Digital Systems
Section 2

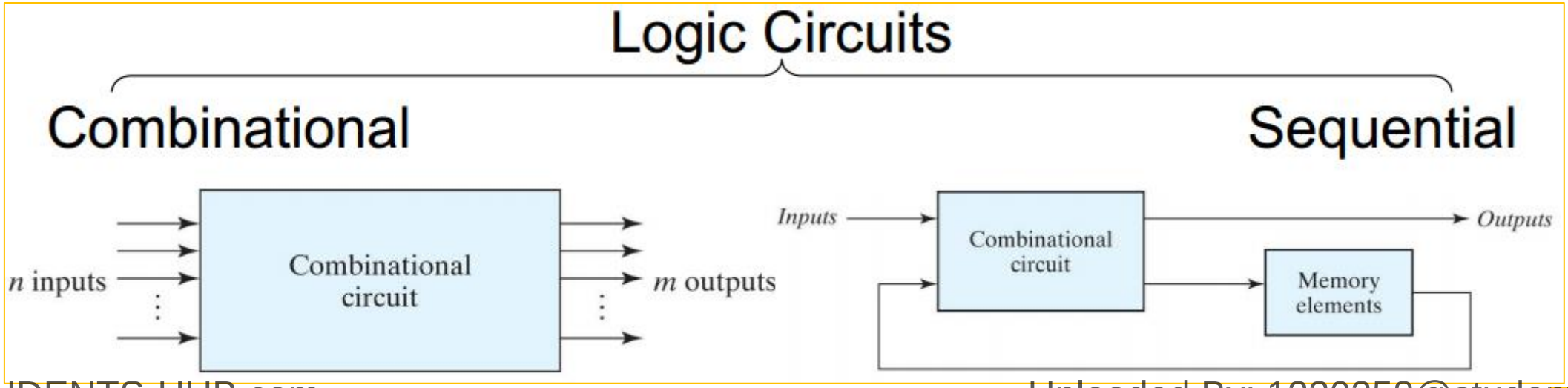
Chapter (4)

Logic Circuits

- In Digital Systems, Logic circuits can be categorized as **combinational** or **sequential**

- **Combinational** Circuit
  - Circuit made of **logic gates only** and perform an operation that can be specified logically by a set of Boolean functions.
  - Circuit output at any time are determined **only by the current combination** (current state/value) of inputs.

- **Sequential** Circuit
  - Circuit is made of **storage/memory** elements and **logic gates**.
  - Circuit output depend on the current combination of inputs and **previously stored values.**

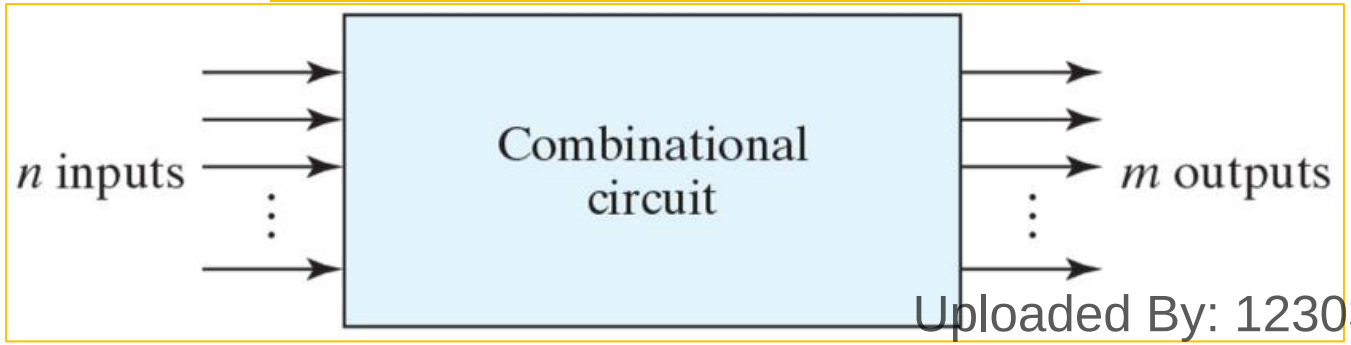


Combinational Circuits

- A logic circuit is combinational if its outputs at any time are a function of **only** the **present** inputs
- A combinational circuit is an interconnection of **logic gates only**
- A combinational circuit does **NOT** have **memory elements or feedback loops**

- A combinational circuit is a **block** of logic gates having:
  1) n inputs: $x_1, x_2 \ldots, x_n$
  2) m outputs: $f_1, f_2 \ldots, f_m$
  3) Logic Gates and wires
- For **n**-input variables, there are $\mathbf{2^n}$ possible **input combinations**.
- There are **m**-outputs, and **m** can be greater than **n**.
- Each output variable can be described with a Boolean function expressed in terms of n or less input variables

Block diagram of a combinational circuit



- For any logic circuit, there are two main activities: **analysis and design**.
- **Analysis**: Examine how the circuit **behaves** by **determining the outputs** based on given inputs and logic gates.
  - Circuit/Logic Gate → Boolean Expression/Truth Table
- **Design**: **Construct** circuits that **deliver the desired outputs** by using logic gates and Boolean expressions.
  - Desired Outputs/Truth Table → Boolean Expression → Circuit/Logic Gates

Analysis determines the **logic function** that a circuit implements

- Given: a logic circuit
- Desired: A **description** of the circuit either in the form of:
  - Boolean functions
  - Truth tables
  - Simply an explanation of the circuit

I. Obtain Boolean expression/function from logic diagram

1) Make sure that the given circuit is **combinational** and **NOT** sequential. The diagram of a combinational circuit has logic gates with **NO feedback** paths **or memory** elements.

2) **Label** all gate outputs that are a function of input variables.
   - Obtain Boolean function for **each gate**.

3) **Label** all gate outputs that are a function of input variables and previously labeled gates.
   - Obtain Boolean function for each of these gates.

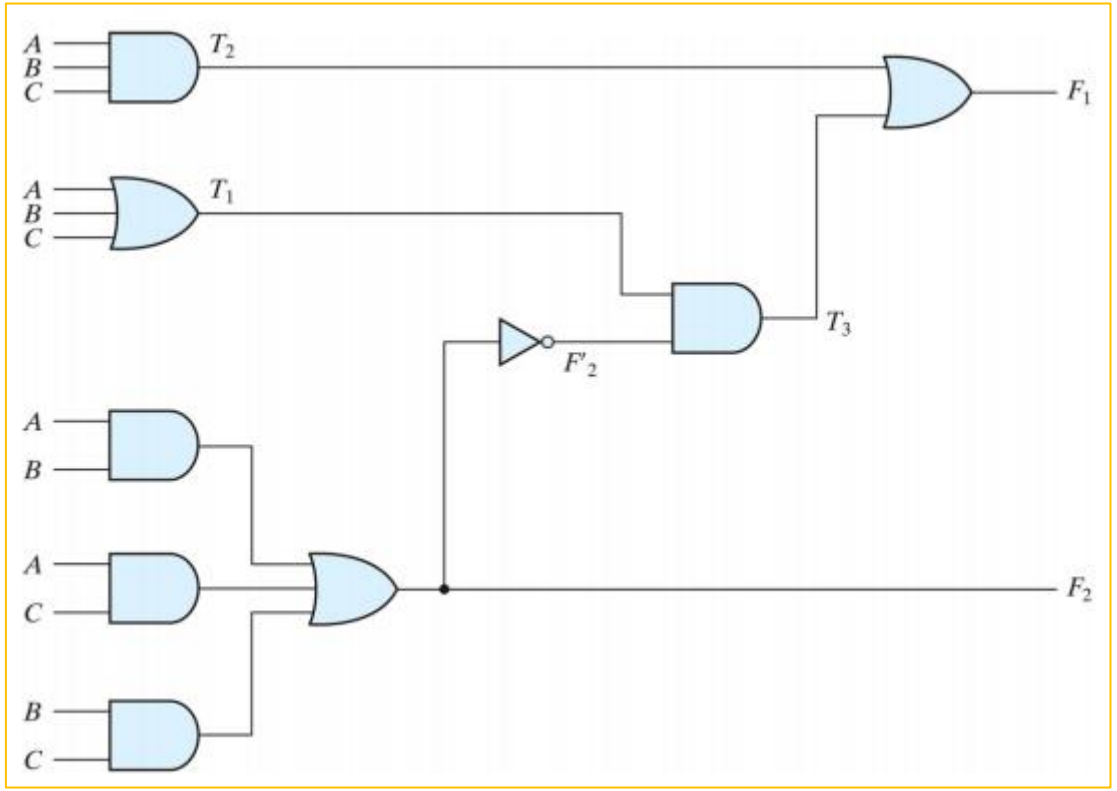
4) **Repeat** step (3) until the outputs of the circuit are obtained.

5) **Substitute** of previously defined variables to obtain the output Boolean functions in terms of **input variables only**
6) **Convert** and/or **simplify** the resultant Output functions to the **required final form** (SOP, POS, SOM,POM) using previously explained methods (Algebraic Manipulation, Expansion, K-Map)

Analysis Procedure

Example:

1) **Combinational** ✓
2) **Label** all gate outputs that are a function of input variables. **(T_1,T_2,F_2)**
3) Label all gate outputs that are a function of input variables and previously labeled gates. **(T_3)**
4) Repeat step (3) until the outputs of the circuit are obtained. **(F1)**
5) **Substitute** → **F_1 (A,B,C)**

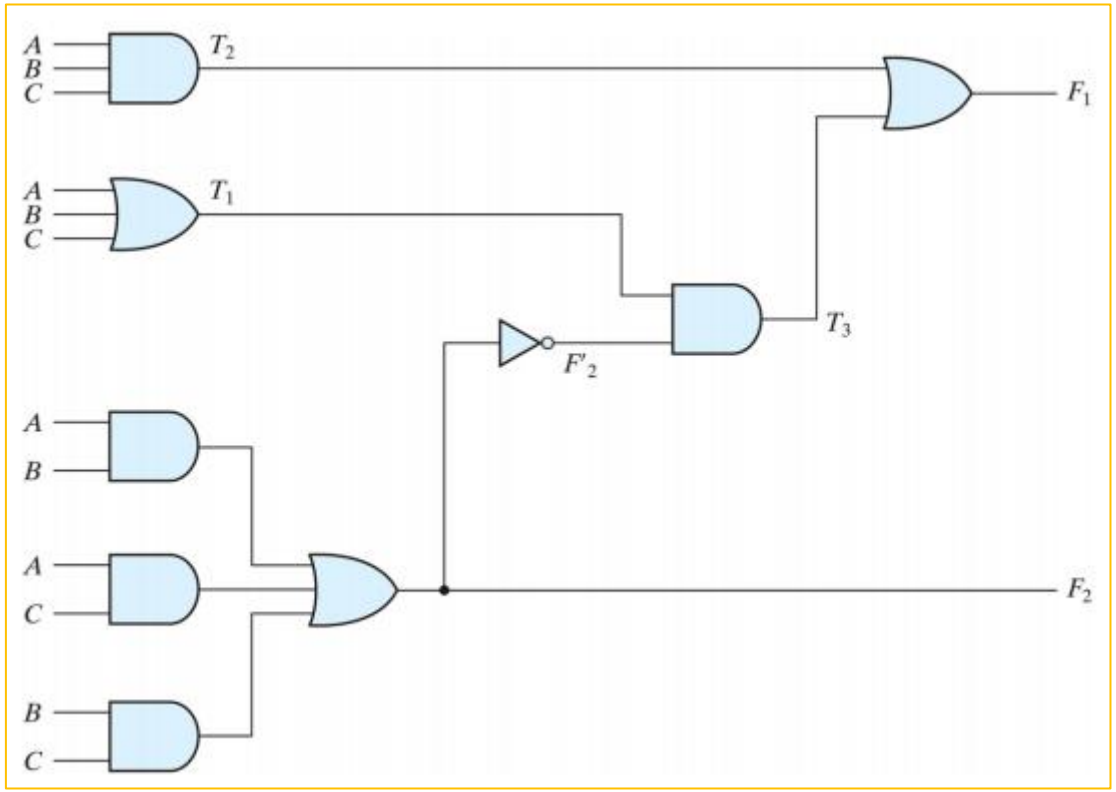


Two outputs of this combinational circuit F_1, F_2
First find **T_1 , T_2 , F_2**
Next find **T_3**
Finally, find **F_1**

Example Cont.:

$$F_2 = AB + AC + BC \checkmark$$

$$T_1 = A + B + C,\ T_2 = ABC$$



$$T_3 = F_2' T_1$$

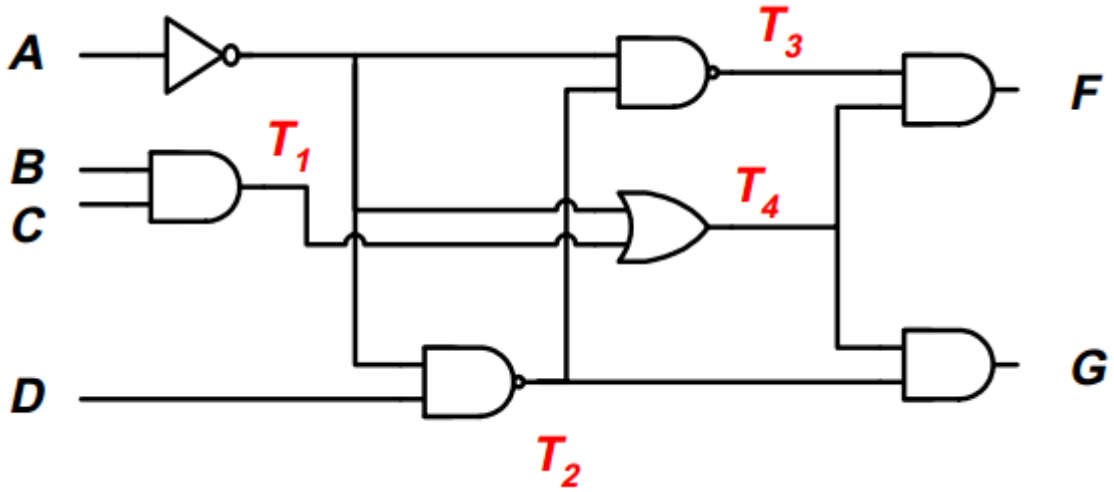
No need to find T1 as a function of input variables only, because it is an **intermediate** gate output

$$\begin{aligned} F_1 &= T_3 + T_2 \\ &= F_2' T_1 + ABC = (AB + AC + BC)'(A + B + C) + ABC \\ &= (A' + B')(A' + C')(B' + C')(A + B + C) + ABC \\ &= A'BC' + A'B'C + AB'C' + ABC \checkmark \end{aligned}$$

Extra Example:

$$T_1 = BC$$

$$T_2 = (A'D)' = A + D'$$



$$T_3 = (A'T_2)' = A + T_2' = A + A'D = A + D$$

$$T_4 = A' + T_1 = A' + BC$$

$$F = T_3 T_4 = (A + D)(A' + BC) = AA' + ABC + A'D + BCD$$
$$= ABC + A'D + BCD = ABC + A'D \quad \checkmark$$

$$G = T_2 T_4 = (A + D')(A' + BC) = AA' + ABC + A'D' + BCD'$$
$$= ABC + A'D' + BCD' = ABC + A'D' \quad \checkmark$$

II. Obtain the Truth Table from the Logic Diagram

1) Prepare the **truth table** for **n** input variables and **2^n** input **combinations**

2) **Label** all gate outputs that are a function of input variables
   - **Fill** in the truth table for these outputs

3) **Label** all gate outputs that are functions of input variables and previously labeled gates
   - **Fill** in the truth table columns for these outputs

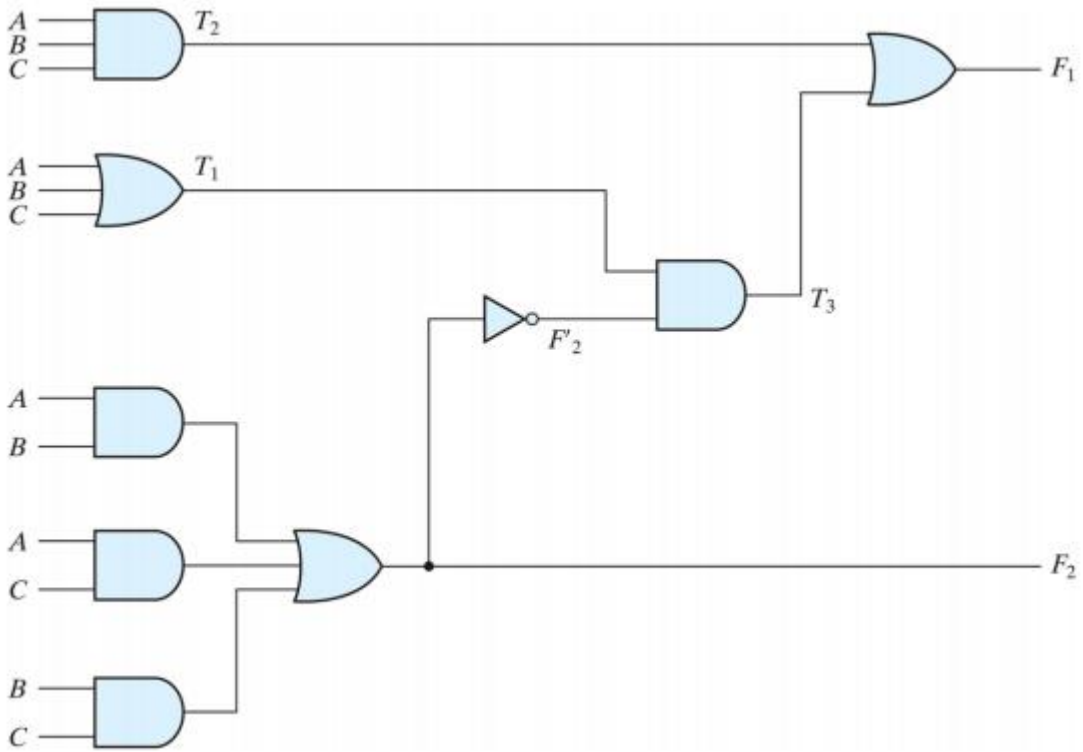
4) **Repeat** step (3) until the columns for **all** the outputs are obtained

5) **Simplify** the obtained Output Function using K-Map (**if required**)

ENCS 2340

Example:

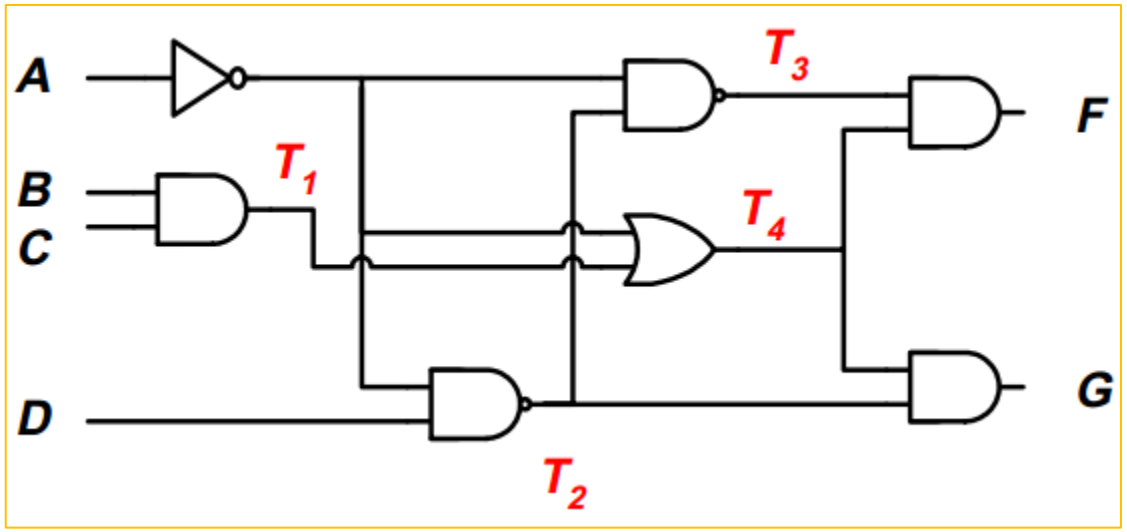
Analyze the below logic circuit by establishing the **truth table** for F_1 and F_2.



A	B	C	F_2	F'_2	T_1	T_2	T_3	F_1
0	0	0	0	1	0	0	0	0
0	0	1	0	1	1	0	1	1
0	1	0	0	1	1	0	1	1
0	1	1	1	0	1	0	0	0
1	0	0	0	1	1	0	1	1
1	0	1	1	0	1	0	0	0
1	1	0	1	0	1	0	0	0
1	1	1	1	0	1	1	0	1

Extra Example:

Analyze the below logic circuit by establishing the **truth table** for **F** and **G**.



Inputs								Outputs	
A	B	C	D	T_1	T_2	T3	T_4	F	G
0	0	0	0	0	1	0	1	0	1
0	0	0	1	0	0	1	1	1	0
0	0	1	0	0	1	0	1	0	1
0	0	1	1	0	0	1	1	1	0
0	1	0	0	0	1	0	1	0	1
0	1	0	1	0	0	1	1	1	0
0	1	1	0	1	1	0	1	0	1
0	1	1	1	1	0	1	1	1	0
1	0	0	0	0	1	1	0	0	0
1	0	0	1	0	1	1	0	0	0
1	0	1	0	0	1	1	0	0	0
1	0	1	1	0	1	1	0	0	0
1	1	0	0	0	1	1	0	0	0
1	1	0	1	0	1	1	0	0	0
1	1	1	0	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1

ENCS 2340

Design involves the specifications of **design objectives** and the creation of the **logic circuit diagram** according to these specifications

1) **Specification**
   - Describe the problem
   - Specify the **number** of inputs and outputs
   - Assign a letter **symbol** to each input/output

2) **Formulation**
   - Convert the specification into **truth tables** for outputs

3) **Logic Minimization**
   - Derive a Boolean function for each output as a function of inputs and minimize these functions using **K-map** or **Boolean algebra**

4) **Technology Mapping**
   - Draw a **logic diagram** using logic gates/functional blocks

5) **Verification**
   - Verify the **correctness** of the design, either **manually** or using **simulation**

Example: Design a circuit that takes **BCD** and convert it to **excess-3**

1) Specification

- **Describe**: Convert BCD code to Excess-3 code.
- **Specify**: Input: **4**-bit BCD code , Output: **4**-bit Excess-3 code
- **Assign**: BCD input: **A, B, C, D** , Excess-3 output: w, x, y, z

2) Formulation

- Done easily with a **truth table**
- **Note**: Output is **don't care** for 1010 to 1111

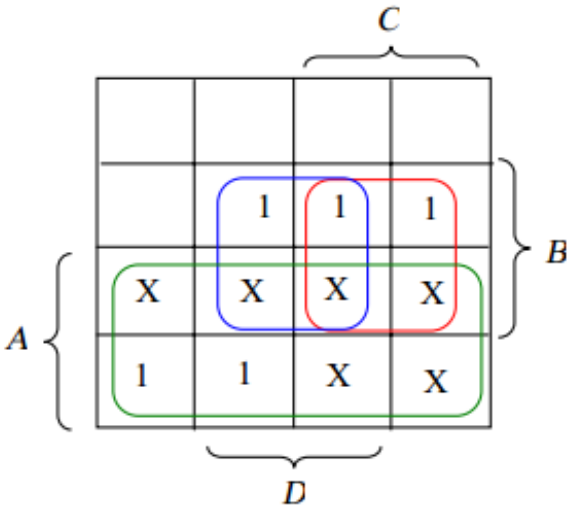
Inputs (BCD Code)				Outputs(Excess-3)			
A	B	C	D	w	x	y	z
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

ENCS 2340

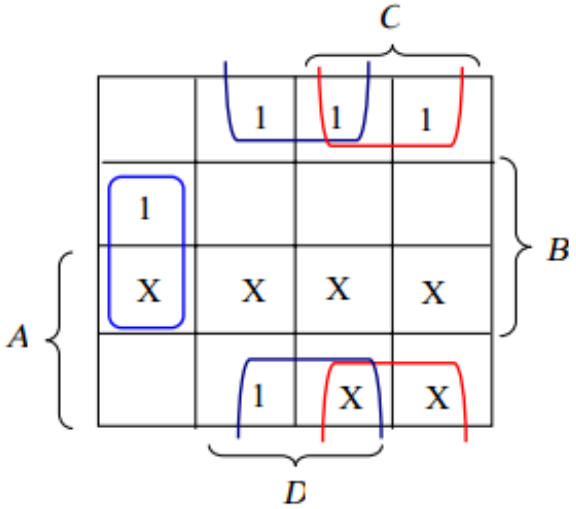
Example Cont.: Design a circuit that takes **BCD** and convert it to **excess-3**

3) Logic Minimization using K-maps

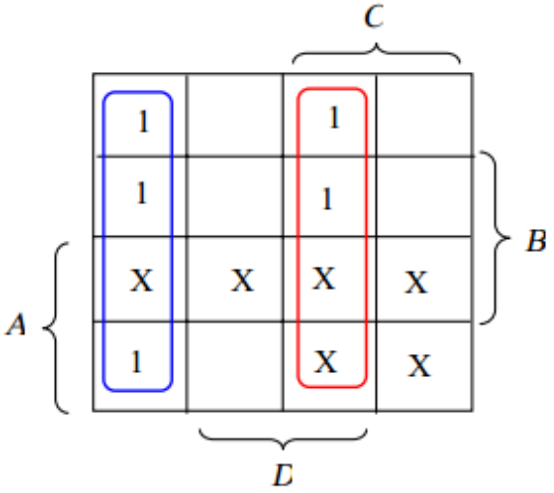
- **4** outputs → **4** **K-Maps**



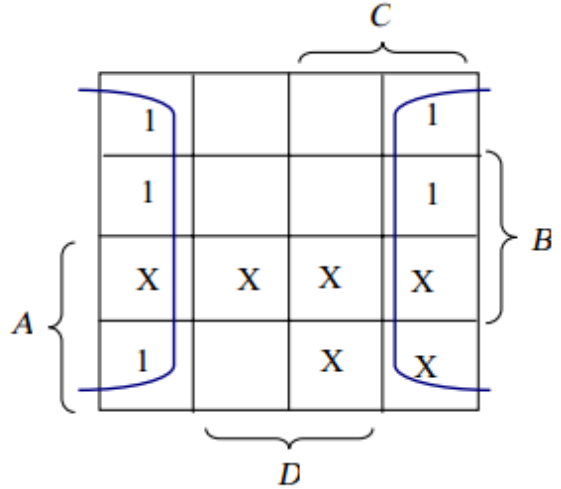
$w = A + BC + BD$



$x = B'C + B'D + BC'D'$

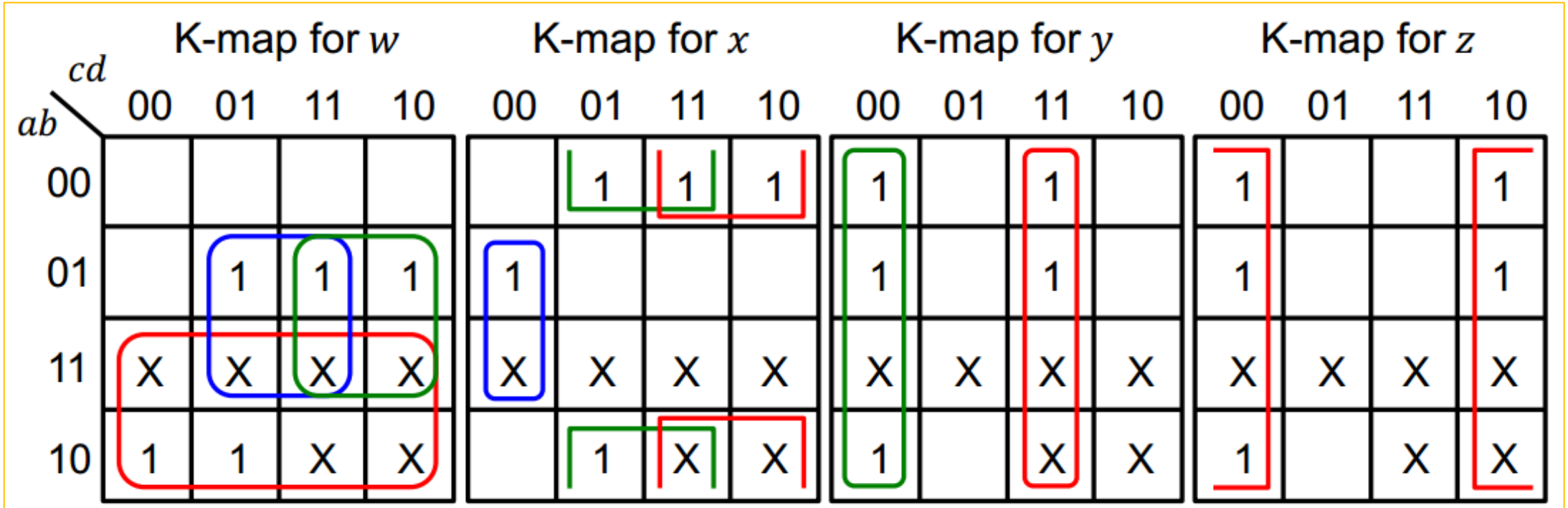


$y = CD + C'D'$



$z = D'$

Example Cont.:



Minimal Sum-of-Product expressions:

$$w = a + bc + bd \;,\; x = b'c + b'd + bc'd' \;,\; y = cd + c'd' \;,\; z = d'$$

Additional 3-Level Optimizations: extract common term $(c + d)$

$$w = a + b(c + d) \;,\; x = b'(c + d) + b(c + d)' \;,\; y = cd + (c + d)'$$

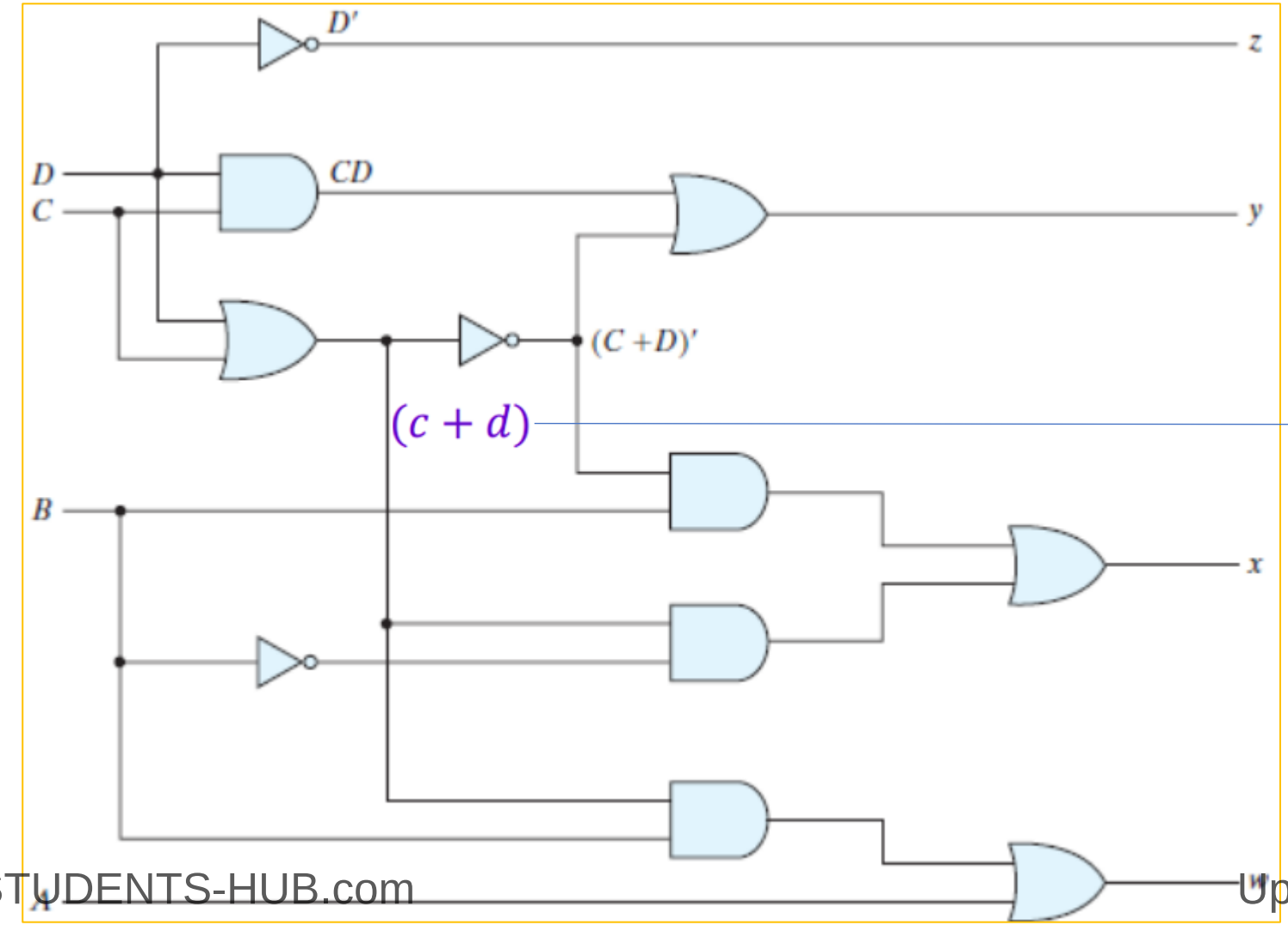
Double DeMorgan's Law (in x , y)

ENCS 2340

Example Cont.: Design a circuit that takes **BCD** and convert it to **excess-3**

4) Technology Mapping (Draw a **logic diagram** using ANDs, ORs, and inverters)

$w = a + b(c + d)$, $x = b'(c + d) + b(c + d)'$, $y = cd + (c + d)'$, $z = d'$



When **multiple** outputs exist, it is common practice to optimize their functions to create **common** gates across them, even if this leads to **nonstandard** forms.

ENCS 2340

Example Cont.: Design a circuit that takes **BCD** and convert it to **excess-3**

5) Verification

- Can be done **manually**
  - **Extract** output functions from circuit diagram
  - Find the **truth table** of the circuit diagram
  - **Match** it against the specification **truth table**
- Verification process can be **automated**
  - Using a **simulator** for complex designs

Standard Logic Circuits

- There are several combinational circuits that are **employed extensively** in the design of digital systems.
- These circuits are available in integrated circuits and are classified as **standard components**. They perform **specific** digital functions commonly needed in the design of digital systems.
- Most **important** standard combinational circuits
  - Adders and Subtractors
  - Comparators
  - Decoders
  - Encoders
  - Multiplexers

Half Adder (HA)

- A combinational circuit that performs the **addition** of two bits is called a **Half Adder**

- **Recall that**
  - 0+0 = 0 and **carry** of 0
  - 0+1= **1** and **carry** of 0
  - 1+0= **1** and **carry** of 0
  - 1+1= 0 and **carry** of **1**

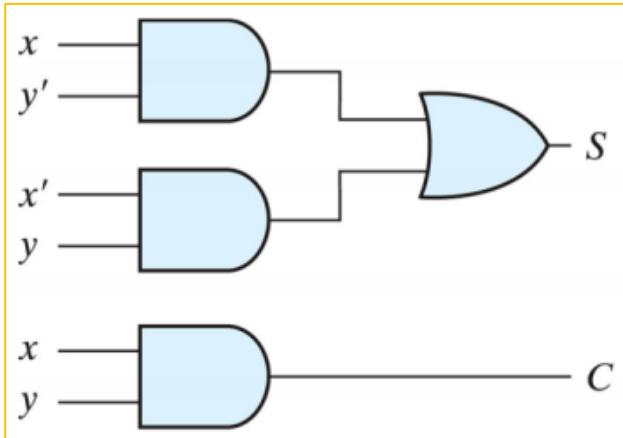
- Inputs: **2** bits (x, y)
- Outputs: **2** bits (**Sum**, **Carry**)

x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Obtain the simplified Boolean functions.

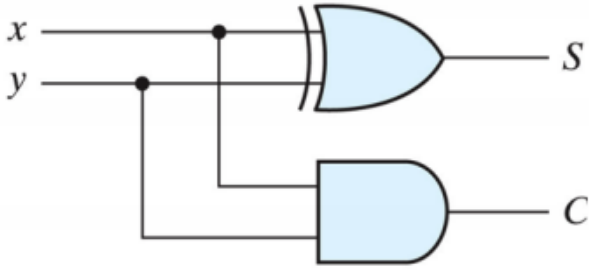
$S = xy' + x'y = x \oplus y$

$C = xy$



$S = xy' + x'y$

$C = xy$



$S = x \oplus y$

$C = xy$

Full Adder (FA)

A **full adder** is a combinational circuit that forms the arithmetic **sum of three bits**

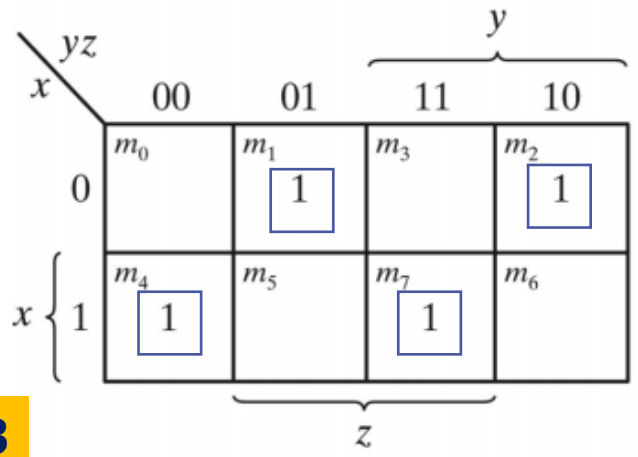
- **Inputs**: 3 bits
  - Two bits → significant **input** bits to be added
  - Third bit → **carry** from the previous stage
- **Outputs**: 2 bits
  - Sum (S)
  - Carry (C)

1

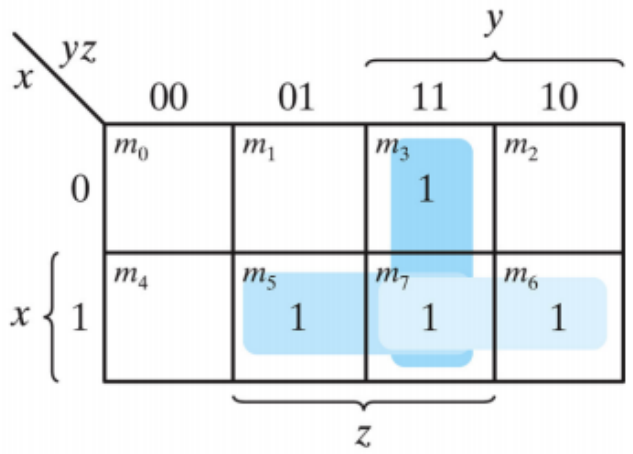
x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

2

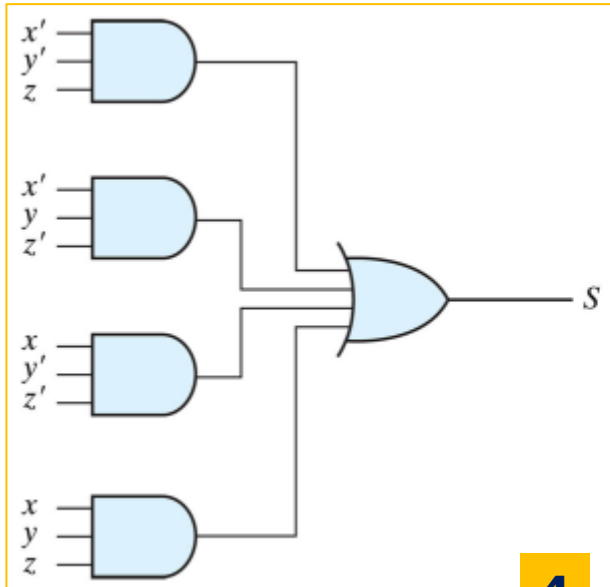
3

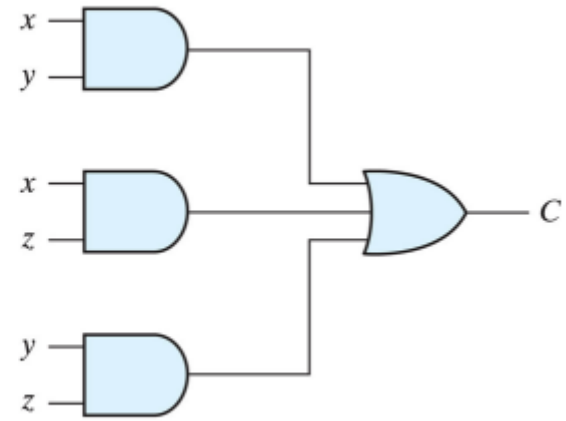


$S = x'y'z + x'yz' + xy'z' + xyz$



$C = xy + xz + yz$





4

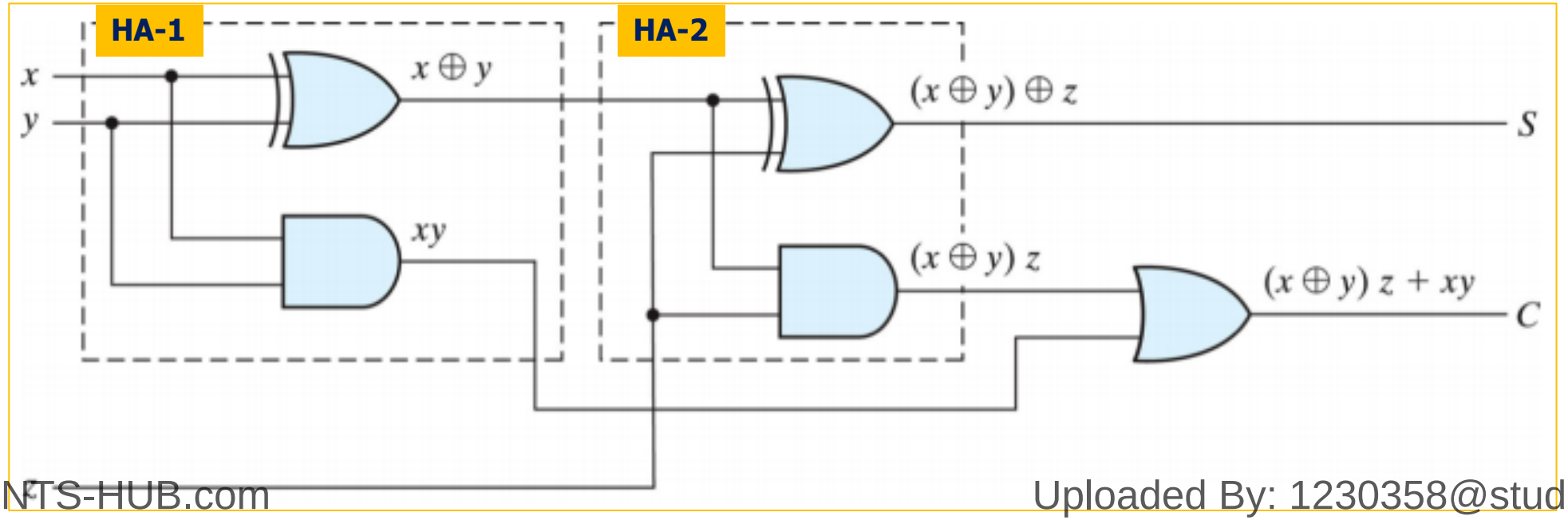
Full Adder (FA)

Manipulate the expressions of **S,C** to get more **familiar** forms

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

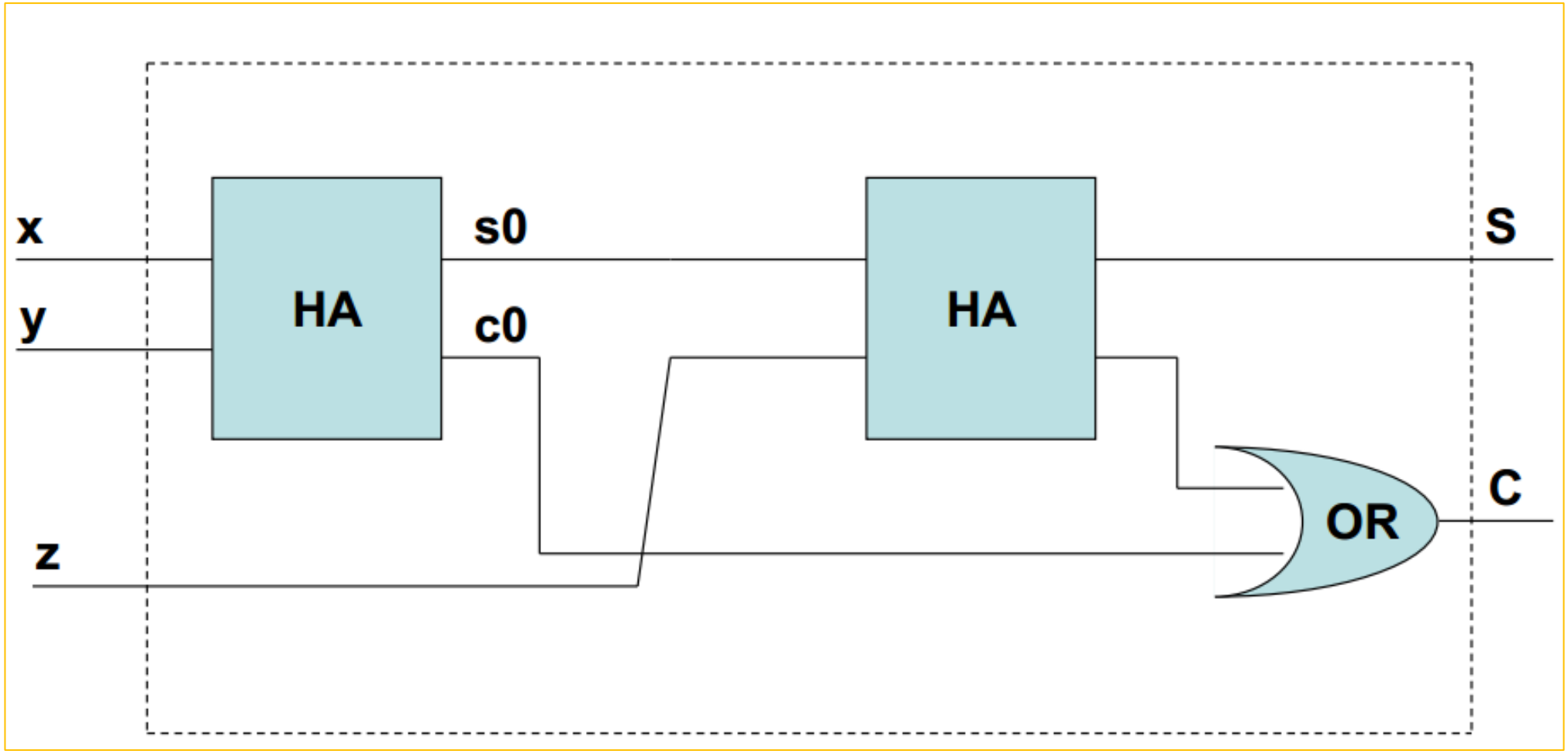
$$S = \sum(1,2,4,7)$$
$$= x'y'z + x'yz' + xy'z' + xyz$$
$$= (x'y' + xy)z + (x; y + xy')z'$$
$$= (x \oplus y)'z + (x \oplus y)z'$$
$$= (x \oplus y) \oplus z$$

$$C = \sum(3,5,6,7)$$
$$= xy + xz + yz$$
$$= x'yz + xy'z + xy$$
$$= (x'y + xy')z + xy$$
$$= (x \oplus y)z + xy$$



Full Adder (FA)

Utilize the Standard **HA** circuit to build **FA**



Practical **Arithmetic/Logical** Functions

- **Bitwise Operations:** Perform operations on binary bit **vectors** (e.g., adding, subtracting, multiplying). Each bit position can utilize the same basic **sub-function**, allowing for consistency across operations.

- **Modular Design Approach:** To simplify the complexity of handling large inputs and outputs, design a **reusable sub-function block** (cell) for each bit. This block can be **replicated** (iterative array) to create larger functional blocks for overall operations, facilitating more manageable and efficient circuit design across various arithmetic and logical functions.

Binary Addition (Recap)

- The process of addition proceeds on a **bit-by-bit** basis, right to left, beginning with the **least significant bit (LSB)**
- Include the **carry** in the addition

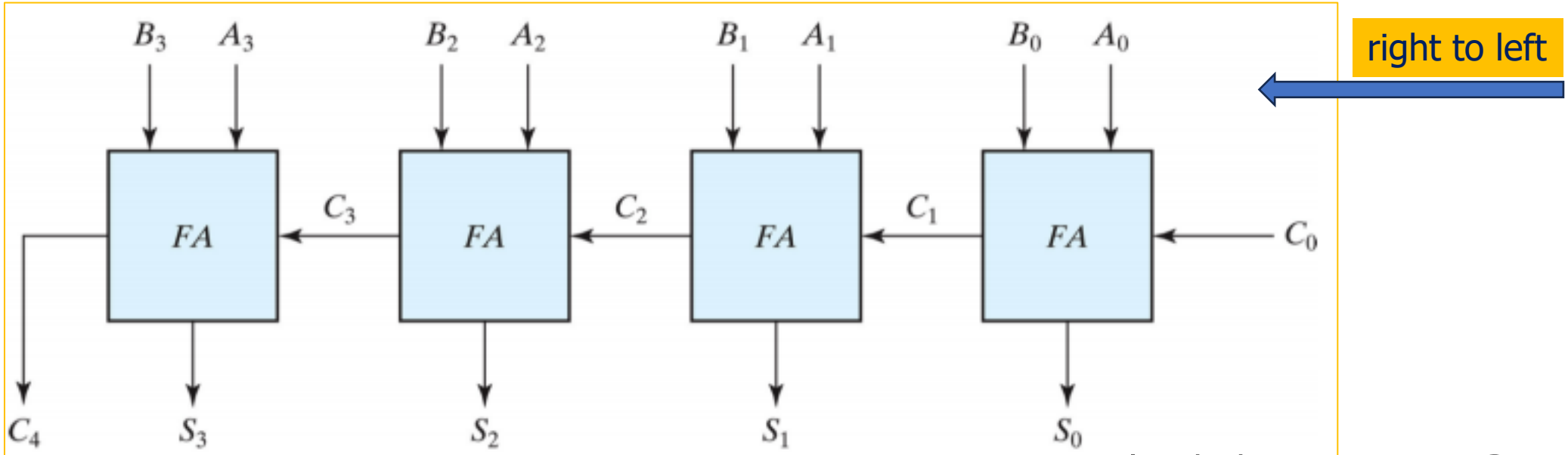
carry		1	1	1	1				
	0	0	1	1	0	1	1	0	(54)
+	0	0	0	1	1	1	0	1	(29)
	0	1	0	1	0	0	1	1	(83)
bit position:	7	6	5	4	3	2	1	0	

Parallel Binary Adder

- A **Parallel Binary Adder** is a digital circuit that produces the arithmetic **sum** of two binary numbers

- We can construct it by **cascading FAs**. We need **n-FA** for an **n**-bit number
  - Each **FA** adds **3** bits: A_i , B_i , C_i → producing: S_i and C_{i+1}
    - (C_i : Carry in, C_{i+1} : Carry out)

For 4-bit numbers $A + B \equiv A_3A_2A_1A_0 + B_3B_2B_1B_0$



Parallel Binary Adder

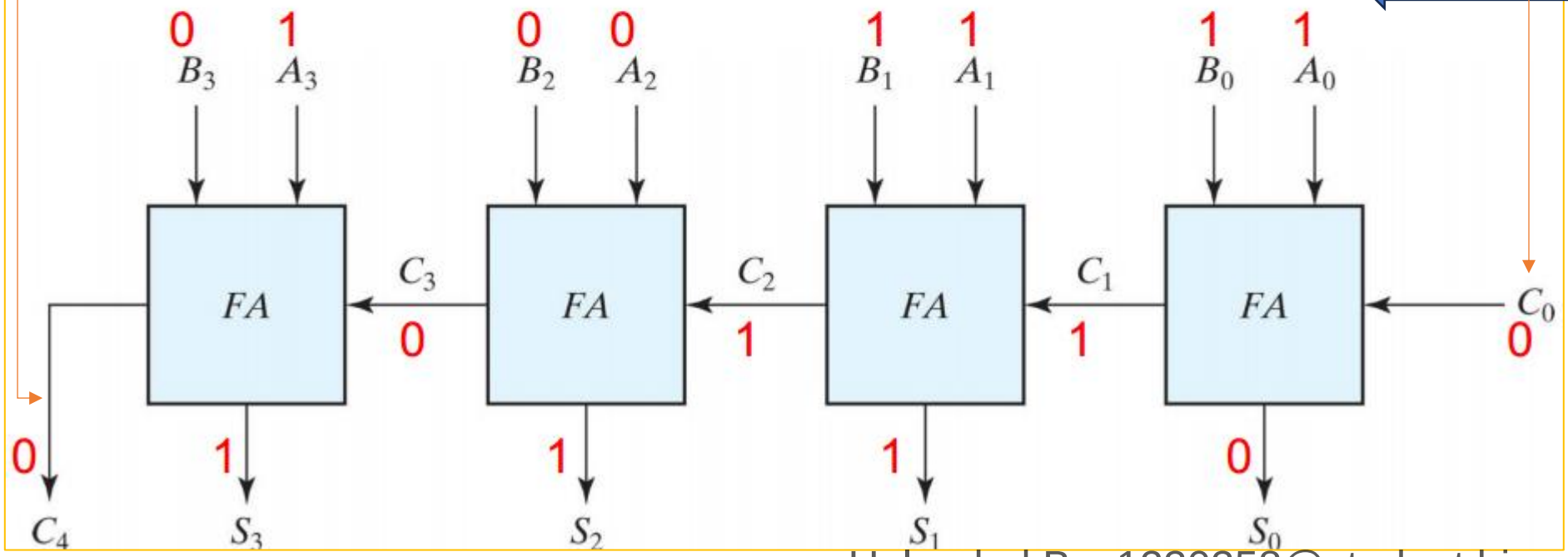
Example:

A = 1011

B = 0011

Subscript *i*:	3	2	1	0	
Input carry	0	1	1	0	C_i
Augend	1	0	1	1	A_i
Addend	0	0	1	1	B_i
Sum	1	1	1	0	S_i
Output carry	0	0	1	1	C_{i+1}

right to left

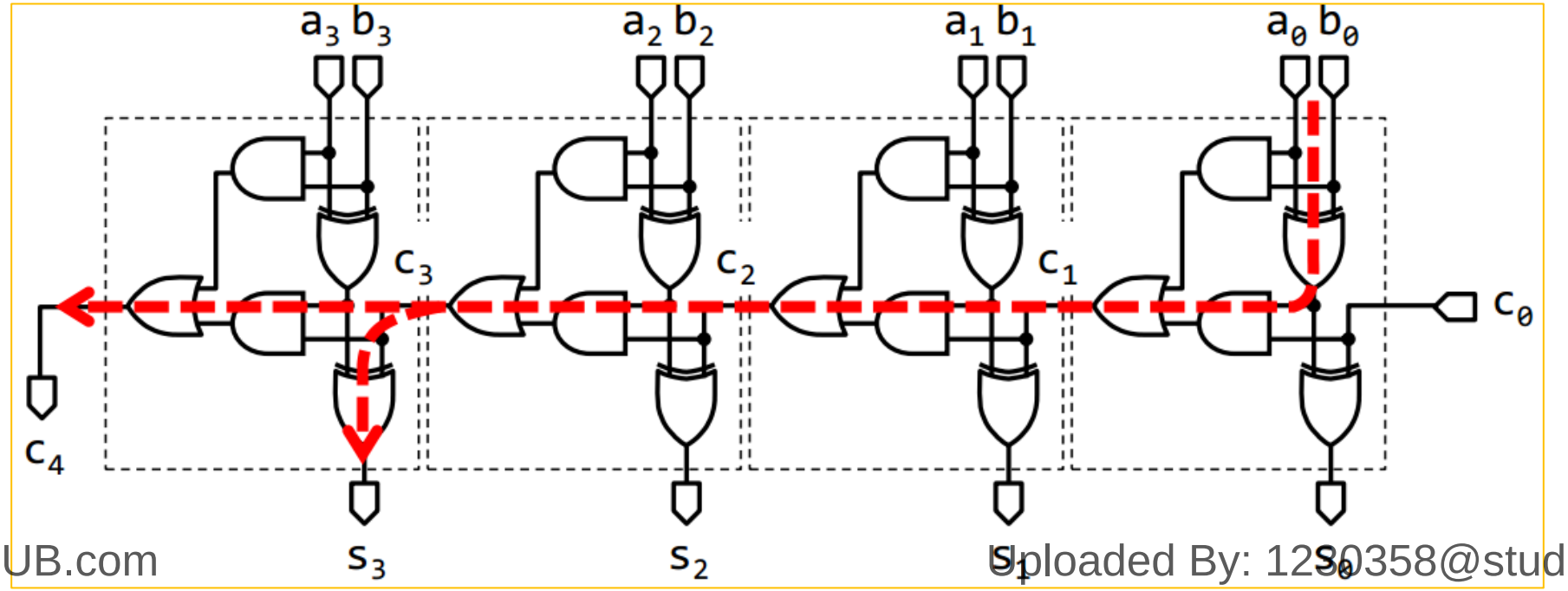


Parallel Binary Adder

- The **four-bit** Parallel Adder is a typical example of a **standard component**. It can be used in many applications involving arithmetic operations
- Observe that the design of this circuit by the **classical** method would require a truth table with $\mathbf{2^9} = 512$ entries (n=9 : $A_0A_1A_2A_3\ B_0B_1B_2B_3\ C_0$)
- It becomes possible to obtain a simple and straightforward implementation by using the previously mentioned **Modular Design Approach** to construct iteratively the **4-bit Parallel Binary Adder** using **sub-function blocks/cells** of the **Full Adder (FA)**
- This Parallel Binary Adder is commonly known as **Ripple-Carry Binary Adder**

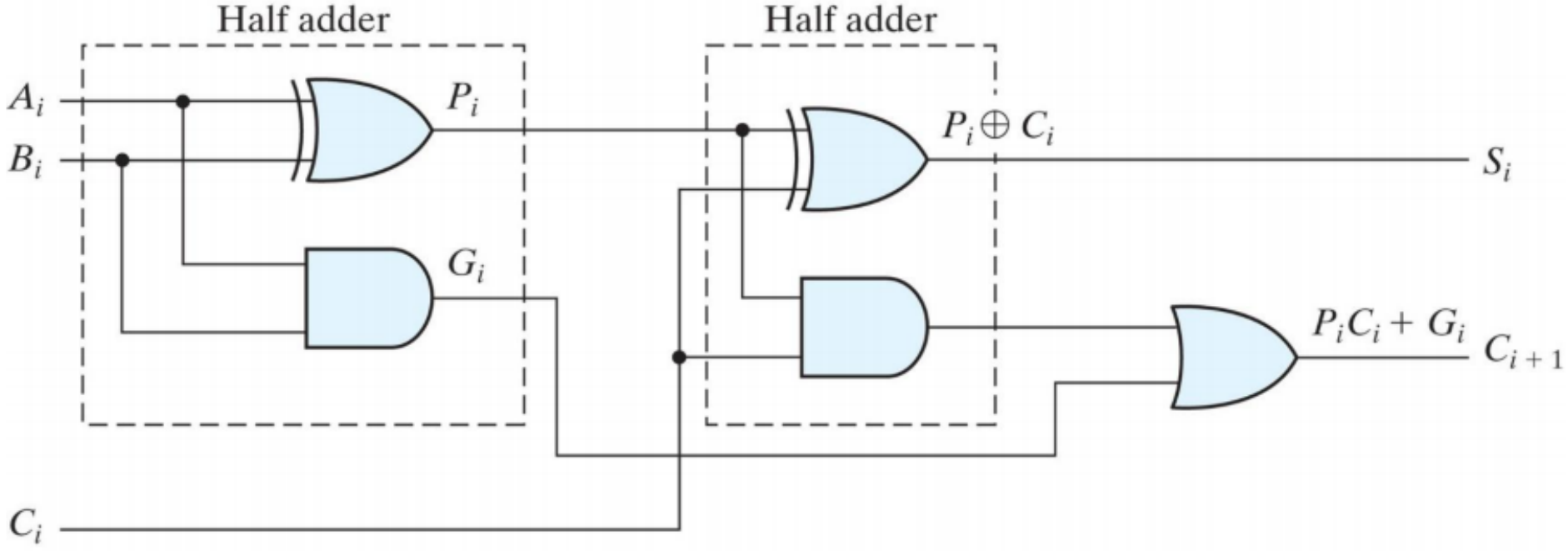
Binary Adder & Carry Propagation

- The **sum bits** are readily **available**
- We need to **wait** for the **last carry bit** (C_4) to be calculated (Propagated)
- Each gate needs **some time** to produce output
- Two gates (one AND & one OR) are used to generate each carry bit
  - For a four-bits Adder, C_4 is generated using (2×4) gates
- This **Carry waiting time** is called **Carry propagation** and it limits the **speed** of overall computations



Carry Propagation

- The most widely used method to reducing the carry propagation in a parallel binary adder is called the **Carry Lookahead Logic**.



$P_i = A_i \oplus B_i$ Carry Propagate

$G_i = A_i . B_i$ Carry Generate (C_{i+1} =1 if G_i=1)

$S_i = P_i \oplus C_i$ Output Sum

$C_{i+1} = G_i + P_i C_i$ Output Carry

Carry Lookahead Logic

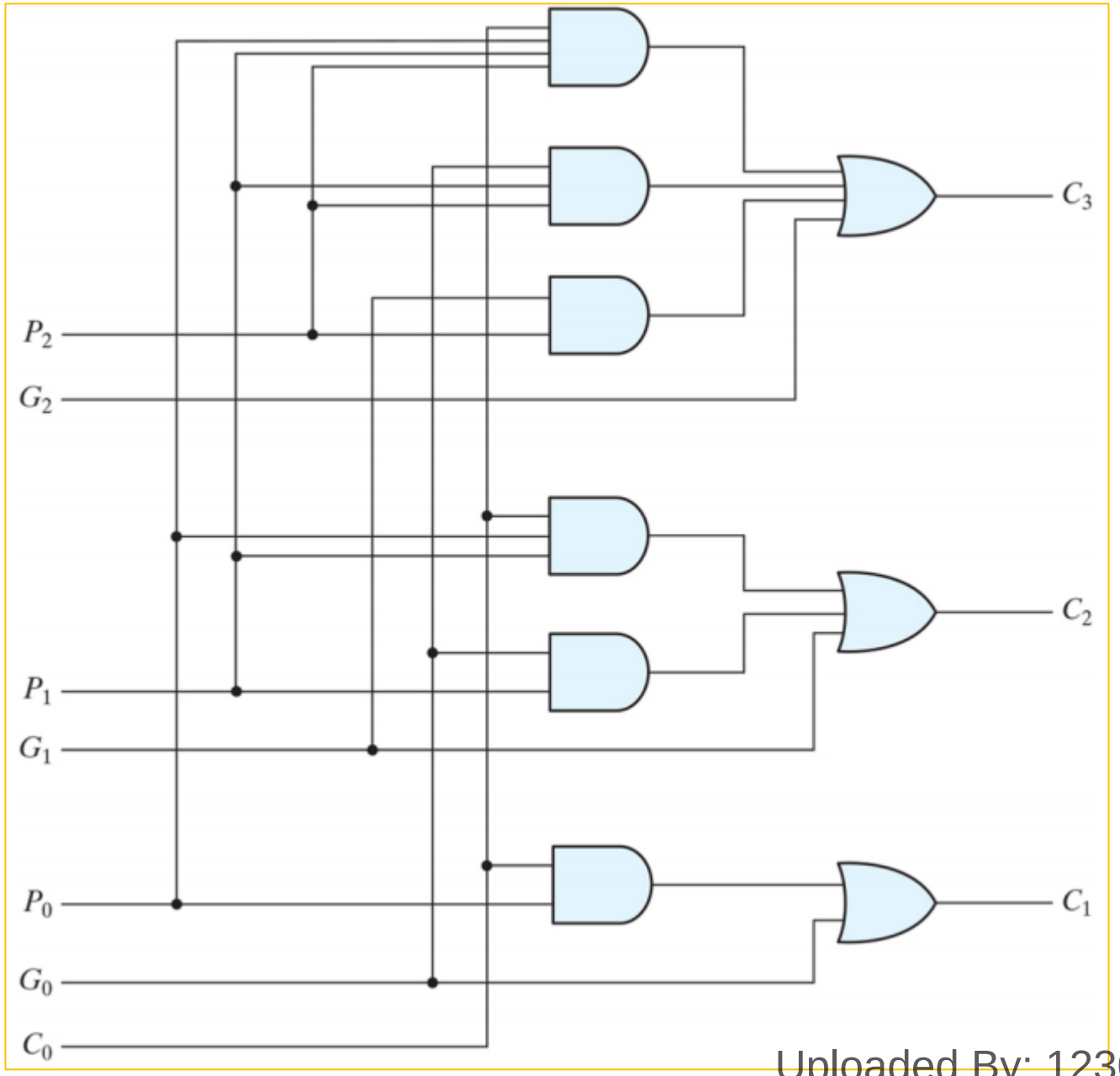
- Write the Boolean functions for the **Carry outputs** of each stage

$$C_{i+1} = G_i + P_i C_i$$

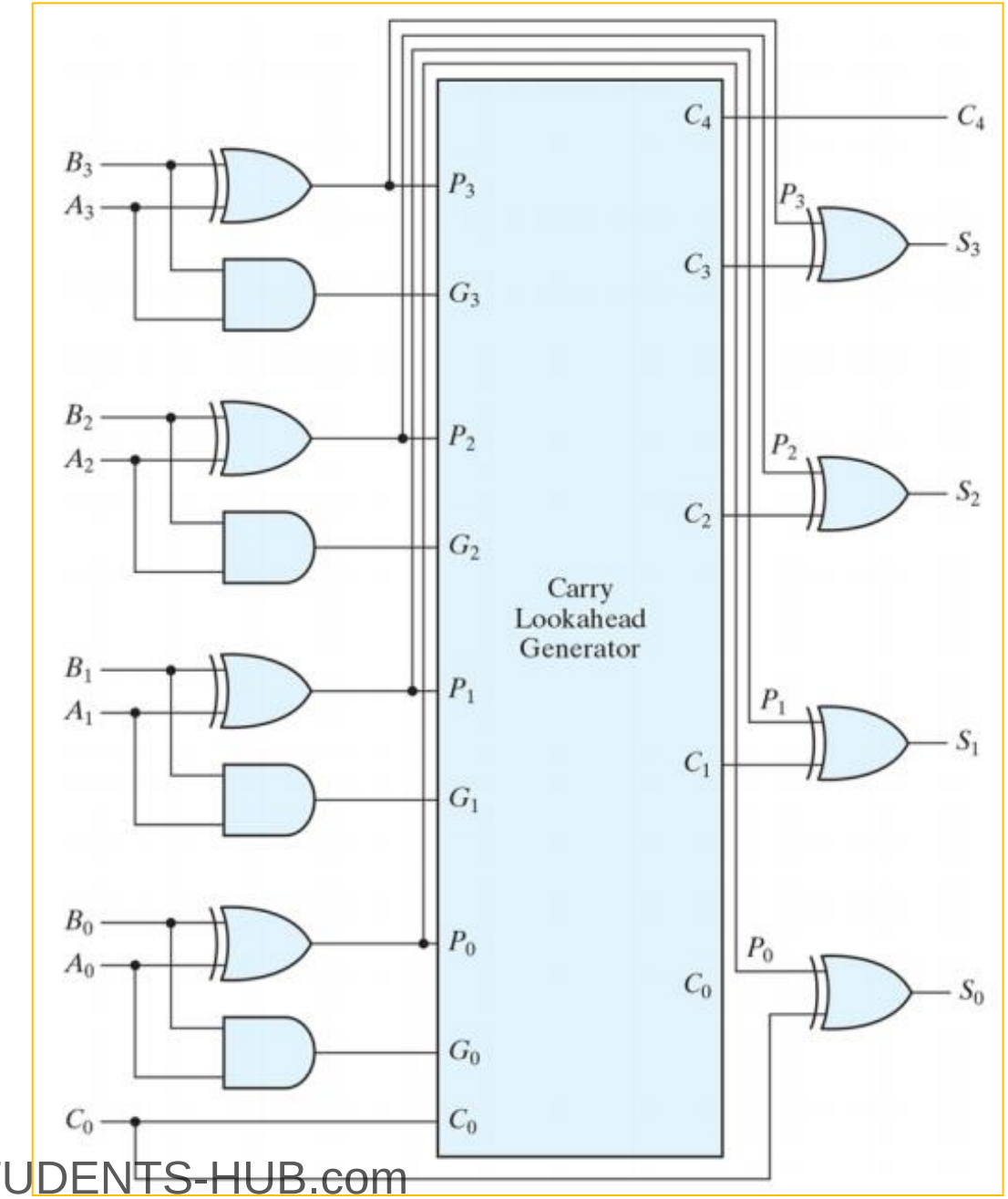
$$\begin{aligned}
C_0 &= \text{input carry} \\
C_1 &= G_0 + P_0 C_0 \\
C_2 &= G_1 + P_1 C_1 \\
&= G_1 + P_1 (G_0 + P_0 C_0) \\
&= G_1 + P_1 G_0 + P_1 P_0 C_0 \\
C_3 &= G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0
\end{aligned}$$

- Note that **P's and G's** are functions of only A's and B's (inputs)
- **ALL** carries are **dependent** on the inputs **only** (C_3 does not have to wait for C_2 and C_1 to become available; C_3 is propagated at the **SAME TIME** as C_2 and C_1).
- This **SPEED GAIN** is traded off with increase in **COMPLEXITY** (**No. of Gates**)

Carry Lookahead Generator



4-bit Carry Lookahead Adder

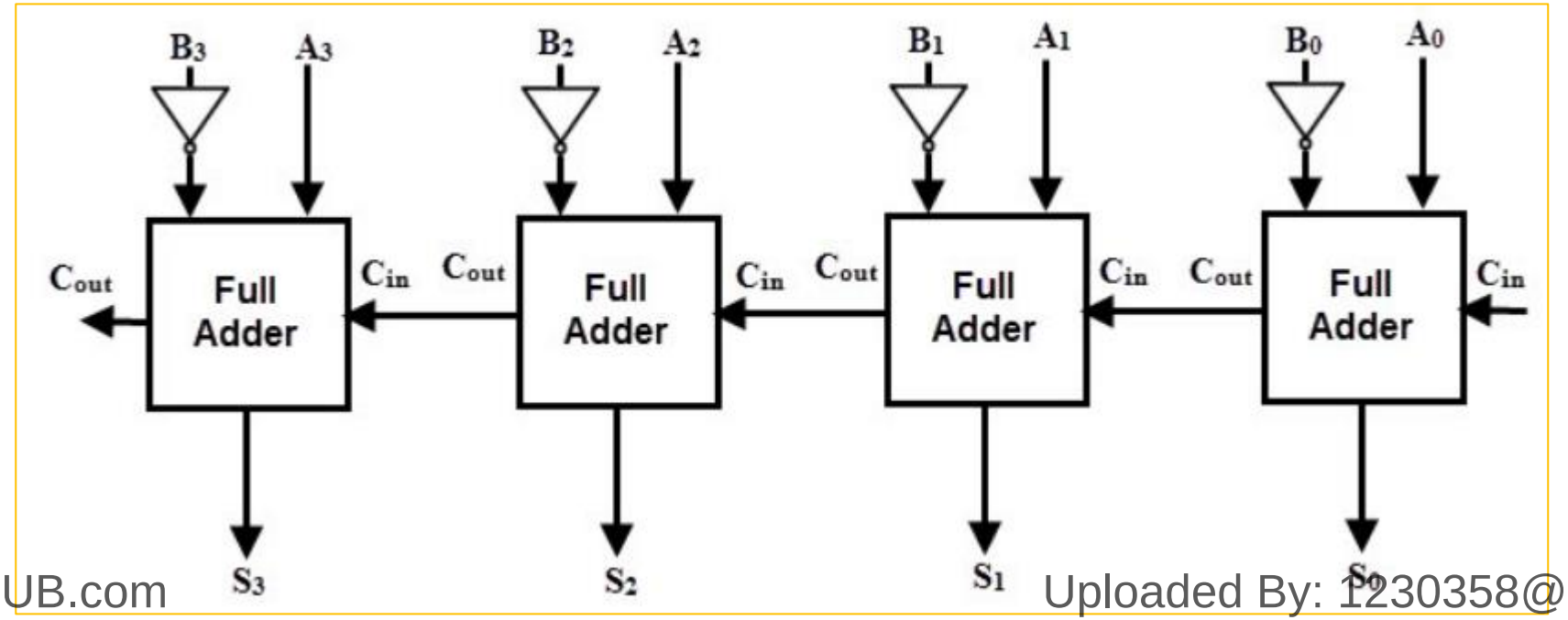


- Each **sum** output requires **two** XOR gates.
- All output **carries** are generated after exactly a **delay** through two levels of gates.
- The **delay** of the carry lookahead adder is **constant**.

Binary Subtractor

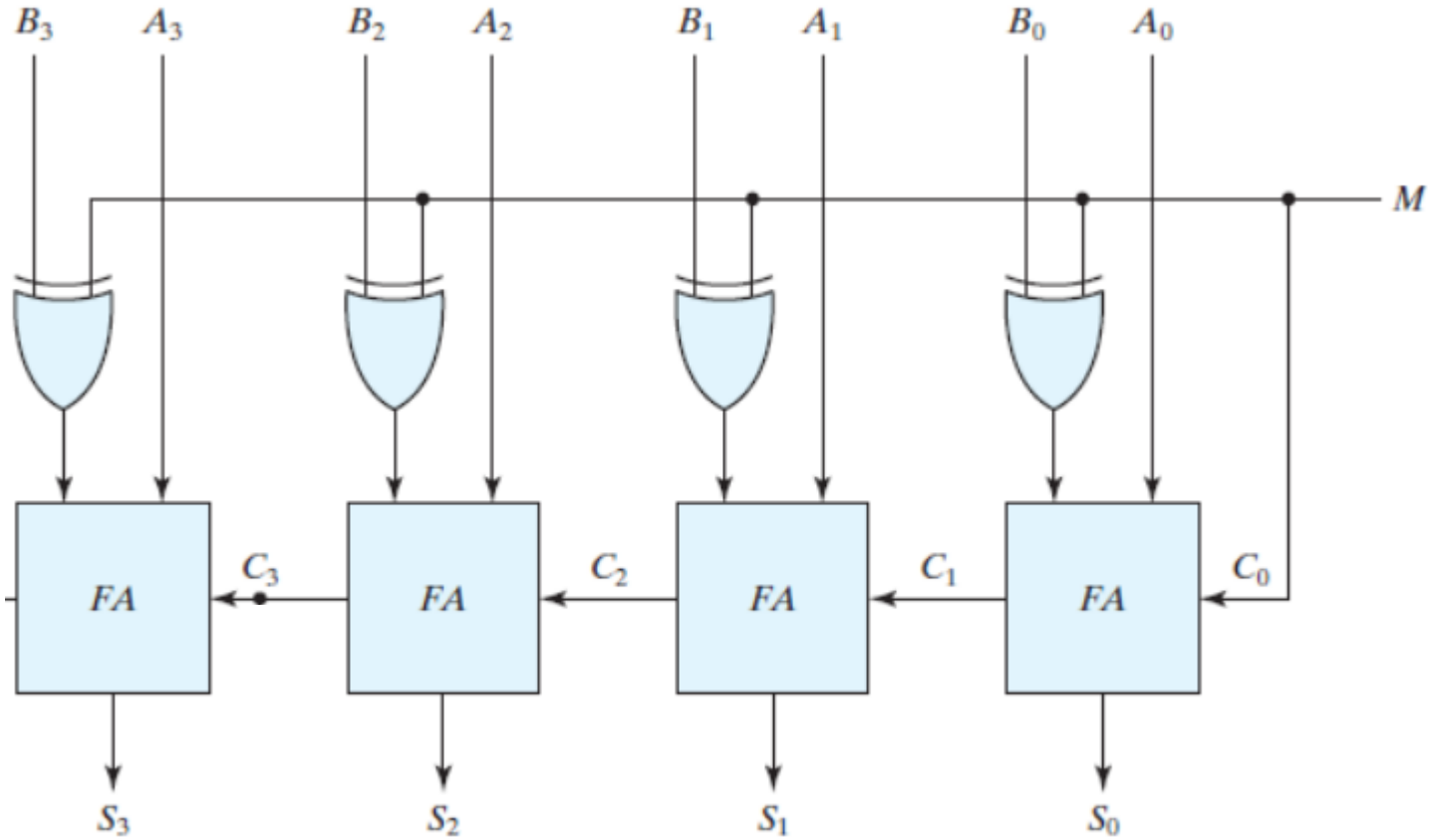
- **Recall**: A − B = A + **2's (B)**
  - 2's complement of B is taken by first finding the 1's complement (**inverting each bit of B**), then a **1** is added.
- This implies that, **Subtraction** can be performed using an **adder by:**
  1) **Invert** the bits of input B *(→ 1's complement of B)*
  2) Change $\mathbf{C_0}$ to **1** *(→ +1)*
  3) Add A & **2's (B)** *(→ A-B)*

  } 2's complement of B
- Binary **subtractor** can be used to perform subtraction for both **signed and unsigned** number systems



Binary Adder-Subtractor

- A four-bit adder can be used to design a circuit that can perform **both** the addition and subtraction operations by Introducing a **Mode Bit (M)** and **4 XOR Gates**
  - **Recall:** B⊕1 = B′ (→ B can be inverted if **Xored** with 1)



1) M = 1 → (A + 1's compliment B + 1)
   I) Unsigned numbers Case:
      if A ≥ B → A - B
      if A < B → 2's compliment of (B - A).
   II) Signed numbers Case:
   → A - B if there is no overflow (V)

2) M = 0 → Adder → (A + B)

The mode bit: M = **0** for **adder**
The mode bit: M = **1** for **subtractor**

Binary Adder-Subtractor

4-bit adder-subtractor example:

					C0	M	S3	S2	S1	S0
A = 6	0	1	1	0	0	0	A + B			
B = 4	0	1	0	0			1	0	1	0
A = 5	0	1	0	1	1	1	A - B			
B = 4	0	1	0	0			0	0	0	1
A = -4	1	1	0	0	1	1	A – B (signed)			
B = -7	1	0	0	1			0	0	1	1

Overflow Review

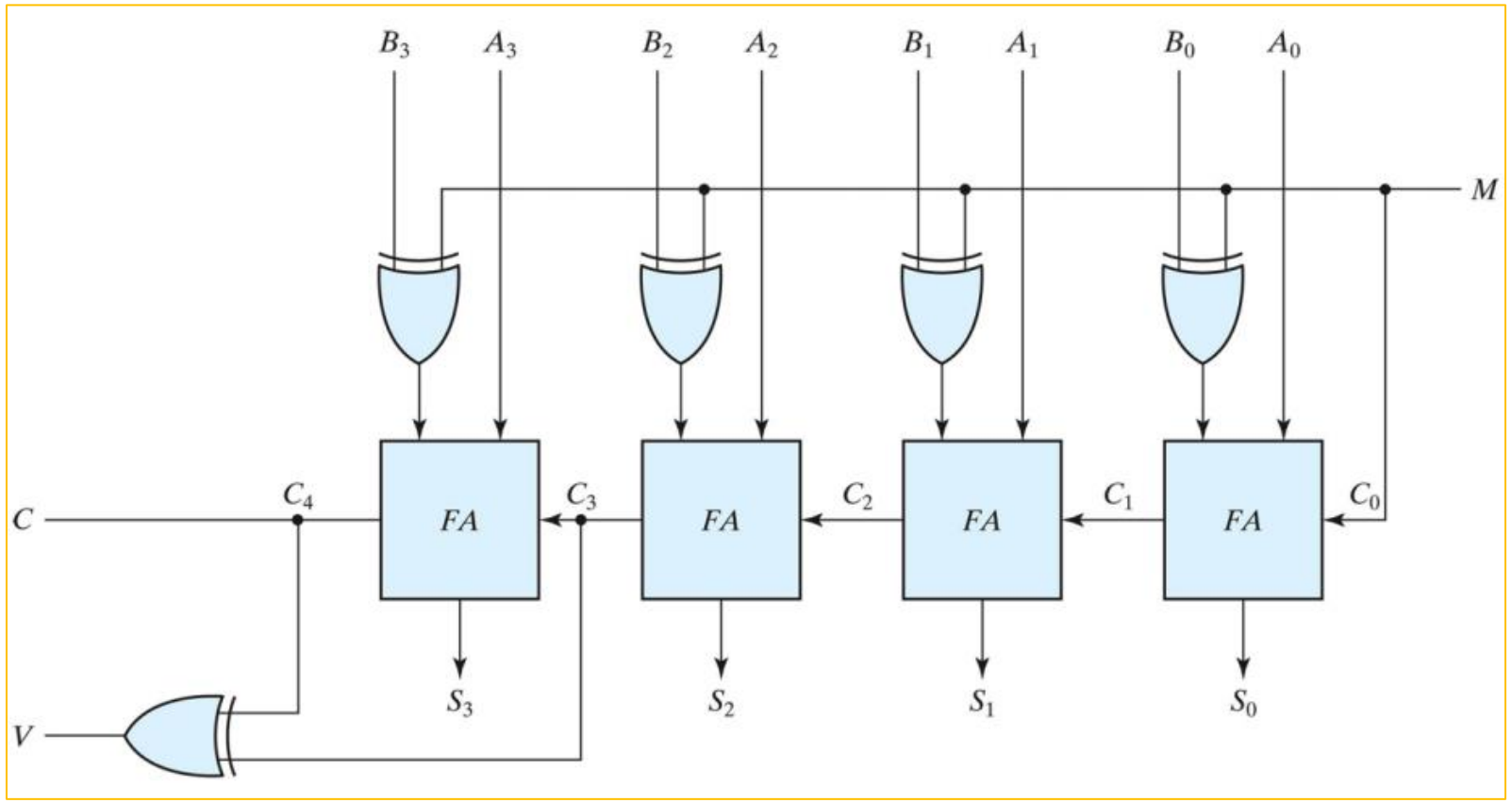
- Overflow occurred when two numbers with **n digits** each are added/subtracted and the result is a number with **n+1 digits**

1) For **Unsigned numbers** an overflow is detected from the **end carry** out of the most significant position.
   - In a 4-bit adder, **A=1111, B=0001** → A+B=1 0000 [S = 0000 , C = 1 → **Overflow**]

2) For Signed Numbers:
   - The **leftmost** bit represents the **sign**
   - When two signed numbers are added, the sign bit is treated as part of the number and the **end carry does NOT** indicate an overflow.
- An overflow may occur if the two numbers added are **both** positive or **both** negative.
  - An overflow condition can be detected by observing the carry into the sign bit position and the carry out of the sign bit position
    - If these **two carries** are **NOT** equal, an overflow has occurred → **XOR gate** function

carries:	0 1	carries:	1 0
+70	0 1000110	−70	1 0111010
+80	0 1010000	−80	1 0110000
+150	1 0010110	−150	0 1101010

Binary Adder-Subtractor



1) M = **1** → (A + 1's compliment B + 1)
   - I) **Unsigned** numbers Case:
     - if A ≥ B → A - B
     - if A < B → **2's** compl. of (**B - A**)
   - II) **Signed** numbers Case:
     - → A - B if there is **NO** overflow (V)

2) M = **0** → Adder → (A + B)

I. **Unsigned** numbers Case: **C** bit detects a carry after addition or a borrow after subtraction
II. **Signed** numbers Case: **V** bit detects an **overflow**
   1) V = 0 means **NO** overflow occurred and the n-bit result is correct
   2) V = 1 means overflow has occurred and the result needs **n + 1** bits to fit→ The n-bit result is **incorrect**

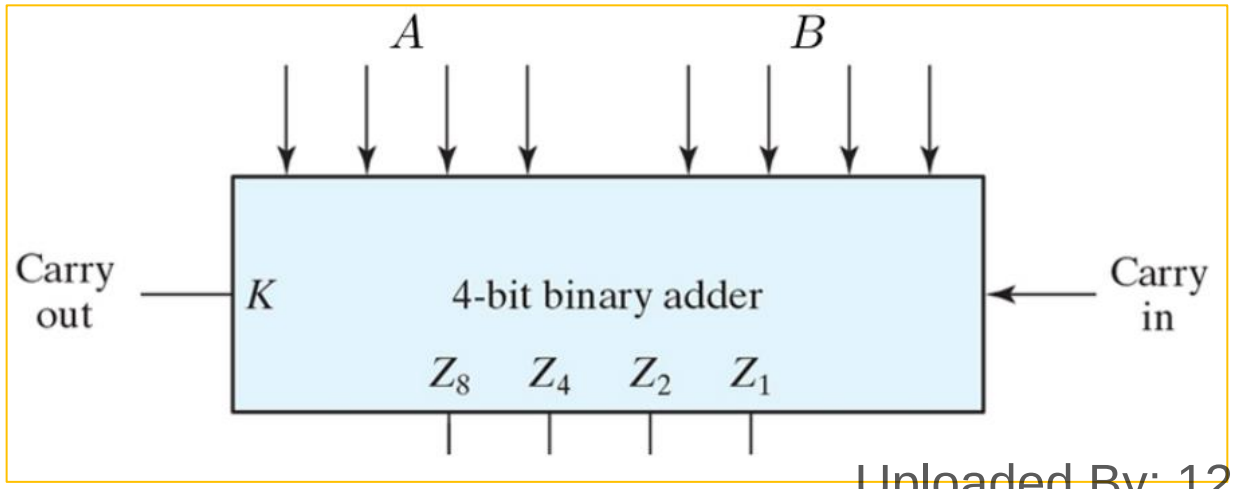
BCD Adder

- Used to add two decimal digits in BCD
- This adder is present in systems used to perform decimal addition **directly** (e.g. calculators, etc.)

- This adder accepts **two** decimal numbers (A and B) in coded form (**BCD**) and **one** carry digit from the previous stage
- The carry digit from the previous stage could be either 0 or 1. → need just **one-bit** for the carry digit

- The **minimum** possible sum at any stage could be 0 + 0 + 0 = 0
- The **maximum** possible sum at any stage could be 9 + 9 + 1 = **19**

- Requires a minimum of **9 inputs** and **5 outputs**
  - Input (A: 4 Bits , B: 4 Bits, Carry in: 1 Bit) Output (Result: 4 Bits, Carry out: 1 Bit)



BCD Adder

Binary Sum					BCD Sum					Decimal
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

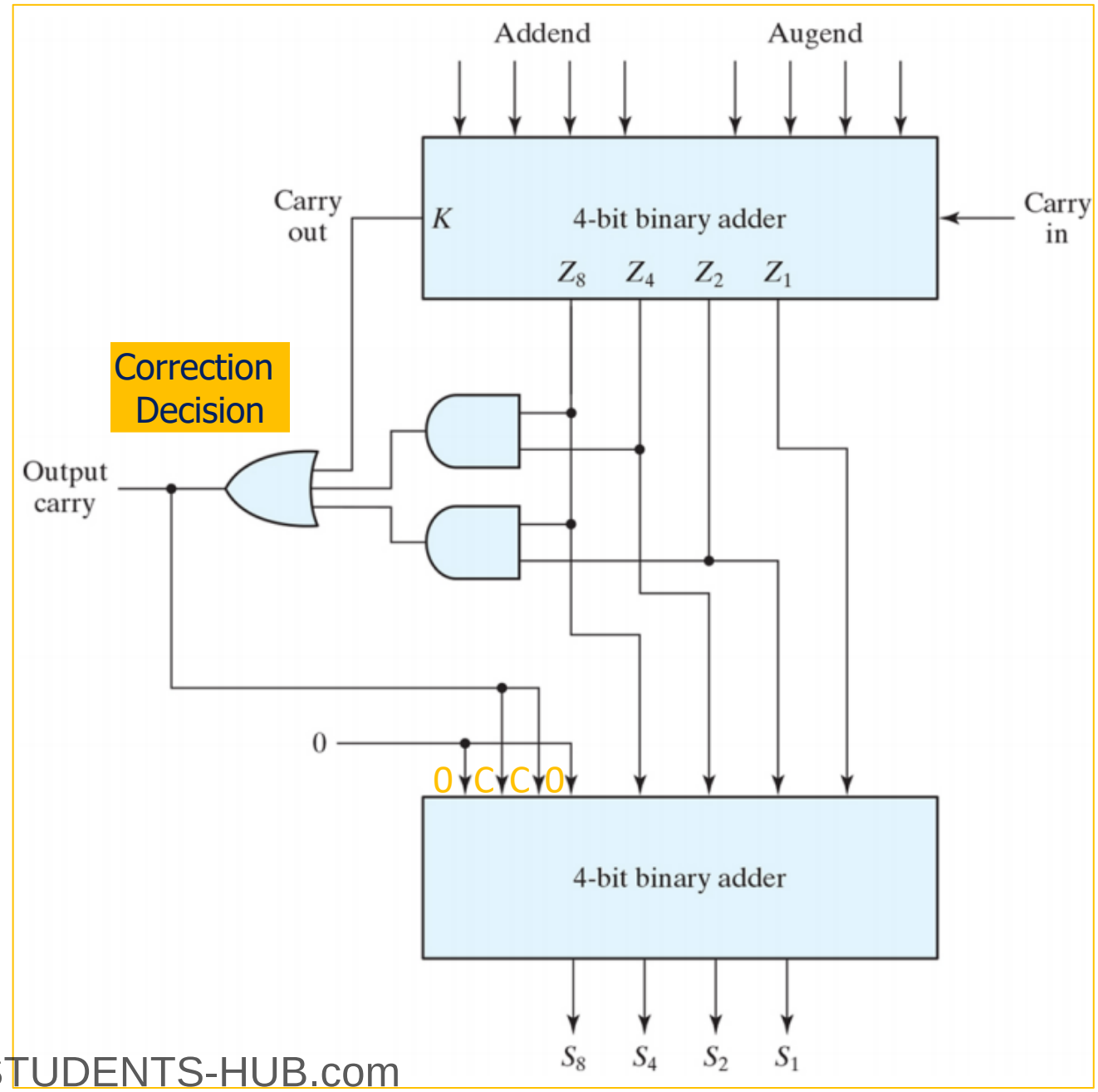
Recall: a **correction** in the sum is needed when the sum is **greater** than 9. The correction is **adding 6** to the sum.

→ The BCD adder will then consist of the 4-bit binary adder. **A second 4-bit** binary adder is needed to add **6** to the sum when it is greater than 9.

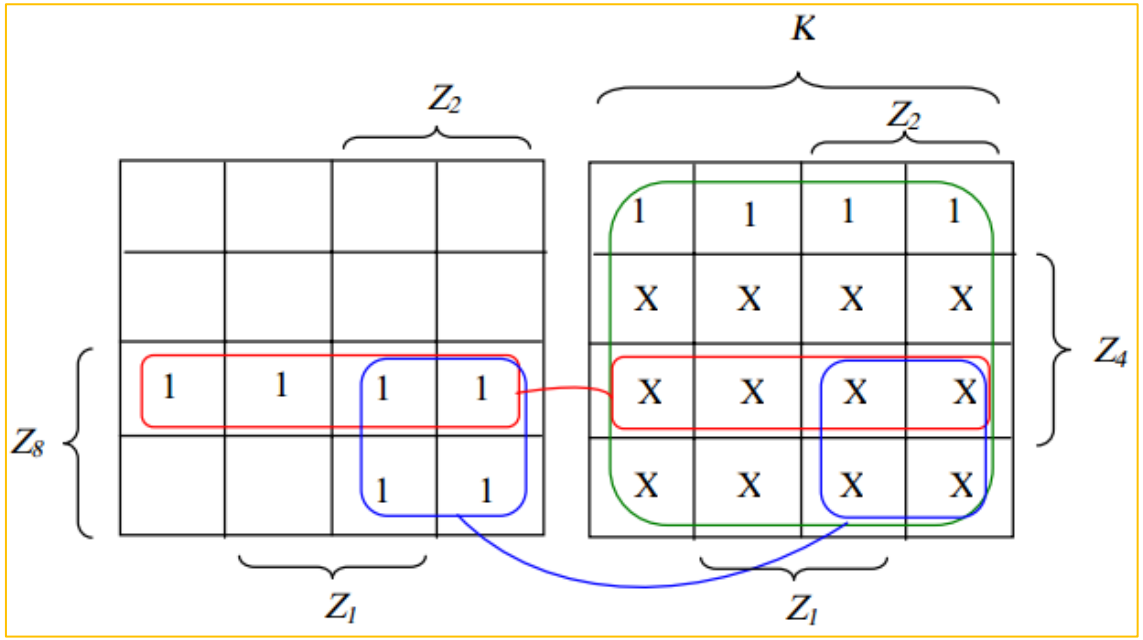
When C = 0, **Do Nothing!**

When C = 1, **Add 0110** to the binary sum and provide an output carry for the next stage.

BCD Adder

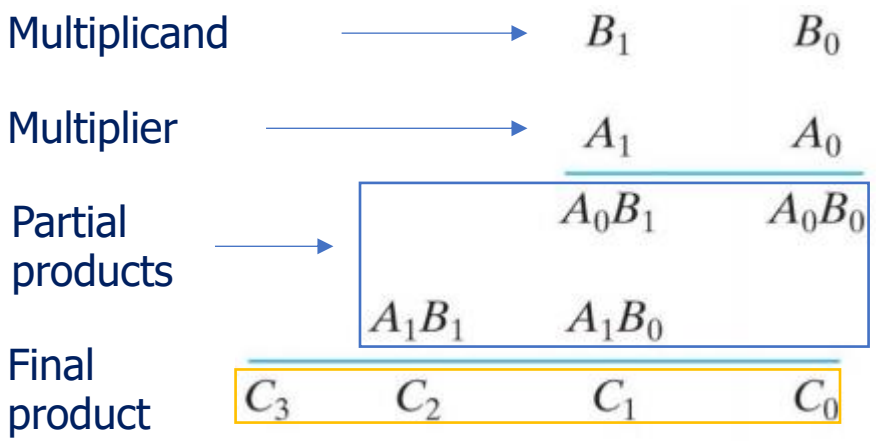


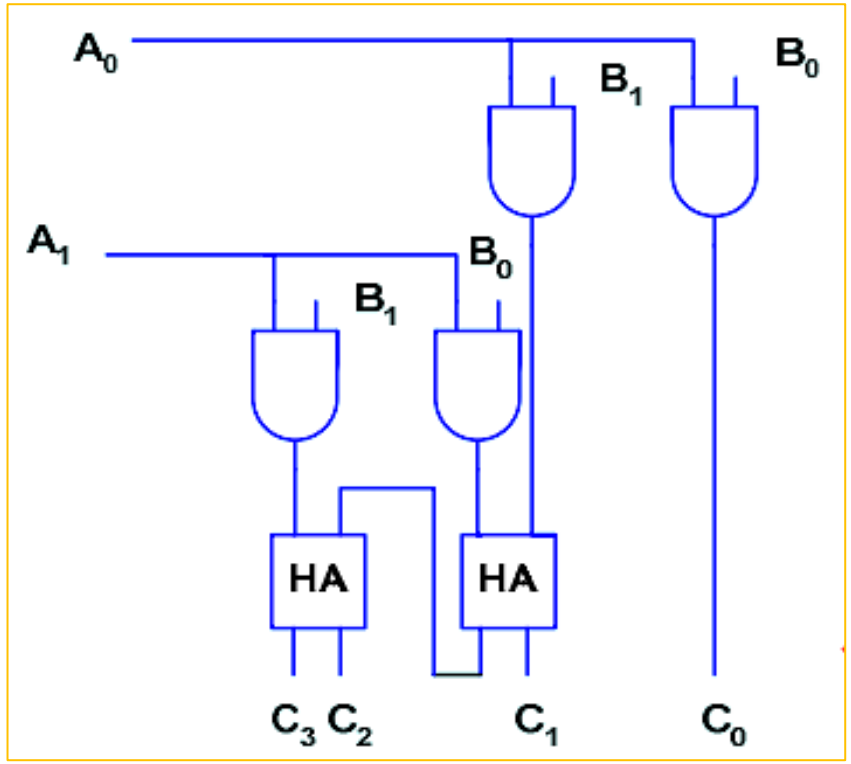
$$C = K + Z_8Z_4 + +Z_8Z_2$$



Binary Multiplier

- **Recall**: Binary multiplication is done in the same way as decimal multiplication
- When multiplying two binary numbers, A and B, the **multiplicand** is multiplied by **each** bit of the **multiplier** starting from the least significant bit.
- Each such multiplication forms a **partial product**.
- **Successive** partial products are **shifted one** position to the **left**.
- The **final** product is obtained from the **sum** of the **partial** products.





Observe: The **multiplication** of two bits such as A_0 and B_0 produces a **1** if **both** bits are 1; otherwise, it produces a **0**. This is identical to an **AND** operation.

Implementation could be done using **Half** adders & **AND** gates

Binary Multiplier

- For **J-bits** Multiplier and **K-bits** multiplicand we need:
  - **J x K** AND gates, and
  - **(J – 1)** K-bit adders
  - The result will be a product of (**J + K**) bits.

Example:

	K=4		B_3	B_2	B_1	B_0
	J=3			A_2	A_1	A_0
			A_0B_3	A_0B_2	A_0B_1	A_0B_0
		A_1B_3	A_1B_2	A_1B_1	A_1B_0	+
	A_2B_3	A_2B_2	A_2B_1	A_2B_0		+
C_6	C_5	C_4	C_3	C_2	C_1	C_0

JxK=3x4 =**12** AND Gates

J-1=3-1 =**2** (**4**-bit) Adders

J+K=3+4 =**7** bits Result

Binary Multiplier

Example:

Design a 4-bit by 3-bit Binary Multiplier

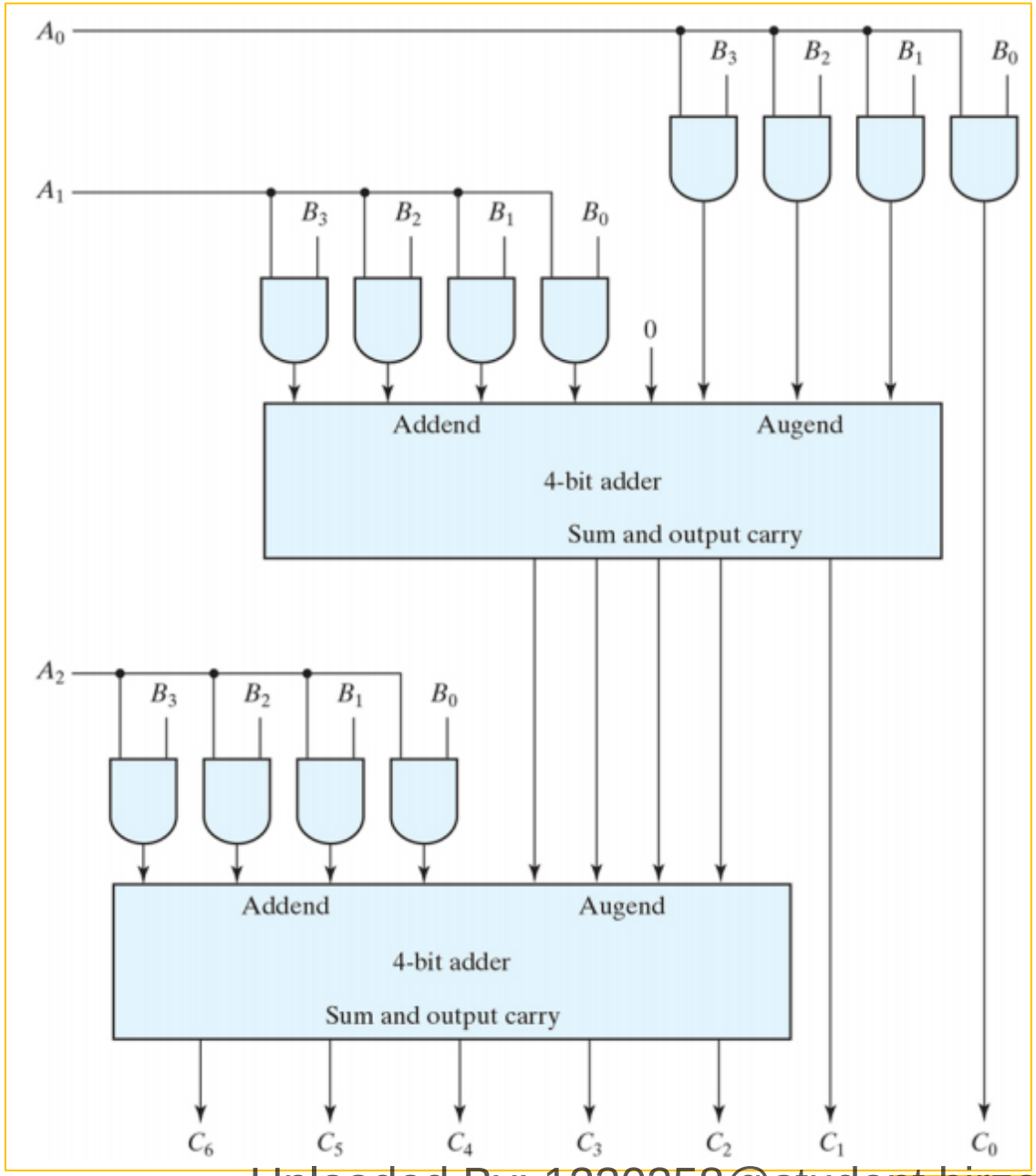
K=4

J=3

12 AND Gates

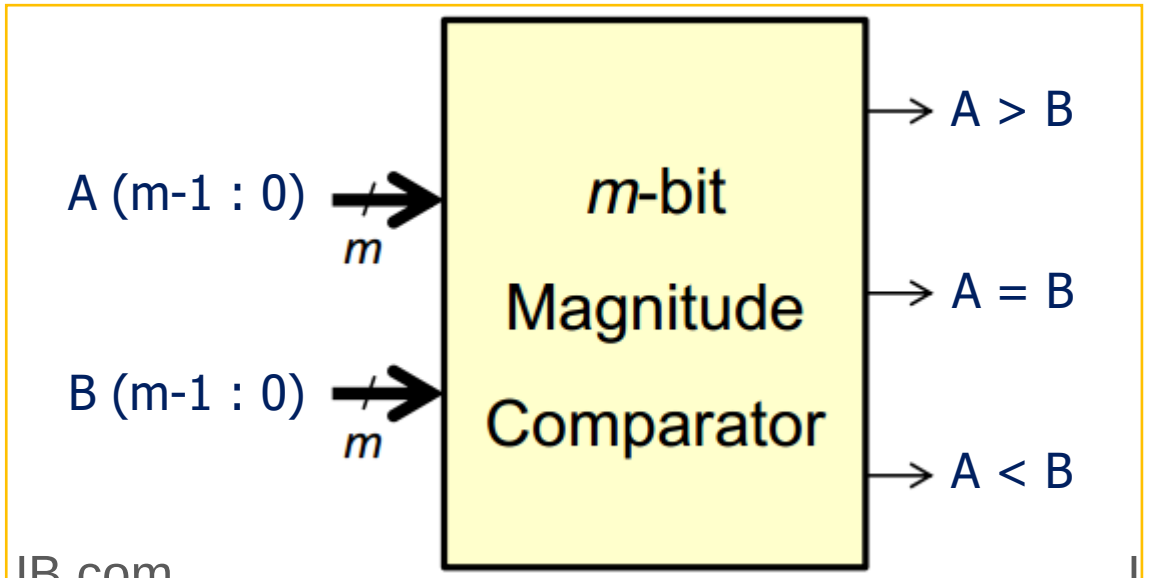
2 (4-Bit) Adders

7 bits Result



Magnitude Comparator

- A magnitude Comparator: a combinational circuit that **compares** two unsigned numbers A and B and determines their relative magnitudes.
  - Two Inputs:
    1) Unsigned integer A (m-bit number)
    2) Unsigned integer B (m-bit number)
  - Three outputs:
    1) A > B (GT output)
    2) A = B (EQ output)
    3) A < B (LT output)
  - **Exactly one** of the three outputs must be equal to **1** while the **remaining two** outputs must be equal to **0**



Magnitude Comparator

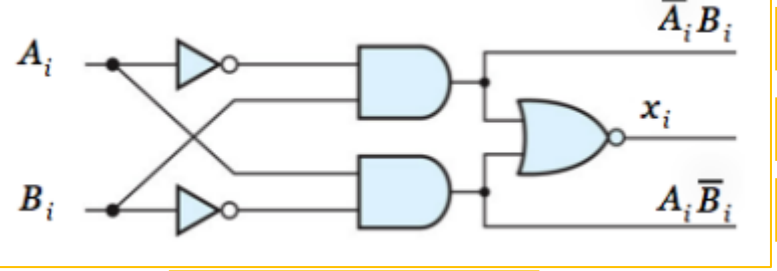
- **Consider:** a circuit to compare two **4-bit** numbers A and B
  - Input/output: 8 inputs, 3 outputs ⇒ **huge** truth table (256 Rows)

$$A = A_3\ A_2\ A_1\ A_0$$
$$B = B_3\ B_2\ B_1\ B_0$$

- A better method to design this circuit is to follow the **systematic** way of comparison, where we compare **each pair** of bits starting from the **most significant** bit.
  - If **all pairs** are **equal** → A=B.
  - If we find a **difference** in the compared bits (i.e. one is 1 and the other is 0), → the number containing the **1** is **larger**

Case 1 (A = B):

- **All pairs** of bits should be equal
  ($A_i = B_i$, i = {0, 1, 2, 3})
- Equality using **XNOR** operation
  $x_i = A_i B_i + A_i' B_i'$, i = {0, 1, 2, 3}
- A = B **only if all** x_i's are **1**
  → **(A = B) = $x_3x_2x_1x_0$**



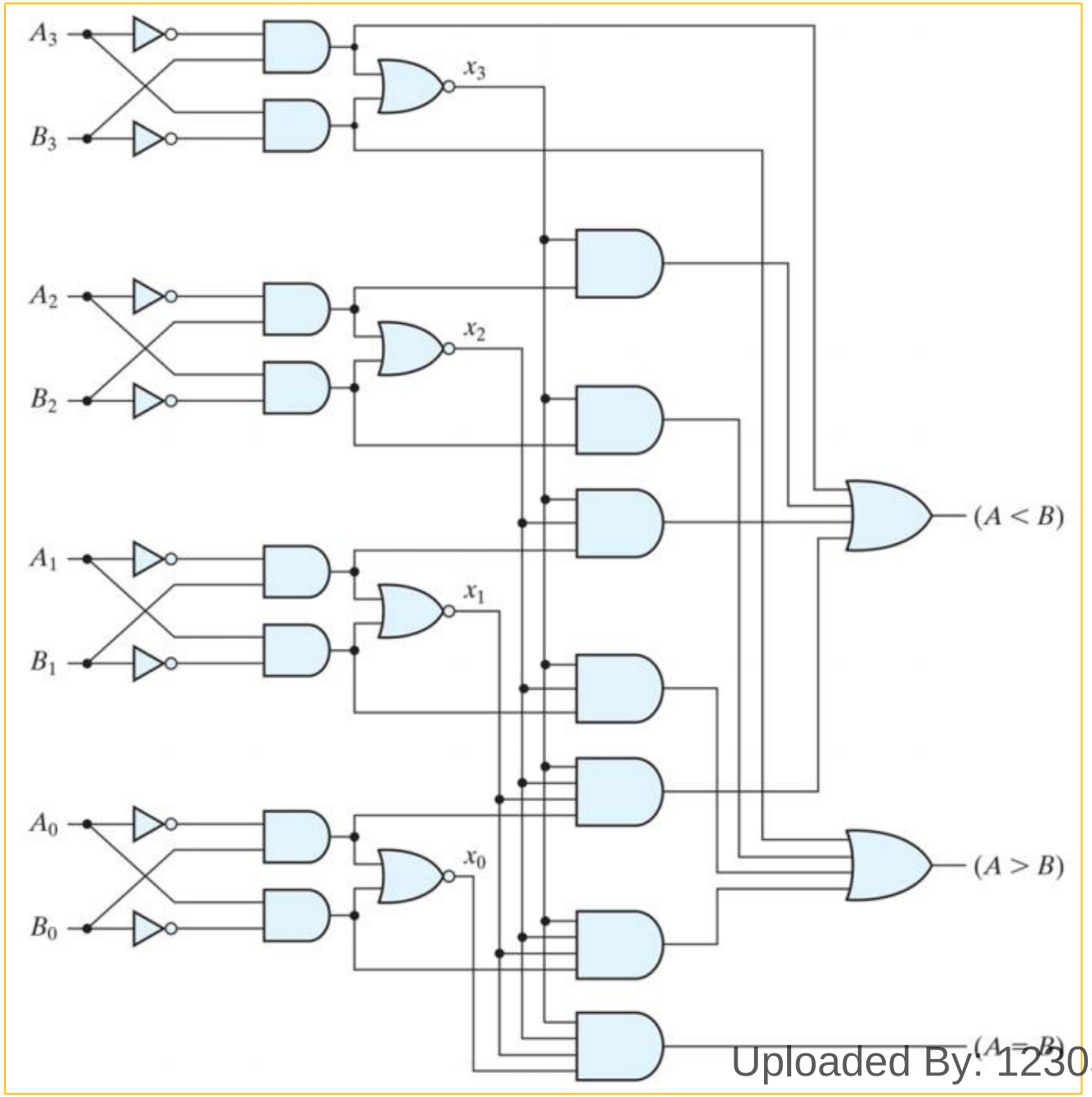
1 bit Comparator

Case 2 and 3 (A < B or A > B):

- Compare the **most-significant** bits (A_3 & B_3)
- If the two bits are **equal**, Compare the **next pair** of bits (A_2 & B_2)
- Continue comparing the subsequent pairs of bits if the previous comparisons are **equal** (A_1 & B_1) and (A_0 & B_0)
- If **Ai≠Bi** at any stage, then:
  A < B if A_i = 0 and B_i = 1 or, simply, if $A_i' B_i$ = 1
  OR **A > B** if A_i = 1 and B_i = 0 or, simply, if $A_i B_i'$ = 1
- → In logical expressions:

$$(A < B) = A_3' B_3 + x_3 A_2' B_2 + x_3 x_2 A_1' B_1 + x_3 x_2 x_1 A_0' B_0$$
$$(A > B) = A_3 B_3' + x_3 A_2 B_2' + x_3 x_2 A_1 B_1' + x_3 x_2 x_1 A_0 B_0'$$

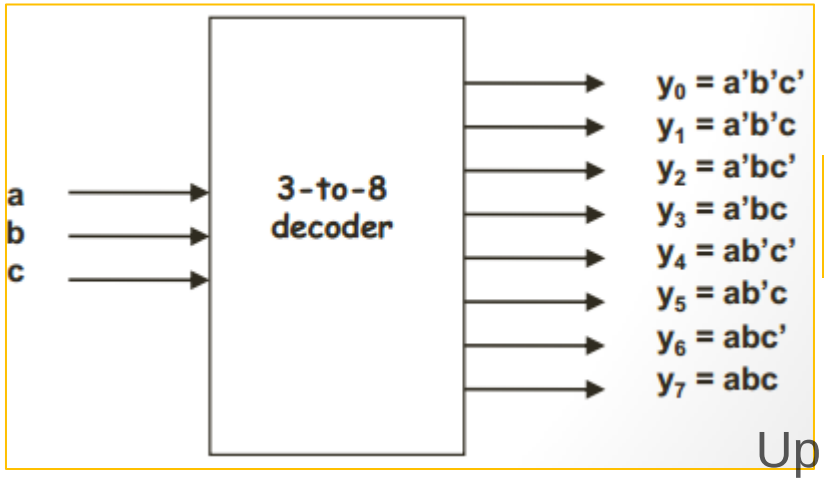
Magnitude Comparator



Decoders

- Imagine that we have a circuit with a **3-bit** input A and a **single-bit** output X. The circuit should **indicate** when a particular code of 110 appears at the input. In other words, the circuit should give **X=1** at its output when binary number $\mathbf{A_2A_1A_0 = 110}$ occurs at its inputs
- This called a **decoding** process as the circuit indicates when a particular code appears at the inputs → that's why we may call it a **decoder**
- This could be extended to consider **all possible** combinations of the input bits

- **Recall**: A binary code of **n** bits is capable of representing up to $\mathbf{2^n}$ **distinct** elements
- A **decoder** is a combinational circuit that **converts** binary information from **n** input lines to a maximum of $\mathbf{2^n}$ **unique** output lines
  - This is called an **n-to-m** line decoder. ($m \leq \mathbf{2^n}$ **,** with $\mathbf{m = 2^n}$ we call it a Full Decoder)
- The output whose value is **1** represents the **minterm** equivalent to the **binary input**.
- Each **combination** of inputs will activate a **unique** output



Binary **decoder** has **n** inputs and $\mathbf{2^n}$ outputs
All $\mathbf{2^n}$ minterms are generated

Example: 3x8 Decoder

- The 3 inputs are **decoded** into 8 outputs, each representing **one** of the **minterms** of the three input variables (**Minterms Indicator**)
- Could be considered as **Binary-to-Octal** 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

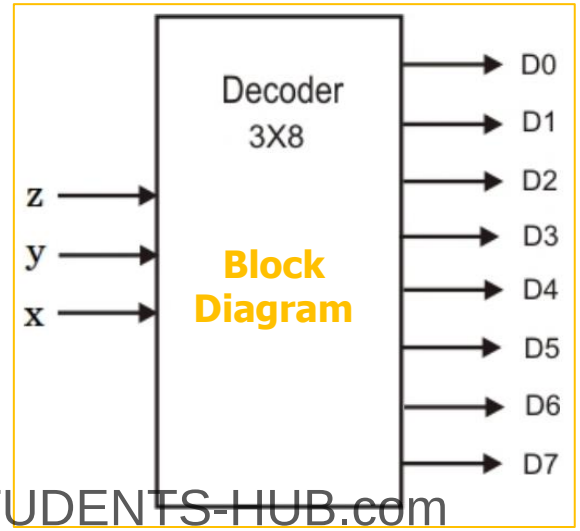
ONLY **ONE** Output is Activated (=1)

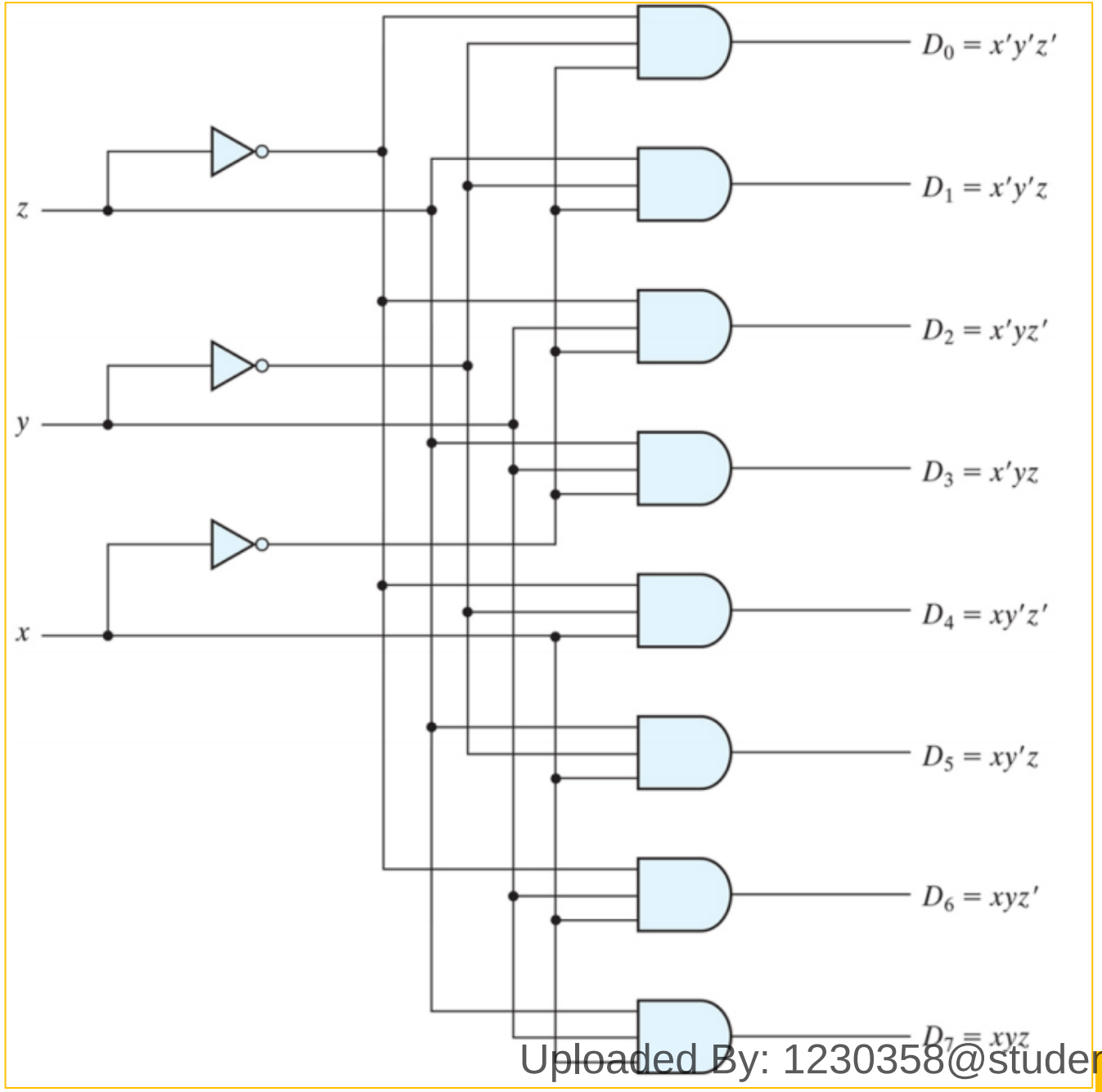
Example: 3x8 Decoder

This decoder can be implemented using three **inverters** and eight **AND** gates

Only **one** output is **1** while the **other** seven are **0**

The **output** whose value = 1 represents the **minterm** equivalent of the binary number currently **available** in the input lines

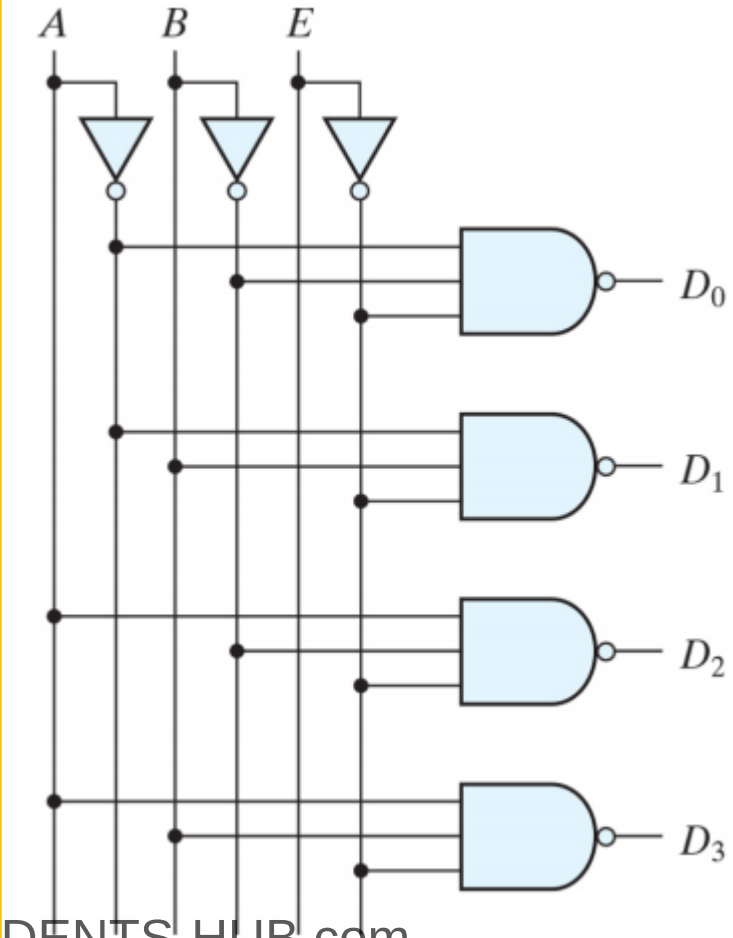




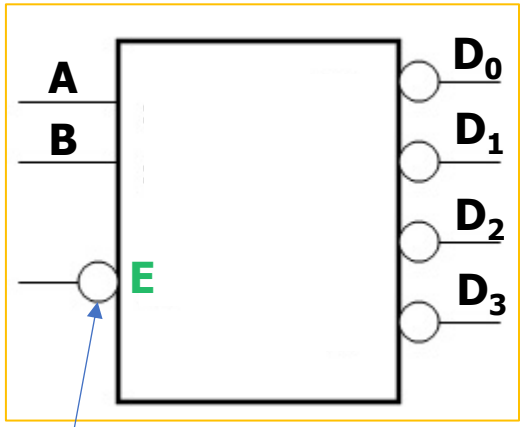
Decoders

- Decoders may constructed using **NAND** gates (**Inverted** Output Decoders)
- Decoders may include one or more **enable** inputs to control the circuit operation (**Demultiplexer**)
  - An **enable** is an **extra** input that will **activate** or **shut off** the Decoder

Inverted Output 2 x 4 Decoder with **Enable**



E	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0



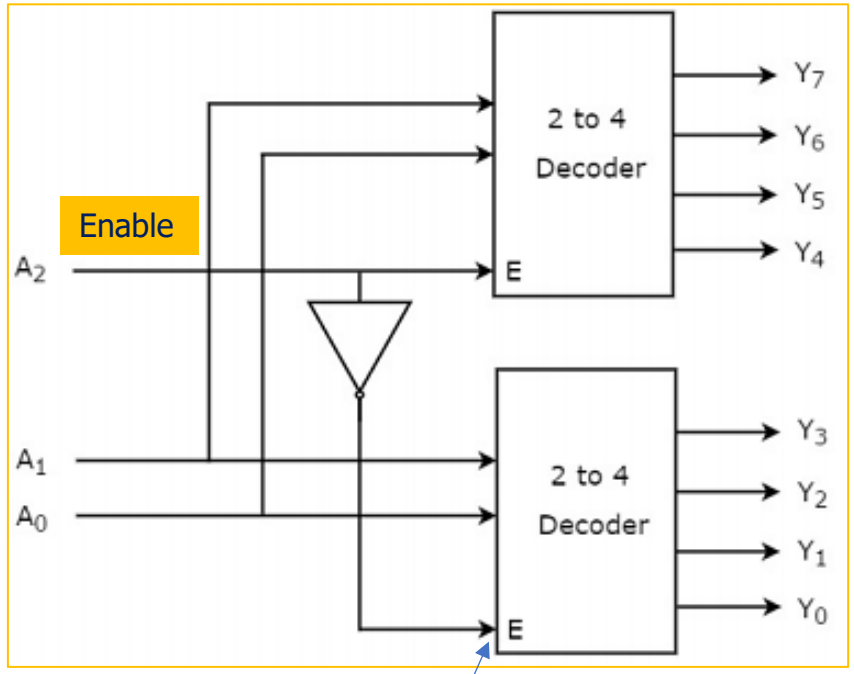
The **bubble** indicates that the decoder is **enabled** when E=**0**
Sometime denoted as **E**

Active High: When Outputs are **1** (**Normal** Output, **AND** Gates Used)
Active Low: When Outputs are **0** (**Inverted** Output, **NAND** Gates Used)

Active High → **minterms Generator**
Active Low → **maxterms Generator**

Decoders with **enable** inputs can be **connected** together to form a **larger** decoder circuit

Active high **3x8** Decoder using **2x4** Decoders



Enable

A_2	A_1	A_0	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

No bubble → the decoder is **enabled** when E=**1**

Be careful: **Don't get confused** between **E** and the **input** variable (A_2)

Combinational Logic with Decoders

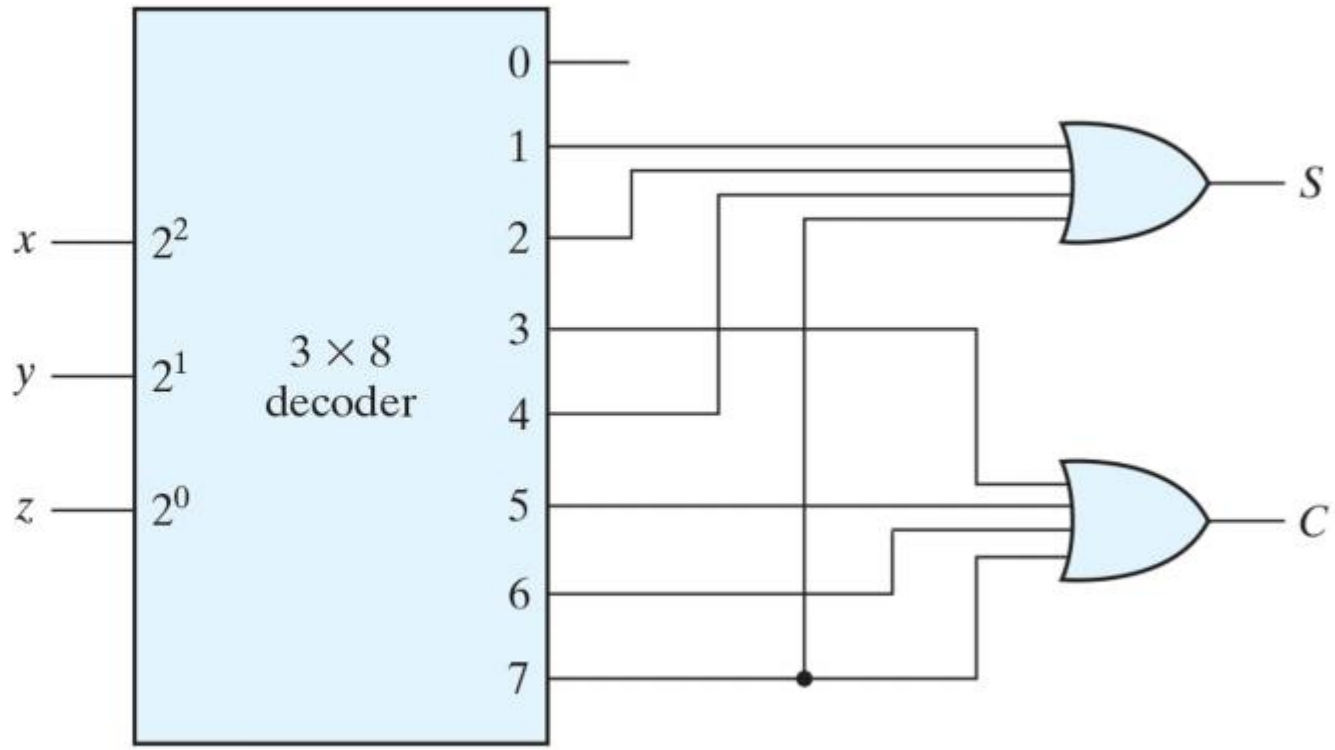
- **Recall**: A decoder provides the **2^n minterms** of **n** input variables.
- **Recall**: Any Boolean function can be expressed in **sum-of-minterms** form.
- A decoder together with an external OR gate provides a logic implementation of the function
  - Since **all** the **minterms** of the function are **available** at the output then there is **NO** need for simplification
  - **Inputs** to each **OR** gate are selected from the **decoder outputs** according to the **list of minterms** of each function

A combinational circuit with **n** inputs and **m** outputs can be implemented with an **n-to-2^n decoder** and **m OR** gates

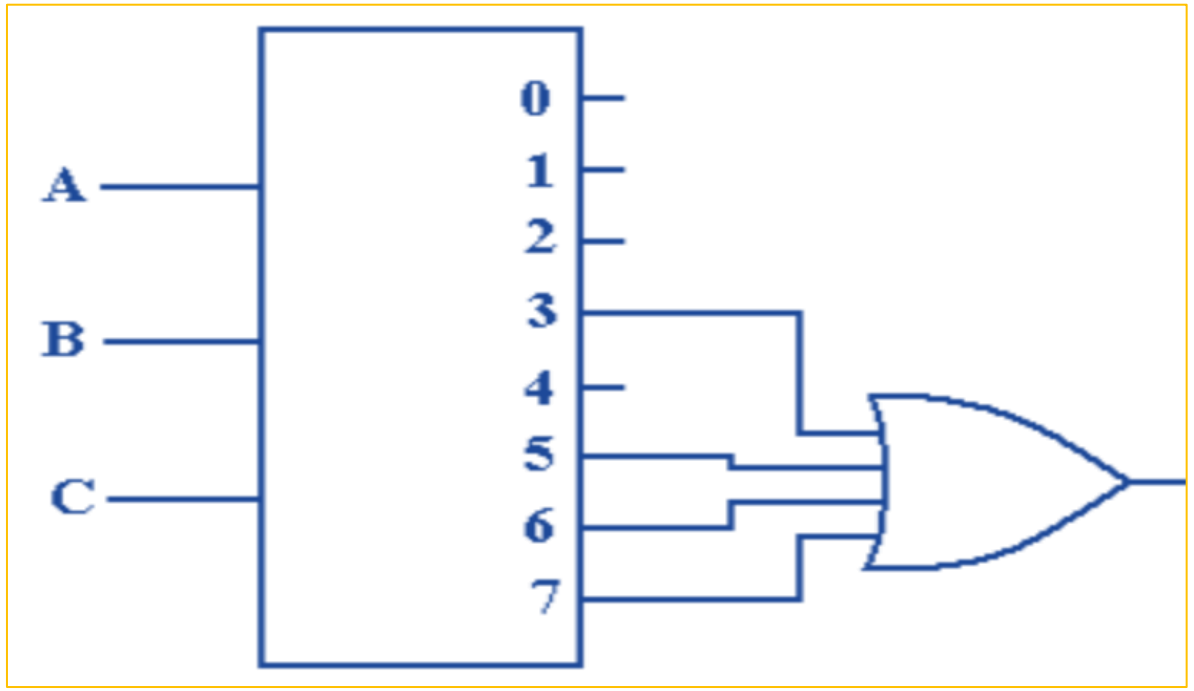
Example: **Full Adder** Implementation Using Active High (**And**) Decoders

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

$$S(x, y, z) = \Sigma(1, 2, 4, 7)$$
$$C(x, y, z) = \Sigma(3, 5, 6, 7)$$

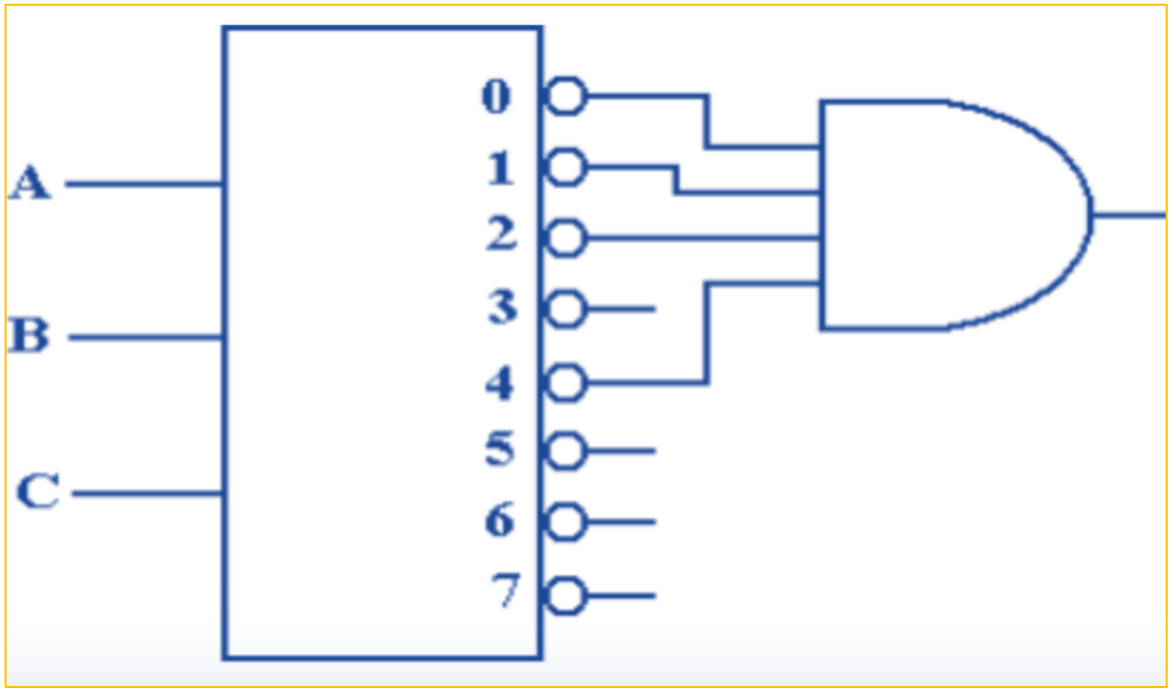


Example: Implementation of $F = \Sigma(3, 5, 6, 7)$, Using Active Low (**NAND**) Decoders



Active High

$F = \Sigma(3, 5, 6, 7)$



Active Low

$F = \Pi(0, 1, 2, 4)$

F=0, if any of these terms is **selected**
Remember: **Selected** in Active **Low** → **0**

General Considerations

- A function with a **long** list of **minterms** requires an **OR** gate with a **large** number of inputs.
- A function having a list of **k minterms** can be expressed in its **complemented** form **F'** with ($2^n - k$) **minterms**
- If the number of **minterms** in the function is **greater** than $2^n/2$ → **F'** can be expressed with **fewer minterms**
  - Same Applied in the case of **maxterms**
- **Recall:** Active-**High** Decoder: **minterm** generator & Active-**Low** Decoder: **maxterm** generator

Important!!

Steps for Optimized Implementation Using Decoders

1) Select **Simplest/Minimal** Form:
   - Evaluate both **F and F'** → choose the one with the **fewest** terms
2) Choose **Decoder and External Gate**
   - Minterm Generation (Active-**High** Decoder): Use **OR** with **F**, **NOR** with **F'**
   - Maxterm Generation (Active-**Low** Decoder): Use **AND** with **F**, **NAND** with **F'**
3) Optimize with Smaller Decoders (if needed/required)
   - Assign the **most significant variable** as an **enable input** for smaller decoders.
   - Share **remaining inputs** across the decoders.

Decoder Type Selection and External Gate Combination

Function Type/Form	Decoder Type	Combining Gate	Explanation
F (fewest minterms)	Active-High (Minterm Generator)	**OR Gate** to combine minterm outputs (of F)	OR gate groups outputs representing **F** minterms
F' (fewest minterms)	Active-High (Minterm Generator)	**NOR Gate** to combine minterm outputs of (F')	NOR gate groups outputs for **F'** minterms → F
F (fewest maxterms)	Active-Low (Maxterm Generator)	**AND Gate** to combine maxterm outputs (of F)	AND gate groups outputs representing **F** maxterms
F' (fewest maxterms)	Active-Low (Maxterm Generator)	**NAND Gate** to combine maxterm outputs (of F')	NAND gate groups outputs for **F'** maxterms, → F

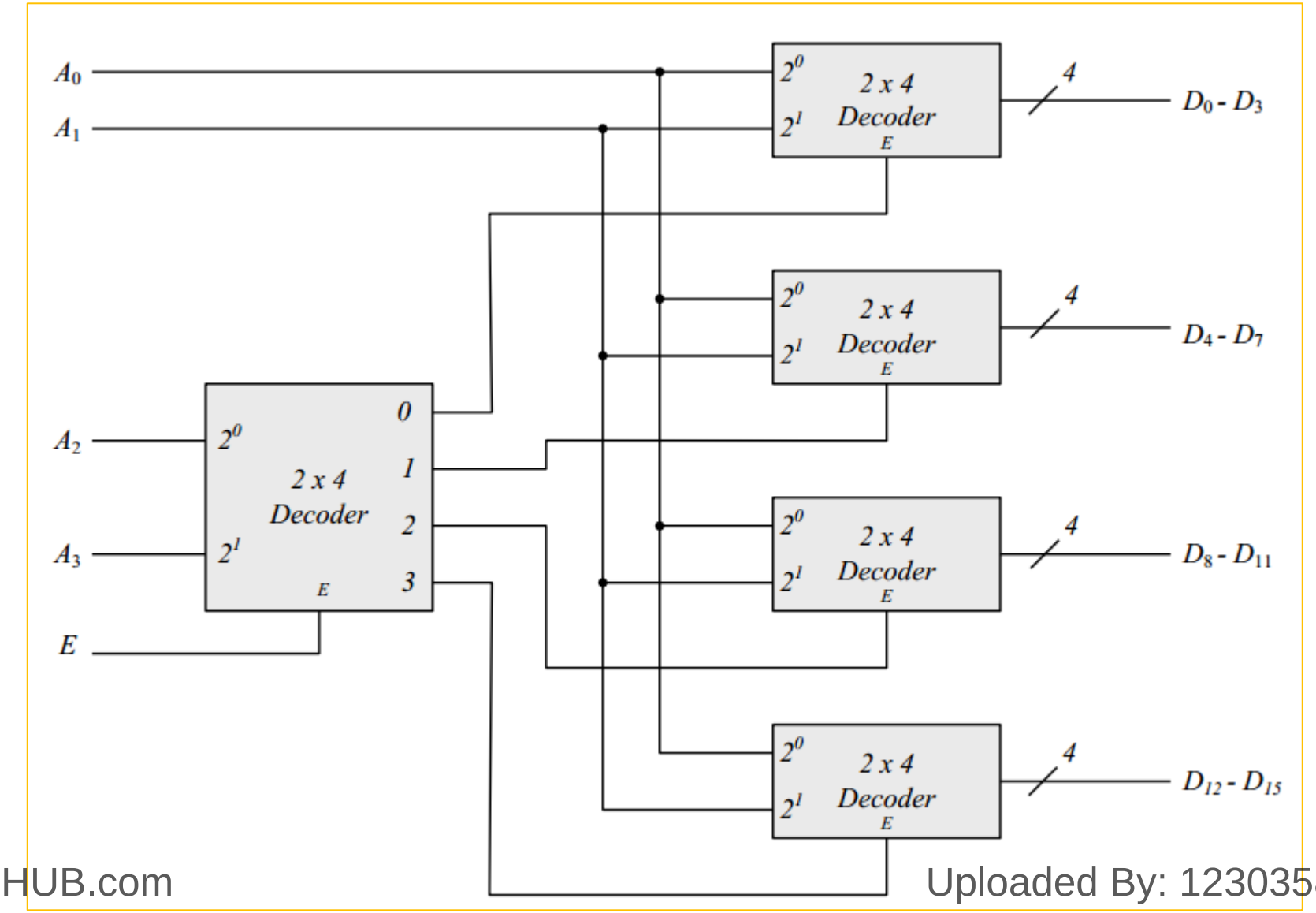
Summary

Step	Description
1. Simple Form	Determine whether F or F' has **fewer** terms (minterms/maxterms).
2. Select Decoder	Choose Active-**High** for minterms, Active-**Low** for maxterms.
3. Combine with Gate	Use OR/NOR for Active-High (F/F'), AND/NAND for Active-Low (F/F')
4. Smaller Decoders	Enable input used for MSB, others shared among decoders (As Required)

Example:

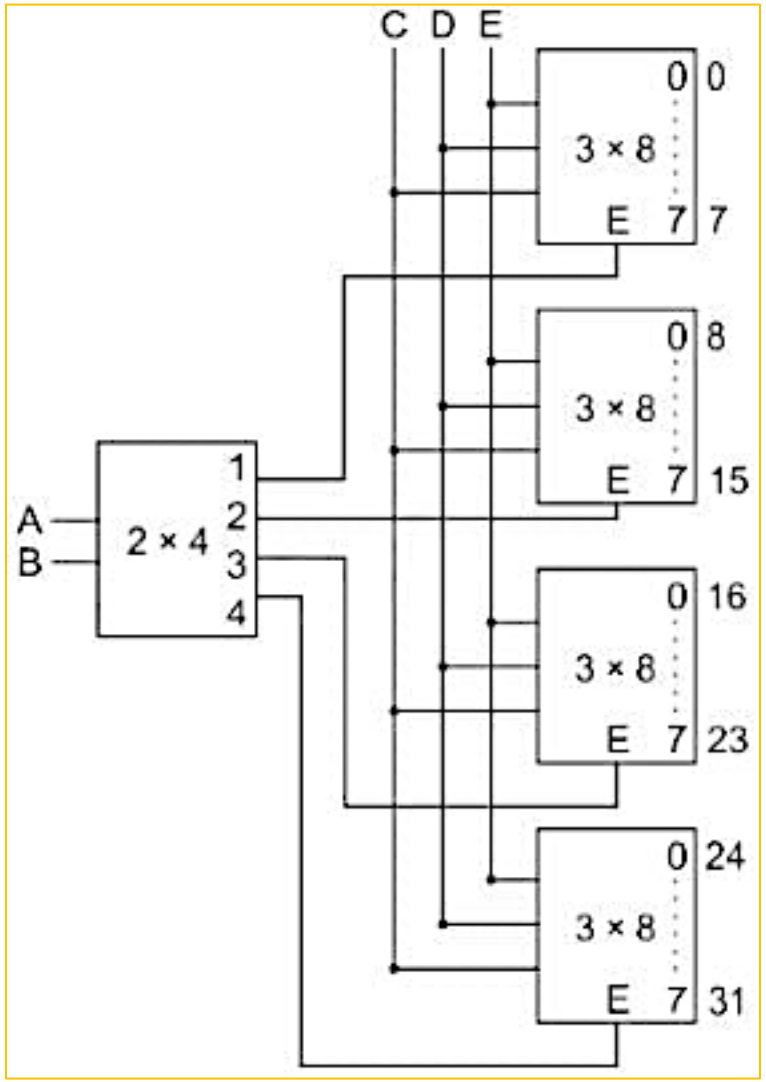
$$F_1(A, B, C) = \sum(3, 5)$$
$$F_2(A, B, C) = \sum(2, 4, 5, 6, 7)$$

Extra Example: Construct a **4-to-16** decoder with **Five 2-to-4** decoders with **enable**



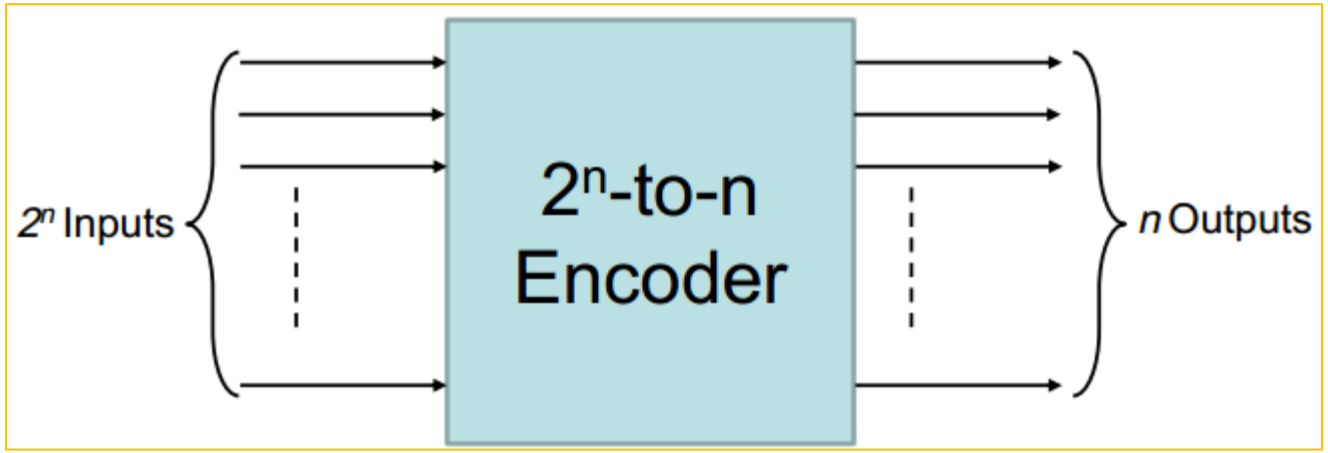
Extra Example:

Construct a **5-to-32** decoder with **four** **3-to-8** decoders with **enable** and a **2-to-4** decoder



Encoder

- An **Encoder** is a digital circuit that performs the **inverse** operation of a Decoder
  - It has **maximum** of 2^n **input** lines and **n output** lines
  - **Output** lines give the **binary code** of the **input** lines
  - **Only 1 input** line should be **active** at a time



Encoder

Example: Design an Octal to Binary **Encoder** (8-to-3 Encoder)

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

$Z = D1 + D3 + D5 + D7$

$Y = D2 + D3 + D6 + D7$

$X = D4 + D5 + D6 + D7$

Limitations

1) If **two** inputs are **active simultaneously** (say D_3=D_6= 1), then the output = 111 which does **NOT** represent either binary 3 or binary 6 (**Wrong Code**)
   - Encoder circuits must establish an **input priority** to ensure that **only one** input is encoded

2) If ALL Input = 0s → ALL output = 0s which is the **same** output when D_0=1
   - This discrepancy can be resolved by providing **one more output** to indicate whether at **least** one input is equal to 1

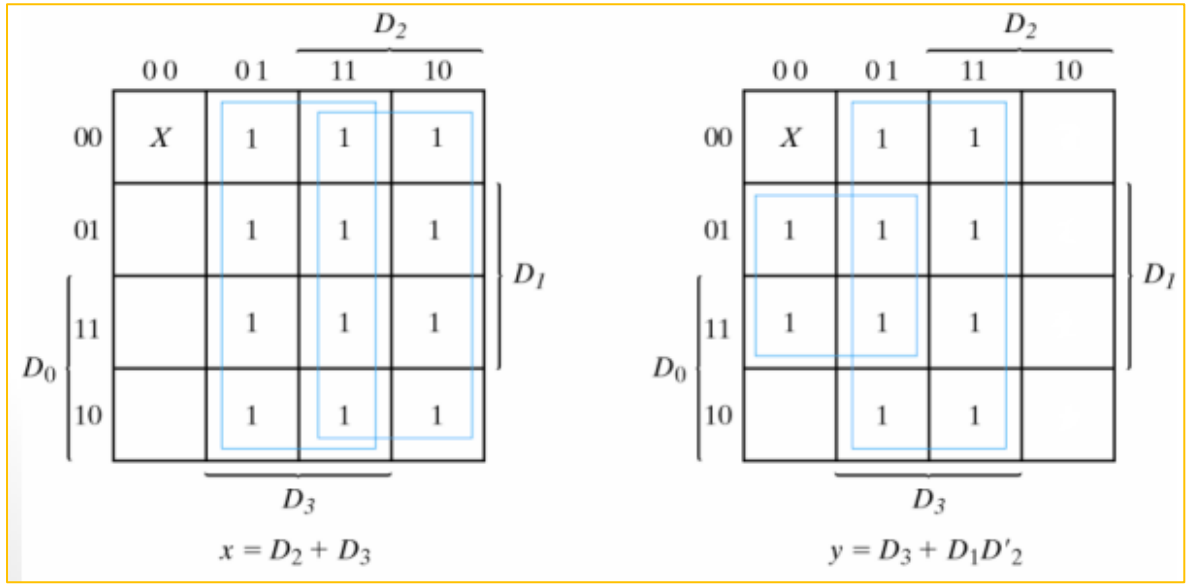
To overcome these limitations, we may use a **priority encoder**

Priority Encoder

- A **priority encoder** is an encoder circuit that includes the **priority** function
- if **two or more inputs** are equal to **1** at the same time →
  - the **input** having the **highest priority** will take precedence
- A **valid** bit (**v**) is introduced at output to indicate the **invalid all 0s** input combination

4-to-2 Priority Encoder

Inputs				Outputs		
D_0	D_1	D_2	D_3	x	y	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1



In place of the '**X**', you **substitute** '**1**' then a '**0**':
X → **2** minterms (0,1)
XX → **4** minterms (00,01,10,11)
XXX → **8** minterms (000,001,010,011,100,101,110,111)

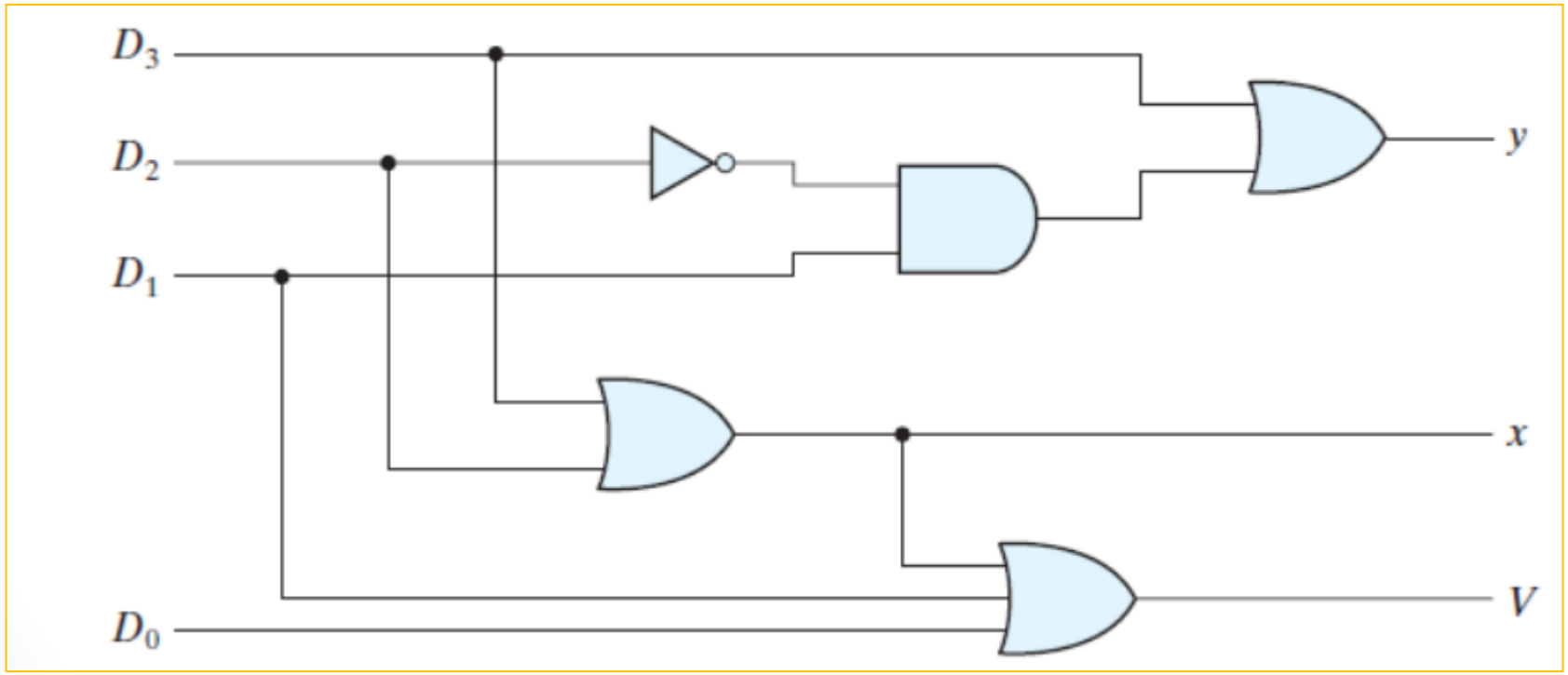
4-to-2 Priority Encoder

$$X = D_2 + D_3$$

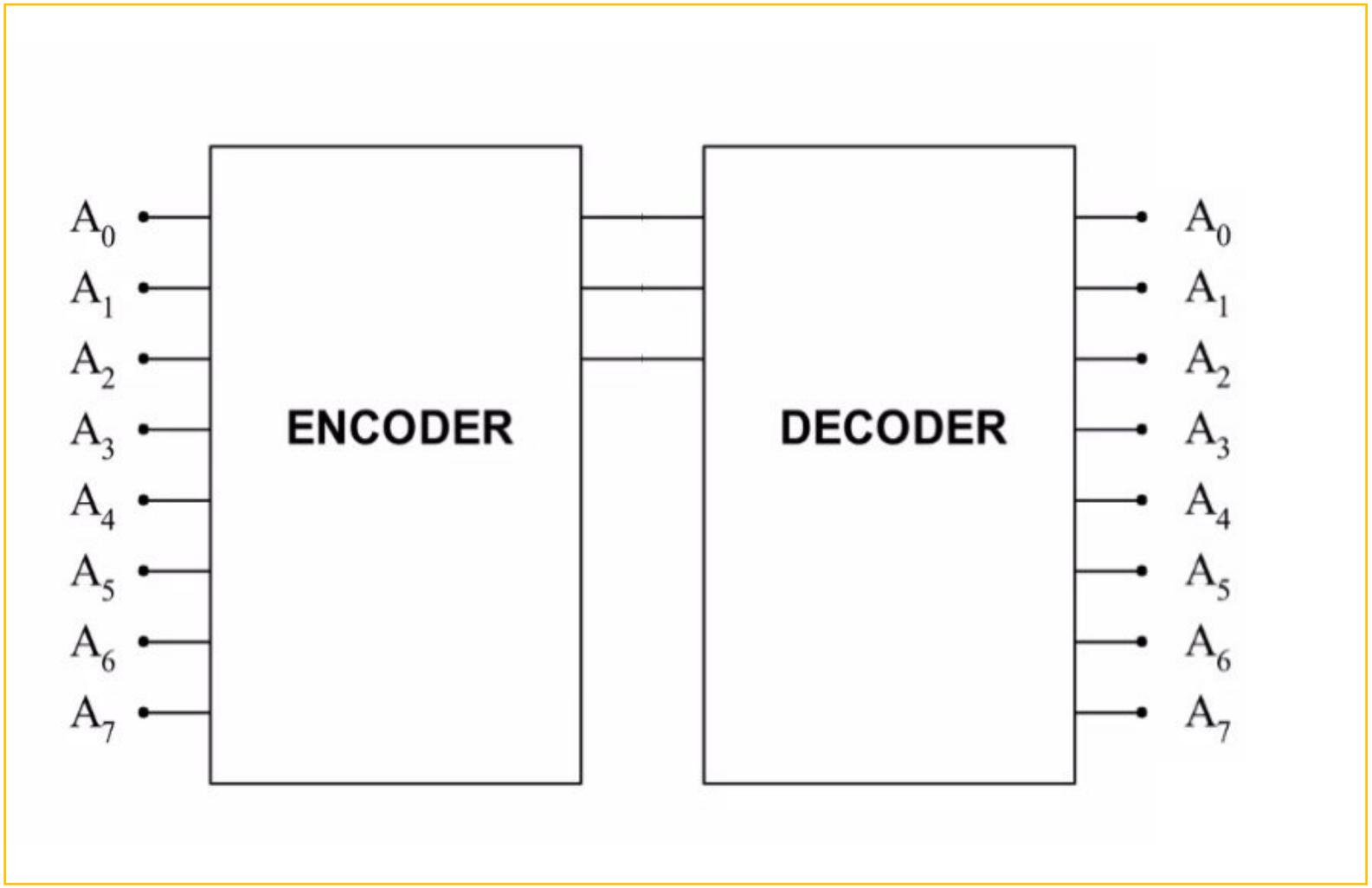
$$Y = D_3 + D_1D_2'$$

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

V is **0** **only** when **all** Inputs are 0 (Inactive)

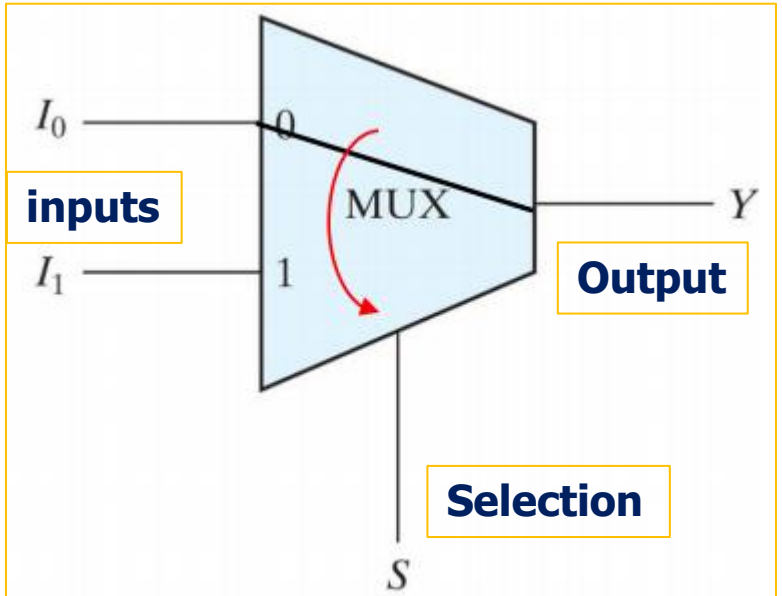


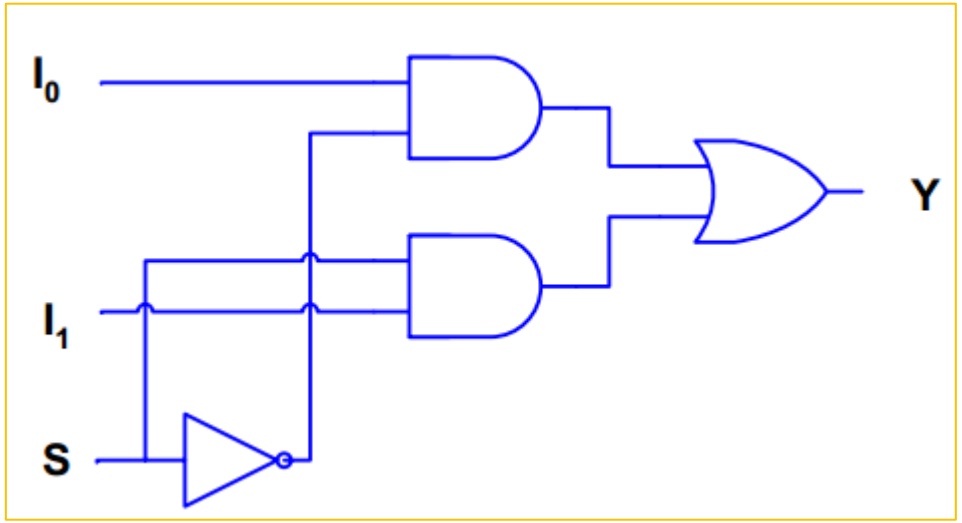
Encoder/Decoder



Multiplexer

- A **multiplexer** (MUX) is a combinational circuit that **selects** binary information from one of **many input lines** and directs it to a **single output line**.
- The selection is performed using **selection control lines**.
- Normally, there are 2^n **input** lines and **n selection** lines.
- A MUX acts as an electronic **switch** that **selects one** of several sources.

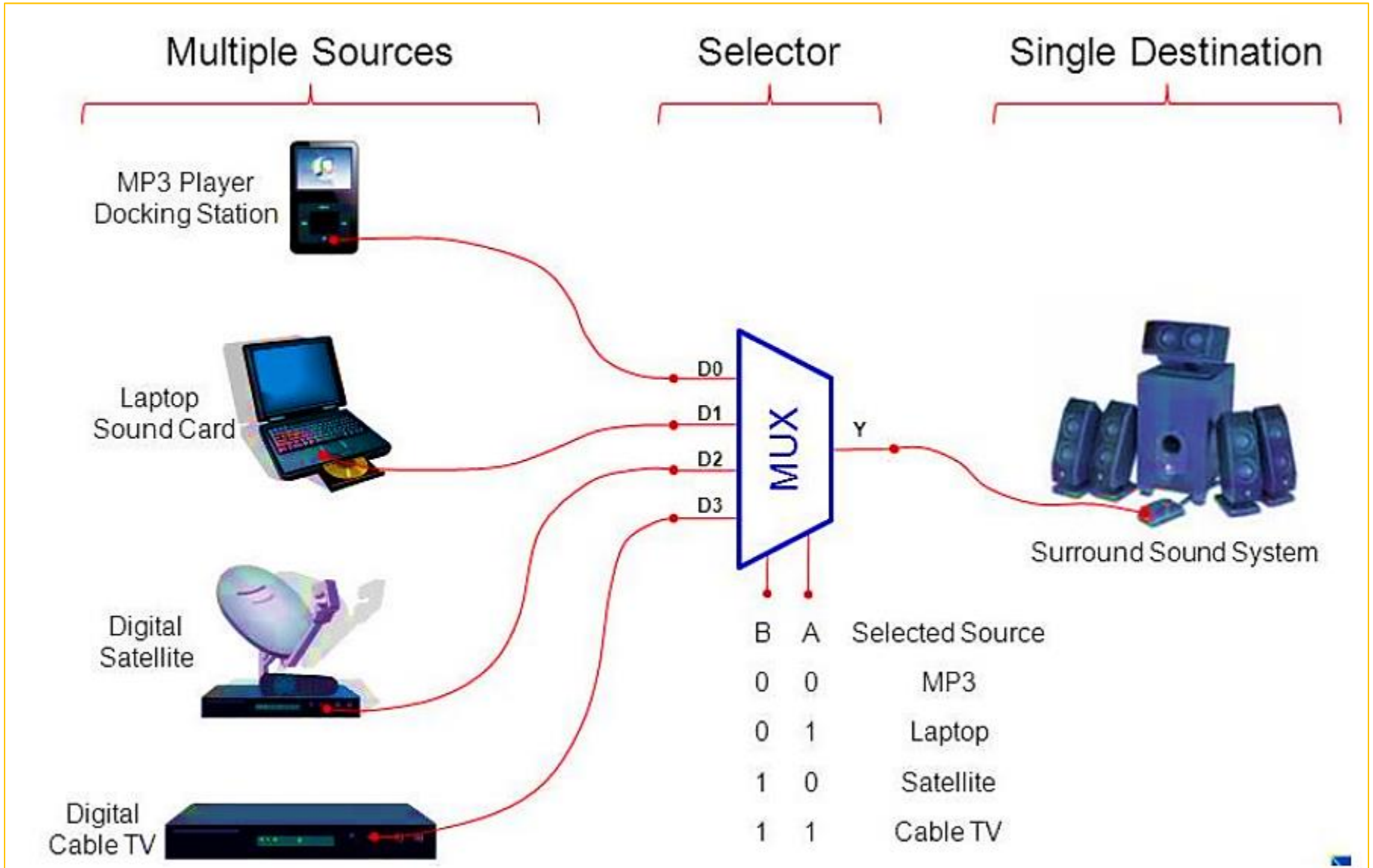




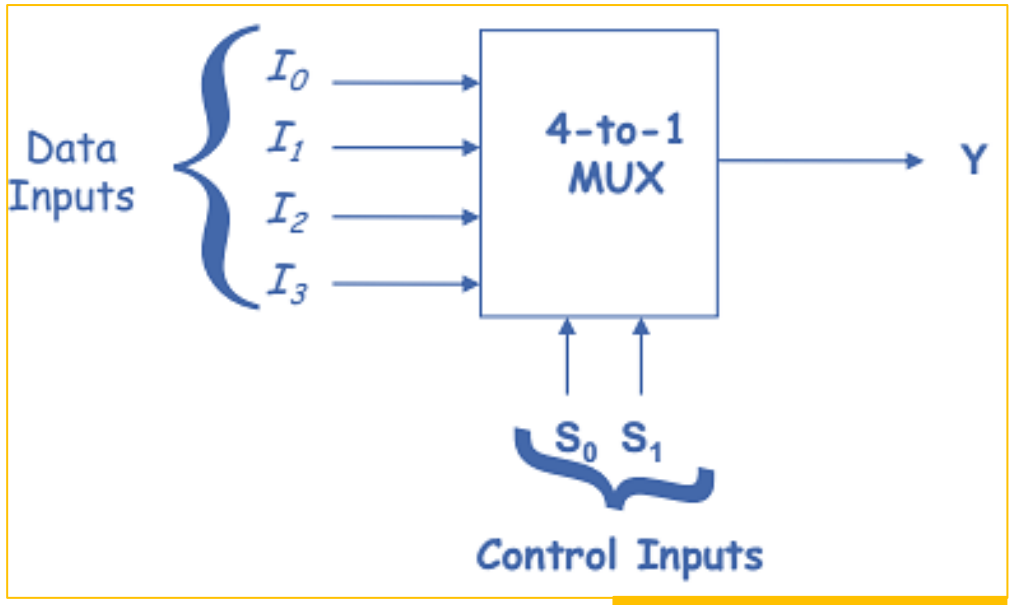
I_0 will move to the output Y when $S = 0$

I_1 will move to the output Y when $S = 1$

Applications of MUX

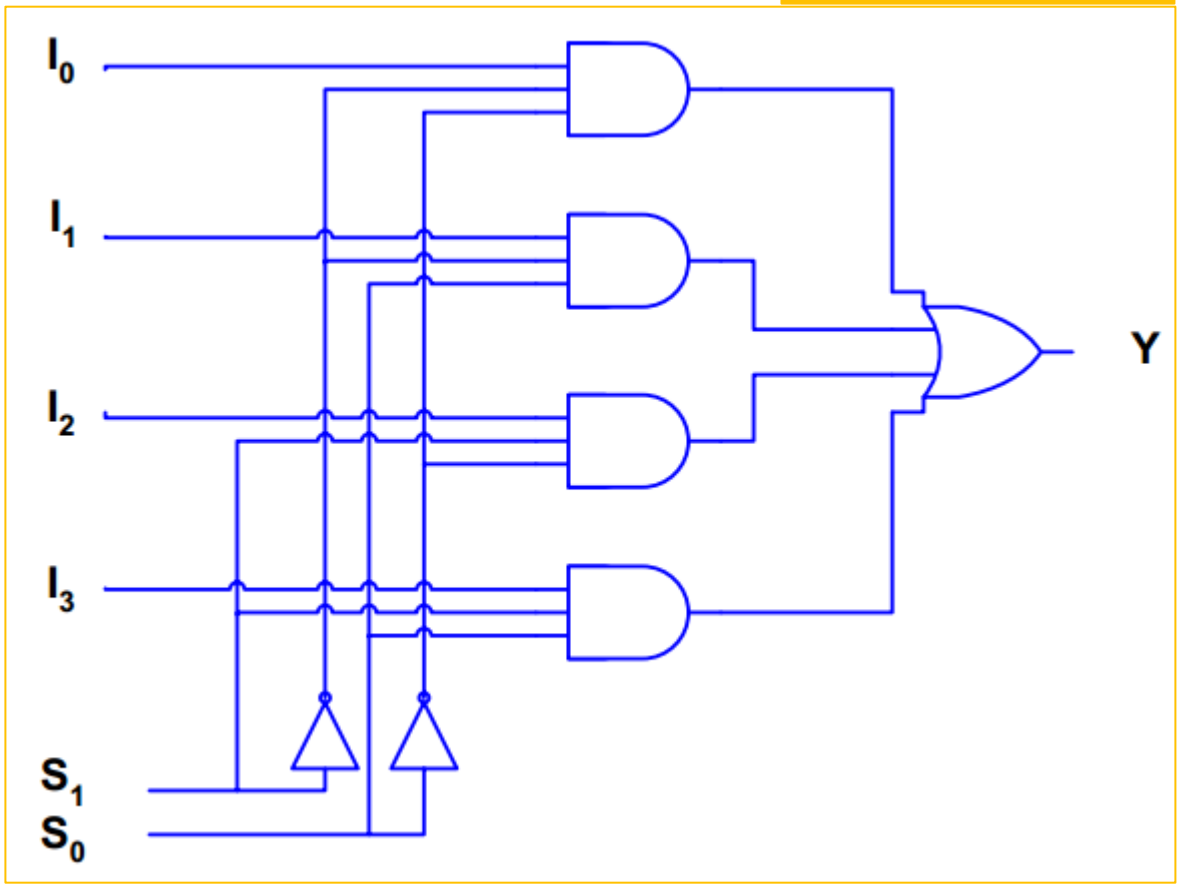


Example: **4-to-1 MUX**



Block Diagram

Logic Diagram



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

Function Table

$$Y = S_1' S_0' I_0 + S_1' S_0 I_1 + S_1 S_0' I_2 + S_1 S_0 I_3$$

Expression

Multiplexer

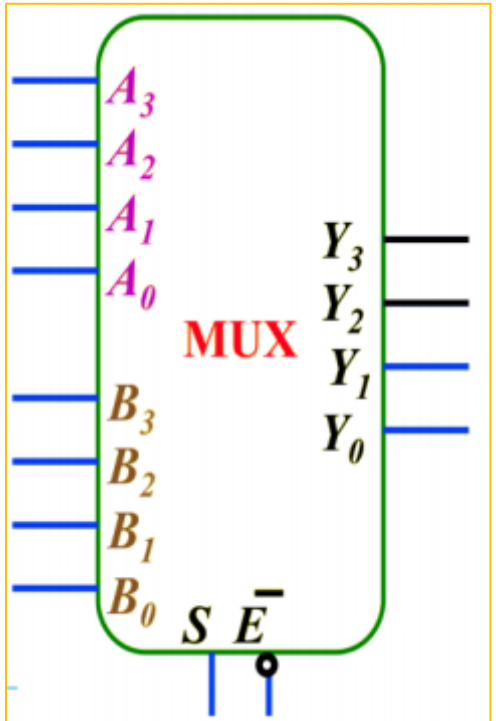
- A **multiplexer** is also called a **data selector**, since it selects one of many inputs and steers the binary information to the **single** output line
- The **AND gates and inverters** in the multiplexer resemble a **decoder** circuit, and they decode the selection input lines.
- In general, for **2^n-to-1** multiplexer
  - Data **selection** lines → **n**
  - **Input** lines → **2^n**
  - **Output** lines → **always 1**
- **2^n-to-1** multiplexer is constructed from
  - **n-to-2^n Decoder**
  - **2^n** input lines connected to the **AND** gates.
  - The outputs of the AND gates are applied to a **single OR** gate

$$Y = m_0 I_0 + m_1 I_1 + m_2 I_2 + \ldots + m_{2^n-1} I_{2^n-1}$$

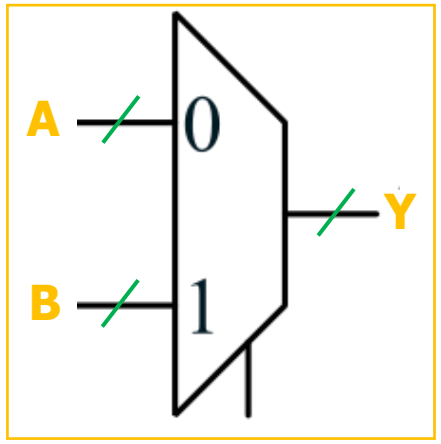
MUX Output (General)

Multi-bit Multiplexer

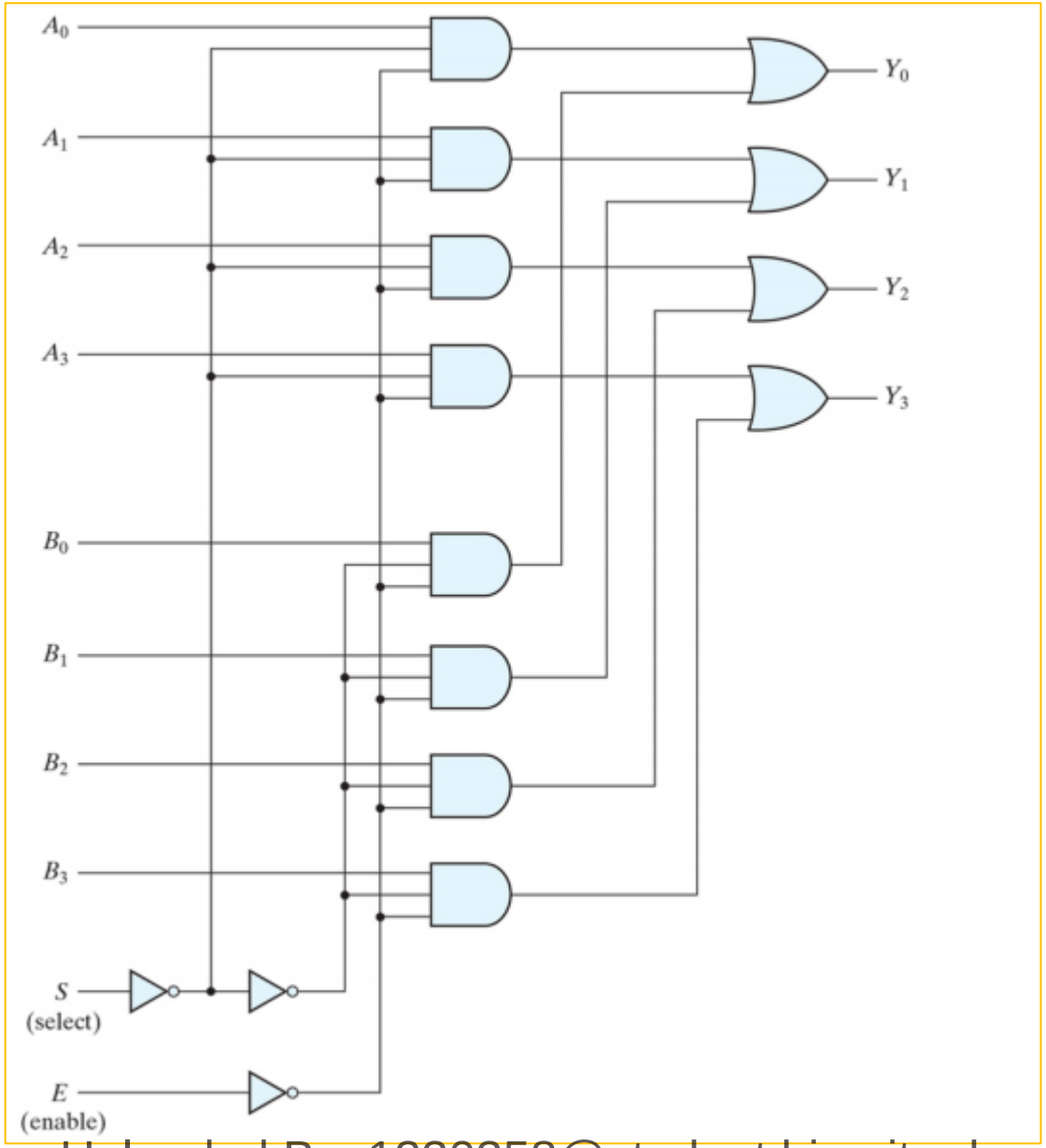
- Multiplexers may have an **enable** input, similar to decoders, to **control** the **operation** of the unit
- **M**-bit (2-to-1) multiplexer is equivalent to **M** parallel mux's **share** a common selection line
  - It is viewed as a circuit that selects **one of two M**-bit sets of data lines



E	S	Y
1	X	All 0's
0	0	Select A
0	1	Select B



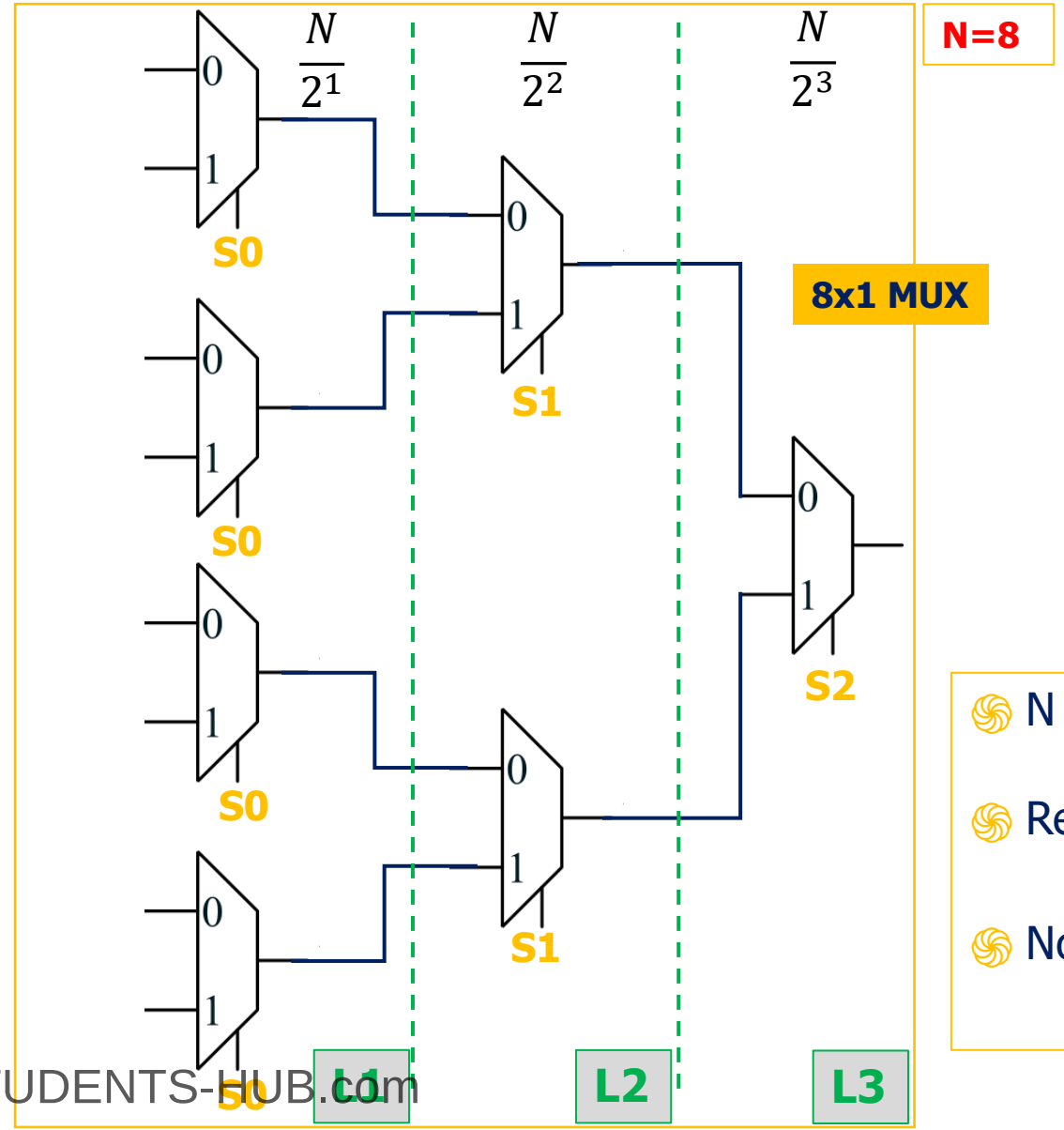
/ → **M**-bit

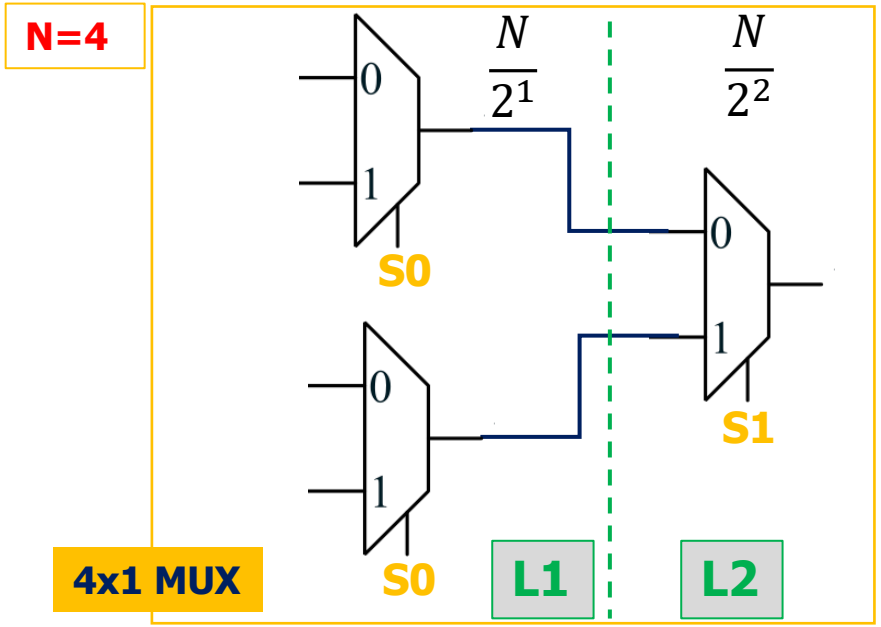


Remember: The **bubble** indicates that the mux is **enabled** when E=**0**

Multiplexers

- Muxes can be **connected** together to form a **larger** Mux circuit



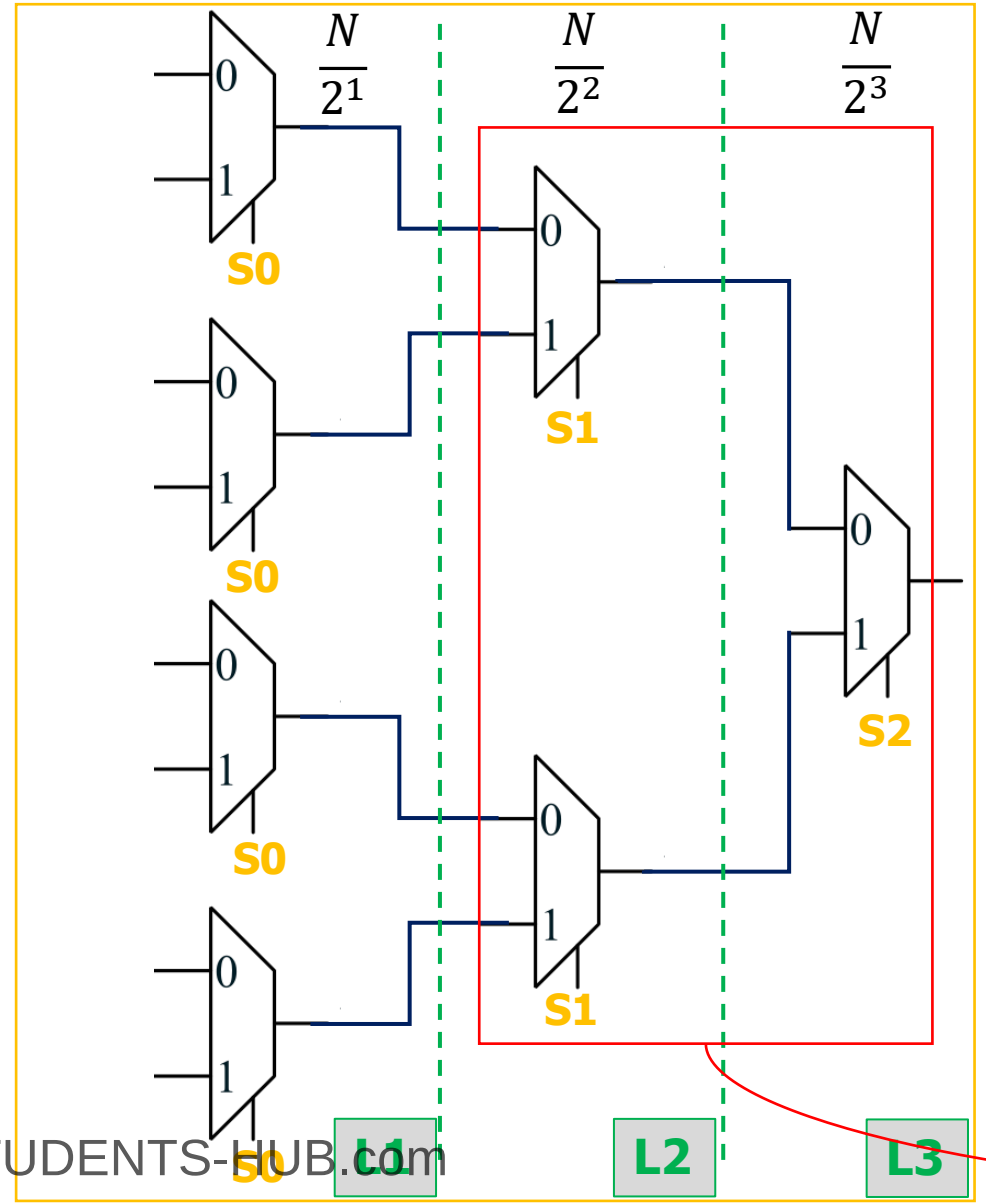


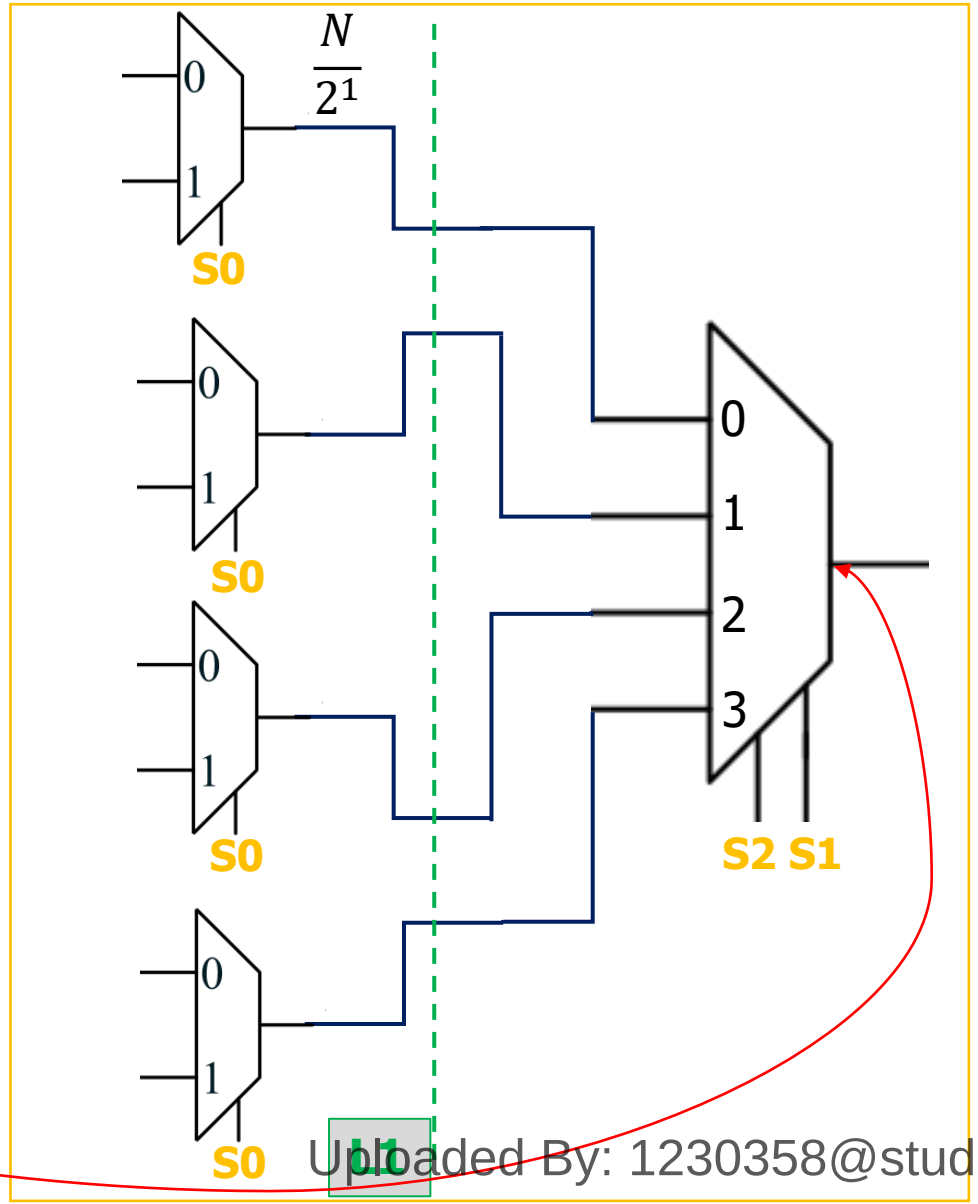
- N → No. of Inputs (2^n)
- Required Levels using 2x1 Mux = $\log_2(N)$
- No. of Muxes in level (i) = $\frac{N}{i}$, i=0,1,2…

Multiplexers

Muxes can be **connected** together to form a **larger** Mux circuit

8x1 MUX

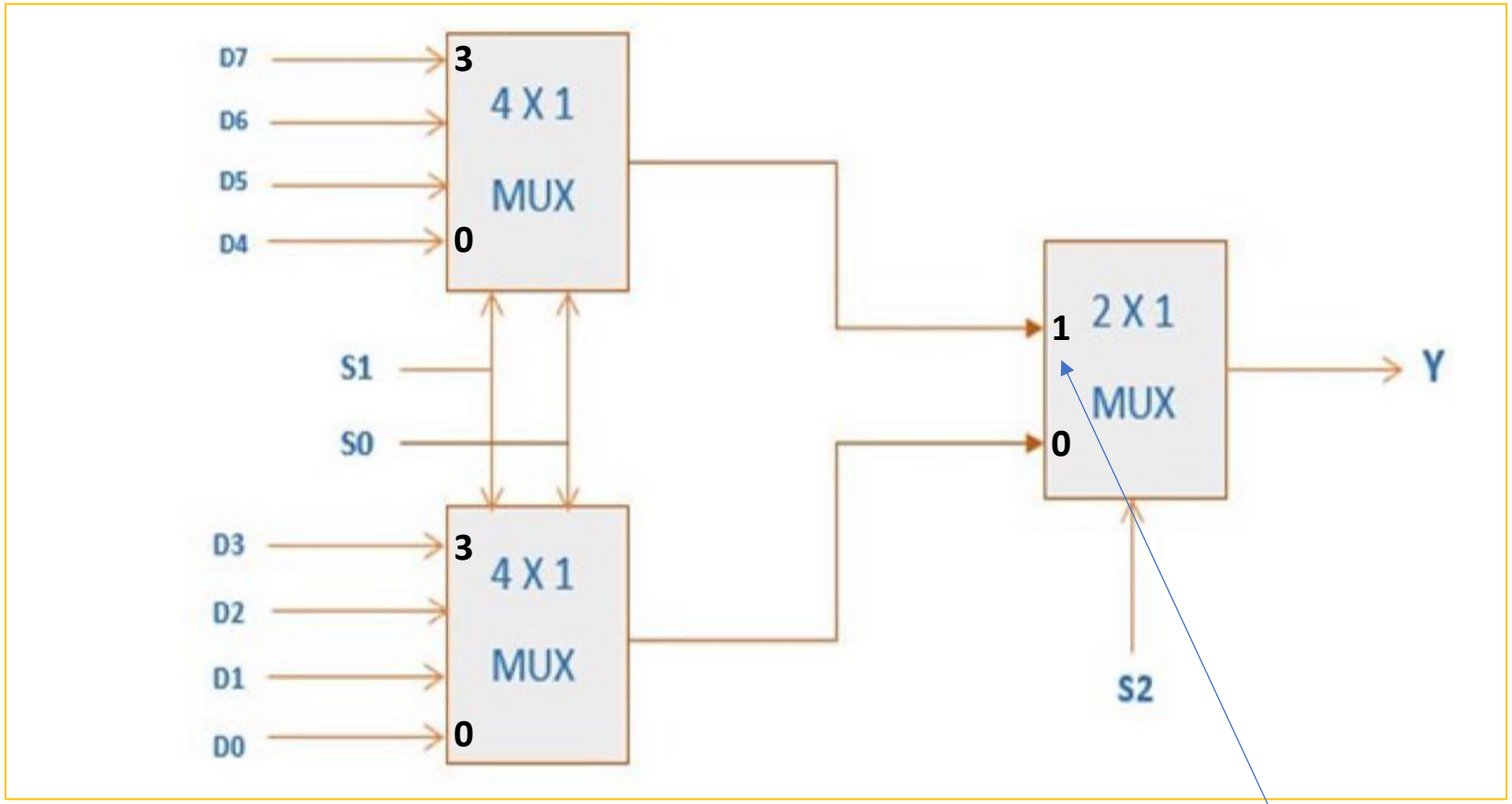




Multiplexers

Muxes can be **connected** together to form a **larger** Mux circuit

8x1 MUX

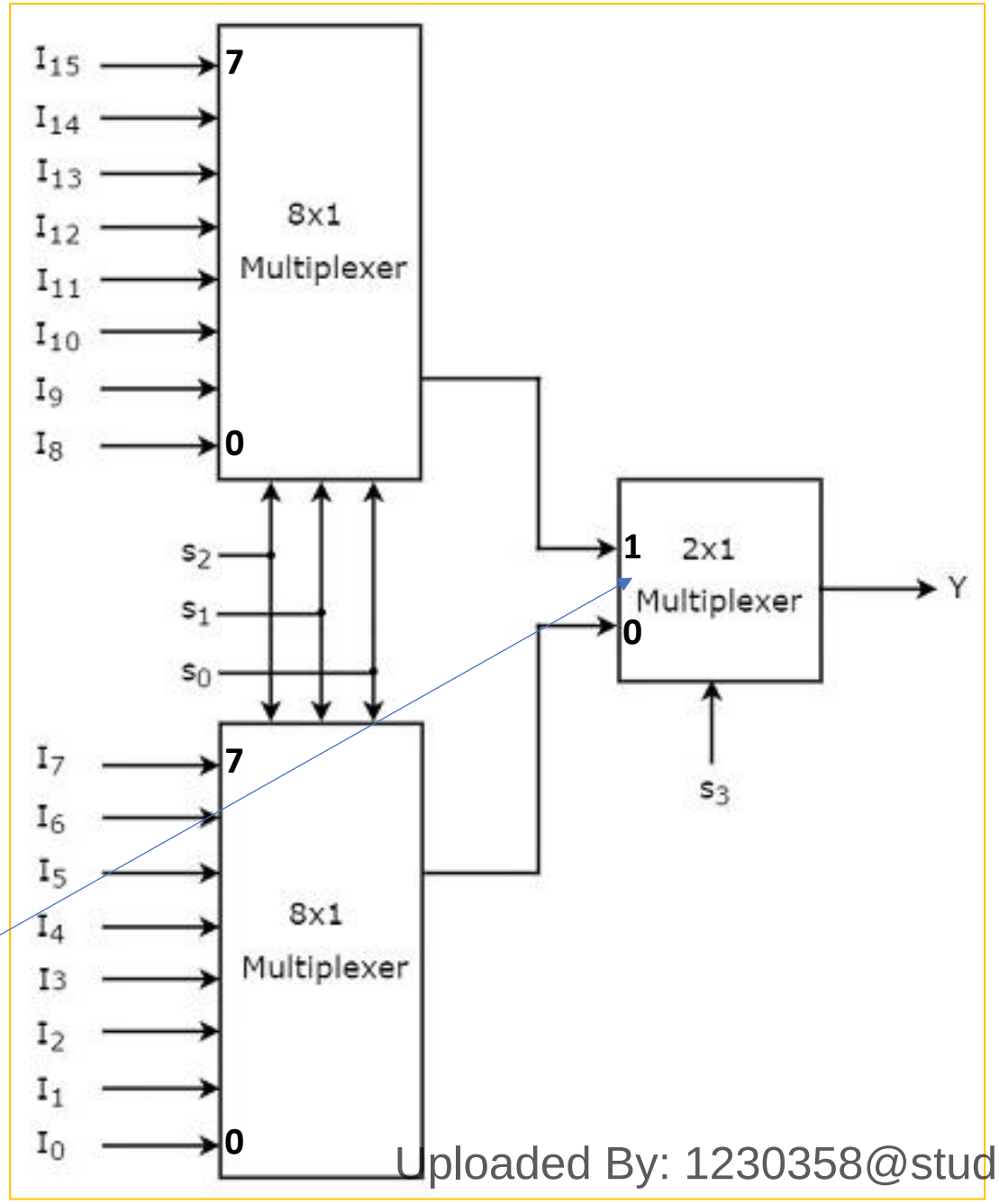


Always Follow/Consider the internal Labels To determine MSB & LSB (Connection Order)

Multiplexers

- Muxes can be **connected** together to form a **larger** Mux circuit

16x1 MUX

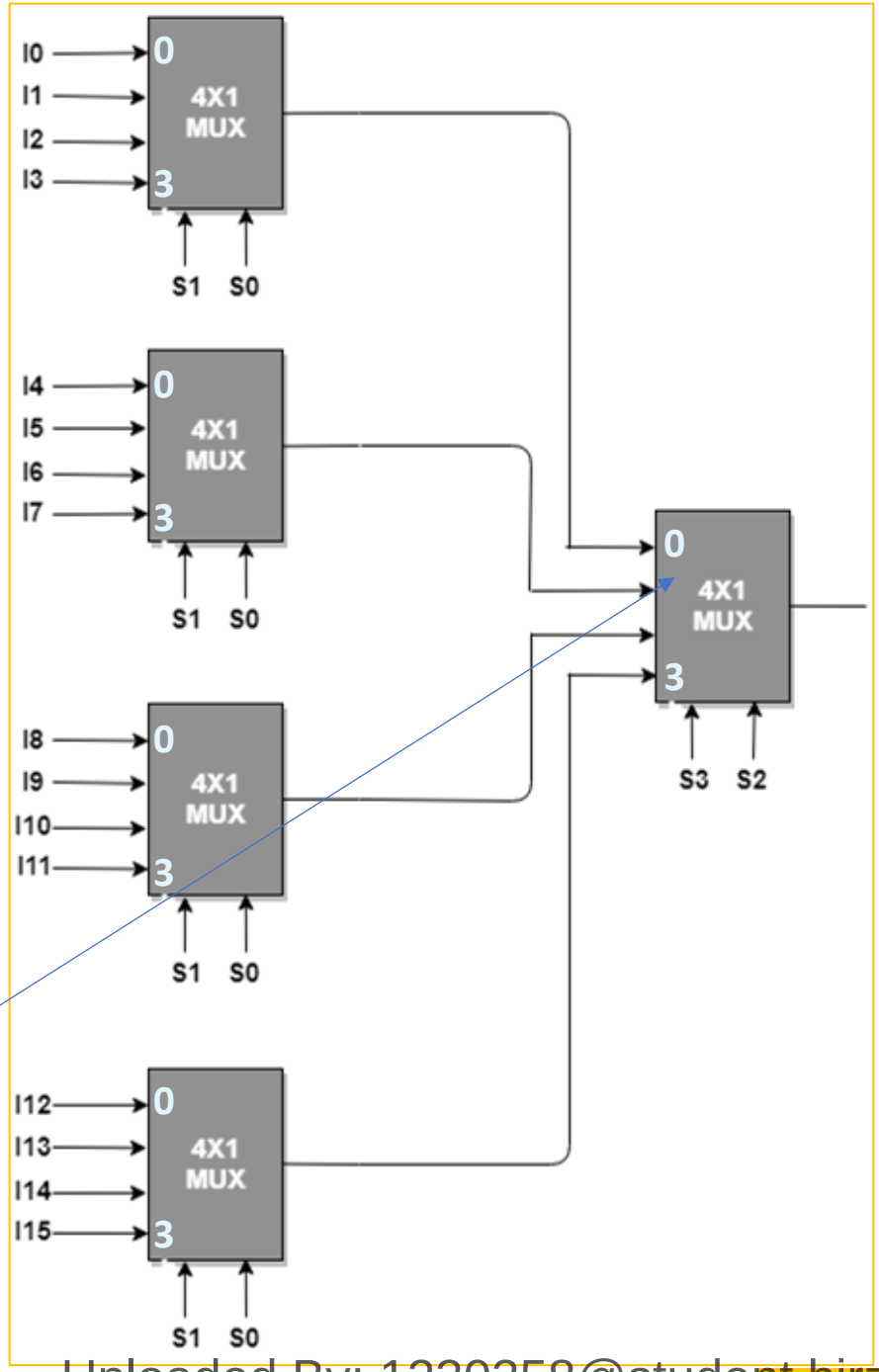


Always Follow/Consider the internal Labels To determine MSB & LSB (Connection Order)

Multiplexers

- Muxes can be **connected** together to form a **larger** Mux circuit

16x1 MUX



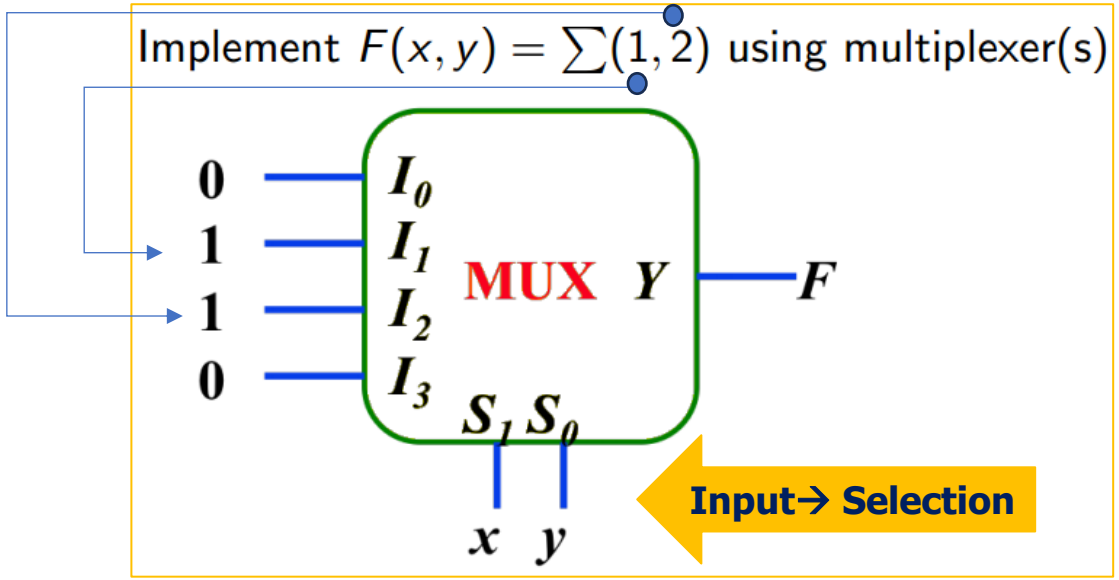
Always Follow/Consider the internal Labels To determine MSB & LSB (Connection Order)

Function Implementation using MUXs

- **RECALL:** We learned how to implement Boolean functions using **decoders**, by adding external **OR** gate
- **RECALL:** A multiplexer is a **decoder** and an **OR** gate that provides the output
  - Multiplexer inputs are the **minterms**

We can implement any **n** variable Boolean function using a **MUX** with **n** select lines (**2^n** input lines)

- The **n** variables are connected to the **n selection** lines.
- Each input of the multiplexer is **set to 0 or 1**, depending on which minterm of the function is present.

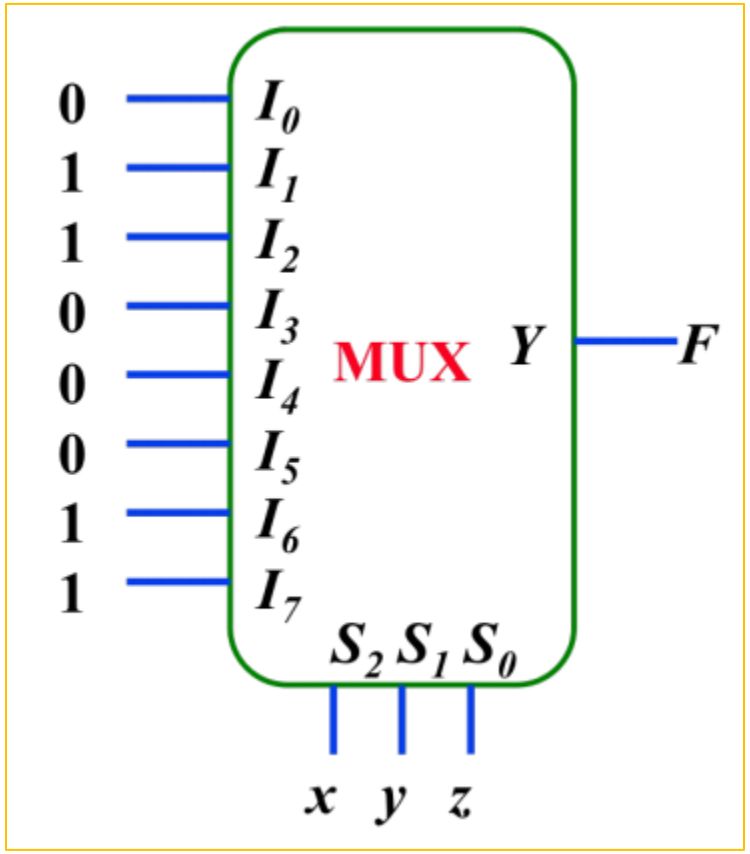


Function Implementation using MUXs

Example: Implement F(**x,y,z**) = Σ(1,2,6,7) using **8-to-1** multiplexer.

n Variables → (2^n x 1)Mux

Solution: Connect the variables x, y, z to the selection inputs S_2, S_1, and S_0. Then set $I_0 = I_3 = I_4 = I_5 = 0$ and $I_1 = I_2 = I_6 = I_7 = 1$.



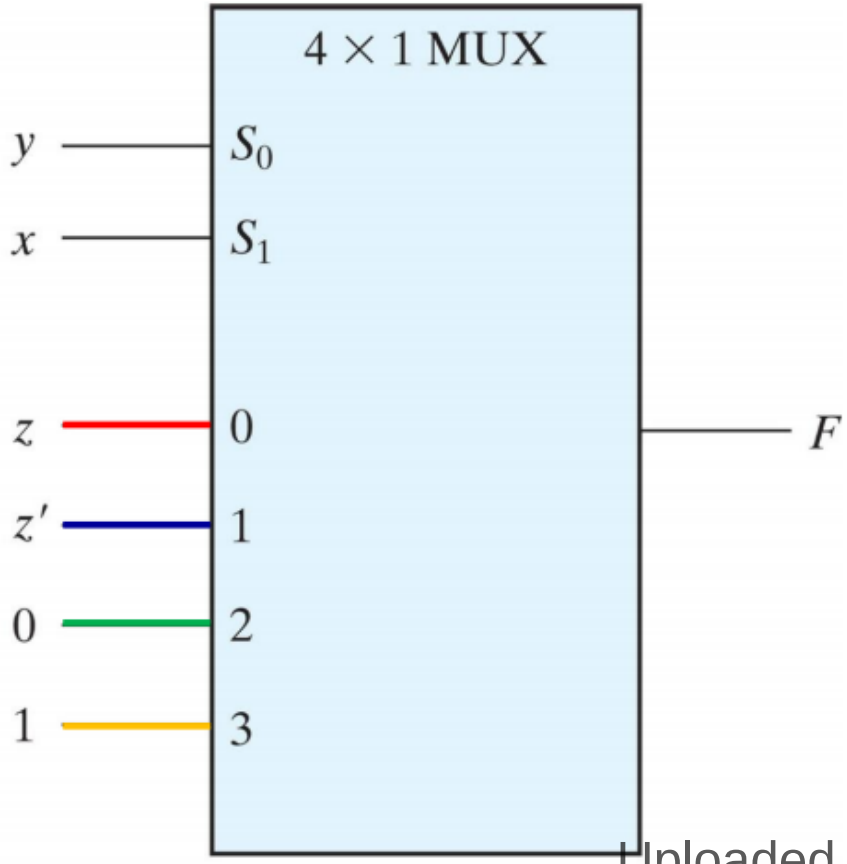
Function Implementation using MUXs

- We can **efficiently** implement any **n** variable Boolean function using a MUX with **(n-1) select** lines (**2^{n-1}** input lines)
  1) Connect the first **(n-1)** variables to the **select** lines
  2) The **remaining single** variable of the function is used for the **data inputs (x, x', 1, 0)**

Example: Implement F(**x,y,z**) = Σ(1,2,6,7) using **4-to-1** multiplexer.

n Variables → (2^{n-1} x 1) Mux

x	y	z	F	
0	0	0	0	$F = z$
0	0	1	1	
0	1	0	1	$F = z'$
0	1	1	0	
1	0	0	0	$F = 0$
1	0	1	0	
1	1	0	1	$F = 1$
1	1	1	1	



Function Implementation using MUXs

Alternative Method : **Implementation Table** [Simpler for 2^{n-1} x 1 Mux]

A. List the **input** of the multiplexer (**z**)

B. List under it all the **minterms** in (**2**) Rows and (**4**) Columns

C. The **first half** of the minterms is associated with the **Primed Variable (z')** and the **second half** with the **Normal Variable (z)**

D. The given function is implemented by circling the minterms of the function and applying the following **rules** to find the values for the inputs of the multiplexer

1) If **both** the minterms in the column are **not** circled, apply **0** to the corresponding input
2) If **both** the minterms in the column are circled, apply **1** to the corresponding input
3) If **the bottom** minterm is circled and the **top is not** circled, apply **z** to the input
4) **If the top** minterm is circled and the **bottom is not** circled, apply **z'** to the input

No need for **TT**
Could be derived
directly from **minterms**

Z'	0	(2)	4	(6)
Z	(1)	3	5	(7)
	Z	Z'	0	1
	I_0	I_1	I_2	I_3

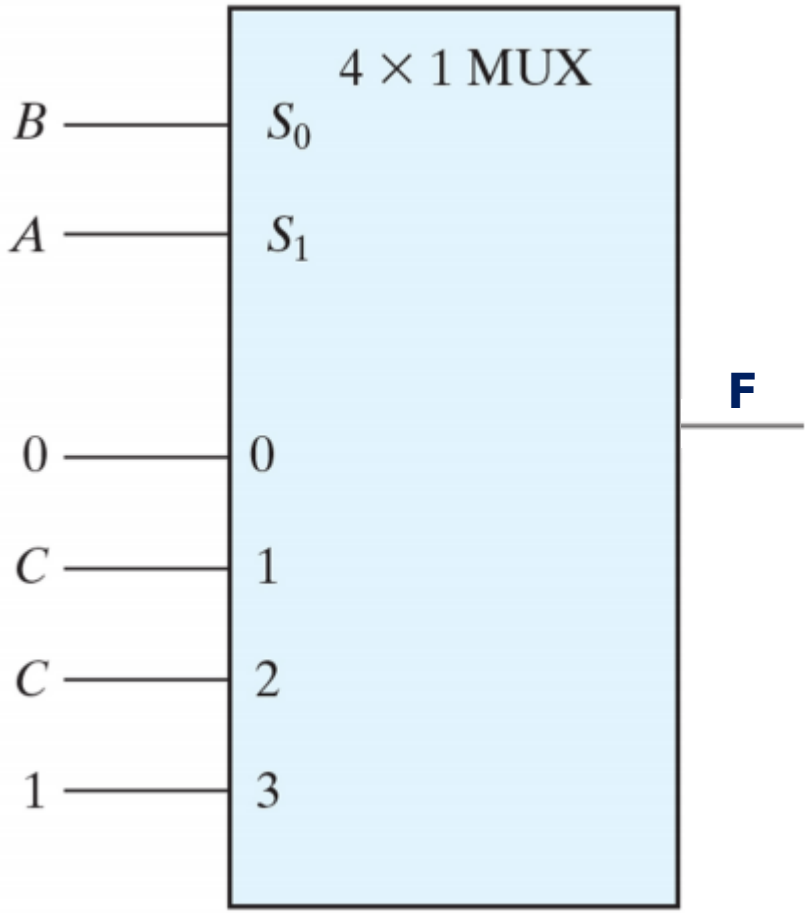
Don't Get Confused
It has **NO** relation to K-Map

Function Implementation using MUXs

Extra Example: Implement F(**A,B,C**) = Σ(3,5,6,7) using **4-to-1** multiplexer.

n Variables → (2^{n-1} x 1) Mux

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



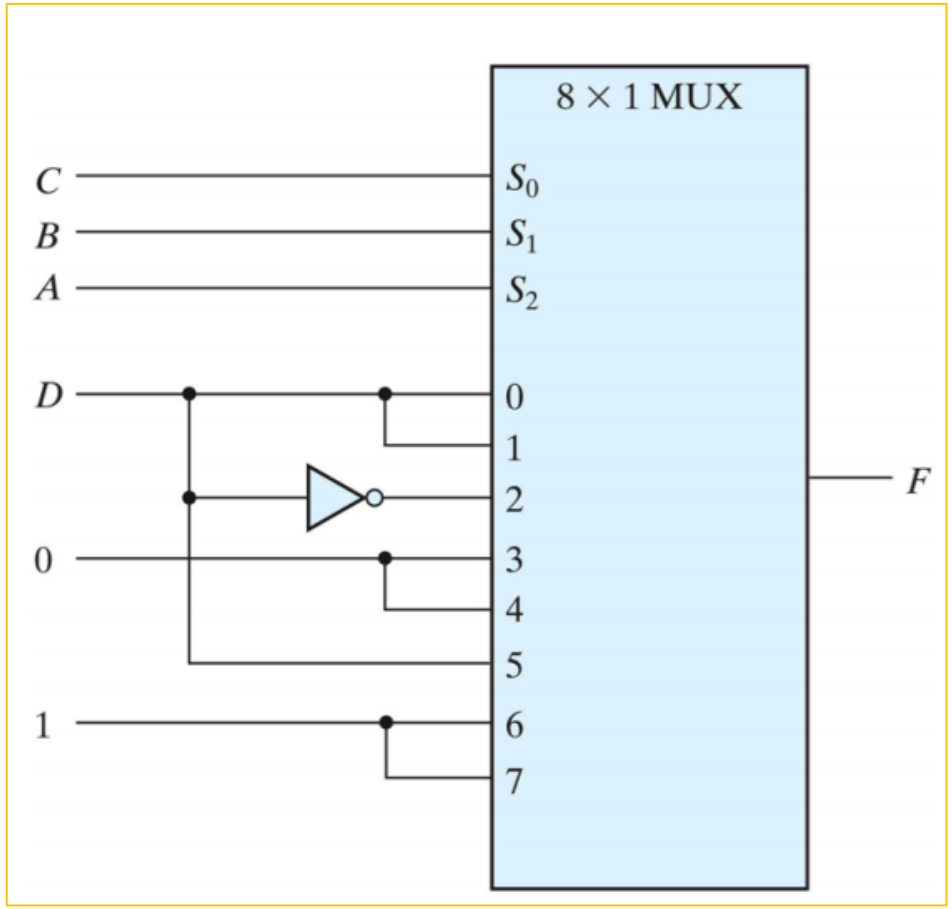
C′	0	2	4	6
C	1	3	5	7
	0	C	C	1
	I_0	I_1	I_2	I_3

Function Implementation using MUXs

n Variables → (2^{n-1} x 1) Mux

Extra Example: Implement F (A, B , C , D) = Σ(1, 3, 4, 11, 12, 13, 14, 15) using **8-to-1** multiplexer

ABC	A	B	C	D	F	
0	0	0	0	0	0	$F = D$
	0	0	0	1	1	
1	0	0	1	0	0	$F = D$
	0	0	1	1	1	
2	0	1	0	0	1	$F = D'$
	0	1	0	1	0	
3	0	1	1	0	0	$F = 0$
	0	1	1	1	0	
4	1	0	0	0	0	$F = 0$
	1	0	0	1	0	
5	1	0	1	0	0	$F = D$
	1	0	1	1	1	
6	1	1	0	0	1	$F = 1$
	1	1	0	1	1	
7	1	1	1	0	1	$F = 1$
	1	1	1	1	1	



D′	0	2	4	6	8	10	12	14
D	1	3	5	7	9	11	13	15
	D	D	D′	0	0	D	1	1
	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7

Function Implementation using MUXs

n Variables → (2^{n-1} x 1) Mux

Extra Example:

Implement F (A, B , C , D) = Σ(3, 5, 10, 11, 12, 15) + Σ(4, 8, 14) using **8-to-1** multiplexer (**Use A,C,D as selection lines**)

A	B	C	D	F	
0	0	0	0	0	I_0
0	0	0	1	0	I_1
0	0	1	0	0	I_2
0	0	1	1	1	I_3
0	1	0	0	X	I_0
0	1	0	1	1	I_1
0	1	1	0	0	I_2
0	1	1	1	0	I_3
1	0	0	0	X	I_4
1	0	0	1	0	I_5
1	0	1	0	1	I_6
1	0	1	1	1	I_7
1	1	0	0	1	I_4
1	1	0	1	0	I_5
1	1	1	0	X	I_6
1	1	1	1	1	I_7

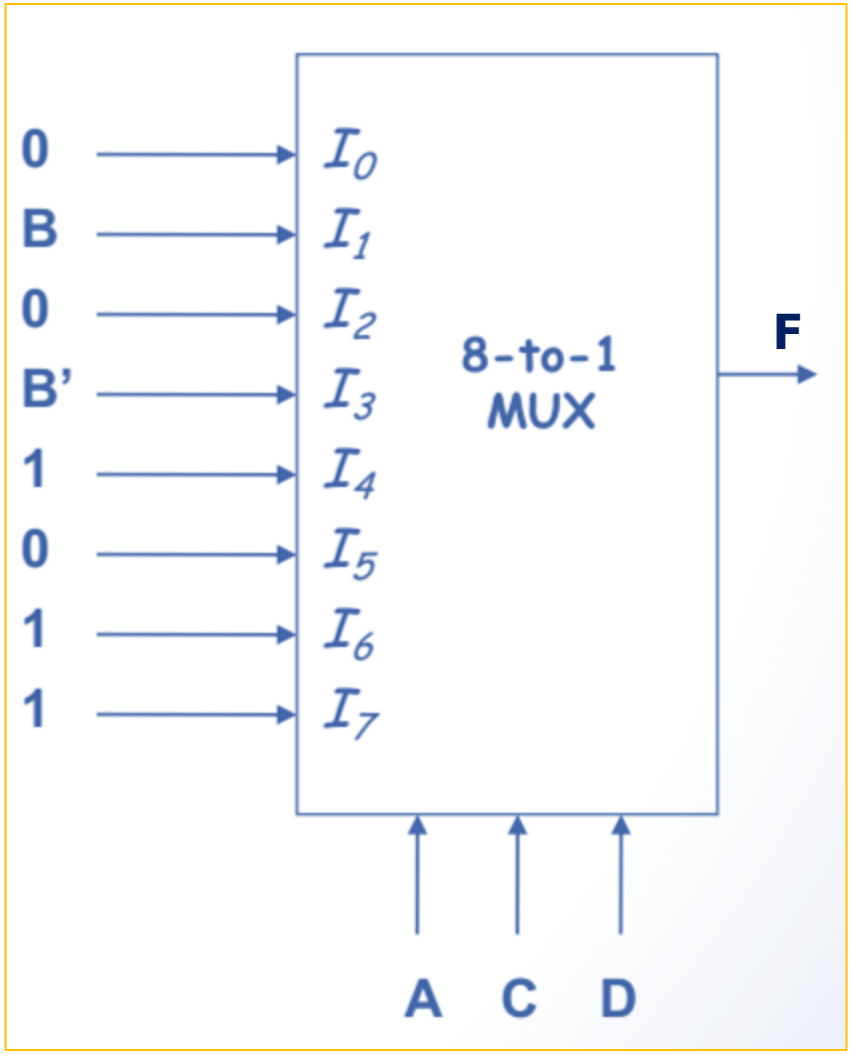
Both Values **Similar** → **Constant** (0,1)

Different → Check **B**

Be Carful about the Order

B'	0	1	2	3	8	9	10	11
B	4	5	6	7	12	13	14	15
	0	B	0	B'	1	0	1	1
	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7

ONLY Consider **Don't Care**, when **other** circles exist in the **same** Column



Function Implementation using MUXs

n Variables → (2^{n-2} x 1) Mux

Extra Example:

Implement F (A, B , C , D) = Σ(3, 5, 10, 11, 12, 15) + Σ(4, 8, 14) using **4-to-1** multiplexer (**Use C,D as selection lines**)

A	B	C	D	F	
0	0	0	0	0	I_0
0	0	0	1	0	I_1
0	0	1	0	0	I_2
0	0	1	1	1	I_3
0	1	0	0	X	I_0
0	1	0	1	1	I_1
0	1	1	0	0	I_2
0	1	1	1	0	I_3
1	0	0	0	X	I_0
1	0	0	1	0	I_1
1	0	1	0	1	I_2
1	0	1	1	1	I_3
1	1	0	0	1	I_0
1	1	0	1	0	I_1
1	1	1	0	X	I_2
1	1	1	1	1	I_3

Both Values **Similar** → **Constant** (0,1)

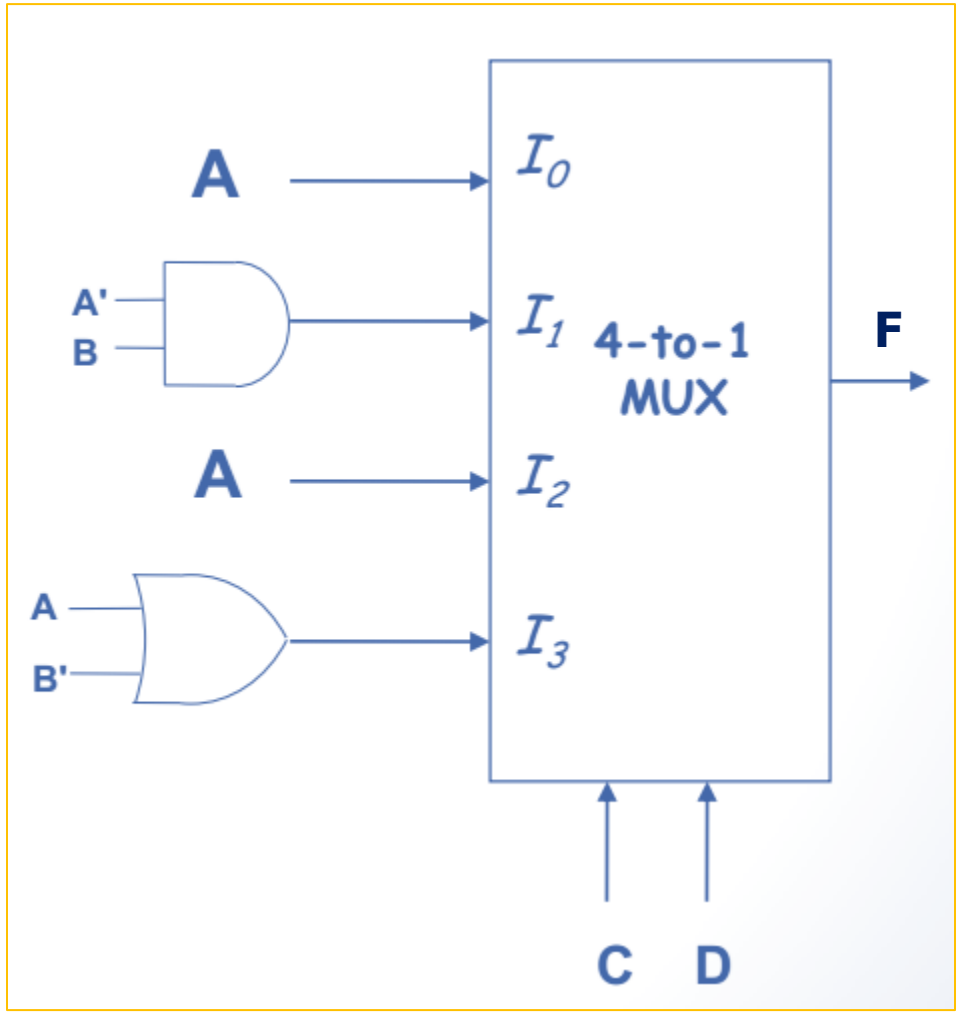
Different → Check **A,B**

A	B	C	D	F	
0	0	0	1	0	I_1
0	1	0	1	1	I_1
1	0	0	1	0	I_1
1	1	0	1	0	I_1

A'B

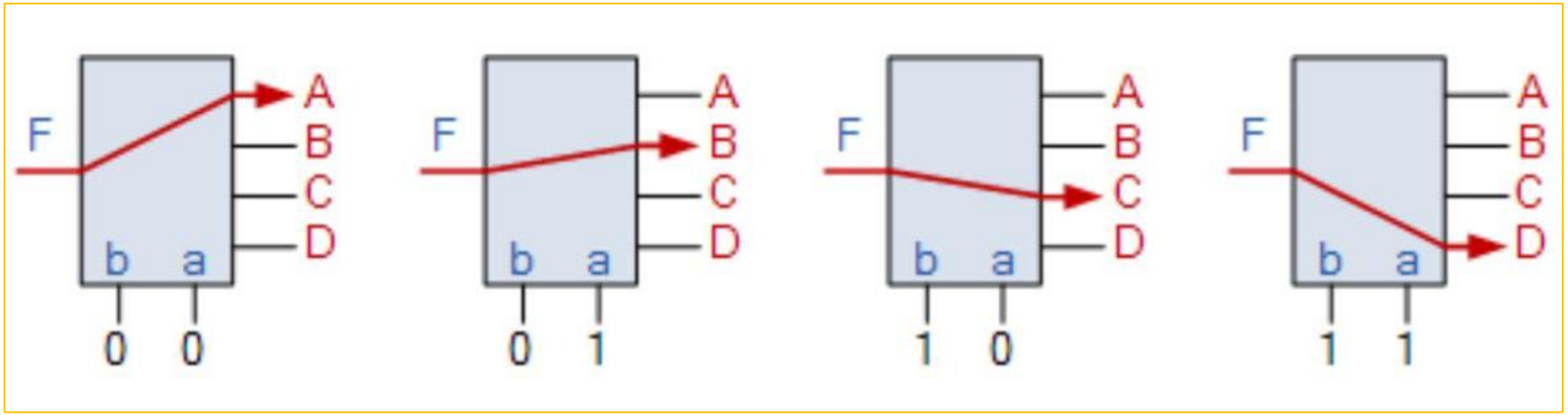
A	B	C	D	F	
0	0	1	1	1	I_3
0	1	1	1	0	I_3
1	0	1	1	1	I_3
1	1	1	1	1	I_3

A+B'

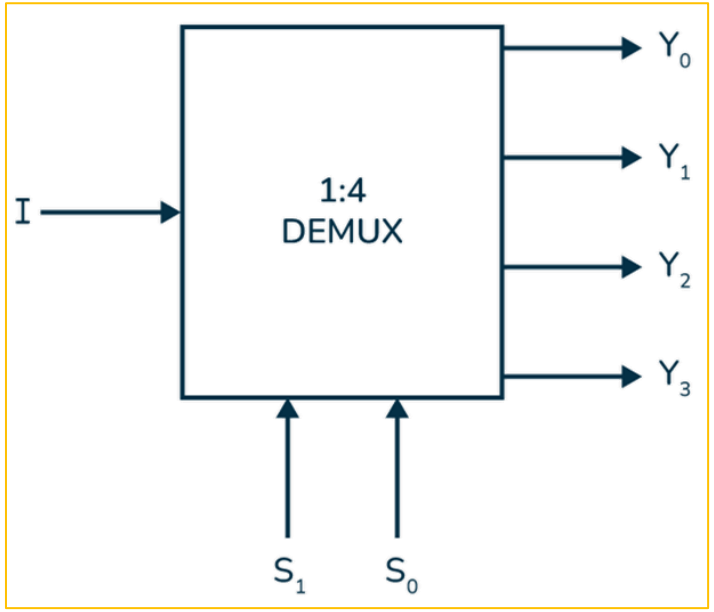


Demultiplexer

- **Recall**: A decoder with **enable** input can function as a **demultiplexer**
- **Demultiplexer (Demux)**: It is a circuit that receives information from a **single line** and **directs** it to **ONE** of **2^n output** lines.
- The **selection** of a **specific output** is controlled by the **bit** combination of **n selection** lines.
- A **demultiplexer** of **2^n outputs** has **n selection** lines, which are used to **select** which output line to **send** the **input**.
- A **demultiplexer** is also called a **data distributor**.



Demultiplexer



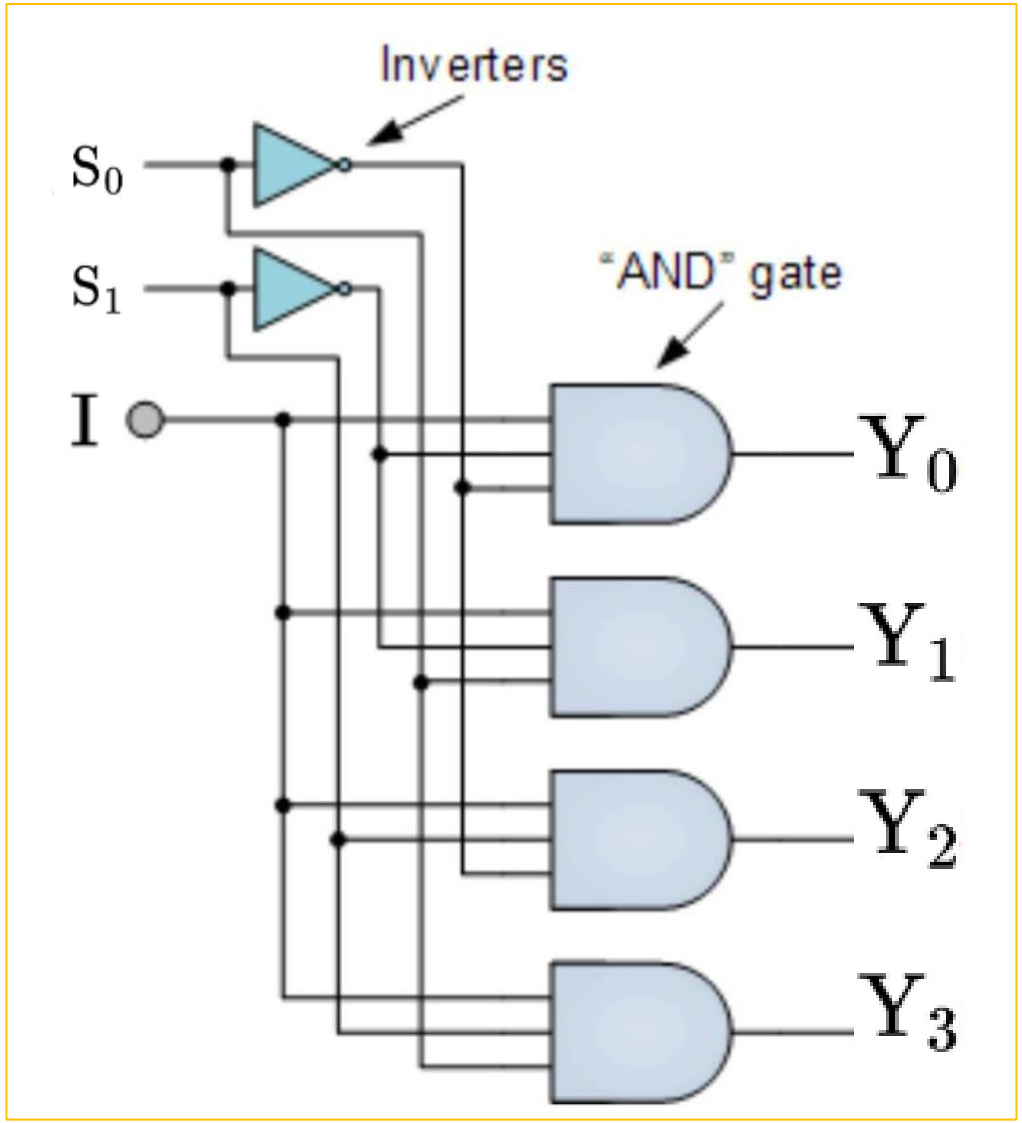
$$Y_0 = \bar{S_1}\,\bar{S_0}\,I$$

$$Y_1 = \bar{S_1}\,S_0\,I$$

$$Y_2 = S_1\,\bar{S_0}\,I$$

$$Y_3 = S_1\,S_0\,I$$

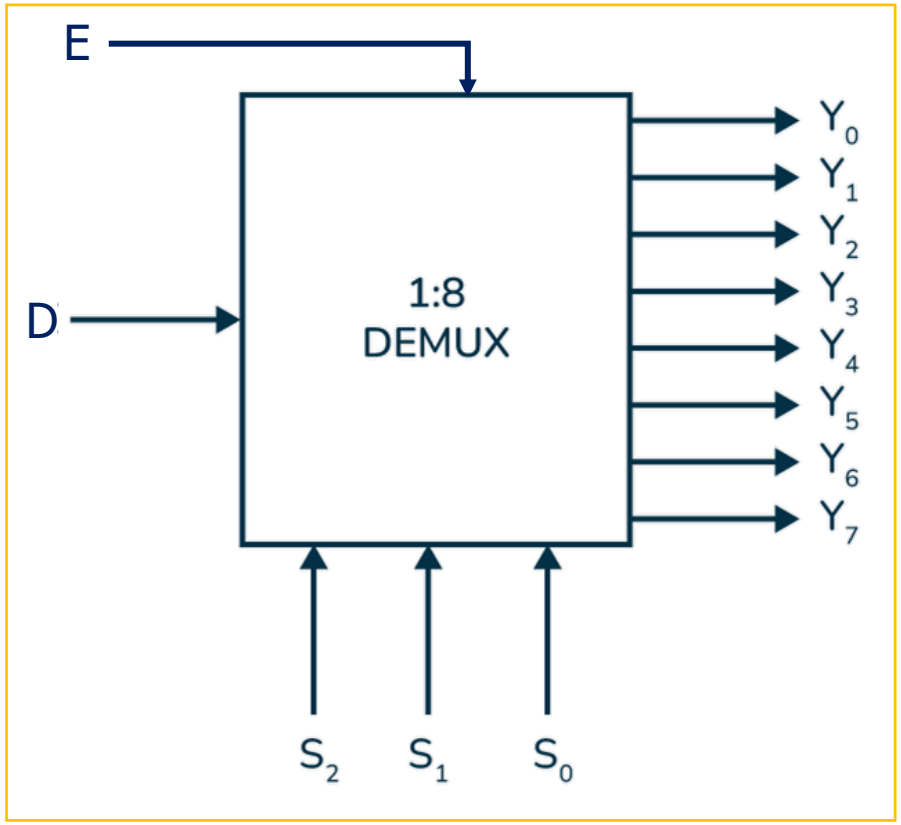
Select Line		Outputs			
S_1	S_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	I
0	1	0	0	I	0
1	0	0	I	0	0
1	1	I	0	0	0



1:8 DeMux with Enable

A 1:8 DEMUX takes a single data **input D**, enables it through an **Enable** signal **E**, and sends the data to one of the eight **outputs** Y_0–Y_7 based on the 3-bit **selection** lines S_0, S_1, S_2

8 Outputs → **3** Selection Lines

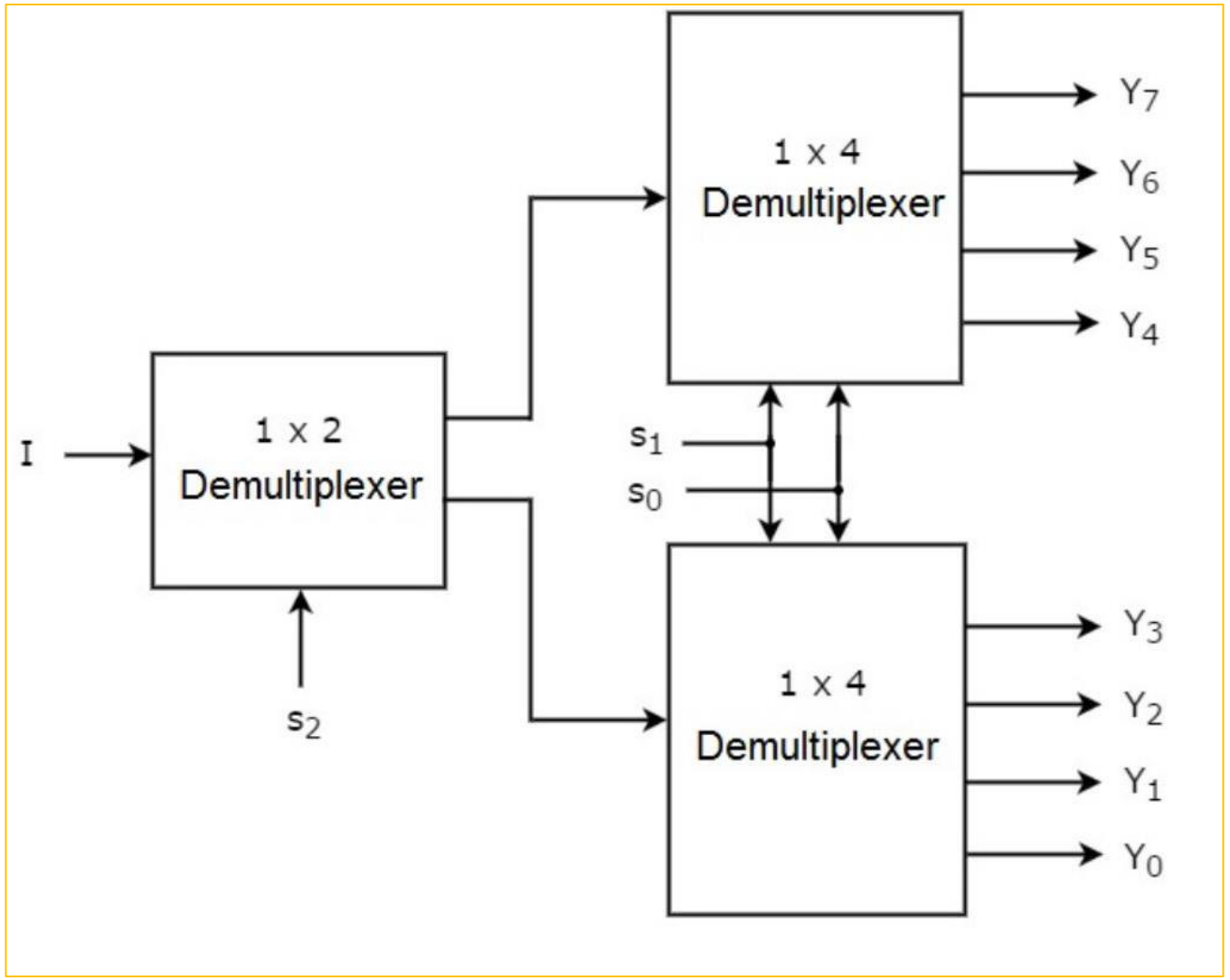


E	S_2	S_1	S_0	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7
0	X	X	X	0	0	0	0	0	0	0	0
1	0	0	0	**D**	0	0	0	0	0	0	0
1	0	0	1	0	**D**	0	0	0	0	0	0
1	0	1	0	0	0	**D**	0	0	0	0	0
1	0	1	1	0	0	0	**D**	0	0	0	0
1	1	0	0	0	0	0	0	**D**	0	0	0
1	1	0	1	0	0	0	0	0	**D**	0	0
1	1	1	0	0	0	0	0	0	0	**D**	0
1	1	1	1	0	0	0	0	0	0	0	**D**

E = 0: All outputs are **0**, regardless of selection lines.
E = 1: Data **D** is routed to the output **selected** by $S_2S_1S_0$

Demultiplexer

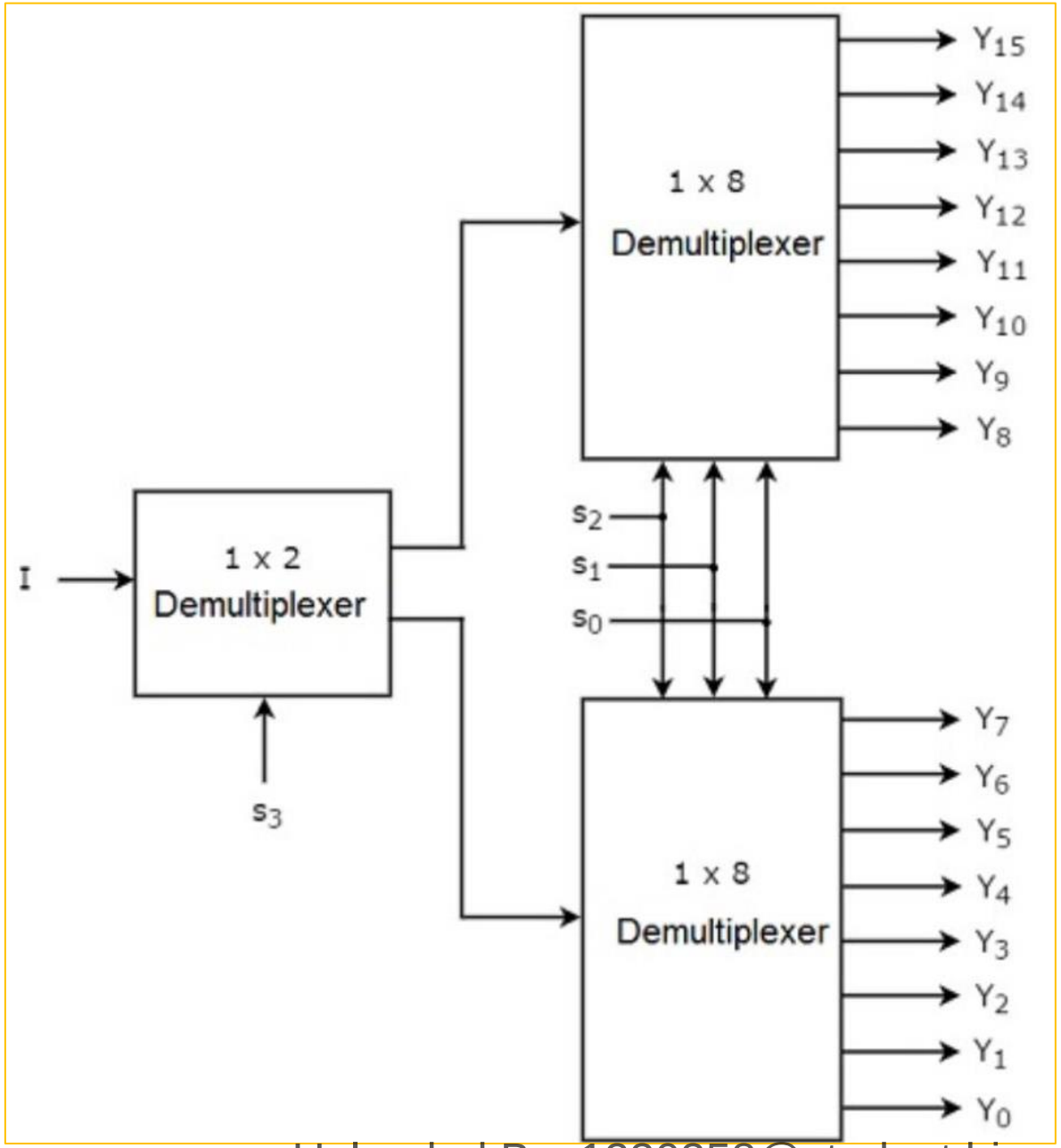
DeMuxes can be **connected** together to form a **larger** DeMux circuit



1x8 DEMUX

Demultiplexer

- DeMuxes can be **connected** together to form a **larger** DeMux circuit

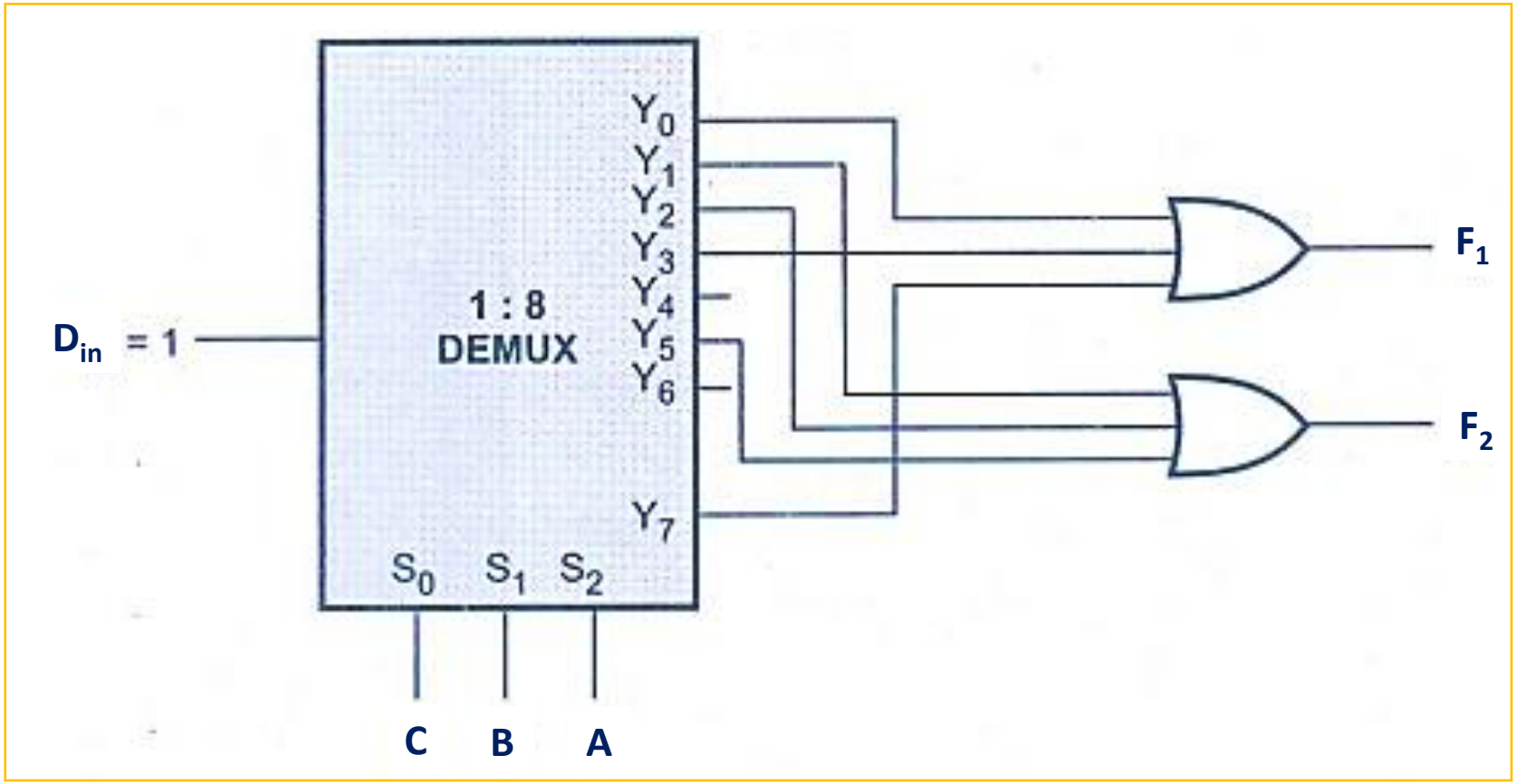


1x16 DEMUX

Function Implementation using DeMux

n Variables → (1-to-2^n) DeMux

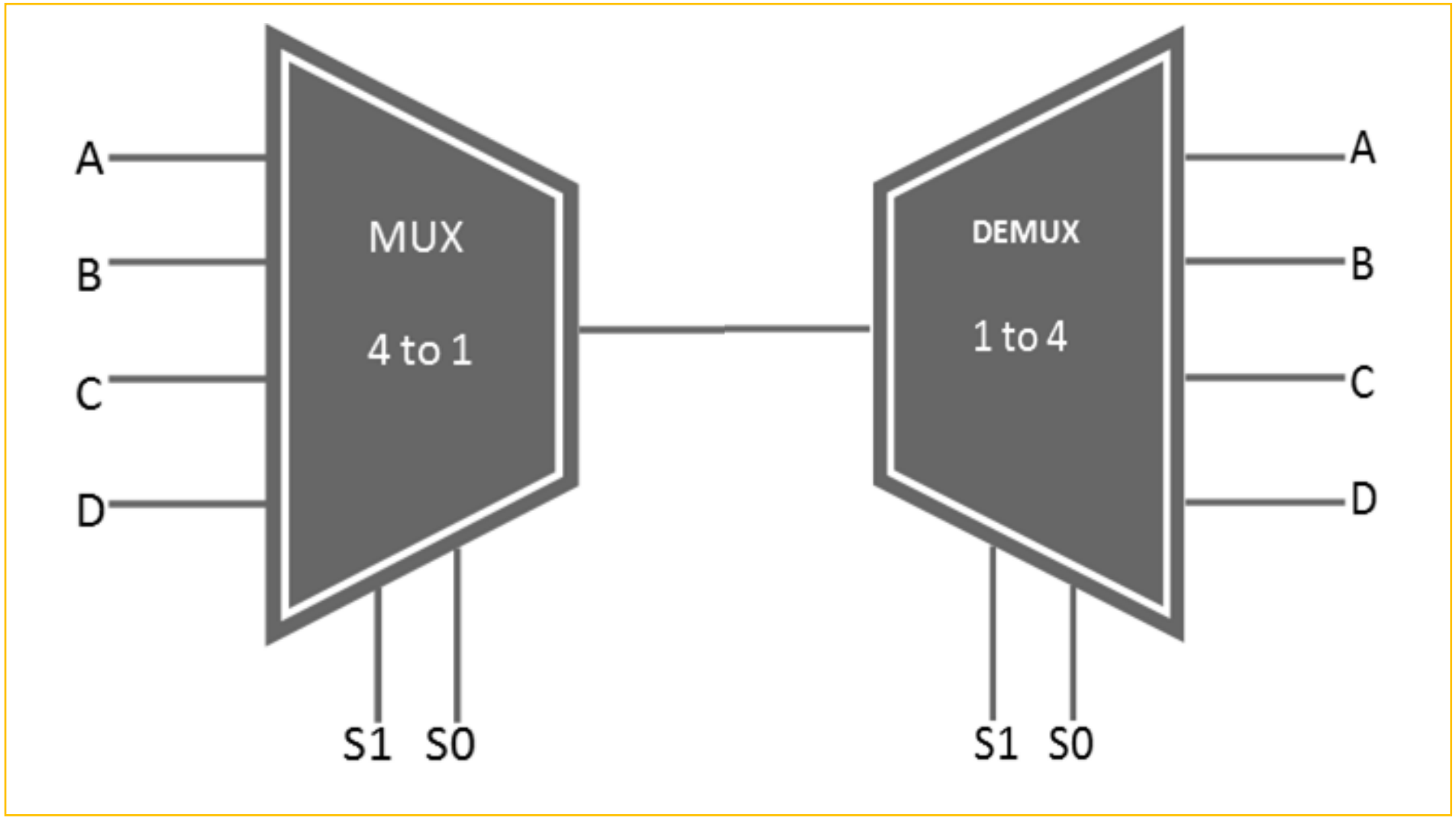
Example: Implement $F_1(A,B,C) = \Sigma(0,3,7)$, $F_2(A,B,C) = \Sigma(1,2,5)$ using **1-to-8** demultiplexer.



What About Using 1-to-4 DeMux??

Same as **Decoder** with:
Decoder Inputs → Selection Lines
DeMux Input → 1

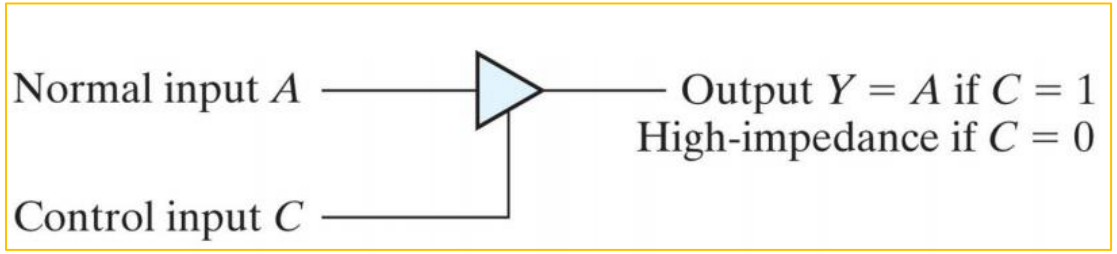
Mux/DeMux

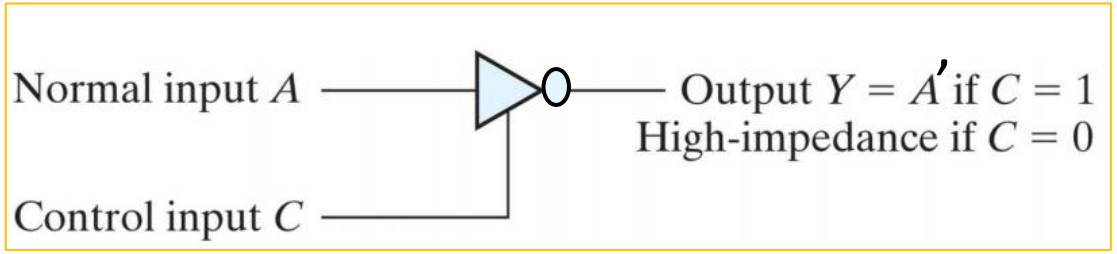


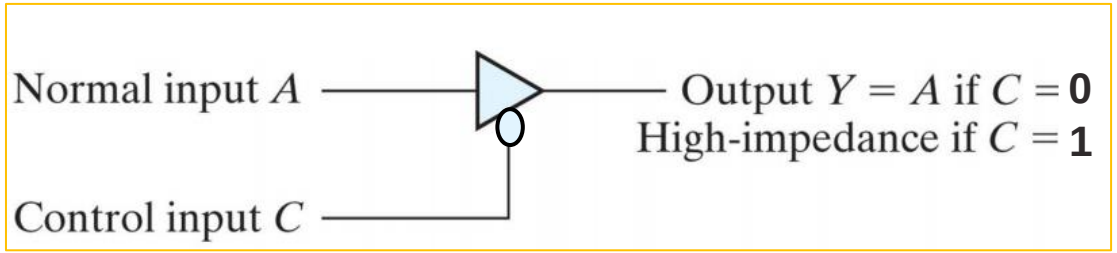
Three State Gates (Buffers)

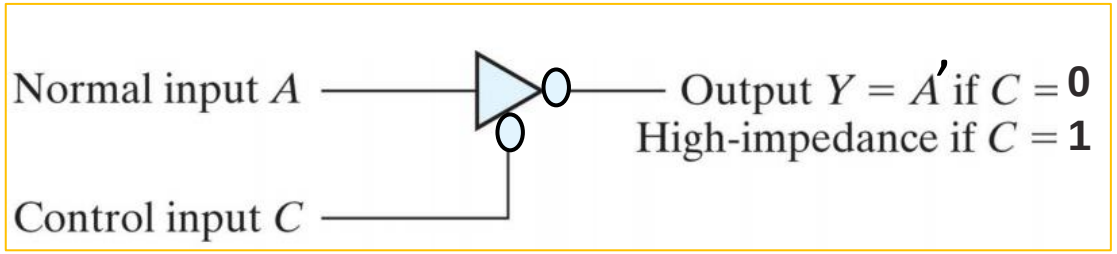
Three State or Tri-State or 3-State

- These gates can be in one of **2 possible states**
  1) An **enabled** state where the output may assume one of two possible values (**0, 1**)
  2) A **disabled** state where the gate output is in a the **Hi-impedance** (Hi-Z) state
     - The circuit behaves like an **open circuit**, which means that the output appears to be **disconnected**
     - The circuit has **NO** logic **significance**
     - The circuit connected to the output of the three-state gate is **NOT** affected by the inputs to the gate
- A control input (**C**) is used to **control** the gate into either the **enabled** or **disabled** state.
  - C could be **Normal** (Active **High**) or **Inverted** (Active **Low**)
  - Output could be **Normal** (**Buffer**) or **Complemented** (**Inverter**)





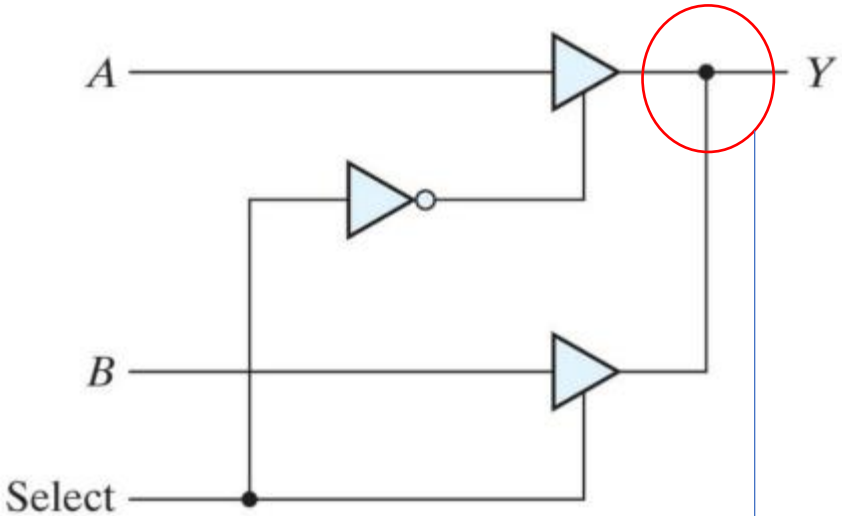




MUX with Three-States Gates

It is possible to implement **multiplexers** using **3-state** buffers

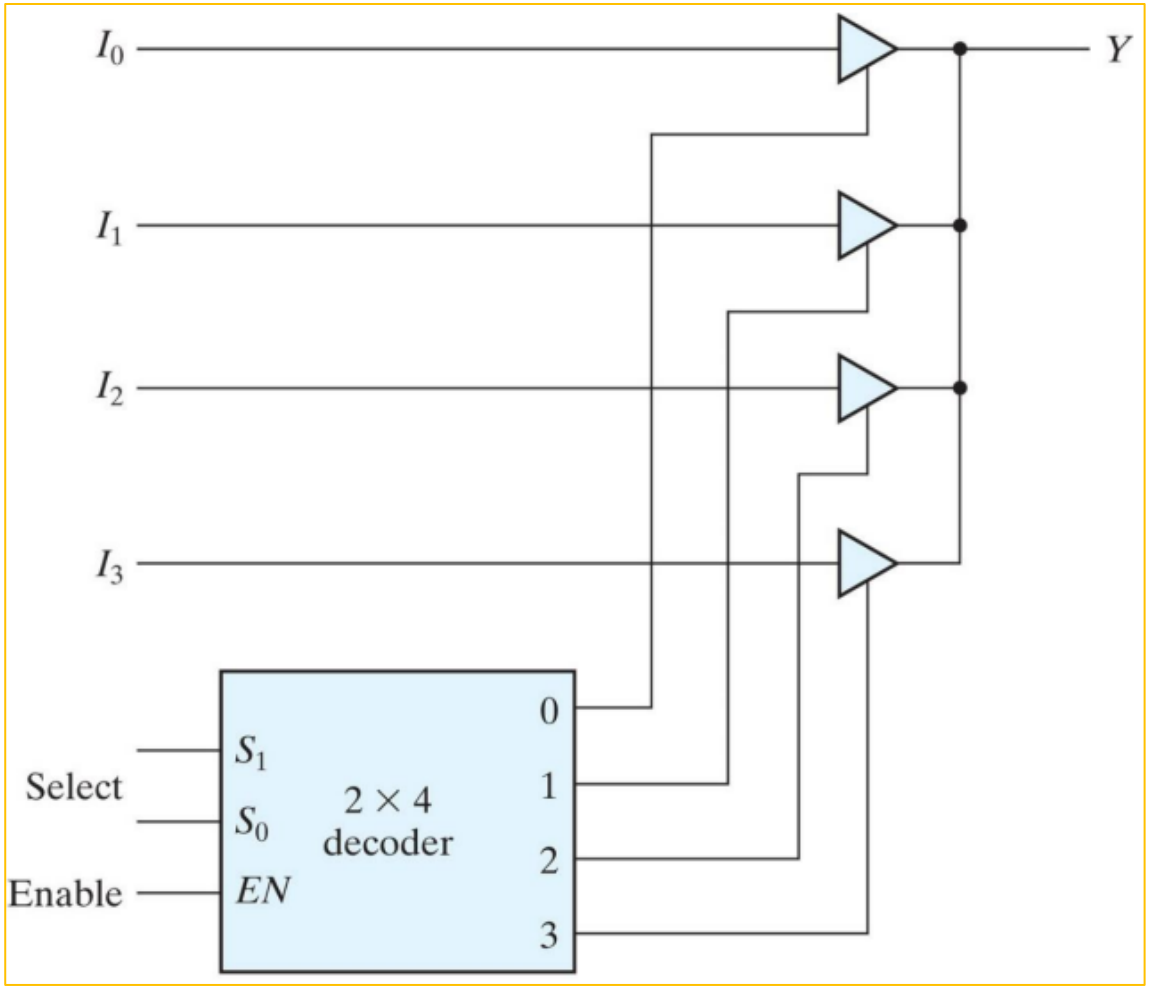
A **4-to-1** multiplexer may be constructed using **four** **3-state** buffers and a **2-to-4** decoder



2-to-1 Mux

Problem: **Direct** Connected/Wired Outputs → Short Circuit → Damage (**Burn**)

Solution: Make sure **ONLY ONE** Input is **enabled** by utilizing the **3-States Buffer**



4-to-1 Mux

Implement the following Boolean functions (Together): (With minimum number of inputs in the external gates)

$$F_1(A, B, C) = \sum(3, 5)$$
$$F_2(A, B, C) = \sum(2, 4, 5, 6, 7)$$

Using:

A. 3x8 decoder constructed with AND gates.
B. 3x8 decoder constructed with NAND gates.
C. 2x4 decoders constructed with NAND gates.

Implement the following Boolean function:

$$F_1(A, B, C, D) = \sum(0, 1, 2, 4, 6, 9, 12, 14)$$

Using:

A. 8-to-1 MUX.
B. 4-to-1 MUXes, with minimum external gates.

Implement each of the following Boolean functions (**Separately**):

$$F_1(A, B, C) = \sum(0, 1, 3, 5),\ F_2(A, B, C) = \sum(0, 1, 4, 5)$$

Using:

A. 4-to-1 MUX.
B. 1-to-4 DEMUX with one external gate.