OF THE
DEPARTMENT OF
INFORMATION ENGINEERING

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Digital Systems

Combinational Logic Design

Marta Bagatin, marta.bagatin@unipd.it

Degree Course in Information Engineering

Academic year 2023-2024

Summary of the Previous Episodes

1. Digital systems process information represented as **binary numbers**
2. The basic building blocks of digital systems are **logic gates**, which implement logic functions on binary numbers
3. Binary information can be handled with **Boolean algebra**
4. The **simplification of Boolean functions** allows us to get minimum cost implementations
5. A logic circuit can be described and simulated through a VHDL code

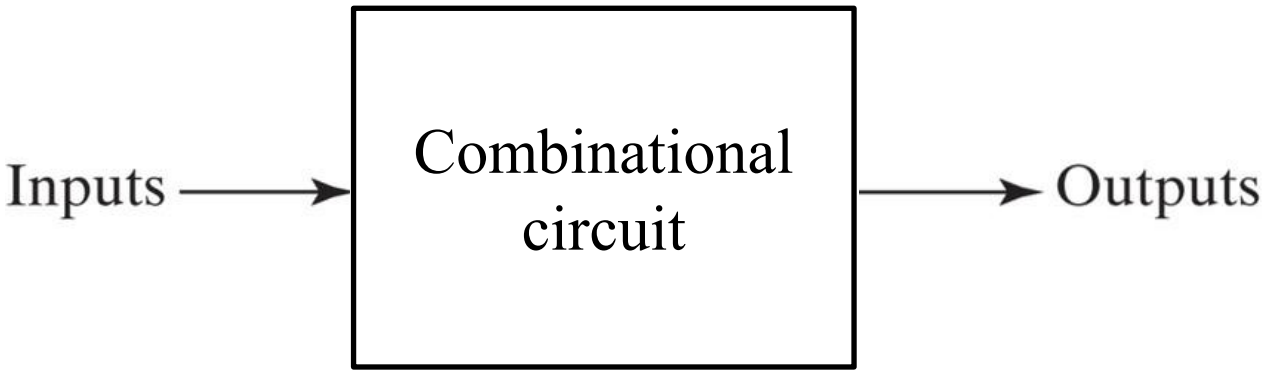
Purpose of the Lesson

- Define a combinational logic system
- Using the concepts studied until now, we will study the steps to **design a combinational system**
- Study the **basic combinational blocks** and their **VHDL description**
  - **Decoder**
  - **Encoder**
  - **Multiplexer**

Combinational Logic vs. Sequential Logic

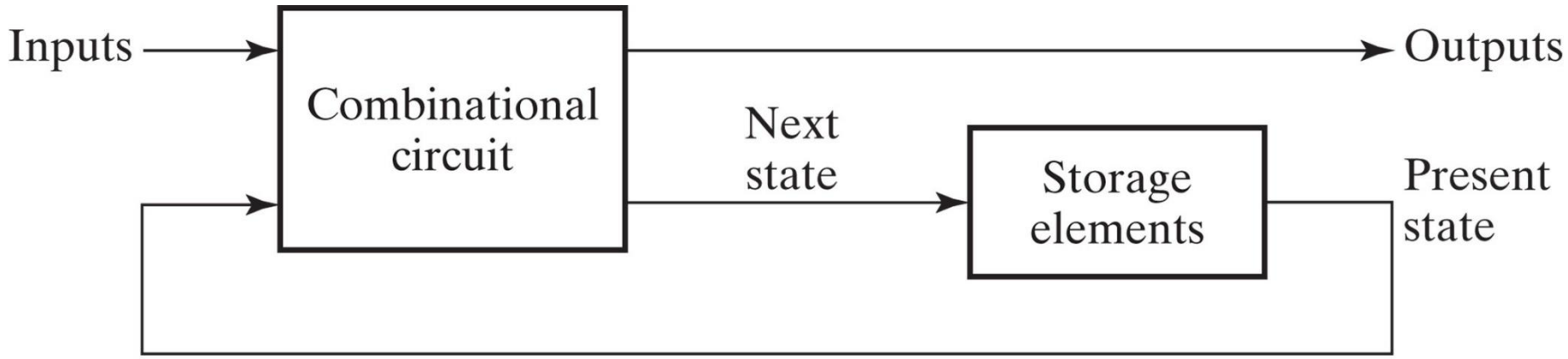
Combinational Logic

- In a combinational logic circuit, the **output depends ONLY on the present value of the inputs** and NOT on their past history
  - A combinational system **does not have memory**, i.e. the input-output relationship does not contain the time variable
- A combinational circuit is made of logic gates interconnected with each other, with **NO feedback paths** (i.e. connections between the output and the input)



Sequential Logic

- In a sequential circuit, the **output depends *both on the current value of the inputs and their past history*** **(also known as system state)**
- A sequential circuit consists of combinational blocks and memory elements, i.e. devices capable of storing binary information
  - Sequential logic is characterized by a feedback path which connects the system state to the input



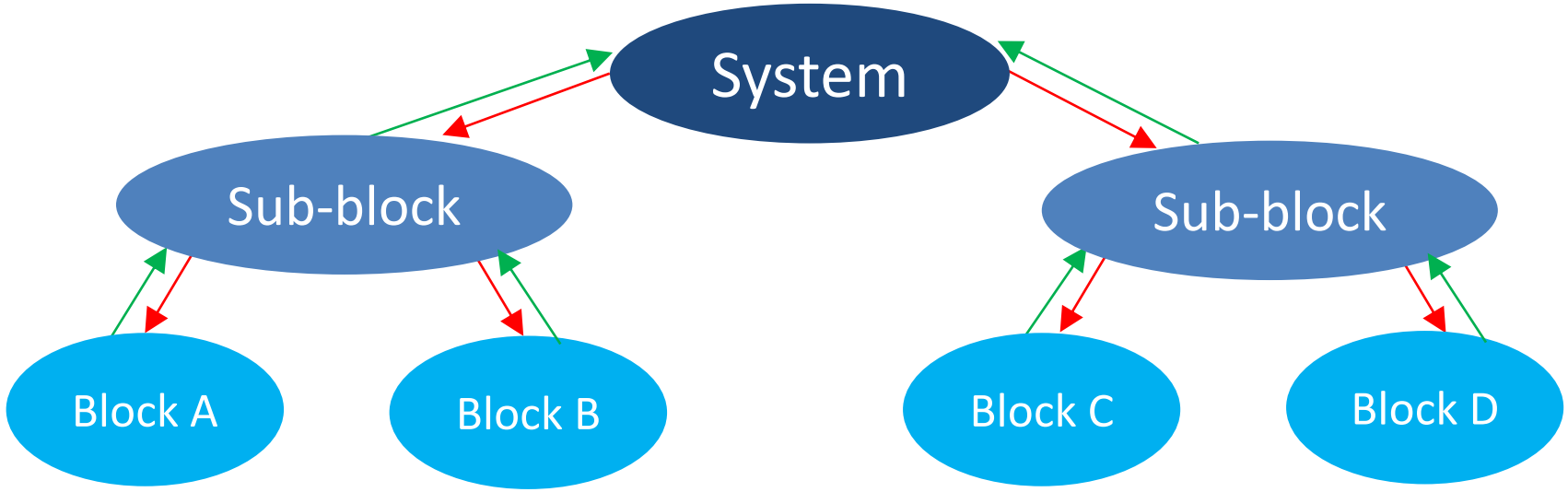
Steps for the Design of a Combinational Circuit

1) Identify the **project specifications**, determine the number of **inputs and outputs of the system**, and assign a name to each of them
2) Find the **truth table** describing the relation between inputs and outputs
3) Determine the Boolean **minimum-cost function for each output as a function of the inputs** (K-map)
4) Draw the logic **diagram** of the circuit, using the blocks available in the chosen technology
5) Verify the correctness of the design

Hierarchical Approach

"Divide and Conquer" Approach

- "Divide and Conquer" is a hierarchical approach
- The system is **recursively divided into sub-blocks**. The procedure is repeated until such blocks become simple enough so that they can be **designed individually**
- The blocks are designed: starting from the specifications, the input-output relationship is formulated, the representation is optimized and the minimum-cost function is mapped into the gates or logic blocks available in the chosen technology
- Finally, the blocks are **connected to each other** to build the complete system



Example 3.1: 4-bit Equality Comparator

- **Specifications**
  - **2 inputs A and B** (4 bits)
  - **1 output E** (1 bit)
  - E is '1' if A and B are equal, E is '0' if A and B are unequal

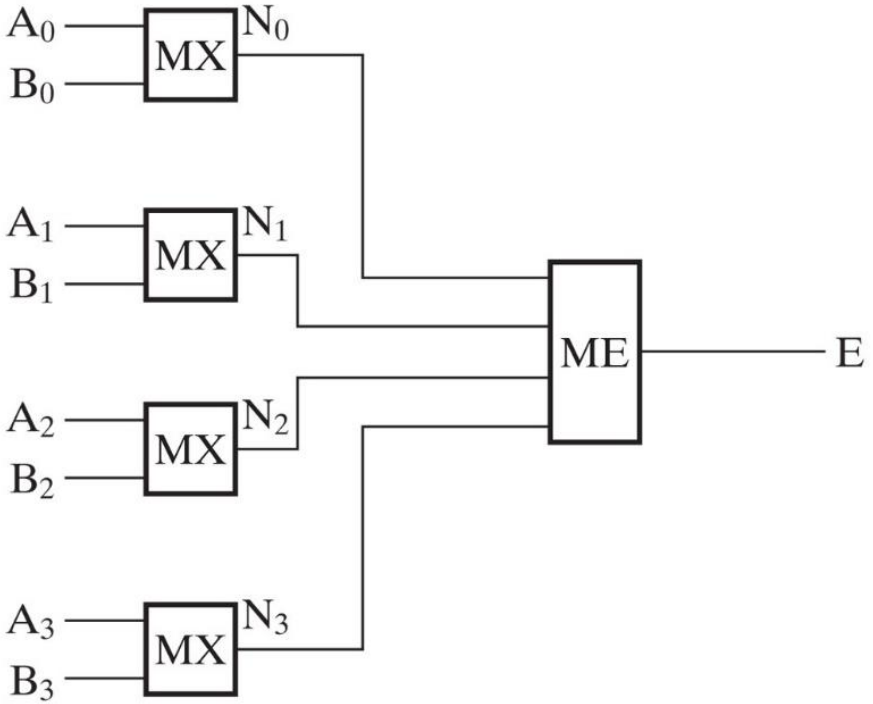
- The function has 8 inputs ($2^8 = 256$ possible combinations), so the problem is not manageable via truth tables and K-maps. Let's use a hierarchical approach
  - Two vectors of n bits are equal if all the bits in the corresponding positions are equal, i.e. $A_0 = B_0$, $A_1 = B_1$, …, $A_{n-1} = B_{n-1}$

Example 3.1: 4-bit Equality Comparator

- 1st hierarchical level (higher abstraction)

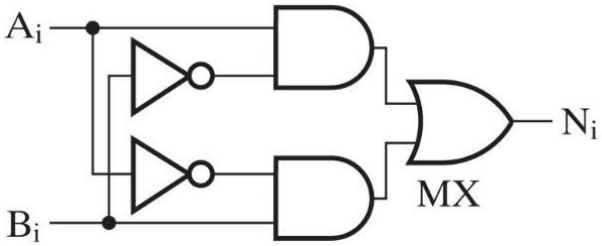
The system is divided into blocks:

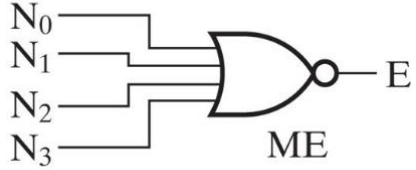
  - four 1-bit comparators (MX) that compare a bit of input A with the corresponding bit of input B
  - a block (ME) that combines the 4 results of the comparison, in order to obtain the output E



- 2nd hierarchical level (lower abstraction)

  - Detail of the 1-bit comparator
  - Detail of the block that combines the results of the 4 comparisons



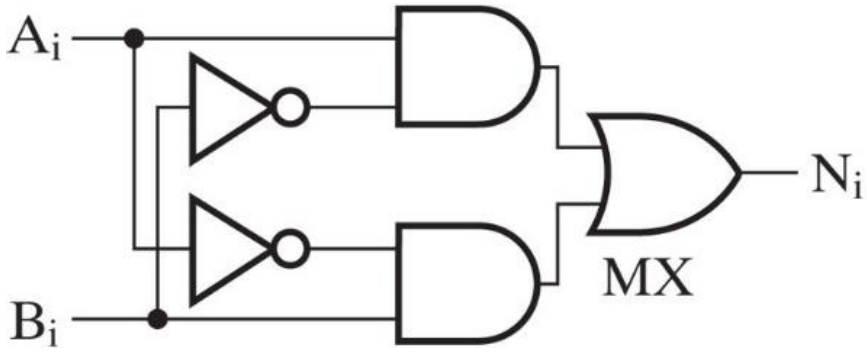


Example 3.1: 4-bit Equality Comparator

- 2nd hierarchical level
- Detail of the 1-bit comparator:

  - Output is '0' if the two input bits are equal
    -> $N_i = 0$ if $A_i = B_i$

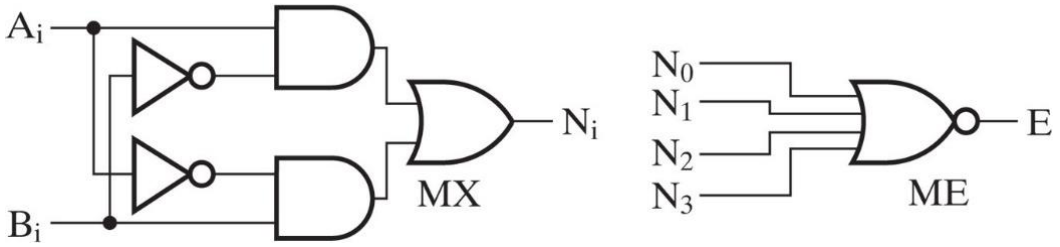
  - Output is '1' if the two input bits are different
    ->$N_i = 1$ if $A_i \neq B_i$



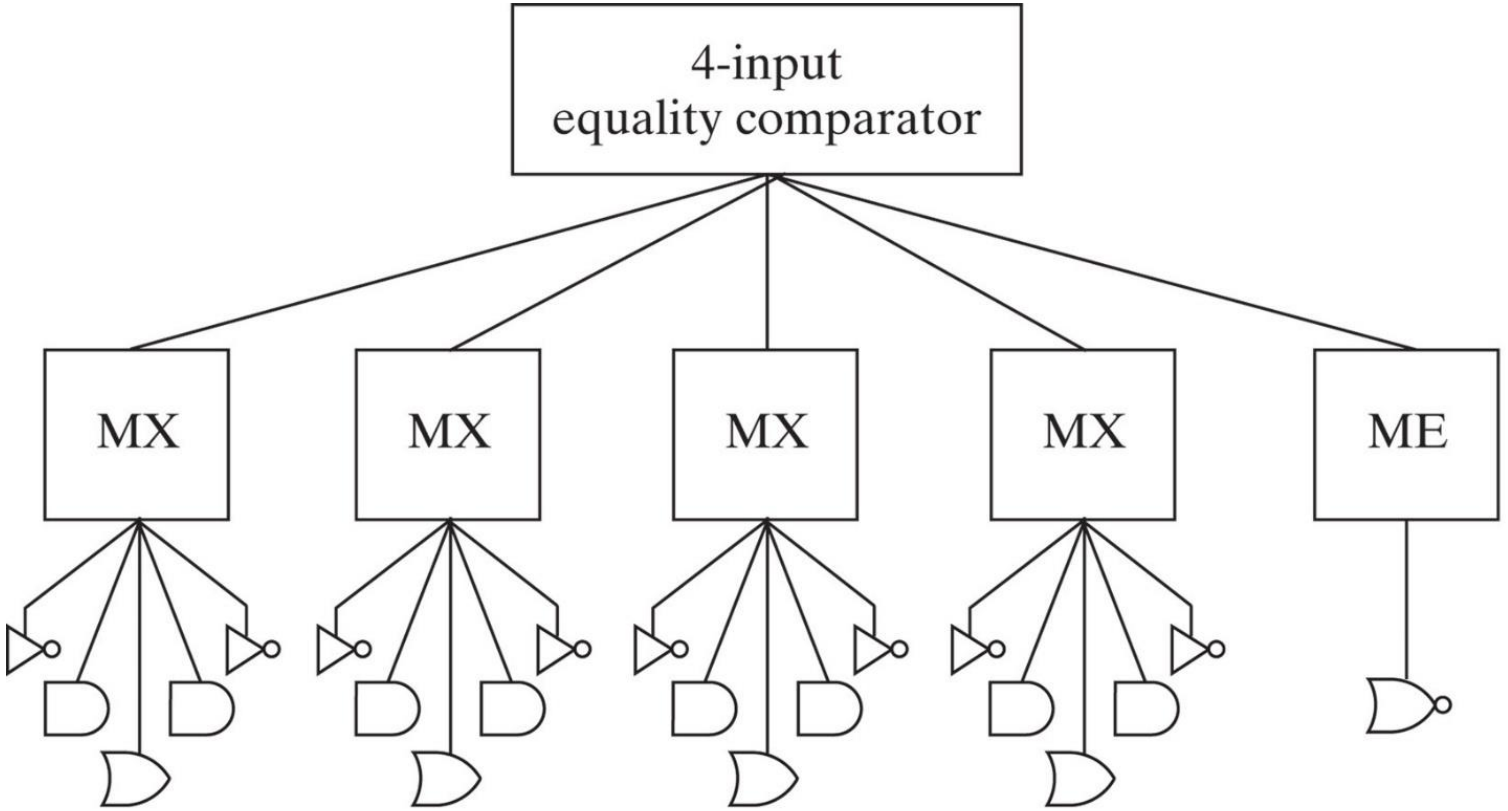
$$N_i = \overline{A_i} B_i + A_i \overline{B_i}$$

- Detail of the block that combines the results of the comparisons:
  - Output is '1' if **all** inputs are '0' (A = B) and is equal to '0' if **at least** one of the inputs is '1'

$$E = \overline{N_0 + N_1 + N_2 + N_3}$$

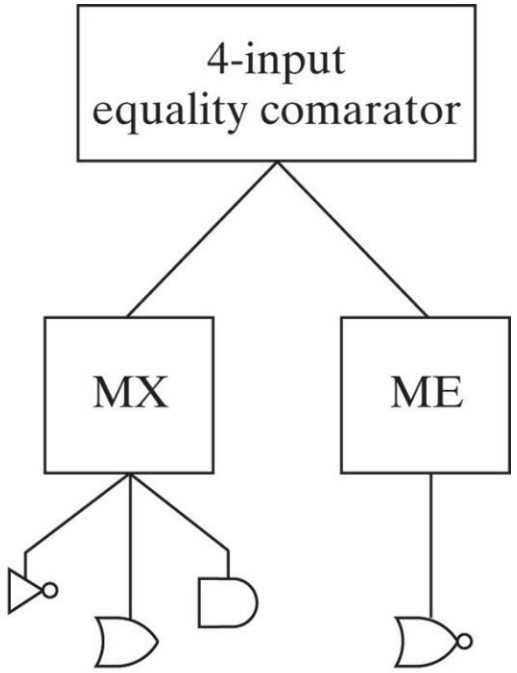


Example 3.1: Hierarchical Representation



- The **hierarchical representation (tree)** provides a simplified scheme
- As we move down in the hierarchy, the level of abstraction decreases, ending with the **leaves**, i.e. the logic gates (**primitive blocks**)
- Note: lines are not interconnections!

Example 3.1: Hierarchical Representation

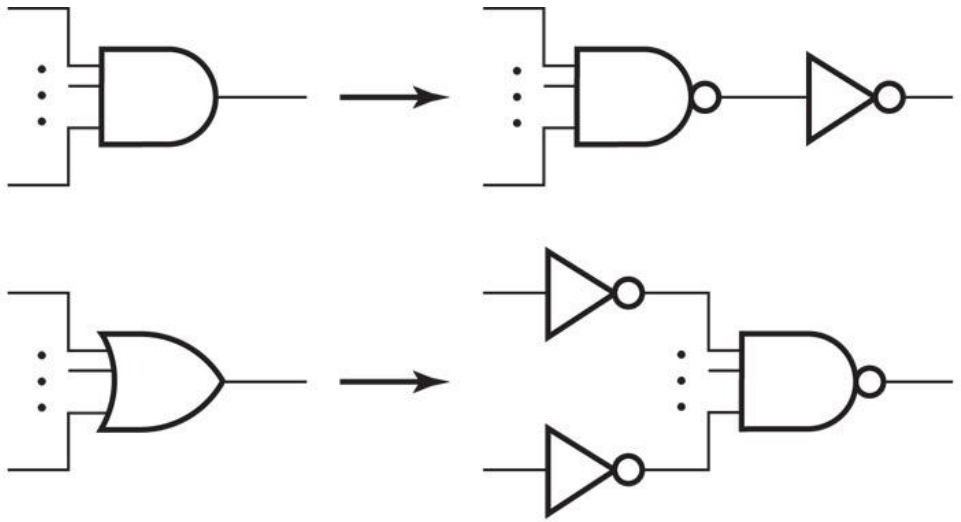


- An even more simplified representation shows identical blocks only once
- This representation highlights the fact that these blocks, once designed, can be reused (each appearance is an **instance**)
- **Regular circuits** (with identical blocks repeated multiple times) require less effort from the designer point of view, when dealing with complex systems

- A hierarchical representation does not provide implementation details, but only the **number of blocks to be designed**: the ratio between the number of primitive blocks (logic gates) in the final circuit and the number of blocks in a hierarchical diagram, including primitive ones, provides a measure of the **regularity of the circuit**

Technology Mapping: Hints

- Technology mapping is the step in which the logic (or schematic) diagram is replaced by a diagram that uses the components available in the particular technology chosen
- For reasons related to currently available technologies (CMOS, Complementary CMOS), **NAND and NOR gates are smaller and faster** with respect to the AND, OR, NOT gates. For this reason, circuits are typically implemented only with NAND gates or NOR gates



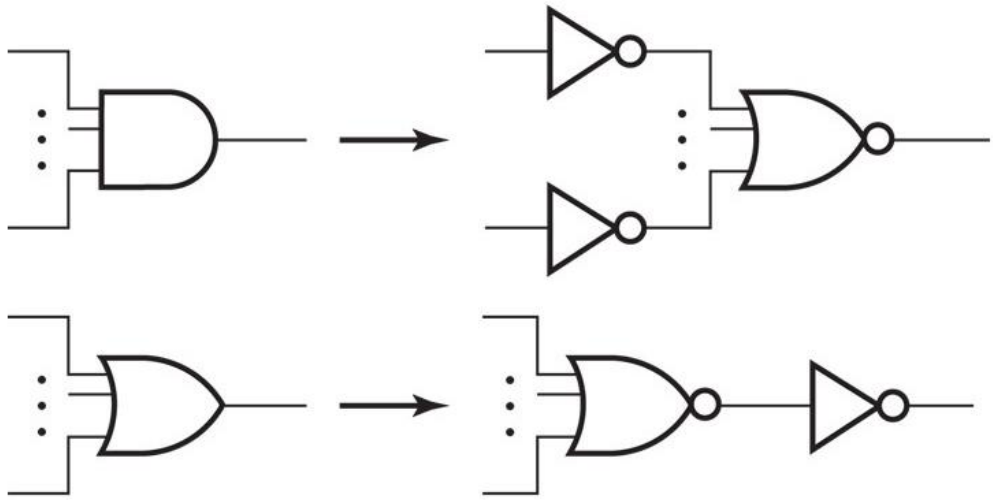
(a) Mapping to NAND gates

NAND are functionally complete:
Any Boolean function can be transformed to a circuit contains only NAND gates and inverters

$$X \cdot Y = \overline{\overline{X \cdot Y}}$$

$$X + Y = \overline{\overline{X + Y}} = \overline{\overline{X} \cdot \overline{Y}}$$

Technology Mapping: Hints



(b) Mapping to NOR gates

In a dual way, any logic circuit can be transformed to a circuit that containing only NOR gates and inverters

$$X \cdot Y = \overline{\overline{X \cdot Y}} = \overline{\overline{X} + \overline{Y}}$$

$$X + Y = \overline{\overline{X + Y}}$$

- It is common practice to design circuits containing only NAND and NOT gates (or only NOR and NOT): the less expensive implementation is chosen, based on the cost (number of gates, inputs, delays)

Combinational Logic Blocks

Single-bit Functions

- Single-bit functions are functions of one variable
- **Value-fixing**: a constant value ('1' or '0') is assigned to the output, regardless of the value of the input variable
- **Value-transferring**: the input value is transferred to the output
- **Value-inverting**: the complement of the input value is transferred to the output

- Truth table:

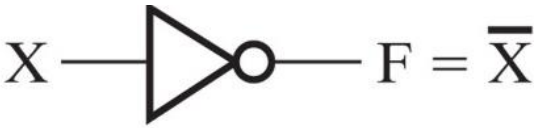
X	F = 0	F = X	F = $\overline{X}$	F = 1
0	0	0	1	1
1	0	1	0	1

Single-bit Functions: Implementation

- **Value-fixing**: the output is connected to a constant logic value '0' or '1'

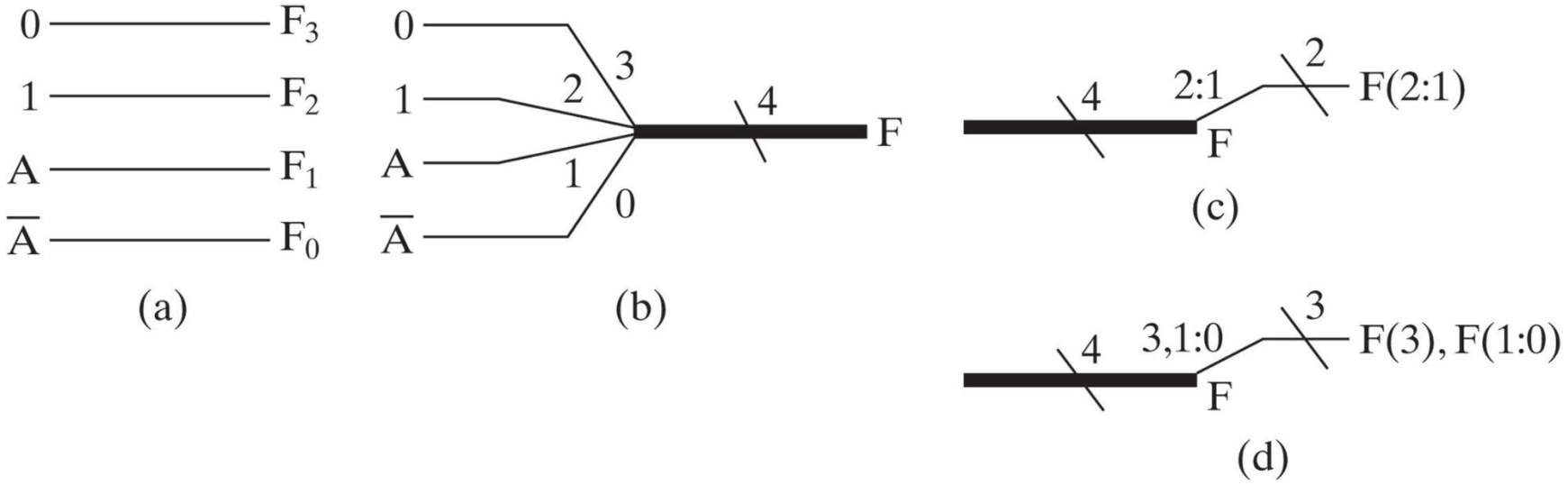
- **Value-transferring**: the output is directly connected to the input

- **Value-inverting**: the output is connected to the input through an inverter



Multiple-bit Functions

- Multi-bit functions can be thought as **vectors of single-bit functions**
- The notation to represent vectors in logic circuits is a slash on the wire indicating the number of bits of the vector
- If only a few bits are transferred (subvector), the index of their position in the vector is specified, see Figs. (c) and (d)



Enabling

- An enabling circuit
  - Allows the signal (X) to reach the output if ENABLE (EN) is '1', and
  - Connects a fixed value ('0' or '1') to the output if ENABLE is '0'
- Enabling is typically realized with one of these solutions:

(a)

F = X if EN = '1'

F = '0' if EN = '0'

(b)

F = X if EN = '1'

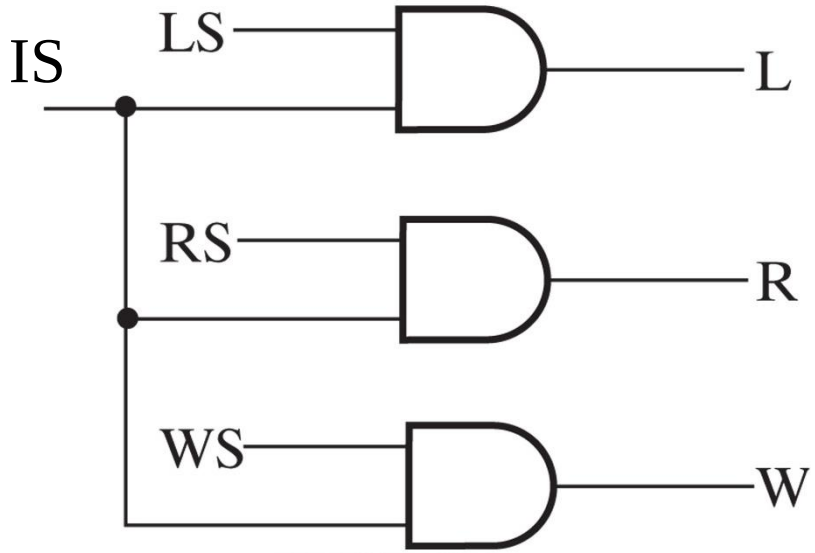
F = '1' if EN = '0'

Example 3.5: Enabling

- Design of a car electrical control using enabling
  - Lights, radio, and windows are enabled only when the car is on
  - The ignition switch (IS) acts as the ENABLE signal
    - Ignition switch (IS): to switch on the car '0': OFF, '1': ON
    - Light switch (LS): to switch on the lights '0': OFF, '1': ON
    - Radio switch (RS): to switch on the radio '0': OFF, '1': ON
    - Window switch (WS): to operate electric windows '0': OFF, '1': ON
    - Lights (L): if '0' lights are off, if '1' lights are on
    - Radio (R): if '0' radio is off, if '1' radio is on
    - Power windows (W): if '0’ power windows is idle, if '1' power windows is active

Example 3.5: Enabling

Input Switches				Accessory Control		
IS	LS	RS	WS	L	R	W
0	X	X	X	0	0	0
1	0	0	0	0	0	0
1	0	0	1	0	0	1
1	0	1	0	0	1	0
1	0	1	1	0	1	1
1	1	0	0	1	0	0
1	1	0	1	1	0	1
1	1	1	0	1	1	0
1	1	1	1	1	1	1



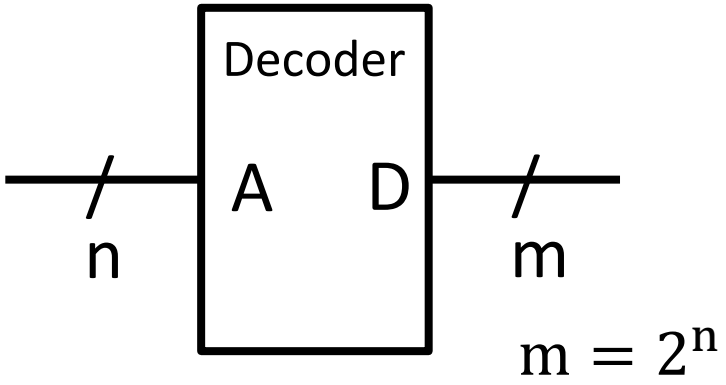
➔ IS = '1': the devices (radio, lights, windows) are controlled by the corresponding switch

➔ IS = '0': all devices are off, regardless of the status of the corresponding switch. The status of the 3 switches is indicated with an X in the truth table (X represents a don't care input, i.e. a product term that is not a minterm. For instance, the first line 0XXX represents the product term $\overline{I}S$)

Decoder

Decoder

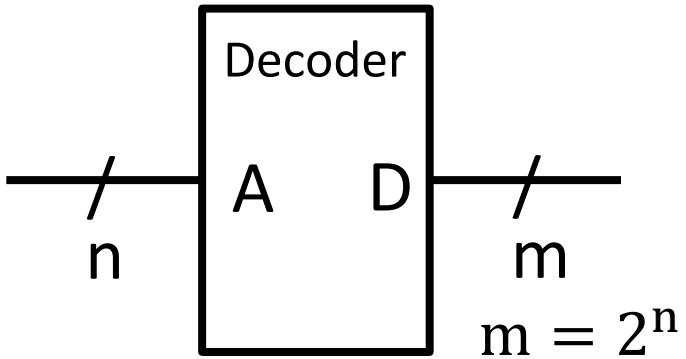
- A decoder is a combinational block that implements a **conversion from one code to another code**
- A **n-to-m decoder converts an n-bit input to an m-bit output representing its 1-hot decoding** (i.e., a binary number containing a single '1' in the position encoded by the input bits)
  - Example: Input 3 (11), Output 1000
- In other words, the decoder sets to '1' the output whose index is encoded by input bits



- 2^n values can be represented with n bits: $m = 2^n$ if all the input combinations are used

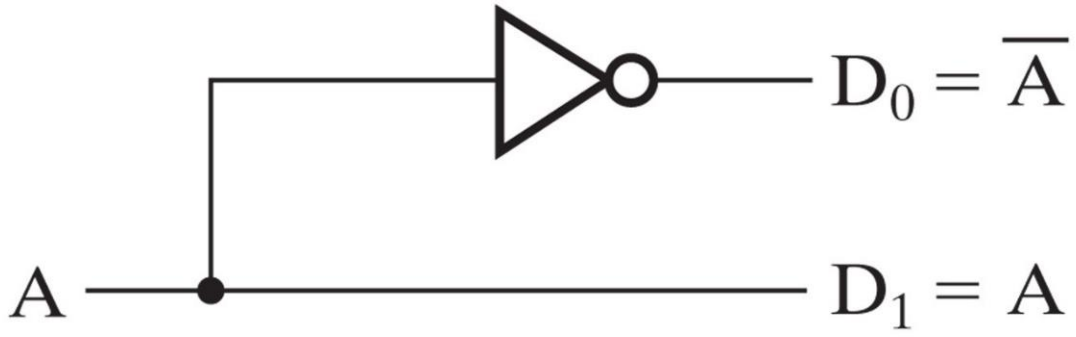
One-hot Coding

Binary code (n bits)	1-hot code (2^n bit)
000	00000001
001	00000010
010	00000100
011	00001000
100	00010000
101	00100000
110	01000000
111	10000000



1-to-2 Decoder (n = 1, m = 2)

A	D_0	D_1
0	1	0
1	0	1

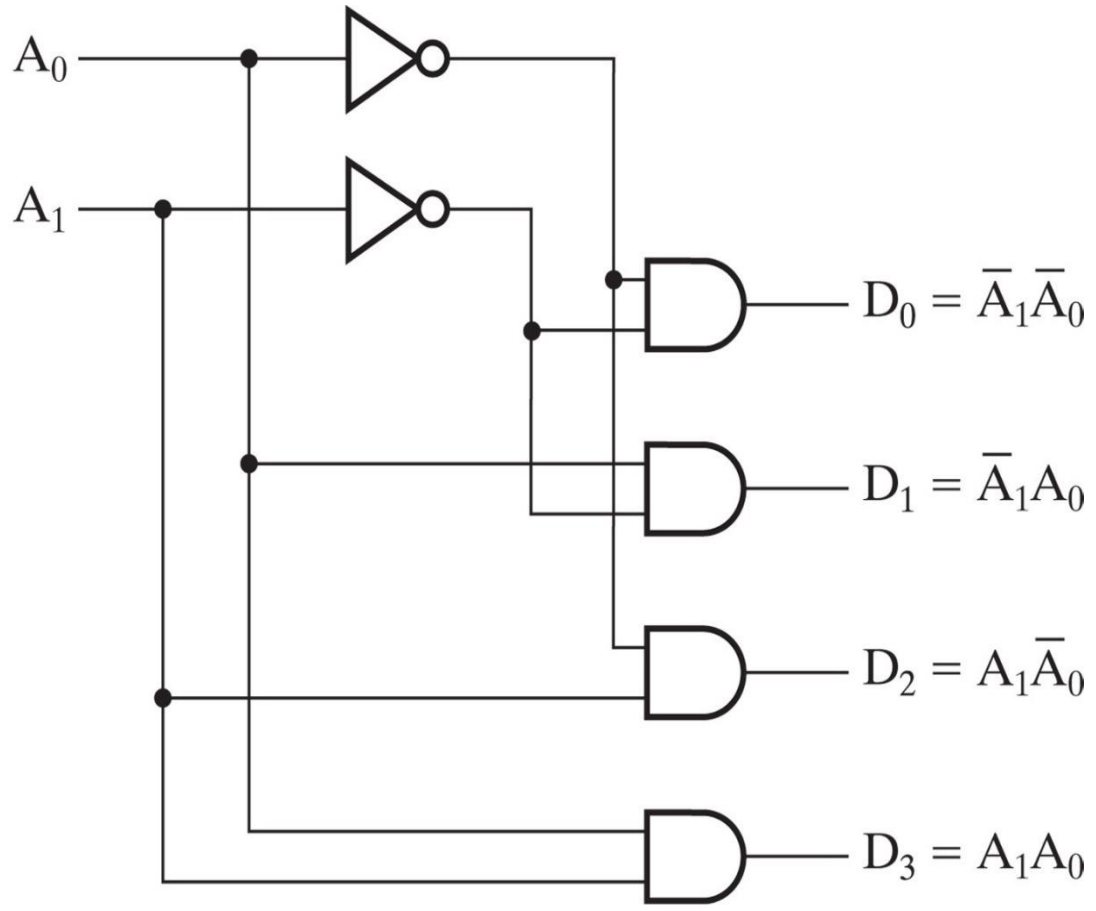


→ The outputs of an **n-to-m decoder are the 2^n minterms corresponding to the input variables**

The decoder is a logic block in charge of generating the 2^n minterms from the n input variables!

2-to-4 Decoder (n = 2, m = 4)

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

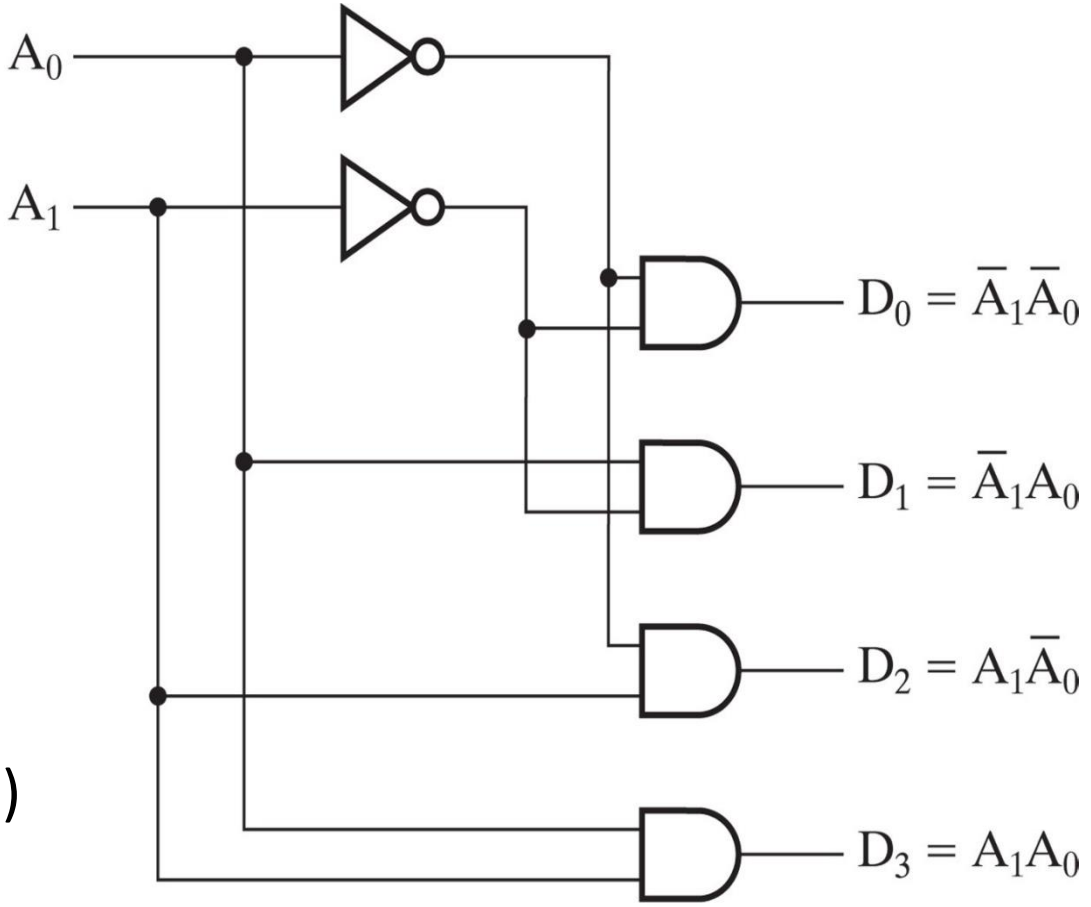


An **n-to-m decoder** can be implemented with *m* AND gates with *n* inputs.

The inputs of each of the AND gates are a proper combination of the A_i inputs (direct or complemented)

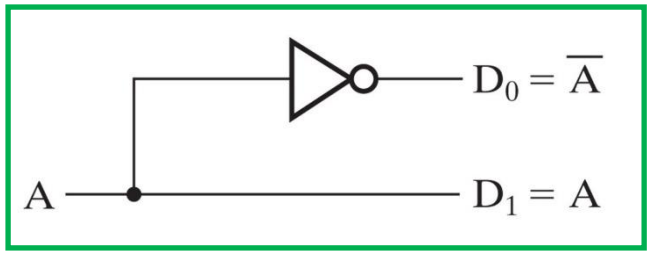
2-to-4 Decoder (n = 2, m = 4)

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 2-to-4 decoder can be seen as a **system consisting of: 2 *1-to-2 decoders* and 4 *AND gates*** (with the 4 combinations of the 2 decoders' outputs connected to the inputs of the AND gates)

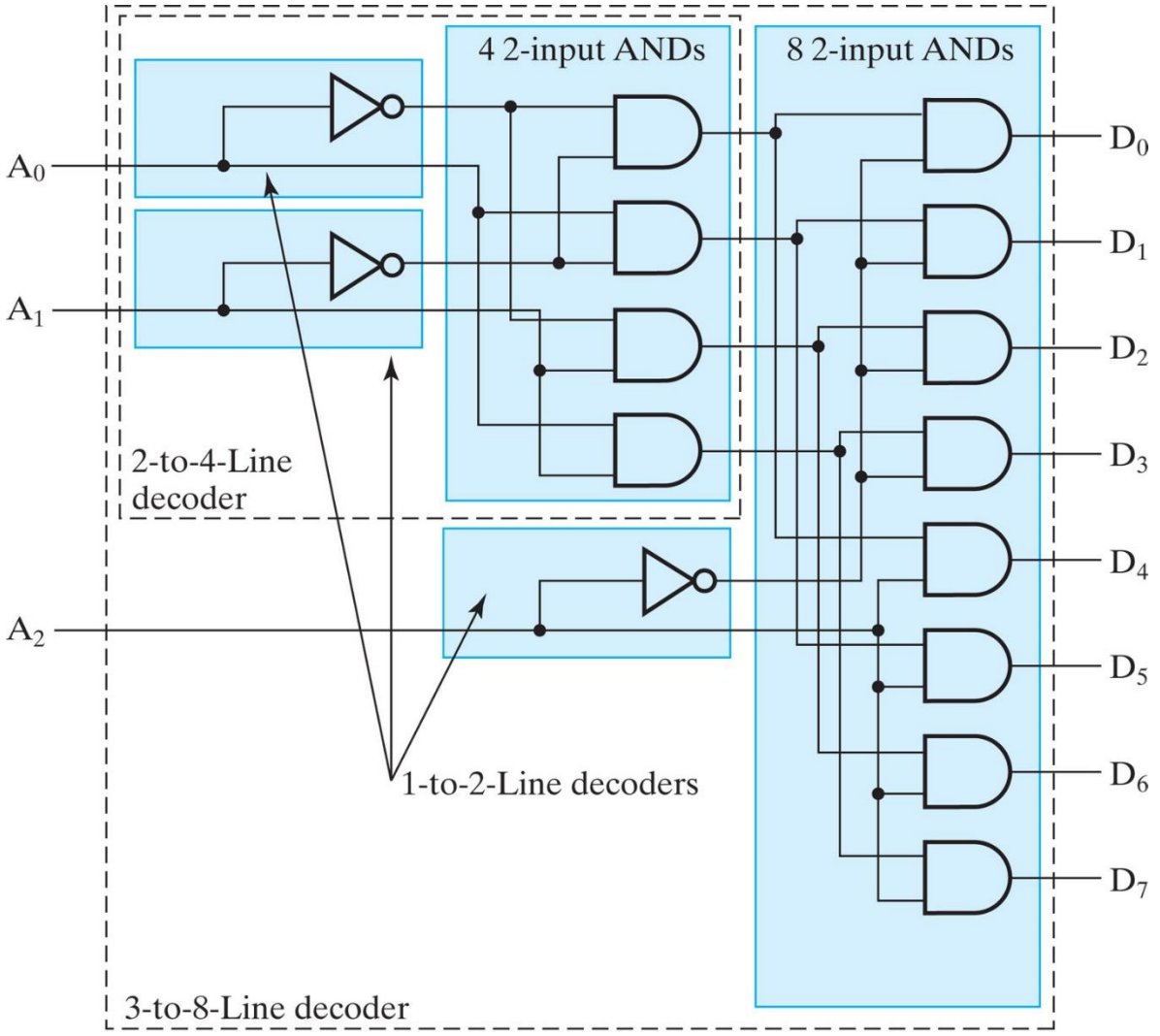
1-to-2 decoder



Decoder: Hierarchical Approach

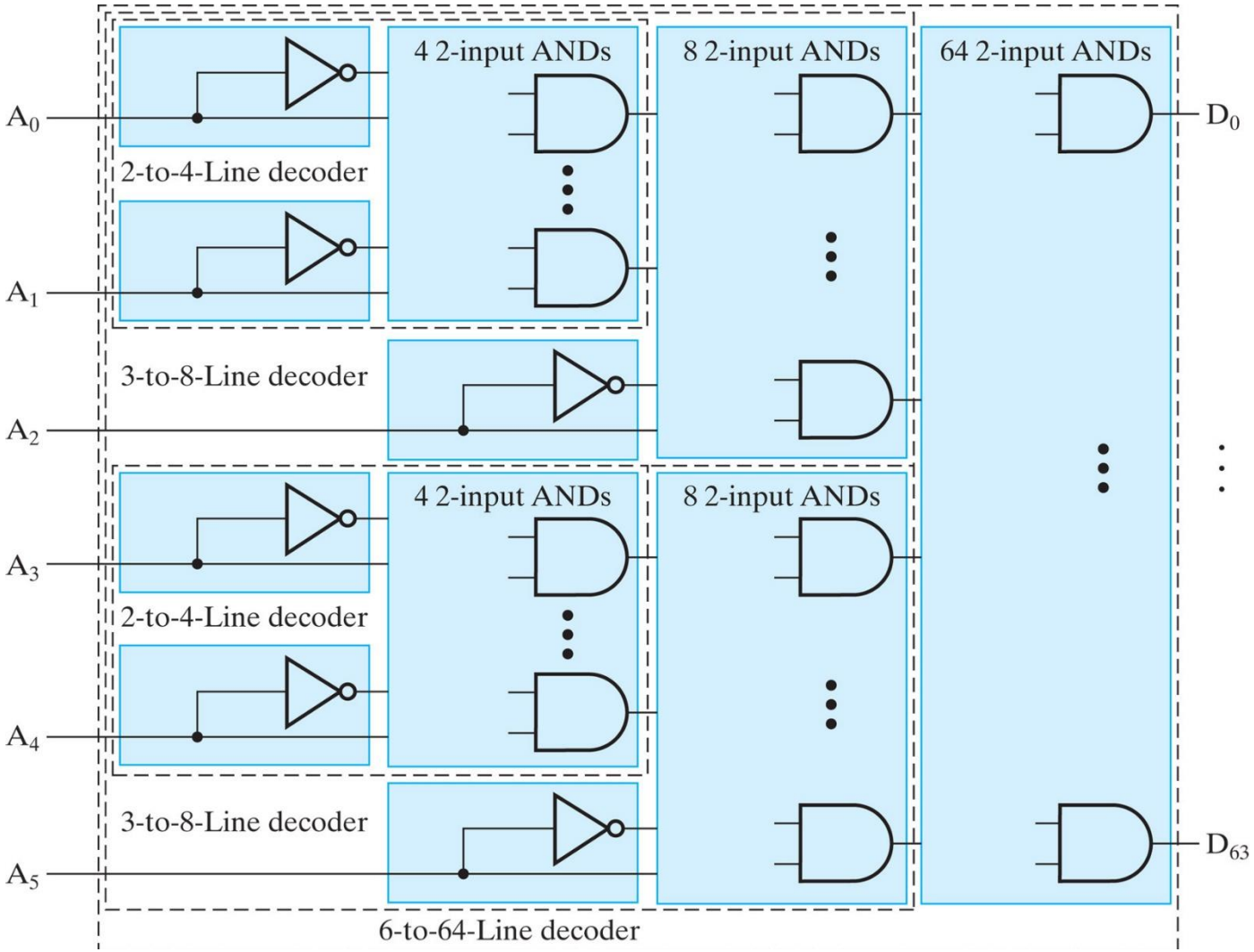
- As the decoder size grows, the approach just seen becomes more and more expensive (in terms of number of AND gates input cost)
- For larger decoders, a hierarchical approach is used, connecting smaller decoders together
- Examples
  - A *3-to-8 decoder* can be implemented with 1 *2-to-4 decoder* and 1 *1-to-2 decoder* feeding 8 2-input *AND gates*
  - A *6-to-64 decoder* can be implemented with 2 *3-to-8 decoder*, feeding 64 2-input *AND gates*
  - Different combinations can be choosen for a hierarchical implementation, with a variable number of stages in cascade, depending on what needs to be optimized (number of logic gates/ number of AND gate inputs)

3-to-8 Decoder



Logic diagram of a 3-to-8 decoder with hierarchical structure

6-to-64 Decoder



Hierarchical diagram of a 6-to-64 decoder

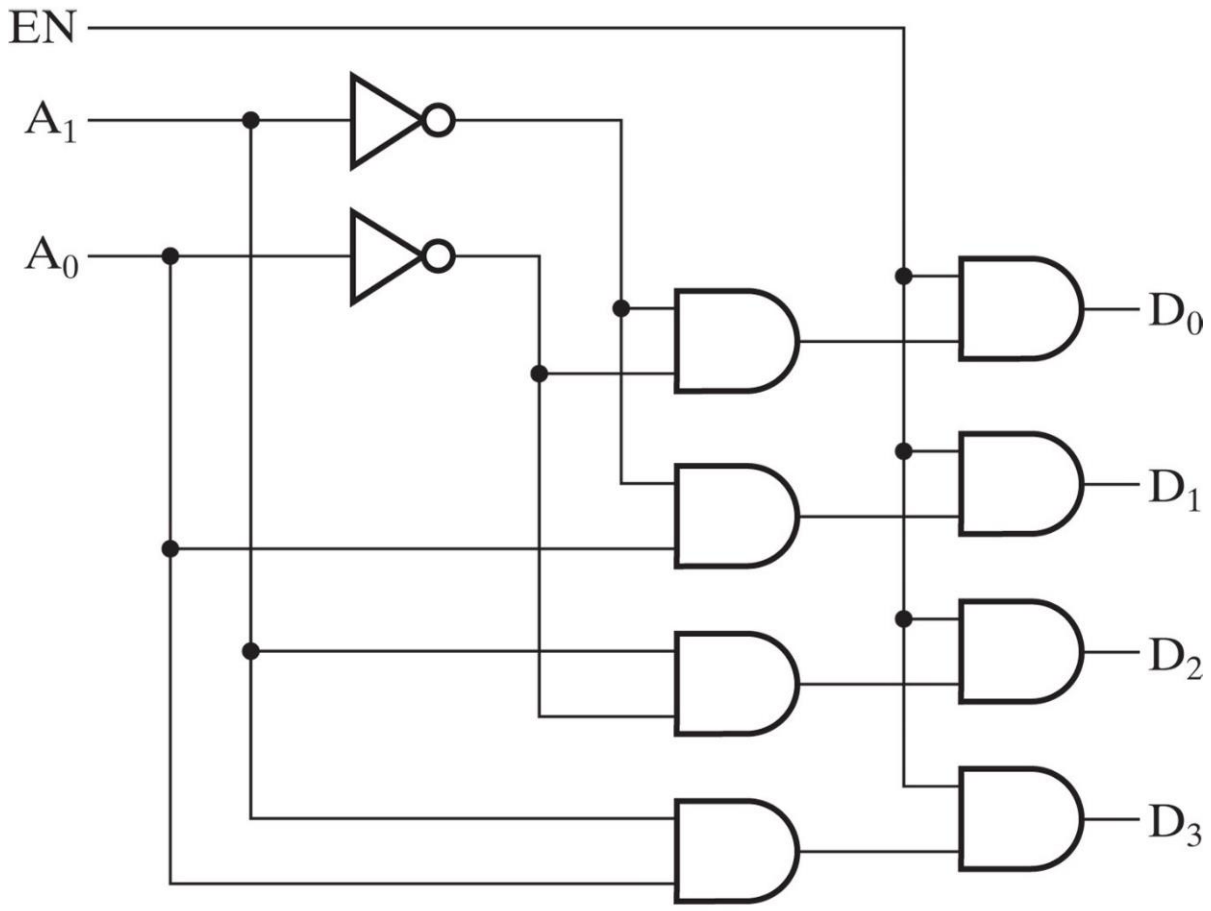
The gate input cost is 182
(4+2+4*2*2+8*2*2+64*2)

Decoder with Enable

- The enabling circuit allows us to enable or disable the decoder outputs
- A decoder with enabling circuit can be implemented by connecting **m circuits of enable to the decoder outputs**
- For large decoders ($n \geq 4$), the minimum cost solution (in terms of gate input number) is to directly connect the enable circuit to the decoder inputs rather than to the outputs

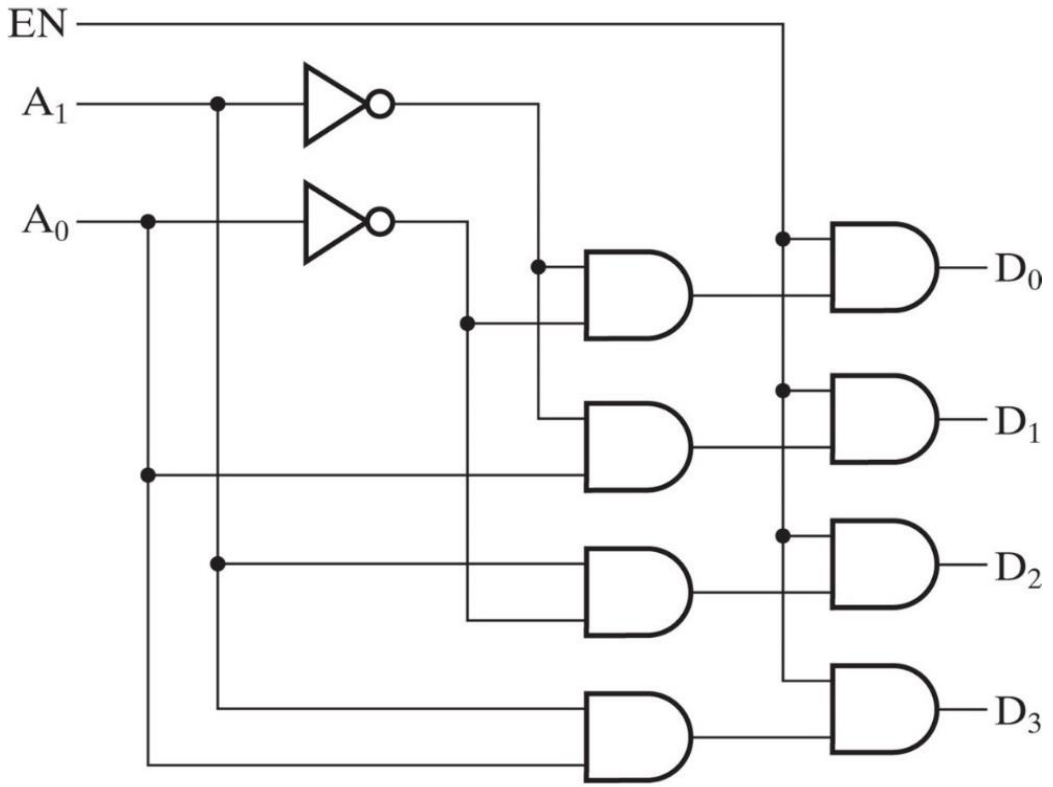
2-to-4 Decoder with Enable

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



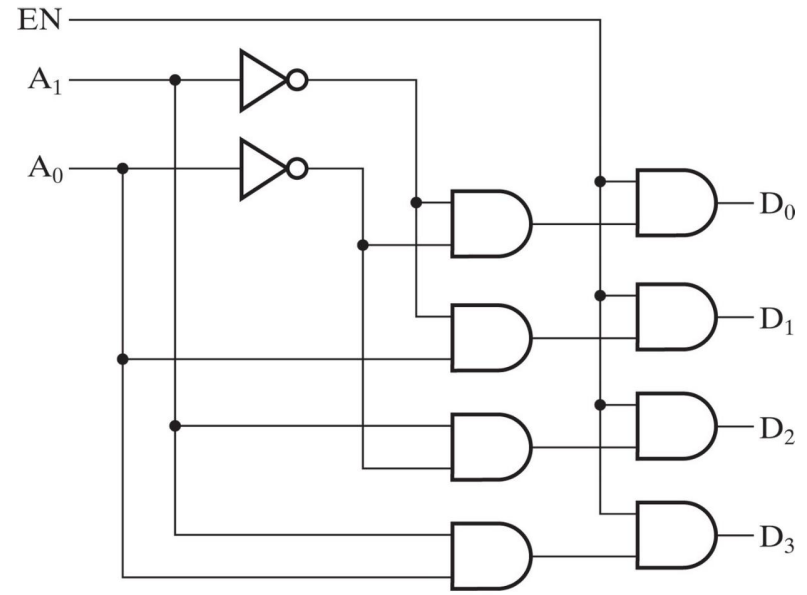
2-to-4 Decoder with Enable: VHDL Entity

```
-- 2-to-4-Line Decoder with Enable: Structural VHDL Description
-- (See Figure 3-16 for logic diagram)
library ieee, lcdf_vhdl;
use ieee.std_logic_1164.all, lcdf_vhdl.func_prims.all;
entity decoder_2_to_4_w_enable is
   port (EN, A0, A1: in std_logic;
         D0, D1, D2, D3: out std_logic);
end decoder_2_to_4_w_enable;
```



2-to-4 Decoder with Enable: Structural Architecture

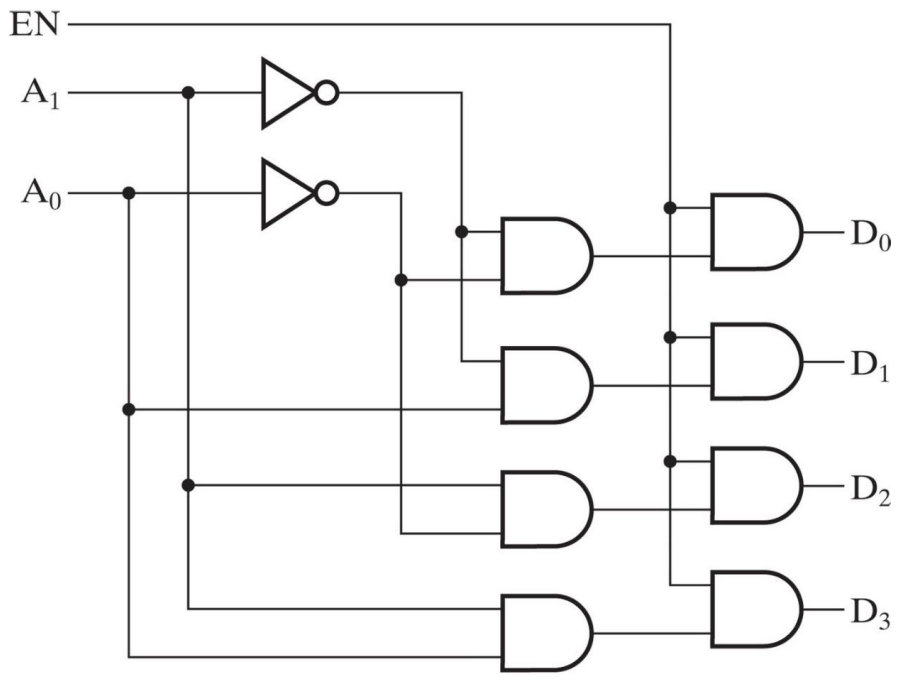
```
architecture structural_1 of decoder_2_to_4_w_enable is
   component NOT1
      port (in1: in std_logic;
            out1: out std_logic);
   end component;
   component AND2
      port (in1, in2: in std_logic;
            out1: out std_logic);
   end component;
   signal A0_n, A1_n, N0, N1, N2, N3: std_logic;
   begin
      g0: NOT1 port map (in1 => A0, out1 => A0_n);
      g1: NOT1 port map (in1 => A1, out1 => A1_n);
      g2: AND2  port map  (in1 => A0_n, in2 => A1_n, out1 => N0);
      g3: AND2  port map  (in1 => A0, in2 => A1_n, out1 => N1);
      g4: AND2  port map  (in1 => A0_n, in2 => A1, out1 => N2);
      g5: AND2 port map   (in1 => A0, in2 => A1, out1 => N3);
      g6: AND2 port map   (in1 => EN, in2 => N0, out1 => D0);
      g7: AND2 port map   (in1 => EN, in2 => N1, out1 => D1);
      g8: AND2 port map   (in1 => EN, in2 => N2, out1 => D2);
      g9: AND2 port map   (in1 => EN, in2 => N3, out1 => D3);
end structural_1;
```



2-to-4 Decoder with Enable: Dataflow Architecture

```
architecture dataflow_1 of decoder_2_to_4_w_enable is

signal A0_n, A1_n: std_logic;
begin
    A0_n <= not A0;
    A1_n <= not A1;
    D0 <= A0_n and A1_n and EN;
    D1 <= A0 and A1_n and EN;
    D2 <= A0_n and A1 and EN;
    D3 <= A0 and A1 and EN;
end dataflow_1;
```



More compact than the structural description: dataflow description does not use components, but uses the predefined operators to describe the circuit in the form of Boolean equations

Combinational Circuits Using the Decoder

- We have seen that the decoder, given n bits as an input, provides the corresponding 2^n minterms as an output
- All logic functions can be expressed in SOP form (sum of minterms)

Any logic function can be realized with a decoder plus an OR gate!

(the decoder provides the minterms, the OR gate implements the sum)

Example: 1-bit Binary Adder with Decoder

Truth Table for 1-Bit Binary Adder

	X	Y	Z	C	S
m_0	0	0	0	0	0
m_1	0	0	1	0	1
m_2	0	1	0	0	1
m_3	0	1	1	1	0
m_4	1	0	0	0	1
m_5	1	0	1	1	0
m_6	1	1	0	1	0
m_7	1	1	1	1	1

- Adder with 3 inputs (1 bit)
- C: carry
- S: sum

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

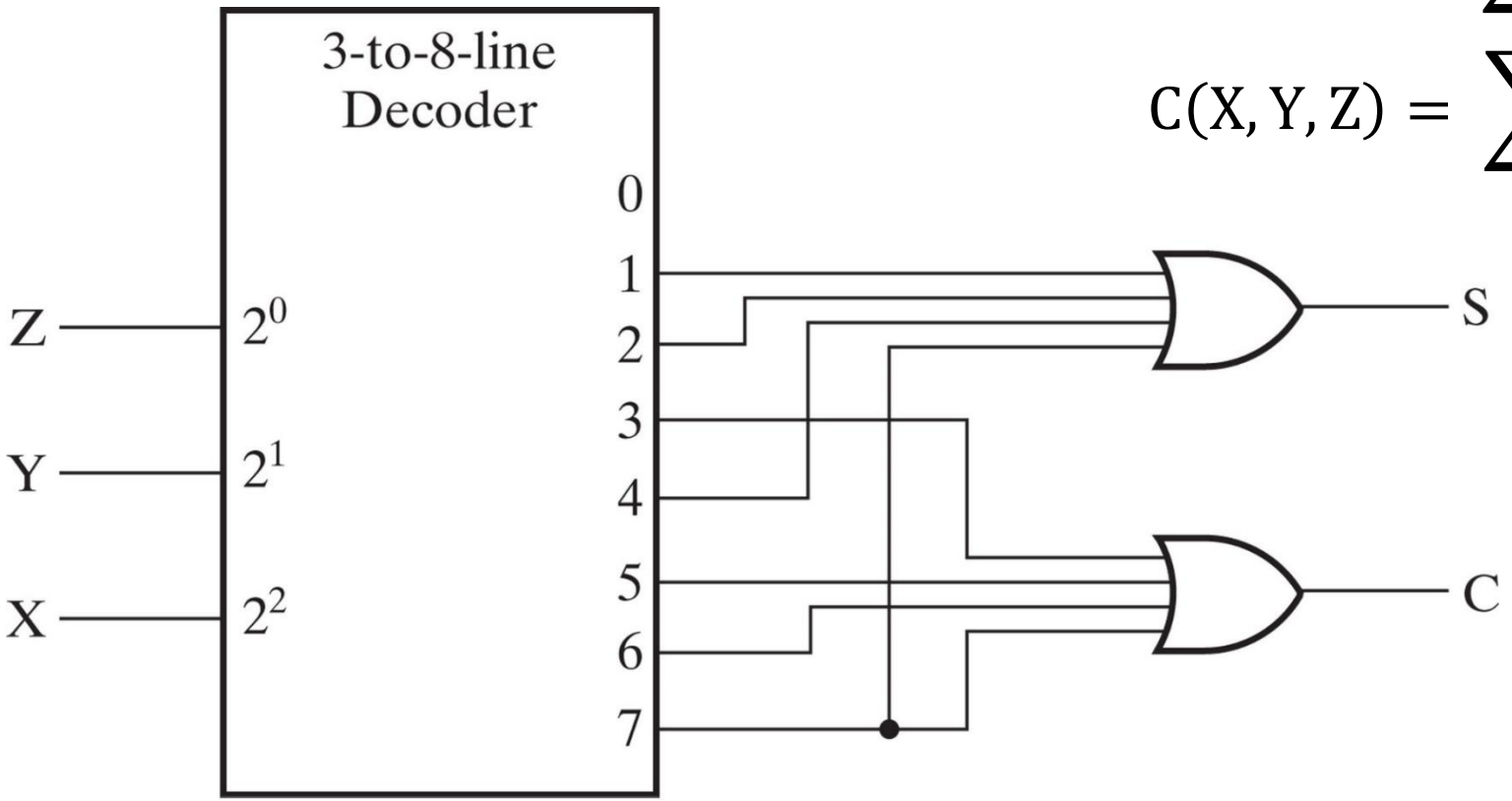
$$S(X, Y, Z) = \sum m(1, 2, 4, 7)$$

Example: 1-bit Adder with Decoder

The decoder generates all minterms, the 2 OR gates provides the sum of the minterms needed for the two outputs S and C

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

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



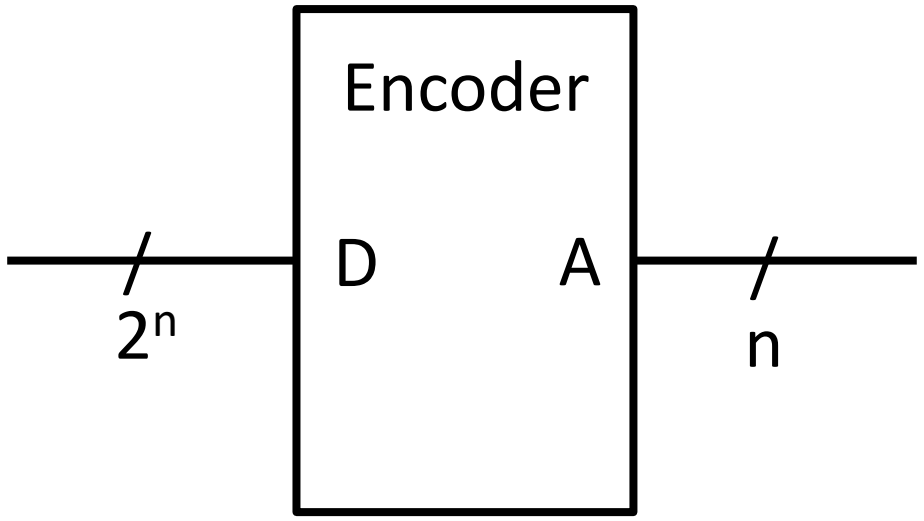
Combinational Circuits using a Decoder

- The implementation of a logic function with decoder and OR gates is convenient if the function can be expressed with a small number of minterms
- In the presence of a large number of minterms, the number of inputs of the OR gate (then the cost) increases and it may be convenient to replace the OR gate with a NOR gate having the minterms of the complemented function as inputs

Encoder

Encoder

- The encoder performs the inverse function of a decoder
- An encoder has a **2^n-bit input** and an **n-bit output**
- We assume that the **input is 1-hot** (only one input at a time can be equal to '1'), the **output is the binary code corresponding to the input**

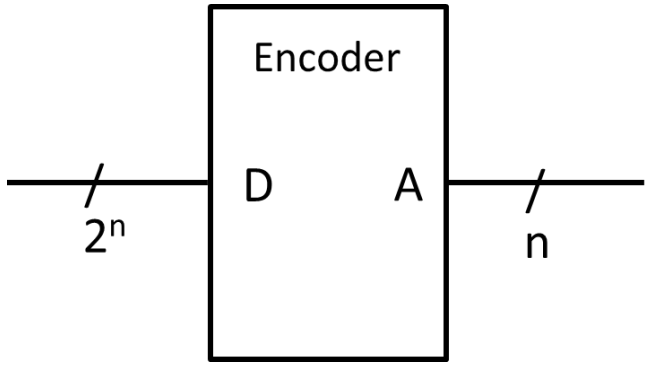


Octal to Binary Encoder

Truth Table for 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

Input: 1-hot code corresponding to octal digits 0 to 7
Output: 3-bit binary code corresponding to the input



Octal to Binary Encoder

Truth Table for 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

Only one input at a time can be equal to '1': the truth table has 8 lines.
For the remaining lines (not shown in the table), the outputs are don't care

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

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

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

Encoder: Issues

Truth Table for 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

If two inputs are simultaneously equal to '1', the circuit produces a wrong output.
Example: $D_3 = D_6 =$ '1' produces $A_2\ A_1\ A_0 =$ "111" (same output corresponding to $D_7 =$ '1')

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

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

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

Encoder: Issues

Truth Table for 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

Other issue: if all inputs are equal to '0', the output is "000" (same output corresponding to D_0= 1 and D_1 =D_2 = ... D_7 = 0)

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

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

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

Priority Encoder

- The priority encoder can manage the situations in which the input is not a 1-hot input:

1) A **priority function** is introduced **to prevent more than one input from taking the value of '1'**
   - Only the highest priority '1' is considered: if more than one input is equal to '1', only the input with higher priority determines the value of the output (independent of the other inputs)

2) An additional output V is added (V = valid) **to prevent all inputs from being equal to '0'**. V is set to '1' when at least one of the inputs is '1'. If all inputs are '0', V is set to '0' and the output is set to "don't care"

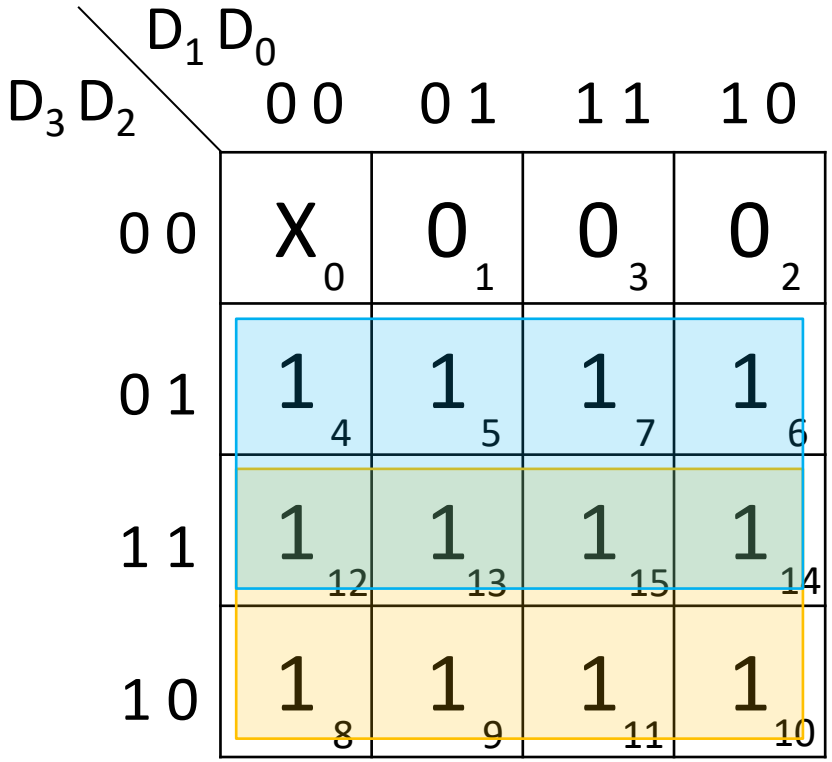
4-input Priority Encoder

Truth Table of Priority Encoder

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

- Input D_3 has the **highest priority**: when D_3 is '1', the other inputs are not considered (regardless of their value, the output is set to "11", i.e. the binary code corresponding to 3)
- After D_3, input D_2 has the next priority level: if D_3 is '0' and D_2 is '1', the output is set to "01" (binary code corresponding to '2'), regardless of D_1 and D_0
- The output V indicates if the output A is valid. V is equal to '0' if all inputs are '0'

4-input Priority Encoder: K-map

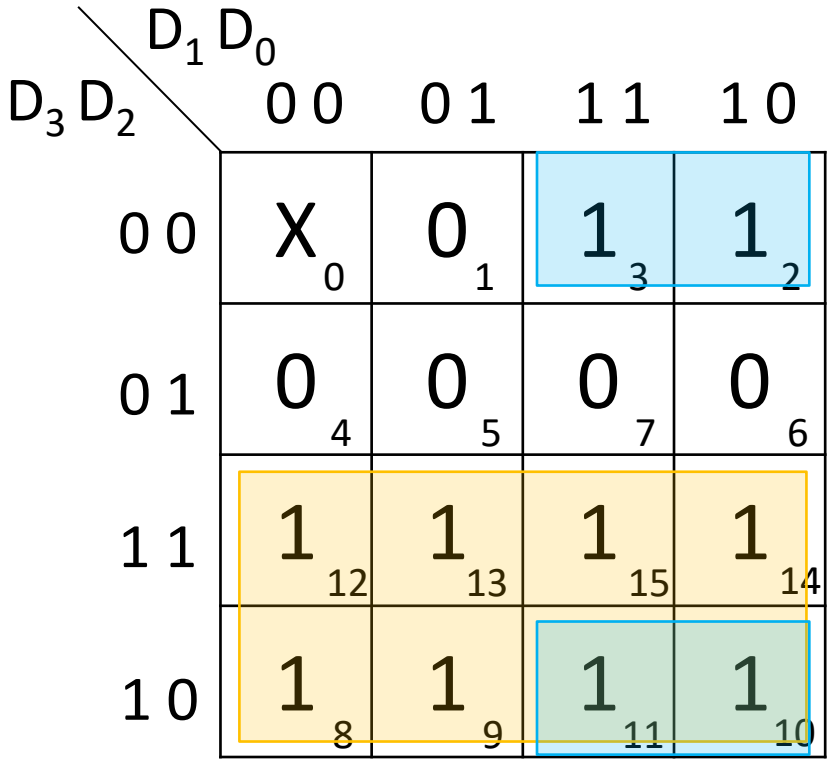


Truth Table of Priority Encoder

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

$$A_1 = D_2 + D_3$$

4-input Priority Encoder: K-map



Truth Table of Priority Encoder

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

$$A_0 = D_3 + \overline{D_2}\, D_1$$

4-input Priority Encoder: K-map

$D_3 D_2$ \ $D_1 D_0$	0 0	0 1	1 1	1 0
0 0	0_0	1_1	1_3	1_2
0 1	1_4	1_5	1_7	1_6
1 1	1_{12}	1_{13}	1_{15}	1_{14}
1 0	1_8	1_9	1_{11}	1_{10}

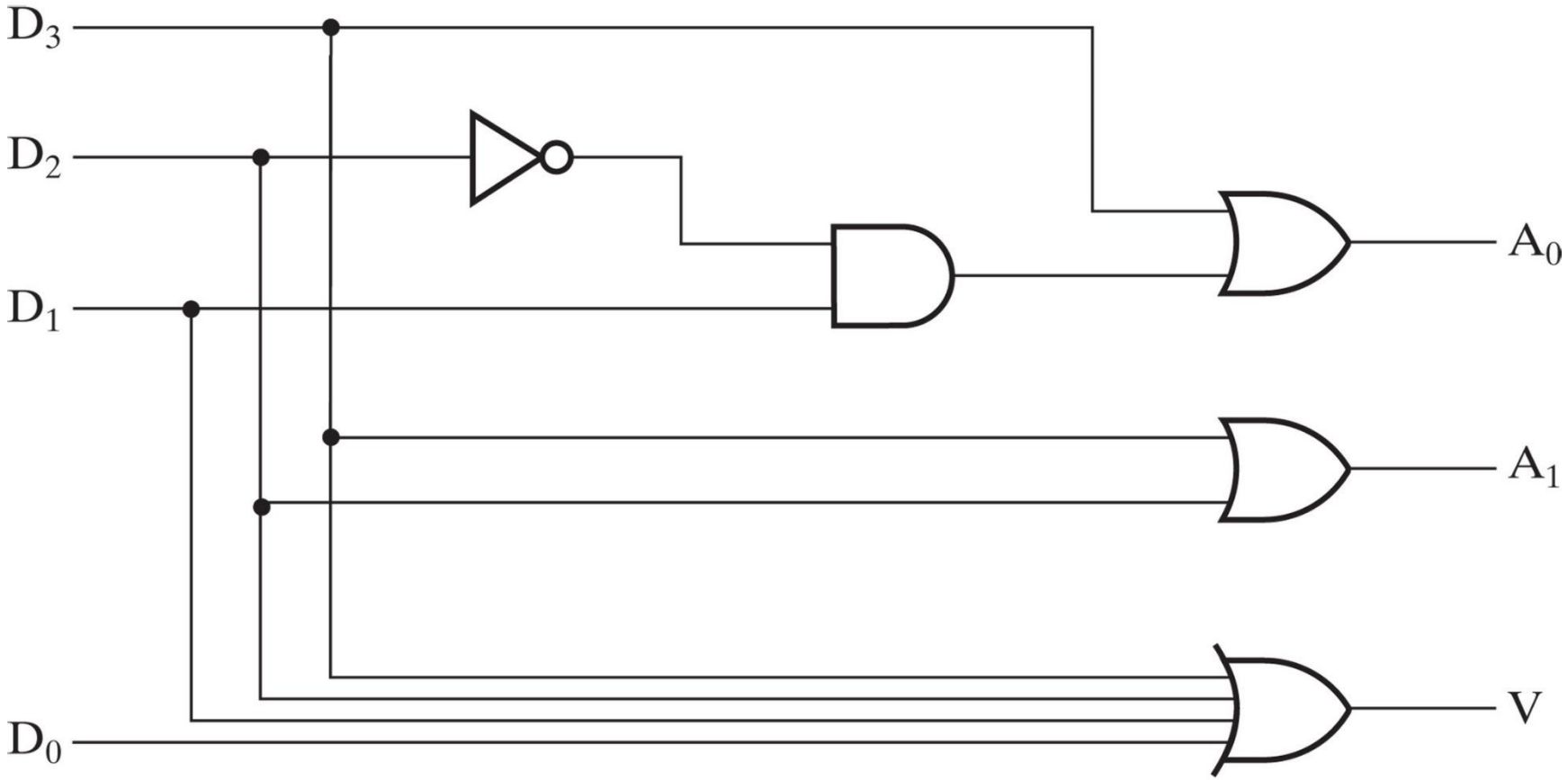
Truth Table of Priority Encoder

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

$$\overline{V} = \overline{D_3} \cdot \overline{D_2} \cdot \overline{D_1} \cdot \overline{D_0}$$

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

4-input Priority Encoder: Circuit



$$A_1 = D_2 + D_3$$

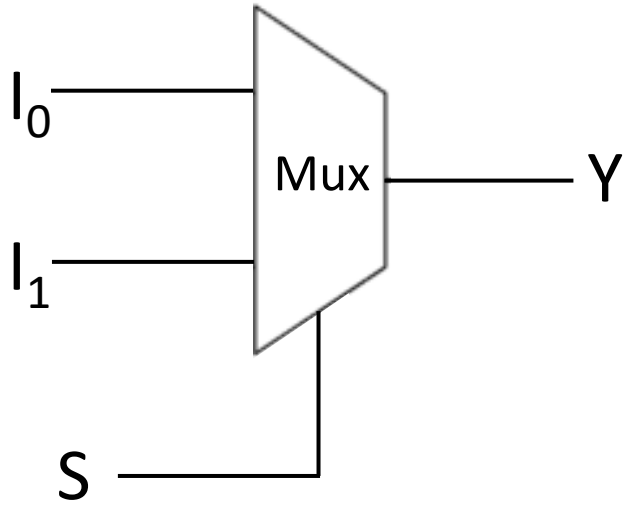
$$A_0 = D_3 + \overline{D_2}\, D_1$$

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

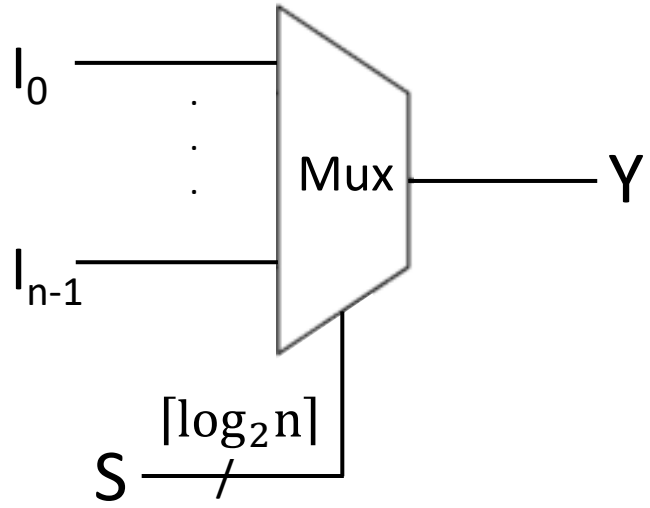
Multiplexer

Multiplexer

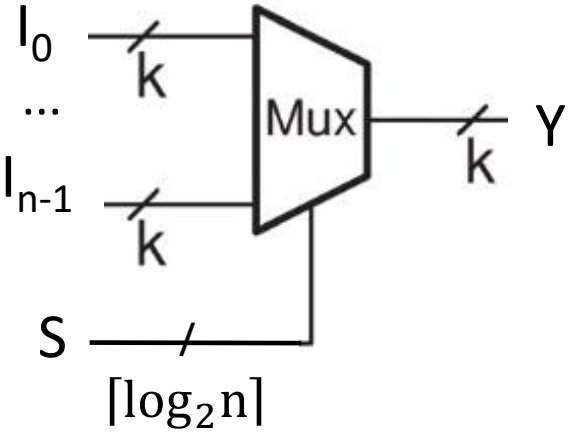
- The multiplexer (MUX) is a combinational block used for **data selection**
- The multiplexer **selects one between several input lines**, **based on the value of a selection signal**, and directs the data to a single output line. The selection input can be a 1-hot signal or a binary signal



Single-bit 2-to-1 Mux



Single-bit n-to-1 Mux

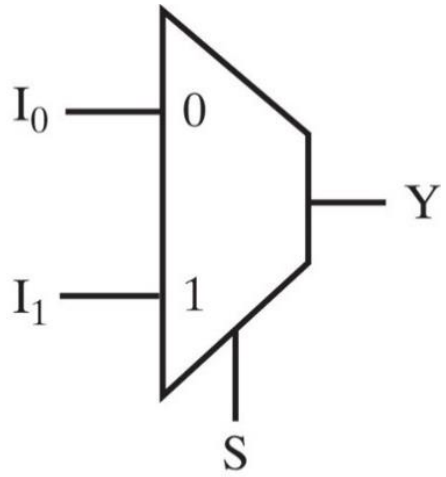


k-bit n-to-1 Mux

2-to-1 Multiplexer

Truth Table for 2–to–1-Line Multiplexer

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



n=1

Data input: $2^n = 2$

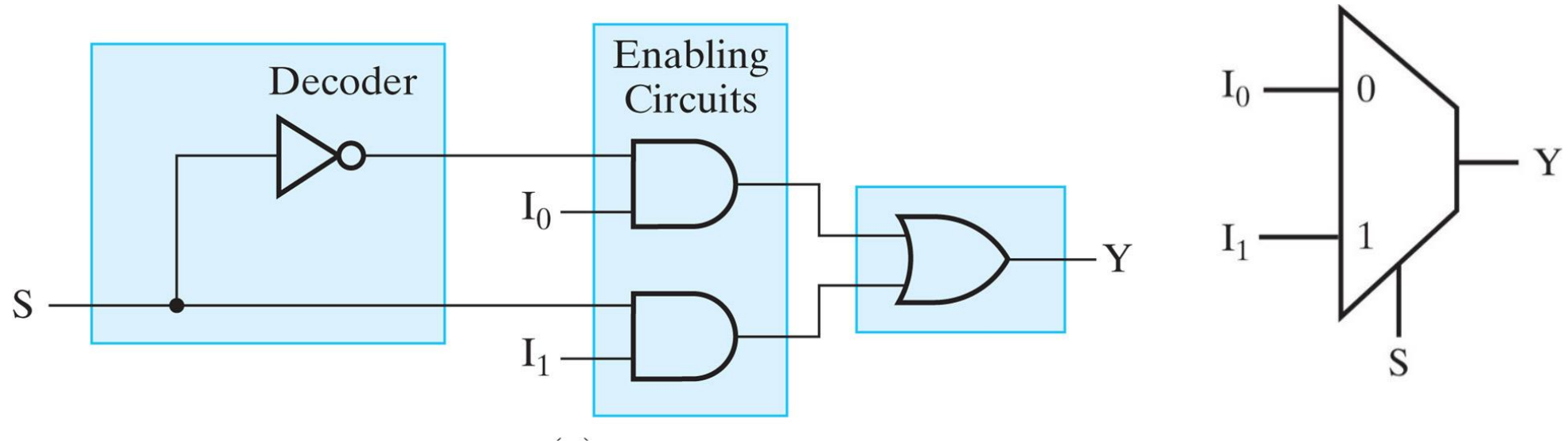
Selection input: 1

If S = '1': I_1 is selected

if S = '0': I_0 is selected

$$Y = \bar{S}\, I_0 + S\, I_1$$

2-to-1 Multiplexer: Implementation



- A 2-to-1 mux can be implemented connecting the output of a 1-to-2 decoder with two enabling circuits followed by a 2-input an OR gate
  - The decoder generates the minterms of the selection inputs. The AND gates provide enable circuits which transmit one of the inputs I_i to the OR gate

4-to-1 Multiplexer

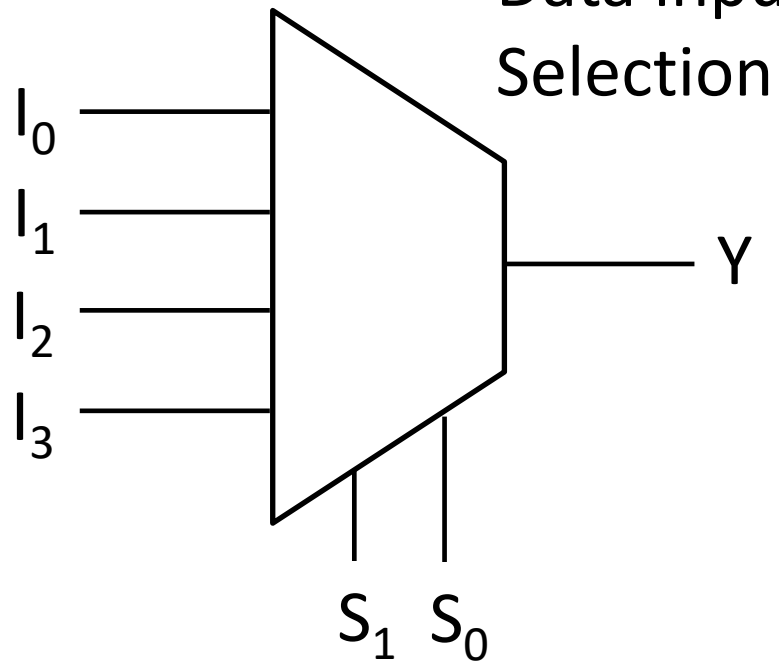
Condensed Truth Table for 4-to-1-Line Multiplexer

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

n=2

Data inputs: $4 = 2^2$

Selection inputs: 2

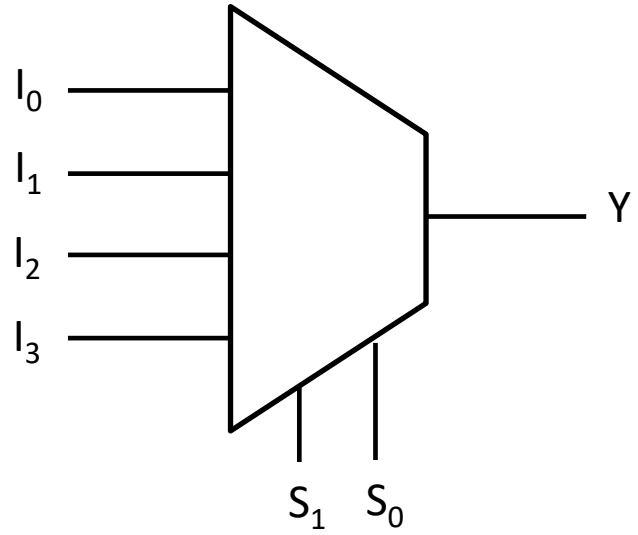


The implemented function is:

$$Y = \overline{S_1}\,\overline{S_0}\ I_0 + \overline{S_1} S_0\ I_1 + S_1 \overline{S_0}\ I_2 + S_1 S_0 I_3$$

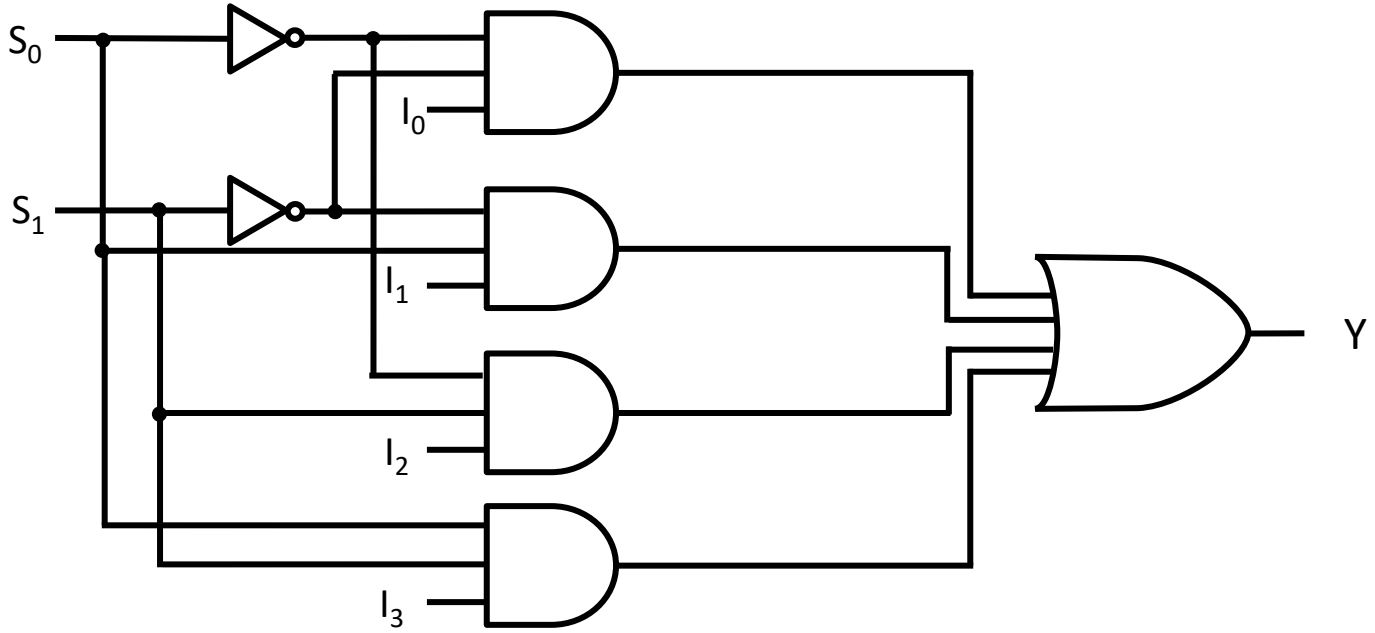
4-to-1 Multiplexer: Implementation 1)

$$Y = \overline{S_1}\,\overline{S_0}\ I_0 + \overline{S_1}S_0\ I_1 + S_1\overline{S_0}\ I_2 + S_1 S_0 I_3$$



The function can be implemented in different ways:

1) With 2 *inverters*, 4 *3-input AND gates* and 1 *4-input OR gates*: **gate input cost = 18**

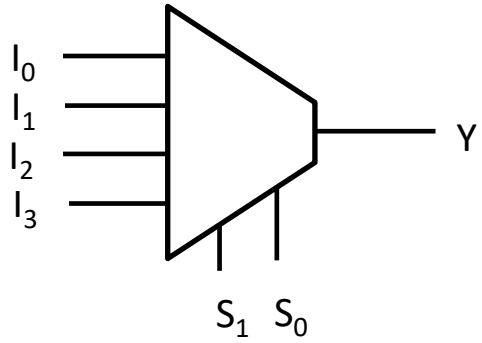


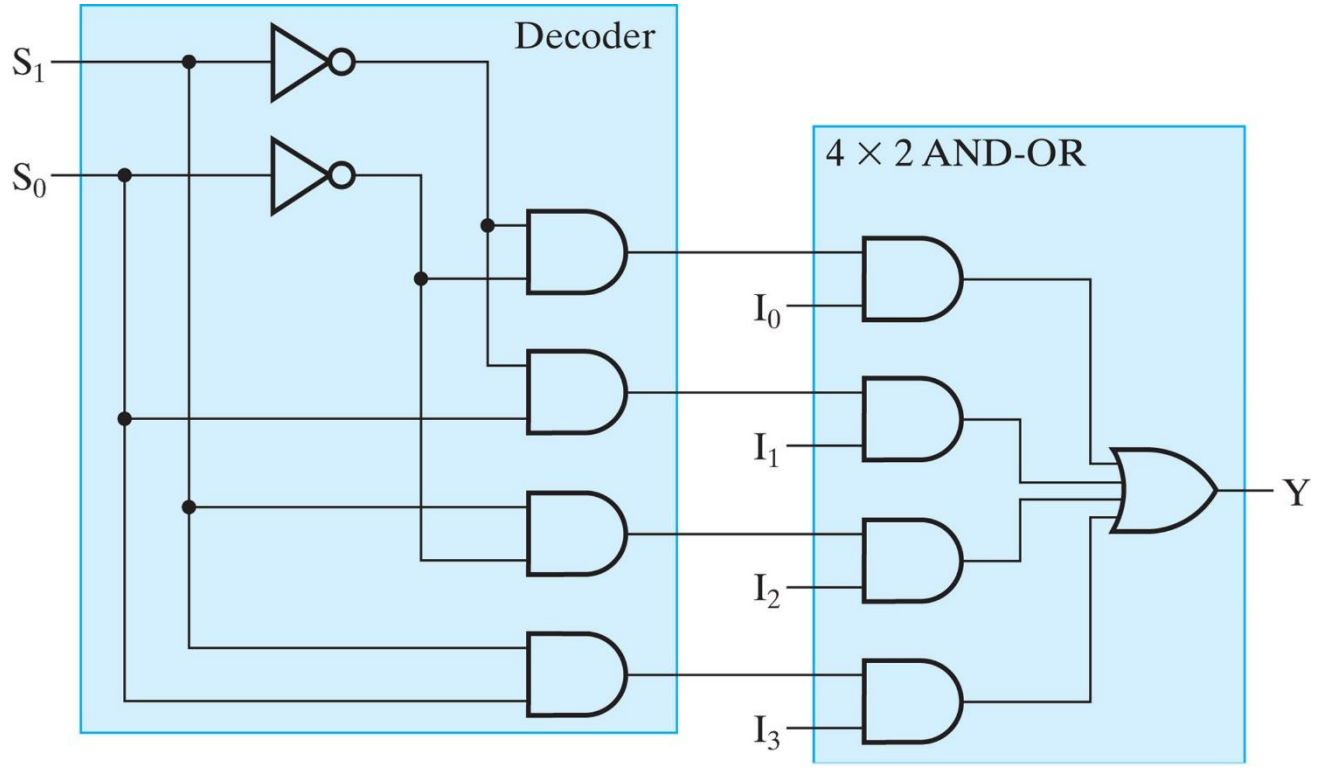
4-to-1 Multiplexer: Implementation 2)

$$Y = \overline{S_1}\,\overline{S_0}\ I_0 + \overline{S_1}S_0\ I_1 + S_1\overline{S_0}\ I_2 + S_1S_0I_3$$

Associative property:

$$Y = (\overline{S_1}\,\overline{S_0})\ I_0 + (\overline{S_1}S_0)\ I_1 + (S_1\overline{S_0})\ I_2 + (S_1S_0)\ I_3$$





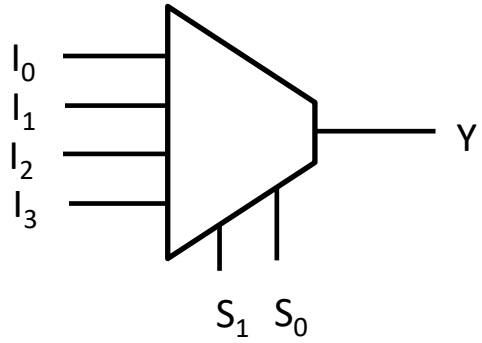
2) A *2-to-4 decoder* followed by 4 *AND gates* (enable circuit) and 1 *4-input OR gate*: **gate input cost = 22**

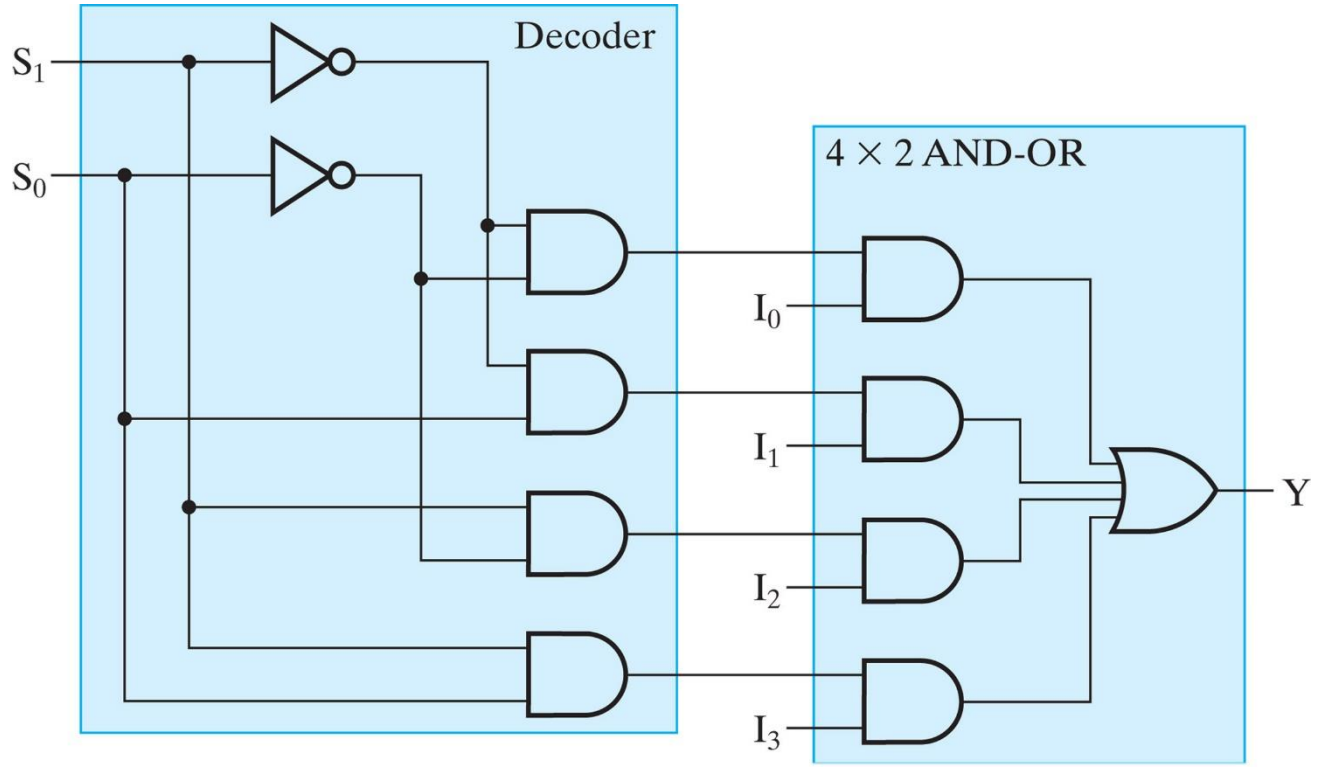
4-to-1 Multiplexer: Implementation 2)

$$Y = \overline{S_1}\,\overline{S_0}\ I_0 + \overline{S_1}S_0\ I_1 + S_1\overline{S_0}\ I_2 + S_1S_0I_3$$

Associative property:

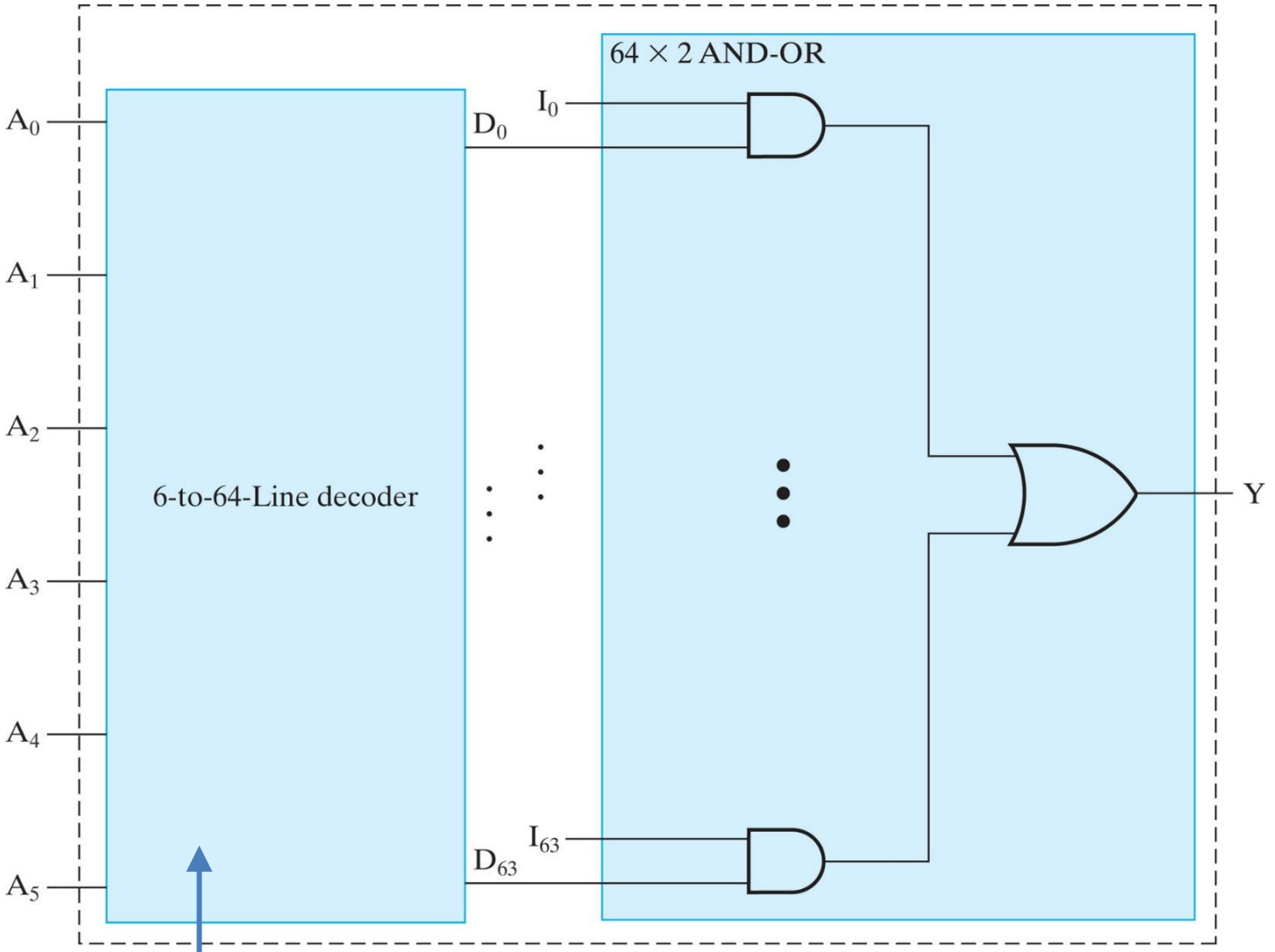
$$Y = (\overline{S_1}\,\overline{S_0})\ I_0 + (\overline{S_1}S_0)\ I_1 + (S_1\overline{S_0})\ I_2 + (S_1S_0)\ I_3$$





2) Although more expensive in terms of gate input cost, this scheme can be **used in hierarchical structures** to implement bigger MUX (2^n data inputs, n selection inputs)

64-to-1 Multiplexer - Implementation 2)



To select one of 64 =2^6 lines,
6 selection inputs are needed

A_0-A_5: selection inputs

I_0-I_{63}: data inputs

Y: output

6-to-64 decoder,

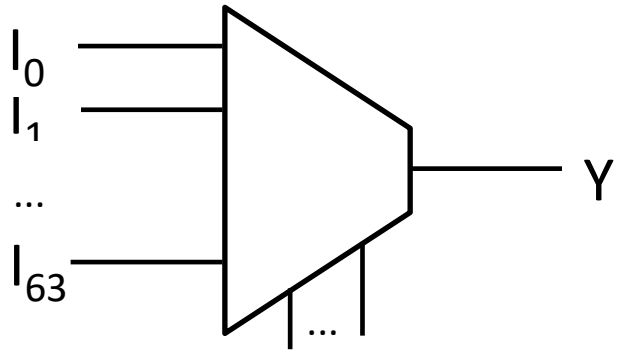
64 2-input AND gates,

1 64-input OR gate:

Gate input cost = **374**

Decoder with hierarchical structure, see slide # 31 (gate input cost = 182)

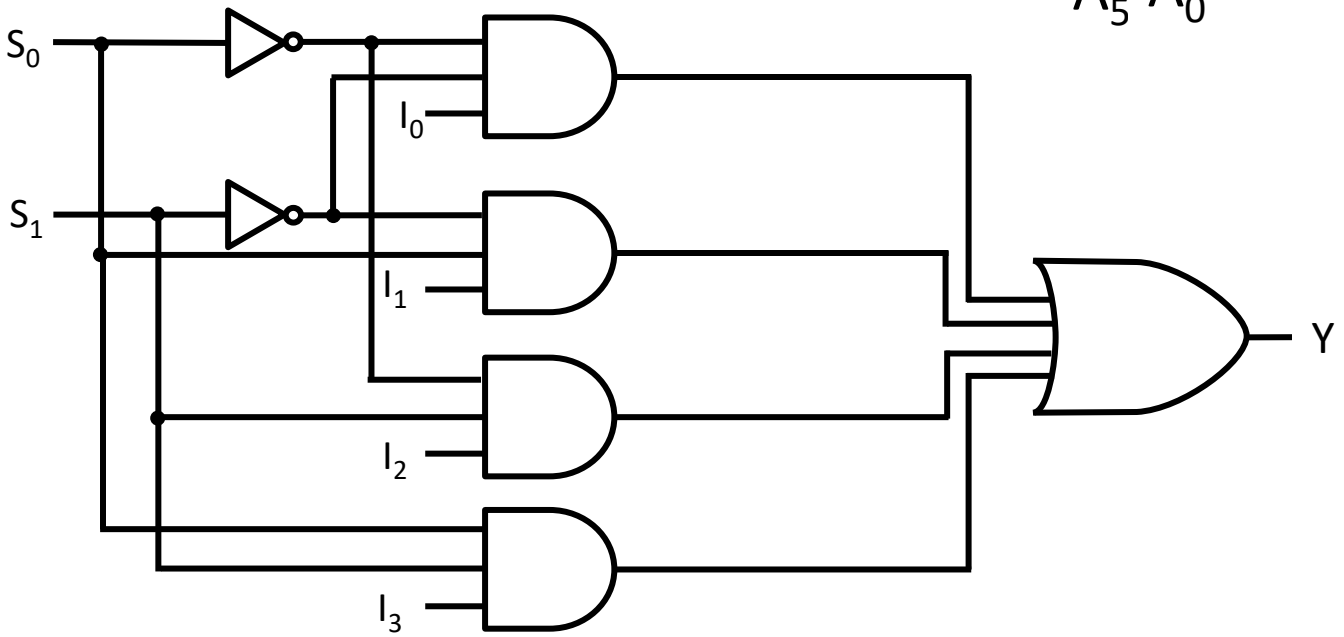
64-to-1 Multiplexer - Implementation 1)



A_0-A_5: selection inputs

I_0-I_{63}: data inputs

Y: output



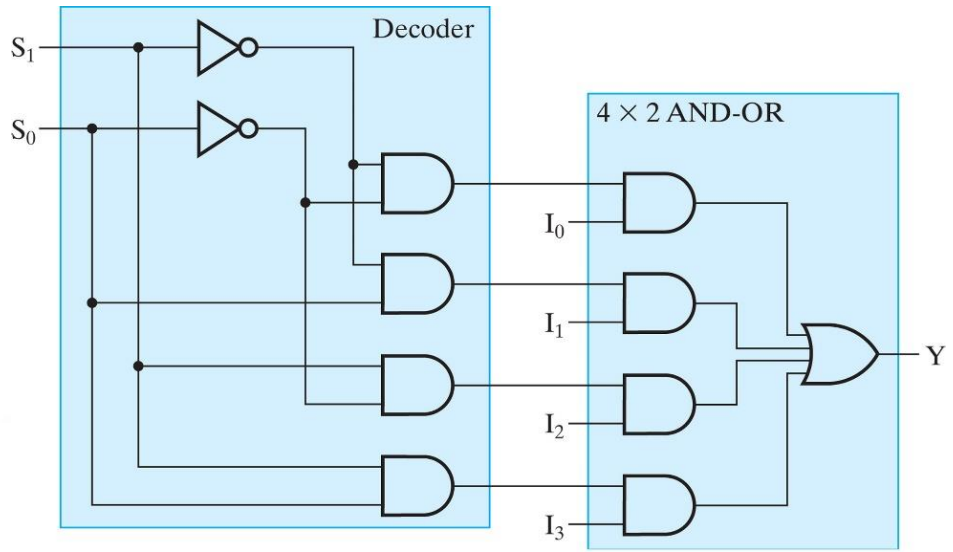
4-to-1 MUX

If the MUX is realized without hierarchical architecture (see slide # 56): 6 inverters, 64 7-input AND gates, 1 64-input OR gate, gate input cost would be **518**

4-to-1 Multiplexer: Structural VHDL (1/2)

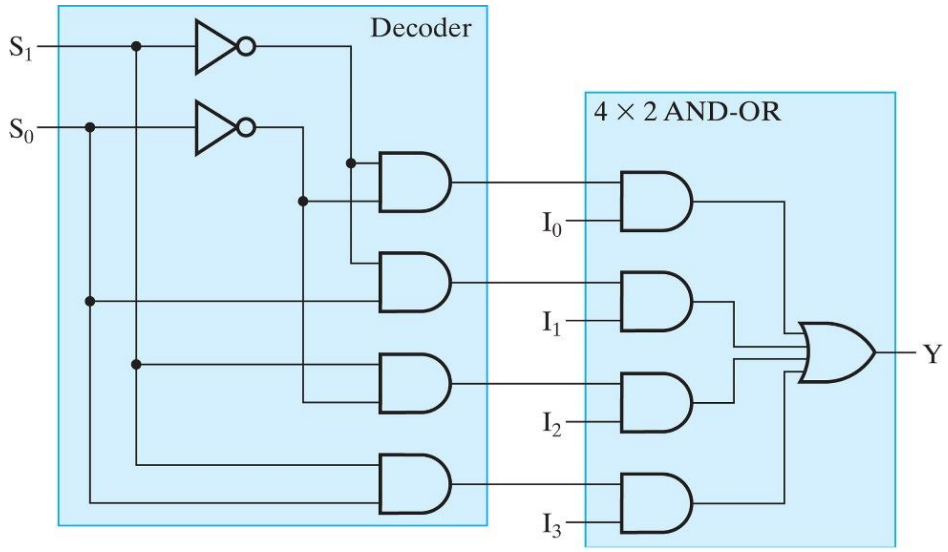
```
-- 4-to-1-Line Multiplexer: Structural VHDL Description
-- (See Figure 3-25 for logic diagram)
library ieee, lcdf_vhdl;
use ieee.std_logic_1164.all, lcdf_vhdl.func_prims.all;
entity multiplexer_4_to_1_st is
   port (S: in std_logic_vector(0 to 1);
         I: in std_logic_vector(0 to 3);
         Y: out std_logic);
end multiplexer_4_to_1_st;

architecture structural_2 of multiplexer_4_to_1_st is
   component NOT1
      port(in1: in std_logic;
           out1: out std_logic);
   end component;
   component AND2
      port(in1, in2: in std_logic;
           out1: out std_logic);
   end component;
```



4-to-1 Multiplexer: Structural VHDL (2/2)

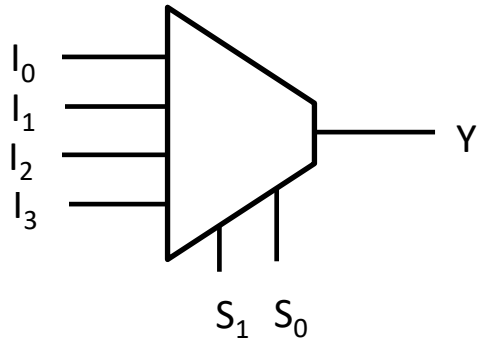
```
component OR4
   port(in1, in2, in3, in4: in std_logic;
        out1: out std_logic);
end component;
signal S_n: std_logic_vector(0 to 1);
signal D, N: std_logic_vector(0 to 3);
begin
   g0: NOT1 port map (S(0), S_n(0));
   g1: NOT1 port map (S(1), S_n(1));
   g2: AND2 port map (S_n(1), S_n(0), D(0));
   g3: AND2 port map (S_n(1), S(0), D(1));
   g4: AND2 port map (S(1), S_n(0), D(2));
   g5: AND2 port map (S(1), S(0), D(3));
   g6: AND2 port map (D(0), I(0), N(0));
   g7: AND2 port map (D(1), I(1), N(1));
   g8: AND2 port map (D(2), I(2), N(2));
   g9: AND2 port map (D(3), I(3), N(3));
   g10: OR4 port map (N(0), N(1), N(2), N(3), Y);
end structural_2;
```



4-to-1 Multiplexer: Dataflow VHDL with Priority Logic "when-else"

```
-- 4-to-1-Line Mux: Conditional Dataflow VHDL Description
-- Using When-Else (See Table 3-8 for function table)
library ieee;
use ieee.std_logic_1164.all;
entity multiplexer_4_to_1_we is
   port (S : in std_logic_vector(1 downto 0);
         I : in std_logic_vector(3 downto 0);
         Y : out std_logic);
end multiplexer_4_to_1_we;

architecture function_table of multiplexer_4_to_1_we is
begin
   Y <= I(0) when S = "00" else
        I(1) when S = "01" else
        I(2) when S = "10" else
        I(3) when S = "11" else
        'X';
end function_table;
```



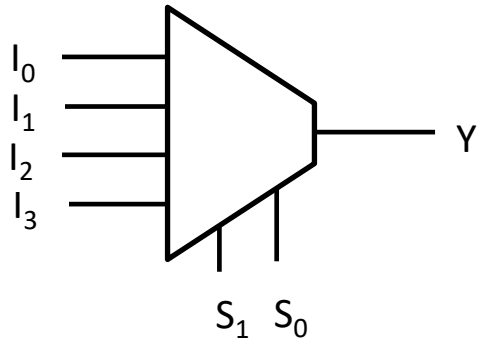
S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

`else 'X'` is needed: in addition to '0' and '1', std_logic_value can take other values ('Z', '-',...) and we must specify what happens also in those conditions (combinational logic)

4-to-1 Multiplexer: Dataflow VHDL with Parallel Logic “with-select"

```
--4-to-1-Line Mux: Conditional Dataflow VHDL Description
Using with Select (See Table 3-8 for function table)
library ieee;
use ieee.std_logic_1164.all;
entity  multiplexer_4_to_1_ws is
   port (S : in std_logic_vector(1 downto 0);
         I : in std_logic_vector(3 downto 0);
         Y : out std_logic);
end multiplexer_4_to_1_ws;

architecture function_table_ws of multiplexer_4_to_1_ws is
begin
   with  S select
      Y <= I(0) when "00",
           I(1) when "01",
           I(2) when "10",
           I(3) when "11",
           'X'when others;
end function_table_ws;
```



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

'X' when others is needed: with-select statement requires the enumeration of all possible cases

Combinational Circuits Using Multiplexer

- A generic **combinational circuit can be realized through a multiplexer** (similar to what we have seen for the decoder)

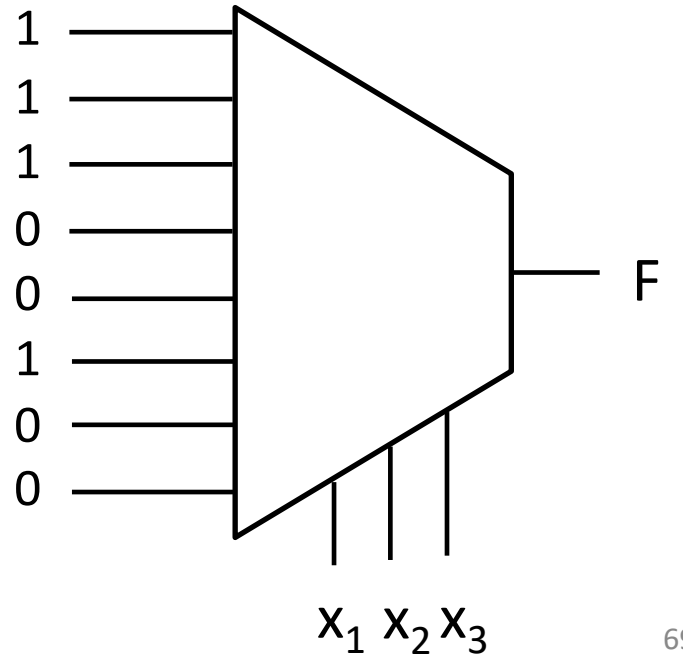
- Possible realization: any **Boolean function of n variables can be implemented with a 2^n-to-1 multiplexer** (2^n data inputs, n selection inputs), fixing the I_i input values appropriately to '0' or '1'

Example: n-variable Function with 2^n-to-1 MUX

x_1	x_2	x_3	F (x_1 , x_2 , x_3)
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

n = 3 → we use an 8-to-1 MUX, applying the 3 inputs of the function to the MUX selection inputs and fixing the MUX data inputs at '1' or '0' based on the truth table

x_1, x_2, x_3 select the appropriate row of the truth table

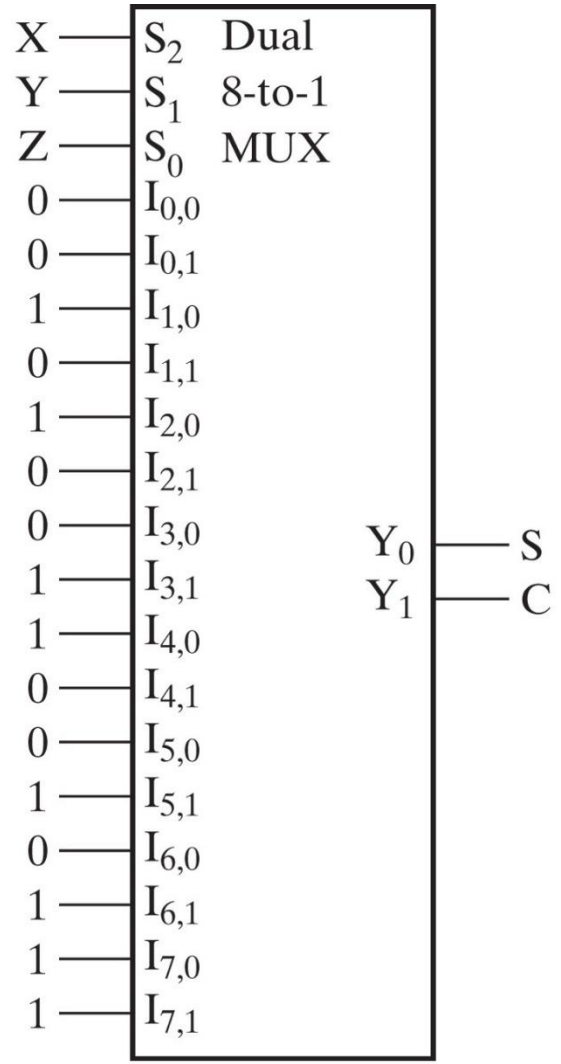


Example 3-15: 1-bit Binary Adder with Dual 8-to-1 MUX

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

Implementation with dual MUX 8-to-1: the XYZ selection signal selects the appropriate row of the truth table



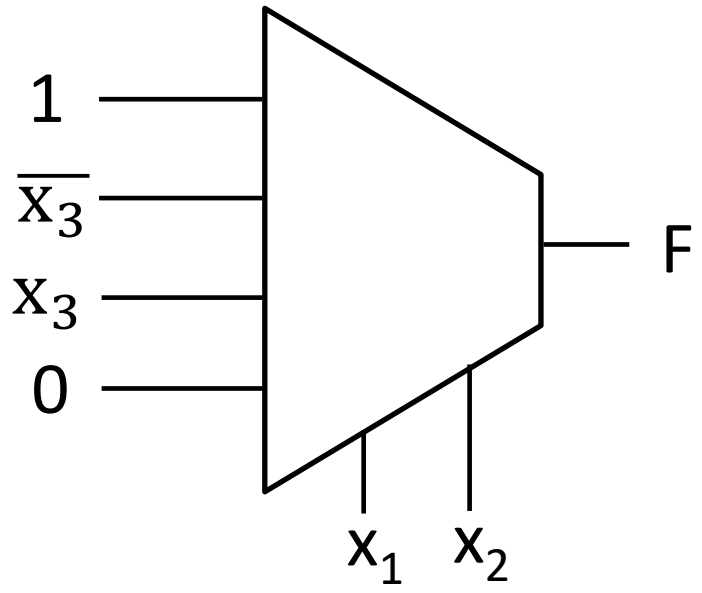
Combinational Circuits Using MUX

- A more efficient implementation can be used to realize a combinational circuit with a multiplexer
- The number of inputs of the multiplexer can be optimized, applying only part of the Boolean function inputs to the selection inputs
  - In the following slides we will see alternative implementations of the functions seen in slides 66-67, using smaller (i.e., cheaper) multiplexers

Example: n-variable Functions with 2^{n-1}-to-1 MUX

x_1	x_2	x_3	F (x_1 , x_2 , x_3)
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

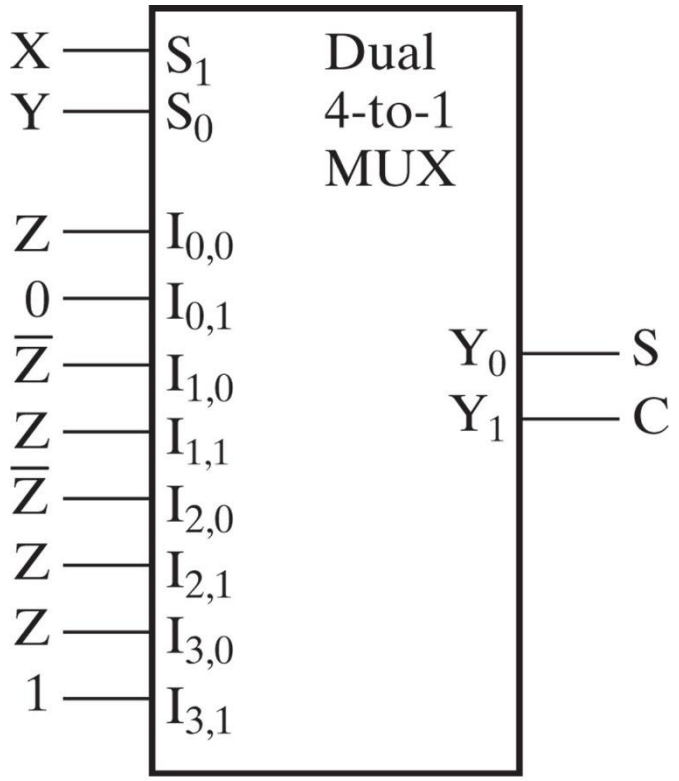
n = 3: We now use a 4-to-1 MUX, applying only 2 of the inputs of the function (x_1, x_2) to the MUX selection inputs.
In this example, the MUX data inputs will be '0', '1', x_3, $\overline{x_3}$



Example 3-16: 1-bit Adder with Dual 4-to-1 MUX

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



Implementation with a dual 4-to-1 MUX: two of the Adder inputs X, Y are connected to the 2 MUX selection inputs. The MUX data inputs will be '0', '1', Z or $\overline{Z}$, based on the values of the outputs in the Adder truth table

Summary

- Definition of **combinational circuit**: the outputs depend only on the current value of the inputs, there is no memory
- Importance of **hierarchical approach** (regularity of a circuit, reusing blocks)
- Signal enabling circuit (**enabling**) allows a signal to be passed or not to the output
- **Decoder**: generates 2^n minterms, starting from n input variables
- **Encoder**: inverse function of the decoder, it generates the n-bit binary coding starting from a 2^n-bit 1-hot input
- **Multiplexer**: selects one between several input data, based on the value of a select input, and directs the data to a single output line
- A generic combinational function can be realized using a decoder or a multiplexer

Disclaimer

Figures from*Logic and Computer Design Fundamentals*, Fifth Edition, GE Mano |Kime| Martin