



## **DVA494**

### Programming of Reliable Embedded Systems

---

Obed Mogaka | Hamid Mousavi | Masoud Daneshtalab

IDD, Mälardalens University

January 19, 2026

## Review of Logic Design Fundamentals

- Boolean Algebra and Algebraic Simplification
- Karnaugh Maps
- Designing With NAND and NOR Gates
- Combinational Logic
- Hazards in Combinational Circuits

**How Computers/Digital/Embedded Systems Work!**

**How Computers/Digital/Embedded Systems Work!**  
**How Can We Design a Digital System!**

### Why Do We Do Computing?

### Why Do We Do Computing?

To Solve Problems, Gain Insight, and Enable a Better Life and Future.

## Why Do We Do Computing?

To Solve Problems, Gain Insight, and Enable a Better Life and Future.

- **How Does a Computer Solve Problems?**

## Why Do We Do Computing?

To Solve Problems, Gain Insight, and Enable a Better Life and Future.

- **How Does a Computer Solve Problems?** Orchestrating Electrons



## Why Do We Do Computing?

To Solve Problems, Gain Insight, and Enable a Better Life and Future.

- **How Does a Computer Solve Problems?** Orchestrating Electrons
- **How Do Problems Get Solved by Electrons?**

## Why Do We Do Computing?

To Solve Problems, Gain Insight, and Enable a Better Life and Future.

- **How Does a Computer Solve Problems?** Orchestrating Electrons
- **How Do Problems Get Solved by Electrons?** I don't know

# The Art of Managing Complexity

- Abstraction
- Discipline
- The Three-Y's
  - Hierarchy
  - Modularity
  - Regularity

Hiding details when they are not important

Abstraction Levels	Examples
<b>Application Software</b>	Programs
<b>Operating Systems</b>	Device drivers
<b>Architecture</b>	Instructions, Registers
<b>Micro architecture</b>	Datapath, Controllers
<b>Logic</b>	Adders, Memories
<b>Digital Circuits</b>	AND gates, NOT gates
<b>Analog Circuits</b>	Amplifiers
<b>Devices</b>	Transistors, Diodes
<b>Physics</b>	Electrons

**Discipline:** Intentionally restricting your design choices to that you can work more productively at higher abstraction levels.

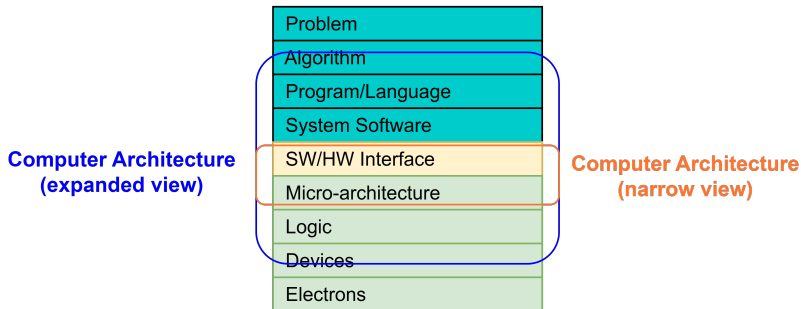
- **Digital Discipline:** We consider only two discrete values: 1's and 0's.
  - How Much Can We Do with 1s and 0s?

# The Three-Y's

- **Hierarchy**
  - A system is divided into modules of smaller complexity
- **Modularity**
  - Having well defined functions and interfaces
- **Regularity**
  - Encouraging uniformity, so modules can be easily re-used

# The Transformation Hierarchy

We are in the Logic and Microarchitecture section.



## The core of computer architecture:

**designing, selecting,** and **interconnecting** hardware components and the hardware/software interface to create a computing system that meets **functional, performance, energy consumption, cost,** and other specific goals.

## Why is it so Important Today?

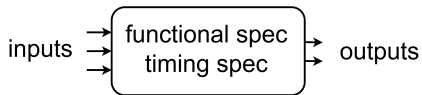
- Performance , Energy Efficiency, Sustainability
- **Reliability**, Safety, Security, Privacy
- New (Device) Technologies



## Logic Gates and Boolean Algebra

---

A logic circuit is composed of: inputs and outputs.

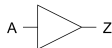


- Functional specification (describes relationship between inputs and outputs)
- Timing specification (describes the delay between inputs changing and outputs responding)

# Basic Logic Gates

They implement simple Boolean functions:

Buffer



A	Z
0	0
1	0

AND



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

OR



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

XOR



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

Inverter



A	Z
0	1
1	0

NAND



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

NOR



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

XNOR



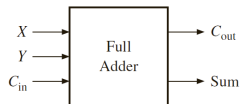
A	B	Z
0	0	1
0	1	0
1	0	0
1	1	1

# Functional Specification

- Functional Specification can be specified by a truth table.
- **Truth Table:** A tabular listing of function values for all possible combinations of values on its input arguments.
- Truth table is the unique signature of a Boolean function but, it is an expensive representation.
- If there are  $n$  inputs, there are  $2^n$  possible combinations.
- A Boolean function can have many alternative Boolean expressions.
- i.e., many alternative Boolean expressions (and gate realizations) may have the same truth table (and function).
- If they all specify the same thing, why do we care? Different Boolean expressions lead to different logic gate implementations → **different cost, latency, and energy properties.**
- **Canonical form:** standard form for a Boolean expression provides a unique algebraic signature.

## Two-Level Canonical (Standard) Forms

Derive algebraic expressions for Sum and  $C_{out}$  from the truth table of a Full Adder:



(a) Full adder module

$X$	$Y$	$C_{in}$	$C_{out}$	$Sum$
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

(b) Truth table

- Sum and  $C_{out}$  can be formed by ORing the **minterms** together.

$$Sum = \bar{X}\bar{Y}C_{in} + \bar{X}Y\bar{C}_{in} + X\bar{Y}\bar{C}_{in} + XYC_{in}$$

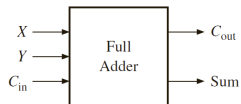
$$C_{out} = \bar{X}YC_{in} + X\bar{Y}C_{in} + XY\bar{C}_{in} + XYC_{in}$$

- These are **Sum of Products (SOP)** expressions.
- They can also be expressed as:

$$Sum = m_1 + m_2 + m_4 + m_7 = \sum m(1, 2, 4, 7)$$

$$C_{out} = m_3 + m_5 + m_6 + m_7 = \sum m(3, 5, 6, 7)$$

## Two-Level Canonical (Standard) Forms



(a) Full adder module

$X$	$Y$	$C_{in}$	$C_{out}$	$Sum$
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

(b) Truth table

- Sum and  $C_{out}$  can also be formed by ANDing the **maxterms**.

$$Sum = (X + Y + C_{in})(X + \bar{Y} + \bar{C}_{in})(\bar{X} + Y + \bar{C}_{in})(\bar{X} + \bar{Y} + C_{in})$$

$$C_{out} = (X + Y + C_{in})(X + Y + \bar{C}_{in})(X + \bar{Y} + C_{in})(\bar{X} + Y + C_{in})$$

- These are **Product of Sums (POS)** expressions.
- They can also be expressed as:

$$Sum = M_0 \cdot M_3 \cdot M_5 \cdot M_6 = \prod M(0, 3, 5, 6)$$

$$C_{out} = M_0 \cdot M_1 \cdot M_2 \cdot M_4 = \prod M(0, 1, 2, 4)$$

# Using Boolean Equations to Represent a Logic Circuits

Boolean equations enable us to do the following:

- Represent the function of a combinational logic block (Functional Specification)
- Methodically transform the function into simpler functions
  - which lead to different hardware realizations
  - Logic Minimization or Logic Simplification
  - We can automate this process Computer-Aided Design or Electronic Design Automation
  - Different Boolean expressions lead to different logic gate implementations → Different hardware area, cost, latency, energy properties

# Boolean Algebra and Algebraic Simplification

The basic mathematics used for logic design is Boolean algebra.

Summary of the laws and theorems of Boolean algebra:

Variable dominant rule	$X \cdot 1 = X, X + 0 = X$
Commutative rule	$X \cdot Y = Y \cdot X, X + Y = Y + X$
Complement rule	$X \cdot \bar{X} = 0, X + \bar{X} = 1$
Idempotency	$X \cdot X = X, X + X = X$
Identity Element	$X \cdot 0 = 0, X + 1 = 1$
Double negation	$\bar{\bar{X}} = X$
Associative rule	$X \cdot (Y \cdot Z) = (X \cdot Y) \cdot Z, X + (Y + Z) = (X + Y) + Z$
Distributive rule	$X \cdot (Y + Z) = X \cdot Y + X \cdot Z, X + Y \cdot Z = (X + Y)(X + Z)$
<b>Absorption</b>	$X \cdot (X + Y) = X \cdot X + X \cdot Y = X$ $X + X \cdot Y = X \cdot (1 + Y) = X$
<b>Adjacency</b>	$X \cdot Y + X \cdot \bar{Y} = X$ $(X + Y)(X + \bar{Y}) = X$
<b>Consensus</b>	$X \cdot Y + \bar{X} \cdot Z + Y \cdot Z = X \cdot Y + \bar{X} \cdot Z$ $(X + Y)(\bar{X} + Z)(Y + Z) = (X + Y)(\bar{X} + Z)$ <i>Corollary:</i> $(X + Y)(\bar{X} + Z) = \bar{X} \cdot Y + X \cdot Z$
<b>DeMorgan</b>	$X \cdot Y = \overline{\bar{X} + \bar{Y}}$ $X + Y = \overline{\bar{X} \cdot \bar{Y}}$
<b>Simplification</b>	$X \cdot (\bar{X} + Y) = X \cdot Y$ $X + \bar{X} \cdot Y = X + Y$



# Algebraic Simplifications

Four ways of simplifying a logic expression using the theorems are as follows:

1. Combining terms.
2. Eliminating terms.
3. Eliminating literals.
4. Adding redundant terms.

**Practice! Practice!**

# Simplification Examples

**Example 1:**

$$F = (A + \overline{B}C + D + EF)(A + \overline{B}C + \overline{D + EF})$$

**Example 2:**

$$F = \overline{(\overline{X + Y})Z + (X\overline{Y}Z)}$$

# Simplification Examples

## Example 1:

$$F = (A + \bar{B}C + D + EF)(A + \bar{B}C + \overline{D + EF})$$

$$F = (X + Y)(X + \bar{Y}),$$

$$\text{where } X = A + \bar{B}C, Y = D + EF$$

$$F = (X + Y)(X + \bar{Y}) = X$$

$$\rightarrow F = X = A + \bar{B}C$$

## Example 2:

$$F = \overline{(\bar{X} + Y)Z + (X\bar{Y}Z)}$$

# Simplification Examples

## Example 1:

$$F = (A + \bar{B}C + D + EF)(A + \bar{B}C + \overline{D + EF})$$

$$F = (X + Y)(X + \bar{Y}),$$

$$\text{where } X = A + \bar{B}C, Y = D + EF$$

$$F = (X + Y)(X + \bar{Y}) = X$$

$$\rightarrow F = X = A + \bar{B}C$$

## Example 2:

$$F = \overline{(\bar{X} + Y)Z + (X\bar{Y}Z)}$$

$$F = \overline{\bar{X}\bar{Y}Z + X\bar{Y}Z}$$

$$F = \overline{\bar{Y}Z(X + \bar{X})}$$

$$\rightarrow F = \overline{\bar{Y}Z} = Y + \bar{Z}$$

# Simplification Examples

**Example 3:**

$$F = x_1x_2 + \overline{x_1x_2} + x_2\overline{x_1}$$

**Example 4:**

$$F = \overline{A(B + \overline{C}) + \overline{A}}$$

# Simplification Examples

## Example 3:

$$F = x_1x_2 + \overline{x_1x_2} + x_2\overline{x_1}$$

$$F = x_1x_2 + \overline{x_1}(x_2 + \overline{x_2}) = x_1x_2 + \overline{x_1}$$

$$F = \overline{x_1} + x_1x_2 = (\overline{x_1} + x_1)(\overline{x_1} + x_2)$$

$$\rightarrow F = \overline{x_1} + x_2$$

## Example 4:

$$F = \overline{A(B + \overline{C}) + \overline{A}}$$

# Simplification Examples

## Example 3:

$$F = x_1x_2 + \overline{x_1x_2} + x_2\overline{x_1}$$

$$F = x_1x_2 + \overline{x_1}(x_2 + \overline{x_2}) = x_1x_2 + \overline{x_1}$$

$$\begin{aligned} F &= \overline{x_1} + x_1x_2 = (\overline{x_1} + x_1)(\overline{x_1} + x_2) \\ &\rightarrow F = \overline{x_1} + x_2 \end{aligned}$$

## Example 4:

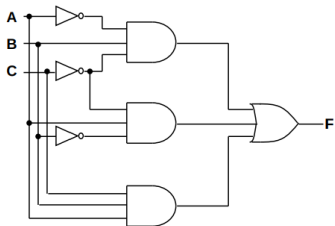
$$F = \overline{A(B + \overline{C})} + \overline{A}$$

$$\begin{aligned} F &= \overline{A(B + \overline{C})}.A = (\overline{A} + \overline{B + \overline{C}}).A \\ &\rightarrow F = \overline{(B + \overline{C})}.A = A\overline{B}C \end{aligned}$$

## Deriving Circuits Functions from Truth Tables

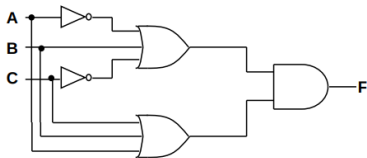
A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

$$F = \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + \bar{A}BC$$



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

$$F = (A + B + C)(\bar{A} + B + \bar{C})$$





# Logic Simplification using Karnaugh Maps (K-Maps)

- Karnaugh Map (K-map) method K-map is an alternative method of representing the truth table that helps visualize adjacencies in up to 6 dimensions.
- Physical adjacency means Logical adjacency

<i>A</i> \ <i>B</i>	0	1
0	00	01
1	10	11

3-Variable K-Map

<i>A</i> \ <i>BC</i>	00	01	11	10
0	000	001	011	010
1	100	101	111	110

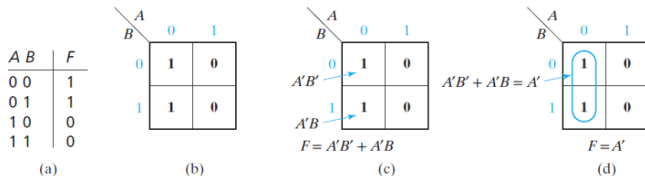
4-Variable K-Map

<i>AB</i> \ <i>CD</i>	00	01	11	10
00	0000	0001	0011	0010
01	0100	0101	0111	0110
11	1100	1101	1111	1110
10	1000	1001	1011	1010

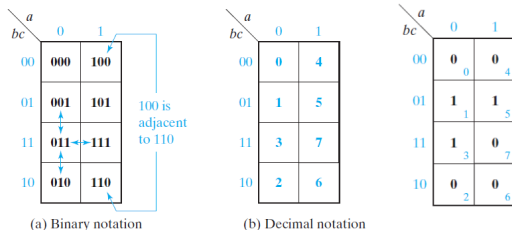
# Two- and Three-Variable Karnaugh Maps

## Two-variable K-Map:

- Minterms in adjacent squares of the map can be combined since they differ in only one variable.
- Thus,  $\overline{A}B$  and  $\overline{A}\overline{B}$  combine to form  $\overline{A}$ , and this is indicated by looping the corresponding 1's on the map in (d)

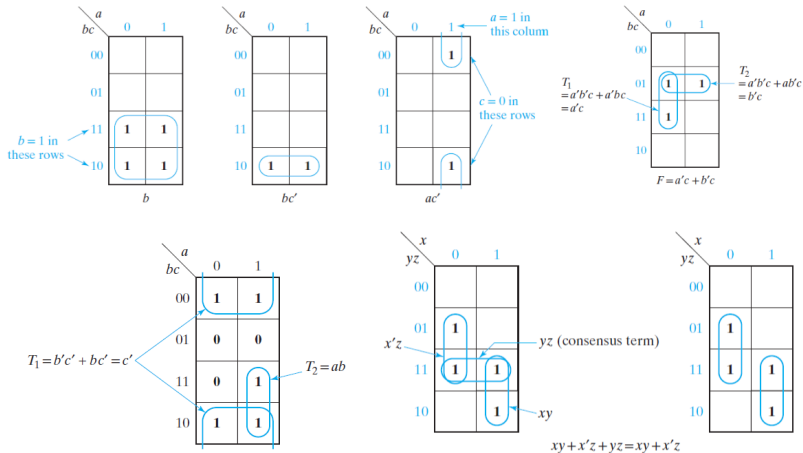


## Three-variable K-Map:



# Three-Variable Karnaugh Maps

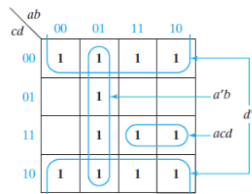
## Simplification of a Three-Variable Function



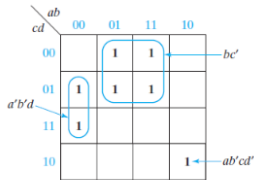
# Three-Variable Karnaugh Maps

## Location of Minterms on Four-Variable Karnaugh Map and Simplification of functions

AB \ CD	00	01	11	10
	0	4	12	8
00	0	4	12	8
01	1	5	13	9
11	3	7	15	11
10	2	6	14	10

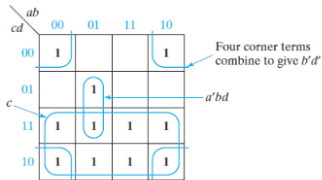


Plot of  $acd + a'b + d'$



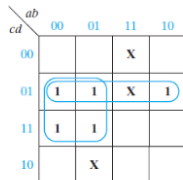
$$f_1 = \sum m(1, 3, 4, 5, 10, 12, 13) \\ = bc' + a'b'd + ab'cd'$$

(a)



$$f_2 = \sum m(0, 2, 3, 5, 6, 7, 8, 10, 11, 14, 15) \\ = c + b'd' + a'bd$$

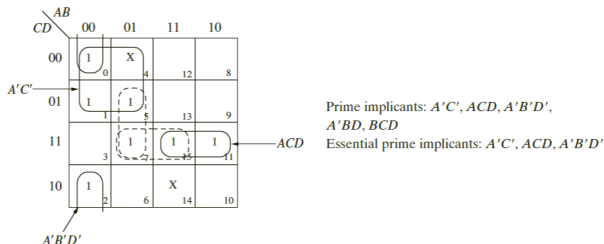
(b)



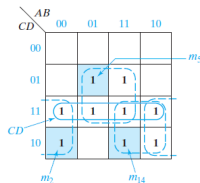
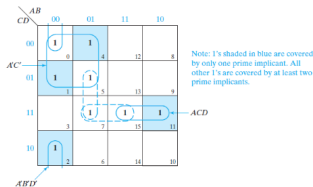
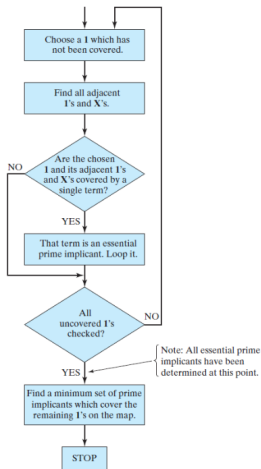
$$f = \sum m(1, 3, 5, 7, 9) + \sum d(6, 12, 13) \\ = a'd + c'd$$

# Determination of Minimum Expressions Using Essential Prime Implicants

- In choosing adjacent squares in a map, we must ensure that:
  - All the minterms of the function are covered when we combine the squares
  - The number of terms in the expression is minimized
  - There are no redundant terms (i.e., minterms already covered by other terms)
- A **prime implicant** is a product term obtained by combining the maximum possible number of adjacent squares in the map.
- If a minterm in a square is covered by only one prime implicant, that prime implicant is said to be **essential**.

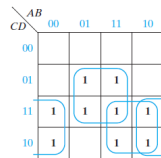


# Determination of Minimum Expressions Using Essential Prime Implicants



$$f = CD + BD + B'C + AC$$

(a)



$$f = BD + B'C + AC$$

(b)

**NOTE:** Simplification using K-Maps is limited. When the number of variables is large or if several functions must be simplified the computer-based **Quine-McCluskey** method is used.

## Designing With NAND and NOR Gates

- In many technologies, implementation of **NAND** gates or **NOR** gates is easier than that of AND and OR gates.
- The bubble at a gate input or output indicates a complement.
- Any logic function can be realized using only NAND gates or only NOR gates.

NAND:

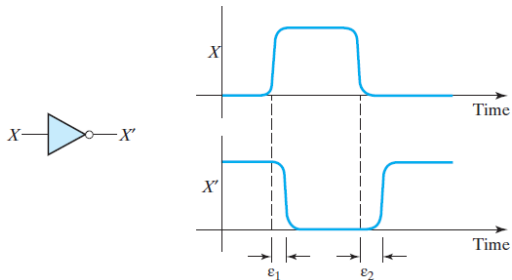


NOR:



## Gate Delays and Timing Diagrams

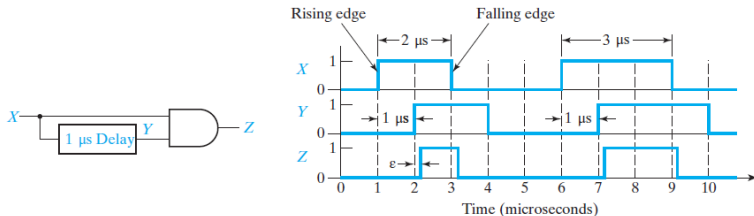
- When the input to a logic gate is changed, the output will not change instantaneously.
- The transistors or other switching elements within the gate take a finite time to react to a change in input-  
propagation delay.





# Timing Diagrams

## Timing Diagram for Circuit with Delay



**Hazard:** An unintended output glitch in a combinational circuit caused by unequal path propagation delays when inputs change:

- **Static-1 hazard:** Output momentarily falls to 0 when it should remain 1.
- **Static-0 hazard:** Output momentarily rises to 1 when it should remain 0.
- **Dynamic hazard:** During an intended  $0 \rightarrow 1$  or  $1 \rightarrow 0$  change, the output toggles multiple times (three or more) before settling.

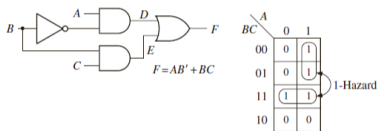
# Hazards in Combinational Circuits



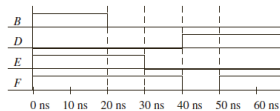
(a) Simple circuit with static 1-hazard



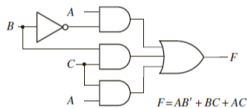
(b) Simple circuit with static 0-hazard



(a) Circuit with 1-hazard



(b) Timing chart



(c) Circuit with hazard removed

	A	BC	0	1
00			0	1
01			0	1
11			1	1
10			0	0

## Combinational Logic

---

- **Combinational Logic**

- Memoryless
- Outputs are strictly dependent on the combination of input values that are being applied to circuit right now

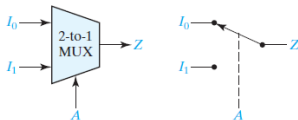
- **Sequential Logic**

- Has memory - Can "store" data values
- Outputs are determined by previous (historical) and current values of inputs

# Multiplexers

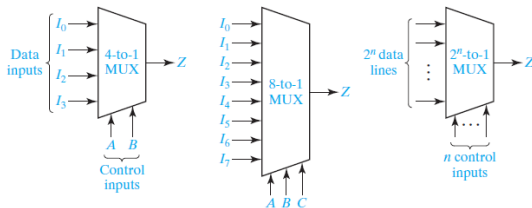
**Multiplexers:** act as data routing switches, needed in many applications.

2-to-1 Multiplexer and Switch Analog  $Z = A'I_0 + AI_1$

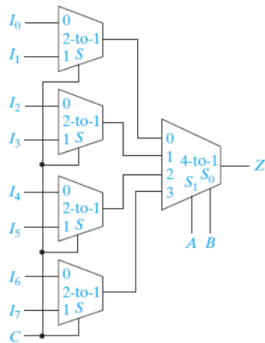
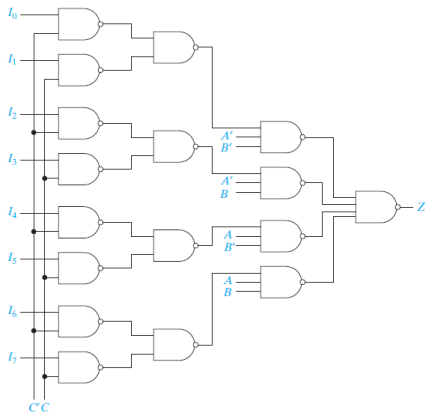


**Multiplexers**

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

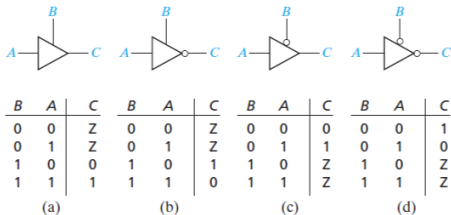
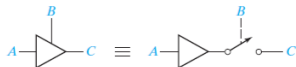


## Implementation of an 8-to-1 MUX.

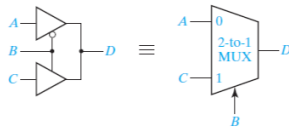


# Three State Buffers

- A gate output can only be connected to a limited number of other device inputs without degrading the performance of a digital system.
- A simple buffer may be used to increase the driving capability of a gate output.

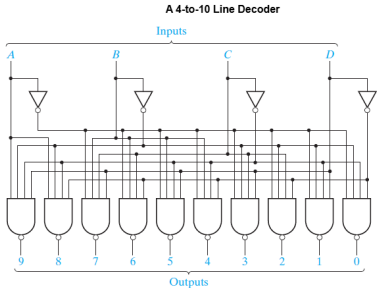
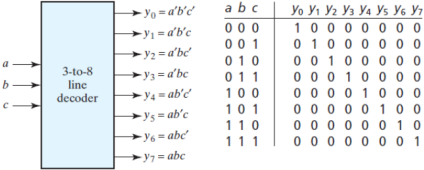


Data Selection Using Three-State Buffers

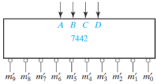


# Decoders and Encoders

Consider a **A 3-to-8 Line Decoder**: it generates all of the minterms of the three input variables.



(a) Logic diagram



(b) Block diagram

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

(c) Truth Table



## Review the following cases:

- Full-adder
- Carry-ripple adder
- Faster adders:
  - Manchester
  - Carry-lookahead
  - Kogge-stone tree
- Adder arrays
- Parallel multiplier
- Comparators (equality and greater-than/equal-to)
- Arithmetic logic unit (ALU)

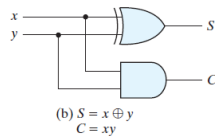
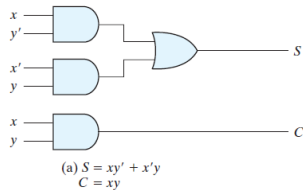
## Half Adder

Half Adder

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

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

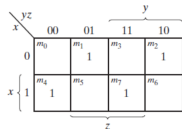
$$C = xy$$



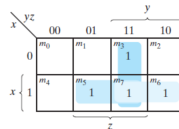
## Full Adder

Full Adder

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

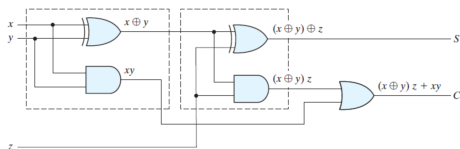
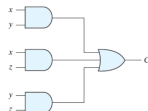
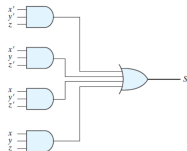


$$(a) S = x'y'z + x'yz' + xy'z' + xyz$$



$$(b) C = xy + xz + yz$$

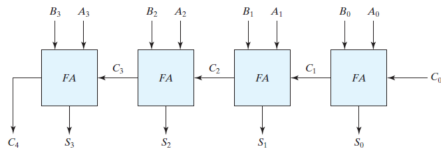
$$\begin{aligned}
 S &= z \oplus (x \oplus y) \\
 &= z'(xy' + x'y) + z(xy' + x'y)' \quad C = z(xy' + x'y) + xy = xy'z + x'yz + xy \\
 &= z'(xy' + x'y) + z(xy + x'y') \\
 &= xy'z' + x'yz' + xyz + x'y'z
 \end{aligned}$$



# Binary Adder-Subtractor

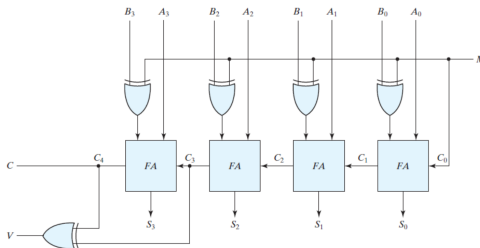
A binary adder produces the arithmetic sum of two binary numbers.

It can be constructed with full adders connected in cascade



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	$C_{i+1}$

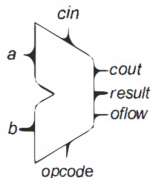
The subtraction of unsigned binary numbers can be done most conveniently by means of complements subtraction  $A - B$  can be done by taking the 2's complement of  $B$  and adding it to  $A$ .



# Arithmetic Logic Unit

**ALU circuits are at the core of any microprocessor or microcontroller.**

Its capable of computing several logic as well as arithmetic functions.



(a)

(b)

Unit	opcode	Instruction	result
Logic	0000	Complement <i>a</i>	not <i>a</i>
	0001	Complement <i>b</i>	not <i>b</i>
	0010	Logic and	<i>a</i> and <i>b</i>
	0011	Logic or	<i>a</i> or <i>b</i>
	0100	Logic nand	<i>a</i> nand <i>b</i>
	0101	Logic nor	<i>a</i> nor <i>b</i>
	0110	Logic xor	<i>a</i> xor <i>b</i>
	0111	Logic xnor	<i>a</i> xnor <i>b</i>
Arithmetic	1000	Transfer <i>a</i>	<i>a</i>
	1001	Transfer <i>b</i>	<i>b</i>
	1010	Increment <i>a</i>	<i>a</i> + 1
	1011	Increment <i>b</i>	<i>b</i> + 1
	1100	Decrement <i>a</i>	<i>a</i> - 1
	1101	Decrement <i>b</i>	<i>b</i> - 1
	1110	Add <i>a</i> and <i>b</i>	<i>a</i> + <i>b</i>
	1111	Add <i>a</i> and <i>b</i> with carry	<i>a</i> + <i>b</i> + <i>cin</i>

**See you next time!**