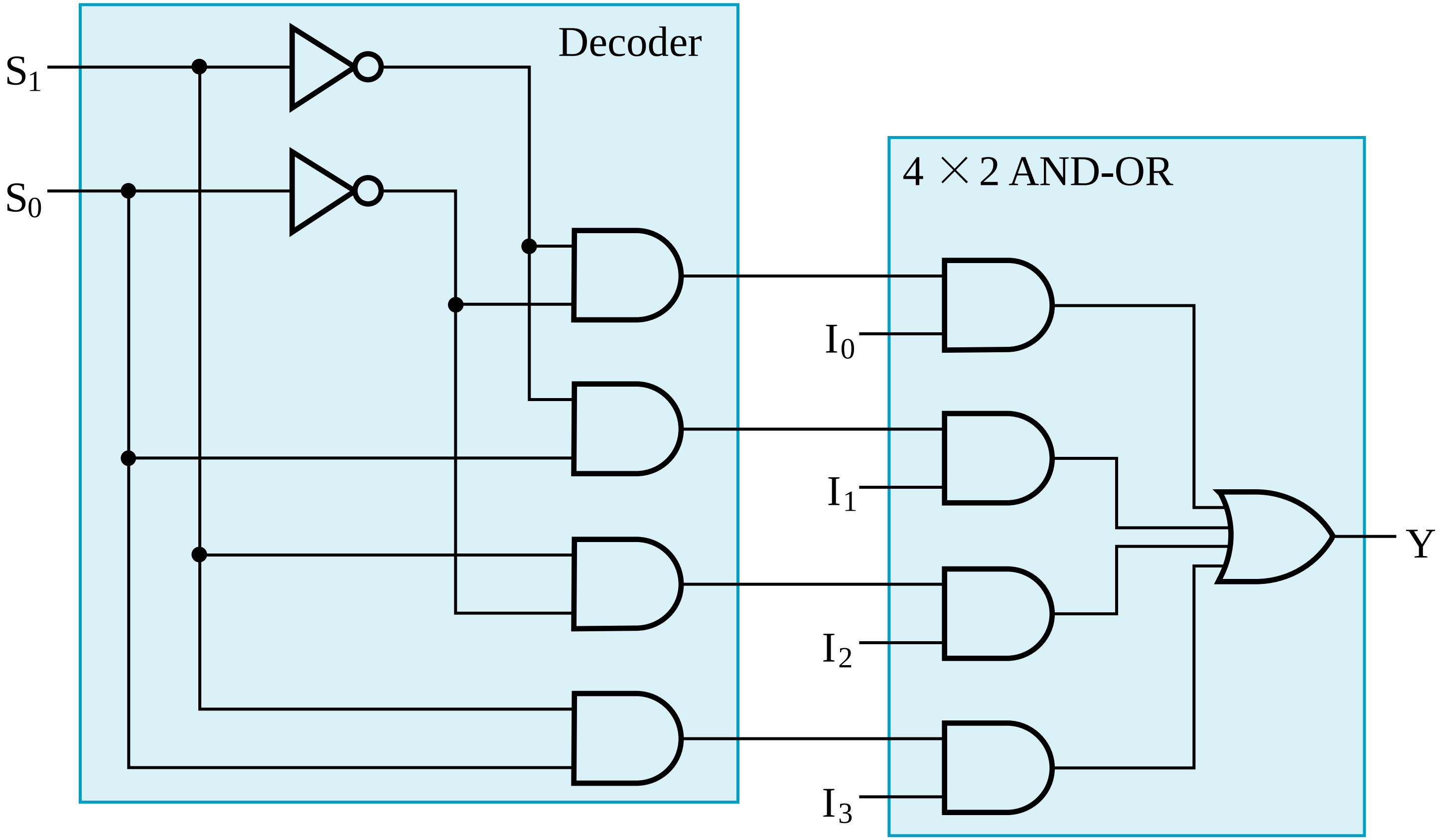


2-to-1-Line Multiplexer (continued)

- **Note the regions of the multiplexer circuit shown:**
  - **1-to-2-line Decoder**
  - **2 Enabling circuits**
  - **2-input OR gate**
- **To obtain a basis for multiplexer expansion, we combine the Enabling circuits and OR gate into a 2 × 2 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 × 2 AND-OR**

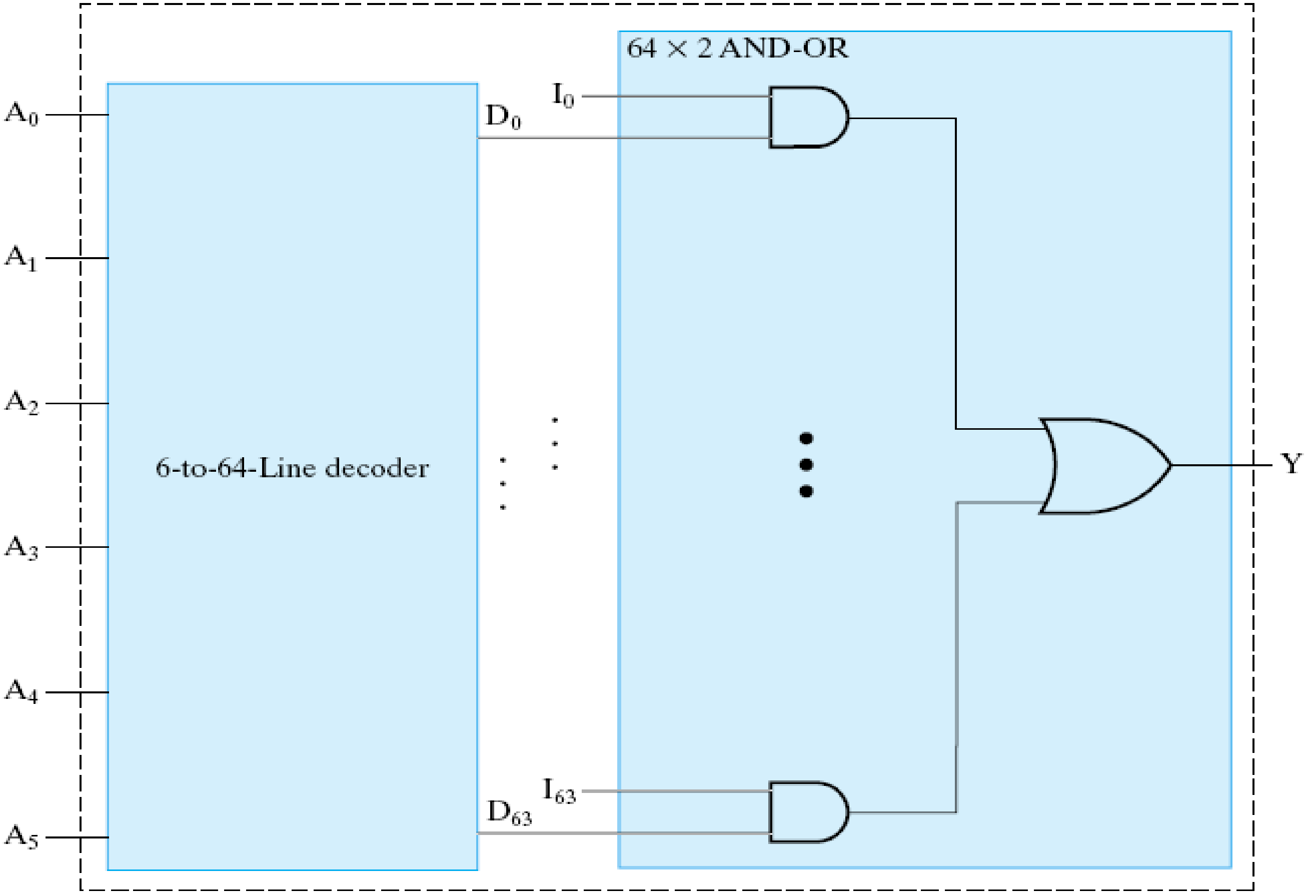
Example: 4-to-1-line Multiplexer

- 2-to-2^2-line decoder
- 2^2 × 2 AND-OR GN = 2+8+8+4 = 22



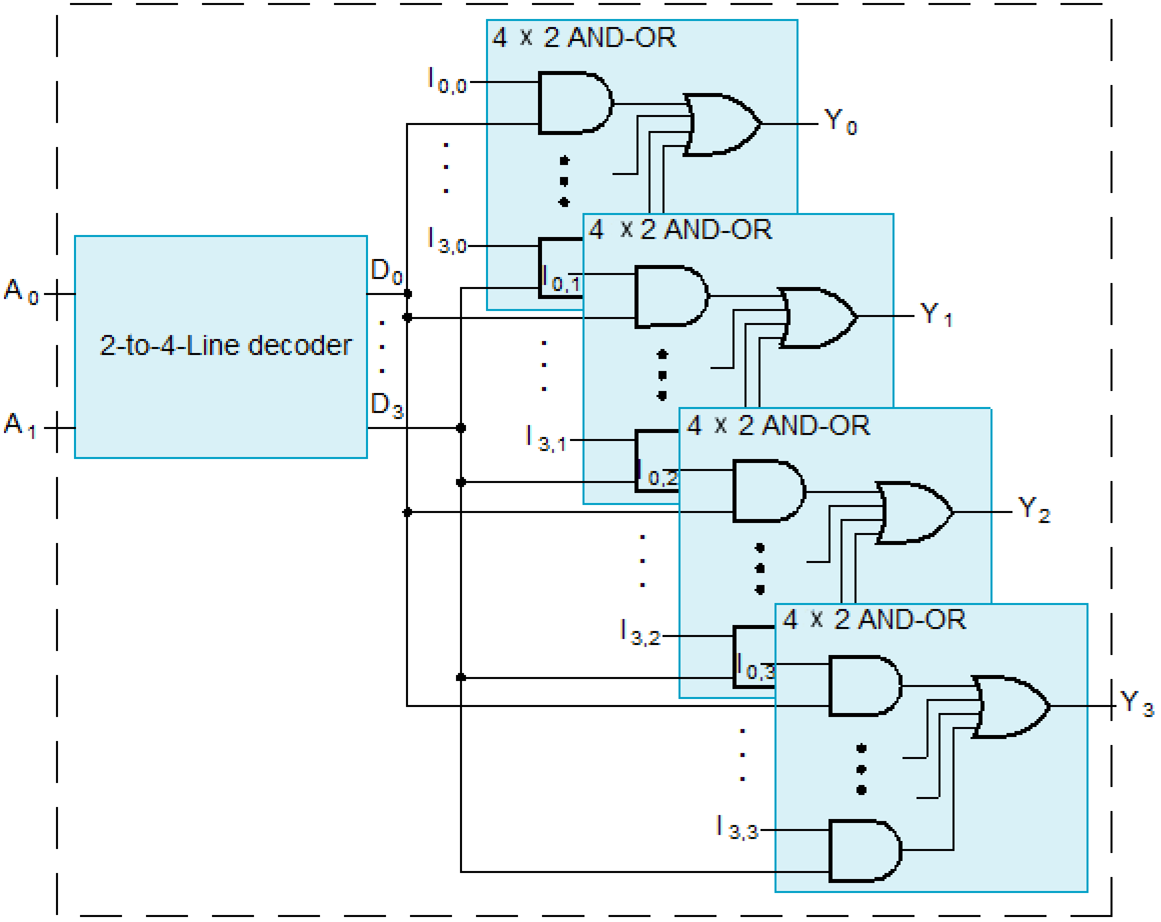
Example: 64-to-1-line Multiplexer

- 6-to-2^6-line decoder
- 2^6 × 2 AND-OR



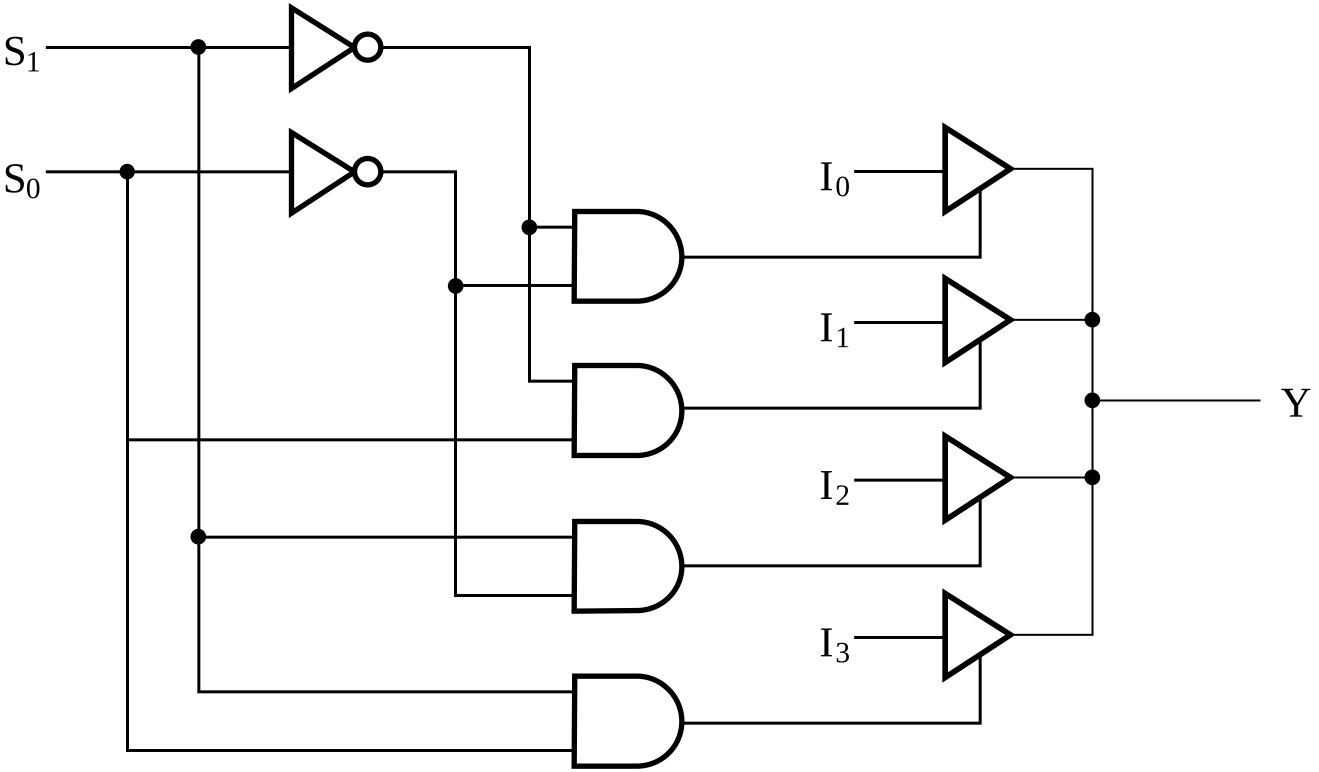
Multiplexer Width Expansion

- Select “vectors of bits” instead of “bits”
- Use multiple copies of $2^n \times 2$ AND-OR in parallel
- Example: 4-to-1-line quad multi-plexer



Other Selection Implementations

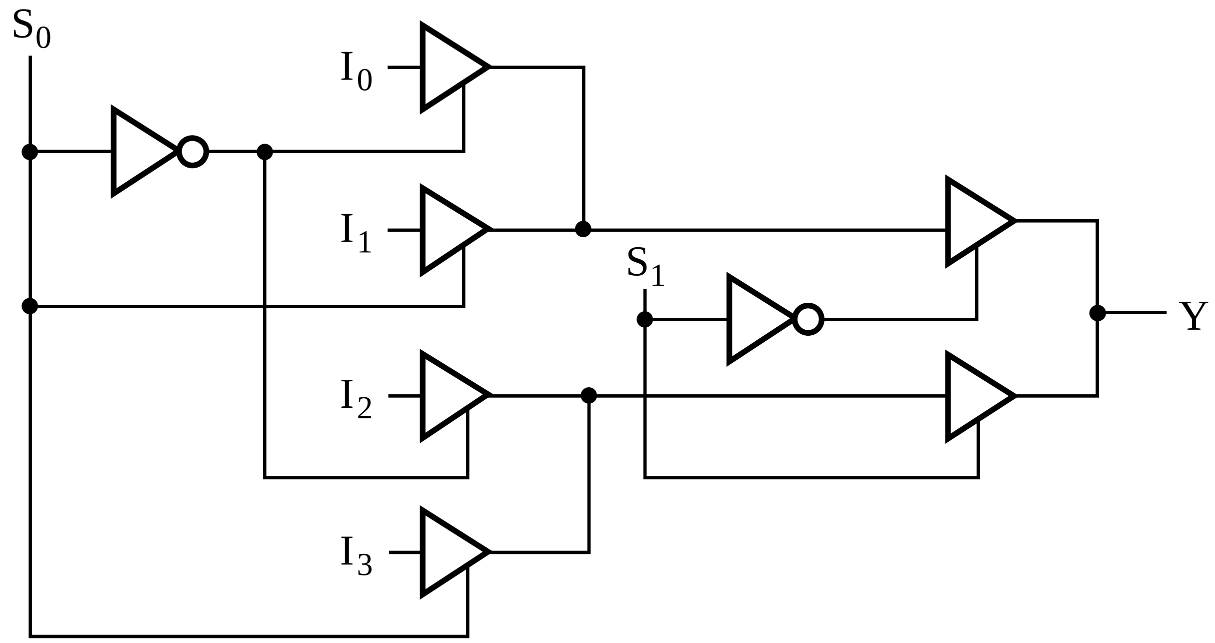
- Three-state logic in place of AND-OR



- Gate input cost with NOTs :18 (2+8+8)
- Gate input cost with NOTs of AND-OR gates: 22

Other Selection Implementations

- **Distributing the decoding across the three-state drivers**



- **Gate input cost with NOTs of AND-OR gates: 22**
- **Gate input cost with NOTs of 3-state drivers: 18**
- **Gate input cost with NOTs: 14 (2+8+4)**

Combinational Logic Implementation - Multiplexer Approach 1

- **Implement *m* functions of *n* variables with:**
  - **Sum-of-minterms expressions**
  - **An *m*-wide 2^n-to-1-line multiplexer**
- **Design:**
  - **Find the truth table for the functions.**
  - **In the order they appear in the truth table:**
    - **Apply the function input variables to the multiplexer selection inputs $S_{n-1}, \ldots, S_0$**
    - **Label the outputs of the multiplexer with the output variables**
  - **Value-fix the information inputs to the multiplexer using the values from the truth table (for don't cares, apply either 0 or 1)**

Example: Gray to Binary Code

- **Design a circuit to convert a 3-bit Gray code to a binary code**
- **The formulation gives the truth table on the right**
- **It is obvious from this table that X = C and the Y and Z are more complex**

Gray **A B C**	**Binary** **x y z**
0 0 0	0 0 0
1 0 0	0 0 1
1 1 0	0 1 0
0 1 0	0 1 1
0 1 1	1 0 0
1 1 1	1 0 1
1 0 1	1 1 0
0 0 1	1 1 1

Gray to Binary (continued)

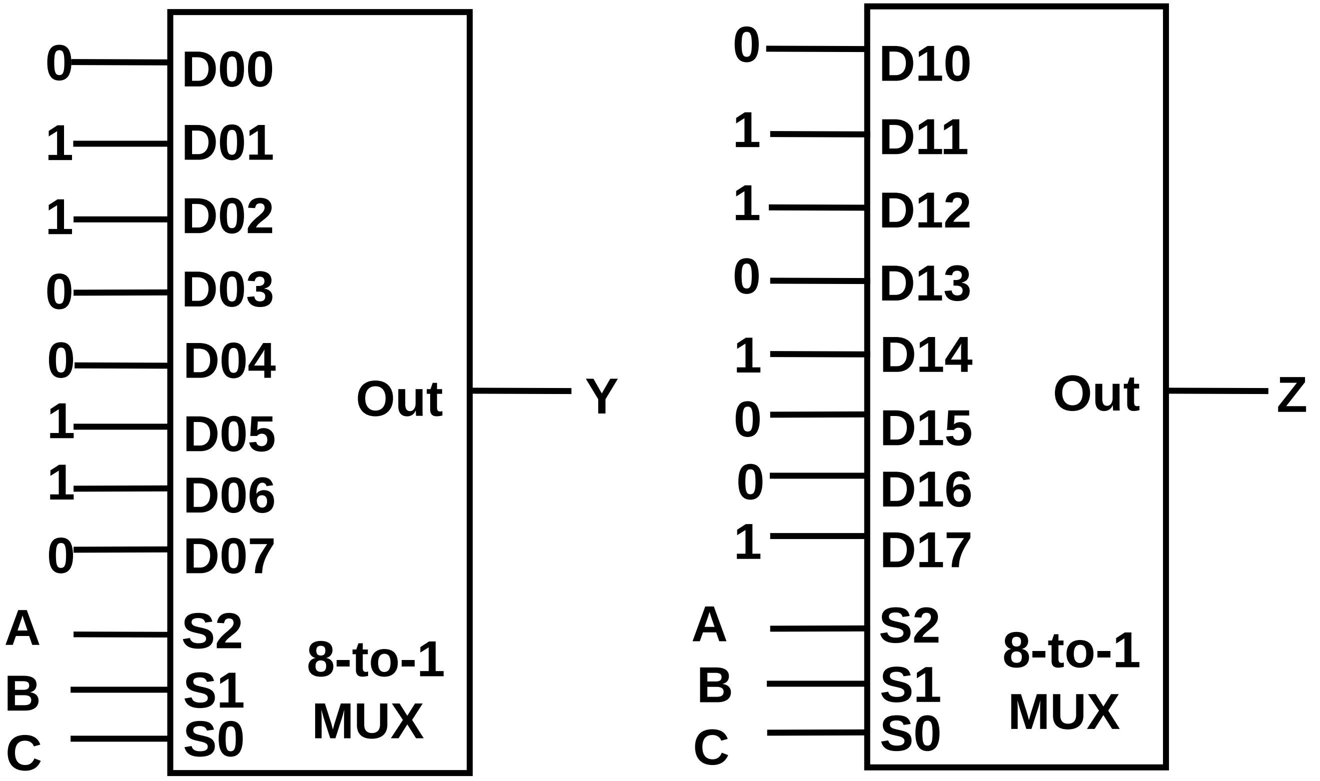
- Rearrange the table so that the input combinations are in counting order

- Functions y and z can be implemented using a dual 8-to-1-line multiplexer by:
  - connecting A, B, and C to the multiplexer select inputs
  - placing y and z on the two multiplexer outputs
  - connecting their respective truth table values to the inputs

Gray	Binary
A B C	x y z
0 0 0	0 0 0
0 0 1	1 1 1
0 1 0	0 1 1
0 1 1	1 0 0
1 0 0	0 0 1
1 0 1	1 1 0
1 1 0	0 1 0
1 1 1	1 0 1

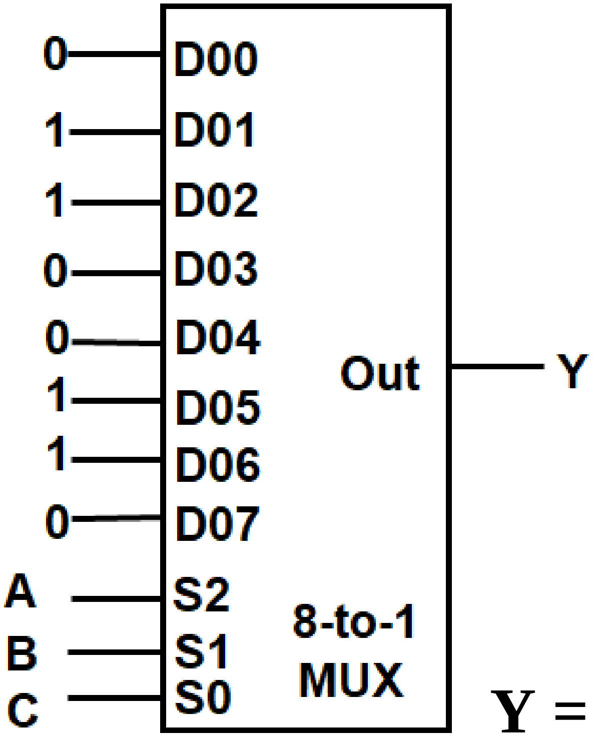
Gray to Binary (continued)

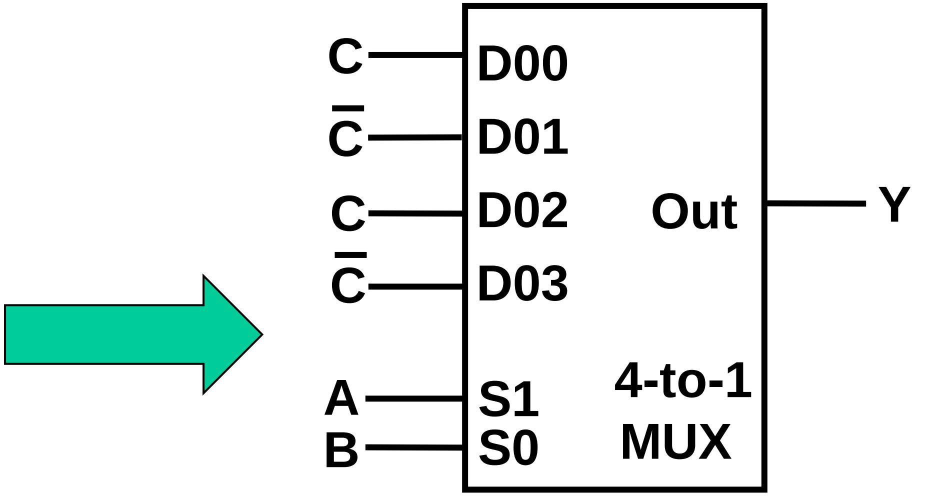
Gray A B C	Binary x y z
0 0 0	0 0 0
0 0 1	1 1 1
0 1 0	0 1 1
0 1 1	1 0 0
1 0 0	0 0 1
1 0 1	1 1 0
1 1 0	0 1 0
1 1 1	1 0 1



- Note that the multiplexer with fixed inputs is identical to a ROM with 3-bit addresses and 2-bit data!

Combinational Logic Implementation - Multiplexer Approach 2





$$
\begin{aligned}
Y = {} & \bar{A}\bar{B}\bar{C}\cdot 0+\bar{A}\bar{B}C\cdot 1+\bar{A}B\bar{C}\cdot 1+\bar{A}BC\cdot 0+A\bar{B}\bar{C}\cdot 0 \\
& +A\bar{B}C\cdot 1+AB\bar{C}\cdot 1+ABC\cdot 0 \\
= {} & \bar{A}\bar{B}\,(\bar{C}\cdot 0+C\cdot 1)+\bar{A}B\,(\bar{C}\cdot 1+C\cdot 0)+A\bar{B}\,(\bar{C}\cdot 0+C\cdot 1) \\
& +AB\,(\bar{C}\cdot 1+C\cdot 0) \\
= {} & \bar{A}\bar{B}C+\bar{A}B\bar{C}+A\bar{B}C+AB\bar{C}
\end{aligned}
$$

Combinational Logic Implementation - Multiplexer Approach 2

- **Implement any *m* functions of *n* variables by using:**
  - **An m-wide 2^{n-1}-to-1-line multiplexer**
  - **A single inverter**
- **Design:**
  - **The first n - 1 variables are applied to the selection inputs.**
  - **For each combination of the selection variables, the output is a function of the last variable (0, 1, X, $\overline{X}$).**
  - **These values are then applied to the appropriate data inputs.**

Example: Gray to Binary Code

- **Design a circuit to convert a 3-bit Gray code to a binary code**
- **The formulation gives the truth table on the right**
- **It is obvious from this table that X = C and the Y and Z are more complex**

Gray	Binary
A B C	x y z
0 0 0	0 0 0
1 0 0	0 0 1
1 1 0	0 1 0
0 1 0	0 1 1
0 1 1	1 0 0
1 1 1	1 0 1
1 0 1	1 1 0
0 0 1	1 1 1

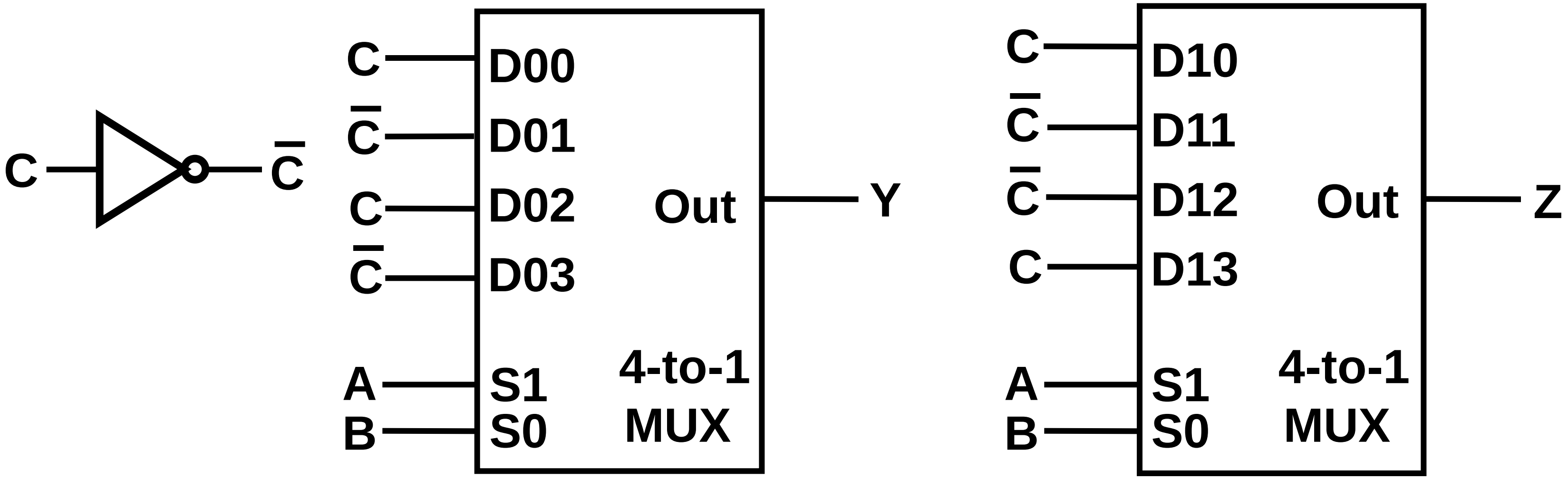
Gray to Binary (continued)

- **Rearrange the table so that the input combinations are in counting order, pair rows, and find rudimentary functions**

Gray A B C	Binary x y z	Rudimentary Functions of C for y	Rudimentary Functions of C for z
0 0 0	0 0 0	$F = C$	$F = C$
0 0 1	1 1 1		
0 1 0	0 1 1	$F = \overline{C}$	$F = \overline{C}$
0 1 1	1 0 0		
1 0 0	0 0 1	$F = C$	$F = \overline{C}$
1 0 1	1 1 0		
1 1 0	0 1 0	$F = \overline{C}$	$F = C$
1 1 1	1 0 1		

Gray to Binary (continued)

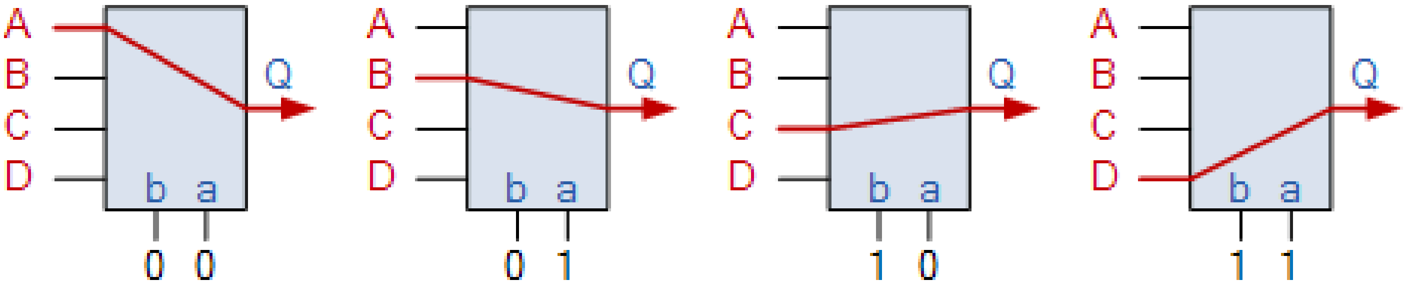
- Assign the variables and functions to the multiplexer inputs:



- Note that this approach (Approach 2) reduces the cost by almost half compared to Approach 1.
- This result is no longer ROM-like.
- Extending, a function of more than *n* variables is decomposed into several sub-functions defined on a subset of the variables. The multiplexer then selects among these sub-functions.

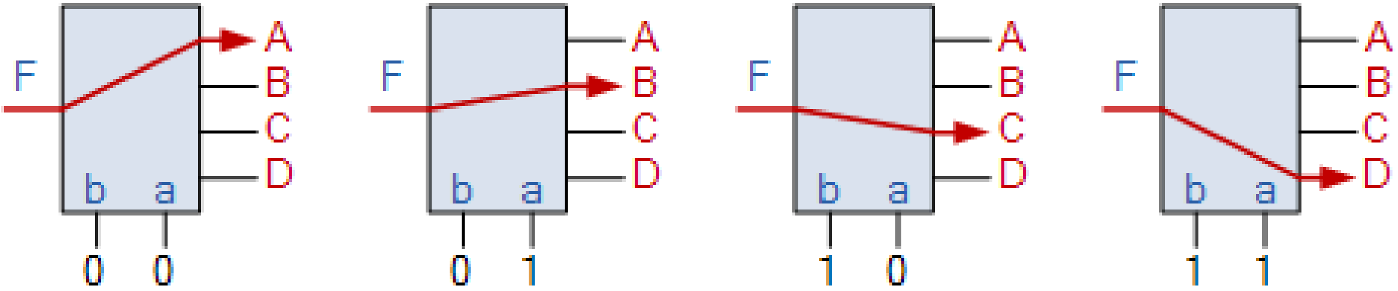
Multiplexer and Demultiplexer

- **Multiplexer means many into one, which is used to select one of the several input signals to a signal output.**



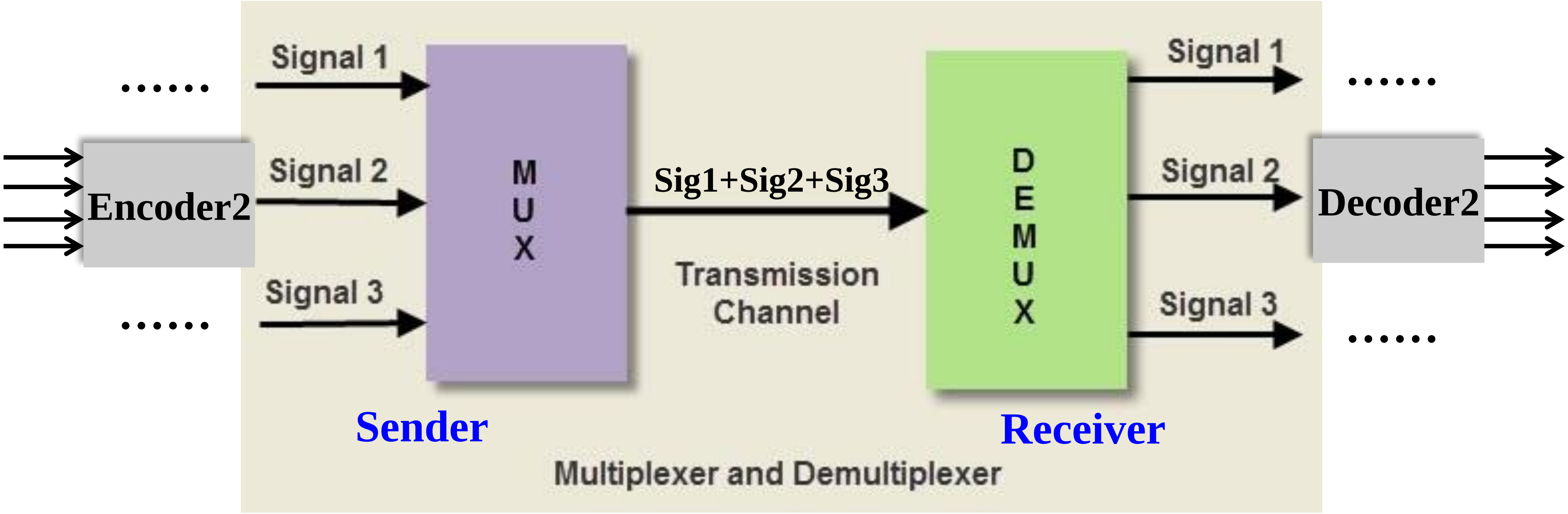
MUX

- **The demultiplexer means one into many, which takes one single input data line and then switches it to any one of a number of individual output lines.**



DEMUX

Multiplexer and Demultiplexer



Assignments

Reading

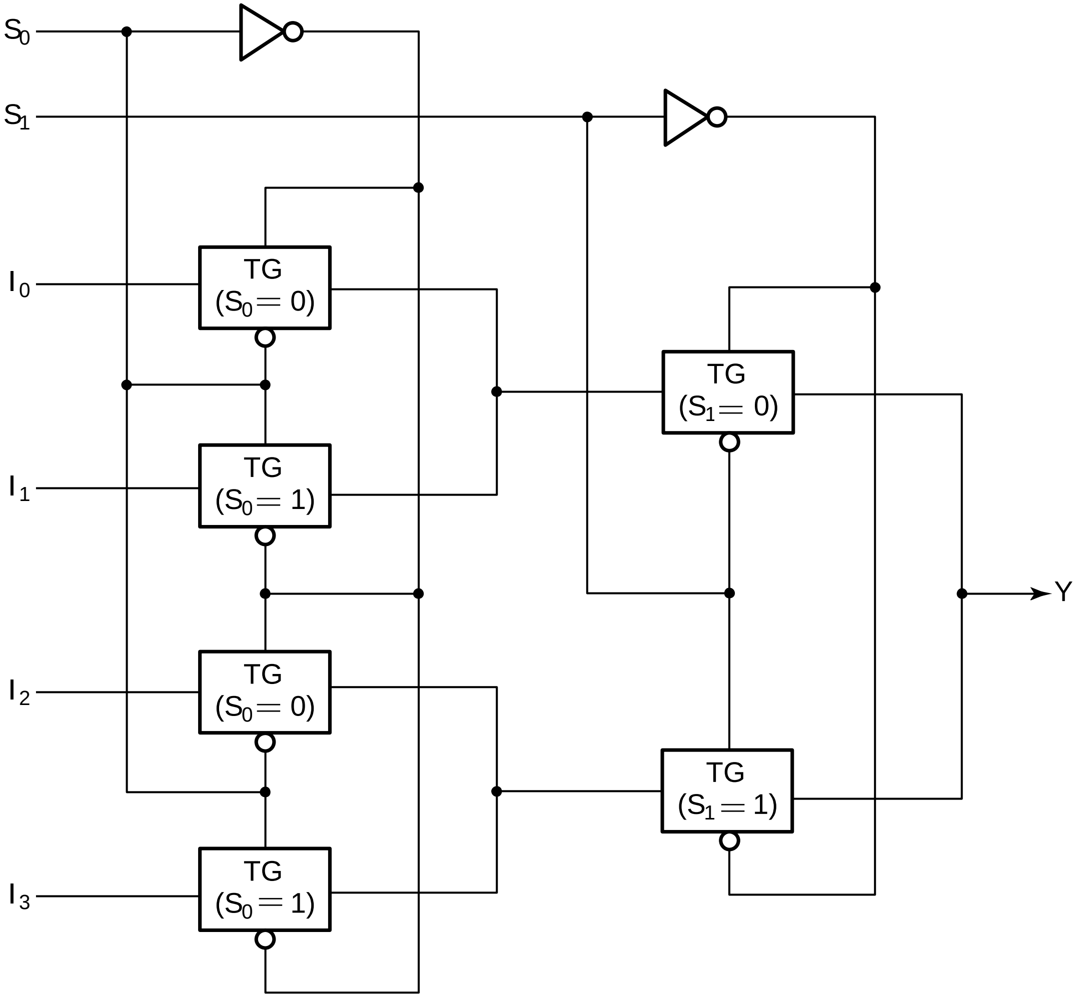
- 3.4 – 3.7

Problem assignment

- 3-28, 3-29, 3-37, 3-44, 3-47

Appendix A: Transmission Gate Multiplexer

- Transmission Gate Multiplexer
- Gate input cost = 8 compared to 14 for 3-state logic and 18 or 22 for gate logic



S_0
S_1
I_0
I_1
I_2
I_3
TG ($S_0 = 0$)
TG ($S_0 = 1$)
TG ($S_0 = 0$)
TG ($S_0 = 1$)
TG ($S_1 = 0$)
TG ($S_1 = 1$)
Y