Lecture 2

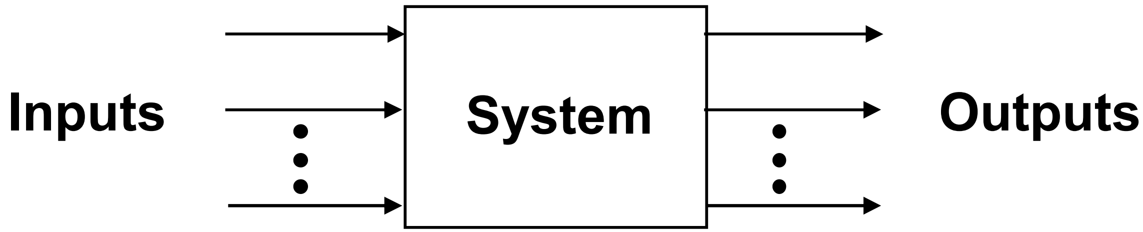
Describing logic circuits

Learning outcomes

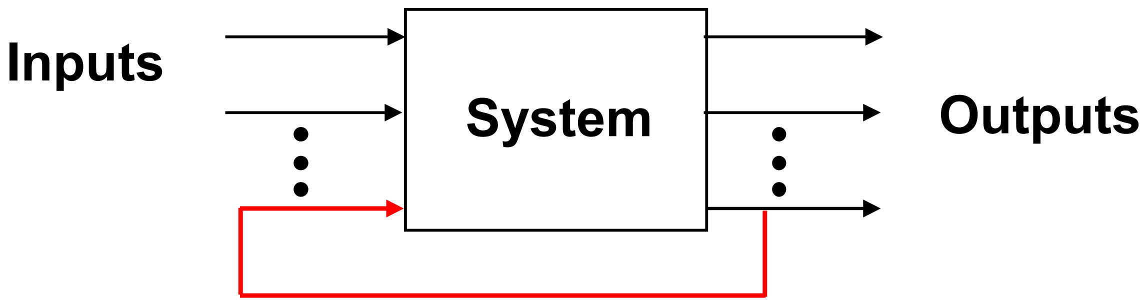
- Introduce the basic laws and rules of Boolean Algebra.
- Apply DeMorgan's theorem to Boolean expression.
- Simplify Boolean expressions using laws and rules of Boolean Algebra.
- Describe combinational logic circuits with Boolean expressions, truth table and/or gate-level circuitry.
- Specify Boolean functions in standard (canonical) forms.

Combinational versus Sequential logic

- Combinational systems are memoryless
  - The outputs depend only on the present inputs



- Sequential systems have memory
  - The outputs depend on the present inputs and on the previous inputs.



Digital circuits/Logic circuits

- The manner in which a digital circuit responds to an input is referred to as the **circuit's logic**.

- Each type of digital circuit **obeys a certain set of logic rules**:

  A fire sprinkler system should spray water if high heat is sensed and the system is set to enabled.

  A car alarm should sound if the alarm is enabled, and either the car is shaken or the door is opened.

- For this reason, **digital circuits** are also called **logic circuits**.

- Both terms are commonly used interchangeably.

Combinational Device

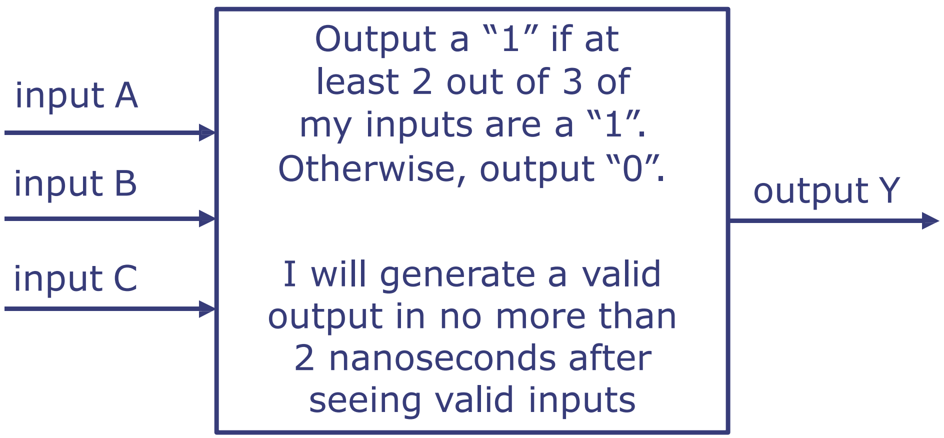
- A combinational device is a circuit element that has:

  one or more **digital inputs** "0" $\leq V_{IL}$, "1" $\geq V_{IH}$

  one or more **digital outputs** "0" $\leq V_{OL}$, "1" $\geq V_{OH}$

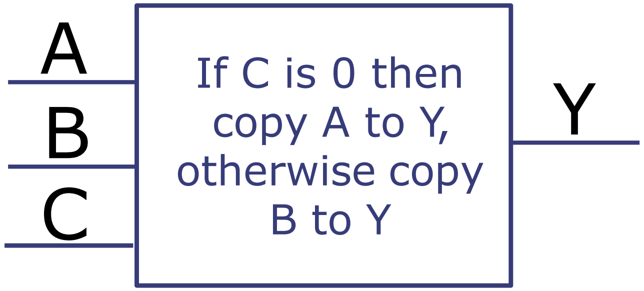
  a **functional specification** that details the value of each output for every possible combination of valid input values

  a **timing specification** consisting (at a minimum) of a propagation delay (t_{PD}): an upper bound on the required time to produce valid, stable output values from an arbitrary set of valid, stable input values



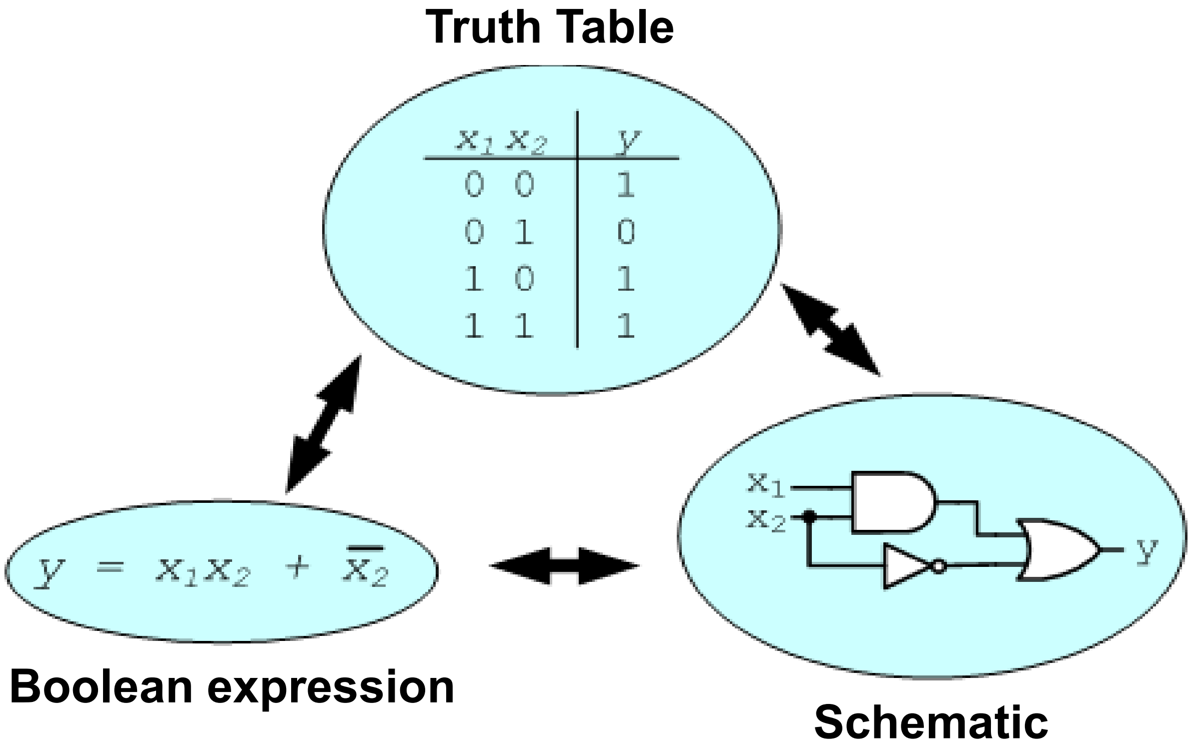
Functional Specifications

- Besides language, there are other ways to specify and represent the function of a combinational device including:
  - Truth Table
  - Boolean Algebra
  - Schematics



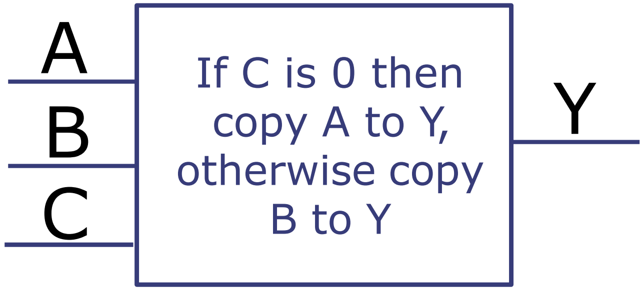
- Can convert between representations

- Truth table is the only unique representation



Truth Table Representation

- A truth table **- a tabular listing of the values of a function for** ***all possible input combinations.*** **For a N-input logic circuit, the truth table will have 2^N rows**.



Truth Table

C	B	A	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Boolean Algebra

- Boolean algebra provides a convenient mathematical framework to describe the relationship between a digital circuit's output and its inputs.

- Boolean algebra uses symbols to represent a logical expression that has one of two possible values: true or false.

- The logical expression might be *door is closed, button is pressed or fuel is low*. Writing these expressions is very tedious, and so we tend to substitute symbols such as A, B and C, etc…

- Boolean algebra allows us to specify the relationships between these symbols referred to as Boolean variables.

- Boolean algebra can be used for the design, analysis and optimization of digital circuits.

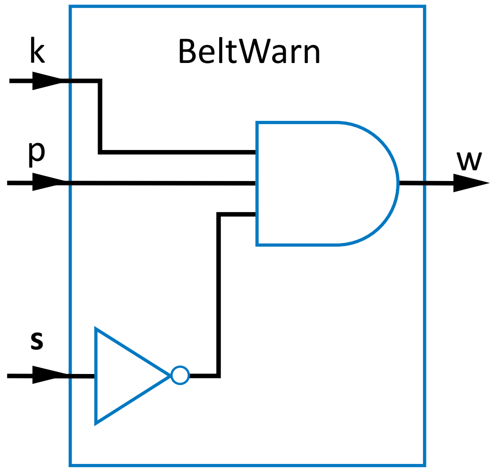
Boolean Algebra to describe Logic

- Design seat belt warning light circuit
- Sensors
  - s=1: seat belt fastened
  - k=1: key inserted
  - p=1: person in seat
- Capture Boolean expression
  - person in seat, and seat belt not fastened, and key inserted
- If we could build a digital circuit which implements this Boolean expression we could sell it as a simple seat Belt warning light circuit

w = p AND NOT(s) AND k

w=1 ➡ light ON

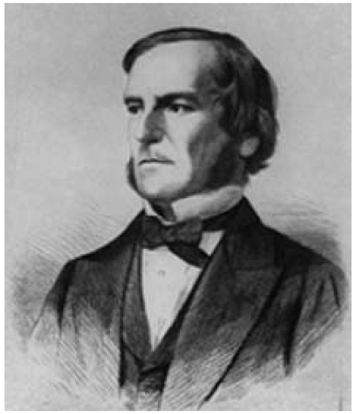
w=0 ➡ light OFF



Boolean Algebra

- Boolean Algebra can thus represent the language of logic in symbols

- Boolean logic is a branch of mathematics that deals with rules for manipulating the two logical truth values true and false.



- Named after George Boole (1815-1864)

  An English mathematician, who was first to develop and describe a formal system to work with truth values.

- Why is Boolean logic so relevant to computers?

  Direct mapping to binary digits!

  1 = true, 0 = false

Boolean Algebra

- "Traditional" algebra
  - Variable represent real numbers
  - Operators operate on variables, return real numbers

- ***Boolean Algebra***
  - Variables represent 0 or 1 only
  - Operators return 0 or 1 only
  - Basic operators
    - AND: *a AND b* returns 1 only when both a=1 and b=1
    - OR: *a OR b* returns 1 if either (or both) a=1 or b=1
    - NOT: *NOT a* returns the opposite of a (1 if a=0, 0 if a=1)

AND Operations With AND Gates

- The Boolean expression for the AND operation is

  $$X = A \bullet B$$

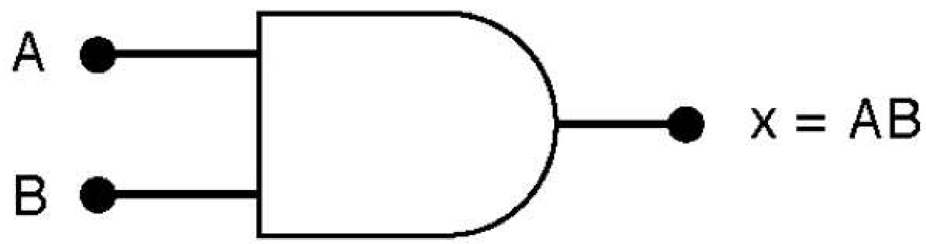
  This is read as “x equals A and B.”

  x = 1 when A = 1 and B = 1.

- Truth table and circuit symbol for a 2-input AND gate are shown below:

AND

A	B	$x = A \cdot B$
0	0	0
0	1	0
1	0	0
1	1	1



AND gate

OR Operations With OR Gates

- The Boolean expression for the OR operation is

  $$X = A + B$$

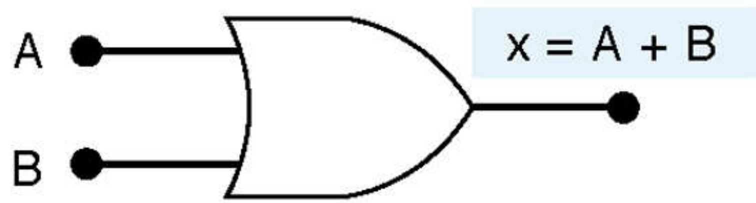
  This is read as "x equals A or B."

  X = 1 (true) when A = 1 (true) or B = 1 (true).

- Truth table and circuit symbol for a 2-input OR gate are shown below. Notice the difference between OR and AND gates.

OR

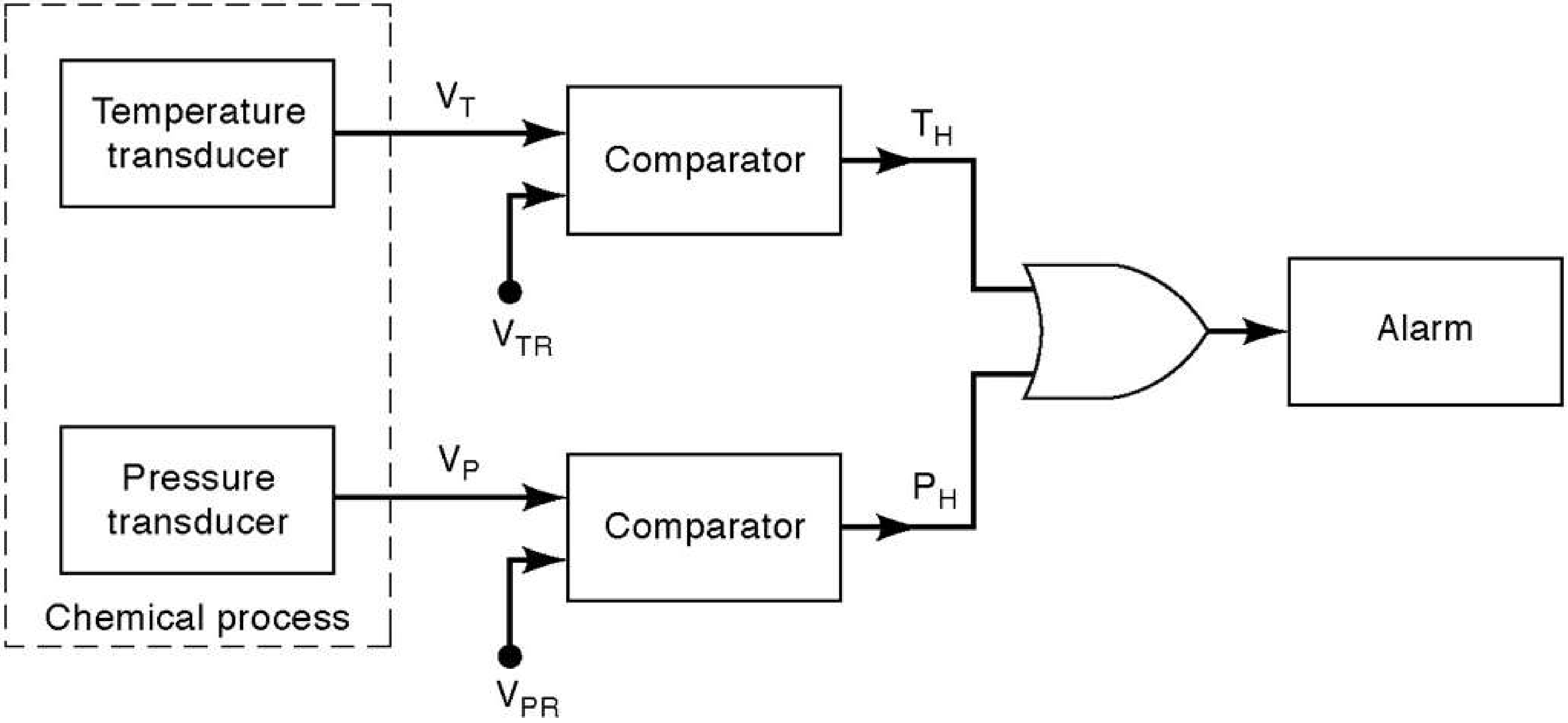
A	B	x = A + B
0	0	0
0	1	1
1	0	1
1	1	1



OR Gate

OR Operations With OR Gates

- There are many examples of applications where an output function is desired when one of multiple inputs is activated.



- In many industrial control systems, it is required to activate an output function whenever one of several inputs is activated. For example, in a chemical process, it may be desired that an alarm is activated whenever the process temperature exceeds a maximum value or whenever the pressure goes above a certain limit.

NOT Operation

- The Boolean expression for the NOT operation is

$$X = \overline{A}$$

- This is read as:

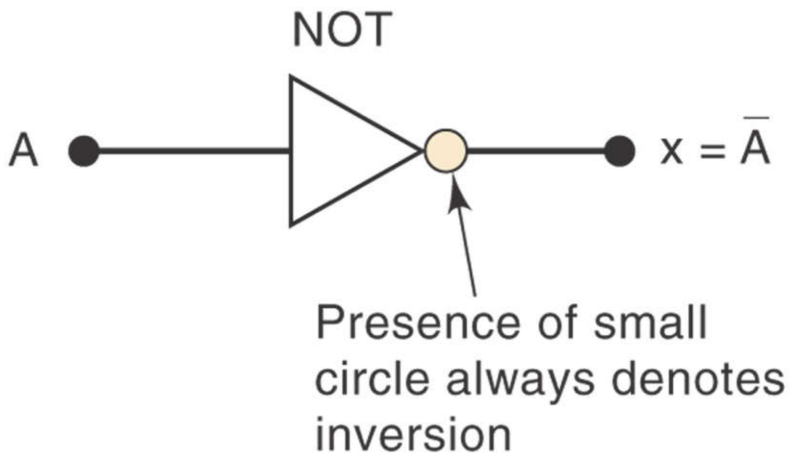
  x equals NOT A, or x equals the complement of A

  The NOT operation is most often designated by an overbar and sometimes indicated by a prime mark (‘)

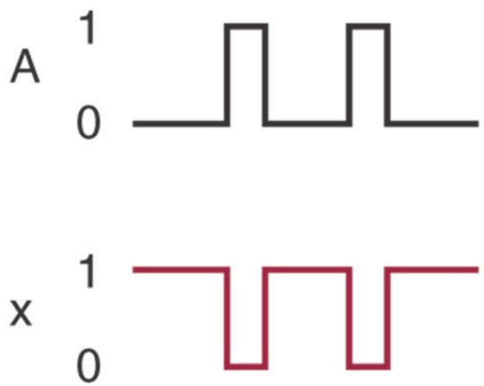
NOT

A	$x = \overline{A}$
0	1
1	0

(a)



(b)

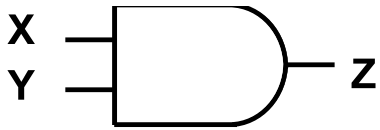


(c)

Combinational logic gates

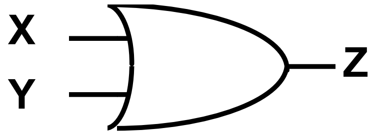
- The three basic Boolean operations (OR, AND, NOT) can describe any logic circuit.

- AND $X \cdot Y$



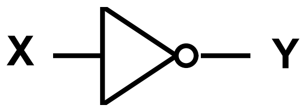
X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

- OR $X + Y$



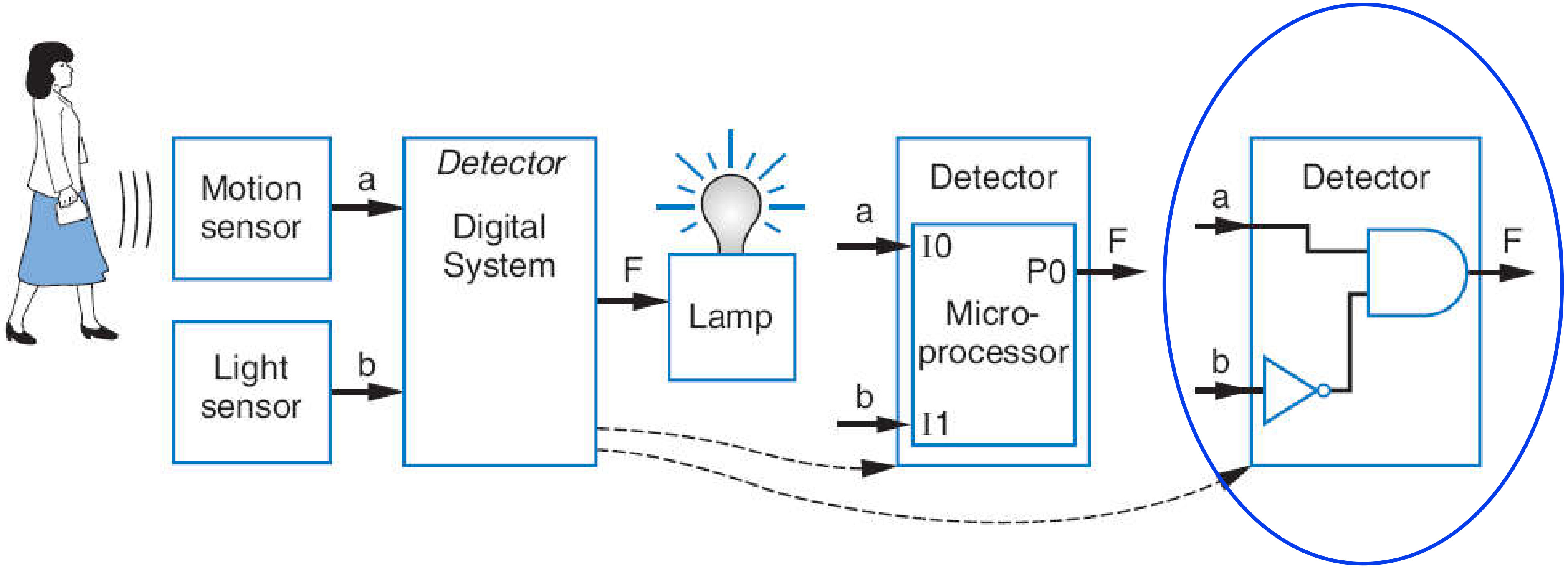
X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

- NOT $\overline{X}$



X	Y
0	1
1	0

Building Circuits Using Gates



- Motion-in-dark example
  - Turn on lamp (F=1) when motion sensed (a=1) and no light (b=0)
  - F = a AND NOT(b)
  - Build using logic gates, AND and NOT, as shown

Boolean Functions

- Boolean functions are described by expressions that consist of:

  Boolean variables, such as: x, y, etc.

  Boolean constants: 0 and 1

  Boolean operators: AND ($\cdot$), OR (+), NOT (')

  Parentheses, which can be nested

- Example: $f = x(y + w'z)$

  The dot operator is implicit and need not be written.

  A literal is a variable that may or may not be complemented.

Boolean Operator Precedence

- Rules for evaluating a Boolean expression.

- The order of evaluation in a Boolean expression is:

  1. Parentheses
  2. NOT
  3. AND
  4. OR

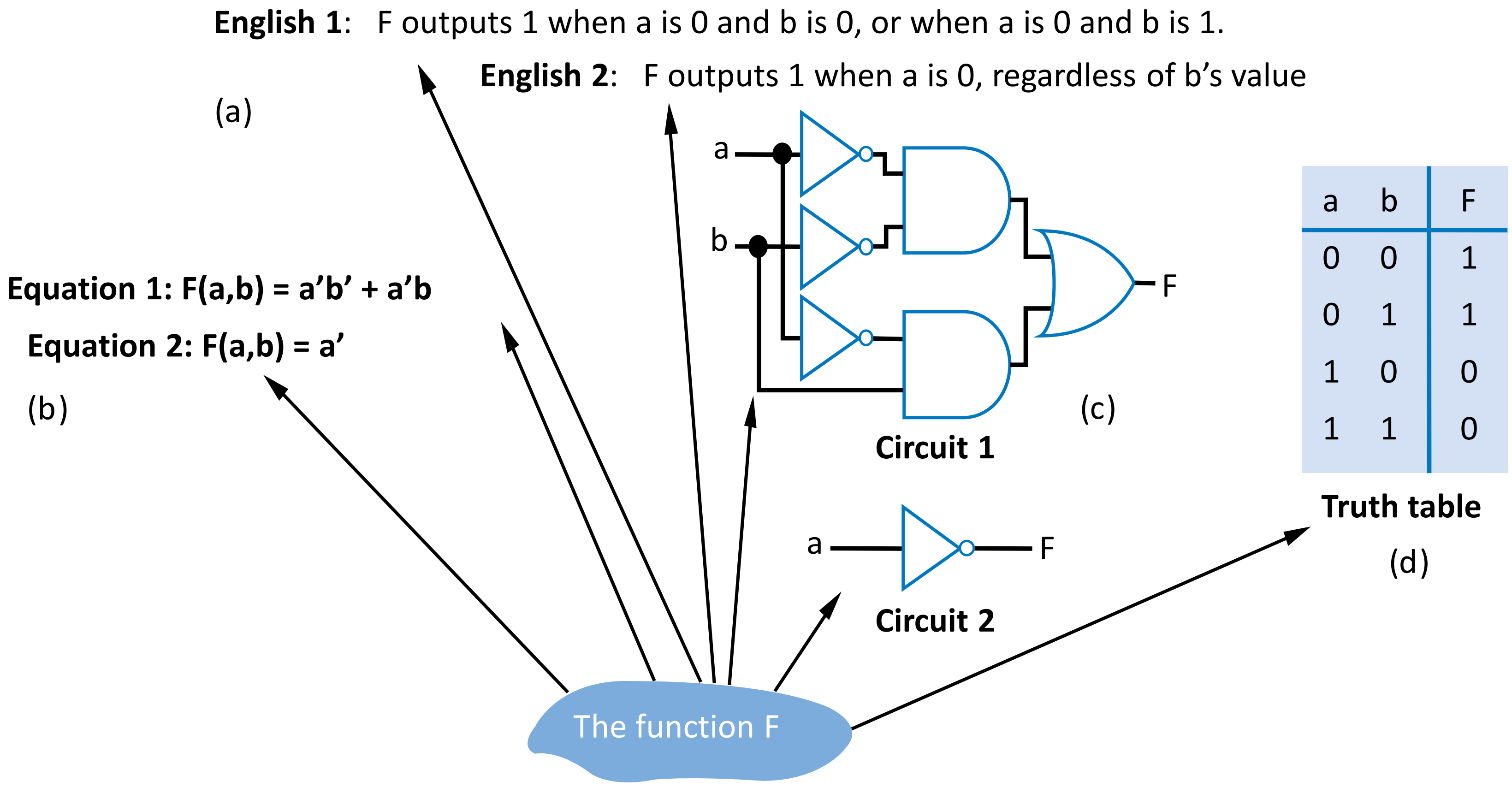
$$A = 0,\ B = 1,\ C = 1, \text{ and } \ D = 1$$

$$\begin{aligned} x &= \overline{A}BC\overline{(A + D)} \\ &= \overline{0} \cdot 1 \cdot 1 \cdot \overline{(0 + 1)} \\ &= 1 \cdot 1 \cdot 1 \cdot \overline{(0 + 1)} \\ &= 1 \cdot 1 \cdot 1 \cdot (\overline{1}) \\ &= 1 \cdot 1 \cdot 1 \cdot 0 \\ &= 0 \end{aligned}$$

- **Consequence:** Parentheses need to appear around OR expressions

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

Boolean Function Representations

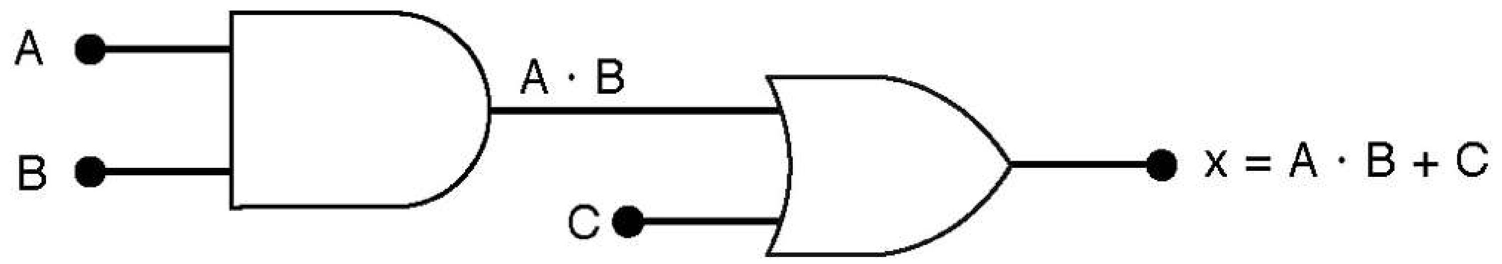


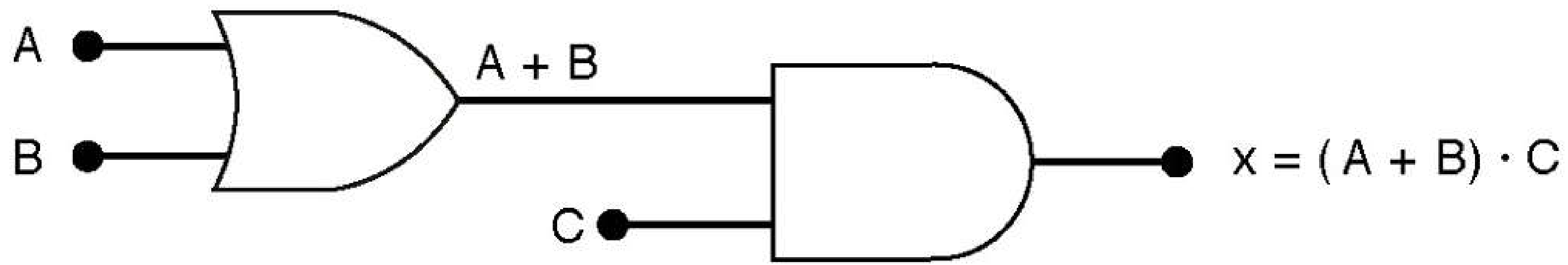
a	b	F
0	0	1
0	1	1
1	0	0
1	1	0

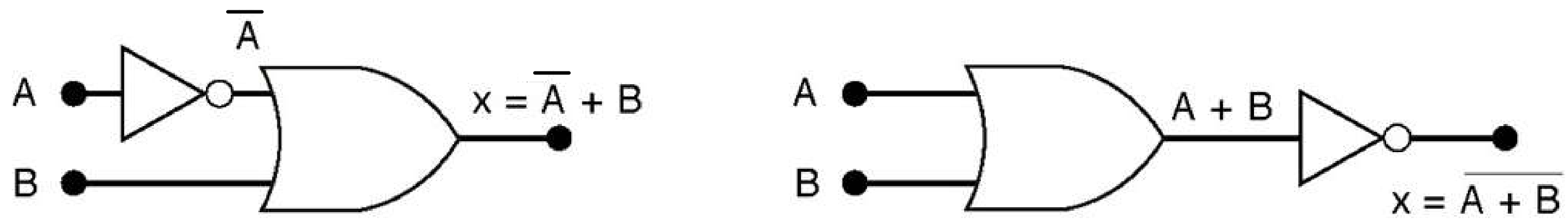
- A function can be represented in different ways
  - Above shows seven representations of the same functions F(a,b), using four different methods: English, Equation, Circuit, and Truth Table

Describing Logic Circuits Algebraically

- Examples of Boolean expressions for logic circuits:







(a) (b)

Duality Principle

- Every truth statement can yield another truth statement:

  *I exercise if I have time **AND** energy (original statement)*

  *I don't exercise if I don't have time **OR** don't have energy (dual statement)*

- The **dual** of a Boolean expression can be obtained by:

  Interchanging AND ($\cdot$) and OR (+) operators

  Interchanging constants 0's and 1's (e.g. identity: x+0=x → x.1=x)

  The complement operator does not change

- **Example**: the dual of $x(y+z')$ is $x+yz'$

- The properties of Boolean algebra appear in **dual pairs**

  If a property is proven to be true then its dual is also true, as shown in previous slide.

  The "dual" of an expression is **NOT** equal to the original expression.

Axioms and theorems of Boolean algebra

	Property	Dual Property
Identity	X + 0 = X	X • 1 = X
Null	X + 1 = 1	X • 0 = 0
Idempotency	X + X = X	X • X = X
Involution	(X')' = X	
Complementarity	X + X' = 1	X • X' = 0
Commutativity	X + Y = Y + X	X • Y = Y • X
Associativity	(X + Y) + Z = X + (Y + Z)	(X • Y) • Z = X • (Y • Z)
Distributivity	X • (Y + Z) = (X • Y) + (X • Z)	X + (Y • Z) = (X + Y) • (X + Z)
Uniting	X • Y + X • Y' = X	(X + Y) • (X + Y') = X
Absorption	X + X • Y = X (X + Y') • Y = X • Y	X • (X + Y) = X (X • Y') + Y = X + Y
Factoring	(X + Y) • (X' + Z) =X • Z + X' • Y	X • Y + X' • Z =(X + Z) • (X' + Y)
Consensus	(X • Y) + (Y • Z) + (X' • Z) =X • Y + X' • Z	(X + Y) • (Y + Z) • (X' + Z) =(X + Y) • (X' + Z)
DeMorgan's	(X + Y + ...)' = X' • Y' • ... (X • Y • ...)' = X' + Y' +…	(X • Y • ...)' = X' + Y' +… (X + Y + ...)' = X' • Y' • ...

DeMorgan's Theorem

- $(x+y)' = x'\,y'$
- $(x\,y)' = x' + y'$

Can be verified
Using a Truth Table

x	y	x'	y'	x+y	(x+y)'	x'y'	x y	(x y)'	x'+ y'
0	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	0	0	1	1
1	0	0	1	1	0	0	0	1	1
1	1	0	0	1	0	0	1	0	0

Identical

Identical

- Generalized DeMorgan's Theorem:

f '(x_1, x_2,..., x_n,0,1,+,•) = f(x_1', x_2',..., x_n',1,0,•,+)

$$(x_1 + x_2 + \cdots + x_n)' = x_1' \cdot x_2' \cdot \cdots \cdot x_n'$$

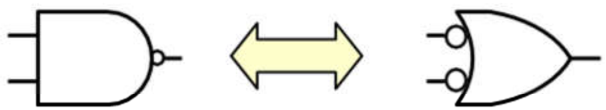
$$(x_1 \cdot x_2 \cdot \cdots \cdot x_n)' = x_1' + x_2' + \cdots + x_n'$$

DeMorgan Equivalents

- Inverting output of AND gate = inverting the inputs of OR gate
- Inverting output of OR gate = inverting the inputs of AND gate
- A function's inverse is equivalent to inverting all the inputs and changing AND to OR and vice versa

A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0

$\overline{A \bullet B}$

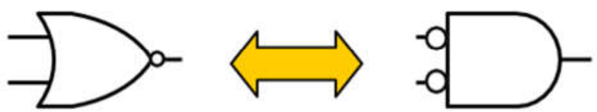


$\overline{A}+\overline{B}$

A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0

A	B	Out
0	0	1
0	1	0
1	0	0
1	1	0

$\overline{A+B}$



$\overline{A} \bullet \overline{B}$

A	B	Out
0	0	1
0	1	0
1	0	0
1	1	0

Complementing Boolean Functions

- What is the complement of $f = x'yz' + xy'z'$?
- Use DeMorgan's Theorem:

  Complement each variable and constant

  Interchange AND and OR operators

- So, what is the complement of $f = x'yz' + xy'z'$?

  Answer: $f' = (x + y' + z)(x' + y + z)$

- **Example 2:** Complement $g = (a' + bc)d' + e$
- **Answer:** $g' = (a(b' + c') + d)e'$

Algebraic Manipulation of Boolean Expressions

- The objective is to acquire skills in manipulating Boolean expressions, to transform them into simpler form.

- **Example 1:** prove $x + xy = x$ (absorption theorem)

- **Proof:** $x + xy = x \cdot 1 + xy$ — $x \cdot 1 = x$

  $= x \cdot (1 + y)$ — Distributivity

  $= x \cdot 1 = x$ — $(1 + y) = 1$

- **Example 2:** prove $x + x'y = x + y$

Proof: $x + x'y = (x + x')(x + y)$ — Distributivity

$= 1 \cdot (x + y)$ — $(x + x') = 1$

$= x + y$

Simplifying Boolean functions

- We can use Boolean algebra to simplify a function so that it contains the smallest number of literals
- A literal is a variable that may or may not be complemented
- **Example**: simplify $f = ab + a'cd + a'bd + a'cd' + abcd$

Solution:

$f = ab + abcd + a'cd + a'cd' + a'bd$ (15 literals)

$f = ab + ab(cd) + a'c(d + d') + a'bd$ (13 literals)

$f = ab + a'c + a'bd$ (7 literals)

$f = ba + ba'd + a'c$ (7 literals)

$f = b(a + a'd) + a'c$ (6 literals)

$f = b(a + d) + a'c$ (5 literals only)

Algebraic Manipulation of Boolean Expressions

- **Prove that:** $xy + x'z + yz = xy + x'z$ (consensus theorem)

- **Proof:** $xy + x'z + yz$

$= xy + x'z + 1 \cdot yz$	$yz = 1 \cdot yz$
$= xy + x'z + (x + x')yz$	$1 = (x + x')$
$= xy + x'z + xyz + x'yz$	Distributive $\cdot$ over +
$= xy + xyz + x'z + x'yz$	Associative commutative +
$= xy \cdot 1 + xyz + x'z \cdot 1 + x'zy$	$xy = xy \cdot 1,\ \ x'yz = x'zy$
$= xy(1 + z) + x'z(1 + y)$	Distributive $\cdot$ over +
$= xy \cdot 1 + x'z \cdot 1$	$1 + z = 1,\ \ 1 + y = 1$
$= xy + x'z$	$xy \cdot 1 = xy,\ \ x'z \cdot 1 = x'z$

Canonical Forms

Canonical (Standard) forms

- For a Boolean function, the truth table constitutes a unique signature
- However, the same truth table can have many gate realizations and many Boolean expressions
- **It is thus useful to specify Boolean functions in a standard (canonical) form that:**
  - Allows comparison for equality.
  - Has a correspondence to the truth tables
- **Canonical forms** provides a unique algebraic signature.
- There are two **canonical forms in common usage:**
  - **Sum of Products (SOP),** also called **Sum of Minterms (SOM)**
  - **Product of Sum (POS),** also called **Product of Maxterms (POM)**

Minterms and Maxterms

- Minterm (or product term)

  ANDed product of literals

  when input is '**1**', the variable is **Not Complemented**

  When input is '**0**' , the variable is **Complemented**

  each variable appears exactly once, true or inverted (but not both)

- Maxterm (or sum term)

  ORed sum of literals

  when input is '**1**', the variable is **Complemented**

  when input is '**0**' , the variable is **Not Complemented**

  each variable appears exactly once, true or inverted (but not both)

index	x	y	Minterm	Maxterm
0	0	0	$m_0 = x'y'$	$M_0 = x + y$
1	0	1	$m_1 = x'y$	$M_1 = x + y'$
2	1	0	$m_2 = xy'$	$M_2 = x' + y$
3	1	1	$m_3 = xy$	$M_3 = x' + y'$

Minterms and Maxterms

- For n variables, there are 2^n Minterms and Maxterms indexed from 0 to $2^n - 1$.

x	y	z	index	Minterm	Maxterm
0	0	0	0	$m_0 = x'y'z'$	$M_0 = x + y + z$
0	0	1	1	$m_1 = x'y'z$	$M_1 = x + y + z'$
0	1	0	2	$m_2 = x'yz'$	$M_2 = x + y' + z$
0	1	1	3	$m_3 = x'yz$	$M_3 = x + y' + z'$
1	0	0	4	$m_4 = xy'z'$	$M_4 = x' + y + z$
1	0	1	5	$m_5 = xy'z$	$M_5 = x' + y + z'$
1	1	0	6	$m_6 = xyz'$	$M_6 = x' + y' + z$
1	1	1	7	$m_7 = xyz$	$M_7 = x' + y' + z'$

Maxterm M_i is the **complement** of Minterm m_i

$M_i = mi'$ and $m_i = M_i'$

Sum-of-products (SoP) canonical form

- Also known as Sum-Of-Minterms (SOM)

x y z	f	Minterm
0 0 0	0	
0 0 1	0	
0 1 0	1	$m_2 = x'yz'$
0 1 1	1	$m_3 = x'yz$
1 0 0	0	
1 0 1	1	$m_5 = xy'z$
1 1 0	0	
1 1 1	1	$m_7 = xyz$

Sum of Minterm entries that evaluate to '**1**'

Focus on the '**1**' entries

$$f = m_2 + m_3 + m_5 + m_7$$

$$f = \sum m(2, 3, 5, 7)$$

$$f = x'yz' + x'yz + xy'z + xyz$$

Examples of SoP expressions

- $f(a,b,c,d) = \sum m(2,3,6,10,11)$

- $f(a,b,c,d) = m_2 + m_3 + m_6 + m_{10} + m_{11}$

- $f(a,b,c,d) = a'b'cd' + a'b'cd + a'bcd' + ab'cd' + ab'cd$

- $g(a,b,c,d) = \sum m(0,1,12,15)$

- $g(a,b,c,d) = m_0 + m_1 + m_{12} + m_{15}$

- $g(a,b,c,d) = a'b'c'd' + a'b'c'd + abc'd' + abcd$

Product-of-Sums (PoS) canonical form

- Also known as Product-Of-Maxterms (POM)

x y z	f	Maxterm
0 0 0	0	$M_0 = x + y + z$
0 0 1	0	$M_1 = x + y + z'$
0 1 0	1	
0 1 1	1	
1 0 0	0	$M_4 = x' + y + z$
1 0 1	1	
1 1 0	0	$M_6 = x' + y' + z$
1 1 1	1	

Product of Maxterm entries that evaluate to '**0**'

Focus on the '**0**' entries

$$f = M_0 \cdot M_1 \cdot M_4 \cdot M_6$$

$$f = \prod M(0, 1, 4, 6)$$

$$f = (x + y + z)(x + y + z')(x' + y + z)(x' + y' + z)$$

Examples of PoS expressions

- $f(a,b,c,d) = \prod M(1,3,11)$
- $f(a,b,c,d) = M_1 \cdot M_3 \cdot M_{11}$
- $f(a,b,c,d) = (a+b+c+d')(a+b+c'+d')\ (a'+b+c'+d')$
- $g(a,b,c,d) = \prod M(0,5,13)$
- $g(a,b,c,d) = M_0 \cdot M_5 \cdot M_{13}$
- $f(a,b,c,d) = (a+b+c+d)(a+b'+c+d')\ (a'+b'+c+d')$

S-o-P, P-o-S, and de Morgan's theorem

- Sum-of-products

  $F' = A'B'C' + A'BC' + AB'C'$

- Apply de Morgan's

  $(F')' = (A'B'C' + A'BC' + AB'C')'$

  $F = (A + B + C)\ (A + B' + C)\ (A' + B + C)$

- Product-of-sums

  $F' = (A + B + C')\ (A + B' + C')\ (A' + B + C')\ (A' + B' + C)\ (A' + B' + C')$

- Apply de Morgan's

  $(F')' = (\ (A + B + C')(A + B' + C')(A' + B + C')(A' + B' + C)(A' + B' + C')\ )'$

  $F = A'B'C + A'BC + AB'C + ABC' + ABC$

Conversion between canonical forms

- To convert a Boolean function f from one canonical form to another, interchange the symbols $\sum$ and $\prod$ and list those indices missing from the original form:

  SOP: $f = m_0 + m_2 + m_3 + m_5 + m_7 = \sum m(0, 2, 3, 5, 7)$

  POS: $f = M_1 \cdot M_4 \cdot M_6 = \prod M(1, 4, 6)$

x y z	f	Minterms	Maxterms
0 0 0	1	$m_0 = x'y'z'$	
0 0 1	0		$M_1 = x + y + z'$
0 1 0	1	$m_2 = x'yz'$	
0 1 1	1	$m_3 = x'yz$	
1 0 0	0		$M_4 = x' + y + z$
1 0 1	1	$m_5 = xy'z$	
1 1 0	0		$M_6 = x' + y' + z$
1 1 1	1	$m_7 = xyz$	

Complement of a boolean function

- Given a Boolean function f:

$$f(x, y, z) = \sum m(0, 2, 3, 5, 7) = \prod M(1, 4, 6)$$

- Then, the complement f' is:

$$f'(x, y, z) = \prod M(0, 2, 3, 5, 7) = \sum m(1, 4, 6)$$

x y z	f	f'
0 0 0	1	0
0 0 1	0	1
0 1 0	1	0
0 1 1	1	0
1 0 0	0	1
1 0 1	1	0
1 1 0	0	1
1 1 1	1	0

The complement of a Sum-of-Products is a Product-of-Sums with the same indices.

The complement of a Product-of-Sums is a Sum-of-Products with the same indices.

Incompletely specified functions – Don’t Care

- Up until now we have assumed that for every possible input condition the circuit specification explicitly requires a particular output from the circuit.
- However, in some applications of logic circuits it is impossible for certain input conditions to occur.
- In these cases, the output value need not be defined
- Instead, the output value is defined as a “don't care”
- By placing “don't cares” (an “X” entry) in the function table or map, the cost of the logic circuit may be lowered.

Incompletely specified functions – Don't Care

- Consider a Boolean function f defined only for inputs ranging from 0 to 9 with:
  - f outputting a zero if the input is 0 to 4.
  - f outputting a one if the input is 5 to 9.
- The output of the function is a "don't care" (an "X" entry) if an input from 10 to 15. should never be encountered in practice.

a b c d	f
0 0 0 0	0
0 0 0 1	0
0 0 1 0	0
0 0 1 1	0
0 1 0 0	0
0 1 0 1	1
0 1 1 0	1
0 1 1 1	1
1 0 0 0	1
1 0 0 1	1
1 0 1 0	X
1 0 1 1	X
1 1 0 0	X
1 1 0 1	X
1 1 1 0	X
1 1 1 1	X

$$f = \sum[\underbrace{m(5,6,7,8,9)}_{\textbf{Minterms}} + \underbrace{d(10,11,12,13,14,15)}_{\textbf{Don't Cares}}]$$

$$f = \prod[\underbrace{M(0,1,2,3,4)}_{\textbf{Maxterms}} . \underbrace{D(10,11,12,13,14,15)}_{\textbf{Don't Cares}}]$$

Notation for incompletely specified functions

- Don't cares and canonical forms
  - so far, only represented on-set
  - also represent don't-care-set
  - need two of the three sets (on-set, off-set, dc-set)

- Another example of canonical representations of function:

  Z = m0 + m2 + m4 + m6 + m8 + d10 + d11 + d12 + d13 + d14 + d15

  $Z = \Sigma$ [m(0,2,4,6,8) + d(10,11,12,13,14,15)]

  Z = M1 • M3 • M5 • M7 • M9 • D10 • D11 • D12 • D13 • D14 • D15

  $Z = \Pi$ [M(1,3,5,7,9) • D(10,11,12,13,14,15)]

Canonical forms

- Any Boolean function can be expressed as a Sum-of-Products and as a Product-of-Sums. Canonical forms can be determined directly from the truth table.
- Canonical forms provide a **unique algebraic signature**.
- Canonical forms contain a larger number of literals

  Because the Minterms (and Maxterms) must contain, by definition, all the variables either complemented or not.
- Canonical form $\neq$ minimal form (which may not be unique)

F in canonical form:

$$\begin{aligned} F(A, B, C) &= \Sigma m(1,3,5,6,7) \\ &= m1 + m3 + m5 + m6 + m7 \\ &= A'B'C + A'BC + AB'C + ABC' + ABC \end{aligned}$$

Canonical form $\neq$ minimal form

$$\begin{aligned} F(A, B, C) &= A'B'C + A'BC + AB'C + ABC + ABC' \\ &= (A'B' + A'B + AB' + AB)C + ABC' \\ &= ((A' + A)(B' + B))C + ABC' \\ &= C + ABC' \\ &= ABC' + C \\ &= AB + C \end{aligned}$$

F in canonical form:

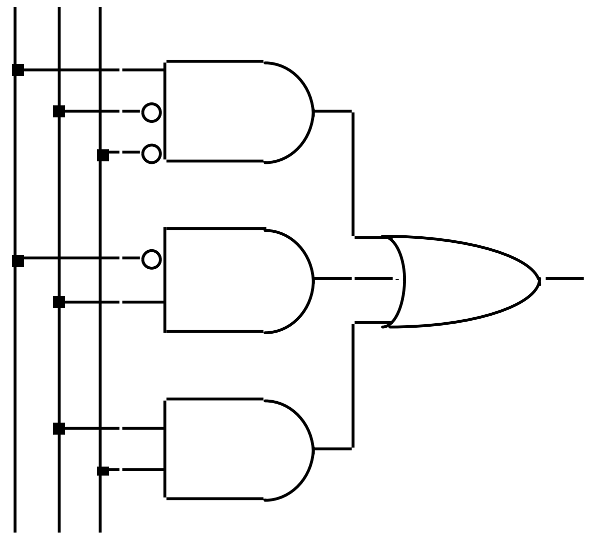
$$\begin{aligned} F(A, B, C) &= \Pi M(0,2,4) \\ &= M0 \bullet M2 \bullet M4 \\ &= (A + B + C)\,(A + B' + C)\,(A' + B + C) \end{aligned}$$

canonical form $\neq$ minimal form

$$\begin{aligned} F(A, B, C) &= (A + B + C)\,(A + B' + C)\,(A' + B + C) \\ &= (A + B + C)\,(A + B' + C) \\ &\quad\ (A + B + C)\,(A' + B + C) \\ &= (A + C)\,(B + C) \end{aligned}$$

Implementations of two-level logic

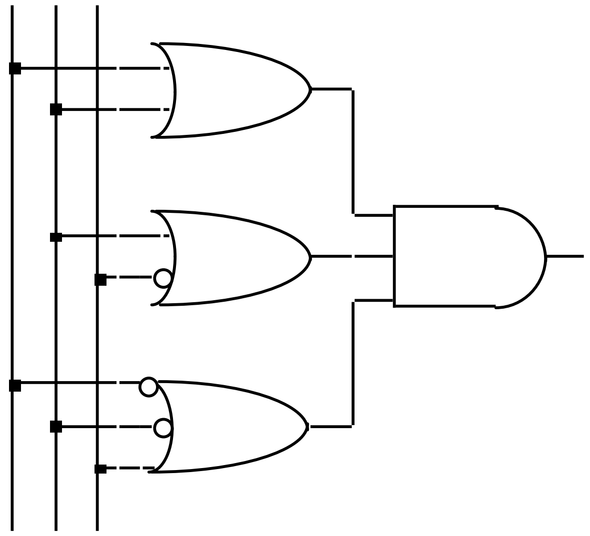
- We can implement directly any canonical form with two levels of gates.



- Sum-of-products

  AND gates to form product terms (minterms)

  OR gate to form sum

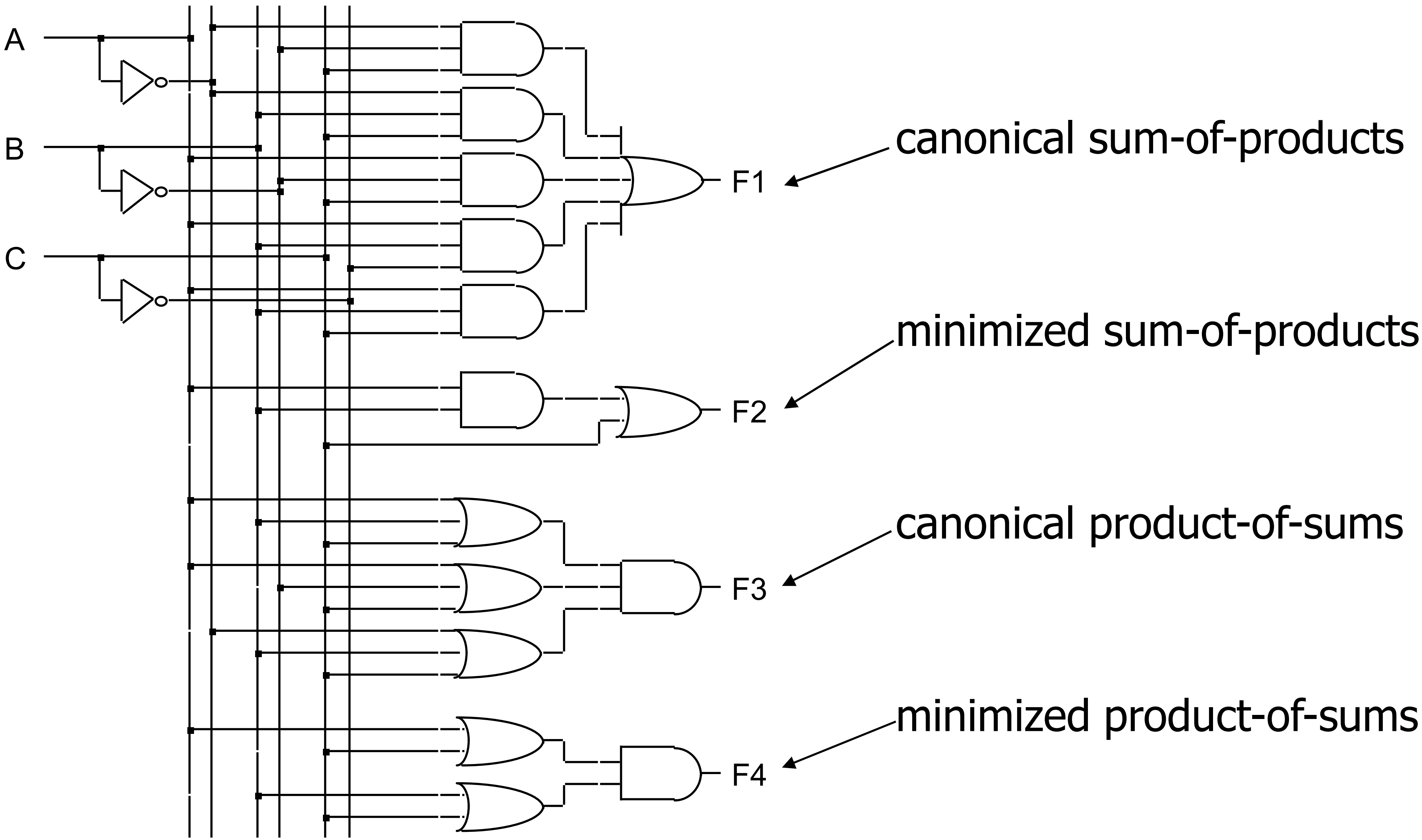


- Product-of-sums

  OR gates to form sum terms (maxterms)

  AND gates to form product

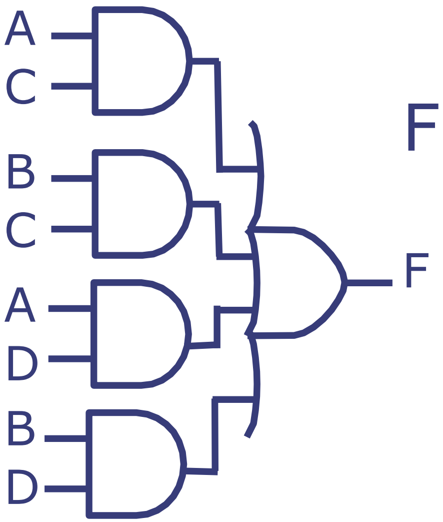
2-level implementations of F = AB + C



Multi-level logic

- We can often reduce the number of gates by using more logic levels than an SOP:

  Find common subexpressions and factor them out into independent variables

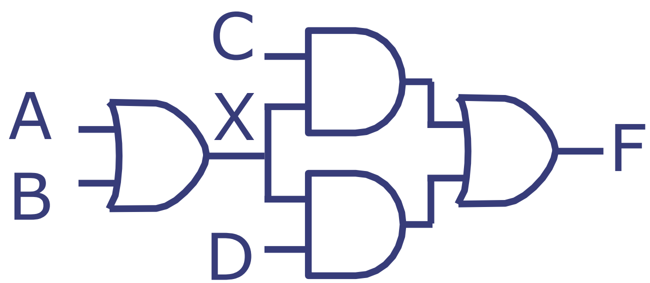


$$F = A \cdot C + B \cdot C + A \cdot D + B \cdot D$$

$$F = (A+B) \cdot C + (A+B) \cdot D$$

$$X = A + B$$

$$F = X \cdot C + X \cdot D$$



- Multi-level simplification has no well-defined optimum
  - Adding levels may reduce gates but increase delay

Multi-level logic

- Advantages
  - circuits may be smaller
  - gates have a smaller number of inputs
  - circuits may be faster

- Disadvantages
  - more difficult to design
  - tools for optimization are not as good as for two-level
  - analysis is more complex

Optimization Strategy to simplify implementation

- Reduce number of inputs

  literal: input variable (complemented or not)

  - Examples (all the same function):
    - $F = BD + AB'C + AC'D'$ $\quad L = 8$
    - $F = BD + AB'C + AB'D' + ABC'$ $\quad L = 11$
    - $F = (A + B)(A + D)(B + C + D')(B' + C' + D)$ $\quad L = 10$

  fewer literals means less transistors (smaller circuits) with cost of logic gate approximately 2 transistors per literal

  fewer inputs implies faster gates (gates are smaller and thus also faster)

- Reduce number of gates

  fewer gates (and the packages they come in) means smaller circuits, which directly influences manufacturing costs

- Reduce number of levels of gates

  fewer level of gates implies reduced signal propagation delays

Summary

- Boolean algebra is a mathematical tool used in the analysis, design and optimization of digital circuits.
- The basic Boolean operations are the OR, AND and NOT operations.
- In practice, tools using Boolean simplification and other techniques are used to synthesize a circuit that meets certain area, delay, and power specifications.
- For a combinational circuit, the truth table constitutes a unique signature.
- However, the same truth table can have many gate realizations and many Boolean expressions.
- Any combinational circuit can be described using the two general forms for logic expressions: the sum-of-products form and product-of-sums form.

Acknowledgments

- Credit is acknowledged where credit is due. Please refer to the full list of references.