Computer Logic Design Fundamentals

Chapter 3 – Combinational Logic Design

Part 2 – Combinational Logic

Prof. Yueming Wang
ymingwang@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

Functions and Functional Blocks

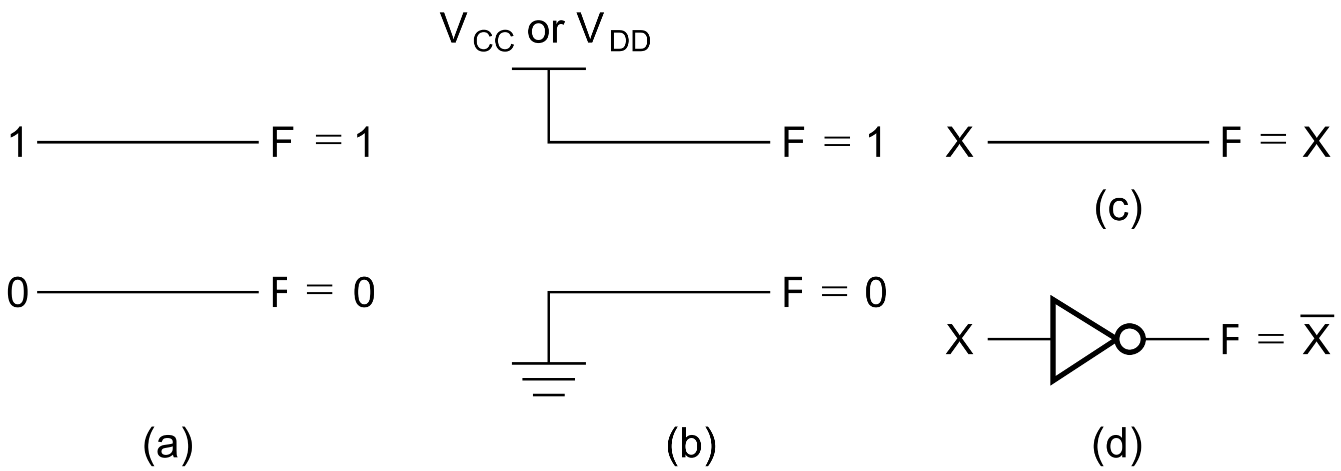
- The functions considered are those found to be very useful in design
- Corresponding to each of the functions is a combinational circuit implementation called a *functional block*.
- In the past, functional blocks were packaged as small-scale-integrated (SSI), medium-scale integrated (MSI), and large-scale-integrated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

Rudimentary Logic Functions

- Functions of a single variable X
- Can be used on the inputs to functional blocks to implement other than the block's intended function

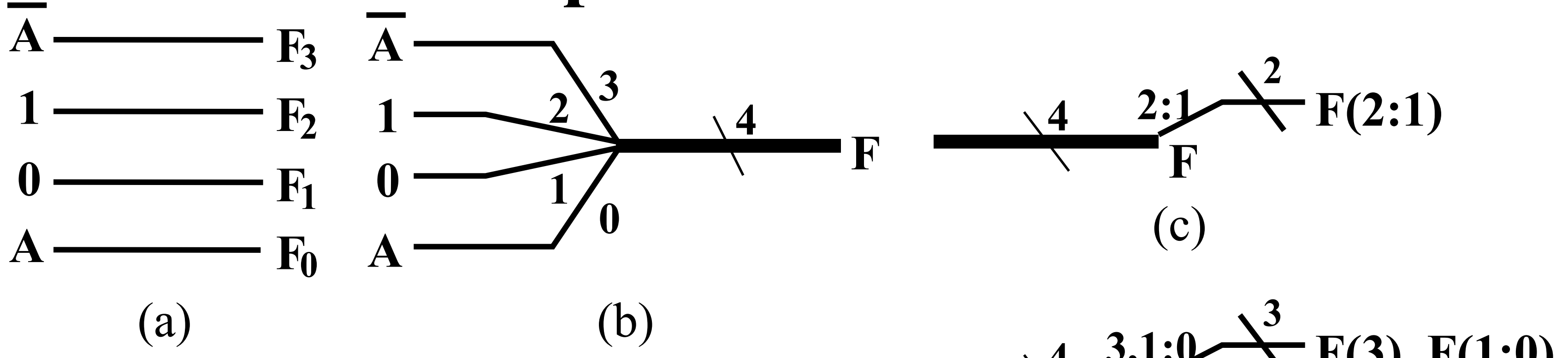
- TABLE 4-1
  Functions of One Variable

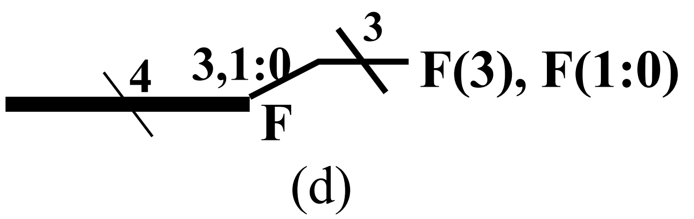
X	F = 0	F = X	F = $\overline{X}$	F = 1
0	0	0	1	1
1	0	1	0	1



Multiple-bit Rudimentary Functions

- Multi-bit Examples:

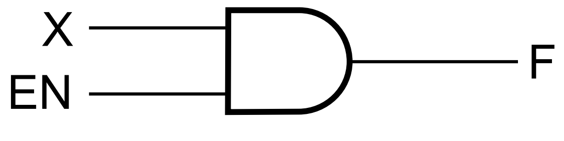




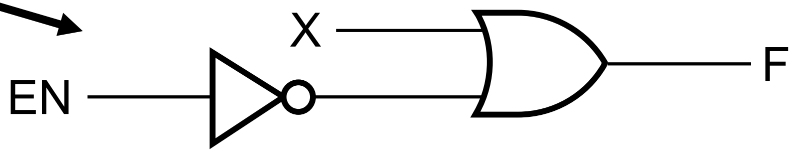
- A wide line is used to represent a *bus* which is a vector signal
- In (b) of the example, F = (F_3, F_2, F_1, F_0) is a bus.
- The bus can be split into individual bits as shown in (b)
- Sets of bits can be split from the bus as shown in (c) for bits 2 and 1 of F.
- The sets of bits need not be continuous 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 Hi-Z (as for three-state buffers and transmission gates), 0 , or 1
- When disabled, 0 output
- When disabled, 1 output
- See Enabling App in text



(a)

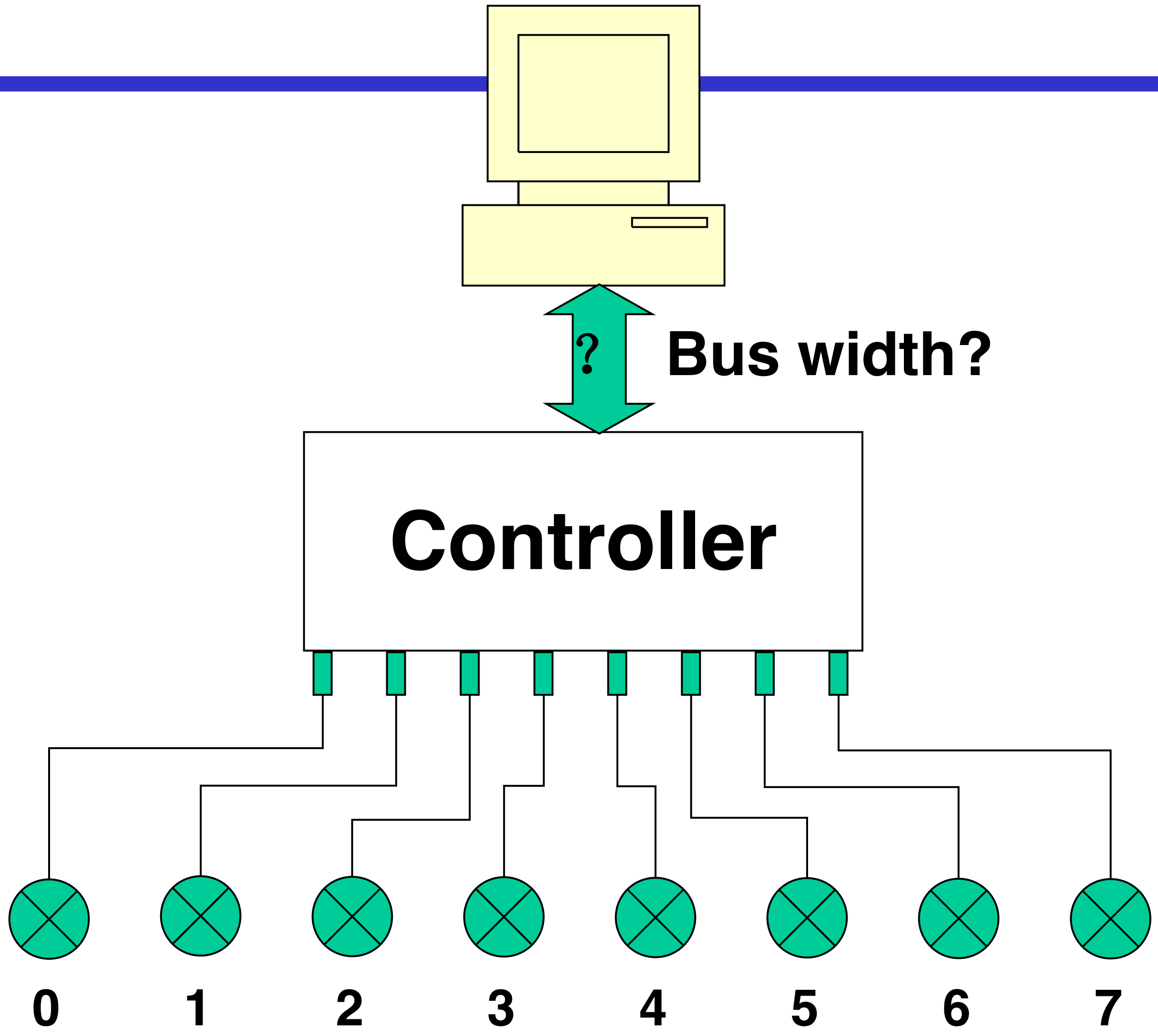


(b)

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**
- **Circuits that perform decoding are called *decoders***
- **Here, 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



Decoder

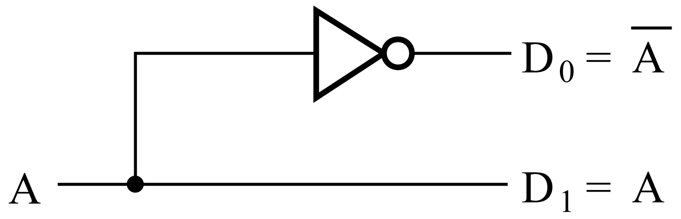
A	B	C	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Decoder Examples

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

A	**D_0**	**D_1**
0	1	0
1	0	1

(a)

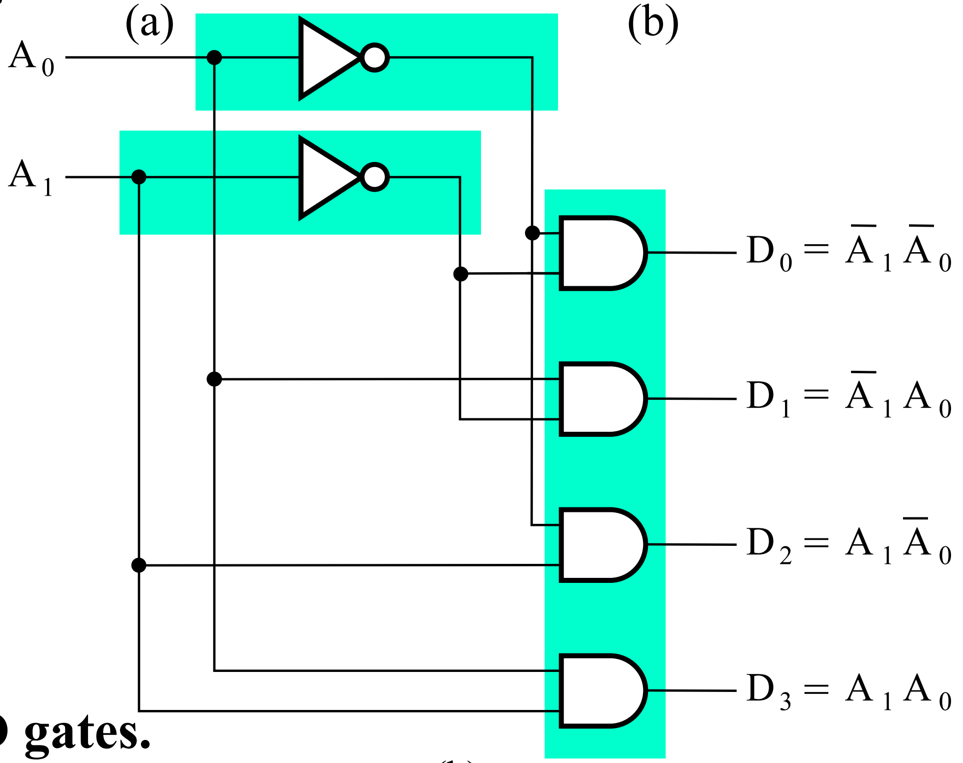


(b)

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

(b)

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

Decoder Expansion

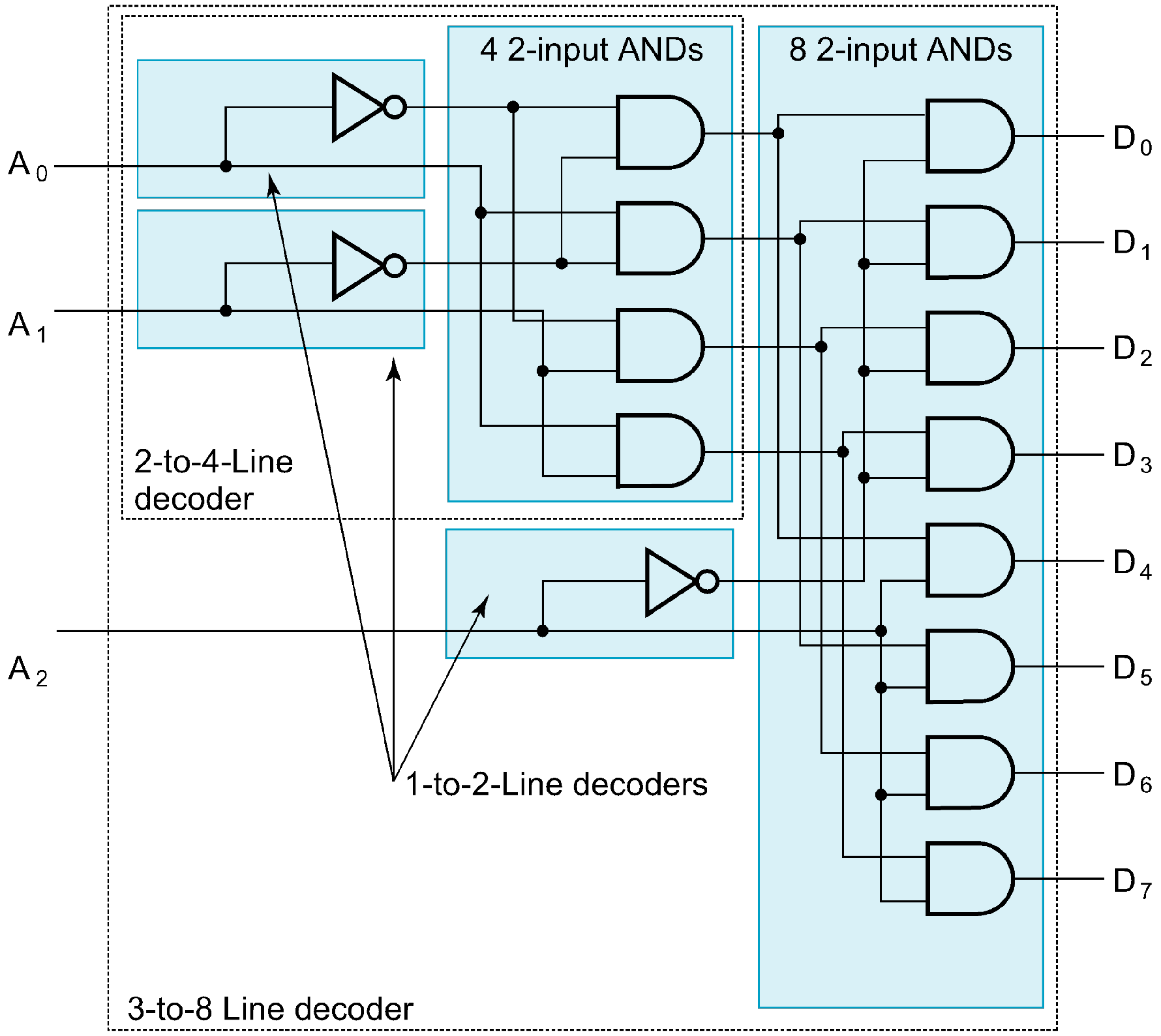
- **General procedure given in book for any decoder with *n* inputs and 2^n outputs.**
- **This procedure builds a decoder backward from the outputs.**
- **The output AND gates are driven by two decoders with their numbers of inputs either equal or differing by 1.**
- **These decoders are then designed 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
  - Number of inputs to decoders driving output ANDs = 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 decoders driving output ANDs = 2
    - Closest possible split to equal
      - Two 1-to-2-line decoders
- See next slide for result

Decoder Expansion - Example 1

- **Result**



Decoder Expansion - Example 2

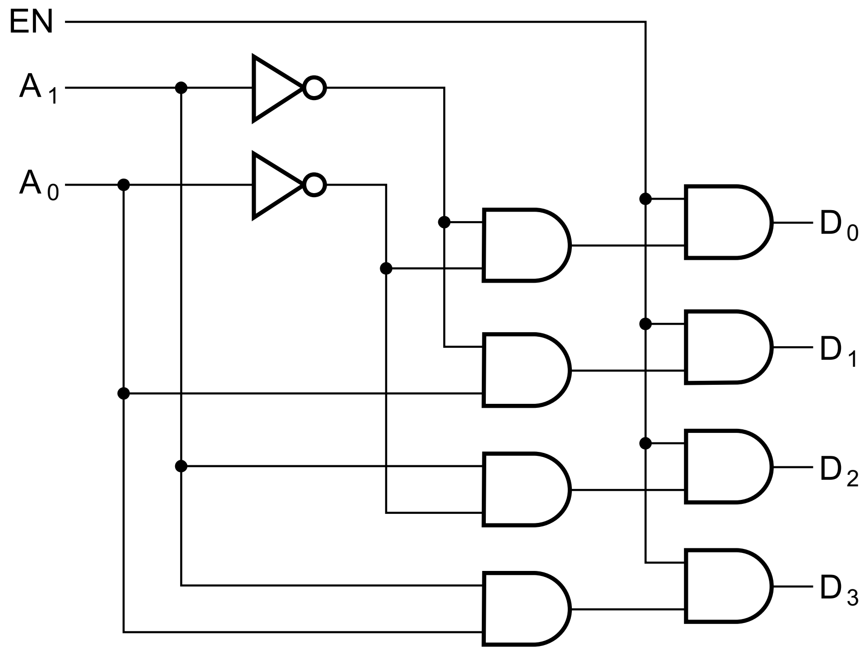
- 7-to-128-line decoder
  - Number of output ANDs = 128
  - Number of inputs to decoders driving output ANDs = 7
  - Closest possible split to equal
    - 4-to-16-line decoder
    - 3-to-8-line decoder
  - 4-to-16-line decoder
    - Number of output ANDs = 16
    - Number of inputs to decoders driving output ANDs = 2
    - Closest possible split to equal
      - 2 2-to-4-line decoders
  - Complete using known 3-8 and 2-to-4 line decoders

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 1:
  - Find the truth table for the functions
  - Make a connection to the corresponding OR from the corresponding decoder output wherever a 1 appears in the truth table
- Approach 2
  - Find the minterms for each output function
  - OR the minterms together

Decoder and OR Gates Example

- **Implement a binary Adder**

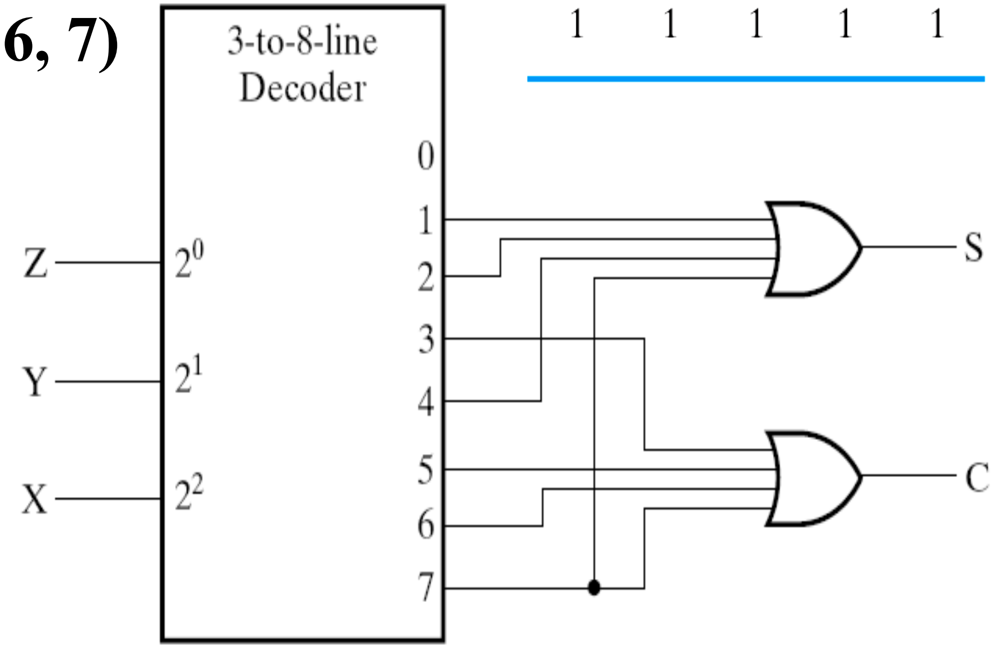
Truth Table

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

- **Finding sum of minterms expressions**

$S(X, Y, Z) = \Sigma_m(1, 2, 4, 7)$
$C(X, Y, Z) = \Sigma_m(3, 5, 6, 7)$

Find circuit



Decoder and OR Gates Example

- Implement the following set of odd parity functions of (A_7, A_6, A_5, A_3)

  $P_1 = A_7 \oplus A_5 \oplus A_3$

  $P_2 = A_7 \oplus A_6 \oplus A_3$

  $P_4 = A_7 \oplus A_6 \oplus A_5$

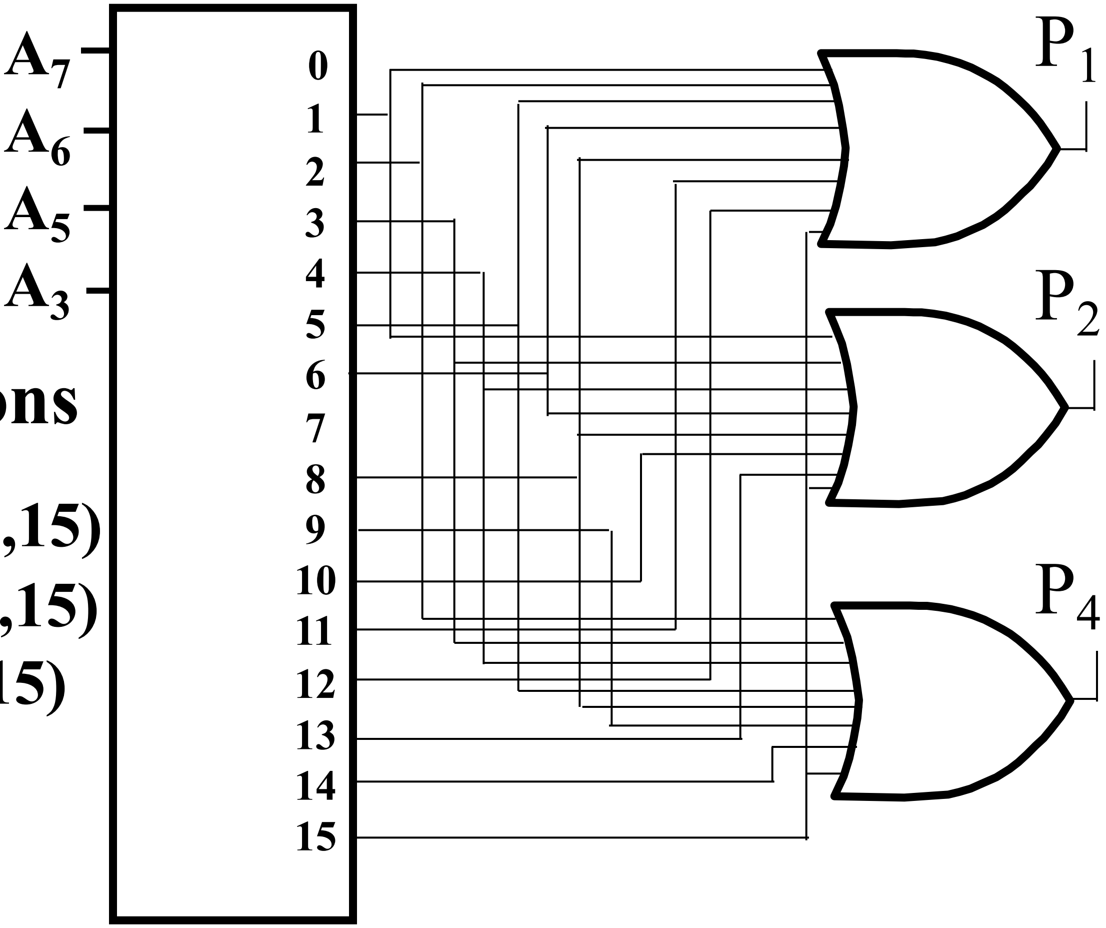
- Finding sum of minterms expressions

  $P_1 = \Sigma_m(1,2,5,6,8,11,12,15)$

  $P_2 = \Sigma_m(1,3,4,6,8,10,13,15)$

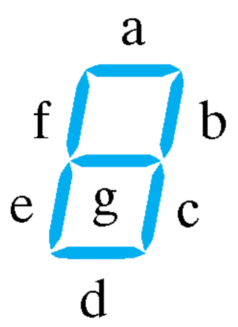
  $P_4 = \Sigma_m(2,3,4,5,8,9,14,15)$

- Find circuit
- Is this a good idea?

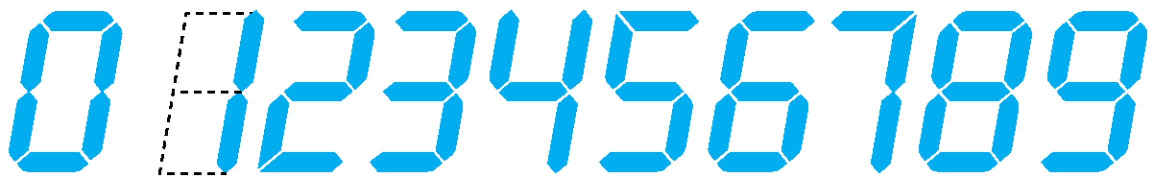


BCD-to-Segment Decoder

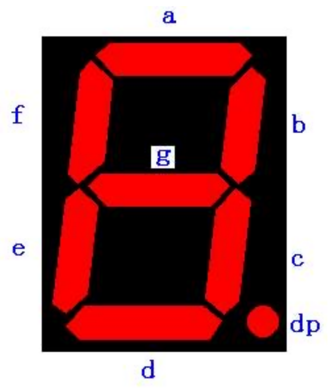
- **Seven-Segment Displayer**



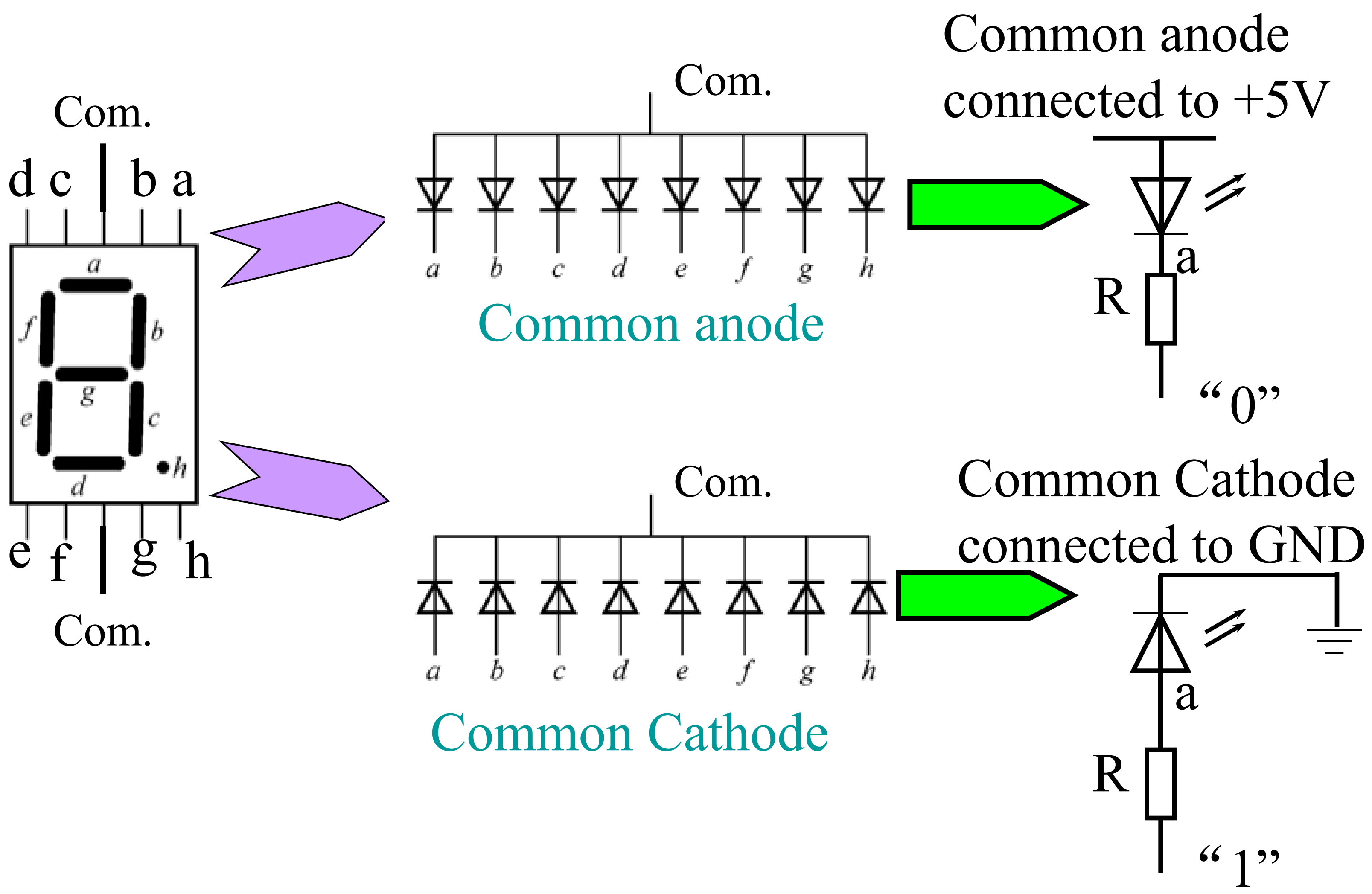
(a) Segment designation



(b) Numeric designation for display



Seven-Segment Displayer



BCD-to-Segment Decoder (Cont.)

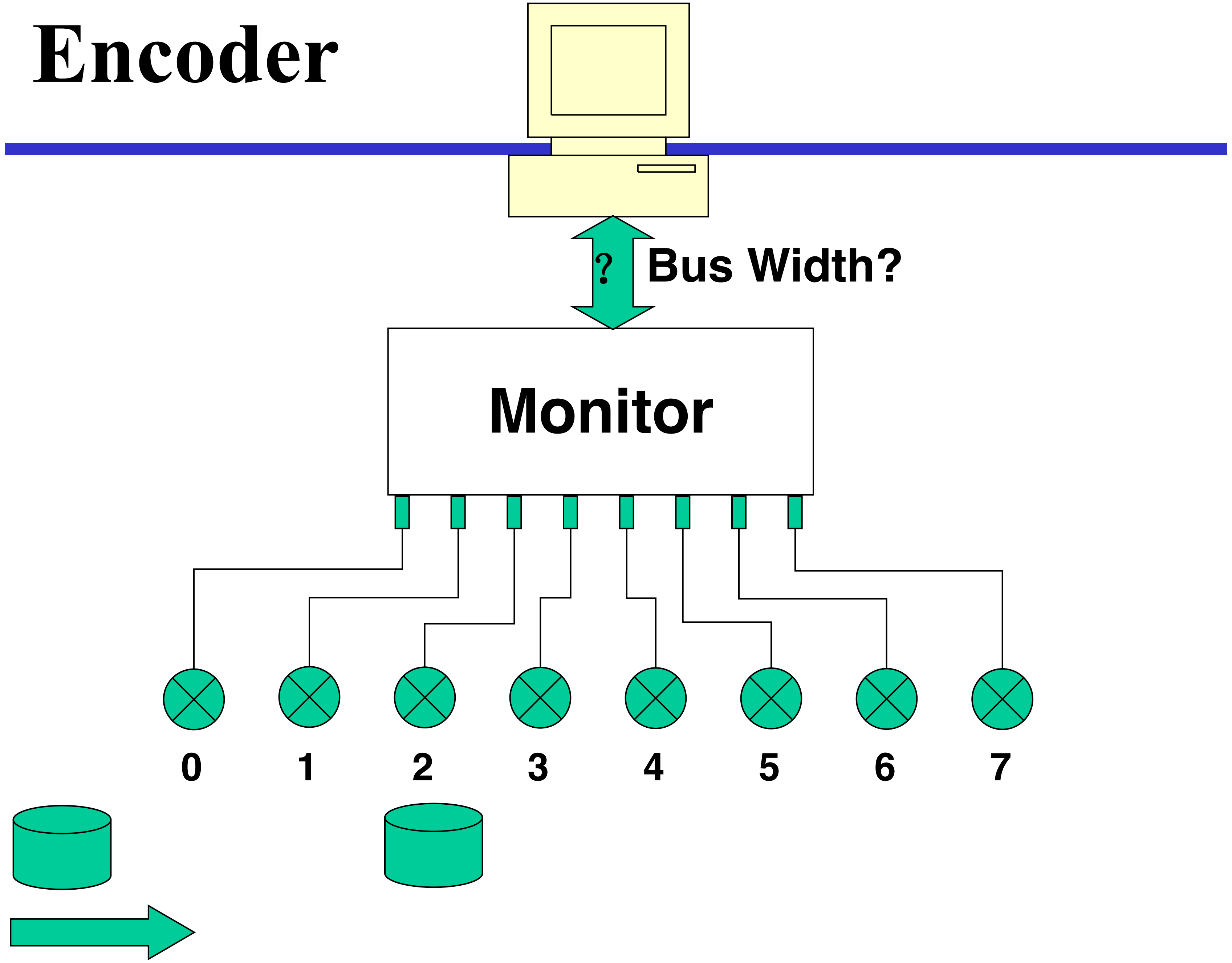
- **Truth Table for BCD-to-Seven-Segment Decoder**

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

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**
- **Typically, an encoder converts a code containing exactly one bit that is 1 to a binary code corres-ponding to the position in which the 1 appears.**

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.
- 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

- 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
1	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 (continued)

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

$$A_2 = D_4$$

$$A_1 = \overline{D}_4 D_3 + \overline{D}_4 \overline{D}_3 D_2 = \overline{D}_4 F_1, \; F_1 = (D_3 + D_2)$$

$$A_0 = \overline{D}_4 D_3 + \overline{D}_4 \overline{D}_3 \overline{D}_2 D_1 = \overline{D}_4 (D_3 + \overline{D}_2 D1)$$

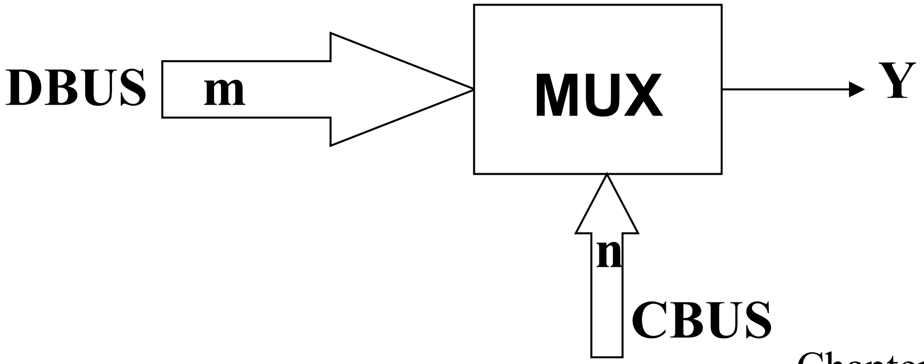
$$V = D_4 + F_1 + D_1 + D_0$$

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 single output
  - A set of control lines for making the selection
- 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**

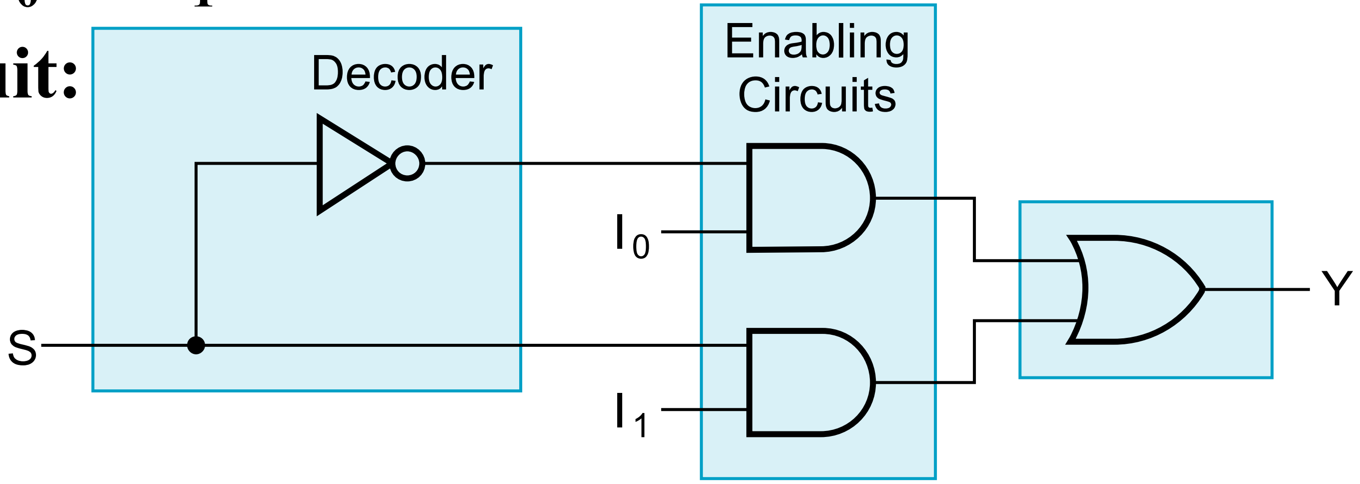


2-to-1-Line Multiplexer

- 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:

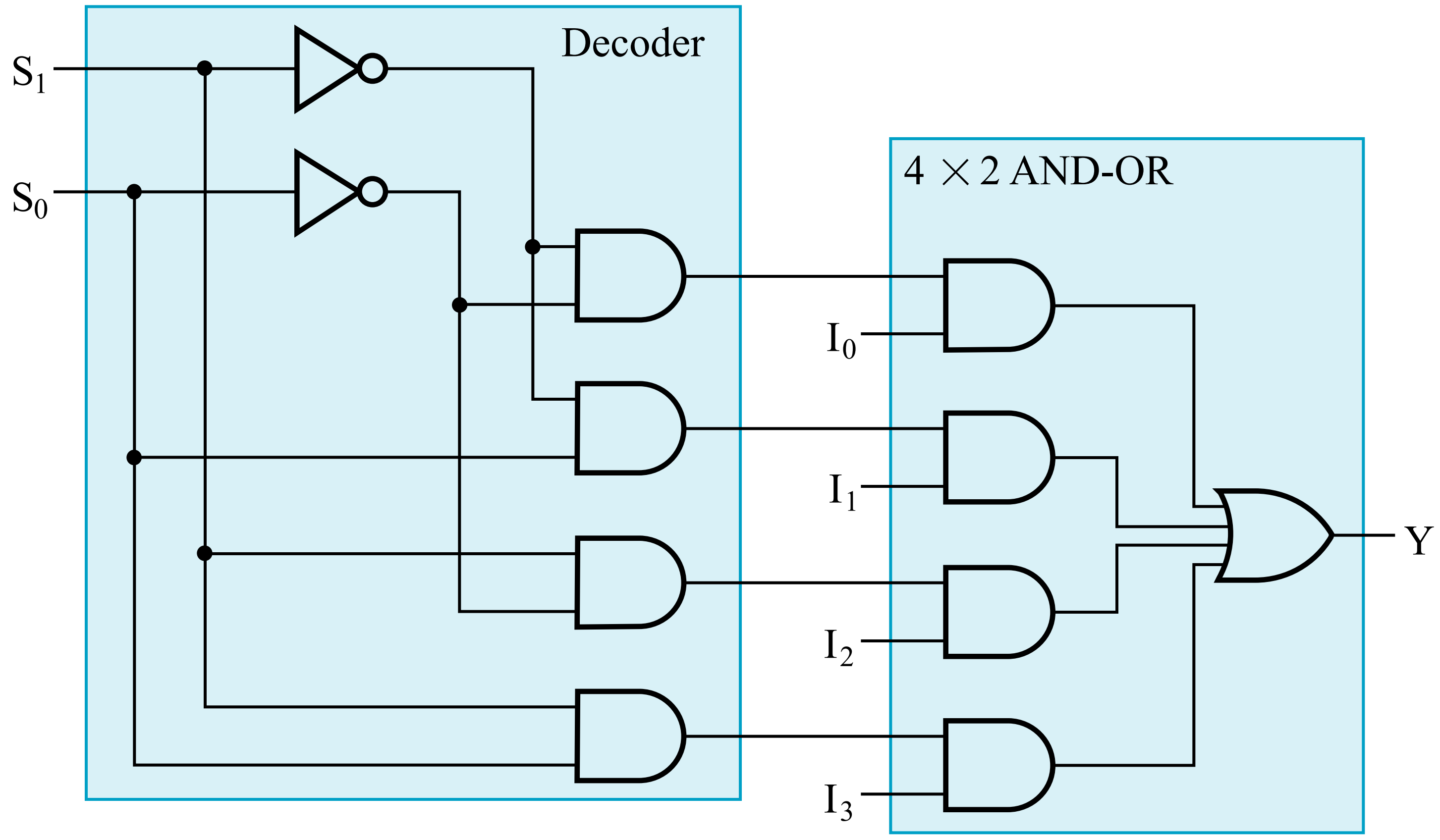


2-to-1-Line Multiplexer (continued)

- **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 $\times$ 2 AND-OR circuit:**
  - **1-to-2-line decoder**
  - **2 $\times$ 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**

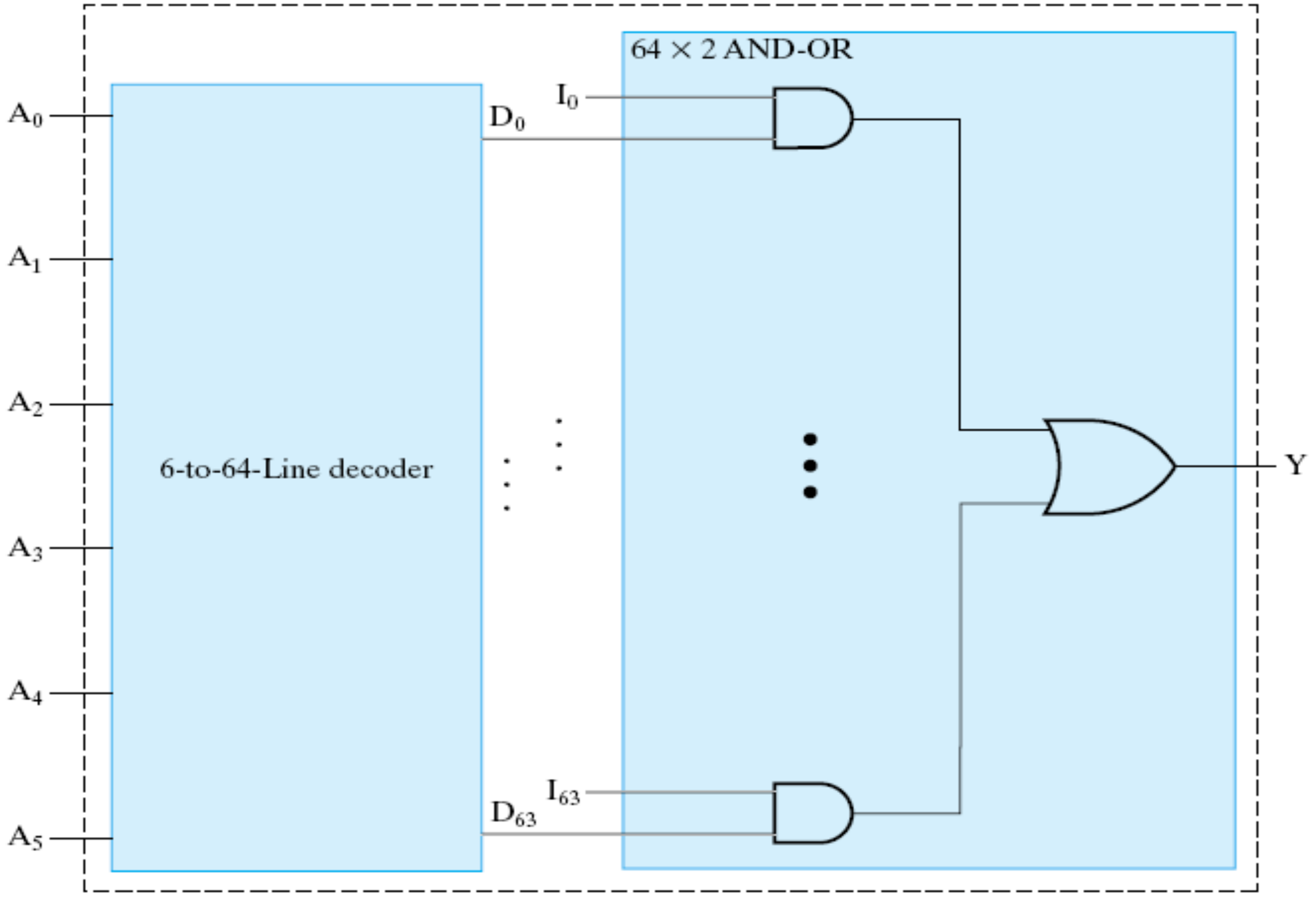
Example: 4-to-1-line Multiplexer

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



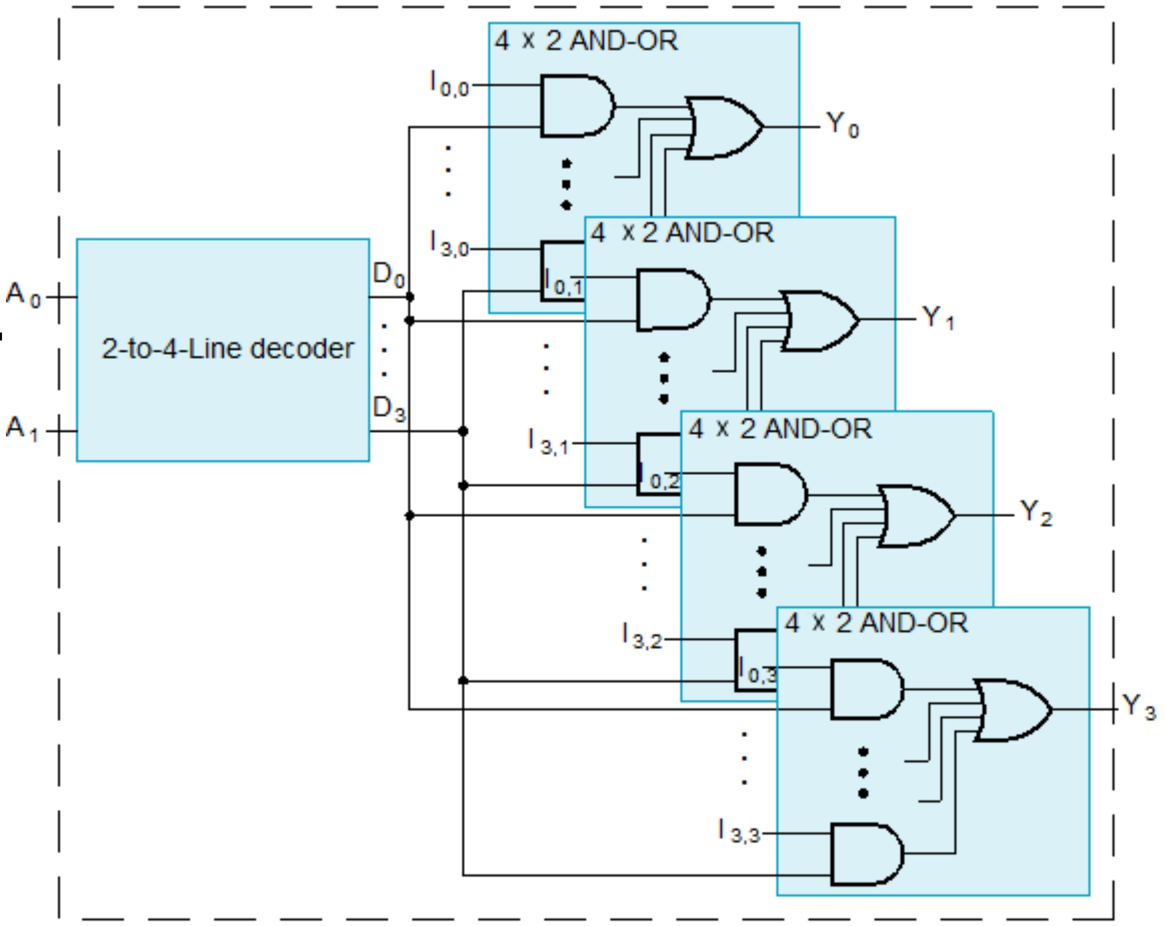
Example: 64-to-1-line Multiplexer

- 6-to-2^6-line decoder
- $2^6 \times 2$ AND-OR



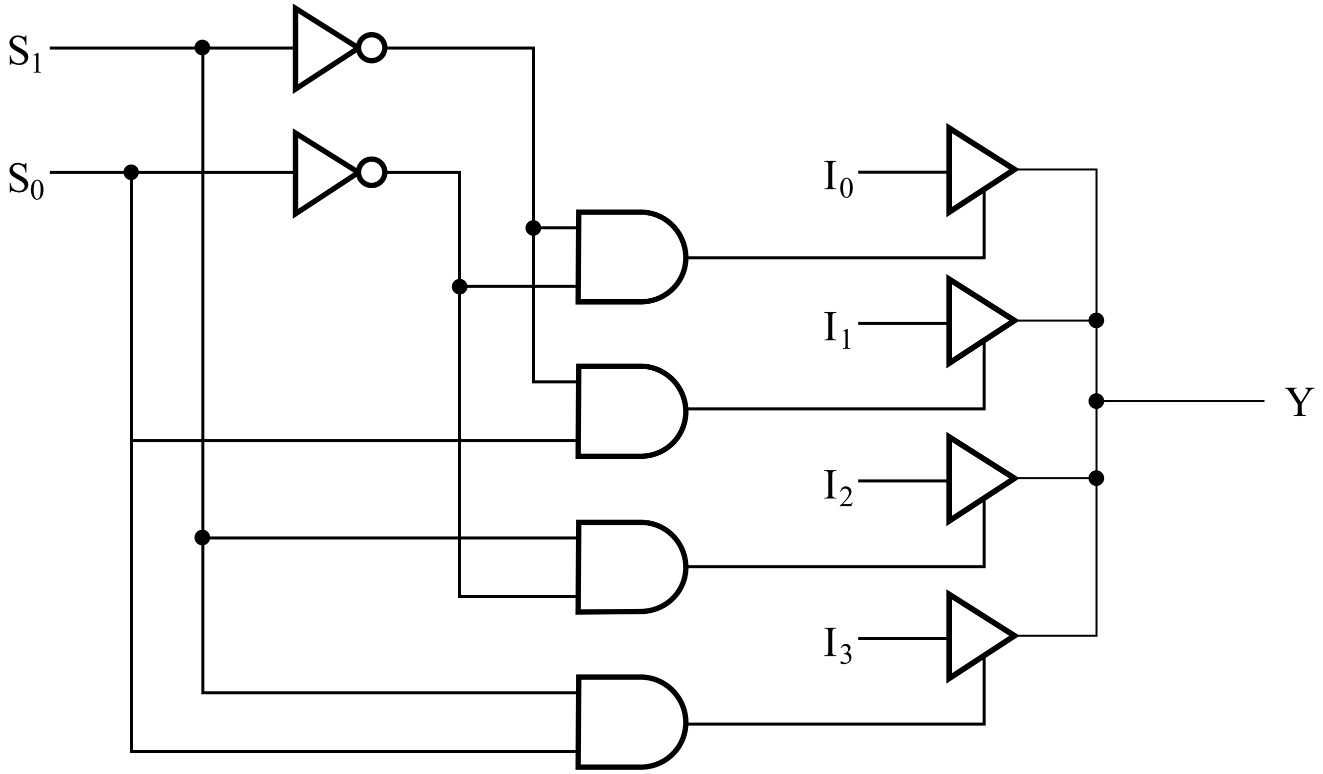
Multiplexer Width Expansion

- 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

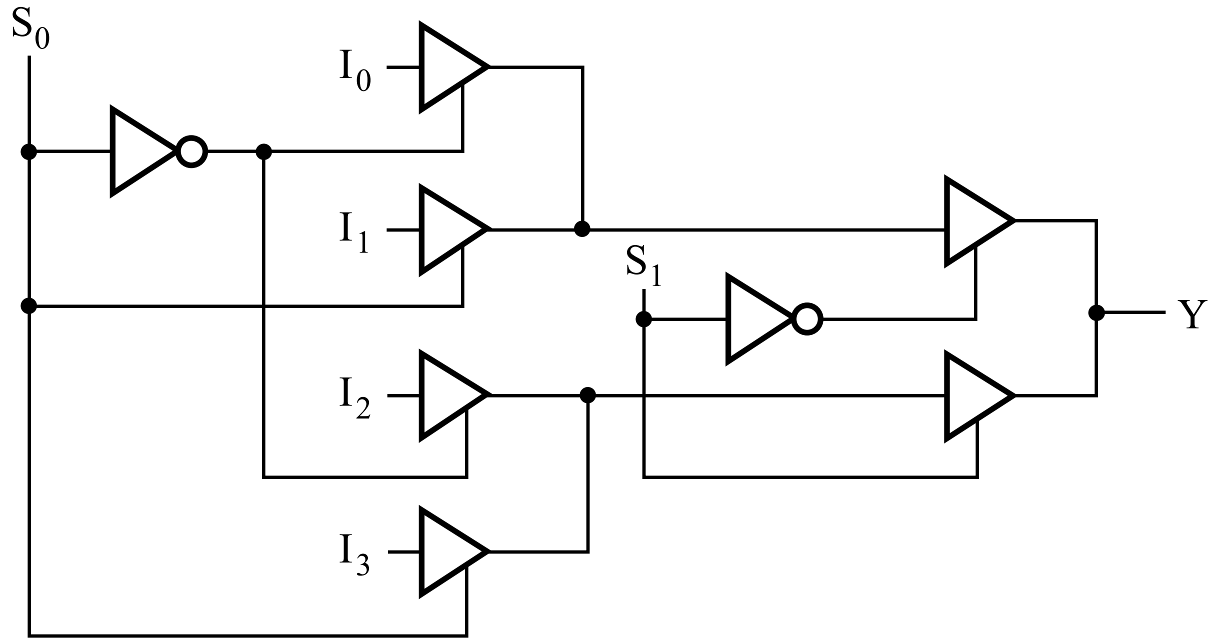
- Three-state logic in place of AND-OR



- Gate input cost = 18

Other Selection Implementations

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



- **Gate input cost = 14 compared to 22 (or 18) for gate implementation**

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 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

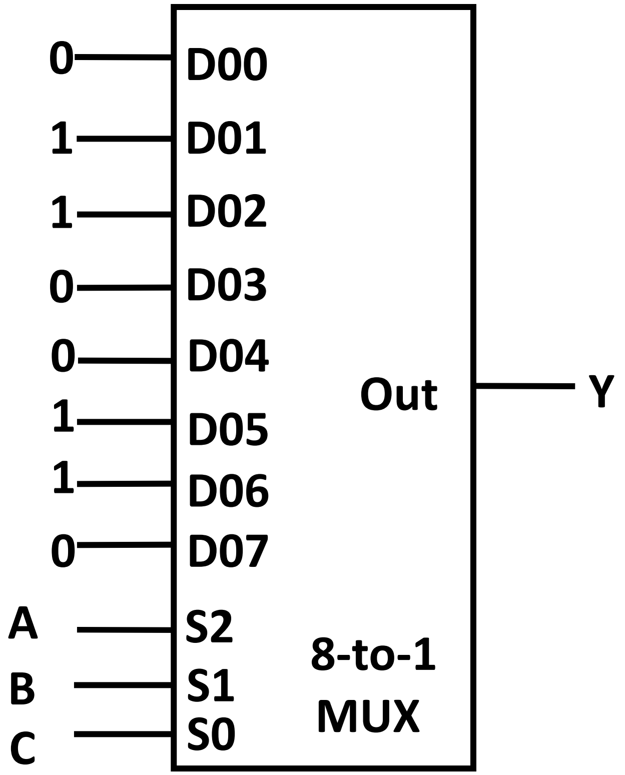
Gray to Binary (continued)

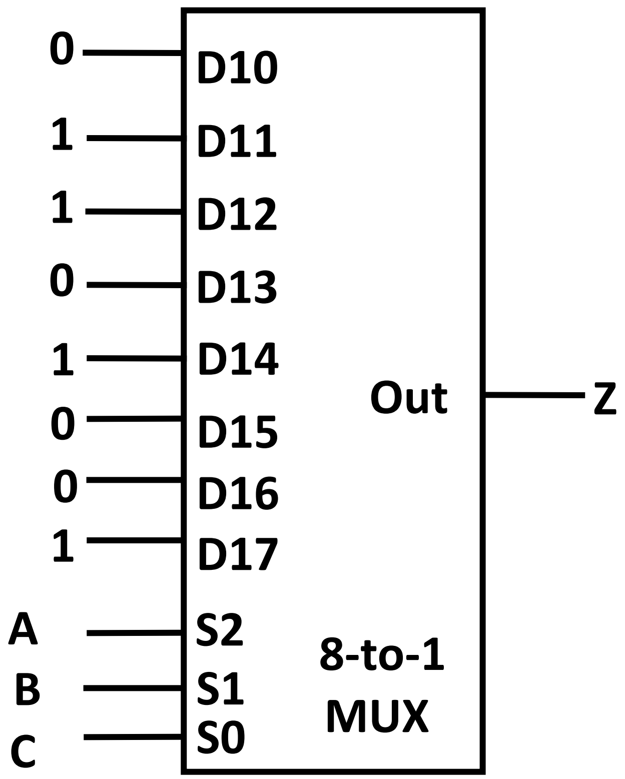
- **Rearrange the table so that the input combinations are in counting order**
- **Functions y and z can be implemented using a dual 8-to-1-line multiplexer by:**
  - **connecting A, B, and C to the multiplexer select inputs**
  - **placing y and z on the two multiplexer outputs**
  - **connecting their 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

- Implement any *m* functions of *n* + 1 variables by using:
  - An m-wide 2^n-to-1-line multiplexer
  - A single inverter
- Design:
  - Find the truth table for the functions.
  - Based on the values of the first *n* variables, separate the truth table rows into pairs
  - For each pair and output, define a rudimentary function of the final variable (0, 1, X, $\overline{X}$)
  - Using the first *n* variables as the index, value-fix the information inputs to the multiplexer with the corresponding rudimentary functions
  - Use the inverter to generate the rudimentary function $\overline{X}$

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

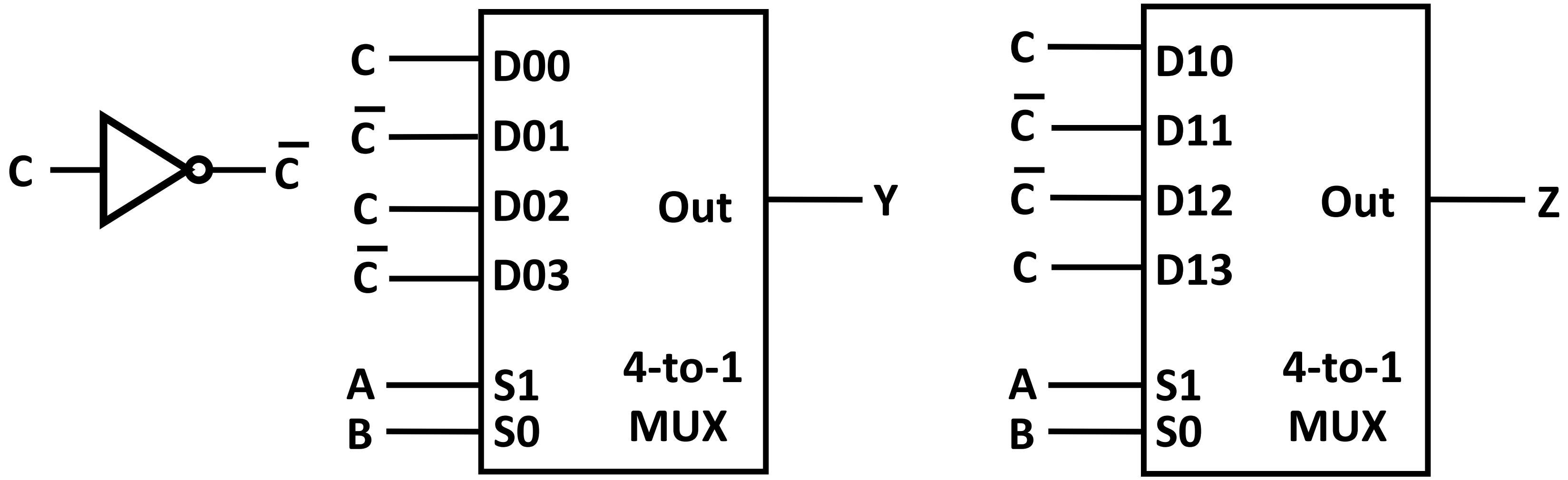
Gray to Binary (continued)

- **Rearrange the table so that the input combinations are in counting order, 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 <u>sub-functions</u> defined on a subset of the variables. The multiplexer then selects among these sub-functions.

Combinational Logic Implementation - Multiplexer Approach 2

- **Another Example**
  Using a 8-to-1 MUX to implement the function:

Input: A B C D	Output: X
0 0 0 0	0
0 0 0 1	1
0 0 1 0	1
0 0 1 1	0
0 1 0 0	0
0 1 0 1	0
0 1 1 0	0
0 1 1 1	1
1 0 0 0	1
1 0 0 1	1
1 0 1 0	1
1 0 1 1	1
1 1 0 0	1
1 1 0 1	0
1 1 1 0	0
1 1 1 1	0

Assignments

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