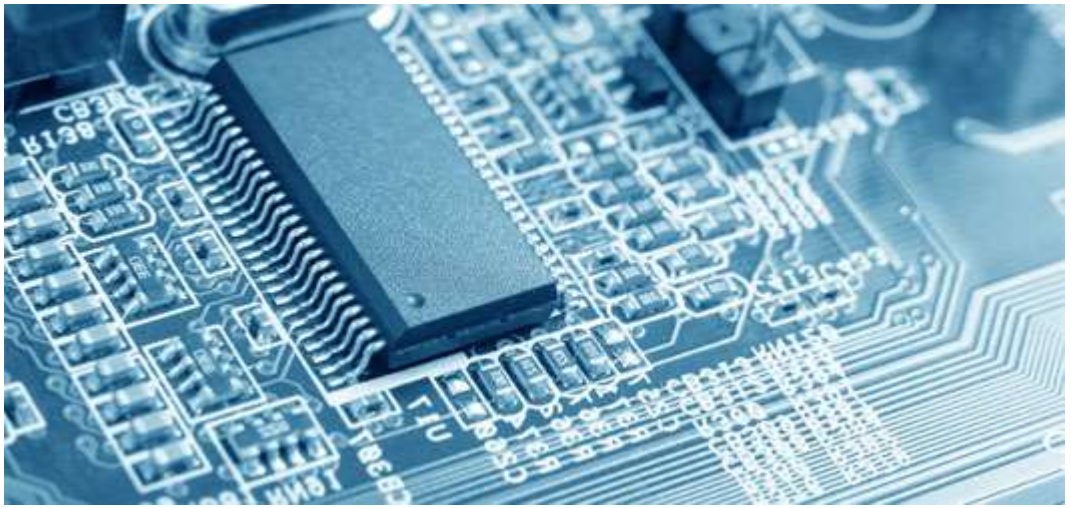
NAZARBAYEV UNIVERSITY
SCHOOL OF SCIENCE AND TECHNOLOGY



ROBT206 – Microcontrollers with Lab

Lecture 10 – Combinational Logic Design

8 February, 2018

Course Logistics

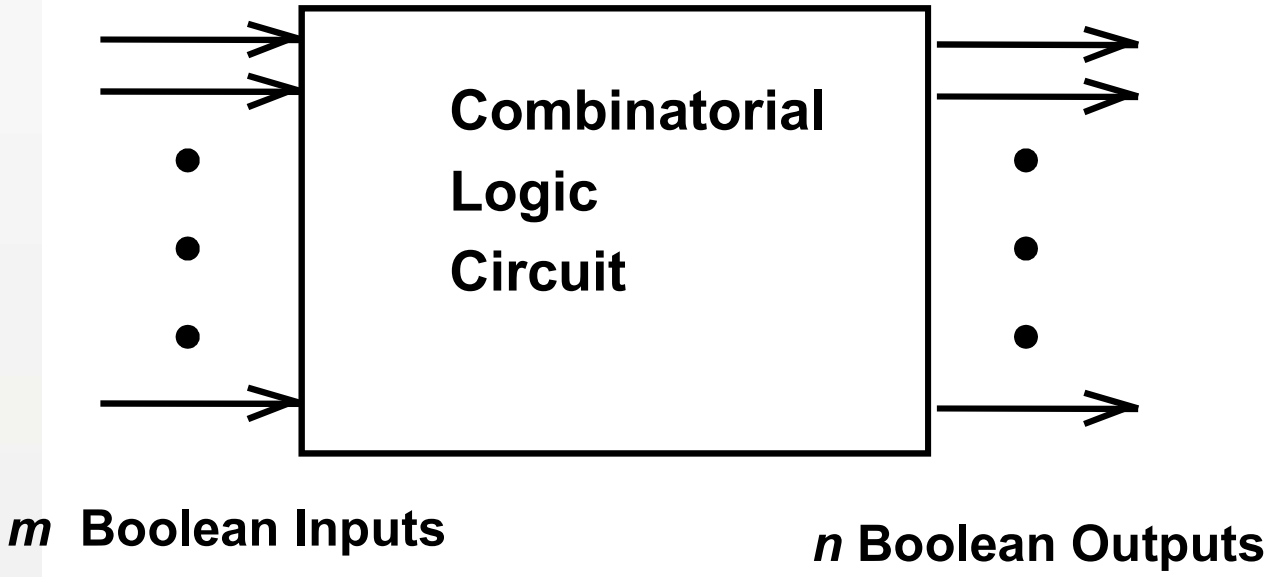
Important Dates and Tasks
Reading Assignment: Mano Chapters 3 and 4
Quiz #2 on 15 February: Till the end of Other gates lecture
Midterm exam on 22 Feb : Mano Chapters 1-4 (till the end of combinational logic)

Topics

Today's Topics
Design Procedure for Combinational Logic Design • Specification • Formulation • Optimization
Technology Mapping: • AND, OR and NOT • NAND and NOR
Verification: Manual and Simulation
Rudimentary Logic Functions
Decoding

Combinational Circuits

- A combinational logic circuit has:
  - A set of *m* Boolean inputs,
  - A set of *n* Boolean outputs, and
  - *n* switching functions, each mapping the 2^m input combinations to an output such that the current output depends only on the current input values
- A block diagram:



Design Procedure

1. **Specification**
   - Write a specification for the circuit if one is not already available
2. **Formulation**
   - Derive a truth table or initial Boolean equations that define the required relationships between the inputs and outputs, if not in the specification
   - Apply hierarchical design if appropriate
3. **Optimization**
   - Apply 2-level and multiple-level optimization
   - Draw a logic diagram or provide a netlist for the resulting circuit using ANDs, ORs, and inverters

Design Procedure

4. **Technology Mapping**
   - Map the logic diagram or netlist to the implementation technology selected
5. **Verification**
   - Verify the correctness of the final design manually or using simulation

Design Example

1. **Specification**
   - BCD to Excess-3 code converter
   - Transforms BCD code for the decimal digits to Excess-3 code for the decimal digits
   - BCD code words for digits 0 through 9: 4-bit patterns 0000 to 1001, respectively
   - Excess-3 code words for digits 0 through 9: 4-bit patterns consisting of 3 (binary 0011) added to each BCD code word
   - Implementation:
     - multiple-level circuit
     - AND, OR and NOT gates

Design Example (Continued)

2. **Formulation**

- Conversion of 4-bit codes can be most easily formulated by a truth table
- Variables
  - BCD: A,B,C,D
- Variables
  - Excess-3 W,X,Y,Z
- Don't Cares
  - BCD 1010 to 1111

Input BCD A B C D	Output Excess-3 WXYZ
0 0 0 0	0 0 1 1
0 0 0 1	0 1 0 0
0 0 1 0	0 1 0 1
0 0 1 1	0 1 1 0
0 1 0 0	0 1 1 1
0 1 0 1	1 0 0 0
0 1 1 0	1 0 0 1
0 1 1 1	1 0 1 0
1 0 0 0	1 0 1 1
1 0 0 1	1 1 0 0

Design Example (Continued)

3. **Optimization**

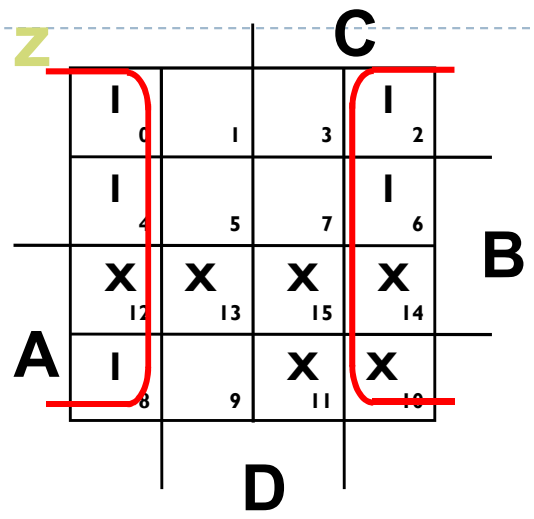
   2-level using K-maps

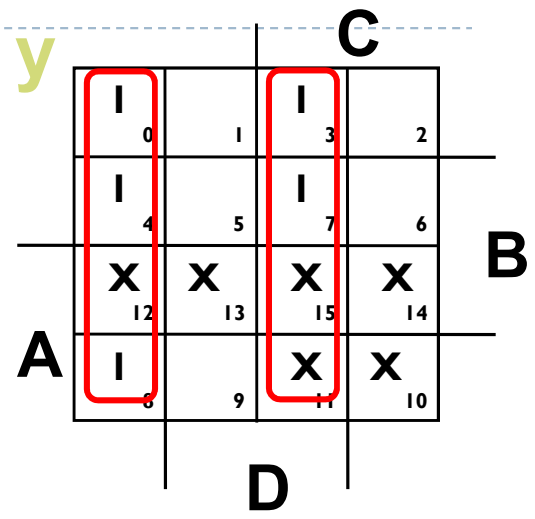
$W = A + BC + BD$

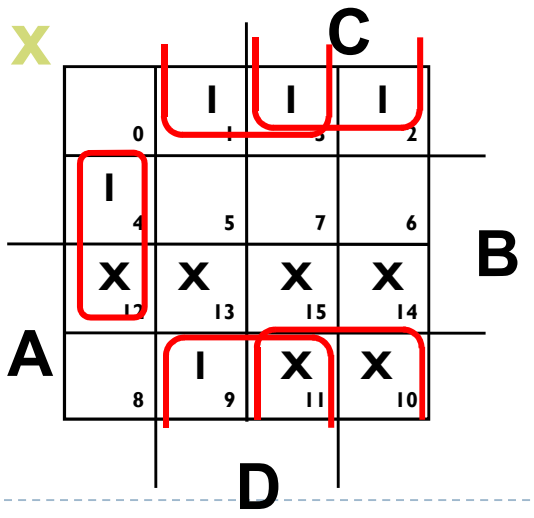
$X = \overline{B}D + \overline{B}C + B\overline{C}\,\overline{D}$

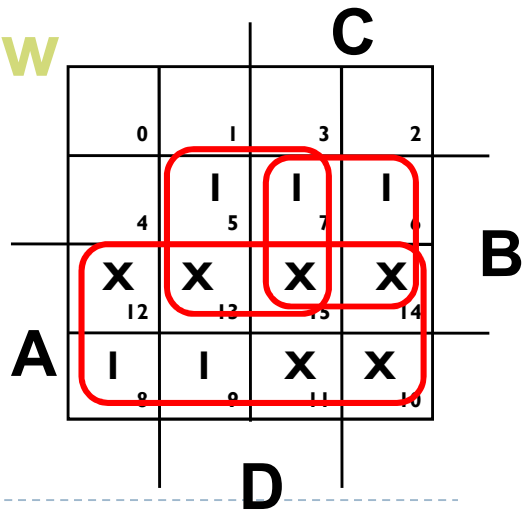
$Y = CD + \overline{C}\,\overline{D}$

$Z = \overline{D}$









Design Example (Continued)

3. **Optimization (continued)**

- Multiple-level using transformations

$$W = A + BC + BD$$
$$X = \overline{B}\,C + \overline{B}\,D + B\,\overline{C}\,\overline{D}$$
$$Y = CD + \overline{C}\,\overline{D}$$
$$Z = \overline{D}$$

- Perform extraction, finding factor:

$$T_1 = C + D$$
$$W = A + BT_1$$
$$X = \overline{B}\,T_1 + B\,\overline{C}\,\overline{D}$$
$$Y = CD + \overline{C}\,\overline{D}$$
$$Z = \overline{D}$$

Design Example (Continued)

3. Optimization (continued)

- Multiple-level using transformations

$T_1 = C + D$

$W = A + BT_1$

$X = \overline{B}\,T_1 + B\overline{C}\,\overline{D}$

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

$Z = \overline{D}$

- An additional extraction not shown in the text since it uses a Boolean transformation: ($\overline{C}\,\overline{D} = \overline{C + D} = \overline{T}_1$):

$W = A + BT_1$

$X = \overline{B}\,T_1 + B\overline{T}_1$

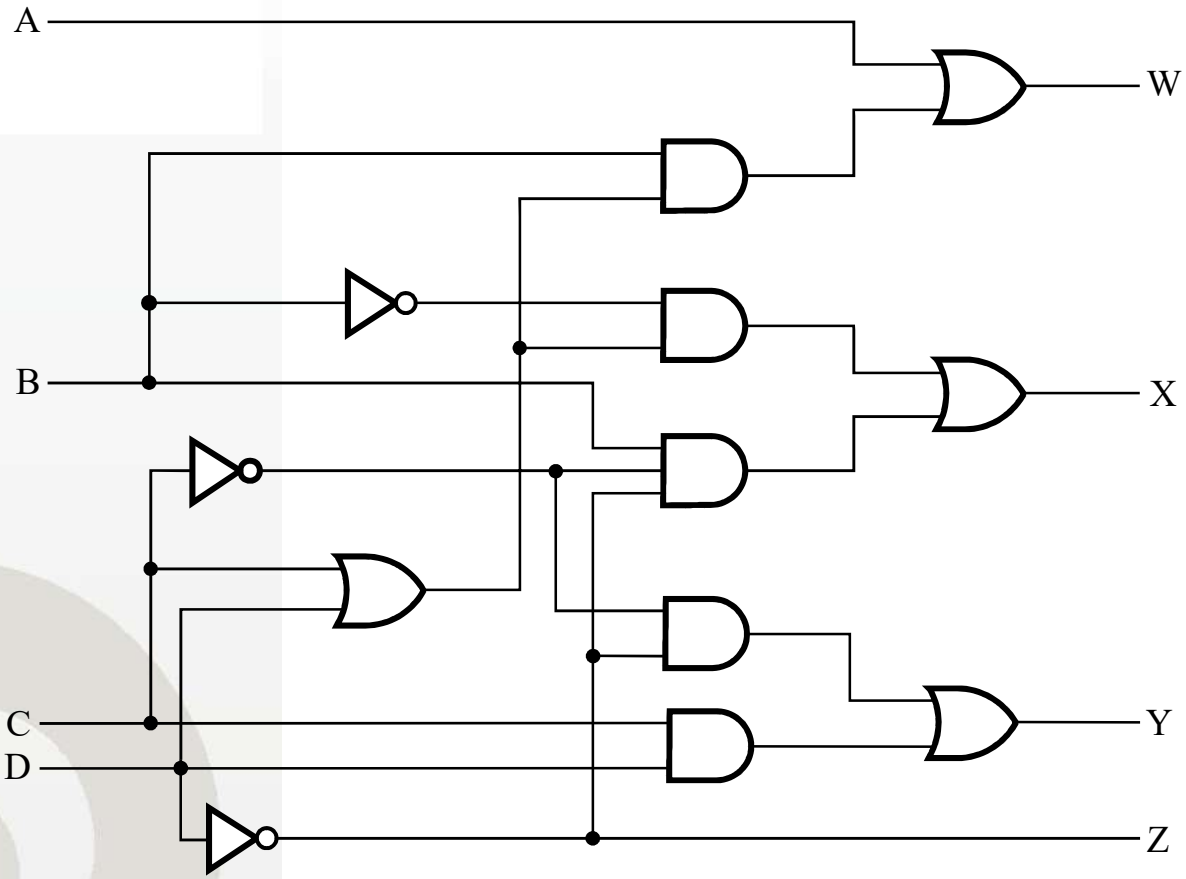
$Y = CD + \overline{T}_1$

$Z = \overline{D}$

Design Example (Continued)

4. Technology Mapping

Mapping with a library containing inverters, 2-input AND and OR gates

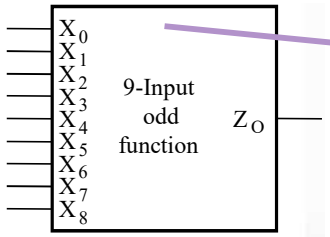


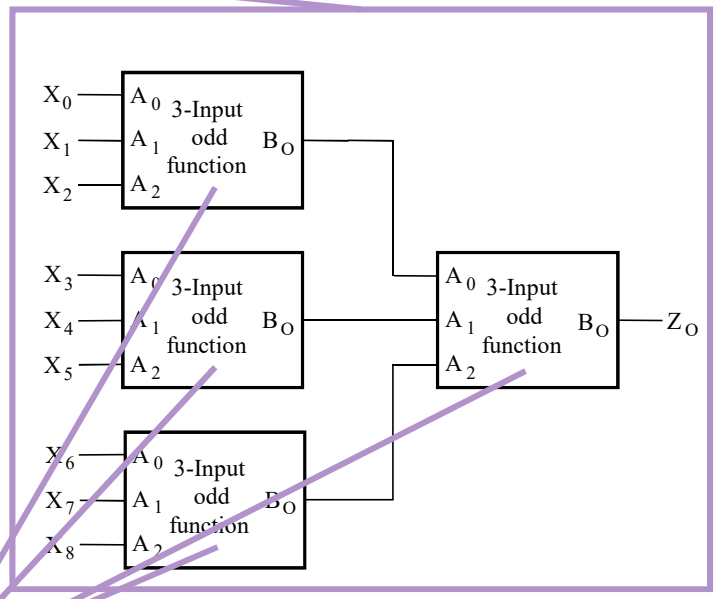
Beginning Hierarchical Design

- 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 parity tree (see next slide)
  - 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

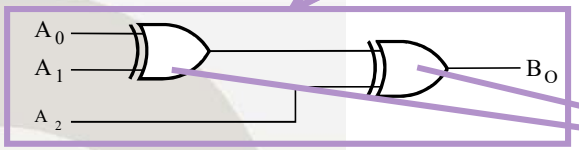
Hierarchy for Parity Tree Example

(a) Symbol for circuit

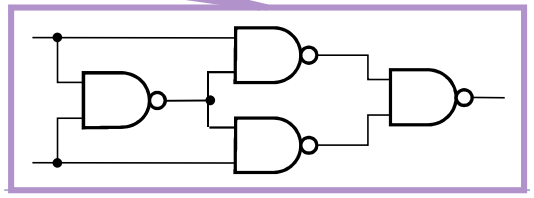




(b) Circuit as interconnected 3-input odd function blocks



(c) 3-input odd function circuit as interconnected exclusive-OR blocks



(d) Exclusive-OR block as interconnected NANDs

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

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.
- Much of the material devoted to learning about combinational blocks uses the top-down design.

Technology Mapping

- Mapping Procedures
  - To NAND gates
  - To NOR gates

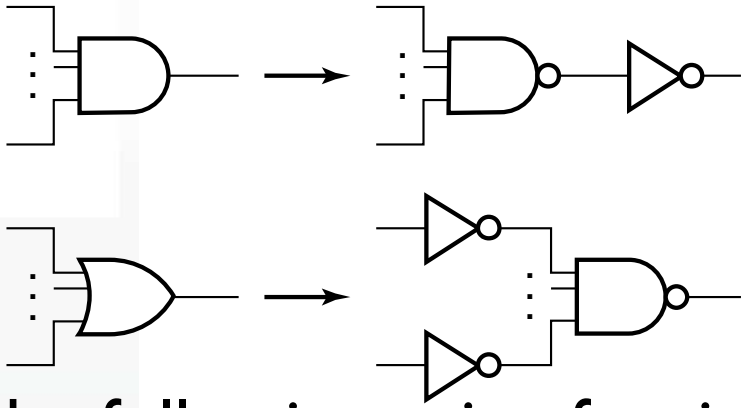
Mapping to NAND gates

- Assumptions:
  - Gate loading and delay are ignored
  - Cell library contains an inverter and *n*-input NAND gates, *n* = 2, 3, …
  - An AND, OR, inverter schematic for the circuit is available

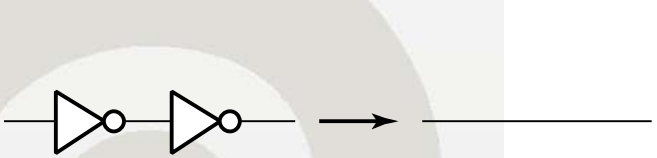
- The mapping is accomplished by:
  - Replacing AND and OR symbols,
  - Pushing inverters through circuit fan-out points, and
  - Canceling inverter pairs

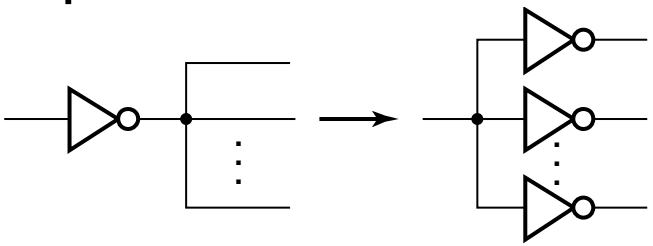
NAND Mapping Algorithm

1. Replace ANDs and ORs:

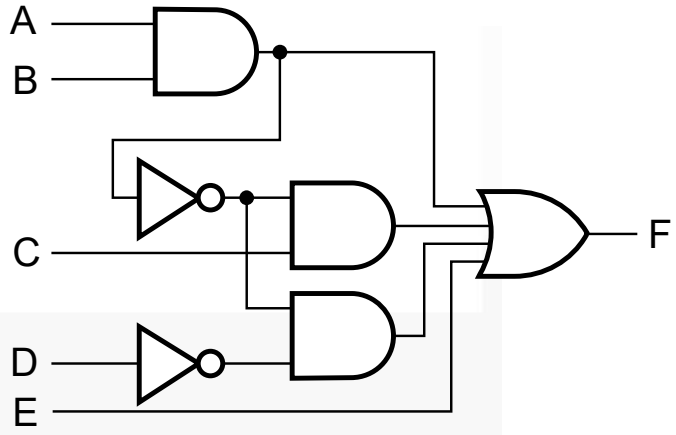


2. Repeat the following pair of actions until there is at most one inverter between:
   a. A circuit input or driving NAND gate output, and
   b. The attached NAND gate inputs.

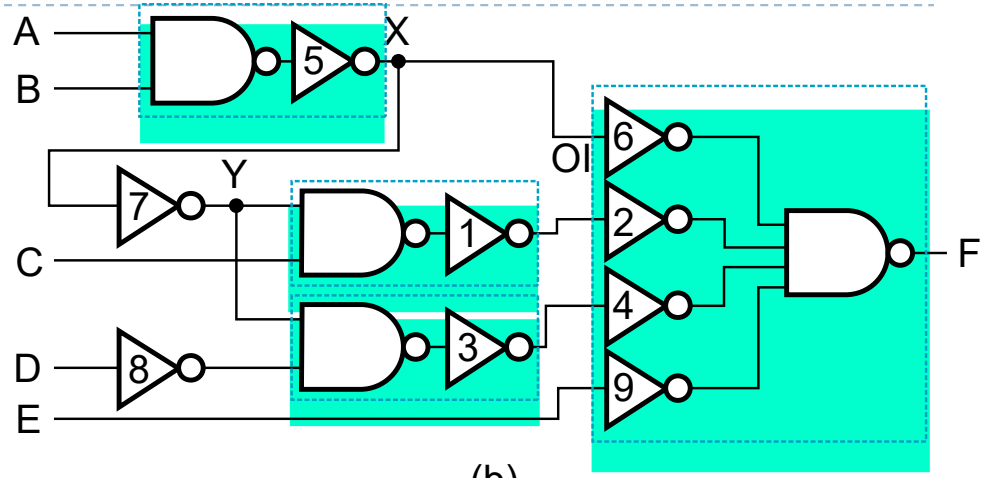




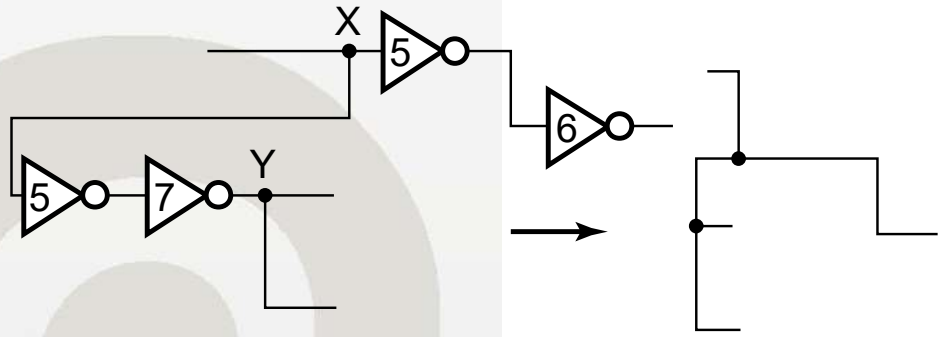
NAND Mapping Example



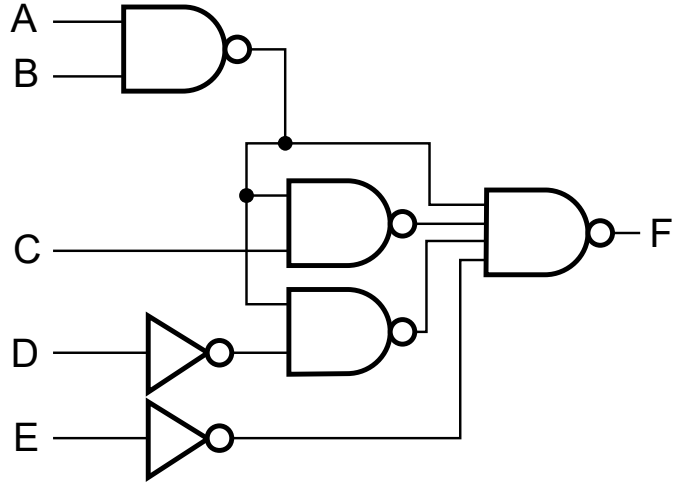
(a)



(b)



(c)



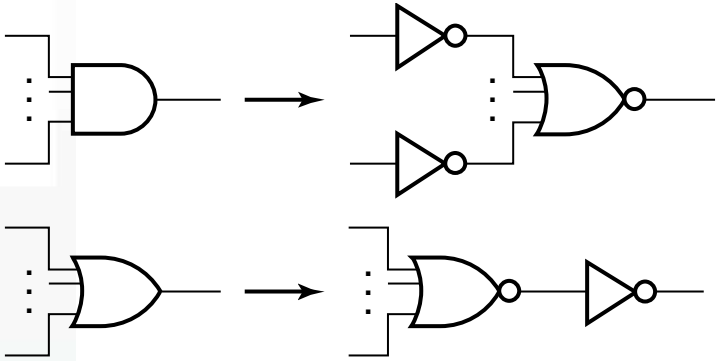
(d)

Mapping to NOR gates

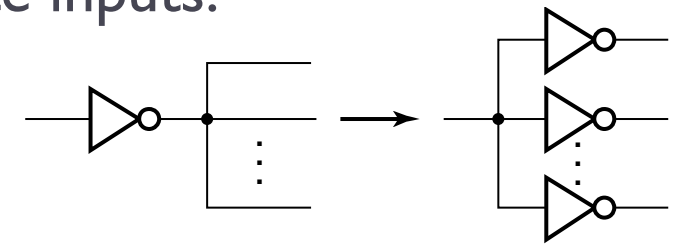
- Assumptions:
  - Gate loading and delay are ignored
  - Cell library contains an inverter and *n*-input NOR gates, *n* = 2, 3, …
  - An AND, OR, inverter schematic for the circuit is available
- The mapping is accomplished by:
  - Replacing AND and OR symbols,
  - Pushing inverters through circuit fan-out points, and
  - Canceling inverter pairs

NOR Mapping Algorithm

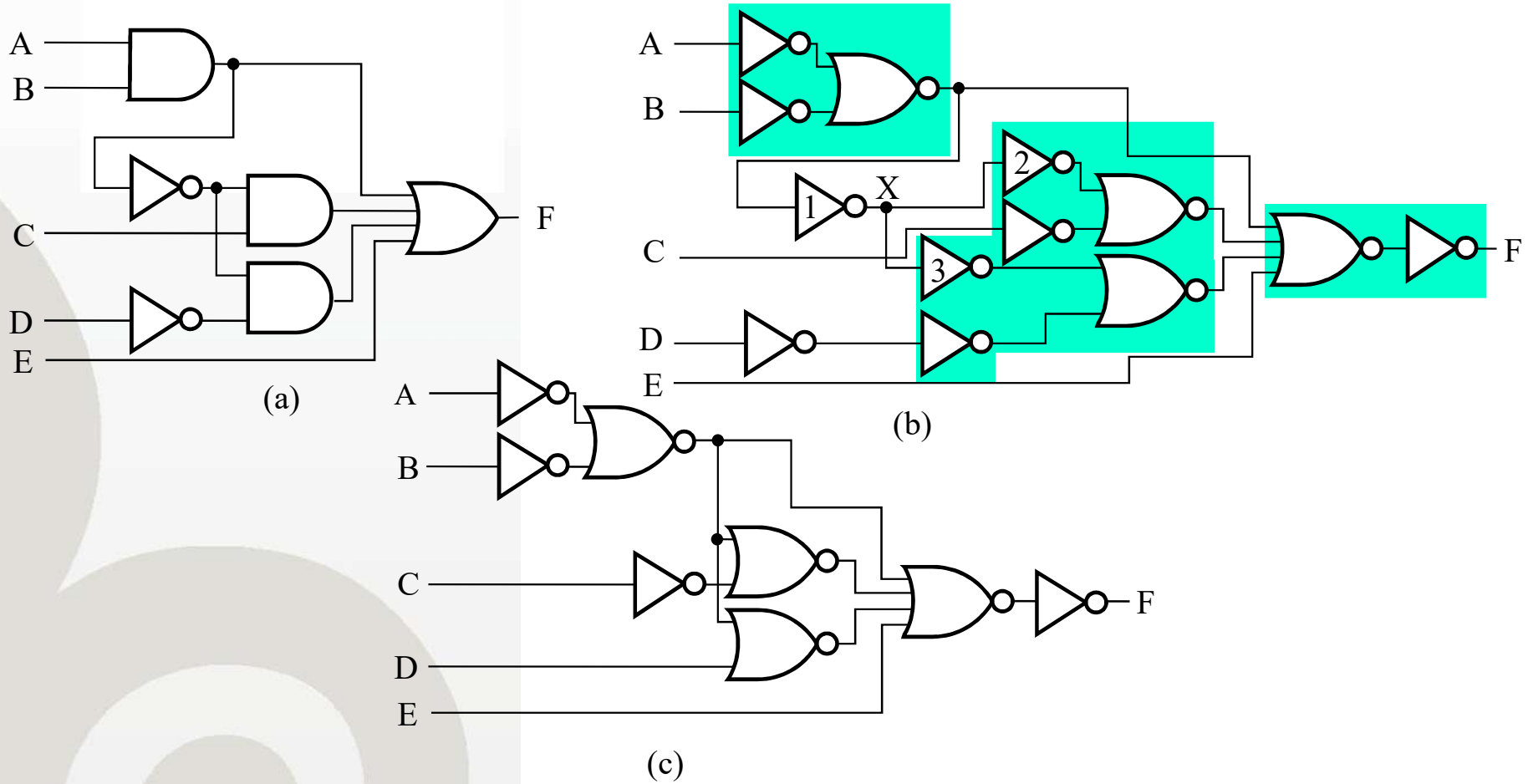
1. Replace ANDs and ORs:



2. Repeat the following pair of actions until there is at most one inverter between :
   a. A circuit input or driving NAND gate output, and
   b. The attached NAND gate inputs.



NOR Mapping Example



Verification

- Verification - show that the final circuit designed implements the original specification
- Simple specifications are:
  - truth tables
  - Boolean equations
  - HDL code
- If the above result from formulation and are not the original specification, it is critical that the formulation process be flawless for the verification to be valid!

Basic Verification Methods

- Manual Logic Analysis
  - Find the truth table or Boolean equations for the final circuit
  - Compare the final circuit truth table with the specified truth table, or
  - Show that the Boolean equations for the final circuit are equal to the specified Boolean equations
- Simulation
  - Simulate the final circuit (or its netlist, possibly written as an HDL) and the specified truth table, equations, or HDL description using test input values that fully validate correctness.
  - The obvious test for a combinational circuit is application of all possible "care" input combinations from the specification

Verification Example: Manual Analysis

- Find the circuit truth table from the equations and compare to specification truth table:

Input BCD A B C D	Output Excess-3 WXYZ
0 0 0 0	0 0 1 1
0 0 0 1	0 1 0 0
0 0 1 0	0 1 0 1
0 0 1 1	0 1 1 0
0 1 0 0	0 1 1 1
0 1 0 1	1 0 0 0
0 1 1 0	1 0 0 1
0 1 1 1	1 0 1 0
1 0 0 0	1 0 1 1
1 0 0 1	1 0 1 1

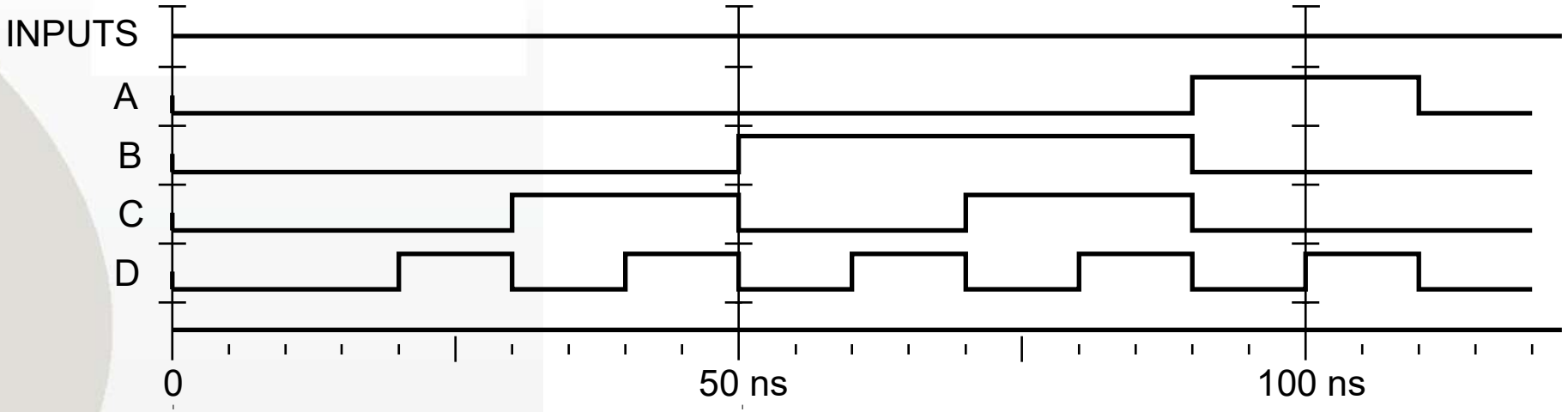
The tables match!

Verification Example: Simulation

- Simulation procedure:
  - Use a schematic editor or text editor to enter a gate level representation of the final circuit
  - Use a waveform editor or text editor to enter a test consisting of a sequence of input combinations to be applied to the circuit
    - This test should guarantee the correctness of the circuit if the simulated responses to it are correct
    - Short of applying all possible "care" input combinations, generation of such a test can be difficult

Verification Example: Simulation

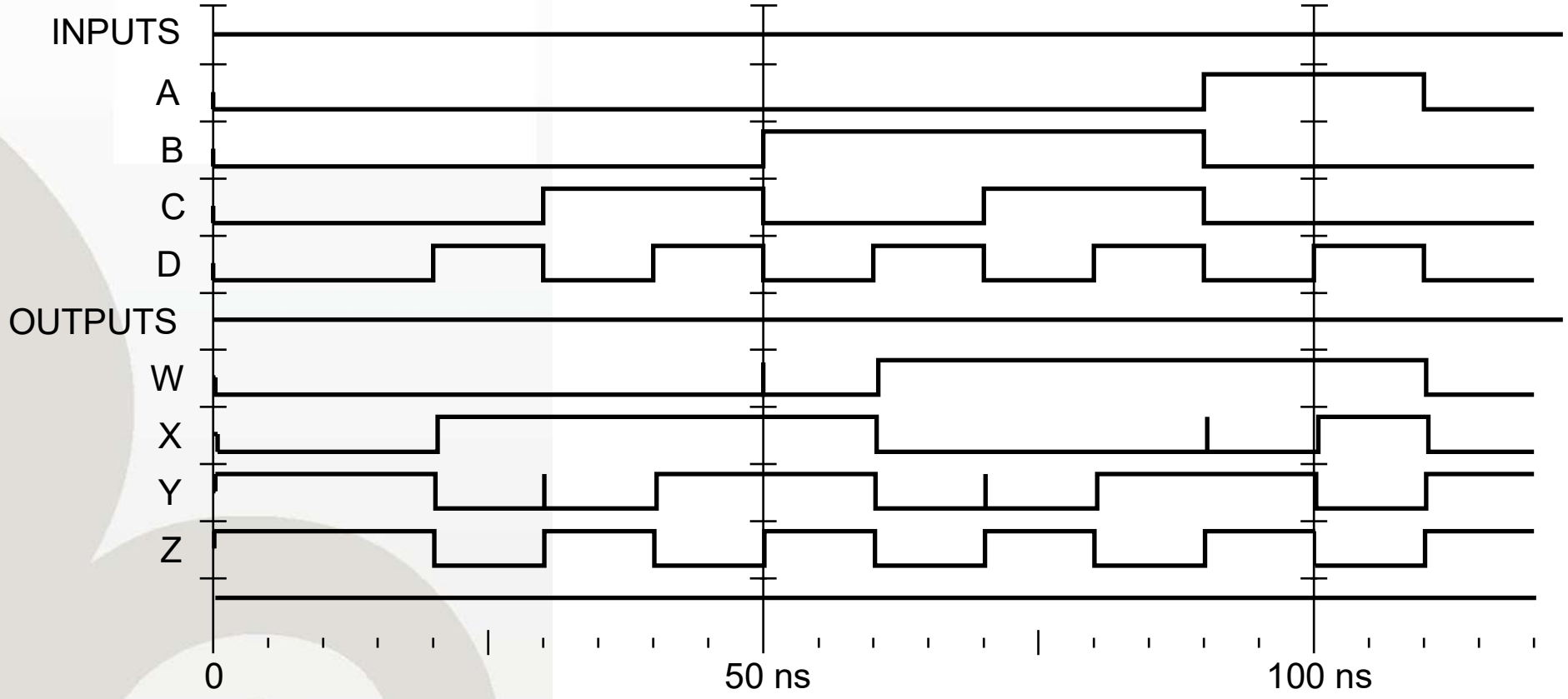
- Enter waveform that applies all possible input combinations:



- Are all BCD input combinations present? (Low is a 0 and high is a one)

Verification Example: Simulation

- Run the simulation of the circuit for 120 ns

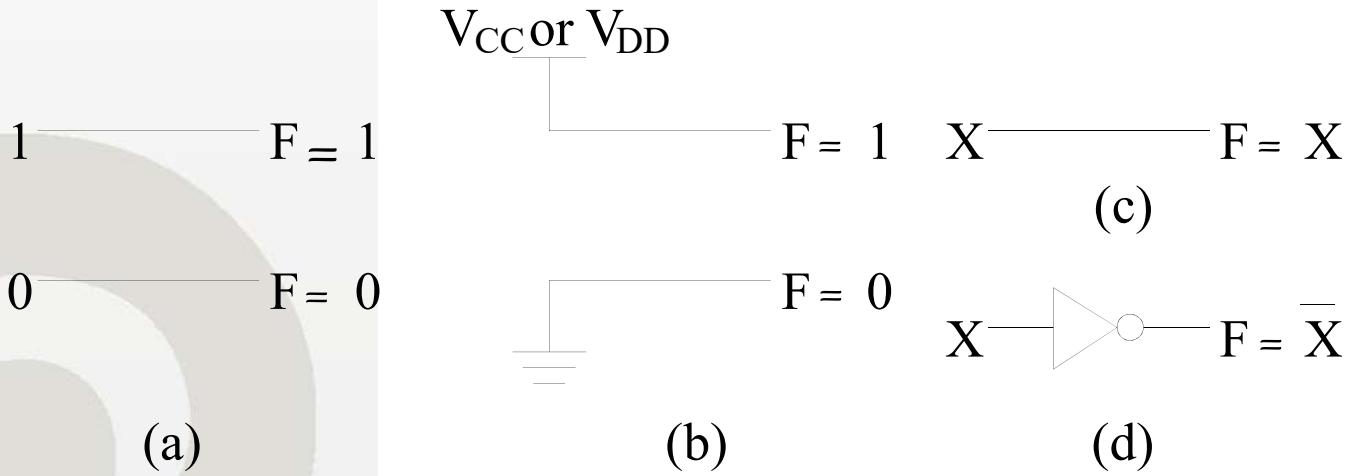


Rudimentary Logic Functions

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

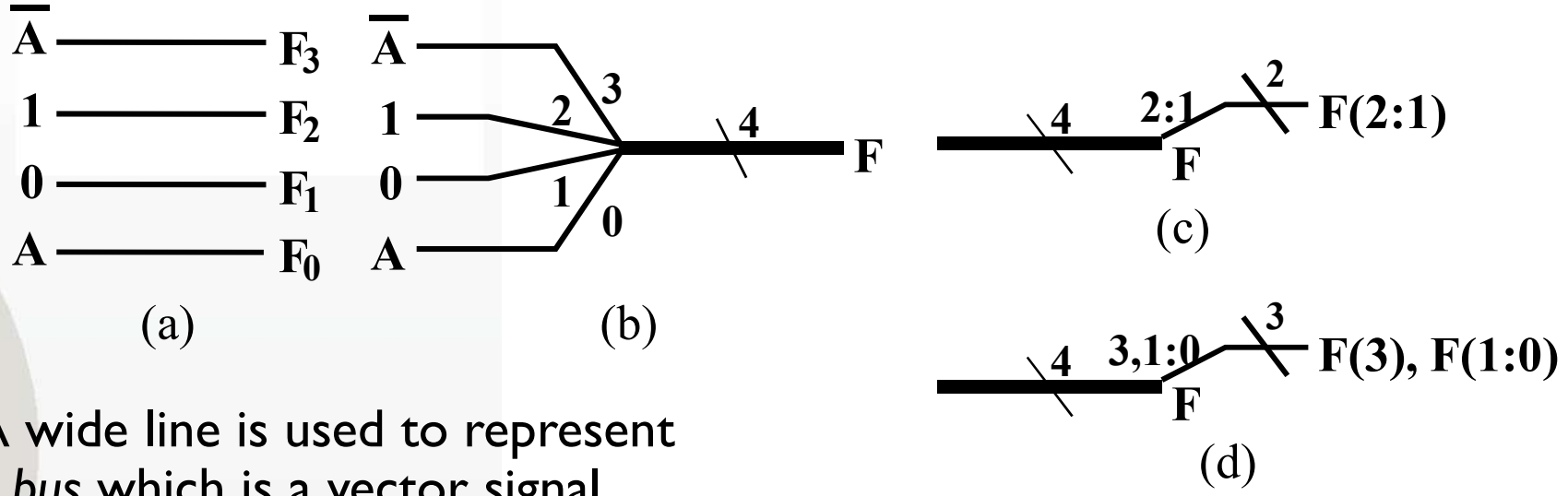
TABLE 4-1
Functions of One Variable

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



Multiple-bit Functions

- Multi-bit Examples:

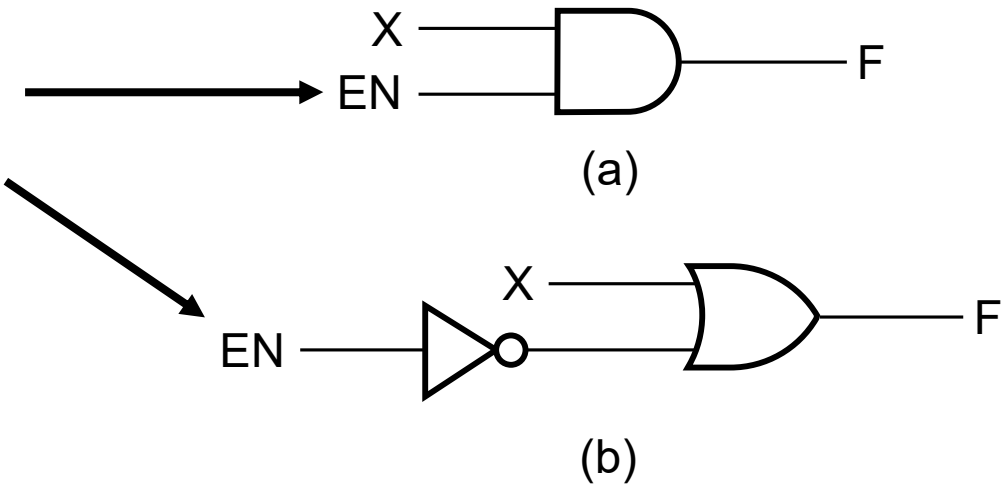


- A wide line is used to represent a *bus* which is a vector signal
- In (b) of the example, F = (F_3, F_2, F_1, F_0) is a bus.
- The bus can be split into individual bits as shown in (b)
- Sets of bits can be split from the bus as shown in (c) for bits 2 and 1 of F.
- The sets of bits need not be continuous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



Decoding

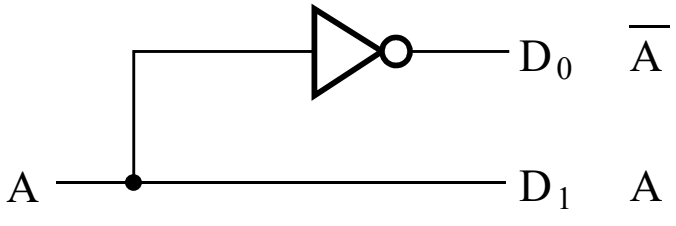
- Decoding – the conversion of an *n*-bit input code to an *m*-bit output code with $n \leq m \leq 2^n$ such that each valid code word produces a unique output code
- Circuits that perform decoding are called ***decoders***
- Here, functional blocks for decoding are
  - called *n*-to-*m* line decoders,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

Decoder Examples

- 1-to-2-Line Decoder

A	D_0	D_1
0	1	0
1	0	1

(a)

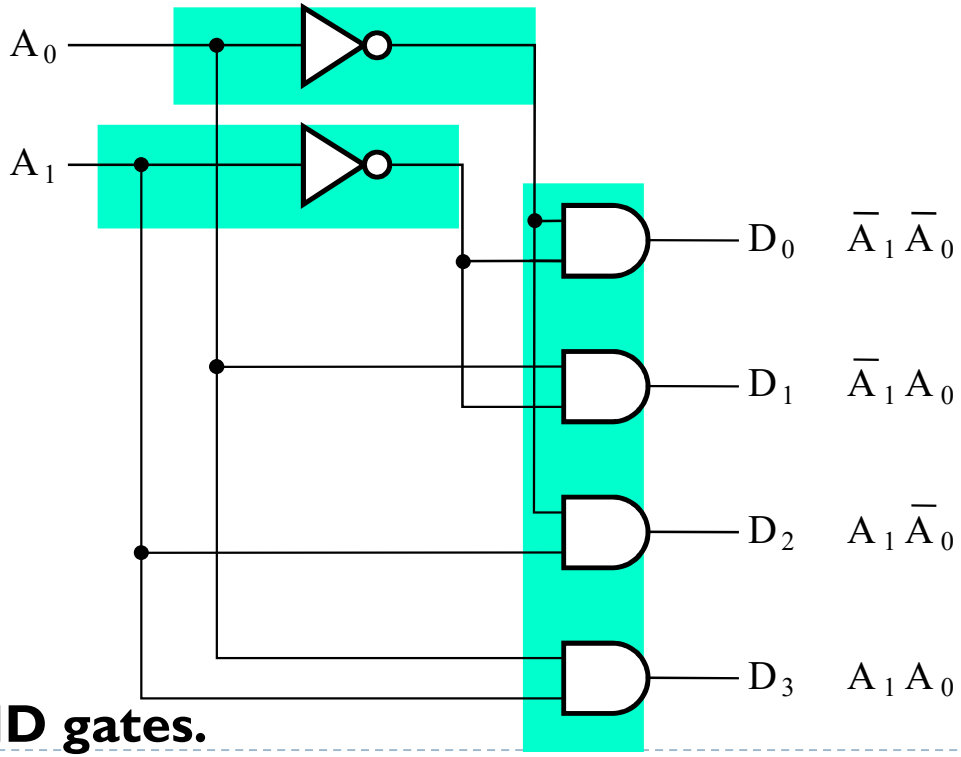


(b)

- 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)

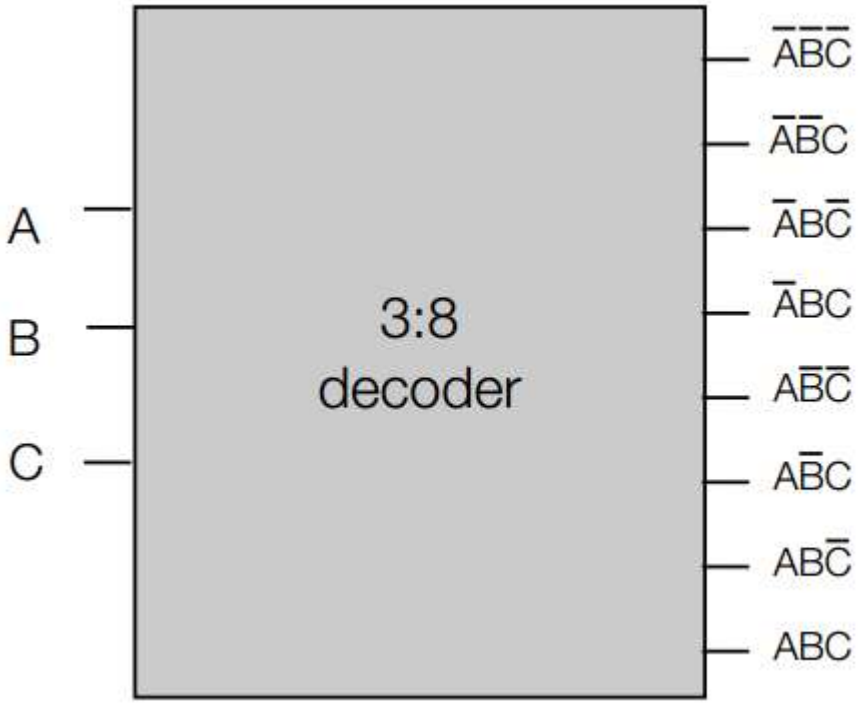
- **Note that the 2-4-line made up of 2 1-to-2-line decoders and 4 AND gates.**

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 $\neq 2^n$

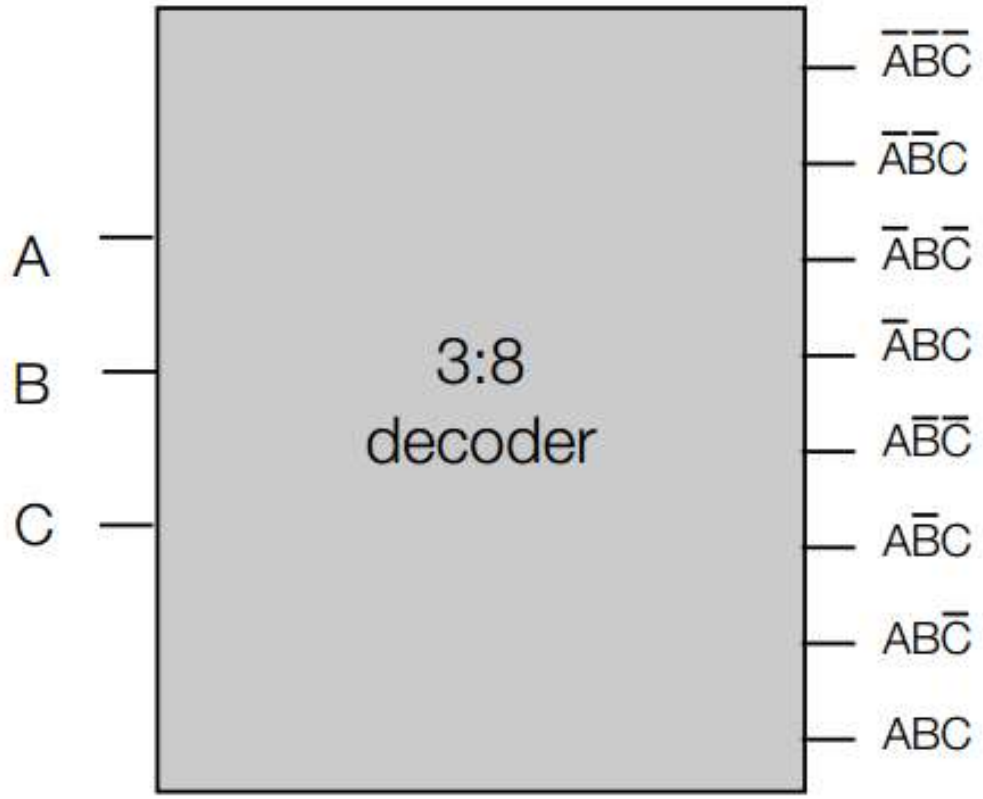
Decoder Expansion Example

- **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



Decoder Expansion Example

Converts n-bit input to m-bit output, where $n <= m <= 2^n$

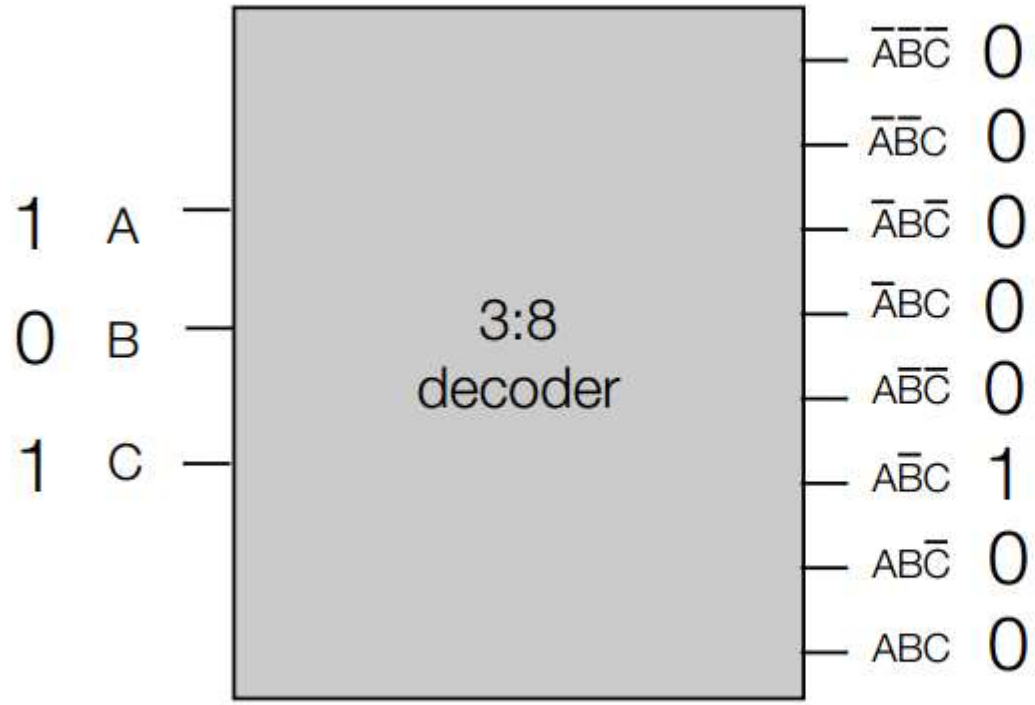


"Standard" Decoder: i^{th} output = 1, all others = 0,
where i is the binary representation of the input (ABC)

Decoder Expansion Example

Converts n-bit input to m-bit output, where $n <= m <= 2^n$

e.g., ABC = 101 (i=5)

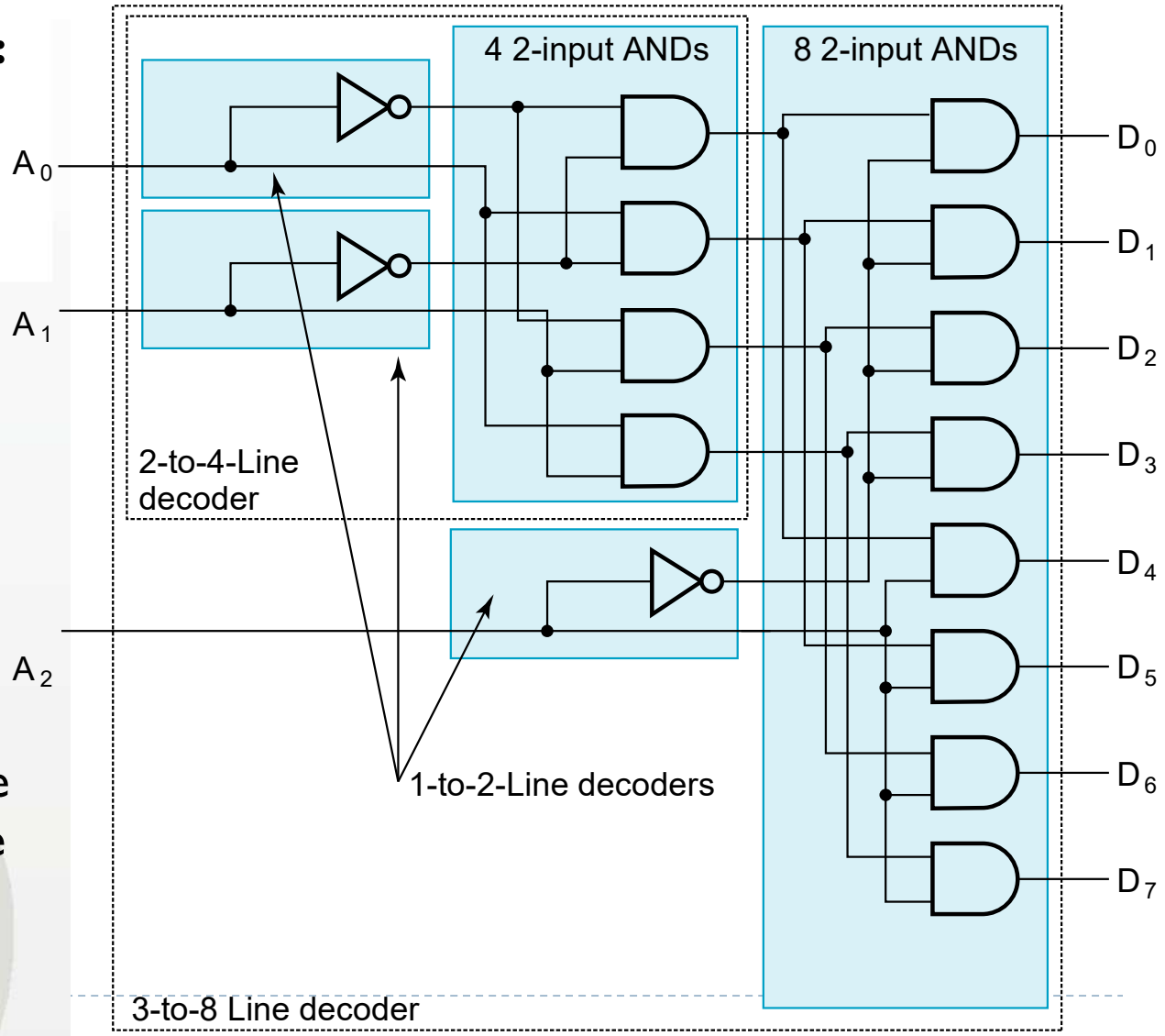


"Standard" Decoder: i^{th} output = 1, all others = 0,

where i is the binary representation of the input (ABC)

Decoder Expansion Example

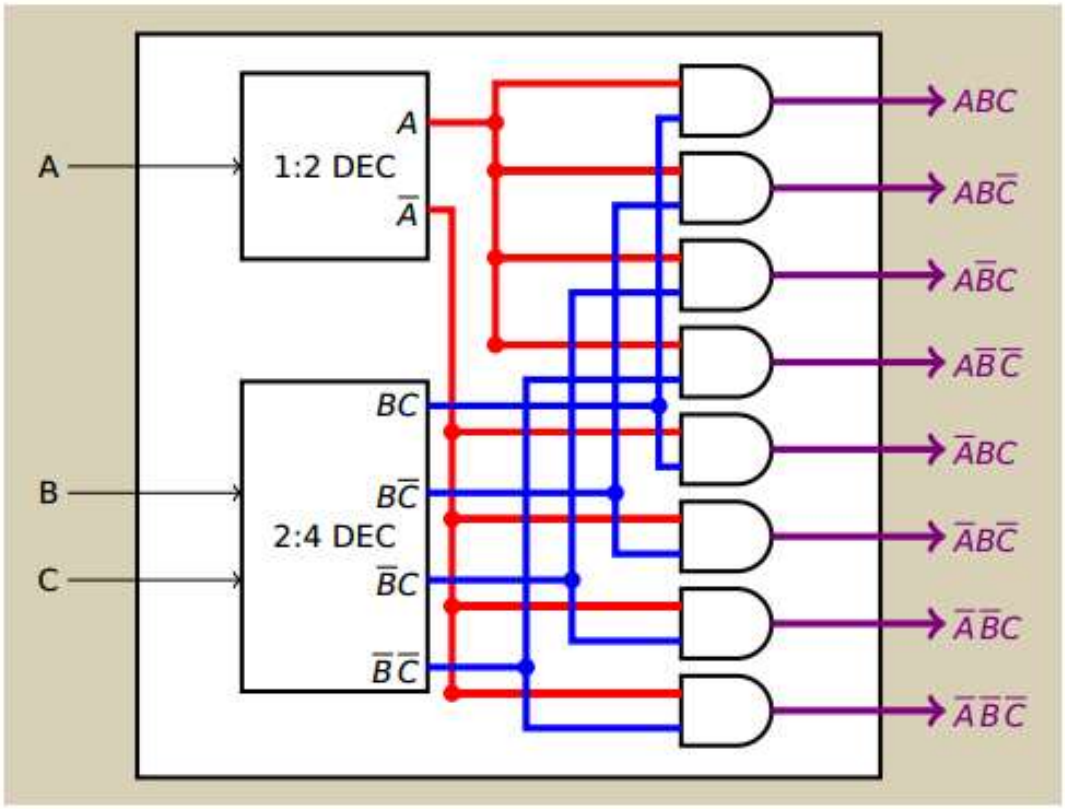
Hierarchical design: use small decoders to build bigger decoder



Note: A2 "selects" whether the 2-to-4 line decoder is active in the top half (A2=0) or the bottom (A2=1)

Decoder Expansion Example

Applying *hierarchical design* again, the 2:4 DEC helps construct a 3:8 DEC.

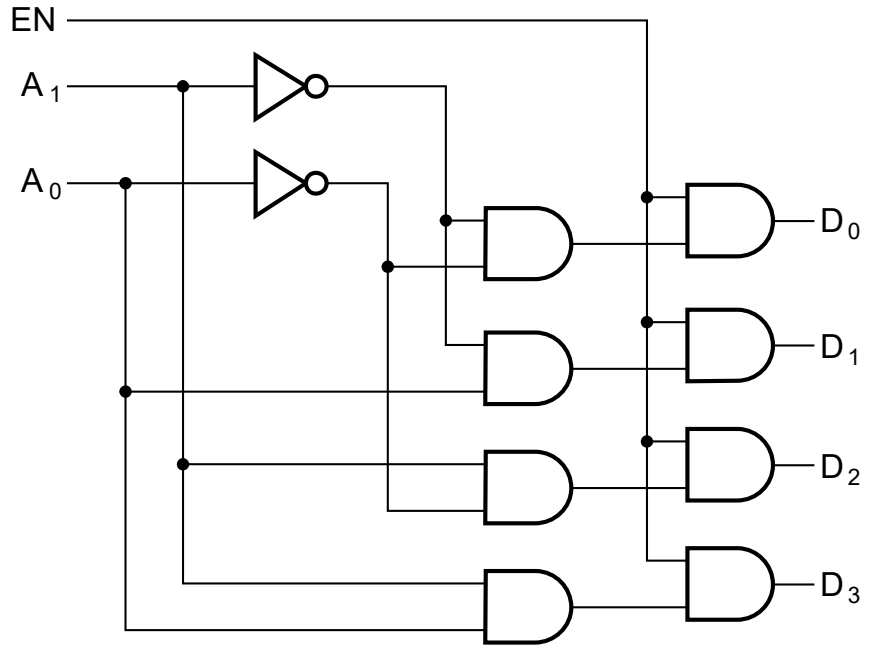


Decoder with Enable

- 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

- Find the minterms for each output function
- OR the minterms together

Combinational Logic Implementation
Binary Adder Bit Example

- Inputs: Bits X and Y being added with the
- incoming carry Z from the right
- Outputs: The sum bit S and the carry bit C

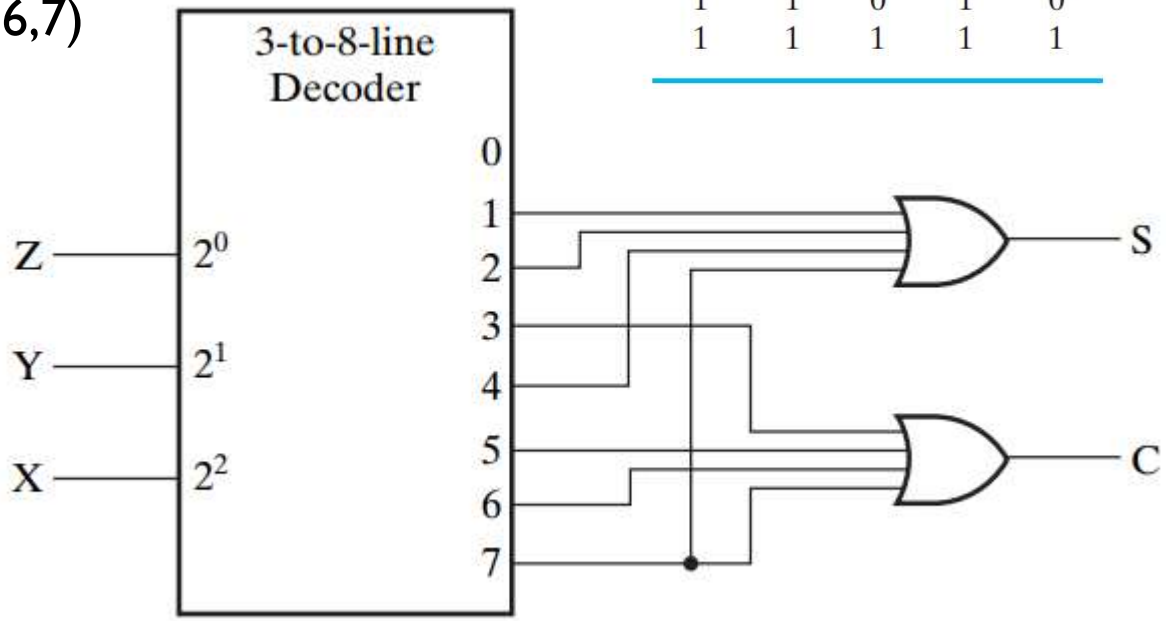
- From the truth table, we obtain the functions:

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

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

TABLE 3-6
Truth Table for 1-bit Binary Adder

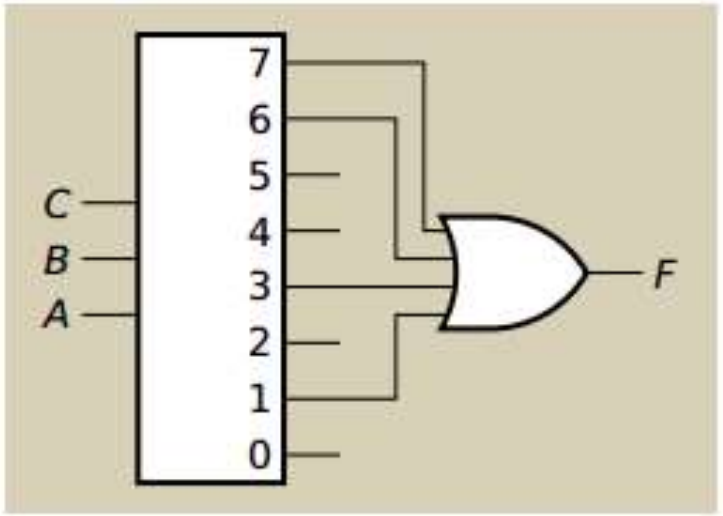
X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



Implementing a Function Using Decoder Example

E.g., $F = A\overline{C} + BC$

C	B	A	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



Warning: Easy, but not a minimal circuit.

Any Questions?