5. Boolean Algebra

5.0 Boolean Algebra

- Boolean algebra is a system for working with variables that have values of either 0 or 1.
- Boolean algebra is closely related to propositional logic.
- The definitions for Boolean operations are similar to logical operations but with different notation.
- In Boolean algebra, 1 represents T and 0 represents F.

5.1 Introduction to Boolean Algebra

Operations

- Boolean operations are similar to propositional operations.

Multiplication

- Boolean multiplication is denoted by • and follows the same rules as regular multiplication.
- The results of Boolean multiplication are the same as the logical "and" operation.

Boolean •	Logical ∧
0 • 0 = 0	F ∧ F = F
0 • 1 = 0	F ∧ T = F
1 • 0 = 0	T ∧ F = F
1 • 1 = 1	T ∧ T = T

Addition

- Boolean addition uses the symbol "+" and follows the same rules as standard addition.
- Except 1 + 1 ≠ 2, since 2 is not an element of {0, 1}.
- The results of Boolean addition are the same as the logical "or" operation.

Boolean +	Logical ∨
0 + 0 = 0	F ∨ F = F
0 + 1 = 1	F ∨ T = T
1 + 0 = 1	T ∨ F = T
1 + 1 = 1	T ∨ T = T

Complement

- Complement of an element is denoted by a bar symbol.
- Complementing a Boolean value is similar to applying the "not" operation in logic.
- The expression under a complement operation is evaluated before the operation is applied.

Boolean complement	Logical ¬
0 = 1	¬F = T
1 = 0	¬T = F

George Boole's and Digital Logic

- George Boole defined Boolean algebra in the mid-19th century.
- Computers use 0s and 1s as their native language.
- Boolean algebra is the basis of computer circuit design.

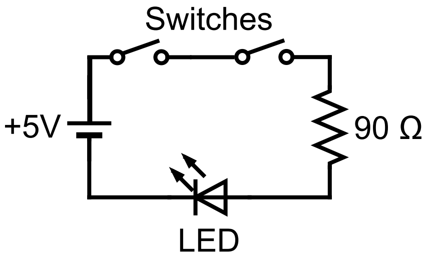
- Vacuum tubes were used to store information as on or off states (1s and 0s).
- Today, high and low voltage states represent 1s and 0s in computer components.
- The foundation of computer circuitry is still based on Boolean algebra.
- The field of computer science that designs computer circuitry is called digital logic.

Claude Shannon and Telephone Switching Networks

- Claude Shannon noticed the connection between Boolean algebra and telephone switching networks as a graduate student at MIT in the late 1930s.
- He showed that Boolean algebra could be used to simplify electronic switches used in telephone exchanges at the time.
- Shannon's study focused on electrical circuits, which have a light that illuminates if the circuit forms a closed loop.
- A circuit contains switches that are either closed or open, and the setting of the switches determines whether the circuit is closed.

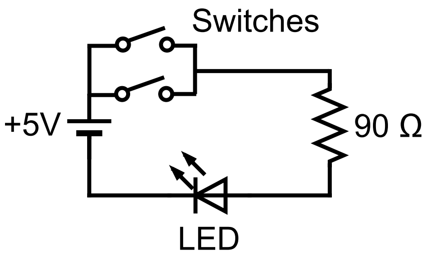
Boolean Logic in Circuits

And Circuit



- When both switches are closed, the LED will recieve current.

Or Circuit



- When either switch is closed, the LED will recieve current.

Variables and Expressions

- Boolean variables have a value of 1 or 0.
- Boolean expressions are built up by applying Boolean operations to Boolean variables or constants.
- The value of a Boolean expression can depend on the order of operations.

Operator Precedence

- Precedence rules exist to determine the correct order of operations.

Precedence	Boolean Operator	Name
0	a · b	Multiplication
1	a + b	Addition
*	(a + b) · c	Parentheses override precedence rules
**	$a + \overline{b + c}$	Complement are given precedence as parentheses

Best Practices

- It is good practice to use parentheses for readability when applying the complement operation, except when it is applied to a single variable or the entire expression.

- When both inputs to the • operation are variables, the expression x • y can be written as xy for more compact expressions.

Boolean Equivalence

- Equivalence between two Boolean expressions is determined by checking if they have the same value for every possible combination of assigned variable values.
- The symbol (≡) denotes logical equivalence in propositional logic, while (=) is used in Boolean algebra.
- Boolean algebra follows the same laws (pairs of equivalent expressions) as propositional logic.

Laws

Law	Definition	
Idempotent	$x + x = x$	$x \bullet x = x$
Associative	$(x + y) + z = x + (y + z)$	$(x \bullet y)z = x(y \cdot z)$
Commutative	$x + y = y + x$	$x \bullet y = y \cdot x$
Distributive	$x + (y \bullet z) = (x + y)(x + z)$	$x \bullet (y + z) = (x \bullet y) + (x \bullet z)$
Identity	$x + 0 = x$	$x \bullet 1 = x$
Domination	$x + 1 = 1$	$x \bullet 0 = 0$
Double complement	$\overline{\overline{x}} = x$	
Complement	$x = \overline{x} = 1$ $\overline{0} = 1$	$x \bullet \overline{x} = 0$ $\overline{1} = 0$
De Morgan's	$\overline{x + y} = \overline{x} \bullet \overline{y}$	$\overline{x \bullet y} = \overline{x} + \overline{y}$
Absorption	$x + (x \bullet y) = x$	$x \bullet (x + y) = x$

- Boolean algebra laws apply similarly to propositional logic laws to demonstrate equivalence between two Boolean expressions.

Set Operation Analogs

Boolean operation/constant	Set operation
Multiplication	Intersection: ∩
Addition	Union: ∪
Complement	Set complement: $\overline{X}$ = A - X
1	the whole set A
0	∅

- The power set of A satisfies the conditions of a Boolean algebra.
- The {0, 1} set along with complement, +, and • operations is also an instance of a Boolean algebra.
- The {0, 1} Boolean algebra is commonly used in modern digital logic.
- The term "Boolean algebra" can refer to either the {0, 1} set or a general class of sets and operations.
- It is important to be aware of both meanings to avoid confusion.

Examples

1) $(\overline{x + 0})(\overline{y} + \overline{z})$

$$\text{Let } x = 0,\, y = 1,\, z = 0$$

$$\begin{aligned}
(\overline{x + 0})(\overline{y} + \overline{z}) &= (\overline{0 + 0})(\overline{1} + \overline{0}) \\
&= (\overline{0})(\overline{1} + \overline{0}) \\
&= (\overline{0})(0 + 1) \\
&= 1(0 + 1) \\
&= 1 \cdot 1 \\
&= 1
\end{aligned}$$

2) $(a + \overline{b + c}) \cdot d$

- The tuple (a, b, c, d) represents a 4-bit number, so each entry can be either 1 or 0.
- Since a 4-bit number has are 2^4 possible values, there are 16 possible values for (a, b, c, d) .

<u>Symbolic Conversion</u>

	Expression	Reasoning
0	$(a + \overline{b + c}) \cdot d$	
1	$(a \vee \overline{b \vee c}) \wedge d$	Addition equivalence
2	$(a \vee (\overline{b} \wedge \overline{c})) \wedge d$	De Morgan's Law
3	$(a \vee (\neg b \wedge \neg c)) \wedge d$	Compliment equivalence

<u>Truth Table</u>

a	b	c	d	$((a \vee (\neg b \wedge \neg c)) \wedge d)$
F	F	F	F	F
				...
T	F	F	T	T
				...
T	F	T	F	F
				...
T	T	T	T	T

3) $xy + \overline{x}y + zy$

<u>Reduced the Expression</u>

	Expression	Reason

	Expression	Reason
0	$(x \bullet y) + (\overline{x} \bullet y) + (z \bullet y)$	
1	$(x \bullet \overline{x} \bullet z) + y$	Distributive Law
2	$(0 \bullet z) + y$	Complement Laws
3	$0 + y$	Domination Law
4	y	Identity Law

5.2 Boolean Functions

A boolean function operates on boolean values (true or false) and returns a boolean value as a result.

A Boolean function maps Boolean input values to the set {0, 1}.

Let B = {0, 1}

- B^k is the set of all k-tuples over B.
- A Boolean function with k input variables maps B^k to B.

E.g.

f: B^3 → B, where B = {0, 1}
$f(u, r, t) = \overline{(u + r)} - t$

$f(1, 1, 0) = 0$

Input/Output Table

- An input/output table showing the output value for every possible combination of input values can be used to define a Boolean function.

Example: How to Safely Enrich Uranium

- Consider a Boolean function with 3 inputs that returns a boolean indicating whether a uranium enrichment process is operating within safe parameters.

Let
$f(v, r, h) = v \bullet r \bullet h$

Where
v = Centrifuge is operating at 50k RPM
r = Flow Rate is 100 grams per hour
h = Temperature is 30° C

The function can be transliterated to propositional logic.
$V \land R \land H$

The input/output table is just the truth table of this expression.

v	r	h	f(v, r, h)
T	T	T	T
T	T	F	F
T	F	T	F
T	F	F	F
F	T	T	F
F	T	F	F
F	F	T	F
F	F	F	F

- An input/output table for a Boolean function with k input variables requires 2^k rows.
- Using a table to define a Boolean function is only feasible for functions with a small number of input variables due to the large number of rows required.

Reversing Input/Output Tables

Consider the following table

x	y	z	f(x, y, z)	
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	1	$\overline{x}\bullet y\bullet z$
1	0	0	1	$x\bullet\overline{y}\bullet\overline{z}$
1	0	1	1	$x\bullet\overline{y}\bullet z$
1	1	0	0	
1	1	1	1	$x\bullet y\bullet z$

Find an equivalent Boolean expression for f

1. Construct a term of the form x•y•z for each row where f(x, y, z) = 1
2. Add together terms.

$$f(x, y, z) = \overline{x}yz + x\overline{y}\overline{z} + x\overline{y}z + xyz$$

Formal Description of Constructing Boolean Expression

- To construct a Boolean expression equivalent to a Boolean function defined by an input/output table, minterms and literals must be defined.

- A literal is a Boolean variable or the complement of a Boolean variable.
- A minterm is a product of literals where each literal is either a variable or the complement of a variable.
- E.g. xyz and $\overline{x}\overline{y}\overline{z}$ are minterms for a Boolean function with input variables x, y, and z.
- However, yz is not a minterm because variable x is missing from the term.
- To create a Boolean function equivalent to a function defined in a table, create a sum of minterms where the set of minterms included in the sum correspond to rows in the table where the function's value is 1.

5.3 Disjunctive & Conjunctive Normal Form

- For computer storage and manipulation of Boolean expressions (such as in automated reasoning systems), it is useful to have standardized expressions.

Disjunctive Normal Form

- A Boolean expression that is a sum of products of literals is said to be in disjunctive normal form (DNF).
- The term "disjunctive" comes from propositional logic and reflects the mathematical equivalence between Boolean algebra and propositional logic.

Definition of Disjunctive Normal Form

A disjunctive normal form (DNF) expression has the form:

$$c_1 + c_2 + ... + c_m$$

where each c_j for $j \in \{1, ..., m\}$ is a product of literals.

E.g.

$$\overline{x}y\overline{z} + x\overline{y} + w + w\overline{y}z$$

- The expression is a sum of terms.
- Each term is the product of literals (x, y, z, w).
- Complement operators are applied to variables singularly.
- There is no addition within a term.

- A term in a disjunctive normal form expression can be a single literal or a product of literals.
- An expression with only one term, such as xyz, is also considered to be in disjunctive normal form.
- In a DNF expression, multiplication can only be applied to literals, and expressions of the form (x + y)z are not allowed.
- Every Boolean function can be represented in disjunctive normal form.
- The procedure to turn a table representation (i.e. input/output tables) of a Boolean function into a Boolean expression produces an expression in DNF.
- It is important to note that there may be multiple expressions in DNF that correspond to the same function.

Examples

Which expressions are in disjunctive normal form?

1. $xy + \overline{w}$
2. $\overline{x + y}$
3. $x + y + z$
4. $xy\overline{w}$
5. $(x + y)\overline{w}$

Answer Key

1. DNF
2. Not DNF
3. DNF
4. DNF
5. Not DNF

Conjunctive Normal Form

- A Boolean expression that is a product of sums of literals is said to be in conjunctive normal form (CNF).
- Each term in the product that is a sum of literals is called a clause.
- CNF expressions are often used to express a set of constraints in a computational problem using logic.
- In a CNF expression, addition can only be applied to literals, and expressions of the form $(x \cdot y) + z$ are not allowed.

Definition of Conjunctive Normal Form

A conjunctive normal form (CNF) expression has the form:

$$d_1 \bullet d_2 \bullet .... \bullet d_m$$

where each d_j for $j \in \{1, \ldots, m\}$ is a sum of literals.
Each d_j is called a clause.

E.g.

$$(x + \overline{y} + z) \bullet (x + y) \bullet (w\overline{x}) \bullet (z)$$

- The expression is a product of four clauses.
- Complement operators are applied to variables singularly.
- There is no multiplication within a clause.

- The clauses in a CNF expression can be single literals or a sum of literals.
- A CNF expression with only one clause, such as $x + y$, is also considered to be in conjunctive normal form.
- In a CNF expression, addition can only be applied to literals, and expressions of the form $xy + z$ are not allowed.
- It is important to note that a CNF expression can represent any Boolean function, but there may be multiple CNF expressions that correspond to the same function.

Examples

Which expressions are in conjunctive normal form?

1. $z(xy + \overline{w})$
2. $(x + y + z)(\overline{w} + \overline{y})$
3. $\overline{(x + y)}(x + y)$
4. $(x + y + z)\overline{w}$
5. $x + y + z$
6. $xy\overline{w}$

Answer Key

1. Not CNF
2. CNF
3. Not CNF
4. CNF
5. CNF
6. CNF

5.4 Functional completeness

- A set of operations is functionally complete if any Boolean function can be expressed using only operations from the set.
- The set {addition, multiplication, complement} is functionally complete because any Boolean function can be expressed in disjunctive normal form which only uses these three operations.

Incomplete Set of Operations

- Some hardware may lack either addition or multiplication operations.
- If addition is lacking, but multiplication and complement operations are available, De Morgan's law can be used to achieve functional completeness.
- Similarly, if multiplication is lacking, but addition and complement operations are available, the same technique can be applied.
- De Morgan's law allows for the replacement of the missing operation to achieve functional completeness in either case.

De Morgan's Law with 2-Terms

$$x + y = \overline{\overline{x} \bullet \overline{y}}$$

$$x \bullet y = \overline{\overline{x} + \overline{y}}$$

De Morgan's Law with n-Terms

$$a_1 + a_2 + \cdots + a_n = \overline{\overline{a_1} \cdot \overline{a_2} \cdots \overline{a_n}}$$

$$a_1 \cdot a_2 \cdots a_n = \overline{\overline{a_1} + \overline{a_2} + \cdots + \overline{a_n}}$$

Missing Addition?

- An arbitrary Boolean function can be expressed using only multiplication and complement operations by following these steps:
    1. Start with the input/output table for the function.
    2. Find a DNF expression that is equivalent to the function.
    3. Repeatedly apply De Morgan's law to eliminate each addition operation.

$$\text{De Morgan's Law}$$

$$x + y = \overline{\overline{x} \cdot \overline{y}}$$

- The second step uses the sum of minterms method to generate a DNF expression equivalent to a given input/output table.

Examples

Use De Morgan's law to find an equivalent Boolean expression which lacks addition operations.

1. $\overline{x} \cdot \overline{y} + \overline{x}y + xy$

2. $\overline{x}y\overline{z} + x\overline{y}z + xyz$

Solutions

1.

$$\overline{x} \cdot \overline{y} + \overline{x}y + xy$$

$$\overline{(\overline{\overline{x} \cdot \overline{y}} \cdot \overline{\overline{x}y})} + xy \quad \text{De Morgan's Law}$$

$$\overline{\overline{\overline{\overline{\overline{x} \cdot \overline{y}} \cdot \overline{\overline{x}y}}} \cdot \overline{xy}} \quad \text{De Morgan's Law}$$

$$\overline{\overline{\overline{x} \cdot \overline{y}} \cdot \overline{\overline{x}y} \cdot \overline{xy}} \quad \text{Double Complement Law}$$

2.

$$\overline{x}y\overline{z} + x\overline{y}z + xyz$$

$$\overline{\overline{\overline{x}y\overline{z}} \cdot \overline{x\overline{y}z} \cdot \overline{xyz}} \quad \text{De Morgan's Law}$$

Missing Multiplication?

- An arbitrary Boolean function can be expressed using only addition and complement operations by following these steps:
  1. Start with the input/output table for the function.
  2. Find a DNF expression that is equivalent to the function.
  3. Repeatedly apply De Morgan's law to eliminate each multiplication operation.

$$\text{De Morgan's Law}$$

$$x \cdot y = \overline{\overline{x} + \overline{y}}$$

- The resulting Boolean expression uses only addition and complement and is functionally complete with the set {addition, complement}.

Examples

Use De Morgan's law to find an equivalent Boolean expression which lacks multiplication operations.

1. x • y + z

2. x • y • z

Solutions

1. $\overline{\overline{x} + \overline{y}} + z$

2. $\overline{\overline{x} + \overline{y} + \overline{z}}$

NAND and NOR Operations

- Using a single type of circuit to implement each operation can improve hardware efficiency and costs.
- Addition, multiplication, and complement operations are not functionally complete individually.
- To achieve functional completeness in such a circuit, a new operation must be introduced.
- The NAND (Not And) operation is represented by ↑ and is equivalent to $\overline{x \bullet y}$.
- The NOR (Not Or) operation is represented by ↓ and is equivalent to $\overline{x + y}$.

NAND		
x	y	x ↑ y

NOR		
x	y	x ↓ y

NAND		
1	1	0
1	0	1
0	1	1
0	0	1

NOR		
1	1	0
1	0	0
0	1	0
0	0	1

Proving NAND is Functionally Complete

- To prove that a set of operations is functionally complete, a known functionally complete set can be used to demonstrate how to compute each operation with the new set.

- The set {multiplication, complement} can be used to illustrate this approach and show that {NAND} is functionally complete.

Proof of Complement

- The identity $\overline{x}$ = x ↑ x can be verified by filling out a simple input/output table.

x	**x ↑ x**	$\overline{x}$
1	0	0
0	1	1

Proof of Multiplication

- Show that $xy = (x \uparrow y) \uparrow (x \uparrow y)$

$$\begin{aligned}
x \cdot y &\equiv\ x \wedge y && \text{By Equality} \\
&\equiv \neg\neg(x \wedge y) && \text{Double Negation} \\
&\equiv\ \neg(x \uparrow y) && \text{Definition of Nand} \\
&\equiv\ \overline{x \uparrow y} && \text{By Equality} \\
&\equiv\ \bar{a} && \text{Let } a = x \uparrow y \\
&\equiv\ a \uparrow a && \text{Compliment Identity} \\
x \cdot y &\equiv\ (x \uparrow y) \uparrow (x \uparrow y) && \text{Substitution. } \blacksquare
\end{aligned}$$

Proof of Addition

- Show that x + y = (x ↑ x) ↑ (y ↑ y)

$$\begin{aligned}
x + y &\equiv\ x \vee y && \text{By Definition of Addition} \\
&\equiv\ \overline{(\ \overline{x} \wedge \overline{y}\ )} && \text{De Morgan's Law} \\
&\equiv\ \overline{(\ a \wedge b\ )} && \text{Let } a = \overline{x}, b = \overline{y} \\
&\equiv\ a \uparrow b && \text{By Definition of NAND} \\
&\equiv\ \overline{x} \uparrow \overline{y} && \text{Substitution} \\
&\equiv\ (x \uparrow x) \uparrow (y \uparrow y) && \text{Complement Identity. } \blacksquare
\end{aligned}$$

Proving NOR is Functionally Complete

- The nor operation is proven in the same way as the nand operation by demonstrating how to compute each operation with nor alone.

Proof of Complement

- The identity $\overline{x} = x \downarrow x$ can be verified by filling out a simple input/output table.

x	**$x \downarrow x$**	**$\overline{x}$**
1	0	0
0	1	1

- Note the congruencies:
  $x \downarrow x = x \uparrow x = \overline{x}$

- This makes sense grammatically, since "not x and y" is the same as saying "not x or y".

Proof of Multiplication

- Show that $x \bullet y = (x \downarrow x) \downarrow (y \downarrow y)$

$$
\begin{aligned}
x \cdot y \equiv\ & x \wedge y && \text{By Equality} \\
\equiv\ & \overline{(\ \overline{x} \vee \overline{y}\ )} && \text{De Morgan's Law} \\
\equiv\ & \overline{(\ a \vee b\ )} && \text{Let } a = \overline{x}, b = \overline{y} \\
\equiv\ & a \downarrow b && \text{By Definition of Nor} \\
\equiv\ & \overline{x} \downarrow \overline{y} && \text{Substitution} \\
\equiv\ & (x \downarrow x) \downarrow (y \downarrow y) && \text{Complement Identity. } \blacksquare
\end{aligned}
$$

Proof of Addition

- Show that x + y = (x ↓ y) ↓ (x ↓ y)

$$
\begin{aligned}
x + y \equiv\ & x \vee y && \text{By Equality} \\
\equiv\ & \neg\neg(x \vee y) && \text{Double Negation} \\
\equiv\ & \neg(x \downarrow y) && \text{Definition of Nor} \\
\equiv\ & \overline{x \downarrow y} && \text{By Equality} \\
\equiv\ & \overline{a} && \text{Let } a = x \downarrow y \\
\equiv\ & a \downarrow a && \text{Compliment Identity} \\
x + y \equiv\ & (x \downarrow y) \downarrow (x \downarrow y) && \text{Substitution. } \blacksquare
\end{aligned}
$$

5.5 Boolean Satisfiability

Boolean satisfiability (SAT) determines if a given Boolean expression can be assigned variable values that make it evaluate to 1.

Satisfiable vs Unsatisfiable

- A Boolean expression is *satisfiable* if it can be evaluated to 1 by assigning values to its input variables.
- If it is impossible to assign values to the input variables to evaluate the expression to 1, the expression is *unsatisfiable*.
- An assignment *satisfies* a Boolean expression if it causes the expression to evaluate to 1.

Challenges and Limits

- To prove that a Boolean expression is unsatisfiable, we must consider all possible variable assignments.
- The time required for this task is proportional to 2^n, where n is the number of variables in the expression.
- Despite many years of research, there is no known provably efficient method for solving SAT.

Examples

Are the following Boolean expressions satisfiable?

1. $f(x, y) = (x + y)\overline{x}(x + \overline{y})$
2. $g(x, y, z) = (x + y + z)\overline{y}(\overline{x} + y)$

Solutions

1. $f(x, y) = (x + y)\overline{x}(x + \overline{y})$

x	**y**	$f(x, y) = (x + y)\overline{x}(x + \overline{y})$	**$f(x, y)**
1	1	$f(1, 1) = (1 + 1)\overline{1}(1 + \overline{1})$	1•0•1 = 0
1	0	$f(1, 0) = (1 + 0)\overline{1}(1 + \overline{0})$	1•0•1 = 0
0	1	$f(0, 1) = (0 + 1)\overline{0}(0 + \overline{1})$	1•1•0 = 0

x	y	$f(x,y) = (x+y)\overline{x}(x+\overline{y})$	$f(x, y)
0	0	$f(0,0) = (0+0)\overline{0}(0+\overline{0})$	0•1•1 = 0

- There are no possible variable assignments that make the expression evaluate to 1, therefore it is unsatisfiable.

2. $g(x,y,z) = (x+y+z)\overline{y}(\overline{x}+y)$

x	y	z	$(x+y+z)\overline{y}(\overline{x}+y)$	g(x, y, z)
1	1	1	$(1+1+1)\overline{1}(\overline{1}+1)$	1•0•1 = 0
1	1	0	$(1+1+0)\overline{1}(\overline{1}+1)$	1•0•1 = 0
1	0	1	$(1+0+1)\overline{0}(\overline{1}+0)$	1•1•0 = 0
1	0	0	$(1+0+0)\overline{0}(\overline{1}+0)$	1•1•0 = 0
0	1	1	$(0+1+1)\overline{1}(\overline{0}+1)$	1•0•1 = 0
0	1	0	$(0+1+0)\overline{1}(\overline{0}+1)$	1•0•1 = 0
0	0	1	$(0+0+1)\overline{0}(\overline{0}+0)$	1•1•1 = 1
0	0	0	$(0+0+0)\overline{0}(\overline{0}+0)$	0•1•1 = 0

- The expression g evaluates to 1 when (x, y, z) = (0, 0, 1), therefore the expression is satisfiable.

Satisfying Disjunctive Normal Form

- If a Boolean expression is in Disjunctive Normal Form (DNF), determining its satisfiability is straightforward.

- An expression in DNF is satisfiable if and only if there is a term that does not contain a variable and its own negation.

I.e. a Boolean expression in DNF is satisfiable if at least one term is not a contradiction.

Example: $x\bar{y}z\bar{x} + \overline{w}xy\bar{z} + \overline{w}xw\bar{x} + xyz\bar{z}$

- The second term of the expression, $\overline{w}xy\bar{z}$, does not contain a variable and its negation.
- The expression can be satisfied by assigning values of: $w = 0$, $x = 1$, $y = 1$, and $z = 0$.

Satisfying Conjunctive Normal Form

- Unlike DNF expressions, real-world problems are often naturally expressed in Conjunctive Normal Form (CNF) which makes it more difficult to determine their satisfiability.

- In CNF, each clause represents a constraint that must be satisfied.

- An individual clause is satisfied by a variable assignment if the assignment causes the clause to evaluate to 1.

- The entire CNF expression is the product of all the clauses, and it evaluates to 1 only if all clauses are satisfied.

Example: $(w + \overline{x} + z)(x + y + \overline{z})(\overline{x} + \overline{y} + z)(w + \overline{y} + \overline{z})$

Let w = 0, x = 0, y = 0, z = 0

(0 + 0 + 0)(0 + 0 + 1)(0 + 1 + 0)(0 + 1 + 1)
0•1•1•1 = 0

Let w = 1, x = 1, y = 1, z = 1

(1 + 1 + 1)(1 + 1 + 0)(1 + 0 + 1)(1 + 0 + 0)
1•1•1•1 = 1

- The expression is satisfiable when (w, x, y, z) = (1, 1, 1, 1)

Example: Scheduling Classes

- The problem of scheduling classes in a school can be formulated as a Boolean satisfiability problem.

For example

Consider a school with 26 classes, offered throughout the day across 5 periods.

Assuming that each class is represented by a unique starting character in the alphabet, we can represent the entire set of classes as a collection of alphabetic characters.

E.g.
{ Astronomy, ..., Zoology} ≡ { A, ..., Z }

Rules:

1. A class must be assigned to only one period.
2. All class must be assigned to some period.
3. Scheduling conflicts exist, and some classes must not belong to the same period.
   E.g. (A, B), (B, E), (D, C), and (C, E).

- As the collection of classes grows longer, it might be necessary to use more complex sorting algorithms that can take into account other factors such as distance between classes or the availability of teaching staff.

Solution

- To solve the class scheduling problem, a Boolean expression will be created, described by the set of rules established.

- Since there are 26 classes and 5 periods, there are 130 (26 • 5) class to period assignments required to describe a complete schedule.

- Every variable assignment represents a possible schedule, but not all combinations are valid due to conflicts.

Let
X_{an} = class a is scheduled for period n.
C_{ab} = class a is in conflict with class b.

E.g.
X_{U1} = underwater-basket-weaving is scheduled for period 1.
C_{FN} = fight-club is in conflict with nap-time.

- These variables can be put into a 2D matrix, which then describes a complete schedule.

- A Boolean expression will be created to determine if there are any conflicts in the class schedule.

- The expression is satisfiable only if a valid, conflict-free schedule can be found.

- The function will be a product of terms, with each term representing a constraint that must be satisfied for the schedule to be valid.

- Asserting that X_{an} must be assigned to a period, but must not be assigned to more than one period, constrains the expression:

  $(x_{B1} + \cdots + x_{B5})\overline{(x_{B1} \cdots x_{B5})}$

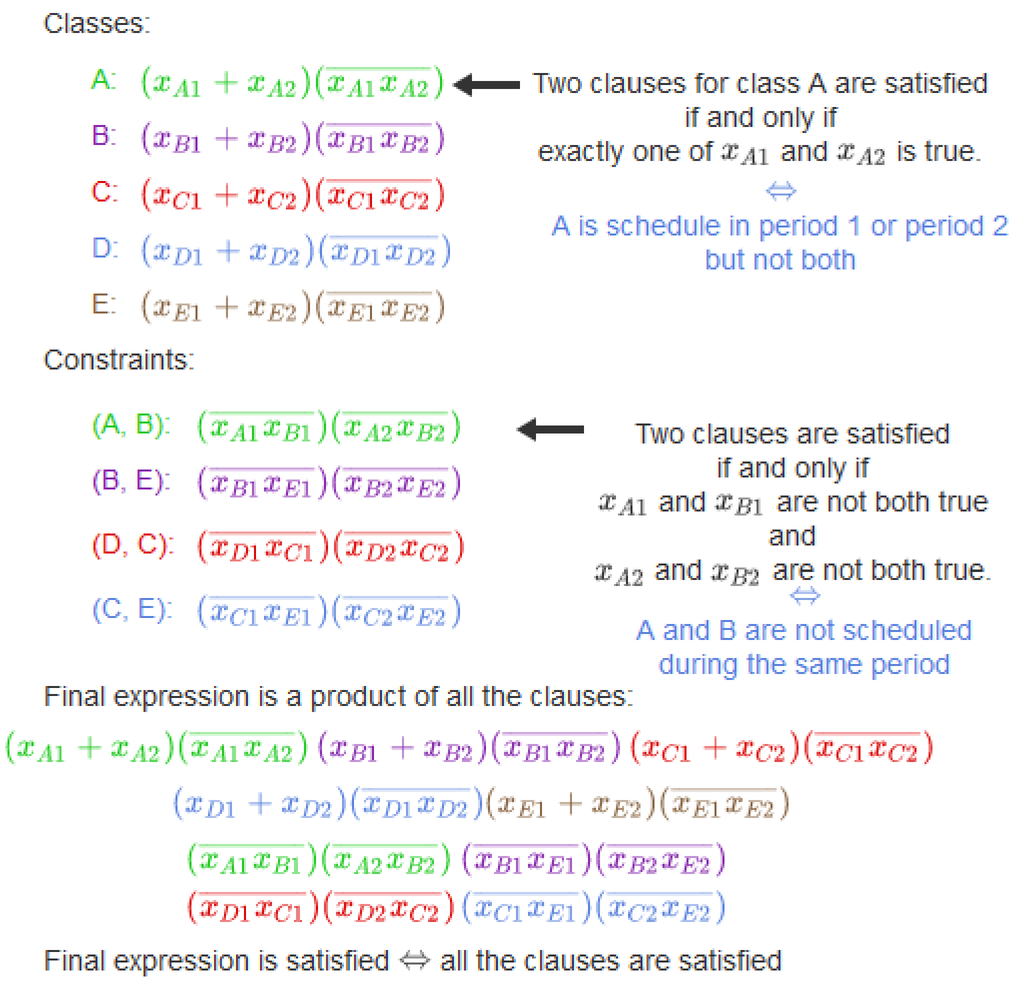
  If $x_{B1} = 1$ and $x_{B5} = 1$, then the expression evaluates to 0.

- Asserting that elements a and b of the tuple C_{ab} do not conflict with one another can be constrained the same way:

  $\overline{(x_{B1} \bullet x_{C1})} \bullet \overline{(x_{F1} \bullet x_{Z1})}$

  If $x_{B1} = 1$ and $x_{C1} = 1$, then the expression evaluates to 0.

Constructing an Expression



Classes:

A: $(x_{A1} + x_{A2})(\overline{x_{A1}x_{A2}})$

B: $(x_{B1} + x_{B2})(\overline{x_{B1}x_{B2}})$

C: $(x_{C1} + x_{C2})(\overline{x_{C1}x_{C2}})$

D: $(x_{D1} + x_{D2})(\overline{x_{D1}x_{D2}})$

E: $(x_{E1} + x_{E2})(\overline{x_{E1}x_{E2}})$

Two clauses for class A are satisfied if and only if exactly one of x_{A1} and x_{A2} is true. ⇔ A is schedule in period 1 or period 2 but not both

Constraints:

(A, B): $(\overline{x_{A1}x_{B1}})(\overline{x_{A2}x_{B2}})$

(B, E): $(\overline{x_{B1}x_{E1}})(\overline{x_{B2}x_{E2}})$

(D, C): $(\overline{x_{D1}x_{C1}})(\overline{x_{D2}x_{C2}})$

(C, E): $(\overline{x_{C1}x_{E1}})(\overline{x_{C2}x_{E2}})$

Two clauses are satisfied if and only if x_{A1} and x_{B1} are not both true and x_{A2} and x_{B2} are not both true. ⇔ A and B are not scheduled during the same period

Final expression is a product of all the clauses:

$$(x_{A1} + x_{A2})(\overline{x_{A1}x_{A2}})\,(x_{B1} + x_{B2})(\overline{x_{B1}x_{B2}})\,(x_{C1} + x_{C2})(\overline{x_{C1}x_{C2}})$$
$$(x_{D1} + x_{D2})(\overline{x_{D1}x_{D2}})(x_{E1} + x_{E2})(\overline{x_{E1}x_{E2}})$$
$$(\overline{x_{A1}x_{B1}})(\overline{x_{A2}x_{B2}})\,(\overline{x_{B1}x_{E1}})(\overline{x_{B2}x_{E2}})$$
$$(\overline{x_{D1}x_{C1}})(\overline{x_{D2}x_{C2}})\,(\overline{x_{C1}x_{E1}})(\overline{x_{C2}x_{E2}})$$

Final expression is satisfied ⇔ all the clauses are satisfied

The final expression is a product of all the clauses. The final expression is satisfied if and only if all the clauses are satisfied.

Credit: Zybooks

5.6 Gates and circuits

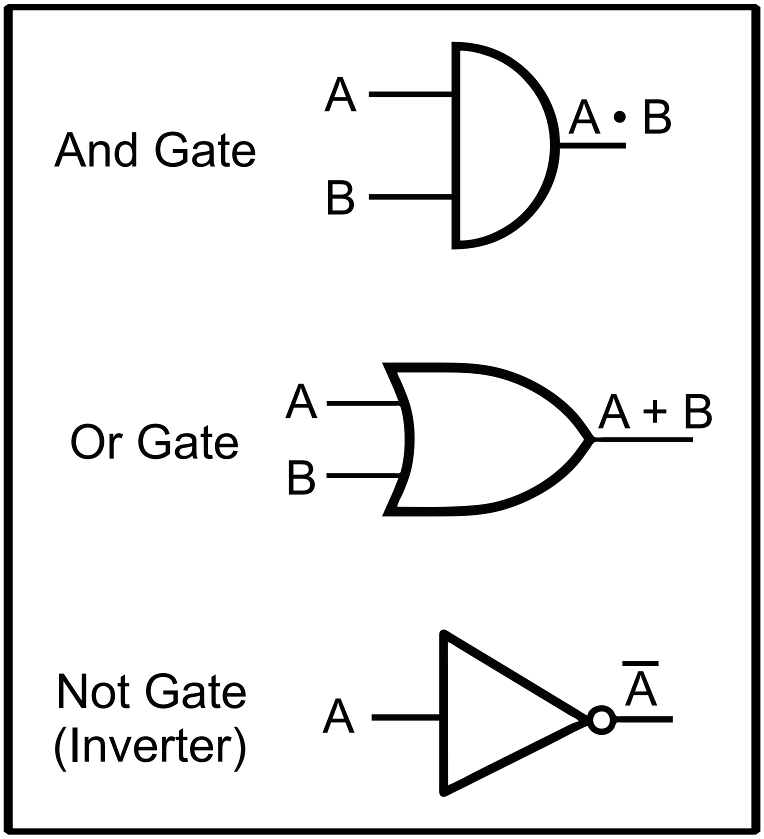
- Boolean algebra is used in the design and description of circuit, and circuits are constructed from electrical devices called gates.
- Gates receive Boolean input values and produce an output based on the input values, thereby implementing a simple Boolean function.

Three Basic Gates: And, Or, Not (Inverter)

- There are three fundamental types of gates:
    1. the AND gate computes Boolean multiplication,

2. the OR gate computes Boolean addition,
3. the inverter computes the complement.

- The shape of a gate indicates the specific operation that it performs.

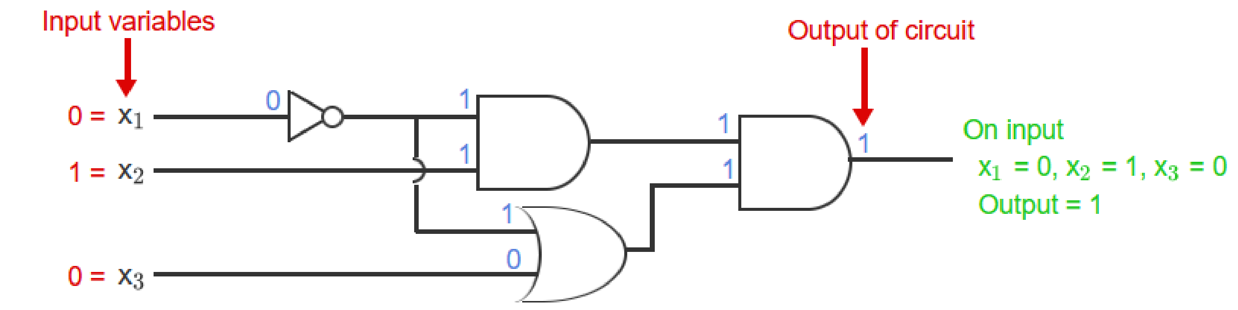


Expressing Functions as Gates

- Connecting gates together can enable computation of more complex functions.

- The output of one gate can serve as input to another gate, and the output of a gate can branch and feed into multiple gates.

- By designing and connecting gates in specific ways, complex computations can be performed within an electronic device.

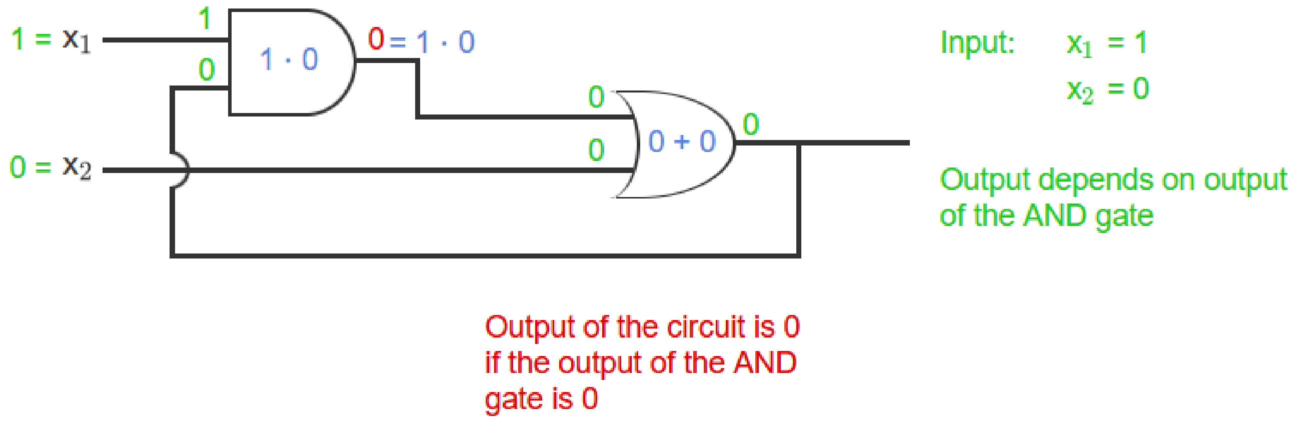


The output of the second AND gate is the output of the circuit. Therefore, on input $x_1 = 0, x_2 = 1$, and $x_3 = 0$, the output of the circuit is 1.

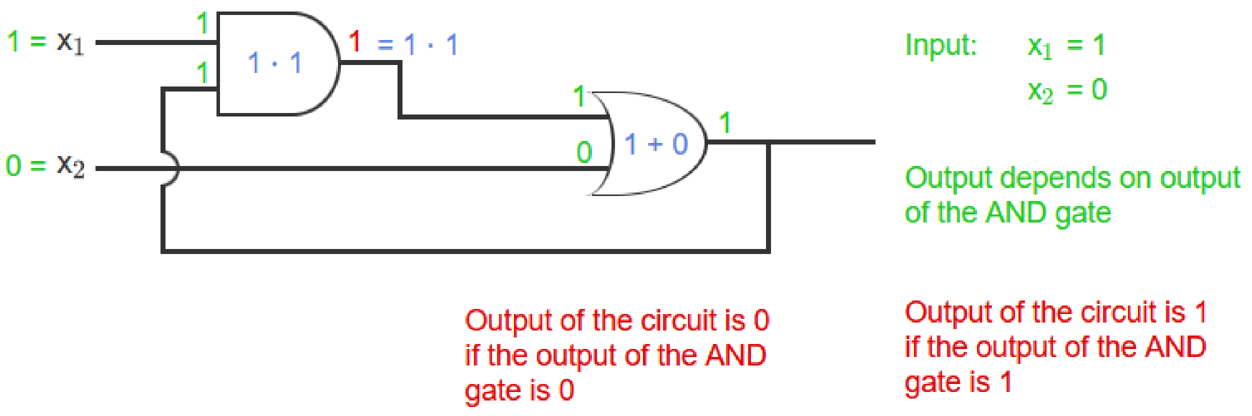
Combinatorial Circuits vs Stateful Circuitry

- Combinatorial circuits are a type of circuit with an output value that is dependent only on the current combination of input values.
- Combinatorial circuits do not store information over time and therefore cannot be stateful.
- The distinction can be illustrated with an example of a non-combinatorial circuit, which can store one bit of information and has the ability to stabilize over time.

Storing 1-Bit


1 = x_1
0 = x_2
1
0
1 · 0
0 = 1 · 0
0
0
0 + 0
0
Input: x_1 = 1
x_2 = 0
Output depends on output of the AND gate
Output of the circuit is 0 if the output of the AND gate is 0

On input $x_1 = 1$ and $x_2 = 0$, the output of the circuit depends on the output of the AND gate. When the output of the AND gate is 0, the output of the circuit is 0.



Input: $x_1 = 1$

$x_2 = 0$

Output depends on output of the AND gate

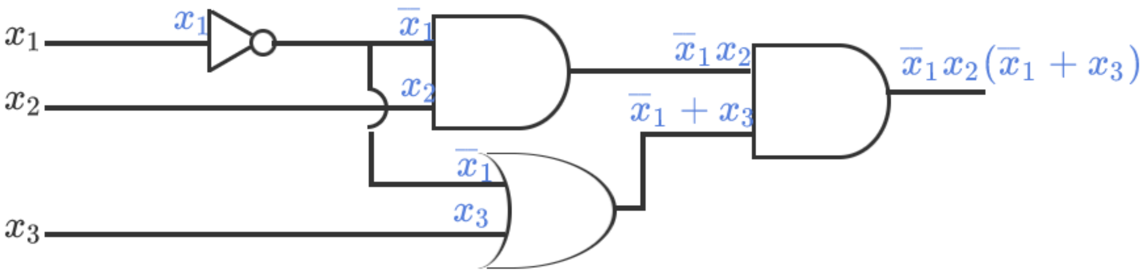
Output of the circuit is 0 if the output of the AND gate is 0

Output of the circuit is 1 if the output of the AND gate is 1

When the output of the AND gate is 1, the output of the circuit is 1.

Constructing Boolean Functions from Circuits

- Every combinatorial circuit calculates a Boolean function.



The circuit computes the function:

$$f(x_1, x_2, x_3) = \overline{x}_1 x_2(\overline{x}_1 + x_3)$$

Therefore, the circuit computes the function $f(x_1, x_2, x_3) = \overline{x}_1 \cdot x_2(\overline{x}_1 + x_3)$.

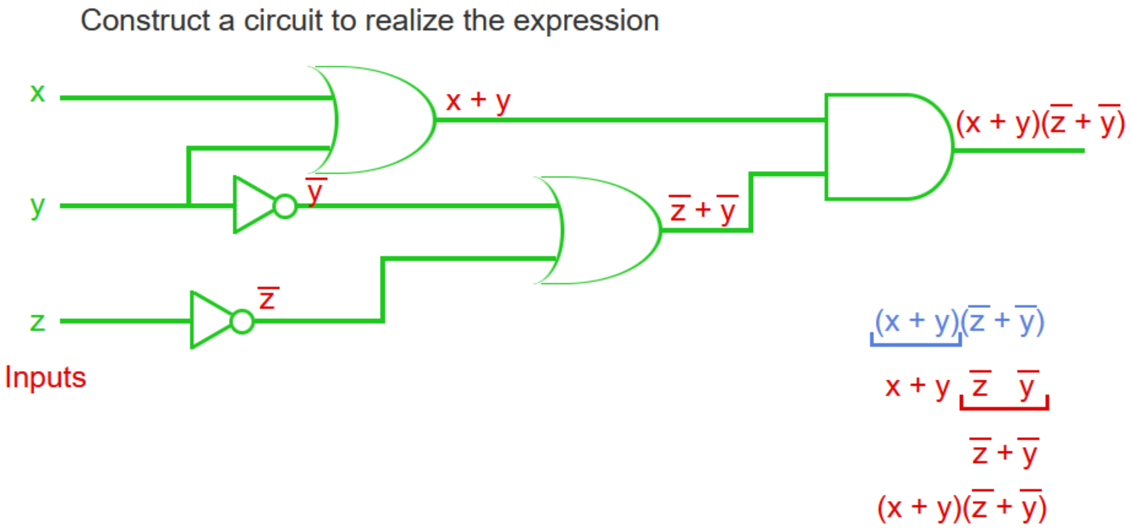
Constructing Circuits from Boolean Functions

- Any Boolean expression can be implemented by a circuit such that the input/output behavior of the circuit matches the function specified by the expression.

- A systematic approach for creating a diagram of the corresponding circuit based on a Boolean expression involves arranging input variables on the left side of the diagram.

- Each operation in the Boolean expression is then applied by adding a gate that performs that operation to the diagram, following the standard precedence rules.

- Input wires to the gate originate from the gates or input values where the operation is applied, and the output wire is labeled with the outcome of the Boolean operation.


Construct a circuit to realize the expression
x
y
z
Inputs
$x + y$
$\overline{y}$
$\overline{z}$
$\overline{z} + \overline{y}$
$(x + y)(\overline{z} + \overline{y})$
$(x + y)(\overline{z} + \overline{y})$
$x + y$ $\overline{z}$ $\overline{y}$
$\overline{z} + \overline{y}$
$(x + y)(\overline{z} + \overline{y})$

Finally, the outputs of the two OR gates are routed to an AND gate whose output is $(x + y)(\overline{z} + \overline{y})$.

Designing circuits

Circuit design typically begins with an informal description of the desired function, followed by creating an input/output table describing every possible input combination.

Steps:

1. Describe the circuit in terms of inputs and outputs.
2. Create an input/output table to ensure the circuit works for every possible input combination.
3. Develop a Boolean expression equivalent to the table.
4. Construct the circuit based on the Boolean expression.
5. (Optional) Optimize the circuit.

Circuit: the Sum of Two Bits in Binary

Consider a circuit that computes the sum of two bits and outputs the result in binary. The circuit has two input variables, x and y, representing the bits to be added in base-10.

1. Circuit Description

   - Input variables: x and y.
   - Output variables: m and l.
   - The concated output is a 2-bit binary number.
   - Addition operation used is the usual addition, 1 + 1 = 2 in this context.

2. Input/Output Table

   - Specifies desired input/output relationships for each output variable.
   - Output labeled 'm' denotes most significant (or high-order) bit of sum.
   - Output labeled 'l' denotes least significant (or low-order) bit of sum.

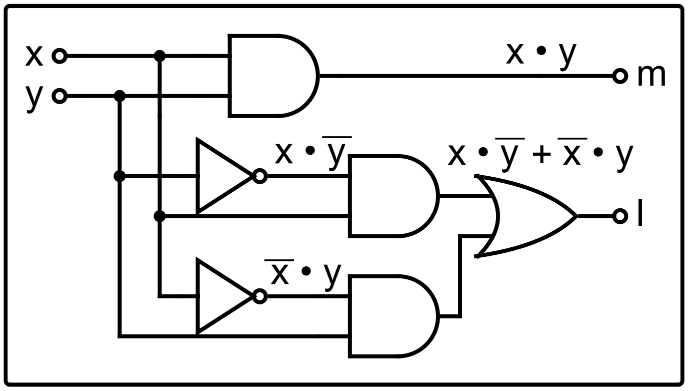
x	y	m	l	x + y
x	**y**	**m**	**l**	**x + y**
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	0	2

3. Develop Boolean Expression(s)

   The circuit has two outputs, so there must be two expressions describing their behavior.
   a) $m = x \cdot y$
   b) $l = (x \cdot \overline{y}) + (\overline{x} \cdot y)$

4. Construct the Circuit

- Without considering optimization, the following diagram describes a circuit that will add two binary inputs algebraically.

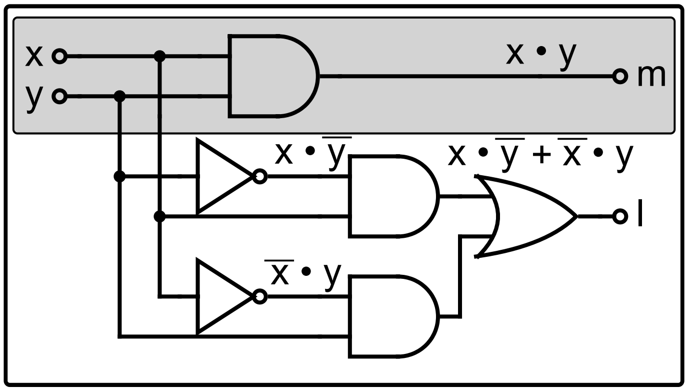


5. Optimize the Circuit

- The previous diagram can be optimized by reducing the number of components needed to achieve the same results.

- Computational work is done by each component and the output of a component can be split to feed into multiple inputs.

- Consider the AND gate responsible for the expression $x \cdot y$.



- The work to compute $x \cdot y$ is done with the AND gate here, so using that output twice could lead to optimization.

- Find an expression that uses the term $x \cdot y$ and is equivalent to $(x \cdot \overline{y}) + (\overline{x} \cdot y)$

$$(x \cdot \overline{y}) + (\overline{x} \cdot y)$$

$$\overline{\overline{(x \cdot \overline{y})} \cdot \overline{(\overline{x} \cdot y)}} \equiv \overline{(\overline{x} + y) \cdot (x + \overline{y})} \quad \text{De Morgan's Law}$$

$$\overline{(z) \cdot (x + \overline{y})} \quad \text{Let } z = \overline{x} + y$$

$$\overline{(z \cdot x) + (z \cdot \overline{y})} \quad \text{Distributive And Over Or}$$

$$\overline{((\overline{x} + y) \cdot x) + ((\overline{x} + y) \cdot \overline{y})} \quad \text{Substitute } z$$

$$\overline{(\overline{x} \cdot x) + (x \cdot y) + (\overline{x} \cdot \overline{y}) + (y \cdot \overline{y})} \quad \text{Distributive And Over Or}$$

$$\overline{(x \cdot y) + (\overline{x} \cdot \overline{y})} \quad \text{Disjunction of Contradictions}$$

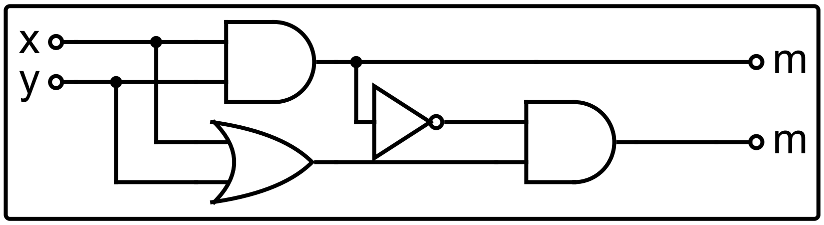
$$\overline{(x \cdot y)} \cdot \overline{(\overline{x} \cdot \overline{y})} \quad \text{De Morgan's Law}$$

$$\overline{(x \cdot y)} \cdot (x + y) \quad \text{De Morgan's Law}$$

- The final expression is equivalent to the initial expression, and contains the term $x \cdot y$.

- The optimized expression has fewer operators than the initial expression.

- With optimization, the following diagram describes a circuit that will add two binary inputs algebraically, using fewer components than before.


x
y
m
m