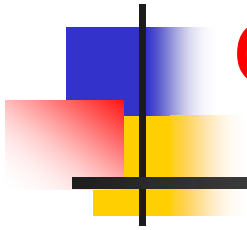
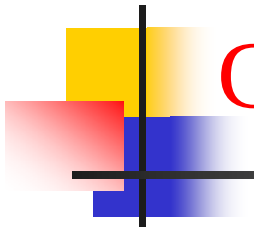


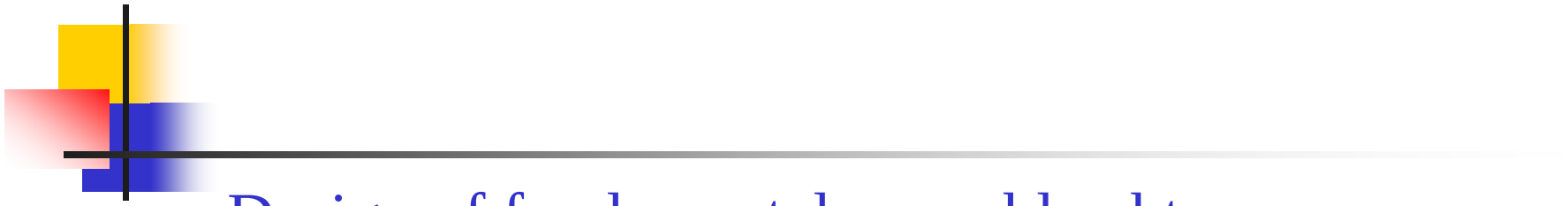
Chapter 4

Combinational Functions and Circuits

J.J. Shann



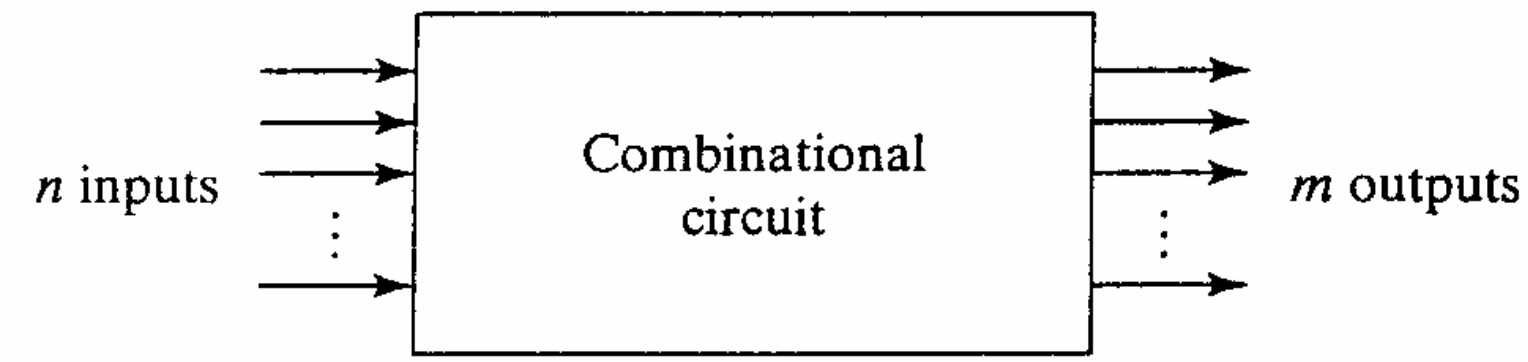
Chapter Overview



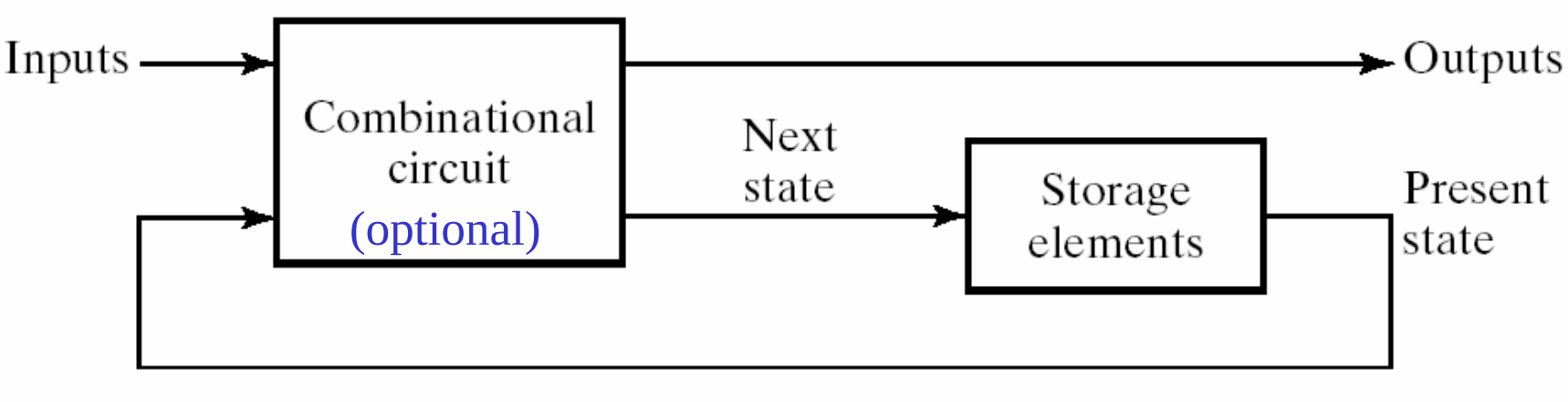
- Design of fundamental, reusable ckts:
  - called *functional blocks*
  - implementing functions of
    - a single variable,
    - decoders,
    - encoders,
    - code converters,
    - multiplexers, and
    - programmable logic

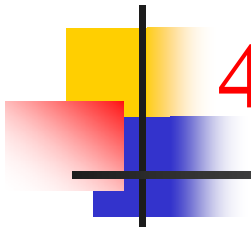
4-1 Combinational Circuits

- Logical circuits for digital systems:
  - Combinational ckts
  - Sequential ckts: (Ch6~)
- Combinational ckt: logic gates
  - It outputs at any time are determined from the present inputs. (no feedback paths or memory elements)



- Sequential ckt: storage elements & logic gates
  - It outputs are a function of the present inputs and the state of the storage elements.
    - The state of storage elements is a function of previous inputs.
  - The ckt behavior must be specified by a time sequence of inputs and internal states.





4-2 Rudimentary Logic Functions

- Most elementary combinational logic functions:
  - Value-fixing
  - Transferring
  - Inverting
  - Enabling

A. Value-Fixing, Transferring, & Inverting

- Single-bit function:
  - a function that has one-bit output
- Functions depend on a single variable at most:
  - 4 different functions are possible

Functions of One Variable

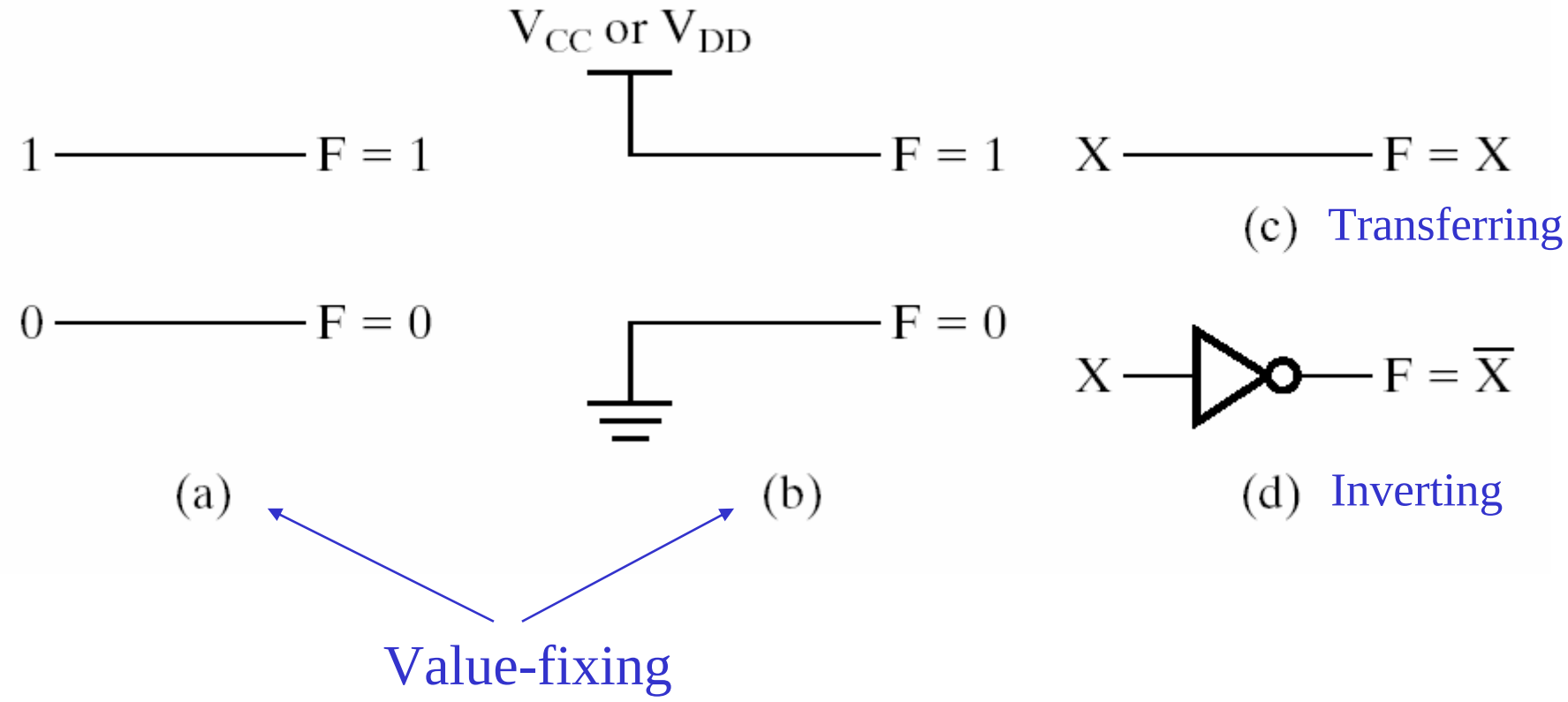
X	F = 0	F = X	F = $\bar{X}$	F = 1
0	0	0	1	1
1	0	1	0	1

value-fixing

transferring

inverting

- Implementation of functions of a single variable *X*:



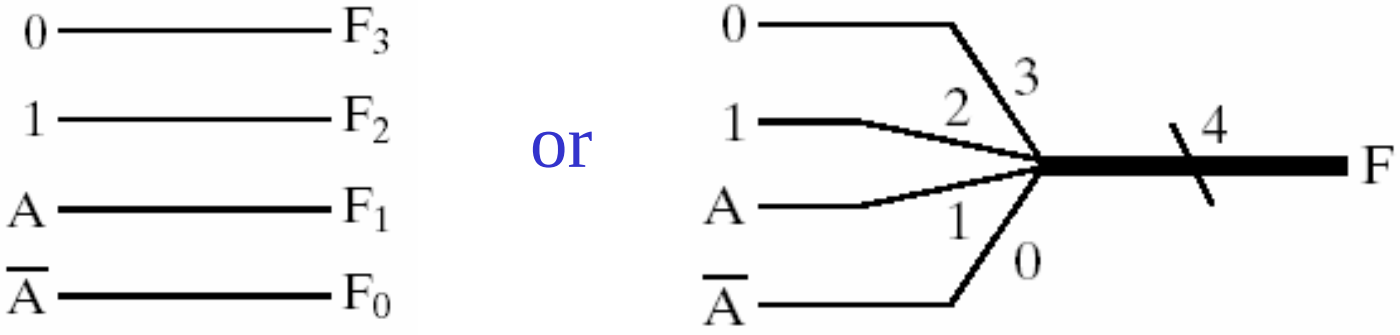
Multiple-Bit Functions

- Multiple-bit function: a vector of single bit functions
- Applying value fixing, transferring, and inverting to multiple bits on a bitwise basis:
  - E.g.: a four bit function F

$F = \mathrm{F}(3{:}0) = (F_3, F_2, F_1, F_0) = (0, 1, A, A')$

For $A = 0$, $F = (0, 1, 0, 1)$

For $A = 1$, $F = (0, 1, 1, 0)$



- Subset of a multiple-bit function:
  - E.g.s:

- Value-fixing, transferring and inverting have a variety of applications in logic design.

Value-Fixing for Implementing a Function

- Example 4-1: Value-fixing for implementing a function

$$Y(A, B, I_0, I_1, I_2, I_3)$$

A	B	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

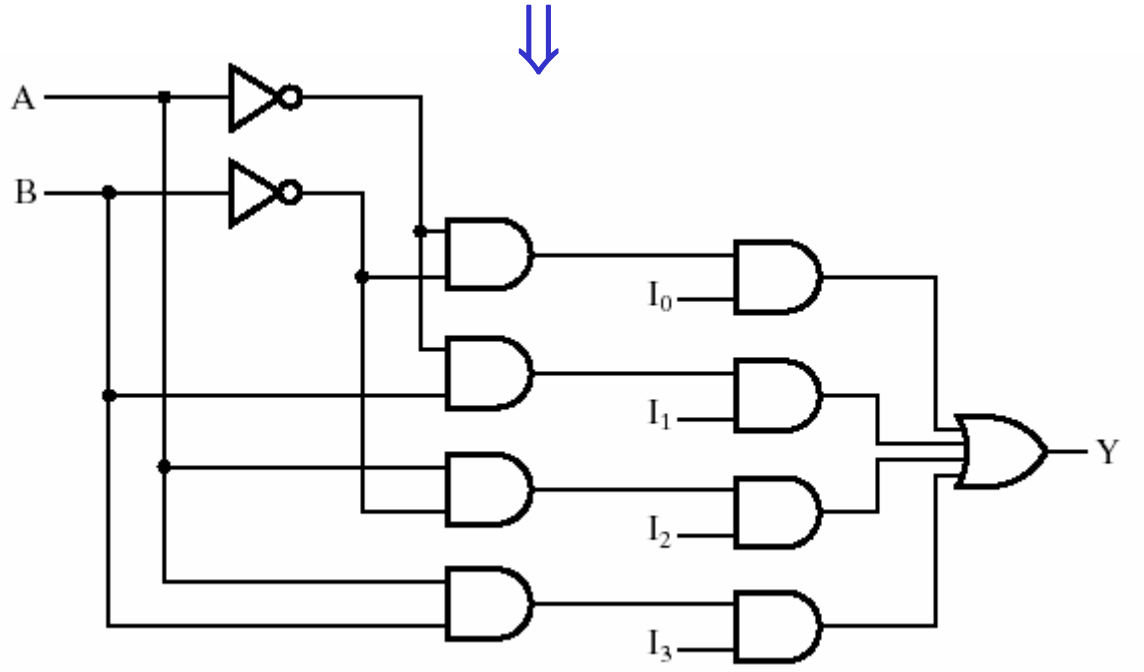
6 input variables

$\Rightarrow$ 64-row truth table (impractical)

<Ans.> Fix the values of I_0 through I_3

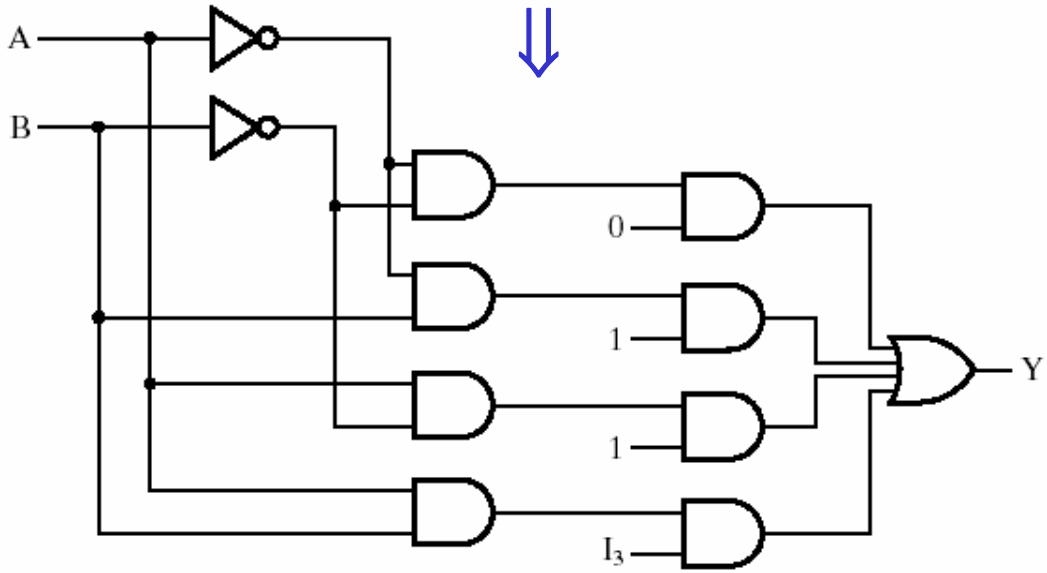
A	B	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

$$\Rightarrow \quad Y(A,B,I_0,I_1,I_2,I_3) = \overline{A}\,\overline{B}I_0 + \overline{A}BI_1 + A\overline{B}I_2 + ABI_3$$



- E.g.: Function implementation by value-fixing

A	B	Y = A + B	Y = A $\bar{B}$ + $\bar{A}$ B	Y = A + B (I_3 = 1) or Y = A $\bar{B}$ + $\bar{A}$ B (I_3 = 0)
0	0	0	0	0
0	1	1	1	1
1	0	1	1	1
1	1	1	0	I_3



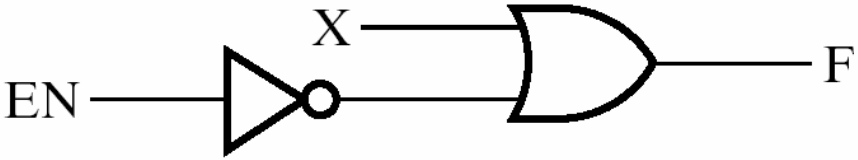
B. Enabling

- Enabling:
  - permits an input signal to pass through to an output
  - disabling can replace the input signal w/ a fixed output value, either 0 or 1
  - requires an additional input signal: *ENABLE* or *EN*
- Enabling ckt:

(a) EN = 1 (enable), F = X
= 0 (disable), F = 0

(a)

(b) EN = 1 (enable), F = X
= 0 (disable), F = 1



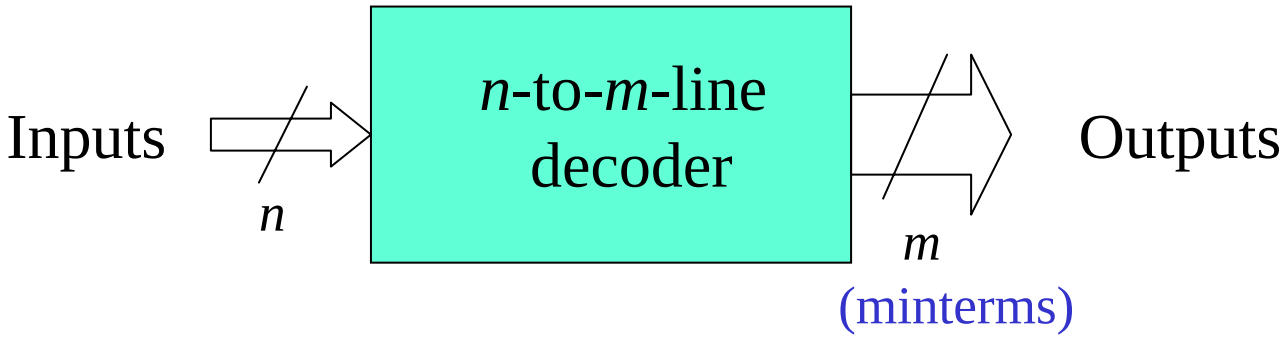
(b)

4-3 Decoder

- *n*-bit binary code:
  - is capable of representing up to 2^n distinct elements of coded information.
- Decoding:
  - the conversion of an *n*-bit input code to an *m*-bit output code w/ $n \leq m \leq 2^n$ s.t. each valid input code word produces a unique output code.
- Decoder:
  - a combinational ckt w/ an *n*-bit binary code applied to its inputs and an *m*-bit binary code appearing at the output, i.e., converts binary information from *n* input lines to a maximum of 2^n unique output lines
  - may have unused bit combinations on it inputs for which no corresponding m-bit code appears at the output.

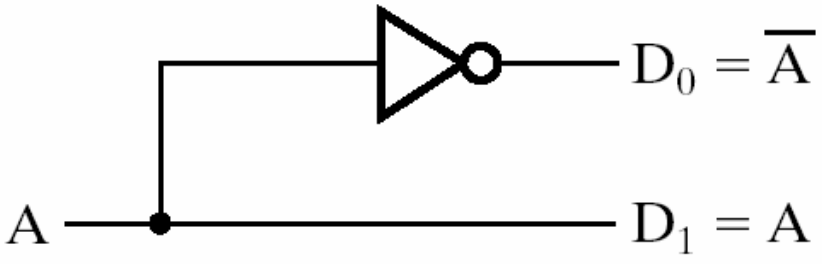
Line Decoder

- *n*-to-*m*-line decoder: $m \leq 2^n$
  - generate the 2^n (or fewer) minterms of *n* input variables



  - E.g.: a 1-to-2-line decoder

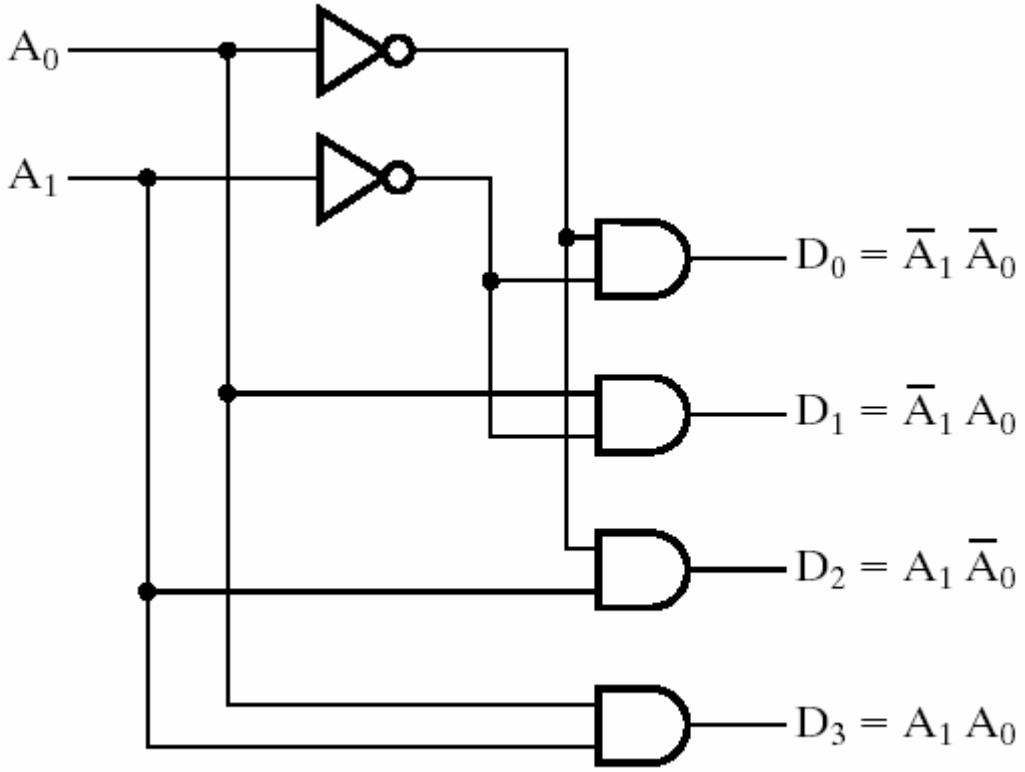
A	D_0	D_1
0	1	0
1	0	1



– E.g.: a 2-to-4-line decoder

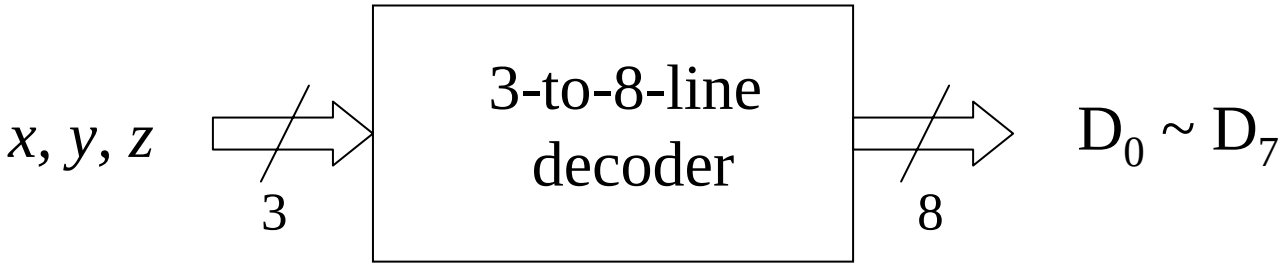
A_1	A_0	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

(a)



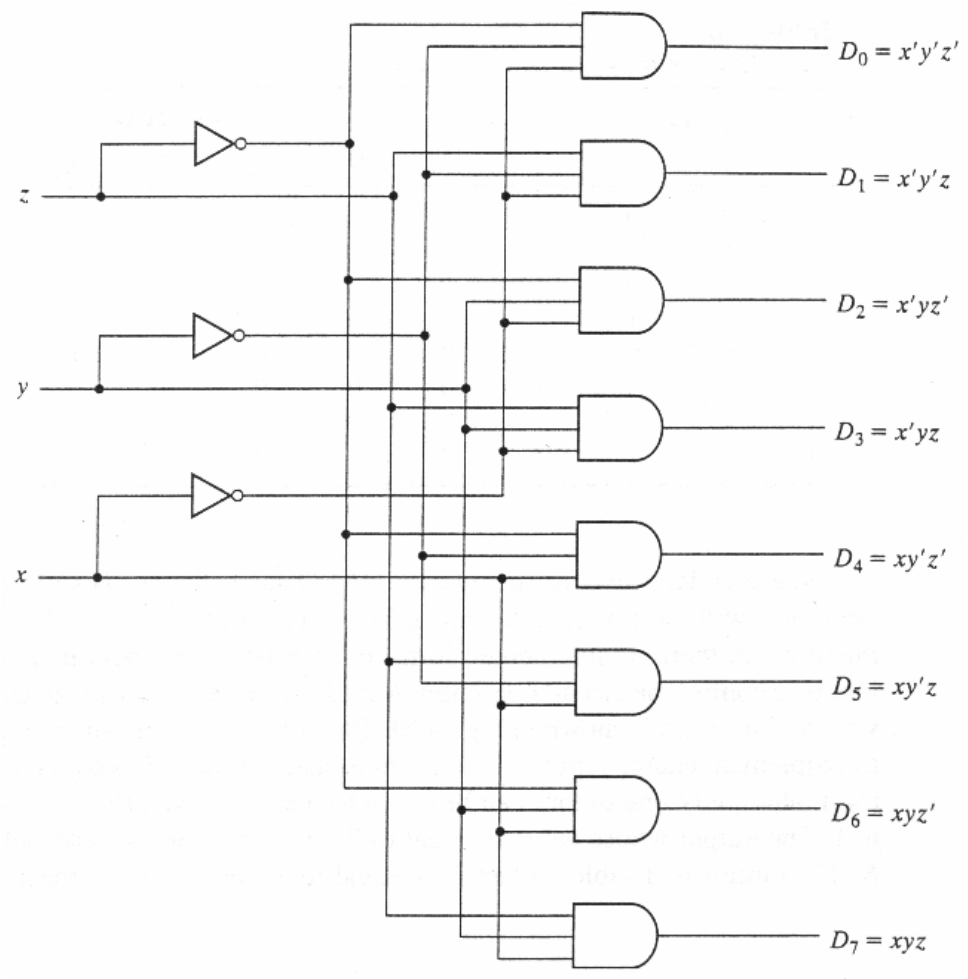
(b)

– E.g.: a 3-to-8-line decoder

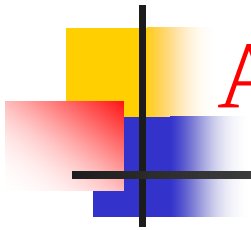


Truth Table of a 3-to-8-Line Decoder

Inputs			Outputs							
x	*y*	*z*	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_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



- Application: a binary-to-octal conversion



A. Decoder Expansion

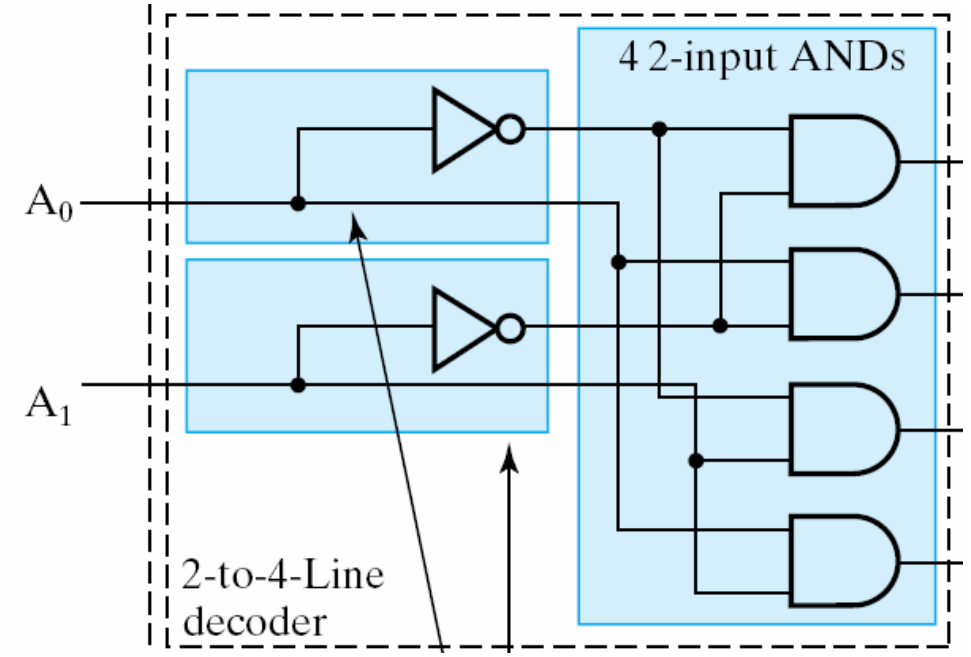
- Construction of larger decoder:
  - Approach 1: Enlarge each AND gate
    - implement each minterm function using a single AND gate w/ more inputs
    - Disadv.: high gate input count
  - Approach 2: Use design hierarchy and collections of AND gates
    - Adv.: The resulting decoder has the same or a lower gate input count than the one constructed by Approach 1.

- E.g.: Construct a 2-to-4 line decoder (n = 2)

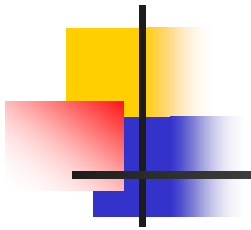
<Ans.>

Use 2 1-to-2-line decoders feeding

4 2-input AND gates.



1-to-2 line decoders



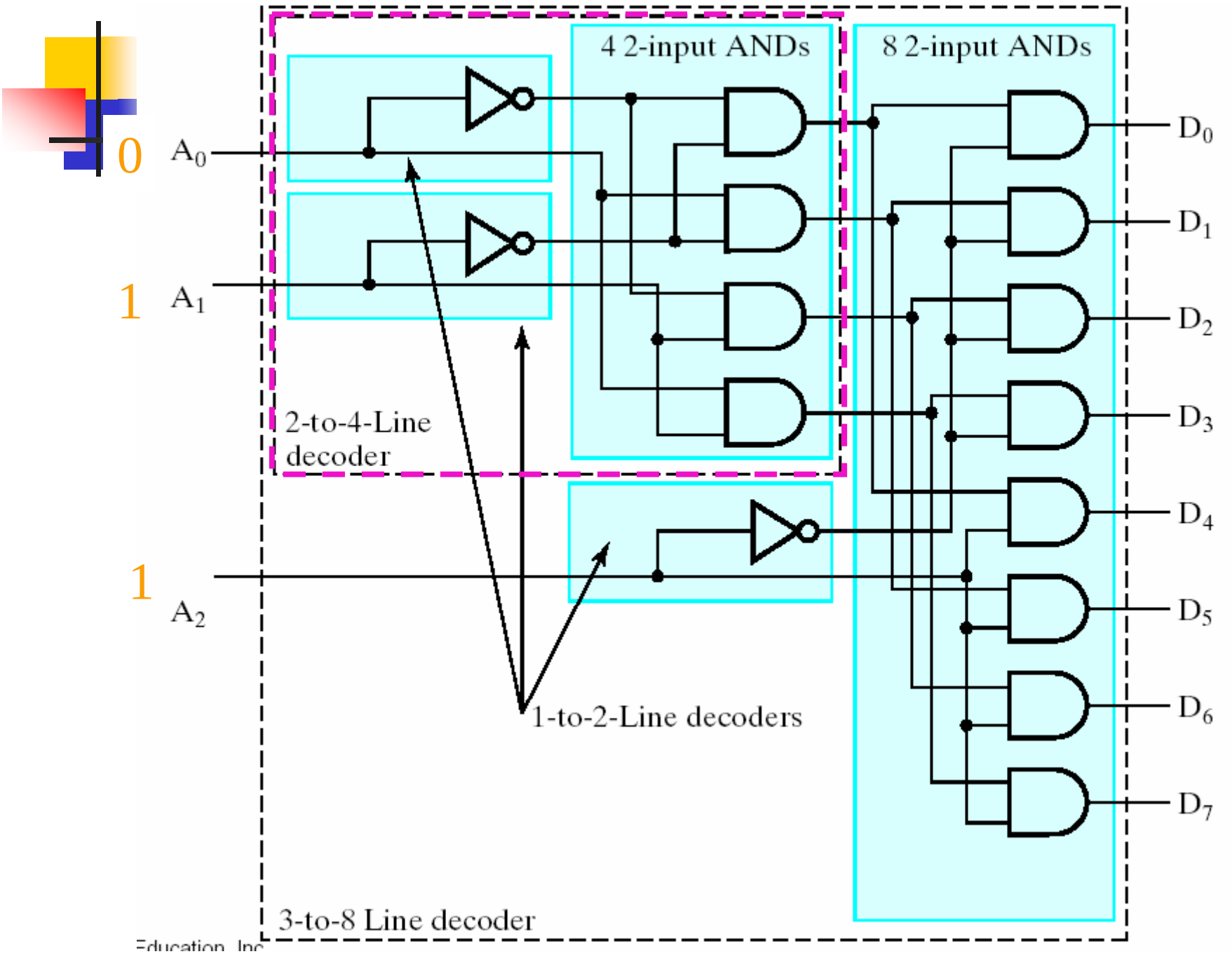
- E.g.: Construct a 3-to-8 line decoder (n = 3)

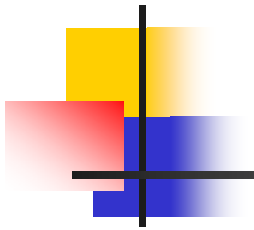
  <Ans.>

  Use a 2-to-4-line decoder and

  a 1-to-2-line decoder feeding

  8 2-input AND gates to form the minterms.


4 2-input ANDs
8 2-input ANDs
0
A_0
1
A_1
1
A_2
D_0
D_1
D_2
D_3
D_4
D_5
D_6
D_7
2-to-4-Line decoder
1-to-2-Line decoders
3-to-8 Line decoder



General procedure:

1. Let $k = n$.
2. If k is even, divide k by two to obtain $k/2$. Use 2^k AND gates driven by 2 decoders of output size $2^{k/2}$.

   If k is odd, obtain $(k+1)/2$ and $(k-1)/2$. Use 2^k AND gates driven by a decoder of output size $2^{(k+1)/2}$ and a decoder of output size $2^{(k-1)/2}$.
3. For each decoder resulting from step 2, repeat step 2 with k equal to the values obtained in step 2 until $k = 1$. For $k = 1$, use a 1-to-2 decoder.

Example 4-3

- E.g.: 6-to-64-line decoder

$k = 6$

$\Rightarrow$ 2 3-to-8 decoders + 64 ANDs

$k' = 3$

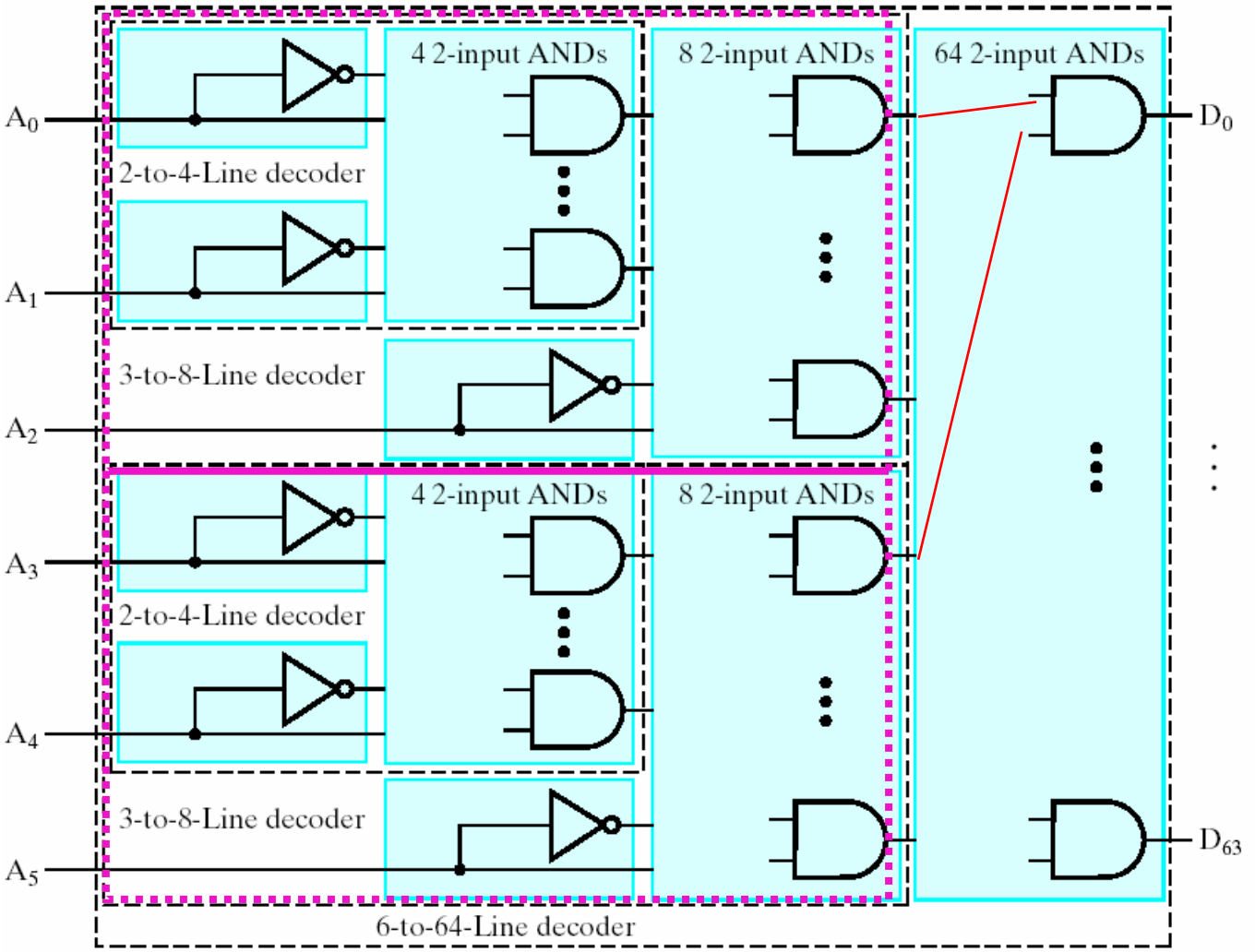
$\Rightarrow$ a 2-to-4 decoders + a 1-to-2 decoders + 8 ANDs

$k'' = 2$

$\Rightarrow$ 2 1-to-2 decoders

<Ans.> 6-to-64-line decoder

$k = 6$
⇒ 2 3-to-8 decoders + 64 ANDs
$k' = 3$
⇒ a 2-to-4 decoders + a 1-to-2 decoders + 8 ANDs
$k'' = 2$
⇒ 2 1-to-2 decoders



Sharing of Decoders

- Sharing of decoders:
  - If multiple decoders are needed and the decoders have common input variables $\Rightarrow$ Parts of the decoders can be shared.
  - E.g.: Suppose that 3 decoders d_a, d_b, and d_c are functions of input variables as follows

$$d_a(A,B,C,D)$$
$$d_b(A,B,C,E) \quad \Rightarrow \quad \text{Sharing of decoders?}$$
$$d_c(C,D,E,F)$$

Sharing of Decoders

<Ans.>

$d_a(A,B,C,D)$
$d_b(A,B,C,E)$
$d_c(C,D,E,F)$

$\Rightarrow$

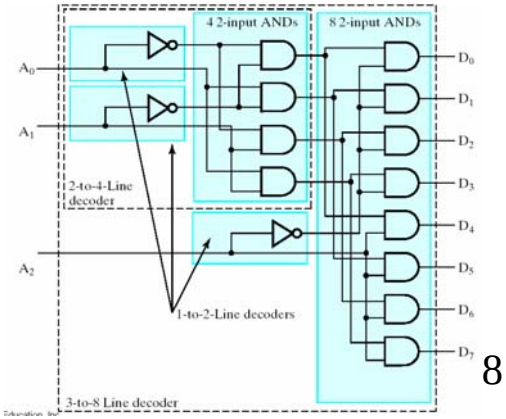
- (A,B,C) *shared by* d_a *and* d_b,
- (C,D) *shared by* d_a *and* d_c
- (C,E) *shared by* d_b *and* d_c

Consider the following cases:

1. (A,B) *shared by* d_a *and* d_b, *and* (C,D) *shared by* d_a *and* d_c
2. (A,B) *shared by* d_a *and* d_b, *and* (C,E) *shared by* d_b *and* d_c, *or*
3. (A,B,C) *shared by* d_a *and* d_b.

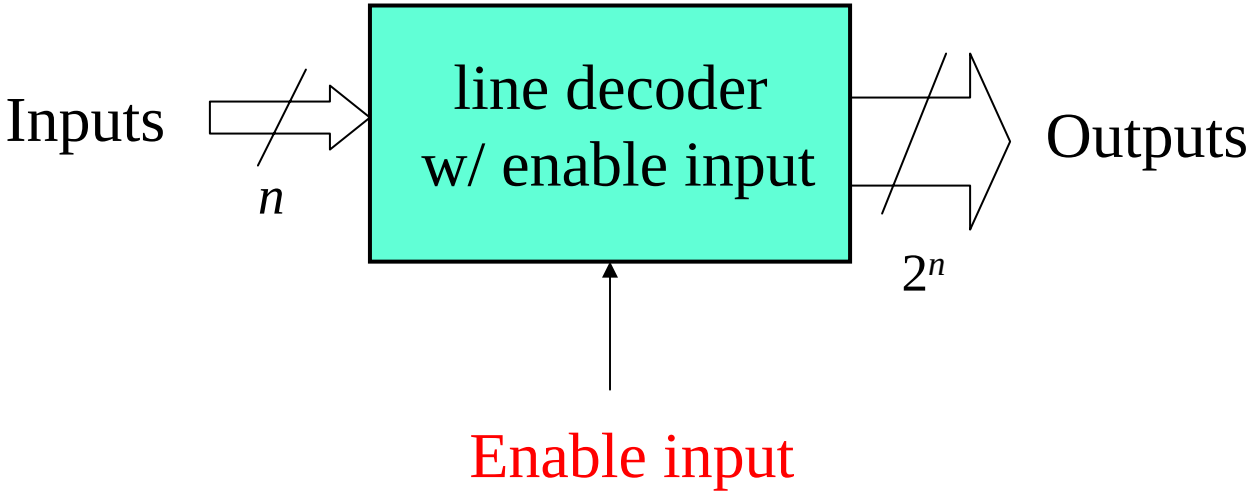
$\Rightarrow$ 1, 2: 16 gate inputs reduced

3: 24 gate inputs reduced (✓)



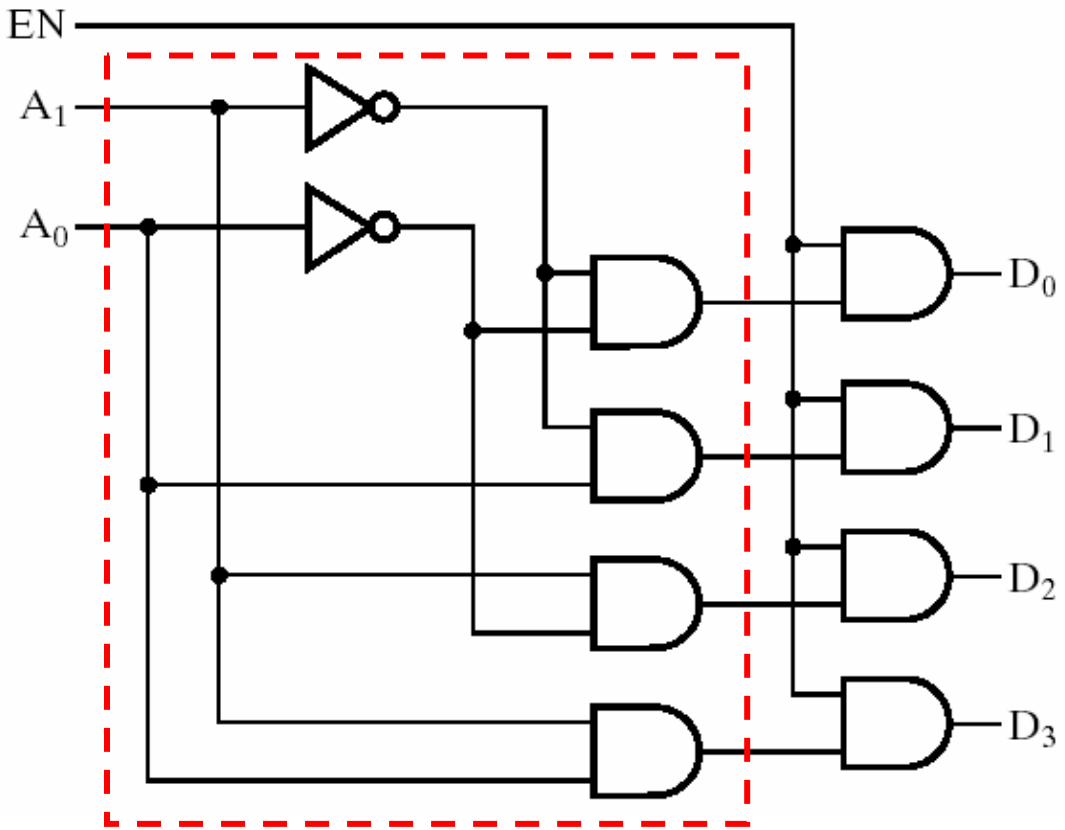
B. Decoder and Enabling Combinations

- Line decoder w/ enable input



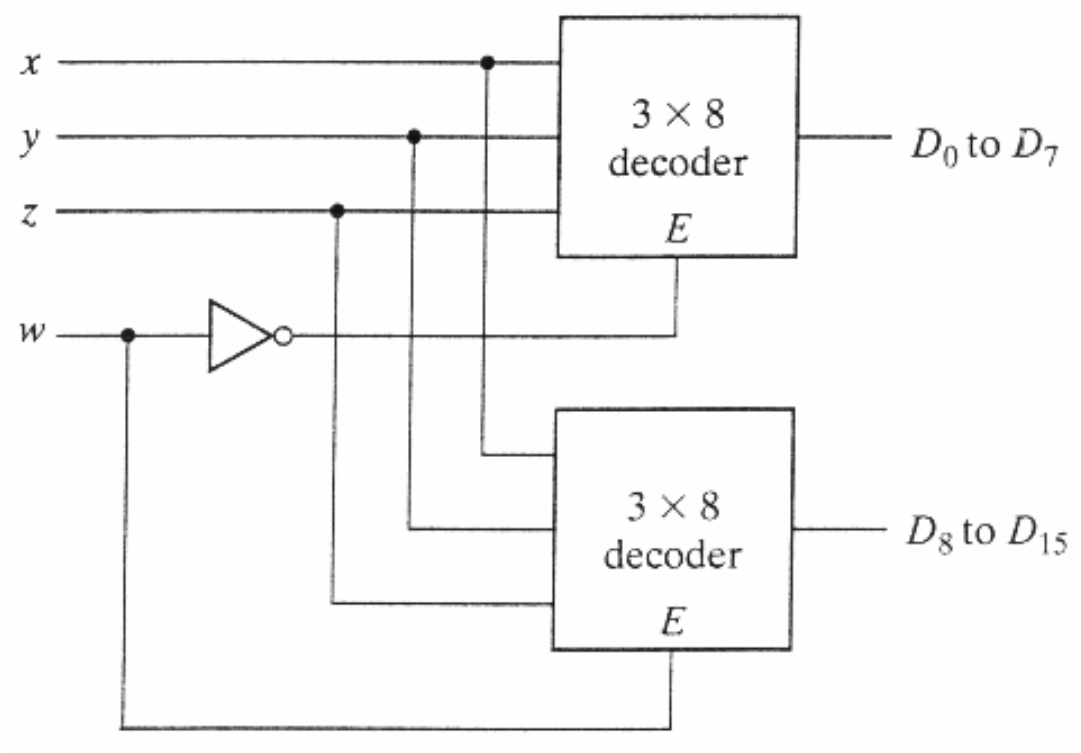
- E.g.: 2-to-4-line decoder w/ enable input

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



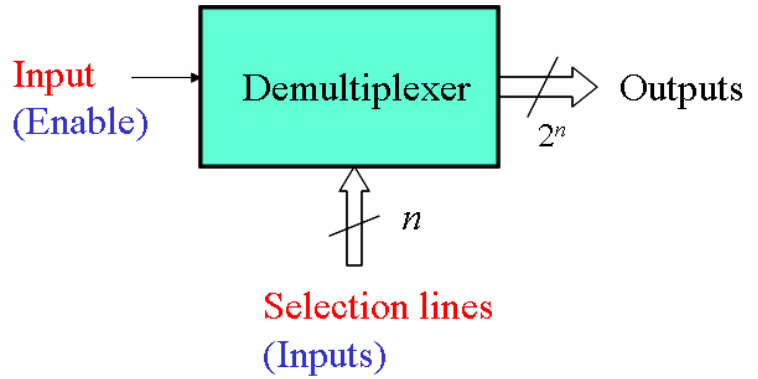
2-to-4 line decoder

- Construction of larger decoder:
  - Decoders w/ enable inputs can be connected together to form a larger decoder ckt.
- E.g.: 2 3-to-8-line decoders ⇒ a 4-to-16-line decoder

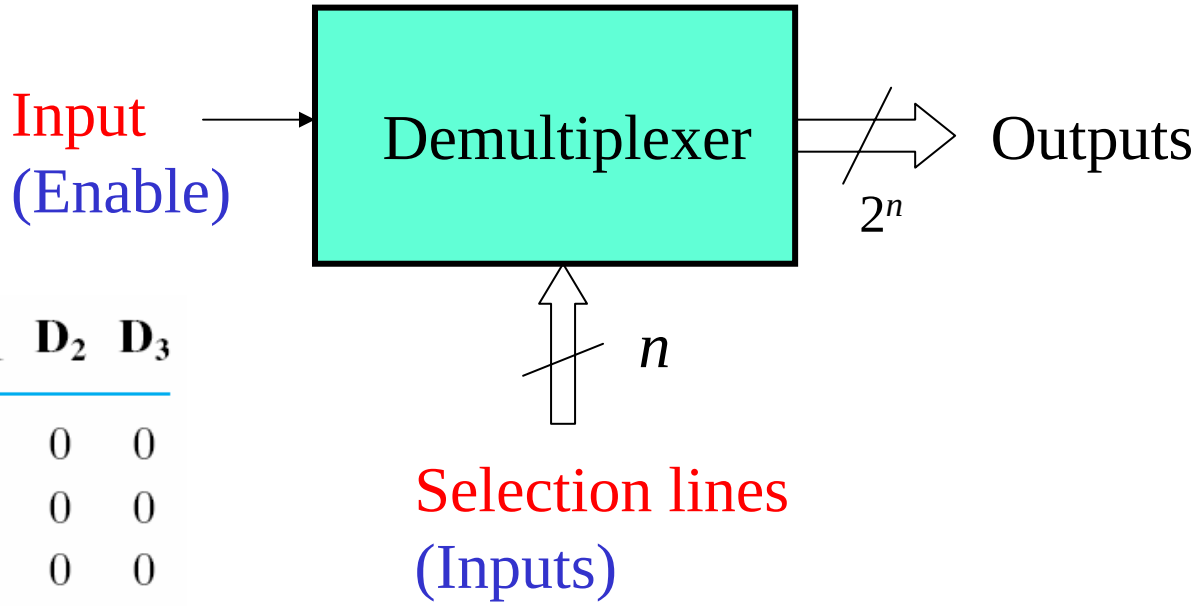


- Demultiplexer: ⇔ a line decoder w/ enable input
  - a line decoder w/ enable input:

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



- Demultiplexer: ⇔ a line decoder w/ enable input
  - a ckt that receives information from a single line and directs it to one of 2^n possible output lines according to the bit combination of n selection lines.



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

4-4 Encoding

- Encoder
  - the inverse function of a decoder
  - Has 2^n (or fewer) input lines & n output lines.
  - The output lines generate the binary code corresponding to the input value.
- Example: an octal-to-binary encoder

Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

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

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

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

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

- limitation of the encoder:

  One and only one input can be active at any given time

  - When more than one input are active simultaneously

    E.g.: illegal input $D_3 = D_6 = 1$

    the output = 111 (≡ the output when D_7 is equal to 1)

    ⇒ Priority encoder

  - When all the inputs are 0:

    the output is 000 (≡ the output when D_0 is equal to 1)

    ⇒ Provide one more output to indicate that at least one input is equal to 1

    (Valid-output indicator)

A. Priority Encoder

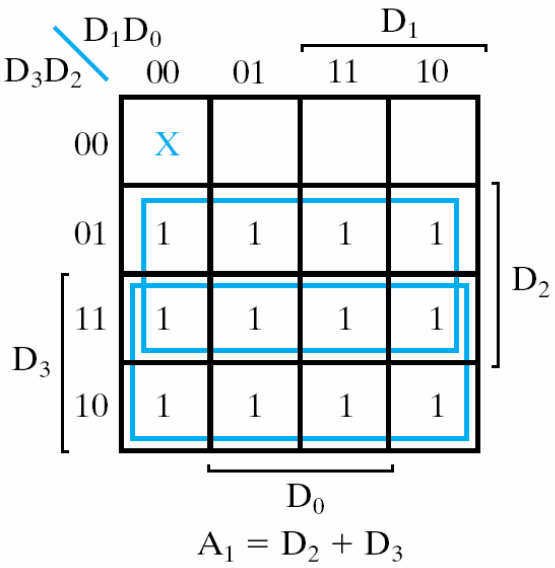
- Priority Encoder:
  - An encoder ckt that includes the priority function,
    i.e., if two or more inputs are equal to 1 at the same time,
    the input having the highest priority will take precedence.
- Example:

  Priority: $D_3 > D_2 > D_1 > D_0$, V: valid-output indicator

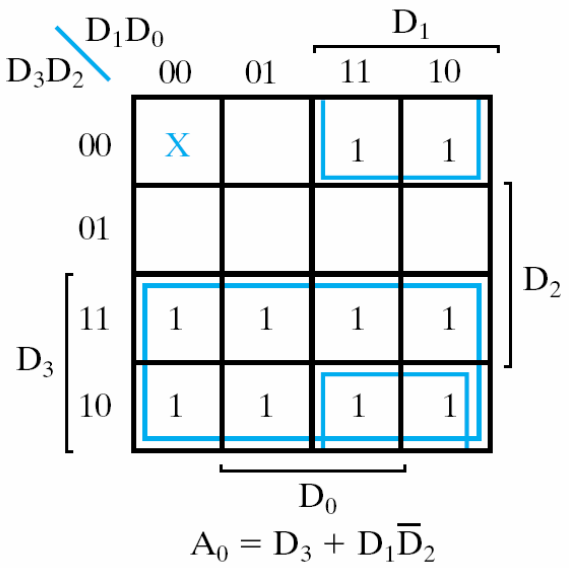
Inputs				Outputs		
D_3	D_2	D_1	D_0	A_1	A_0	V
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

×: don't-care condition

Inputs				Outputs		
D_3	D_2	D_1	D_0	A_1	A_0	V
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1



$A_1 = D_2 + D_3$

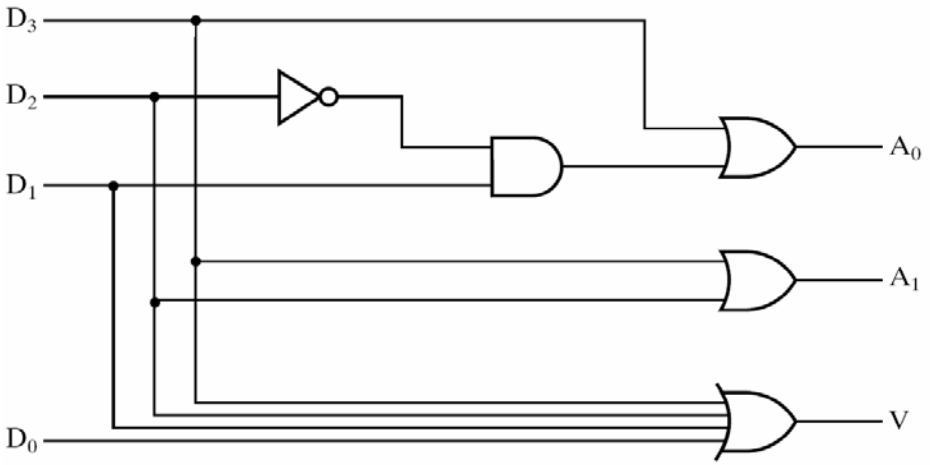


$A_0 = D_3 + D_1\overline{D}_2$

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

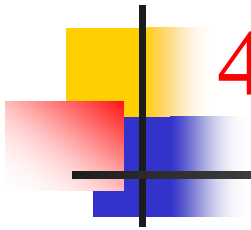
$$A_1 = D_2 + D_3$$

$$V = D_0 + D_1 + D_2 + D_3$$



B. Encoder Expansion

- Construction of larger encoder:
  - Encoders can be expanded to larger # of inputs by expanding OR gates.

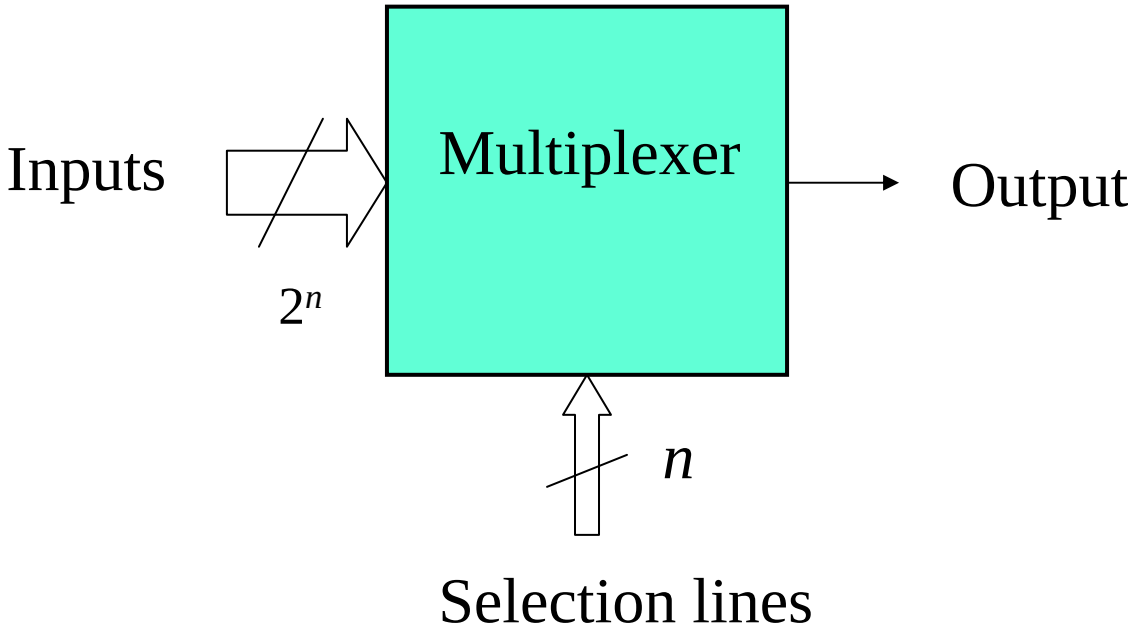


4-5 Selecting

- Selection ckts:
  - typically have a set of inputs from which selections are made, a single output, and a set of control lines for making the selection.
- Implementation of selection ckts:
  - Multiplexers
  - using three-state drivers or transmission gates

A. Multiplexers

- Multiplexer (MUX): Data selector
  - a combinational ckt that selects binary information from one of many input lines & directs it to a single output line.
  - 2^n input lines, n selection lines, and one output line



Example: 2-to-1-line MUX

- E.g.: 2-to-1-line MUX

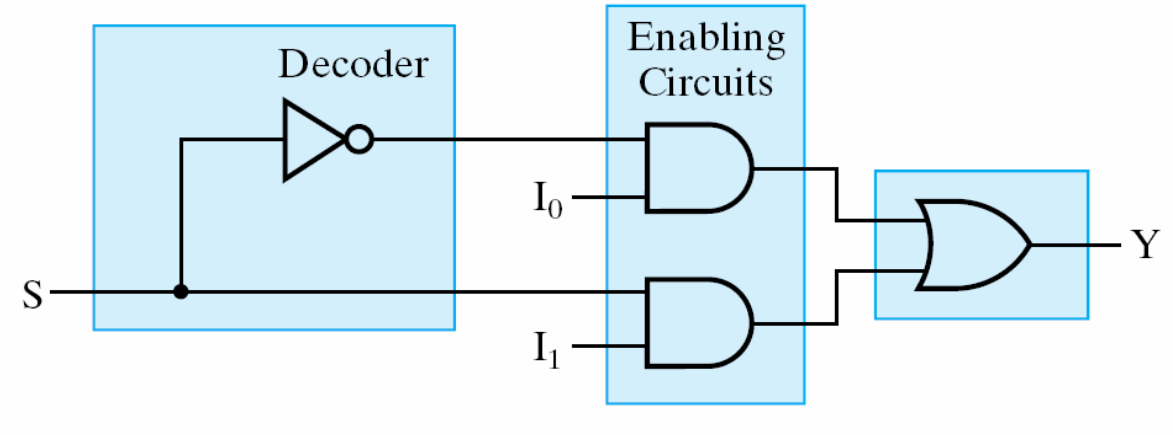
Truth table

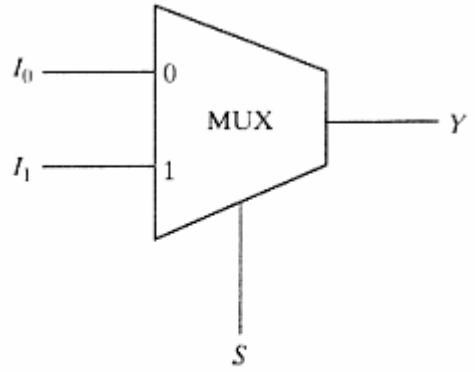
S	I_0	I_1	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Condensed truth table

S	Y
0	I_0
1	I_1

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





Block diagram

Example: 4-to-1-line MUX

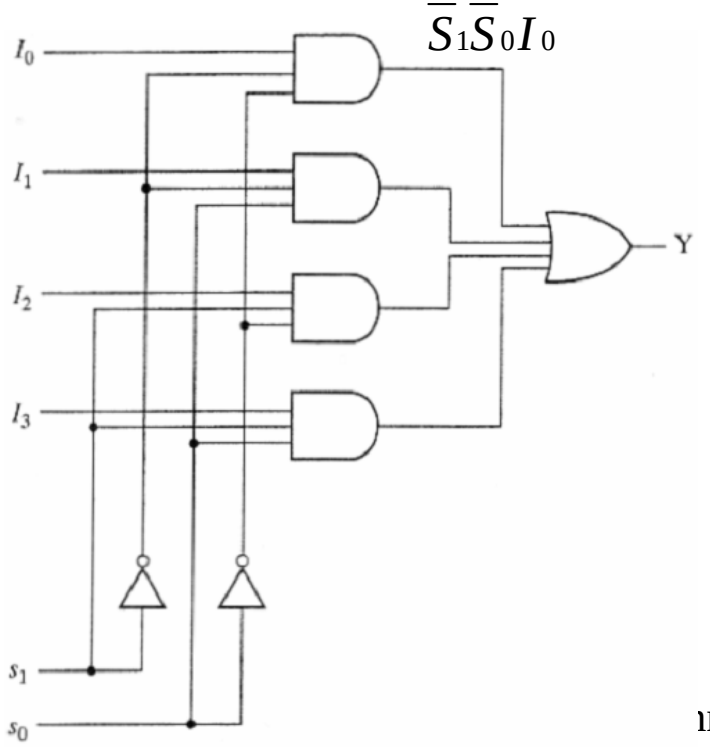
- E.g.: 4-to-1-line MUX

S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

<Approach 1>

$$Y = \overline{S}_1\overline{S}_0 I_0 + \overline{S}_1 S_0 I_1 + S_1\overline{S}_0 I_2 + S_1 S_0 I_3$$

GIC = 18 (including inverter inputs)



<Approach 2>

$$Y = \overline{S}_1\overline{S}_0 I_0 + \overline{S}_1 S_0 I_1 + S_1\overline{S}_0 I_2 + S_1 S_0 I_3$$

$$= (\overline{S}_1\overline{S}_0) I_0 + (\overline{S}_1 S_0) I_1 + (S_1\overline{S}_0) I_2 + (S_1 S_0) I_3$$

$\Rightarrow$ a 2-to-4 decoder

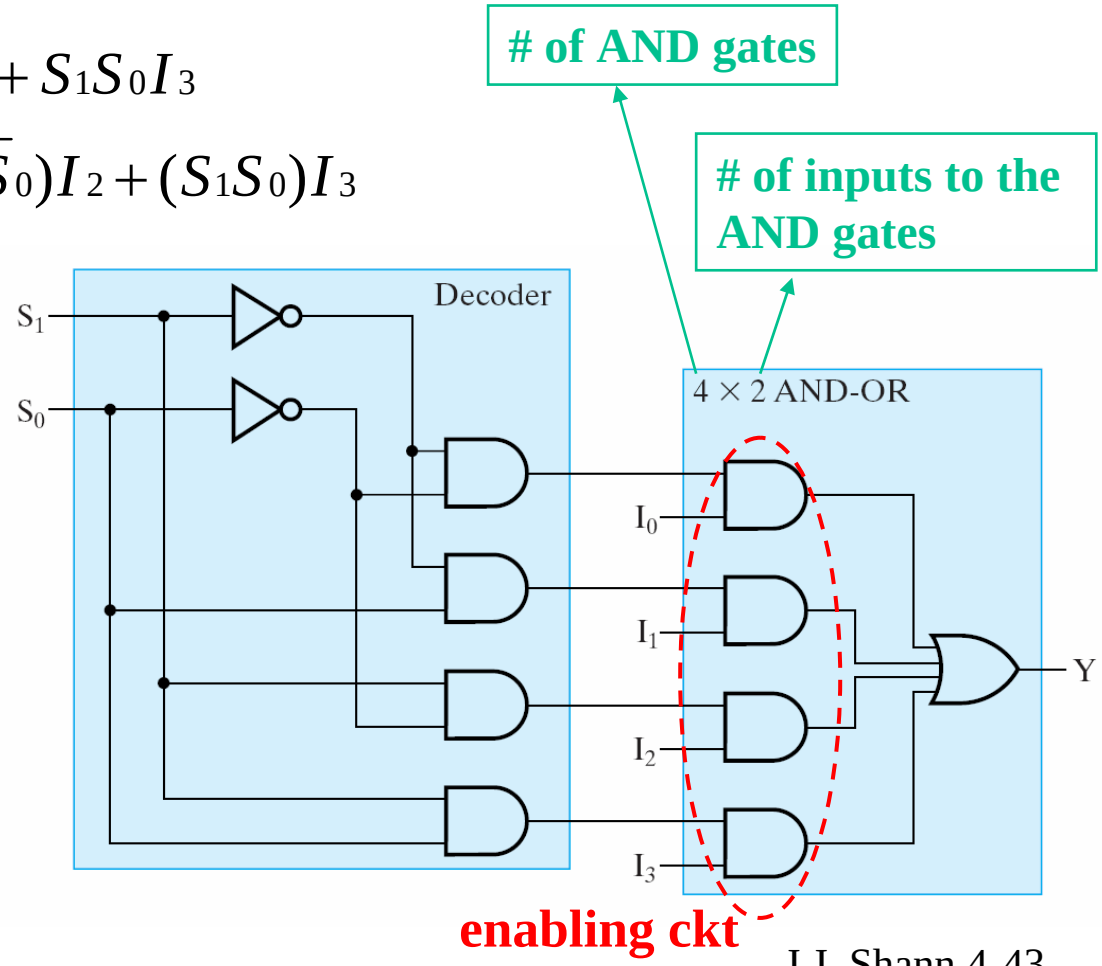
+ 4 AND gates (enabling ckt)

+ a 4-input OR gates

$\Rightarrow$ GIC = 22

* is more costly but

provides a structural basis for constructing larger n-to-2^n-line multiplexers by expansion.



B. Multiplexer Expansion

- Multiplexer expansion:
  - is based upon the ckt structure consists of (Approach 2)

    a decoder,

    enabling ckts, and

    an OR gate.

Example 4-4

- E.g.: 64-to-1-line multiplexer

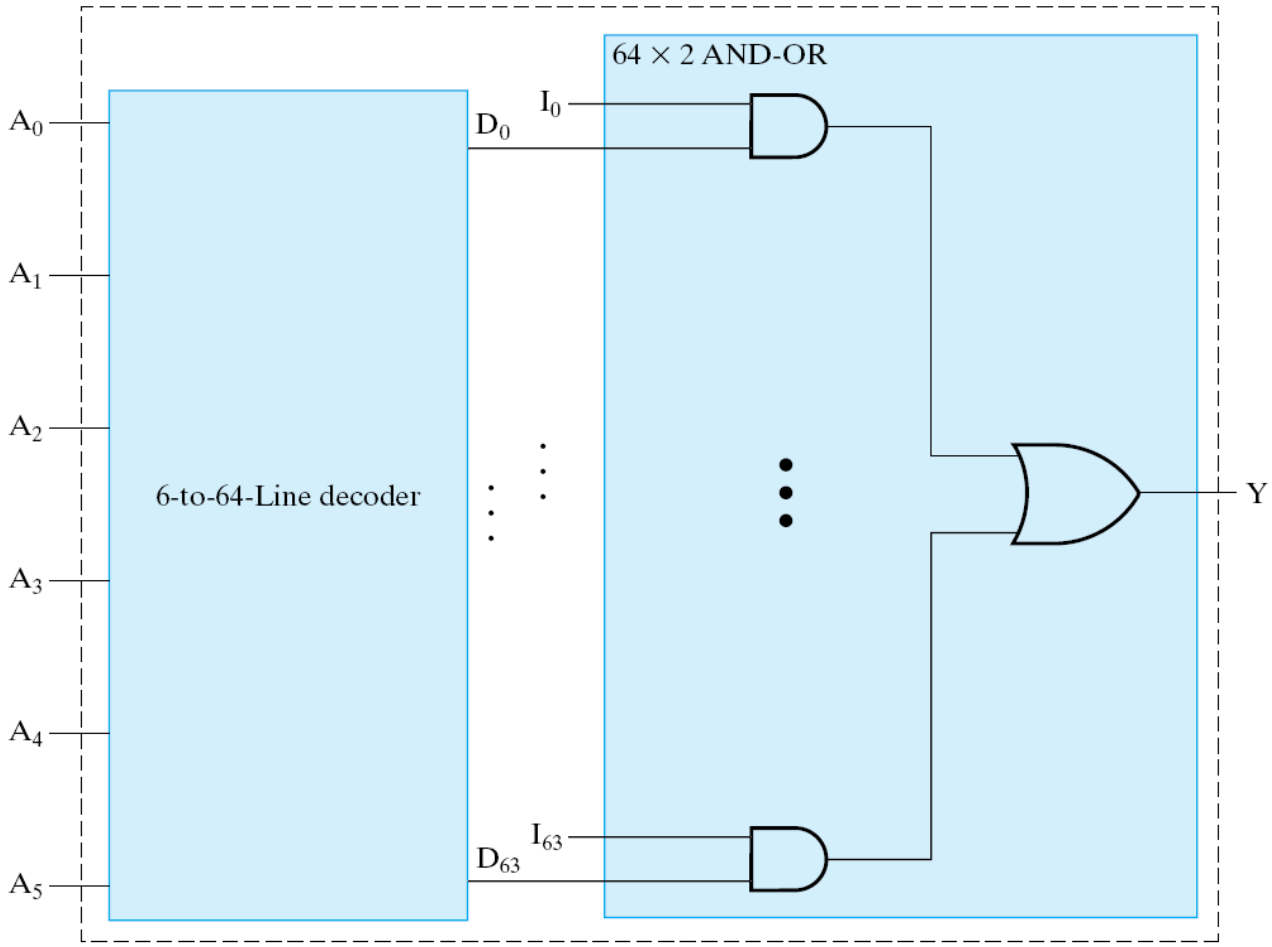
<Ans.>

$n = 6$

$\Rightarrow$ a 6-to-64-line decoder

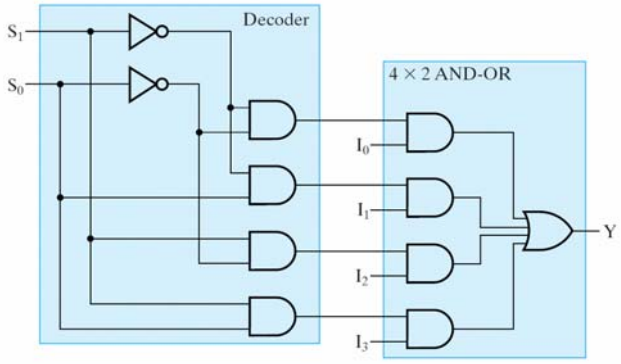
+

a 64×2 AND-OR gate



Example 4-5

4-to-1-line MUX

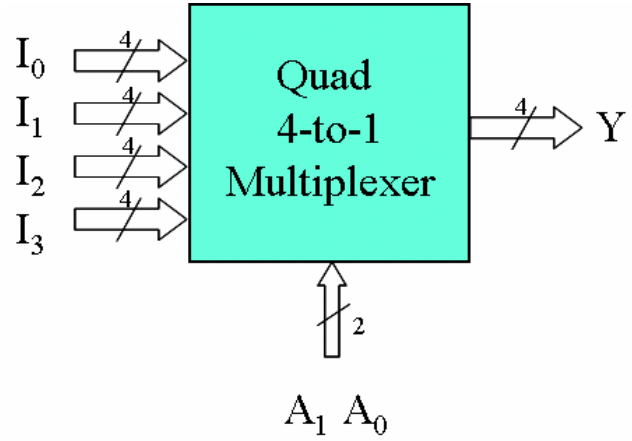


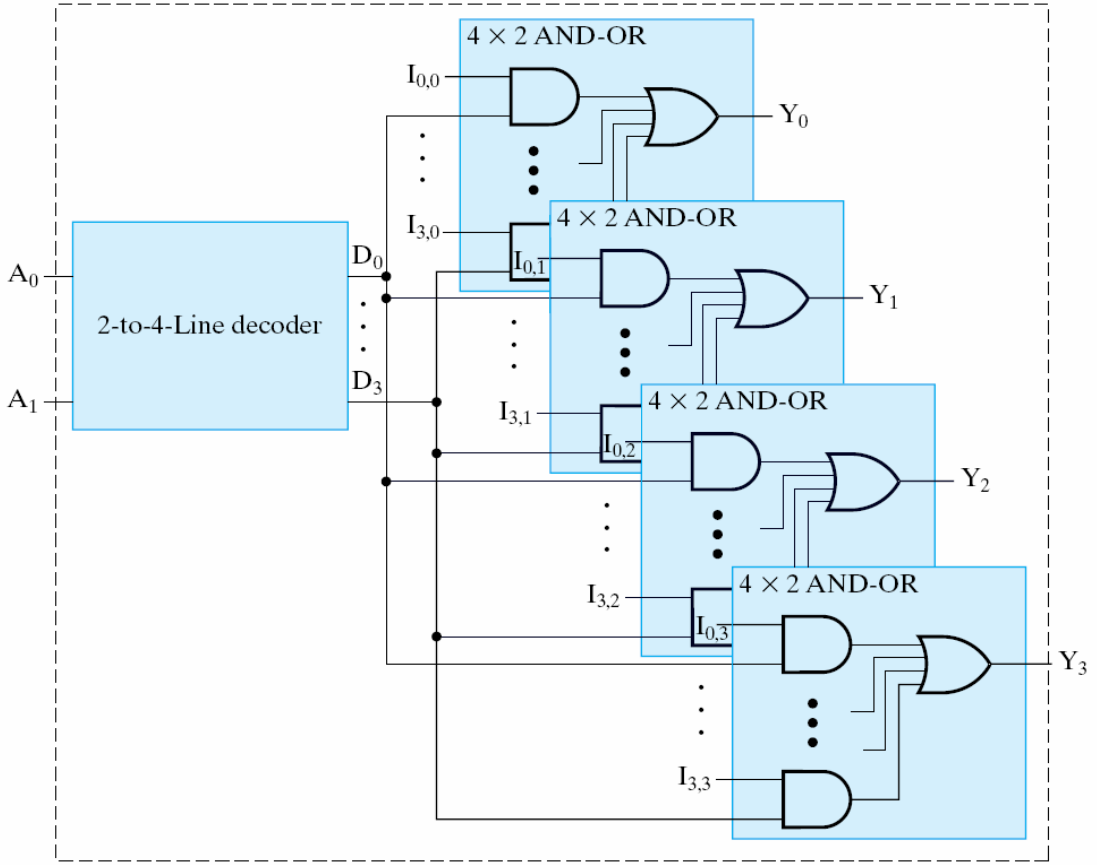
- E.g.: 4-to-1-line quad MUX
  - a quad 4-to-1 MUX: has 2 selection inputs and each information input replaced by a vector of 4 inputs

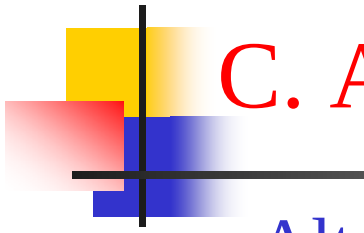
a 2-to-4-line decoder

+

4 4x2 AND-OR gates





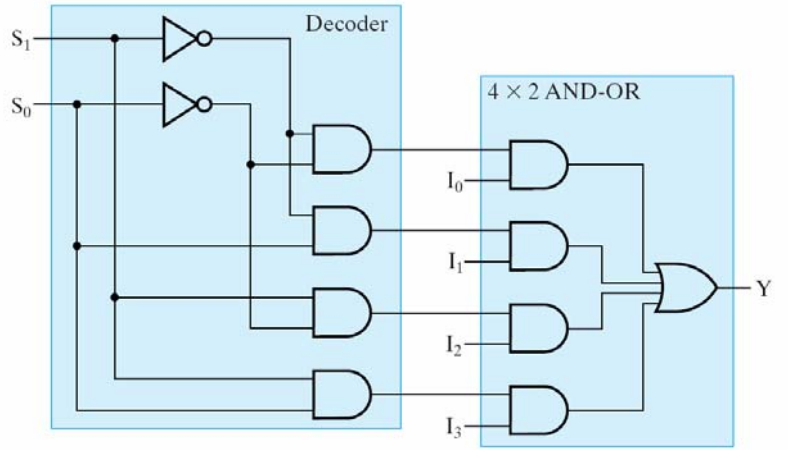


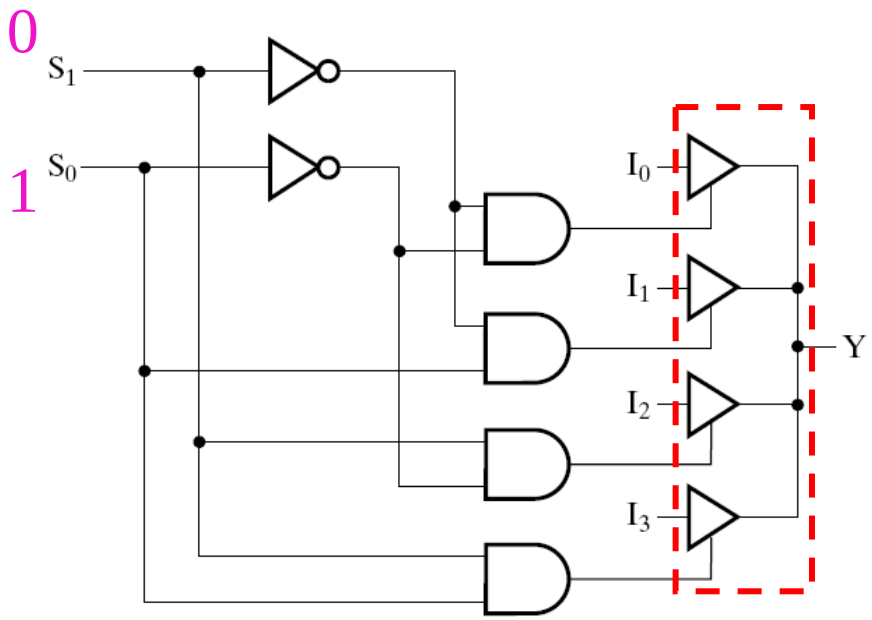
C. Alternative Selection Implementation

- Alternative selection implementation:
  - Three-state implementation
  - Transmission gate implementation
  * may have even lower cost than is achievable w/ gates.

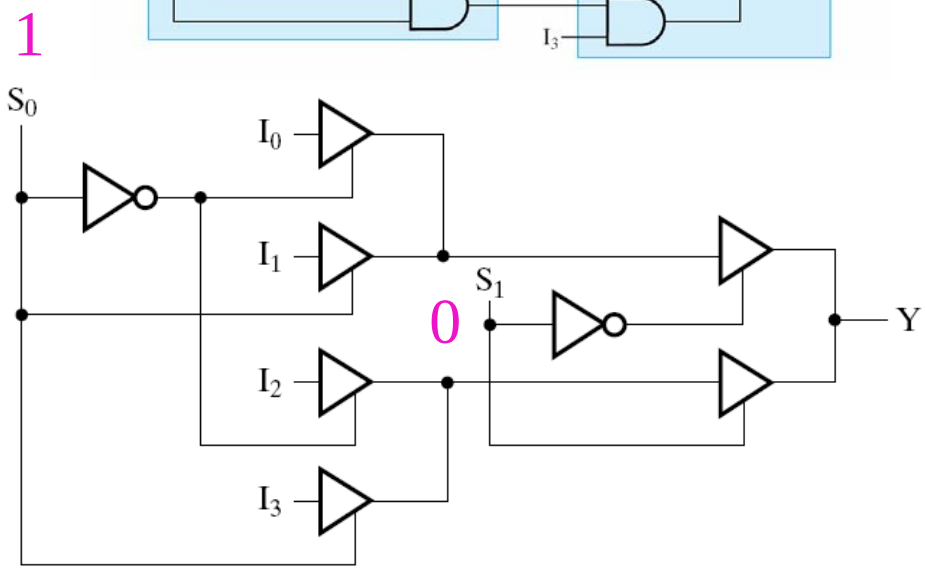
Three-State Implementation

- E.g.: a 4-to-1-line MUX
  using 3-state drivers





GIC = 18

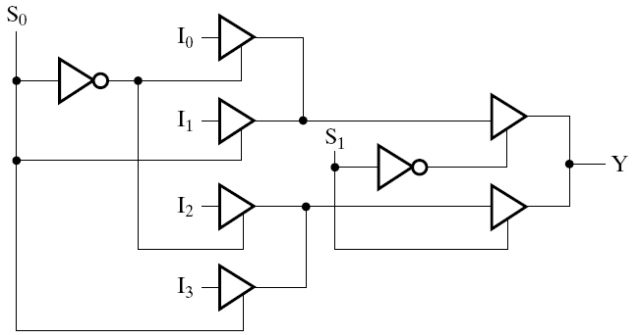


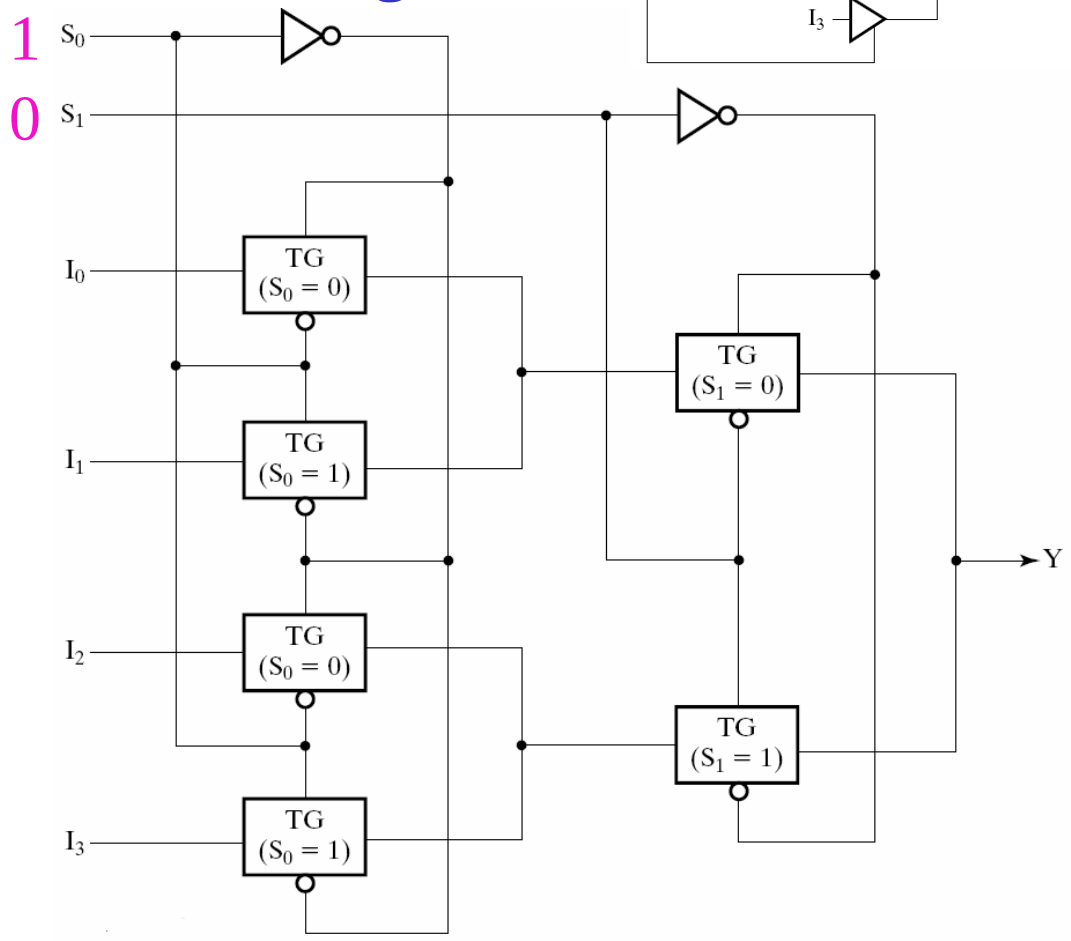
GIC = 14

Transmission Gate Implementation

- E.g.: a 4-to-1-line MUX

  using transmission gates





GIC = 8

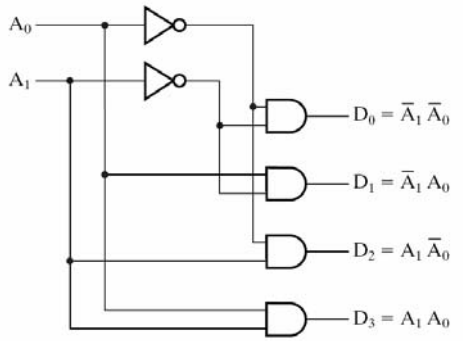
* The cost of a transmission gate = a GIC of 1

4-6 Combinational Function Implementation

- Implementing combinational logic functions using:
  - Decoders
  - Multiplexers (MUXs)
  - Read-only memories (ROMs)
  - Programmable logic arrays (PLAs)
  - Programmable array logics (PALs)
  - Lookup tables (LUTs)

§3-6

A. Using Decoders



- Combinational logic implementation
  - A line-decoder provides the 2^n minterms of n input variables.
    - each output = a minterm (active-HIGH outputs)
  - Any Boolean function of n input variables

    → Sum of minterms expression

    → Use an n-to-2^n-line decoder to generate the minterms & an external OR gate to form the logical sum
  - Any combinational ckt w/ n inputs and m outputs

    → Use an n-to-2^n-line decoder & m OR gates

Example 4-6

- E.g.: Decoder and OR gate implementation of a binary adder bit (Full adder)

<Ans.>

3 inputs:

X, Y (the two bits being added) ,

Z (the incoming carry from the right)

2 outputs:

S (the sum bit)

C (the carry bit)

$\Rightarrow$ Need a 3-to-8-line decoder and 2 OR gates.

Truth Table for 1-bit Binary Adder

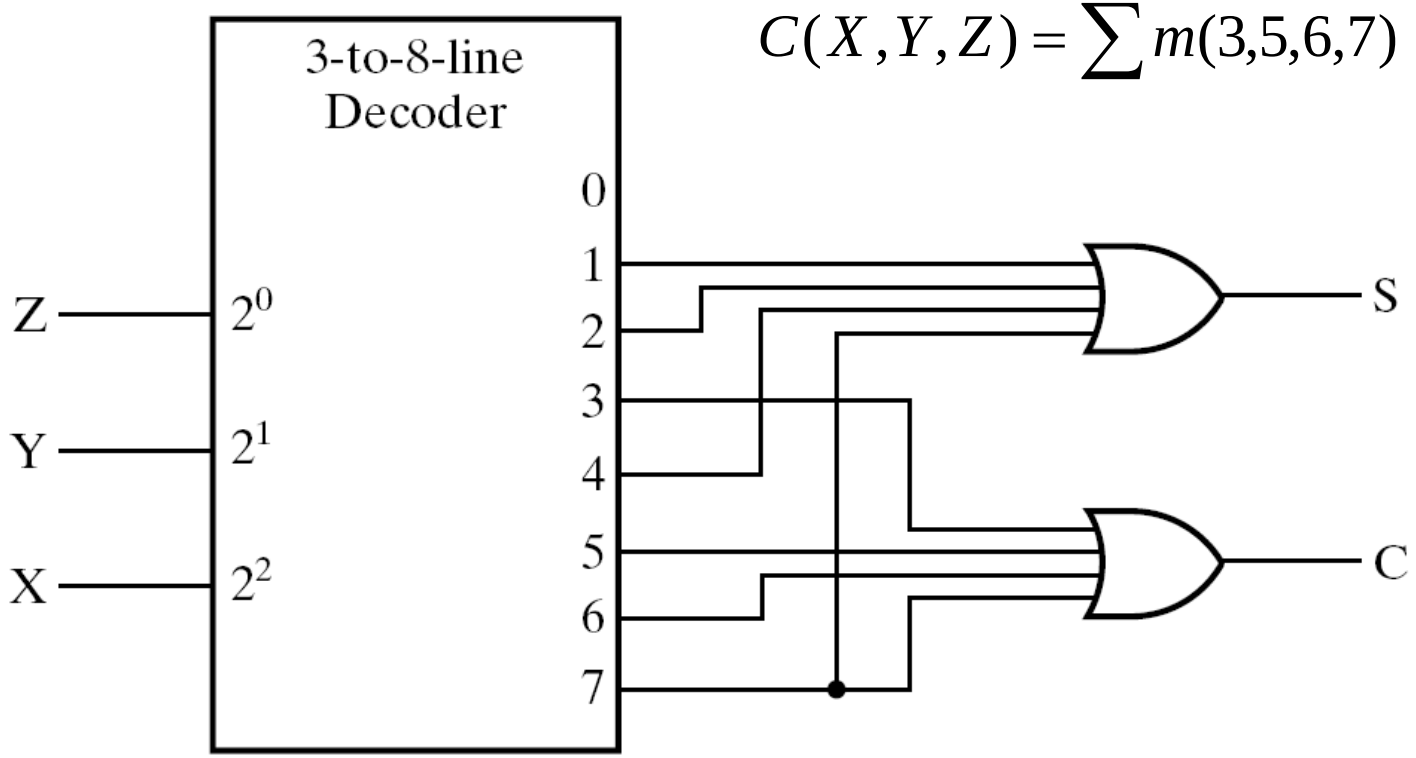
X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

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

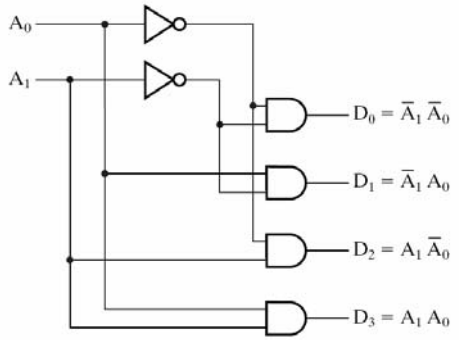
$$C(X,Y,Z) = \sum m(3,5,6,7)$$

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

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



Possible approaches using decoder:



- For decoder w/ active HIGH outputs:

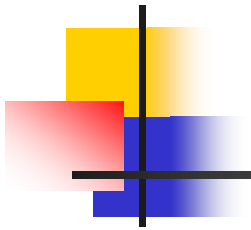
  (Assumption: function *F* has *k* minterms)

  - ORed the minterms of F : *k*-input OR gate
  - NORed the minterms of F′ : $(2^n - k)$-input NOR gate
  - * If $k \leq 2^n/2$, then use an OR gate; otherwise, use a NOR gate.

- For decoder w/ active LOW outputs:

  (Assumption: function *F* has *k* maxterms)

  - ANDed the maxterms of F : *k*-input AND gate
  - NANDed the maxterms of F′ : $(2^n - k)$-input NAND gate
  - * If $k \leq 2^n/2$, then use an AND gate; otherwise, use a NAND gate.

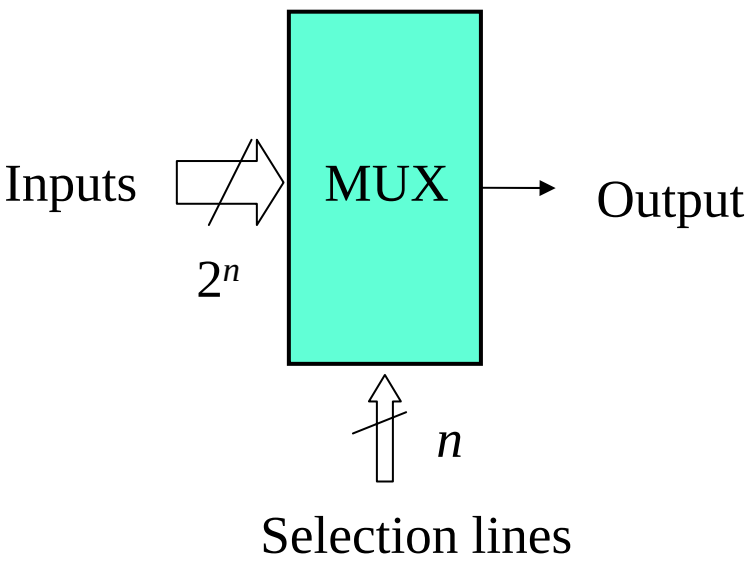


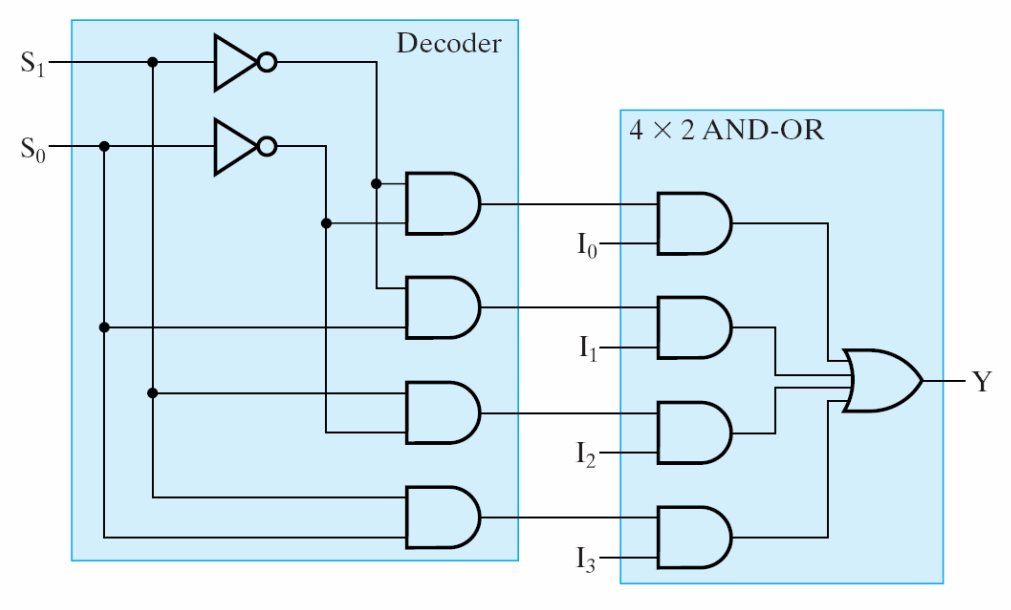
- Summary:
  - The decoder method can be used to implement any combinational ckt.
  - When the decoder method may provide the best solution:
    - If the combinational ckt has many outputs and

      if each output function (or its complement) is expressed w/ a small # of minterms.

B. Using Multiplexers

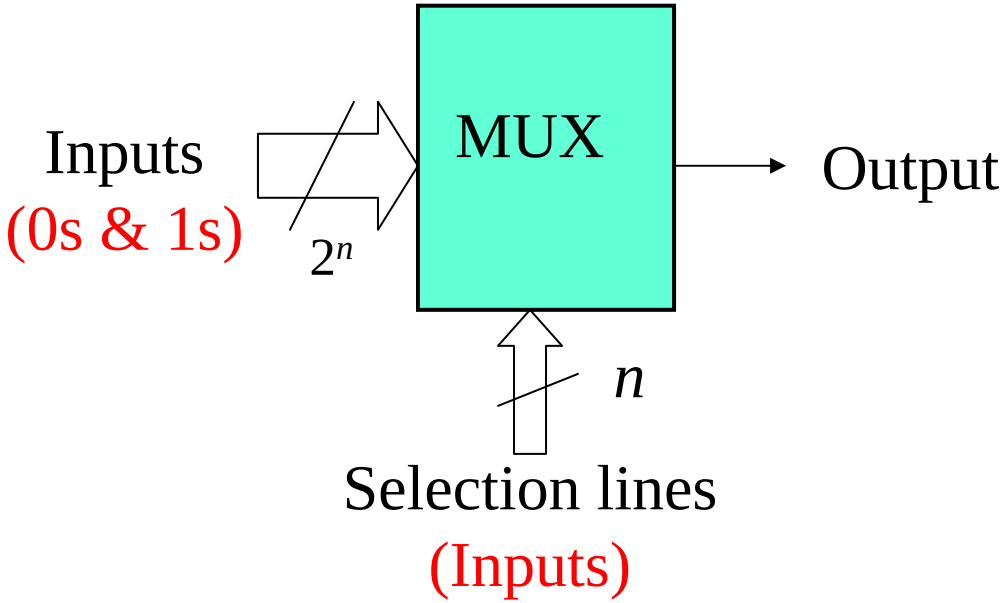
- MUX: a decoder + an $m \times 2$ AND-OR gate
  - Decoder: generates the minterms of the selection inputs.
  - AND-OR gate: provides enabling ckts that determine whether the minterms are "attached" to the OR gate w/ the information inputs (I_i) used as the enabling signals





Implementing *n*-variable Boolean function with 2^n-to-1 MUX

- Implement a Boolean function of *n* variables w/ 2^n-to-1 MUX (a MUX that has *n* selection inputs):
  - The minterms to be included w/ the function are chosen by making their corresponding input lines equal to 1, those minterms not included in the function are disabled by making their input lines equal to 0.



Example 4-7

- E.g.: MUX implementation of a binary adder bit

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

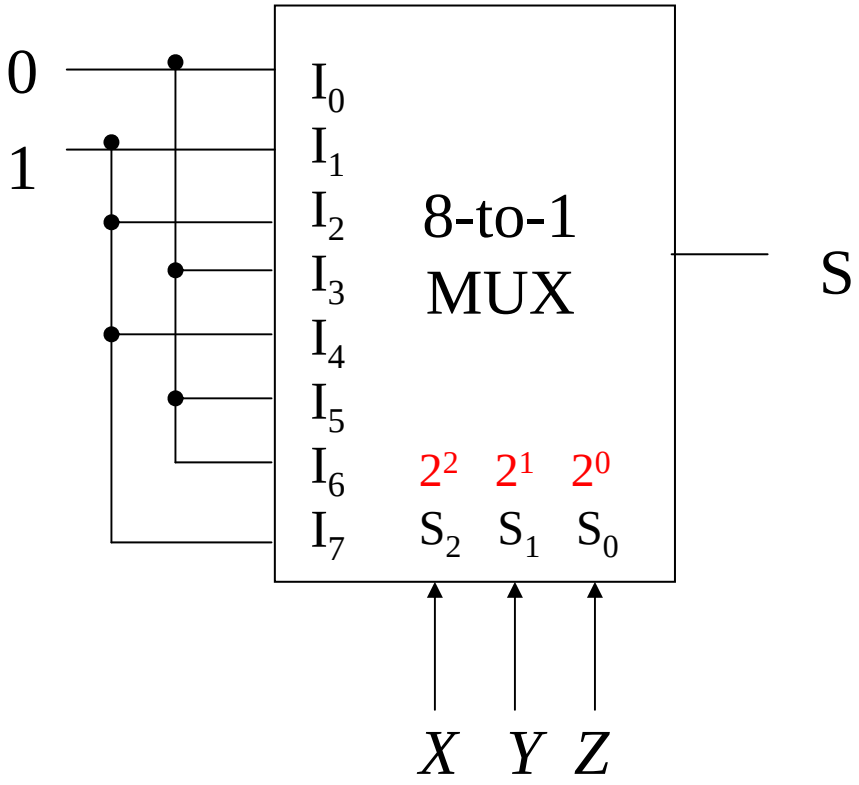
$$S(X,Y,Z) = \sum m(1,2,4,7)$$

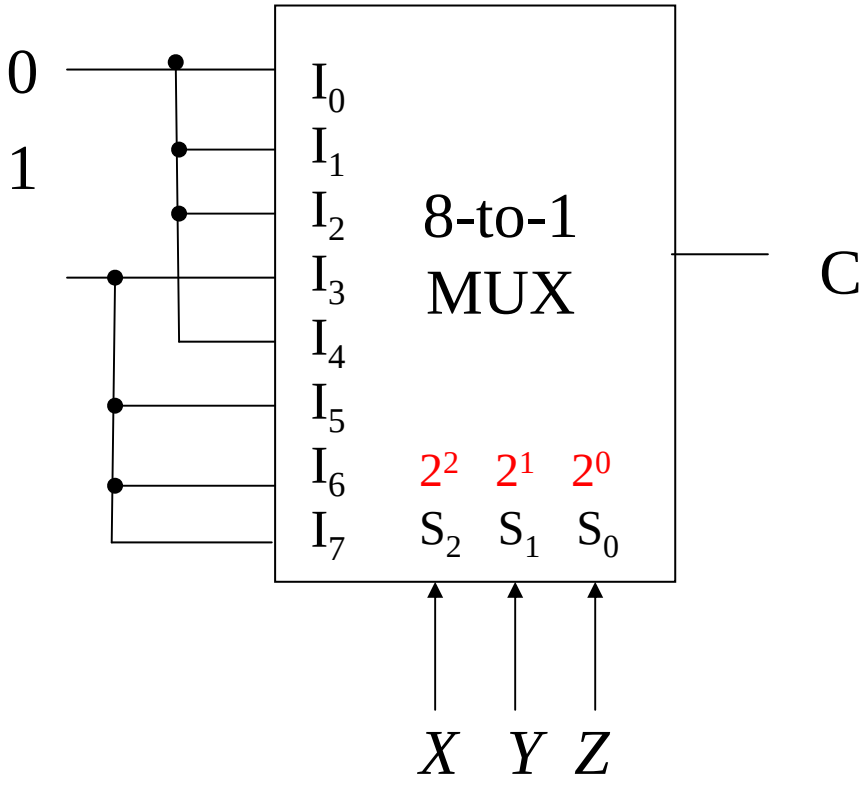
$$C(X,Y,Z) = \sum m(3,5,6,7)$$

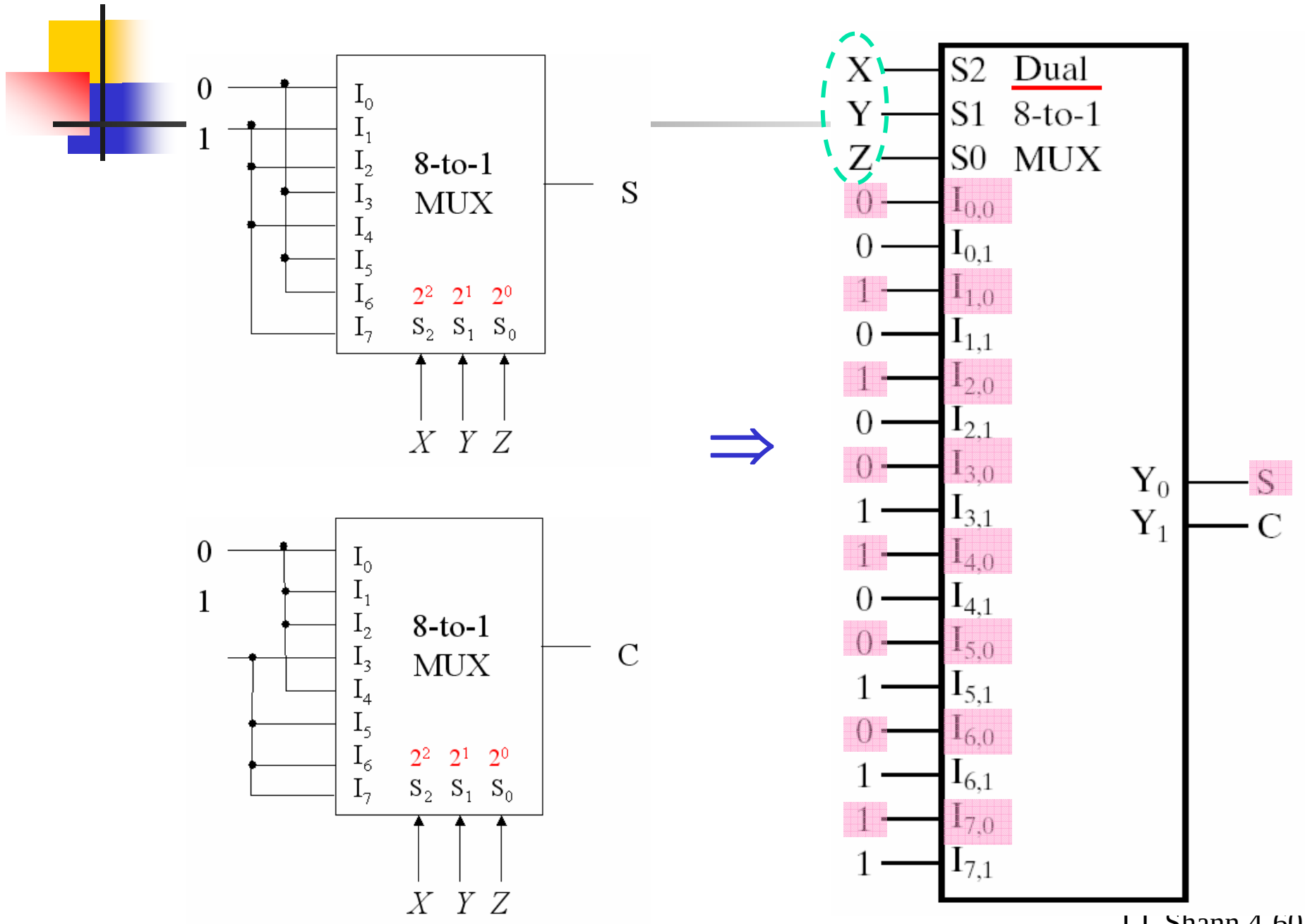
<Ans.>

Need two 8-to-1-line MUXs

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






0
1
I_0
I_1
I_2
I_3
I_4
I_5
I_6
I_7
8-to-1
MUX
2^2 2^1 2^0
S_2 S_1 S_0
S
X Y Z
0
1
8-to-1
MUX
C
X Y Z
X
Y
Z
S2 Dual
S1 8-to-1
S0 MUX
$I_{0,0}$ $I_{0,1}$ $I_{1,0}$ $I_{1,1}$ $I_{2,0}$ $I_{2,1}$ $I_{3,0}$ $I_{3,1}$ $I_{4,0}$ $I_{4,1}$ $I_{5,0}$ $I_{5,1}$ $I_{6,0}$ $I_{6,1}$ $I_{7,0}$ $I_{7,1}$
Y_0
Y_1
S
C

Implementing *n*-variable Boolean function with 2^{n-1}-to-1 MUX

- Implement a Boolean function of *n* variables w/ a MUX that has $(n - 1)$ selection inputs:

  2^{n-1}-to-1 MUX

  - The first $n - 1$ variables of the function are connected to the selection inputs of the MUX.
  - The remaining single variable (Z) of the function is used for the data inputs: Z, Z', 1, or 0.
  - Procedure:
    - The Boolean function is first listed in a truth table.
    - The first $n - 1$ variables in the table are applied to the selection inputs of the MUX.
    - For each combination of the selection variables, we evaluate the output as a function of the last variable Z: Z, Z', 1, or 0
    - These values are then applied to the data inputs in the proper order.

Example 4-8

- E.g.: Alternative MUX implementation of a binary adder bit

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

Use two 4-to-1-line MUXs to implement the ckt. Choose *X* and *Y* as the selection inputs.

* It is possible to use any other variable of the function for the MUX selection inputs.

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

Z

Z′

Z′

Z

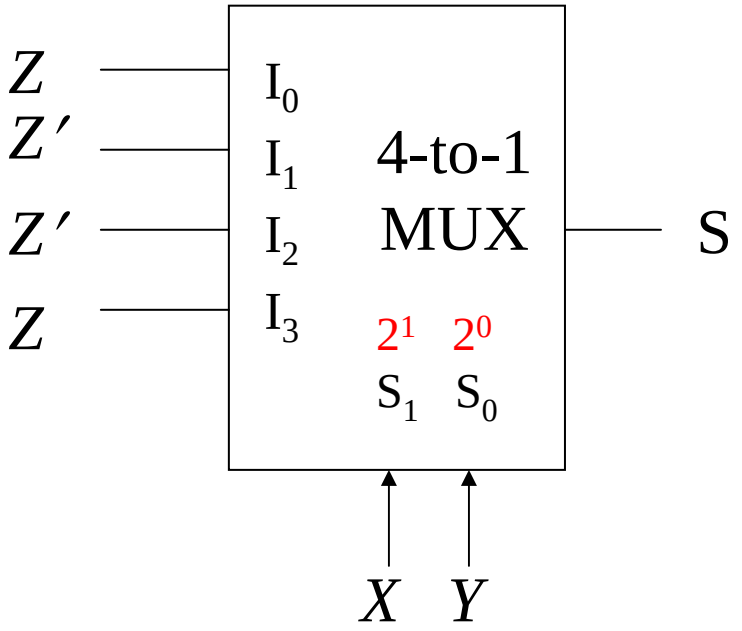
X	Y	Z	C
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

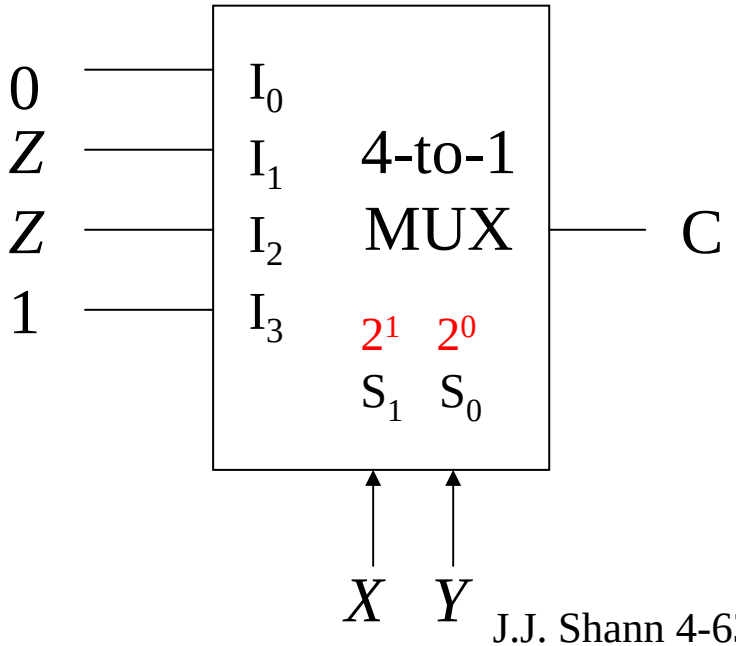
0

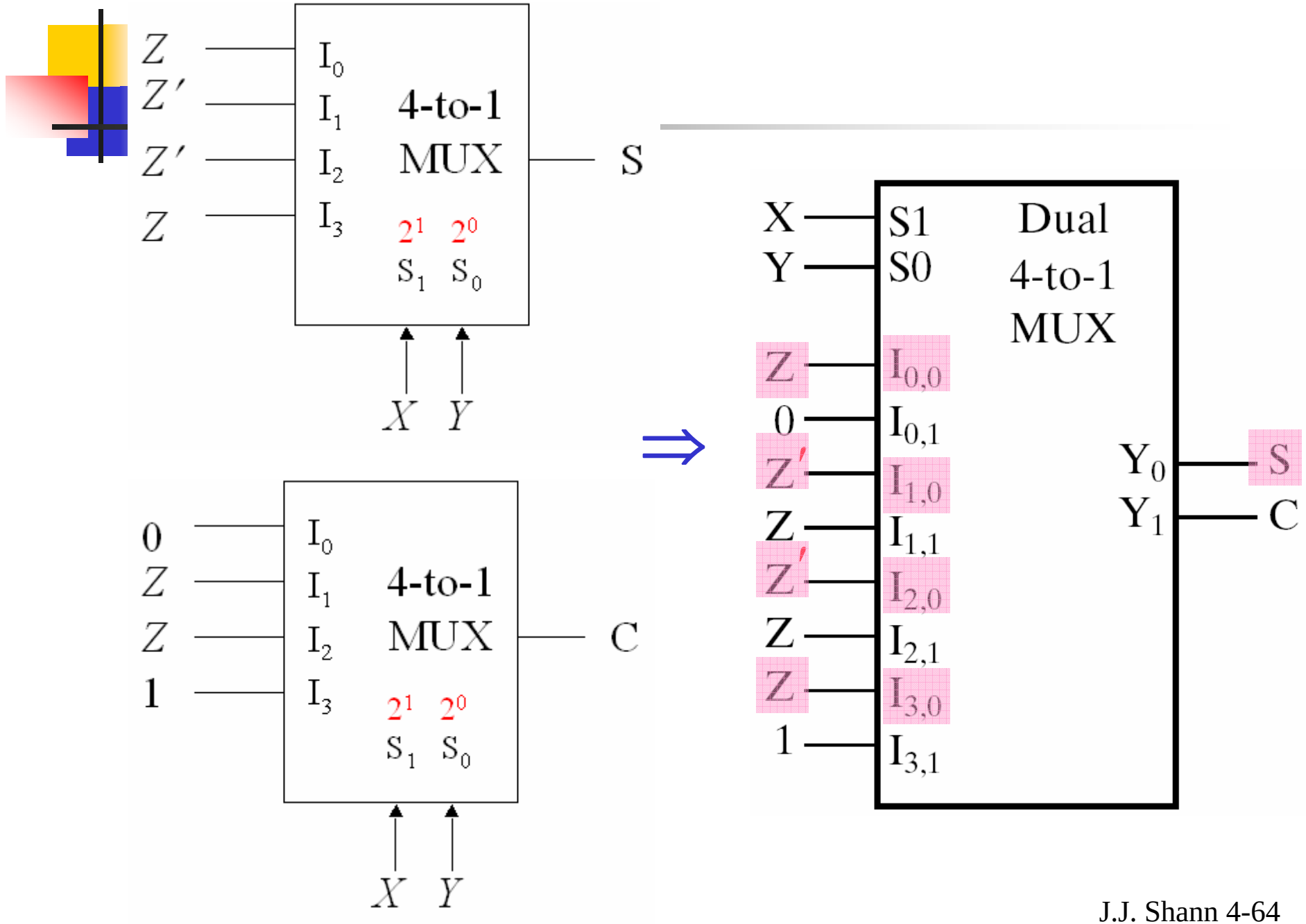
Z

Z

1

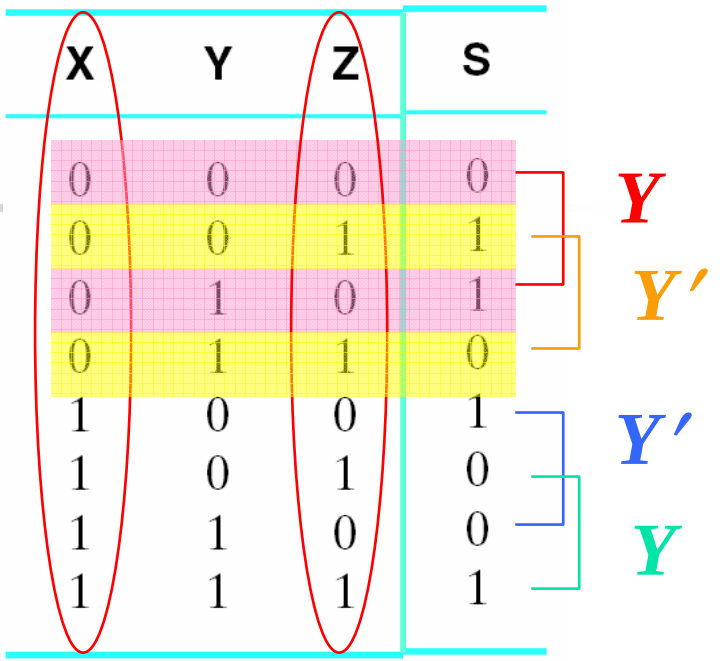


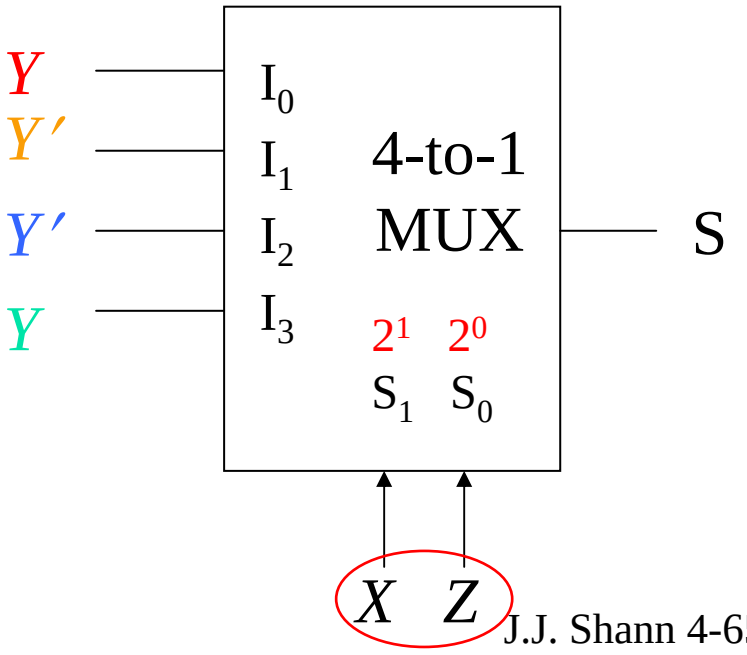



Z
Z′
Z′
Z
I0
I1
I2
I3
4-to-1
MUX
2^1 2^0
S1 S0
S
X Y
0
Z
Z
1
I0
I1
I2
I3
4-to-1
MUX
2^1 2^0
S1 S0
C
X Y
X
Y
S1
S0
Dual
4-to-1
MUX
Z
0
Z′
Z
Z′
Z
Z
1
I0,0
I0,1
I1,0
I1,1
I2,0
I2,1
I3,0
I3,1
Y0
Y1
S
C

- It is possible to use any other variable of the function for the MUX selection inputs.
  - E.g.: Realize *S* by using *X* and *Z* as the selection lines.

X	Z	Y	S





Example 4-9

- E.g.: MUX implementation of 4-variable function

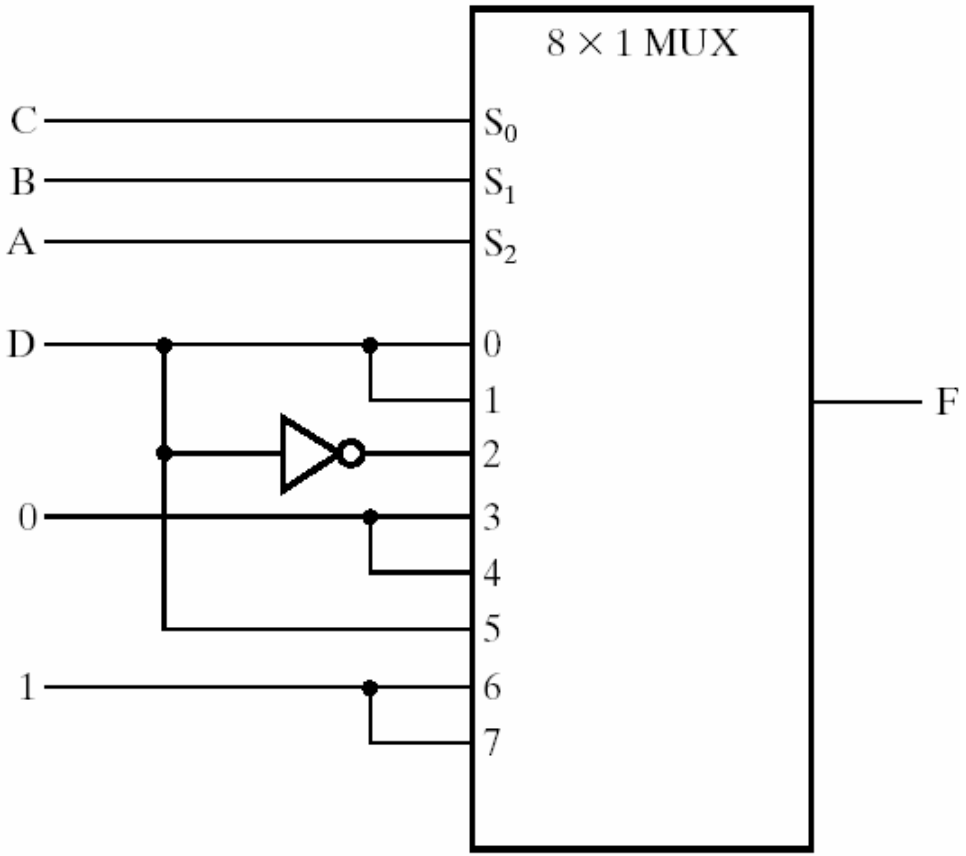
  Use an 8-to-1 MUX to realize the function

  $F(A,B,C,D) = \Sigma(1,3,4,11,12,13,14,15)$

$F(A,B,C,D) = \Sigma(1,3,4,11,12,13,14,15)$

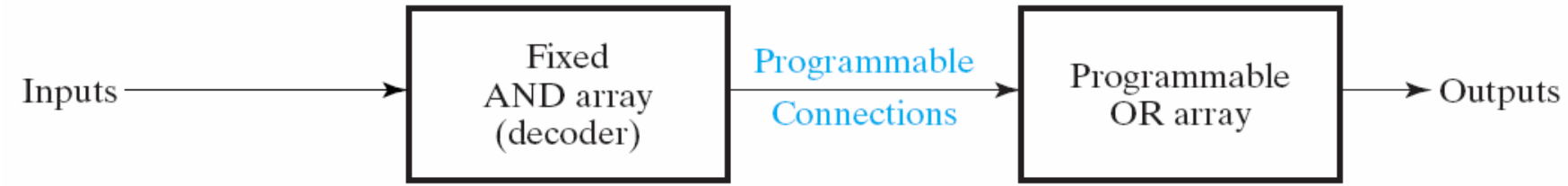
<Ans.>

A	B	C	D	F	
0	0	0	0	0	F = D
0	0	0	1	1	
0	0	1	0	0	F = D
0	0	1	1	1	
0	1	0	0	1	$F = \overline{D}$
0	1	0	1	0	
0	1	1	0	0	F = 0
0	1	1	1	0	
1	0	0	0	0	F = 0
1	0	0	1	0	
1	0	1	0	0	F = D
1	0	1	1	1	
1	1	0	0	1	F = 1
1	1	0	1	1	
1	1	1	0	1	F = 1
1	1	1	1	1	



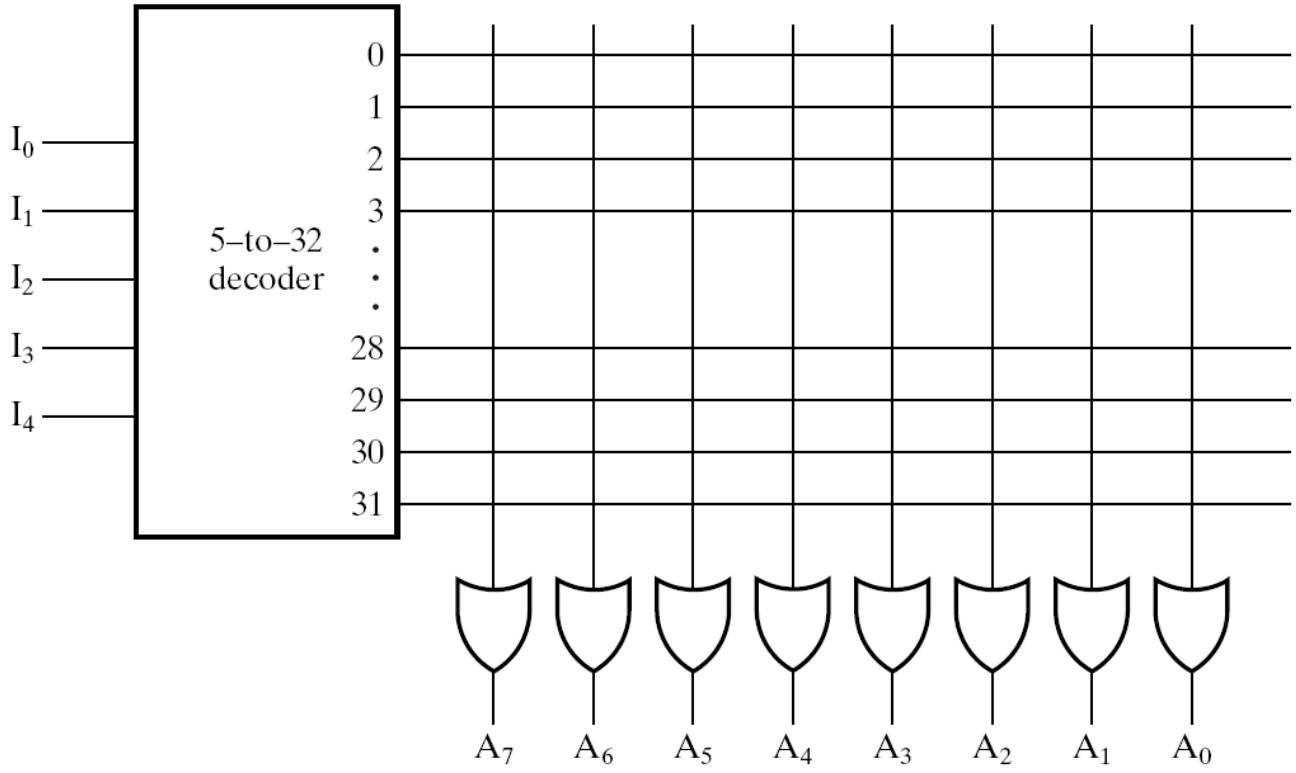
C. Using Read-Only Memories

- ROM:
  - a memory device in which permanent binary information is stored



- ROM Block diagram:

- Internal logic of a $2^k \times n$ ROM: comb. ckt.
  - have an internal $k \times 2^k$ decoder & n OR gates
  - E.g.: a 32×8 ROM



- ROM truth table:
  - shows the word content in each address
  - gives all the information for programming the ROM
  - E.g.:

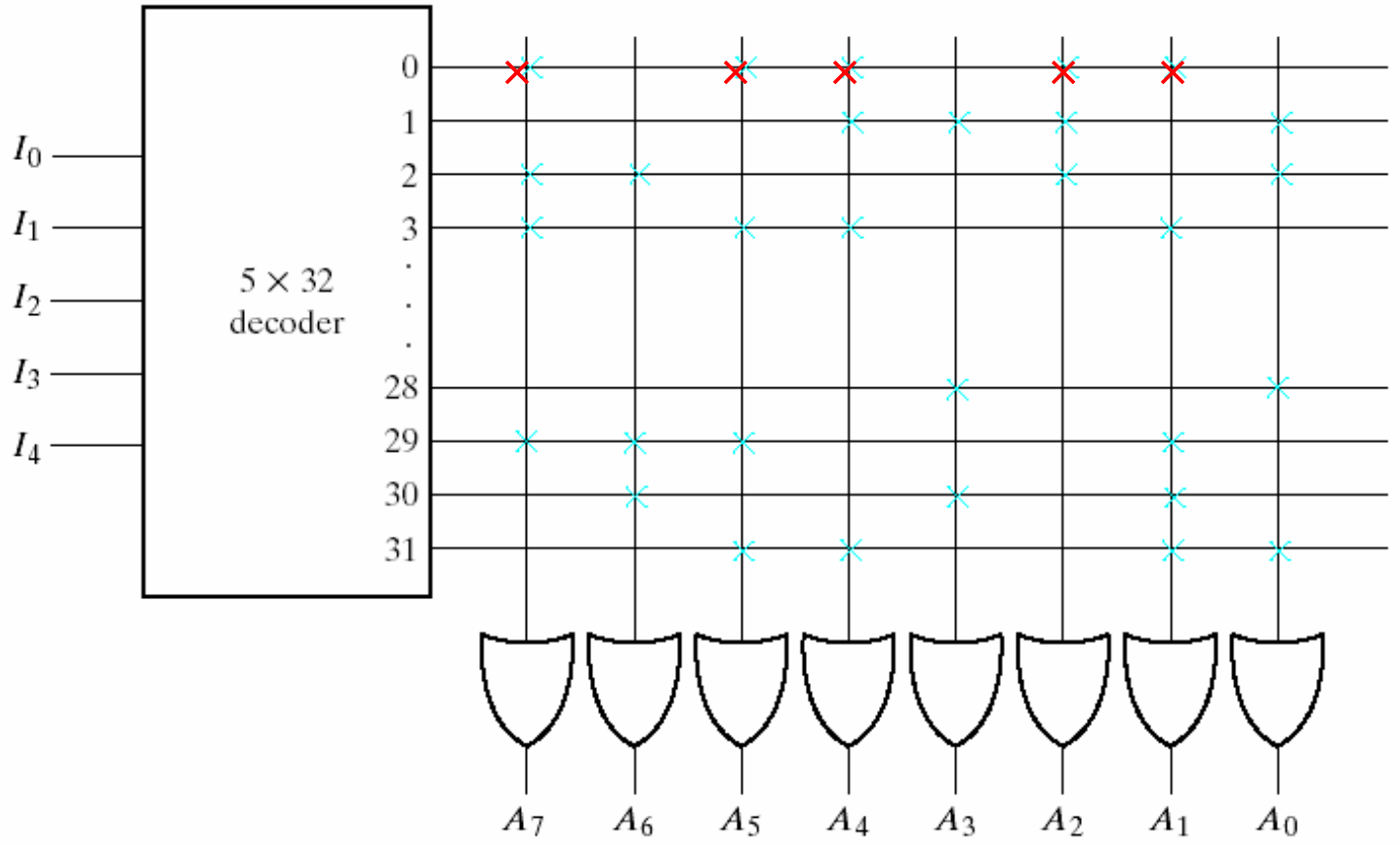
ROM Truth Table (Partial)

Inputs					Outputs							
I4	I3	I2	I1	I0	A7	A6	A5	A4	A3	A2	A1	A0
0	0	0	0	0	1	0	1	1	0	1	1	0
0	0	0	0	1	0	0	0	1	1	1	0	1
0	0	0	1	0	1	1	0	0	0	1	0	1
0	0	0	1	1	1	0	1	1	0	0	1	0
		⋮					⋮					
1	1	1	0	0	0	0	0	0	1	0	0	1
1	1	1	0	1	1	1	1	0	0	0	1	0
1	1	1	1	0	0	1	0	0	1	0	1	0
1	1	1	1	1	0	0	1	1	0	0	1	1

- Programming the ROM:
  - E.g.:

ROM Truth Table (Partial)

Inputs					Outputs							
I4	I3	I2	I1	I0	A7	A6	A5	A4	A3	A2	A1	A0
0	0	0	0	0	1	0	1	1	0	1	1	0
0	0	0	0	1	0	0	0	1	1	1	0	1
0	0	0	1	0	1	1	0	0	0	1	0	1
0	0	0	1	1	1	0	1	1	0	0	1	0
⋮					⋮							
1	1	1	0	0	0	0	0	0	1	0	0	1
1	1	1	0	1	1	1	1	0	0	0	1	0
1	1	1	1	0	0	1	0	0	1	0	1	0
1	1	1	1	1	0	0	1	1	0	0	1	1



Combinational Circuit Implementation

- ROM: a decoder + OR gates
  - Boolean functions $\rightarrow$ "sum of minterms" form
    $\rightarrow$ ROM truth table
    $\rightarrow$ ROM implementation
  - For an n-input, m-output combinational ckt
    $\Rightarrow 2^n \times m$ ROM
- Design procedure:
  1. Determine the size of ROM required from the # of inputs and outputs of the comb. ckt.
  2. Obtain the programming truth table of the ROM.
  3. The 0's (or 1's) in the output functions of the truth table directly specify those links that must be removed to provide the required comb ckt in sum of minterms form.

Example 4-10

- E.g.: Implementing a combinational ckt using a ROM

  The ckt accepts a 3-bit number and generates an output binary number equal to the square of the input number.

<Ans.>

Truth table:

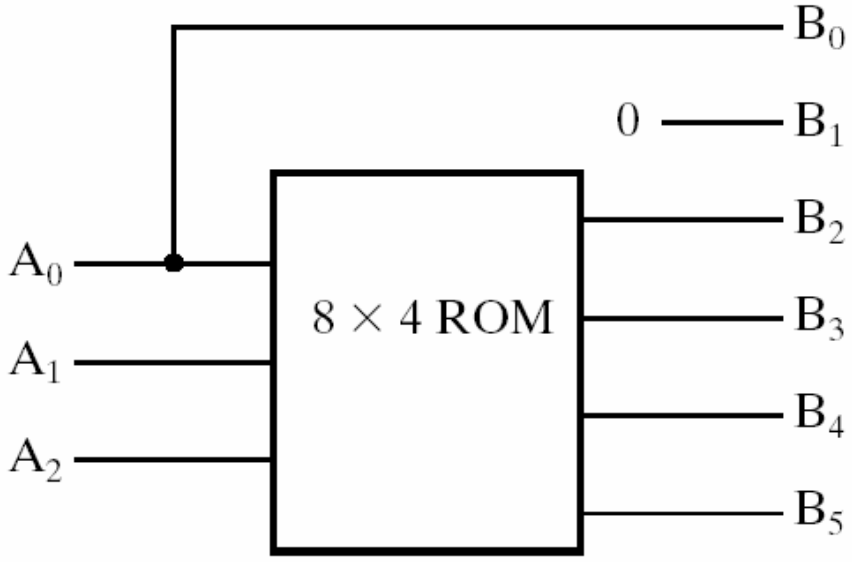
Inputs			Outputs						
A_2	A_1	A_0	B_5	B_4	B_3	B_2	B_1	B_0	Decimal
0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	1	1
0	1	0	0	0	0	1	0	0	4
0	1	1	0	0	1	0	0	1	9
1	0	0	0	1	0	0	0	0	16
1	0	1	0	1	1	0	0	1	25
1	1	0	1	0	0	1	0	0	36
1	1	1	1	1	0	0	0	1	49

ROM implementation:

$B_1 = 0, B_0 = A_0$

Truth table $\Rightarrow$ 3 inputs, 4 outputs

$\Rightarrow$ a 8×4 ROM



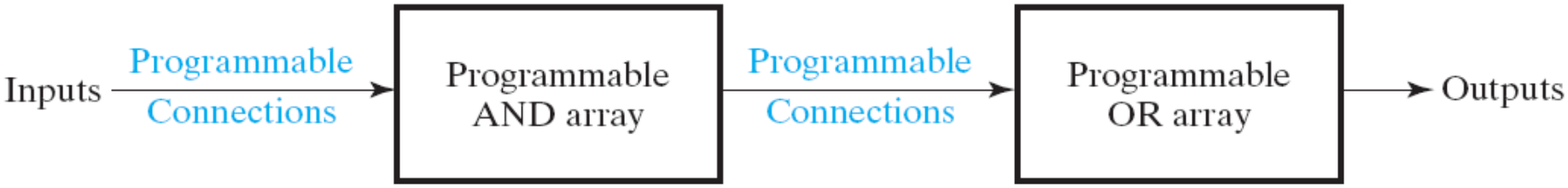
(a) Block diagram

A_2	A_1	A_0	B_5	B_4	B_3	B_2
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	1
0	1	1	0	0	1	0
1	0	0	0	1	0	0
1	0	1	0	1	1	0
1	1	0	1	0	0	1
1	1	1	1	1	0	0

(b) ROM truth table

D. Using Programmable Logic Arrays

- PLA:



– The array of AND gates can be programmed to generate any product terms of the input variables.

– The product terms are then connected to OR gates to provide the sum of products for the required Boolean functions.

* Compare to ROM:

does not provide full decoding of the variables

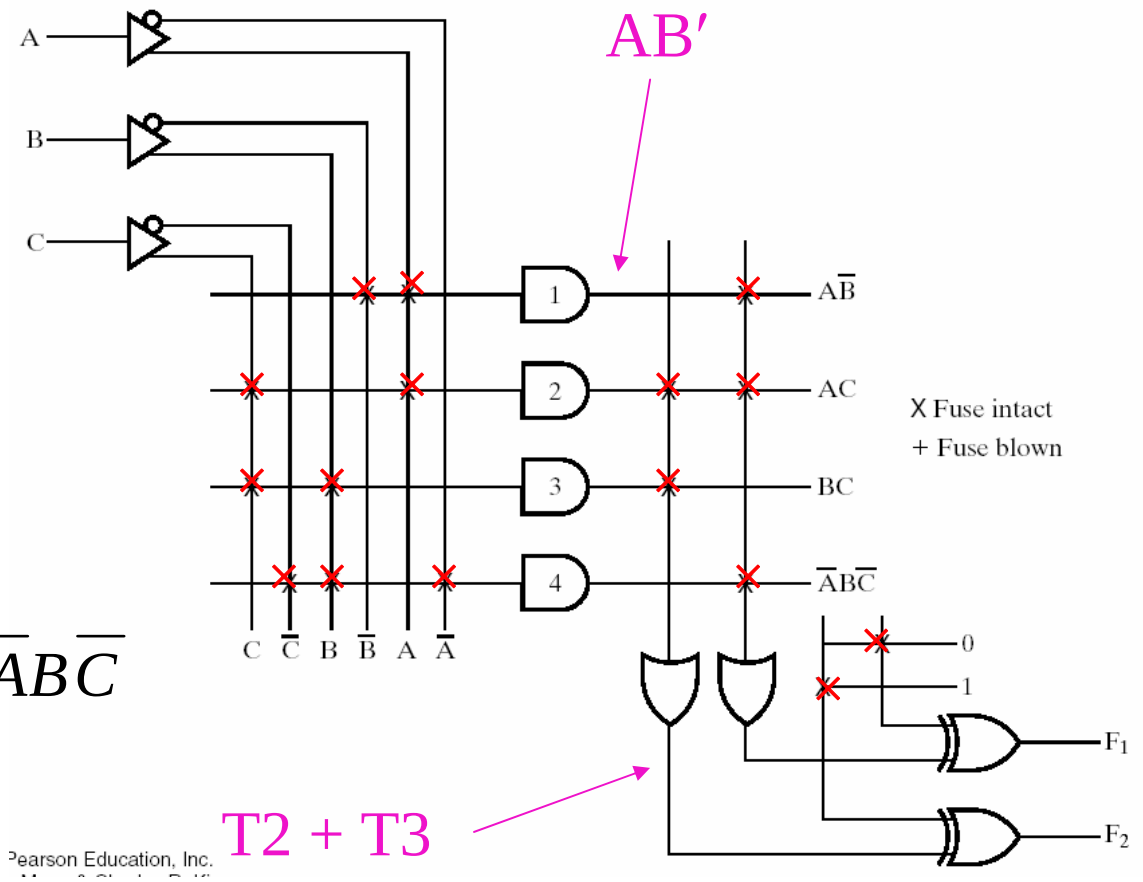
$\Rightarrow$ does not generate all the minterms

- Internal logic of a PLA:
  - E.g.: a $3 \times 4 \times 2$ PLA

3 inputs

4 product terms

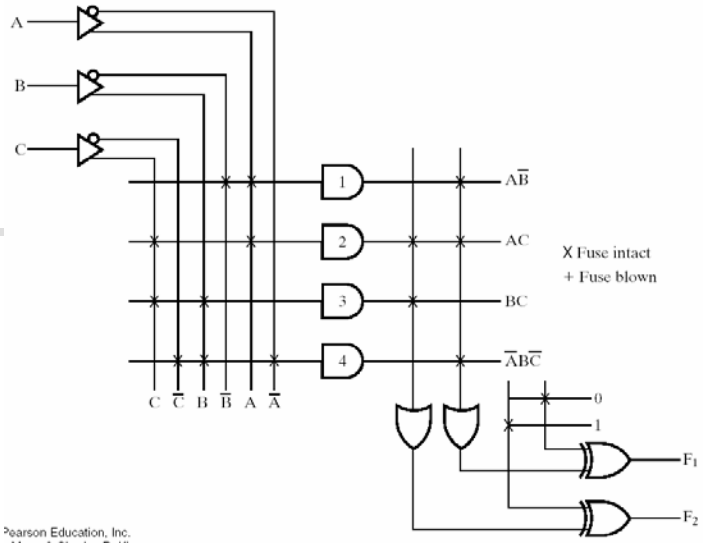
2 outputs



$$F_1 = A\overline{B} + AC + \overline{A}B\overline{C}$$

$$F_2 = \overline{AC + BC}$$

- PLA programming table:
  - shows the internal connections (the fuse map) of a PLA
  - E.



Programming Table for the PLA in Figure 4-24

		Inputs			Outputs	
	Product term	A	B	C	(T) F_1	(C) F_2
$A\bar{B}$	1	1	0	—	1	—
AC	2	1	—	1	1	1
BC	3	—	1	1	—	1
$\bar{A}B\bar{C}$	4	0	1	0	1	—

(AND array) (OR array)

Combinational Circuit Implementation

- Design method:
  - A PLA has a finite # of AND gates

    $\Rightarrow$ Reduce the # of distinct product terms.
  - Both the true and complement of each function should be simplified to see which one can be expressed w/ fewer product terms and which one provides product terms that are common to other functions.
  - The # of literals in a term is not important.

Example 4-11

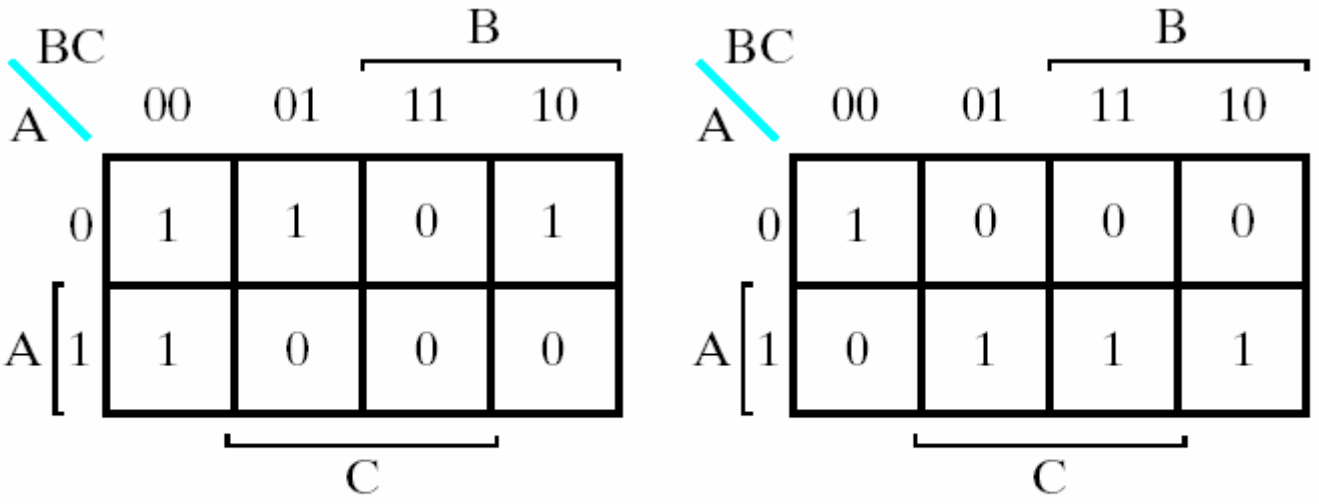
- E.g.: Implementing a combinational ckt using a PLA

  Implement the following two Boolean functions w/ a PLA:

  $F_1(A, B, C) = \Sigma\,(0, 1, 2, 4)$

  $F_2(A, B, C) = \Sigma\,(0, 5, 6, 7)$

  <Ans.>

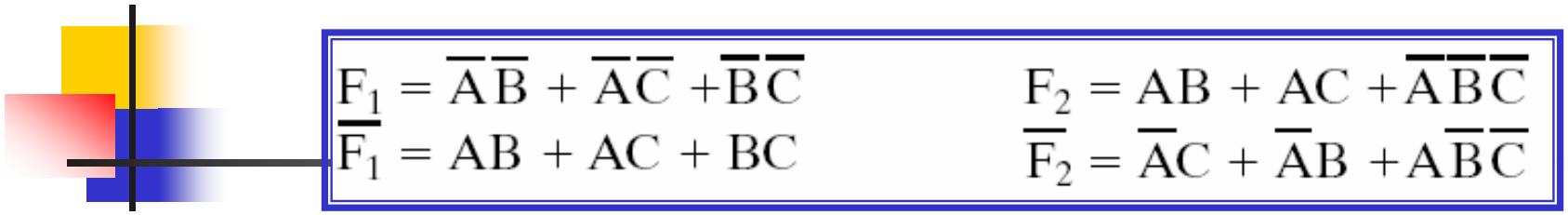


$F_1 = \overline{A}\,\overline{B} + \overline{A}\,\overline{C} + \overline{B}\,\overline{C}$

$\overline{F}_1 = AB + AC + BC$

$F_2 = AB + AC + A\overline{B}\,\overline{C}$

$\overline{F}_2 = \overline{A}C + \overline{A}B + A\overline{B}\,\overline{C}$



$F_1 = A'B' + A'C' + B'C'$ (6)

$\overline{F}_1 = (AB + AC + BC)'$ (6)

(4)

(6)

$F_2 = AB + AC + A'B'C'$

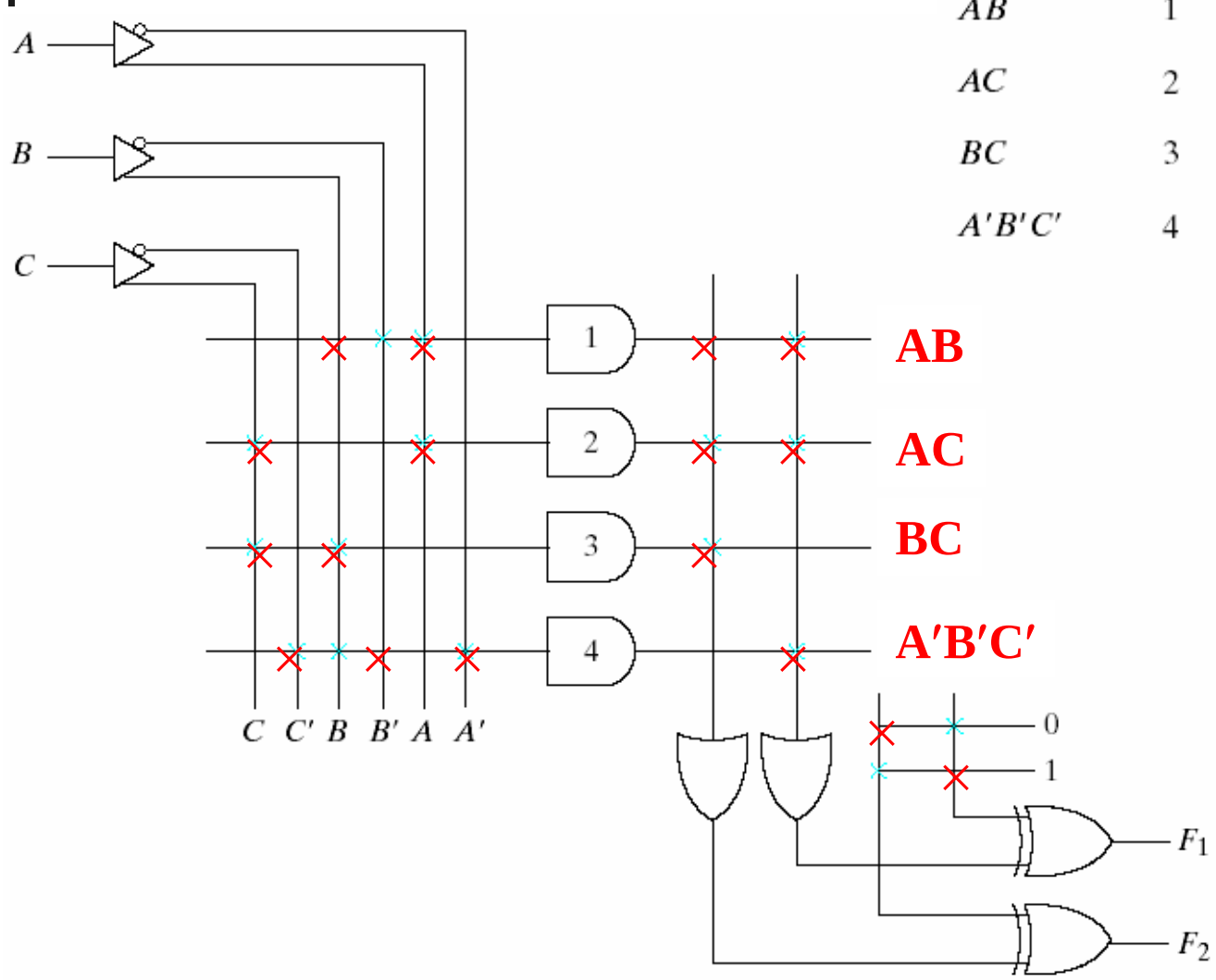
$F_2 = (A'C + A'B + AB'C')'$

PLA programming table

	Product term	Inputs A	B	C	Outputs (C) F_1	(T) F_2
AB	1	1	1	–	1	1
AC	2	1	–	1	1	1
BC	3	–	1	1	1	–
$\overline{A}\,\overline{B}\,\overline{C}$	4	0	0	0	–	1

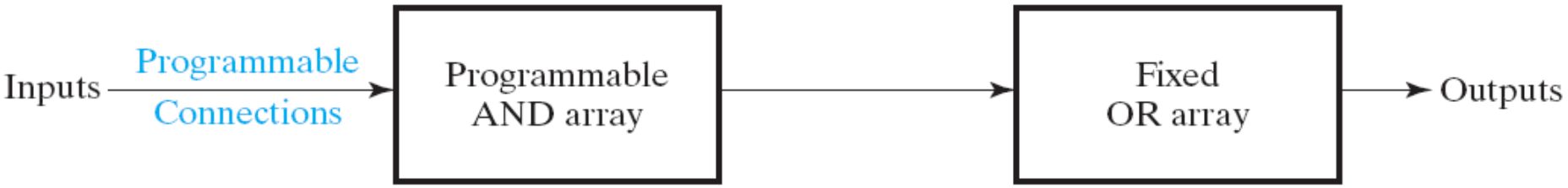
PLA programming table

	Product term	Inputs A	B	C	Outputs (C) F_1	(T) F_2
AB	1	1	1	–	1	1
AC	2	1	–	1	1	1
BC	3	–	1	1	1	–
$A'B'C'$	4	0	0	0	–	1



E. Using Programmable Array Logic Devices

- PAL:



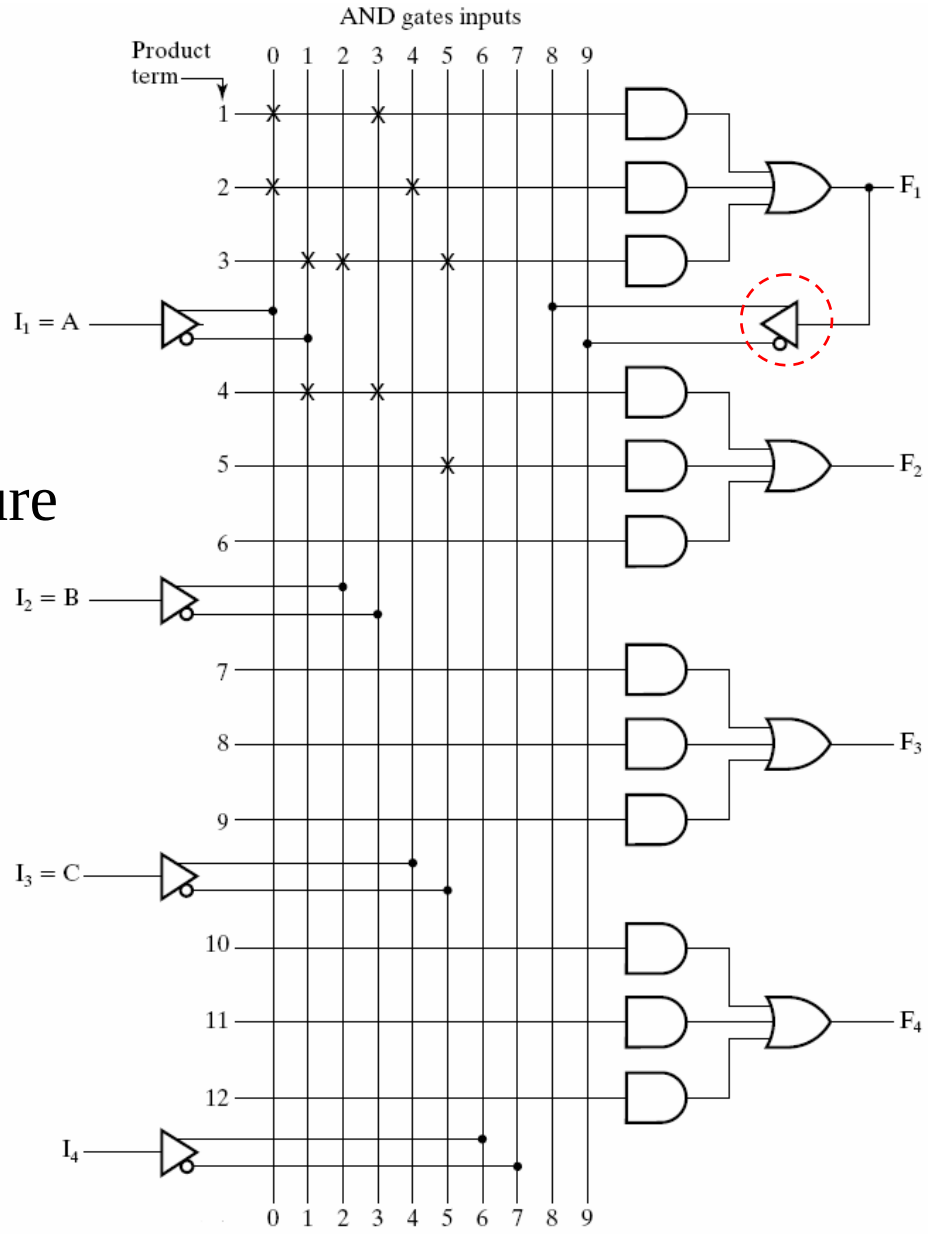
  - Only the AND gates are programmable.
  - The PAL is easier to program, but is not as flexible as the PLA.

- Internal logic of a PAL:
  - E.g.:

    4 inputs

    4 outputs (4 sections)

    3-wide AND-OR structure



$$F_1 = A\overline{B} + AC + \overline{A}B\overline{C}$$

$$F_2 = \overline{AC + BC} = \overline{A}\,\overline{B} + \overline{C}$$

Combinational Circuit Implementation

- Design method:
  - The Boolean functions must be simplified to fit into each section.
  - Each function can be simplified by itself.

    (No sharing of product terms)
  - If the # of terms in a function is too large, it may be necessary to use two sections to implement the function.
- PAL programming table

Example 4-12

- E.g.: Implementing a combinational ckt using a PAL

  Implement the following Boolean functions w/ a PAL:

  $W(A,B,C,D) = \Sigma(2,12,13)$

  $X(A,B,C,D) = \Sigma(7,8,9,10,11,12,13,14)$

  $Y(A,B,C,D) = \Sigma(0,2,3,4,5,6,7,8,10,11,15)$

  $Z(A,B,C,D) = \Sigma(1,2,8,12,13)$

  <Ans.>

  $W = ABC' + A'B'CD'$

  $X = A + BCD$

  $Y = A'B + CD + B'D'$

  $Z = ABC' + A'B'CD' + AC'D' + A'B'C'D$

  $= W + AC'D' + A'B'C'D$

$W = ABC' + A'B'CD'$

$X = A + BCD$

$Y = A'B + CD + B'D'$

$Z = W + AC'D' + A'B'C'D$

PAL programming table

Product term	AND Inputs A	B	C	D	W	Outputs
1	1	1	0	—	—	$W = A B \overline{C}$
2	0	0	1	0	—	$+ \overline{A}\,\overline{B} C \overline{D}$
3	—	—	—	—	—	
4	1	—	—	—	—	$X = A$
5	—	1	1	1	—	$+ BCD$
6	—	—	—	—	—	
7	0	1	—	—	—	$Y = \overline{A} B$
8	—	—	1	1	—	$+ CD$
9	—	0	—	0	—	$+ \overline{B}\,\overline{D}$
10	—	—	—	—	1	$Z = W$
11	1	—	0	0	—	$+ A \overline{C}\,\overline{D}$
12	0	0	0	1	—	$+ \overline{A}\,\overline{B}\,\overline{C} D$

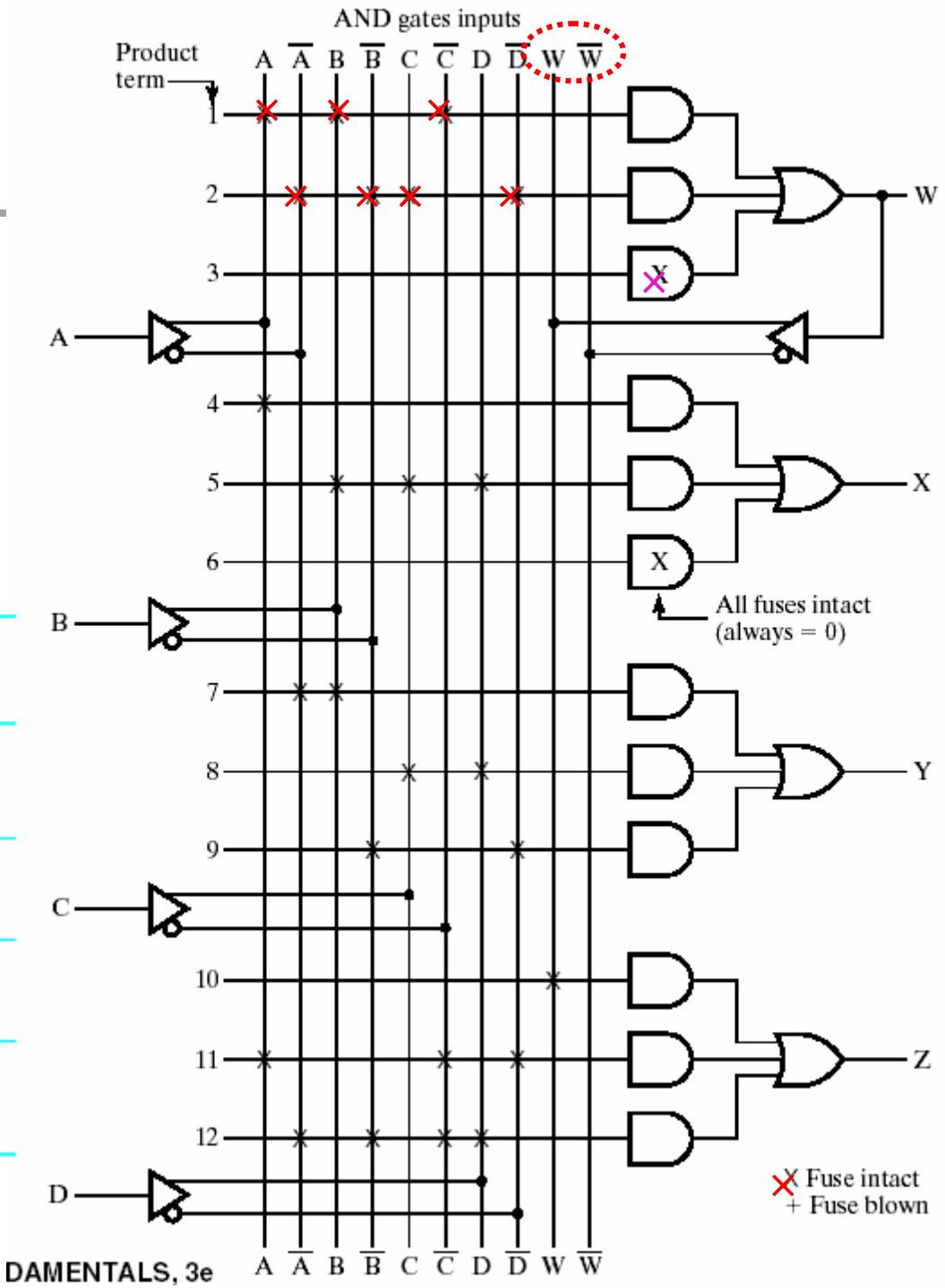
$W = ABC' + A'B'CD'$

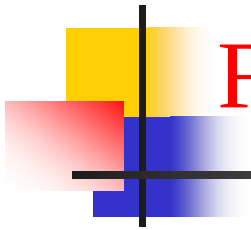
$X = A + BCD$

$Y = A'B + CD + B'D'$

$Z = W + AC'D' + A'B'C'D$

Product term	AND Inputs A	B	C	D	W	Outputs
1	1	1	0	—	—	$W = A\overline{B}\overline{C}$
2	0	0	1	0	—	$+\overline{A}\overline{B}C\overline{D}$
3	—	—	—	—	—	
4	1	—	—	—	—	$X = A$
5	—	1	1	1	—	$+BCD$
6	—	—	—	—	—	
7	0	1	—	—	—	$Y = \overline{A}B$
8	—	—	1	1	—	$+CD$
9	—	0	—	0	—	$+\overline{B}\overline{D}$
10	—	—	—	—	1	$Z = W$
11	1	—	0	0	—	$+A\overline{C}\overline{D}$
12	0	0	0	1	—	$+\overline{A}\overline{B}\overline{C}D$





F. Using Lookup Tables

- Combinational ckt implementation using LUT:
  - An *m*-input LUT can implement any function of *m* or fewer variables.
    - stores the truth table of a function

      (the same as programming a single output ROM)
    - Typically, $m = 4$.
- Key problems in programming LUT:
  - deal w/ functions w/ $> m$ input variables
  - share LUTs among multiple functions
- Support *s*:
  - the # of input variables for a function

- Optimization method:
  - use multiple-level logic transformations:

    primarily decomposition & extraction
- Optimization goal:
  - implement the function or functions by using a min # of LUTs w/ the constraint that a single LUT can implement a subfunction of at most *m* variables.

  $\Rightarrow$ Find a min # of eqs, each w/ at most *m* variables, that implement the desired function or functions.

Example 4-13

- Eg: Implementing a single-output function w/ LUTs

$$F1(A,B,C,D,E,F,G,H,I) = ABCDE + \overline{F}GHI\,\overline{D}\,\overline{E}$$

<Ans.>

$s = 9$, $m = 4$

$\Rightarrow$ the min # of LUTs k needed $= \lceil 9/4 \rceil = 3$

$\Rightarrow$ Decompose F_1 into 3 eqs, each w/ at most $s = 4$

$$F_1 = (ABC)DE + (\overline{F}GHI)\overline{D}\,\overline{E}$$

$\Downarrow$ decomposition

$$F_1(D,E,X_1,X_2) = X_1DE + X_2\overline{D}\,\overline{E}$$
$$X_1(A,B,C) = ABC$$
$$X_2(F,G,H,I) = \overline{F}GHI$$

$\Rightarrow$ 3 LUTs

Example 4-14

- E.g.: Implementing a multiple-output function w/ LUTs

$$F_1(A,B,C,D,E,F,G,H,I) = ABCDE + \overline{F}GHI\,\overline{D}\,\overline{E}$$

$$F_2(A,B,C,D,E,F,G,H,I) = ABCEF + \overline{F}GHI$$

<Ans.>

$s = 9$, $m = 4$

$\Rightarrow$ At least $\lceil 9/4 \rceil = 3$ LUTs are required for each function.

Two of the LUTs can be shared

$\Rightarrow$ the min # of LUTs k needed = 6 – 2 = 4

Factoring F_2 w/ sharing of eqs w/ the decomposition of F_1.

E.g. 4-13

$$F_1 = (ABC)DE + (\overline{F}GHI)\overline{D}\,\overline{E}$$

$\Downarrow$ decomposition

$$F_1(D, E, X_1, X_2) = X_1DE + X_2\overline{D}\,\overline{E}$$

$$X_1(A, B, C) = ABC$$

$$X_2(F, G, H, I) = \overline{F}GHI$$

$$F_2 = (ABC)EF + (\overline{F}GHI)$$

$$= X_1EF + X_2$$

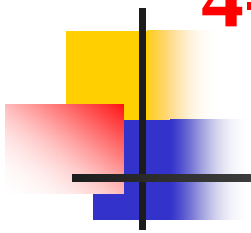
$\Downarrow$

$$F_1(D, E, X_1, X_2) = X_1DE + X_2\overline{D}\,\overline{E}$$

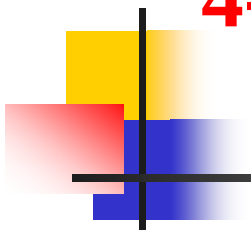
$$F_2(E, F, X_1, X_2) = X_1EF + X_2$$

$$X_1(A, B, C) = ABC$$

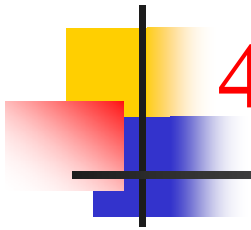
$$X_2(F, G, H, I) = \overline{F}GHI$$



4-7 HDL Representation for Combinational Circuits – VHDL



4-8 HDL Representation for Combinational Circuits – Verilog



4-9 Chapter Summary

- Function blocks used frequently to design larger ckts:
  - Rudimentary ckts that implement functions of a single variable: value-fixing, transferring, inverting
  - Decoders
  - Encoders
  - Multiplexers
- Design of combinational logic ckts using
  - Decoders
  - Multiplexers
  - Programmable logic: ROM, PLA, PAL LUT

Problems

Sections	Exercises
§4-1	
§4-2	4-1~4-4
§4-3	4-5~4-9
§4-4	4-10, 4-11, 4-14
§4-5	4-12, 4-13, 4-15~4-20
§4-6	4-21~4-35
§4-7 (×)	4-36~4-45
§4-8 (×)	4-46~4-55

Homework

- 4-2 -->（§4-2, power and ground symbol的觀念）
- 4-8 -->（§4-3, line decoder設計）
- 4-12 -->（§4-5, multiplexer設計）
- 4-17 -->（§4-5, multiplexer設計）
- 4-21 -->（§4-6, 使用decoder and OR gates來設計組合電路）
- 4-24 -->（§4-6, 使用multiplexerand OR gates來設計組合電路）
- 4-30 -->（§4-6, 使用ROM來設計組合電路）
- 4-31 -->（§4-6, 使用PLA來設計組合電路）
- 4-34 -->（§4-6, 使用PAL來設計組合電路）