Introduction to Logic Circuits

VOLKAN KURSUN

Some material from McGraw Hill

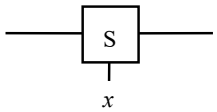
Outline

- Variables, Functions, and Truth Tables
- Logic Gates and Networks
- Boolean Algebra
- The Venn Diagram
- Synthesis Using AND, OR, and NOT Gates
- Sum-of-Products and Product-of-Sums Forms
- NAND and NOR Logic Networks
- XOR and XNOR
- Multiplexer and Tri-State Driver
- Introduction to VHDL

A Binary Switch

- The dominance of binary circuits in digital systems is a consequence of their simplicity: signals assume only two possible values in steady-state
- The simplest binary element is a switch that has two states
- Assume a switch is controlled by an input signal x
  - Switch is open if x = 0
  - Switch is closed if x = 1

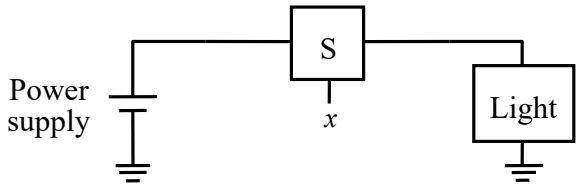
(a) Two states of a switch



(b) Symbol for a switch

A Light Controlled By One Switch

- A simple application where the switch turns a lightbulb on or off: the lightbulb glows when sufficient current passes through it
- The current flows when the switch is closed: when the **switch control input x** = 1
- The **output is the state (condition) of the lightbulb: L**
- If x = 1, the light is ON, L = 1
- If x = 0, the light is OFF, L = 0



(on)
L = 1 if x = 1
(off)
L = 0 if x = 0
L = x

- This simple logic expression describes the output as a function of the input
- L(x) = x is a **logic function** and **x is an input variable**

Light Controlled By Two Series Switches

- The light will be turned ON only if both switches are closed
- If either switch is open, the light is OFF

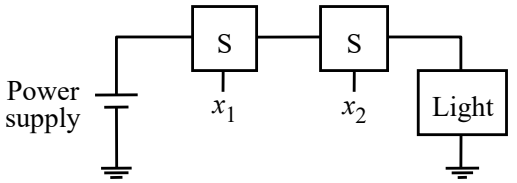
AND operator ↑

- $L(x_1, x_2) = x_1 \cdot x_2$

AND logic function with two input variables

- L = 1 if x_1 = 1 AND x_2 = 1,
- L = 0 otherwise

The logical AND function (series connection)

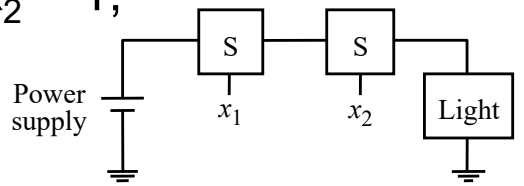


Two-input AND Function TRUTH TABLE

- $L(x_1, x_2) = x_1 \cdot x_2$

AND function with two input variables

- L = 1 if x_1 = 1 **AND** x_2 = 1,
- L = 0 otherwise



- **All possible inputs and outputs are listed in a truth table**
- For a logic function with 2 inputs, there are 2^2 = 4 different input combinations: 4 rows in the truth table
- **For a logic function with n inputs, there are 2^n different input combinations: 2^n rows in the truth table**

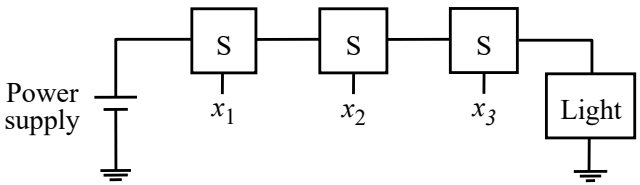
x_1	x_2	$x_1 \cdot x_2$
0	0	0
0	1	0
1	0	0
1	1	1

Three-input AND Function TRUTH TABLE

- $L(x_1, x_2, x_3) = x_1 \cdot x_2 \cdot x_3$

AND function with three input variables

- L = 1 if x_1 = 1 **AND** x_2 = 1 **AND** x_3 = 1
- L = 0 otherwise



For a logic function with 3 inputs, there are 2^3 = 8 different input combinations: 8 rows in the truth table

x_1	x_2	x_3	$x_1 \cdot x_2 \cdot x_3$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Light Controlled By Two Parallel Switches

- The light will be turned ON if either x_1, or x_2, or both switches are closed
- If both switches are open, the light is OFF

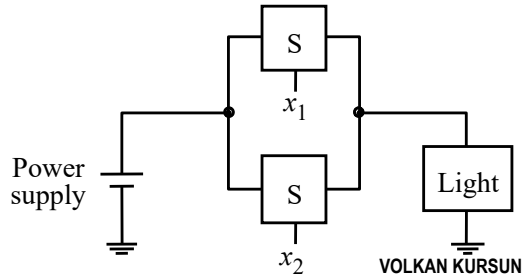
OR operator ↑

- $L(x_1, x_2) = x_1 + x_2$

OR logic function with two input variables

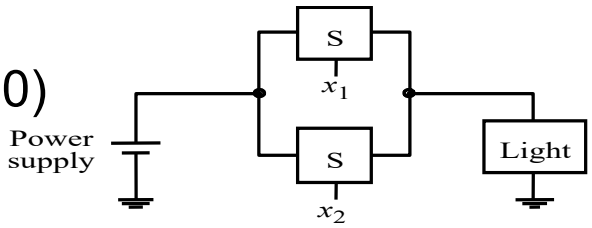
- L = 1 if x_1 = 1 **OR** x_2 = 1 **OR** x_1 = x_2 = 1
- L = 0 if x_1 = x_2 = 0

The logical OR function (parallel connection)



Two-Input OR Function TRUTH TABLE

- $L(x_1, x_2) = x_1 + x_2$
- L = 1 if (x_1 = 1, x_2 = 0) OR (x_2 = 1, x_1 = 0) OR (x_1 = x_2 = 1)
- L = 0 if (x_1 = x_2 = 0)



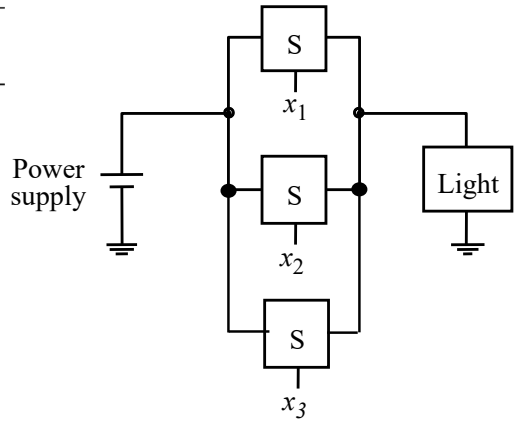
x_1	x_2	$x_1 + x_2$
0	0	0
0	1	1
1	0	1
1	1	1

- **All possible inputs and outputs are listed in a truth table**
- For a logic function with 2 inputs, there are 2^2 = 4 different input combinations: 4 rows in the truth table
- For a logic function with n inputs, there are 2^n different input combinations: 2^n rows in the truth table

Three-Input OR Function TRUTH TABLE

- $L(x_1, x_2, x_3) = x_1 + x_2 + x_3$

x_1	x_2	x_3	$x_1 + x_2 + x_3$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1



For a logic function with 3 inputs, there are 2^3 = 8 different input combinations: 8 rows in the truth table

Light Controlled By Three Switches

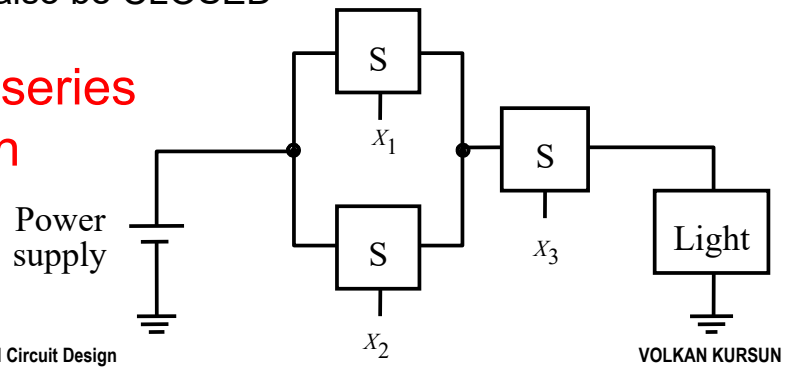
- The **parallel-series connection** of switches realizes the logic function:

$L(x_1, x_2, x_3) = (x_1 + x_2) \cdot x_3$

Logic function with three input variables

- L = 1 if x_3 = 1 **AND** (x_1 **OR** x_2) = 1
- For the light to turn ON, switch controlled by x_3 must be CLOSED **AND** at least one of the switches controlled by x_1 **OR** x_2 MUST also be CLOSED

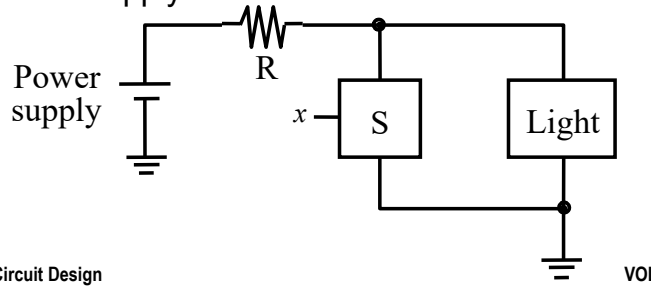
A parallel-series connection



INVERSION

- Can we design a logic circuit where the light is turned ON when a switch is opened?
- Connect the switch in parallel with the light
- **Assumption: the switch resistance is much smaller than the light bulb**
- When the switch is closed, the switch prevents the current flowing through the light (switch short-circuits the light): light is TURNED OFF
- When the switch is open, the current flows through the light bulb and the light is TURNED ON
- An extra resistor is used to ensure that the closed switch does not short-circuit the power supply

An inverting circuit



INVERSION (COMPLEMENT) FUNCTION

- L(x) = $\bar{x}$

where L = 1 if x = 0, L = 0 if x = 1

x	L
0	1
1	0

- The value of the function is inverse (complement) of the input variable: this is commonly called the **NOT** operation
- When typing the complement by keyboard, overbars cannot be used. Instead, use the following notations which are easier:

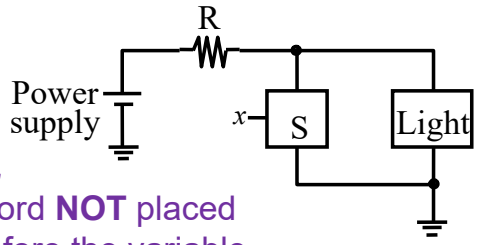
$\bar{x} = x' = !x = \sim x = \text{NOT x}$

Apostrophe

Exclamation mark

Tilde

Word **NOT** placed before the variable



TRUTH TABLES TO PROVE PROPERTIES

$f = (x_1 \cdot x_2) \cdot x_3$ $g = x_1 \cdot (x_2 \cdot x_3)$

f = g (associative property)

x_1	x_2	x_3	$x_1 \cdot x_2$	f	$x_2 \cdot x_3$	g
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	1	0
1	0	0	0	0	0	0
1	0	1	0	0	0	0
1	1	0	1	0	0	0
1	1	1	1	1	1	1

general form:

input variables x_1 x_2 --- x_n	output $Y = f(x_1, \ldots, x_n)$

2^n rows

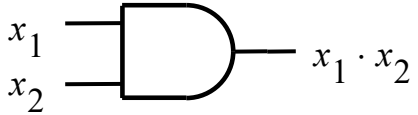
Outline

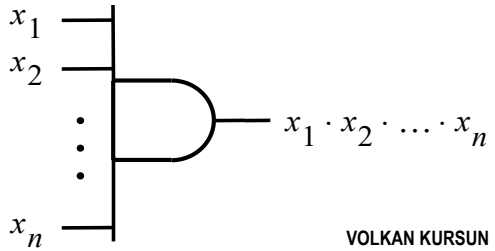
- Variables, Functions, and Truth Tables
- Logic Gates and Networks
- Boolean Algebra
- The Venn Diagram
- Synthesis Using AND, OR, and NOT Gates
- Sum-of-Products and Product-of-Sums Forms
- NAND and NOR Logic Networks
- XOR and XNOR
- Multiplexer and Tri-State Driver
- Introduction to VHDL

Logic Gates

- A logic gate has **one or more inputs** and **one output** that is a function of the inputs
- The three basic logic functions (AND, OR, and NOT) introduced in previous section can be used to implement logic functions of any complexity
- A complex function would require a network of many logic gates that can individually perform these basic logic functions for implementation
- Schematic: a circuit diagram consisting of graphical symbols representing logic gates

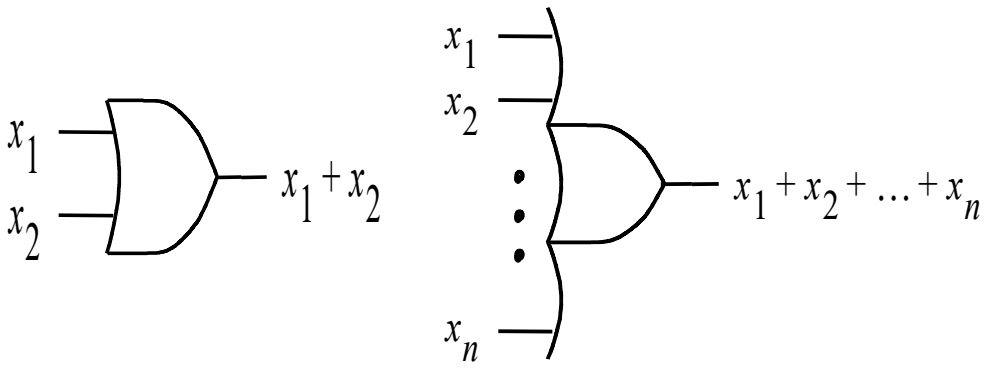
Graphical symbols representing **AND gates**:



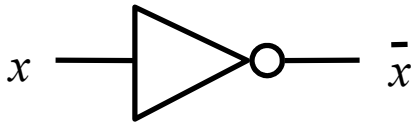


Logic Gates: OR, NOT

- Graphical symbols representing OR gates:

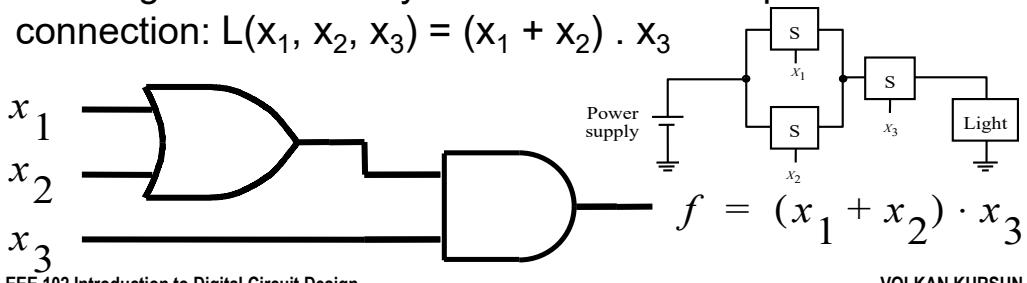


- Graphical symbol representing inverter (NOT gate):



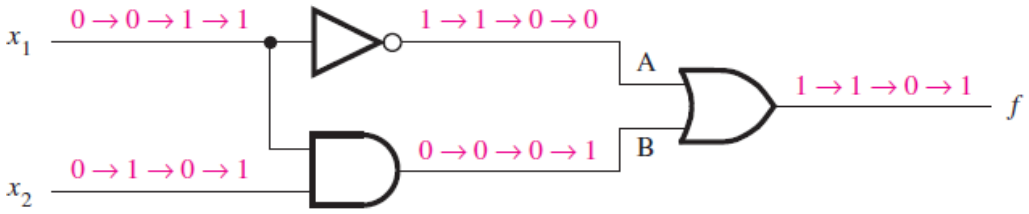
Networks of Logic Gates

- Larger circuits are implemented as networks of logic gates: called a **logic network** or **logic circuit**
- A given logic function can be implemented with a number of different logic networks
- **Complexity** of a given network has a direct impact on **cost**: find ways to implement functions with **simpler logic networks to lower the cost**
- Example: the logic network that implements the function of the light controlled by three switches with parallel-series connection: $L(x_1, x_2, x_3) = (x_1 + x_2) \cdot x_3$



Analysis versus Synthesis of Logic Networks

- **Synthesis**: designing a new logic network that implements a desired function
- **Analysis**: determine the function performed by an existing logic network
- Analysis example: consider the following network of 3 logic gates

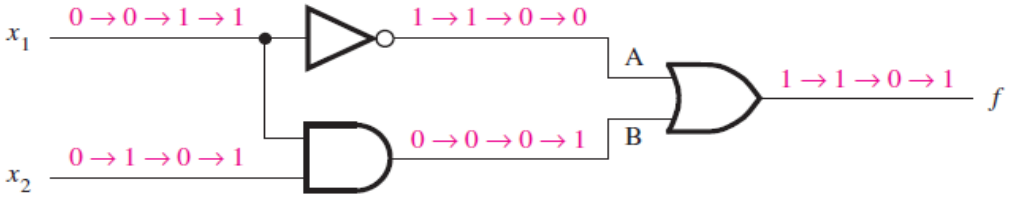


(a) Network that implements $f = \bar{x}_1 + x_1 \cdot x_2$

- To analyze the functional behavior, consider what happens if all possible input combinations are applied: form the truth table (next slide)

Truth Table for Network Analysis

- To analyze the functional behavior, consider what happens if all possible input combinations are applied: form the truth table



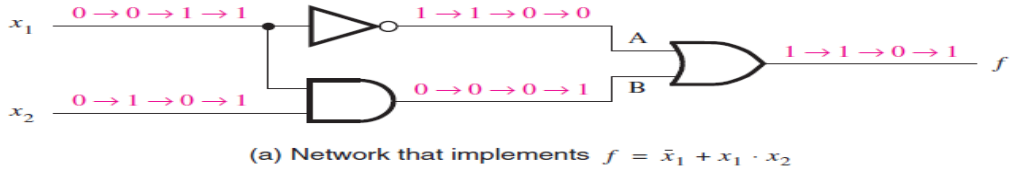
(a) Network that implements $f = \bar{x}_1 + x_1 \cdot x_2$

x_1	x_2	$f(x_1, x_2)$	A	B
0	0	1	1	0
0	1	1	1	0
1	0	0	0	0
1	1	1	0	1

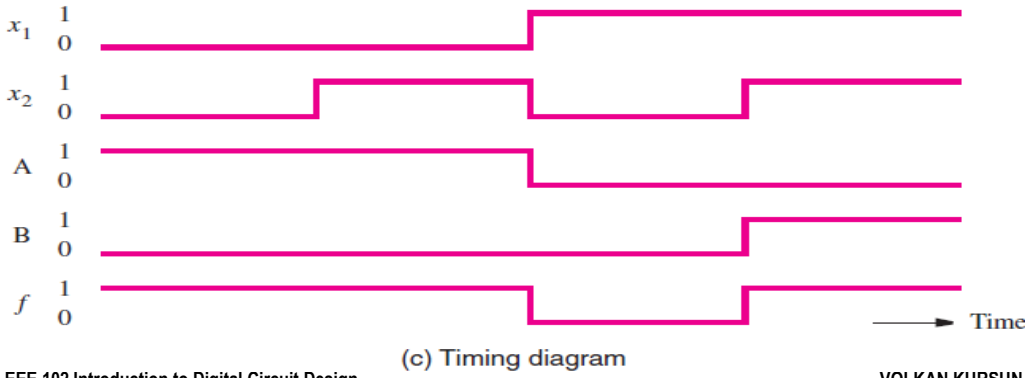
(b) Truth table

Timing Diagram for Network Analysis

- Perform graphical analysis with timing diagrams: waveforms of the inputs (x_1, x_2), output (f), and internal nodes (A, B) are shown



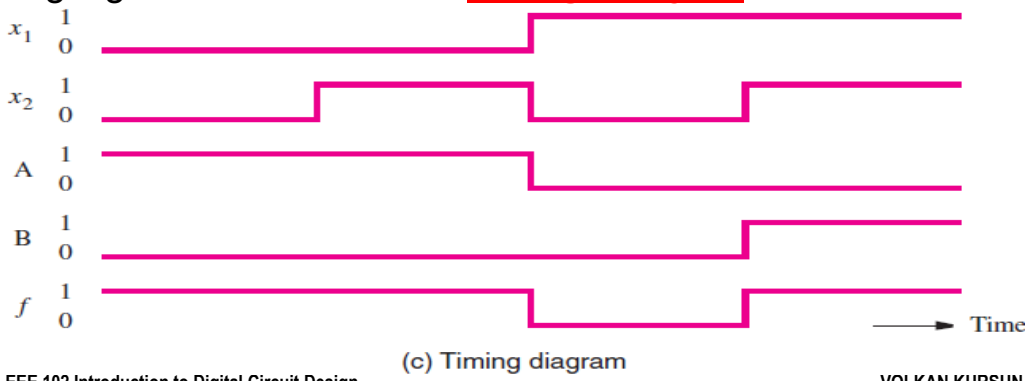
(a) Network that implements $f = \bar{x}_1 + x_1 \cdot x_2$



(c) Timing diagram

Timing Diagram for Network Analysis

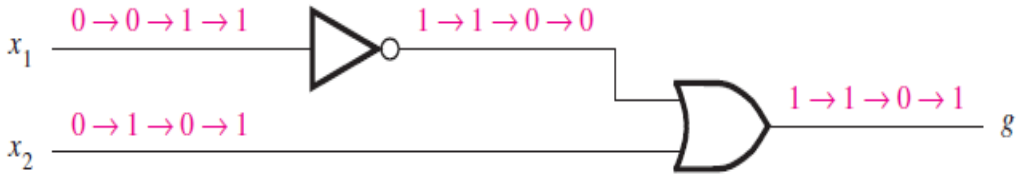
- These idealized waveforms assume that the logic gates respond to changes in inputs instantaneously (zero delay)
- Such **idealized timing diagrams** that do not consider delay are **useful for only functional analysis**
- Real logic circuits have non-zero propagation delays and more **realistic timing diagrams** that **consider delays** of logic gates are needed for **timing analysis**



(c) Timing diagram

Functionally Equivalent Networks

- Consider the following network:

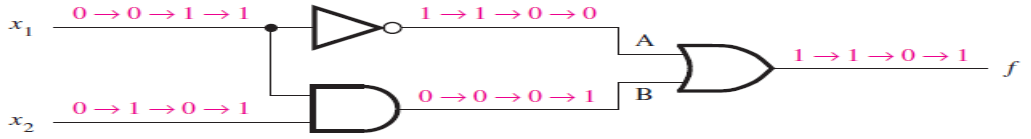


(d) Network that implements $g = \bar{x}_1 + x_2$

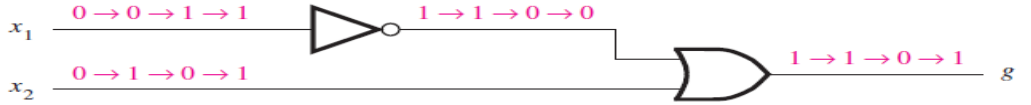
- By performing analysis using truth tables and/or timing diagrams, you can show that the output g changes in exactly the same way as f does in the previous example
- Therefore, g(x_1, x_2) = f(x_1, x_2): **the two networks are functionally equivalent**
- When two networks are functionally equivalent, you should **choose the simpler network to lower the cost**

Algebraic Rules to Simplify Logic Functions

- In this example, the equivalence of the following two networks was established through construction of truth tables



(a) Network that implements $f = \bar{x}_1 + x_1 \cdot x_2$

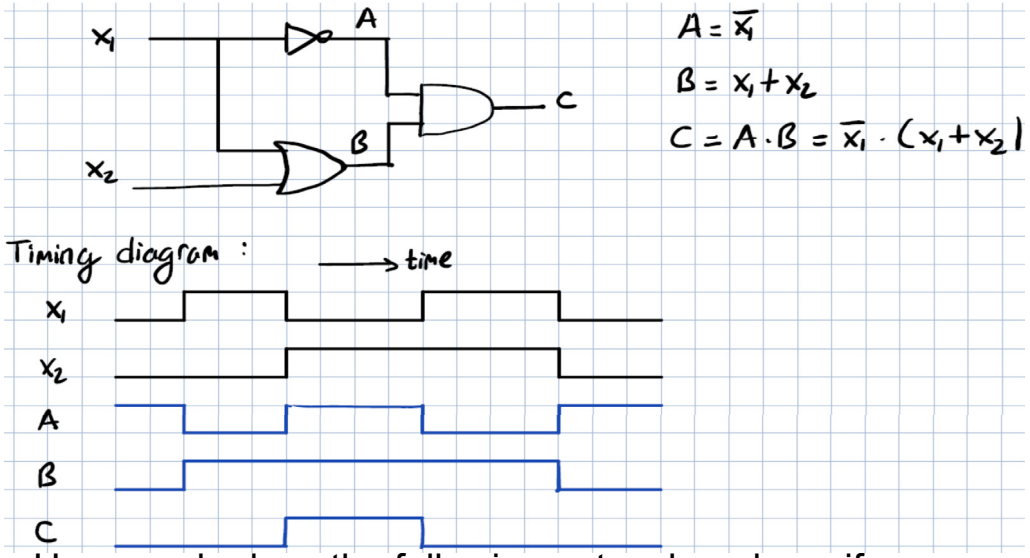


(d) Network that implements $g = \bar{x}_1 + x_2$

- The equivalence can also be shown through algebraic manipulation of the logic expressions
- Rules that can be used to show the following equivalence will be seen soon

$$\bar{x}_1 + x_1 \cdot x_2 = \bar{x}_1 + x_2$$

Functionally Equivalent Networks



- Homework: draw the following network and see if you can prove that it is equivalent to the network given above

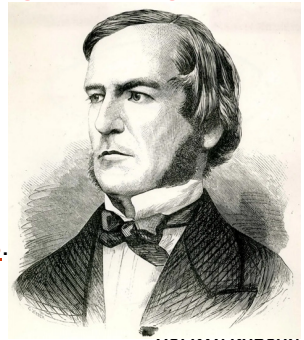
$$\overline{x_1} \cdot x_2 = \overline{x_1} \cdot (x_1 + x_2)$$

Outline

- Variables, Functions, and Truth Tables
- Logic Gates and Networks
- Boolean Algebra
- The Venn Diagram
- Synthesis Using AND, OR, and NOT Gates
- Sum-of-Products and Product-of-Sums Forms
- NAND and NOR Logic Networks
- XOR and XNOR
- Multiplexer and Tri-State Driver
- Introduction to VHDL

Boolean Algebra

- Boolean algebra: **George Boole established the algebraic description of processes in logical thought and thinking in 1840s**. Developing novel ideas on **logical method** and confident in the **symbolic reasoning** he had derived from his mathematical investigations, he published in 1847 a pamphlet, "The Mathematical Analysis of Logic, being an Essay towards a Calculus of Deductive Reasoning," in which **he argued persuasively that logic should be allied with mathematics, not philosophy**
- In 1854 he published "An Investigation into the Laws of Thought, on Which Are Founded the Mathematical Theories of Logic and Probabilities"
- With Boole's publications in 1847 and 1854, began the **algebra of logic**, or what is now called Boolean algebra

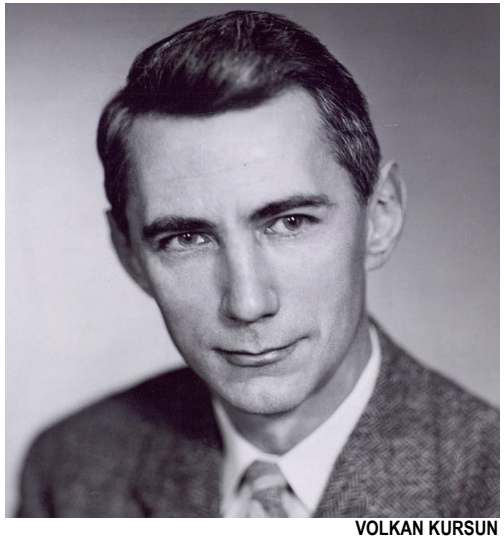


George Boole (born November 2, 1815, Lincoln, Lincolnshire, England—died December 8, 1864, Ballintemple, County Cork, Ireland) English mathematician who helped establish modern symbolic logic and whose algebra of logic, now called Boolean algebra, is basic to the design of digital computer circuits.

*From Encyclopaedia Britannica

Boolean Algebra and Logic Circuits

- **Claude Shannon** demonstrated that the Boolean algebra can be used to describe logic circuits in 1930s. Shannon's master's thesis, *A Symbolic Analysis of Relay and Switching Circuits* (1940), used Boolean algebra to establish the theoretical foundations of digital circuits.



Claude Shannon (born April 30, 1916, Petoskey, Michigan, U.S.—died February 24, 2001, Medford, Massachusetts) American mathematician and electrical engineer who laid the theoretical foundations for digital circuits and information theory, a mathematical communication model.

*From Encyclopaedia Britannica

Principle of Duality

- **Dual expressions**: dual of a logic expression is obtained by replacing all + operators with . operators AND all . operators with + operators AND replacing all 0s with 1s AND all 1s with 0s
- **Principle of duality**: the dual of any true statement (axiom or theorem) in Boolean algebra is also true
- Principle of duality implies that **at least two different ways exist to express every logic function** with Boolean algebra: **often** one expression that leads to a simpler physical implementation is preferable to the dual expression

Axioms of Boolean Algebra

- Boolean algebra is based on a set of rules that are derived from a small number of assumptions (axioms)

1) $0 \cdot 0 = 0$ $\quad 1+1=1$

2) $1 \cdot 1 = 1$ $\quad 0+0=0$

3) $0 \cdot 1 = 1 \cdot 0 = 0$ $\quad 1+0 = 0+1 = 1$

4) if $x=0$ then $\bar{x}=1$ if $x=1$ then $\bar{x}=0$

duality: replace + with ·, · with +, $0 \leftrightarrow 1$, then all statements remain valid.

★ Precedence of operations: in absence of parenthesis NOT > AND > OR

Axiom (AKA **proposition**, **postulate**, accepted truth, general truth, **assumption**): a statement or proposition (onerme) which is regarded as being **established**, **accepted**, or **self-evidently true**, to serve as a premise or starting point for further reasoning and arguments.

Single Variable Theorems

- These **theorems** can be **proven with induction using the axioms**: substitute the values x = 0 and x = 1 into the expressions and use the axioms for proof

Single variable theorems

5)	$x \cdot 0 = 0$	$x+1=1$	null elements
6)	$x \cdot 1 = x$	$x+0=x$	identities
7)	$x \cdot x = x$	$x+x=x$	idempotency
8)	$x \cdot \bar{x} = 0$	$x+\bar{x}=1$	complements
9)	$\bar{\bar{x}} = x$		involution

Proof of 6 by truth table

x	$x \cdot 1$	$x+0$
0	$0 \cdot 1 = 0$	$0+0=0$
1	$1 \cdot 1 = 1$	$1+0=1$

Theorem (kuram): a general proposition **not self-evident but proved by a chain of reasoning**; a truth established by means of accepted truths.

Two- and Three-Variable Properties

- The validity of these properties (algebraic identities) can be proven either by induction or by algebraic manipulation

10)	$x \cdot y = y \cdot x$	$x+y = y+x$	commutation
11)	$x \cdot (y \cdot z) = (x \cdot y) \cdot z$	$x+(y+z) = (x+y)+z$	association
12)	$x \cdot (y+z) = xy + xz$	$x+yz = (x+y) \cdot (x+z)$	distribution
13)	$x + x \cdot y = x$	$x \cdot (x+y) = x$	absorption
14)	$x \cdot y + x \cdot \bar{y} = x$	$(x+y) \cdot (x+\bar{y}) = x$	combining
15)	$\overline{x \cdot y} = \bar{x} + \bar{y}$	$\overline{(x+y)} = \bar{x} \cdot \bar{y}$	DeMorgan's theorem

- Example: use induction with truth table to prove DeMorgan's theorem

x	y	$x \cdot y$	$\overline{x \cdot y}$	$\bar{x}$	$\bar{y}$	$\bar{x} + \bar{y}$
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

Two- and Three-Variable Properties

16) $x + \bar{x}y = x + y$ $\quad x \cdot (\bar{x} + y) = x \cdot y$

17) $x \cdot y + y \cdot z + \bar{x} \cdot z = x \cdot y + \bar{x} \cdot z$ $\quad (x+y) \cdot (y+z) \cdot (\bar{x}+z) = (x+y) \cdot (\bar{x}+z)$

consensus

Huntington's basic postulates : 5, 8, 10, 12 (other identities can be derived from them)

Postulate (AKA axiom, assumption): a thing suggested or assumed as true as the basis for reasoning, discussion, or belief.

Example: Show that

$$Q = XY + YZ + \bar{Y} = X + \bar{Y} + Z$$

$$= Y(X+Z) + \bar{Y} \quad (10 \text{ and } 12)$$

$$= \bar{Y} + X + Z \quad (16)\ a + a'b = a + b$$

Axioms, Theorems, and Properties of Boolean Algebra

- Axioms, theorems, and properties of Boolean algebra **provide the basis for automating the synthesis of logic functions with the CAD tools**

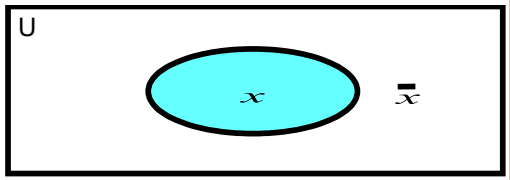
Outline

- Variables, Functions, and Truth Tables
- Logic Gates and Networks
- Boolean Algebra
- The Venn Diagram
- Synthesis Using AND, OR, and NOT Gates
- Sum-of-Products and Product-of-Sums Forms
- NAND and NOR Logic Networks
- XOR and XNOR
- Multiplexer and Tri-State Driver
- Introduction to VHDL

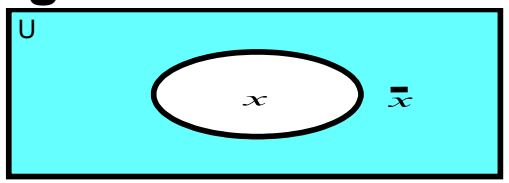
Venn Diagrams

- Although induction can be used to prove theorems and properties of Boolean algebra, the induction procedure is quite tedious
- Venn diagrams provide an **alternative more intuitive and easier way to prove the theorems and properties of Boolean algebra**
- Venn diagrams are particularly useful for providing **intuitive understanding of how two expressions may be equivalent**
- Represent a variable x by a circle such that the **area inside the circle corresponds to x = 1** while the **area outside the circle corresponds to x = 0**
- **Universal set (U):** a larger set that contains all of the elements in all of the sets being considered (an outer box –or rectangle- that includes all the sets)
- **Inverse of universal set (U'):** empty set (null set) {}
- An expression involving one or more variables is depicted by **shading the area where the value of the expression is 1**

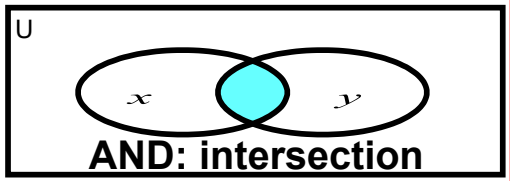
Venn Diagrams



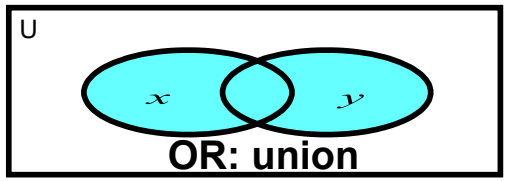
(c) Variable x



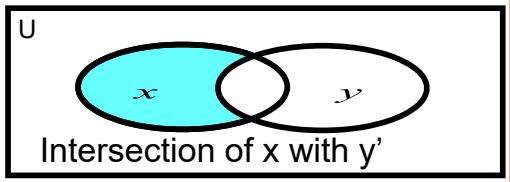
(d) $\bar{x}$



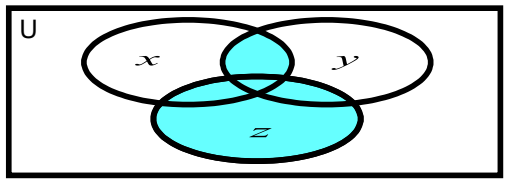
(e) $x \cdot y$



(f) $x + y$



(g) $x \cdot \bar{y}$

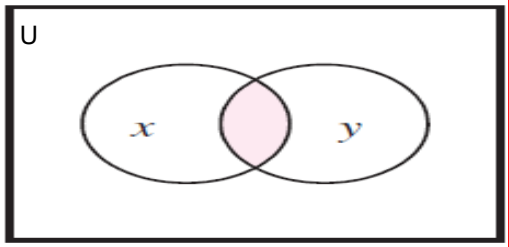


(h) $x \cdot y + z$

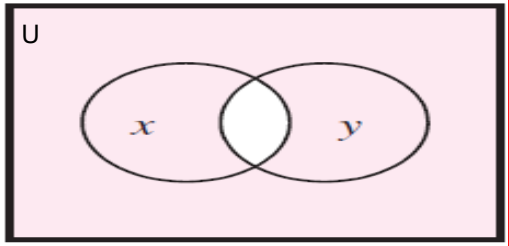
Union of z with the intersection of x and y

Venn Diagrams to Verify Equivalence

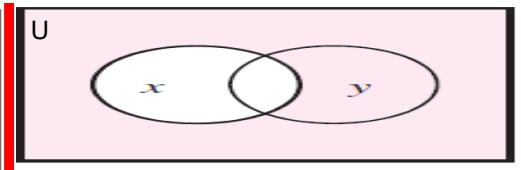
Example-1: verification of DeMorgan's theorem



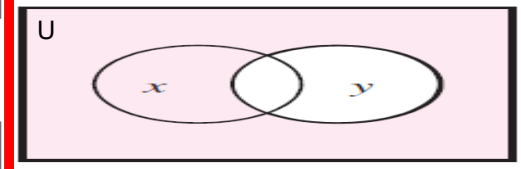
(a) $x \cdot y$



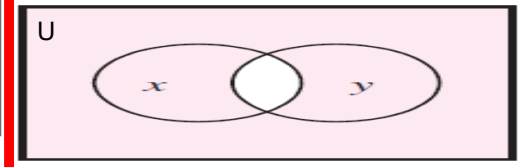
(b) $\overline{x \cdot y}$



(c) $\bar{x}$



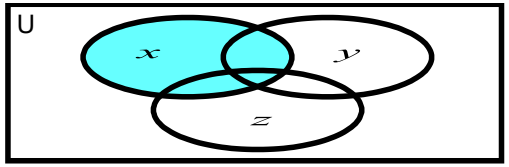
(d) $\bar{y}$



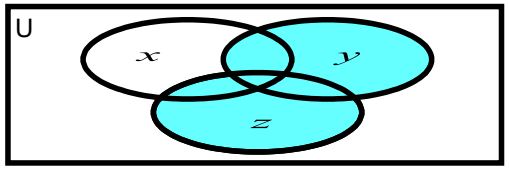
(e) $\bar{x} + \bar{y}$

Venn Diagrams to Verify Equivalence

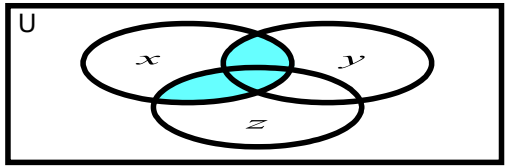
Example-2: verification of the distributive property $x \cdot (y + z) = x \cdot y + x \cdot z$



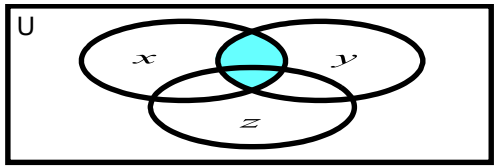
(a) x



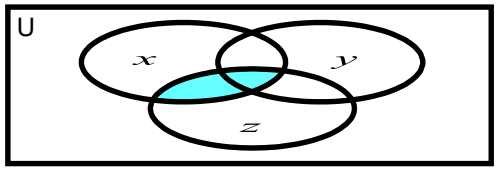
(b) $y + z$



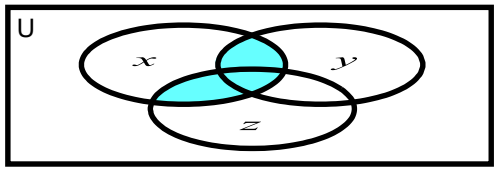
(c) $x \cdot (y + z)$



(d) $x \cdot y$



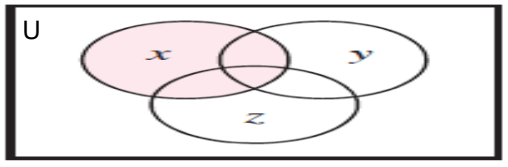
(e) $x \cdot z$



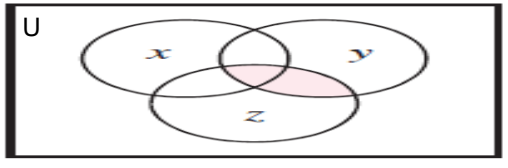
(f) $x \cdot y + x \cdot z$

Venn Diagrams to Verify Equivalence

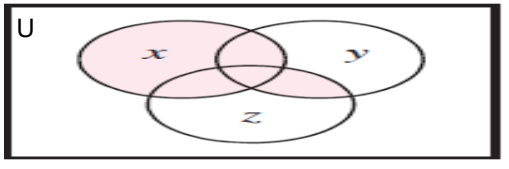
Example-3: verification of the distributive property $x + (y \cdot z) = (x + y) \cdot (x + z)$



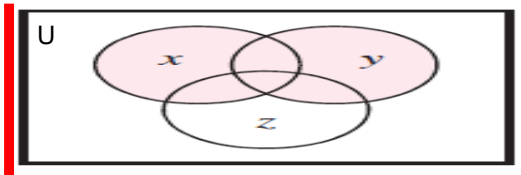
(a) x



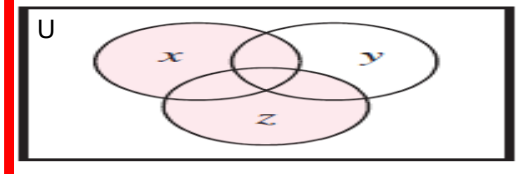
(b) $y \cdot z$



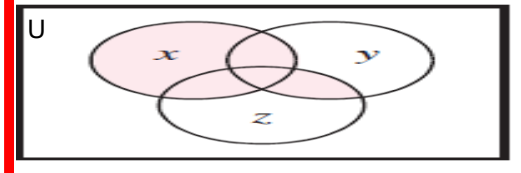
(c) $x + y \cdot z$



(d) $x + y$



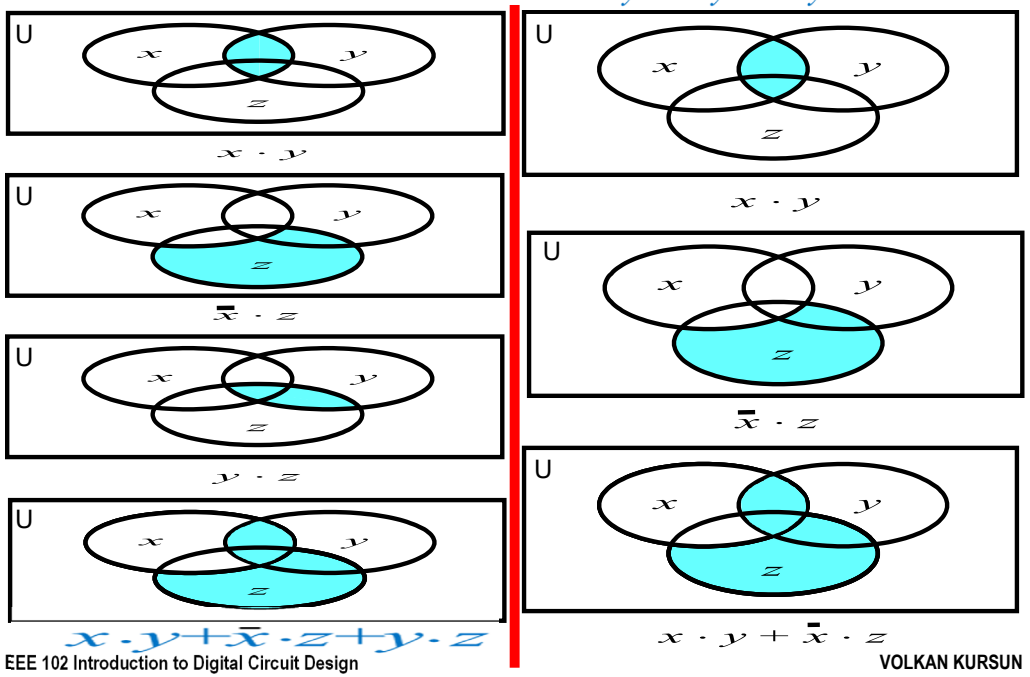
(e) $x + z$



(f) $(x + y) \cdot (x + z)$

Venn Diagrams to Verify Equivalence

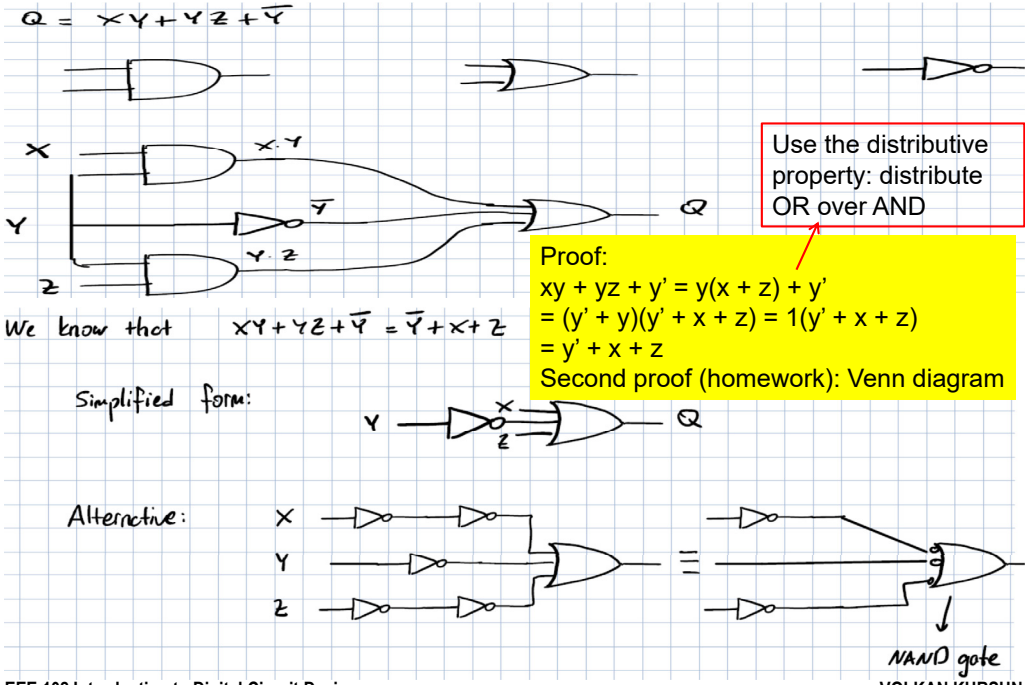
Example-4: verification of the consensus property $x \cdot y + \bar{x} \cdot z + y \cdot z = x \cdot y + \bar{x} \cdot z$



Outline

- Variables, Functions, and Truth Tables
- Logic Gates and Networks
- Boolean Algebra
- The Venn Diagram
- Synthesis Using AND, OR, and NOT Gates
- Sum-of-Products and Product-of-Sums Forms
- NAND and NOR Logic Networks
- XOR and XNOR
- Multiplexer and Tri-State Driver
- Introduction to VHDL

Synthesis Using AND, OR, NOT Gates



Use the distributive property: distribute OR over AND

Proof:
xy + yz + y' = y(x + z) + y'
= (y' + y)(y' + x + z) = 1(y' + x + z)
= y' + x + z
Second proof (homework): Venn diagram

Synthesis From Truth Table

- A straightforward implementation of a logic function can be obtained by using a product term (AND gate) for each row of the truth table where the output function is 1
- **Each product term contains all input variables**: if an input variable x_i is 1 in a row, then x_i is entered in the logical product term. Alternatively, if x_i is 0 in a row, then x_i' is entered in the logical product term
- The logical sum (OR) of these logical product terms realizes the desired logical function
- **There are many different networks that can realize the same function**: some of the networks may be simpler than others and preferred for smaller silicon area and lower cost
- Algebraic manipulation, Venn diagrams, and truth tables can be used to **derive simplified logic expressions and networks, thereby lowering the area and cost**

Synthesis Example-2

- Design a logic circuit with two inputs x_1 and x_2. The function of the circuit is to continuously monitor the state of the two inputs and to produce an output logic value 1 whenever the inputs (x_1, x_2) are (0, 0), (0, 1), or (1, 1). The output should be 0 when the inputs are (1, 0).
- Solution: Let us first express the required behavior with a truth table

x_1	x_2	$f(x_1, x_2)$
0	0	1
0	1	1
1	0	0
1	1	1

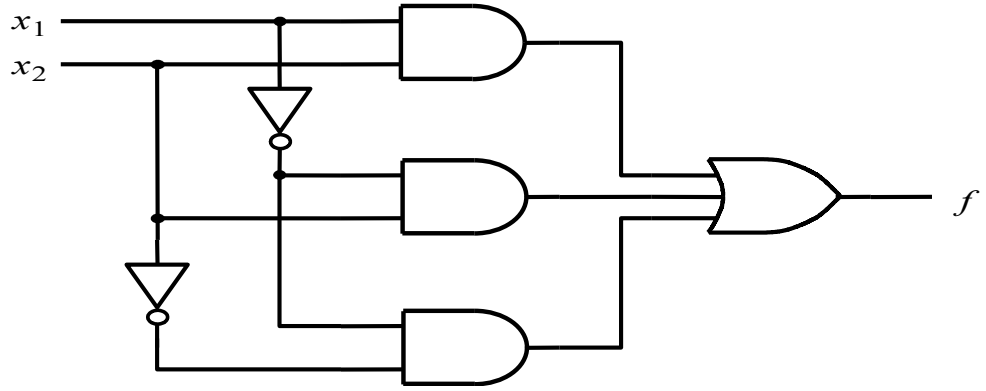
One way to design a logic circuit that implements this truth table is to create a logical product term (AND) for each input combination where the output function f is 1.

Logical product term-1: $x_1' \cdot x_2'$
Logical product term-2: $x_1' \cdot x_2$
Logical product term-3: $x_1 \cdot x_2$

Take the logical sum (OR) of these logical product (AND) terms to realize f: **$f(x_1, x_2) = x_1' \cdot x_2' + x_1' \cdot x_2 + x_1 \cdot x_2$**

Synthesis Example-2: Logic Network

$f(x_1, x_2) = x_1' \cdot x_2' + x_1' \cdot x_2 + x_1 \cdot x_2$



- Although this design implements the required function, the network is not the simplest
- To find a **simpler network**, **manipulate the given obtained expression using the theorems and properties of Boolean algebra**

Synthesis Example-2: Logic Network

$f(x_1, x_2) = x_1' \cdot x_2' + x_1' \cdot x_2 + x_1 \cdot x_2$

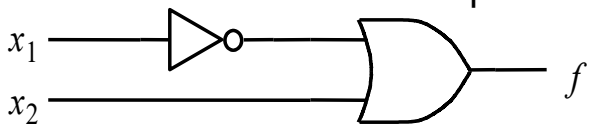
- To find a simpler network, manipulate the given obtained expression using the theorems and properties of Boolean algebra
- Theorem 7: any term can be replicated in a logical sum expression. Let us replicate the second product term:

$f(x_1, x_2) = x_1' \cdot x_2' + x_1' \cdot x_2 + x_1 \cdot x_2 + x_1' \cdot x_2$

- Next, use the commutative and distributive properties:

$f(x_1, x_2) = x_1' \cdot (x_2' + x_2) + x_2 \cdot (x_1 + x_1') = x_1' + x_2$

- The network described with this simplified expression is:



- **The cost of this simplified network is much less than the original design**

Outline

- Variables, Functions, and Truth Tables
- Logic Gates and Networks
- Boolean Algebra
- The Venn Diagram
- Synthesis Using AND, OR, and NOT Gates
- Sum-of-Products and Product-of-Sums Forms
- NAND and NOR Logic Networks
- XOR and XNOR
- Multiplexer and Tri-State Driver
- Introduction to VHDL

Minterms

- If a logic function is specified in the form of a truth table, then an expression that realizes the logic function can be obtained by considering the rows for which the function is 1
- **Minterm:** for a function of n variables, a logical **product (AND) term** in which **each of the n variables appears once** is called a minterm
- The variables may appear in a minterm in either uncomplemented (true) or complemented (false) form: for a given row the minterm is formed by including x_i if $x_i = 1$ and by including x_i' if $x_i = 0$.

Minterms

- Consider the following truth table of a three-variable logic function

Row number	x_1	x_2	x_3	Minterm
0	0	0	0	$m_0 = \overline{x}_1\overline{x}_2\overline{x}_3$
1	0	0	1	$m_1 = \overline{x}_1\overline{x}_2x_3$
2	0	1	0	$m_2 = \overline{x}_1x_2\overline{x}_3$
3	0	1	1	$m_3 = \overline{x}_1x_2x_3$
4	1	0	0	$m_4 = x_1\overline{x}_2\overline{x}_3$
5	1	0	1	$m_5 = x_1\overline{x}_2x_3$
6	1	1	0	$m_6 = x_1x_2\overline{x}_3$
7	1	1	1	$m_7 = x_1x_2x_3$

- Each **minterm** is identified with an **index** that corresponds to the **row number** and the row number is simply the **decimal equivalent** of the corresponding binary number in that row

Sum-of-Products (SOP) Representation

- Sum-of-products (SOP) form: A logic expression consisting of logical product (AND) terms that are logically summed (ORed)
- **Sum-of-products circuit implementations are two-level: a first level of AND gates followed by a second level of OR gate**
- **ANY LOGIC FUNCTION** can be represented by a **sum of minterms** that correspond to the rows **where the function is 1**
- **Canonical sum-of-products: each product term is a minterm** in a sum-of-products representation of a function
- The resulting **canonical implementation is** functionally correct and **unique** but not necessarily the simplest (lowest-cost) implementation of the logic function: the first step in synthesis process is to derive a canonical sum-of-products expression for a given function and then to manipulate this canonical expression using the theorems and properties to find a simpler but functionally equivalent sum-of-products expression that has a lower cost

Canonical Sum-of-Products Example

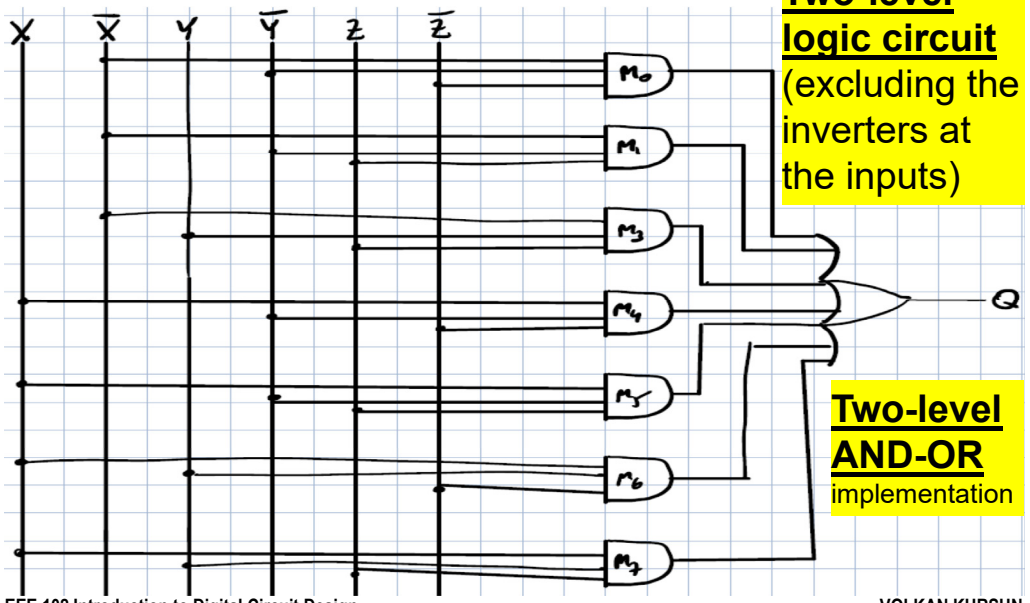
$Q = XY + YZ + \overline{Y} = X + \overline{Y} + Z$

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

$Q = m_0 + m_1 + m_2 + m_4 + m_5 + m_6 + m_7 = \sum m(0,1,2,4,5,6,7)$

Canonical Sum-of-Products Example

- Logic circuit implementation of the canonical sum-of-products form of the logic function



Two-level logic circuit (excluding the inverters at the inputs)

Two-level AND-OR implementation

Canonical Sum-of-Products Example-2

- Consider the three-variable function $f(x_1, x_2, x_3)$ specified by the following truth table:

Row number	x_1	x_2	x_3	$f(x_1, x_2, x_3)$
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

To synthesize this logic function in canonical form, include the minterms m_1, m_4, m_5, and m_6.

- Canonical sum-of-products expression for f:

$f(x_1, x_2, x_3) = x_1'x_2'x_3 + x_1x_2'x_3' + x_1x_2'x_3 + x_1x_2x_3'$

$f(x_1, x_2, x_3) = \sum(m1, m4, m5, m6) = \sum m(1,4,5,6)$

Simplified Sum-of-Products (SOP)

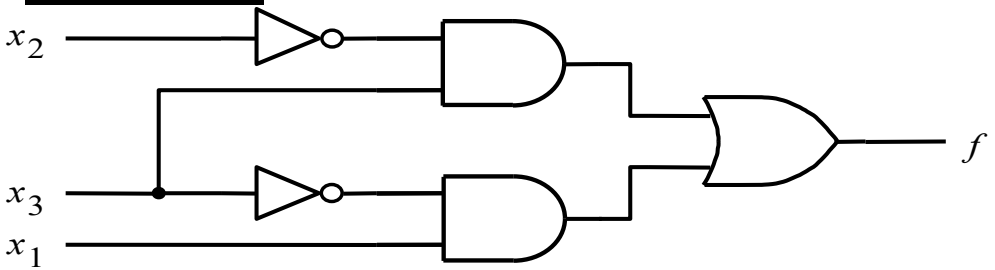
$f(x_1, x_2, x_3) = x_1'x_2'x_3 + x_1x_2'x_3' + x_1x_2'x_3 + x_1x_2x_3'$

- This canonical expression can be simplified with algebraic manipulation:

$f(x_1, x_2, x_3) = x_1'x_2'x_3 + x_1x_2'x_3' + x_1x_2'x_3 + x_1x_2x_3'$

$= (x_1' + x_1)x_2'x_3 + (x_2' + x_2)x_1x_3' = x_2'x_3 + x_1x_3'$

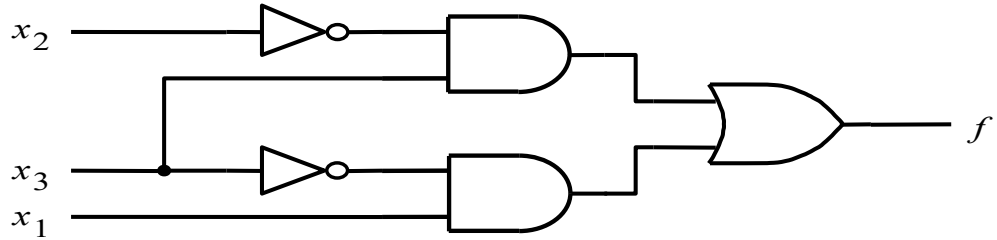
- This is the **minimum cost sum-of-products expression** for f



Comparison of Two Sums-of-Products

1) $f(x_1, x_2, x_3) = x_1'x_2'x_3 + x_1x_2'x_3' + x_1x_2'x_3 + x_1x_2x_3'$

2) $f(x_1, x_2, x_3) = x_2'x_3 + x_1x_3'$



- **Cost**: the **total number of inputs to all the gates** in the synthesized logic circuit

Cost of the canonical sum-of-products (3 inverters + 4*3-input AND gates + 1*4-input OR gate)= 3 + 3*4 + 4 = 19

Cost of the simplified sum-of-products (2 inverters + 2*2-input AND gates + 1*2-input OR gate = 2 + 2*2 + 2 = 8

- The simplified implementation lowers the cost by **58%**

Maxterms

- The complements of minterms are called maxterms
- Each maxterm is identified with an index that corresponds to the row number: M_i

Row number	x_1	x_2	x_3	Minterm	Maxterm
0	0	0	0	$m_0 = \overline{x}_1\overline{x}_2\overline{x}_3$	$M_0 = x_1 + x_2 + x_3$
1	0	0	1	$m_1 = \overline{x}_1\overline{x}_2x_3$	$M_1 = x_1 + x_2 + \overline{x}_3$
2	0	1	0	$m_2 = \overline{x}_1x_2\overline{x}_3$	$M_2 = x_1 + \overline{x}_2 + x_3$
3	0	1	1	$m_3 = \overline{x}_1x_2x_3$	$M_3 = x_1 + \overline{x}_2 + \overline{x}_3$
4	1	0	0	$m_4 = x_1\overline{x}_2\overline{x}_3$	$M_4 = \overline{x}_1 + x_2 + x_3$
5	1	0	1	$m_5 = x_1\overline{x}_2x_3$	$M_5 = \overline{x}_1 + x_2 + \overline{x}_3$
6	1	1	0	$m_6 = x_1x_2\overline{x}_3$	$M_6 = \overline{x}_1 + \overline{x}_2 + x_3$
7	1	1	1	$m_7 = x_1x_2x_3$	$M_7 = \overline{x}_1 + \overline{x}_2 + \overline{x}_3$

Product-of-Sums (POS) Representation

- Product-of-sums (POS) form: a logic expression consisting of logical product (AND) of logical sum (OR) terms
- **ANY LOGIC FUNCTION** can be represented by a **product of maxterms** that correspond to the rows **where the logic function is 0**
- **Canonical product-of-sums: each sum term is a maxterm** in a product-of-sums representation of a function
- The resulting canonical implementation is functionally correct and **unique** but not necessarily the simplest (lowest-cost) implementation of the logic function: the first step in synthesis process is to derive a canonical product-of-sums expression for a given function and then to manipulate this canonical expression using the theorems and properties to find a simpler but functionally equivalent product-of-sums expression that has a lower cost

Canonical Product-of-Sums (POS) Form

- Just like a function f can be represented as the sum of minterms for which f = 1, the complement of the function can be represented as the sum of minterms for which f' = 1 (f = 0)
- Example: Consider the following truth table

Row number	x_1	x_2	x_3	$f(x_1, x_2, x_3)$
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

$f'(x_1, x_2, x_3) = m_0 + m_2 + m_3 + m_7 = x_1'x_2'x_3' + x_1'x_2x_3' + x_1'x_2x_3 + x_1x_2x_3$

Using Theorems 9 and DeMorgan:

$(f'(x_1, x_2, x_3))' = f(x_1, x_2, x_3)$

$= (m_0 + m_2 + m_3 + m_7)'$

$= m_0' \cdot m_2' \cdot m_3' \cdot m_7'$

$= M_0 \cdot M_2 \cdot M_3 \cdot M_7 =$

$\prod(M_0, M_2, M_3, M_7) = \prod M(0, 2, 3, 7)$

$= (x_1+x_2+x_3)(x_1+x_2'+x_3)(x_1+x_2'+x_3')(x_1'+x_2'+x_3')$

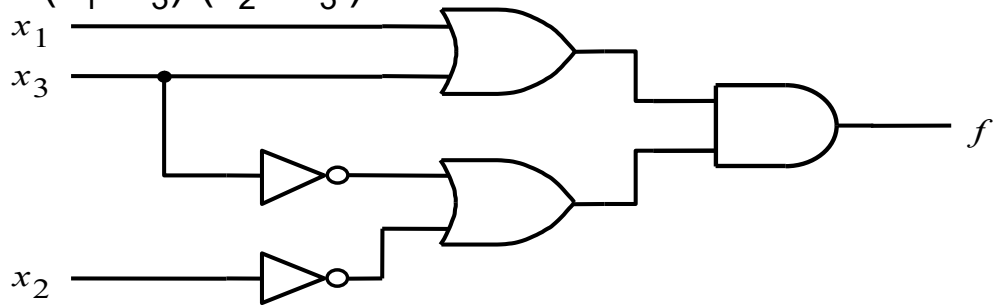
Simplified Product-of-Sums (POS) Form

Canonical $f(x_1, x_2, x_3) = (x_1+x_2+x_3).(x_1+x_2'+x_3).(x_1+x_2'+x_3').(x_1'+x_2'+x_3')$

- Use the commutative, associative, and combining properties to **simplify this canonical expression with algebraic manipulation**:

$f(x_1, x_2, x_3) = ((x_1+x_3)+(x_2x_2')).((x_2'+x_3')+(x_1x_1'))$

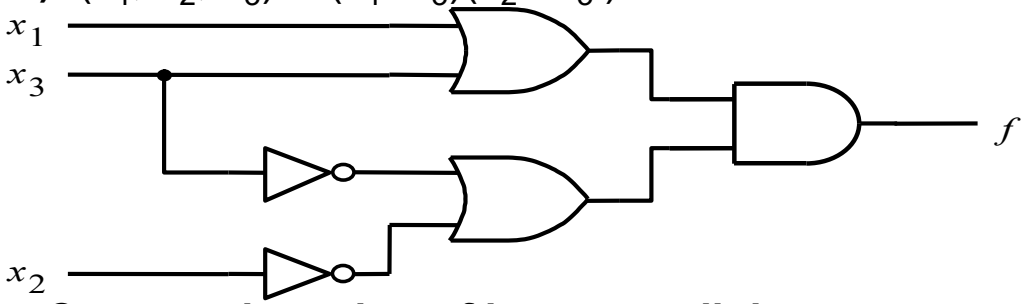
$= (x_1+x_3).(x_2'+x_3')$



Comparison of Two Products-of-Sums

1) $f(x_1, x_2, x_3) = (x_1+x_2+x_3)(x_1+x_2'+x_3)(x_1+x_2'+x_3')(x_1'+x_2'+x_3')$

2) $f(x_1, x_2, x_3) = (x_1+x_3)(x_2'+x_3')$



- **Cost**: **total number of inputs to all the gates**

Cost of the canonical product-of-sums (3 inverters + 4*3-input OR gates + 1*4-input AND gate)= 3 + 4*3 + 4 = 19

Cost of the simplified sum-of-products (2 inverters + 2*2-input OR gates + 1*2-input AND gate = 2 + 2*2 + 2 = 8

- The simplified implementation lowers the cost by **58%**

Canonical Product-of-Sums Example-2

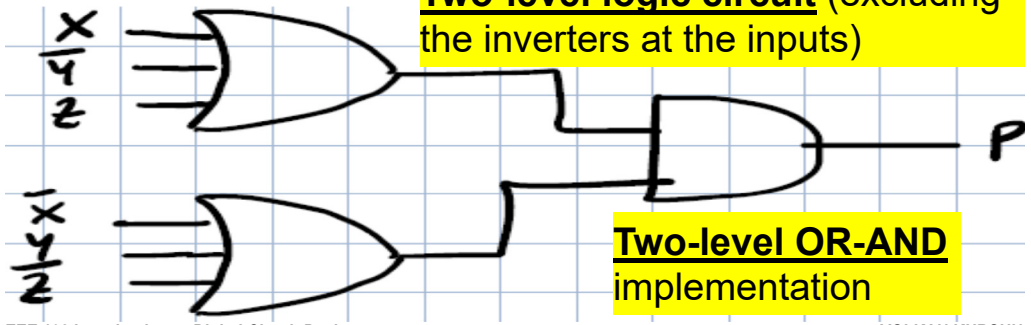
X	Y	Z	P	$\bar{P}$
0	0	0	1	0
0	0	1	1	0
0	1	0	0	1
0	1	1	1	0
1	0	0	1	0
1	0	1	0	1
1	1	0	1	0
1	1	1	1	0

$\bar{P} = M_2 + M_5$

$P = \overline{\bar{P}} = \overline{(M_2 + M_5)} = \bar{M}_2 \cdot \bar{M}_5 = M_2 \cdot M_5$

P = (X + Y' + Z) . (X' + Y + Z')

Two-level logic circuit (excluding the inverters at the inputs)



Two-level OR-AND implementation

Outline

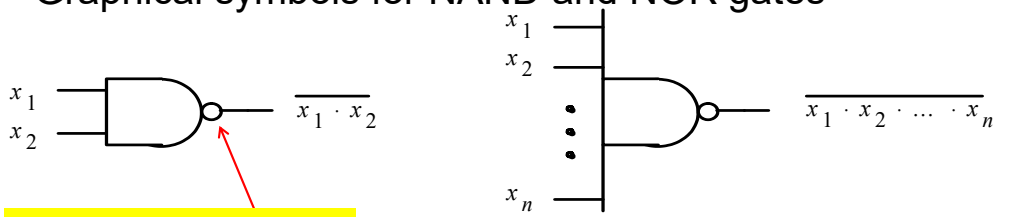
- Variables, Functions, and Truth Tables
- Logic Gates and Networks
- Boolean Algebra
- The Venn Diagram
- Synthesis Using AND, OR, and NOT Gates
- Sum-of-Products and Product-of-Sums Forms
- NAND and NOR Logic Networks
- XOR and XNOR
- Multiplexer and Tri-State Driver
- Introduction to VHDL

NAND and NOR Gates

- NAND function is obtained by complementing the AND operation
- NOR function is obtained by complementing the OR operation
- In actual CMOS circuit implementations, a single stage AND or OR gate does not exist
- NAND and NOR gates have single stage CMOS circuit implementations
- In actual CMOS integrated circuits, AND function is obtained by inverting a NAND gate's output
- In actual CMOS integrated circuits, OR function is obtained by inverting a NOR gate's output
- Logic networks with only NAND, NOR, and inverters are preferred for implementation on silicon

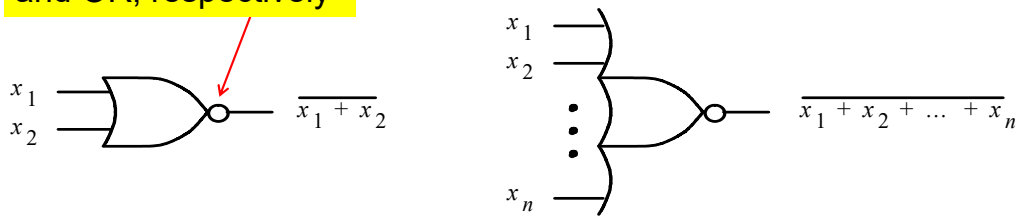
NAND and NOR Gates

- Graphical symbols for NAND and NOR gates



Bubbles at the output distinguish NAND and NOR from AND and OR, respectively

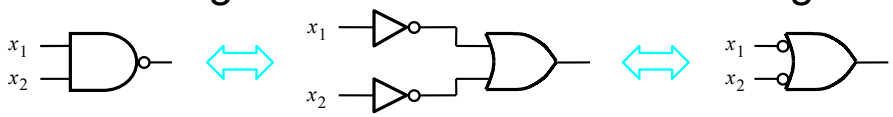
(a) NAND gates



(b) NOR gates

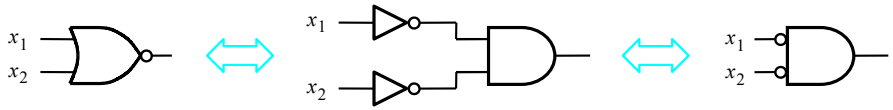
DeMorgan's Theorem in Logic Gates

- A NAND of two variables x_1 and x_2 is equivalent to first inverting the variables and then ORing them



(a) $\overline{x_1 x_2} = \bar{x}_1 + \bar{x}_2$

- A NOR of two variables x_1 and x_2 is equivalent to first inverting the variables and then ANDing them

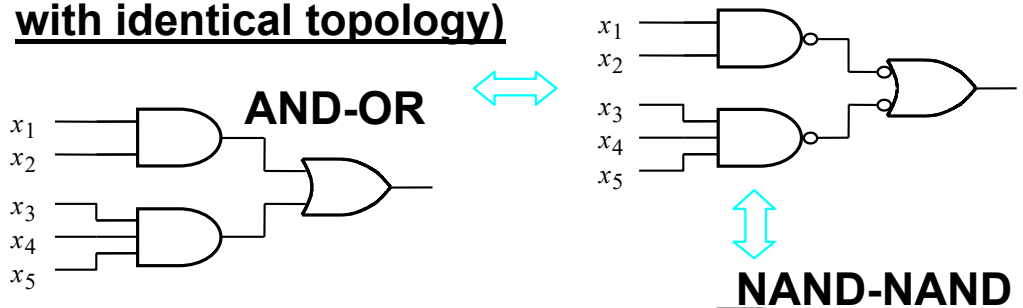


(b) $\overline{x_1 + x_2} = \bar{x}_1 \bar{x}_2$

NOT gates are shown as bubbles in the figures indicating inversion of signals

Implementation Using Only NAND

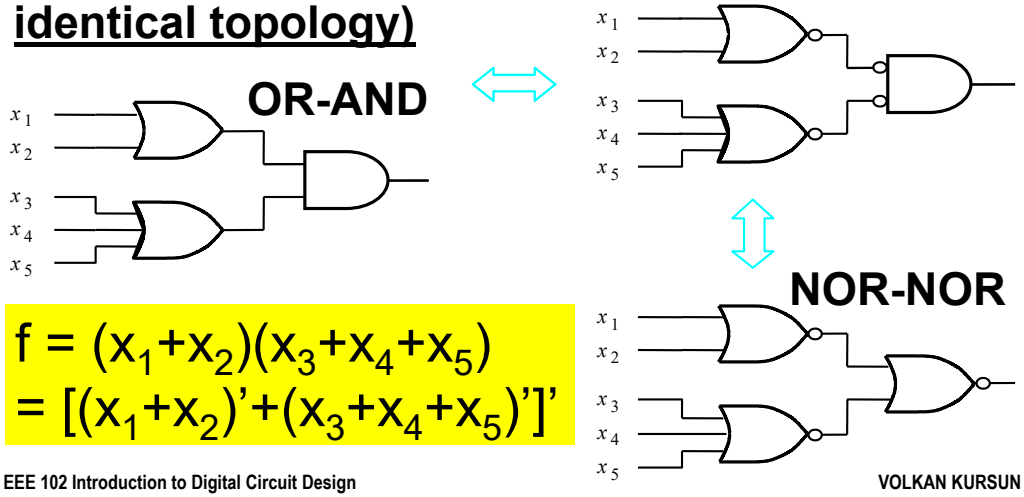
- **Any logic function** can be implemented in sum-of-products (two-stage AND-OR) form: these networks can also be **implemented with only NAND gates (two-stage NAND-NAND network with identical topology)**



$f = x_1x_2 + x_3x_4x_5$
$= [(x_1x_2)'(x_3x_4x_5)']'$

Implementation Using Only NOR

- **Any logic function** can be implemented in product-of-sums (two stage OR-AND) form: these networks can also be **implemented with only NOR gates (two-stage NOR-NOR network with identical topology)**



$f = (x_1+x_2)(x_3+x_4+x_5)$
$= [(x_1+x_2)'+(x_3+x_4+x_5)']'$

Implementation Using Only NOR

- Example: Consider the function

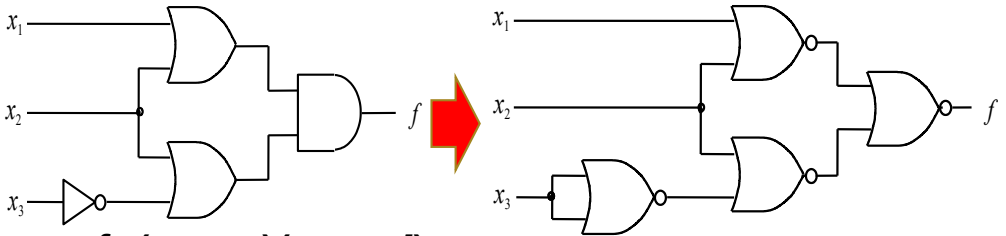
$$f(x_1, x_2, x_3) = \sum m(2,3,4,6,7) = \prod M(0,1,5)$$

- The **canonical product-of-sums** is:

$f(x_1, x_2, x_3) = M_0.M_1.M_5 = (x_1+x_2+x_3)(x_1+x_2+x_3')(x_1'+x_2+x_3')$

- **Simplify**: $f = (x_1+x_2+x_3)(x_1+x_2+x_3')(x_1+x_2+x_3')(x_1'+x_2+x_3')$

$=((x_1+x_2)+(x_3x_3'))((x_2+x_3')+(x_1x_1'))=\mathbf{(x_1+x_2)(x_2+x_3')}$



$f=(x_1+x_2)(x_2+x_3')$ (a) POS implementation

$f=[(x_1+x_2)'+(x_2+x_3')']'$ (b) NOR implementation

Implementation Using Only NAND

- Example: Consider the function

$$f(x_1, x_2, x_3) = \sum m(2,3,4,6,7) = \prod M(0,1,5)$$

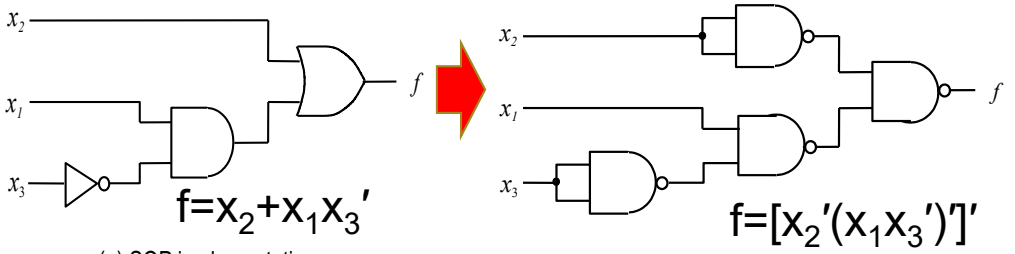
- The **canonical sum-of-products** is:

$f(x_1, x_2, x_3) = m_2+m_3+m_4+m_6+m_7$

$=x_1'x_2x_3'+x_1'x_2x_3+x_1x_2'x_3'+x_1x_2x_3'+x_1x_2x_3$

- **Simplify**: $f = x_1'x_2(x_3'+x_3)+x_1x_3'(x_2+x_2')+x_1x_2(x_3'+x_3)$

$f=x_1'x_2+x_1x_3'+x_1x_2=(x_1'+x_1)x_2+x_1x_3'=x_2+x_1x_3'$



$f=x_2+x_1x_3'$ (a) SOP implementation

$f=[x_2'(x_1x_3')']'$ (b) NAND implementation

Outline

- Variables, Functions, and Truth Tables
- Logic Gates and Networks
- Boolean Algebra
- The Venn Diagram
- Synthesis Using AND, OR, and NOT Gates
- Sum-of-Products and Product-of-Sums Forms
- NAND and NOR Logic Networks
- XOR and XNOR
- Multiplexer and Tri-State Driver
- Introduction to VHDL

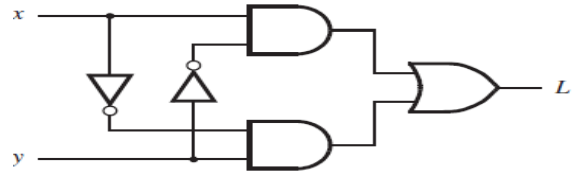
2-Input XOR

SOP of XOR = $m_1 + m_2$ = x'y + xy'

POS of XOR = $M_0 \cdot M_3$ = (x + y) . (x' + y')

x	y	L
0	0	0
0	1	1
1	0	1
1	1	0

(b) Truth table

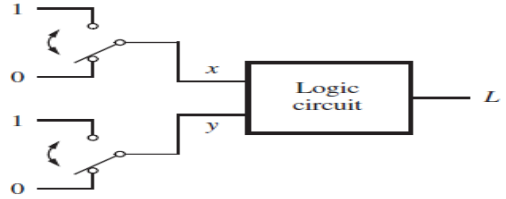


(c) Logic network

Checks if inputs are different

$L(x, y) = x \oplus y$

(d) XOR gate symbol

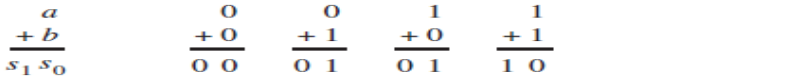


a) Two switches that control a light

- Example: two toggle switches that control the values of x and y. x and y are inputs to an XOR gate that controls a light L

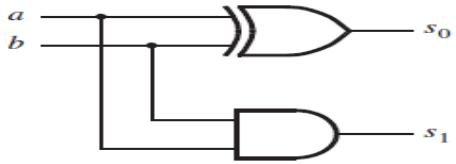
Binary Adder (Half Adder)

- Consider addition of two one-digit binary numbers a and b. The four possible valuations of a, b, and the resulting sums are given in the following truth table. $S = s_1 s_0$ is a two-digit number since 1 + 1 = 0b10



(a) Evaluation of $S = a + b$ ← Arithmetic addition operator

a	b	s_1	s_0
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



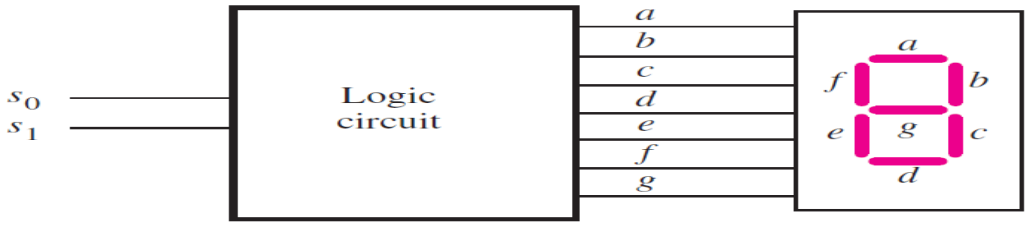
$s_1(a, b) = m_3 = a.b$

$s_0(a, b) = m_1 + m_2 = a'.b + a.b' = a \oplus b$

Binary Adder with 7-Segment Display

- Consider a half adder connected to a 7-segment display. The display shows the value of sum as a decimal number (0, 1, or 2). Design the logic circuit to drive the display.
- The display has 7 segments, labelled a, b, c, d, e, f, g, where each segment is a light-emitting diode (LED)
- The output of the adder has one of three possible values: 0b00, 0b01, or 0b11
- The logic circuit has two inputs, s1, s0, and seven outputs. Each output drives one segment of the display.

Binary Adder with 7-Segment Display



(a) Logic circuit and 7-segment display

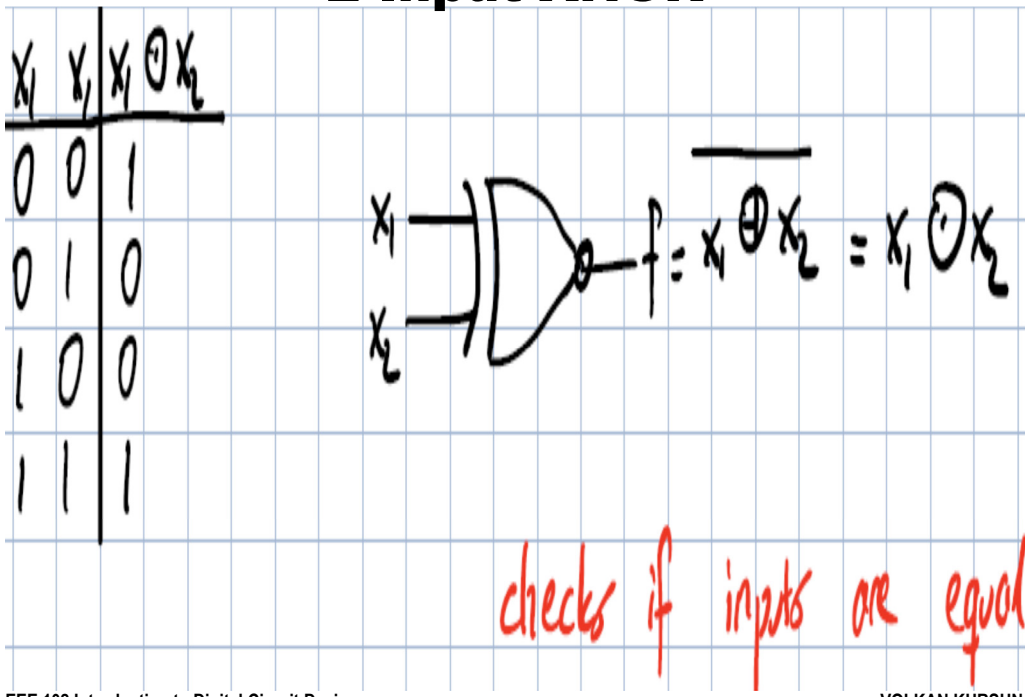
	s_1	s_0	a	b	c	d	e	f	g
0	0	0	1	1	1	1	1	1	0
1	0	1	0	1	1	0	0	0	0
2	1	0	1	1	0	1	1	0	1

(b) Truth table

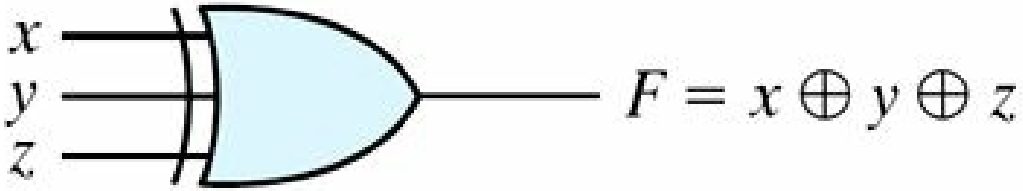
a = d = e = $s_1'.s_0' + s_1.s_0' = s_0'$, **b** = 1,

c = $s_1'.s_0' + s_1'.s_0 = s_1'$, $f = s_1'.s_0'$, **g** = $s_1.s_0'$

2-Input XNOR



3-Input XOR

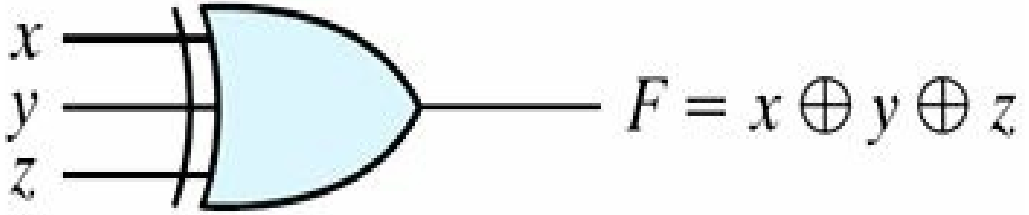


x	y	z	x ⊕ y ⊕ z
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

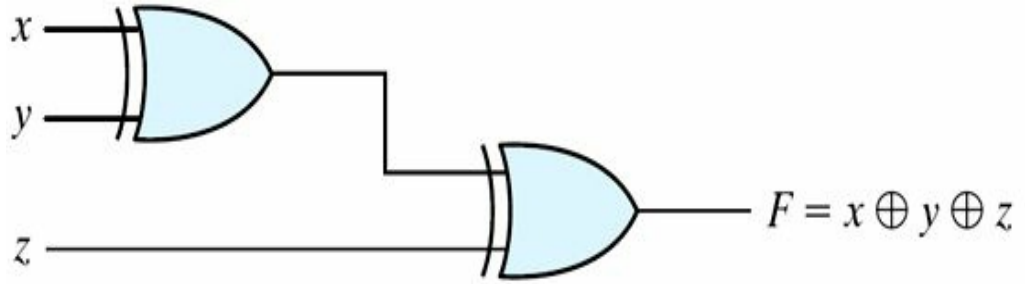
- All 0s: output is 0
- Even number of 1s: output is 0
- Odd number of 1s: output is 1

3-Input XOR

- 3-input XOR gate:



- 3-input XOR function can be implemented with two-stage 2-input XOR gates:



3-Input XNOR

- 3-input XNOR gate:

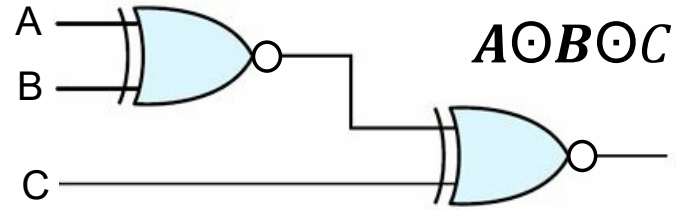
$$g \stackrel{\text{def}}{=} \overline{A \oplus B \oplus C}$$

A	B	C	A ⊕ B ⊕ C	$\overline{A \oplus B \oplus C}$
0	0	0	0	1
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	0

- All 0s: output is 1
- Even number of 1s: output is 1
- Odd number of 1s: output is 0

3-Input XNOR =? 2*2-Input XNOR

- Consider the following logic circuit:



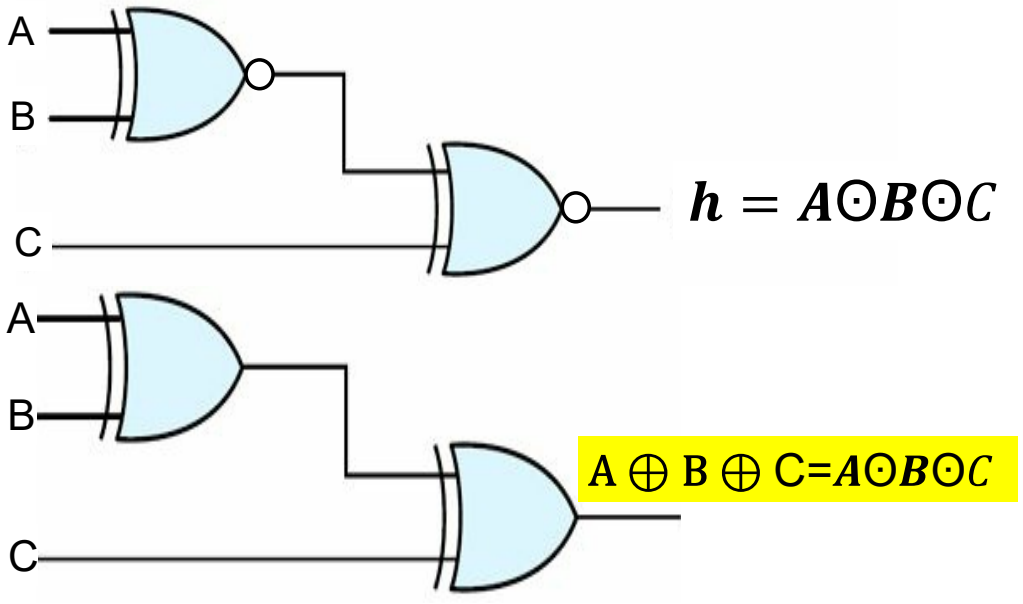
$$A \odot B \odot C \neq \overline{A \oplus B \oplus C}$$

A	B	C	A ⊕ B ⊕ C	$\overline{A \oplus B \oplus C}$	$A \odot B \odot C$
0	0	0	**0**	1	**0**
0	0	1	**1**	0	**1**
0	1	0	**1**	0	**1**
0	1	1	**0**	1	**0**
1	0	0	**1**	0	**1**
1	0	1	**0**	1	**0**
1	1	0	**0**	1	**0**
1	1	1	**1**	0	**1**

$$A \odot B \odot C = A \oplus B \oplus C$$

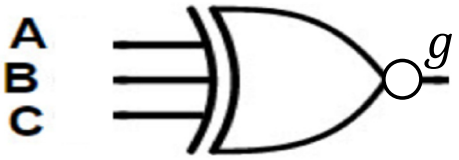
Equivalent XOR and XNOR Networks

- The following two logic circuits are equivalent:



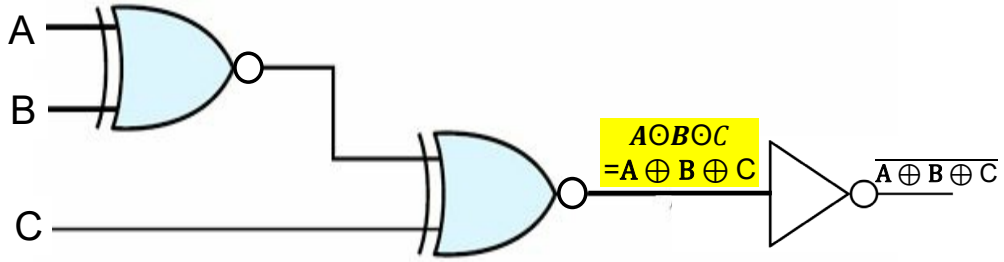
3-Input XNOR Implementation

- 3-input XNOR gate:



$$g \overset{\text{def}}{=} \overline{A \oplus B \oplus C}$$

- 3-input XNOR function can be implemented with cascaded 2-input XNOR gates and an inverter as follows:



2, 3, and 4-Input XOR and XNOR Functions

- **2-input XNOR**: $a \odot b \overset{\text{def}}{=} \overline{a \oplus b}$
- **3-input XOR**: $a \odot b \odot c = \overline{(a \oplus b)} \odot c = \overline{(a \oplus b)}c + \overline{(a \oplus b)}\bar{c} = a \oplus b \oplus c$
- **4-input XNOR**: $a \odot b \odot c \odot d = (a \oplus b \oplus c) \odot d = \overline{(a \oplus b \oplus c) \oplus d} = \overline{a \oplus b \oplus c \oplus d}$
- 4-input XNOR gate output:

$$XNOR4 \overset{\text{def}}{=} \overline{a \oplus b \oplus c \oplus d}$$

- By cascading 3*2-input XNOR gates, the 4-input XNOR function can be implemented:

$$a \odot b \odot c \odot d = \overline{a \oplus b \oplus c \oplus d}$$

5-Input XNOR Gate

$$a \odot b \odot c \odot d = \overline{a \oplus b \oplus c \oplus d}$$

- **5-input XOR**: $a \odot b \odot c \odot d \odot e = \overline{(a \oplus b \oplus c \oplus d)} \odot e = \overline{(a \oplus b \oplus c \oplus d)}e + \overline{\overline{(a \oplus b \oplus c \oplus d)}}\bar{e} = a \oplus b \oplus c \oplus d \oplus e$
- 5-input XNOR gate output $\overset{\text{def}}{=} \overline{a \oplus b \oplus c \oplus d \oplus e}$
- 5-input XNOR gate can be implemented by:

A) Cascading 4*2-input XOR gates and **inverting the output**

B) Cascading 4*2-input XNOR gates and **inverting the output**

n-Input XOR and XNOR Functions

- General rule, if (n-1)*2-input XNOR gates are cascaded:

1) If n is odd,

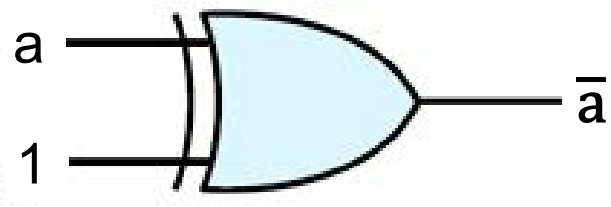
$$a_1 \odot a_2 \odot \ldots \odot a_{n-1} \odot a_n = \overline{a_1 \oplus a_2 \oplus \cdots \oplus a_{n-1} \oplus a_n} \neq a_1 \oplus a_2 \oplus \cdots \oplus a_{n-1} \oplus a_n$$

By inverting the final output of cascaded (n-1)*2-input XNOR gates, an n-input XNOR gate can be implemented

2) If n is even, $a_1 \odot a_2 \odot \ldots \odot a_{n-1} \odot a_n = \overline{a_1 \oplus a_2 \oplus \cdots \oplus a_{n-1} \oplus a_n}$

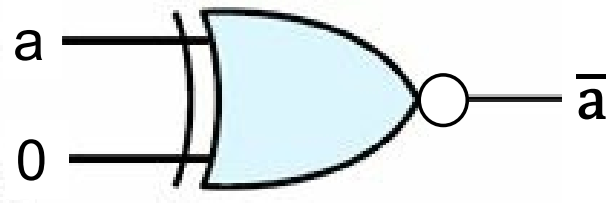
Inversion with XOR and XNOR

- $a \oplus 1 = \bar{a}.1 + a.\bar{1} = \bar{a}$



XOR gate acting as an inverter

- $a \odot 0 = \bar{a}.\bar{0} + a.0 = \bar{a}$



XNOR gate acting as an inverter

THREE-WAY LIGHT CONTROL

- Assume a room with three switches that control a light L. The light is off when all switches are open. Closing any one of the switches will turn the light on. Then turning on any other switch will turn the light off. Thus, the light will be on if exactly one switch is closed and will be off if two or no switches are closed. If the light is off with two switches that are closed, then the light will be turned on by closing the third switch.

x_1	x_2	x_3	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

$f(x_1, x_2, x_3)=m_1+m_2+m_4+m_7$
$=x_1'x_2'x_3+x_1'x_2x_3'+x_1x_2'x_3'+x_1x_2x_3$
$= x_1 \oplus x_2 \oplus x_3$

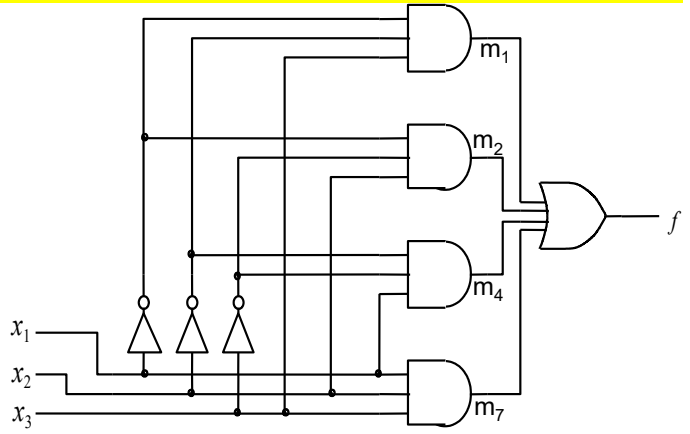
$f(x_1, x_2, x_3)=M_0M_3M_5M_6$
$=(x_1+x_2+x_3)(x_1+x_2'+x_3')(x_1'+x_2+x_3')$
$(x_1'+x_2'+x_3)$

THREE-WAY LIGHT CONTROL

- **Sum-of-products realization** (cannot be further simplified into a lower cost sum-of-products expression)

$f(x_1, x_2, x_3)=m_1+m_2+m_4+m_7$
$=x_1'x_2'x_3+x_1'x_2x_3'+x_1x_2'x_3'+x_1x_2x_3 = x_1 \oplus x_2 \oplus x_3$

x_1	x_2	x_3	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



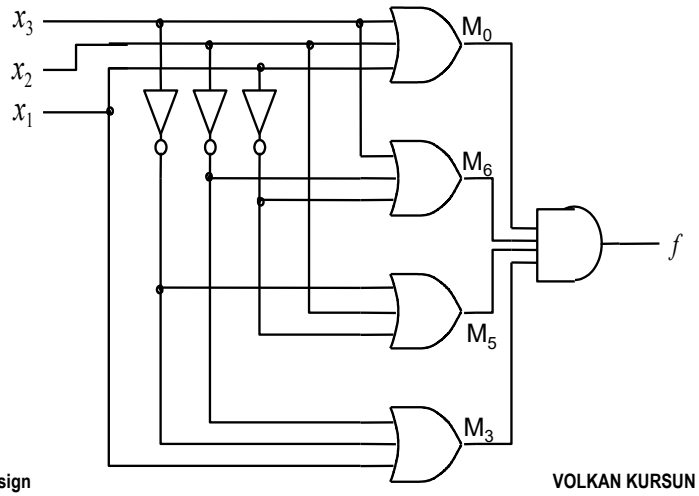
(a) Sum-of-products realization

THREE-WAY LIGHT CONTROL

- **Product-of-sums realization** (cannot be further simplified into a lower cost product-of-sums expression)

$$f(x_1, x_2, x_3)=M_0M_3M_5M_6$$
$$=(x_1+x_2+x_3)(x_1+x_2'+x_3')(x_1'+x_2+x_3')(x_1'+x_2'+x_3)$$

x_1	x_2	x_3	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



Outline

- Variables, Functions, and Truth Tables
- Logic Gates and Networks
- Boolean Algebra
- The Venn Diagram
- Synthesis Using AND, OR, and NOT Gates
- Sum-of-Products and Product-of-Sums Forms
- NAND and NOR Logic Networks
- XOR and XNOR
- Multiplexer and Tri-State Driver
- Introduction to VHDL

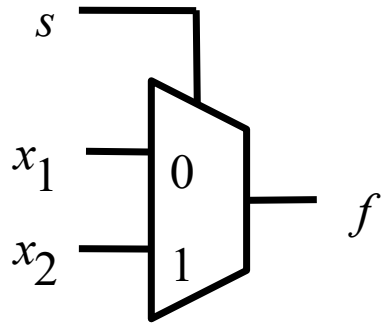
Multiplexer

- In digital systems, it is often necessary to choose data from a number of possible sources: multiplexers are used to steer data from one of many inputs to a single output
- A multiplexer circuit multiplexes input signals onto a single output
- A **two-to-one multiplexer example**: two data inputs x_1 and x_2. Design a circuit that produces an output that has the same value as either x_1 or x_2, depending on the value of a select signal s. The select signal is the third input of the multiplexer. Assume that the output of the multiplexer will be the same value as x_1 if s = 0 and the same value as x_2 if s = 1.
- Based on the design requirements, specify the desired circuit in the form of a truth table

2-to-1 Multiplexer Example

s	x_1	x_2	$f(s, x_1, x_2)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Truth table



Graphical symbol

- Write the function in **canonical sum-of-products** form:

$$f(s, x_1, x_2)=m_2+m_3+m_5+m_7=\sum m(2,3,5,7)$$
$$=s'x_1x_2'+s'x_1x_2+sx_1'x_2+sx_1x_2$$

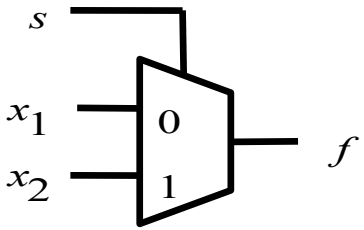
Simplify the expression using the distributive property:

$$f(s, x_1, x_2) =s'x_1(x_2'+x_2)+sx_2(x_1'+x_1)= s'x_1+sx_2$$

Simplified 2-to-1 Multiplexer Circuit

s	x_1	x_2	$f(s, x_1, x_2)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

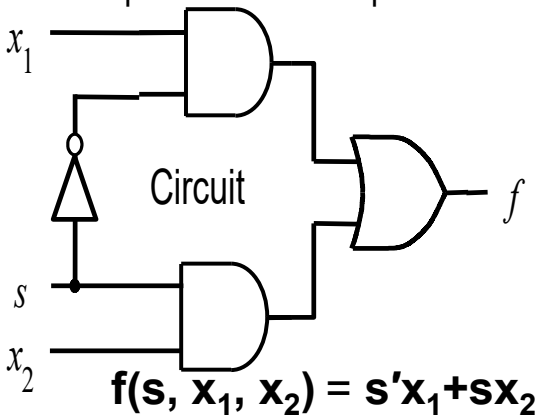
Truth table



Graphical symbol

s	$f(s, x_1, x_2)$
0	x_1
1	x_2

More compact truth-table representation

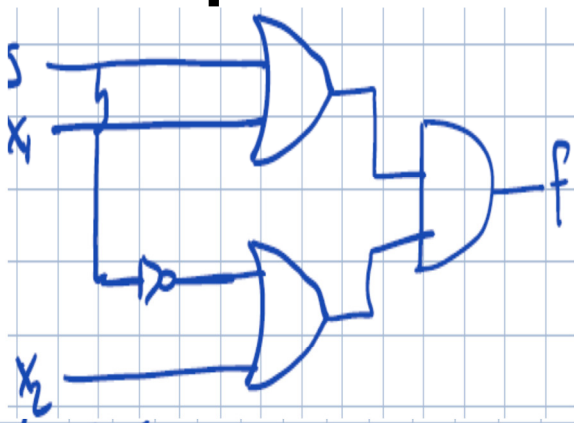


$f(s, x_1, x_2) = s'x_1 + sx_2$

Simplified 2-to-1 Multiplexer: POS

s	x_1	x_2	$f(s, x_1, x_2)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Truth table



$f = \Pi M(0,1,4,6)$ canonical POS

$= (s + x_1 + x_2).(s + x_1 + \bar{x}_2).(\bar{s} + x_1 + x_2).(\bar{s} + \bar{x}_1 + x_2)$

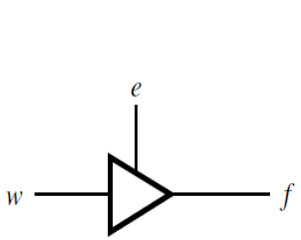
$= (s + x_1).(\bar{s} + x_2)$ POS

More Complex Multiplexers

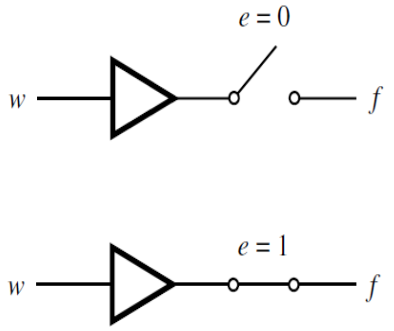
- 4-to-1 multiplexer: four data inputs, two select inputs to choose one of four data inputs, and one data output
- 8-to-1 multiplexer: eight data inputs, three select inputs to choose one of eight data inputs, and one data output
- 2^n-to-1 multiplexer: 2^n data inputs, n select inputs to choose one of 2^n data inputs, and one data output

Tri-State Driver

- Outputs of two gates cannot be connected together
- Specialized interface circuits are needed if the outputs of gates need to be connected to a **common bus**
- Use tri-state drivers to interface the gates to the common wires of a bus



(a) Symbol



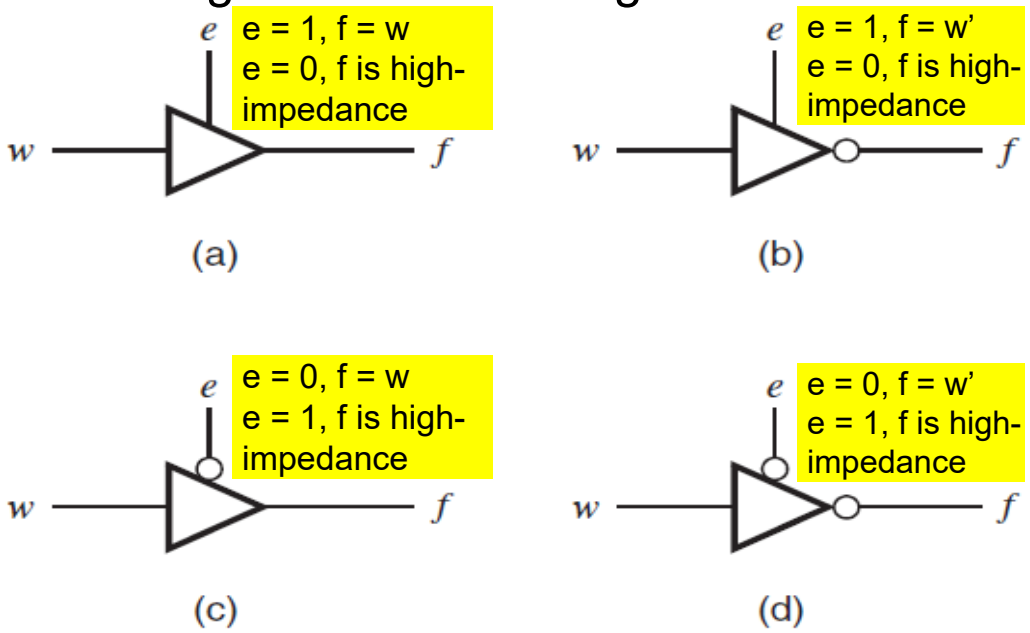
(b) Equivalent circuit

e	w	f
0	0	Z
0	1	Z
1	0	0
1	1	1

(c) Truth table

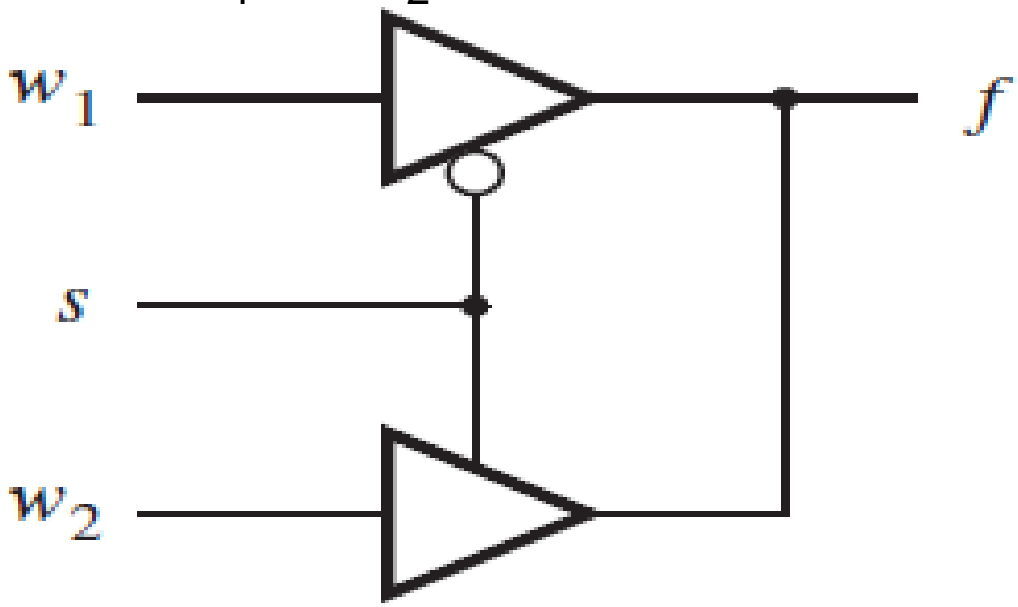
Tri-State Driver Types

- Inverting and non-inverting tri-state drivers:



2-to-1 Mux with Tri-State Drivers

- $f = s'w_1 + sw_2$



Outline

- Variables, Functions, and Truth Tables
- Logic Gates and Networks
- Boolean Algebra
- The Venn Diagram
- Synthesis Using AND, OR, and NOT Gates
- Sum-of-Products and Product-of-Sums Forms
- NAND and NOR Logic Networks
- XOR and XNOR
- Multiplexer and Tri-State Driver
- Introduction to VHDL

Computer-Aided Design (CAD)

- Complex logic circuits cannot be designed manually: CAD tools are needed
- CAD tools are used for various purposes at different phases of the design process

Phase-1) Design Entry: description of the circuit being designed. There are **two methods for design entry**:

1A) Schematic Capture: A logic circuit can be defined by drawing logic gates and interconnecting them with wires. A schematic capture tool allows user to draw a schematic diagram. To facilitate schematic capture, the tool provides a library of gates of various types and inputs. Any subcircuits that have been previously created can be represented as graphical symbols and included in the schematic of a larger logic circuit: **hierarchical design** (allows dealing with the complexities of digital systems)

Computer-Aided Design (CAD)

Phase-1) Design Entry: description of the circuit being designed. There are **two methods for design entry**:

1B) Hardware Description Languages: Schematic capture tools are quite useful. However, for very large circuits, schematic entry (just like manual design) is also not feasible. A better method to design large circuits is to represent the circuit in textual form using a hardware description language (HDL). One of the two HDLs that are IEEE standards is Very High Speed Integrated Circuit Hardware Description Language (VHDL).

Advantages of VHDL:

1B1) Design portability to new technologies and different CAD tools: a circuit specified in VHDL can be implemented in different types of semiconductor technologies and with CAD tools provided by different companies. Portability is very important since the digital circuit technology changes very rapidly (technology scaling). By **using a standard language**, the **designer can focus on the functionality of the desired circuit** without being concerned about the details of the technology that will eventually be used to implement the design.

Computer-Aided Design (CAD)

Advantages of VHDL:

1B1) Design portability to new technologies and different CAD tools:

1B2) Reusability for faster development of new products: design entry is done by writing a VHDL code that is plain text. Documentation that explains how the circuit works can be included in the code. Sharing and reuse of existing VHDL code is easy. Faster development of new products is allowed by adapting existing VHDL code.

1B3) Modularity and hierarchical design: VHDL code can be written in a modular way that facilitates hierarchical design.

Phase-2) Logic Synthesis: process of generating a logic circuit from an initial specification in the form of a schematic diagram or HDL code. Compilation of VHDL code: translation of VHDL code into a network of logic gates. The output of the synthesis tool is a set of logic expressions that describe the logic functions required by the design entry.

Logic circuit optimization: initial logic expressions produced by the synthesis tools may not be in optimum form. The synthesis tools may manipulate these initial logic expressions to automatically generate better circuits. Optimization goals: smaller area, lower cost, higher speed, lower power consumption.

Computer-Aided Design (CAD)

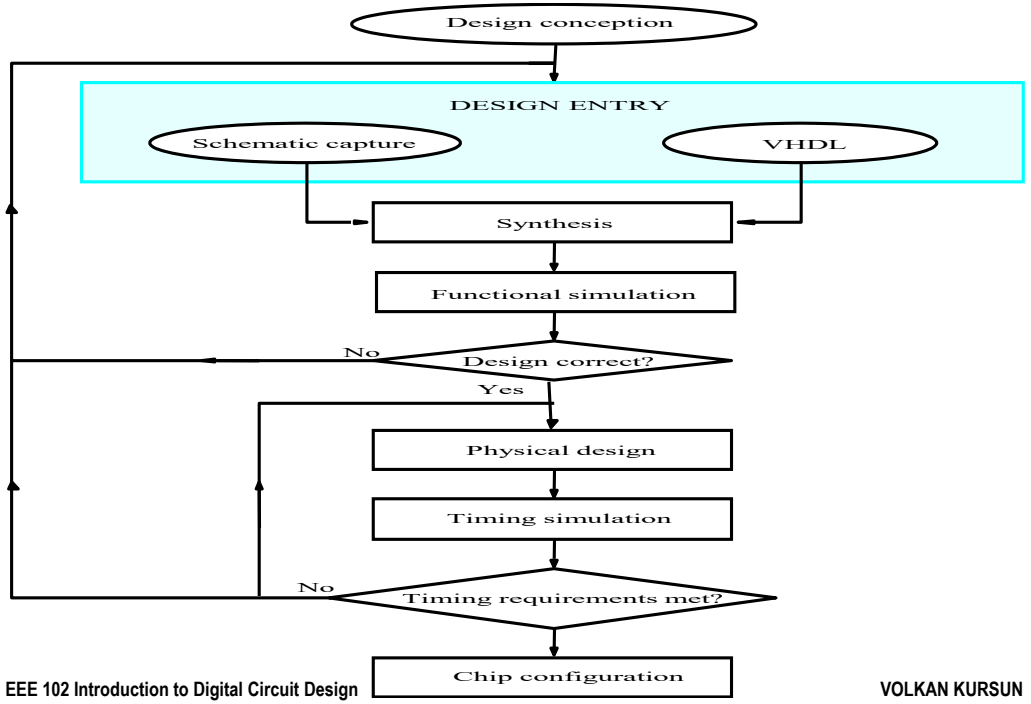
Phase-3) Functional Simulation: functional simulator is used to verify functionality. The logic expressions produced by the synthesis tool are used by the functional simulator, assuming the expressions will be implemented with **perfect gates (without any propagation delays)**. The inputs to the circuit are determined by the user (stimulus generation). For each input combination, the functional simulator determines the outputs produced by the circuit. Timing diagrams can be used to examine the outputs and verify functionality.

Phase-4) Physical Design: automatically map a circuit (that is specified in the form of logic expressions) onto a target technology. **Placement** of specific logic circuits from a library and **routing** of wires to connect the components.

Phase-5) Timing Analysis: timing analizer evaluates the propagation delays based on physical design. Detailed physical information from the logic circuits and wires are used for accurate timing analysis.

Phase-6) Circuit Implementation: after verifying both functionality and timing, the circuit is implemented on an actual chip. There are two ways for implementation: **6A) Custom implementation**: a custom chip is manufactured for implementing the design. **6B) Programmable devices**: a programmable hardware device is used for mapping the design onto silicon.

Complete Design Flow



VHDL

- VHDL was adopted as an IEEE standard (IEEE 1076) in 1987. The standard was revised in 1993, 2000, and 2002.
- Documentation language for describing complex digital circuits: **provides a common (standard) way** of documenting circuits
- A desired circuit can be described in two primary ways:

1) Structural Modelling: use VHDL statements to describe the structure of a circuit using logic gates such as AND and OR.

2) Behavioural Modelling: provides features to model the behaviour of a digital circuit. More abstract (high level) description of a logic circuit without the structural details of the actual implementation.

- CAD tools are used to synthesize the VHDL code into hardware implementation

Structural Specification

- Logic signals are represented as data objects
- Data objects can be of various types
- BIT data type: BIT objects can have only two values, 0 or 1
- **Two constructs** are used for design entry:

1) Entity construct: input and output signals are declared in the entity. The input and output signals are called ports (keyword PORT). Each PORT represents a signal with an associated mode that specifies whether it is an input (IN) or an output (OUT) signal. An entity must also be assigned a name.

VHDL has **rules for signal names**: signal names **must start with a letter** and can include any letter or number as well as the underscore character. A signal name cannot be a keyword.

Structural Specification

Two constructs are used for design entry:

1) Entity construct:

2) Architecture construct: An entity specifies the input and output signals but does not provide any information about what the circuit does. The **circuit's functionality must be specified with the architecture construct**.

Architecture must be given a name. VHDL has built-in support for the following Boolean operators: AND, OR, NOT, NAND, NOR, XOR, and XNOR.

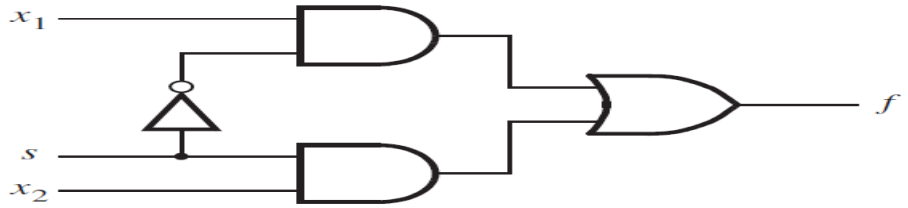
Note: VHDL does not assume any precedence for the logic operators. Parentheses must be used to clarify the precedence of operations in the expressions.

Example: f <= NOT s AND x_1 OR s AND x_2 would cause a **compile-time error**

Instead write: f <= (NOT s AND x_1) OR (s AND x_2)

Structural Specification Example

- Write the structural VHDL code for a 2-to-1 multiplexer



Step-1: **write the entity construct** to declare the inputs and output

```
ENTITY example1 IS
    PORT ( x1, x2, s  : IN   BIT ;
           f          : OUT  BIT ) ;
END example1 ;
```

Entity name

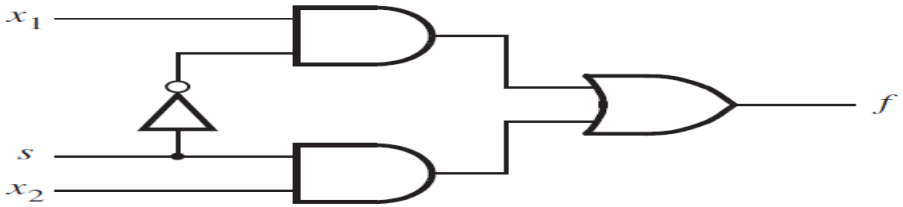
Input ports

Inputs and output are BIT type

Output port

Structural Specification Example

- Write the structural VHDL code for a 2-to-1 multiplexer



Step-2: **write the architecture construct** to specify the functionality

ARCHITECTURE name

```
ARCHITECTURE LogicFunc OF example1 IS
BEGIN
    f <= (NOT s AND x1) OR (s AND x2) ;
END LogicFunc ;
```

ARCHITECTURE of the ENTITY named example1

VHDL assignment operator

STD_LOGIC Data Type

- Was introduced in the IEEE 1164 standard revision as the standard data type for representing logic signals
- Unlike the BIT data type which can have only two values, the STD_LOGIC data type can have four values: 0, 1, Z, and –
- Z: high-impedance
- -: unspecified logic value (don't care)
- STD_LOGIC data type did not exist in the original IEEE 1076 standard. Compiler directives are needed to be able to use the STD_LOGIC type: the definition of STD_LOGIC is in a set of files (a package) in the IEEE library. The package is named std_logic_1164 and all the files in this package will be included in the following example

STD_LOGIC Data Type Example

- Change the data type from BIT to STD_LOGIC in the 2-to-1 multiplexer example:

Declare that the code will make use of the LIBRARY named ieee

All files within the std_logic_1164 package are included

Inputs and output are STD_LOGIC type

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY example3 IS
    PORT ( x1, x2, s  : IN  STD_LOGIC ;
           f          : OUT STD_LOGIC ) ;
END example3 ;

ARCHITECTURE LogicFunc OF example3 IS
BEGIN
    f <= (NOT s AND x1) OR (s AND x2) ;
END LogicFunc ;
```


VHDL Code for 2-to-1 MUX in POS

```
ENTITY example IS
    PORT(x1,x2,s: IN STD_LOGIC;
         f      :OUT STD_LOGIC);
END example;

ARCHITECTURE example_arch OF example IS
BEGIN
    f <= (s OR x1) AND ( (NOT s) OR x2);
END example_arch;
```

declare inputs & outputs

describe the function of the circuit

assignment operator

Behavioral Specification

- Instead of describing the structure of the multiplexer, describe the behavior of the multiplexer with an if-else statement

```
-- Behavioral specification
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY example4 IS
      PORT ( x1, x2, s  : IN   STD_LOGIC ;
             f          : OUT  STD_LOGIC ) ;
END example4 ;

ARCHITECTURE Behavior OF example4 IS
BEGIN
    PROCESS ( x1, x2, s )
    BEGIN
          IF s = '0' THEN
                f <= x1 ;
          ELSE
                f <= x2 ;
          END IF ;
    END PROCESS ;
END Behavior ;
```

Comment line

Include the entire std_logic_1164 package from the LIBRARY named ieee

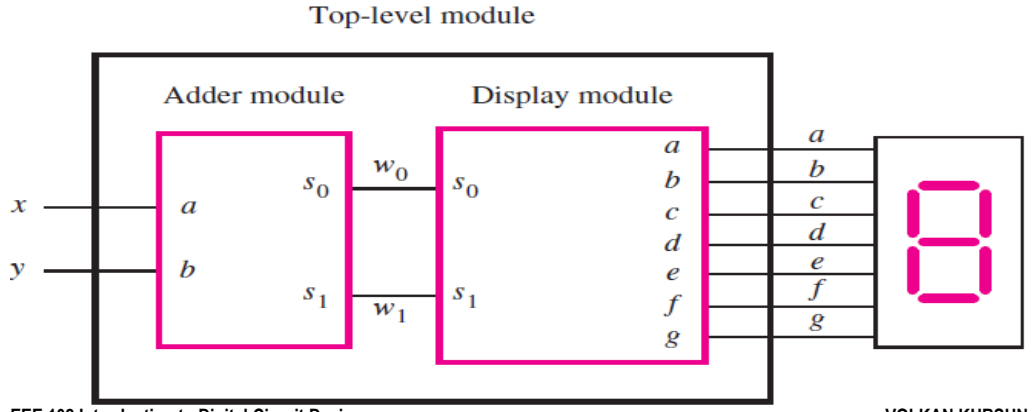
Sensitivity list: identifies the signals that affect the outputs produced by the PROCESS

Sequential assignment with if-else statement

Sequential assignment statements must be placed inside a PROCESS construct

Hierarchical VHDL Code

- Hierarchical design can help cope with complexity: a top level entity includes multiple instances of lower level entities
- Example: describe the circuit that generates the arithmetic sum of two inputs x and y and displays the decimal sum on a 7-segment display



Hierarchical VHDL Code Example

- VHDL codes of the adder and display entities:

```
-- An adder circuit
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY adder IS
     PORT ( a, b     : IN   STD_LOGIC ;
            s1, s0   : OUT  STD_LOGIC ) ;
END adder ;

ARCHITECTURE HalfAdd OF adder IS
BEGIN
     s1 <= a AND b ;
     s0 <= a XOR b ;
END Behavior ;
```

```
-- A circuit for driving a 7-segment display
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY display IS
     PORT ( s1, s0             : IN   STD_LOGIC ;
            a, b, c, d, e, f, g : OUT  STD_LOGIC ) ;
END display ;

ARCHITECTURE HalfAdd OF display IS
BEGIN
     a <= NOT s0 ;
     b <= '1' ;
     c <= NOT s1 ;
     d <= NOT s0 ;
     e <= NOT s0 ;
     f <= NOT s1 AND NOT s0 ;
     g <= s1 AND NOT s0 ;
END Behavior ;
```


Hierarchical VHDL Code Example

- Top level VHDL entity:

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY adder_display IS
     PORT ( x, y               : IN   STD_LOGIC ;
            a, b, c, d, e, f, g : OUT  STD_LOGIC ) ;
END adder_display ;

ARCHITECTURE Structure OF adder_display IS
     SIGNAL w1, w0 : STD_LOGIC ;
     COMPONENT adder IS
          PORT ( a, b     : IN   STD_LOGIC ;
                 s1, s0   : OUT  STD_LOGIC ) ;
     END COMPONENT ;
     COMPONENT display IS
          PORT ( s1, s0             : IN   STD_LOGIC ;
                 a, b, c, d, e, f, g : OUT  STD_LOGIC ) ;
     END COMPONENT ;
BEGIN
     U1: adder PORT MAP ( x, y, w1, w0 ) ;
     U2: display PORT MAP ( w1, w0, a, b, c, d, e, f, g ) ;
END Structure ;
```

Internal signals w_1, w_0 are declared within the architecture (they are not PORTs but rather internal SIGNALs)

COMPONENT declaration for the entity named adder: allows the adder entity to be used as a COMPONENT (subcircuit) in the architecture body

Instantiates the adder entity as a subcircuit

x, y input ports of the top-level adder_display entity are connected to the a, b input ports of the adder entity

w1, w0 internal signals of the top-level adder_display entity are connected to the s1, s0 output ports of the adder entity