freebooks.pk

Computer Science Class 9 — Punjab Board (English Medium)

Chapter 3

Digital Systems and Logic Design

Boolean Algebra • Logic Gates • Half & Full Adders • Karnaugh Maps

Free Notes for Class 9 Computer Science Students

Chapter 3: Digital Systems and Logic Design

This chapter introduces digital systems and the branch of mathematics that underlies them: Boolean algebra. It begins by distinguishing analog signals (continuous, infinite values) from digital signals (discrete, only 0 or 1), and explains how Analog-to-Digital (ADC) and Digital-to-Analog (DAC) conversion allow real-world signals like sound to be processed by digital devices.

The chapter then covers the fundamental Boolean operations — AND, OR, and NOT — and the logic gates (AND, OR, NOT, NAND, XOR) that implement them in electronic circuits, along with truth tables that describe their behaviour. It introduces Boolean algebra's simplification laws and De Morgan's theorems, then applies these ideas to two important digital circuits, the half-adder and full-adder, and finally to Karnaugh maps (K-maps), a graphical technique for simplifying Boolean expressions.

Learning Objectives

- Differentiate between analog and digital signals, and explain ADC and DAC conversion
- Explain the fundamentals of digital logic and how binary values are represented as voltage levels
- Construct and evaluate Boolean functions and expressions using AND, OR, and NOT operations
- Build truth tables for logical expressions and Boolean functions
- Identify logic gates (AND, OR, NOT, NAND, XOR) and describe their functions
- Apply Boolean algebra laws and De Morgan's theorems to simplify Boolean expressions
- Describe the operation of half-adder and full-adder circuits using truth tables and Boolean expressions
- Use Karnaugh maps (K-maps) to minimize Boolean expressions

Key Concepts

3.1 Analog and Digital Signals

An analog signal changes smoothly and continuously over time and can take any value within a given range — examples include voice signals, body temperature, and radio waves. A digital signal, by contrast, has only two discrete values, represented as '0' and '1', and is used throughout digital electronics and computing systems.

Analog-to-Digital Conversion (ADC) converts continuous analog signals into discrete digital signals that can be processed by computers and smartphones; Digital-to-Analog Conversion (DAC) converts digital signals back into analog form so that humans can perceive the information, such as through speakers. For example, a microphone uses ADC to convert your voice into digital data for transmission, while a speaker at the receiving end uses DAC to convert that digital data back into sound waves.

3.2 Fundamentals of Digital Logic

Digital logic is the basis of digital systems, using binary values (0 and 1) to represent and manipulate information. In digital circuits, these two states are represented by different voltage levels — conventionally, a higher voltage (e.g., 5 volts) represents binary '1', while a low voltage (e.g., 0 volts) represents binary '0'. These voltage levels, called logic levels, allow digital circuits to switch devices on and off and to process information.

3.3 Boolean Functions: AND, OR and NOT Operations

Boolean algebra is a branch of mathematics dealing with logic and symbolic computation using two values, True and False. The AND operation requires two binary inputs and produces '1' only when both inputs are '1' (written A·B); otherwise the output is '0'. The OR operation produces '1' when at least one input is '1' (written A+B), and '0' only when both inputs are '0'. The NOT operation takes a single input and negates it (written $\bar{A}$ or ¬A): if the input is 1, the output is 0, and vice versa.

A Boolean function has one or more binary inputs and produces a single binary output, constructed by combining AND, OR, and NOT operations. For example, F(A,B,C) = A·B + Ā·C combines all three operations; its truth table is built by evaluating the function for every possible combination of A, B, and C. Boolean functions underpin the Arithmetic Logic Units (ALUs) of CPUs, data processing in memory and storage, and control logic throughout computers, phones, and calculators.

3.4 Logic Gates and their Functions

Logic gates are physical devices in electronic circuits that implement Boolean operations. The AND gate outputs true only when both inputs are true. The OR gate outputs true when at least one input is true. The NOT gate outputs the opposite of its single input. The NAND gate is an AND gate combined with a NOT gate — it is the inverse of AND, outputting true when at least one input is false.

The XOR (Exclusive OR) gate outputs true only when exactly one of its two inputs is true — unlike the OR gate, it outputs false when both inputs are true. For example, in a scenario where you can either play video games OR do homework but not both, XOR models this 'one or the other, not both' logic.

3.5 Boolean Algebra Laws and Simplification

Boolean expressions can be simplified using standard algebraic laws: Identity Laws (A+0=A, A·1=A), Null Laws (A+1=1, A·0=0), Idempotent Laws (A+A=A, A·A=A), Complement Laws (A+Ā=1, A·Ā=0), Commutative Laws (A+B=B+A, A·B=B·A), Associative Laws, Distributive Laws (A·(B+C)=A·B+A·C), and Absorption Laws (A+(A·B)=A).

De Morgan's Theorems state that the complement of a sum equals the product of the complements (Ā+B̄ = complement of A·B), and the complement of a product equals the sum of the complements (Ā·B̄ = complement of A+B). Simplifying Boolean expressions reduces the number of gates a circuit needs, making it faster, cheaper, and more energy-efficient.

3.6 Half-Adder and Full-Adder Circuits

A half-adder is a basic circuit that adds two single-bit binary digits, with two inputs (A, B) and two outputs: Sum (S) and Carry (C). Its Boolean expressions are S = A⊕B (XOR) and C = A·B (AND) — the sum is high when only one input is high, while the carry is high only when both inputs are high.

A full-adder is a more complex circuit that adds three single-bit values: two data bits plus a carry-in bit (Cin) from a previous addition. It has three inputs (A, B, Cin) and two outputs: Sum and Carry-out (Cout). Its Boolean expressions are Sum = A⊕B⊕Cin and Carry = (A·B) + (Cin·(A⊕B)) — the sum is high when an odd number of inputs are high, while the carry is high when at least two inputs are high. Chaining full-adders together allows computers to add multi-bit binary numbers.

3.7 Karnaugh Maps (K-Maps)

A Karnaugh map (K-map) is a graphical method for simplifying Boolean expressions without lengthy algebraic manipulation, by plotting the truth values of a function in a grid so that patterns can be visually identified and combined. The size of the grid depends on the number of variables: a 2-variable K-map uses a 2×2 grid, a 3-variable K-map uses a 2×4 grid, and a 4-variable K-map uses a 4×4 grid.

To build a K-map: create a grid sized for the number of variables, fill each cell with the output value (0 or 1) from the truth table, then group adjacent 1s into the largest possible groups (sized as a power of 2: 1, 2, 4, 8...). Each group of 1s corresponds to a simplified term. For example, simplifying $A\cdot\bar{B} + \bar{A}\cdot B + A\cdot B$ with a 2-variable K-map identifies two groups that reduce to the much simpler final expression F(A,B) = A + B.

3.8 Minterms and Logic Diagrams

A minterm is a product term in which every variable of a Boolean function appears exactly once, in either its true or complemented form; each minterm corresponds to exactly one combination of variable values that makes the function equal to 1. For a 3-variable function A, B, C, the minterm where A=1, B=0, C=1 is written as $A\cdot\bar{B}\cdot C$.

A logic diagram shows the physical arrangement of logic gates needed to implement a Boolean function, with each gate's inputs and outputs correctly connected. To build one: identify the gates needed for the function, arrange them according to the function's structure, then connect the inputs and outputs correctly — this is the final step that translates Boolean algebra into an actual, buildable digital circuit.

Important Definitions

What is a digital signal?

A signal that has only two discrete values, represented as '0' and '1', used throughout digital electronics and computing systems.

What is an analog signal?

A signal that changes smoothly and continuously over time and can take any value within a given range, such as sound waves or temperature.

What is Boolean algebra?

A branch of mathematics dealing with logic and symbolic computation, using two values (True/False or 1/0), forming the basis for digital circuit analysis and design.

What is a logic gate?

A physical device in an electronic circuit that implements a basic Boolean operation, such as AND, OR, NOT, NAND, or XOR.

What is a truth table?

A table that lists every possible combination of input values for a Boolean expression or logic gate, along with the corresponding output for each combination.

What is a half-adder?

A basic digital circuit that adds two single-bit binary digits, producing a Sum output ($A \oplus B$) and a Carry output ($A \cdot B$).

What is a full-adder?

A digital circuit that adds three single-bit values — two data bits and a carry-in bit — producing a Sum output and a Carry-out output, used to build multi-bit binary adders.

What is a Karnaugh map (K-map)?

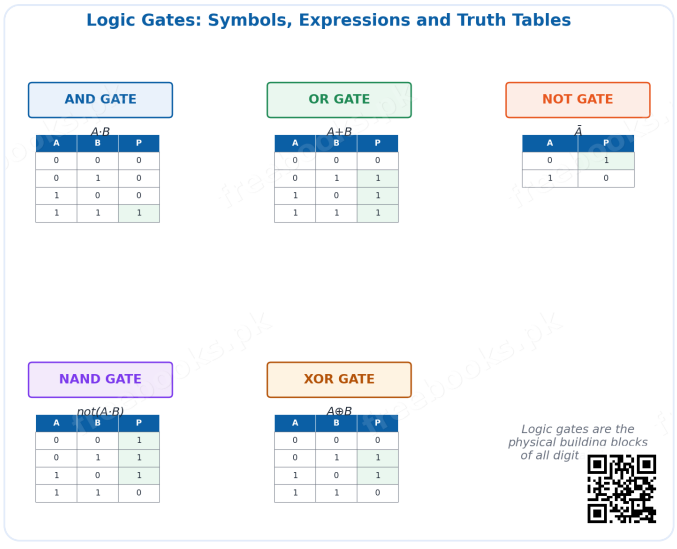
A graphical grid-based method for simplifying Boolean expressions by plotting truth-table values and grouping adjacent 1s into the largest possible groups.

Key Facts and Relations

Topic	Key Fact / Relation
AND operation	A · B → 1 only if both A=1 and B=1
OR operation	A + B → 0 only if both A=0 and B=0
NOT operation	Ā (or ¬A) → inverts the single input
XOR operation	A ⊕ B → 1 only if exactly one input is 1
De Morgan's Theorems	Ā+B̄ = complement of (A·B); Ā·B̄ = complement of (A+B)
Half-adder	Sum (S) = A ⊕ B; Carry (C) = A · B
Full-adder	Sum = A ⊕ B ⊕ Cin; Carry = (A·B) + (Cin·(A⊕B))
K-map grid size	2 variables = 2×2; 3 variables = 2×4; 4 variables = 4×4

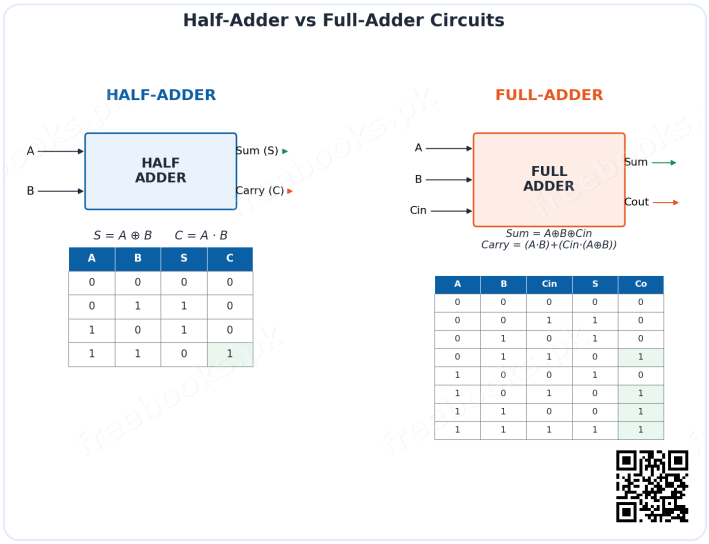
Diagrams

Logic Gates: Symbols and Truth Tables: A comparison of the AND, OR, NOT, NAND, and XOR logic gates, showing their standard symbols and truth tables side by side



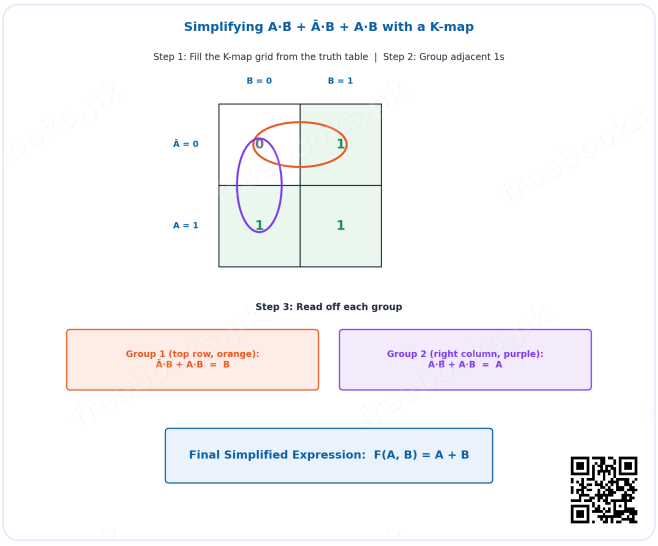
Logic Gates: Symbols and Truth Tables

Half-Adder and Full-Adder Circuits: Block diagrams of the half-adder (2 inputs, Sum/Carry outputs) and full-adder (3 inputs, Sum/Carry-out outputs) with their truth tables and Boolean expressions



Half-Adder and Full-Adder Circuits

Simplifying a Boolean Expression with a K-map: A worked 2-variable Karnaugh map example showing how A·B̄ + Ā·B + A·B is grouped and simplified to F(A,B) = A + B



Simplifying a Boolean Expression with a K-map

Short Questions & Answers

Define a Boolean function and give an example.

A Boolean function is a function with one or more binary inputs that produces a single binary output, built using AND, OR, and NOT operations; for example, F(A,B) = A·B represents the AND operation between A and B.

What is the significance of the truth table in digital logic?

A truth table lists every possible combination of input values along with the corresponding output, making it possible to fully understand, verify, and communicate the behaviour of a Boolean expression or logic circuit.

Explain the difference between analog and digital signals.

An analog signal changes continuously and can take any value within a range (like sound waves), while a digital signal has only discrete values, 0 or 1, making it more resistant to noise and easier for computers to process.

Describe the function of a NOT gate with its truth table.

A NOT gate takes a single binary input and outputs its opposite: when the input is 0, the output is 1, and when the input is 1, the output is 0. Its truth table has two rows: A=0→P=1, A=1→P=0.

What is the purpose of a Karnaugh map in simplifying Boolean expressions?

A K-map provides a visual, grid-based way to identify and group adjacent 1s in a truth table, allowing a Boolean expression to be simplified into a shorter, equivalent form without lengthy algebraic manipulation.

What is the difference between a half-adder and a full-adder?

A half-adder adds only two single-bit inputs (A, B) and has no carry-in, while a full-adder adds three inputs (A, B, and a carry-in bit Cin), allowing full-adders to be chained together to add multi-bit binary numbers.

What is a minterm in Boolean algebra?

A minterm is a product term in which every variable of a Boolean function appears exactly once, in either its true or complemented form, corresponding to exactly one input combination that makes the function equal to 1.

Why are digital signals preferred over analog signals for data transmission and storage?

Digital signals are much less affected by noise and signal degradation than analog signals, making them better suited for reliably transmitting and storing information over long distances.

Long Questions & Answers

Explain the usage of Boolean functions in computers, with examples of where they are applied.

Boolean functions, which are mathematical expressions built from binary variables combined using the fundamental logical operations of AND, OR, and NOT, form an absolutely essential foundation underlying the operation of virtually every modern computing device, from the simplest pocket calculator to the most powerful supercomputer, precisely because every piece of information inside a digital computer is ultimately represented using nothing more than binary values of 0 and 1, and Boolean functions provide the exact mathematical framework needed to meaningfully combine, manipulate, and process these binary values in order to produce useful results. One of the most fundamental and widespread applications of Boolean functions within computers is found in arithmetic operations, where the Arithmetic Logic Unit (ALU) inside a computer's CPU relies extensively on carefully designed Boolean functions, implemented directly as physical logic-gate circuits, in order to perform essential operations such as addition, subtraction, multiplication, and division on binary numbers — for instance, as covered elsewhere in this chapter, half-adder and full-adder circuits, which are themselves built entirely from Boolean AND, OR, and XOR gates, form the core building blocks that allow a computer's ALU to add together binary numbers of any length. Beyond pure arithmetic, Boolean functions are likewise essential for data processing more broadly, since binary data stored in a computer's memory and storage devices must constantly be manipulated, filtered, compared, and retrieved, and Boolean logic provides the precise mechanism by which a computer can efficiently make these kinds of binary decisions and comparisons at extremely high speed. Boolean functions additionally play a crucial role in what is known as control logic, where they are used extensively throughout a computer system to coordinate and control the various different parts of the system, ensuring that different hardware and software components operate together in the correct, carefully synchronized

sequence rather than working against one another. Even outside of computers in the strictest sense, Boolean functions are present in a great many everyday digital devices that people interact with constantly, such as cell phones and calculators: in a cell phone, for example, every single button press or touchscreen tap that a user makes is ultimately evaluated internally by Boolean functions that determine whether specific conditions are true or false, and use this evaluation to decide on and generate the correct corresponding output or action; similarly, when a person enters numbers and mathematical operations into a basic calculator, the device relies internally on Boolean logic circuits in order to correctly process this binary input and arrive at the correct final numerical result to display back to the user.

Describe how to construct a truth table for a Boolean expression, using F(A, B, C) = A·B + Ā·C as a worked example.

Constructing a truth table for any given Boolean expression is a systematic, entirely mechanical process that involves working through every single possible combination of the input variables involved in that expression, one at a time, and carefully calculating the corresponding output value that the expression would produce for each individual combination, ultimately organizing all of these calculated results together into a single, clearly structured table. The first essential step in this process is to identify precisely how many distinct binary input variables the given Boolean expression actually contains, since this number directly determines exactly how many separate rows the resulting truth table will need to contain in total — specifically, for any Boolean expression containing n distinct binary variables, the complete truth table will always require exactly 2^n separate rows, since this is the total number of distinct combinations possible when each of the n variables can independently take on one of exactly two possible binary values (0 or 1). Applying this general principle directly to the specific example function given, F(A, B, C) = A·B + Ā·C, we can see that this particular expression contains exactly three distinct binary input variables — A, B, and C — meaning that its complete truth table will necessarily require exactly $2^3 = 8$ separate rows in total, one row for every one of the eight possible combinations of values that these three variables can take on together. The next step is to systematically list out every one of these eight possible input combinations in a clear, consistently ordered sequence, which is most commonly

done by counting upward in standard binary order from 000 through to 111 (that is: 0,0,0 then 0,0,1 then 0,1,0 then 0,1,1 then 1,0,0 then 1,0,1 then 1,1,0 and finally 1,1,1), ensuring that every single one of the eight possible input combinations is fully accounted for and none are accidentally omitted or duplicated. For each of these eight individual input combinations, it is often extremely helpful, particularly with a somewhat more complex expression like this specific example, to additionally calculate and separately record any relevant intermediate values as their own individual helper columns within the table, before attempting to calculate the final overall output value — specifically, for this particular expression, it is very useful to calculate and record the intermediate value of the sub-expression A·B in one dedicated column, the intermediate value of $\bar{A}$ (meaning simply the logical negation, or the direct opposite, of whatever value A currently holds) in a second dedicated column, and then the intermediate value of the second sub-expression $\bar{A}$·C in a third dedicated column, since breaking the overall calculation down step-by-step in this way substantially reduces the chance of making a careless calculation error along the way. Only once every one of these necessary intermediate helper values has been correctly calculated and recorded for a given row can the final overall output column, F(A, B, C), then finally be calculated for that specific row, simply by taking the two previously calculated intermediate values found in the A·B column and the $\bar{A}$·C column, and then correctly combining these two values together using the final OR operation specified in the original expression; repeating this same complete process very carefully and systematically for every single one of the eight rows in turn ultimately produces a fully completed truth table that correctly and completely describes the exact behaviour of the original Boolean function $F(A, B, C) = A \cdot B + \bar{A} \cdot C$ across absolutely every single one of its possible input combinations.

Compare and contrast half-adders and full-adders, including their truth tables, Boolean expressions, and typical circuit diagrams.

Half-adders and full-adders are both fundamental digital circuits specifically designed to perform binary addition, and while they share this same core underlying purpose, they differ from one another in several genuinely important and consequential ways that directly determine how and where each one can actually be practically used within a larger digital computing system. A half-adder

is the comparatively simpler of the two circuits, designed specifically to add together exactly two individual single-bit binary digits at a time, commonly labelled A and B; it has exactly two binary inputs (A and B) and produces exactly two binary outputs, specifically a Sum output, labelled S, and a Carry output, labelled C. The complete truth table describing a half-adder's behaviour therefore contains exactly four distinct rows, covering all four possible combinations of its two binary inputs: when A=0 and B=0, both the Sum and Carry outputs are 0; when A=0 and B=1, the Sum output becomes 1 while the Carry output remains 0; when A=1 and B=0, the Sum output is likewise 1 while the Carry remains 0; and finally, when A=1 and B=1, the Sum output becomes 0 while the Carry output becomes 1, correctly reflecting the fact that 1+1 in binary equals 10 (that is, a sum of 0 with a carry of 1). These specific input-output relationships can be captured algebraically using two corresponding Boolean expressions: the Sum output is calculated as $S = A \oplus B$, using the XOR (Exclusive OR) operation, while the Carry output is calculated as $C = A \cdot B$, using the simple AND operation. A full-adder, by clear contrast, is meaningfully more complex than a half-adder, since it is specifically designed to add together three individual single-bit binary values simultaneously, rather than only two: it takes as its inputs not only the two primary data bits, A and B, but additionally a third input bit, commonly labelled Cin (short for 'carry-in'), which specifically represents any carry value that may have already been generated and produced by a separate, preceding addition step elsewhere in a larger multi-bit addition calculation. Because a full-adder has one additional input compared to a half-adder, its complete truth table correspondingly requires exactly eight distinct rows in total, rather than only four, in order to fully cover every single one of the possible combinations of its three separate binary inputs. A full-adder's behaviour can likewise be captured using two corresponding Boolean expressions, though these are noticeably more complex than a half-adder's expressions: its Sum output is calculated as $Sum = A \oplus B \oplus Cin$, while its Carry-out output (commonly labelled Cout) is calculated using the more complex expression $Carry = (A \cdot B) + (Cin \cdot (A \oplus B))$. In terms of their typical circuit diagrams, a half-adder can be constructed using just a single XOR gate (to directly produce the Sum output) together with a single AND gate (to directly produce the Carry output), whereas a full-adder's more complex circuit diagram is instead most commonly and efficiently constructed by combining together two separate half-adder circuits along with one additional OR gate,

which correctly combines the two individual carry outputs generated by these two internal half-adders into the full-adder's single final Carry-out output. Crucially, and most practically significantly of all, it is precisely this additional carry-in input that a full-adder possesses, but which a half-adder critically lacks, that specifically allows multiple individual full-adder circuits to be chained directly together in a sequential row, with the carry-out output produced by one full-adder stage being directly connected onward to serve as the carry-in input for the very next full-adder stage in the sequence — and it is precisely this specific chaining capability that ultimately allows a computer to correctly and reliably add together binary numbers of essentially any arbitrary length, well beyond the single-bit limitation that a lone half-adder circuit would otherwise impose.

Simplify the Boolean function F(A, B) = A·B + A·$\bar{B}$ using Boolean algebra rules, showing each step and the law applied. Also simplify F(A,B,C) = $\bar{A}$ + $\bar{B}$ + AC using De Morgan's laws.

Simplifying the first given Boolean function, F(A, B) = A·B + A·$\bar{B}$, can be accomplished through a short, clearly justified sequence of individual algebraic steps, each one relying on a specific, well-established Boolean algebra law. The starting expression is A·B + A·$\bar{B}$. As the very first step, we can identify that the variable A is a common factor present in both of the two terms being added together in this expression, and applying the Distributive Law (which states that X·Y + X·Z = X·(Y+Z)) in the reverse, 'factoring' direction, allows this common factor A to be pulled outside of a single set of parentheses, transforming the original expression into the equivalent factored form A·(B + $\bar{B}$). As the second and final step, we can now recognize that the specific sub-expression contained within these parentheses, B + $\bar{B}$, exactly matches the standard form covered by the Complement Law (which states that X + $\bar{X}$ = 1 for any variable X), meaning that B + $\bar{B}$ can therefore be directly and immediately simplified to the constant value 1; substituting this simplified value of 1 directly back into our partially-simplified expression transforms A·(B + $\bar{B}$) into A·1, and then applying the Identity Law (which states that X·1 = X for any variable X) one final time allows this expression to be simplified down to its final, fully simplified form of simply A. Therefore, the fully simplified form of the original function is F(A, B) = A. As a second example, simplifying F(A,B,C) = $\bar{A}$ + $\bar{B}$ + AC using De Morgan's laws:

applying De Morgan's Theorem to the first two terms, $\bar{A} + \bar{B}$ is equivalent to the complement of (A·B), i.e. $\bar{A} + \bar{B}$ = complement(A·B); the expression therefore becomes complement(A·B) + AC. This form shows how a sum-of-complements can always be rewritten as the complement of a product using De Morgan's theorem, which is a standard technique used to convert a Boolean expression into a form using only NAND-style logic (complement-of-AND), which is useful because NAND gates are among the cheapest and most common gates to manufacture, so digital circuit designers frequently use De Morgan's laws specifically to redesign a circuit so that it can be built using only NAND gates, reducing the number of different gate types a manufacturer needs to stock and assemble.

Multiple Choice Questions (MCQs)

Which of the following Boolean expressions represents the OR operation? (A) A · B (B) A + B (C) $\bar{A}$ (D) A ⊕ B

Correct answer: (B) A + B. The OR operation is represented using the '+' symbol: A + B, producing 1 when at least one input is 1.

What is the dual of the Boolean expression A · 0 = 0? (A) A + 1 = 1 (B) A + 0 = A (C) A · 1 = A (D) A · 0 = 0

Correct answer: (A) A + 1 = 1. The dual of an expression is found by swapping AND with OR and swapping 0 with 1; the dual of A · 0 = 0 is A + 1 = 1.

Which logic gate outputs true only if both inputs are true? (A) OR gate (B) AND gate (C) XOR gate (D) NOT gate

Correct answer: (B) AND gate. The AND gate outputs true (1) only when both of its inputs are true (1); otherwise it outputs false (0).

In a half-adder circuit, the carry output is generated by which operation? (A) XOR operation (B) AND operation (C) OR operation (D) NOT operation

Correct answer: (B) AND operation. In a half-adder, the Carry output is generated by the AND operation: C = A·B, which is 1 only when both A and B are 1.

What is the decimal equivalent of the binary number 1101? (A) 11 (B) 12 (C) 13 (D) 14

Correct answer: (C) 13. $1101_2 = 1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0 = 8+4+0+1 = 13$.

Which gate is the inverse of the AND gate? (A) OR gate (B) NOR gate (C) NAND gate (D) XOR gate

Correct answer: (C) NAND gate. The NAND gate is an AND gate combined with a NOT gate, producing the exact inverse output of a plain AND gate.

What is the output of the XOR gate when both inputs are 1? (A) 0 (B) 1 (C) Undefined (D) Depends on the circuit

Correct answer: (A) 0. The XOR gate outputs true only when exactly one input is true; when both inputs are 1, XOR outputs 0.

A K-map for a Boolean function with 3 variables uses which grid size? (A) 2×2 (B) 2×4 (C) 4×4 (D) 4×8

Correct answer: (B) 2×4. A 3-variable Karnaugh map uses a 2×4 grid, covering all $2^3 = 8$ possible input combinations.

Which of the following is an example of Analog-to-Digital Conversion (ADC)? (A) A speaker converting digital audio to sound (B) A microphone converting voice into digital data (C) A printer printing a document (D) A monitor displaying an image

Correct answer: (B) A microphone converting voice into digital data. ADC converts a continuous analog signal (like voice captured by a microphone) into discrete digital data; this is the opposite of DAC.

In a full-adder circuit, how many binary inputs are there in total? (A) 1 (B) 2 (C) 3 (D) 4

Correct answer: (C) 3. A full-adder has three binary inputs: A, B, and a carry-in bit (Cin) from a previous addition stage.

Quick Revision Summary

- Analog = continuous, infinite values; Digital = discrete, only 0 or 1; ADC converts analog→digital, DAC converts digital→analog
- AND ($A \cdot B$) = 1 only if both are 1; OR ($A+B$) = 0 only if both are 0; NOT ($\bar{A}$) = inverts the input; XOR ($A \oplus B$) = 1 only if exactly one is 1
- NAND = inverse of AND; logic gates are physical implementations of Boolean operations
- Key Boolean laws: Identity, Null, Idempotent, Complement, Commutative, Associative, Distributive, Absorption, De Morgan's

- De Morgan's: complement of (A+B) = $\bar{A}\cdot\bar{B}$; complement of (A·B) = $\bar{A}+\bar{B}$
- Half-adder: S=A⊕B, C=A·B (2 inputs, no carry-in); Full-adder: Sum=A⊕B⊕Cin, Carry=(A·B)+(Cin·(A⊕B)) (3 inputs, has carry-in)
- K-map grid size: 2 variables=2×2, 3 variables=2×4, 4 variables=4×4; group adjacent 1s in powers of 2 (1,2,4,8...) to simplify
- A minterm includes every variable of a function, in true or complemented form, corresponding to exactly one row where the function = 1

Exam Tips

- Memorize the four basic truth tables (AND, OR, NOT, XOR) — they are the foundation for every other question in this chapter
- For duality questions: swap every AND↔OR and every 0↔1 in the original expression
- Half-adder has NO carry-in; Full-adder HAS a carry-in (Cin) — this is the single most tested distinction in this chapter
- When simplifying Boolean expressions, always name the law used at each step (Distributive, Complement, Identity, etc.) — this is usually required for full marks
- For K-map questions, remember: group size must always be a power of 2 (1, 2, 4, 8...), and groups must be adjacent (including wrap-around edges)
- Practice building truth tables methodically: for n variables, there are always exactly 2^n rows, listed in binary counting order