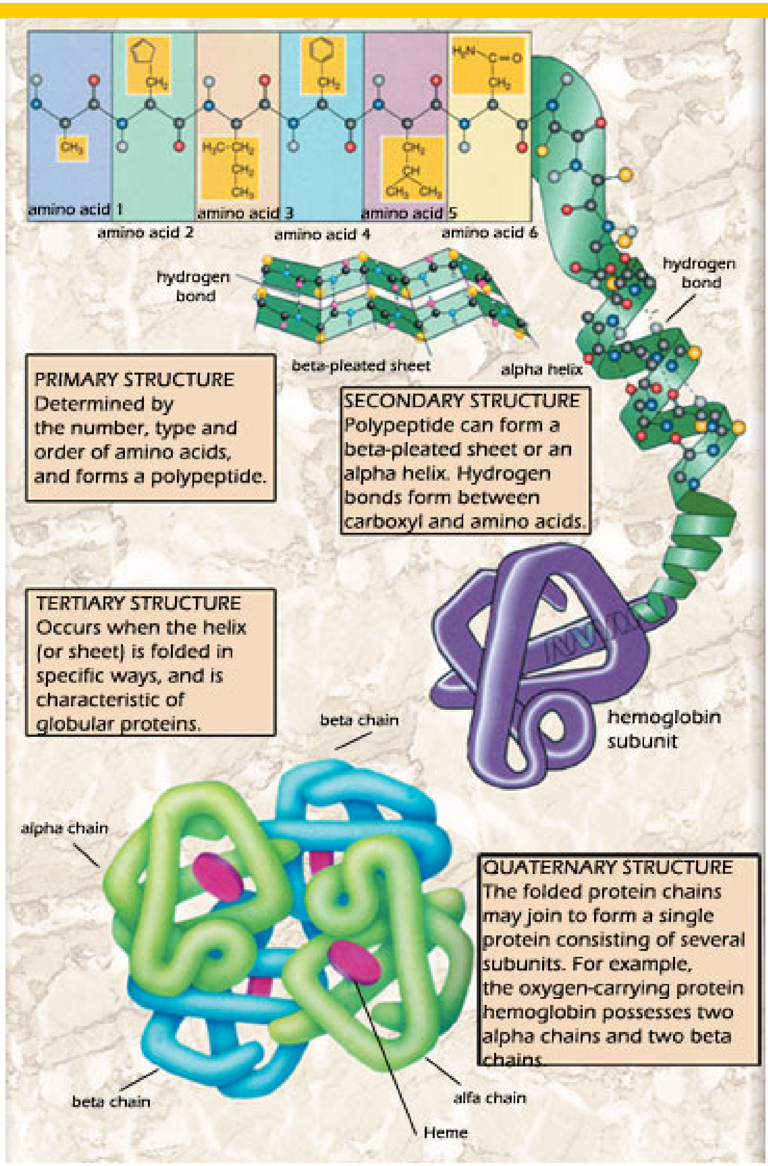
Switching Circuits & Logic Design

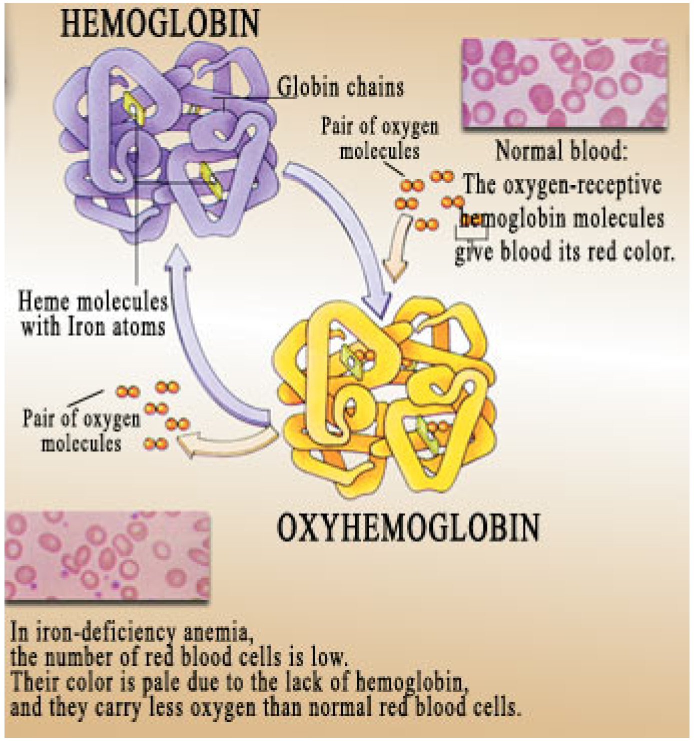
Jie-Hong Roland Jiang
江介宏

Department of Electrical Engineering
National Taiwan University

Fall 2012

§9 Multiplexers, Decoders, and Programmable Logic Devices





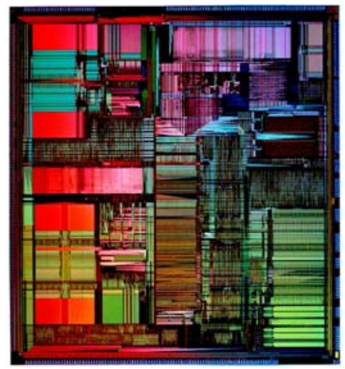
www.miracleofthebloodandheart.com

Outline

- Introduction
- Multiplexers
- Three-state buffers
- Decoders and encoders
- Read-only memories
- Programmable logic devices
- Decomposition of switching functions

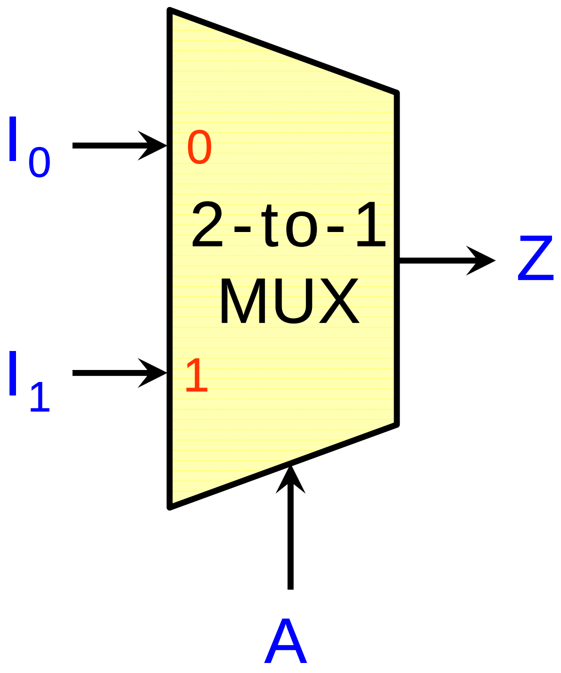
Introduction

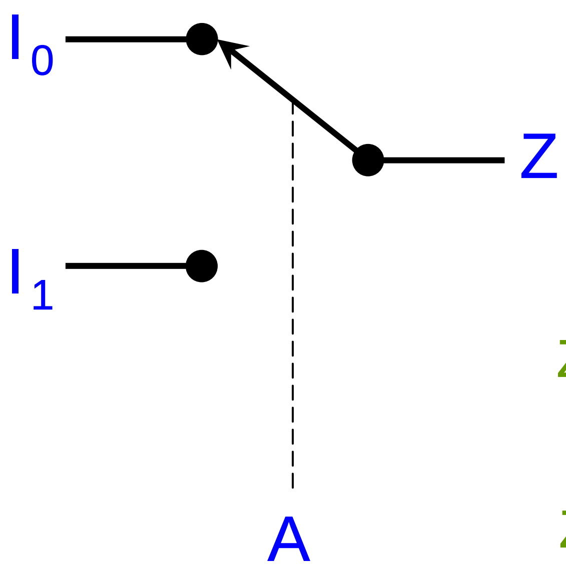
- Building blocks of logic design
  - Gates
  - More complex integrated circuits
    - Multiplexers, decoders, encoders, 3-state buffers, ROMs, PLAs, PAL, CPLDs, FPGAs, etc.
- Integrated circuits
  - Small-scale integration (SSI)
    - Roughly 1~10 gates
    - NAND, NOR, AND, OR, inverters, flip-flops
  - Medium-scale integration (MSI)
    - Roughly 10~100 gates
    - Adders, multiplexers, decoders, registers, counters
  - Large-scale integration (LSI)
    - Roughly 100~1000 gates
    - Memories, microprocessors
  - Very-large-scale integration (VLSI)
    - Roughly 1000+ gates



Multiplexers

- Multiplexer (data selector, MUX)
  - Select one of the data inputs and connect it to the output terminal

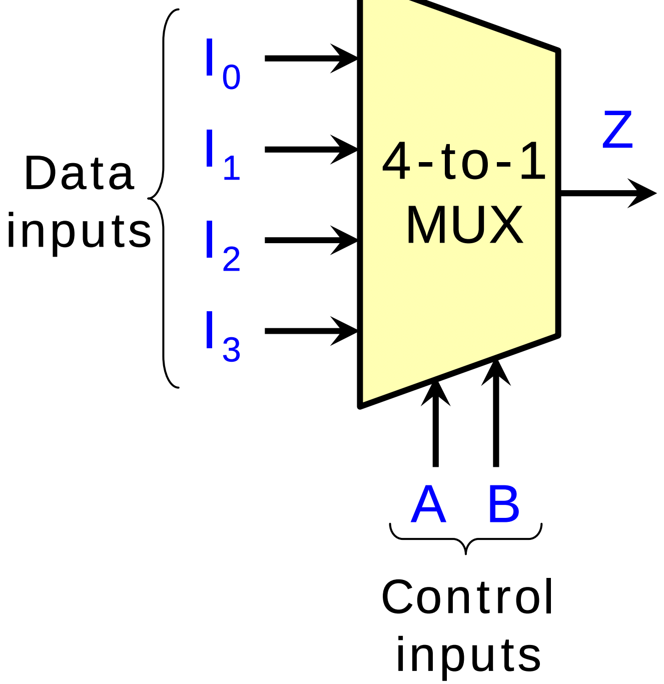


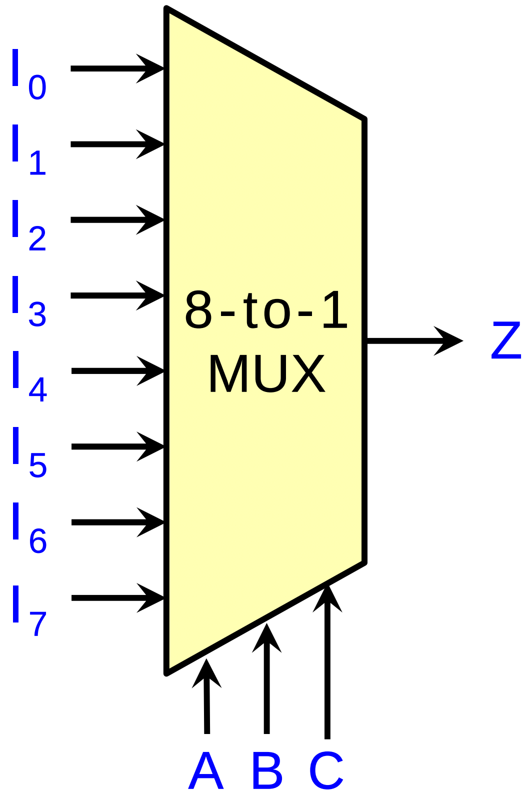


$$Z = \begin{cases} I_0 \text{ when } A = 0 \\ I_1 \text{ when } A = 1 \end{cases}$$

$$Z = A'I_0 + AI_1$$

Multiplexers





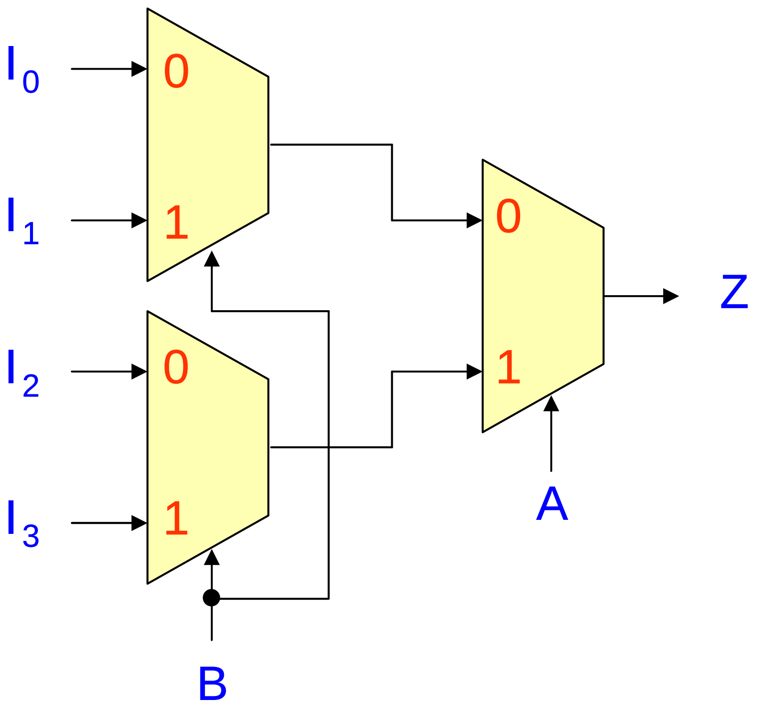
$Z = A'B'I_0 + A'BI_1 + AB'I_2 + ABI_3$

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

Multiplexers

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

$Z = A'B'I_0 + A'BI_1 + AB'I_2 + ABI_3$

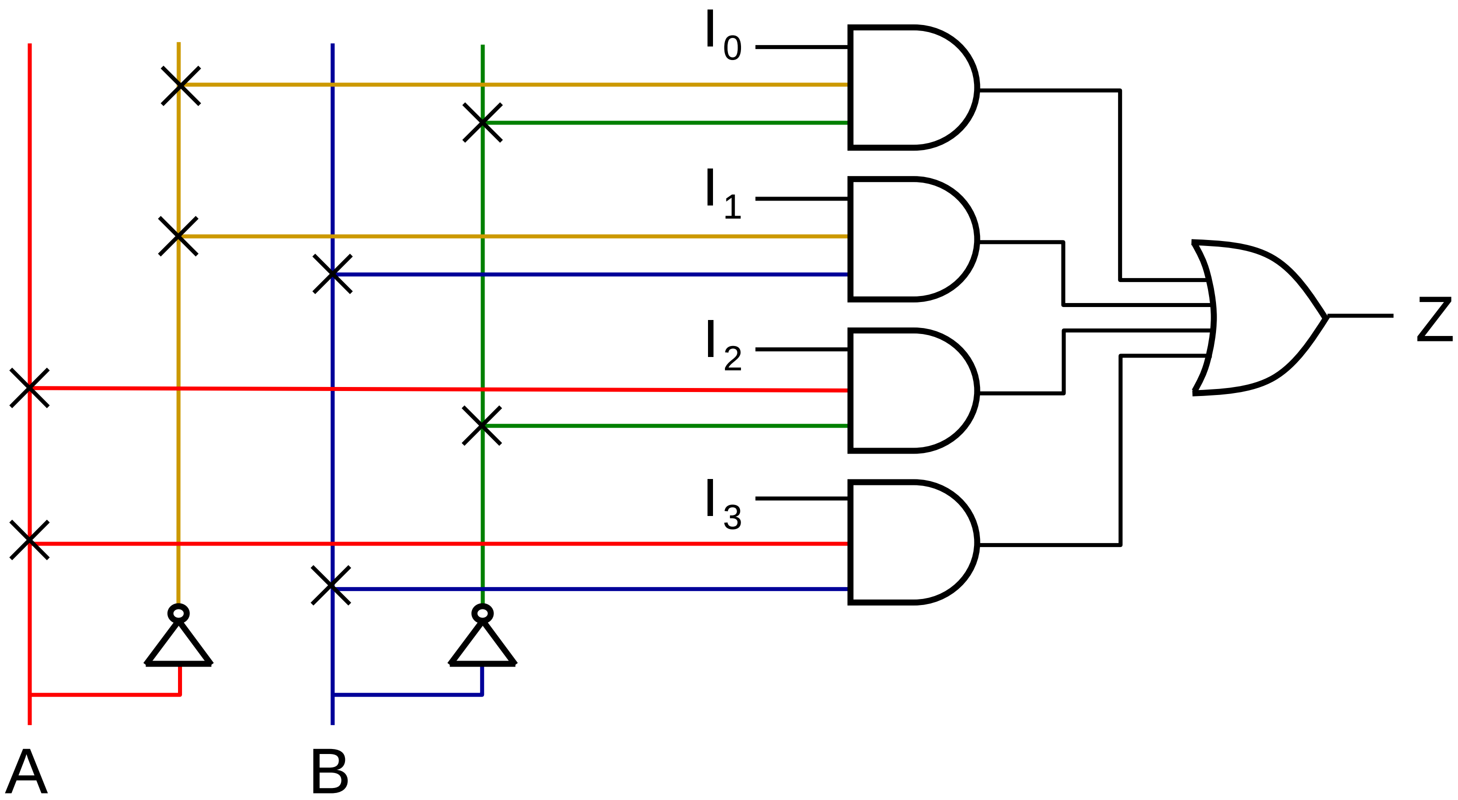


We will learn later how to realize any Boolean function with MUXes (Shannon expansion)

Multiplexers

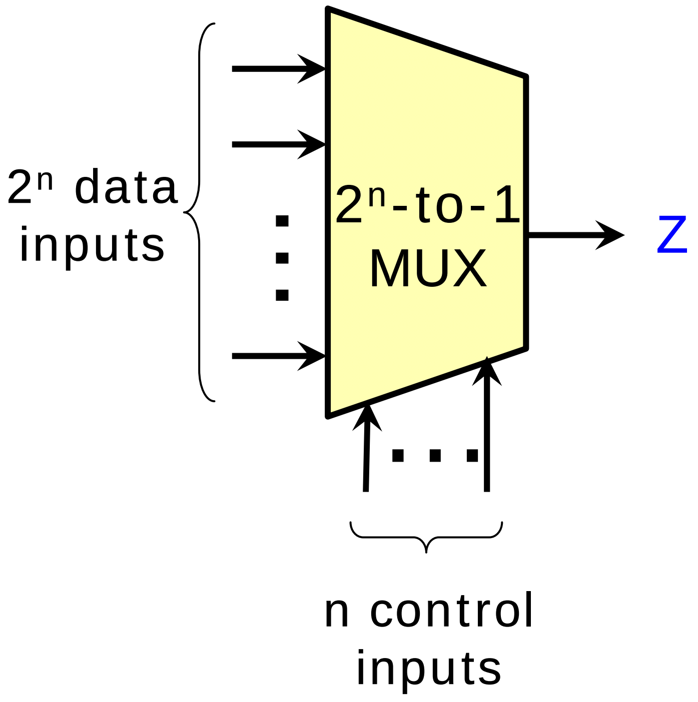
- 4-to-1 MUX realization using two-level AND-OR logic

  $Z = A'B'I_0 + A'BI_1 + AB'I_2 + ABI_3$



Multiplexers

- General 2^n-to-1 MUX



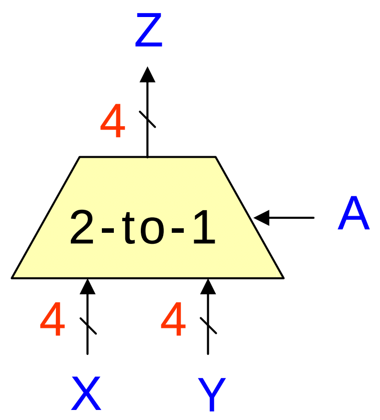
$$Z = \sum_{k=0}^{2^n-1} m_k I_k$$

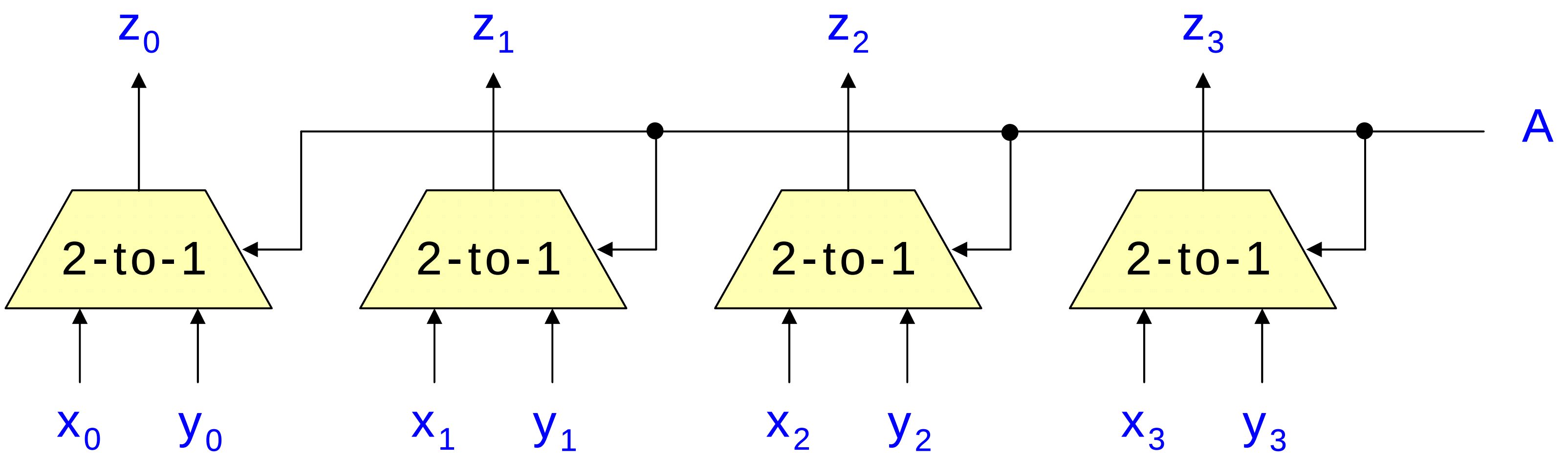
Multiplexers

- Multi-bit data selection

$$Z = \begin{cases} X \text{ when } A = 0 \\ Y \text{ when } A = 1 \end{cases}$$

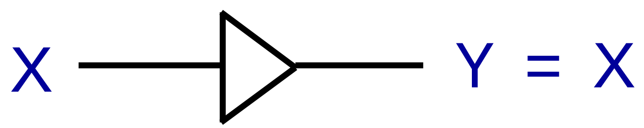
X, Y are multi-bit words



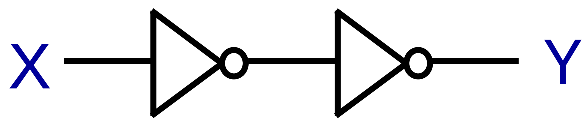


Buffers

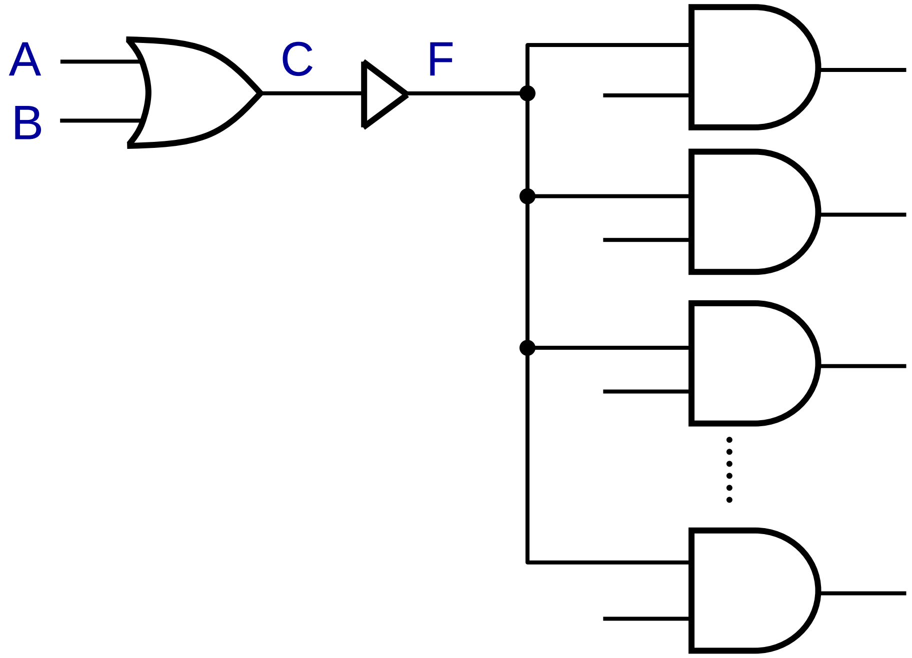
- Logic gates have limited driving capability
  - Gate output can only drive a limited number of other gate inputs without degrading circuit performance
  - **Buffers** can be inserted to increase the driving capability of a gate output



Buffer

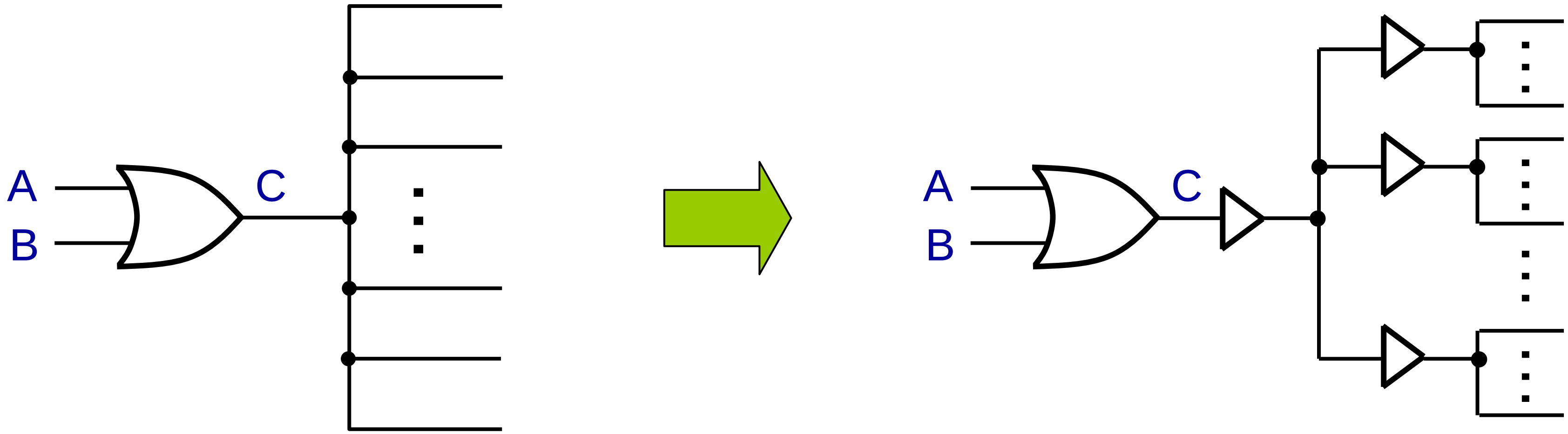


Inverter pair



Buffers

- Multi-stage buffer insertion for high fan-out gates
  - Amortize capacitive loads



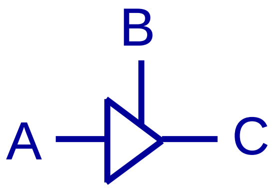
Three-State Buffers

- Direct connection of two or more gate outputs together is dangerous
  - Use of **three-state (tri-state) buffers** permits the connection

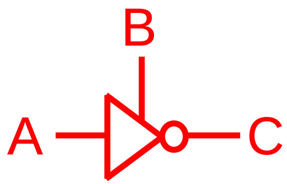
$$C = \begin{cases} A, & \text{when } B = 1 \\ \text{Hi-Z}, & \text{when } B = 0 \end{cases}$$

Three-State Buffers

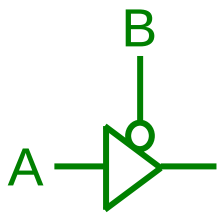
- Four kinds of three-state buffers



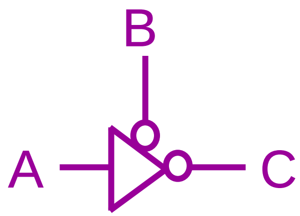
B A	C
0 0	Z
0 1	Z
1 0	0
1 1	1



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



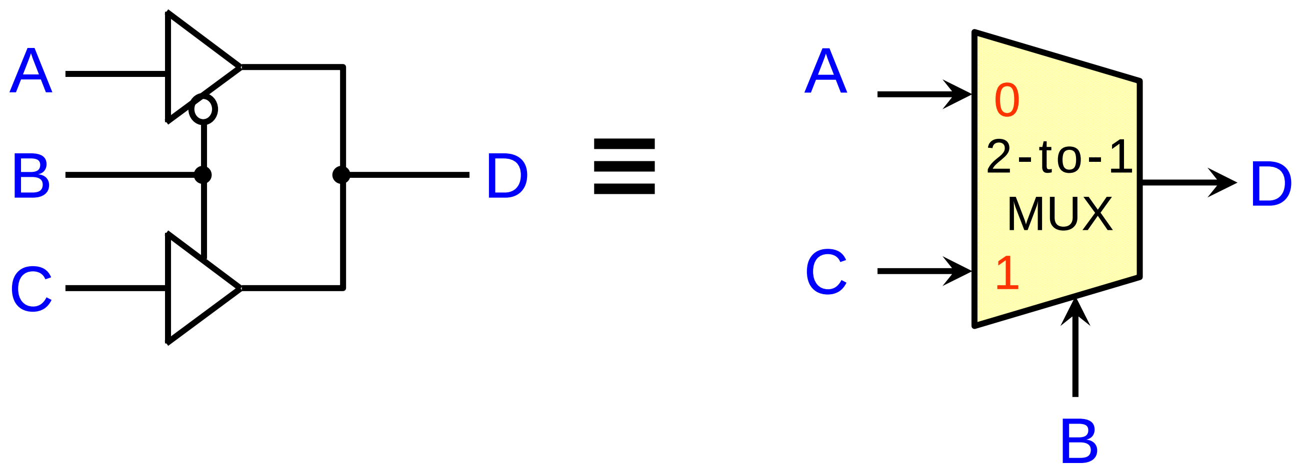
B A	C
0 0	0
0 1	1
1 0	Z
1 1	Z



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

Three-State Buffers

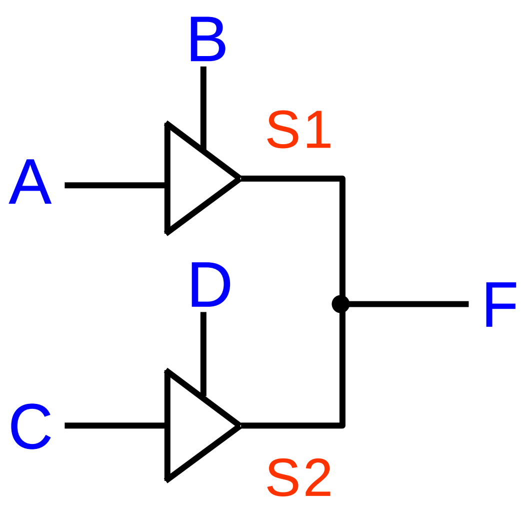
- Data selection using three-state buffers



Are the set of three-state buffers functionally complete?

Three-State Buffers

- Circuit with two three-state buffers

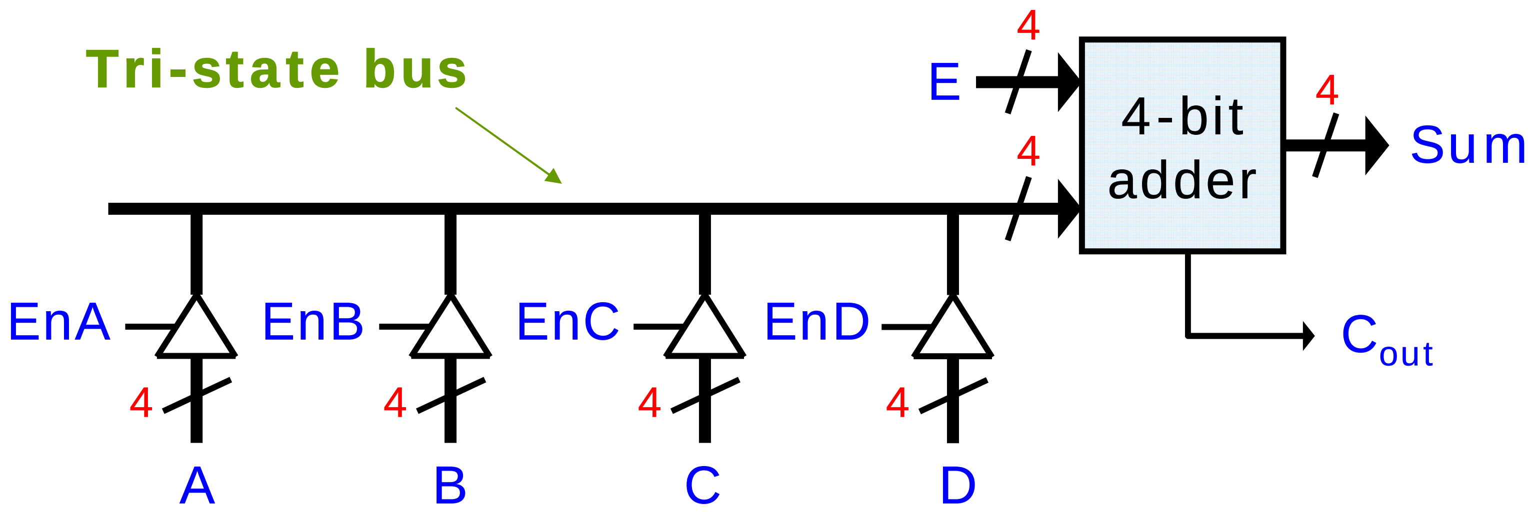


F		S2			
		X	0	1	Z
S1	X	X	X	X	X
	0	X	0	X	0
	1	X	X	1	1
	Z	X	0	1	Z

Three-State Buffers Applications

- Data selection using tri-state buffers (alternative to MUXes)

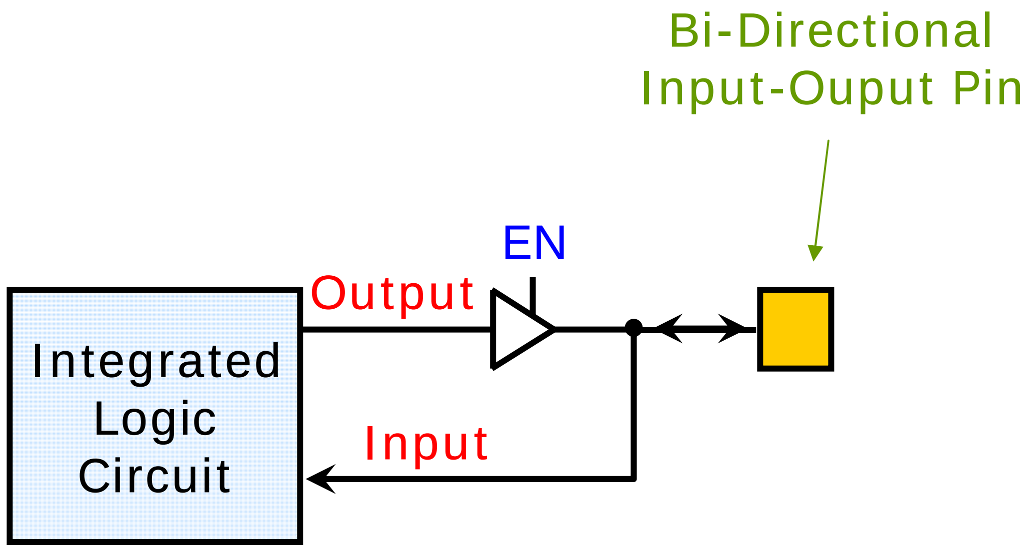
  Example
  - Data communication over a shared channel (**bus**)



Only one of {EnA, EnB, EnC, EnD} is active,
i.e., they are mutually exclusive

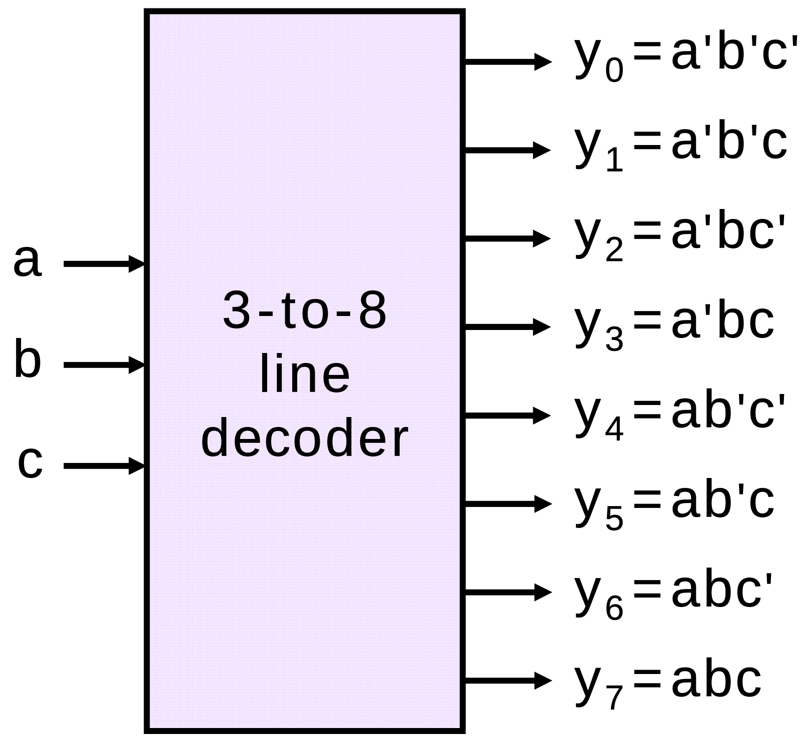
Three-State Buffers Applications

- Integrated circuit with bi-directional input/output pin
  - **Bi-directional I/O pin** can be used as an input pin and as an output pin, but not both at the same time



Decoders and Encoders
Decoders

- 3-to-8 line decoder
  - Exactly one output line will be 1 for each combination of input values

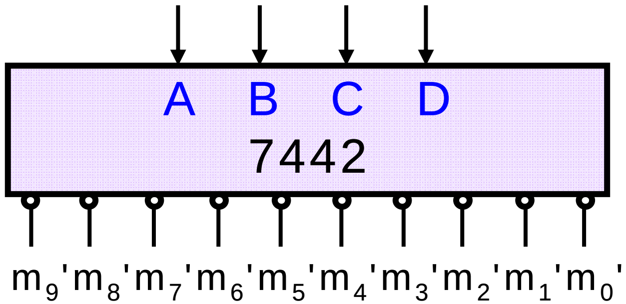


a	b	c	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

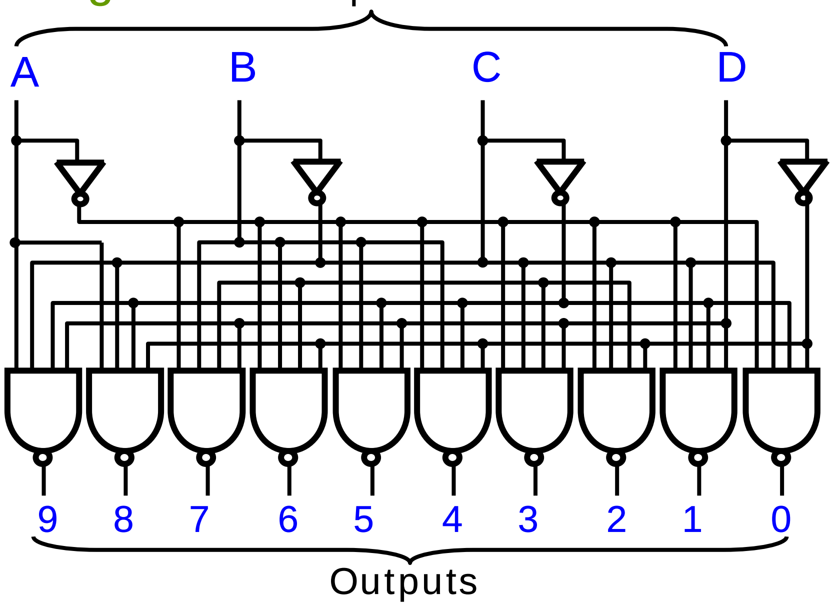
Decoders and Encoders
Decoders

- 4-to-10 decoder with inverted outputs (indicated by circles)
  - Exactly one output line will be 0 for each combination of input values

Block diagram



Logic diagram



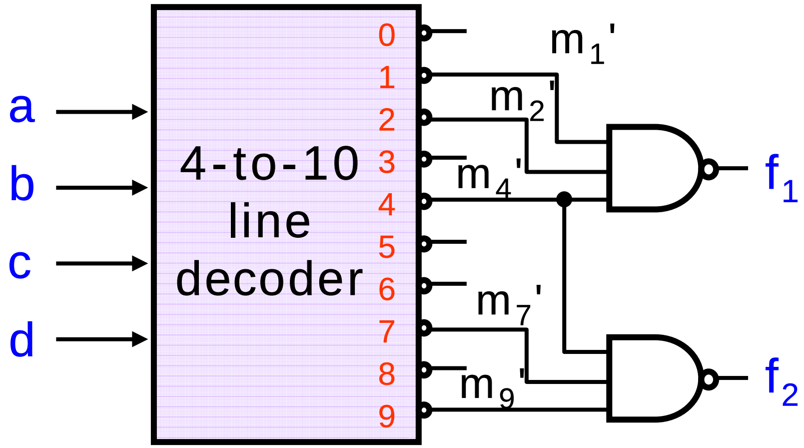
Truth table

BCD input				Decimal Output									
A	B	C	D	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	1	1	1	1	1	1	1	1	1
0	0	0	1	1	0	1	1	1	1	1	1	1	1
0	0	1	0	1	1	0	1	1	1	1	1	1	1
0	0	1	1	1	1	1	0	1	1	1	1	1	1
0	1	0	0	1	1	1	1	0	1	1	1	1	1
0	1	0	1	1	1	1	1	1	0	1	1	1	1
0	1	1	0	1	1	1	1	1	1	0	1	1	1
0	1	1	1	1	1	1	1	1	1	1	0	1	1
1	0	0	0	1	1	1	1	1	1	1	1	0	1
1	0	0	1	1	1	1	1	1	1	1	1	1	0
1	0	1	0	1	1	1	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1	1	1	1	1	1
1	1	0	0	1	1	1	1	1	1	1	1	1	1
1	1	0	1	1	1	1	1	1	1	1	1	1	1
1	1	1	0	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1

Decoders and Encoders
Decoders

- n-to-2^n line decoder generates all 2^n minterms (or maxterms) of the n input variables
  - The outputs are defined by the equations
    $y_i = m_i$, for $i = 0, \ldots, 2^n-1$ (noninverted outputs)
    $y_i = m_i' = M_i$, for $i = 0, \ldots, 2^n-1$ (inverted outputs)
  - Can be used for function construction
    Example

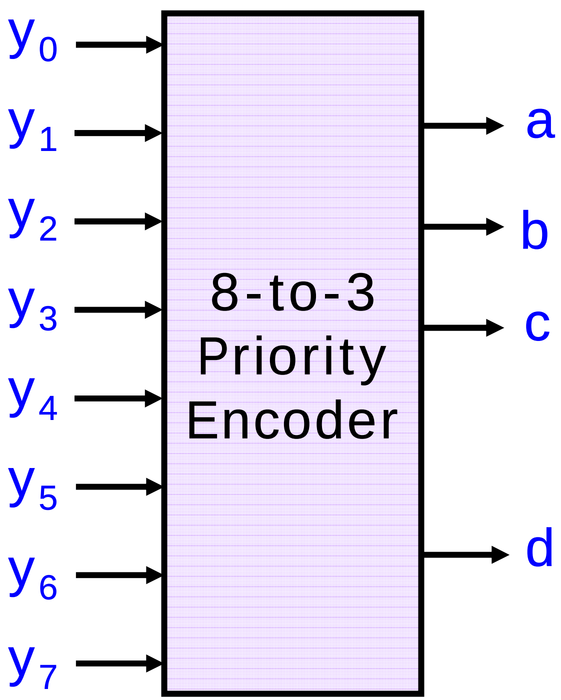
$$f1(a,b,c,d) = m_1+m_2+m_4 = (m_1'm_2'm_4')'$$
$$f2(a,b,c,d) = m_4+m_7+m_9 = (m_4'm_7'm_9')'$$



Note that m_i' with $i \geq 10$ is not available in the 4-to-10 line decoder

Decoders and Encoders
Encoders

- An encoder performs the inverse function of a decoder
- 8-to-3 priority encoder
  - If input y_i is 1 with $y_j = 0$ for all $j > i$, then the outputs (a,b,c) represent a binary number equal to i
  - Output d is 1 if any input is 1, otherwise, d is 0



y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	a	b	c	d
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

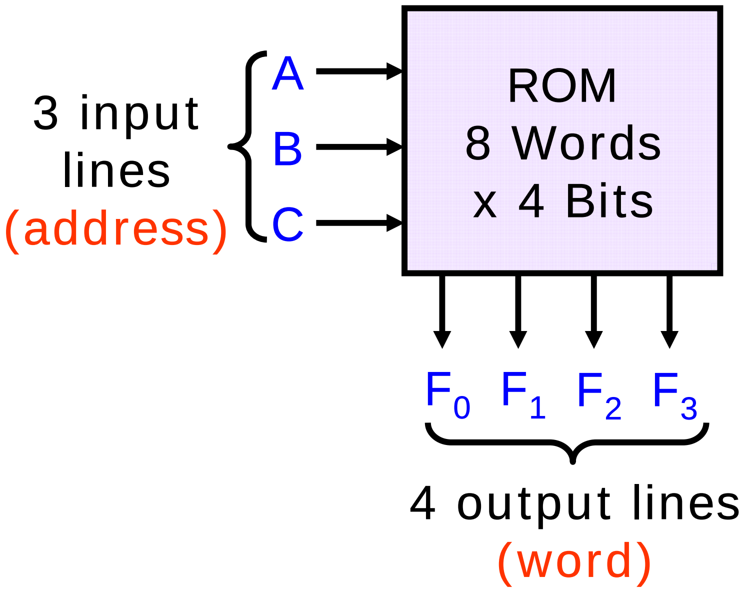
distinguished by d

don't care

Read-Only Memories

- A **read-only memory (ROM)** stores an array of binary data, which can be read out whenever desired, but cannot be changed under normal operating conditions
  - An output pattern stored in the ROM is called a **word**
  - An input combination serves as an **address** which can select one of the words stored in the ROM

Block diagram



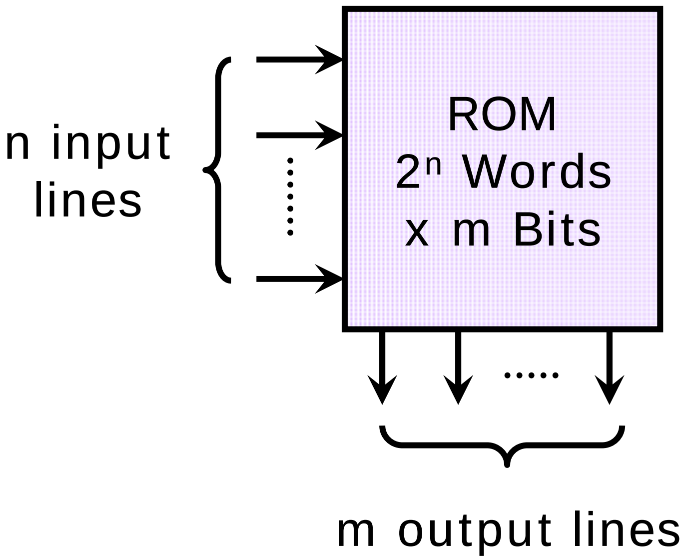
Truth table for ROM

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

typical data stored in a ROM (here, 2^3 words of 4 bits each)

Read-Only Memories

- A ROM which has n input lines and m output lines contains an array of 2^n words, and each word is m bits long
  - The input lines serve as an address to select one of the 2^n words
  - A $(2^n \times m)$ ROM can realize m functions of n variables



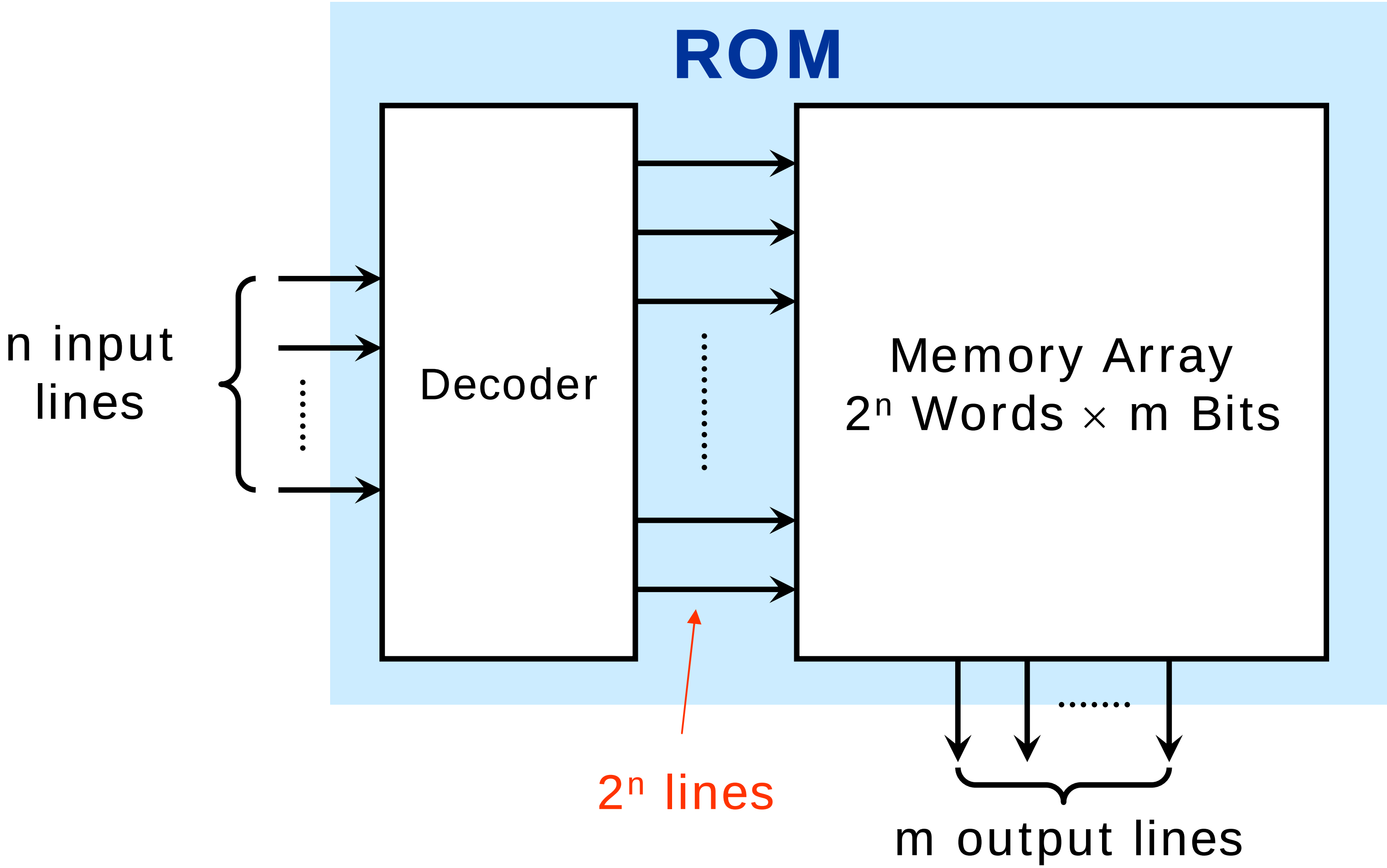
size = $2^n \times m$ (bits)

n input variables	m output variables
00…00	100…110
00…01	010…111
00…10	101…101
00…11	110…010
:	:
11…00	001…011
11…01	110…110
11…10	011…000
11…11	111…101

typical data array stored in ROM (here, 2^n words of m bits each)

Read-Only Memories
Basic Structure

- A ROM consists of a decoder and a memory array

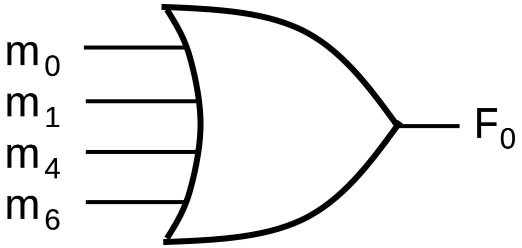


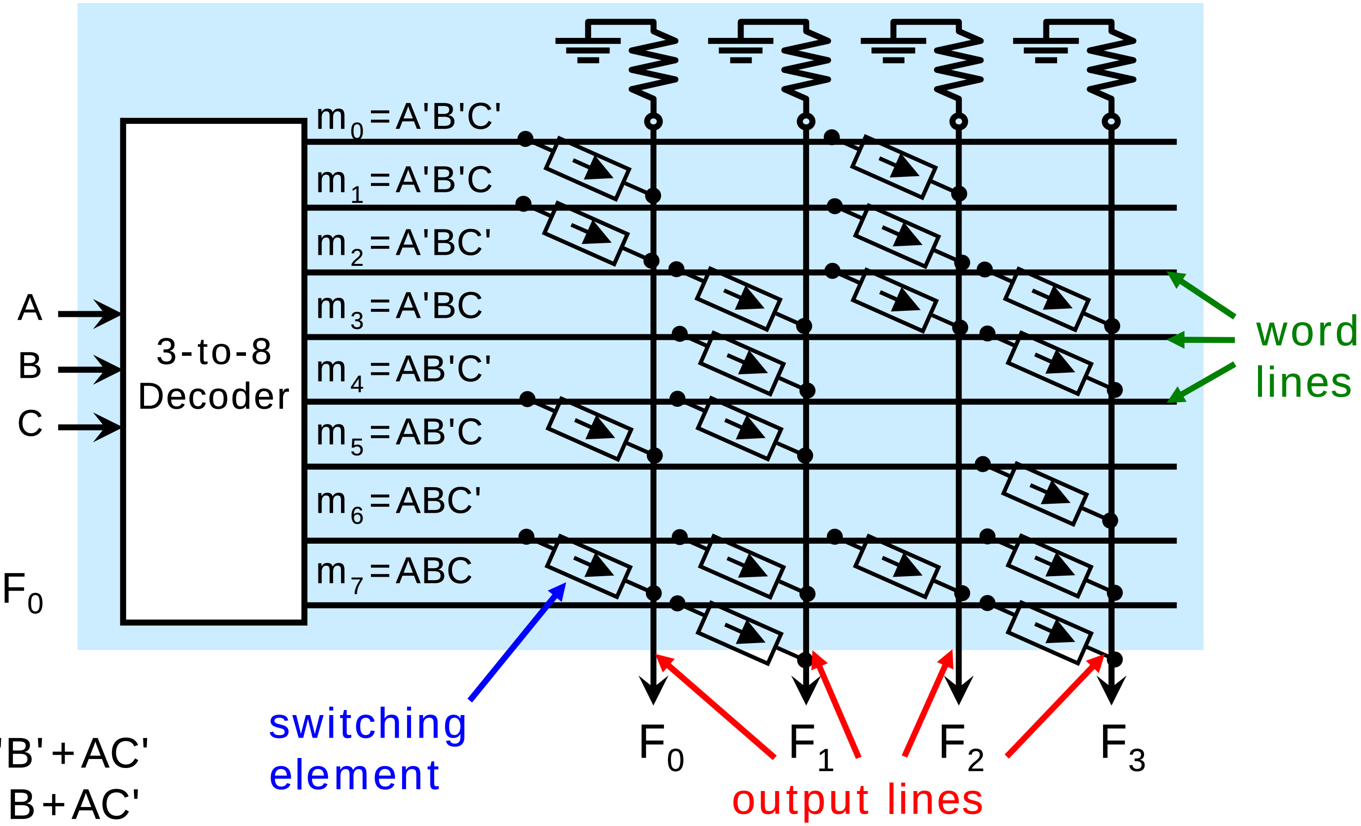
Read-Only Memories
Basic Structure

Example

- The contents of a ROM are usually specified by a truth table

m_i	$F_0F_1F_2F_3$
0	1 0 1 0
1	1 0 1 0
2	0 1 1 1
3	0 1 0 1
4	1 1 0 0
5	0 0 0 1
6	1 1 1 1
7	0 1 0 1



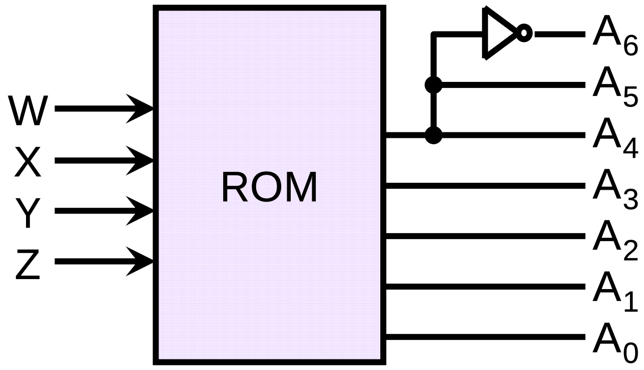


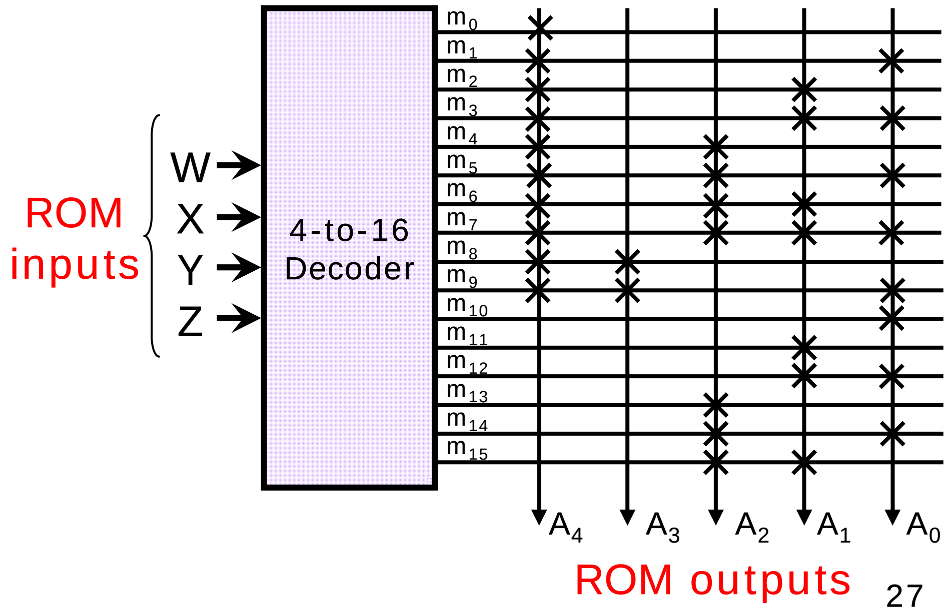
$F_0 = \sum m(0,1,4,6) = A'B'+AC'$
$F_1 = \sum m(2,3,4,6,7) = B+AC'$
$F_2 = \sum m(0,1,2,6) = A'B'+BC'$
$F_3 = \sum m(2,3,5,6,7) = AC+B$

Read-Only Memories
Basic Structure

- Multiple-output combinational circuits can be realized using ROMs
  Example

Input WXYZ	Hex Digit	ASCII Code for Hex Digit A_6 A_5 A_4 A_3 A_2 A_1 A_0
0000	0	0 1 1 0 0 0 0
0001	1	0 1 1 0 0 0 1
0010	2	0 1 1 0 0 1 0
0011	3	0 1 1 0 0 1 1
0100	4	0 1 1 0 1 0 0
0101	5	0 1 1 0 1 0 1
0110	6	0 1 1 0 1 1 0
0111	7	0 1 1 0 1 1 1
1000	8	0 1 1 1 0 0 0
1001	9	0 1 1 1 0 0 1
1010	A	1 0 0 0 0 0 1
1011	B	1 0 0 0 0 1 0
1100	C	1 0 0 0 0 1 1
1101	D	1 0 0 0 1 0 0
1110	E	1 0 0 0 1 0 1
1111	F	1 0 0 0 1 1 0





Read-Only Memories

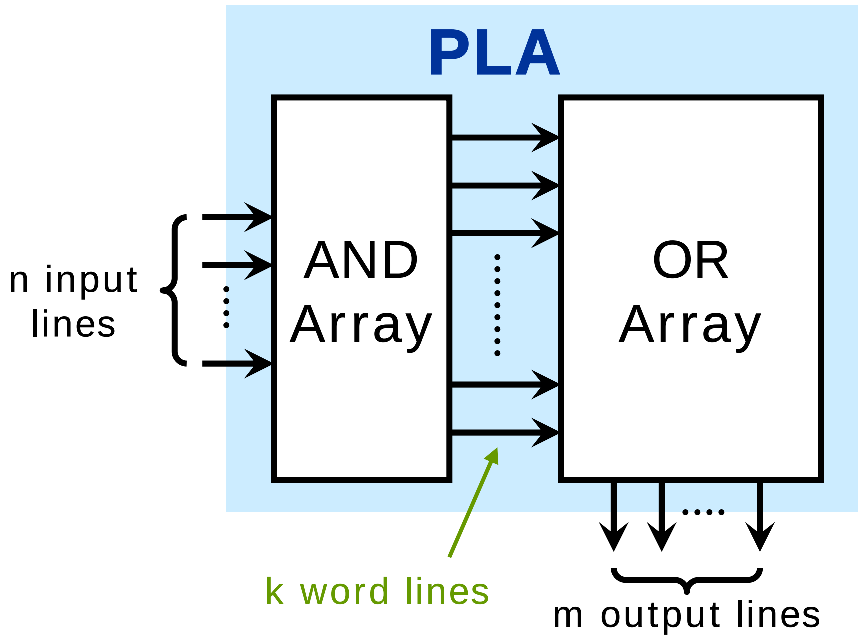
- Common types of ROMs
  - Mask-programmable ROMs
    - At the time of manufacture, data array is permanently stored in a mask-programmable ROM
  - Programmable ROMs (PROMs)
    - Can be programmed once
  - Electrically erasable programmable ROM (EEPROMs); flash memories
    - Can be erased and reprogrammed a limited number of times

Programmable Logic Devices

- A **programmable logic device (PLD)** is a general name for a digital IC capable of being programmed to provide a variety of different logic functions
  - For a digital system designed using a PLD, changes in the design can easily be made by changing the programming of the PLD without changing the wiring
  - Common types of PLD
    - Programmable logic arrays (PLAs)
    - Programmable array logic (PAL)
      - A special case of PLAs

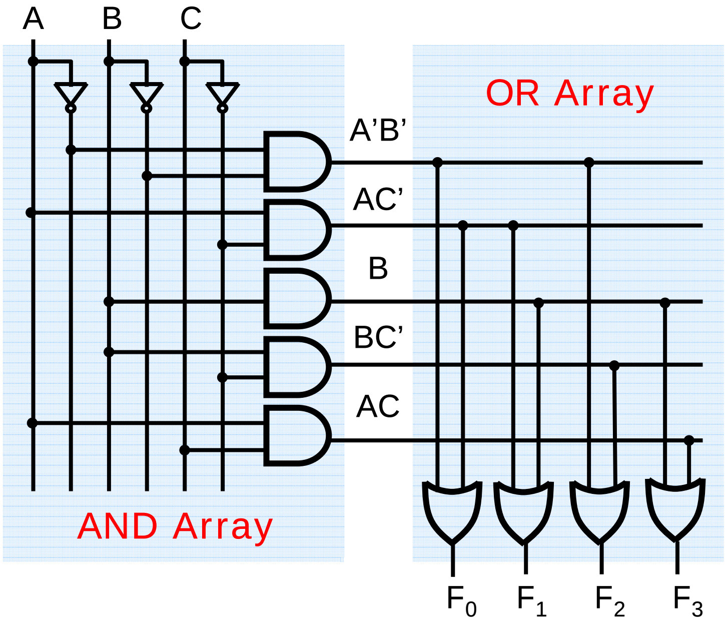
Programmable Logic Devices
Programmable Logic Arrays

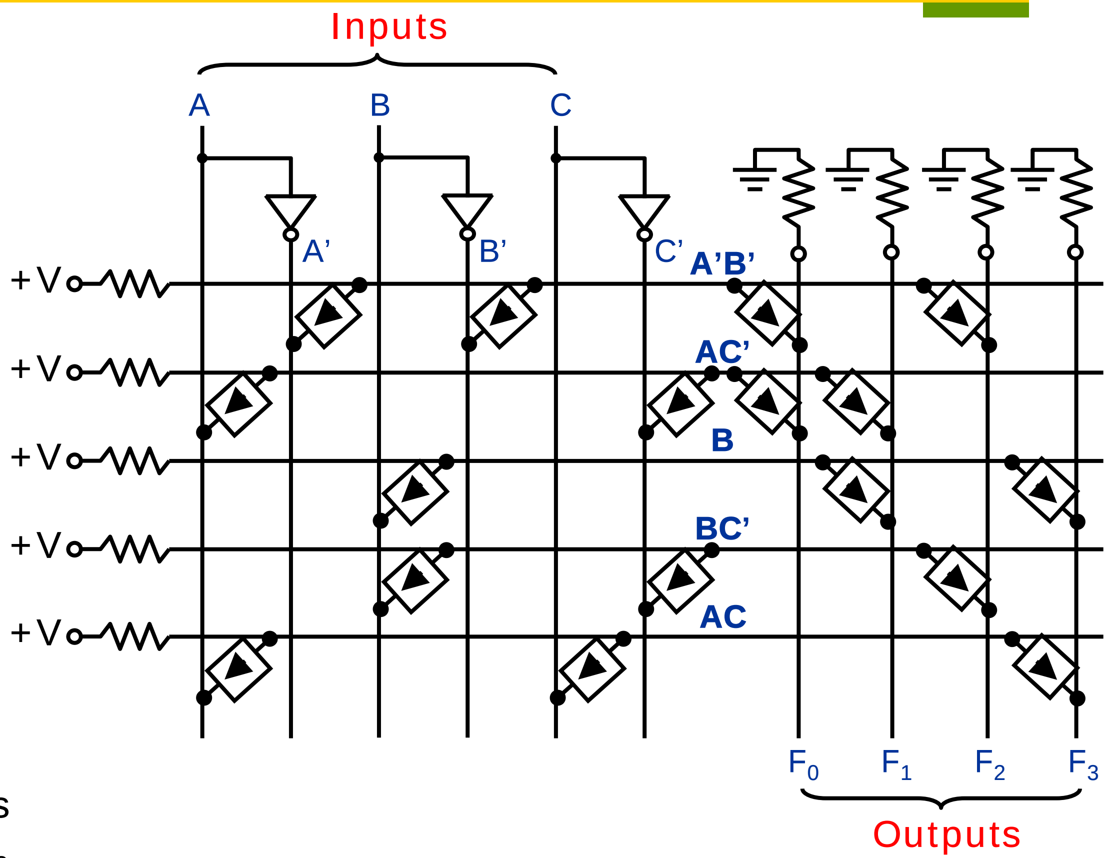
- A **programmable logic array** (**PLA**) performs the same basic function as a ROM
  - A PLA with n inputs and m outputs can realize m functions of n variables
  - A PLA implements an SOP expression, while a ROM implements a truth table
    - The decoder of a ROM is replaced with an AND array realizing selected product terms of the input variables; the OR array ORs together the product terms



Programmable Logic Devices
Programmable Logic Arrays

Example





PLA table

Product Term	Inputs A B C	Outputs $F_0F_1F_2F_3$
A'B'	0 0 –	1 0 1 0
AC'	1 – 0	1 1 0 0
B	– 1 –	0 1 0 1
BC'	– 1 0	0 0 1 0
AC	1 – 1	0 0 0 1

$$F_0 = A'B' + AC'$$
$$F_1 = AC' + B$$
$$F_2 = A'B' + BC'$$
$$F_3 = B + AC$$

Programmable Logic Devices
Programmable Logic Arrays

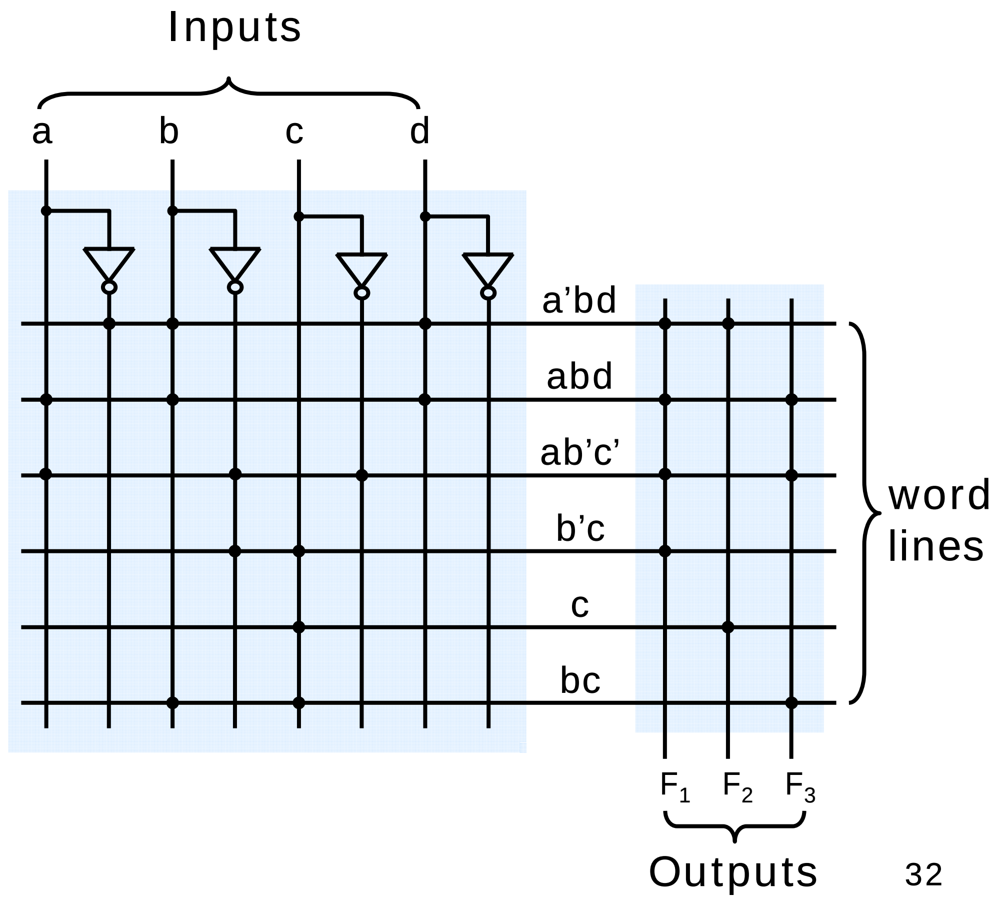
Example (Eq. 7-23b)

$f_1 = a'bd + abd + ab'c' + b'c$
$f_2 = c + a'bd$
$f_3 = bc + ab'c' + abd$

PLA table

	a b c d	f_1 f_2 f_3
a'bd	0 1 – 1	1 1 0
abd	1 1 – 1	1 0 1
ab'c'	1 0 0 –	1 0 1
b'c	– 0 1 –	1 0 0
c	– – 1 –	0 1 0
bc	– 1 1 –	0 0 1

PLA structure



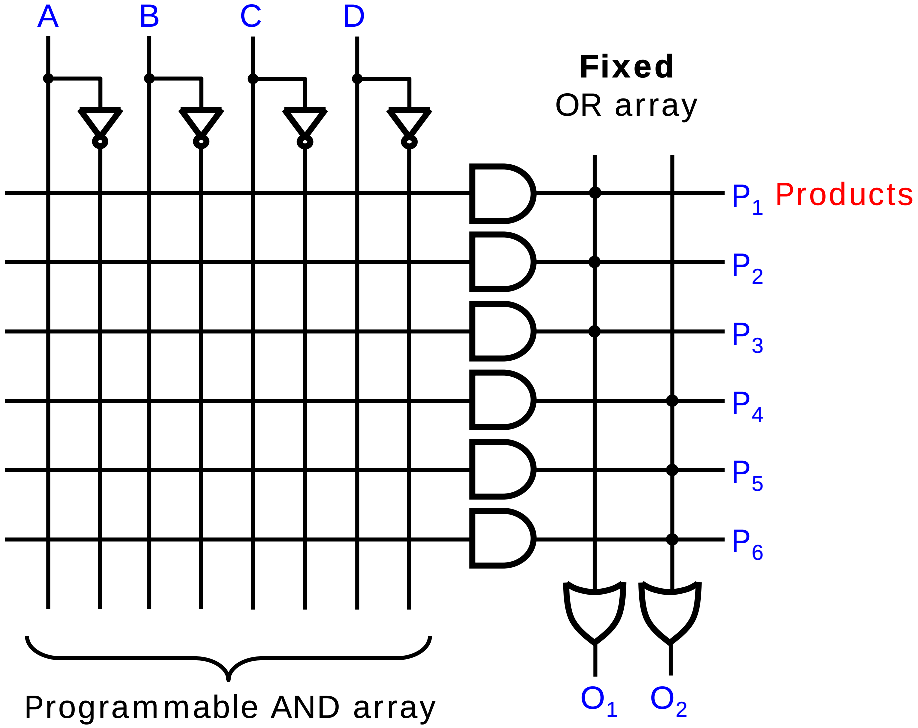
Programmable Logic Devices
Programmable Logic Arrays

- A PLA table for a PLA differs from a truth table for a ROM
  - In a truth table, exactly one row will be selected by each combination of input values
  - In a PLA table, zero, one, or more rows may be selected by each combination of input values (since each row represents a general product term)
    - To determine the value of f_i for a given input combination, the values in the selected rows must be ORed together

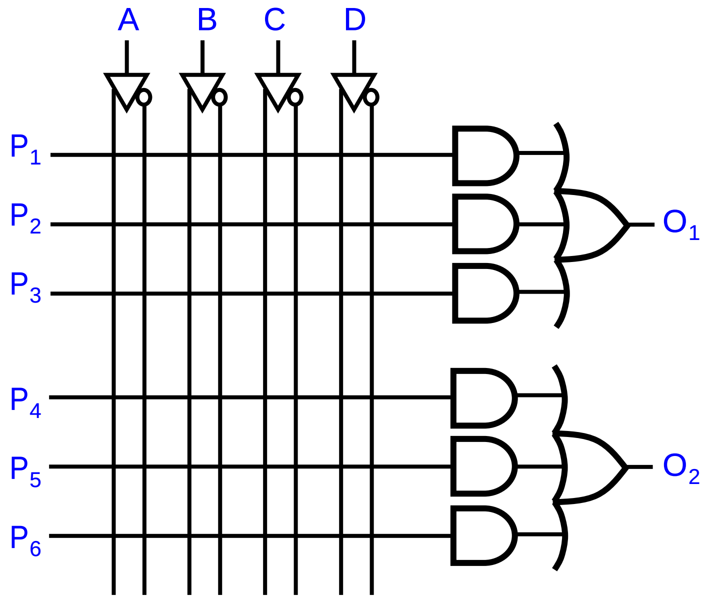
- Common types of PLAs
  - Mask-programmable PLAs
  - Field-programmable PLAs (FPLAs)

Programmable Logic Devices
Programmable Array Logic

- The **programmable array logic** (**PAL**) is a special case of PLA in which the AND array is programmable and the OR array is fixed
  - PAL is less expensive and easier to program



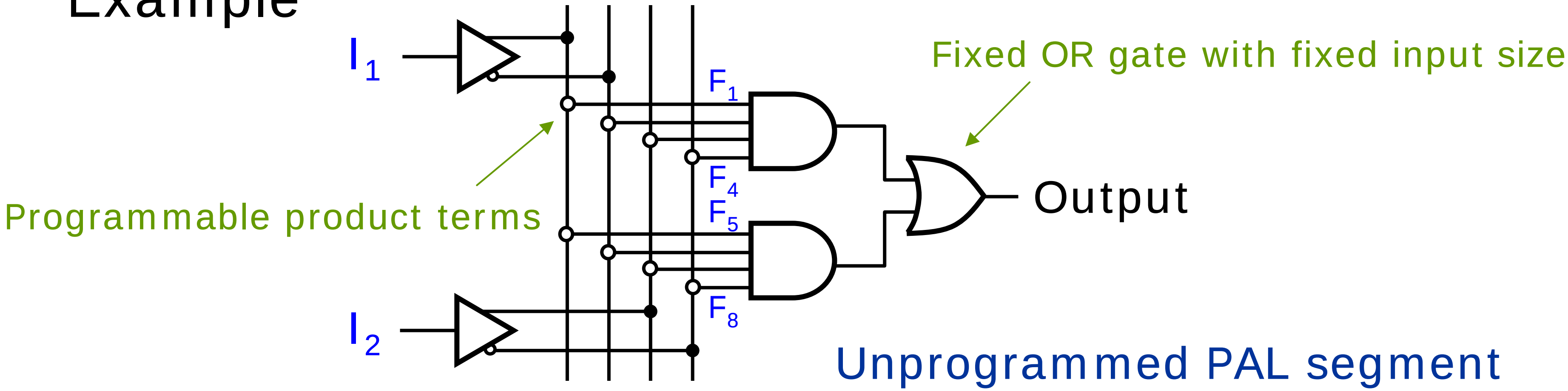
PAL device



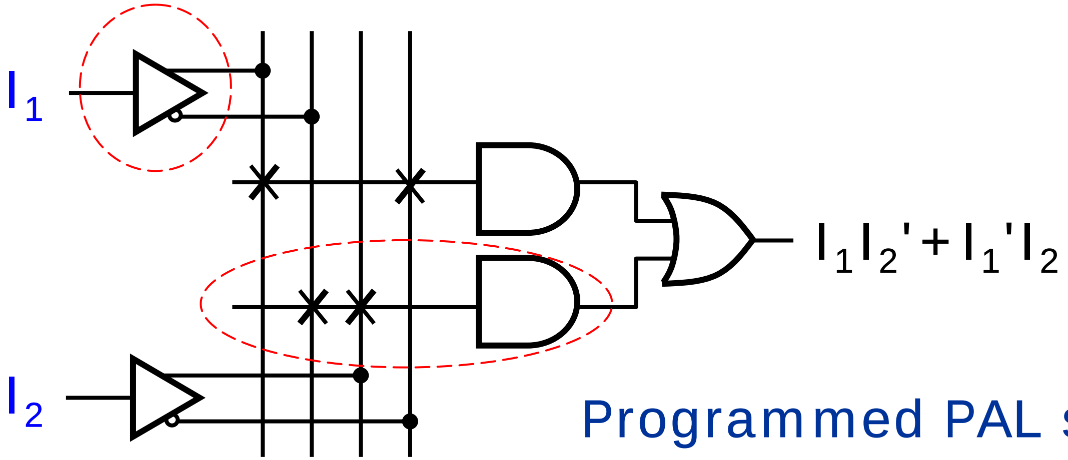
Standard PAL representation

Programmable Logic Devices
Programmable Array Logic

Example



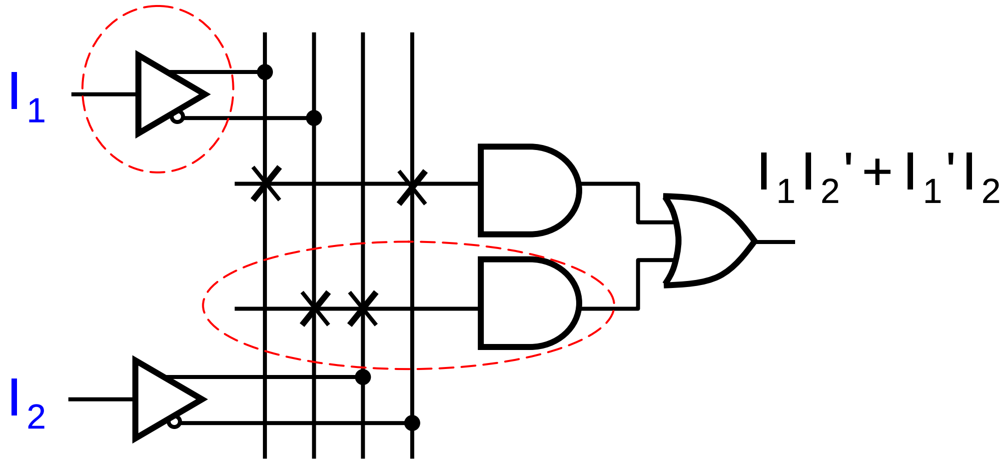
Unprogrammed PAL segment

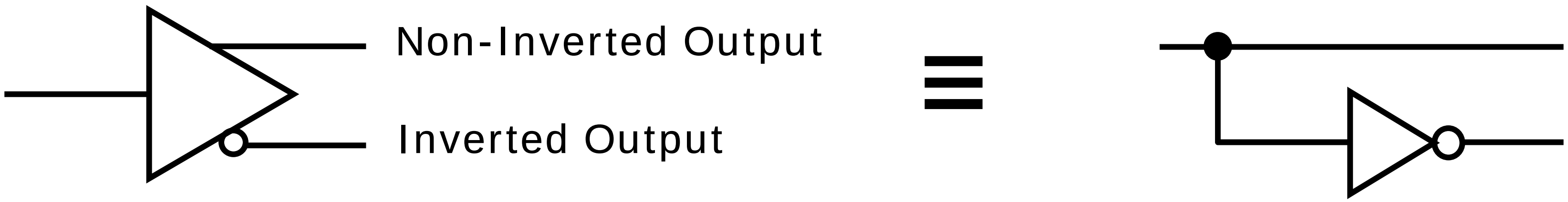


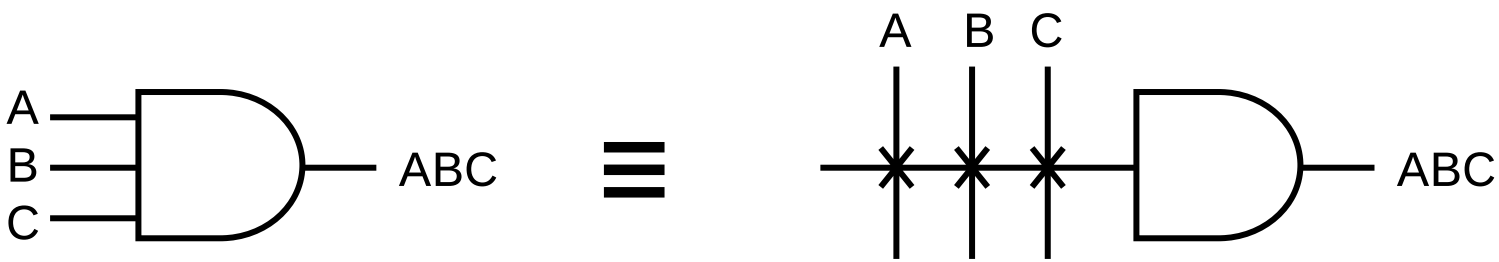
Programmed PAL segment

Programmable Logic Devices
Programmable Array Logic

- Notation





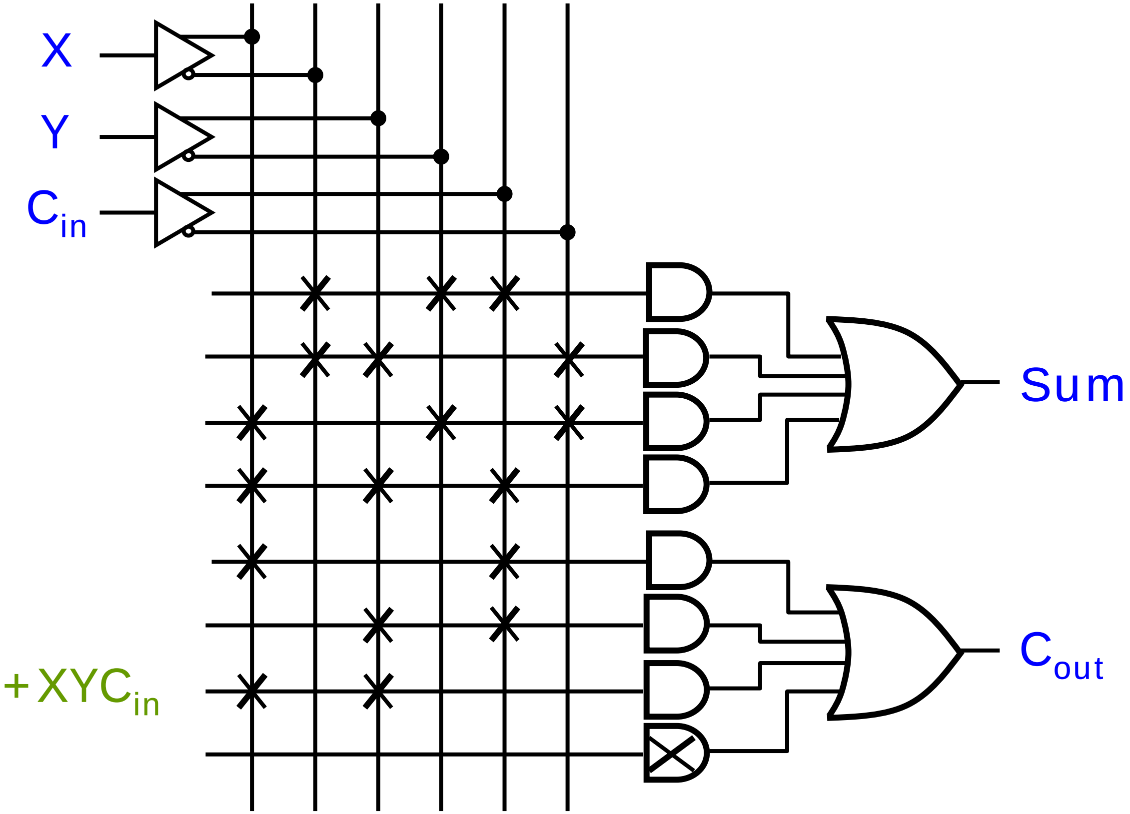


Programmable Logic Devices
Programmable Array Logic

Example

- PAL implementation of full adder

X	Y	C_{in}	Sum	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



$$Sum = X'Y'C_{in} + X'YC_{in}' + XY'C_{in}' + XYC_{in}$$
$$C_{out} = XC_{in} + YC_{in} + XY$$

Programmable Logic Devices
Programmable Array Logic

- Unlike PLA, the AND terms cannot be shared among two or more OR gates
  - Each function to be realized can be simplified by itself without regard to common terms
- For a given type of PAL, the number of AND terms that feed each output OR gate is fixed and limited
  - Need to choose a PAL with an appropriate number of OR-gate inputs

Other Programmable Logic Devices

- Complex programmable logic devices (CPLDs)
  - §9.7 (skipped)

- Field programmable gate arrays (FPGAs)
  - §9.8 (skipped)

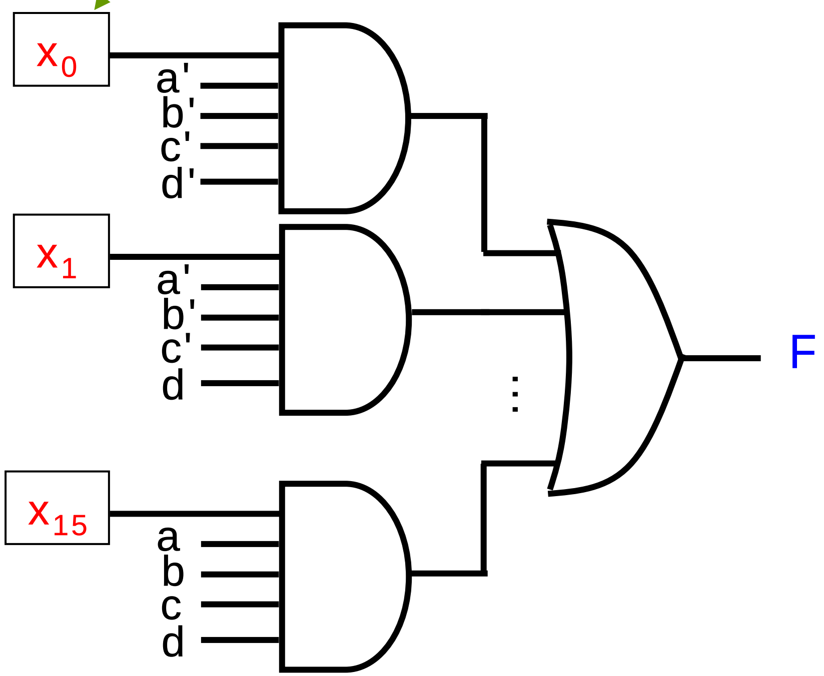
Function Generators

- Implementation of a 4-input Lookup Table (LUT)

a b c d	F
0 0 0 0	x_0
0 0 0 1	x_1
⋮	⋮
1 1 1 1	x_{15}

$x_i \in \{0,1\}$

bits stored in the LUT
(enable/disable minterms)

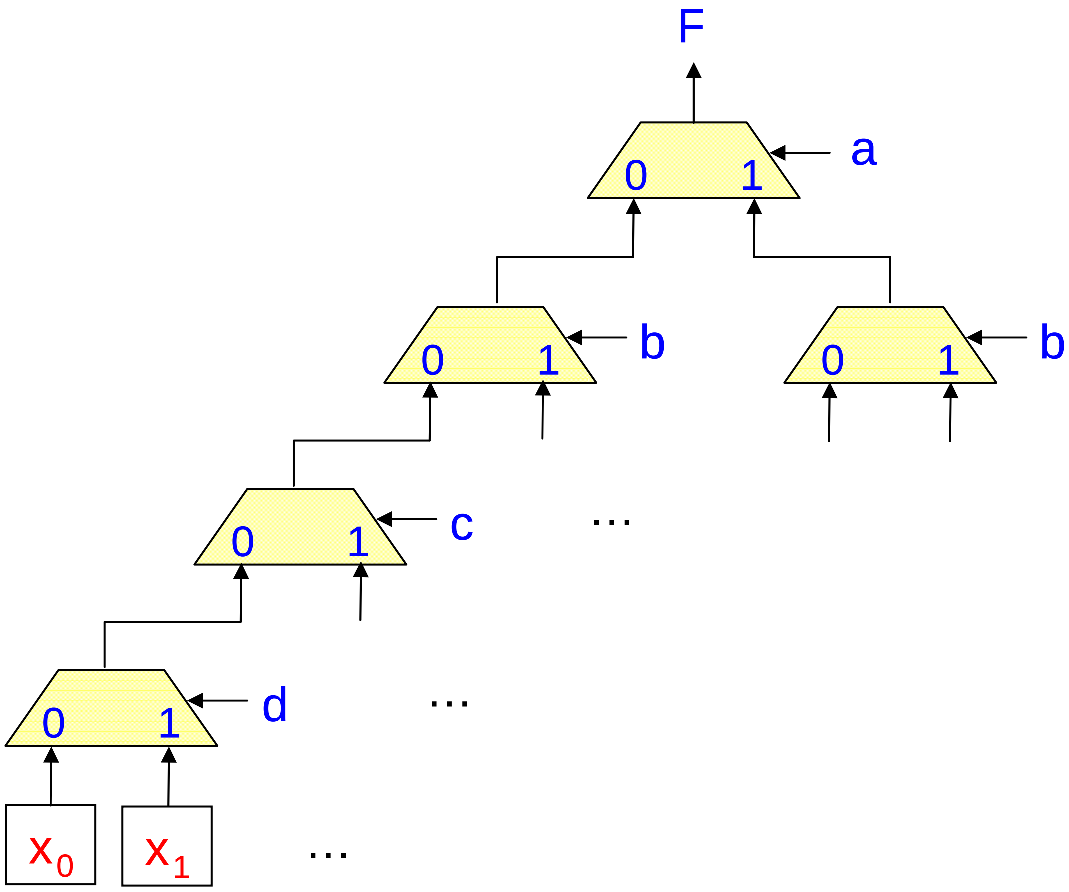


Function Generators

- Another implementation of a Lookup Table (LUT)

a b c d	F
0 0 0 0	x_0
0 0 0 1	x_1
⋮	⋮
1 1 1 1	x_{15}

$a_i \in \{0,1\}$



Decomposition of Switching Functions

- To implement a Boolean function with more than 4 inputs using 4-variable function generators, it has to be decomposed into subfunctions, each with $\leq$ 4 inputs
  - **Shannon's expansion theorem** provides a decomposition method

Decomposition of Switching Functions
Shannon's Expansion Theorem

- Shannon's expansion theorem
  - By expanding a function f(a,b,c,d) about variable a, the following equality holds:

    $f(a,b,c,d) = a' \cdot f(0,b,c,d) + a \cdot f(1,b,c,d) = a'f_0 + af_1$

Example

$$f(a,b,c,d) = c'd' + a'b'c + bcd + ac'$$
$$= a'(c'd' + b'c + bcd) + a(c'd' + bcd + c')$$
$$= a'\underbrace{(c'd' + b'c + cd)}_{f_0} + a\underbrace{(c' + bd)}_{f_1}$$

fewer inputs

Decomposition of Switching Functions
Shannon's Expansion Theorem

- Shannon's expansion using K-map

f:

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

f_0 (a=0) | f_1 (a=1):

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

$$\begin{cases} a=0, & f_0=c'd'+b'c+cd \\ a=1, & f_1=c'+bd \end{cases}$$

Decomposition of Switching Functions Shannon's Expansion Theorem

- Shannon's expansion theorem
  - By expanding a function f(a,b,c,d) about variable a, the following equality holds:

    $f(x_1,\ldots,x_{i-1},x_i,x_{i+1},\ldots,x_n)$

    $= x_i' \cdot f(x_1,\ldots,x_{i-1},0,x_{i+1},\ldots,x_n) + x_i \cdot f(x_1,\ldots,x_{i-1},1,x_{i+1},\ldots,x_n)$

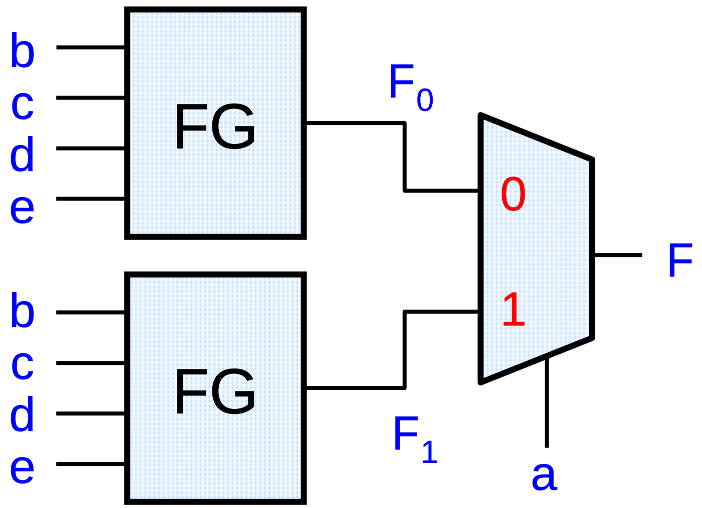
    $= x_i' \cdot f_0 + x_i \cdot f_1$

    fewer inputs

Decomposition of Switching Functions
Multiplexers and Shannon Expansion

- Any 5-variable function can be realized using two 4-variable function generators and a 2-to-1 MUX
  - Apply Shannon's expansion once:

    $f(a,b,c,d,e) = a' \cdot f(0,b,c,d,e) + a \cdot f(1,b,c,d,e) = a'f_0 + af_1$



Decomposition of Switching Functions
Multiplexers and Shannon Expansion

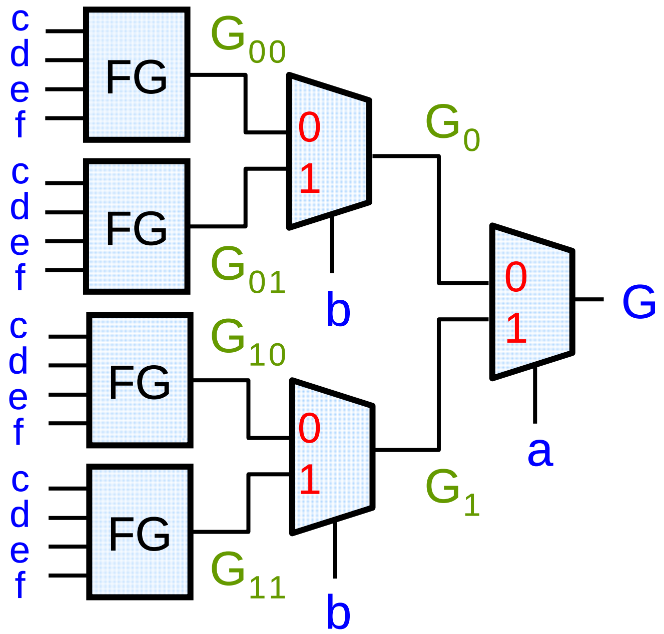
- Any 6-variable function can be realized using four 4-variable function generators and three 2-to-1 MUXes
  - Apply Shannon's expansion twice:

$$G(a,b,c,d,e,f) = a' \cdot G(0,b,c,d,e,f) + a \cdot G(1,b,c,d,e,f) = a'G_0 + aG_1$$

$$G_0 = b' \cdot G(0,0,c,d,e,f) + b \cdot G(0,1,c,d,e,f) = b'G_{00} + bG_{01}$$

$$G_1 = b' \cdot G(1,0,c,d,e,f) + b \cdot G(1,1,c,d,e,f) = b'G_{10} + bG_{11}$$

$$G(a,b,c,d,e,f) = a'b'G_{00} + a'bG_{01} + ab'G_{10} + abG_{11}$$



Decomposition of Switching Functions
Multiplexers and Shannon Expansion

Exercises

- Realize $F=A'B'+AC$ using a 4-to-1 MUX

- Realize $F=A'B'C'+B'CD+A'BC+BCD'+AC'D'$ using an 8-to-1 MUX

- Realize $F=A'B'C'+B'CD+A'BC+BCD'+AC'D'$ using a 4-to-1 MUX and 2-input function generators