

EEE 3342C: Digital Systems

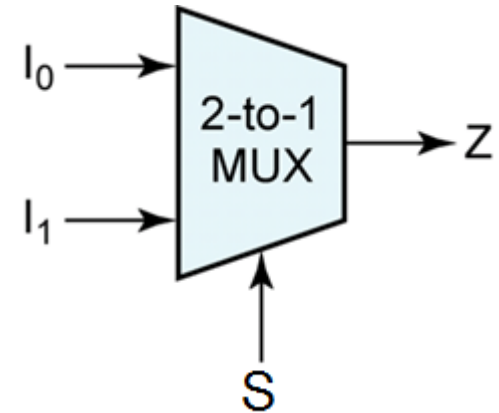
Chapter 9: Multiplexers, Decoders and Programmable Logic Devices

Instructor: Suboh A Suboh

Department of Electrical Engineering and Computer Science
University of Central Florida

Multiplexer

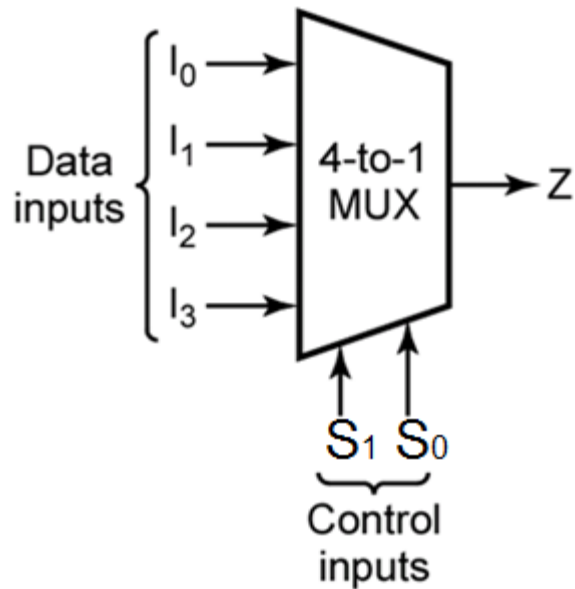
- Multiplexer, abbreviated as MUX
- Also called data selector
- Has 2^n or more inputs ($n \geq 1$)
- Has 1 output
- Has n selectors
- The output is equal to one of the inputs
 - Based on the value of the selector(s)



This is a 2-to-1 multiplexer
(or 2-to-1 MUX)

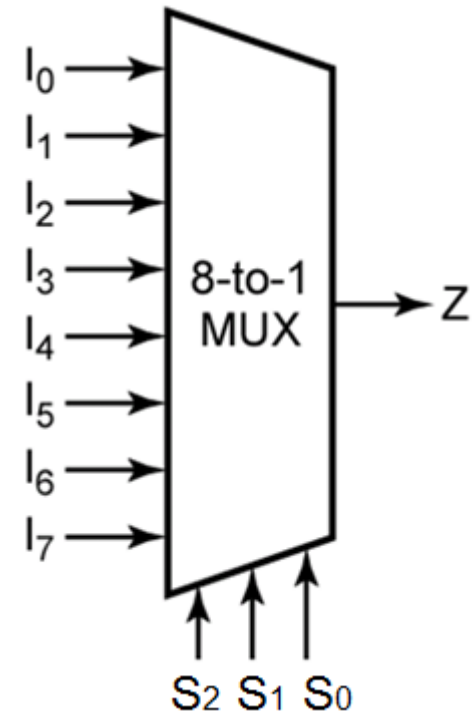
If $S=0$, $Z=I_0$

If $S=1$, $Z=I_1$



This is a 4-to-1 multiplexer
It has 4 data inputs and 1 output

If $S_1=0$ and $S_0=0$, then $Z = I_0$
 If $S_1=0$ and $S_0=1$, then $Z = I_1$
 If $S_1=1$ and $S_0=0$, then $Z = I_2$
 If $S_1=1$ and $S_0=1$, then $Z = I_3$



This is an 8-to-1 multiplexer
It has 8 data inputs and 1 output

If $S_2=0$, $S_1=0$, $S_0=0$, then $Z = I_0$
 If $S_2=0$, $S_1=0$, $S_0=1$, then $Z = I_1$
 ...
 If $S_2=1$, $S_1=1$, $S_0=1$, then $Z = I_7$

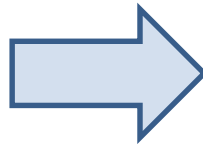
16-to-1 MUX has 4 selectors
 32-to-1 MUX has 5 selectors
 2^n -to-1 MUX has n selectors...

Boolean Expression for the Multiplexer

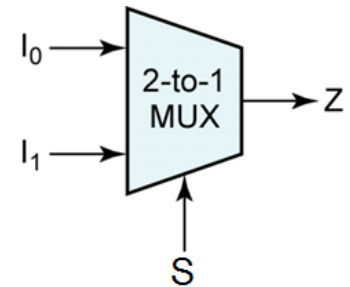
- For 2-to-1 multiplexer

Truth table for
2-to-1 MUX

S	I ₁	I ₀	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



I ₁ I ₀	S	
	0	1
00	0	0
01	1	0
11	1	1
10	0	1

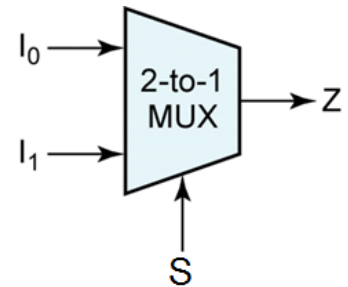


$$Z = S'.I_0 + S.I_1$$

Boolean Expression for the Multiplexer

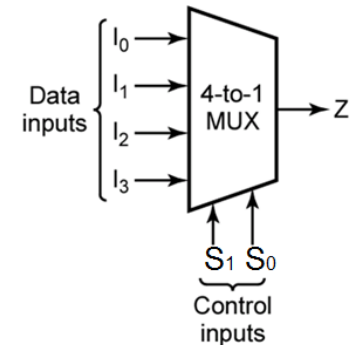
- For 2-to-1 multiplexer:

$$Z = S'.I_0 + S.I_1$$



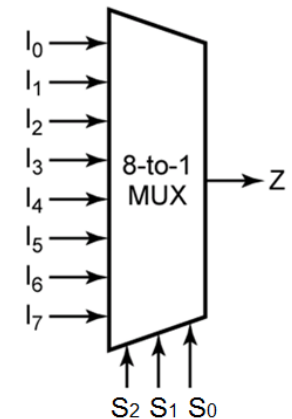
- For 4-to-1 multiplexer:

$$Z = S_1'S_0'I_0 + S_1'S_0I_1 + S_1S_0'I_2 + S_1S_0I_3$$



- For 8-to-1 multiplexer:

$$Z = S_2'S_1'S_0'I_0 + S_2'S_1'S_0I_1 + S_2'S_1S_0'I_2 + S_2'S_1S_0I_3 + S_2S_1'S_0'I_4 + S_2S_1'S_0I_5 + S_2S_1S_0'I_6 + S_2S_1S_0I_7$$

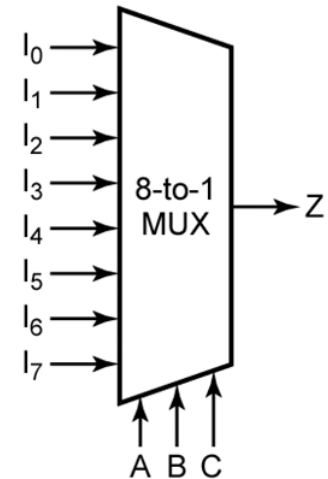
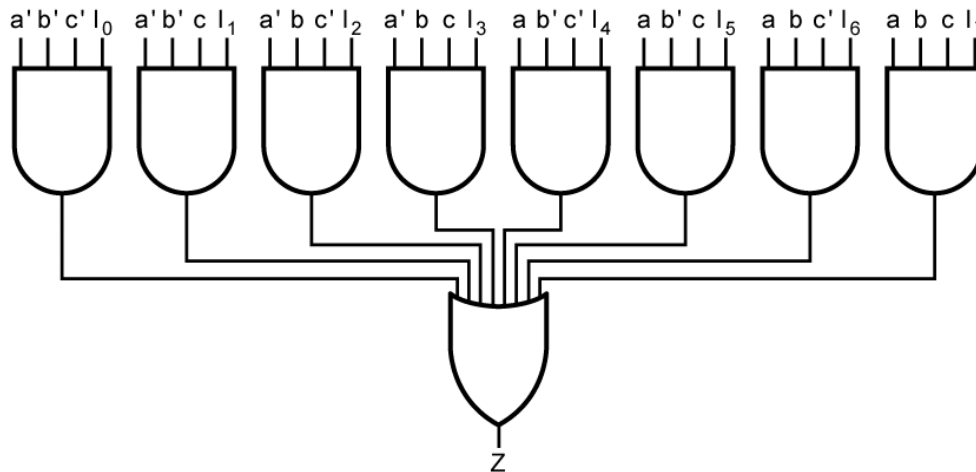


Circuit for 8-to-1 Multiplexer

- The Boolean expression is:

$$Z = A'B'C'I_0 + A'B'CI_1 + A'BC'I_2 + A'BCI_3 + AB'C'I_4 + AB'CI_5 + ABC'I_6 + ABCI_7$$

$$Z = \sum_{k=0}^{2^n-1} m_k I_k$$

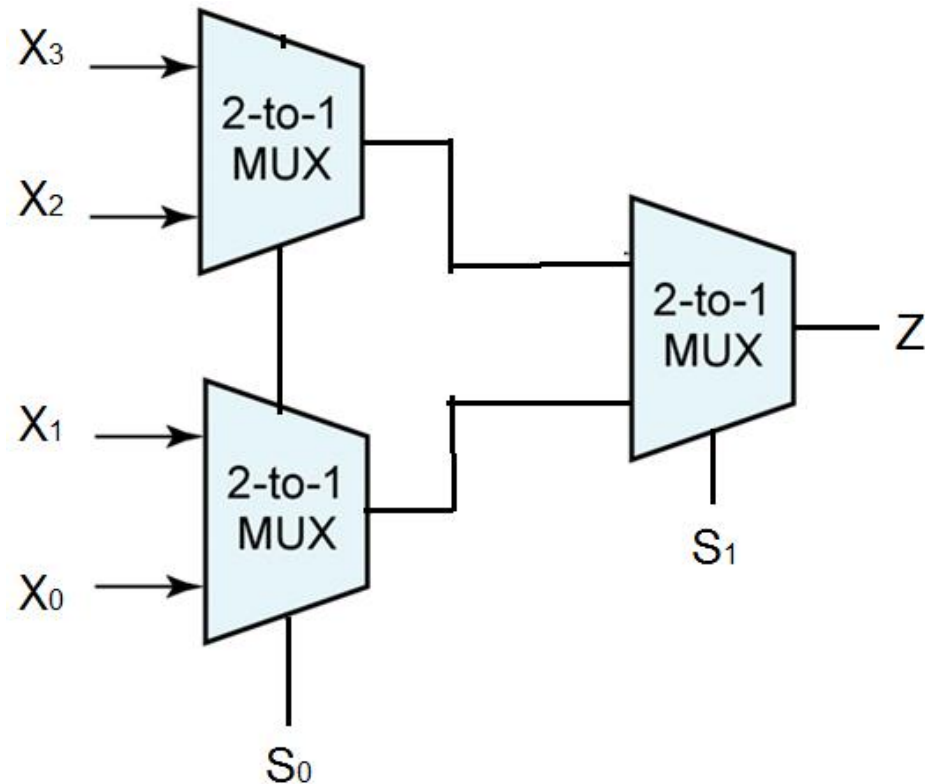
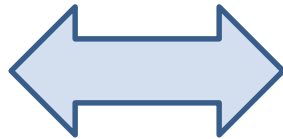
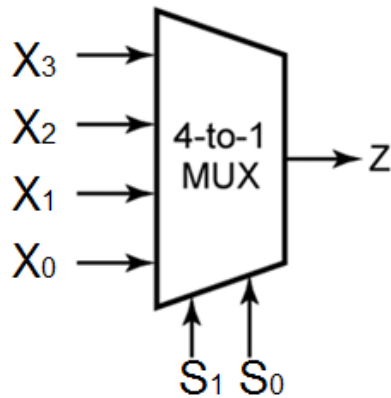


It is a Sum-Of-Products (SOP) circuit

4-to-1 MUX using 2-to-1 MUXes

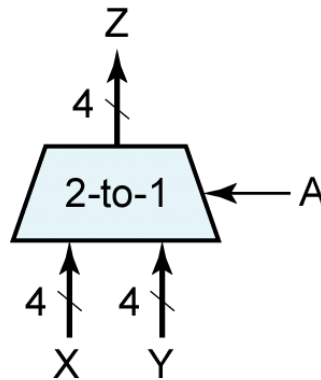
- We need to build a 4-to-1 mux using several 2-to-1 MUXes
- The configuration is shown in the block diagram on the right side

S_1	S_0	Z
0	0	X_0
0	1	X_1
1	0	X_2
1	1	X_3



Multiplexer Input (multiple bits)

- In a multiplexer, the inputs and the output have the same number of bits
- It could be 1 bit, or it could be higher

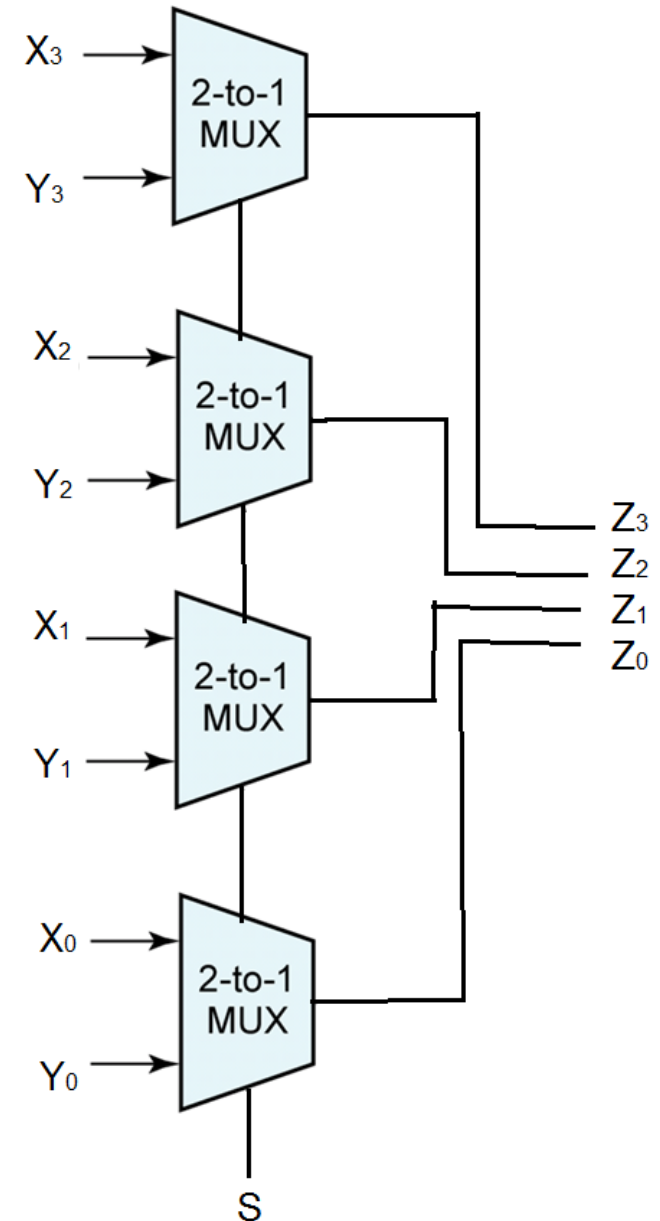
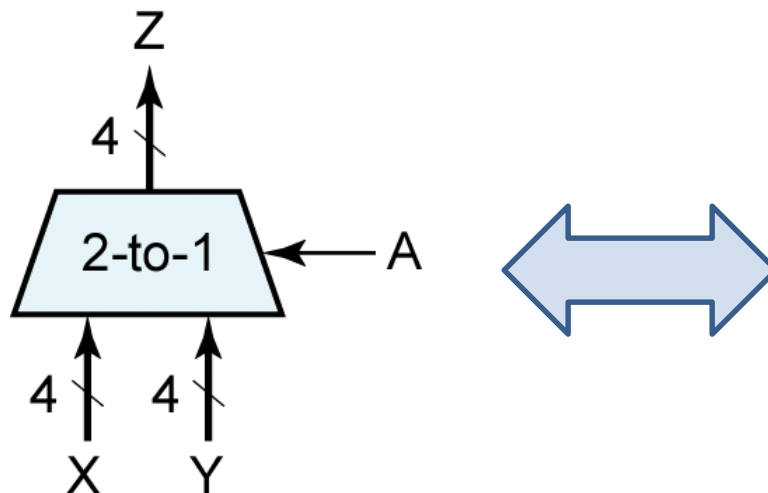


- In this 2-to-1 multiplexer,
 - Each input is 4 bits
 - The output is 4 bits

If $A=1$, $Z = X_3X_2X_1X_0$
If $A=0$, $Z = Y_3Y_2Y_1Y_0$

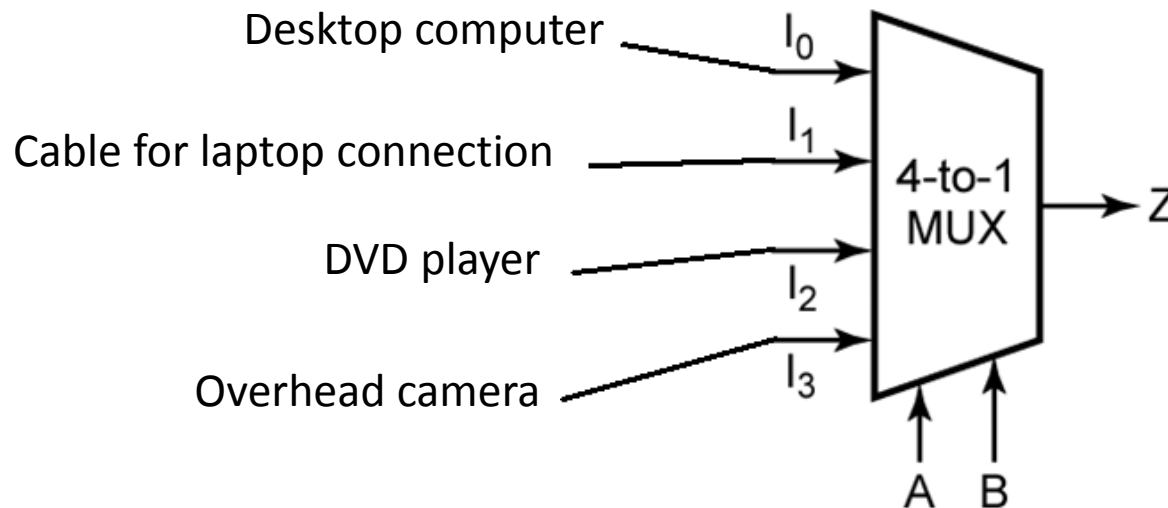
2-to-1 Multiplexer (with 4-bit inputs)

- We need to build a 2-to-1 mux with 4-bit inputs and output
- We have 2-to-1 muxes with 1-bit input
- The configuration is shown in the block diagram on the right side



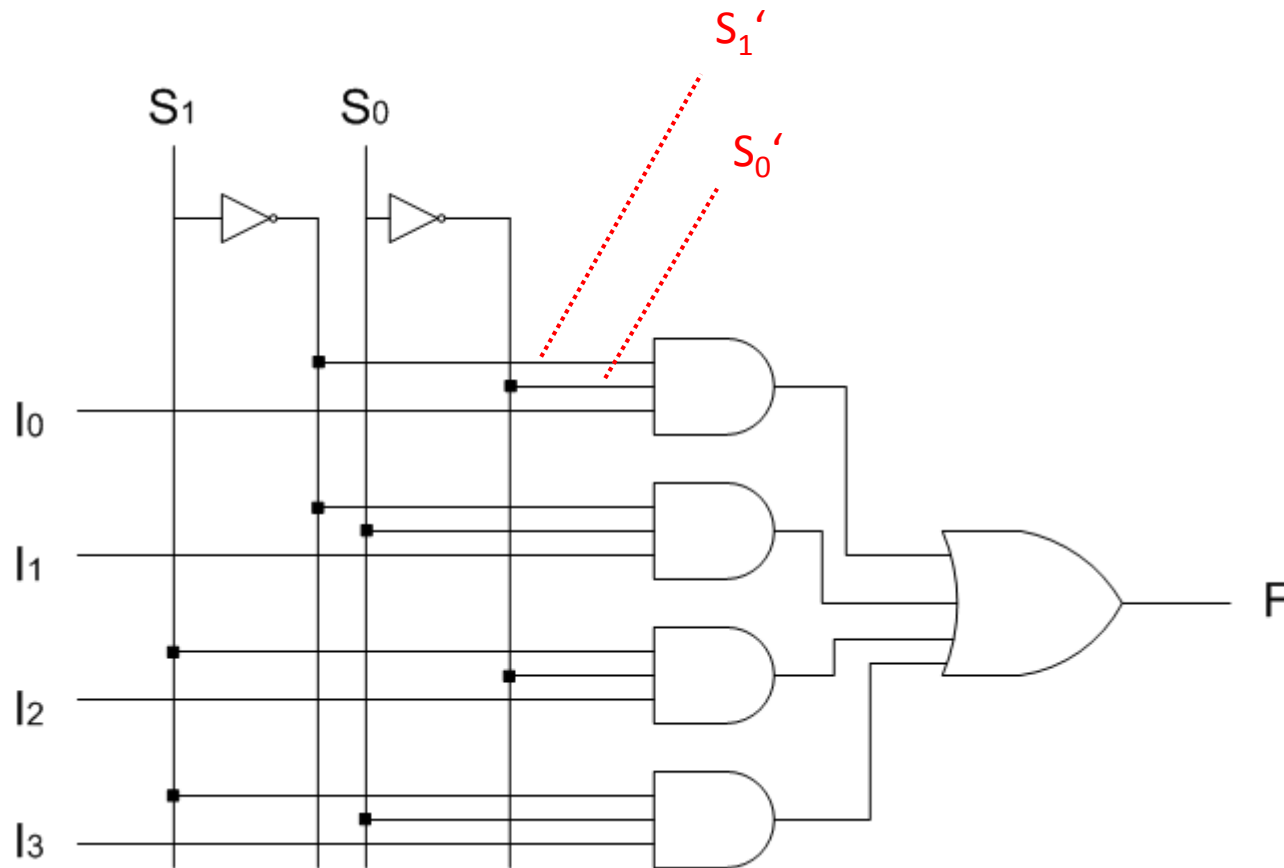
Example of Multiplexer Use

- Classroom projector display



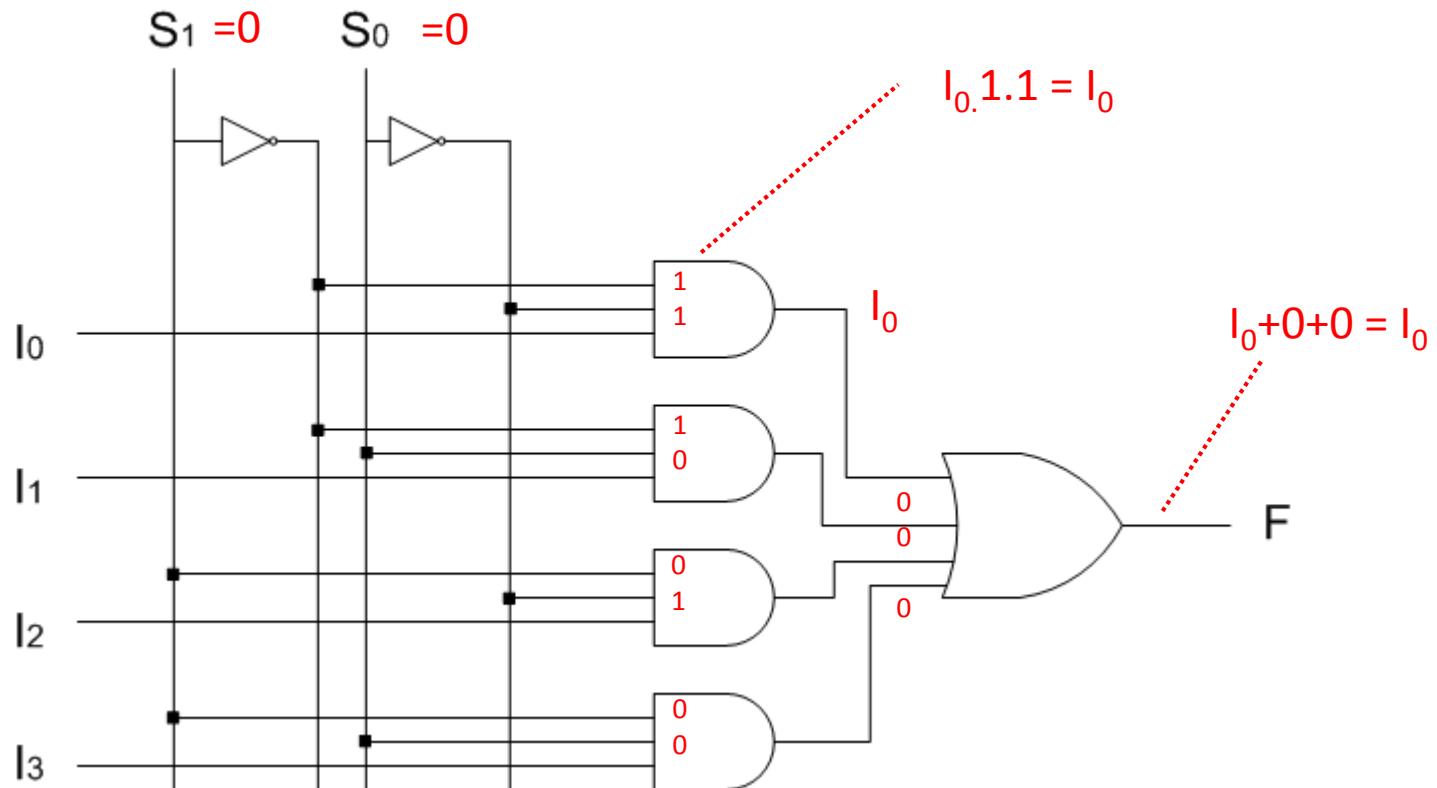
If $AB=00$, the projector shows the display of the desktop computer (slides)
If $AB=01$, the projector shows the display of the laptop that's connected via the cable
If $AB=10$, the projector shows the image from the DVD player
If $AB=11$, the projector shows the image from the over head camera

Logic Circuit for a 4-to-1 Multiplexer

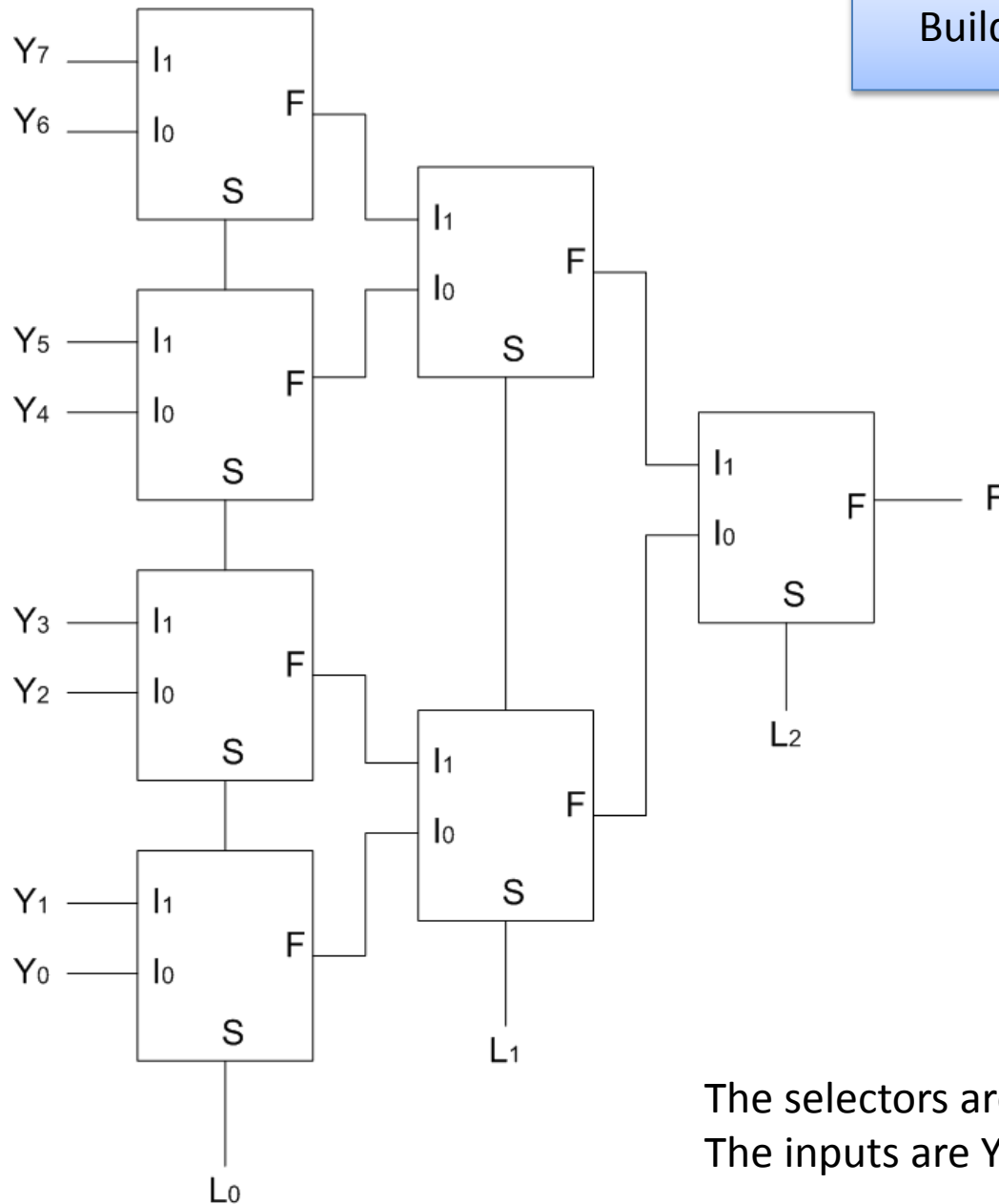


Logic Circuit for a 4-to-1 Multiplexer

Let's consider the case when $S_1=0$ and $S_0=0$



Build an 8-to-1 MUX using 2-to-1 MUXes

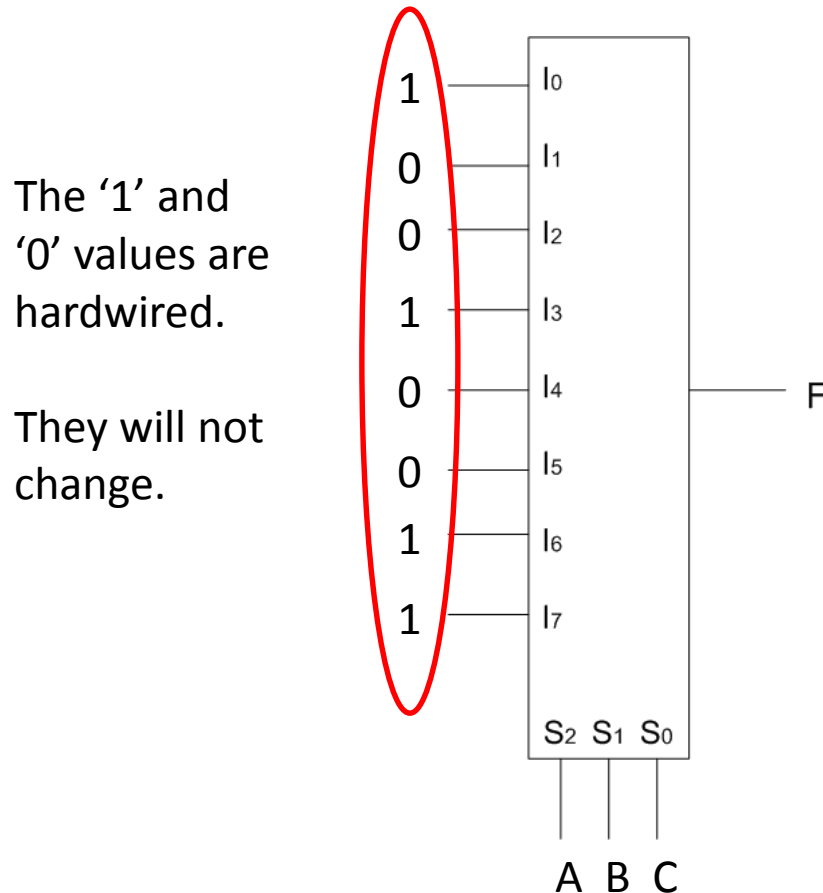


Truth table

L_2	L_1	L_0	F
0	0	0	I_0
0	0	1	I_1
0	1	0	I_2
0	1	1	I_3
1	0	0	I_4
1	0	1	I_5
1	1	0	I_6
1	1	1	I_7

Implementing a Function Using a Multiplexer

- We have the function: $F(A,B,C) = A'B'C' + A'BC + ABC' + ABC$
- This function is $F(A,B,C) = \sum m(0, 3, 6, 7)$
- This function has 8 minterms; we need an 8-to-1 multiplexer



We put the column of F from the truth table at the inputs of the multiplexer.

When ABC varies from 000 to 111, the result will be the column F from the truth table.

So, the multiplexer will give the correct value.

Truth table of F

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

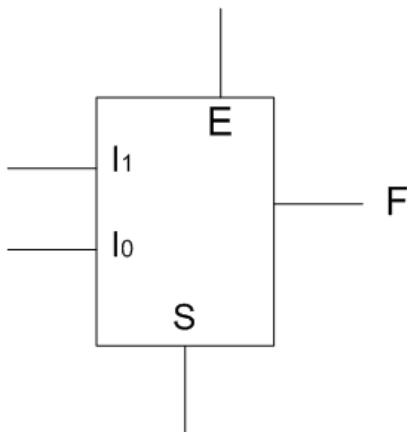
Multiplexer with Enable Signal

- When the enable signal is active, the multiplexer will operate as usual
- When the enable signal is inactive, the output is always 0

Active high enable:

If $E=1$, operates as usual

If $E=0$, output is always zero

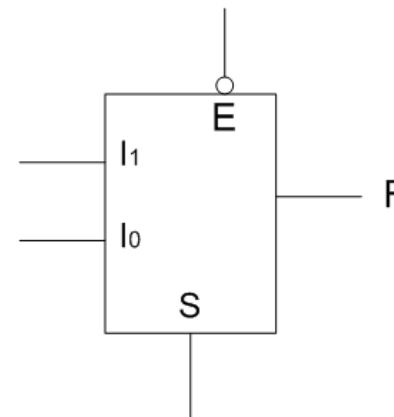


E	S	I ₁	I ₀	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1

Active low enable:

If $E=0$, operates as usual

If $E=1$, output is always zero



E	S	I ₁	I ₀	F
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

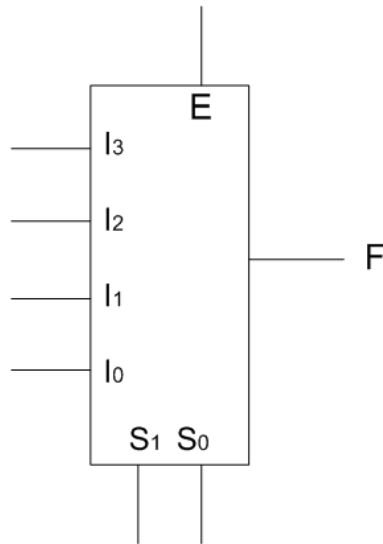
Multiplexer with Enable Signal

- 4-to-1 multiplexer with enable signal

Active high enable:

If $E=1$, operates as usual

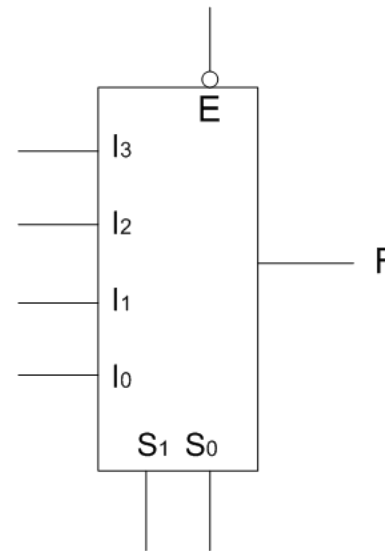
If $E=0$, output is always zero



Active low enable:

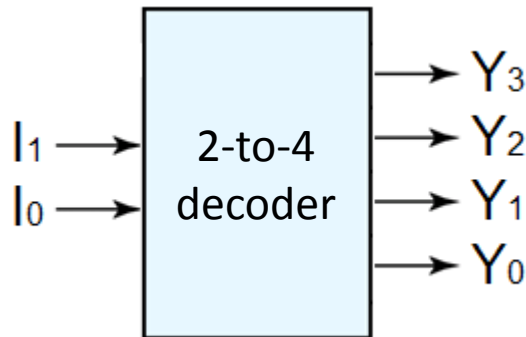
If $E=0$, operates as usual

If $E=1$, output is always zero



Decoder

- A decoder has n inputs and 2^n outputs
- It generates the 2^n minterms; the variables are the inputs
- For a given input, only one output is 1, the others are 0



Truth table for the 2-to-4 encoder

I_1	I_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

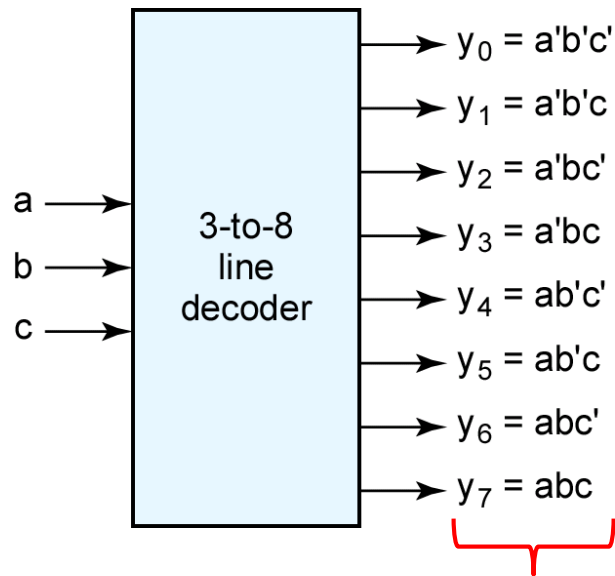
$$\begin{aligned} Y_3 &= I_1 \cdot I_0 = m_3 && \text{(minterm 3)} \\ Y_2 &= I_1 \cdot I_0' = m_2 && \text{(minterm 2)} \\ Y_1 &= I_1' \cdot I_0 = m_1 && \text{(minterm 1)} \\ Y_0 &= I_1' \cdot I_0' = m_0 && \text{(minterm 0)} \end{aligned}$$

Only 1 output is equal to 1 in a line

The outputs of the decoder are the minterms of the inputs

Decoder

- The 3-to-8 decoder



Each output has the Boolean expression of a minterm.

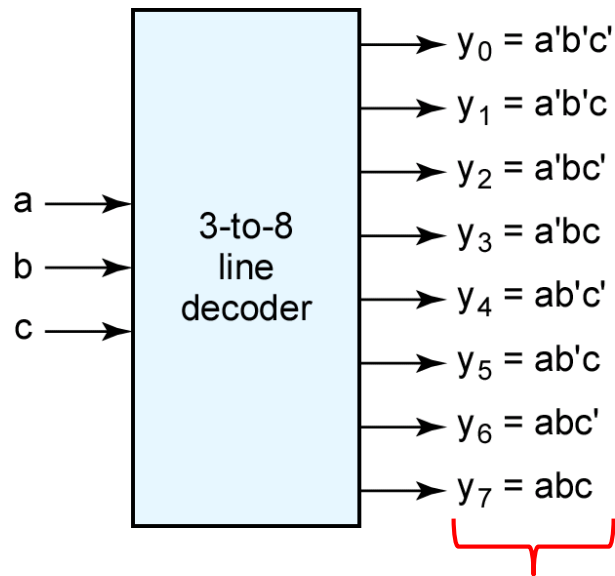
Truth table for the 3-to-8 encoder

a	b	c	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_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 1 output is equal to 1 in a line

Decoder

- The 3-to-8 decoder



Each output has the Boolean expression of a minterm.

Truth table for the 3-to-8 encoder

a	b	c	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_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 1 output is equal to 1 in a line

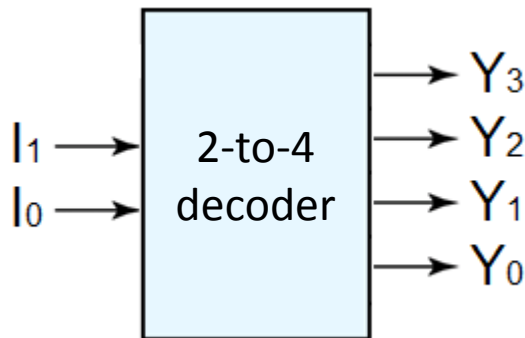
Decoder with Enable Signal

- When the enable signal is active, the decoder will operate as usual
- When the enable signal is inactive, all the outputs are 0

Active high enable:

If $E=1$, operate as usual

If $E=0$, all outputs are zero

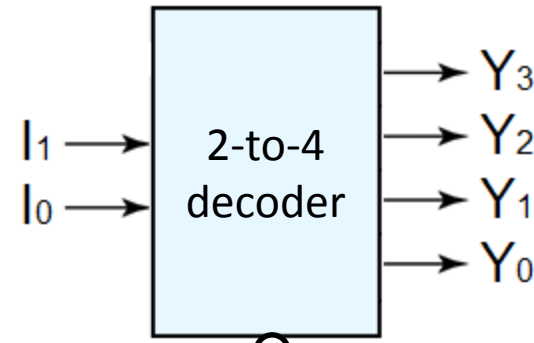


E	I ₁	I ₀	Y ₃	Y ₂	Y ₁	Y ₀
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

Active low enable:

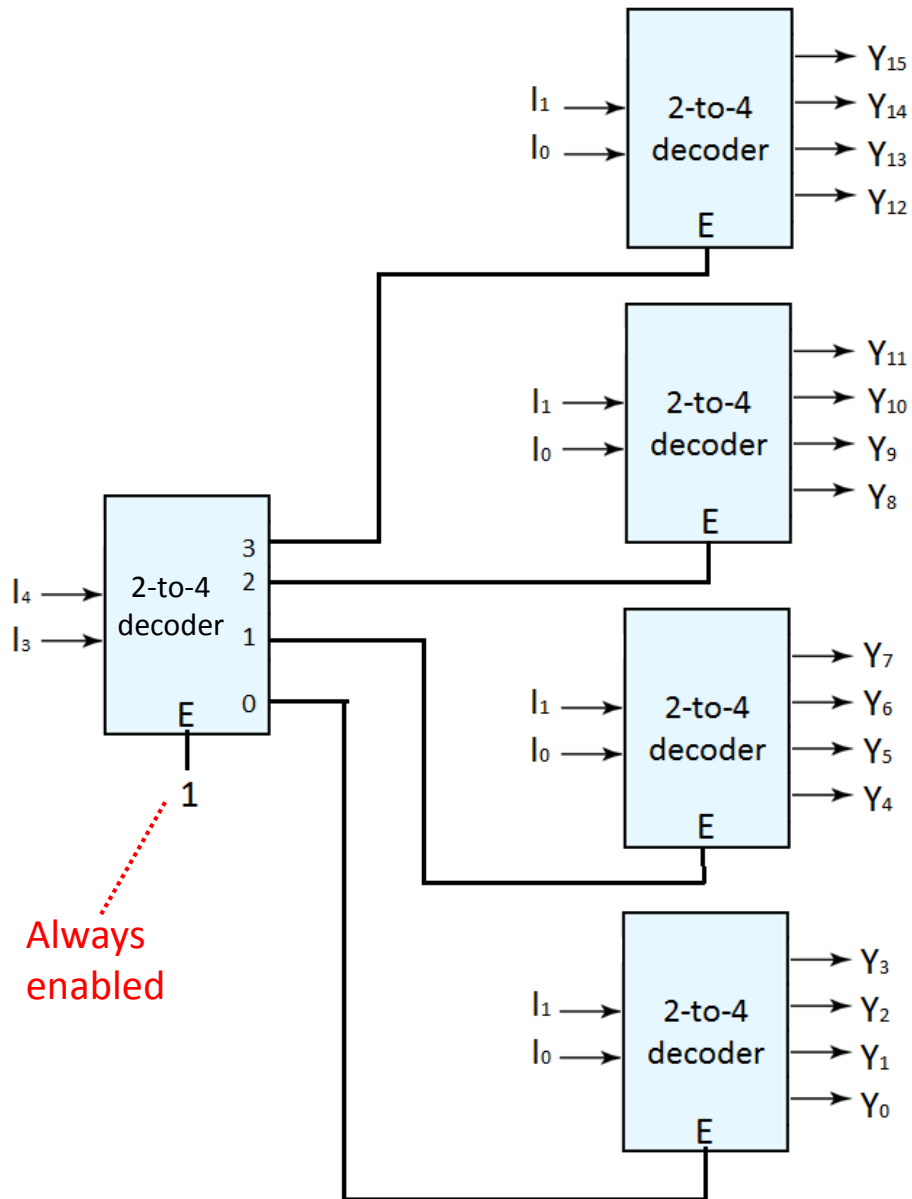
If $E=0$, operate as usual

If $E=1$, all outputs are zero



E	I ₁	I ₀	Y ₃	Y ₂	Y ₁	Y ₀
0	0	0	0	0	0	1
0	0	1	0	0	1	0
0	1	0	0	1	0	0
0	1	1	1	0	0	0
1	0	0	0	0	0	0
1	0	1	0	0	0	0
1	1	0	0	0	0	0
1	1	1	0	0	0	0

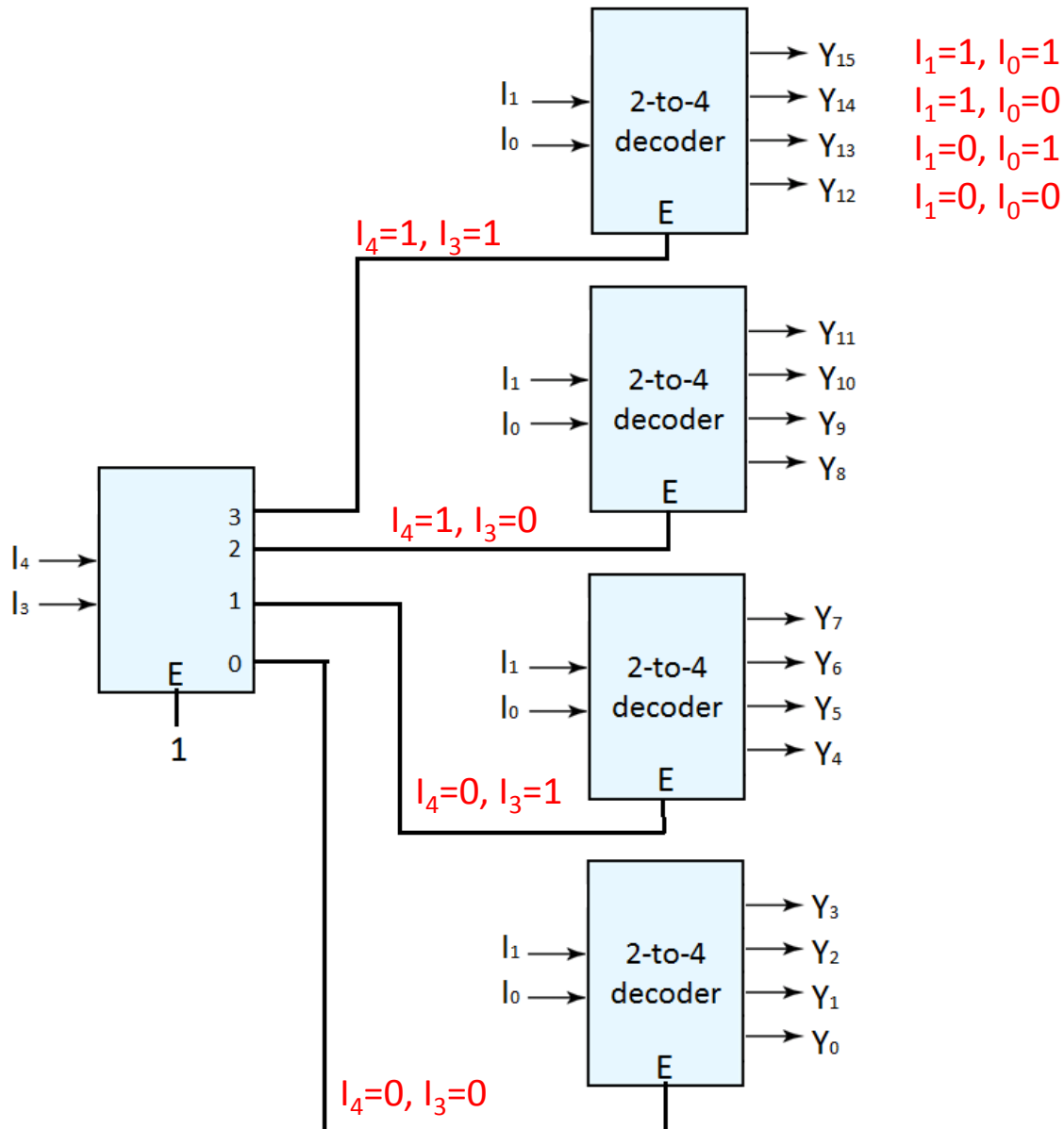
Build a 4-to-16 decoder using 2-to-4 decoders



Only one of the 4 decoders on the right side is enabled.

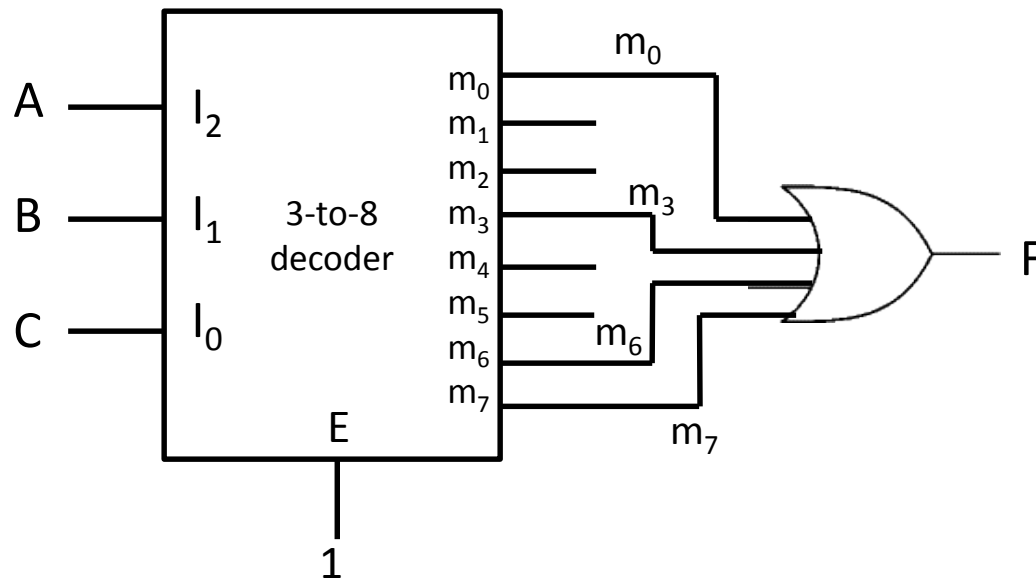
The ones that are not enabled will give zero in all outputs.

Build a 4-to-16 decoder using 2-to-4 decoders



Implementing a Function Using a Decoder

- We have the function: $F(A,B,C) = A'B'C' + A'BC + ABC' + ABC$
- This function is $F(A,B,C) = \sum m(0, 3, 6, 7)$
- This function has 8 minterms; we need a 3-to-8 decoder

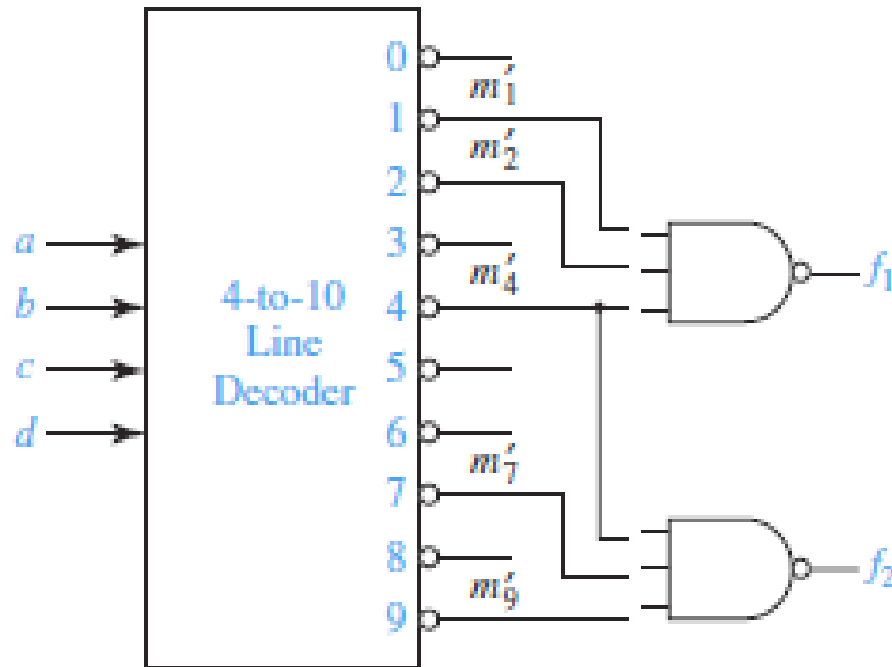


Truth table of F

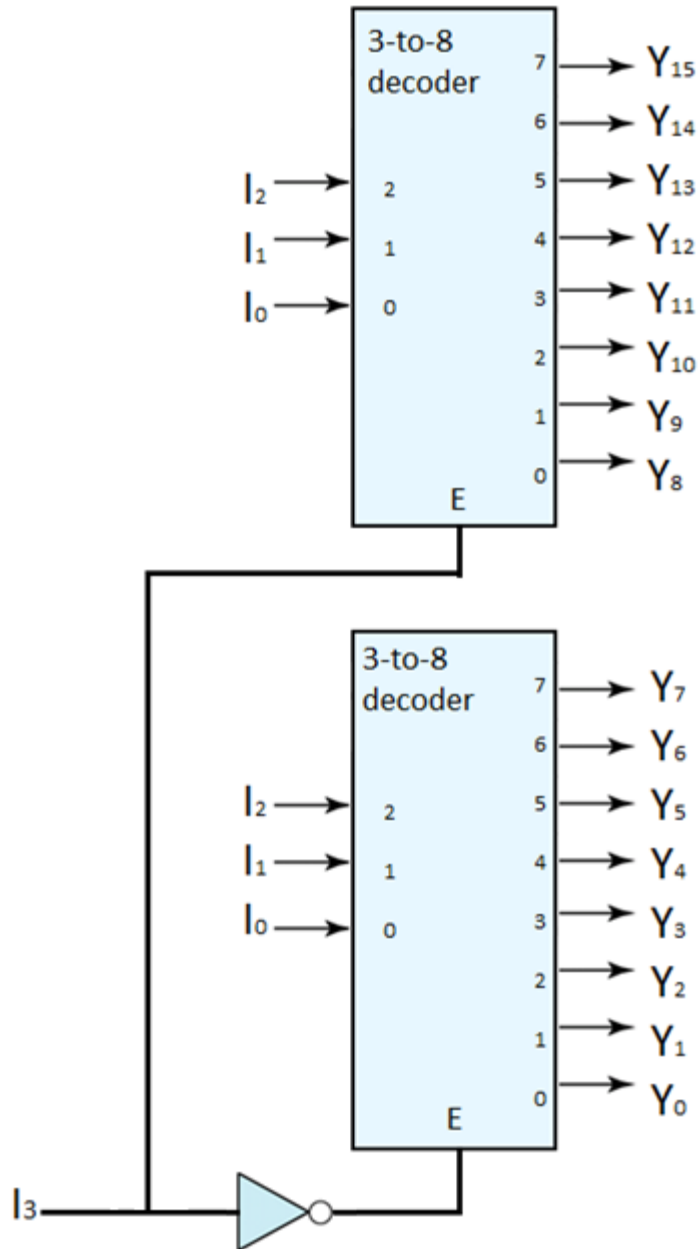
A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Implementing a Function Using a Decoder

- Decoder 7442 4-10 decoder. Inverted outputs. **Decimal digits in BCD**
- We have the function: $f_1(a,b,c,d) = m_1 + m_2 + m_4$, $f_2(a,b,c,d) = m_4 + m_7 + m_9$
- This function is $f_1 = (m_1' \cdot m_2' \cdot m_4')'$, and $f_2 = (m_4' \cdot m_7' \cdot m_9')'$
- This function has 8 minterms; we need a 4-to-16(or less) decoder



Build a 4-to-16 decoder using 3-to-8 decoders

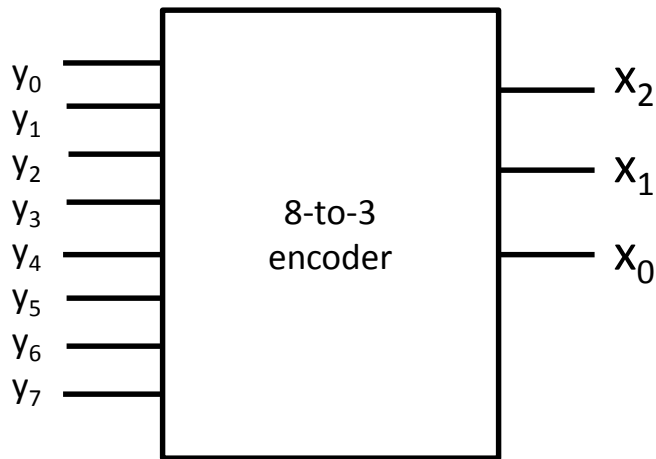


Truth table

I_3	I_2	I_1	I_0	Y_{15}	Y_{14}	Y_{13}	Y_{12}	Y_{11}	Y_{10}	Y_9	Y_8	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	0	0	0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Encoder

- An encoder is the inverse function of the decoder
- It has n outputs and 2^n inputs
- Only 1 input should be 1, the others should be 0
- The output indicates which input is 1
 - If $y_0=1$, then $x_2x_1x_0=000$; if $y_1=1$, then $x_2x_1x_0=001$; etc.

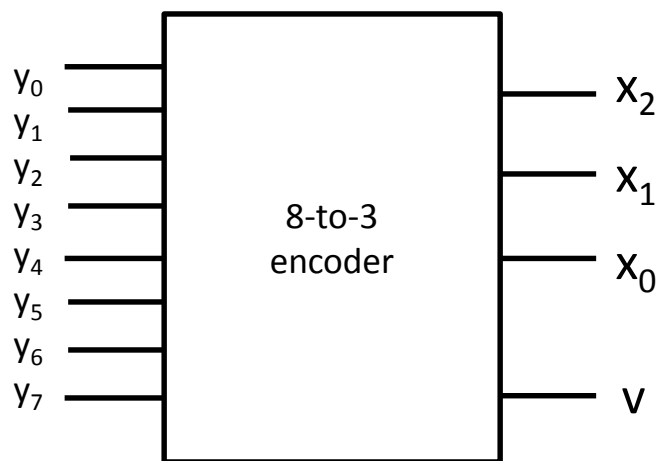


Truth table

y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	x_2	x_1	x_0
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

Priority Encoder

- It is similar to an encoder
- If more than one input is 1; the output is based on the priority
- We can give priority (1) to the larger number, or (2) to the smaller number
- It has a signal (v) that indicates if all the inputs are zero



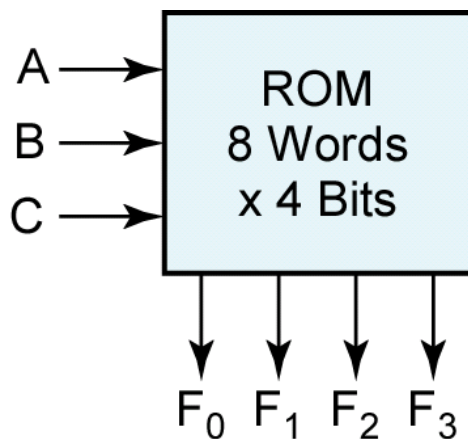
This encoder gives priority to the signal with the highest number

Truth table

Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7	X_2	X_1	X_0	v
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
X	1	0	0	0	0	0	0	0	0	1	1
X	X	1	0	0	0	0	0	0	1	0	1
X	X	X	1	0	0	0	0	0	1	1	1
X	X	X	X	1	0	0	0	1	0	0	1
X	X	X	X	X	1	0	0	1	0	1	1
X	X	X	X	X	X	1	0	1	1	0	1
X	X	X	X	X	X	X	1	1	1	1	1

Read-Only Memory (ROM)

- ROM is a type of memory; data is stored in it
- It can be read only; the data in it cannot be changed
- Below is the diagram of a ROM
- There are 8 words in it (a word is a data element; here, each word is 4 bits)
- The address is 3 bits; valid addresses are from 0 (000) to 7 (111)
- Write the address at 'ABC'; the word at address ABC goes on the output at "F₀F₁F₂F₃"

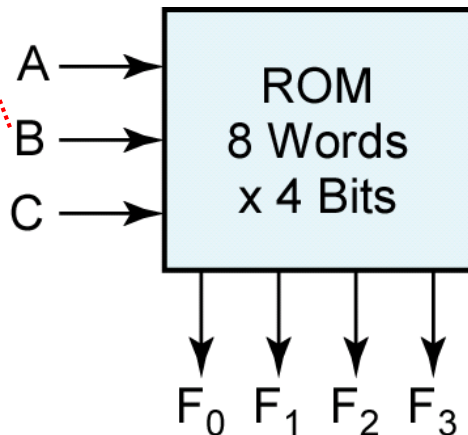


Note: In contrast, the memory in computers is called Random Access Memory (RAM). It can be read and written. Any program or file that is loaded goes into the RAM

Implementing Multiple Functions Using ROM

- We have a ROM that stores 8 words; each word is 4 bits
- The address has 3 bits; so the function could have 3 variables
- The word has 4 bits; so we can implement 4 functions

The variables of the functions go in the address bits



Truth table for ROM

A	B	C	F ₀	F ₁	F ₂	F ₃
0	0	0	1	0	1	0
0	0	1	1	0	1	0
0	1	0	0	1	1	1
0	1	1	0	1	0	1
1	0	0	1	1	0	0
1	0	1	0	0	0	1
1	1	0	1	1	1	1
1	1	1	0	1	0	1

How to implement 4 functions in the ROM?

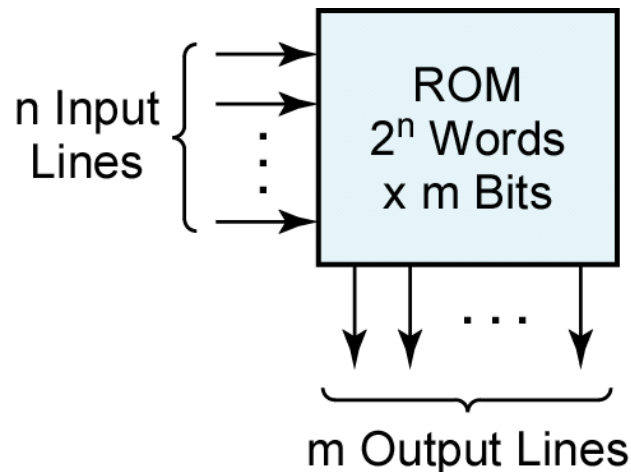
Write the truth table of the 4 functions.

Take the output of the functions and store it in the ROM.

The output of the functions are on the output lines of the ROM

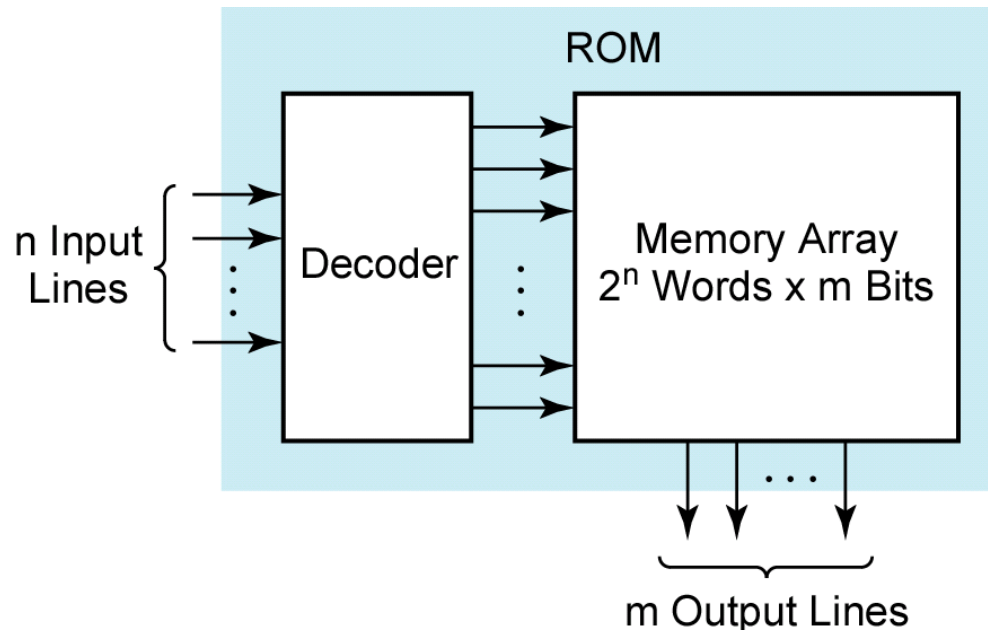
A ROM of Large Size

- The diagram below shows a ROM of arbitrary size
- The address is n bits; so it contains 2^n words
- Each word has m bits (so, there are m output lines)
- How many functions can we implement with this ROM?
 - Answer: m , because there are m output lines
- How many variables can each function have?
 - Answer: n , because the address has n bits



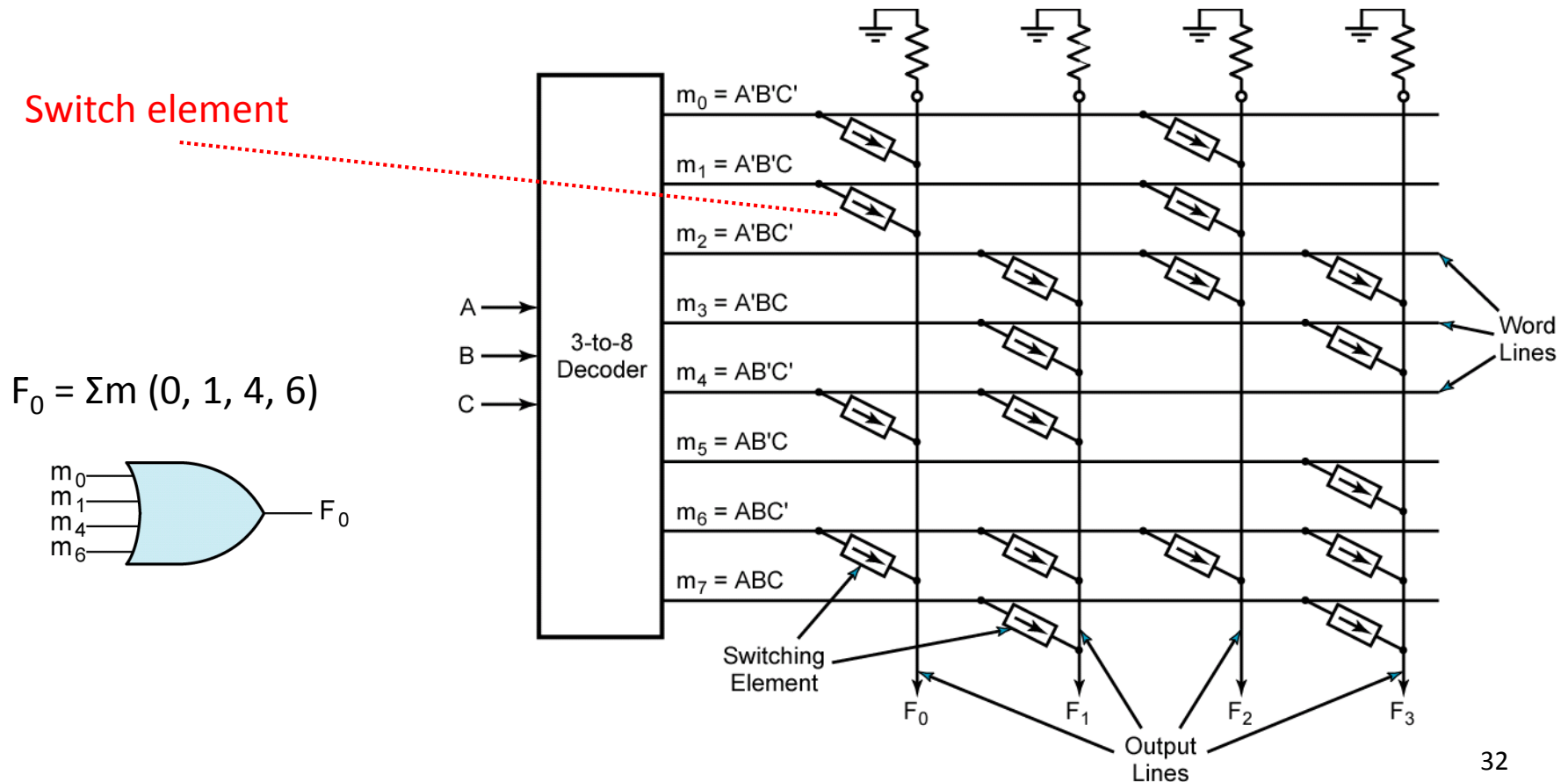
Structure of a ROM

- There is a decoder; it takes the address
- One of the decoder's outputs is 1, the others are 0
- There is a Memory Array; the data is stored in it
- The value of 1 from the decoder selects one word; this word goes to the output



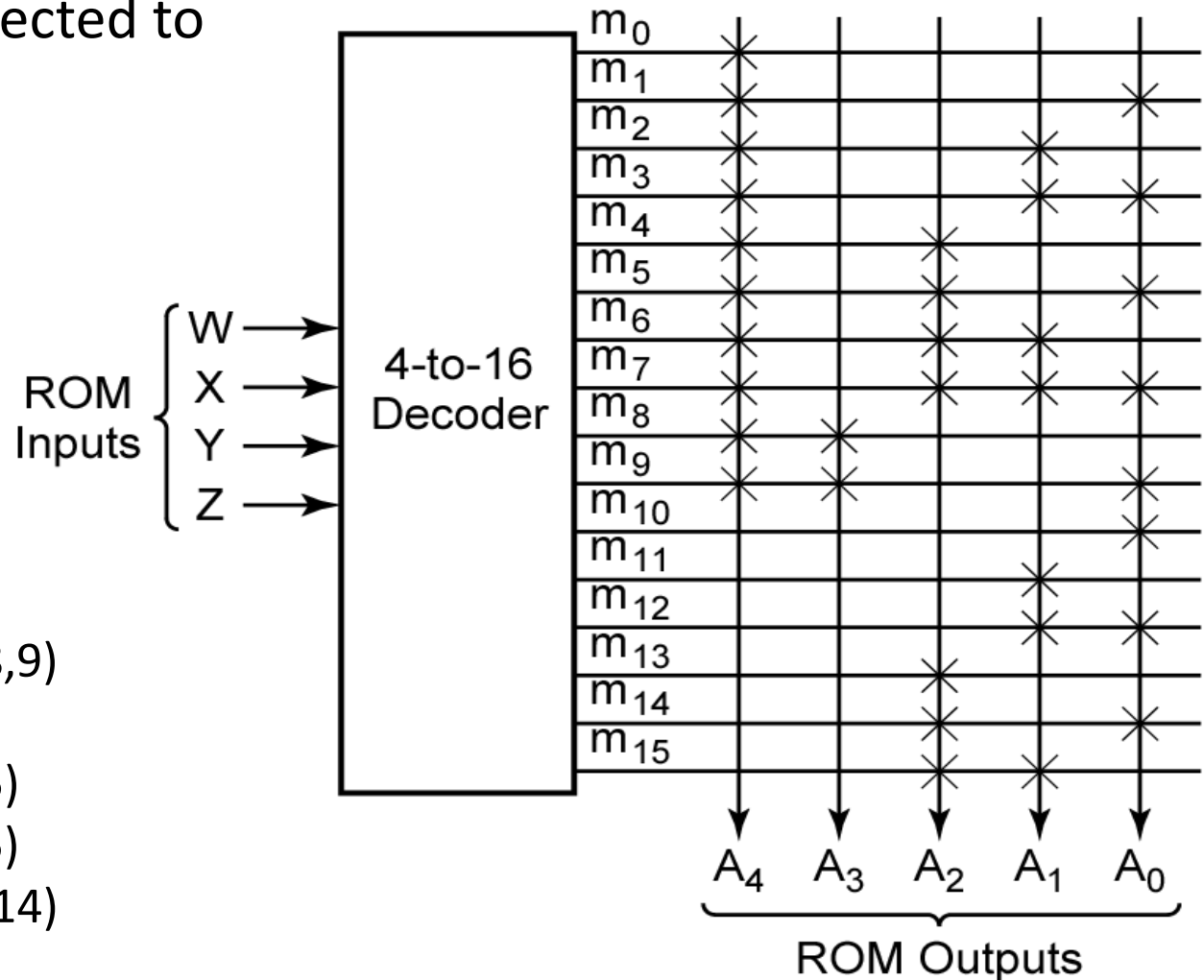
Structure of a ROM (more details)

- Every function is connected to ground; passing by a resistor
- The vertical line of a function is also connected to the minterms of the function
- We have: $F_0 = \sum m(0, 1, 4, 6)$; the vertical line of F_0 is connected to minterms 0, 1, 4, 6
- If the input of the decoder is one of the minterms, F_0 will take a value of 1
- Otherwise, F_0 will take a value of 0 from the ground



Structure of a ROM (another way)

- The X sign means the minterm is connected to the output line



The ROM in the figure,

$$A_4 = \sum m(0,1,2,3,4,5,6,7,8,9)$$

$$A_3 = \sum m(8,9)$$

$$A_2 = \sum m(4,5,6,7,13,14,15)$$

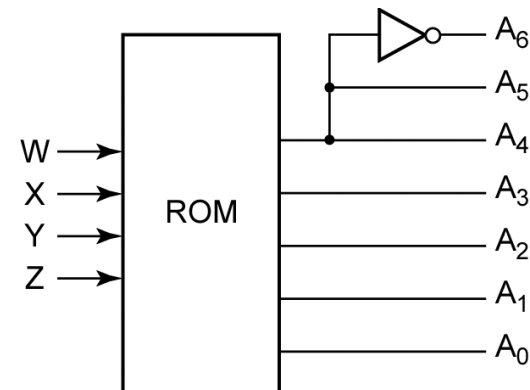
$$A_1 = \sum m(2,3,6,7,11,12,15)$$

$$A_0 = \sum m(1,3,5,7,9,10,12,14)$$

Implementing a Function with a ROM

Input				Output						
W	X	Y	Z	A ₆	A ₅	A ₄	A ₃	A ₂	A ₁	A ₀
0	0	0	0	0	1	1	0	0	0	0
0	0	0	1	0	1	1	0	0	0	1
0	0	1	0	0	1	1	0	0	1	0
0	0	1	1	0	1	1	0	0	1	1
0	1	0	0	0	1	1	0	1	0	0
0	1	0	1	0	1	1	0	1	0	1
0	1	1	0	0	1	1	0	1	1	0
0	1	1	1	0	1	1	0	1	1	1
1	0	0	0	0	1	1	1	0	0	0
1	0	0	1	0	1	1	1	0	0	1
1	0	1	0	1	0	0	0	0	0	1
1	0	1	1	1	0	0	0	0	1	0
1	1	0	0	1	0	0	0	0	1	1
1	1	0	1	1	0	0	0	1	0	0
1	1	1	0	1	0	0	0	1	0	1
1	1	1	1	1	0	0	0	1	1	0

- The ROM has 16 words; each word is 5 bits
- We can use this ROM to implement 5 functions
- Can we implement 7 functions?
- Look at the truth table.
- Observe that A₅ is equivalent to A₄ and A₆ is the inverse of A₄
- So we can duplicate A₄ to obtain A₅ and we can invert A₄ to obtain A₆



Types of ROM

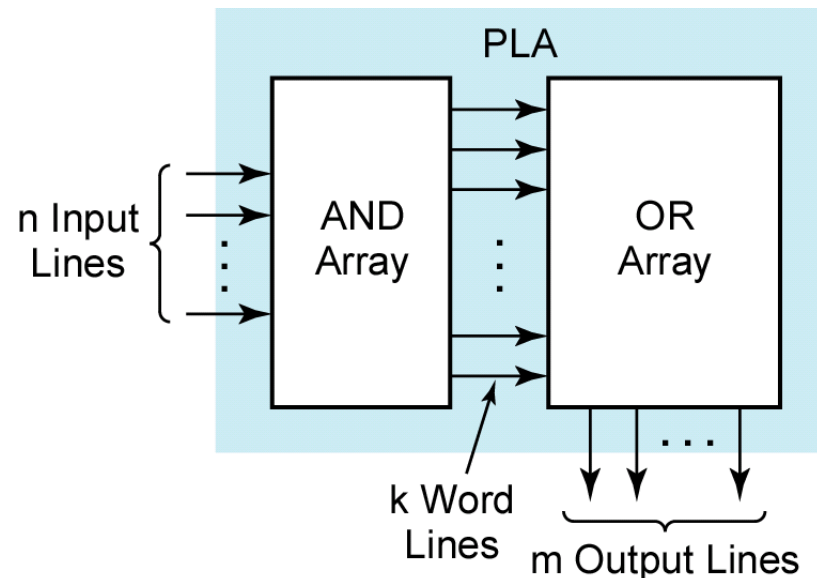
- Mask-programmable ROM
 - When the ROM is manufactured, the data in it is stored
 - The data cannot be changed
- Programmable ROM (PROM), also called Field Programmable ROM (FPRM)
 - When the ROM is manufactured, it is empty
 - It can be programmed only once
- Electrically Erasable Programmable ROM (EEPROM)
 - This ROM can be programmed, erased and reprogrammed
 - It can be erased and rewritten for a limited number of times (between 100 and 1000 times)
- Flash memory
 - There is no limit to how often we can erase it and write to it

Programmable Logic Devices (PLD)

- A PLD is an integrated circuit; a simple PLD can implement from 2 to 10 functions, each having from 4 to 16 variables
- Instead of using a large number of gates, a PLD can be used
- If the functions we are implementing change, we don't need to rewire all the system; we reprogram the PLD
- Types of PLD
 - Programmable Logic Arrays (PLA)
 - Programmable Array Logic (PAL)

Programmable Logic Array (PLA)

- A PLA with n input and m outputs can implement m functions with n variables each
 - (like a ROM)
- A PLA has an AND array and an OR array, shown in the figure
- We have a function that is written in SOP form
- Each product is realized in the AND array
- The products go in the OR array and are ORed together



PLA Structure

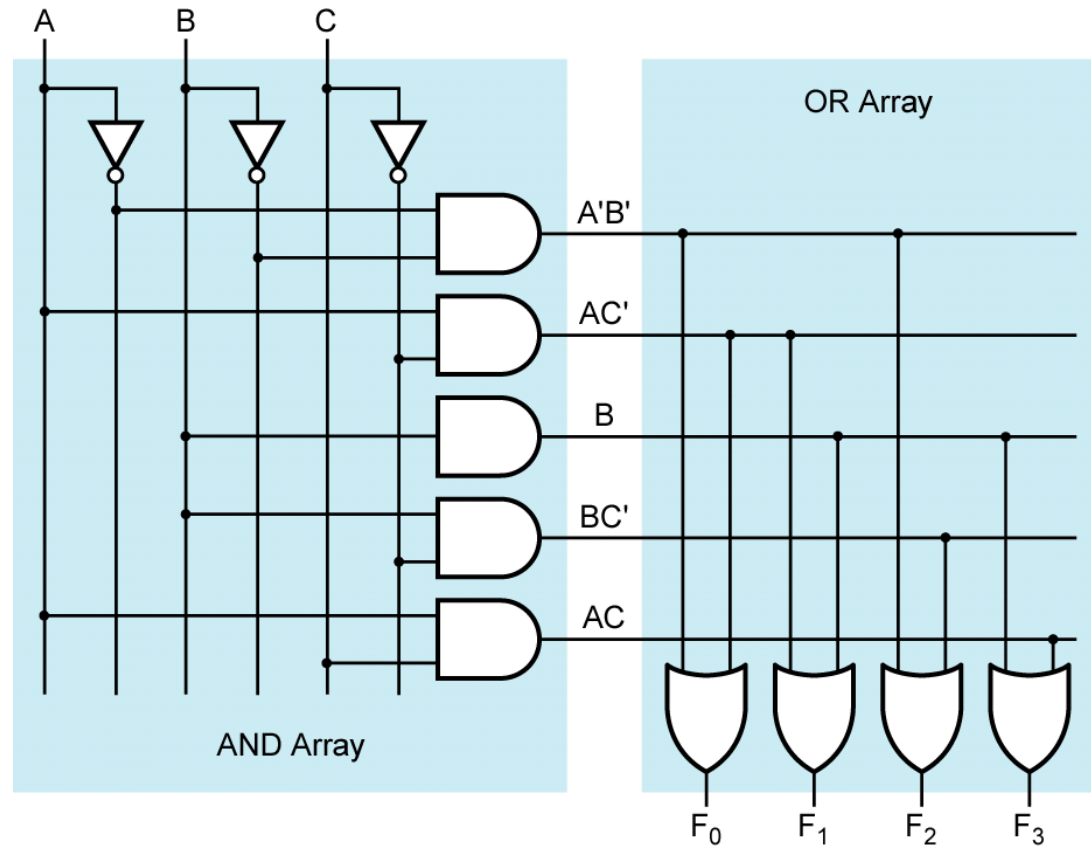
- We have the functions:

$$F_0 = A'B' + AC'$$

$$F_1 = AC' + B$$

$$F_2 = A'B' + BC'$$

$$F_3 = B + AC$$



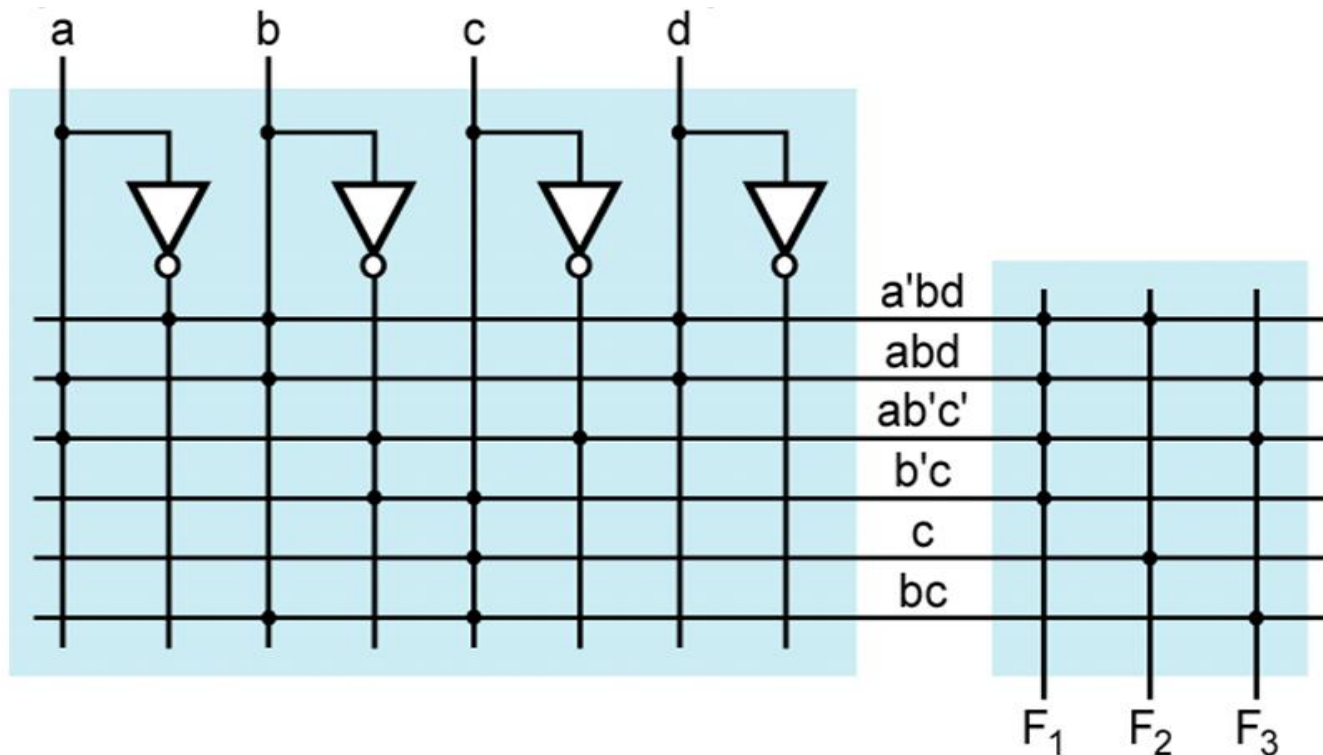
PLA Structure (another way)

- The figure shows another way to represent the structure of a PLA
- The functions are:

$$F_1 = a'bd + abd + ab'c' + b'c$$

$$F_2 = a'bd + c$$

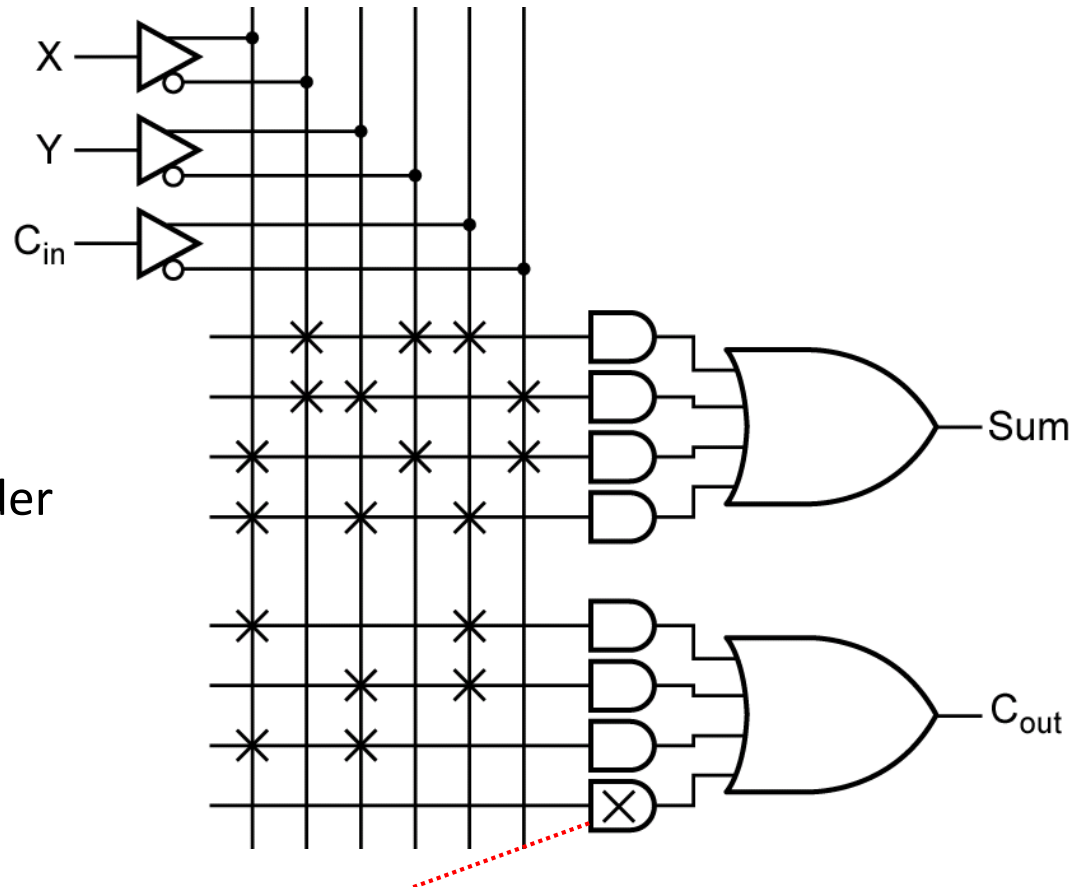
$$F_3 = abd + ab'c' + bc$$



Programmable Array Logic (PAL)

- It is a special case of PLA
- Similar to PLA, the PAL has an AND array and an OR array
- The AND array is programmable; the OR array is fixed

- The AND gate can take as input any variable or its complement; therefore, any product can be realized
- But, the OR gate takes 4 AND gates; therefore, there can be at most 4 products in the SOP expression



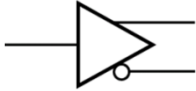
The PAL in the figure implements a 1-bit full adder

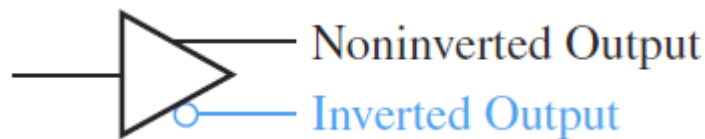
$$\text{Sum} = X'Y'C_{in} + X'Yc_{in}' + XY'C_{in}' + XYC_{in}$$

$$C_{out} = Xc_{in} + Yc_{in} + XY$$

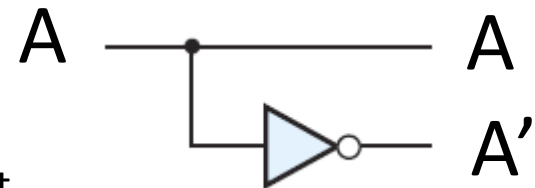
This AND gate is not used

Input Buffer

- In the previous figure, this symbol was used 
- It is an input buffer
- It keeps the voltage high enough; this is needed because in the PAL, many gates take the same input signal
- If the signal is logic 1 (usually around 5 Volts) and the voltage drops too much, it might be mistaken as logic 0
- The input buffer also provides the complement

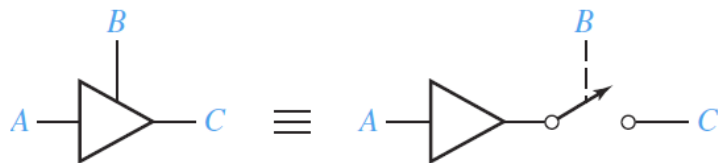


Logically Equivalent

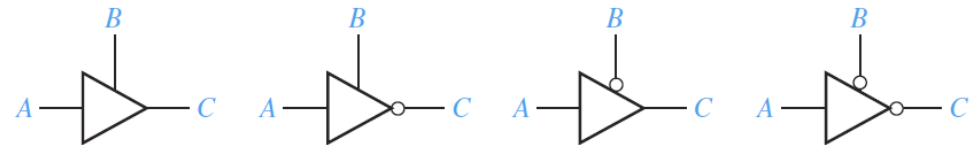


Three-State Buffers

- if one gate has 0 output(low voltage) and another has 1 output (high voltage)
 - When gates connected together- output voltage is intermediate value between(0-1)
 - Three-State(tri-state) buffer is the solution
 - Also used in IC :Bidirectional Input-output pins



Three State buffer



B	A	C
0	0	Z
0	1	Z
1	0	0
1	1	1

(a)

B	A	C
0	0	Z
0	1	Z
1	0	1
1	1	0

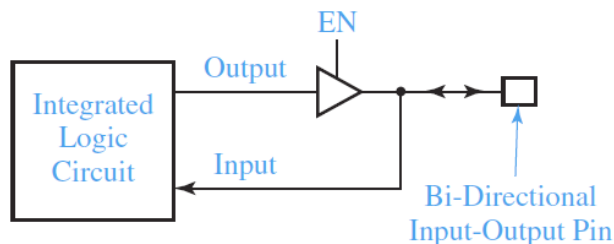
(b)

B	A	C
0	0	0
0	1	1
1	0	Z
1	1	Z

(c)

B	A	C
0	0	1
0	1	0
1	0	Z
1	1	Z

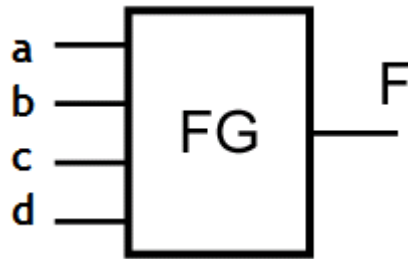
(d)



Four kinds of Three- State buffer
High-Z(High-impedance)

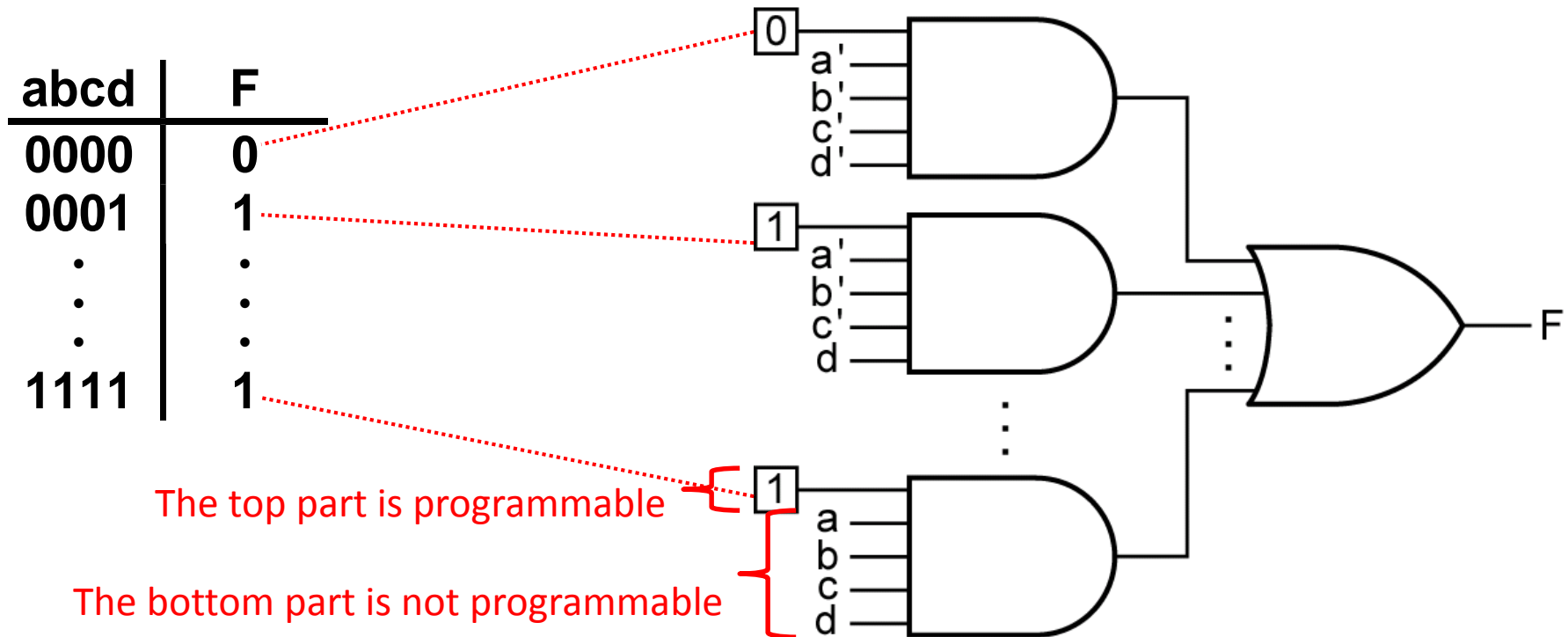
Function Generator

- A function generator gives the output of one function
- A function generator with inputs a,b,c,d has this symbol



Function Generator Implementation

- This is one possible implementation
- Using the truth table of the function F
- The output from the truth table is filled in the programmable part of the AND gate



Shannon's Expansion Theorem

- This theorem splits the function into two part based on one of its variables
- We have the function $f(a,b,c,d)$, we split it here into two parts based on its variable a
 - The two parts we get are: f_0 and f_1

$$\begin{aligned}f(a, b, c, d) &= a'f(0, b, c, d) + af(1, b, c, d) \\ &= a'f_0 + af_1\end{aligned}$$

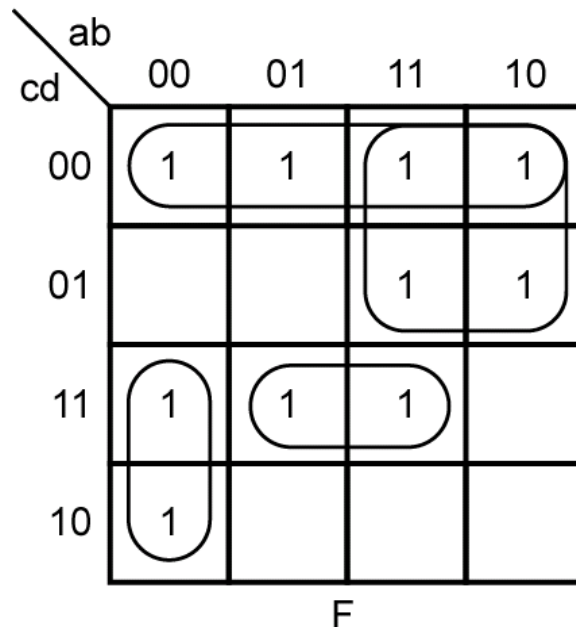
$$\begin{aligned}f(a, b, c, d) &= c'd' + a'b'c + bcd + ac' \\ &= a'(c'd' + b'c + bcd) + a(c'd' + bcd + c) \\ &= a'(c'd' + b'c + cd) + a(c' + bd) \\ &= a'f_0 + af_1\end{aligned}$$

$$f_0 = c'd' + b'c + cd$$

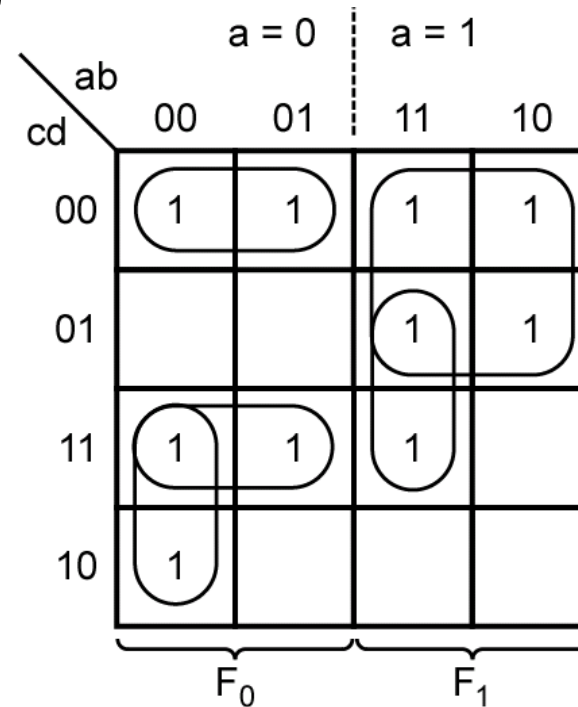
$$f_1 = c' + bd$$

Shannon's Expansion by K-map

- Draw the function on a K-map



$$F = c'd' + ac' + a'b'c + bcd$$



$$\begin{aligned} \text{Left side} &= a'c'd' + a'cd + a'b'c \\ &= a'(c'd' + cd + b'c) \end{aligned}$$

$$f_0 = c'd' + cd + b'c$$

$$\begin{aligned} \text{Right side} &= ac' + abd \\ &= a(c' + bd) \end{aligned}$$

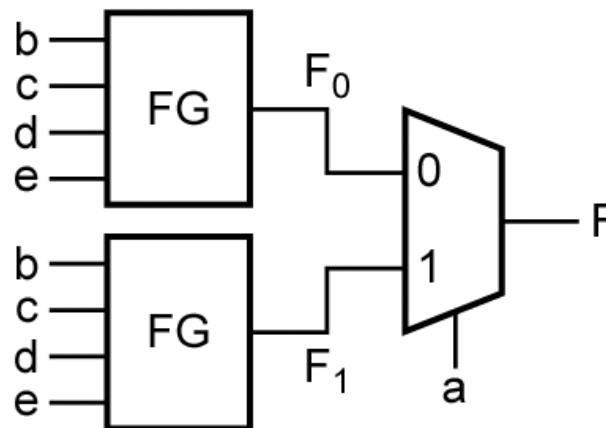
$$f_1 = c' + bd$$

Using Shannon's Expansion Theorem

- We have a function with five variables, $f(a,b,c,d,e)$
- We have function generators that take 4 inputs each
- We can implement f on multiple function generators by doing a Shannon's expansion

$$\begin{aligned} f(a, b, c, d, e) &= a' f(0, b, c, d, e) + a f(1, b, c, d, e) \\ &= a' f_0 + a f_1 \end{aligned}$$

- We got f_0 and f_1
- Multiplex f_0 and f_1



Using Shannon's Expansion

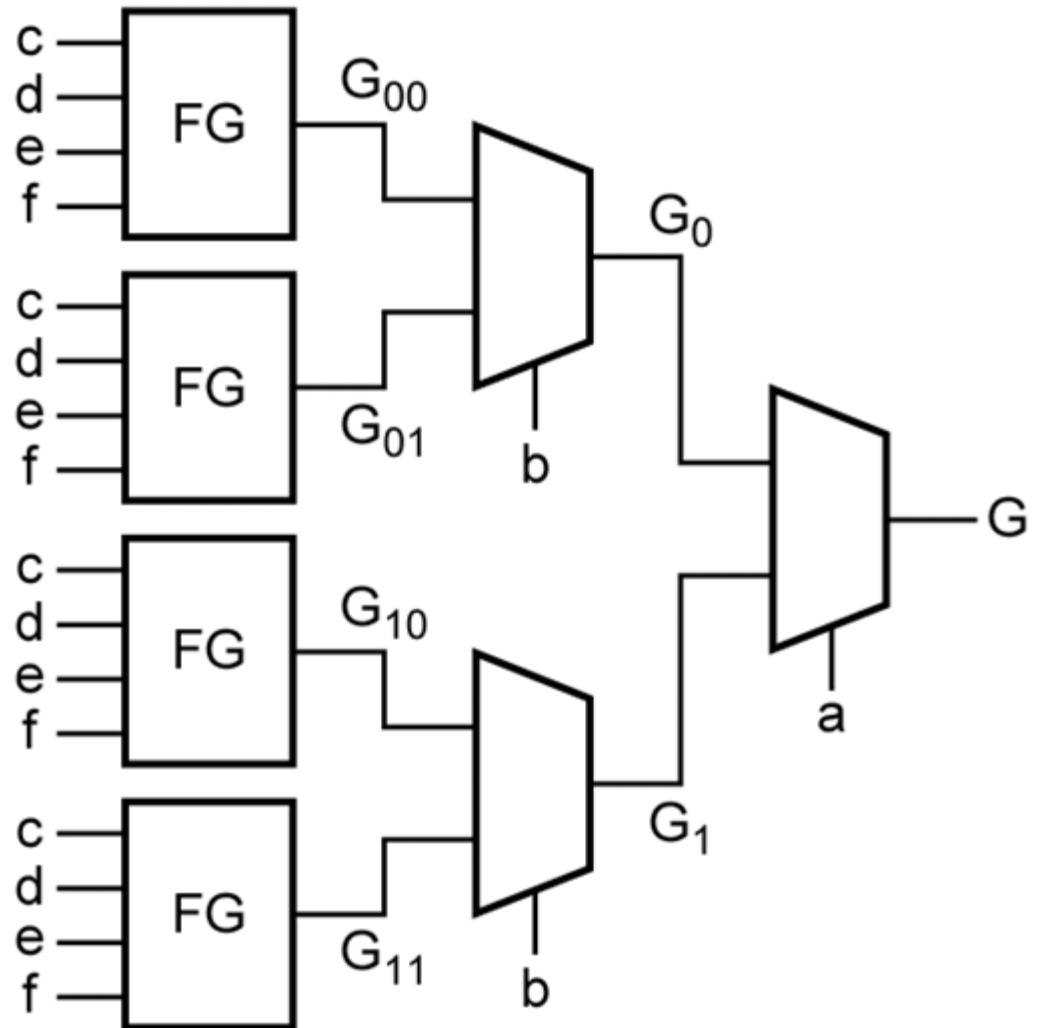
- We have a function with six variables, $G(a,b,c,d,e,f)$
- We have function generators with 4 inputs each
- First, do a Shannon's expansion on G over a , we get G_0 and G_1
- Then, do a Shannon's expansion on G_0 over b , we get G_{00} and G_{01}
- Then, do a Shannon's expansion on G_1 over b , we get G_{10} and G_{11}

$$G(a, b, c, d, e, f) = a'G_0 + aG_1$$

$$G_0 = b'G_{00} + bG_{01}$$

$$G_1 = b'G_{10} + bG_{11}$$

- G_{00} means $a=0, b=0$
- G_{01} means $a=0, b=1$
- G_{10} means $a=1, b=0$
- G_{11} means $a=1, b=1$
- Do a function generator for each of $G_{00}, G_{01}, G_{10}, G_{11}$
- Connect them using multiplexers



Function Generator using Truth Table

- Instead of using Shannon's expansion to fill the function generators, we can look at the truth table
- We have the function $F(A,B,C,D)$
- We have function generators with 3 inputs each
- We split F over A

After filling the function generators, they should be connected with multiplexers, like in previous slides.

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

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

This will go in the first function generator (F_0)

This will go in the second function generator (F_1)

Using Function Generator using Truth Table

- We have the function $F(A,B,C,D)$
- We have function generators with 2 inputs each
- We split F over A and B

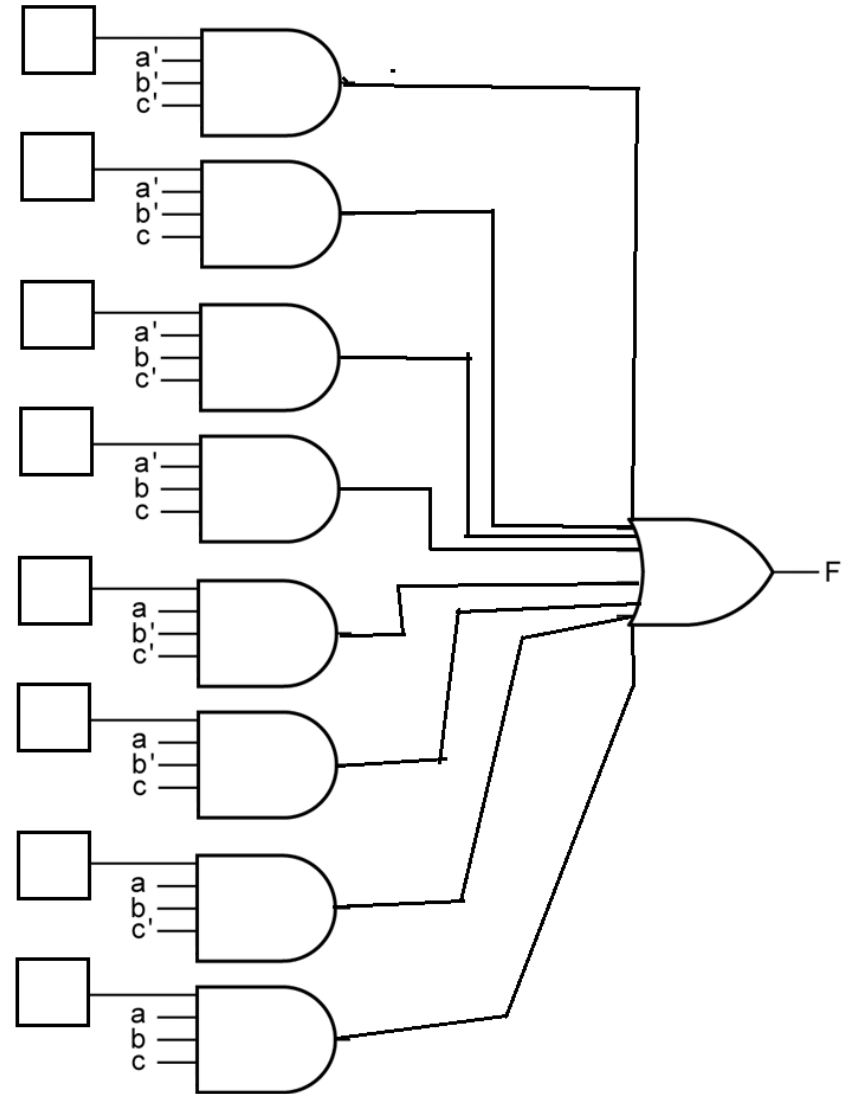
A	B	C	D	F
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

After filling the function generators, they should be connected with multiplexers, like in previous slides.

A	B	C	D	F	
0	0	0	0	0	This will go in the first function generator (F_{00})
0	0	0	1	1	
0	0	1	0	1	
0	0	1	1	0	
0	1	0	0	1	This will go in the second function generator (F_{01})
0	1	0	1	1	
0	1	1	0	0	
0	1	1	1	0	
1	0	0	0	0	This will go in the third function generator (F_{10})
1	0	0	1	0	
1	0	1	0	1	
1	0	1	1	1	
1	1	0	0	0	This will go in the fourth function generator (F_{11})
1	1	0	1	1	
1	1	1	0	0	
1	1	1	1	1	

Example

- Implement the function F in the function generator
- $F = a \text{ xor } b \text{ xor } c$



Example

- Implement the function F in the function generator
- $F = a \text{ xor } b \text{ xor } c$

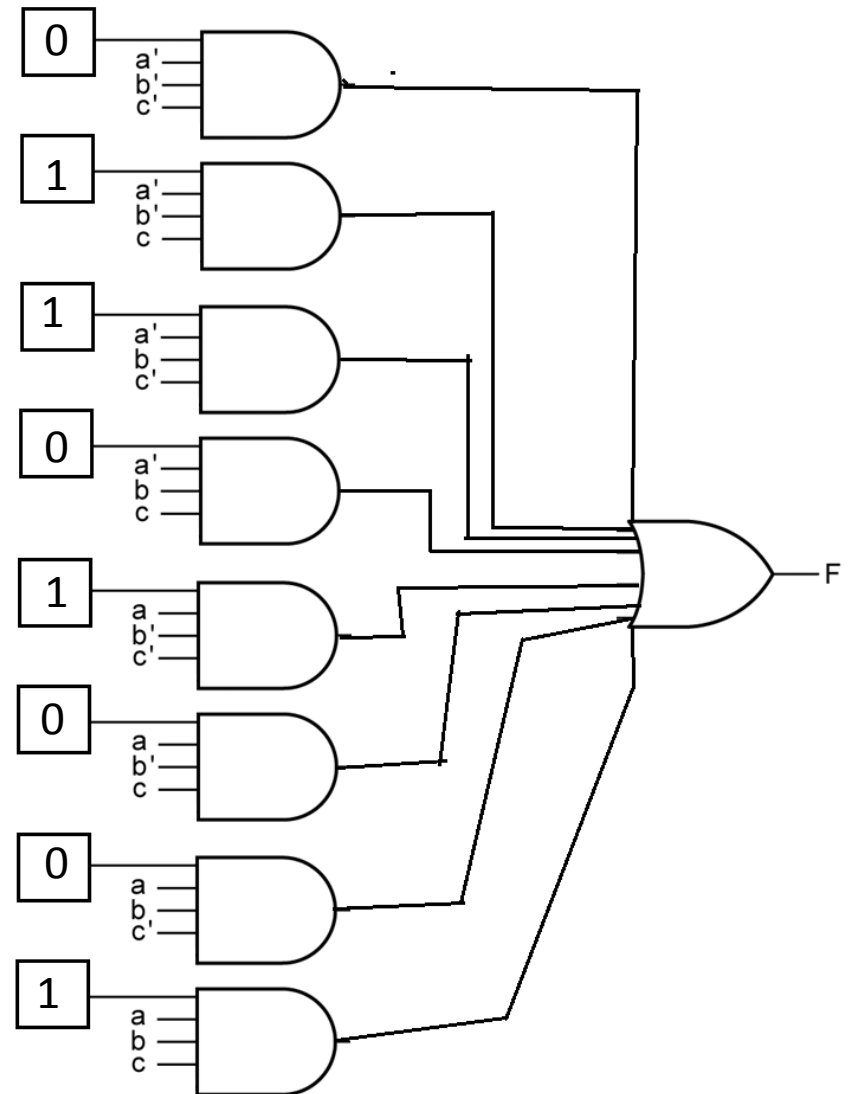
F=1 if the number of 1's is odd

F=0 if the number of 1's is even

Truth Table

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

Copy F from the truth table in the corresponding place in the function generator



Example

- We have the function: $F(a,b,c,d) = abc + a'bd + bc'd$
- Do a Shannon's expansion on a to find F_0 and F_1
 - Do it by algebra
 - Also, do it by K-map
 - Also, do it by truth table

$$F(a,b,c,d) = abc + a'bd + bc'd$$

$$= a'. F_0 + a.F_1$$

$$= a'. F(0,b,c,d) + a.F(1,b,c,d)$$

$$= a' (bd + bc'd) + a (bc + bc'd)$$

$$= a' [bd(1+c')] + a [b (c+c'd)]$$

$$= a' (bd) + a [b (c+c')(c+d)]$$

$$= a' (bd) + a (bc + bd)$$

So we find:

$$F_0 = bd$$

$$F_1 = bc + bd$$

Example

- We have the function: $F(a,b,c,d) = abc + a'bd + bc'd$
- Do a Shannon's expansion on a to find F_0 and F_1
 - Do it by algebra
 - Also, do it by K-map
 - Also, do it by truth table

		cd			
ab		00	01	11	10
00					
01			1	1	
11			1	1	1
10					

A red dashed horizontal line separates the top two rows (a=0) from the bottom two rows (a=1). Red ovals highlight the 1s in the top part (a=0) and the 1s in the bottom part (a=1).

Top part:
 $a'bd = a'(bd)$
 So, $F_0 = bd$

Bottom part:
 $abd + abc = a(bc + bd)$
 So, $F_1 = bc + bd$

Example

continued

- We have the function: $F(a,b,c,d) = abc + a'bd + bc'd$
- Do a Shannon's expansion on a to find F_0 and F_1
 - Do it by algebra
 - Also, do it by K-map
 - Also, do it by truth table

	abcd	F
a=0	0000	0
	0001	0
	0010	0
	0011	0
	0100	0
	0101	1
	0110	0
	0111	1
a=1	1000	0
	1001	0
	1010	0
	1011	0
	1100	0
	1101	1
	1110	1
	1111	1

cd \ b	0	1
00		
01		1
11		1
10		

The top part is F_0
 $F_0 = bd$

cd \ b	0	1
00		
01		1
11		1
10		1

The bottom part is F_1
 $F_1 = bc + bd$

Example

- We have the function: $F(a,b,c,d) = abc + a'bd + bc'd$
- We need to implement F with Function Generators (FG)
- We don't have an FG with 4 inputs, we only have FGs with 3 inputs

Same function as
previous slide

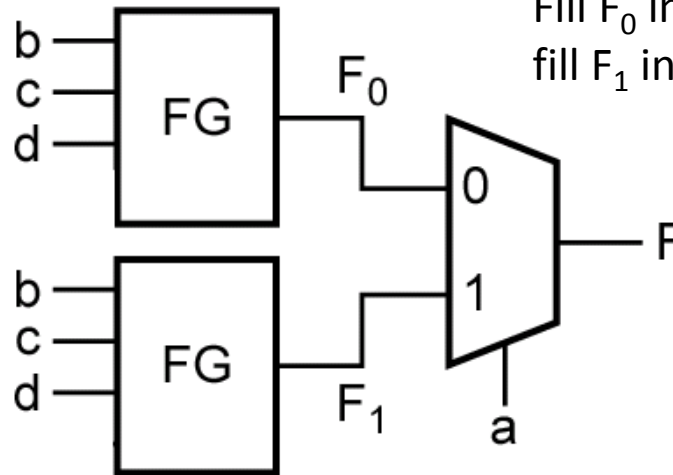
abcd	F
0000	0
0001	0
0010	0
0011	0
0100	0
0101	1
0110	0
0111	1
1000	0
1001	0
1010	0
1011	0
1100	0
1101	1
1110	1
1111	1

Write the top 8 values in the function generator

F_0

So we find F_0 and F_1 ; the Shannon's expansion of F over a .

Fill F_0 in the top FG and fill F_1 in the bottom FG.



Write the bottom 8 values in the function generator

Example

- We have the function $G(a,b,c,d) = a'b'c' + abd' + bc'd + acd$
- We need to implement G but we have only 2-input function generators
- We need to do Shannon's expansion over a and then over b

$$\begin{aligned} G(a,b,c,d) &= a'.G(0,b,c,d) + a.G(1,b,c,d) \\ &= a'(b'c' + bc'd) + a(bd' + bc'd + cd) \\ &= a'(b'c' + c'd) + a(b + cd) \end{aligned}$$

$$\begin{aligned} &b'c' + bc'd \\ &= c' (b' + bd) \\ &= c' (b' + d) \\ &= b'c' + c'd \end{aligned}$$

$$\begin{aligned} &bd' + bc'd + cd \\ &= bd' + d (bc' + c) \\ &= bd' + d (b+c) \\ &= bd' + bd + cd \\ &= b + cd \end{aligned}$$

$$G_0 = b'c' + c'd$$

$$G_1 = b + cd$$

$$\begin{aligned} G_0(b,c,d) &= b'.G_0(0,c,d) + b.G_0(1,c,d) \\ &= b'(c' + c'd) + b(c'd) \\ &= b'(c') + b(c'd) \end{aligned}$$

$$G_{00} = c'$$

$$G_{01} = c'd$$

$$\begin{aligned} G_1(b,c,d) &= b'.G_1(0,c,d) + b.G_1(1,c,d) \\ &= b'(cd) + b(1) \end{aligned}$$

$$G_{10} = cd$$

$$G_{11} = 1$$

We found:

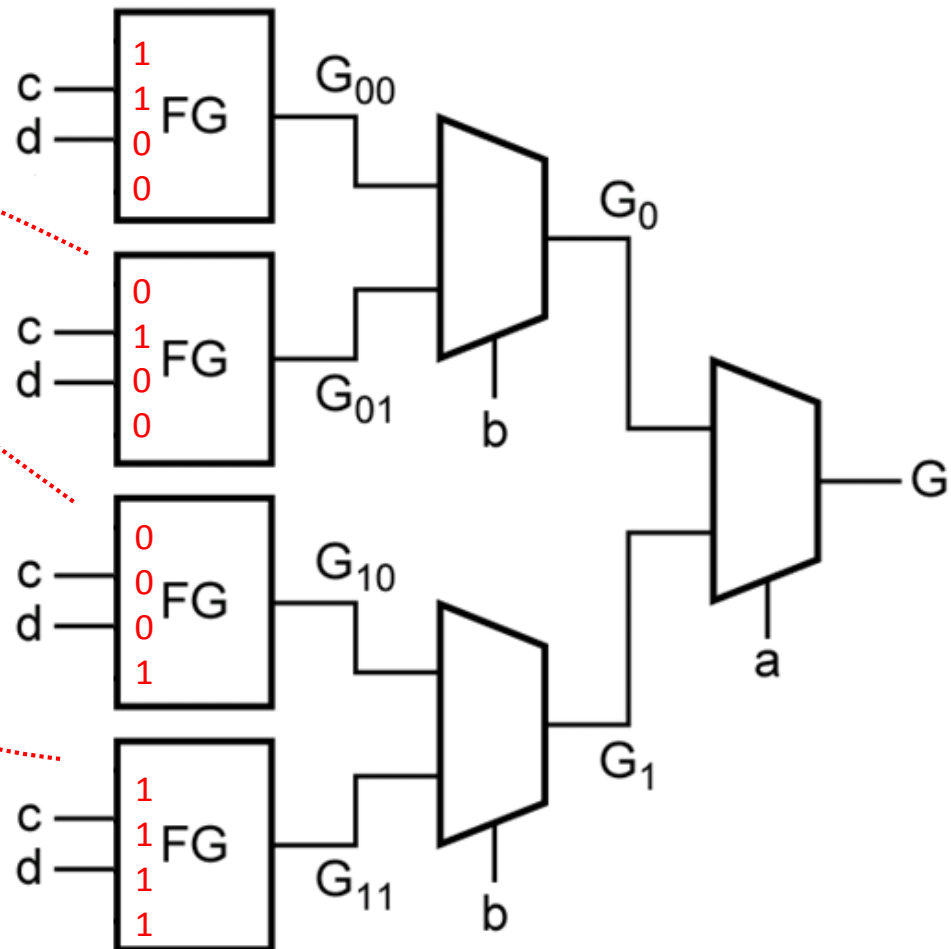
$$G_{00}(c,d) = c'$$

$$G_{01}(c,d) = c'd$$

$$G_{10}(c,d) = cd$$

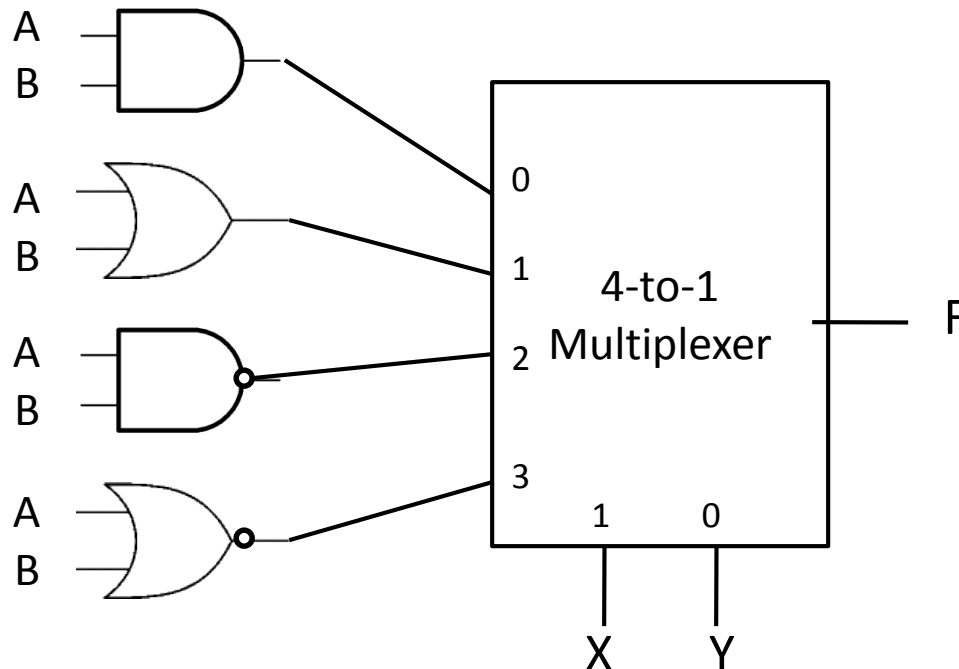
$$G_{11}(c,d) = 1$$

This is the configuration of
the logic circuit



Example

- We need to implement a multigate that will do AND, OR, NAND and NOR
- The inputs are A and B (1 bit each)
- The operation selectors are X and Y
- Implement the multigate using 4 gates and a multiplexer



If $XY=00$, $F=A.B$

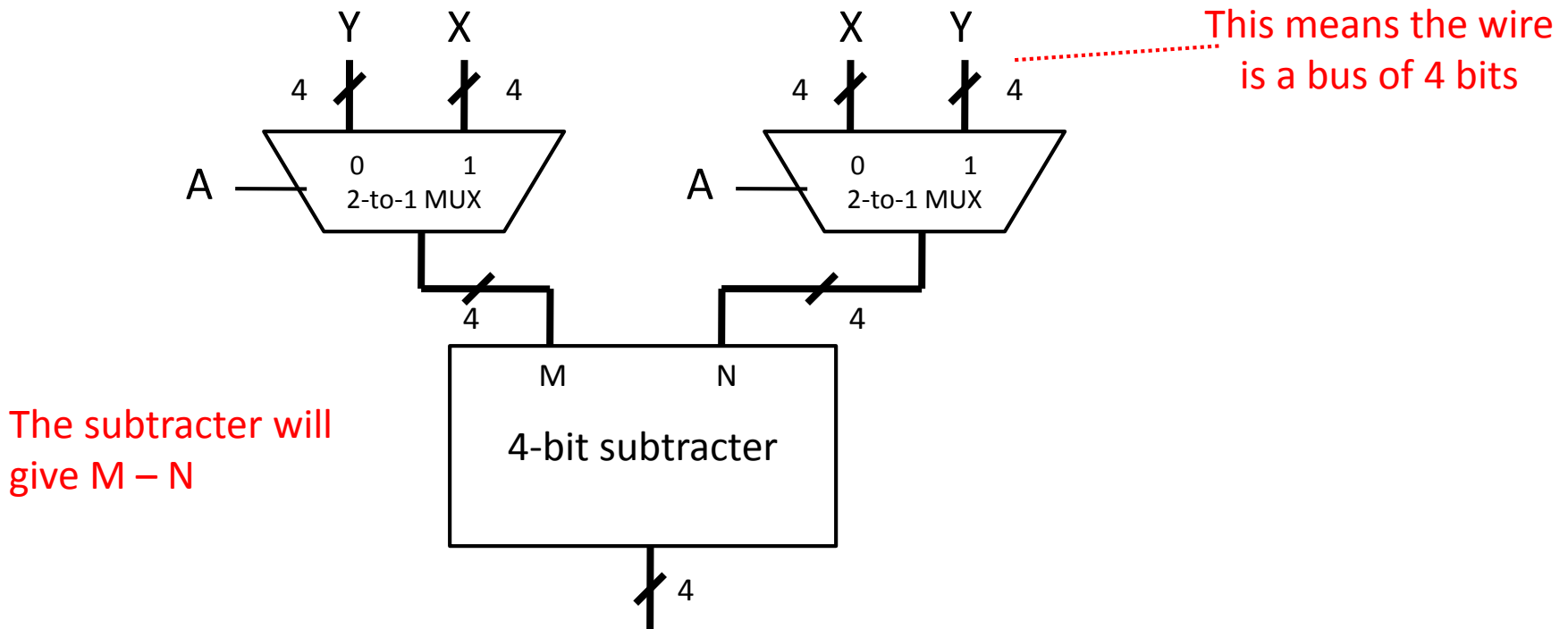
If $XY=01$, $F=A+B$

If $XY=10$, $F=A \text{ nand } B$

If $XY=11$, $F=A \text{ nor } B$

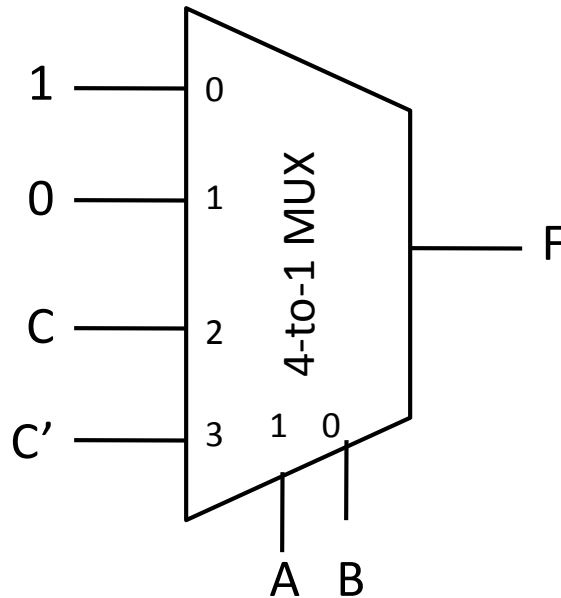
Example

- Design a circuit that will either subtract $X-Y$ or $Y-X$ depending on the value of A (each of X and Y is 4 bits)
- If $A=1$, the output should be $X-Y$; if $A=0$, the output should be $Y-X$
- Use one 4-bit subtracter and two 2-to-1 MUXes with bus input



Example

- We have the function $F(A, B, C)$
- If we had an 8-to-1 multiplexer, we can put the truth table result of F (8 lines) in the 8 inputs of the multiplexer
- However, we have a 4-to-1 multiplexer



Truth table

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

Red annotations on the left of the table indicate the function value for each AB combination:

- $AB=00; F=1$ (for rows 000 and 001)
- $AB=01; F=0$ (for rows 010 and 011)
- $AB=10; F=C$ (for rows 100 and 101)
- $AB=11; F=C'$ (for rows 110 and 111)

Example of Implementing Boolean Functions Using a 4:1 MUX

Implement $F(A,B,C) = \Sigma(,,)$ using one 4:1 MUX, use ____ and ____ for select lines.

Truth table

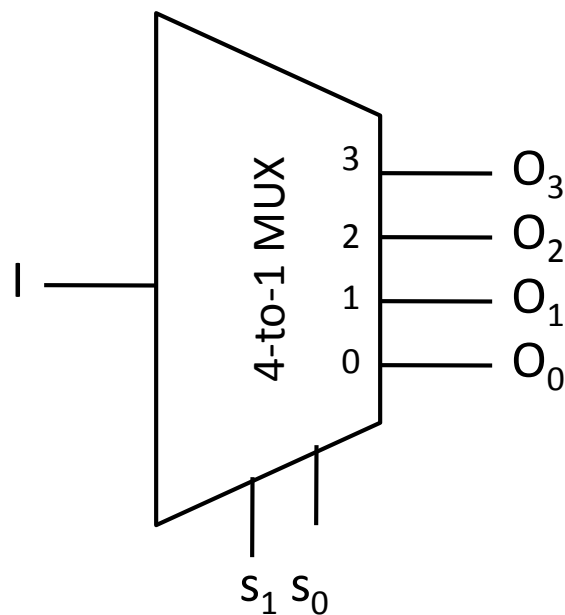
A B C	F
0 0 0	1
0 0 1	1
0 1 0	0
0 1 1	0
1 0 0	0
1 0 1	1
1 1 0	1
1 1 1	0

Implementation table

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

Demultiplexer

- Also abbreviated as DMUX or DEMUX
- It does the inverse function of a multiplexer
- There is 1 input, n selectors and 2^n outputs
- The input will go to the output designated by the selectors
- The other outputs are zero

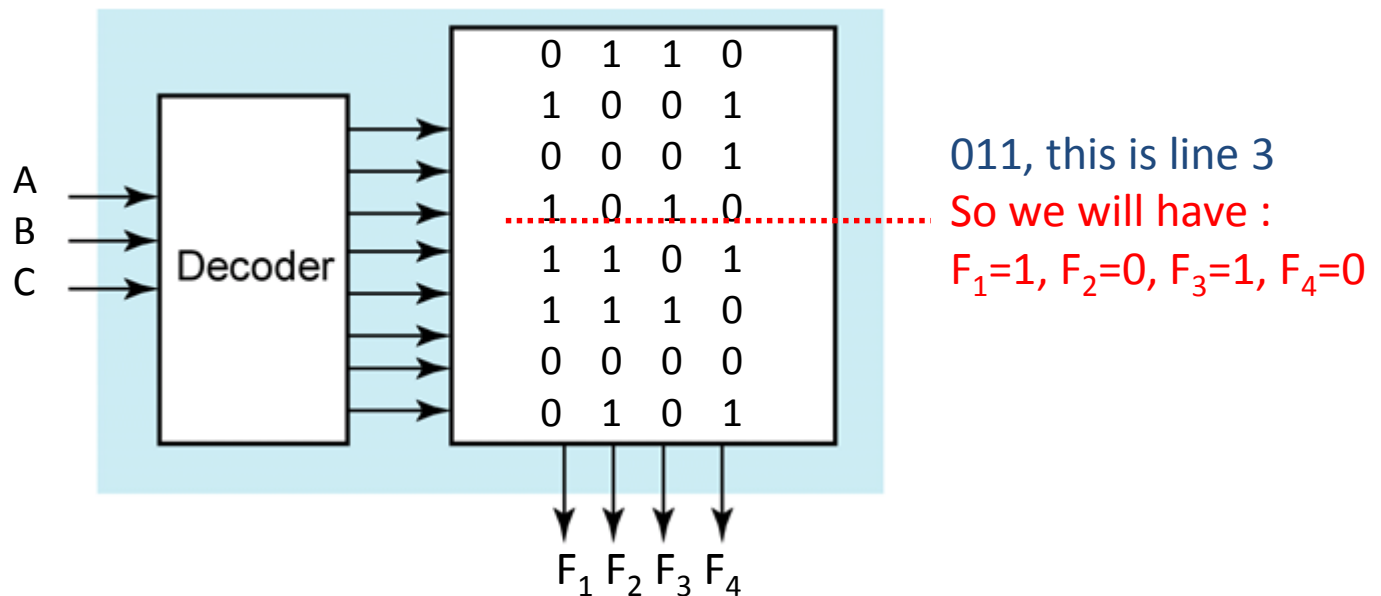


Truth table

s_1	s_0	I	O_3	O_2	O_1	O_0
0	0	0	0	0	0	0
0	0	1	0	0	0	1
0	1	0	0	0	0	0
0	1	1	0	0	1	0
1	0	0	0	0	0	0
1	0	1	0	0	1	0
1	1	0	0	0	0	0
1	1	1	1	0	0	0

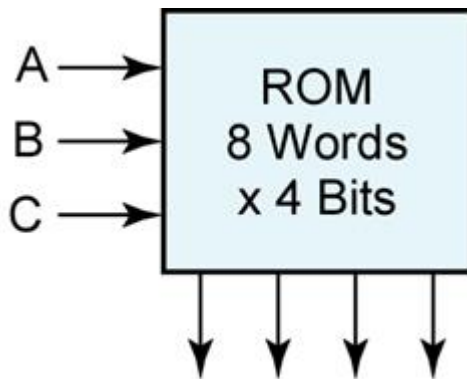
Example

- The data stored in this ROM implements the four functions
- If A=0, B=1, C=1, what will be the value of F_1, F_2, F_3, F_4 ?
- Give the minterm expressions of F_1 and F_2



Example

- We have the ROM in the figure below
- It has 3 bits for the address
- It's able to store 8 words
- Each word is 4 bits
- What is the total number of bits that can be stored in this ROM?



Total number of bits:

$$8 \text{ word} * 4 \text{ bits/word} = 32 \text{ bits}$$

Example

- What size ROM is required to implement 4 functions of 5 variables?
- What is the total number of bits in this ROM?

There are 5 variables, we need 2^5 words = 32 words

Each word should be 4 bits

The total number of bits is:
 $32 \text{ words} * 4 \text{ bits/word} = 128 \text{ bits}$

- What size ROM is required to implement 8 functions of 10 variables?
- What is the total number of bits in this ROM?

There are 10 variables, we need 2^{10} words = 1024 words

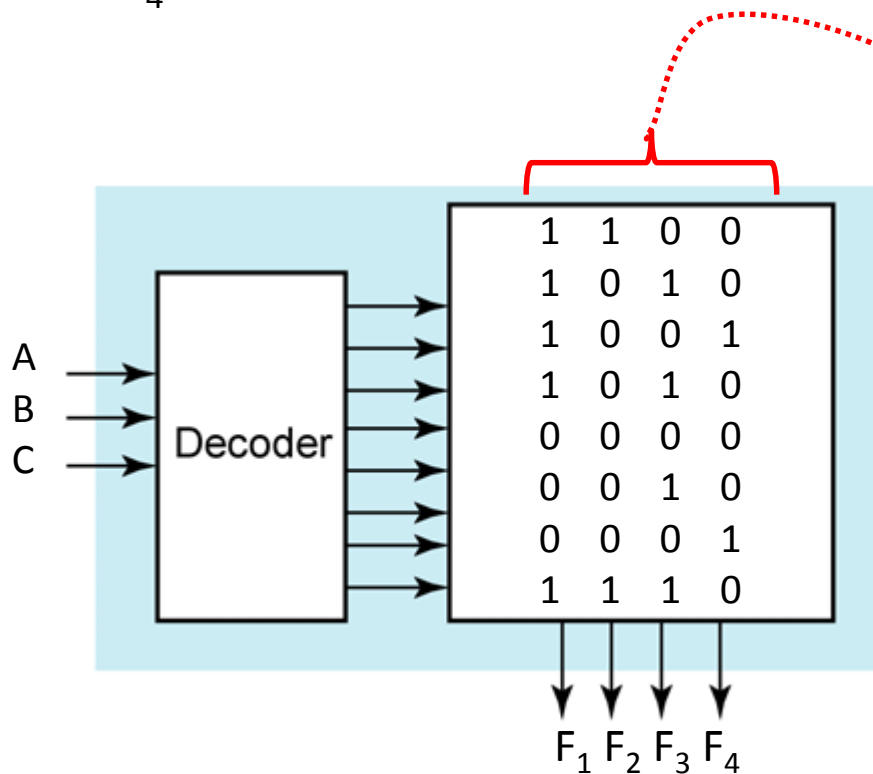
Each word should be 8 bits

The total number of bits is:
 $1024 * 8 \text{ bits} = 1 \text{ KiloByte}$

8 bits = 1 byte
 $2^{10} = 1 \text{ kilo}$

Example

- Implement the following functions in a ROM
- $F_1 = A' + BC$
- $F_2 = A'B'C' + ABC$
- $F_3 = C$
- $F_4 = BC'$



Truth Table

A	B	C	F ₁	F ₂	F ₃	F ₄
0	0	0	1	1	0	0
0	0	1	1	0	1	0
0	1	0	1	0	0	1
0	1	1	1	0	1	0
1	0	0	0	0	0	0
1	0	1	0	0	1	0
1	1	0	0	0	0	1
1	1	1	1	1	1	0

Take all the values from the truth table and copy them into the ROM

Example

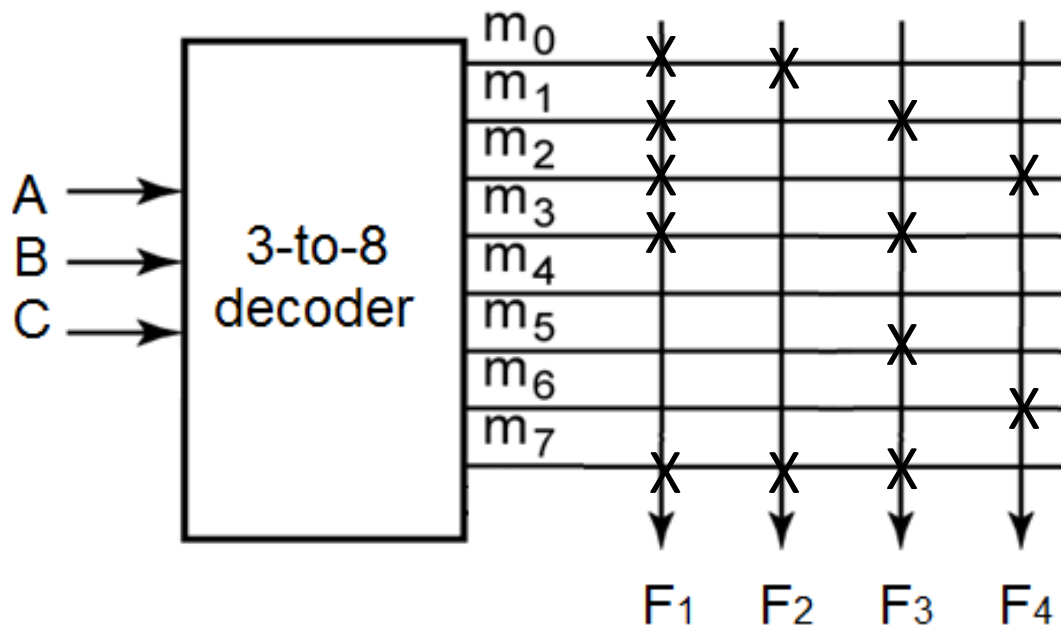
- Implement the following functions in a ROM using the ROM structure diagram

- $F_1 = A' + BC$
- $F_2 = A'B'C' + ABC$
- $F_3 = C$
- $F_4 = BC'$

Place an X on the line intersection to select a minterm

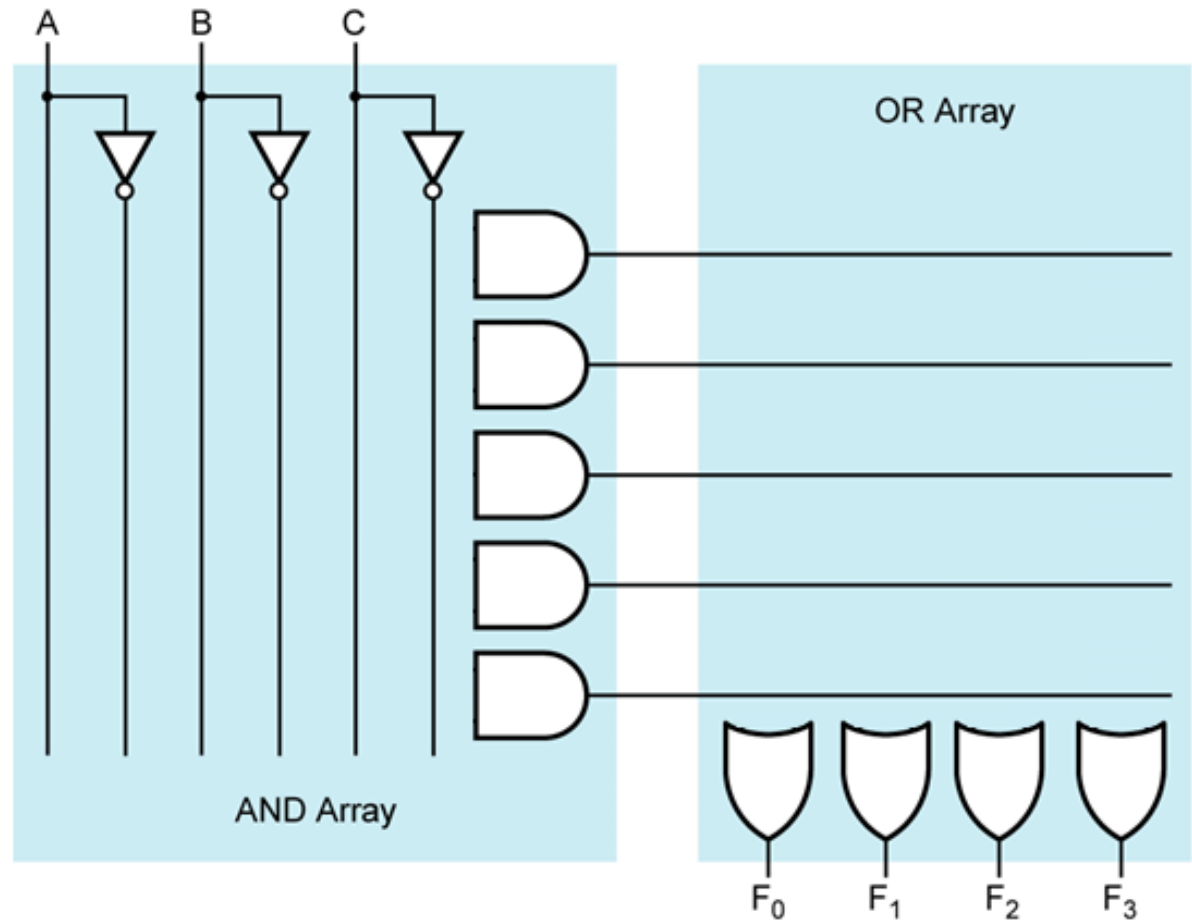
Truth Table

A	B	C	F_1	F_2	F_3	F_4
0	0	0	1	1	0	0
0	0	1	1	0	1	0
0	1	0	1	0	0	1
0	1	1	1	0	1	0
1	0	0	0	0	0	0
1	0	1	0	0	1	0
1	1	0	0	0	0	1
1	1	1	1	1	1	0



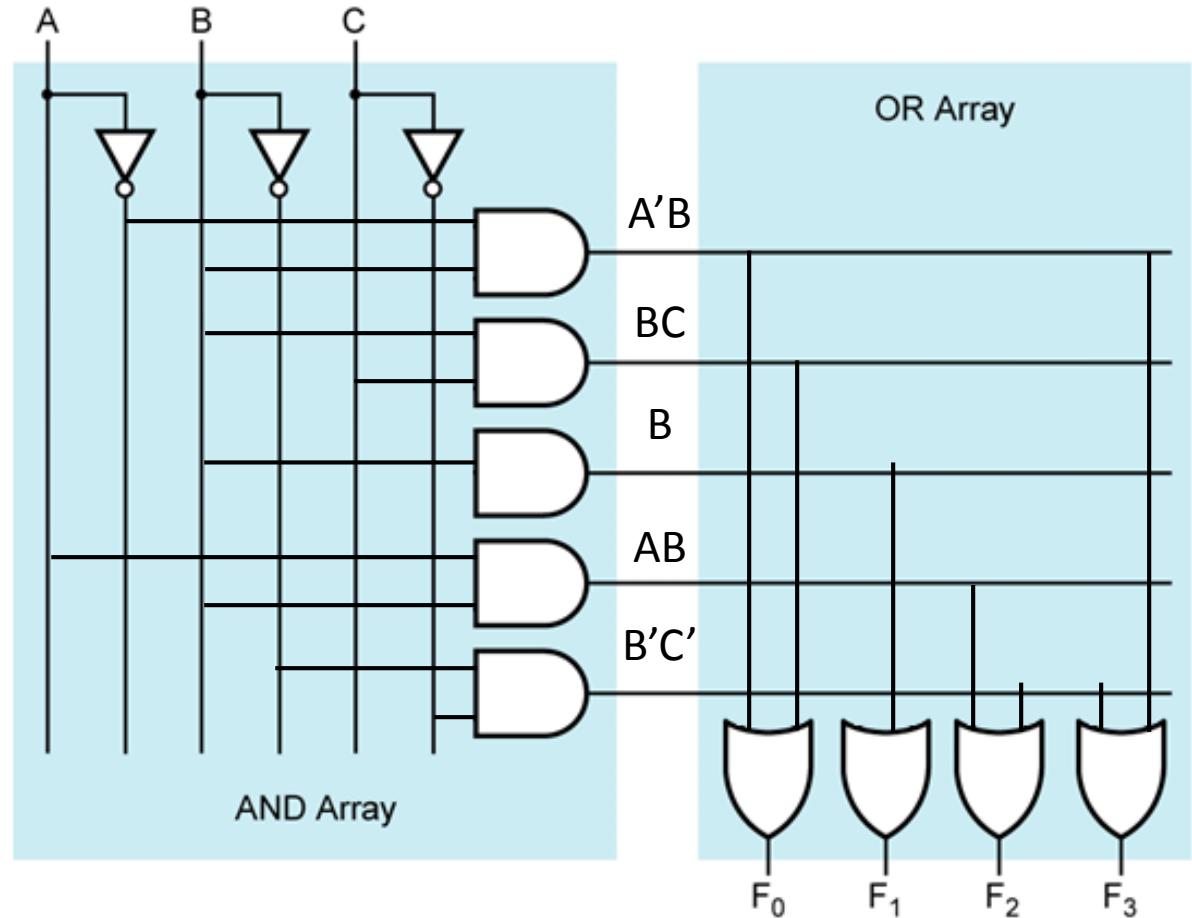
Example

- Implement the following functions by drawing the lines in the Programmable Logic Array (PLA)
- $F_0 = A'B + BC$
- $F_1 = B + AB$
- $F_2 = AB + B'C'$
- $F_3 = B'C' + A'B$



Example

- Implement the following functions by drawing the lines in the PLA
- $F_0 = A'B + BC$
- $F_1 = B$
- $F_2 = AB + B'C'$
- $F_3 = B'C' + A'B$



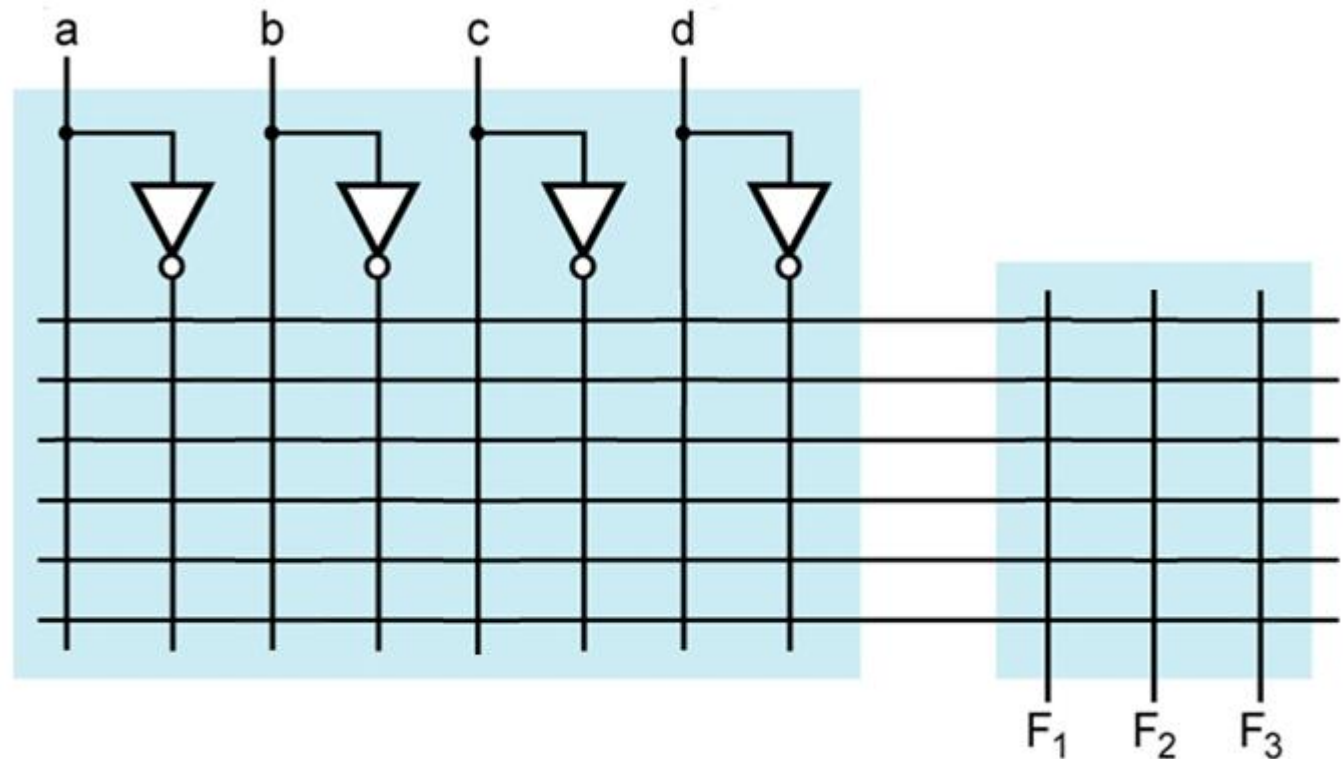
The AND gates make the products that we need

We can reuse the products

Or the products together

Example

- Implement the following functions in the PLA by selecting the needed intersections
- $F_1 = A'B + BC$
- $F_2 = AB + B'C'$
- $F_3 = B'C' + A'B$

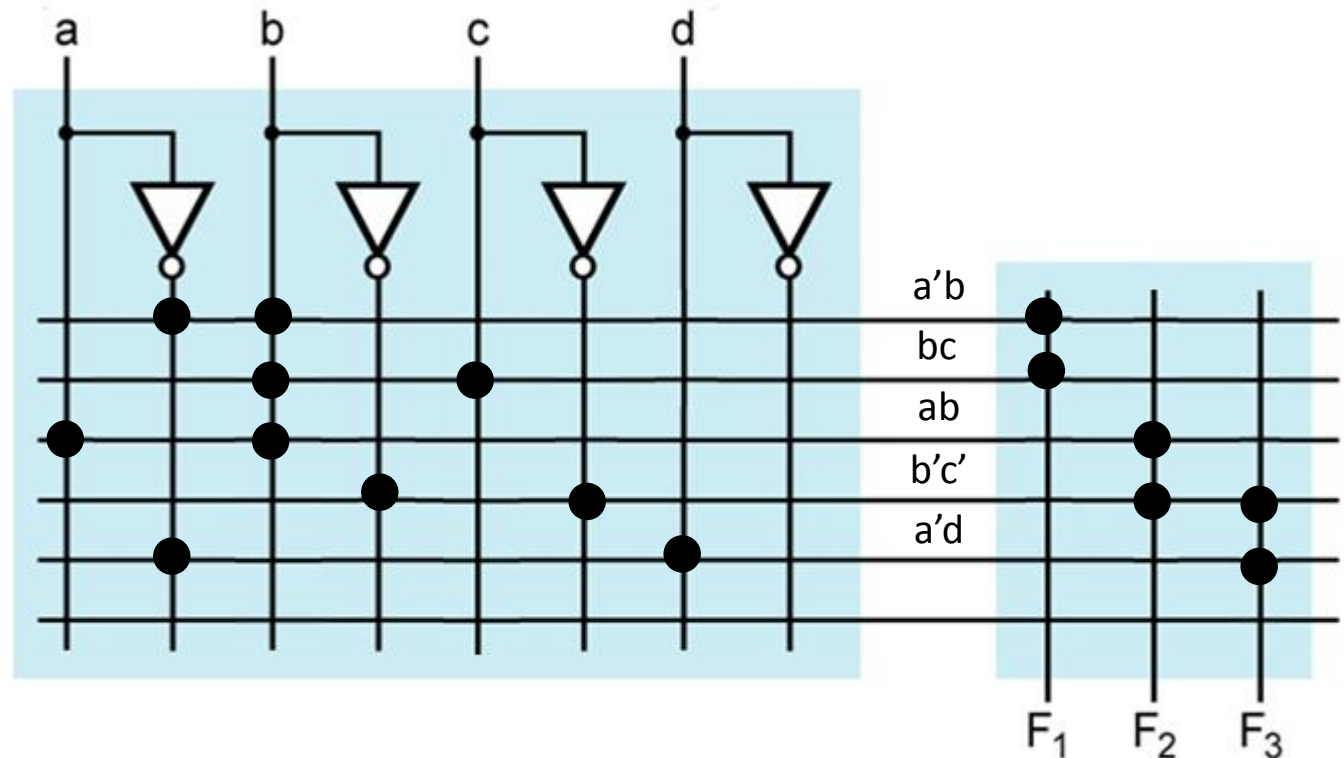


Example

- Implement the following functions by selecting the needed intersections
- $F_1 = a'b + bc$
- $F_2 = ab + b'c'$
- $F_3 = b'c' + a'd$

The horizontal line makes a product

One or more product are selected by the vertical line of the function

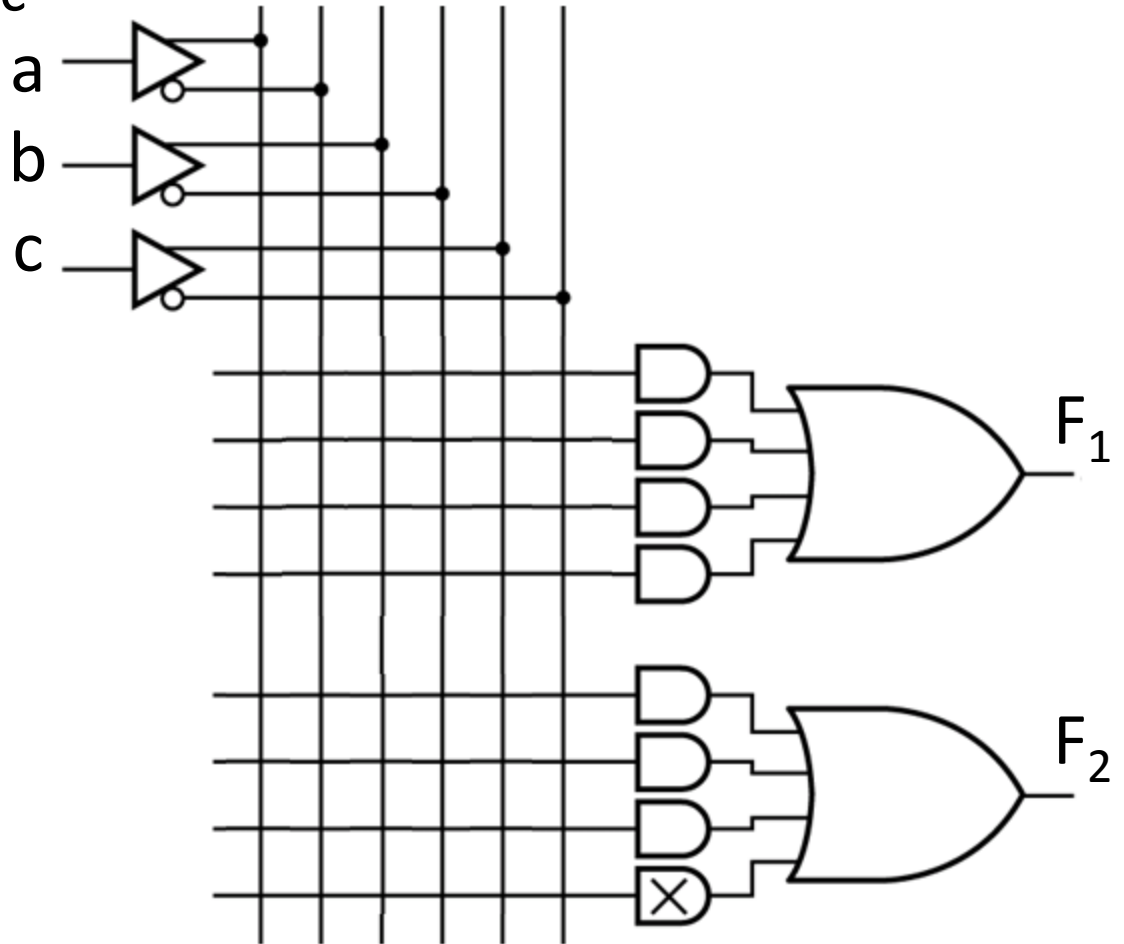


Example

- Implement these functions on the Programmable Array Logic (PAL)

- $F_1 = ab' + a'b + bc' + b'c$

- $F_2 = ab + bc + a'b'$

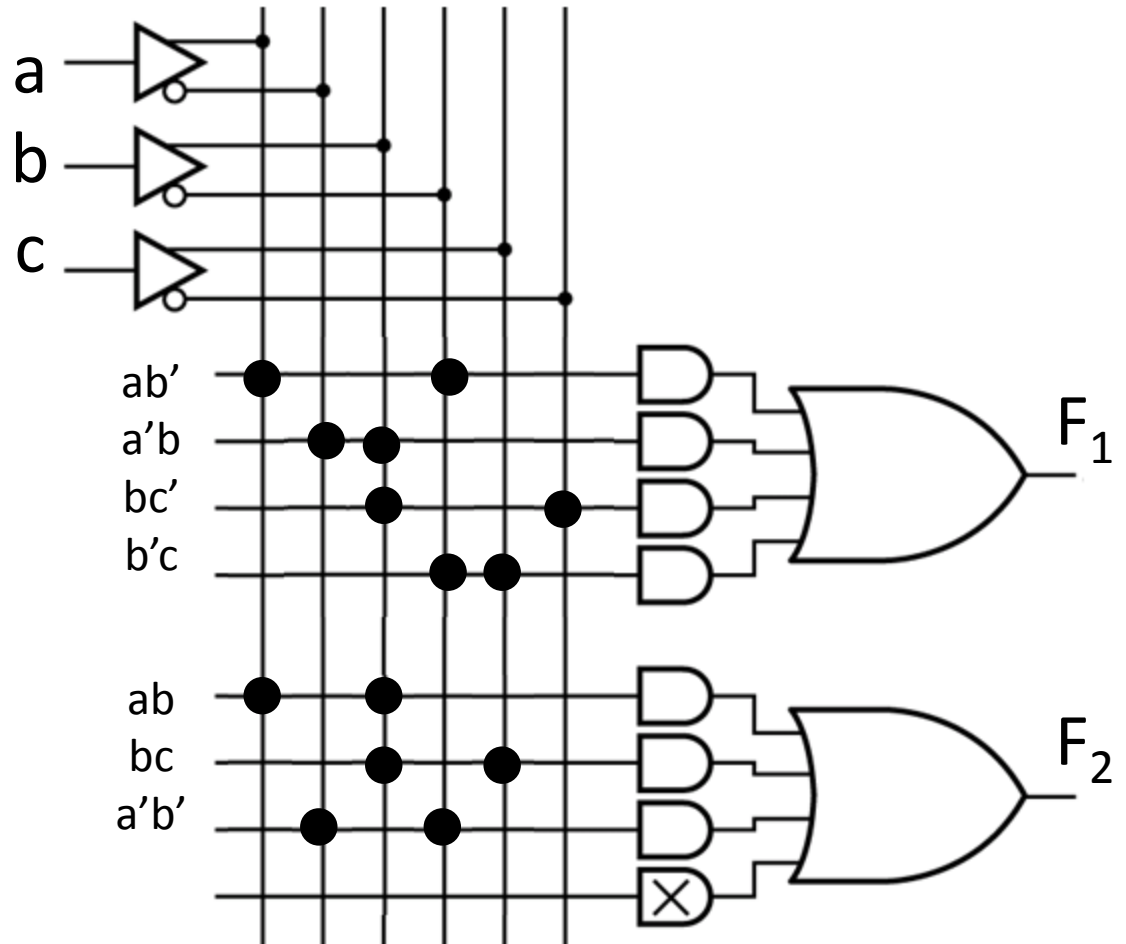


Example

- Implement these functions on the PAL
- $F_1 = ab' + a'b + bc' + b'c$
- $F_2 = ab + bc + a'b'$

The horizontal line
makes a product

One or more product
are selected by the
vertical line of the
function



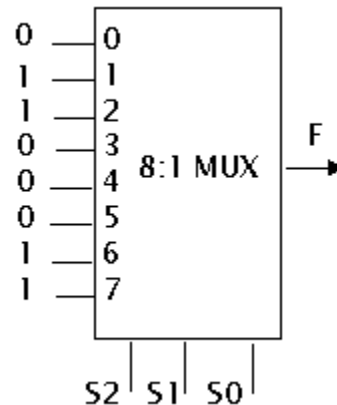
Boolean function Implementation

Using 2^n -to-1 multiplexer

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.

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

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 $I_1 = I_2 = I_6 = I_7 = 1$.



Boolean function Implementation

Using $2^{(n-1)}$ -to-1 multiplexer

- Assume Boolean function has n variables;
- Choose $n-1$ variables to be the selection lines for the $n:1$ MUX;
- The remaining single variable (let us say, the least Significant bit, z) is used for data input.
- Data input lines (MUX inputs) can take on one of the following values

$z, z', 1$ or 0

- Construct “the implementation table to find out which value should be assigned to each input of the MUX.
- Another way of determining the values.
 - From truth table we have to find the relation of F and z to be able to design input lines. For example : $f(x,y,z) = \sum(1,2,6,7)$

Example of Implementing Boolean Functions Using a 4:1 MUX

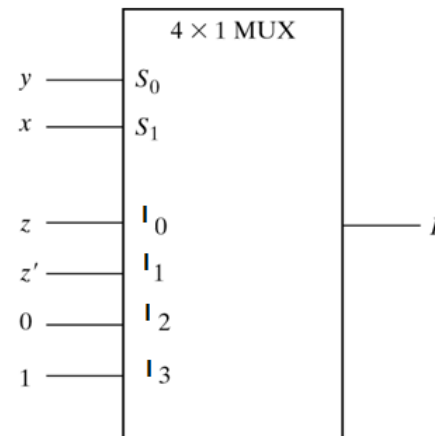
Implement $F(x,y,z) = \sum(1,2,6,7)$ using one 4:1 MUX, use x and y for select lines.

Implementation table

	I_0	I_1	I_2	I_3
z'	0	2	4	6
z	1	3	5	7
	z	z'	0	1

Truth table

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	



Example of Implementing Boolean Functions Using MUX

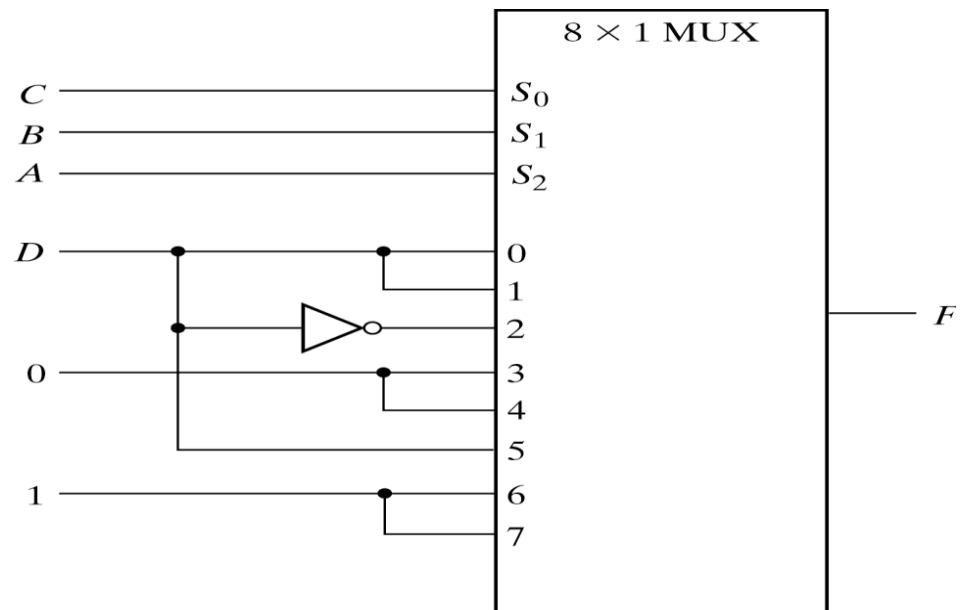
Implement $F(A,B,C,D) = \sum (1,3,4,11,12,13,14,15)$ using one 8:1 MUX

Choose A, B, and C for select lines.

Implementation table

	I ₀	I ₁	I ₂	I ₃	I ₄	I ₅	I ₆	I ₇
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

A	B	C	D	F	
0	0	0	0	0	
0	0	0	1	1	$F = D$
0	0	1	0	0	
0	0	1	1	1	$F = D$
0	1	0	0	1	
0	1	0	1	0	$F = D'$
0	1	1	0	0	
0	1	1	1	0	$F = 0$
1	0	0	0	0	
1	0	0	1	0	$F = 0$
1	0	1	0	0	
1	0	1	1	1	$F = D$
1	1	0	0	1	
1	1	0	1	1	$F = 1$
1	1	1	0	1	
1	1	1	1	1	$F = 1$



- **Problem 1:**
 - Realize a full adder using a 3-to-8 line decoder WITH INVERTED OUTPUT.
- **Problem 2:**
 - Use 2-to-1 multiplexers with active high output and active high enable to implement a 4-to-1 multiplexer with active high output and no enable line.
- **Problem 3:**
 - Use an 8-to-1 MUX and minimum number of external gates to realize the function
$$F(w, x, y, z) = \sum(3, 4, 5, 7, 10, 14)$$