Computers, Algorithms, and Boolean Logic

Algorithms, Programming, and Data Representation · Module 92313 · Chapter 2

From Fundamentals to AI Apocalypse · personal study guide

Contents

Start from memory 2

How to use this guide 2

How the chapter fits together 3

2.1 Algorithms are finite, executable procedures 5

2.2 A stored-program computer executes instructions 8

2.3 Programs have syntax, static meaning, and behaviour 10

2.4 Propositions turn logic into two-valued inputs 12

2.5 Boolean algebra and truth tables make the rules checkable 13

2.6 Boolean laws simplify expressions 16

2.7 Logic gates implement Boolean functions 18

2.8 Physical circuits take time to switch 21

Formula map and symbols 22

Derive it cold 22

Explain it 22

Mixed practice 23

Exam-style problems 24

Slips in the notes 25

Self-check 26

Answers 27

Made by Thanate Amornjiraporn (Focus)

This lecture connects a recipe for solving a problem to a machine that executes instructions, then shows how Boolean logic can be represented as algebra and circuits. The numbered sections below group the slides by topic because the supplied deck has slide numbers but no numbered topic sections.

Start from memory

Try these before reading. Write a concrete answer first; a blank or uncertain answer is useful feedback.

1. What makes a procedure an algorithm rather than an informal idea?
2. What is the difference between syntax and semantics?
3. How many input rows are needed for a truth table with three variables?
4. When is A ⇒ B false?
5. What does it mean for NAND to be functionally complete?

Answers: see "Start from memory" under Answers at the end.

How to use this guide

Read one short topic, hide it, and reconstruct its definition, rule, or diagram from memory. Attempt the mixed problems before opening their worked answers.

1. **Day 0:** Do the cold start, study §§2.1–2.3, explain the algorithm and stored-program ideas without looking, and solve practice 1, 3, 5, 8.
2. **Day 1:** Rebuild the propositional logic rules in §2.4 and solve practice 2, 12, 17, 22, 23.
3. **Day 3:** Work §§2.5–2.6 from memory; solve practice 4, 6, 7, 9, 11, 14, 15, 18, 20, 24, 25 and check every truth-table row.
4. **About day 7:** Reconstruct the gate circuits in §2.7, study §2.8, then solve practice 10, 13, 16, 19, 21.
5. **About day 14:** Re-derive the NAND circuit and take one exam-style problem under a timer.
6. **Before the exam:** Complete the self-check and both exam problems. Revisit each section named in a missed answer.

When stuck: attempt one small input by hand, name the rule you are applying, then check the result against a table or the definitions. If the prerequisites are missing, study one worked example and solve a fresh one cold.

Made by Thanate Angoriiraporn (Focus)

How the chapter fits together

The route begins with precise procedures and machine execution, then moves from statements to Boolean expressions and finally to gates that implement those expressions.

Measured · describe what happens

- **Algorithm** §2.1
  input → finite steps → output
  Rests on: instructions that are defined, executable, and able to halt.
- **Stored-program machine** §2.2
  fetch · execute · test · stop
  Rests on: instructions stored in memory and executed by the machine.
- **Program language** §2.3
  syntax · static semantics · meaning
  Rests on: legal expressions that a compiler or interpreter can evaluate.

Defined · set the rules

- **Propositions** §2.4
  true = 1 · false = 0
  Rests on: statements with exactly one truth value.
- **Boolean expressions** §2.5
  $\neg \cdot \wedge \cdot \vee$
  Rests on: defined operations over the two truth values.
- **Truth tables** §2.5
  n variables → 2^n rows
  Rests on: checking every possible input assignment.

Derived · simplify and build

- **Boolean laws** §2.6
  $(A + B)' = A' \cdot B'$
  Rests on: equal outputs for every assignment.
- **Logic gates** §2.7
  NAND alone is complete
  Rests on: gate outputs implementing Boolean functions.
- **Physical circuits** §2.8
  switching takes time
  Rests on: real components that need time to change state.

Made by Thanate Amornjirapoln (Focus)

Question	Tool
Does a procedure meet the algorithm definition?	Check definition, executability, finite resources, and halting.
What does a Boolean expression produce?	Evaluate it row by row in a truth table.
Can a circuit use only NAND gates?	Rewrite NOT, AND, and OR using NAND and Boolean laws (§2.7).

Made by Thanate Amornjiraporn (Focus)

2.1 Algorithms are finite, executable procedures

An algorithm turns a problem into steps that a person or machine can carry out. A precise procedure also makes it possible to decide whether it finishes and what output it should produce.

> **In plain words:** an algorithm is a recipe precise enough for a machine to follow: it says what goes in, what to do at each step, and what comes out. It must also stop. “Keep improving the guess” only becomes an algorithm once it says when the guess is good enough.

Made by Thanate Amornjiraporn (Focus)

From declarative facts to imperative steps

Declarative knowledge states facts. **Imperative knowledge** describes how to do something: a sequence of actions with a flow of control and a stopping condition. An algorithm is a finite, well-defined sequence of executable instructions for a class of problems or a computation.

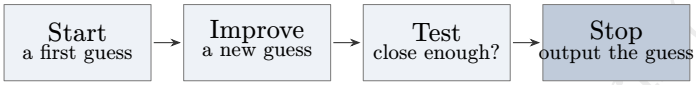
Algorithm checklist

$$\text{input} \rightarrow \text{defined instructions} \rightarrow \text{output}$$

$$\text{finite text} \cdot \text{executable steps} \cdot \text{finite memory} \cdot \text{finite steps}$$

Read it as: "input goes through clearly defined instructions to output", and the four checks: the text is finite, each step can be carried out, memory use stays finite, and it stops after finitely many steps.

The slides call finite memory use dynamic finiteness and a finite number of steps halting. A procedure that never stops does not deliver its required result.



An iteration as an algorithm: improve the guess, test it, and repeat until the test passes; then output the guess.

The slides give a tiny example: even "1 + 2 + 3 =" is an algorithm, because it fixes the steps, their order and when you are finished.

An algorithm is more specific than "find a square root": it says how to make and check successive guesses. The procedure in the notes starts with a guess g; if g^2 is close enough to x, it stops. Otherwise it replaces g with the average of g and x/g.

Worked example · follow the square-root recipe

For $x = 16$ and initial guess $g = 3$, calculate two updates.

1. First compare $3^2 = 9$ with 16, so continue. The next guess is $(3 + 16/3)/2 \approx 4.167$.
2. Now $4.167^2 \approx 17.36$, so update again: $(4.167 + 16/4.167)/2 \approx 4.003$.
3. The next estimate is about 4.000. The procedure approaches $\sqrt{16}$ and stops when its chosen closeness test succeeds. §2.1
4. Sanity check: $4^2 = 16$, so the guesses are heading for the true root.

Trap: an iteration rule is not complete until it states what counts as close enough and what happens when that test is not met.

Near miss: "start at 1 and keep adding 1 until you reach the largest whole number" has finite text and steps anyone can carry out, but it never stops, because there is no largest whole number. It fails the finite-steps check, so it is not an algorithm.

Recall check · answer without looking

1. Name the three practical ingredients of an algorithm from the slides.
2. What does the square-root procedure do when its current guess is not close enough?
3. How do finiteness and halting differ?

Answers: see "2.1 Algorithms are finite, executable procedures" under Answers at the end.

Made by Thanate Amornjiraporn (Focus)

2.2 A stored-program computer executes instructions

At bottom a computer does two things: it calculates and it remembers results, and it only knows what it is told. A stored-program computer can carry out many different procedures because its instructions are data stored in memory. The same machine follows a different algorithm when the stored program changes.

> **In plain words:** a computer only calculates and remembers. A stored-program computer keeps its instructions in the same memory as its data, so loading a different program turns the same hardware into a different machine. The processor takes each instruction from memory and carries it out.

Made by Thanate Amornjiraporn (Focus)

Instructions, control flow, and architecture

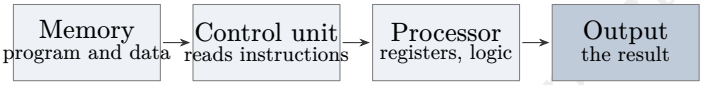
A **fixed-program computer**, such as a calculator, does one built-in job. A **stored-program computer** stores and executes instructions, so its job can change. It keeps a sequence of instructions in memory. A processor executes them using primitive operations such as arithmetic or logic, simple tests, and moving data. Tests can change which instruction runs next; execution stops when the program says it is done or reaches its stopping condition.

Machine-level view

$$\text{program in memory} \rightarrow \text{processor executes instructions} \rightarrow \text{input becomes output}$$

Read it as: "the program sits in memory, the processor executes its instructions one after another, and that turns the input into the output".

The slide's diagram: the CPU holds a control unit and a processor; the processor holds registers and combinational logic. Main memory, input and output sit outside the CPU, and instructions pass from memory to the control unit. It is a simplified model, not a particular modern computer.



A stored-program machine: instructions come from memory to the control unit, which has the processor carry them out on the data.

The lecture lists six abstract Turing-machine primitives: move right, move left, print a symbol, scan the current symbol, erase it, and do nothing (halt). Modern programming languages package low-level actions into more convenient operations, and you can use abstract methods to create new primitives. A language gives you primitives; expressions are legal combinations of them; expressions and computations have values and meanings. A computable procedure can be expressed in different general-purpose languages, even though syntax and tools differ.

Memory hook: the algorithm is the recipe; the stored program is that recipe represented as instructions the machine can execute.

Near miss: a basic calculator with fixed buttons for $+$, $-$, $\times$ and $\div$ calculates and remembers a result, but its procedures are built into its circuits rather than stored as instructions in memory; giving it a new procedure would mean building a new device. That is the difference a stored program makes.

Recall check · answer without looking

1. What does the stored-program idea place in computer memory?
2. What are the three broad kinds of primitive instructions listed in the slides?
3. What role can a test play in the flow of a program?

Answers: see "2.2 A stored-program computer executes instructions" under Answers at the end.

2.3 Programs have syntax, static meaning, and behaviour

A computer needs instructions that are legal in its language and have a defined interpretation. Separating the kinds of errors helps explain why a program can be rejected, crash, or run while still giving the wrong answer.

> **In plain words:** a program can go wrong at three levels. **Syntax** is grammar: are the symbols arranged legally? **Static semantics** is sense: do the legal pieces fit together, such as adding a number to a number? **Semantics** is meaning: what the program actually does when it runs, which may differ from what you intended.

Made by Thanate Amornjiraporn (Focus)

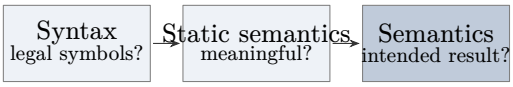
Three checks on a program expression

Syntax asks whether the symbols form a legal expression. **Static semantics** asks whether a syntactically legal expression is meaningful under the language's rules, such as whether the operands have compatible types. **Semantics** is the meaning associated with an expression that passes those checks.

Examples from the slides

$$\text{"hi"}5 \rightarrow \text{not syntactically valid}$$
$$3.2 * 5 \rightarrow \text{syntactically valid and meaningful}$$
$$3 + \text{"hi"} \rightarrow \text{syntactically valid, but a static semantic error}$$

English shows the same split: "cat dog boy" is not syntactically valid, and "I are hungry" is valid syntax with a static semantic error. The exact type rules depend on the language.



Three checks, in order: a program can pass the first two and still fail the third.

A syntactic error is common and easily caught. Only some languages check for static semantic errors before running, and these errors can cause unpredictable behaviour. A semantic error can be harder: the program may stop, run forever, or return an answer different from the programmer's intention. English can be ambiguous: "Flying planes can be dangerous" has two meanings. A program statement has exactly one meaning, but it may not be the one its author intended.

A program can contain definitions and commands. An interpreter evaluates them in a shell, or a compiler translates the program into processor instructions. Commands may be entered directly or saved in a file for the shell to read.

Trap: "it runs" does not mean "it is right". A program with no syntax or static semantic errors can still give the wrong answer.

Near miss: a program meant to average three marks that computes a + b + c / 3 has legal syntax and no static semantic error, and it runs, but the division happens before the additions, so it returns the wrong value. It passes every check except doing what was meant.

Recall check · answer without looking

1. What kind of error is "3 + 'hi'" meant to illustrate in the slides?
2. How can a program with no syntax or static-semantic errors still fail its purpose?
3. What is one difference between a compiler and an interpreter in the lecture's description?

Answers: see "2.3 Programs have syntax, static meaning, and behaviour" under Answers at the end.

2.4 Propositions turn logic into two-valued inputs

Boolean circuits calculate with truth values. Propositional logic gives precise rules for combining statements before those rules are translated into algebra or gates.

> **In plain words:** a **proposition** is a statement that is either true or false, such as "it is raining". Connectives build bigger statements from smaller ones: NOT flips the value, AND is true only when both parts are true, and OR is true when at least one is. Writing 1 for true and 0 for false turns these rules into arithmetic a circuit can do.

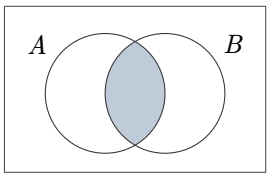
Statements and connectives

A **proposition** is a statement that is either true or false. Compound propositions use **NOT**, **AND**, and **OR**. In the notes' example, the umbrella output is true when it is raining or the forecast is bad.

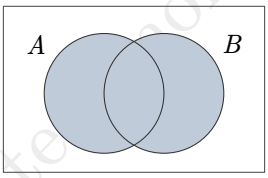
Core connectives

$$\neg A \text{ is true exactly when } A \text{ is false}$$
$$A \wedge B \text{ is true exactly when both } A \text{ and } B \text{ are true}$$
$$A \vee B \text{ is true when at least one of } A \text{ or } B \text{ is true}$$

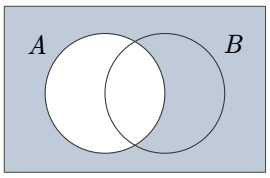
Read it as: "not A", "A and B", "A or B"; this OR includes the case where both are true.
The OR in this lecture is inclusive: it is also true when both inputs are true.



A $\wedge$ **B** is true only where both are true.



A $\vee$ **B** is true where at least one is true, including both.



$\neg$**A** is true wherever A is not.

For example, let R mean "it is raining" and W mean "the weather forecast is bad". The umbrella decision is $U = W \vee R$. If $W = 0$ and $R = 1$, then $U = 1$.

More complex decisions combine several operations. The lecture gives $U = C' \cdot (W + R)$: take an umbrella when you are not taking the car and either the forecast is bad or it is raining. Parentheses matter because the operations have different precedence.

> **Near miss:** in everyday speech "tea or coffee" often means one but not both, but this lecture's OR is inclusive: $A \vee B$ is also true when both are true. Exclusive or is a separate operation, XOR (§2.7).

> **Recall check · answer without looking**
>
> 1. What must be true for A $\wedge$ B to be true?
> 2. Can A $\vee$ B be true when A and B are both true?
> 3. Write the umbrella rule "raining or bad forecast" with R and W.
>
> *Answers: see "2.4 Propositions turn logic into two-valued inputs" under Answers at the end.*

2.5 Boolean algebra and truth tables make the rules checkable

Boolean algebra replaces long verbal propositions with symbols and gives every operation a precise meaning. Truth tables then provide a direct way to calculate an expression for every possible input.

> **In plain words:** Boolean algebra writes logic as short formulas: a prime $'$ for NOT, $\cdot$ for AND and $+$ for OR, with 1 for true and 0 for false. A **truth table** checks a formula by listing every combination of inputs with the output for each; with n inputs there are 2^n rows.

Translate propositions into algebra

Boolean values use 1 for TRUE and 0 for FALSE. Variables stand for propositions. The supplied slides write NOT as a prime, OR as $+$, and AND as $\cdot$. Read the notation carefully: here $+$ is logical OR, not ordinary arithmetic addition.

Notation and precedence

$$A' = \text{NOT } A$$
$$A \cdot B = A \text{ AND } B$$
$$A + B = A \text{ OR } B$$
$$\text{NOT first} \cdot \text{AND next} \cdot \text{OR last}$$

Read it as: "A prime is not A", "A dot B is A and B", "A plus B is A or B"; NOT is applied first, then AND, then OR.

Use parentheses to make grouping explicit. For example, $C' \cdot (W + R)$ is different from $C' \cdot W + R$. In C, logical operators are &&, ||, !; bitwise operators are &, |, (the slide labels these "Binary"; see Slips in the notes).

Made by Thanate A

Enumerate every input

A truth table lists an expression's output for every assignment of its input variables. With n Boolean variables there are 2^n assignments. Two expressions are equivalent when their output columns match in every row.

A	**B**	**A ∧ B**	**A ∨ B**	**¬A**
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Worked example · the umbrella truth table

Build the table for $U = C' \cdot (W + R)$ with partial results.

1. Three variables give $2^3 = 8$ rows. List (R,W,C) from 000 to 111.
2. Name the partial results: X1 = R + W and X2 = C′.
3. Fill X1, then X2, then U = X1 · X2.

R	**W**	**C**	**X1 = R + W**	**X2 = C′**	**U = X1 · X2**
0	0	0	0	1	0
0	0	1	0	0	0
0	1	0	1	1	1
0	1	1	1	0	0
1	0	0	1	1	1
1	0	1	1	0	0
1	1	0	1	1	1
1	1	1	1	0	0

U = 1 only for (R,W,C) = 010, 100, 110. Sanity check: those are exactly the rows with no car and some bad weather. §2.5

Made by

Implication (beyond the notes)

Implication is not in these slides; the "=>" on slide 24 only means "we could write". Take this truth table as its definition: A ⇒ B is false only for A = 1, B = 0, so its column for (A,B) = 00, 01, 10, 11 is 1, 1, 0, 1. Evaluating $A' + B$ gives 1, 1, 0, 1 as well, so the two are equivalent.

Trap: three input variables require eight rows, not six. List all assignments before calculating an output column. And + means OR here, so 1 + 1 = 1.

Near miss: 1 + 1 looks as if it should be 2, but here + means OR, and 1 OR 1 = 1. A two-valued system has no 2.

Recall check · answer without looking

1. How many rows are needed for five input variables?
2. What does it mean for two Boolean expressions to be equivalent?
3. Which operation has lower precedence: AND or OR?

Answers: see "2.5 Boolean algebra and truth tables make the rules checkable" under Answers at the end.

Made by Thanate Amornjirap

2.6 Boolean laws simplify expressions

Boolean laws let you transform an expression while preserving its output. Each identity can be checked with a truth table, and the same law can simplify the design of a circuit.

> **In plain words:** Boolean laws are rewriting rules: each swaps an expression for another that gives the same output for every input, which a truth table can confirm. They shorten formulas, and a shorter formula means a circuit with fewer gates.

Core identities

Negation, identity, and idempotence

$$(A')' = A$$
$$A \cdot A' = 0$$
$$A + A' = 1$$
$$A \cdot A = A$$
$$A + A = A$$
$$A \cdot 0 = 0$$
$$A \cdot 1 = A$$
$$A + 0 = A$$
$$A + 1 = 1$$

Read it as: "A or not A is always 1" and "A and not A is always 0": a statement and its opposite are never both true and never both false.

The slides use · for AND and + for OR. A and B may themselves be compound Boolean expressions. (A′)′ = A is listed under "Negation", and the NAND slides call it the Involution law.

Commutative, associative, and distributive

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

Parentheses keep the grouping visible. The two distributive laws mirror each other with AND and OR exchanged.

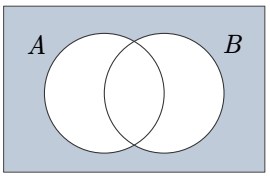
De Morgan's rules

Negating a compound expression changes its operation and negates each input: NOT of an OR becomes an AND of the negations, and NOT of an AND becomes an OR of the negations.

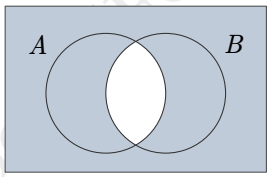
De Morgan's laws

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

Read it as: "not (A or B) equals not A and not B", and "not (A and B) equals not A or not B".
The rules hold for any number of inputs.



$(A + B)' = A' \cdot B'$: outside both circles.



$(A \cdot B)' = A' + B'$: everything except the overlap.

Worked example · simplify a negated expression

Rewrite (A + B)′ using De Morgan's law, then evaluate it when A = 0 and B = 1.

1. Apply (A + B)′ = A′ · B′.
2. Substitute the inputs: 1 · 0 = 0.
3. The original expression is false because at least one input to the OR was true. §2.6

Trap: De Morgan swaps the operator as well as priming each input. (A + B)′ is not A′ + B′.

Near miss: $(A \cdot B)' = A' \cdot B'$ looks natural, but try $A = 1$, $B = 0$: the left side is $(0)' = 1$ and the right side is $0 \cdot 1 = 0$. De Morgan's law also swaps the operator: $(A \cdot B)' = A' + B'$.

Recall check · answer without looking

1. State the De Morgan rule for NOT(A OR B).
2. What are the idempotent laws?
3. How can you check a proposed Boolean identity for two variables?

Answers: see "2.6 Boolean laws simplify expressions" under Answers at the end.

2.7 Logic gates implement Boolean functions

A gate is a circuit block whose output follows a Boolean rule. Algebra connects a circuit diagram to its expression and makes it possible to build several operations from a universal gate.

> **In plain words:** a **logic gate** is a small circuit that computes one Boolean operation; its output stands for 1 or 0 according to its inputs. A small circle on an output means the result is inverted, turning AND into NAND and OR into NOR. NAND alone can build every other gate, so it is called universal.

Read and combine gates

AND, OR, and NOT gates represent the corresponding operations. A small circle at a gate output indicates inversion, producing NAND or NOR. Follow a multi-gate circuit from its inputs toward its output, giving each intermediate signal a name.

Gate	Shape
AND	Flat input side, round output side.
OR	Curved input side, pointed output side.
NOT	Triangle with a circle. The circle means NOT; the triangle only shows the direction.
NAND	AND shape with a circle on the output.
NOR	OR shape with a circle on the output.
XOR	OR shape with an extra curve at the inputs.
XNOR	XOR shape with a circle on the output.

Inverted gates and exclusive gates

$$\text{NAND}(A, B) = (A \cdot B)'$$
$$\text{NOR}(A, B) = (A + B)'$$
$$\text{XOR}(A, B) = A' \cdot B + A \cdot B'$$
$$\text{XNOR}(A, B) = A' \cdot B' + A \cdot B$$

Read it as: "NAND of A and B is not (A and B)"; "XOR is true when exactly one input is true".
XOR is true when exactly one input is true. XNOR is true when both inputs agree.

Made for Amazon.com (Focus)

NAND is a universal gate

The lecture calls a set of functions complete when combinations of those functions can generate any Boolean function. NAND alone is complete, so it is also called a universal gate. NAND and NOR are the commonly used building blocks because they are easy to construct from transistors.

Building gates from NAND

$$\text{NOT}: (A \cdot A)' = A' \,(\text{idempotent law})$$
$$\text{AND}: ((A \cdot B)')' = A \cdot B \,(\text{involution law})$$
$$\text{OR}: A + B = (A' \cdot B')' \,(\text{De Morgan})$$
$$\text{NOR}: \text{invert the OR output}$$

NOT takes one NAND with tied inputs, AND takes two, and OR takes three: one to invert each input and one to combine them.

Worked example · translate a two-stage NAND circuit

The first NAND receives B and C and produces I. A second NAND receives A and I and produces X.

1. Name the intermediate output: $\text{I} = (\text{B} \cdot \text{C})'$.
2. Write the final gate: $\text{X} = (\text{A} \cdot \text{I})' = (\text{A} \cdot (\text{B} \cdot \text{C})')'$.
3. Use $(\text{P} \cdot \text{Q})' = \text{P}' + \text{Q}'$: $\text{X} = \text{A}' + ((\text{B} \cdot \text{C})')'$.
4. Apply double negation: $\text{X} = \text{A}' + \text{B} \cdot \text{C}$. §2.7

A	B	C	B · C	X = A′ + B · C
0	0	0	0	1
0	0	1	0	1
0	1	0	0	1
0	1	1	1	1
1	0	0	0	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Sanity check with the gates, row 1,1,0: $\text{I} = (1 \cdot 0)' = 1$, then $\text{X} = (1 \cdot 1)' = 0$, which matches the table.

Memory hook: bubbles on outputs mean "invert this result"; translate them as a prime before simplifying.

Near miss: XOR and OR agree on three of the four input rows; they differ only at $A = B = 1$, where OR gives 1 and XOR gives 0. A circuit that mixes them up is right three times out of four and still wrong.

Recall check · answer without looking

1. When is XOR true?
2. How do you express NAND using AND and NOT?
3. What does functional completeness mean for NAND?

Answers: see "2.7 Logic gates implement Boolean functions" under Answers at the end.

Made by Thanate Amornjiraporn (Focus)

2.8 Physical circuits take time to switch

A Boolean truth table describes the stable input-output rule. Real circuits also have timing behaviour because their physical components need time to change state.

> **In plain words:** a truth table says which output a gate settles on, but not when. A real gate is built from physical parts that take a short time to switch, so for a moment after an input changes, the output can still show the old value.

From ideal logic to a real signal

The final slides contrast an ideal, instantaneous gate with real switching. They identify time delay and saturation as limits on how quickly a physical circuit can change its output. Slide 48 shows the output of a NOT built from NAND following a gradual curve across a time Δt rather than jumping. Real gates are built from transistors; slide 49 shows a transistor-transistor logic (TTL) NAND gate. A logic diagram therefore describes function, while a timing diagram is needed to reason about when a change becomes visible.

Two useful models

$$\text{Boolean model} \rightarrow \text{which stable output follows each input?}$$
$$\text{Physical model} \rightarrow \text{how long does the output take to reach its new state?}$$

Read it as: the Boolean model answers "which output?"; the physical model answers "how long until the output gets there?".

The supplied slides introduce the distinction but do not provide numerical delay values or a timing-analysis procedure.



the output follows after a delay

Gate delay: the input changes at one moment and the output follows a little later.

Trap: do not claim that a real gate changes output at exactly the same instant as its input. The lecture explicitly notes switching delay.

Near miss: a circuit whose truth table is exactly right can still show a wrong value for a moment: if two signals reach a gate along paths with different delays, the gate briefly sees the new value on one input and the old value on the other. Correct logic is not the same as correct timing.

Recall check · answer without looking

1. What two real-circuit limits are named in the lecture?
2. What question does a truth table answer that a timing diagram does not?
3. Does this lecture supply a numerical propagation delay for the example gate?

Answers: see "2.8 Physical circuits take time to switch" under Answers at the end.

Formula map and symbols

Idea	Rule	Check
Truth-table size	n inputs $\rightarrow 2^n$ rows	Enumerate every assignment once.
Implication (beyond the notes)	$A \Rightarrow B = A' + B$	False only for A=1, B=0.
Complement of OR	$(A + B)' = A' \cdot B'$	Both inputs must be 0.
Complement of AND	$(A \cdot B)' = A' + B'$	At least one input must be 0.
XOR	$A' \cdot B + A \cdot B'$	Inputs differ.
XNOR	$A' \cdot B' + A \cdot B$	Inputs agree.
NAND constructions	$A' = (A \cdot A)'$; $A \cdot B = ((A \cdot B)')'$; $A + B = (A' \cdot B')'$	Name the law used.
Two-stage NAND example	$X = A' + B \cdot C$	Label the intermediate output first.

Check every answer: confirm the operator precedence, list every input assignment, and test the expression against one row that should make it true and one that should make it false.

Derive it cold

Rebuild each result from its definition before opening the worked examples.

1. Run the square-root update twice from $x = 16$ and $g = 3$; show each average. §2.1
2. Explain why the sequence, control flow, and stop rule all matter to a recipe. §2.1
3. Build the umbrella table for $U = C' \cdot (W + R)$ with its partial results. §2.5
4. Take the truth-table definition of $A \Rightarrow B$ and show its equivalence to $A' + B$. §2.5
5. Prove one De Morgan law by comparing its truth-table outputs. §2.6
6. Build NOT, AND, OR and NOR from NAND, naming the law for each. §2.7
7. Translate the two-stage NAND circuit into an expression, then simplify it to $X = A' + B \cdot C$. §2.7

Explain it

Answer each question aloud in your own words, then compare with the model answer.

1. Why is a stopping condition part of an algorithm?
2. Why can the same computer execute different algorithms?
3. Why is "3 + 'hi'" different from an expression with invalid syntax?
4. Why does a truth table with n inputs have 2^n rows?
5. Why do parentheses change $C' \cdot (W + R)$ compared with $C' \cdot W + R$?
6. Why is NAND called functionally complete?
7. Why is a Boolean truth table not enough to describe a real circuit's timing?

Answers: see "Explain it" under Answers at the end.

Made by Thanate Angonjiraporn (Thanus)

Mixed practice

Try all 25 problems before opening the worked answers. They alternate across topics so you must choose the rule, not just repeat the previous one.

1. List the three ingredients of a recipe that make it an algorithm in the practical definition from the lecture.
2. Let A=1 and B=0. Evaluate A ∧ B, A ∨ B, and ¬A.
3. A machine stores instructions and uses tests to change which step runs next. Which computer model from the lecture does this describe?
4. How many rows are needed for a truth table with four Boolean inputs?
5. Classify "3 + 'hi'" using the lecture's distinction between syntax and static semantics.
6. Rewrite $(A \cdot B)'$ using a De Morgan law.
7. For A=1 and B=0, evaluate A ⇒ B and $A' + B$. What do you observe?
8. For x=9 and current guess g=2, calculate the next square-root guess using $(g + x/g)/2$.
9. Rewrite $(A + B + C)'$ using De Morgan's law.
10. State the difference between an ideal Boolean gate model and a physical circuit model.
11. Let A=0, B=1, C=1. Evaluate $A' + B \cdot C$ using the stated precedence.
12. Give one possible behaviour of a semantic error even when a program is syntactically valid.
13. For A=1 and B=1, find A XOR B and A XNOR B.
14. Simplify $A \cdot A + A$ using Boolean laws.
15. In C notation, which operator is logical AND: & or &&?
16. A two-stage NAND circuit has $I=(A \cdot B)'$ and $X=(I \cdot C)'$. Simplify X.
17. After the first square-root update, explain what the algorithm does if the stopping test still fails.
18. Simplify $(A + B)' \cdot (A + B)$ using a complement law.
19. For the circuit in problem 16, find X when A=0, B=1, C=0, working through the gates.
20. What is the maximum number of Boolean functions of three input variables? Treat a Boolean function as one choice of output bit for each truth-table row.
21. Write A + B using only NAND operations, then name the law that makes it work.
22. What can a stored-program computer do that a calculator, as a fixed-program computer, cannot?
23. Classify each as a syntax error, a static semantic error, or valid: "hi"5, 3 + "hi", 3.2 * 5.
24. Let $V=(C + R) \cdot W'$. List every input (C,R,W) that gives V=1.
25. For C=1, W=1, R=1, evaluate $C' \cdot W + R$ and $C' \cdot (W + R)$.

Answers: see "Mixed practice" under Answers at the end.

Made by Thanos Antoniou (Focus)

Exam-style problems

Set a timer for about 20 minutes per problem. Show intermediate steps and identify the rule behind each conclusion.

1 · From a recipe to an algorithm

For this practice variation on the lecture's square-root recipe, make the stopping test explicit. For positive x, start with a positive guess g, test whether $|g^2-x|$ is below a tolerance ε, stop if so, and otherwise set g to $(g+x/g)/2$ and repeat.

1. Write the sequence of steps in plain language, including the control flow and stop condition.
2. For x=25, g=4, and ε=0.01, calculate two successive guesses and decide whether the second estimate passes the stopping test. If not, does the third?
3. Explain which algorithm properties are demonstrated by the finite instruction list, the tolerance test, and the repeated update. Name one additional condition needed to justify that every run halts.
4. Let the proposition S mean $|g^2-x|<\varepsilon$. Write the condition for doing another update as a Boolean expression, and state its value at the first guess of part 2.

Answers: see "Exam-style problems" under Answers at the end.

2 · Simplify and check a NAND circuit

A circuit has $I=(A\cdot C)'$ and $X=(B\cdot I)'$. Assume Boolean inputs.

1. Use Boolean algebra to simplify X.
2. Write the full truth table of X for A,B,C from 000 to 111.
3. Explain why the circuit can be built using NAND gates alone and why its physical output may not change instantaneously.

Answers: see "Exam-style problems" under Answers at the end.

Made by Thanate Amornjiraporn (Focus)

Slips in the notes

These were checked against the slides. Learn the corrected form, and expect that the slide wording may still be used in the course.

Slide	Issue	This guide
5	Row g = 4.17 prints g*g = 17.36, but $4.17^2 = 17.39$; 17.36 is the unrounded $(25/6)^2$. Row g = 4.0035 prints 16.0277, but $4.0035^2 = 16.0280$.	Rounding slip. §2.1 uses 4.167 and 17.36, which agree.
9, 18	"Interpreter or compiler executes each instruction." A compiler translates a program; it does not execute it. Slide 18 says commands are converted into machine language, which is the accurate part.	§§2.2–2.3 say the processor executes and the compiler translates.
10	"Compute anything" and "any other programming language" are loose: the claim holds for what is computable, in general-purpose (Turing-complete) languages.	§2.2 says "computable" and "general-purpose".
31	"Binary: AND & OR | NOT ~" should read bitwise. ~ takes one operand, so it is not binary in the usual sense.	§2.5 says bitwise.
35, 42	Notation clash, not an error: $(A')' = A$ is under "Negation" on slide 35 and "Involution law" on slide 42.	§2.6 gives both names.
24	Not an error, but a trap: "=>" here means "we could write", not implication. The slides never define implication.	§2.5 marks implication as beyond the notes.
49	Uncertain: the slide titled TTL NAND gate also shows what appears to be a MOSFET cross-section (gate, source, drain), which is probably unrelated to TTL.	§2.8 mentions only the TTL NAND gate.
7, 20, 40	Typing slips with no effect on meaning: "An Algorithms", "this this part", "combination these functions".	§§2.1 and 2.7 use the corrected wording.

Made by The Armoured Trapeze (Focus)

Self-check

Use day 0 to learn and practise. Tick a skill only in a later session, at least one day afterward, when you can demonstrate it without notes. Listed problems provide evidence.

- ☐ Explain the algorithm requirements and run the square-root update. (1, 8, 17)
- ☐ Describe the stored-program model and instruction classes. (3, 22)
- ☐ Distinguish syntax, static semantics, and semantic errors. (5, 12, 23)
- ☐ Evaluate NOT, AND, and inclusive OR, and name their C operators. (2, 7, 15)
- ☐ Determine truth-table size and enumerate inputs. (4, 20, 24)
- ☐ Apply precedence and parentheses correctly. (11, 25)
- ☐ Show $A \Rightarrow B = A' + B$ with a complete truth table (beyond the notes). (7)
- ☐ Use complement, idempotent, and De Morgan laws. (6, 9, 14, 18)
- ☐ Translate, simplify and build NAND circuits. (16, 19, 21)
- ☐ Distinguish XOR from XNOR by input agreement. (13)
- ☐ Explain the difference between Boolean behaviour and switching time. (10)
- ☐ Rebuild the listed derivations from definitions.
- ☐ Solve both multi-part problems and justify each step.

Made by Thanate Amornjiraporn (Focus)

Answers

Start from memory

1. It has well-defined executable steps, finite resource use, and halts with an output for its input. §2.1
2. Syntax concerns whether a symbol sequence is grammatically valid; semantics concerns the meaning of a valid expression. Static semantics checks whether a valid-looking expression is meaningful in the language. §2.3
3. $2^3 = 8$ rows, one for each assignment. §2.5
4. Only when A is true and B is false. Implication is not in these slides; §2.5 gives this as a definition and says so. §2.5
5. Every Boolean function can be built using NAND gates alone. §2.7

2.1 Algorithms are finite, executable procedures

1. Simple steps, control flow that determines when they run, and a rule for stopping. §2.1
2. It averages the guess g with x/g and repeats the test. §2.1
3. Finiteness says the description and each resource use stay finite; halting says the run takes a finite number of steps and terminates. §2.1

2.2 A stored-program computer executes instructions

1. A sequence of instructions that the machine can execute. §2.2
2. Arithmetic and logic, simple tests, and moving data. §2.2
3. It can determine which instruction or branch executes next. §2.2

2.3 Programs have syntax, static meaning, and behaviour

1. A static semantic error: the expression's pieces are legal-looking but their combination is not meaningful under the language's rules. §2.3
2. Its behaviour may differ from the programmer's intention: it can crash, fail to halt, or return an unexpected answer. §2.3
3. An interpreter executes/evaluates instructions in a shell; a compiler translates a program into machine instructions. The slide gives this as a broad distinction. §2.3

2.4 Propositions turn logic into two-valued inputs

1. Both A and B must be true. §2.4
2. Yes. The lecture uses inclusive OR. §2.4
3. $U = R \vee W$ (or $U = R + W$ in the Boolean notation introduced next). §2.4

Made by Thanate Amornjiraporn (Focus)

2.5 Boolean algebra and truth tables make the rules checkable

1. $2^5 = 32$ rows. §2.5
2. Their outputs match for every possible input assignment. §2.5
3. OR has lower precedence; NOT is highest and AND is in the middle. §2.5

2.6 Boolean laws simplify expressions

1. $(A + B)' = A' \cdot B'$. §2.6
2. $A \cdot A = A$ and $A + A = A$. §2.6
3. Make a truth table for both sides and compare the output columns in all four input rows. §2.6

2.7 Logic gates implement Boolean functions

1. When exactly one of its two inputs is true. §2.7
2. $(A \cdot B)'$. §2.7
3. Any Boolean function can be built from combinations of NAND gates alone. §2.7

2.8 Physical circuits take time to switch

1. Time delay and saturation limit the switching speed. §2.8
2. It gives the stable logical output for each input assignment. §2.8
3. No. The slides make the qualitative point but give no numeric delay value. §2.8

Made by Thanate Amornjiraporn (Focus)

Explain it

1. Without a stopping condition, the procedure may keep repeating and never return an answer. Halting is part of the definition, so a recipe that may run forever is not an algorithm for that input. The stop test is what turns repetition into a result. §2.1

2. The stored instructions can change while the hardware remains the same; the machine follows the program it is given. A calculator cannot do this, because its job is built in. §2.2

3. The second expression is syntactically valid but has a static semantic problem: the operands cannot be combined meaningfully under the illustrated rules. Invalid syntax, such as ”hi”5, is not a legal expression at all. The first problem concerns meaning, the second form. §2.3

4. Each variable has two possible values, so all combinations number 2 multiplied by itself n times. Every extra variable doubles the rows, because each old row appears once with 0 and once with 1. §2.5

5. NOT, AND, and OR have a precedence order; without parentheses the expression groups differently and can produce a different output. $C' \cdot W + R$ means $(C' \cdot W) + R$, so $R = 1$ gives 1 even when $C = 1$. With the parentheses the same input gives 0. §2.5

6. Any Boolean function can be assembled using NAND gates alone. NOT, AND and OR can each be built from NAND, and those three express any truth table. That is the slides' meaning of a complete set. §2.7

7. A truth table describes stable input-output values, while real gates have switching delay and saturation limits. It says where the output ends up, not when it gets there. Two circuits with the same table may differ in timing. §2.8

Made by Thanate Amornjaroen (Focus)

Mixed practice

1. A sequence of simple steps, control flow that determines when each step runs, and a way to know when to stop. §2.1
2. A ∧ B=0, A ∨ B=1, and ¬A=0. §2.4
3. A stored-program computer. §2.2
4. 2^4=16 rows. §2.5
5. It is syntactically valid but has a static semantic error in the slide's example. §2.3
6. $A' + B'$. §2.6
7. Both are 0. Their matching outputs for this row are consistent with the equivalence; check all four rows to prove it. §2.5
8. $(2 + 9/2)/2 = 6.5/2 = 3.25$. It is above $\sqrt{9} = 3$ and closer than 2 was. §2.1
9. $(A + B + C)' = A' \cdot B' \cdot C'$. §2.6
10. The Boolean model gives stable outputs for inputs; a physical circuit also takes time to switch and has limits such as saturation. §2.8
11. NOT first gives A′=1; AND next gives B·C=1; OR last gives 1+1=1. §2.5
12. It may crash, run forever, or return an answer different from the programmer's intention. §2.3
13. XOR=0 and XNOR=1 because the inputs agree. §2.7
14. A·A=A, so A·A+A=A+A=A. §2.6
15. && is logical AND; & is bitwise AND. §2.5
16. X=((A·B)′·C)′=((A·B)′)′+C′=A·B+C′. §2.7
17. Replace g by the average of g and x/g, then test the new guess again. §2.1
18. Let Y=A+B. Then Y′·Y=0 by the complement law. §2.6
19. I=(0·1)′=1, then X=(1·0)′=1. The simplified form agrees: 0·1+1=1. §2.7
20. There are 8 truth-table rows and two possible output choices per row, so 2^8=256 functions. §2.5
21. A + B=((A·A)′·(B·B)′)′. De Morgan gives (A′·B′)′=A+B, and each (X·X)′ is a NOT by the idempotent law. §2.7
22. Run different programs on the same hardware, because its instructions are stored and can change. A calculator does one built-in job. §2.2
23. "hi"5 is a syntax error; 3 + "hi" is a static semantic error; 3.2 * 5 is valid. §2.3
24. W must be 0 and C or R must be 1: (0,1,0), (1,0,0), (1,1,0). §2.5
25. C′=0, so C′·W+R=0+1=1, but C′·(W+R)=0·1=0. The parentheses change the result. §2.5

Made by Thanate Amornjirapom (Focus)

Exam-style problems

1. Set the inputs x, g, and ε. Compute $|g^2-x|$. If it is less than ε, output g and stop; otherwise replace g with $(g+x/g)/2$ and repeat the test. §2.1
2. First update: $(4+25/4)/2=5.125$. Second: $(5.125+25/5.125)/2\approx5.00152$. Its residual is about 0.0152, above $\varepsilon=0.01$, so another update is needed. The third guess is about 5.0000002, with residual about 0.000002, which passes. §2.1
3. The finite list is a finite description; its defined arithmetic and comparison steps are executable; the tolerance defines a stopping test; the repeated update is the control flow that repeats. To establish halting for every allowed input, one must show the update eventually makes the residual less than ε (and specify valid inputs such as $x>0$ and $\varepsilon>0$). §2.1
4. Update when S′ is 1. At $g=4$, $|16-25|=9$, which is not below 0.01, so $S=0$ and $S'=1$. §2.4 §2.5

Exam-style problems

1. Substitute I: $X=(B\cdot(A\cdot C)')'$. De Morgan gives $X=B'+((A\cdot C)')'$. Double negation gives $X=B'+A\cdot C$. §2.6 §2.7
2. X for 000 to 111 is 1, 1, 0, 0, 1, 1, 0, 1. $X=1$ whenever $B=0$, and for $B=1$ only when $A=C=1$. §2.5
3. The circuit as given is two NAND gates, and NAND is functionally complete, so any Boolean function can be built from it. Physical switching has delay and saturation limits, so the output needs time to change. §2.7 §2.8

Made by Thanate Amornjiraporn (Focus)

Personal study guide based on the supplied APDR lecture slides, “Algorithms, programming and data representation” and “Boolean algebra and logic,” in 02_computers.pdf (slides 1–49, dated 14 April 2024; the Boolean-logic section credits Bernhard Kainz). Written in my own words; no lecture pages or figures are reproduced. The guide’s section numbers group the unnumbered slide topics. Not an official FAU publication; check course details against the current notes. Made by **Thanate Amornjiraporn (Focus)**.