

ATME COLLEGE OF ENGINEERING

13th KM Stone, Bannur Road, Mysore - 560 028



A T M E
College of Engineering

DEPARTMENT OF CSE- CYBER SECURITY



LECTURE NOTES

COURSE: DIGITAL DESIGN AND COMPUTER

ORGANISATION

COURSE CODE: 1BCS303

SEMESTER: III

COURSE COORDINATOR

Mrs. Razikha Amreen M I

(ACADEMIC YEAR 2026-27)

INSTITUTIONAL VISION AND MISSION

Vision

Development of academically excellent, culturally vibrant, socially responsible and globally competent human resources.

Mission

- To keep pace with advancements in knowledge and make the students competitive and capable at the global level.
- To create an environment for the students to acquire the right physical, intellectual, emotional and moral foundations and shine as torch bearers of tomorrow's society.
- To strive to attain ever-higher benchmarks of educational excellence.

DEPARTMENT VISION AND MISSION

Vision

To nurture competent and ethical cyber security professionals capable of contributing to secure and resilient digital ecosystems.

Mission

M1: To impart strong foundational knowledge in cyber security through effective teaching–learning practices and a continuously evolving curriculum that addresses emerging cyber threats and technologies.

M2: To promote research, innovation, and industry collaboration in emerging areas of cyber security to address real-world digital security challenges.

M3: To instill professional ethics, teamwork, leadership qualities, and a sense of social responsibility in addressing real-world cyber challenges.

M4: To foster a culture of lifelong learning among students by encouraging higher studies, professional development, and active engagement with the global cyber security community.

Program Educational Objectives

PEO1 –Excel as cyber security professionals or entrepreneurs by applying strong fundamentals of cyber security engineering.

PEO2 – Pursue higher studies, research, and professional certifications while engaging in lifelong learning to keep pace with evolving cyber technologies and threats.

PEO3 – Exhibit leadership, teamwork, professional ethics, and social responsibility while addressing cyber security challenges in multidisciplinary environments.

Program Specific Outcomes

PSO1 – Apply principles of computer science, programming, algorithms, operating systems, and computer networks to design and develop efficient and reliable computing solutions.

PSO2 – Identify, analyze, and mitigate cyber threats and vulnerabilities using techniques related to network security, ethical hacking, penetration testing, and digital forensics.

PSO3 – Apply cyber security policies, cyber laws, risk management practices, and incident response strategies to ensure the protection and resilience of digital systems.

Digital Design and Computer Organization

Module 1 : Introduction to Digital Design

1. Binary Logic

- Binary logic deals with variables that take on two discrete values and with operations that assume logical meaning.
- The two values the variables take may be called by different names (e.g. true/false, yes/no, etc.), but for our purpose it is convenient to think in terms of bits and assign the values 1 and 0.
- Binary logic is used to describe the manipulation and processing of binary information. It is particularly suited for the analysis and design of digital systems.
- The digital logic circuits that perform binary arithmetic are circuits whose behaviour is most conveniently expressed by means of binary variables and logical operations.

Definition of Binary Logic

- Binary logic consists of binary variables and logical operations. The variables are designated by letters of the alphabet such as A, B, C, X, Y, Z, etc., with each variable having two and only two distinct possible values: 1 and 0.
- There are three basic logical operations: (1) AND (2) OR (3) NOT.

(1) AND —

The operation is represented by a dot, or by the absence of an operator. E.g. $x \cdot y = z$ or $xy = z$ is read “x AND y is equal to z”. The logical operation AND is interpreted to mean that $z = 1$ if and only if $x = 1$ and $y = 1$; otherwise $z = 0$.

(2) OR —

This operation is represented by a plus sign. E.g. $x + y = z$ is read “x OR y is equal to z”, meaning that $z = 1$ if $x = 1$, or if $y = 1$, or if both $x = 1$ and $y = 1$. If both $x = 0$ and $y = 0$, then $z = 0$.

(3) NOT —

This operation is represented by a prime (sometimes by a bar over the variable). E.g. $x' = z$ is read “x not is equal to z”, meaning that z is what x is not. In other words, if $x = 1$, then $z = 0$; but if $x = 0$, then $z = 1$.

For each combination of the values of x and y , there is a value of z specified by the definition of the logical operation. These definitions may be listed in a compact form using truth tables. A truth table is a table of all possible combinations of the variables, showing the relation between the values that the variables may take and the result of the operation.

Truth tables for AND, OR and NOT:

x	y	$x \cdot y$
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$x + y$
0	0	0
0	1	1
1	0	1
1	1	1

x	x'
0	1
1	0

2. Switching Circuits and Binary Signals

- The use of binary variables and the application of binary logic are demonstrated by simple switching circuits: switches in series realize the AND function, and switches in parallel realize the OR function.
- Let the manual switches A and B represent two binary variables with values equal to 0 when the switch is open and 1 when the switch is closed.
- Let the lamp L represent a third binary variable equal to 1 when the light is ON and 0 when OFF.
- For switches in series, the light turns ON if A and B are closed. For switches in parallel, the light turns ON if A or B is closed.
- The two circuits can be expressed by means of binary logic with the AND and OR operations respectively:

$$L = A \cdot B \quad (\text{switches in series — logic AND})$$

$$L = A + B \quad (\text{switches in parallel — logic OR})$$

- Electronic digital circuits are sometimes called switching circuits because they behave like a switch, with the active element (such as a transistor) either conducting (switch closed) or not conducting (switch open).
- Instead of changing the switch manually, an electronic switching circuit uses binary signals to control the conduction or non-conduction state of the active element.
- Electrical signals such as voltages or currents exist throughout a digital system in either one of two recognizable binary values, i.e. logic 1 or logic 0.

3. Logic Gates

- Electronic digital circuits are also called logic circuits because, with the proper input, they establish logical manipulation paths.
- Any desired information for computing or control can be operated on by passing binary signals through various combinations of logic circuits, each signal representing a variable and carrying one bit of information.
- Logic circuits that perform the logical operations of AND, OR and NOT are known respectively as the AND gate, OR gate and NOT gate (inverter). These circuits, called gates, are blocks of hardware that produce a logic-1 or logic-0 output signal if input logic requirements are satisfied.
- The input signals x and y of a two-input gate may exist in one of four possible states: 00, 10, 11 or 01.
- AND and OR gates may have more than two inputs. A 3-input AND gate is written $F = ABC$, and a 4-input OR gate is written $G = A + B + C + D$.
- The mathematical system of binary logic is better known as Boolean or switching algebra. This algebra is conveniently used to describe the operation of complex networks of digital circuits.
- Designers of digital systems use Boolean algebra to transform circuit diagrams into algebraic expressions and vice versa.

4. Boolean Algebra

- Boolean algebra may be defined with a set of elements, a set of operators, and a number of unproved axioms or postulates.

- The postulates of a mathematical system form the basic assumptions from which it is possible to deduce the rules, theorems and properties of the system.

(a) Duality

- The Huntington postulates have been listed in pairs and designated by part (a) and part (b).
- One part may be obtained from the other if the binary operators and the identity elements are interchanged. This important property of Boolean algebra is called the duality principle.
- It states that every algebraic expression deducible from the postulates of Boolean algebra remains valid if the operators and identity elements are interchanged.

Consider the Boolean laws $A + 0 = A$ and $A \cdot 1 = A$.

The first of these laws implies Boolean addition (logical OR), and the second implies Boolean multiplication (logical AND). These laws can be derived directly from each other on the basis of the principle of duality.

The rules are:

- (i) Replace all (+) symbols by (.) symbols
- (ii) Replace all (.) symbols by (+) symbols
- (iii) Replace every 0 by 1, and every 1 by 0.

Example — consider the Boolean law $A + 0 = A$ (A is a variable). Following the rule stated above, we get $A \cdot 1 = A$, which is the dual of $A + 0 = A$ (+ is replaced by . and 0 is replaced by 1).

E.g. $A + A = A$ in relation to the logical OR operation; its dual is $A \cdot A = A$ in relation to the logical AND operation. Also, $A + A' = 1$ and $A \cdot A' = 0$; $A + 1 = 1$ and $A \cdot 0 = 0$.

Table 1 — Postulates and Theorems of Boolean Algebra

	(a)	(b)
Postulate 2	$A + 0 = A$	$A \cdot 1 = A$
Postulate 5	$A + A' = 1$	$A \cdot A' = 0$
Theorem 1	$A + A = A$	$A \cdot A = A$
Theorem 2	$A + 1 = 1$	$A \cdot 0 = 0$
Theorem 3 (Involution)	$(A')' = A$	
Postulate 3 (Commutative)	$A + B = B + A$	$AB = BA$
Theorem 4 (Associative)	$A + (B + C) = (A + B) + C$	$A(BC) = (AB)C$
Postulate 4 (Distributive)	$A(B + C) = AB + AC$	$A + BC = (A + B)(A + C)$
Theorem 5 (DeMorgan)	$(A + B)' = A' \cdot B'$	$(AB)' = A' + B'$
Theorem 6 (Absorption)	$A + AB = A$	$A(A + B) = A$

- Table 1 lists 6 theorems of Boolean algebra and 4 of the 15 postulates. These are the most basic relationships in Boolean algebra.
- The theorems, like the postulates, are listed in pairs; each relation is the dual of the one paired with it.
- The postulates are basic axioms of the algebraic structure and need no proof. The theorems must be proved from the postulates.

Proof of Basic Theorems

Theorem 1(a): $A + A = A$

Step	Justification
$A + A = (A + A) \cdot 1$	by postulate 2(b)
$= (A + A)(A + A')$	postulate 5(a)
$= A + AA'$	postulate 4(b)
$= A + 0$	postulate 5(b)
$A + A = A$	postulate 2(a)

Theorem 1(b): $A \cdot A = A$ (dual of Theorem 1(a))

Step	Justification
$A \cdot A = A \cdot A + 0$	by postulate 2(a)
$= AA + AA'$	postulate 5(b)
$= A(A + A')$	postulate 4(a)
$= A \cdot 1$	postulate 5(a)
$A \cdot A = A$	postulate 2(b)

Note: Theorem 1(b) is the dual of Theorem 1(a). Any dual theorem can be similarly derived from the proof of its corresponding pair.

Theorem 2(a): $A + 1 = 1$

Step	Justification
$A + 1 = 1 \cdot (A + 1)$	by postulate 2(b)
$= (A + A')(A + 1)$	postulate 5(a)
$= A + A' \cdot 1$	postulate 4(b)
$= A + A'$	postulate 2(b)
$A + 1 = 1$	postulate 5(a)

Theorem 2(b): $A \cdot 0 = 0$ — by duality.

Theorem 3 (Involution): $(A')' = A$

From postulate 5, we have $A + A' = 1$ and $A \cdot A' = 0$, which defines the complement of A. The complement of A' is A, and it is also $(A')'$. Therefore, since the complement is unique, we have $(A')' = A$.

Theorem 6(a): $A + AB = A$

Step	Justification
$A + AB = A \cdot 1 + AB$	postulate 2(b)
$= A(1 + B)$	postulate 4(a)
$= A(B + 1)$	postulate 3(a)
$= A \cdot 1$	theorem 2(a)
$A + AB = A$	postulate 2(b)

Theorem 6(b): $A(A + B) = A$ — by duality.

Theorem 5 — DeMorgan's Theorem

DeMorgan's first theorem states that the complement of a sum is equal to the product of the complements:

$$(A + B)' = A' \cdot B'$$

A	B	A + B	(A+B)'	A'	B'	A'·B'
0	0	0	1	1	1	1
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	0

Both columns $(A+B)'$ and $A' \cdot B'$ match for every row — proved.

DeMorgan's second theorem states that the complement of a product is equal to the sum of the complements:

$$(A \cdot B)' = A' + B'$$

A	B	A·B	(A·B)'	A'	B'	A'+B'
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

Both columns $(A.B)'$ and $A'+B'$ match for every row — proved.

Theorem 4 (Associative): $A + (B + C) = (A + B) + C$

Proof (LHS):

$$\begin{aligned} A + (B+C) &= [A + (B+C)] \cdot 1 = [A + (B+C)][(B+C) + (B+C)'] \\ &= A + BC + BC' + CB + CC' = A + B(C+C') + BC = A + B + BC = A + (B+C) \end{aligned}$$

Proof (RHS):

$$(A+B) + C = (A+B) + C \cdot (C+C') = (A+B) + CC + CC' = (A+B) + C + 0 = A + (B+C)$$

Since LHS = RHS = $A + (B + C)$, the associative law is verified.

5. Operator Precedence

The operator precedence for evaluating Boolean expressions is:

- (1) Parentheses
- (2) NOT
- (3) AND
- (4) OR

The expression inside a parenthesis must be evaluated before all other operations. The next operation that holds precedence is the complement, then follows the AND, and finally the OR.

Example — consider the expression $(A' + B)' = A' \cdot B'$. Interpreted two ways:

- The expression inside the parentheses is evaluated first, and the result is then complemented (left side).
- The complements of A and B are both evaluated first, and the result is then ANDed (right side).

Example 1 — Using basic Boolean theorems, prove:

$$(x + y)(x + z) = x + yz$$

Step	Reason
$(x+y)(x+z) = x.x + x.z + y.x + y.z$	
$= x + xz + yx + yz$	$x \cdot x = x$
$= x(1+z) + yx + yz$	
$= x + yx + yz$	$1 + z = 1$
$= x(1+y) + yz$	
$= x + yz$	$1 + y = 1$ (proved)

Example 2 — Prove:

$$xy + xz + yz' = xz + yz'$$

Step	Reason
$xy + xz + yz' = xy(z+z') + xz(y+y') + yz'(x+x')$	$x+x'=1, y+y'=1, z+z'=1$
$= xyz + xyz' + xyz + xy'z + xyz' + x'y'z'$	
$= xyz + xyz' + xy'z + x'y'z'$	$xyz+xyz' = xyz; xyz'+xy'z'=xyz'$
$= xz(y+y') + yz'(x+x')$	rearranging
$= xz + yz'$	$y+y'=1, x+x'=1$ (proved)

6. Boolean Functions

- A binary variable can take the value of 0 or 1.
- A Boolean function is an expression formed with binary variables, the two binary operators OR and

AND, the unary operator NOT, parentheses and an equal sign.

- For a given value of the variables, the function can be either 0 or 1. E.g. the Boolean function $F_1 = xyz'$. The function F_1 is equal to 1 if $x = 1$, $y = 1$ and $z' = 1$ (i.e. $z = 0$); otherwise $F_1 = 0$.
- A Boolean function may also be represented in a truth table. To represent a function of n binary variables in a truth table, we need a list of the 2^n combinations of 1's and 0's, and a column showing the combinations for which the function is equal to 1 or 0.

As shown below, there are 8 possible distinct combinations for assigning bits to three variables x , y , z :

x	y	z	$F_1 = xyz'$	$F_2 = x + y'z$	$F_3 = x'y'z + x'yz + xy'$	$F_4 = xy' + x'z$
0	0	0	0	0	0	0
0	0	1	0	1	1	1
0	1	0	0	0	0	0
0	1	1	0	0	1	1
1	0	0	0	1	1	1
1	0	1	0	1	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	0

- F_1 is equal to 1 only when $x = 1$, $y = 1$ and $z = 0$. It is 0 otherwise.
- $F_2 = 1$ if $x = 1$, or if $y = 0$ while $z = 1$.
- F_3 and F_4 each have four 1's and four 0's — in fact $F_3 = F_4 = xy' + x'z$ (F_4 is the simplified form of F_3).
- A Boolean function may be transformed from an algebraic expression into a logic diagram composed of AND, OR and NOT gates.
- From the gate-level diagrams, the implementation of F_4 requires fewer gates and fewer inputs than F_3 — F_3 has 3 terms and 8 literals, while F_4 has 2 terms and 4 literals.
- To find simpler circuits, one must know how to manipulate Boolean functions to obtain equal and simpler expressions.

Algebraic Manipulation

- When a Boolean expression is implemented with logic gates, each term requires a gate and each variable within the term designates an input to that gate.
- A literal is defined as a single variable within a term, in complemented or uncomplemented form.
- By reducing the number of terms, the number of literals, or both, in a Boolean expression, it is often possible to obtain a simpler circuit.
- The manipulation of Boolean algebra consists mostly of reducing an expression for the purpose of obtaining a simpler circuit.
- Functions of up to 5 variables can be simplified by the map method. For more complex Boolean functions, designers use computer minimization programs capable of producing optimal circuits with millions of logic gates.

Complement of a Function

- The complement of a function F is F' and is obtained from an interchange of 0's for 1's and 1's for 0's in the value of F .
- The complement of a function may be derived algebraically through DeMorgan's theorems.
- DeMorgan's theorems can be extended to 3 or more variables. The three-variable form of the first theorem is derived as follows, from postulates and theorems:

Step	Reason
$(A+B+C)' = (A+X)'$ [let $B+C = X$]	
$= A'X'$	theorem 5(a) — DeMorgan
$= A'(B+C)'$	substitute $B+C = X$

$= A'(B'C')$	theorem 5(a) — DeMorgan
$= A'B'C'$	theorem 4(b) — associative

DeMorgan's theorems for any number of variables resemble the two-variable case in form, and can be derived by successive substitutions similar to the method above. Generalized form:

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

$$(ABCD \dots F)' = A' + B' + C' + D' + \dots + F'$$

The generalized form of DeMorgan's theorems states that the complement of a function is obtained by interchanging AND and OR operators and complementing each literal.

Example 1 — Find the complement of the functions:

$$F_1 = x'yz' + x'y'z \quad F_2 = x(y'z' + yz)$$

Step	Result
$F_1' = (x'yz' + x'y'z)'$	$= (x'yz')' \cdot (x'y'z)'$
	$= (x + y' + z)(x + y + z')$
$F_2' = [x(y'z' + yz)]'$	$= x' + (y'z' + yz)'$
	$= x' + (y'z')' \cdot (yz)'$
	$= x' + (y+z)(y'+z')$
	$= x' + yz' + y'z$

Example 2 — Find the complement by taking the dual and complementing each literal:

(a) $F_1 = x'yz' + x'y'z$. Dual of F_1 is $(x'+y+z')(x'+y'+z)$. Complementing each literal: $(x+y'+z)(x+y+z') = F_1'$

(b) $F_2 = x(y'z' + yz)$. Dual of F_2 is $x + (y'+z')(y+z)$. Complementing each literal: $x' + (y+z)(y'+z') = F_2'$

7. Digital Logic Gates

- Since Boolean functions are expressed in terms of AND, OR and NOT operations, it is easier to implement a Boolean function with these types of gates.
- Factors to be weighed when considering the construction of other types of logic gates are: (1) the feasibility and economy of producing the gate with physical components; (2) the possibility of extending the gate to more than two inputs; (3) the basic properties of the binary operator such as commutativity and associativity; (4) the ability of the gate to implement Boolean functions alone or in conjunction with other gates.
- The gates except for the inverter and buffer can be extended to have more than two inputs.

The eight logical functions used as standard gates in digital design are:

Function	Standard gate
Complement	NOT
Transfer	Buffer
Logical AND	AND
Logical OR	OR
Logical NAND	NAND
Logical NOR	NOR
Logical Ex-OR	EX-OR
Equivalence	EX-NOR

Standard gates — symbol equation and truth table:

Gate	Equation	x	y	F
AND	$F = xy$	0/0/1/1	0/1/0/1	0/0/0/1
OR	$F = x + y$	0/0/1/1	0/1/0/1	0/1/1/1
NAND	$F = (xy)'$	0/0/1/1	0/1/0/1	1/1/1/0

NOR	$F = (x+y)'$	0/0/1/1	0/1/0/1	1/0/0/0
X-OR	$F = xy' + x'y = x \oplus y$	0/0/1/1	0/1/0/1	0/1/1/0
X-NOR	$F = xy + x'y' = x \odot y$	0/0/1/1	0/1/0/1	1/0/0/1

x, y, F columns list the 4 input/output rows for $x y = 00, 01, 10, 11$ in order.

NOT (inverter): $F = x'$ ($x: 0 \rightarrow 1, 1 \rightarrow 0$). Buffer: $F = x$ (output equals input).

Three-Input Gates

- 3-input NOR gate: output = $(x + y + z)'$
- 3-input NAND gate: output = $(xyz)'$
- Cascaded NAND gates: $F = [(ABC)'] \cdot (DE)'] = ABC + DE$
- 3-input X-OR gate: $F = x \oplus y \oplus z$ (equivalent to two cascaded 2-input X-OR gates)

x	y	z	$F = x \oplus y \oplus z$
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

Example — Construct an AND gate using NOR gates.

Solution: Using DeMorgan's theorem, $(A.B)'$ complemented is $(A'+B')' = A.B$.

So A and B are each first passed through a NOR gate wired as an inverter (giving A' and B'), and the two outputs A' and B' are then fed into a third NOR gate: $Y = (A'+B')' = A.B$.

8. Simplification Examples

(1) $Y = ABC + A'BC + AB'C$

Step	Reason
$Y = BC(A+A') + AB'C$	$A+A'=1$
$= BC.1 + AB'C$	$1+A=1$
$= BC + ABC + AB'C$	
$= BC + AC(B+B')$	
$= BC + AC.1$	$B+B'=1$
$Y = C(A+B)$	final simplified form

(2) $Y = (A+B)(A'+B)$

Step	Reason
$Y = AA' + AB + A'B + BB$	
$= AB + A'B + B$	$AA'=0, BB=B$
$= B(A'+A) + B$	
$= B.1 + B$	$A+A'=1$
$Y = B$	$B+B=B$

(3) $Y = AB + A'B'$ (using DeMorgan's theorem)

Step	Reason
$Y = (AB)' + (A'B)'$ -complemented form used as:	expressed via DeMorgan
$Y = (AB)' \cdot (A'B)'$ → complement of Y	DeMorgan 1st theorem

$= (A'+B')(A+B)$	DeMorgan 2nd theorem
$= A'A + A'B + B'A + B'B$	
$Y = A'B + B'A$	$A'A=0, B'B=0$

(4) Simplify $ABC + A'BC + AB'C + ABC'$ and realize using basic gates.

Step	Reason
$Y = BC(A+A') + AB'C + ABC'$	$A+A'=1$
$= BC + AB'C + ABC'$	$1+A=1$
$= BC + AC(B+B') + ABC'$	
$= BC + AC + ABC'$	$B+B'=1$
$= BC + AC(1+B) +$	$1+B=1$ (absorbed)
$= BC + AC + ACB + ABC'$	
$= BC + AC + AB(C+C')$	
$Y = AB + BC + CA$	$C+C'=1$ (final SOP form)

Realization: this final expression $Y = AB + BC + CA$ is implemented with three 2-input AND gates (for AB, BC, CA) feeding a single 3-input OR gate.

(5) Find the complement of $AB + AC$ using DeMorgan's theorem and realize using basic gates only.

Step	Reason
$Y = (AB+AC)' = (AB)' \cdot (AC)'$	DeMorgan 1st theorem
$= (A'+B')(A'+C)$	DeMorgan 2nd theorem
$= A'A' + A'C + B'A' + B'C$	
$= A' + A'C + A'B' + B'C$	$A'A'=A'$
$= A'(1+C) + A'B' + B'C$	$1+C=1$
$= A' \cdot 1 + A'B' + B'C$	
$= A'(1+B') + B'C$	$1+B'=1$
$Y = A' + B'C$	final simplified form

Realization: A is passed through an inverter to give A' ; B and C are each inverted to give B' , C' , which are ANDed to give $B'C'$; A' and $B'C'$ are then ORed to give $Y = A' + B'C$.

9. Gate-Level Minimization

- Gate-level minimization is the design task of finding an optimal gate-level implementation of the Boolean functions describing a digital circuit.
- This task is well understood but is difficult to execute by manual methods when the logic has more than a few inputs.
- Fortunately, computer-based logic synthesis tools can minimize a large set of Boolean equations efficiently and quickly.
- Nevertheless, it is important that a designer understand the underlying mathematical description and solution of the problem.

The Map Method (Karnaugh Map)

- The complexity of the digital logic gates that implement a Boolean function is directly related to the complexity of the algebraic expression from which the function is implemented.
- Although the truth-table representation of a function is unique, when expressed algebraically it can appear in many different but equivalent forms.
- Boolean expressions may be simplified by algebraic means, but this procedure is awkward because it lacks specific rules to predict each succeeding step in the manipulative process.
- The map method provides a simple, straightforward procedure for minimizing Boolean functions. This

method may be regarded as a pictorial form of a truth table, and is also known as the Karnaugh map or K-map.

- A K-map is a diagram made up of squares, with each square representing one minterm of the function to be minimized. The map presents a visual diagram of all possible ways a function may be expressed in standard form.
- The simplified expressions produced by the map are always in one of two standard forms: (1) Sum of Products (SOP) (2) Product of Sums (POS).

Two-Variable K-Map

There are 4 minterms for two variables; hence the map consists of 4 squares, one for each minterm (m_0 , m_1 , m_2 , m_3). The 0's and 1's marked in each row and column designate the values of the variables x and y .

$x \setminus y$	0	1
0	$x'y' (m_0)$	$x'y (m_1)$
1	$xy' (m_2)$	$xy (m_3)$

Three-Variable K-Map

There are 8 minterms for three binary variables; therefore the map consists of 8 squares. The minterms are arranged not in binary sequence but in a Gray-code sequence, so that only one bit changes in value from one adjacent column to the next.

$x \setminus yz$	00	01	11	10
0	$x'y'z'$	$x'y'z$	$x'yz$	$x'yz'$
1	$xy'z'$	$xy'z$	xyz	xyz'

Key property: any two adjacent squares in the map differ by only one variable, which is primed in one square and unprimed in the other — this is the basis for simplification by grouping.

Example 1 — Simplify $F(x,y,z) = \Sigma(2,3,4,5)$

$x \setminus yz$	00	01	11	10
0	0	0	1	1
1	1	1	0	0

$$F(x,y,z) = x'yz + x'yz' + xy'z' + xy'z = x'y(z+z') + xy'(z'+z) = x'y + xy'$$

Example 2 — Simplify $F(x,y,z) = \Sigma(3,4,6,7)$

$x \setminus yz$	00	01	11	10
0	0	0	1	0
1	1	0	1	1

$$F = x'yz + xy'z' + xyz + xyz' = yz(x'+x) + xz'(y'+y) = yz + xz'$$

Pairs, Quads and Octets

By grouping together adjacent 1's in the K-map, it is possible to eliminate some of the variables and obtain directly the simplified form of a Boolean expression.

2-Variable K-map:

$A \setminus B$	0	1
0	0	1
1	0	1

$m_0=0, m_1=1, m_2=0, m_3=1 \rightarrow f(A,B) = \Sigma(1,3) = A'B + AB = B(A'+A) = B$. A variable can be eliminated.

3-Variable K-map — Example 1:

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

$$f = A'B + AB'$$

Verification: $f = A'BC + A'BC' + AB'C' + AB'C = A'B(C+C') + AB'(C+C') = A'B + AB' = A \oplus B$

3-Variable K-map — Example 2:

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

$$f = B'C + AB'$$

$$f = B'C(A+A')$$

3-Variable K-map — Example 3 (quad):

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

$$f = C$$

Verification: $f = A'BC + A'BC' \dots = B'C(A'+A) + BC(A'+A) = B'C + BC = C(B'+B) = C$

Four-Variable K-Map — examples

Note: pairs eliminate one variable, quads eliminate two variables, octets eliminate three variables.

Example 1 — Simplify $F(w,x,y,z) = \Sigma(0,1,2,4,5,6,8,9,12,13,14)$

$$F = y' + w'z' + xz'$$

(2 quads and 1 octet are used in the grouping.)

Example 2 — Simplify $F = A'B'C' + BCD' + A'BCD' + AB'C'$

$$F = B'C' + B'D + ACD'$$

(2 quads and 1 pair are used in the grouping.)

Example 3 — Simplify $F(A,B,C,D) = \Sigma(0,2,3,5,7,8,9,10,11,13,15)$

$$F = CD + AB' + BD + B'D'$$

(4 quads used in the grouping.)

Example 4 — Simplified K-map:

$$F(A,B,C) = C'D + AB'C + A'BD'$$

10. NAND and NOR Implementation

- Digital circuits are more frequently constructed with NAND or NOR gates than with AND and OR gates.
- NAND and NOR gates are easier to fabricate with electronic components and are the basic gates used in all IC digital logic families.
- Because of the prominence of NAND and NOR gates, rules and procedures have been developed for converting Boolean functions given in terms of AND, OR and NOT into equivalent NAND or NOR logic diagrams.

NAND gate can be drawn as AND-Invert [$F=(xyz)'$] or Invert-OR [$F=x'+y'+z' = (xyz)'$]. NOR gate can be drawn as OR-Invert [$F=(x+y+z)'$] or Invert-AND [$F=x'y'z' = (x+y+z)'$]. An inverter can equivalently be drawn as a

Buffer-Invert, AND-Invert or OR-Invert with a single input.

NAND Implementation

- The implementation of a Boolean function with NAND gates requires that the function be simplified into sum-of-products (SOP) form.

Example: $F = AB + CD + E$ is implemented in sum-of-products form with AND and OR gates. Replacing the AND gates with NAND gates and the output OR gate with a NAND gate (drawn with an invert-OR symbol) gives an equivalent NAND-only circuit:

$$F = [(AB)'. (CD)'. E']' \rightarrow \text{redrawn with standard NAND symbol} \rightarrow F = AB + CD + E$$

Rule for obtaining the NAND logic diagram from a Boolean function:

- (1) Simplify the function and express it in sum-of-products form.
- (2) Draw a NAND gate for each product term of the function that has at least two literals. The inputs to each NAND gate are the literals of that term — this constitutes a group of first-level gates.
- (3) Draw a single NAND gate (using the AND-invert or invert-OR symbol) in the second level, with inputs coming from the outputs of the first-level gates.
- (4) A term with a single literal requires an inverter in the first level, or may be complemented and applied directly as an input to the second-level NAND gate.

Example — Implement $F(x,y,z) = \Sigma(0,6)$ with NAND gates.

$x \setminus yz$	00	01	11	10
0	1	0	0	0
1	0	0	0	1

$$F = x'y'z' + xyz'$$

Two-level NAND implementation: this is obtained directly from the SOP terms — one first-level NAND gate for $x'y'z'$, one for xyz' , both feeding a second-level NAND gate that produces F .

NOR Implementation

- The NOR function is the dual of the NAND function. For this reason, all procedures and rules for NOR logic are the dual of the corresponding procedures and rules developed for NAND logic.
- The implementation of a Boolean function with NOR gates requires that the function be simplified into product-of-sums (POS) form.

Consider the product-of-sums expression $F = (A+B)(C+D)E$. Using NOR gates (OR-invert then a second-level NOR with an inverter for the single-literal term E): $F = [(A+B)' + (C+D)' + E']'$.

Example — Implement $F' = x'y + xy' + z$ with NOR gates.

This is the complement of the function in sum-of-products form. Complementing F' gives the simplified function in product-of-sums form, as required for NOR implementation:

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

First-level OR-invert (NOR) gates produce $(x+y)'$, $(x'+y)'$ and z' ; a second-level NOR gate combines them to give F , and its complement is also available as $F' = (x+y+z)(x'+y'+z)$.

Rules for NAND and NOR implementation:

Case	Function to simplify	Standard form to use	How to derive	Implement with	No. of levels to F
(a)	F	SOP	Combine 1's in map	NAND	2
(b)	F'	SOP	Combine 0's in map	NAND	3
(c)	F	POS	Complement F' in (b)	NOR	2
(d)	F'	POS	Complement F in (a)	NOR	3

11. K-Map for Boolean Expressions in POS Form

Example: $F(A,B,C) = \Sigma m(1,3,4,6)$. F is expressed in terms of minterms; minterm values are 1 in m_1, m_3, m_4 and m_6 . Hence in the K-map, 1's are entered in cells 1, 3, 4, 6. In all other cells (0, 2, 5, 7), 0's are entered.

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

$$F'(A,B,C) = A'C + AC$$

In order to obtain the Boolean expression for $F(A,B,C)$ in POS form, DeMorgan's theorem is applied:

$$F(A,B,C) = (F')' = (A'C + AC)' = (A'C)'. (AC)' = (A+C')(A'+C)$$

Note: if $f(A,B,C) = \Sigma m(1,3,4,6)$, then (i) $f'(A,B,C) = \Pi M(1,3,4,6)$; and (ii) $f(A,B,C) = \Pi M(0,2,5,7)$.

Example — $F(A,B,C,D) = \Pi M(3,7,10,11,15)$

$$F = CD + AB'C$$

In POS form, taking the complement of F :

$$F' = f' = (CD + AB'C)' = (CD)'. (AB'C)' = (C'+D')(A'+B+C')$$

$$f = (C'+D')(A'+B+C')$$

Example — obtain, using K-map, the minimized Boolean expression in (i) SOP form and (ii) POS form for a given truth table.

Minterms with value 1: $m_1, m_5, m_6, m_7, m_{11}, m_{12}, m_{13}, m_{15}$.

Minterms with value 0: $m_0, m_2, m_3, m_4, m_8, m_9, m_{10}, m_{14}$.

(i) SOP form:

$$F = A'CD + A'BC + ABC' + ACD$$

(ii) POS form:

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

12. Additional Simplification Examples

Simplify $Y = (AB + A'B + B')'$ using DeMorgan's theorems:

Step	Reason
$Y = (AB)' \cdot (A'B)' \cdot (B)'$	DeMorgan 1st theorem
$= (A'+B')(A+B')(B)$	DeMorgan 2nd theorem on each factor
$= (A'+B')(AB + BB')$	$(A+B')B = AB + BB'$
$= (A'+B')(AB)$	$BB'=0$
$= A'AB + B'AB$	
$Y = 0 + 0 = 0$	$A'A=0$

Simplify $AB + A'C + B'C$:

Step	Reason
$Y = AB + C(A'+B')$	
$= AB + C(AB)'$	DeMorgan 2nd theorem
$= AB.1 + AB' C \dots = AB(1+C) + A'B'C$	$1+C=1$
$= AB + ABC + A'B'C$	
$= AB + C(AB + A'B')$	

$Y = AB + C$ (since $AB + A'B' = 1$ is not generally true — result follows directly from absorption: $AB + C(AB)' = AB + C$).

Simplify $Y = (A'+B)(A'+C)(B+C)$:

Step	Reason
$Y = (A'B)' \cdot (C'D)'$	DeMorgan 1st theorem
$Y = (A+B)(C+D)$	DeMorgan 2nd theorem, applied to each factor
$Y = (A+B')(C+D')$	final form — fewer gates

Step	Reason
$Y = (A'.B)(A'+C)(B+C)$	
$= A'.B'[A'B' + A'C + B'C + CC']$	
$= A'B' + A'B'C + A'B'C' + 0$	$A'A=A', CC'=0$
$= A'B'(1+C) + A'B'C'$	$1+C=1$
$= A'B'.1 + A'B'C'$	
$= A'B'(1+C')$	$1+C'=1$
$Y = A'B'$	final simplified form

Realize the following expressions using basic gates after simplification (i) $Y = (A'B + C'D)'$:

the expanded SOP form $Y = AC + AD' + B'C + B'D'$ requires more logic gates, so the factored form $(A+B')(C+D')$ is preferred for the circuit realization (two OR gates feeding a single AND gate, with inverters on B and D).

(ii) $Y = [(A+B) \cdot C \cdot D']'$:

$$Y = (A+B)' + C' + (D')' = A'B' + C' + D$$

Realized using inverters on A, B, C and D; an AND gate combining A' and B'; and a 3-input OR gate combining (A'.B'), C' and D to produce the final output Y.

13. Prime Implicants

- In choosing adjacent squares in a map, we must ensure that: (1) all the minterms of the function are covered when we combine the squares; (2) the number of terms in the expression is minimized; (3) there are no redundant terms (i.e. minterms already covered by other terms).
- Sometimes there may be two or more expressions that satisfy the simplification criteria. The procedure for combining squares in the map may be made more systematic if we understand the meaning of two special types of terms.
- A prime implicant is a product term obtained by combining the maximum possible number of adjacent squares in the map.
- If a minterm in a square is covered by only one prime implicant, that prime implicant is said to be essential.
- The prime implicants of a function can be obtained from the map by combining all possible maximum numbers of squares: a single 1 on a map represents a prime implicant if it is not adjacent to any other 1's; two adjacent 1's form a prime implicant provided they are not within a group of 4 adjacent squares; four adjacent 1's form a prime implicant if they are not within a group of 8 adjacent squares, and so on.

Example — $F(A,B,C,D) = \Sigma(0,2,3,5,7,8,9,10,11,13,15)$

A \ CD	00	01	11	10
00	1	0	1	1
01	0	1	1	0
11	0	1	1	0
10	1	1	1	1

All groupings of 1's are implicants: CD, AB', AD, B'C, BD, and AB'D.

Selecting the maximal (prime) groupings and eliminating redundant terms yields the minimized expression using the prime implicants of the map (as derived in the earlier POS/SOP examples for this same function): $F = CD + AB' + BD + B'D'$.