

Boolean Algebra

Discrete Math, Fall 2025

Konstantin Chukharev

Boolean Algebra

*“Мы почитаем всех нулями,
А единицами — себя.”*

— А.С. Пушкин, «Евгений Онегин»



Gottfried
Wilhelm
Leibniz



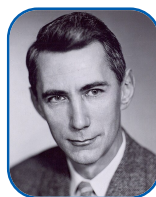
George Boole



Augustus De
Morgan



Charles
Sanders Peirce



Claude
Shannon

From Algebra to Digital Circuits

1854: George Boole publishes *Laws of Thought*.

- Radical idea: treat logic as *algebra* with operations on $\{0, 1\}$.

1937: Claude Shannon's Master's thesis:

- Shows Boolean algebra can *systematically design* relay circuits.
- No more trial-and-error engineering!

1948: Transistor invented.

- Boolean algebra becomes the foundation of digital electronics.

Today: Every processor runs on 200+ billion transistors, all following Boolean algebra rules.

Abstract symbols $(0, 1, \wedge, \vee, \neg) \implies$ Physical reality (voltages, silicon gates).

Modern Applications: Where Boolean Algebra Rules

Why it matters: Boolean algebra isn't just theory — it's the foundation of the industry.

Proving Program Correctness:

- SAT solvers decide NP-complete problems
- Model checkers verify Intel/AMD CPUs
- Formal methods prevent rocket crashes
- Automated theorem proving

Building Verified Hardware:

- Synthesis tools: RTL $\Rightarrow$ 10 billion gates
- Circuit optimizers save millions in silicon area
- AES encryption (S-box)
- Google TPUs: optimized integrated circuits

Scale: Apple M3 chip has 25 *billion* transistors. Every single gate was designed using Boolean minimization algorithms.

Lecture Overview

Foundations:

- Variables, operations, expressions
- Truth tables & equivalence
- Algebraic laws (with proofs!)
- Duality principle

Minimization:

- K-maps (visual method)
- Quine-McCluskey (systematic)
- Prime implicants & covers
- Don't-care optimization

Representations:

- Normal forms: DNF, CNF, ANF
- Canonical forms (minterms/maxterms)
- Shannon expansion
- Functional completeness (Post's theorem)

Implementation:

- Logic gates & circuits
- Sequential logic (latches, flip-flops)
- Hazards & timing
- Real hardware considerations

Variables and Expressions

Syntax: Building Blocks

Boolean variables: $x, y, z, \dots$ — represents a binary value (0 or 1).

Boolean constants: 0 (false) and 1 (true).

Boolean operations:

- $\neg x$ — negation (NOT)
- $x \wedge y$ — conjunction (AND)
- $x \vee y$ — disjunction (OR)

Boolean expressions: Built recursively from variables, constants, and operations.

Example: Simple but powerful:

- $x \wedge \neg x$ — *always* 0 (contradiction)
- $x \vee \neg x$ — *always* 1 (tautology)
- $\neg(x \wedge y)$ — NAND gate (universal!)
- $(x \vee y) \wedge (\neg z \vee w)$ — typical circuit logic

With just 3 operations ($\wedge, \vee, \neg$), you can express *any* Boolean function. We'll prove this later!

Semantics: What Expressions Mean

Every Boolean expression $f(x_1, \dots, x_n)$ defines a *function* $f : \{0, 1\}^n \rightarrow \{0, 1\}$.

Example (Access control system): Expression: $\text{admin} \wedge (\text{weekday} \vee \text{emergency})$ with 3 variables.

Truth table with $2^3 = 8$ rows determines when door unlocks.

Interpretations:

- **Logic:** predicates, propositions, proofs
- **Hardware:** voltages ($\approx 0V$ vs $\approx 5V$), transistor states
- **Programming:** `if (x && y)`, branch prediction
- **Databases:** `WHERE admin=1 AND (weekday=1 OR emergency=1)`
- **Cryptography:** S-box substitutions, linear approximations

Warning: With n variables, there are 2^{2^n} possible Boolean functions. For $n = 4$: already 65,536 functions! We need *structure* to navigate this explosion.

The State Explosion Problem

The problem: System with n Boolean state variables has 2^n reachable states.

- Real Intel CPU verification: $\approx 2^{10^9}$ states (more than you can imagine!).
- Per-component verification: from 10^{50} to 10^{10000} states (more than atoms in the *universe!*).

Explicit enumeration: *physically impossible*.

The solution: Symbolic model checking represents state sets as *Boolean formulas* (in BDDs).

- Instead of listing 1 billion states, store the formula, e.g., $x_1 \wedge (x_2 \vee \neg x_3)$.
- Ken McMillan's SMV verified hardware with 10^{20} states in 1992. Major breakthrough in verification!

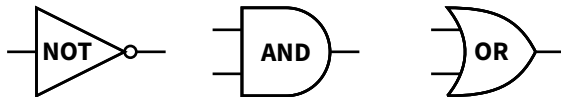
Real impact: Intel's Pentium FDIV bug (1994): \$475 million loss.

Modern CPUs are *exhaustively verified* using Boolean methods before fabrication.

Boolean Operations

Definition 1: Basic Boolean operations:

- **NOT** ($\neg x$) – negation
- **AND** ($x \wedge y$) – conjunction
- **OR** ($x \vee y$) – disjunction

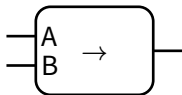


Bubble notation: A small circle (bubble) on a gate input or output indicates logical negation.

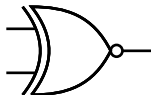
Derived Operations

Definition 2:

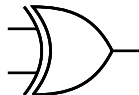
- **Implication:** $x \rightarrow y \equiv \neg x \vee y$
- **Equivalence:** $x \iff y \equiv (x \rightarrow y) \wedge (y \rightarrow x)$
- **XOR:** $x \oplus y \equiv (x \wedge \neg y) \vee (\neg x \wedge y)$



IMPLY



XNOR



XOR

Implication has no standard gate — shown as a block and can be decomposed into NOT/AND/OR.
XOR and XNOR (equivalence) have dedicated gate symbols.

CNF and SAT Solvers

SAT solvers (DPLL, CDCL) work exclusively with CNF formulas.

Why CNF? It enables *unit propagation* – a fast inference rule.

Unit propagation rule: When a clause has only one unassigned literal, that literal must be true.

Example: Clause: $(x \vee y \vee z)$ with assignments $x = 0, y = 0$

Only z remains unassigned $\Rightarrow$ must set $z = 1$ to satisfy the clause.

To use modern SAT solvers, express your problem in CNF!

Caveat: Some operations (XOR, majority) cause exponential blowup when converted to CNF.

The XOR Problem in SAT Solving

Easy case: AND/OR

Formula: $(x \vee y) \wedge z$

Already in CNF: 2 clauses

Solver handles it efficiently

Hard case: XOR

Constraint: $x \oplus y \oplus z = 1$ (odd parity)

CNF: 4 clauses!

$(x \vee y \vee z) \wedge (x \vee \neg y \vee \neg z) \wedge (\neg x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee z)$

The explosion: AES cipher: 1000+ XOR gates $\Rightarrow$ millions of CNF clauses $\Rightarrow$ SAT solver is confused!

Solution: CryptoMiniSat handles XOR natively via Gaussian elimination over $\mathbb{F}_2$.

Recursive Structure of Expressions

Definition 3: A *Boolean expression* is defined recursively:

- Variables $(x, y, z, \dots)$ and constants $(0, 1)$ are expressions
- If f and g are expressions, then so are $\neg f$, $(f \wedge g)$, $(f \vee g)$

Note: We could also include $(f \rightarrow g)$, $(f \iff g)$, $(f \oplus g)$ and many other operations, but these can be built from the basic three, so we keep the definition simple.

Note: The braces in binary expression are mandatory, except when there is no ambiguity.

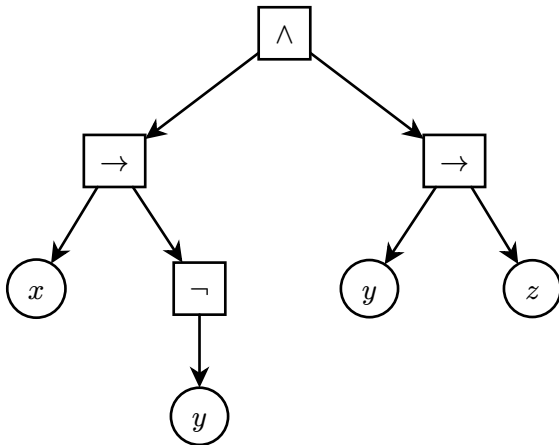
Structural induction: To prove property P for all expressions:

1. Prove P holds for variables and constants
2. Prove: if P holds for f and g , then P holds for $\neg f$, $f \wedge g$, $f \vee g$

This is THE proof technique for Boolean algebra!

Expression as Syntax Tree

Example: Expression: $(x \rightarrow \neg y) \wedge (y \rightarrow z)$



Truth Tables: Complete Function Specification

Definition 4: A *truth table* completely specifies a Boolean function $f : \{0, 1\}^n \rightarrow \{0, 1\}$ by listing output for every input combination.

Size: n variables $\Rightarrow 2^n$ rows

Example: AND operation ($x \wedge y$):

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

The Exponential Explosion

Problem: Truth tables grow exponentially!

- 3 variables: 8 rows (manageable)
- 10 variables: 1,024 rows (tedious)
- 32 variables: 4 billion rows (impossible!)

Solution: Compact representations

- Algebraic laws (rewriting)
- Normal forms (DNF, CNF)
- Symbolic methods (BDDs)
- SAT solvers (work with CNF formulas, not truth tables)

Building Complex Truth Tables

Example: Formula: $(x \wedge y) \vee (\neg x \wedge z)$

x	y	z	$x \wedge y$	$\neg x \wedge z$	Result ($\vee$)
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	0	0
0	1	1	0	1	1
1	0	0	0	0	0
1	0	1	0	0	0
1	1	0	1	0	1
1	1	1	1	0	1

Strategy: Evaluate subexpressions first, then combine.

Pattern: Result has 1s at rows 1, 3, 6, 7 (0-based) $\Rightarrow$ sum of minterms (we'll formalize this later!)

Tautologies and Contradictions

Definition 5:

- **Tautology:** $f \equiv 1$ (always true)
- **Contradiction:** $f \equiv 0$ (always false)
- **Contingency:** Sometimes true, sometimes false

Example:

- Tautology: $x \vee \neg x$ (law of excluded middle)
- Contradiction: $x \wedge \neg x$ (see Aristotle's non-contradiction law)
- Contingency: $x \wedge y$ (depends on values)

SAT Solving Connection

A formula f is:

- **Satisfiable (SAT):** has at least one solution
- **Unsatisfiable (UNSAT):** has no solutions (contradiction!)
- **Valid:** all assignments satisfy it (tautology!)

Complexity:

- “Is f satisfiable?” is NP-complete!
- “Is f valid?” is co-NP-complete (check if $\neg f$ is UNSAT)

Logical Equivalence

Definition 6: Two expressions f and g are *logically equivalent* ($f \equiv g$) if they have identical truth tables:

$$\forall x_1, \dots, x_n \in \{0, 1\} : f(x_1, \dots, x_n) = g(x_1, \dots, x_n)$$

Why this matters:

- Optimization: simpler equivalent formula
- Verification: check if circuits compute same function
- Rewriting: transform to desired form (CNF, DNF)

Verifying Equivalence: De Morgan's Law

Example: Claim: $\neg(x \wedge y) \equiv \neg x \vee \neg y$

x	y	$\neg(x \wedge y)$	$\neg x \vee \neg y$	Equal?
0	0	1	1	✓
0	1	1	1	✓
1	0	1	1	✓
1	1	0	0	✓

All rows match $\Rightarrow$ formulas are equivalent!

Expression Evaluation

Example: Formula: $f(x, y, z) = (x \vee y) \wedge (\neg x \vee z)$ with $x = 1, y = 0, z = 1$

$$\begin{aligned} f(1, 0, 1) &= (1 \vee 0) \wedge (\neg 1 \vee 1) \\ &= 1 \wedge (0 \vee 1) \\ &= 1 \wedge 1 \\ &= 1 \end{aligned}$$

Substitute values, then evaluate step-by-step from innermost operations outward.

Algebraic Simplification

Example: Simplify $f(x, y) = (x \wedge y) \vee (x \wedge \neg y)$:

$$\begin{aligned} f(x, y) &= (x \wedge y) \vee (x \wedge \neg y) \\ &= x \wedge (y \vee \neg y) && \text{distributivity} \\ &= x \wedge 1 && \text{excluded middle} \\ &= x && \text{identity} \end{aligned}$$

Reduced from 3 operations to 0!

Hardware impact: Fewer operations = fewer gates = lower cost, power, and delay.

Dual Form Simplification

Example: Simplify $g(x, y) = (x \vee y) \wedge (x \vee \neg y)$:

$$\begin{aligned} g(x, y) &= (x \vee y) \wedge (x \vee \neg y) \\ &= x \vee (y \wedge \neg y) && \text{distributivity (dual)} \\ &= x \vee 0 && \text{contradiction} \\ &= x && \text{identity} \end{aligned}$$

Complexity: Checking equivalence is co-NP-complete! Finding shortest formula is even harder — no polynomial algorithm known.

This is why circuit minimization uses heuristics (K-maps, Espresso, ABC).

Boolean Algebra Structure

The Power of Abstraction

One proof, many applications!

Once we prove a law in Boolean algebra, it applies to:

- Logical propositions ($\wedge$ = “and”, $\vee$ = “or”)
- Set operations ($\wedge$ = $\cap$, $\vee$ = $\cup$)
- Switching circuits ($\wedge$ = series, $\vee$ = parallel)
- Binary arithmetic (GF(2): $\wedge$ = $\times$, $\oplus$ = $+$)

1854: George Boole creates algebra of logic

1880s: Augustus De Morgan discovers duality laws

1937: Claude Shannon applies to switching circuits

Today: Foundation of AI, databases, hardware synthesis

Boolean Algebra: The Axiomatic Definition

Definition 7: A *Boolean algebra* is $\langle B, \vee, \wedge, \neg, 0, 1 \rangle$ where:

- $\vee$ (join), $\wedge$ (meet) – binary operations
- $\neg$ (complement) – unary operation
- 0 (bottom), 1 (top) – constants

satisfying 5 axiom groups.

Example (The prototypical Boolean algebra): $B = \{0, 1\}$ with standard operations:

- $\vee = \text{OR}$, $\wedge = \text{AND}$, $\neg = \text{NOT}$
- $\neg 0 = 1$, $\neg 1 = 0$

Stone's representation theorem (1936): Every Boolean algebra is isomorphic to a field of sets!

So, set theory and Boolean algebra are “the same” mathematically.

The Five Axiom Groups

Definition 8: For all $x, y, z \in B$:

1. **Commutativity:** $x \vee y = y \vee x, \quad x \wedge y = y \wedge x$
2. **Associativity:** $(x \vee y) \vee z = x \vee (y \vee z), \quad (x \wedge y) \wedge z = x \wedge (y \wedge z)$
3. **Distributivity:** $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z), \quad x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$
4. **Identity:** $x \vee 0 = x, \quad x \wedge 1 = x$
5. **Complement:** $x \vee \neg x = 1, \quad x \wedge \neg x = 0$

Distributivity: The Key Property

Crucial difference from arithmetic:

In $\mathbb{R}$: only $\times$ distributes over $+$

$$a \times (b + c) = ab + ac$$

But $+$ does NOT distribute over $\times$:

$$a + (b \times c) \neq (a + b) \times (a + c)$$

In Boolean algebra, BOTH operations ($\wedge$ and $\vee$) distribute over each other!

This symmetry is what makes duality possible.

Binary Strings Boolean Algebra

Example: $\mathbb{B}_n$ = n -bit strings with bitwise operations:

- $B = \{0, 1\}^n$ (all 2^n bit-vectors)
- Join: bitwise OR — $0101 \text{ OR } 0011 = 0111$
- Meet: bitwise AND — $0101 \text{ AND } 0011 = 0001$
- Complement: bitwise NOT — $\text{NOT } 0101 = 1010$
- $0 = \underbrace{00\dots0}_{n \text{ bits}}, \quad 1 = \underbrace{11\dots1}_{n \text{ bits}}$

This Boolean algebra has 2^n elements!

For $n = 8$: 256 elements (bytes)

For $n = 64$: $2^{64} \approx 10^{19}$ elements

Surprising Boolean Algebras

Example (*Divisors of 30*): $B = \{1, 2, 3, 5, 6, 10, 15, 30\}$

- Join: $a \vee b = \text{lcm}(a, b)$
- Meet: $a \wedge b = \text{gcd}(a, b)$
- Complement: $\neg a = 30 / a$
- Bottom: $\perp = 1$
- Top: $\top = 30$

Example: $6 \vee 10 = 30$, $6 \wedge 10 = 2$

Check axioms yourself!

Commutativity, associativity work because gcd/lcm have these properties.

Distributivity: $\text{gcd}(a, \text{lcm}(b, c)) = \text{lcm}(\text{gcd}(a, b), \text{gcd}(a, c))$

Derived Laws

Theorem 1: From the 5 axioms, we can prove:

1. **Idempotence:** $x \vee x = x, \quad x \wedge x = x$
2. **Absorption:** $x \vee (x \wedge y) = x, \quad x \wedge (x \vee y) = x$
3. **Domination:** $x \vee 1 = 1, \quad x \wedge 0 = 0$
4. **Involution:** $\neg\neg x = x$

These are NOT axioms — they are consequences!

We prove them using only the 5 axiom groups.

Proving Idempotence

Theorem 2: $x \vee x = x$ for all x in a Boolean algebra.

Proof:

$x \vee x = (x \vee x) \wedge 1$	identity axiom
$= (x \vee x) \wedge (x \vee \neg x)$	complement axiom
$= x \vee (x \wedge \neg x)$	distributivity
$= x \vee 0$	complement axiom
$= x$	identity axiom

□

Standard technique: Introduce 1 or 0, apply distributivity, eliminate with complements.

Proving Absorption

Theorem 3: $x \vee (x \wedge y) = x$ for all x, y in a Boolean algebra.

Proof:

$$\begin{aligned}x \vee (x \wedge y) &= (x \wedge 1) \vee (x \wedge y) && \text{(identity)} \\&= x \wedge (1 \vee y) && \text{(distributivity)} \\&= x \wedge 1 && \text{(domination)} \\&= x && \text{(identity)}\end{aligned}$$

□

Similarly, $x \wedge (x \vee y) = x$ by dual reasoning.

Duality: swap $\wedge \Leftrightarrow \vee$ and $0 \Leftrightarrow 1$.

Circuit Optimization via Absorption

Example: Expression: “enable OR (enable AND ready)”

Simplify using absorption law:

$$\text{enable} \vee (\text{enable} \wedge \text{ready}) = \text{enable}$$

Result: just “enable”!

Hardware impact:

Before: 1 AND gate + 1 OR gate = 2 gates

After: 0 gates (direct wire)

Saved: chip area, power consumption, propagation delay

Algebraic Laws and Duality

The Duality Principle

Theorem 4: If an identity holds in any Boolean algebra, then the *dual* identity (obtained by swapping $\vee \Leftrightarrow \wedge$ and $0 \Leftrightarrow 1$) also holds.

Original	Dual
$x \vee 0 = x$	$x \wedge 1 = x$
$x \vee (x \wedge y) = x$	$x \wedge (x \vee y) = x$
$\overline{x \vee y} = \overline{x} \wedge \overline{y}$	$\overline{x \wedge y} = \overline{x} \vee \overline{y}$

Free theorems: Every proof gives TWO results!

Duality is built into the axioms: AND and OR appear symmetrically in all five axiom groups. This structural symmetry makes duality automatic, not a coincidence.

De Morgan's Laws

Theorem 5 (De Morgan): For all x, y in a Boolean algebra:

$$\neg(x \vee y) = \neg x \wedge \neg y$$

$$\neg(x \wedge y) = \neg x \vee \neg y$$

Intuition: Negation “flips” everything — distributes by swapping AND $\Leftrightarrow$ OR!

Example: Database query:

`NOT (admin OR vip)  $\equiv$  NOT admin AND NOT vip`

“Neither admin nor VIP”

Connection to Lattices

Definition 9: A *lattice* is a partially ordered set where any two elements x, y have:

- A *least upper bound* (join): $x \vee y$
- A *greatest lower bound* (meet): $x \wedge y$

Example: Power set $\mathcal{P}(\{a, b\})$ ordered by $\subseteq$:

- Join = union: $\{a\} \vee \{b\} = \{a\} \cup \{b\} = \{a, b\}$
- Meet = intersection: $\{a\} \wedge \{a, b\} = \{a\} \cap \{a, b\} = \{a\}$

Boolean Algebras Are Special Lattices

Definition 10: A *bounded distributive lattice* is a lattice with bottom $\perp$, top $\top$, and distributivity:

$$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$$

Definition 11: A *complemented lattice* is a bounded lattice where every element x has a complement y such that $x \vee y = \top$ and $x \wedge y = \perp$.

Theorem 6: Every Boolean algebra is a complemented bounded distributive lattice.

Translation: $\wedge \Rightarrow \wedge$ (meet), $\vee \Rightarrow \vee$ (join), $0 \Rightarrow \perp$ (bot), $1 \Rightarrow \top$ (top).

Boolean algebras add complement uniqueness + extra structure.

Complement Uniqueness

Theorem 7: In a Boolean algebra, each element has a *unique* complement.

Proof: Suppose y and z are both complements of x . Then:

$$\begin{aligned}y &= y \wedge 1 && \text{identity} \\&= y \wedge (x \vee z) && z \text{ is complement} \\&= (y \wedge x) \vee (y \wedge z) && \text{distributivity} \\&= 0 \vee (y \wedge z) && y \text{ is complement} \\&= (x \wedge z) \vee (y \wedge z) && z \text{ is complement} \\&= (x \vee y) \wedge z && \text{distributivity} \\&= 1 \wedge z && y \text{ is complement} \\&= z && \text{identity}\end{aligned}$$

Therefore $y = z$.

□

Summary: Boolean Algebra Structure

Core points:

- A Boolean algebra is an algebraic structure $(B, \vee, \wedge, \neg, 0, 1)$ satisfying five axiom groups.
- From the axioms follow the essential laws used in practice: De Morgan, idempotence, absorption.
- Duality: every valid identity yields a dual identity (swap OR $\Leftrightarrow$ AND, 0 $\Leftrightarrow$ 1).

Duality is built into the axioms — AND and OR appear symmetrically.

Every proof yields TWO results automatically!

What's next? We've mastered the algebra! Now we'll see how to *represent* any function systematically using normal forms (DNF, CNF, ANF).

Normal Forms

The Representation Problem

Challenge: How to represent ANY Boolean function systematically?

For n variables: 2^{2^n} different functions!

Example: $n = 3$ gives 256 different functions.

Solution: Canonical forms

- DNF (Disjunctive Normal Form)
- CNF (Conjunctive Normal Form)
- Minterms and maxterms

Building Blocks: Literals

Definition 12: A *literal* is a variable or its negation.

Example: Variables x, y, z give literals:

- Positive: x, y, z
- Negative: $\neg x, \neg y, \neg z$

Literals are the atoms of Boolean logic.

Every expression is built from literals using $\wedge$ and $\vee$.

Cubes and Clauses

Definition 13:

- **Cube (product):** conjunction (AND) of literals
- **Clause (sum):** disjunction (OR) of literals

Example *cubes*:

- $x \wedge y$
- $x \wedge \neg y \wedge z$
- $\neg x \wedge \neg y \wedge \neg z$

Example *clauses*:

- $x \vee y$
- $x \vee \neg y \vee z$
- $\neg x \vee \neg y \vee \neg z$

Cube: ALL literals must be true.

Clause: at least ONE literal must be true.

Disjunctive Normal Form (DNF)

Definition 14: A formula is in *Disjunctive Normal Form (DNF)* if it is a disjunction (OR) of cubes:

$$f(x_1, x_2, \dots, x_n) = (c_1) \vee (c_2) \vee \dots \vee (c_k)$$

where each c_i is a cube (conjunction of literals).

Example: DNF formulas:

- $(x \wedge y) \vee (\neg x \wedge z)$ – “(x and y) OR (¬x and z)” (2 cubes)
- $(x \wedge \neg y \wedge z) \vee (\neg x \wedge y) \vee z$ – three alternatives (3 cubes)
- $x \vee (y \wedge \neg z)$ – still DNF (mixed form) (2 cubes: x and $y \wedge \neg z$)

Example: Formulas **NOT** in DNF:

- $(x \vee y) \wedge z$ – this is CNF (AND of ORs)
- $x \wedge (y \vee z)$ – OR nested inside AND violates DNF structure

Intuition: DNF says “the output is 1 if ANY of these scenarios (combinations) happen.”

Conjunctive Normal Form (CNF)

Definition 15: A formula is in *Conjunctive Normal Form (CNF)* if it is a conjunction (AND) of clauses:

$$f(x_1, x_2, \dots, x_n) = (c_1) \wedge (c_2) \wedge \dots \wedge (c_k)$$

where each c_i is a clause (disjunction of literals).

Example: CNF formulas:

- $(x \vee y) \wedge (\neg x \vee z)$ — both constraints must hold
- $(x \vee \neg y \vee z) \wedge (\neg x \vee y) \wedge z$ — three constraints
- $x \wedge (y \vee \neg z)$ — still CNF (mixed form)

Example: **NOT** in CNF:

- $(x \wedge y) \vee z$ — this is DNF (OR of ANDs)
- $x \vee (y \wedge z)$ — AND nested inside OR violates CNF structure

Intuition: CNF says “the output is 1 only if ALL of these constraints are satisfied.”

CNF vs DNF

Intuition:

- **CNF** says “ALL of these constraints must be satisfied.”
 - Each constraint (clause) offers multiple ways to be true.
- **DNF** says “ANY of these scenarios can make it true.”
 - Each scenario (cube) is a specific combination (product) of variable values.

Duality: DNF and CNF are *dual* forms.

Swap $\wedge$ and $\vee$ to convert between them.

Minterms and Maxterms

Definition 16: For n variables:

- A *minterm* is a maximal cube containing all n variables
- A *maxterm* is a maximal clause containing all n variables

Example: For variables x, y, z :

- **Minterms:** $(x \wedge y \wedge z), (x \wedge y \wedge \neg z), (x \wedge \neg y \wedge z), \dots$
- **Maxterms:** $(x \vee y \vee z), (x \vee y \vee \neg z), (x \vee \neg y \vee z), \dots$

There are exactly $2^n = 2^3 = 8$ minterms and 8 maxterms for $n = 3$ variables.

Key property:

- Each minterm is 1 for **exactly ONE** input combination
- Each maxterm is 0 for **exactly ONE** input combination

Notation: From Formal to Algebraic

We transition from *formal logical notation* to *algebraic notation* (standard in Boolean algebra):

Formal notation:

- Conjunction: $\wedge$
- Disjunction: $\vee$
- Negation: $\neg$
- Requires parentheses
- Used in *formal logic* where syntax matters

Example: $(x \wedge y) \vee (\neg x \wedge z)$

Algebraic notation:

- Conjunction: juxtaposition xy
- Disjunction: plus $x + y$
- Negation: overline $\overline{x}$
- Precedence: NOT > AND > OR
- Standard in *Boolean algebra* applications

Example: $xy + \overline{x}z$

Why algebraic? More compact, easier to manipulate algebraically, matches circuit notation.

Precedence rules allow omitting many parentheses: $x + yz$ instead of $(x \vee (y \wedge z))$.

Critical: Use notation *consistently!* Never mix $+$, $\vee$, $\cdot$, $\wedge$ in the same formula.

Minterm and Maxterm Indexing

We can index minterms and maxterms by their binary representations:

Example: For x, y, z (interpreting as bits: xyz):

Index	Binary	Minterm m_i	Maxterm M_i
0	000	$\bar{x}\bar{y}\bar{z}$	$x + y + z$
1	001	$\bar{x}\bar{y}z$	$x + y + \bar{z}$
2	010	$\bar{x}y\bar{z}$	$x + \bar{y} + z$
3	011	$\bar{x}yz$	$x + \bar{y} + \bar{z}$
4	100	$x\bar{y}\bar{z}$	$\bar{x} + y + z$
5	101	$x\bar{y}z$	$\bar{x} + y + \bar{z}$
6	110	$xy\bar{z}$	$\bar{x} + \bar{y} + z$
7	111	xyz	$\bar{x} + \bar{y} + \bar{z}$

Note: Why $M_0 = x + y + z$? Because $M_0 = 0$ only when $x = 0, y = 0, z = 0$.

Sum of Products (SoP)

Definition 17: A *sum of products (SoP)* (also called *canonical sum of minterms*) is a DNF where each cube is a minterm.

General form: $f(x_1, \dots, x_n) = \bigvee_{i \in I} m_i$ where $I \subseteq \{0, 1, \dots, 2^n - 1\}$

Note: In SoP, each cube is *maximal* (contains all variables). This form is *unique* for any given function!

Recipe: Find rows where $f = 1 \Rightarrow$ write minterms $m_i \Rightarrow$ OR them together.

Why it works: Each minterm is 1 for exactly one row $\Rightarrow$ OR gives 1 when ANY minterm is 1.

SoP: Example

Example: For the function f with truth table:

Step 1: Identify rows where $f = 1$: rows 1, 3, 6, 7

Step 2: Write corresponding minterms:

- Row 1 (001): $m_1 = \bar{x}\bar{y}z$
- Row 3 (011): $m_3 = \bar{x}yz$
- Row 6 (110): $m_6 = xy\bar{z}$
- Row 7 (111): $m_7 = xyz$

Step 3: OR them together:

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

$$\begin{aligned}f &= m_1 \vee m_3 \vee m_6 \vee m_7 = \\&= \bar{x}\bar{y}z + \bar{x}yz + xy\bar{z} + xyz\end{aligned}$$

Product of Sums (PoS)

Definition 18: A *product of sums (PoS)* (also called *canonical product of maxterms*) is a CNF where each clause is a maxterm.

General form: $f(x_1, \dots, x_n) = \bigwedge_{i \in I} M_i$ where $I \subseteq \{0, 1, \dots, 2^n - 1\}$

Example: For the **same function**, use rows where $f = 0$ (rows 0, 2, 4, 5):

$$\begin{aligned} f &= M_0 \wedge M_2 \wedge M_4 \wedge M_5 \\ &= (x + y + z) \cdot (x + \bar{y} + z) \cdot (\bar{x} + y + z) \cdot (\bar{x} + y + \bar{z}) \end{aligned}$$

Two approaches:

- SoP uses 1s in truth table (where function is true)
- PoS uses 0s in truth table (where function is false)
- Both represent the same function!

Why PoS?

Sometimes PoS is more compact than SoP (when function has fewer 0s than 1s).

Completeness of Normal Forms

Theorem 8: Every Boolean function can be represented in both CNF and DNF.

Proof (*sketch*) for DNF (SoP): Given a truth table:

1. For each row where output is 1, create the corresponding minterm
2. OR all these minterms together
3. The result is the function in DNF (SoP form)

Why this works:

- Each minterm is 1 for exactly one input combination
- ORing them gives 1 exactly when the function should be 1
- This construction is always possible and always correct



Shannon Expansion

Theorem 9 (Shannon Expansion): For any Boolean function f and variable x :

$$f(x, y_1, \dots, y_n) = \bar{x} \cdot f_0(y_1, \dots, y_n) + x \cdot f_1(y_1, \dots, y_n)$$

where $f_0 = f(0, y_1, \dots, y_n)$ and $f_1 = f(1, y_1, \dots, y_n)$ are the *cofactors* of f with respect to x .

Dual form:

$$f(x, y_1, \dots, y_n) = (x + f_0)(\bar{x} + f_1)$$

Shannon Expansion: Example

Example: Expand $f(x, y) = x \oplus y$ by variable x :

Step 1: Compute cofactors:

- $f_0 = f(0, y) = 0 \oplus y = y$
- $f_1 = f(1, y) = 1 \oplus y = \bar{y}$

Step 2: Apply Shannon expansion:

$$\begin{aligned} f(x, y) &= \bar{x} \cdot f_0 + x \cdot f_1 \\ &= \bar{x}y + x\bar{y} \end{aligned}$$

This gives the standard XOR representation!

Key idea: Decompose function by fixing one variable to 0 and 1, then combine results.

The *cofactors* f_0 and f_1 are simpler subfunctions.

Shannon Expansion: Applications

Binary Decision Diagrams (BDDs):

Apply Shannon expansion recursively to build a *decision tree*.

With variable ordering and reduction, this becomes a *canonical* graph representation.

BDDs are used by Intel/AMD to verify chip designs with billions of gates.

SAT Solving (DPLL Algorithm):

DPLL uses Shannon expansion for systematic case-split reasoning: pick variable x , recursively solve $f[x = 0]$ and $f[x = 1]$, exploring the solution space via cofactor decomposition.

Circuit decomposition: Break large circuits into blocks for hierarchical synthesis (EDA tools).

Converting Arbitrary Formulas to Normal Forms

General algorithm for converting ANY Boolean formula to DNF or CNF:

1. Eliminate “non-standard” operations (rewrite in terms of $\wedge$, $\vee$, $\neg$):

- $x \iff y \rightsquigarrow (x \wedge y) \vee (\bar{x} \wedge \bar{y})$
- $x \rightarrow y \rightsquigarrow \bar{x} \vee y$

2. Push negations inward (resulting in NNF – Negation Normal Form):

- $\overline{x \wedge y} \rightsquigarrow \bar{x} \vee \bar{y}$
- $\overline{x \vee y} \rightsquigarrow \bar{x} \wedge \bar{y}$
- Eliminate double negations: $\bar{\bar{x}} \rightsquigarrow x$

3. Distribute to desired form:

- For DNF: $(x \vee y) \wedge z \rightsquigarrow_{\text{DNF}} (x \wedge z) \vee (y \wedge z)$
- For CNF: $(x \wedge y) \vee z \rightsquigarrow_{\text{CNF}} (x \vee z) \wedge (y \vee z)$

4. Simplify: Apply $x \vee \bar{x} = 1$, $x \wedge \bar{x} = 0$, $x \vee 0 = x$, $x \wedge 1 = x$

Converting to DNF: Example

Example: Convert $(x \rightarrow y) \wedge (y \iff z)$ to DNF

Step 1: Eliminate $\rightarrow$ and $\iff$:

$$f = (\bar{x} \vee y) \wedge ((y \wedge z) \vee (\bar{y} \wedge \bar{z}))$$

Step 2: Already in NNF (negations only on literals)

Step 3: Distribute AND over OR to get DNF:

$$\begin{aligned} f &= ((\bar{x} \vee y) \wedge (y \wedge z)) \vee ((\bar{x} \vee y) \wedge (\bar{y} \wedge \bar{z})) \\ &= (\bar{x} \wedge y \wedge z) \vee (y \wedge y \wedge z) \vee (\bar{x} \wedge \bar{y} \wedge \bar{z}) \vee (y \wedge \bar{y} \wedge \bar{z}) \end{aligned}$$

Step 4: Simplify:

$$\begin{aligned} f &= (\bar{x} \wedge y \wedge z) \vee (y \wedge z) \vee (\bar{x} \wedge \bar{y} \wedge \bar{z}) \vee 0 \\ &= (\bar{x} \wedge y \wedge z) \vee (y \wedge z) \vee (\bar{x} \wedge \bar{y} \wedge \bar{z}) \end{aligned}$$

Converting to DNF: Example [2]

Further simplification (absorption): $(\bar{x} \wedge y \wedge z)$ is absorbed by $(y \wedge z)$:

$$f = (y \wedge z) \vee (\bar{x} \wedge \bar{y} \wedge \bar{z})$$

Result: DNF with 2 cubes

Converting to CNF: Example

Example: Convert $(x \rightarrow y) \vee z$ to CNF

Step 1: Eliminate $\rightarrow$:

$$f = (\bar{x} \vee y) \vee z = \bar{x} \vee y \vee z$$

Step 2: Already in NNF

Step 3: Already in CNF (single clause!)

Result: CNF with 1 clause $(\bar{x} \vee y \vee z)$

Key insight: Some formulas are naturally close to one normal form. Choose the form that results in simpler representation.

DNF to CNF: Manageable Case

Now consider converting *between* DNF and CNF (not from arbitrary formulas).

Example (DNF to CNF with modest growth): Given DNF: $(x \wedge y) \vee (\bar{x} \wedge z)$

Step	Expression
Step 1: Distribute OR over AND	$f = (x \vee (\bar{x} \wedge z)) \wedge (y \vee (\bar{x} \wedge z))$
Step 2: Distribute inner ORs	$f = ((x \vee \bar{x}) \wedge (x \vee z)) \wedge ((y \vee \bar{x}) \wedge (y \vee z))$
Step 3: Simplify complements	$f = (1 \wedge (x \vee z)) \wedge ((\bar{x} \vee y) \wedge (y \vee z))$
Step 4: Apply identity	$f = (x \vee z) \wedge (\bar{x} \vee y) \wedge (y \vee z)$

Result: 2 cubes $\Rightarrow$ 3 clauses (modest growth)

DNF to CNF: Exponential Blowup

Example (DNF to CNF with exponential explosion): Consider DNF with 4 cubes:

$$f = (a \wedge b) \vee (c \wedge d) \vee (e \wedge f) \vee (g \wedge h)$$

Converting to CNF requires distributing OR over all ANDs:

$$f = (a \vee c \vee e \vee g) \wedge (a \vee c \vee e \vee h) \wedge (a \vee c \vee f \vee g) \wedge \dots$$

Count the clauses: Each cube contributes 2 literals, giving $2^4 = 16$ possible combinations.

Result: 4 cubes $\Rightarrow$ 16 clauses (exponential blowup!)

Warning: DNF with n cubes, each having k literals, can produce up to k^n clauses in CNF.

For $n = 10$ cubes with $k = 2$ literals each: $2^{10} = 1024$ clauses!

Summary: Canonical Forms

Core concepts:

- Literals, cubes (AND), clauses (OR)
- DNF: OR of cubes (ANY scenario)
- CNF: AND of clauses (ALL constraints)
- Minterms/maxterms: all variables
- SoP/PoS: unique canonical forms
- Shannon expansion: cofactor decomposition

Key properties:

- Every function has both DNF and CNF
- Truth table $\Rightarrow$ circuit via SoP/PoS
- NNF: intermediate for conversion
- DNF $\Leftrightarrow$ CNF can explode exponentially
- Canonical forms are often huge $\Rightarrow$ minimization needed

What's next: Minimization techniques to reduce circuit size, cost, and power consumption.

Minimization

The Minimization Problem

We can express *any* Boolean function from its truth table using SoP or PoS.

But the result is often **not minimal**.

Example: Consider $f(x, y, z) = (\neg x \wedge \neg y \wedge z) \vee (\neg x \wedge y \wedge z) \vee (x \wedge y \wedge \neg z) \vee (x \wedge y \wedge z)$

Canonical DNF: 4 cubes, 12 literals total (7 gates needed).

After minimization: $f = (\neg x \wedge z) \vee (x \wedge y)$ — only 4 literals (3 gates).

Result: 57% fewer gates, less power, faster operation.

Why minimize?

- **Cheaper:** Fewer gates = lower cost
- **Power:** Each gate consumes energy
- **Speed:** Fewer delays = faster
- **Yield:** Fewer components = fewer defects
- **Verification:** Simpler = easier to test

The challenge: Minimization is NP-complete.

Practical techniques:

- K-maps (2-5 vars)
- Quine-McCluskey (6-8 vars)
- ESPRESSO heuristics (100+ vars)

What Does “Minimal” Mean?

Different minimization criteria (goals) exist:

- **Minimum literals:** Fewest total literal occurrences
- **Minimum terms:** Fewest cubes (DNF) or clauses (CNF)
- **Minimum gates:** Fewest logic gates in circuit
- **Minimum levels:** Shortest signal propagation path (depth)

Example: Function: $f = \overline{A}B + AC + BC$

- Has 6 literals, 3 cubes
- Can be reduced to: $f = \overline{A}B + AC$ (using consensus theorem)
- Now 4 literals, 2 cubes

Usually, we minimize the number of literals (most common criterion).

Gray Code: Foundation of K-Maps

Definition 19: A *Gray code* is a binary encoding where consecutive values differ in exactly one bit. This single-distance property eliminates transition errors in electromechanical systems.

Historical note:

- Invented by Frank Gray (Bell Labs, 1947) for shaft encoders
- Prevents spurious signals when multiple bits change simultaneously
- Still used today in rotary encoders, genetics (binary gene representation), puzzle solving (Hamiltonian paths on hypercubes)

Example: 3-bit Gray code sequence:

Decimal	Binary	Gray Code	Bit Changed
0	000	000	(bit 1)
1	001	001	bit 0
2	010	011	bit 1

Gray Code: Foundation of K-Maps [2]

Decimal	Binary	Gray Code	Bit Changed
3	011	010	bit 0
4	100	110	bit 2
5	101	111	bit 0
6	110	101	bit 1
7	111	100	bit 0

K-map connection: Adjacent cells in Karnaugh maps use Gray code ordering. This ensures neighboring cells differ in exactly one variable, making algebraic simplification patterns visually apparent.

The key identity: $(x \wedge y) \vee (x \wedge \neg y) = x$ (variable y cancels out)

Converting Binary and Gray Code

Binary to Gray:

- Keep MSB (most significant bit)
- Each next bit: XOR current binary bit with previous binary bit

Gray to Binary:

- Keep MSB
- Each next bit: XOR current Gray bit with previous binary bit

Example: Binary $1011_2 \rightarrow$ Gray:

- Bit 3: 1 (keep MSB)
- Bit 2: $1 \oplus 0 = 1$
- Bit 1: $0 \oplus 1 = 1$
- Bit 0: $1 \oplus 1 = 0$
- **Result:** 1110_{Gray}

Introduction to Karnaugh Maps

Definition 20: A *Karnaugh map (K-map)* is a 2D grid representation of a truth table, arranged using Gray code so that adjacent cells (including wraparound) differ in exactly one variable.

Core idea: K-maps transform algebraic minimization into visual pattern recognition. Adjacent groups of 1s correspond to cubes that can be simplified by eliminating variables.

Advantages:

- Visual pattern recognition
- Fast for small functions (2-5 variables)
- Educational value

Limitations:

- Practical only for ≤ 6 variables
- Manual grouping required
- Does not scale to large functions

K-maps produce minimal **DNF** by grouping 1s.

For minimal **CNF**, group 0s instead and apply De Morgan's laws to the result.

2-Variable K-Map: Step by Step

Example: Build K-map for $f(x, y) = x \vee y$.

Step 1: Create 2×2 grid and fill in the truth values

Step 2: Group adjacent 1s

- Horizontal pair (bottom row): $x = 1 \rightarrow$ gives term x
- Vertical pair (right column): $y = 1 \rightarrow$ gives term y

Result: $f = x \vee y$

		y	
		0	1
x	0	0 ₀	1 ₁
	1	1 ₂	1 ₃

3-Variable K-Map Structure

For 3 variables, we use a 4×2 grid (two variables for rows, one for columns):

		z	
		0	1
xy	00	0	1
	01	2	3
	11	6	7
	10	4	5

Note: Row order: 00, 01, **11**, 10 (Gray code, NOT binary!)

This ensures top-bottom adjacency and wrap-around (*torus* structure).

Remember: The map *wraps around* — top and bottom rows are adjacent!

3-Variable K-Map: Complete Example

Example: Minimize $f(x, y, z) = \sum m(1, 3, 6, 7)$:

Step 1: Draw 4×2 grid and fill in the truth values

Step 2: Identify groupings

- **Red group:**

- ▶ Variables: $x = 0, z = 1, y$ varies
- ▶ Term: $\neg x \wedge z$

- **Green group:**

- ▶ Variables: $x = 1, y = 1, z$ varies
- ▶ Term: $x \wedge y$

		z	
		0	1
xy	00	0 ₀	1 ₁
	01	0 ₂	1 ₃
	11	1 ₆	1 ₇
	10	0 ₄	0 ₅

Step 3: Write minimal DNF

$$f = (\neg x \wedge z) \vee (x \wedge y) = \bar{x}z + xy$$

How K-Map Grouping Works

Grouping principles:

- Variables that *change* within the group are eliminated
- Variables that stay *constant* remain in the term
- Larger groups $\Rightarrow$ more variables eliminated $\Rightarrow$ simpler terms

Example: Group of 2 cells in 3-var K-map:

Cell	xyz	Analysis
m_1	001	$x = 0, y = 0, z = 1$
m_3	011	$x = 0, y = 1, z = 1$
		$x = 0, z = 1, y \text{ varies}$

Result: $\neg x \wedge z$ (y eliminated)

4-Variable K-Map Structure

For 4 variables, use a 4×4 grid with Gray code on both axes:

		CD			
		00	01	11	10
AB	00	0	1	3	2
	01	4	5	7	6
	11	12	13	15	14
	10	8	9	11	10

Important: Both rows and columns wrap around! Treat the map as a *torus*:

- Top $\iff$ Bottom are adjacent
- Left $\iff$ Right are adjacent
- Even corners can be grouped!

4-Variable K-Map: Complete Example

Example:

		CD			
		00	01	11	10
AB	00	0	1	3	2
	01	4	5	7	6
	11	12	13	15	14
	10	8	9	11	10

Group	Cells	Pattern	Term
Corners	0, 2, 8, 10	$B = 0, D = 0, A \text{ and } C \text{ vary}$	$\overline{B} \overline{D}$
Vertical	5, 13	$B = 1, C = 0, D = 1, A \text{ varies}$	$B \overline{C} D$

Minimal DNF: $f = \overline{B} \overline{D} + B \overline{C} D$

K-Map Grouping: Valid Group Sizes

Rules for valid groups:

1. Size must be a power of 2: 1, 2, 4, 8, or 16 cells
2. Shape must be rectangular (1×2 , 2×2 , 4×4 , 1×8 , *etc.*)
3. Can wrap around edges (torus topology)
4. Larger groups are always better (fewer literals)

Example: Relationship between group size and simplification:

Group Size	Variables Eliminated	Literals Left	Example Term
1 cell	0	4	$ABCD$
2 cells	1	3	ABC
4 cells	2	2	AB
8 cells	3	1	A
16 cells	4	0	1 (tautology)

K-Map Strategy: Optimal Grouping

Grouping rules:

1. Mark all 1s in the K-map
2. Find essential groups (cells covered by only one group)
3. Cover remaining 1s with largest rectangles (8, 4, 2, 1)
4. Use wraparound on edges and corners
5. Minimize overlap but allow if needed

K-Maps with Don't-Care Conditions

Definition 21 (Don't-Care Conditions): Situations where output value doesn't matter (marked as $\times$):

- Invalid input combinations
- Outputs that are never used
- Incompletely specified functions

Example: BCD (Binary Coded Decimal) uses only 0-9:

Inputs 1010-1111 are don't-cares (invalid BCD)

Strategy with don't-cares:

- Treat $\times$ as 0 or 1 to maximize group sizes
- Include $\times$ in groups if it helps
- Don't create groups containing only $\times$ values

Don't-Care Example

Example: Function with don't-cares at positions 9, 11, 12, 15

		CD			
		00	01	11	10
AB	00	0 ₀	1 ₁	1 ₃	0 ₂
	01	0 ₄	0 ₅	0 ₇	0 ₆
	11	× ₁₂	1 ₁₃	× ₁₅	0 ₁₄
	10	1 ₈	× ₉	× ₁₁	0 ₁₀

- Group 1: includes cells 1, 3 (filled with 1s) $\Rightarrow$ gives $\overline{A} \overline{B} D$
- Group 2: includes cells 8, 9, 12, 13 (uses don't-cares) $\Rightarrow$ gives $A \overline{C}$

Without don't-cares, we would need more terms to cover all 1s!

Algebraic Minimization

Beyond K-maps, we can minimize algebraically using Boolean laws:

Essential simplification laws:

Law	Sum Form	Product Form
Idempotent	$X + X = X$	$X \cdot X = X$
Absorption	$X + XY = X$	$X(X + Y) = X$
Combining	$XY + X\bar{Y} = X$	$(X + Y)(X + \bar{Y}) = X$
Consensus theorem	$XY + \bar{X}Z + YZ = XY + \bar{X}Z$	$(X + Y)(\bar{X} + Z)(Y + Z) = (X + Y)(\bar{X} + Z)$

Step-by-Step Algebraic Minimization

Example: Minimize $f = ABC + AB\overline{C} + \overline{A}BC + \overline{A}\overline{B}C$:

Step 1: Look for combining opportunities

$$\begin{aligned}f &= ABC + AB\overline{C} + \overline{A}BC + \overline{A}\overline{B}C \\&= AB(C + \overline{C}) + \overline{A}BC + \overline{A}\overline{B}C \quad (\text{factor}) \\&= AB + \overline{A}BC + \overline{A}\overline{B}C \quad (\text{complement})\end{aligned}$$

Step 2: Apply more combining

$$\begin{aligned}f &= AB + \overline{A}C(B + \overline{B}) \quad (\text{factor}) \\&= AB + \overline{A}C \quad (\text{complement})\end{aligned}$$

Final result: $f = AB + \overline{A}C$ (reduced from 12 to 4 literals!)

Consensus Theorem

Theorem 10 (Consensus Theorem): $XY + \overline{X}Z + YZ = XY + \overline{X}Z$

The term YZ is “absorbed” by the other two terms.

Proof:

$$\begin{aligned} XY + \overline{X}Z + YZ &= XY + \overline{X}Z + (X + \overline{X})YZ && \text{(complement)} \\ &= XY + \overline{X}Z + XYZ + \overline{X}YZ && \text{(distributive)} \\ &= XY(1 + Z) + \overline{X}Z(1 + Y) && \text{(factor)} \\ &= XY + \overline{X}Z && \text{(null: } 1 + X = 1 \text{)} \end{aligned}$$

□

Note: Intuition: If YZ is true, then either XY or $\overline{X}Z$ must be true, so YZ is redundant.

Multi-Level Minimization

Definition 22 (Multi-Level Logic): Instead of two-level SoP/PoS, use multiple levels of gates to share common subexpressions.

Example: Two-level: $f_1 = ABC + ABD$, $f_2 = ABC + ABE$

Total: 12 literals (6 + 6)

Multi-level with factoring:

- $T = AB$ (common factor)
- $f_1 = T(C + D)$
- $f_2 = T(C + E)$

Total: 6 literals + reuse of T

Trade-off: Multi-level uses fewer gates but has longer delay (more levels).

When to Use CNF vs DNF

DNF (Sum of Products):

- Few 1s in truth table
- OR-of-ANDs natural
- Most circuit designs
- Easy synthesis from minterms
- K-map method works well

CNF (Product of Sums):

- Few 0s in truth table
- AND-of-ORs structure
- SAT solver input
- Constraint representation
- Built from maxterms

Example: Consider two functions:

Truth table	SoP cubes	PoS clauses
2 ones, 6 zeros	2 minterms	6 maxterms
6 ones, 2 zeros	6 minterms	2 maxterms

Choose the form with fewer components!

Introduction to Quine-McCluskey

Definition 23: The *Quine-McCluskey algorithm* is a systematic tabular method for finding all prime implicants, guaranteed to produce a minimal form.

The Q-M algorithm has two phases:

1. Generate all prime implicants from minterms
2. Select a minimal set of prime implicants covering all minterms

When to use Q-M:

- More than 4-5 variables (K-maps impractical)
- Need guaranteed minimal solution
- Computer-aided design tools
- Can be automated (unlike K-maps)

Limitation: Complexity grows exponentially with variables — practical for $\leq 6-8$ variables.

Quine-McCluskey Algorithm

The Q-M algorithm has two phases:

Phase 1: Generate Prime Implicants

1. List all minterms in binary
2. Group by number of 1s (Hamming weight)
3. Combine pairs differing in exactly one bit
4. Replace differing bit with dash (-)
5. Repeat until no more combinations
6. Uncombined terms are prime implicants

Phase 2: Select Minimal Cover

1. Build prime implicant chart
2. Find essential prime implicants
3. Use Petrick's method or heuristics for rest

Step-by-Step Example of Q-M Phase 1

Example: Minimize $f(A, B, C) = \sum m(1, 3, 5, 6, 7)$

Step 1: Group minterms by number of 1-bits (Hamming weight).

Group	Minterm	Binary
1	m_1	001
2	m_3	011
	m_5	101
	m_6	110
3	m_7	111

		C	
		0	1
AB	00	0 ₀	1 ₁
	01	0 ₂	1 ₃
	11	1 ₆	1 ₇
	10	0 ₄	1 ₅

Q-M Phase 1: First Combination

Step 2: Try to combine each minterm in group i with each minterm in group $i + 1$.

# of 1s	Minterm	Cube	Size 2 Implicants
1	m_1 ✓	001	$m_1 + m_3 = m_{1,3}$ ✓ 0-1 $m_1 + m_5 = m_{1,5}$ ✓ -01 $m_1 + m_6$ ✗ (differ in 3 positions)
2	m_3 ✓	011	$m_3 + m_7 = m_{3,7}$ ✓ -11
	m_5 ✓	101	$m_5 + m_7 = m_{5,7}$ ✓ 1-1
	m_6 ✓	110	$m_6 + m_7 = m_{6,7}$ ✓ 11-
3	m_7 ✓	111	—

Result: Five size-2 implicants formed: $m_{1,3}$, $m_{1,5}$, $m_{3,7}$, $m_{5,7}$, $m_{6,7}$

Note: Mark all “used” minterms with ✓ and all “unused” with ✗

Q-M Phase 1: Second Combination

Step 3: Try to combine each 2-size implicant with others having dashes in the SAME positions.

Dash pos.	Implicant	Pattern	Size 4 Implicant
pos. 0	$m_{1,5}$ ✓	-01	$m_{1,5} + m_{3,7} = m_{1,3,5,7}$ ✓ --1
	$m_{3,7}$ ✓	-11	—
pos. 1	$m_{1,3}$ ✓	0-1	$m_{1,3} + m_{5,7} = m_{1,3,5,7}$ ✓ --1
	$m_{5,7}$ ✓	1-1	—
pos. 2	$m_{6,7}$ ✗	11-	(no other implicant with dash at pos. 2)

Explanation:

- $m_{1,5}$ (-01) and $m_{3,7}$ (-11): dashes align at pos. 0, differ only in bit 1 $\Rightarrow$ combine to $m_{1,3,5,7}$ (--1) ✓
- $m_{1,3}$ (0-1) and $m_{5,7}$ (1-1): dashes align at pos. 1, differ only in bit 0 $\Rightarrow$ combine to $m_{1,3,5,7}$ (--1) ✓
- $m_{6,7}$ (11-): alone with dash at pos. 2 $\Rightarrow$ cannot combine ✗

Result: One size-4 implicant formed: $m_{1,3,5,7}$ (--1). One size-2 implicant remains uncombined: $m_{6,7}$ (11-).

Q-M Phase 1: Finding Prime Implicants

Step 4: Identify prime implicants (uncombined terms).

From Step 3:

- Size-4 implicant: $m_{1,3,5,7}$ ($--1$) is *prime* since there are no other size-4 implicants to combine with
- Size-2 implicants NOT marked with ✓: $m_{6,7}$ are *prime*

Pattern	Covers	Boolean Term
$--1$	$\{1, 3, 5, 7\}$	C
$11-$	$\{6, 7\}$	AB

These 4 prime implicants cannot be reduced further.

Prime Implicant Chart

Definition 24: A *prime implicant chart* is a table showing which minterms are covered by each prime implicant.

Note: Our goal in Q-M is to select a minimum set of prime implicants covering all minterms.

Example: For our example $\sum m(1, 3, 5, 6, 7)$:

Prime Implicant	m_1	m_3	m_5	m_6	m_7
C ($--1$)	✓	✓	✓		✓
AB ($11-$)				✓	✓

Finding essential prime implicants:

- Look for columns with only ONE ✓ $\Rightarrow$ that PI is **essential**
 - ▶ Columns m_1, m_3, m_5 – only covered by $C \Rightarrow C$ is **essential**
 - ▶ Column m_6 – only covered by $AB \Rightarrow AB$ is **essential**

Prime Implicant Chart [2]

Selecting minimal cover:

- Must include: C (essential), covers $\{1, 3, 5, 7\}$
- Must include: AB (essential), covers $\{6, 7\}$

Minimal solution: $f = C + AB$ (3 literals)

Essential Prime Implicants

Definition 25: An *essential prime implicant* (EPI) is a prime implicant that covers at least one minterm not covered by any other prime implicant.

Example: Consider this prime implicant chart:

	m_i	m_j	m_k
PI ₁	×	×	
PI ₂		×	×
PI ₃			×

Key insight: If a minterm has *only one* ×, the corresponding prime implicant is *essential* and must be included in any minimal cover.

- Minterm m_i is covered only by PI₁ $\Rightarrow$ PI₁ is *essential* $\Rightarrow$ it *must* be included
- Minterm m_j is covered by both PI₁ and PI₂ $\Rightarrow$ no EPI
- Minterm m_k is covered by both PI₂ and PI₃ $\Rightarrow$ no EPI

Conclusion: PI₁ *must* be included. For remaining minterms, choose either PI₂ or PI₃.

Petrick's Method

When multiple prime implicants remain after selecting essentials:

Definition 26: *Petrick's method* finds all combinations of prime implicants that cover the remaining minterms, so you can pick the one with the lowest cost.

1. Express “covering all minterms” as a Boolean formula
2. Each minterm needs at least one of its covering PIs
3. Formula in CNF (product of sums)
4. Convert to DNF to see all possible covers
5. Choose cover with minimum cost

Why it works: The CNF formula is satisfiable if and only if we can cover all minterms. Each satisfying assignment is a valid cover.

Petrick's Method: Example

Example: Prime implicants P_1, P_2, P_3, P_4 cover minterms as follows:

	m_1	m_2	m_3	m_4
P_1	×	×		
P_2	×		×	
P_3		×	×	×
P_4			×	×

Coverage formula (CNF):

- m_1 : $P_1 + P_2$ (needs P_1 OR P_2)
- m_2 : $P_1 + P_3$ (needs P_1 OR P_3)
- m_3 : $P_2 + P_3 + P_4$
- m_4 : $P_3 + P_4$

Formula: $(P_1 + P_2)(P_1 + P_3)(P_2 + P_3 + P_4)(P_3 + P_4)$

Petrick's Method: Solving

Example (*continued*): Expand to DNF:

$$\begin{aligned} & (P_1 + P_2)(P_1 + P_3)(P_2 + P_3 + P_4)(P_3 + P_4) \\ &= (P_1 + P_2P_3)(P_2 + P_3 + P_4)(P_3 + P_4) \\ &= P_1(P_3 + P_4) + P_2P_3(P_3 + P_4) \\ &= P_1P_3 + P_1P_4 + P_2P_3 \end{aligned}$$

Possible covers:

1. P_1P_3 (2 implicants)
2. P_1P_4 (2 implicants)
3. P_2P_3 (2 implicants)

All have same cost! Choose any: $f = P_1 + P_3$ (for example)

Warning: CNF to DNF expansion can explode exponentially!

Quine-McCluskey: Complete Example

Example: Minimize $f(A, B, C, D) = \sum m(0, 1, 2, 5, 6, 7, 8, 9, 10, 14)$

Group	Minterm	Binary	→ Comb 1	→ Comb 2
0	0	0000	0,1: 000–	0,1,8,9: –00–
1	1	0001	0,2: 00–0	0,2,8,10: –0–0
	2	0010	0,8: –000	
	8	1000	1,9: 100–	
2	5	0101	2,6: 0–10	2,6,10,14: ––10
	6	0110	2,10: –010	
	9	1001	5,7: 01–1	
	10	1010	8,9: 100–	
3	7	0111	8,10: 10–0	
	14	1110	6,7: 011–	
			6,14: –110	
			10,14: 1–10	

Q-M Example: Finding Minimal Cover

Prime implicants:

- P_1 : -00- covers $\{0, 1, 8, 9\}$
- P_2 : -0-0 covers $\{0, 2, 8, 10\}$
- P_3 : --10 covers $\{2, 6, 10, 14\}$
- P_4 : 01-1 covers $\{5, 7\}$

Prime implicant chart:

	0	1	2	5	6	7	8	9	10	14
P_1	×	×					×	×		
P_2	×		×				×		×	
P_3			×		×				×	×
P_4				×		×				

- P_4 is essential (only covers $\{5, 7\}$)
- After selecting P_4 , need to cover $\{0, 1, 2, 6, 8, 9, 10, 14\}$

Minimal solution: $f = P_1 + P_3 + P_4$ *or* $f = P_2 + P_3 + P_4$

Complexity of Minimization

Theorem 11: Boolean function minimization is **NP-complete**.

What this means in practice:

- No known polynomial-time algorithm for exact minimization
- As variables increase, exhaustive search becomes impossible
- For n variables: up to 2^{2^n} possible functions!

Example: The scale of the problem:

- 4 variables: $2^{16} = 65536$ functions
- 5 variables: $2^{32} \approx 4.3 \times 10^9$ functions
- 10 variables: 2^{1024} functions (more than atoms in universe!)

Note: For large functions, we use heuristics (ESPRESSO, ABC) instead of exact methods. Modern CAD tools focus on technology mapping and multi-objective optimization (timing, power, area).

Modern Minimization Tools

Industrial-strength tools:

ESPRESSO:

- Heuristic multi-level minimizer
- Handles 100+ variables
- Used in major CAD tools

ABC (Berkeley):

- Academic tool for logic synthesis
- Advanced algorithms for large designs

Commercial tools:

- Synopsys Design Compiler
- Cadence Genus

Modern approach:

- Technology mapping (fit to available gates)
- Timing-driven optimization
- Power minimization
- Multi-objective optimization

Practical Minimization Strategy

Recommended approach by function size:

2-3 variables:

- Use K-maps (instant, visual)
- Or simple algebra

4 variables:

- K-maps work well
- Good for learning and verification

5-6 variables:

- K-maps possible but tedious
- Q-M algorithm or tools

7+ variables:

- Use CAD tools (ESPRESSO, ABC)
- Heuristic methods
- Accept near-optimal solutions

Always:

- Verify by expanding result
- Check against original truth table
- Test edge cases

Summary: Minimization Techniques

What we've mastered:

1. **Why minimize:** Cost, power, speed, clarity.
2. **Gray code:** Foundation for K-map adjacency.
3. **K-maps:** Visual method for 2-5 variables.
 - Grouping rules and strategies
 - Wrap-around and corners
 - Don't-care optimization
4. **Algebraic methods:** Laws, consensus, multi-level, factoring.
5. **Quine-McCluskey:**
 - Systematic prime implicant generation
 - Prime implicant chart
 - Essential implicants
6. **Petrick's method:** Minimum cover selection.
7. **Minimization complexity:** NP-complete.
8. **Practical tools:** ESPRESSO, ABC.

Summary: Minimization Techniques [2]

Key insight:

Minimization transforms algebraic simplification into visual pattern recognition (K-maps) or systematic algorithms (Q-M).

For small functions (≤ 5 vars), K-maps are instant.

For large functions, we use heuristics and accept near-optimal solutions.

What's next:

All our techniques (DNF, CNF, minimization) use AND/OR/NOT.

But there's a completely different approach using only XOR and AND — Algebraic Normal Form (ANF). This gives us *unique* canonical representation, crucial for cryptography.

Algebraic Normal Form

Motivation: A Different Representation

All our normal forms (DNF, CNF) use AND, OR, NOT. They work great for circuits but have redundancy: the same function has infinitely many equivalent representations.

Example: $f = (x \wedge y) \vee (x \wedge \neg y) = x \wedge (y \vee \neg y) = x \wedge 1 = x$

For cryptography and formal verification, we need a *unique* canonical form.

ANF (Algebraic Normal Form): Use only XOR ($\oplus$) and AND ($\wedge$).

Result: Every Boolean function has a *unique* polynomial representation over $\text{GF}(2)$.

Note: $\bar{x} = x \oplus 1$, so NOT is not needed.

Why ANF? Applications and Advantages

So far, we've built Boolean functions using AND ($\wedge$), OR ($\vee$), and NOT ($\bar{x}$).

This led us to:

- DNF and CNF (canonical forms)
- K-maps and Quine-McCluskey (minimization)
- Circuit design with standard gates

Question: Can we represent Boolean functions using only AND ($\wedge$) and XOR ($\oplus$)?

That is, without OR and NOT operations?

The answer is yes, and this leads to a completely different algebraic structure with important theoretical and practical applications.

Why ANF matters:

- **Cryptography:** Analyze security of encryption algorithms (AES S-boxes)

Why ANF? Applications and Advantages [2]

- **Coding theory:** Design error-correcting codes (Reed-Muller codes)
- **Formal verification:** Unique canonical form simplifies equivalence checking
- **Quantum computing:** XOR gates are easier to implement than OR gates

Introducing Algebraic Normal Form

Definition 27: An *Algebraic Normal Form* (ANF), also called a *Zhegalkin polynomial*, represents a Boolean function as a polynomial over $\mathbb{F}_2$ (the field $\{0, 1\}$) using:

- XOR ($\oplus$) for addition (modulo 2)
- AND ($\wedge$) for multiplication
- Constants 0 and 1

Example: The function $f(x, y, z) = xy \oplus xz \oplus yz \oplus x \oplus 1$ is in ANF.

Term structure:

- xy, xz, yz are degree-2 terms (quadratic)
- x is a degree-1 term (linear)
- 1 is a degree-0 term (constant)

The algebraic degree of this function is 2.

Notation: We write XOR as $\oplus$ or $+$, and AND as $\wedge$, $\cdot$, or by juxtaposition (xy).

Properties of XOR

Property	OR	XOR
Characteristic	Idempotent $x \vee x = x$	Self-inverse $x \oplus x = 0$
Negation	Requires NOT	$\bar{x} = x \oplus 1$

These differences enable important applications:

Cryptography

- Stream cipher design
- S-box analysis (DES, AES)
- Algebraic attacks
- Linear cryptanalysis resistance

Coding Theory

- Error-correcting codes
- Reed-Muller codes
- Parity check matrices
- LDPC codes

Note: In $\mathbb{F}_2$, XOR corresponds to addition and AND to multiplication.
ANF treats Boolean functions as polynomials over this field.

ANF Structure

Example: The function $f(x, y, z) = xy \oplus xz \oplus yz \oplus x \oplus 1$ is in ANF.

Terms:

- xy — degree 2 (product of 2 variables)
- xz — degree 2
- yz — degree 2
- x — degree 1 (linear term)
- 1 — degree 0 (constant)

The function has algebraic degree 2.

Note: In ANF, we write addition (XOR) as $\oplus$ or simply $+$.

Multiplication (AND) can be written as $\wedge$, $\cdot$, or juxtaposition (*e.g.*, xy).

Comparing ANF and DNF

Property	DNF	ANF
Operations	AND, OR, NOT	AND, XOR only
Structure	OR of AND-terms	XOR of AND-terms
Example	$xy \vee \bar{x}z$	$xy \oplus xz \oplus 1$
Canonical form	Yes (via minterms)	Yes (unique)
Representations	Many variants possible	Exactly one
Negation	Explicit ($\bar{x}$)	$x \oplus 1$
$x + x =$	x (idempotent)	0 (self-inverse)
Identity element	$x \vee 0 = x$	$x \oplus 0 = x$
Degree notion	Not applicable	Max monomial size
Primary use	Circuit synthesis	Cryptographic analysis

Uniqueness of ANF Representation

Theorem 12: Every Boolean function $f : \{0, 1\}^n \rightarrow \{0, 1\}$ has *exactly one* ANF representation.

Proof: Consider Boolean functions as a vector space over $\mathbb{F}_2$:

1. There are 2^{2^n} Boolean functions on n variables
2. There are 2^n possible monomials, giving 2^{2^n} possible ANFs
3. The mapping $\text{ANF} \rightarrow \text{Boolean function}$ is a linear bijection over $\mathbb{F}_2$

Therefore, each function has exactly one ANF representation. □

Significance: Unlike DNF/CNF (which have multiple equivalent forms), ANF is always canonical.

This eliminates the need for minimization in the ANF context.

Note: The uniqueness follows from the linear algebra structure over $\mathbb{F}_2$.

The Möbius transform (used in Pascal's triangle method) provides an explicit bijection.

Monomials in ANF

Example: For $n = 3$ variables (x, y, z) :

Degree	Count	Monomials
0	1	1
1	3	x, y, z
2	3	xy, xz, yz
3	1	xyz

Total: $2^3 = 8$ possible monomials (including constant 1).

Each monomial either appears in ANF or doesn't $\Rightarrow 2^8 = 256$ possible ANFs for 3 variables.

Note: For n variables, there are $\sum_{k=0}^n \binom{n}{k} = 2^n$ possible monomials.

Algebraic Degree

Definition 28: The *algebraic degree* of a Boolean function in ANF is the maximum number of variables in any monomial with coefficient 1.

Example:

Function	ANF	Degree
NOT	$f(x) = x \oplus 1$	1
XOR	$f(x, y) = x \oplus y$	1
AND	$f(x, y) = xy$	2
NAND	$f(x, y) = xy \oplus 1$	2
Majority	$f(x, y, z) = xy \oplus xz \oplus yz$	2
Full ANF	$f(x, y, z) = xyz \oplus xy \oplus ... \oplus 1$	3

Why Algebraic Degree Matters

Weak: Degree 2-3

$$f(x, y, z) = x \oplus yz$$

Linearization attack:

1. Set $w = yz$
2. Now: $f = x \oplus w$ (linear!)
3. Solve system $\Rightarrow$ broken! ✗

Strong: Degree 7-8

AES S-box: degree 7

Many nonlinear terms

Linearization $\Rightarrow$ huge system

Computationally infeasible ✓

Cryptography lesson: Low degree = vulnerable to algebraic attacks!

Modern ciphers (AES, SHA-256) use maximum practical degree.

Methods for Computing ANF

We will examine three methods for computing ANF from truth tables:

Method	Complexity	Variables	Difficulty	Best suited for
Direct computation	$O(2^{2n})$	1-3	High	Theoretical understanding
Pascal's triangle	$O(2^n)$	1-5	Medium	Hand calculations
Karnaugh map	$O(2^n)$	2-4	Low	Visual intuition

Learning approach:

1. Begin with K-map method (most intuitive)
2. Master Pascal's triangle (most practical)
3. Understand direct method (theoretical foundation)

Each method provides different insights into the structure of ANF.

Method 1: Direct Computation

Idea: Solve a system of linear equations over $\mathbb{F}_2$ (arithmetic mod 2).

For function $f : \mathbb{F}_2^n \rightarrow \mathbb{F}_2$, we want to find coefficients a_S such that:

$$f(x_1, \dots, x_n) = \bigoplus_{S \subseteq \{1, \dots, n\}} \left(a_S \prod_{i \in S} x_i \right)$$

where $\oplus$ denotes XOR and $\prod$ denotes AND.

Each row of the truth table gives one linear equation!

Remember: In $\mathbb{F}_2$, addition is XOR:

- $1 \oplus 1 = 0 \oplus 0 = 0$
- $1 \oplus 0 = 0 \oplus 1 = 1$

Direct Computation: Example (Setup)

Example: Find ANF for $f(x, y)$ with truth table:

x	y	$f(x, y)$
0	0	1
0	1	1
1	0	1
1	1	0

Let ANF be: $f(x, y) = a_0 \oplus a_1x \oplus a_2y \oplus a_3xy$

From truth table, substitute each (x, y) to get 4 equations over $\mathbb{F}_2$:

(x, y)	Substituted equation	f
(0,0)	$a_0 \oplus (a_1 \cdot 0) \oplus (a_2 \cdot 0) \oplus (a_3 \cdot 0 \cdot 0) = a_0$	= 1
(0,1)	$a_0 \oplus (a_1 \cdot 0) \oplus (a_2 \cdot 1) \oplus (a_3 \cdot 0 \cdot 1) = a_0 \oplus a_2$	= 1

Direct Computation: Example (Setup) [2]

(x, y)	Substituted equation	f
(1,0)	$a_0 \oplus (a_1 \cdot 1) \oplus (a_2 \cdot 0) \oplus (a_3 \cdot 1 \cdot 0) = a_0 \oplus a_1$	$= 1$
(1,1)	$a_0 \oplus a_1 \oplus a_2 \oplus a_3$	$= 0$

Direct Computation: Solution

Solving the system step-by-step:

- Equation 1: $a_0 = 1$
- Equation 2: $(a_0 \oplus a_2 = 1) \Rightarrow (1 \oplus a_2 = 1) \Rightarrow a_2 = 0$
- Equation 3: $(a_0 \oplus a_1 = 1) \Rightarrow (1 \oplus a_1 = 1) \Rightarrow a_1 = 0$
- Equation 4: $(1 \oplus 0 \oplus 0 \oplus a_3 = 0) \Rightarrow (1 \oplus a_3 = 0) \Rightarrow a_3 = 1$

Result: $f(x, y) = 1 \oplus xy$, which is equivalent to $f = x \overline{\wedge} y$.

Each row in the truth table gives one linear equation over $\mathbb{F}_2$.

With n variables: 2^n equations in 2^n unknowns (ANF coefficients).

System is always solvable, but for $n > 4$ becomes impractical manually!

Method 2: Pascal's Triangle Method

Fast systematic method using a butterfly/pyramid pattern!

Algorithm:

1. Write function values f in a column (in binary order: 000, 001, 010, ...)
2. Create next column: XOR each adjacent pair ($f_i \oplus f_{i+1}$)
3. Repeat step 2 until only one value remains
4. The **first value** of each column is an ANF coefficient

Note: Also called the *Butterfly method* or *ANF transform*.

Pattern: Coefficients appear in binary order: 1, z , y , yz , x , xz , xy , xyz

Pascal's Triangle Method: Example

Find ANF for $f(x, y, z) = \sum m(0, 1, 3, 7)$:

xyz	f	a_1	a_z	a_y	a_{yz}	a_x	a_{xz}	a_{xy}	a_{xyz}
000	1	1	0	1	1	1	0	1	0
001	1	1	1	0	0	1	1	1	
010	0	0	1	0	1	0	0		
011	1	1	1	1	1	0			
100	0	0	0	0	1				
101	0	0	0	1					
110	0	0	1						
111	1	1							

ANF: $f = 1 \oplus y \oplus yz \oplus x \oplus xy$

Pascal's Triangle: Why It Works

The transformation computes the *Möbius transform* over the Boolean lattice:

$$a_S = \bigoplus_{T \subseteq S} f(T)$$

Each coefficient a_S is the XOR of all function values $f(T)$ where $T \subseteq S$.

Example: For a_{xy} , corresponding to input $110 = \{x, y\}$:

$$\begin{aligned} a_{xy} &= f(000) \oplus f(010) \oplus f(100) \oplus f(110) \\ &= 1 \oplus 0 \oplus 0 \oplus 0 = 1 \end{aligned}$$

Historical note: Related to Reed-Muller codes and Fast Walsh-Hadamard Transform!

Method 3: Karnaugh Map Method

Idea: Process K-map cells by *Hamming weight* (degree). For each cell with value 1, add corresponding monomial to ANF and invert its region.

Algorithm:

1. Fill K-map with function values
2. Order cells by Hamming weight: e.g., 000 $\rightarrow$ 001, 010, 100 $\rightarrow$ 011, 101, 110 $\rightarrow$ 111
3. For each cell in order:
 - If value = 0: skip
 - If value = 1: add monomial, invert its region (where monomial = 1)
4. Result: ANF = XOR of added monomials

Key: Processing by weight isolates each term's contribution via successive inversions.

Example: $f(x, y) = x \oplus y$ – Initial State

Example: Find ANF for $f(x, y) = x \oplus y = \sum m(1, 2)$

	$y = 0 \quad y = 1$	
$x = 0$	0	1
$x = 1$	1	0

Processing order:

1. $(0, 0)$ – weight 0
2. $(1, 0)$ – weight 1
3. $(0, 1)$ – weight 1
4. $(1, 1)$ – weight 2

Current ANF: (empty)

Step 1: Process (0, 0) – Constant

	$y = 0$	$y = 1$
$x = 0$	0	1
$x = 1$	1	0

Cell (0, 0) = 0
Monomial: 1
Region: all cells

Skip

ANF: (empty)

	$y = 0$	$y = 1$
$x = 0$	0	1
$x = 1$	1	0

Step 2: Process (1, 0) — Variable x

	$y = 0$	$y = 1$
$x = 0$	0	1
$x = 1$	1	0

Cell (1, 0) = 1

Monomial: x

Region: $x = 1$

Add x



Invert region $x = 1$

ANF: x

	$y = 0$	$y = 1$
$x = 0$	0	1
$x = 1$	0	1

Step 3: Process (0, 1) – Variable y

	$y = 0$	$y = 1$
$x = 0$	0	1
$x = 1$	0	1

Cell (0, 1) = 1

Monomial: y

Region: $y = 1$

Add y



Invert region $y = 1$

ANF: $x \oplus y$

	$y = 0$	$y = 1$
$x = 0$	0	0
$x = 1$	0	0

Step 4: Process (1, 1) – Product xy

	$y = 0$	$y = 1$
$x = 0$	0	0
$x = 1$	0	0

Cell (1, 1) = 0
Monomial: xy
Region: one cell

Skip

✓ **Done!**

ANF: $x \oplus y$

	$y = 0$	$y = 1$
$x = 0$	0	0
$x = 1$	0	0

Example 2: Three Variables — Initial

Example: Find ANF for $f(x, y, z) = \sum m(3, 5, 6, 7)$

	$z = 0$	$z = 1$
$x = 0, y = 0$	0	0
$x = 0, y = 1$	0	1
$x = 1, y = 1$	1	1
$x = 1, y = 0$	0	1

Processing order:

Weight 0: (0, 0, 0)

Weight 1: (1, 0, 0), (0, 1, 0), (0, 0, 1)

Weight 2: (1, 1, 0), (1, 0, 1), (0, 1, 1)

Weight 3: (1, 1, 1)

ANF: (empty)

Step 1: $(0, 0, 0)$ – Constant 1

	$z = 0$	$z = 1$
$x = 0, y = 0$	0	0
$x = 0, y = 1$	0	1
$x = 1, y = 1$	1	1
$x = 1, y = 0$	0	1

Cell $(0, 0, 0) = 0$

Monomial: 1

Region: all cells

Skip

ANF: (empty)

	$z = 0$	$z = 1$
$x = 0, y = 0$	0	0
$x = 0, y = 1$	0	1
$x = 1, y = 1$	1	1
$x = 1, y = 0$	0	1

Step 2: $(1, 0, 0)$ – Variable x

	$z = 0$	$z = 1$
$x = 0, y = 0$	0	0
$x = 0, y = 1$	0	1
$x = 1, y = 1$	1	1
$x = 1, y = 0$	0	1

Cell $(1, 0, 0) = 0$

Monomial: x

Region: $x = 1$

Skip

ANF: (empty)

	$z = 0$	$z = 1$
$x = 0, y = 0$	0	0
$x = 0, y = 1$	0	1
$x = 1, y = 1$	1	1
$x = 1, y = 0$	0	1

Step 3: $(0, 1, 0)$ – Variable y

	$z = 0$	$z = 1$
$x = 0, y = 0$	0	0
$x = 0, y = 1$	0	1
$x = 1, y = 1$	1	1
$x = 1, y = 0$	0	1

Cell $(0, 1, 0) = 0$

Monomial: y

Region: $y = 1$

Skip

ANF: (empty)

	$z = 0$	$z = 1$
$x = 0, y = 0$	0	0
$x = 0, y = 1$	0	1
$x = 1, y = 1$	1	1
$x = 1, y = 0$	0	1

Step 4: $(0, 0, 1)$ – Variable z

	$z = 0$	$z = 1$
$x = 0, y = 0$	0	0
$x = 0, y = 1$	0	1
$x = 1, y = 1$	1	1
$x = 1, y = 0$	0	1

Cell $(0, 0, 1) = 0$

Monomial: z

Region: $z = 1$

Skip

ANF: (empty)

	$z = 0$	$z = 1$
$x = 0, y = 0$	0	0
$x = 0, y = 1$	0	1
$x = 1, y = 1$	1	1
$x = 1, y = 0$	0	1

Step 5: $(1, 1, 0)$ – Product xy

	$z = 0$	$z = 1$
$x = 0, y = 0$	0	0
$x = 0, y = 1$	0	1
$x = 1, y = 1$	1	1
$x = 1, y = 0$	0	1

Cell $(1, 1, 0) = 1$
Monomial: xy
Region: $x = 1, y = 1$

Add xy



Invert region

ANF: xy

	$z = 0$	$z = 1$
$x = 0, y = 0$	0	0
$x = 0, y = 1$	0	1
$x = 1, y = 1$	0	0
$x = 1, y = 0$	0	1

Step 6: $(1, 0, 1)$ – Product xz

	$z = 0$	$z = 1$
$x = 0, y = 0$	0	0
$x = 0, y = 1$	0	1
$x = 1, y = 1$	0	0
$x = 1, y = 0$	0	1

Cell $(1, 0, 1) = 1$

Monomial: xz

Region: $x = 1, z = 1$

Add xz



Invert region

ANF: $xy \oplus xz$

	$z = 0$	$z = 1$
$x = 0, y = 0$	0	0
$x = 0, y = 1$	0	1
$x = 1, y = 1$	0	1
$x = 1, y = 0$	0	0

Step 7: $(0, 1, 1)$ – Product yz

	$z = 0$	$z = 1$
$x = 0, y = 0$	0	0
$x = 0, y = 1$	0	1
$x = 1, y = 1$	0	1
$x = 1, y = 0$	0	0

Cell $(0, 1, 1) = 1$
Monomial: yz
Region: $y = 1, z = 1$

Add yz



Invert region

ANF: $xy \oplus xz \oplus yz$

	$z = 0$	$z = 1$
$x = 0, y = 0$	0	0
$x = 0, y = 1$	0	0
$x = 1, y = 1$	0	0
$x = 1, y = 0$	0	0

Step 8: (1, 1, 1) – Product xyz

	$z = 0$	$z = 1$
$x = 0, y = 0$	0	0
$x = 0, y = 1$	0	0
$x = 1, y = 1$	0	0
$x = 1, y = 0$	0	0

Cell (1, 1, 1) = 0

Monomial: xyz

Region: one cell

Skip

✓ **Done!**

ANF: $xy \oplus xz \oplus yz$

	$z = 0$	$z = 1$
$x = 0, y = 0$	0	0
$x = 0, y = 1$	0	0
$x = 1, y = 1$	0	0
$x = 1, y = 0$	0	0

Final Result

	$z = 0$	$z = 1$
$x = 0, y = 0$	0	0
$x = 0, y = 1$	0	0
$x = 1, y = 1$	0	0
$x = 1, y = 0$	0	0

All zeros $\Rightarrow$ complete!

ANF: $xy \oplus xz \oplus yz$

Verification:

$$f(0, 1, 1) = 0 \oplus 0 \oplus 1 = 1 \quad \checkmark$$

$$f(1, 0, 1) = 0 \oplus 1 \oplus 0 = 1 \quad \checkmark$$

$$f(1, 1, 0) = 1 \oplus 0 \oplus 0 = 1 \quad \checkmark$$

$$f(1, 1, 1) = 1 \oplus 1 \oplus 1 = 1 \quad \checkmark$$

Matches $\sum m(3, 5, 6, 7) \quad \checkmark$

K-map Method: Summary

Algorithm:

1. **Order:** Process cells by Hamming weight ($0 \rightarrow 1 \rightarrow 2 \rightarrow \dots \rightarrow n$)
2. **Decision:** For each cell:
 - Value = 0 $\Rightarrow$ Skip
 - Value = 1 $\Rightarrow$ Add monomial, invert its region
3. **Region:** All cells where the monomial equals 1
4. **Result:** ANF = XOR of all added monomials

Key: Processing by weight ensures each monomial's contribution is isolated via XOR over $\mathbb{F}_2$.

Reference: [Wikipedia: Zhegalkin polynomial § Using a Karnaugh map](#)

Comparison of ANF Construction Methods

Method	Pros	Cons	Best for
Direct computation	• Systematic • <i>Always works</i> • Clear math	• Tedious • Many equations • Error-prone	1-2 variables, learning
Pascal's triangle	• <i>Fast</i> • Mechanical • Scales well	• Less intuitive • Needs practice • Easy to miscount	3-5 variables, computation
K-map method	• Visual • <i>Intuitive</i> • Good for patterns	• Only 2-4 variables • Can miss groups • Needs care	2-4 variables, understanding

Recommendation: Learn all three! Use Pascal's triangle for exams, K-map for intuition.

Summary of ANF Properties

Key properties of ANF:

1. **Uniqueness:** Every Boolean function has exactly one ANF representation
2. **Completeness:** ANF can represent any Boolean function
3. **Operations:** Uses only XOR ($\oplus$) and AND — no NOT needed
4. **Algebraic degree:** Maximum degree of monomials (crucial for cryptography)
5. **Self-inverse:** $x \oplus x = 0$ (unlike $x \vee x = x$)
6. **Linearity:** XOR is linear over $\mathbb{F}_2$, making polynomial algebra work

Key insight: ANF trades circuit efficiency (uses more gates than DNF/CNF) for mathematical uniqueness. In cryptography, the algebraic degree in ANF is a direct measure of security against algebraic attacks.

Note: Computational efficiency: Pascal's triangle method is the fastest way to compute ANF from truth tables. K-map method is more intuitive but limited to small functions.

ANF in Cryptography: Why It Matters

Why cryptographers care about ANF?

ANF reveals the *algebraic structure* of cryptographic functions. This structure determines vulnerability to algebraic attacks!

Algebraic Attacks

Express cipher as polynomial system over $\mathbb{F}_2$:

- Each round forms polynomial equations
- Solve using Gröbner basis algorithms
- Low degree enables efficient attacks

Note: If an S-box has degree 2, attackers can linearize the system and solve in polynomial time.

S-Box Design Criteria

Secure S-boxes require:

- High algebraic degree (nonlinearity)
- Balanced output distribution
- Absence of linear structures

Note: The AES S-box has degree 7 (near maximum of 8), providing resistance to algebraic attacks.

ANF in Cryptographic Primitives

Stream Cipher Design

Boolean functions in stream ciphers:

- Filter functions for LFSRs
- Combining functions
- Must resist correlation and algebraic attacks

Degree requirements depend on cipher structure.

Important Function Classes

- **Bent functions:** Maximum nonlinearity
- **Balanced functions:** Equal 0s and 1s in output
- **Correlation-immune:** Statistical attack resistance
- **Resilient:** Balanced and correlation-immune

Case study: The E0 stream cipher (used in Bluetooth) was cryptanalyzed due to insufficient algebraic degree in its combining function.

Converting Between ANF and DNF

Two approaches for converting between representations:

Method	Advantages	Disadvantages
Algebraic Direct application of identities	• Direct transformation • Shows structural relationships	• Complex algebraic expansions • Error-prone for large functions • Exponential term growth
Via truth table Truth table as intermediate form	• Systematic procedure • Always succeeds • Clear verification steps	• Requires complete enumeration • Additional computational step

Recommended approach: Use truth table as intermediate representation.

This method is systematic, reliable, and suitable for examination contexts.

Note: Direct algebraic conversion requires expanding $x \oplus y = (x \wedge \bar{y}) \vee (\bar{x} \wedge y)$, which becomes complex for multi-variable functions.

Converting ANF to DNF

Example: Convert $f = x \oplus y \oplus xy$ (ANF) to DNF

Step 1: Build truth table by evaluating ANF:

x	y	$f = x \oplus y \oplus xy$	Minterm
0	0	$0 \oplus 0 \oplus 0 = 0$	—
0	1	$0 \oplus 1 \oplus 0 = 1$	$\bar{x}y$
1	0	$1 \oplus 0 \oplus 0 = 1$	$x\bar{y}$
1	1	$1 \oplus 1 \oplus 1 = 1$	xy

Step 2: Read minterms where $f = 1$:

DNF: $f = \bar{x}y \vee x\bar{y} \vee xy = \sum m(1, 2, 3)$

Converting DNF to ANF

Example: Convert $f = \bar{x}y \vee x\bar{y}$ (DNF) to ANF

Step 1: Build truth table by evaluating DNF

Step 2: Apply Pascal's triangle method to get ANF:

xy	f	a_1	a_x	a_y	a_{xy}
00	0	0	1	1	0
01	1	1	0	1	
10	1	1	1		
11	0	0			

ANF: $f = x \oplus y$ (the XOR function!)

Summary: Algebraic Normal Form

Core concepts:

- ANF uses only XOR and AND (no OR, no NOT)
- Every function has *unique* ANF representation
- Algebraic degree = max monomial size
- Monomials: products of variables ($1, x, xy, xyz, \dots$)
- XOR is self-inverse: $x \oplus x = 0$
- Negation via XOR: $\bar{x} = x \oplus 1$
- Linear functions: degree ≤ 1 (no AND terms)

ANF construction methods:

- Pascal's triangle (systematic, $O(2^n)$)
- K-map regions (visual, intuitive)
- Direct computation (linear system)

Applications:

- Cryptanalysis of S-boxes
- Algebraic attacks on ciphers
- Reed-Muller codes

What's next: We've seen many ways to express Boolean functions, e.g., using $\{\wedge, \vee, \neg\}$ or $\{\oplus, \wedge, 1\}$.

Which sets of operations can express *all* Boolean functions? This leads us to *functional completeness*.

Functional Completeness

The Fundamental Question

We've seen many Boolean operations: AND, OR, NOT, XOR, NAND, NOR, implication, equivalence...

The million-dollar question:

Imagine you're designing a new computer chip. Manufacturing costs depend on how many *different types* of gates you need. Each gate type requires different masks, testing procedures, and inventory.

Question: What's the *minimum* set of operations needed to build *everything*?

Can we manufacture chips with just ONE type of gate? Or do we absolutely need three types (AND, OR, NOT)?

This isn't just theoretical — Intel, AMD, and ARM face this question for every new processor!

Central question: Which sets of operations can express *every* Boolean function?

For example:

- Can we build everything using only $\{\wedge, \vee, \neg\}$? ✓

The Fundamental Question [2]

- Can we build everything using only $\{\text{NAND}\}$? ✓ (just one gate type!)
- Can we build everything using only $\{\wedge, \vee\}$? ✗ (can't make NOT)
- Can we build everything using only $\{\oplus\}$? ✗ (only linear functions)

Note: Post's theorem (1941): There are exactly FIVE “weaknesses” a set can have. If a set avoids all five, it's functionally complete. Having even one weakness means some functions cannot be expressed.

Functional Closure

Definition 29: Let F be a set of Boolean functions. The *functional closure* $[F]$ is the smallest set containing F and closed under:

1. **Composition:** If $f, g \in [F]$, then $h(x) = f(g(x)) \in [F]$
2. **Identification:** If $f(x_1, \dots, x_n) \in [F]$, then $f(x, x, x_3, \dots, x_n) \in [F]$
3. **Permutation:** If $f(x_1, \dots, x_n) \in [F]$, then $f(x_{\sigma(1)}, \dots, x_{\sigma(n)}) \in [F]$ for any permutation σ
4. **Introduction of fictitious variables:** If $f(x_1, \dots, x_n) \in [F]$, then $f(x_1, \dots, x_n, x_{n+1}) \in [F]$

Informally, $[F]$ contains all functions you can build by *combining* functions from F .

Note: The closure operations keep the important structure while allowing you combine functions freely.

Functional Completeness

Definition 30: A set F of Boolean functions is *functionally complete* (or *universal*) if its closure $[F]$ contains all possible Boolean functions.

Equivalently: every Boolean function can be expressed using only operations from F .

Example: **Complete sets:**

- $\{\wedge, \vee, \neg\}$ – the classical basis
- $\{\wedge, \neg\}$ – AND with negation
- $\{\vee, \neg\}$ – OR with negation
- $\{\text{NAND}\}$ – NAND alone (Sheffer stroke)
- $\{\text{NOR}\}$ – NOR alone (Peirce arrow)

Incomplete sets:

- $\{\wedge, \vee\}$ – cannot produce NOT
- $\{\oplus\}$ – only produces linear functions
- $\{\neg\}$ – only produces constants and negations

Why Functional Completeness Matters

Circuit Design

Functionally complete sets determine which gates are sufficient to build any digital circuit.

Note: NAND and NOR gates are universal — entire processors can be built using only NAND gates!

Note: Practical constraint: Manufacturing circuits with a single gate type (NAND or NOR) simplifies design, testing, and production.

Programming Languages

Programming languages must provide functionally complete sets of operators.

Example: C provides: `&&`, `||`, `!`, `^`
Each subset `{&&, !}` or `{||, !}` is complete.

Post's Five Classes

Definition 31: A Boolean function $f : \{0, 1\}^n \rightarrow \{0, 1\}$ belongs to class:

1. T_0 (**preserves 0**): $f(0, 0, \dots, 0) = 0$
2. T_1 (**preserves 1**): $f(1, 1, \dots, 1) = 1$
3. S (**self-dual**): $f(\bar{x}_1, \dots, \bar{x}_n) = \overline{f(x_1, \dots, x_n)}$
4. M (**monotone**): $x \leq y \Rightarrow f(x) \leq f(y)$ (bitwise comparison)
5. L (**linear**): f has ANF of degree ≤ 1 (no AND terms)

Note: These classes are *closed under composition*: combining functions from a class stays within that class. If all operations in F belong to one class, $[F]$ cannot contain all Boolean functions.

Post's Classes: Examples

Function	T_0	T_1	S	M	L
Constant 0	✓	✗	✗	✓	✓
Constant 1	✗	✓	✗	✓	✓
Identity x	✓	✓	✓	✓	✓
NOT $\bar{x}$	✗	✗	✓	✗	✓
AND $x \wedge y$	✓	✓	✗	✓	✗
OR $x \vee y$	✓	✓	✗	✓	✗
XOR $x \oplus y$	✓	✗	✗	✗	✓
NAND $\overline{x \wedge y}$	✗	✗	✗	✗	✗
NOR $\overline{x \vee y}$	✗	✗	✗	✗	✗
Implication $x \rightarrow y$	✗	✓	✗	✗	✗

Note: NAND and NOR belong to *none* of the five classes — this is why they are universal.

Understanding Post's Classes

T_0 and T_1 (Constants)

T_0 : maps all-0s to 0; T_1 : maps all-1s to 1

- AND, OR: both T_0 and T_1
- XOR: T_0 only
- NOT: neither

M (Monotone)

$$x \leq y \Rightarrow f(x) \leq f(y)$$

- AND, OR: monotone
- NOT, XOR: not monotone

L (Linear)

$$\text{ANF degree} \leq 1: f = a_0 \oplus a_1 x_1 \oplus \cdots \oplus a_n x_n$$

- Constants, NOT, XOR: linear
- AND, OR: not linear

S (Self-Dual)

$$f(\bar{x}) = \overline{f(x)}$$

- NOT: self-dual
- Median: $xy + yz + xz$

Note: Each class is closed under composition. If $F \subseteq C$ for some class C , then $[F] \subseteq C$, preventing completeness.

Post's Criterion

Theorem 13: A set F of Boolean functions is functionally complete if and only if F contains at least one function that does *not* belong to each of the five Post classes:

$$F \text{ is complete} \iff \begin{cases} \exists f \in F : f \notin T_0 \\ \exists f \in F : f \notin T_1 \\ \exists f \in F : f \notin S \\ \exists f \in F : f \notin M \\ \exists f \in F : f \notin L \end{cases}$$

Note: Each f can be different, or the *same* function can escape multiple classes.

Equivalently: F is complete $\iff F$ is not contained in any Post class.

Note: Practical test: To prove F is complete, find five functions (possibly the same) escaping each class.

Post's Criterion: Intuition

Each Post class represents a “weakness”:

- T_0 : Cannot create constant 1 from all-0 input
- T_1 : Cannot create constant 0 from all-1 input
- S : Cannot break symmetry between x and $\bar{x}$
- M : Cannot decrease output when input increases
- L : Cannot create nonlinear interactions (AND-like)

Escaping all five weaknesses provides power to build any function.

Post's Criterion: Proof Sketch

The full proof is technical, but the key steps are:

Step 1: Show that if $F \subseteq T_0$, then $[F] \subseteq T_0$ (similarly for T_1, S, M, L).

Proof: Composition preserves each class. For example, if $f, g \in T_0$, then:

$$h(0, \dots, 0) = f(g(0, \dots, 0), \dots) = f(0, \dots, 0) = 0$$

Step 2: If F escapes all five classes, construct $\{\wedge, \neg\}$ from F .

This involves:

- Using non- T_0 and non- T_1 to create constants
- Using non- S to break self-duality
- Using non- M to enable decreasing behavior
- Using non- L to create nonlinear terms

Step 3: Since $\{\wedge, \neg\}$ is complete, $[F]$ contains all functions. ■

Verifying Completeness: Positive Examples

Example: Is $\{\wedge, \vee, \neg\}$ complete?

Check each class:

- NOT $\notin T_0$: NOT(0) = 1 $\neq$ 0 ✓
- NOT $\notin T_1$: NOT(1) = 0 $\neq$ 1 ✓
- AND $\notin S$: non-self-dual ✓
- NOT $\notin M$: NOT is not monotone ✓
- AND $\notin L$: AND has degree 2 ✓

Conclusion: Complete! ✓

Example: Is $\{\text{NAND}\}$ complete?

NAND(x, y) = $\overline{x \wedge y}$:

- NAND(0,0) = 1 $\neq$ 0 $\Rightarrow \notin T_0$ ✓
- NAND(1,1) = 0 $\neq$ 1 $\Rightarrow \notin T_1$ ✓
- NAND($\overline{x}$, $\overline{y}$) $\neq \overline{\text{NAND}(x, y)}$ $\Rightarrow \notin S$ ✓
- NAND(0,1) = 1 $>$ NAND(1,1) = 0 $\Rightarrow \notin M$ ✓
- ANF has degree 2 $\Rightarrow \notin L$ ✓

Conclusion: Complete! ✓

Incomplete Sets: Examples

Example: Is $\{\wedge, \vee\}$ complete?

- $\text{AND} \in T_0$: $\text{AND}(0, 0) = 0$
- $\text{OR} \in T_0$: $\text{OR}(0, 0) = 0$
- All functions in closure stay in T_0
- Cannot escape T_0 (need function with $f(0, \dots, 0) = 1$)

Conclusion: Incomplete. Stuck in T_0 . ✗

Example: Is $\{\oplus, \neg\}$ complete?

- $\text{XOR} \in L$: degree 1
- $\text{NOT} \in L$: $\neg x = x \oplus 1$, degree 1
- All functions in closure stay in L
- Cannot escape L (need nonlinear function like AND)

Conclusion: Incomplete. Stuck in L . ✗

Complete Sets in Practice

Common functionally complete sets:

Set	Applications	Notes
$\{\wedge, \vee, \neg\}$	Textbooks, theory	Classical basis
$\{\wedge, \neg\}$	TTL logic families	Two-level logic
$\{\vee, \neg\}$	Alternative basis	Dual to {AND, NOT}
$\{\neg, \rightarrow\}$	Logic programming	Implication-based
$\{\text{NAND}\}$	Digital circuits	Single gate type
$\{\text{NOR}\}$	Digital circuits	Single gate type
$\{\oplus, \wedge, 1\}$	ANF synthesis	Algebraic forms

Note: Design principle: Minimal complete sets reduce hardware complexity and manufacturing costs.

Definability and Applications

Definition 32: A function f is *definable* from set F if $f \in [F]$.

- XOR is definable from {AND, OR, NOT}:

$$x \oplus y = (x \wedge \bar{y}) \vee (\bar{x} \wedge y)$$

- NOT is definable from {NAND}:

$$\bar{x} = \text{NAND}(x, x)$$

- AND is definable from {NAND}:

$$x \wedge y = \overline{\text{NAND}(x, y)} = \text{NAND}(\text{NAND}(x, y), \text{NAND}(x, y))$$

Applications:

1. **Circuit synthesis:** Convert expressions to available gate types
2. **Gate minimization:** Replace complex gates with universal gates
3. **Hardware verification:** Ensure gate library is sufficient
4. **Compiler optimization:** Transform logical expressions efficiently

Building Functions from NAND

Since $\{\text{NAND}\} = \{\overline{\wedge}\}$ is complete, we can build any function using only NAND gates.

Example: **Construct basic gates from NAND:**

Gate	Formula	NAND equivalent
NOT	$\neg x$	$x \overline{\wedge} x$
AND	$x \wedge y = \neg(x \overline{\wedge} y)$	$(x \overline{\wedge} y) \overline{\wedge} (x \overline{\wedge} y)$
OR	$x \vee y = \neg(\neg x \wedge \neg y)$	$(x \overline{\wedge} x) \overline{\wedge} (y \overline{\wedge} y)$
XOR	$x \oplus y = (x \wedge \neg y) \vee (\neg x \wedge y)$	$(x \overline{\wedge} (x \overline{\wedge} y)) \overline{\wedge} (y \overline{\wedge} (x \overline{\wedge} y))$

Trade-off: Using only NAND gates may increase circuit depth and gate count compared to mixed-gate designs.

Note: Similar constructions exist for NOR gates, providing alternative universal implementations.

Summary: Functional Completeness

Key concepts:

1. **Functional closure** $[F]$: all functions expressible using F
2. **Functional completeness**: $[F]$ contains all Boolean functions
3. **Post's five classes**: T_0, T_1, S, M, L (each closed under composition)
4. **Post's criterion**: F is complete $\Leftrightarrow F$ escapes all five classes
5. **Universal gates**: NAND and NOR are individually complete

Key insight: Post's theorem gives us a *complete characterization* of functional completeness. To check if a set is complete, just verify it has one function escaping each of the five classes. This is why NAND alone is universal — it escapes all five classes.

Summary: Functional Completeness [2]

Practical applications:

- Circuit design with minimal gate types
- Optimization of Boolean expressions
- Hardware verification and gate library design
- Understanding expressiveness limits

What's next: We've mastered the *theory*: Boolean algebra, normal forms, minimization, ANF, completeness. Now we implement these ideas in *hardware*: logic gates, combinational circuits, sequential circuits.

Digital Circuits

*“It is software that gives form and purpose to a programmable machine,
much as a sculptor shapes clay”*

— Alan Key

From Boolean Algebra to Hardware

Boolean algebra provides the mathematical foundation for digital circuit design.

Mathematical abstraction:

- Boolean variables: x, y, z
- Operations: $\wedge, \vee, \neg, \oplus$
- Functions: $f(x, y, z)$
- Truth tables and expressions
- Algebraic laws and transformations

Physical implementation:

- Voltage levels: HIGH (1), LOW (0)
- Transistor circuits (CMOS logic)
- Propagation delays
- Power consumption
- Real constraints: noise, timing, fanout

Connection: Boolean expressions translate directly to logic gate networks. Circuit optimization uses the algebraic laws we've studied.

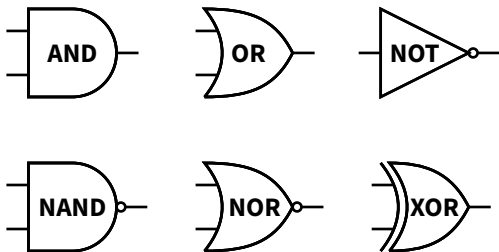
Logic Gate Standards

Two main notation standards for logic gates:

Definition 33: ANSI/IEEE Std 91-1984 and IEC 60617 define standardized symbols for logic gates used in circuit diagrams.

Note: Two styles exist: **Distinctive shapes** (American: AND = D-shape, OR = shield) and **Rectangular** (IEC: rectangles with symbols). We use distinctive shapes. Both are widely used in industry.

Standard Logic Gates: Visual Symbols



Bubble notation: Small circle (bubble) on gate input/output indicates logical negation.

Basic Logic Gates

Gate	Symbol	Expression	Truth Pattern
AND	D-shape	$A \wedge B$	Output 1 only when all inputs are 1
OR	Shield	$A \vee B$	Output 1 when any input is 1
NOT	Triangle + bubble	$\overline{A}$	Inverts the input
NAND	D-shape + bubble	$\overline{A \wedge B}$	NOT-AND
NOR	Shield + bubble	$\overline{A \vee B}$	NOT-OR
XOR	Shield + curve	$A \oplus B$	Output 1 when inputs differ
XNOR	Shield + curve + bubble	$\overline{A \oplus B}$	Output 1 when inputs match

Note: XNOR is also called *equivalence* gate: outputs 1 when inputs are equivalent.

Universal Gates: The Ultimate Minimalism

The manufacturing dream:

In the early days of computing (1950s-60s), every different gate type required:

- Different design specifications
- Different testing procedures
- Different inventory management
- Different failure modes to diagnose

Engineers wondered: “What if we could build EVERYTHING with just ONE gate type?”

The answer: NAND and NOR are *universal* — you can build any circuit using only NAND gates (or only NOR gates)!

This revolutionized chip manufacturing. Today, while modern chips use mixed gates for optimization, the NAND-only design philosophy still influences architecture.

Universal Gates: The Ultimate Minimalism [2]

Theorem 14: NAND and NOR gates are *functionally complete* — any Boolean function can be implemented using only NAND gates (or only NOR gates).

Example (Building the basic gates from NAND (the “construction kit”)): Starting with ONLY the NAND gate, we can build everything:

Step 1 — Build NOT: $\overline{A} = A \overline{\wedge} A$ “NAND a signal with itself to invert it”

Step 2 — Build AND: $A \wedge B = \overline{\overline{A \overline{\wedge} B}} = (A \overline{\wedge} B) \overline{\wedge} (A \overline{\wedge} B)$

“Do NAND, then invert the result with another NAND”

Step 3 — Build OR: $A \vee B = \overline{\overline{A} \overline{\wedge} \overline{B}} = (A \overline{\wedge} A) \overline{\wedge} (B \overline{\wedge} B)$

“Invert both inputs, then NAND them” (De Morgan’s law in action!)

Step 4 — Build anything: Since $\{\wedge, \vee, \neg\}$ is complete, and we can build all three from NAND, we can build *any* Boolean function using only NAND gates! ✓

Universal Gates: The Ultimate Minimalism [3]

Why this matters in practice:

- **Modern FPGA chips:** Built primarily from lookup tables (LUTs) that can implement any function
- **ASIC design:** Standard cell libraries often emphasize NAND/NOR for uniformity
- **Fault testing:** Easier to test one gate type comprehensively
- **Academic value:** Proves minimality — you can't do better than one operation!

Note: In reality, modern chip designers use mixed gates (AND, OR, NOT, XOR, *etc.*) because:

- Lower power consumption
- Faster operation (fewer gate delays)
- Smaller area (1 AND gate vs. 2 NANDs)

But NAND-only designs prove what's *possible* and provide fallback when needed!

Constructing Circuits from Boolean Expressions

General procedure:

1. **Start with Boolean expression:** Obtain from truth table (DNF/CNF) or specifications
2. **Minimize if desired:** Use K-maps, Quine-McCluskey, or algebraic methods
3. **Identify gate structure:** Break expression into hierarchical operations
4. **Draw circuit:** Connect gates according to expression structure
5. **Verify:** Check truth table matches specification

Key principle: Expression evaluation order determines circuit topology (depth and gates).

Circuit Construction: Example

Example: Build circuit for $f(A, B, C) = A \wedge B \vee \overline{A} \wedge C$

Step 1: Identify operations:

- Two AND gates: $A \wedge B$ and $\overline{A} \wedge C$
- One NOT gate: $\overline{A}$
- One OR gate: combining the AND results

Step 2: Determine levels:

1. Level 0 (inputs): A, B, C
2. Level 1: $\overline{A}$ (NOT gate)
3. Level 2: $A \wedge B$ and $\overline{A} \wedge C$ (AND gates)
4. Level 3: Final OR gate

Circuit depth: 3 levels (excluding inputs)

Note: Depth affects propagation delay — critical for circuit speed.

Multi-Output Circuits

We often need to compute *multiple functions* simultaneously, using one circuit with *shared inputs*.

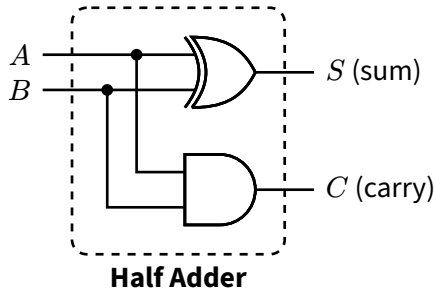
Example (Half Adder): Adds two bits A , B , produces sum and carry:

Outputs:

- Sum: $S = A \oplus B$
- Carry: $C = A \wedge B$

Truth table:

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



Gates needed: 1 XOR, 1 AND

Note: Half adder is the simplest arithmetic circuit — no carry-in means independent bit addition.

Full Adder: Definition

Definition 34: A *full adder* is a combinational circuit that adds three bits: two operand bits A , B and a carry-in bit C_{in} , producing a sum bit S and a carry-out bit C_{out} .

Direct gate-level implementation:

Outputs:

- Sum: $S = A \oplus B \oplus C_{\text{in}}$
- Carry-out: $C_{\text{out}} = (A \wedge B) \vee (C_{\text{in}} \wedge (A \oplus B))$

Alternative carry: $C_{\text{out}} = AB \vee AC_{\text{in}} \vee BC_{\text{in}}$

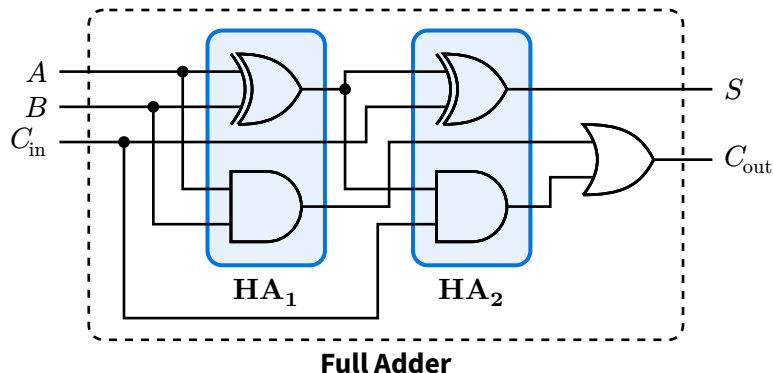
Implementation using half adders:

1. HA_1 : Add A and B
 $S_1 = A \oplus B$, $C_1 = A \wedge B$
2. HA_2 : Add S_1 and C_{in}
 $S = S_1 \oplus C_{\text{in}}$, $C_2 = S_1 \wedge C_{\text{in}}$
3. Combine carries: $C_{\text{out}} = C_1 \vee C_2$

Key insight: Both implementations are equivalent but differ in structure:

- Gate-level uses 5 gates with shared $(A \oplus B)$ term
- Half-adder-based uses 2 modular components + 1 OR gate

Full Adder: Circuit Implementation



Physical implementation:

- 2 XOR gates
- 2 AND gates
- 1 OR gate
- Total: 5 gates

Logical structure:

- Two half adders (blue boxes)
- One OR gate for carry combining
- Modular hierarchical design

Key insight: The same circuit can be viewed two ways:

- **Gate-level view:** 5 individual gates with wire sharing
- **Module-level view:** 2 half-adder components + OR combiner

The blue boxes show how the gates naturally group into half adders!

Other Arithmetic Circuits

Circuit	Function	Key Components
Subtractor	Compute $A - B$	Use 2's complement, add with negation
Comparator	Test $A < B$, $A = B$, $A > B$	XOR for equality, cascaded logic
Multiplexer	Select one of n inputs	$\log_2 n$ select lines, AND-OR structure
Demultiplexer	Route input to one of n outputs	Inverse of multiplexer
Encoder	Convert 2^n inputs to n -bit code	Priority encoding
Decoder	Convert n -bit code to 2^n outputs	Minterm generation

Multiplexers and Demultiplexers

Example (4-to-1 Multiplexer):

Function: Select one of 4 inputs (I_0, I_1, I_2, I_3) based on 2 select lines (S_1, S_0)

Truth: If $(S_1 S_0) = 01_2$, then $\text{Out} = I_1$

Implementation:

- 4 AND gates (each enabled by specific select combination)
- 1 OR gate (combines all enabled inputs)
- $\text{Out} = \overline{S_1} \overline{S_0} I_0 \vee \overline{S_1} S_0 I_1 \vee S_1 \overline{S_0} I_2 \vee S_1 S_0 I_3$

Note: Decoders are useful because they generate all possible minterms. An n -to- 2^n decoder produces all minterms for n variables, so you can build any Boolean function by just connecting the right outputs with OR gates.

Circuit Minimization Goals

Now that we've seen various combinational circuits, let's consider: **How do we make them efficient?**

Minimization objectives:

1. **Gate count:** Fewer gates $\Rightarrow$ lower cost, smaller chip area
2. **Circuit depth:** Fewer levels $\Rightarrow$ faster operation (less delay)
3. **Fanout:** Number of gates driven by one output (affects loading)
4. **Power consumption:** Fewer transitions $\Rightarrow$ less energy

Note: These objectives often conflict! Minimization requires trade-offs.

Example: DNF for $f = AB \vee AC \vee BC$ (3 terms, 3 gates):

- Algebraic identity: $AB \vee AC \vee BC = AB \vee AC$ when $A = 1$
- But for all cases, adding redundant term BC eliminates hazards!
- **Trade-off:** More gates for reliable behavior

Circuit Minimization Techniques

Technique	Best For	Limitations
Algebraic laws	Small expressions, manual	Error-prone, no guarantee
K-maps	2-4 variables, visualization	Does not scale beyond 6 variables
Quine-McCluskey	Exact minimization, automation	Exponential complexity
Espresso	Multi-output, heuristic	May not find optimal solution
Technology mapping	Gate libraries, ASIC/FPGA	Requires EDA tools

Note: Modern synthesis tools (e.g., Synopsys Design Compiler, Cadence Genus) combine multiple techniques and optimize for specific target technologies.

Two-Level vs Multi-Level Logic

Aspect	Two-Level Logic	Multi-Level Logic
Form	SoP or PoS	Factored expressions, nested gates
Advantages	+ Fast (minimal depth = 2) + Easy to minimize (K-maps) + Predictable timing	+ Fewer gates (shared sub-expressions) + Smaller area + Lower power
Disadvantages	- More gates (exponential growth) - Large chip area	- Slower (deeper paths) - Harder to minimize - Complex timing analysis

Modern approach: Use multi-level logic for area/power optimization, then optimize critical paths for speed.

Sequential Circuits: The Need for Memory

Limitation of combinational circuits: No memory!

What combinational circuits CAN do:

- Compute functions of current inputs ✓
- Process data in one pass ✓
- Adders, comparators, multiplexers, ALUs ✓

What combinational circuits CANNOT do:

- Remember past inputs or results ✗
- Count events over time ✗
- Implement algorithms with loops or state ✗
- Build registers, counters, or processors ✗

To build real computers, we need circuits that “remember”!

This needs a completely different approach than what we've seen so far.

Combinational vs Sequential Circuits

Definition 35: A *sequential circuit* has outputs that depend on:

1. Current inputs (like combinational circuits)
2. **Previous state** (memory)

This requires *feedback* — outputs feed back to inputs.

Example:

- **Combinational:** ALU, multiplexer, decoder (no memory)
- **Sequential:** Registers, counters, state machines (with memory)

Note: Truth tables don't work for sequential circuits — we need *state diagrams* or *state tables* instead.

Latches: Basic Memory Elements

Definition 36: A *latch* is a bistable circuit with two stable states, used to store one bit.

Example (SR Latch (Set-Reset)):

Inputs: S (set), R (reset)

Outputs: Q (state), $\overline{Q}$ (complementary)

Behavior:

- $S = 1, R = 0$: Set $Q = 1$
- $S = 0, R = 1$: Reset $Q = 0$
- $S = 0, R = 0$: Hold current state (memory!)
- $S = 1, R = 1$: **Forbidden** (both outputs would be 0)

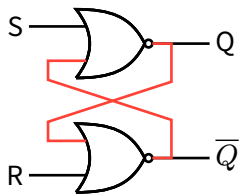
Implementation: Two cross-coupled NOR gates (or NAND gates)

Warning: SR latch has a forbidden state ($S = 1, R = 1$) that must be avoided by circuit design.

SR Latch: Circuit and Behavior

NOR-based SR Latch:

Circuit structure:



Feedback creates memory!

State table:

S	R	Q_t	Q_{t+1}	Action
0	0	0	0	Hold
0	0	1	1	Hold
0	1	0/1	0	Reset
1	0	0/1	1	Set
1	1	?	?	Forbidden

Equations:

- $Q = \overline{S \vee \overline{Q}}$
- $\overline{Q} = \overline{R \vee Q}$

Note: Q_t = current state, Q_{t+1} = next state. The latch “remembers” when $S = R = 0$.

Why Truth Tables Fail for Sequential Circuits

Key insight: An SR latch cannot be described by a simple Boolean function $Q = f(S, R)$ because it depends on *previous state*.

Why not? The output Q depends on:

- Current inputs S, R
- **Previous output** Q_t (feedback)

This is a *state-dependent* behavior:

- $(S, R) = (0, 0)$ with $Q_t = 0 \Rightarrow Q_{t+1} = 0$
- $(S, R) = (0, 0)$ with $Q_t = 1 \Rightarrow Q_{t+1} = 1$

Same inputs, different outputs! This violates the basic idea of a function.

Sequential circuits need:

- State tables (not truth tables)
- Finite state machines (FSMs)
- Temporal logic (next-state functions)
- Timing diagrams showing state transitions

Bottom line: Pure Boolean functions can only describe *combinational* logic.

Flip-Flops: Clocked Memory

Definition 37: A *flip-flop* is an edge-triggered memory element that changes state only on clock transitions (rising or falling edge).

Key difference from latches:

- **Latches:** Level-sensitive (transparent when enabled)
- **Flip-flops:** Edge-triggered (change only on clock edge)

Flip-flops are more predictable and easier to synchronize in complex systems.

D Flip-Flop

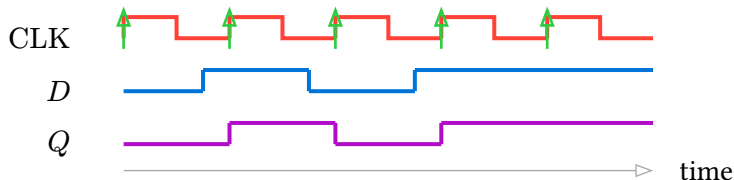
Example (D (Data) Flip-Flop): **Input:** D (data), CLK (clock), **Output:** Q

Behavior:

- On clock edge (e.g., rising edge $\uparrow$): $Q_{t+1} = D_t$ (capture and store D value)
- Between clock edges: Q holds value (ignores D changes)

Characteristic equation: $Q_{t+1} = D$

Timing diagram:



Note: Notice Q changes *only* at rising edges ($\uparrow$), capturing D 's value at that instant, regardless of D changes between edges.

Use cases: Registers, memory buffers, pipeline stages

D flip-flops eliminate the forbidden state problem of SR latches.

Other Flip-Flop Types

JK Flip-Flop

Inputs: J (set), K (reset), CLK

Behavior on clock edge:

- $(J, K) = (0, 0)$: Hold state
- $(J, K) = (0, 1)$: Reset ($Q = 0$)
- $(J, K) = (1, 0)$: Set ($Q = 1$)
- $(J, K) = (1, 1)$: Toggle ($Q = \overline{Q}$)

Equation: $Q_{t+1} = J\overline{Q}_t \vee \overline{K}Q_t$

Toggle mode ($J = K = 1$) ideal for counters and frequency dividers.

T (Toggle) Flip-Flop

Input: T (toggle), CLK

Behavior on clock edge:

- $T = 0$: Hold state
- $T = 1$: Toggle ($Q_{t+1} = \overline{Q}_t$)

Equation: $Q_{t+1} = T \oplus Q_t$

Implementation: JK with $J = K = T$

Common in binary counters (each stage divides frequency by 2).

Note: In practice, however, most designs use *D flip-flops* because they're *simpler* and more *predictable*.

Clock Signals and Synchronization

Definition 38: A *clock signal* is a periodic square wave that synchronizes state changes across a sequential circuit.

Key clock parameters:

- **Frequency:** Clock cycles per second (Hz)
- **Period:** Time for one complete cycle
- **Duty cycle:** Fraction of period when clock is HIGH
- **Rising edge** ($\uparrow$) or **Falling edge** ($\downarrow$): Transition points

Synchronous design: All flip-flops share the same clock signal, so state updates happen together.

Asynchronous design: No global clock (timing analysis is much harder).

Note: Most digital systems use *synchronous* design because it's more predictable and *easier to verify*.

Real Circuit Issues: Hazards

Definition 39: A *hazard* (or *glitch*) is an unwanted transient change in circuit output due to:

- Unequal propagation delays in different paths
- Race conditions between signals

Example (Static-1 Hazard): Circuit: $f = A\overline{B} + BC$ with $A = 1, C = 1$, while B transitions $1 \rightarrow 0$

K-map analysis:

		C	
		0	1
AB	00	0 ₀	0 ₁
	01	0 ₂	1 ₃
	11	0 ₆	1 ₇
	10	1 ₄	1 ₅

Gap between rectangles causes hazard at transition!

Fix: Add redundant term (AC)

		C	
		0	1
AB	00	0 ₀	0 ₁
	01	0 ₂	1 ₃
	11	0 ₆	1 ₇
	10	1 ₄	1 ₅

Now the transition is covered by the new rectangle.

Real Circuit Issues: Hazards [2]

- Expected: f stays 1 (both terms can be 1)
- Actual (without fix): Brief glitch to 0 if delays differ
- Solution: Redundant term eliminates the gap

Hazard explanation: When $A = 1, C = 1$ and B transitions from 1 to 0:

- Initially ($B = 1$): $f = 1 \cdot 0 + 1 \cdot 1 = 1$ (via BC)
- Finally ($B = 0$): $f = 1 \cdot 1 + 0 \cdot 1 = 1$ (via $A\bar{B}$)
- **Problem:** If BC turns off before $A\bar{B}$ turns on, f briefly becomes 0
- **Fix:** Add AC term, which stays 1 throughout: $f = A\bar{B} + BC + AC$

Types of hazards:

- **Static hazards:** Output should stay constant but glitches
- **Dynamic hazards:** Multiple transitions during single input change

Race Conditions and Metastability

Definition 40: A *race condition* occurs when circuit behavior depends on the relative timing of multiple signals, leading to unpredictable outcomes.

Definition 41: *Metastability* occurs when a flip-flop input changes too close to the clock edge, causing the output to hover in an undefined state before settling.

Note: Timing constraints: Setup time (t_{setup}): input stable before clock; Hold time (t_{hold}): input stable after clock. Violating these causes metastability. Proper synchronizer design (multi-stage flip-flops) reduces risk.

Arithmetic Logic Unit (ALU) Overview

Definition 42: An *Arithmetic Logic Unit (ALU)* is a combinational circuit that performs arithmetic and logical operations on integer operands.

Typical ALU operations:

Arithmetic:

- Addition, Subtraction
- Increment, Decrement
- Multiplication (in some ALUs)
- Comparison

Logical:

- AND, OR, XOR, NOT
- Shift left, Shift right
- Rotate operations
- Bit manipulation

Control: Operation select lines determine which function the ALU performs.

Note: ALUs are central components of CPUs, performing the bulk of computational work.

Summary: Digital Circuits

Key concepts covered:

Combinational Logic

1. Logic gates and standards (ANSI/IEC)
2. Circuit construction from Boolean expressions
3. Arithmetic circuits (adders, subtractors, comparators)
4. Multi-level vs two-level logic
5. Circuit minimization techniques

Sequential Logic

1. Latches (SR latch, forbidden states)
2. Flip-flops (D, JK, T, master-slave)
3. Clock signals and edge-triggering
4. State-dependent behavior
5. Timing constraints and hazards

Key insight: Boolean algebra provides the *perfect mathematical model* for combinational circuits. Every gate, every circuit, every processor follows these algebraic laws. Sequential circuits add time and state, requiring flip-flops and temporal reasoning.

Summary: Digital Circuits [2]

From theory to silicon: We've seen how abstract Boolean algebra (1854) connects to physical circuits (transistors, gates, propagation delays). This is the bridge from mathematics to the processors running everything in the digital world.

The Journey Complete

What You've Mastered

Theoretical Mastery:

- Design ANY Boolean function from scratch
- Prove algebraic identities rigorously
- Understand Post's completeness criterion
- Analyze cryptographic security via ANF
- Recognize lattice structure in Boolean algebra

Practical Skills:

- Minimize circuits using K-maps
- Build hardware from NAND gates alone
- Optimize expressions algebraically
- Design adders and arithmetic circuits
- Understand sequential logic and timing

The complete picture:

When you write `if (user.isLoggedIn && !user.isBanned)` in your code, you now understand the ENTIRE chain:

- Boolean algebra: $x \wedge (\neg y)$
- Truth tables: evaluating all $2^2 = 4$ cases
- Circuit synthesis: AND gate + NOT gate
- Physical implementation: transistors switching at GHz speeds
- Optimization: Can we use fewer gates?

What You've Mastered [2]

- Cryptographic perspective: What's the algebraic degree?

You understand it ALL — from philosophy (Boole, 1854) to physics (transistors, 2024)!

Applications of Boolean Algebra

Where Boolean algebra lives in the real world:

Computing & Hardware:

- CPU architecture and processor design
- FPGA programming and digital circuit synthesis
- Memory systems (cache, RAM)
- ALU design and arithmetic circuits

Security & Verification:

- Cryptography (AES S-boxes, stream ciphers)
- Algebraic cryptanalysis and attacks
- Formal verification (SAT/SMT solvers)
- Hardware security modules

Software & Optimization:

- Compiler optimization (Boolean expression simplification)
- Database query optimization
- AI hardware accelerators (TPUs, GPUs)
- Model checking and program analysis

Algorithms & Theory:

- Satisfiability problems (3-SAT, NP-completeness)
- Binary decision diagrams (BDDs)
- Coding theory (Reed-Muller codes)
- Error detection and correction

Summary: The Complete Journey

From George Boole's 1854 "Laws of Thought" to Claude Shannon's 1937 breakthrough connecting algebra to circuits, Boolean algebra became the foundation of digital computing.

What we've mastered:

1. **Algebraic structure:** Axioms, laws, duality, lattices
2. **Normal forms:** DNF/CNF, minterms/maxterms, SoP/PoS, Shannon expansion
3. **Minimization:** K-maps (visual), Quine-McCluskey (systematic), algebraic methods
4. **ANF:** Unique canonical form, algebraic degree, cryptographic applications
5. **Functional completeness:** Post's five classes, universal gates (NAND/NOR)
6. **Digital circuits:** Combinational (gates, adders) and sequential (latches, flip-flops)

Core insight: Simple operations ($\wedge$, $\vee$, $\neg$) on two values (0, 1) generate the entire computational universe. Boolean algebra provides the *perfect mathematical model* for digital computation — every gate, every circuit, every processor follows these algebraic laws.

Summary: The Complete Journey [2]

Tools you can now use: Truth tables (enumeration), algebraic manipulation (laws), K-maps (visual optimization), Quine-McCluskey (systematic), ANF (cryptanalysis), Post's criterion (completeness), circuit design (from abstraction to silicon).