Chapter 3
Combinational Logic Design

CSCM601150

Dosen: Erdefi Rakun dan Tim Dosen PSD

Fasilkom UI

Outline:

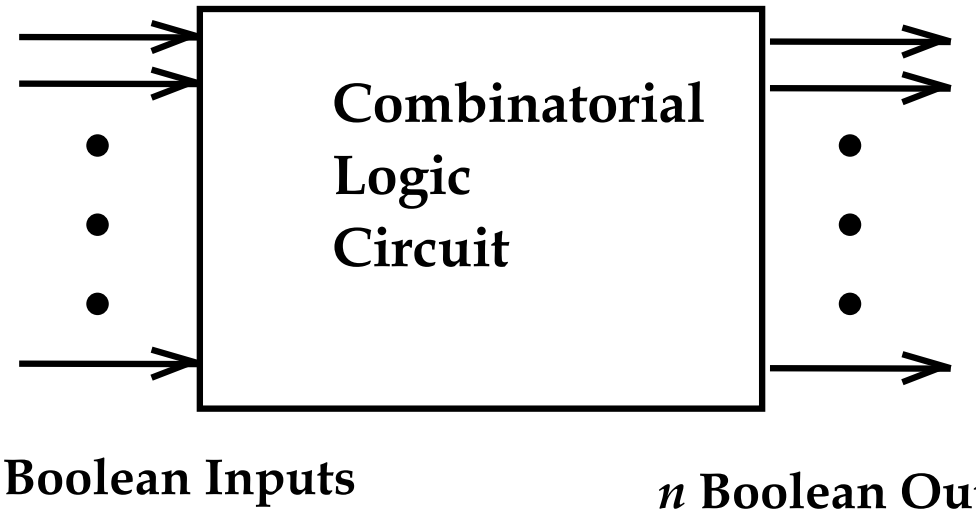
- Beginning Hierarchical Design
- Technology Mapping
- Decoding
- Encoding
- Selecting
- Binary Adder
- Binary Subtraction
- Binary Adder - Subtractors

Note: Portion of these materials are taken from Aaron Tan's slide and other portions of this material © 2008 by Pearson Education,Inc

3-1 Beginning Hierarchical Design

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

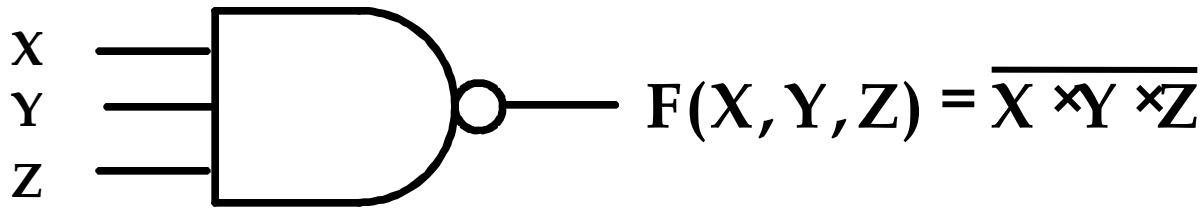
Design Procedure

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

3-2 Technology Mapping

NAND Gate

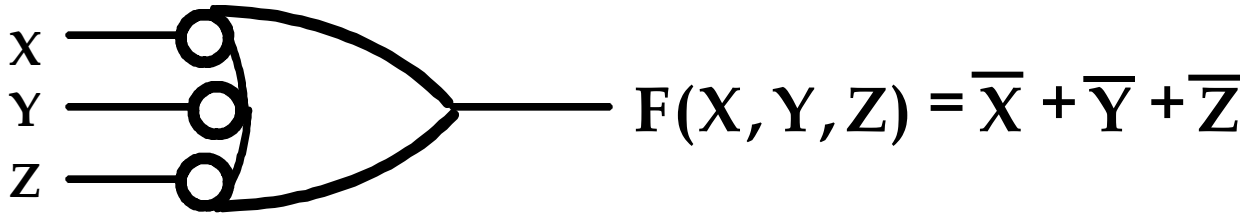
- The basic NAND gate has the following symbol, illustrated for three inputs:
  - AND-Invert (NAND)



- NAND represents NOT AND, i. e., the AND function with a NOT applied. The symbol shown is an AND-Invert. The small circle ("bubble") represents the invert function.

NAND Gates (continued)

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



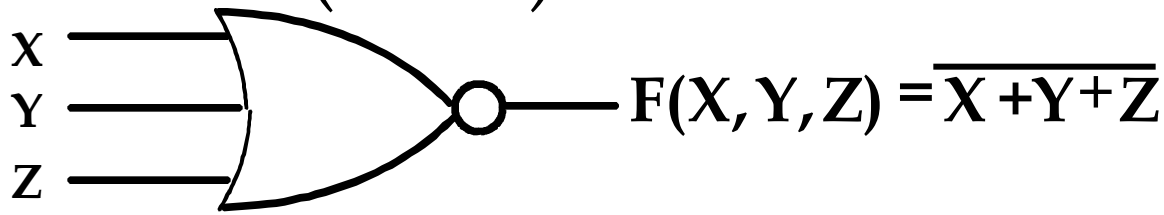
- This NAND symbol is called Invert-OR, since inputs are inverted and then ORed together.
- AND-Invert and Invert-OR both represent the NAND gate. Having both makes visualization of circuit function easier.
- A NAND gate with one input degenerates to an inverter.

NAND Gates (continued)

- 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.
- The NAND gate is a universal gate as shown in Figure 2-24 of the text.
- NAND usually does not have a operation symbol defined since
  - the NAND operation is not associative, and
  - we have difficulty dealing with non-associative mathematics!

NOR Gate

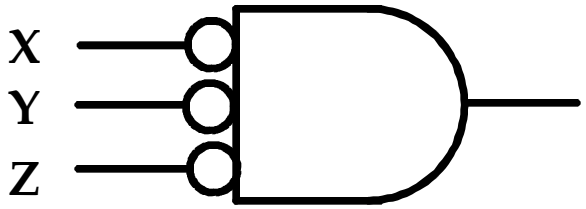
- The basic NOR gate has the following symbol, illustrated for three inputs:
  - OR-Invert (NOR)



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

NOR Gate (continued)

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



- 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. Having both makes visualization of circuit function easier.
- A NOR gate with one input degenerates to an inverter.

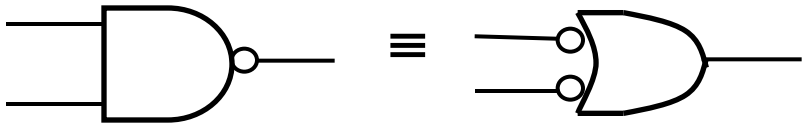
NOR Gate (continued)

- The NOR gate is a natural implementation for some technologies other than CMOS in terms of chip area and speed.
- The NOR gate is a universal gate
- NOR usually does not have a defined operation symbol since
  - the NOR operation is not associative, and
  - we have difficulty dealing with non-associative mathematics!

NAND, NOR Gates

- **NAND gate**

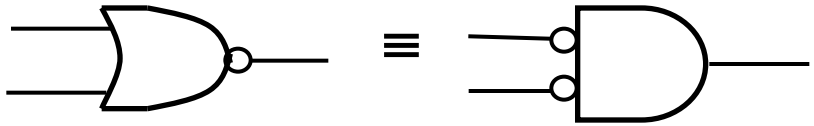
A	B	(A · B)'
0	0	
0	1	
1	0	
1	1	



NAND Negative-OR

- **NOR gate**

A	B	(A + B)'
0	0	
0	1	
1	0	
1	1	



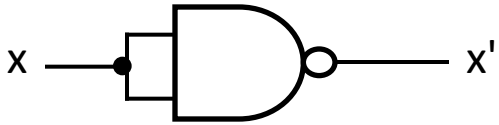
NOR Negative-AND

Universal Gates

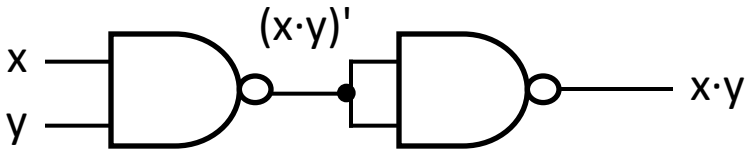
- AND/OR/NOT gates are sufficient for building any Boolean function.
- We call the set {AND, OR, NOT} a complete set of logic.
- However, other gates are also used:
  - Usefulness (eg: XOR gate for parity bit generation)
  - Economical
  - Self-sufficient (eg: NAND/NOR gates)

NAND Gates

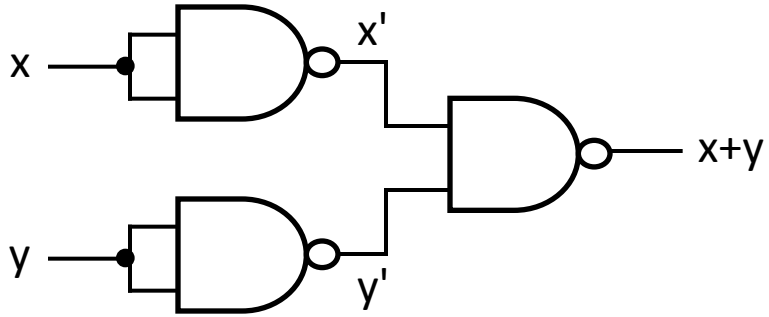
- {NAND} is a complete set of logic.
- Proof: Implement NOT/AND/OR using only NAND gates.



$(x \cdot x)' = x'$ (idempotency)



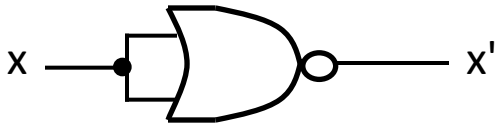
$((x \cdot y)' \cdot (x \cdot y)')' = ((x \cdot y)')'$ (idempotency)
$= x \cdot y$ (involution)



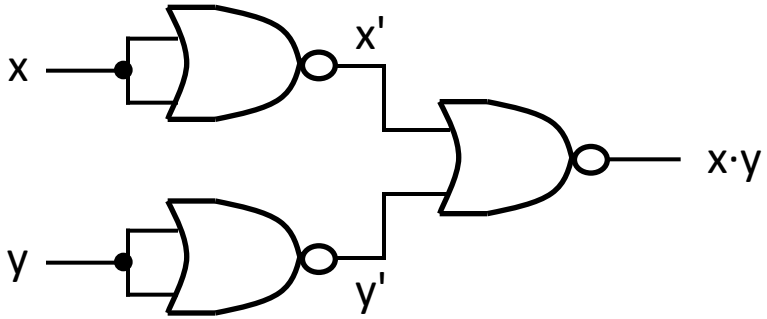
$((x \cdot x)' \cdot (y \cdot y)')' = (x' \cdot y')'$ (idempotency)
$= (x')' + (y')'$ (DeMorgan)
$= x + y$ (involution)

NOR Gates

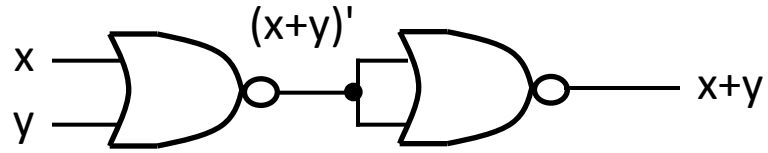
- {NOR} is a complete set of logic.
- Proof: Implement NOT/AND/OR using only NOR gates.



$(x+x)' = x'$ (idempotency)



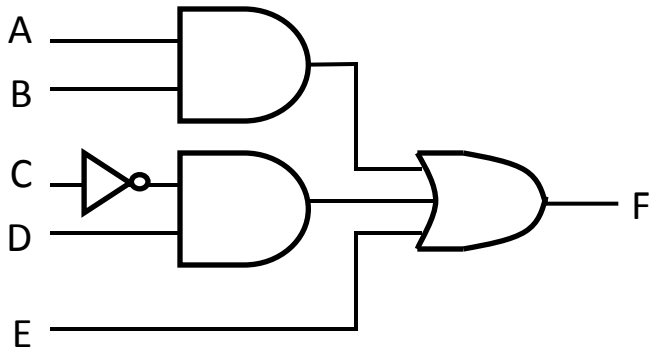
$((x+x)'+(y+y)')' = (x'+y')'$ (idempotency)
$= (x')'\cdot(y')'$ (DeMorgan)
$= x\cdot y$ (involution)



$((x+y)'+(x+y)')' = ((x+y)')'$ (idempotency)
$= x+y$ (involution)

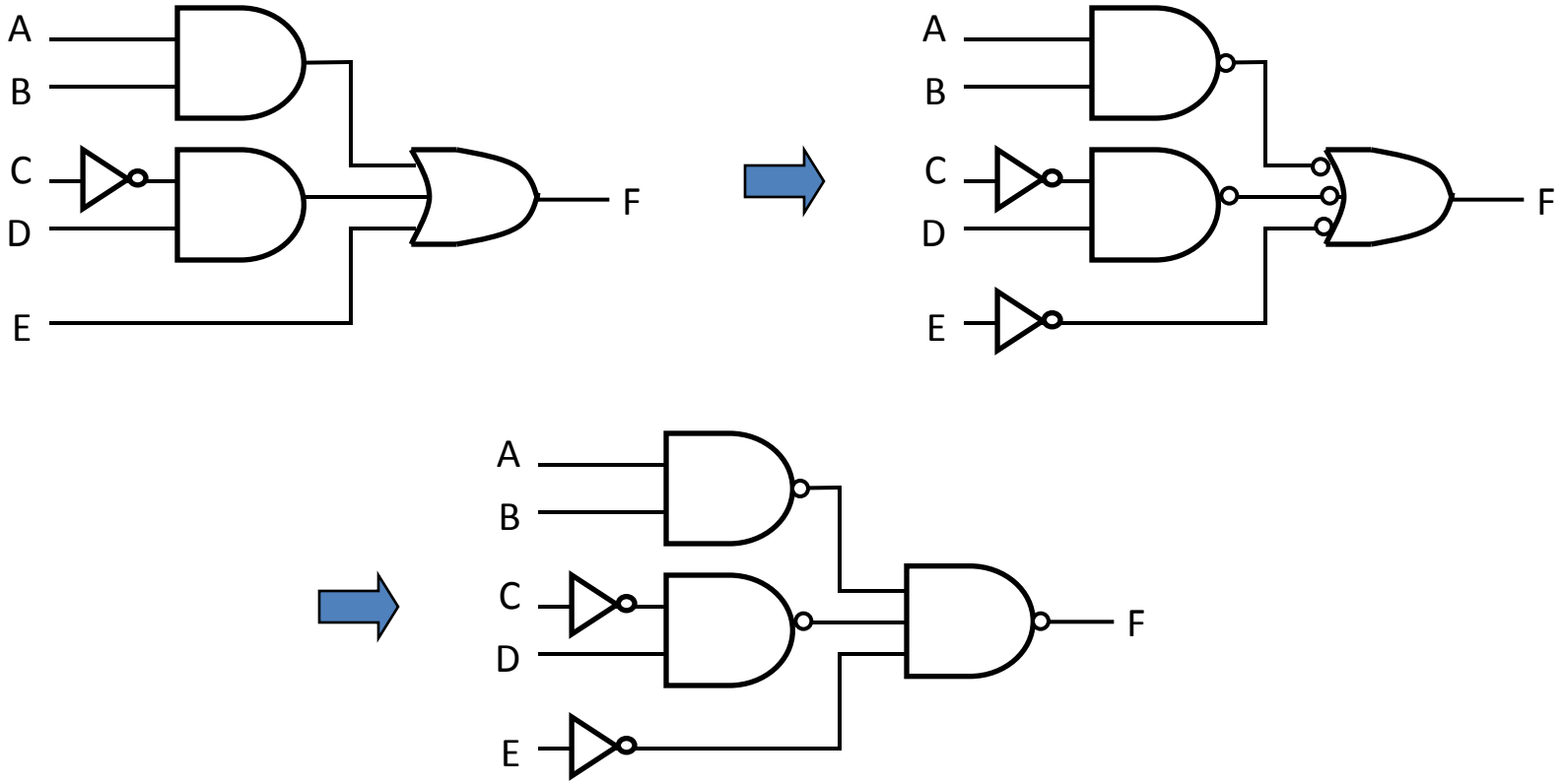
SOP and NAND Circuits (1/2)

- An SOP expression can be easily implemented using
  - 2-level AND-OR circuit
  - 2-level NAND circuit
- Example: F = A·B + C'·D + E
  - Using 2-level AND-OR circuit



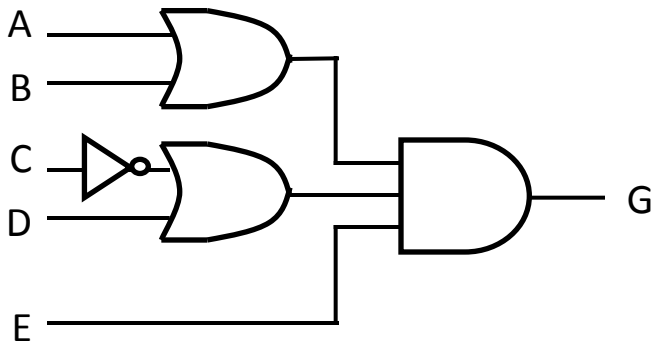
SOP and NAND Circuits (2/2)

- Example: F = A·B + C'·D + E
  - Using 2-level NAND circuit



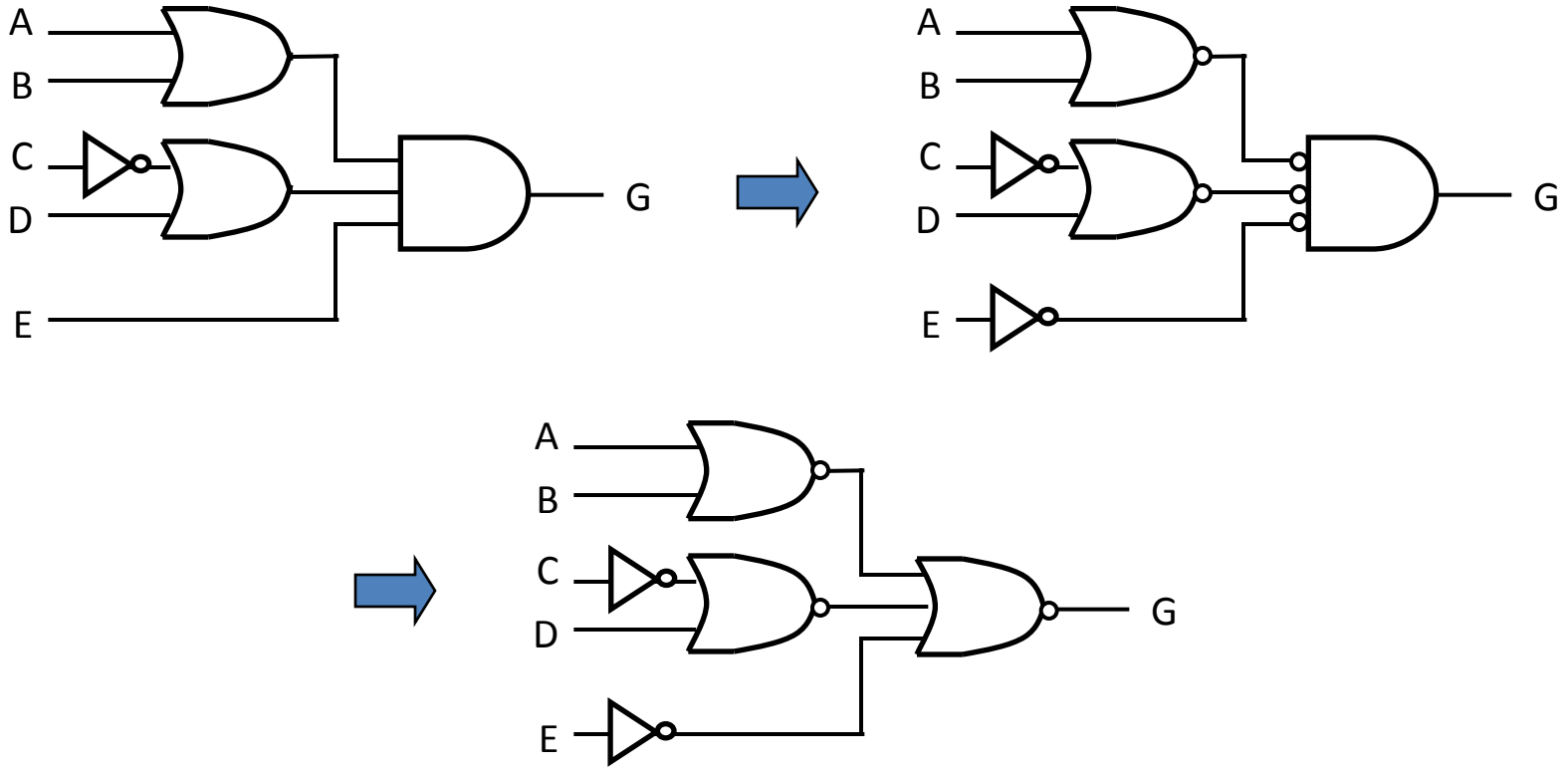
POS and NOR circuits (1/2)

- A POS expression can be easily implemented using
  - 2-level OR-AND circuit
  - 2-level NOR circuit
- Example: $G = (A+B) \cdot (C'+D) \cdot E$
  - Using 2-level OR-AND circuit



POS and NOR circuits (2/2)

- Example: $G = (A+B) \cdot (C'+D) \cdot E$
  - Using 2-level NOR circuit



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
     - NAND gates (including inverters)

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				Output Excess-3			
A	B	C	D	W	X	Y	Z
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
1	0	1	0	X	X	X	X
1	0	1	1	X	X	X	X
1	1	0	0	X	X	X	X
1	1	0	1	X	X	X	X
1	1	1	0	X	X	X	X
1	1	1	1	X	X	X	X

Design Example (continued)

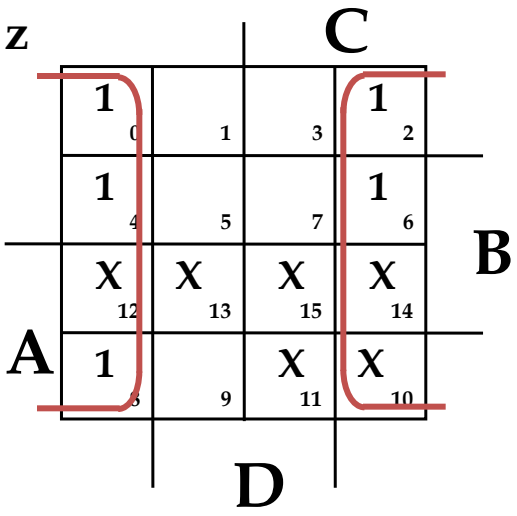
3. Optimization
   a. 2-level using K-maps

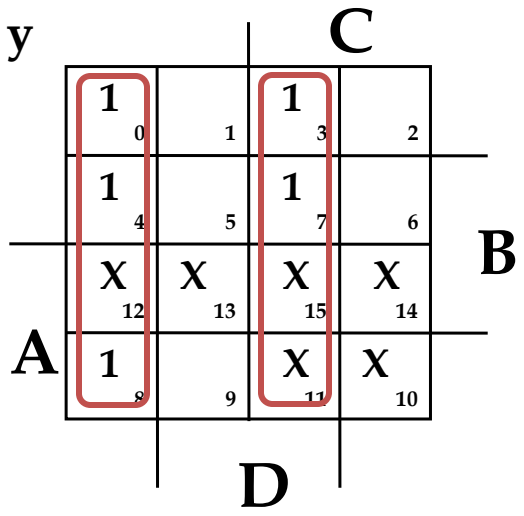
$W = A + BC + BD$

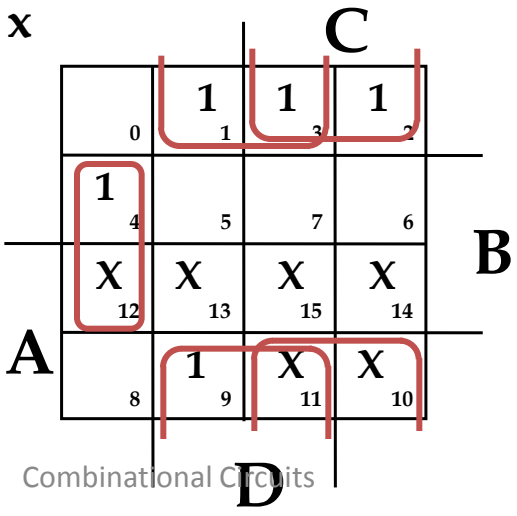
$X = \overline{B}C + \overline{B}D + B\overline{C}\,\overline{D}$

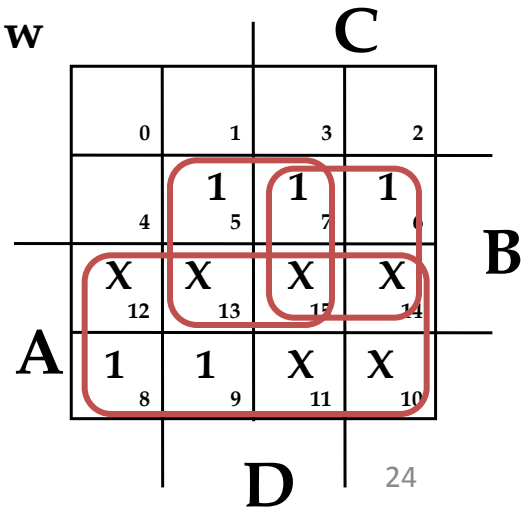
$Y = CD + \overline{C}\,\overline{D}$

$Z = \overline{D}$









Design Example (continued)

3. Optimization (continued)
   b. Multiple-level using transformations

$W = A + BC + BD$

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

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

$Z = \overline{D}$

$G = 7 + 10 + 6 + 0 = 23$

   – Perform extraction, finding factor:

$T_1 = C + D$

$W = A + BT_1$

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

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

$Z = \overline{D}$

$G = 2 + 4 + 7 + 6 + 0 = 19$

Design Example (continued)

3. Optimization (continued)

   b. Multiple-level using transformations

   $T_1 = C + D$

   $W = A + BT_1$

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

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

   $Z = \overline{D}$ $\qquad$ $G = 19$

   – An additional extraction not shown in the text since it uses a Boolean transformation: ($\overline{C}\,\overline{D} = \overline{C + D} = \overline{T_1}$):

   $W = A + BT_1$

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

   $Y = CD + \overline{T_1}$

   $Z = \overline{D}$ $\qquad$ $G = 2 + 4 + 6 + 4 + 0 = 16!$

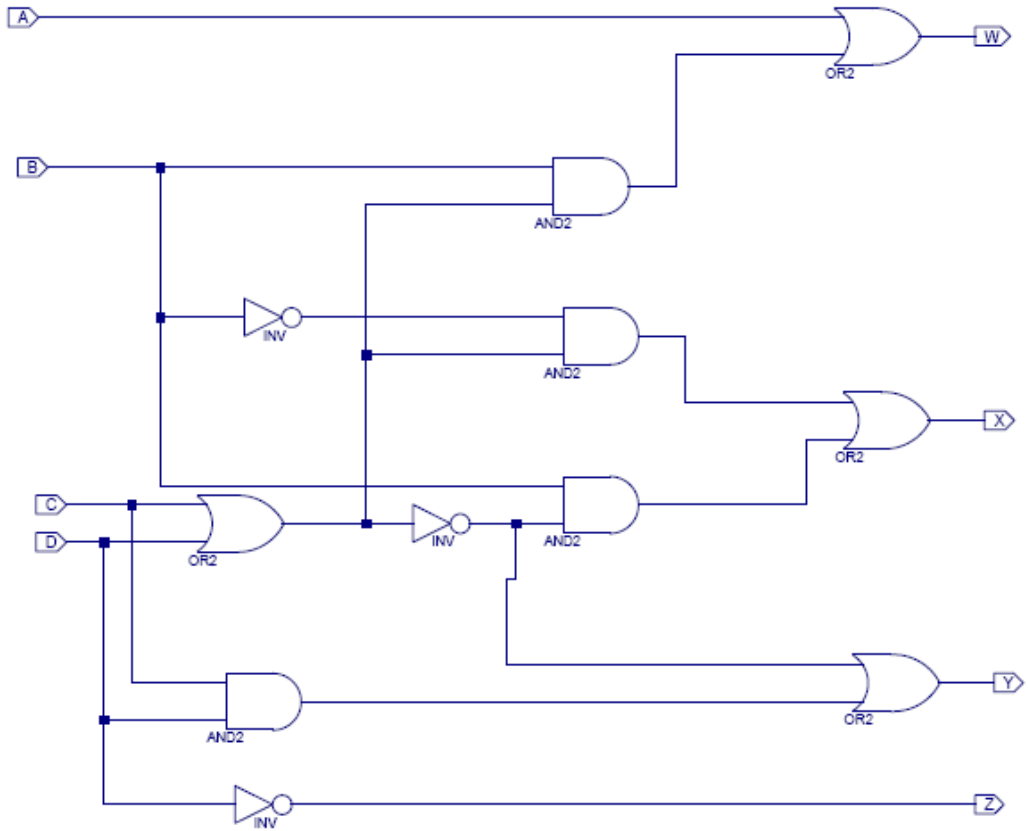
Design Example (continued)

4. Mapping Procedures
   - To NAND gates
   - To NOR gates
   - Mapping to multiple types of logic blocks is covered in the reading supplement: Advanced Technology Mapping.

Design Example (continued)

Technology Mapping

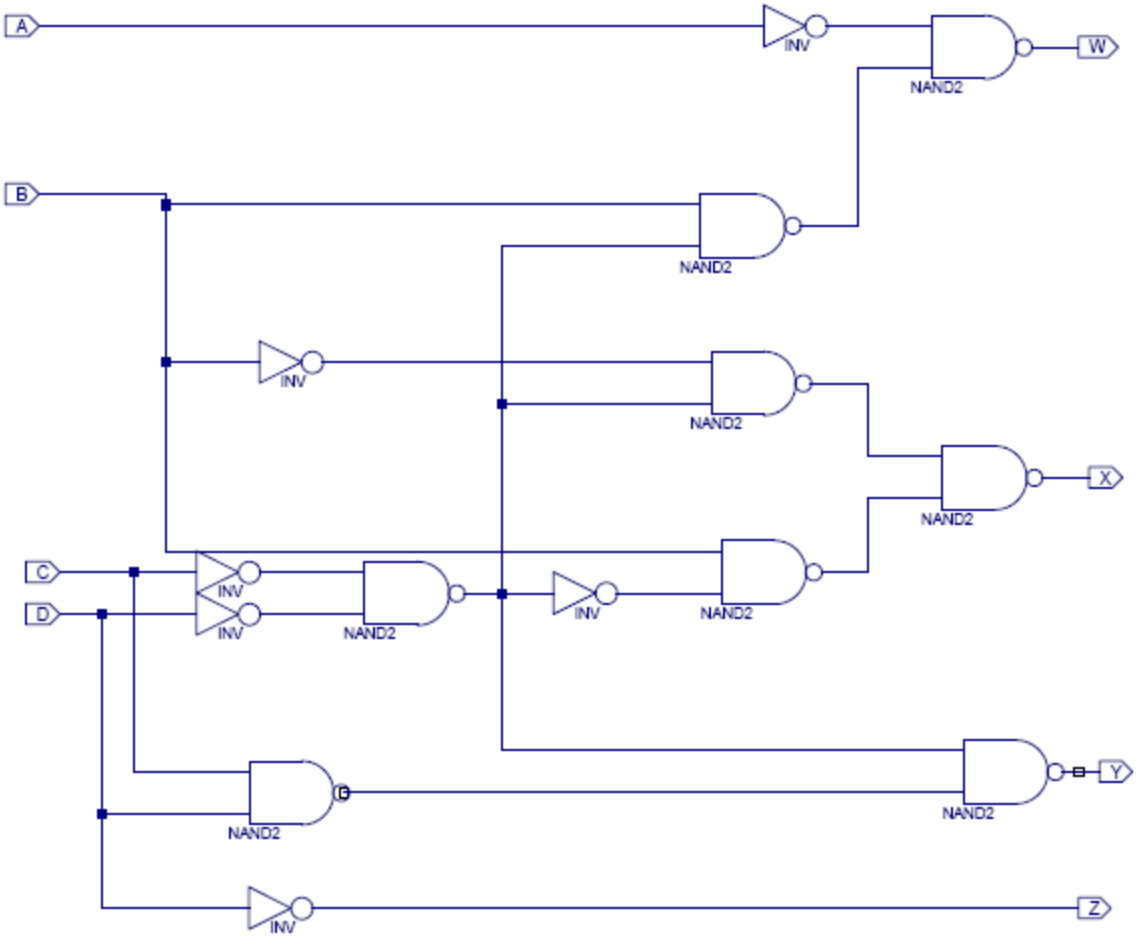
Mapping with a library containing AND, OR, NOT



Design Example (continued)

Technology Mapping

Mapping with a library containing inverters and 2-input NAND



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

- BCD-to-Excess 3 Code Converter
  - Find the SOP Boolean equations from the final circuit.
  - Find the truth table from these equations
  - Compare to the formulation truth table
- Finding the Boolean Equations:

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

$W = A + (T_1 B) = A + B T_1$

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

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

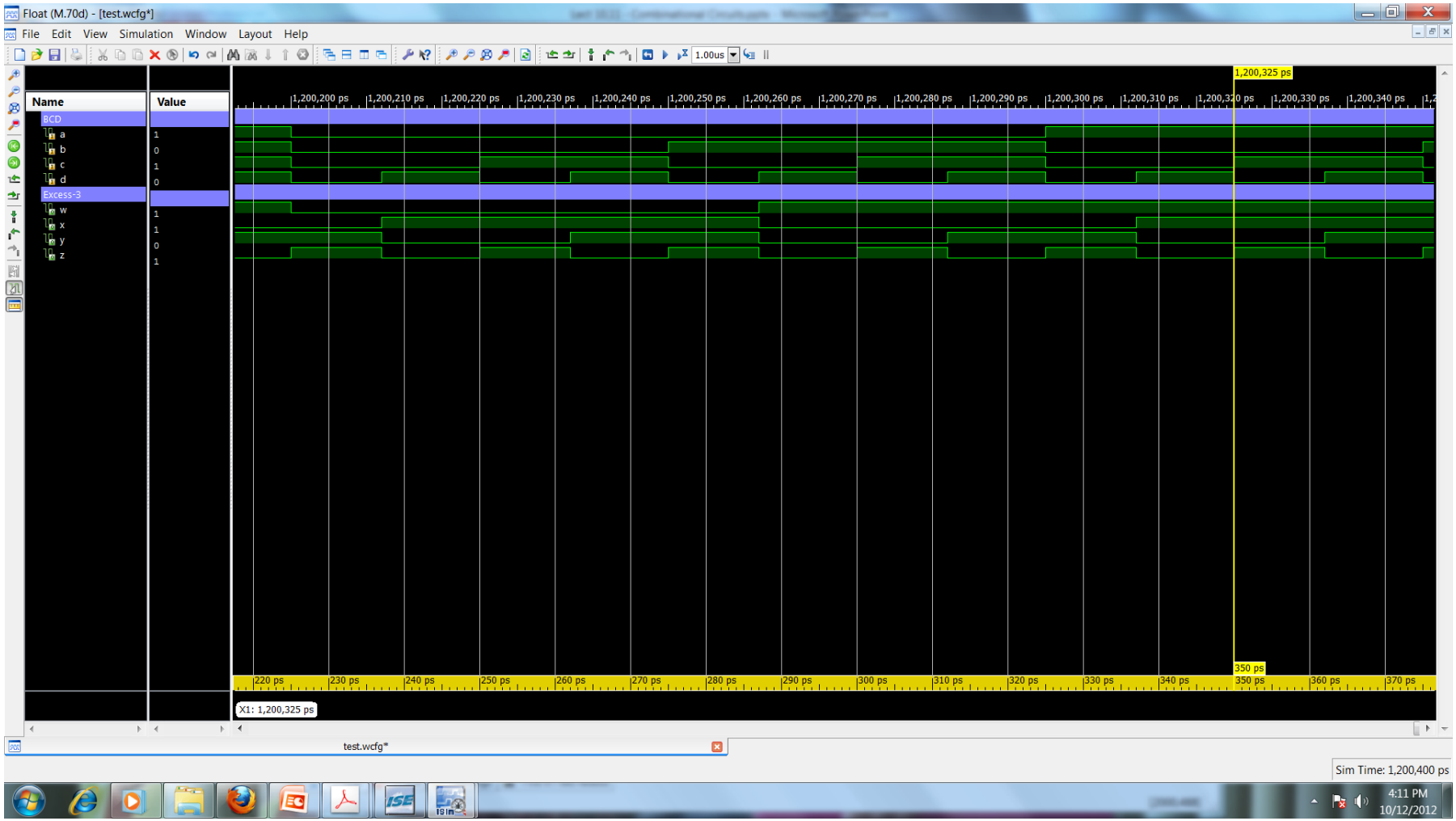
Verification Example: Manual Analysis

- 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 1 0 0

The tables match!

Verification Example: Simulation Analysis



3-3 Functions and Functional Blocks

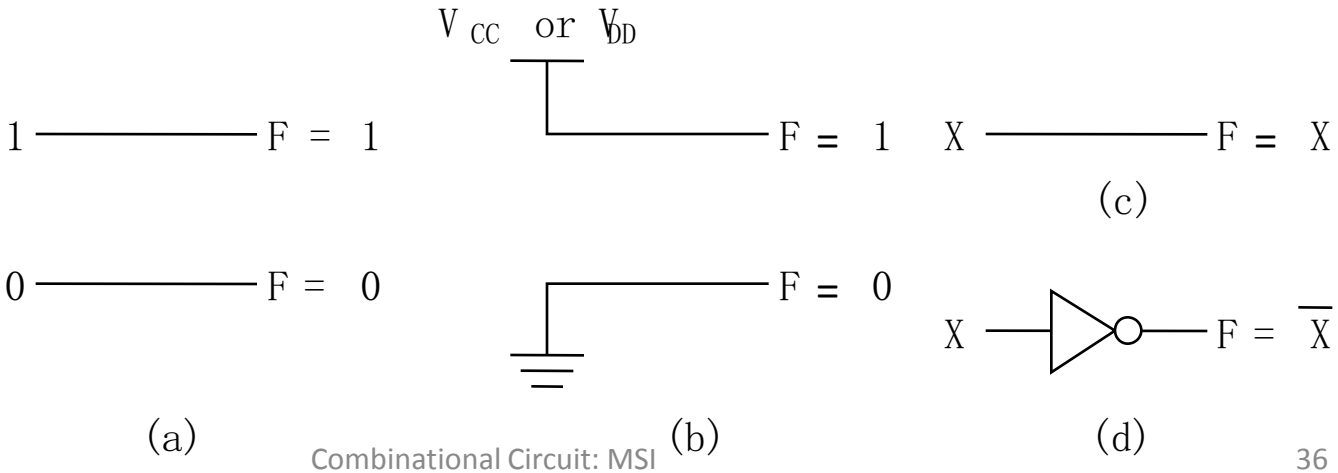
- The functions considered are those found to be very useful in design
- Corresponding to each of the functions is a combinational circuit implementation called a *functional block*.
- In the past, functional blocks were packaged as small-scale-integrated (SSI), medium-scale integrated (MSI), and large-scale-integrated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

3-4 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

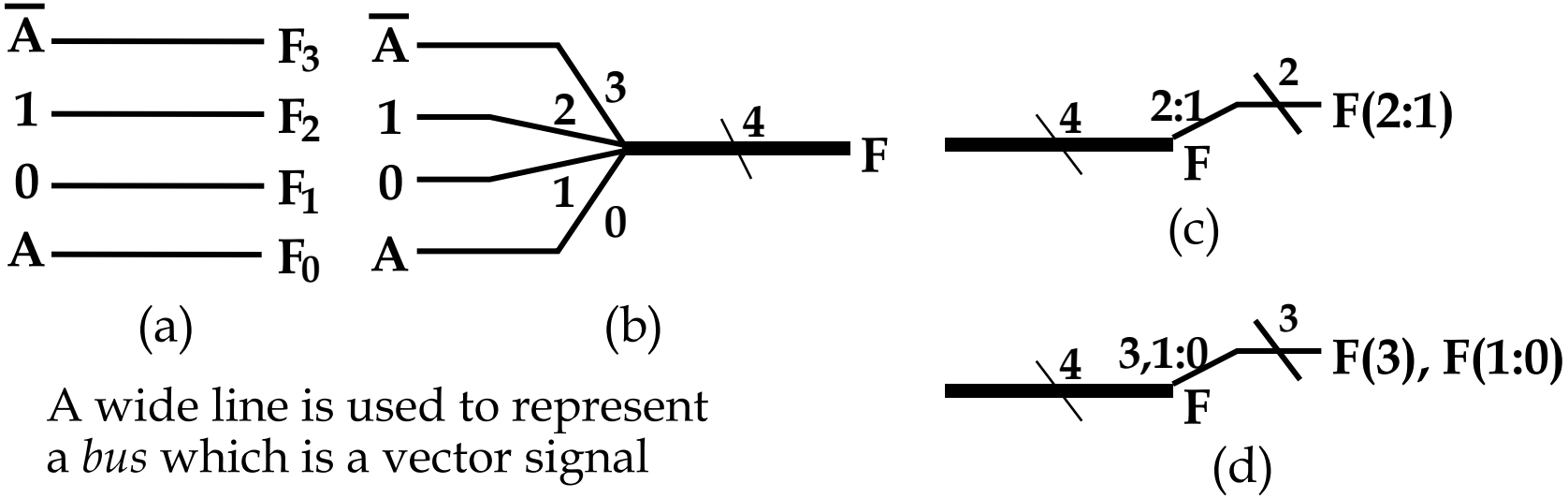
TABLE 4-1
Functions of One Variable

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



Multiple-bit Rudimentary Functions

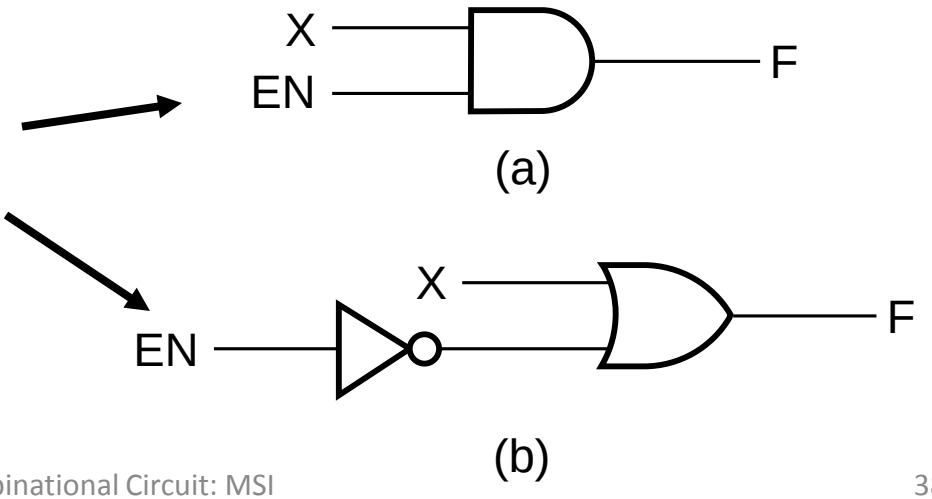
- Multi-bit Examples:



- 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.

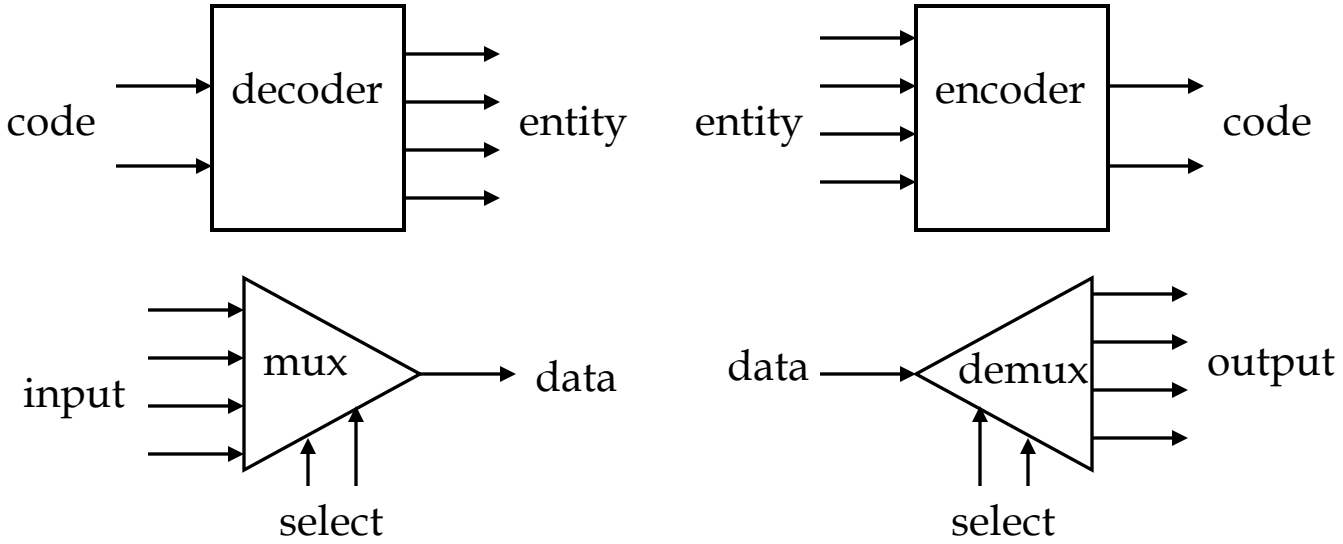
Enabling Function

- *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



Introduction

- Four common and useful MSI circuits:
  - Decoder
  - Demultiplexer
  - Encoder
  - Multiplexer
- Block-level outlines of MSI circuits:



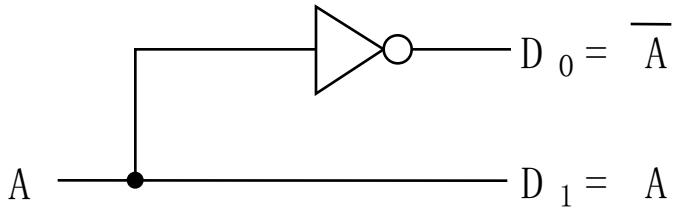
3-5 Decoding

- 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

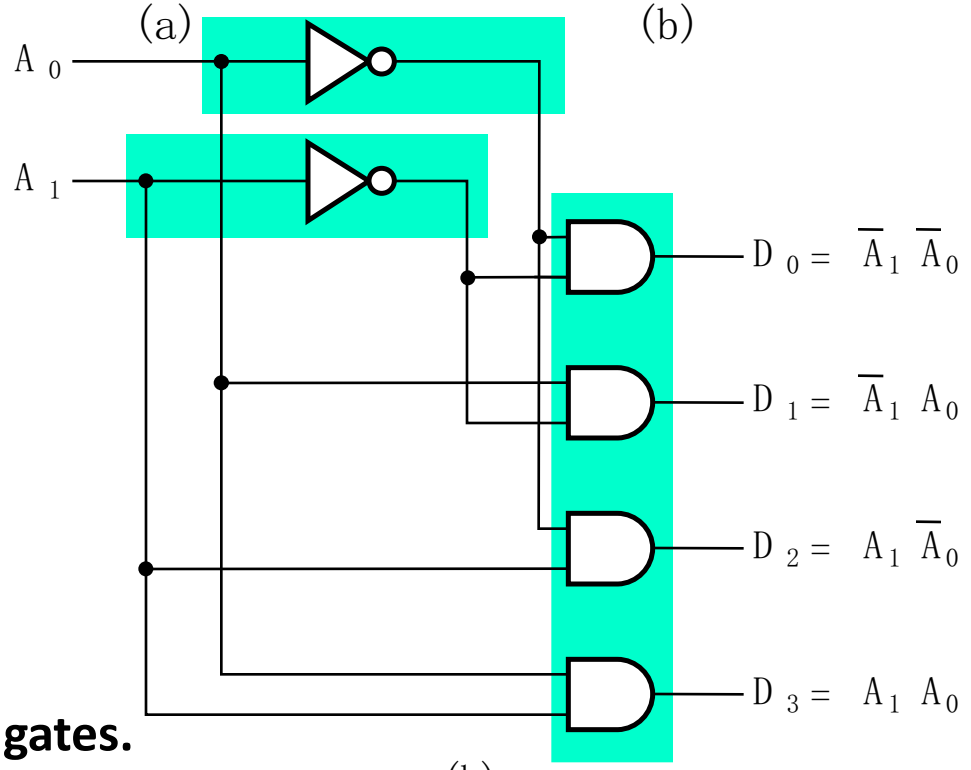


(a) (b)

- 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)



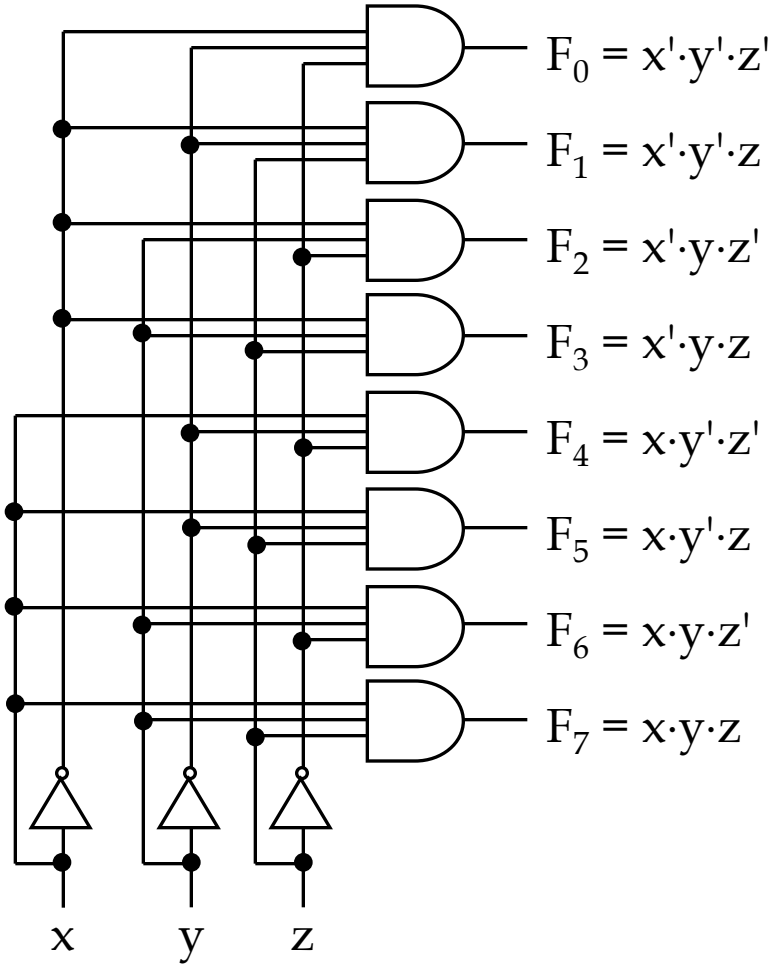
(b)

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

Decoders Examples

- Design a 3×8 decoder.

x	y	z	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_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 Expansion

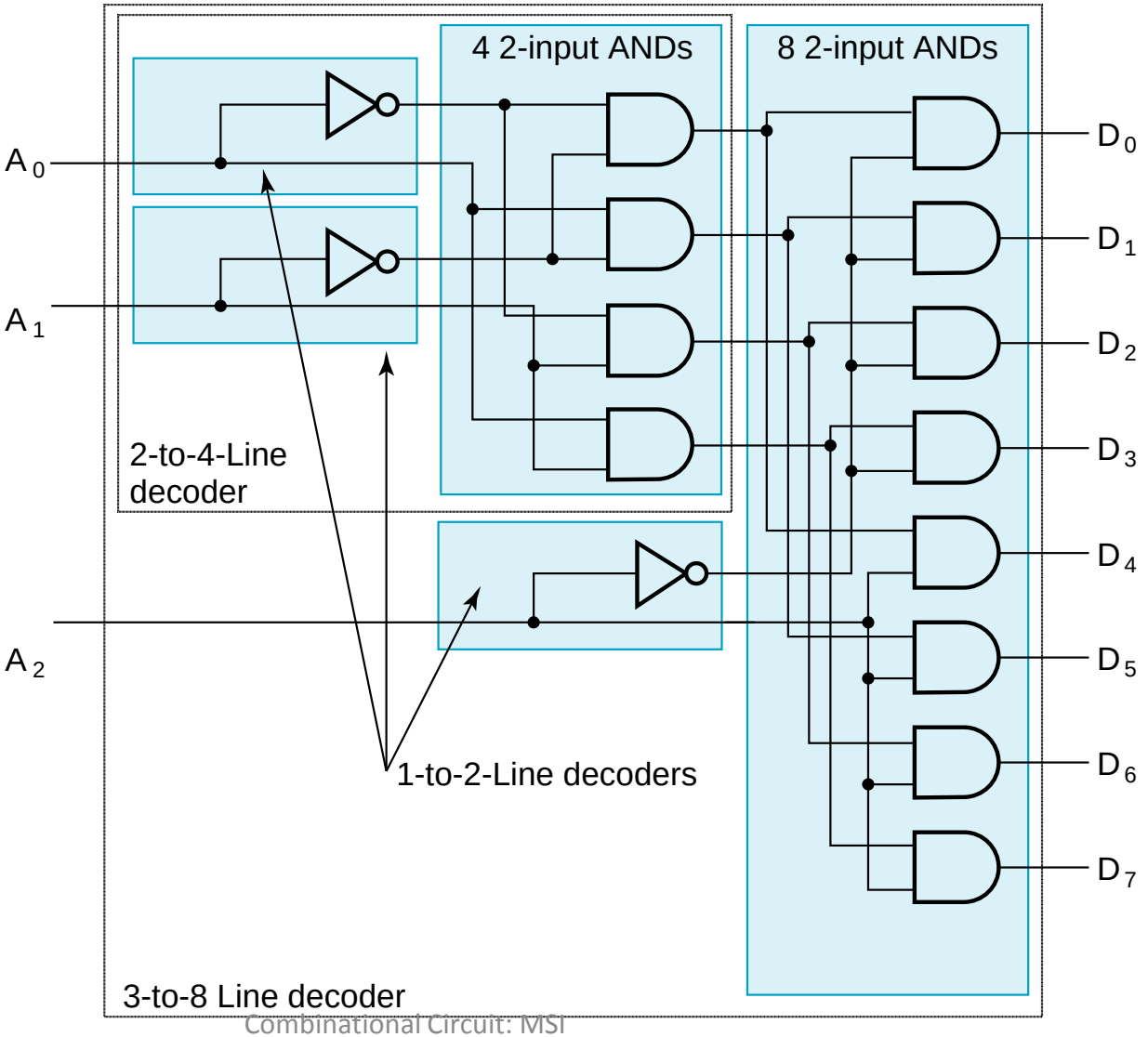
- General procedure given in book for any decoder with n inputs and 2^n outputs.
- This procedure builds a decoder backward from the outputs.
- The output AND gates are driven by two decoders with their numbers of inputs either equal or differing by 1.
- These decoders are then designed using the same 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

- 6-to-64-line decoder
  - Number of output ANDs = 64
  - Number of inputs to decoders driving output ANDs = 6
  - Gate cost = 6 + (6 x 64) = 390
  - Closest possible split to equal
    - 2 3-to-8-line decoder
  - 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
  - New Gate cost =

    6 + 2 (2 x 4) + 2 (2 x 8) + 2 x 64 = 182

Decoder Expansion - Example 2

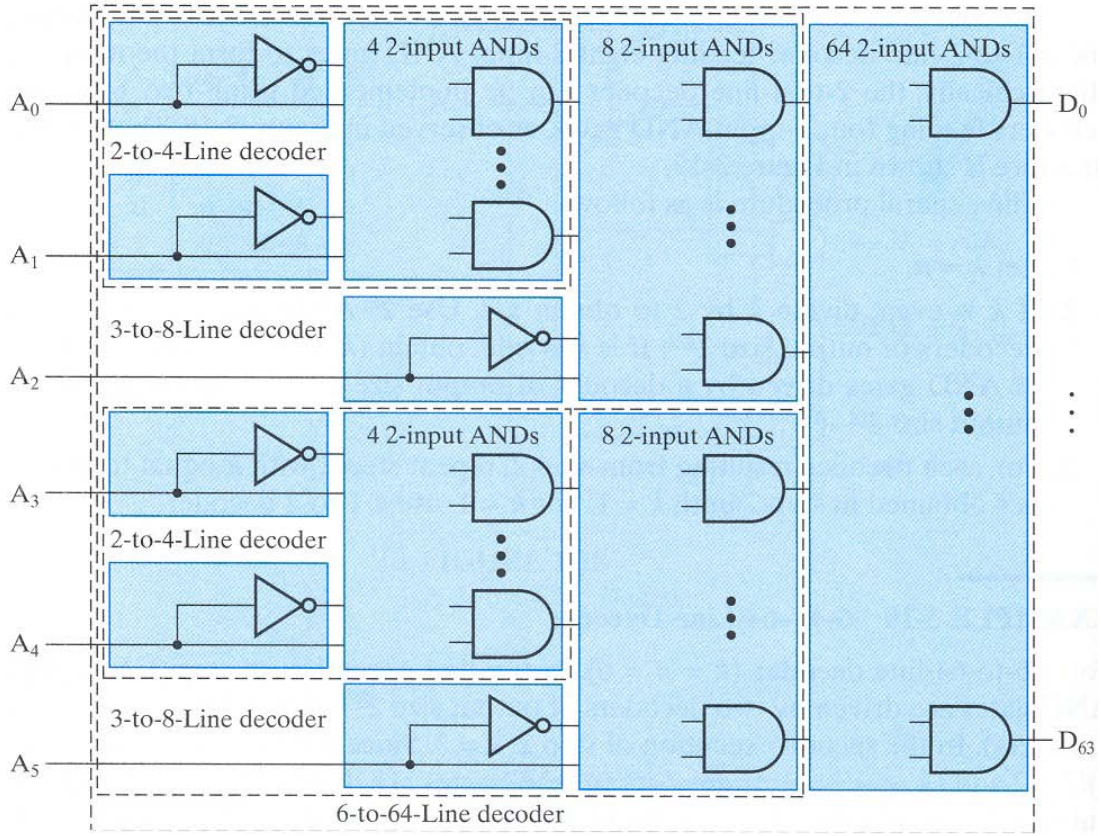


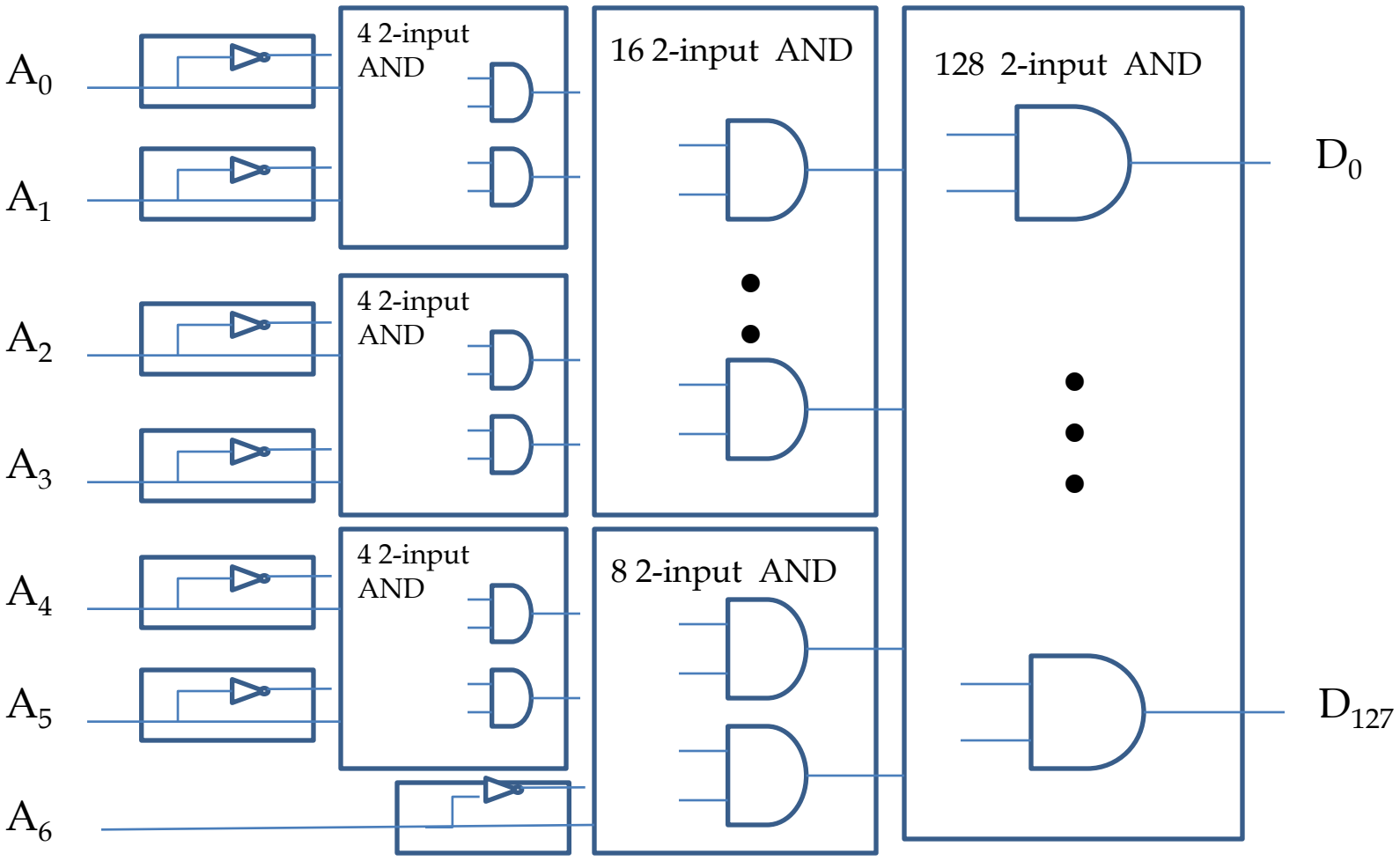
FIGURE 3-20
A 6-to-64-Line Decoder

Decoder Expansion - Example 3

- 7-to-128-line decoder
  - Number of output ANDs = 128
  - Number of inputs to decoders driving output ANDs = 7
  - Gate cost = 7 + 7 x 128 = 903
  - 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 = 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 decoders driving output ANDs = 3
    - Closest possible split to equal
      - 2-to-4-line decoder
      - 1-to-2-line decoder
  - New Gate cost =

    = 7 + 3 (2 x 4) + (2 x 8) + (2 x 16) + 2 x 128 = 335

Decoder Expansion - Example 3

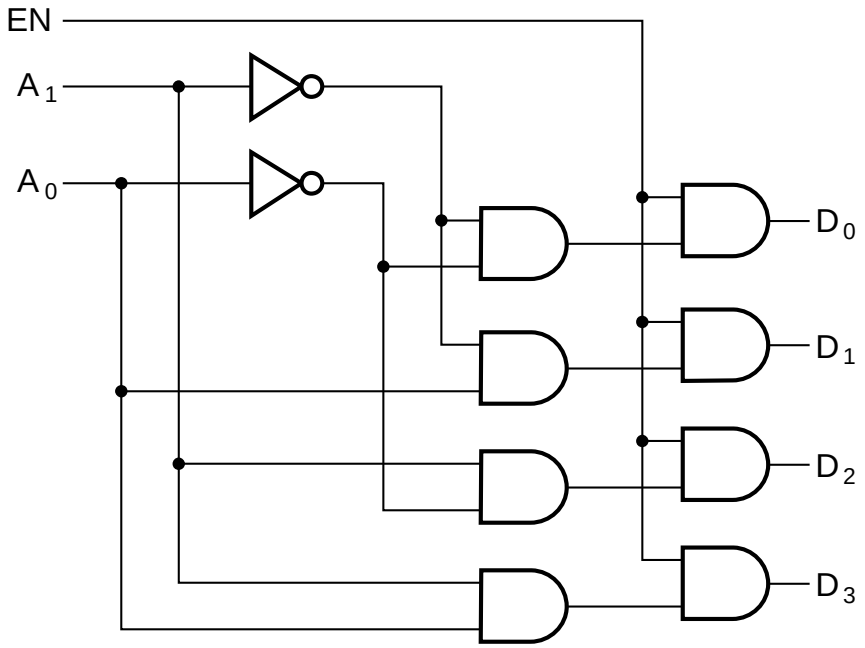


Decoder with Enable

- 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

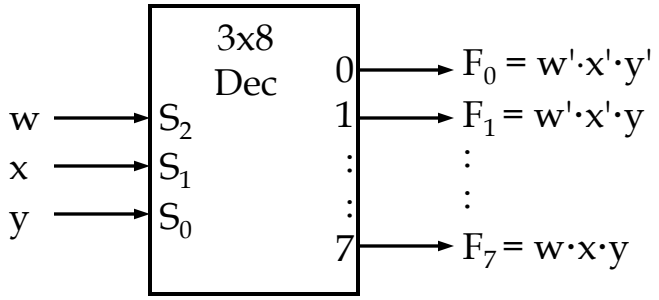
(a)

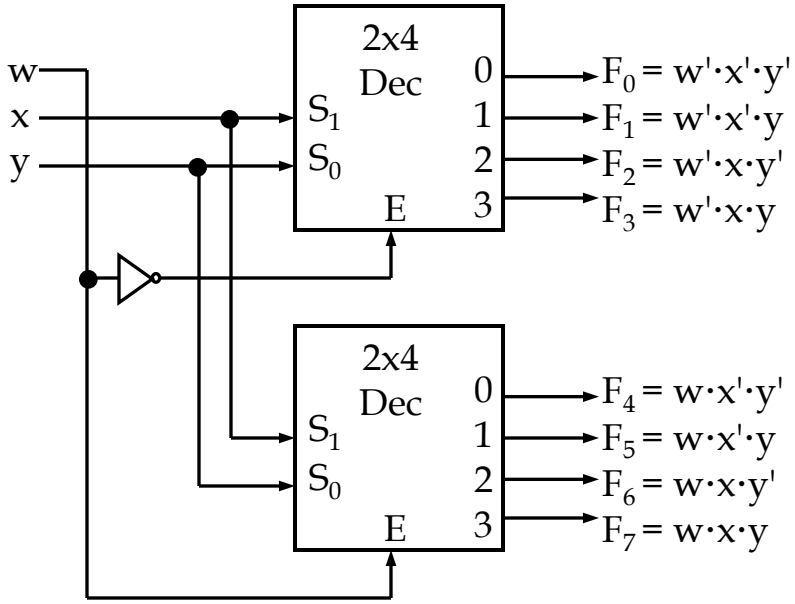


(b)

Larger Decoders (1/3)

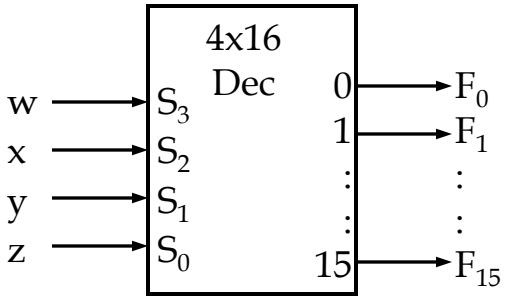
- Larger decoders can be constructed from smaller ones.
- Example: A 3×8 decoder can be built from two 2×4 decoders (with one-enable) and an inverter.

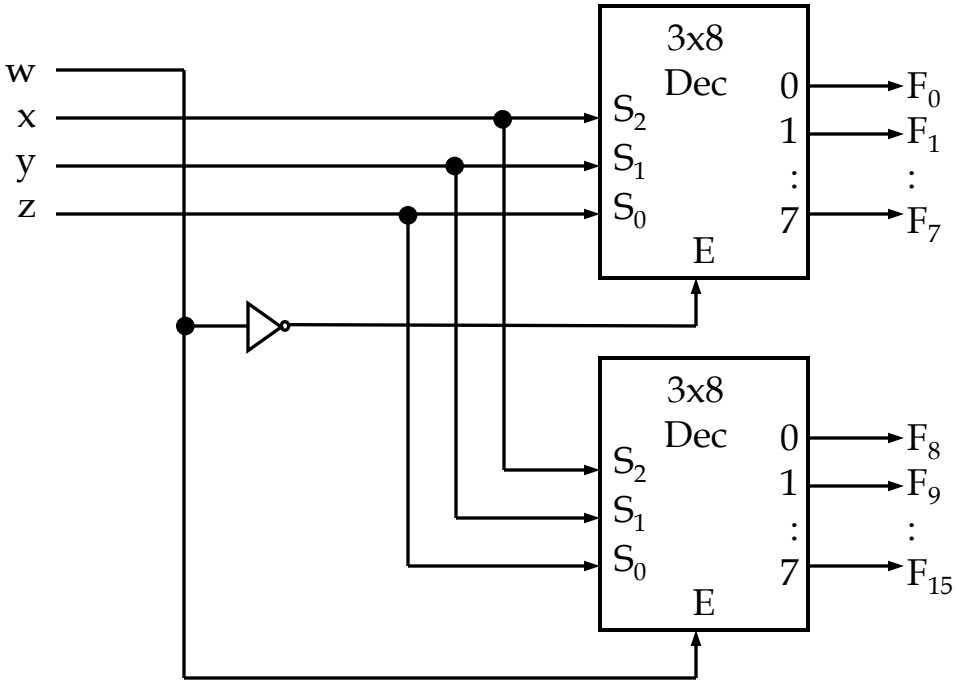




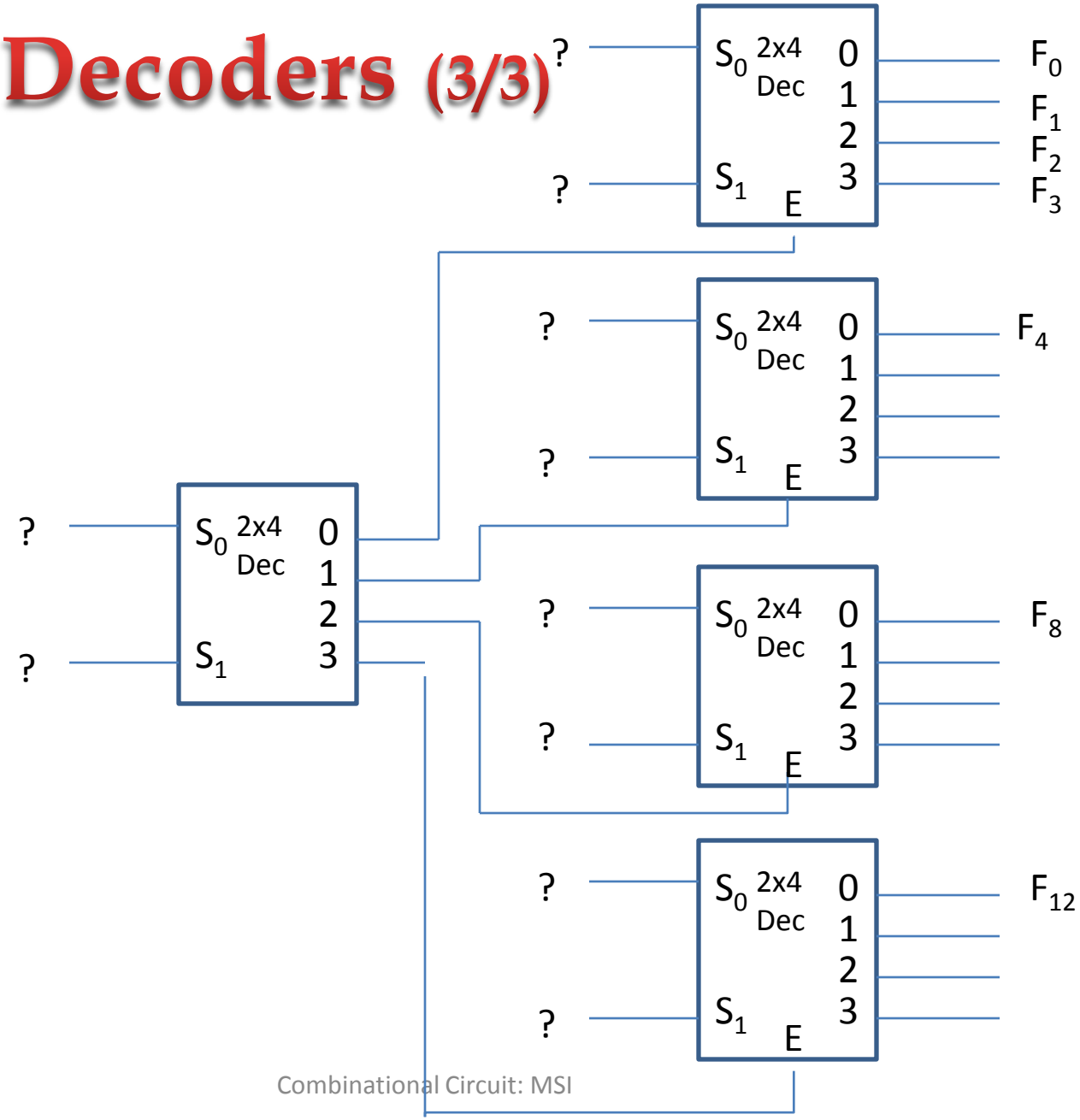
Larger Decoders (2/3)

- Construct a 4×16 decoder from two 3×8 decoders with one-enable.





Larger Decoders (3/3)



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

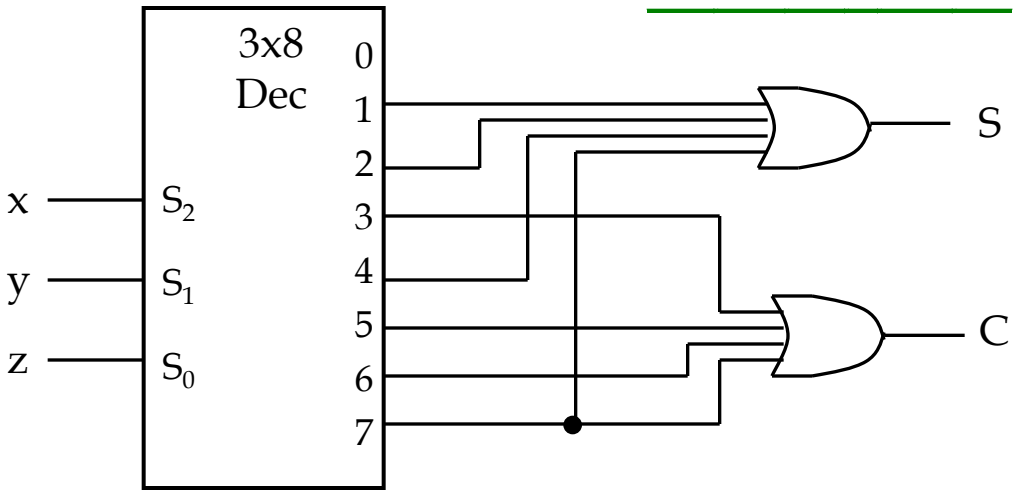
Decoders: Implementing Functions

- Example: Full adder

  $S(x, y, z) = \sum m(1,2,4,7)$

  $C(x, y, z) = \sum m(3,5,6,7)$

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

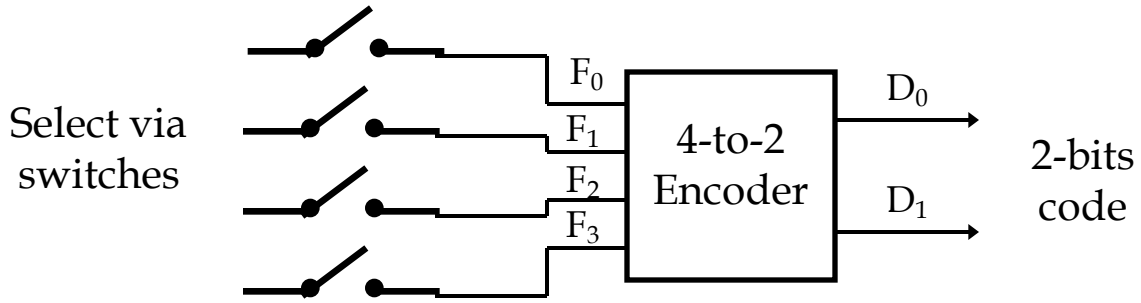


3-6 Encoding

- 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 corresponding to the position in which the 1 appears.

Encoders (1/4)

- Encoding is the converse of decoding.
- Given a set of input lines, of which exactly one is high, the encoder provides a code that corresponds to that input line.
- Contains 2^n (or fewer) input lines and n output lines.
- Implemented with OR gates.
- Example:



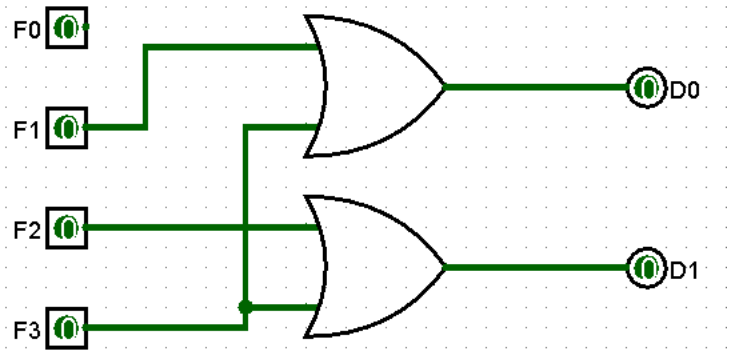
Encoders (2/4)

- Truth table:

- With K-map, we obtain:

  $D_0 = F1 + F3$
  $D_1 = F2 + F3$

- Circuit:



F_0	F_1	F_2	F_3	D_1	D_0
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1
0	0	0	0	X	X
0	0	1	1	X	X
0	1	0	1	X	X
0	1	1	0	X	X
0	1	1	1	X	X
1	0	0	1	X	X
1	0	1	0	X	X
1	0	1	1	X	X
1	1	0	0	X	X
1	1	0	1	X	X
1	1	1	0	X	X
1	1	1	1	X	X

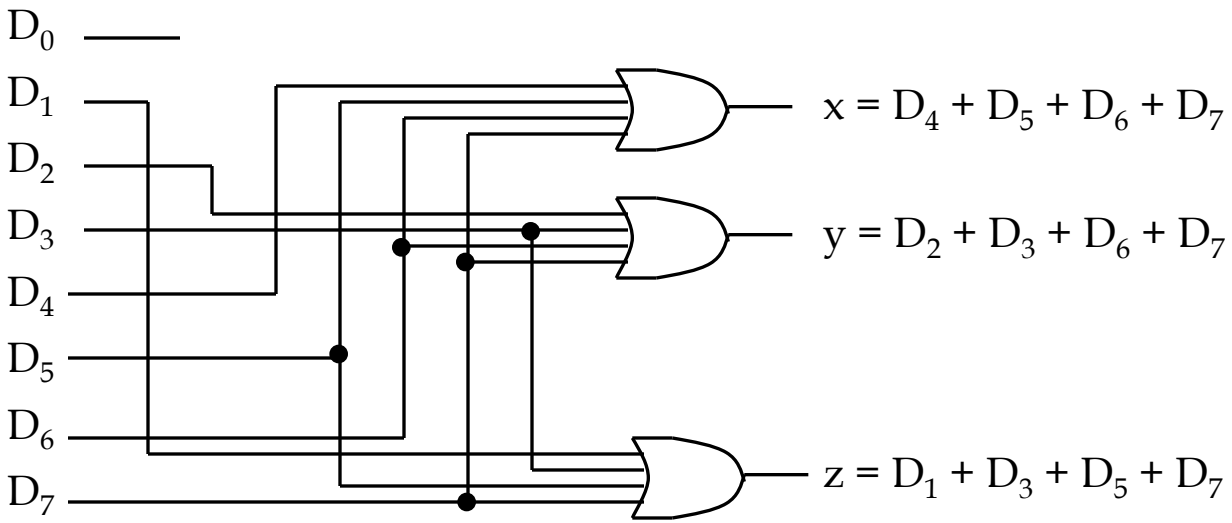
Encoders (3/4)

- Example: Octal-to-binary encoder.
  - At any one time, only one input line has a value of 1.
  - Otherwise, we need priority encoder.

Inputs								Outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

Encoders (4/4)

- Example: Octal-to-binary encoder.



An 8-to-3 encoder

- Exercise: Can you design a 2^n-to-n encoder without using K-map?

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:

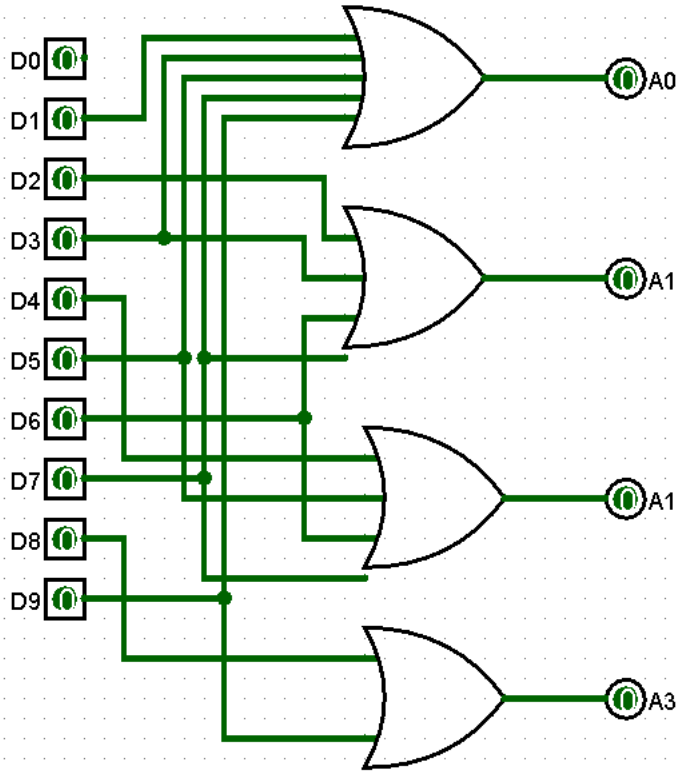
  $A_3 = D_8 + D_9$

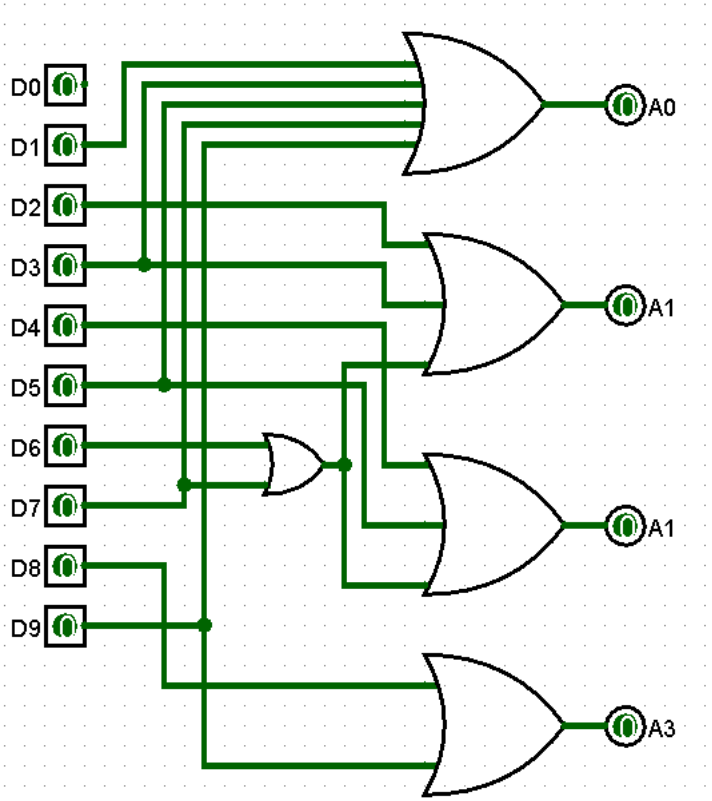
  $A_2 = D_4 + D_5 + D_6 + D_7$

  $A_1 = D_2 + D_3 + D_6 + D_7$

  $A_0 = D_1 + D_3 + D_5 + D_7 + D_9$
- $F_1 = D_6 + D_7$ can be extracted from A_2 and A_1 Is there any cost saving?

Encoder Example (continued)





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:

$A_2 = D_4$

$A_1 = D_4' D_3 + D_4' D_3' D_2 = D_4' F_1, \quad F_1 = (D_3 + D_2)$

$A_0 = D_4' D_3 + D_4' D_3' D_2' D_1 = D_4' (D_3 + D_2' D1)$

$V = D_4 + F_1 + D_1 + D_0$

3-7 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

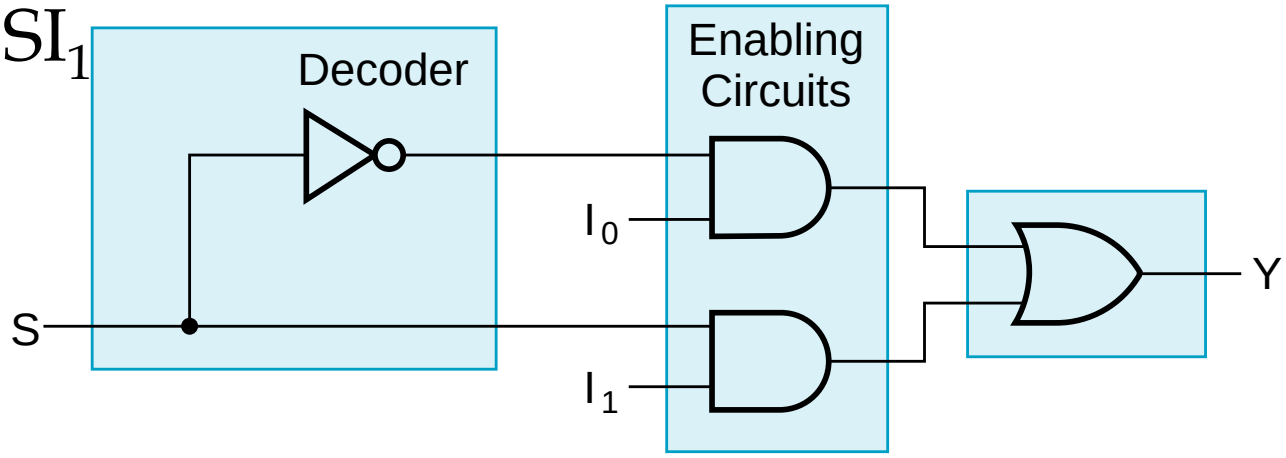
- A multiplexer selects information from an input line and directs the information to an output line
- A typical multiplexer has *n* control inputs (S_{n-1}, … S_0) called *selection inputs*, 2^n information inputs (I_{2^n-1}, … I_0), and one output Y
- A multiplexer can be designed to have *m* information 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:

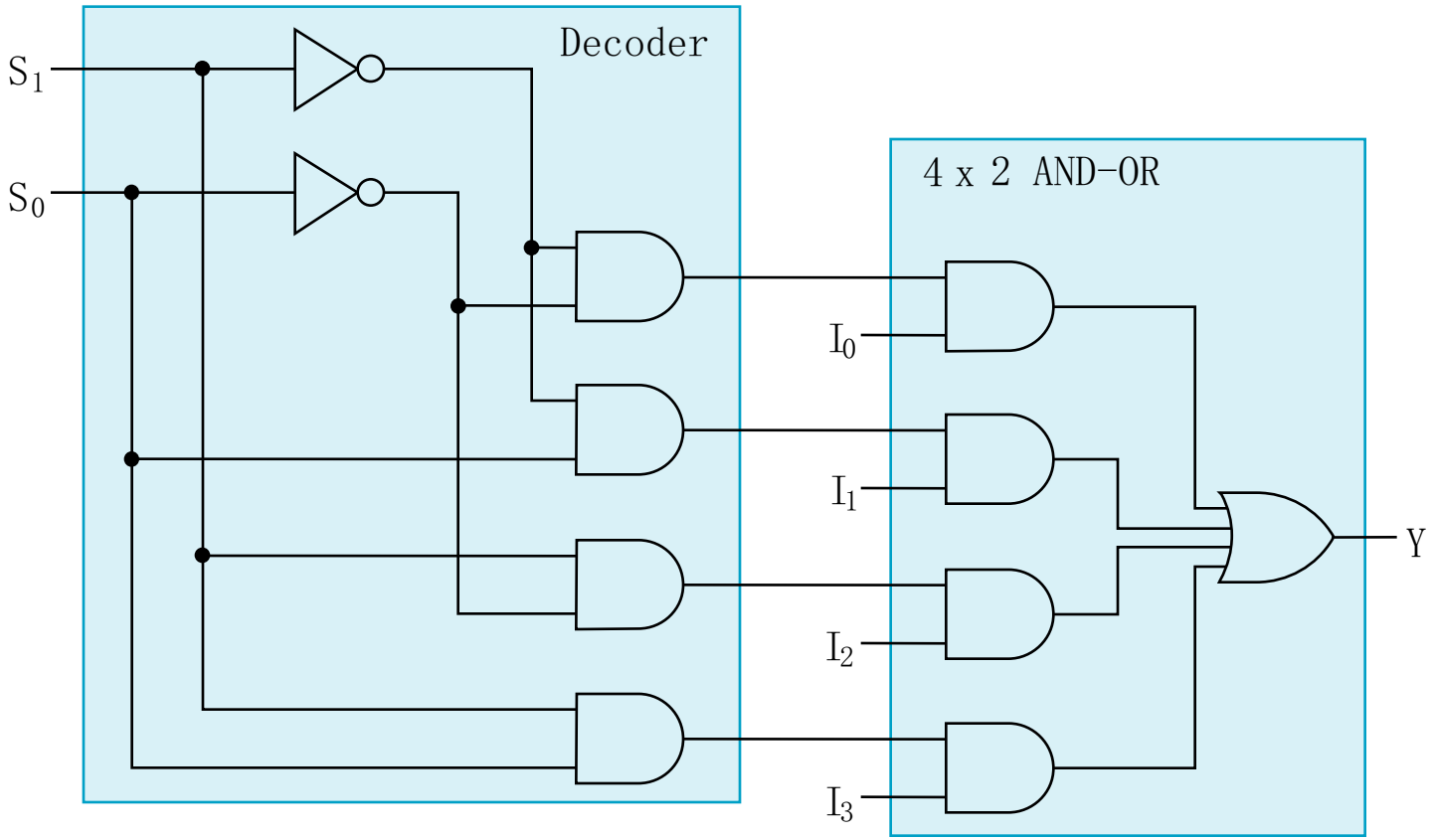


2-to-1-Line Multiplexer (continued)

- 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 x 2 AND-OR circuit:
  - 1-to-2-line decoder
  - 2 x 2 AND-OR
- In general, for an 2^n-to-1-line multiplexer:
  - n-to-2^n-line decoder
  - 2^n x 2 AND-OR

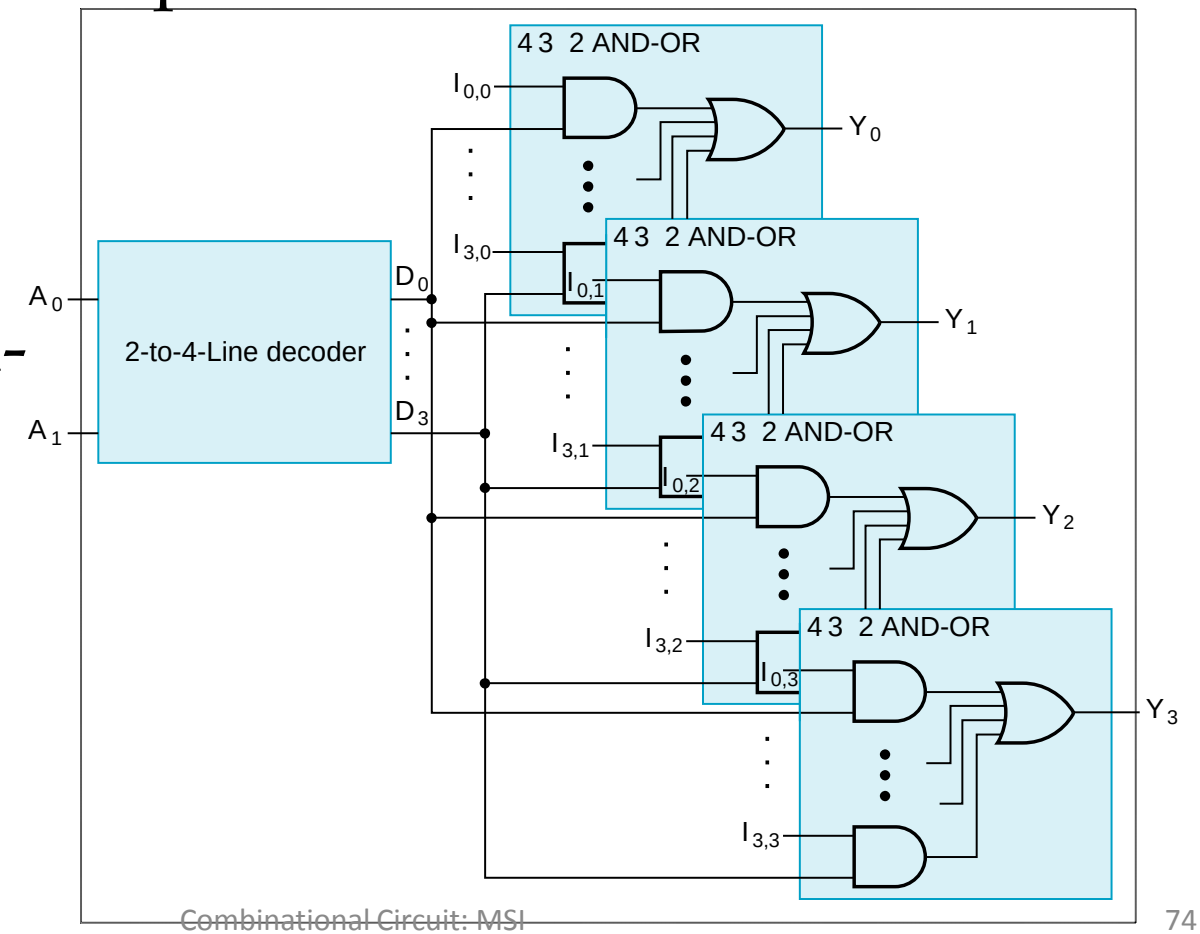
Example: 4-to-1-line Multiplexer

- 2-to-2^2-line decoder
- 2^2 x 2 AND-OR



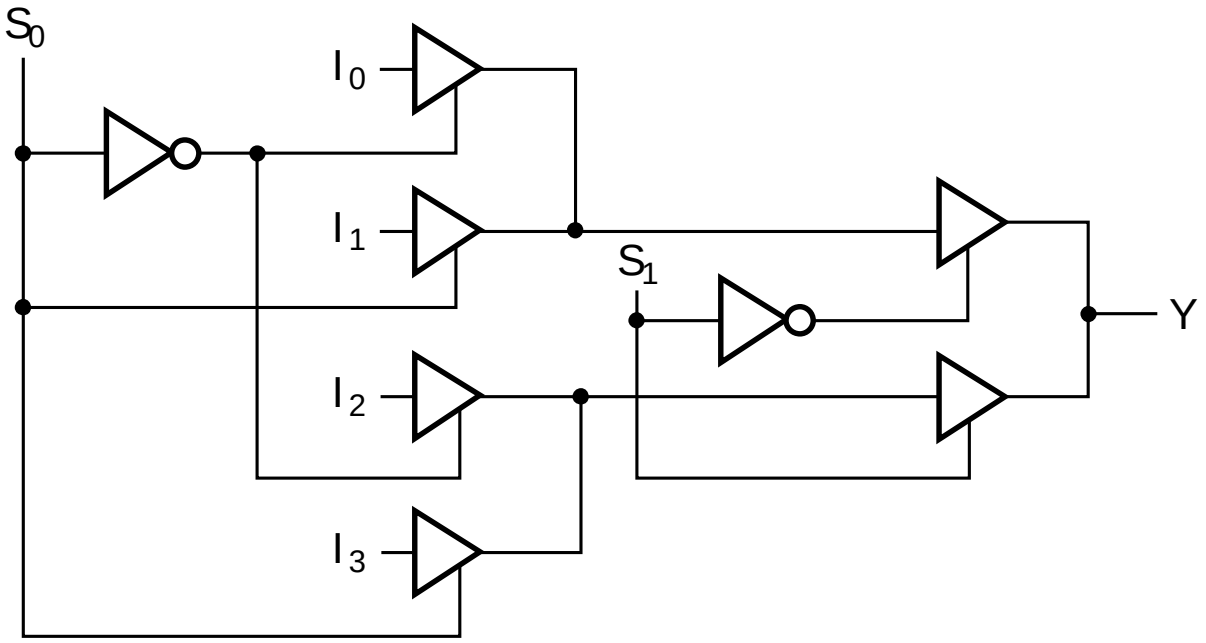
Multiplexer Width Expansion

- Select "vectors of bits" instead of "bits"
- Use multiple copies of 2^n x 2 AND-OR in parallel
- Example: 4-to-1-line quad multi-plexer



Other Selection Implementations

- Three-state logic in place of AND-OR



(b)

- 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 = A and the Y and Z are more complex

Gray Code			Binary		
A	B	C	X	Y	Z
0	0	0	0	0	0
0	0	1	0	0	1
0	1	1	0	1	0
0	1	0	0	1	1
1	1	0	1	0	0
1	1	1	1	0	1
1	0	1	1	1	0
1	0	0	1	1	1

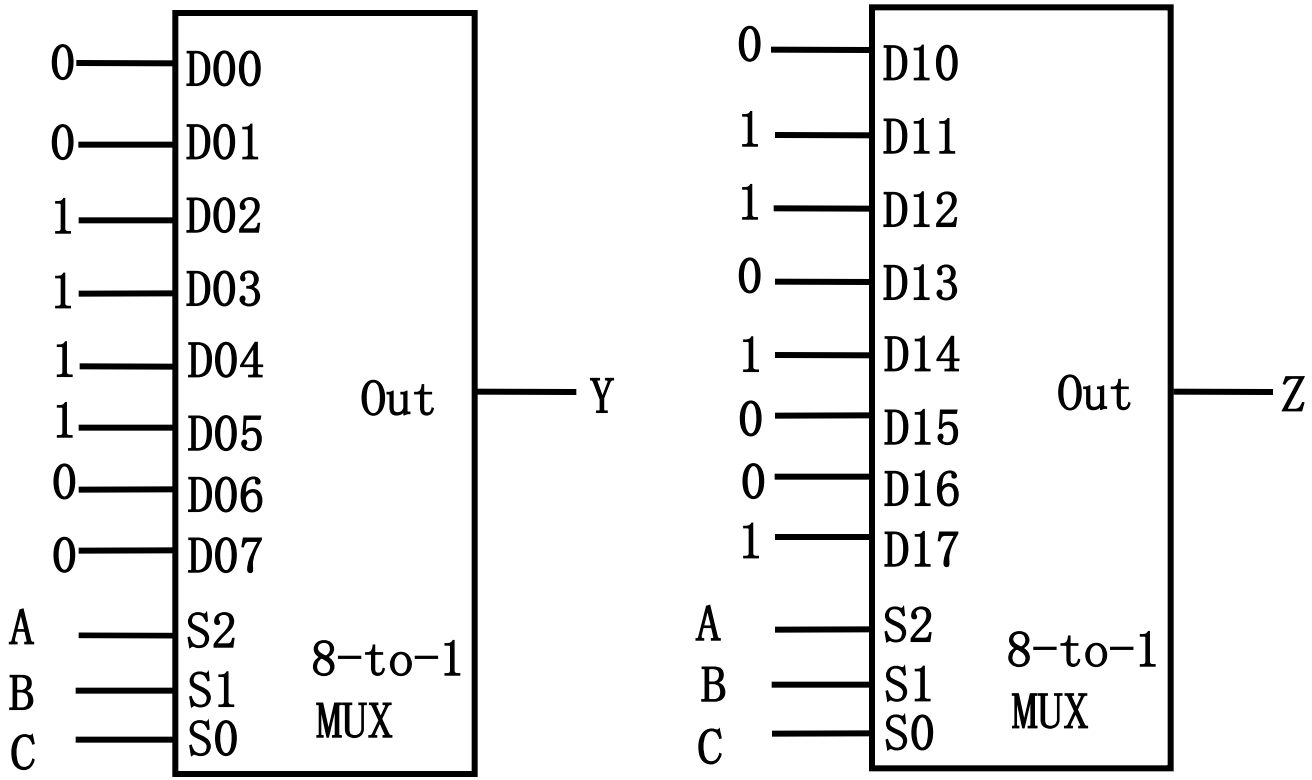
Gray to Binary (continued)

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

Gray Code			Binary		
A	B	C	X	Y	Z
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	1
0	1	1	0	1	0
1	0	0	1	1	1
1	0	1	1	1	0
1	1	0	1	0	0
1	1	1	1	0	1

- 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 to Binary (continued)



- 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, $\overline{\mathbf{X}}$)
  - 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 $\overline{\mathbf{X}}$

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 = A

Gray Code			Binary		
A	B	C	X	Y	Z
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	1
0	1	1	0	1	0
1	0	0	1	1	1
1	0	1	1	1	0
1	1	0	1	0	0
1	1	1	1	0	1

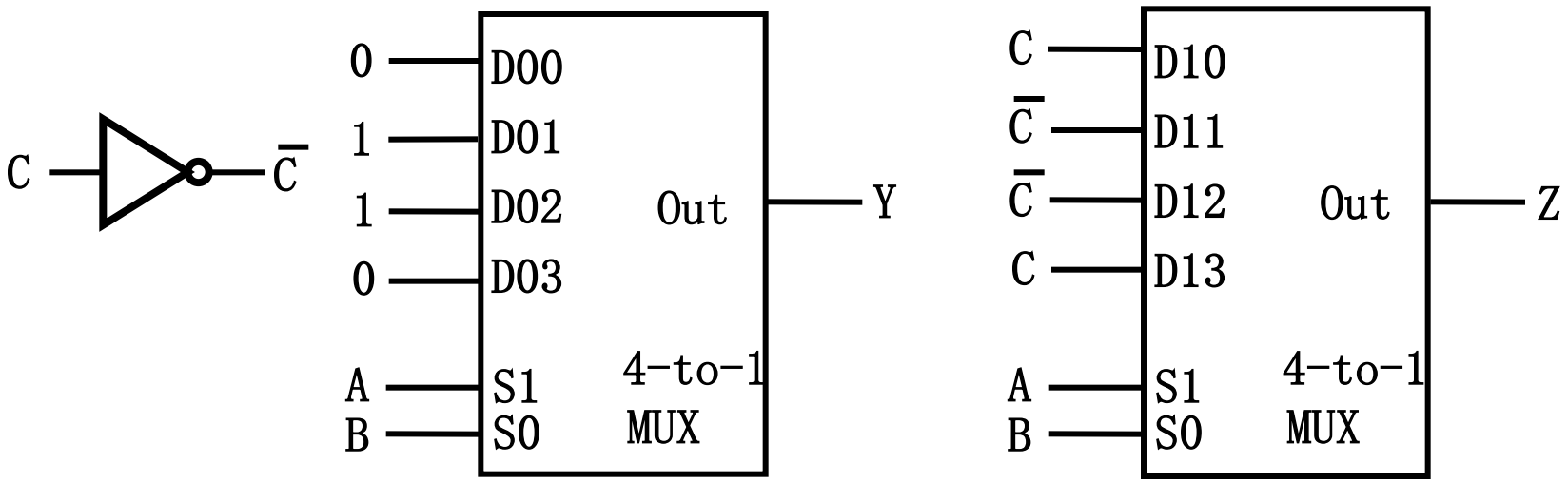
Gray to Binary (continued)

- 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 = 0	F = C
0 0 1	0 0 1		
0 1 0	0 1 1	F = 1	$F = \overline{C}$
0 1 1	0 1 0		
1 0 0	1 1 1	F = 1	$F = \overline{C}$
1 0 1	1 1 0		
1 1 0	1 0 0	F = 0	F = C
1 1 1	1 0 1		

Gray to Binary (continued)

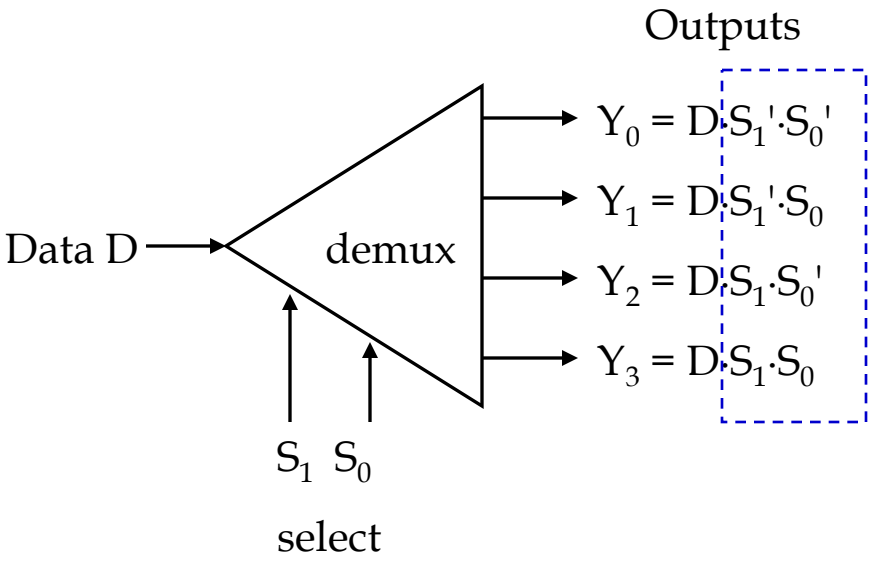
- Assign the variables and functions to the multiplexer inputs:



- 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.

Demultiplexers (1/2)

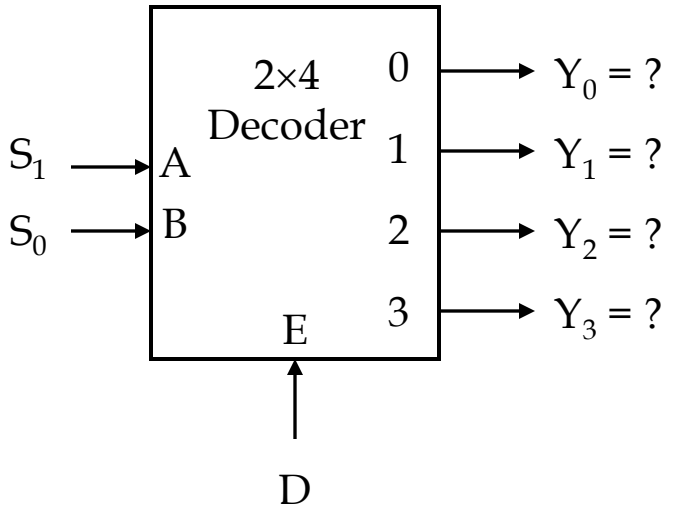
- Given an input line and a set of selection lines, a demultiplexer directs data from the input to one selected output line.
- Example: 1-to-4 demultiplexer.



S_1	S_0	Y_0	Y_1	Y_2	Y_3
0	0	D	0	0	0
0	1	0	D	0	0
1	0	0	0	D	0
1	1	0	0	0	D

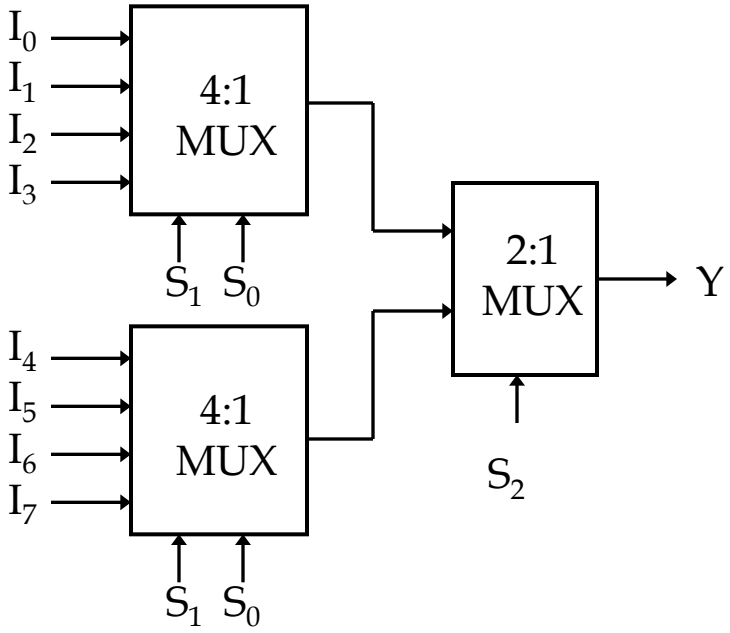
Demultiplexers (2/2)

- It turns out that the demultiplexer circuit is actually identical to a decoder with enable.



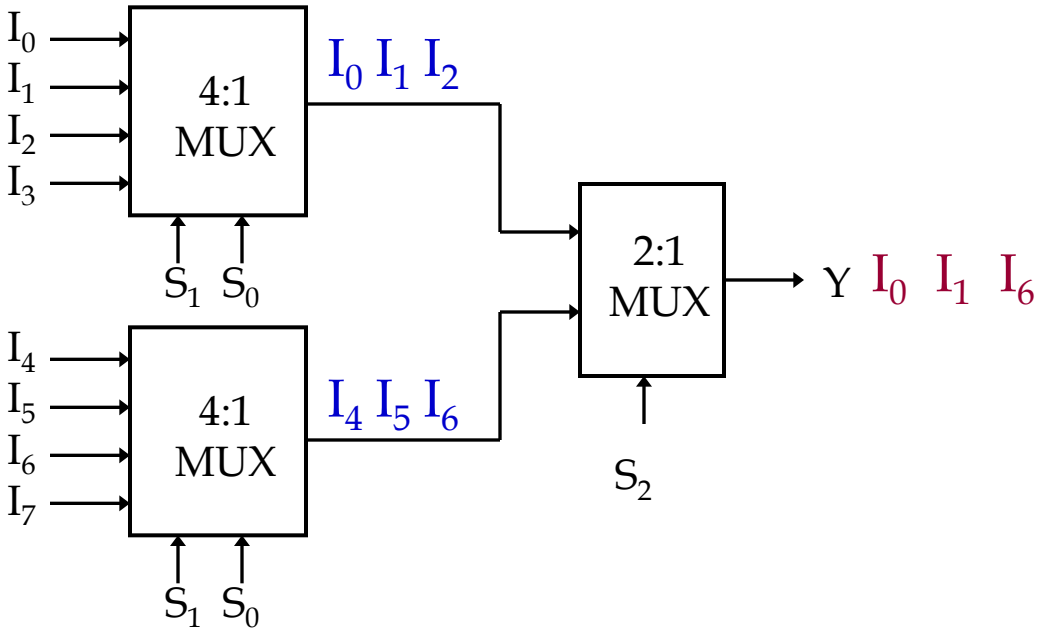
Larger Multiplexers (1/4)

- Larger multiplexers can be constructed from smaller ones.
- An 8-to-1 multiplexer can be constructed from smaller multiplexers like this (note placement of selector lines):



S_2	S_1	S_0	Y
0	0	0	I_0
0	0	1	I_1
0	1	0	I_2
0	1	1	I_3
1	0	0	I_4
1	0	1	I_5
1	1	0	I_6
1	1	1	I_7

Larger Multiplexers (2/4)

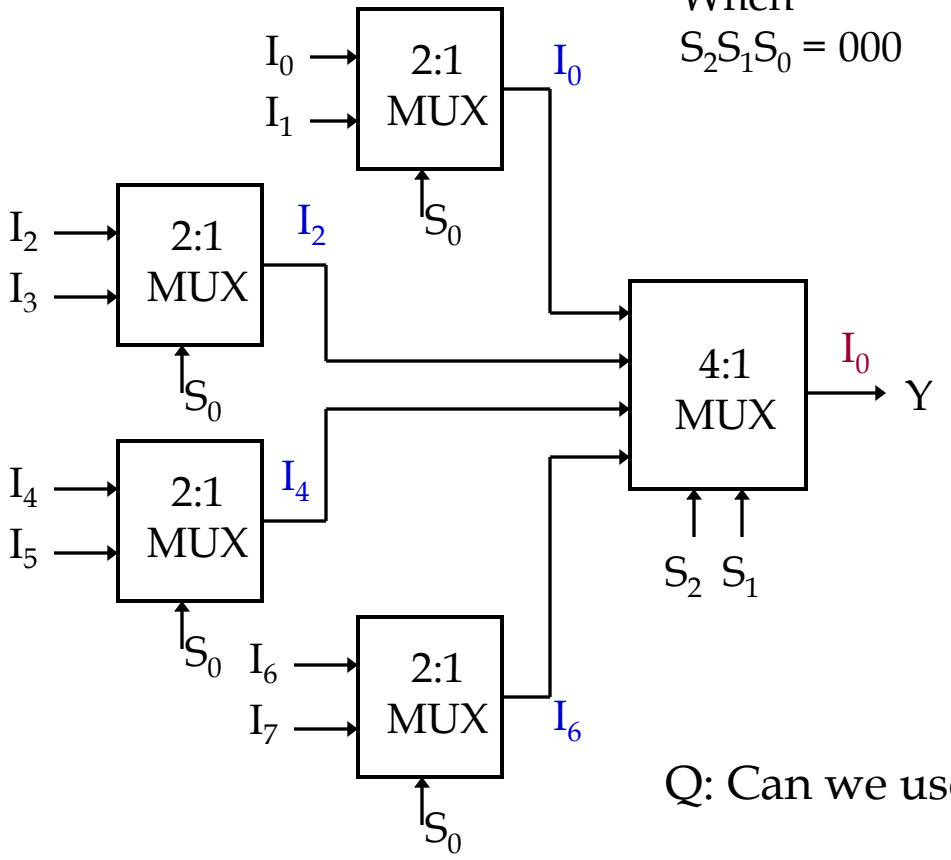


S_2	S_1	S_0	Y
0	0	0	I_0
0	0	1	I_1
0	1	0	I_2
0	1	1	I_3
1	0	0	I_4
1	0	1	I_5
1	1	0	I_6
1	1	1	I_7

- When $S_2S_1S_0 = 000$
- When $S_2S_1S_0 = 001$
- When $S_2S_1S_0 = 110$

Larger Multiplexers (3/4)

- Another implementation of an 8-to-1 multiplexer using smaller multiplexers:

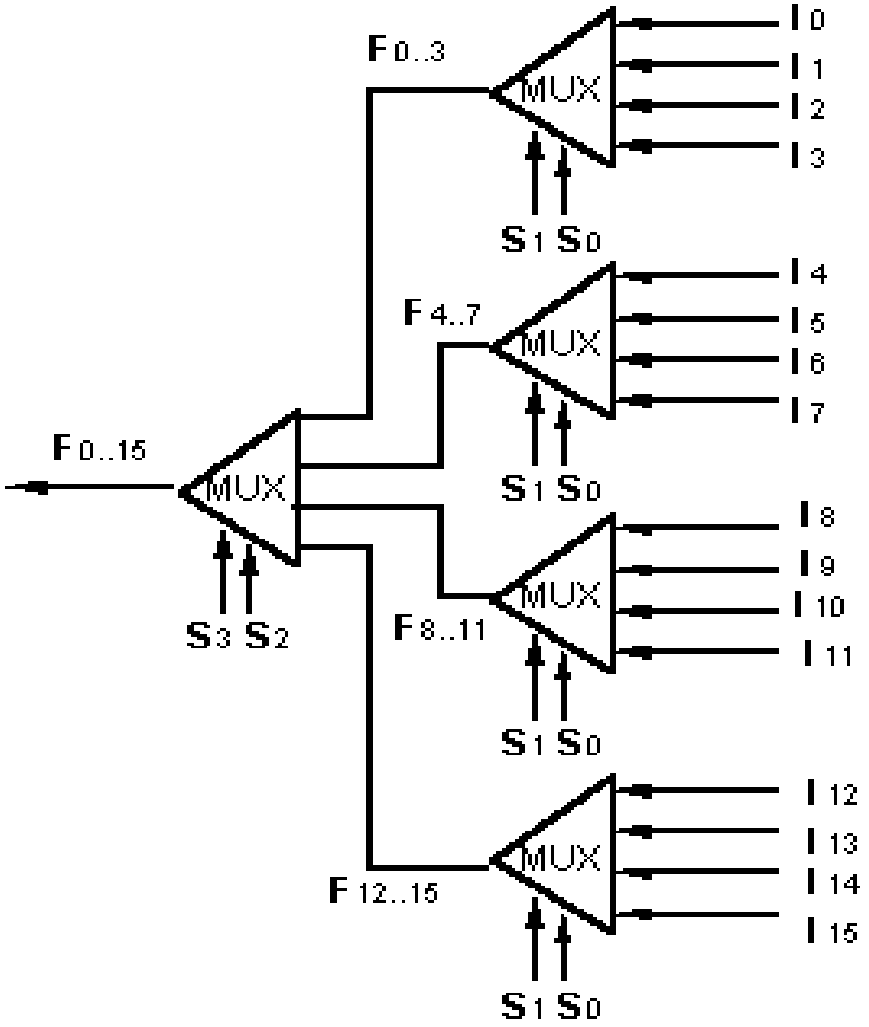


S_2	S_1	S_0	Y
0	0	0	I_0
0	0	1	I_1
0	1	0	I_2
0	1	1	I_3
1	0	0	I_4
1	0	1	I_5
1	1	0	I_6
1	1	1	I_7

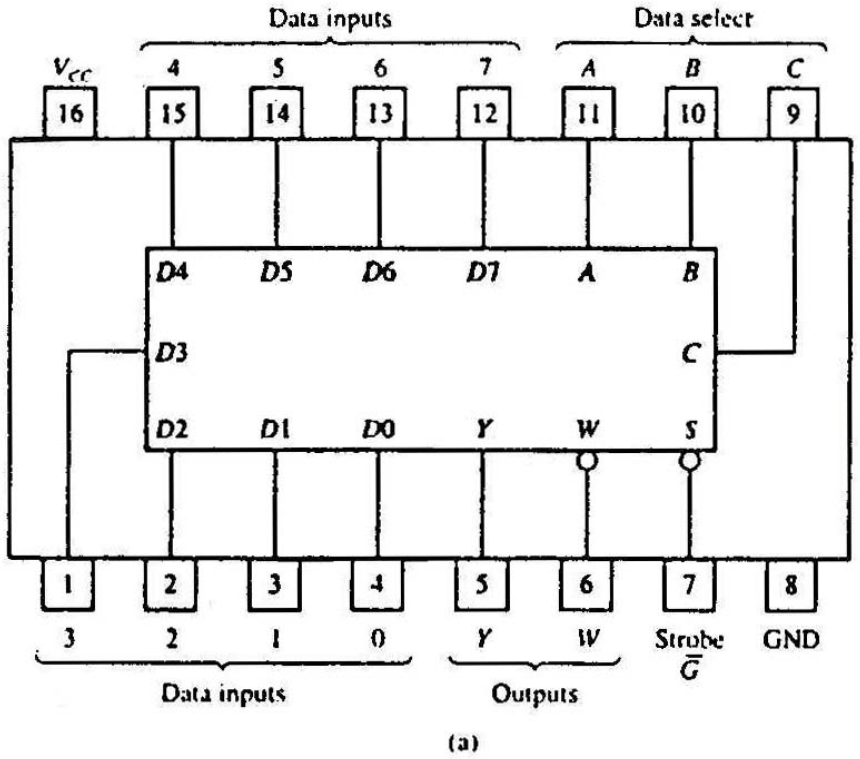
Q: Can we use only 2:1 multiplexers?

Larger Multiplexers (4/4)

- A 16-to-1 multiplexer can be constructed from five 4-to-1 multiplexers:



Standard MSI Multiplexer (1/2)



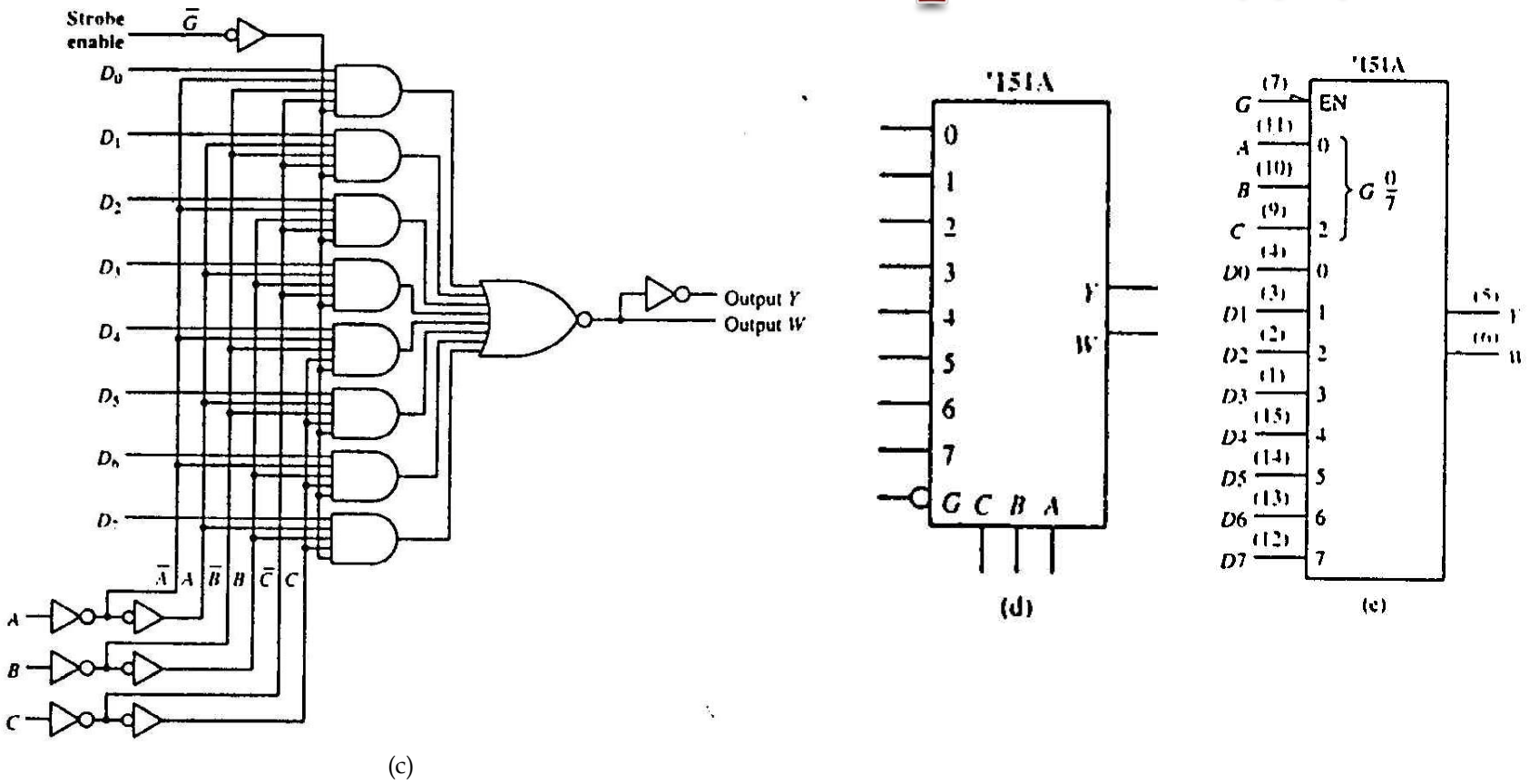
(a)

INPUTS				OUTPUTS	
SELECT			STROBE	Y	W
C	B	A	$\overline{G}$		
X	X	X	H	L	H
L	L	L	L	D0	$\overline{D0}$
L	L	H	L	D1	$\overline{D1}$
L	H	L	L	D2	$\overline{D2}$
L	H	H	L	D3	$\overline{D3}$
H	L	L	L	D4	$\overline{D4}$
H	L	H	L	D5	$\overline{D5}$
H	H	L	L	D6	$\overline{D6}$
H	H	H	L	D7	$\overline{D7}$

(b)

74151A 8-to-1 multiplexer. (a) Package configuration. (b) Function table.

Standard MSI Multiplexer (2/2)



(c)

74151A 8-to-1 multiplexer. (c) Logic diagram. (d) Generic logic symbol. (e) IEEE standard logic symbol.

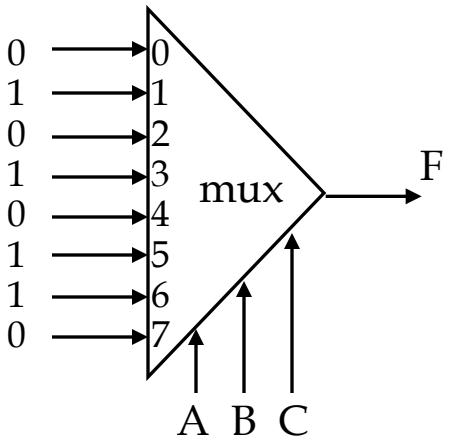
Source: The TTL Data Book Volume 2. Texas Instruments Inc.,1985.

Multiplexers: Implementing Functions (1/3)

- Boolean functions can be implemented using multiplexers.
- A 2^n-to-1 multiplexer can implement a Boolean function of n input variables, as follows:
  1. Express in sum-of-minterms form. Example:
     $F(A,B,C) = A'\cdot B'\cdot C + A'\cdot B\cdot C + A\cdot B'\cdot C + A\cdot B\cdot C'$
     $= \sum m(1,3,5,6)$
  2. Connect n variables to the n selection lines.
  3. Put a '1' on a data line if it is a minterm of the function, or '0' otherwise.

Multiplexers: Implementing Functions (2/3)

- $F(A,B,C) = \sum m(1,3,5,6)$



This method works because:

$$\text{Output} = m_0 \cdot I_0 + m_1 \cdot I_1 + m_2 \cdot I_2 + m_3 \cdot I_3 + m_4 \cdot I_4 + m_5 \cdot I_5 + m_6 \cdot I_6 + m_7 \cdot I_7$$

Supplying '1' to I_1, I_3, I_5, I_6 , and '0' to the rest:

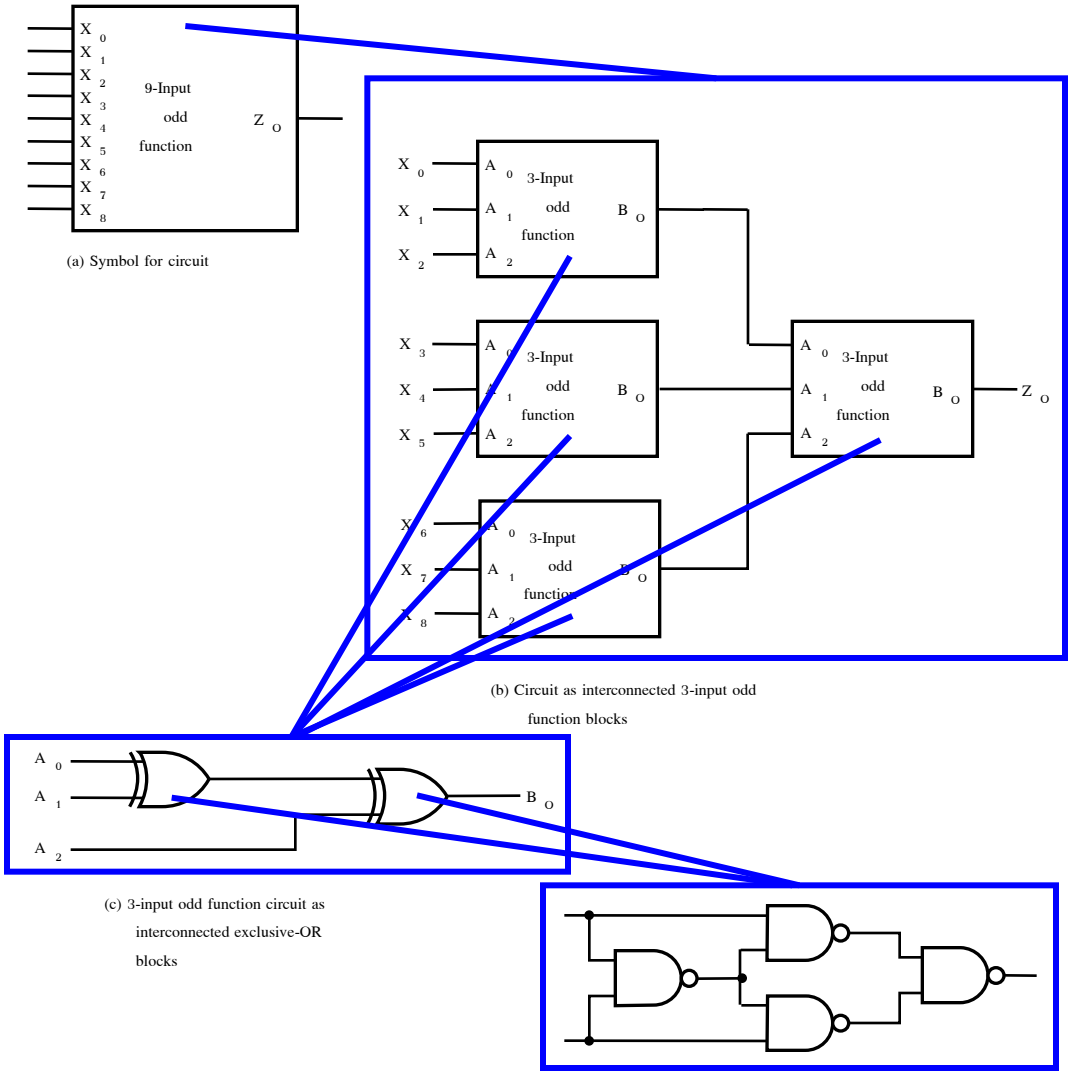
$$\text{Output} = m_1 + m_3 + m_5 + m_6$$

3-9 Binary Adders

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

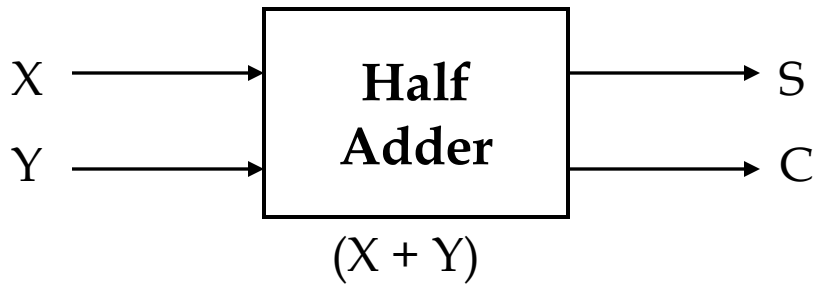
- Whenever possible, we try to decompose a complex design into common, *reusable* function blocks
- These blocks are
  - verified and well-documented
  - placed in libraries for future use

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 in this chapter is devoted to learning about combinational blocks used in top-down design.

Gate-Level (SSI) Design: Half Adder (1/2)

- Design procedure:
  1. State problem
     Example: Build a Half Adder.
  2. Determine and label the inputs and outputs of circuit.
     Example: Two inputs and two outputs labelled, as shown below.



  3. Draw the truth table.

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

Gate-Level (SSI) Design: Half Adder (2/2)

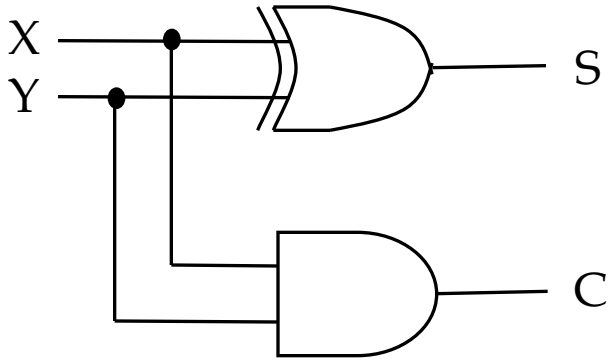
4. Obtain simplified Boolean functions.

   Example: $C = X \cdot Y$

   $S = X' \cdot Y + X \cdot Y' = X \oplus Y$

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

5. Draw logic diagram.



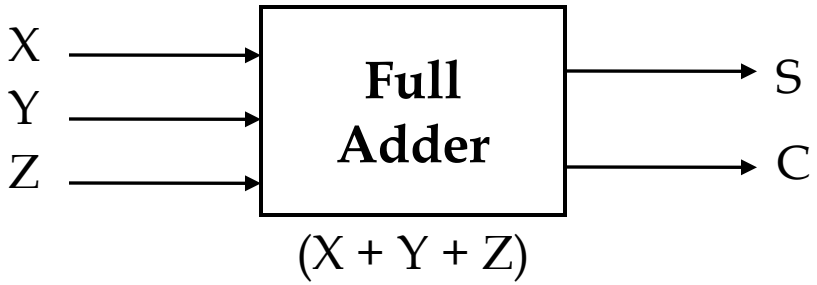
Half Adder

Gate-Level (SSI) Design: Full Adder (1/5)

- Half adder adds up only two bits.
- To add two binary numbers, we need to add 3 bits (including the *carry*).
  - Example:

	1	1	1		carry
	0	0	1	1	X
+	0	1	1	1	Y
	1	0	1	0	S

- Need Full Adder (so called as it can be made from two half adders).



Gate-Level (SSI) Design: Full Adder (2/5)

- 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

Note:

Z - carry in (to the current position)

C - carry out (to the next position)

- Using K-map, simplified SOP form:

  C = ?

  S = ?

C

X \ YZ	00	01	11	10
0			1	
1		1	1	1

S

X \ YZ	00	01	11	10
0		1		1
1	1		1	

Gate-Level (SSI) Design: Full Adder (3/5)

- Alternative formulae using algebraic manipulation:

$$
\begin{aligned}
C &= X\cdot Y + X\cdot Z + Y\cdot Z \\
&= X\cdot Y + (X + Y)\cdot Z \\
&= X\cdot Y + ((X\oplus Y) + X\cdot Y)\cdot Z \\
&= X\cdot Y + (X\oplus Y)\cdot Z + X\cdot Y\cdot Z \\
&= X\cdot Y + X\cdot Y\cdot Z + (X\oplus Y)\cdot Z \\
&= X\cdot Y + (X\oplus Y)\cdot Z
\end{aligned}
$$

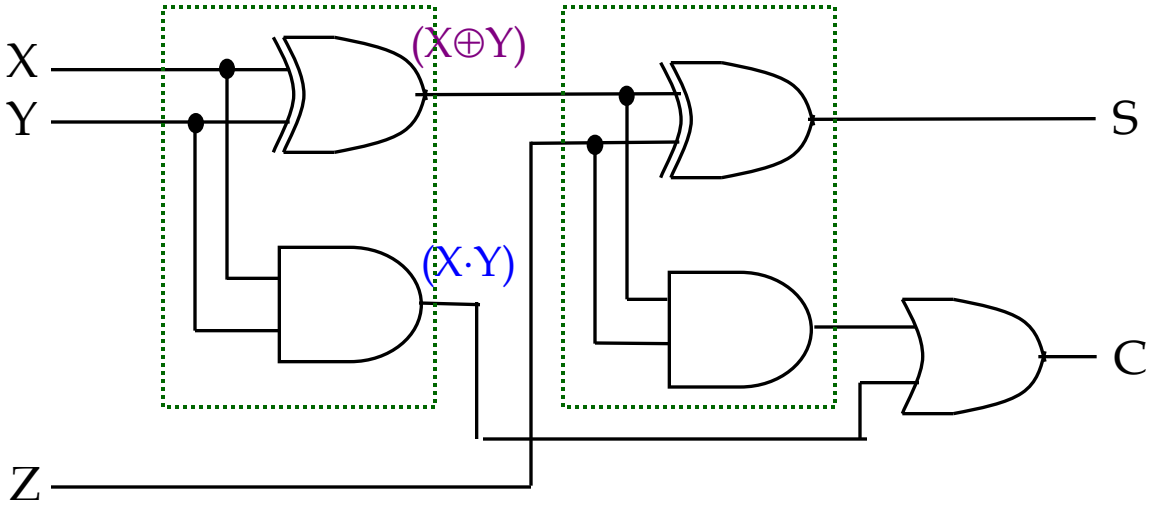
$$
\begin{aligned}
S &= X'\cdot Y'\cdot Z + X'\cdot Y\cdot Z' + X\cdot Y'\cdot Z' + X\cdot Y\cdot Z \\
&= X'\cdot(Y'\cdot Z + Y\cdot Z') + X\cdot(Y'\cdot Z' + Y\cdot Z) \\
&= X'\cdot(Y\oplus Z) + X\cdot(Y\oplus Z)' \\
&= X\oplus(Y\oplus Z)
\end{aligned}
$$

Gate-Level (SSI) Design: Full Adder (4/5)

- Circuit for above formulae:

  $C = X \cdot Y + (X \oplus Y) \cdot Z$

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



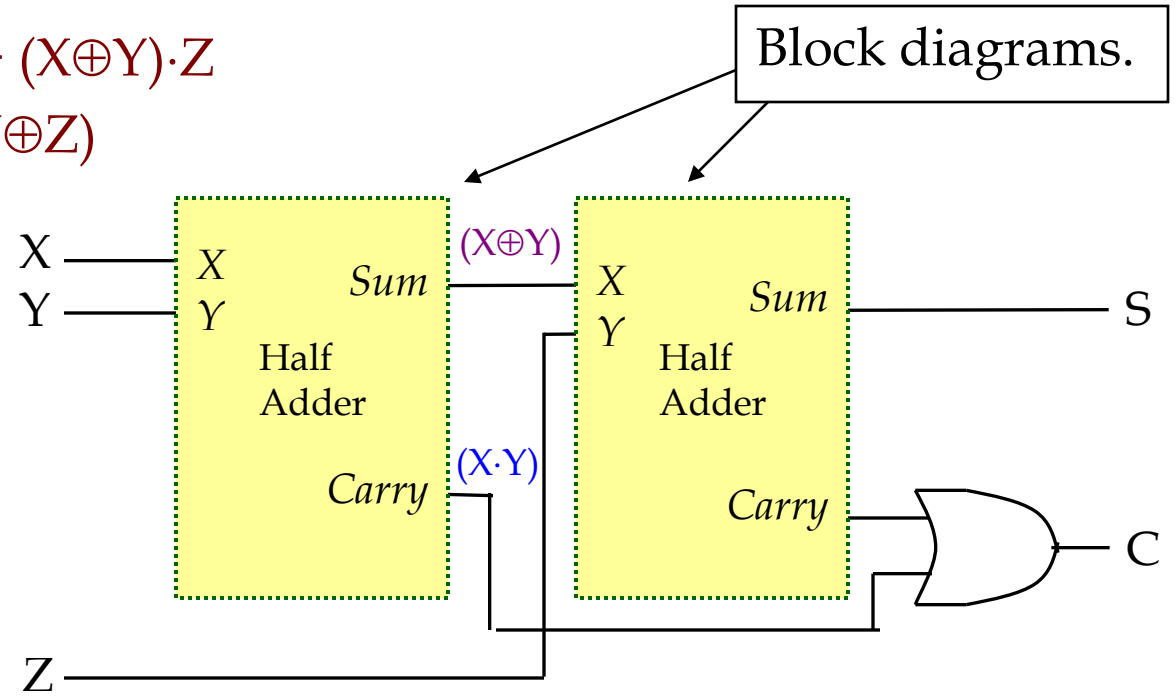
Full Adder made from two Half-Adders (+ an OR gate).

Gate-Level (SSI) Design: Full Adder (5/5)

- Circuit for above formulae:

  C= X·Y + (X⊕Y)·Z

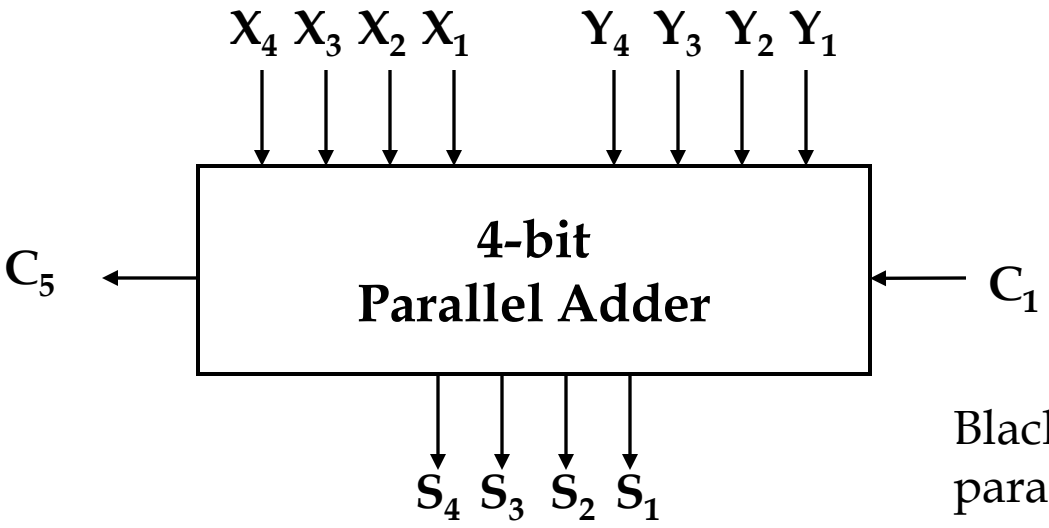
  S = X⊕(Y⊕Z)



Full Adder made from two Half-Adders (+ an OR gate).

4-Bit Parallel Adder (1/4)

- Consider a circuit to add two 4-bit numbers together and a carry-in, to produce a 5-bit result.



Black-box view of 4-bit parallel adder

- 5-bit result is sufficient because the largest result is:

$$1111_2 + 1111_2 + 1_2 = 11111_2$$

4-Bit Parallel Adder (2/4)

- SSI design technique should not be used here.
- Truth table for 9 inputs is too big: $2^9 = 512$ rows!

$X_4X_3X_2X_1$	$Y_4Y_3Y_2Y_1$	C_1	C_5	$S_4S_3S_2S_1$
0 0 0 0	0 0 0 0	0	0	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 0 1
…	…	…	…	…
0 1 0 1	1 1 0 1	1	1	0 0 1 1
…	…	…	…	…
1 1 1 1	1 1 1 1	1	1	1 1 1 1

- Simplification becomes too complicated.

4-Bit Parallel Adder (3/4)

- Alternative design possible.
- Addition formula for each pair of bits (with carry in),

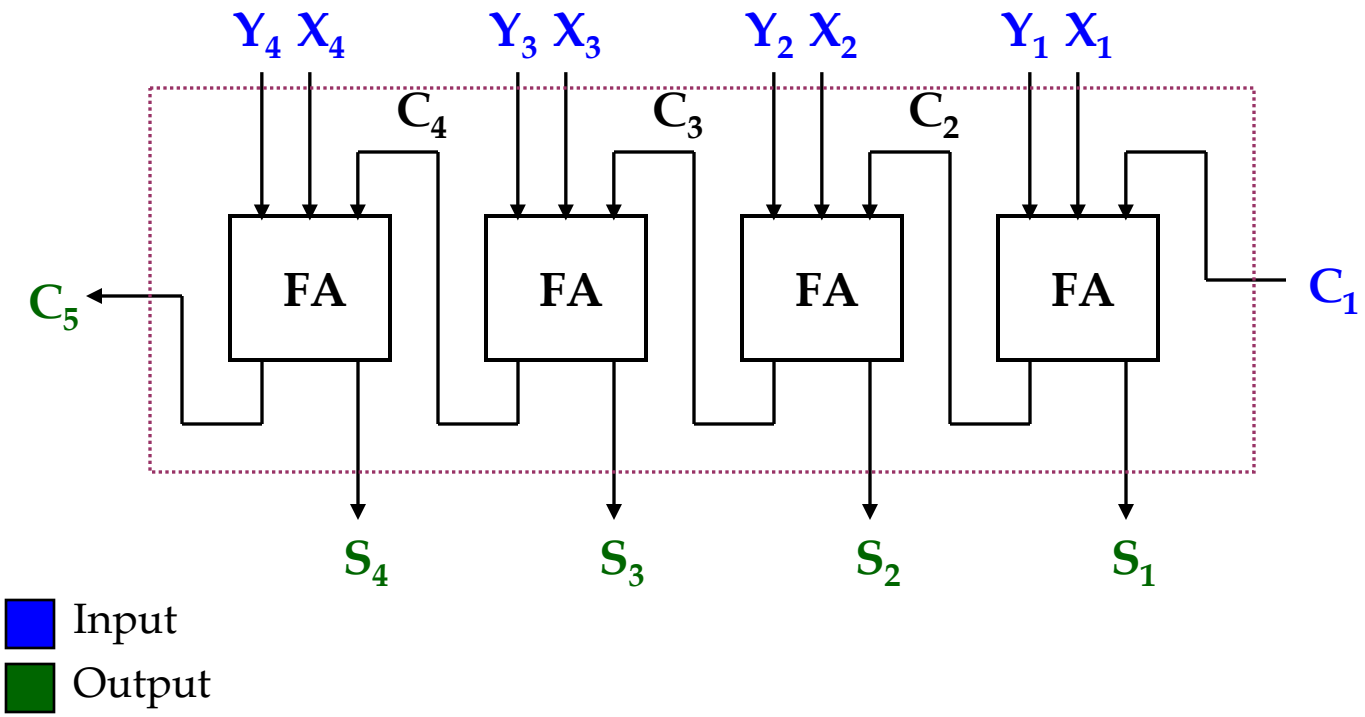
$$C_{i+1}S_i = X_i + Y_i + C_i$$

has the same function as a full adder:

$$C_{i+1} = X_i \cdot Y_i + (X_i \oplus Y_i) \cdot C_i$$
$$S_i = X_i \oplus Y_i \oplus C_i$$

4-Bit Parallel Adder (4/4)

- Cascading 4 full adders via their carries, we get:



Parallel Adder

- Note that carry propagated by cascading the carry from one full adder to the next.
- Called Parallel Adder because inputs are presented simultaneously (in parallel). Also called Ripple-Carry Adder.

3-10 Binary Subtraction

Single Bit Binary Subtraction with Borrow

- Given two binary digits (X,Y), a borrow in (Z) we get the following difference (S) and borrow (B):
- Borrow in (Z) of 0:

Z	**0**	**0**	**0**	**0**
X	**0**	**0**	**1**	**1**
- Y	**-0**	**-1**	**-0**	**-1**
BS	**0 0**	**1 1**	**0 1**	**0 0**

- Borrow in (Z) of 1:

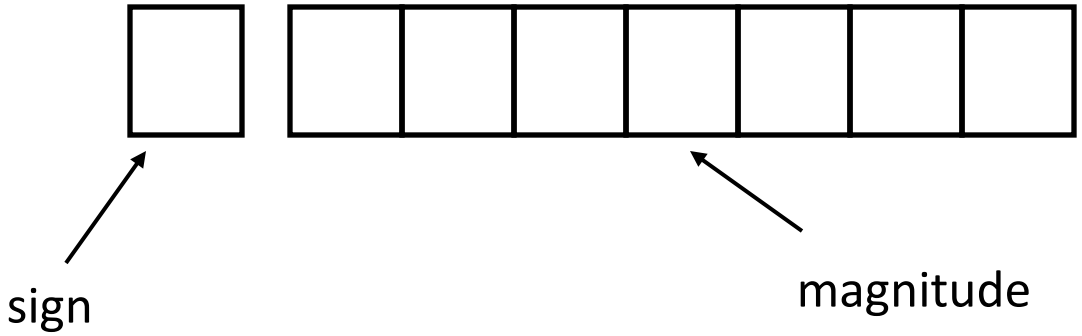
Z	**1**	**1**	**1**	**1**
X	**0**	**0**	**1**	**1**
- Y	**-0**	**-1**	**-0**	**-1**
BS	**11**	**1 0**	**0 0**	**1 1**

Negative Number

- Unsigned numbers: only non-negative values.
- Signed numbers: include all values (positive and negative)
- There are 3 common representations for signed binary numbers:
  - Sign-and-Magnitude
  - 1s Complement
  - 2s Complement

Sign and magnitude (1/3)

- The sign is represented by a 'sign bit'
  - 0 for +
  - 1 for -
- E.g.: a 1-bit sign and 7-bit magnitude format.



- 00110100 ➔ $+110100_2 = +52_{10}$
- 10010011 ➔ $-10011_2 = -19_{10}$

Sign and magnitude (2/3)

- Largest value: $01111111 = +127_{10}$
- Smallest value: $11111111 = -127_{10}$
- Zeros: $00000000 = +0_{10}$
  $10000000 = -0_{10}$
- Range: -127_{10} to $+127_{10}$
- Question:

  For an *n*-bit sign-and-magnitude representation, what is the range of values that can be represented?

Sign and magnitude (3/3)

- To negate a number, just invert the sign bit.
- Examples:
  - How to negate 00100001_{sm} (decimal 33)?
    Answer: 10100001_{sm} (decimal -33)
  - How to negate 10000101_{sm} (decimal -5)?
    Answer: 00000101_{sm} (decimal +5)

1s complement (1/3)

- Given a number x which can be expressed as an n-bit binary number, its negated value can be obtained in **1s-complement** representation using:

$$-x = 2^n - x - 1$$

- Example: With an 8-bit number 00001100 (or 12_{10}), its negated value expressed in 1s-complement is:

-00001100_2 $= 2^8 - 12 - 1$ (calculation in decimal)
$= 243$
$= 11110011_{1s}$

(This means that -12_{10} is written as 11110011 in 1s-complement representation.)

1s complement (2/3)

- Essential technique to negate a value: invert all the bits.
- Largest value: $01111111 = +127_{10}$
- Smallest value: $10000000 = -127_{10}$
- Zeros: $00000000 = +0_{10}$
  $11111111 = -0_{10}$
- Range: -127_{10} to $+127_{10}$
- The most significant (left-most) bit still represents the sign: 0 for positive; 1 for negative.

1s complement (3/3)

Examples (assuming 8-bit numbers):

$(14)_{10} = (00001110)_2 = (00001110)_{1s}$

$-(14)_{10} = -(00001110)_2 = (11110001)_{1s}$

$-(80)_{10} = -(\ ?\ )_2 = (\ ?\ )_{1s}$

2s complement (1/3)

- Given a number x which can be expressed as an n-bit binary number, its negated value can be obtained in **2s-complement** representation using:

$$-x = 2^n - x$$

- Example: With an 8-bit number 00001100 (or 12_{10}), its negated value expressed in 1s-complement is:

-00001100_2 $= 2^8 - 12$ (calculation in decimal)
$= 244$
$= 11110100_{2s}$

(This means that -12_{10} is written as 11110100 in 2s-complement representation.)

2s complement (2/3)

- Essential technique to negate a value: invert all the bits, then add 1.
- Largest value: $01111111 = +127_{10}$
- Smallest value: $10000000 = -128_{10}$
- Zero: $00000000 = +0_{10}$
- Range: -128_{10} to $+127_{10}$
- The most significant (left-most) bit still represents the sign: 0 for positive; 1 for negative

2s complement (3/3)

Examples (assuming 8-bit numbers):

$(14)_{10} = (00001110)_2 = (00001110)_{2s}$

$-(14)_{10} = -(00001110)_2 = (11110010)_{2s}$

$-(80)_{10} = -(\ ?\ )_2 = (\ ?\ )_{2s}$

Comparison

Important!

4-bit system

Positive values

Value	Sign-and-Magnitude	1s Comp.	2s Comp.
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000

Negative values

Value	Sign-and-Magnitude	1s Comp.	2s Comp.
-0	1000	1111	-
-1	1001	1110	1111
-2	1010	1101	1110
-3	1011	1100	1101
-4	1100	1011	1100
-5	1101	1010	1011
-6	1110	1001	1010
-7	1111	1000	1001
-8	-	-	1000

Complement on Fraction

- We can extend the idea of complement on fractions.
- Examples:
  - Negate 0101.01 in 1s-complement
    Answer: 1010.10
  - Negate 111000.101 in 1s-complement
    Answer: 000111.010
  - Negate 0101.01 in 2s-complement
    Answer: 1010.11

2s Complement Addition/Subtraction (1/3)

- Algorithm for addition, A + B:
  1. Perform binary addition on the two numbers.
  2. Ignore the carry out of the MSB.
  3. Check for overflow. Overflow occurs if the 'carry in' and 'carry out' of the MSB are different, or if result is opposite sign of A and B.
- Algorithm for subtraction, A – B:

  A – B = A + (-B)
  1. Take 2s-complement of B.
  2. Add the 2s-complement of B to A.

Overflow

- Signed numbers are of a fixed range.
- If the result of addition/subtraction goes beyond this range, an **overflow** occurs.
- Overflow can be easily detected:
  - *positive add positive* → negative
  - *negative add negative* → positive
- Example: 4-bit 2s-complement system
  - Range of value: -8_{10} to 7_{10}
  - $0101_{2s} + 0110_{2s} = 1011_{2s}$
    $5_{10} + 6_{10} = -5_{10}$?! (overflow!)
  - $1001_{2s} + 1101_{2s} = \underline{1}0110_{2s}$ (discard end-carry) $= 0110_{2s}$
    $-7_{10} + -3_{10} = 6_{10}$?! (overflow!)

2s Complement Addition/Subtraction (2/3)

- Examples: 4-bit system

```
  +3      0011
+ +4    + 0100
----    -------
  +7      0111
----    -------
```

```
  -2      1110
+ -6    + 1010
----    -------
  -8     11000
----    -------
```

```
  +6      0110
+ -3    + 1101
----    -------
  +3     10011
----    -------
```

```
  +4      0100
+ -7    + 1001
----    -------
  -3      1101
----    -------
```

- Which of the above is/are overflow(s)?

2s Complement Addition/Subtraction (3/3)

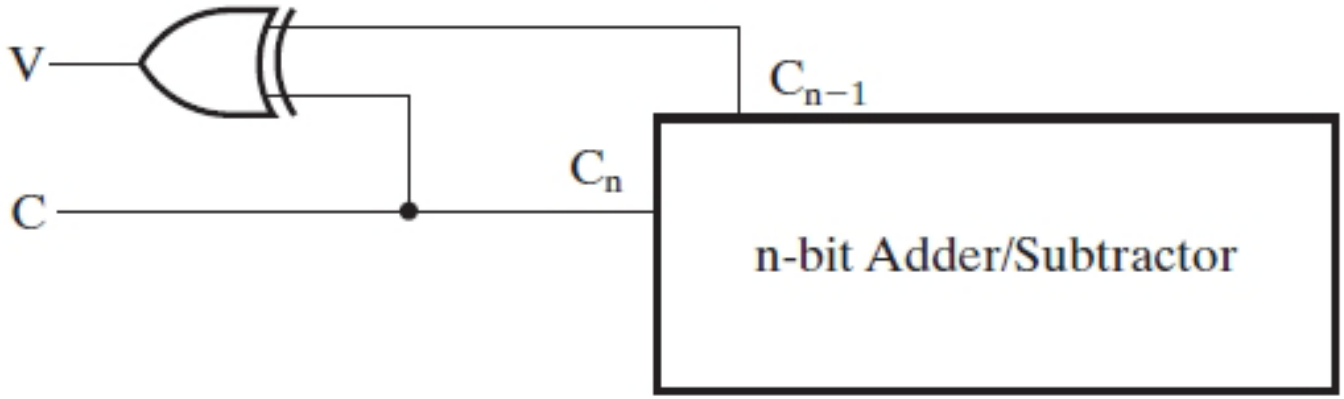
- Examples: 4-bit system

```
   -3        1101
 + -6      + 1010
 ----      -------
   -9       10111
 ----      -------
```

```
   +5        0101
 + +6      + 0110
 ----      -------
  +11        1011
 ----      -------
```

- Which of the above is/are overflow(s)?

Overflow Detection



1s Complement Addition/Subtraction (1/2)

- Algorithm for addition, A + B:
  1. Perform binary addition on the two numbers.
  2. If there is a carry out of the MSB, add 1 to the result.
  3. Check for overflow. Overflow occurs if result is opposite sign of A and B.
- Algorithm for subtraction, A – B:

  A – B = A + (-B)
  1. Take 1s-complement of B.
  2. Add the 1s-complement of B to A.

1s Complement Addition/Subtraction (2/2)

- Examples: 4-bit system

Any overflow?

```
  +3      0011
+ +4    + 0100
----    -------
  +7      0111
----    -------
```

```
  +5      0101
+ -5    + 1010
----    -------
  -0      1111
----    -------
```

```
  -2      1101
+ -5    +  1010
----    ------
  -7     10111
----   +    1
         ------
          1000
```

```
  -3      1100
+ -7    + 1000
----    -------
 -10     10100
----   +    1
       -------
        0101
```

QUICK REVIEW QUESTIONS (5)

- DLD page 37

2-13. In a 6-bit 2's complement binary number system, what is the decimal value represented by (100100)2s?

a. -4 b. 36 c. -36 d. -27 e. -28

2-14. In a 6-bit 1's complement binary number system, what is the decimal value represented by $(010100)_{1s}$?

a. -11 b. 43 c. -43 d. 20 e. -20

2-15. What is the range of values that can be represented in a 5-bit 2's complement binary systems?

a. 0 to 31 b. -8 to 7 c. -8 to 8 d. -15 to 15 e. -16 to 15

QUICK REVIEW QUESTIONS (5)

- DLD page 37

2-16. In a 4-bit 2's complement scheme, what is the result of this operation: $(1011)_{2s}$ + $(1001)_{2s}$?

a. 4 b. 5 c. 20 d. -12 e. overflow

2-17. Assuming a 6-bit 2's complement system, perform the following subtraction operation by converting it into addition operation:

a. $(011010)_{2s}$ - $(010000)_{2s}$

b. $(011010)_{2s}$ - $(001101)_{2s}$

c. $(000011)_{2s}$ - $(010000)_{2s}$

2-18. Assuming a 6-bit 1's complement system, perform the following subtraction operation by converting it into addition operation:

a. $(011111)_{1s}$ - $(010101)_{1s}$

b. $(001010)_{1s}$ - $(101101)_{1s}$

c. $(100000)_{1s}$ - $(010011)_{1s}$

Eight Conditions for Signed-Magnitude Addition/Subtraction

Operation	ADD Magnitudes	SUBTRACT Magnitudes		
		A > B	A < B	A = B
(+A) + (+B)	+ (A + B)			
(+A) + (-B)		+ (A – B)	- (B – A)	+ (A – B)
(-A) + (+B)		- (A – B)	+ (B – A)	+ (A – B)
(-A) + (-B)	- (A + B)			
(+A) - (+B)		+ (A – B)	- (B – A)	+ (A – B)
(+A) - (-B)	+ (A + B)			
(-A) - (+B)	- (A + B)			
(-A) - (-B)		- (A – B)	+ (B – A)	+ (A – B)

Examples

Example of adding two magnitudes when the result is the sign of both operands:

```
   +3   0 011
+  +2   0 010
-------------
   +5   0 101
```

Example of adding two magnitudes when the result is the sign of the larger magnitude:

```
   -3   1 011
+  +2   0 010
-------------
-( +3     011
 - +2)   010
-------------
  - 1   1 001
```

QUICK REVIEW QUESTIONS (6)

By using 8-bit signed magnitude calculate:

a) $0100\ 1111_2 + 0010\ 0011_2$

b) $0100\ 1111_2 + 0110\ 0011_2$

c) $(99)_{10} - (79)_{10}$

d) $(-19)_{10} + (13)_{10}$

e) Subtract $(-24)_{10}$ from $(-43)_{10}$

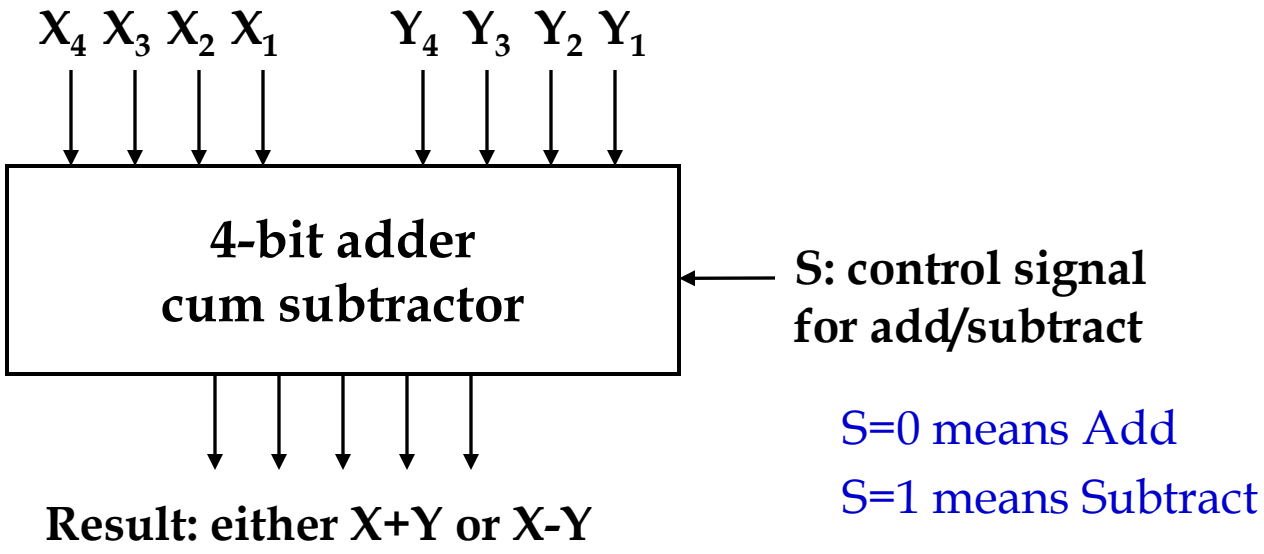
3-11 Binary Adder-Subtractors

Block-Level Design

- More complex circuits can also be built using block-level method.
- In general, block-level design method (as opposed to gate-level design) relies on algorithms or formulae of the circuit, which are obtained by decomposing the main problem to sub-problems recursively (until small enough to be directly solved by blocks of circuits).
- Simple examples using 4-bit parallel adder as building blocks:
  1. BCD-to-Excess-3 Code Converter
  2. 16-bit Parallel Adder
  3. Adder cum Subtractor

4-Bit Adder Cum Subtractor (1/3)

- Recall: Subtraction can be done via addition with 2s-complement numbers.
- Hence, we can design a circuit to perform both addition and subtraction, using a parallel adder and some gates.



4-Bit Adder Cum Subtractor (2/3)

- Recall:

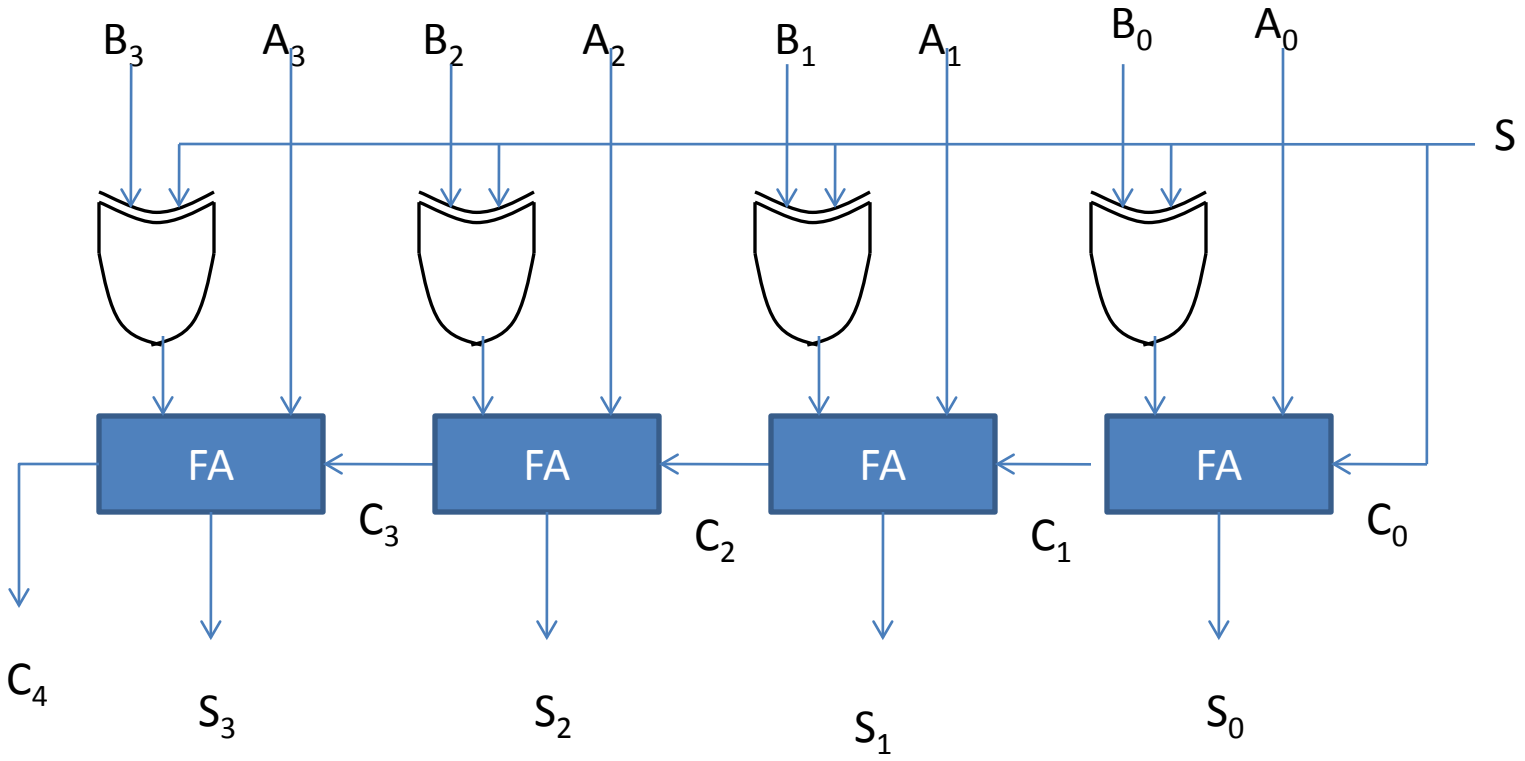
$$X - Y = X + (-Y)$$
$$= X + (\text{2s complement of } Y)$$
$$= X + (\text{1s complement of } Y) + 1$$

- Design requires:

(1) XOR gates, and (2) S connected to carry-in.

4-Bit Adder Cum Subtractor (3/3)

- 4-bit adder-cum-subtractor circuit:



Morris Mano: Fig 4-7