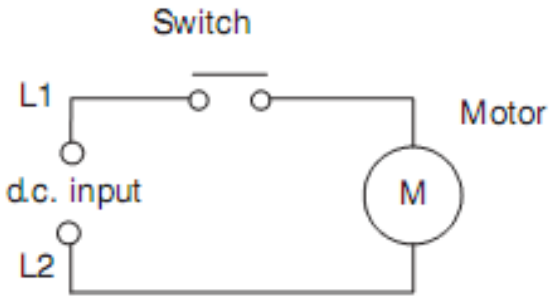
Chapter Four

Ladder Programming

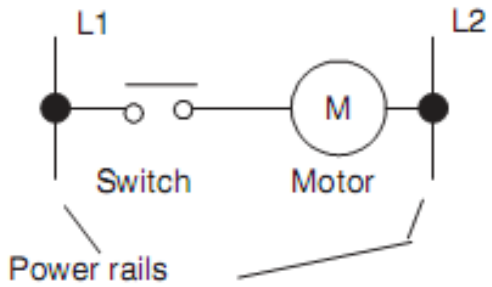
This chapter is an introduction to programming a PLC using ladder diagrams.

Ladder Diagrams

As an introduction to ladder diagrams, consider the simple wiring diagram for an electrical circuit in the figure.



We can redraw this diagram in a different way, using two vertical lines to represent the input power rails and stringing the rest of the circuit between them. The result is a circuit termed **ladder diagram**.



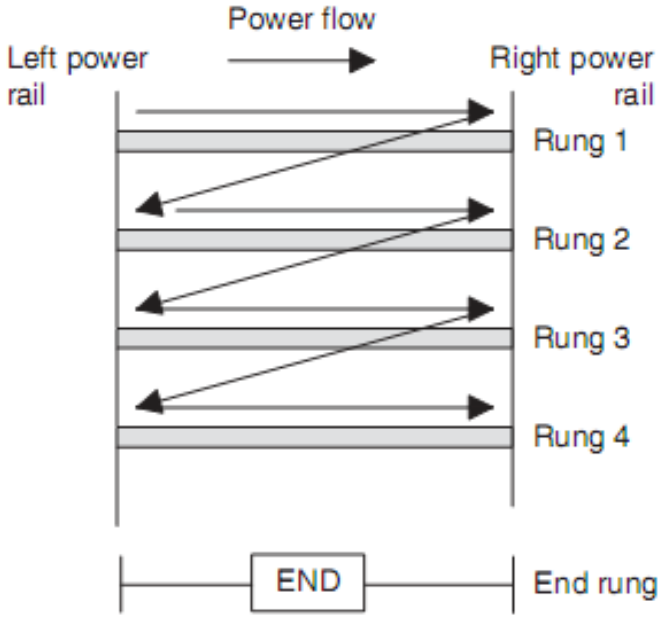
The power lines, or rails, as they are often called, are like the vertical sides of a ladder, with the horizontal circuit lines similar to the rungs of the ladder.

PLC Ladder Programming (LAD)

A very commonly used method of programming PLCs is based on the use of ladder diagrams.

In drawing a ladder diagram, **certain conventions** are adopted:

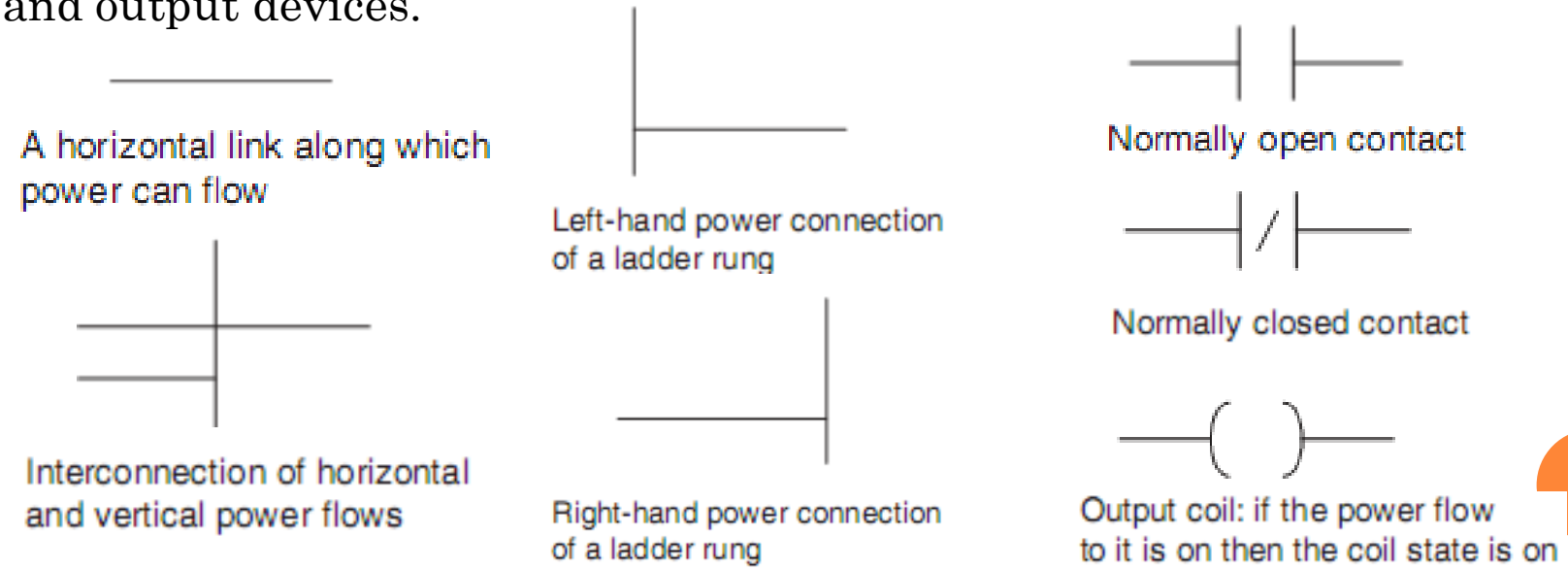
- The vertical lines of the diagram represent the power rails between which circuits are connected. The **power flow is taken to be from the left-hand vertical across a rung**.
- Each rung on the ladder defines **one operation** in the control process.
- A ladder diagram is read from **left to right** and from **top to bottom**. The figure shows the scanning motion employed by the PLC.
- When the PLC is in its run mode, it goes through the entire ladder program to the end, the end rung of the program might be indicated by a block with the word END or RET to return the program to its beginning.



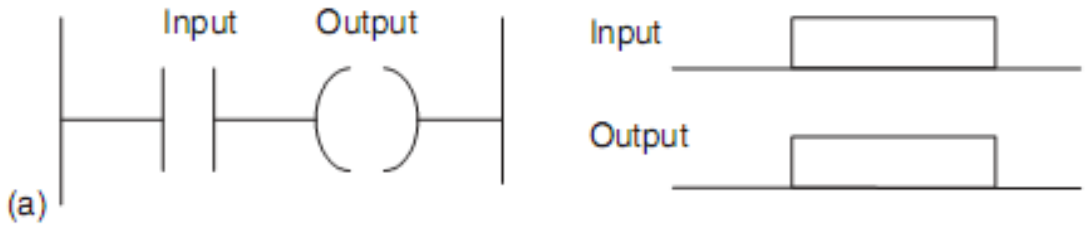
- Each rung must start with an **input or inputs** and must end with at **least one output**.

- As the program is scanned, the outputs are not updated instantly, but the results stored in memory and **all the outputs are updated simultaneously at the end of the program scan**.
- Electrical devices are shown in their **normal condition**. Thus a switch that is normally open until some object closes it is shown as open on the ladder diagram.
- A **particular device can appear in more than one rung of a ladder**. For example, we might have a relay that switches on one or more devices. The same letters and/or numbers are used to label the device in each situation.

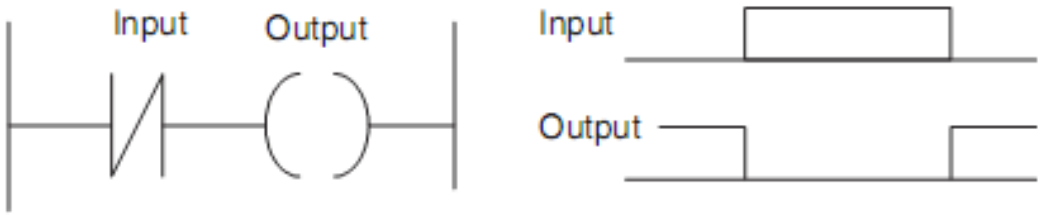
The figure shows **standard IEC 1131-3 symbols** that are used for input and output devices.



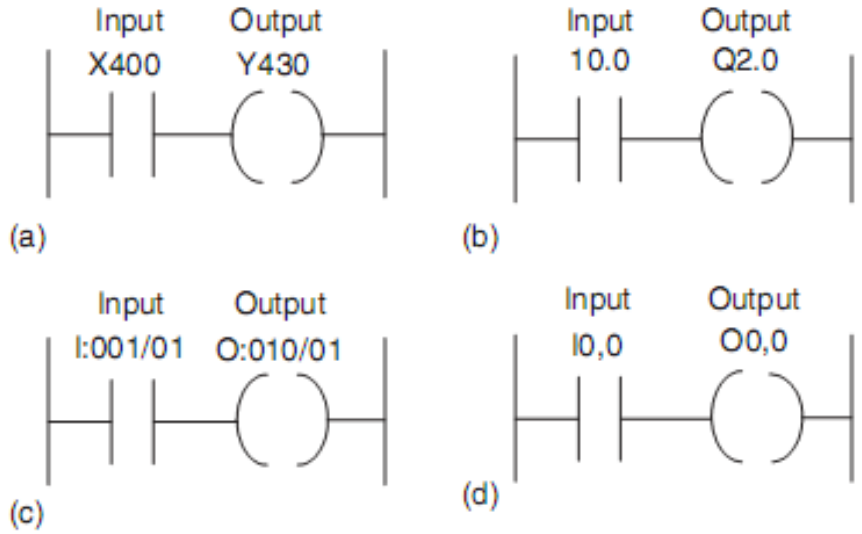
To illustrate the drawing of the rung of a ladder diagram, consider a situation where energizing an output device, such as a motor, depends on a **normally open** start switch. When the switch is closed, that is, there is an input, the output of the motor is activated.



With a **normally closed** switch |/| there will be an output until that switch was opened. The output will be off when the input is activated.



In drawing ladder diagrams, the names of the associated variable and addresses of each element are appended to its symbol. Thus the next figure shows how the ladder diagram of previous example would appear using (a) **Mitsubishi,** (b) **Siemens**, (c) **Allen-Bradley**, and (d) **Telemecanique** notations for the addresses.



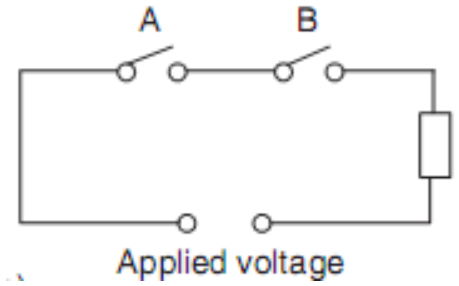
Notation: (a) Mitsubishi, (b) Siemens, (c) Allen-Bradley, and (d) Telemecanique.

Logic Functions

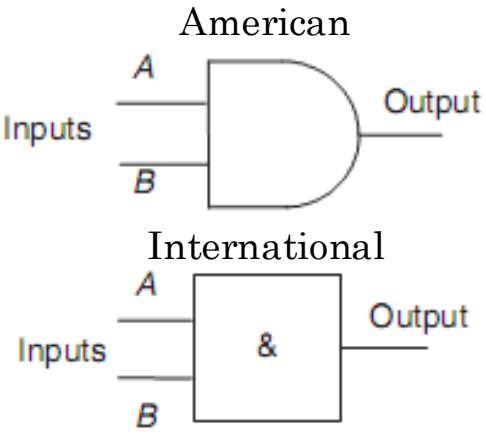
There are many control situations requiring actions to be initiated when a **certain combination of conditions** is realized.

1- AND

An output is not energized unless two normally open switches A and B are both closed.



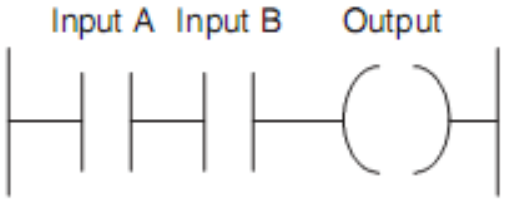
Electrical circuit



Logic gate symbol

Inputs		Output
A	B	
0	0	0
0	1	0
1	0	0
1	1	1

Truth table

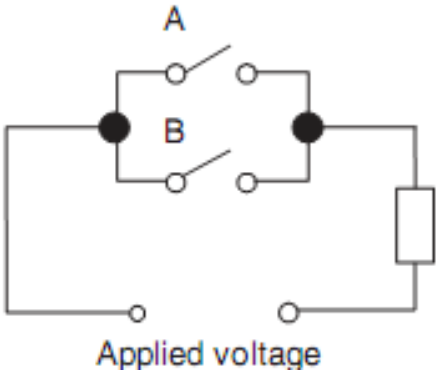


Ladder diagram

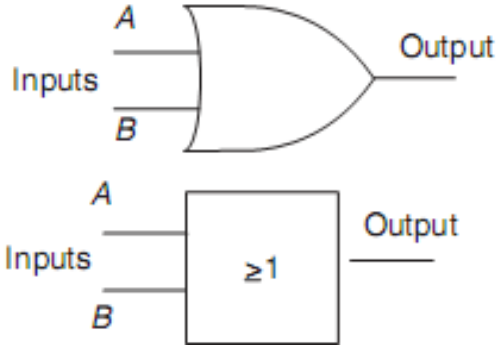
"**On a ladder diagram, contacts in a horizontal rung, that is, contacts in series, represent the logical AND operations**."

2- OR

An output is energized when switch A, B or, both are closed.



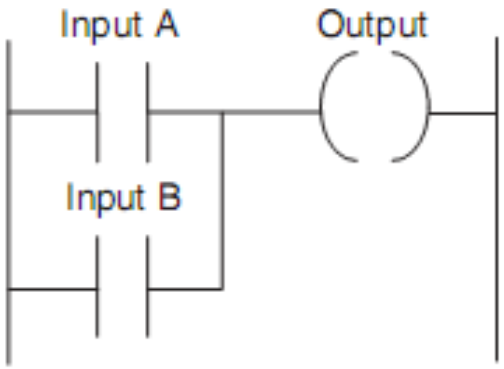
Electrical circuit



Logic gate symbol

Inputs		Output
A	B	
0	0	0
0	1	1
1	0	1
1	1	1

Truth table

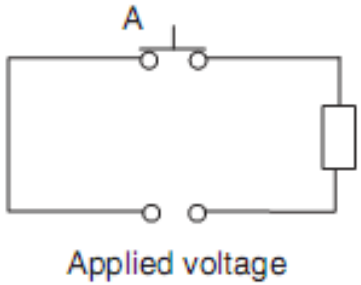


Ladder diagram

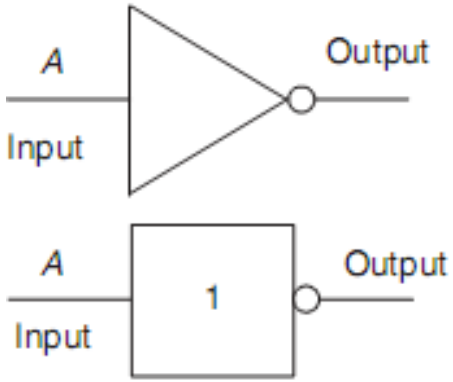
"**Paths in parallel, represent logical OR operations**."

3- NOT

There is an output when there is no input and no output when there is an input. The gate is sometimes referred to as an **inverter**.



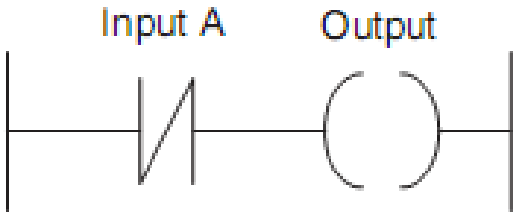
Electrical circuit



Logic gate symbol

Input	
A	Output
0	1
1	0

Truth table

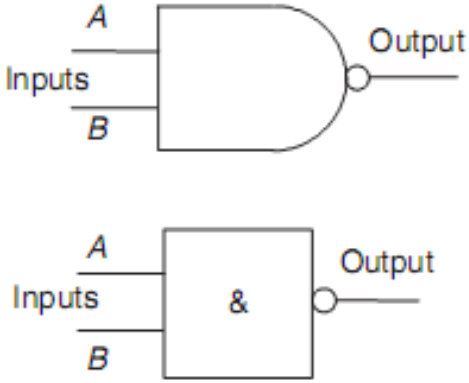


Ladder diagram

4- NAND

NAND gate is an AND gate followed by a NOT gate. The consequence of having the NOT gate is to invert all the outputs from the AND gate. **Either input A or input B (or both) have to be 0 for there to be a 1 output**. When both inputs A and input B are 1, the output is 0.

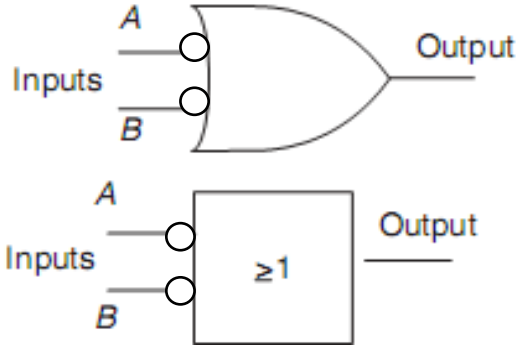
Logic gate symbol

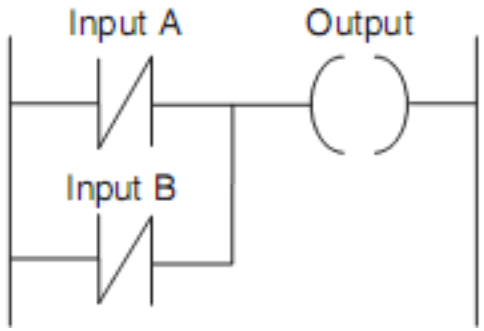


Truth table

Inputs		Output
A	B	
0	0	1
0	1	1
1	0	1
1	1	0

An alternative that gives exactly the same result is to put a NOT gate on each input and then follow that with an OR gate.



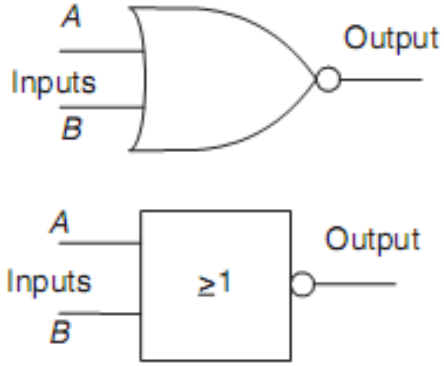


Ladder diagram

5- NOR

NOR gate is an OR gate followed by a NOT gate. The consequence of having the NOT gate is to invert the outputs of the OR gate**. There is an output when neither input A nor input B is 1**.

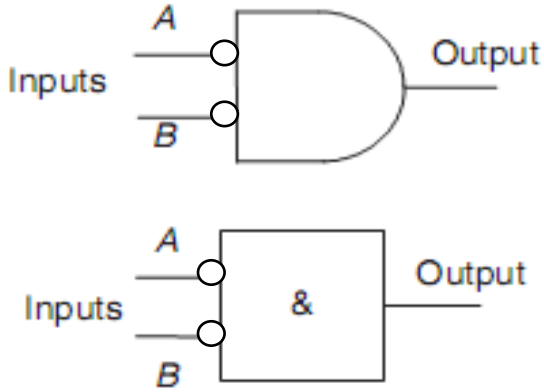
Logic gate symbol

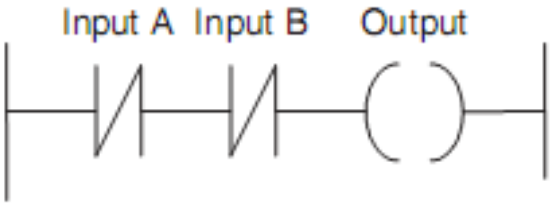


Truth table

Inputs		Output
A	B	
0	0	1
0	1	0
1	0	0
1	1	0

An alternative, which gives exactly the same results, is to put a NOT gate on each input and then an AND gate.



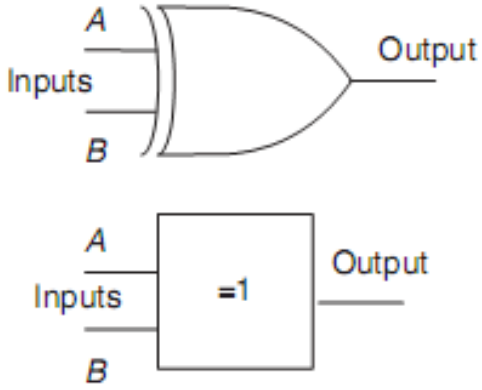


Ladder diagram

6- Exclusive OR (XOR)

The OR gate gives an output when either or both of the inputs are 1. However, sometimes there is a need for a **gate that gives an output when either of the inputs is 1 but not when both are 1**.

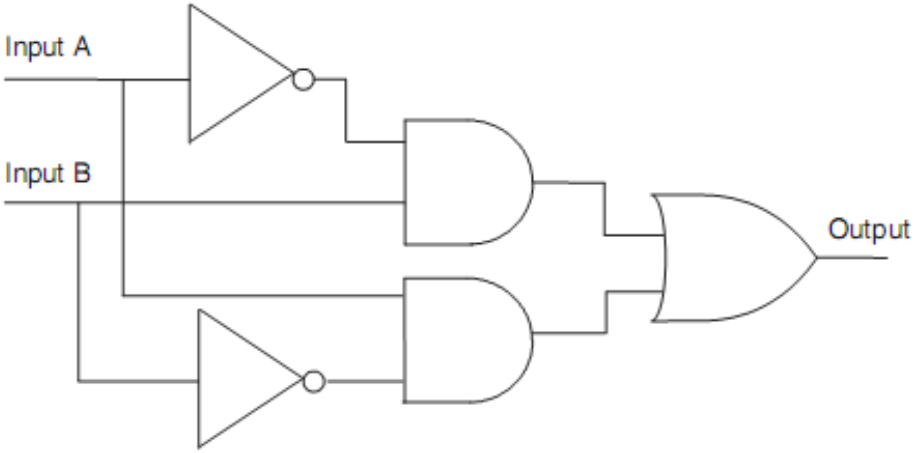
Logic gate symbol

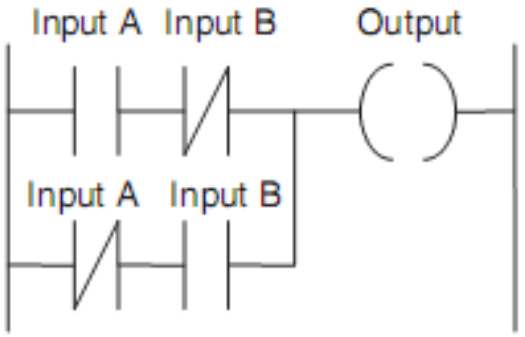


Truth table

Inputs		Output
A	B	
0	0	0
0	1	1
1	0	1
1	1	0

One way of obtaining such a gate is by using NOT, AND, and OR gates as shown in figure.



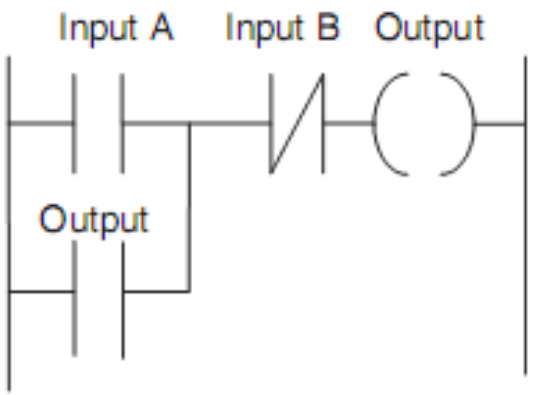


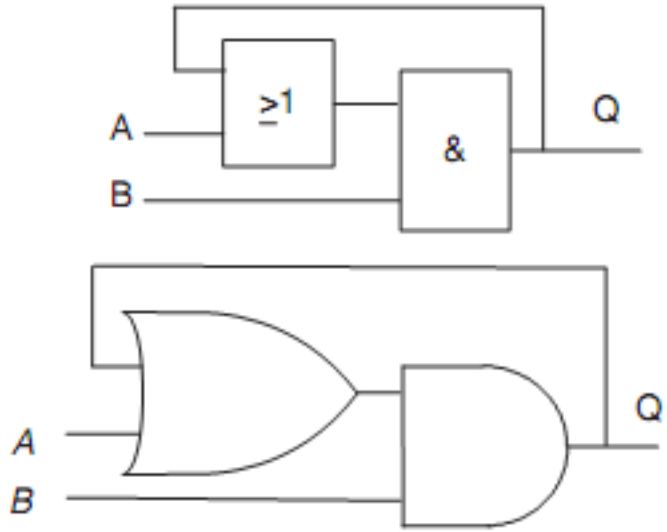
Ladder diagram

Latching

There are often situations in which **it is necessary to hold an output energized, even when the input ceases**. The term **latch circuit** is used for the circuit that carries out such an operation.

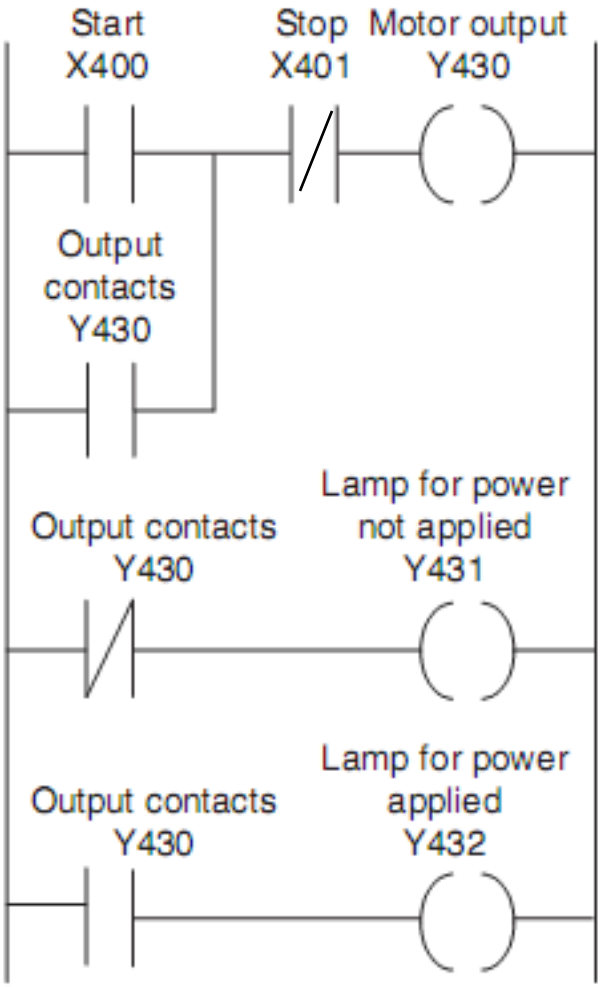
An example of a latch circuit is shown in figure. When the input A contacts close, there is an output. However, when there is an output, **another set of contacts associated with the output closes**. These contacts form an OR logic gate system with the input contacts. Thus, even if input A opens, the circuit will still maintain the output energized. The only way to release the output is by operating the normally closed contact B.





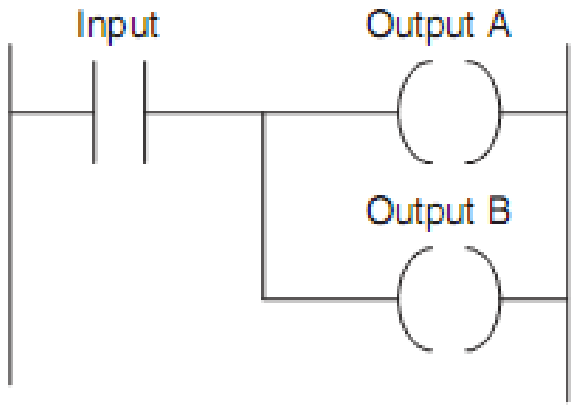
As an illustration of the application of a latching circuit, **consider a motor controlled by stop and start push-button switches and for which one signal light must be illuminated when the power is applied to the motor and another when it is not applied.**

X401 is closed when the program is started. When X400 is momentarily closed, Y430 is energized and its contacts close. This results in latching as well as the switching off of Y431 and the switching on of Y432. To switch the motor off, X401 is pressed and opens. Y430 contacts open in the top rung and third rung but close in the second rung. Thus Y431 comes on and Y432 goes off

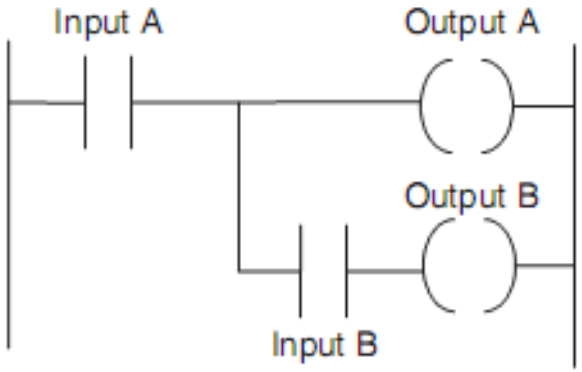


Multiple Outputs

With ladder diagrams, there can be more than one output connected to a contact. The figure shows a ladder program with two output coils. When the input contacts close, both the coils give outputs.

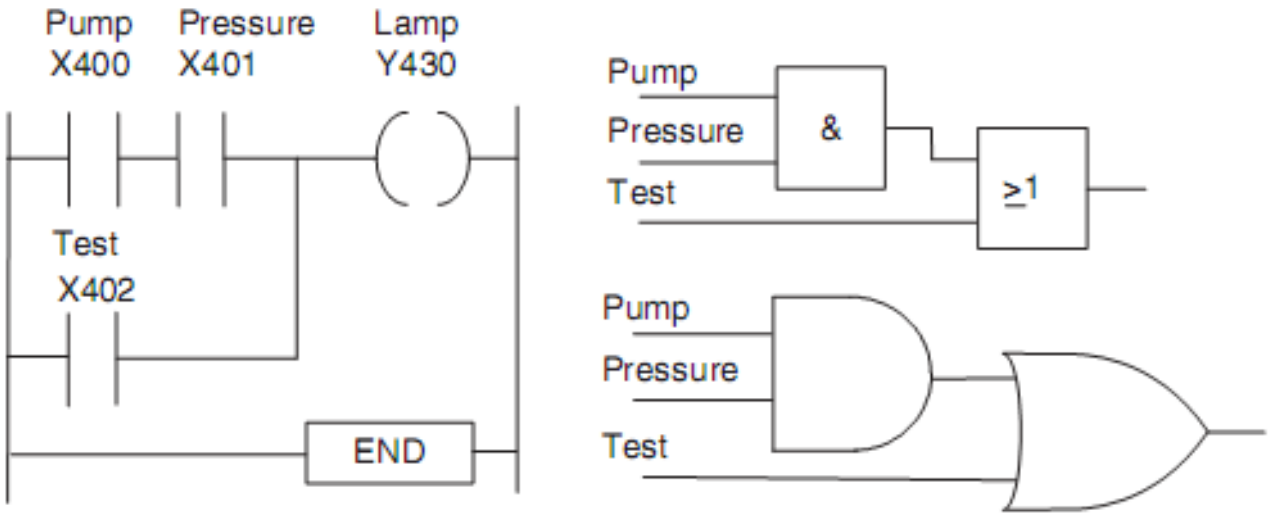


For the ladder rung shown in the figure, output A occurs when input A occurs. Output B occurs only when both input A and input B occur.



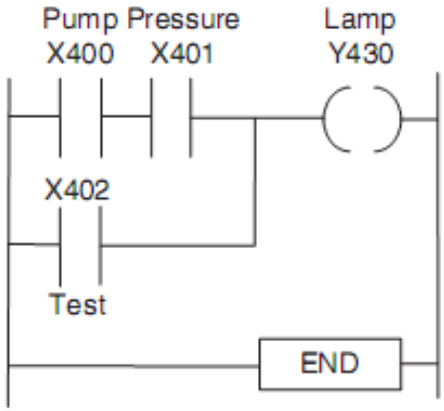
Example

A signal lamp is required to be switched on if **a pump is running** **and** the **pressure is satisfactory**, **or** if the **lamp test switch is closed**.



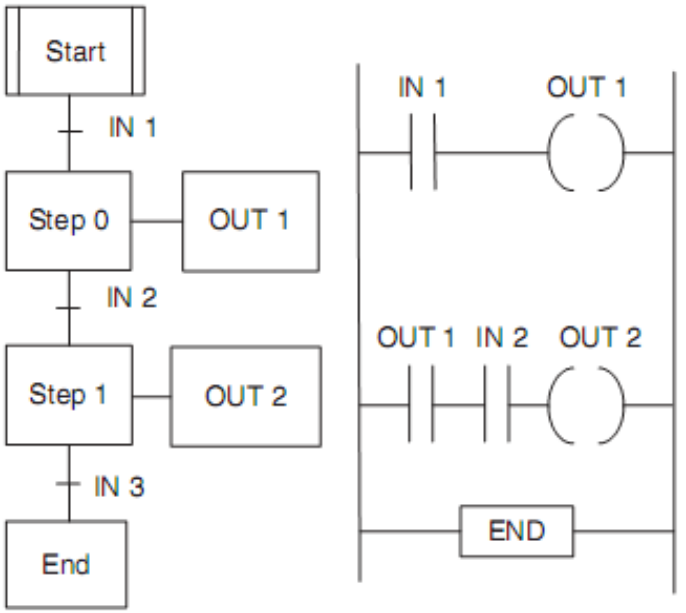
Other Programming Languages

Other languages that are less popular include **instruction list (IL)**, **sequential function chart (SFC),** and **structured text (ST).**



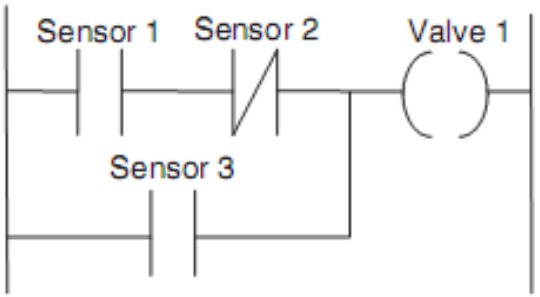

```
LD      X400
AND     X401
LD      X402
ORB
OUT     Y430
END
```

instruction list (IL),



sequential function chart (SFC),

structured text (ST).




```
Valve_1 := (Sensor_1 AND NOT Sensor_2)
           OR Sensor_3

IF Sensor_1 AND NOT Sensor_2 THEN
    Valve_1 := 1;
ELSEIF Sensor_3 THEN
    Valve_1 := 1
END_IF
```

Boolean Algebra

Ladder programs can be derived from Boolean expressions since we are concerned with a mathematical system of logic.

In Boolean algebra there are just two digits, 0 and 1. When we have an **AND operation** for inputs A and B, we can write:

$$A \cdot B = Q$$

The **OR operation** for inputs A and B is written as:

$$A + B = Q$$

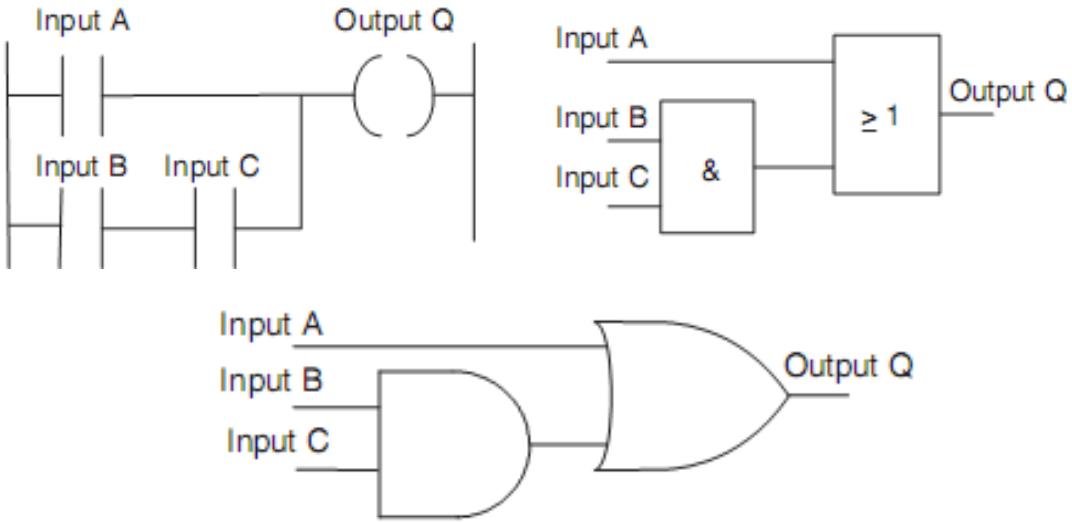
The **NOT operation** for an input A is written as:

$$\bar{A} = Q$$

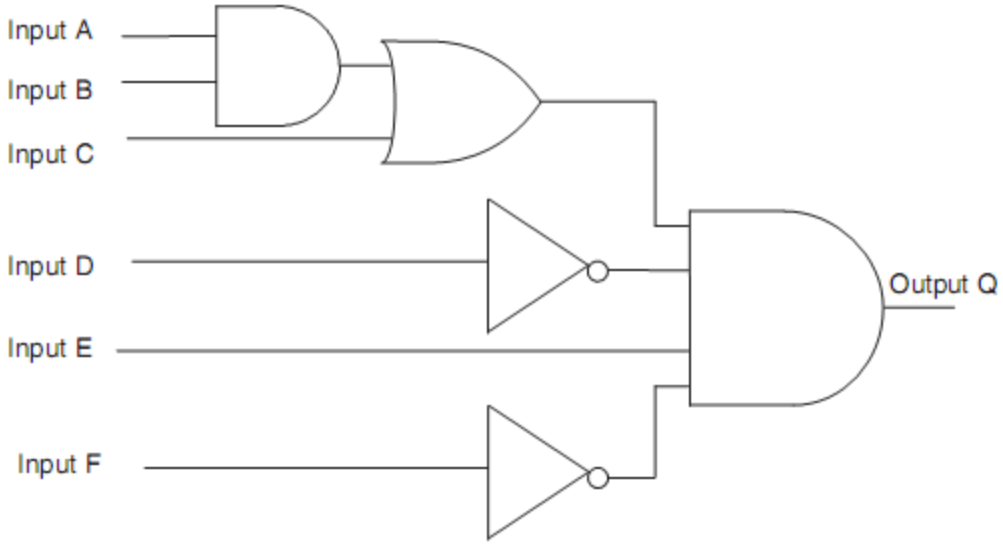
As an illustration of how we can relate Boolean expressions with ladder diagrams, consider the expression:

$$A + B \cdot C = Q$$

The figure shows the ladder diagram and the corresponding logic circuit.

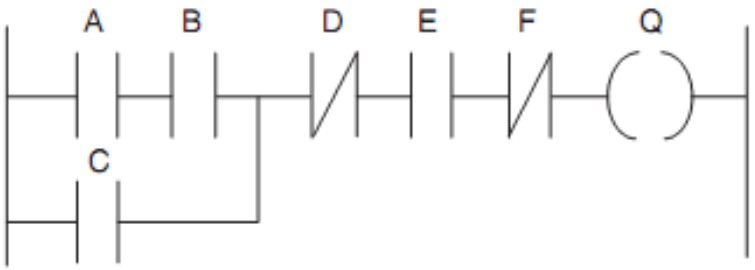


Consider a logic diagram with many inputs, as shown in figure below.



Its representation by a Boolean expression and a ladder rung.

$$(A \cdot B + C) \cdot \bar{D} \cdot E \cdot \bar{F} = Q$$



Axioms

The ***axioms*** (or *postulates*) of a mathematical system are a **minimal set of basic definitions that we assume to be true**, from which all other information about the system can be derived. The next table lists the Boolean algebra axioms.

Table 1

(A1)	$1 + 1 = 1$	(A1′)	$0 \cdot 0 = 0$
(A2)	$0 + 0 = 0$	(A2′)	$1 \cdot 1 = 1$
(A3)	$1 + 0 = 0 + 1 = 1$	(A3′)	$0 \cdot 1 = 1 \cdot 0 = 0$

Single-variable Theorems

During the analysis or synthesis of logic circuits, we often write algebraic expressions that characterize a circuit's actual or desired behavior. Boolean algebra *theorems* are statements that **allow us to manipulate algebraic expressions to get simpler ones**. For example, the theorem $X + 0 = X$ allows us to substitute every occurrence of $X + 0$ in an expression with X. The next table lists Boolean algebra theorems involving a single variable X. Try to prove it yourself.

Table 2

(T1)	$X + 0 = X$	(T1′)	$X \cdot 1 = X$
(T2)	$X + 1 = 1$	(T2′)	$X \cdot 0 = 0$
(T3)	$X + X = X$	(T3′)	$X \cdot X = X$
(T4)	$X + X' = 1$	(T4′)	$X \cdot X' = 0$
(T5)	$(X')' = X$		

Two- And Three Variable Theorems

Boolean algebra theorems with two or three variables are listed in the next table. Each of these theorems is easily proved by evaluating the theorem statement for the four possible combinations of X and Y, or the eight possible combinations of X, Y, and Z.

Table 3

Commutative Laws	(T6)	$A + B = B + A$	(T6′)	$A \cdot B = B \cdot A$
Associative Laws	(T7)	$A + (B + C) = (A + B) + C$	(T7′)	$A \cdot (B \cdot C) = (A \cdot B) \cdot C$
Distributive Laws	(T8)	$A \cdot (B + C) = A \cdot B + A \cdot C$	(T8′)	$A + B \cdot C = (A + B) \cdot (A + C)$
DeMorgan's Laws	(T9)	$(A + B)' = A' \cdot B'$	(T9′)	$(A \cdot B)' = A' + B'$
Absorption Laws	(T10)	$X + X \cdot Y = X$	(T10′)	$X \cdot (X + Y) = X$
Combining	(T11)	$X \cdot Y + X \cdot Y' = X$	(T11′)	$(X + Y) \cdot (X + Y') = X$

The first two theorem pairs (T6, T6′) and (T7, T7′) concern commutativity and associativity of logical addition and multiplication and are identical to the commutative and associative laws for addition and multiplication of integers and reals. Taken together, they indicate that the parenthesization or order of terms in a logical sum or logical product is irrelevant.

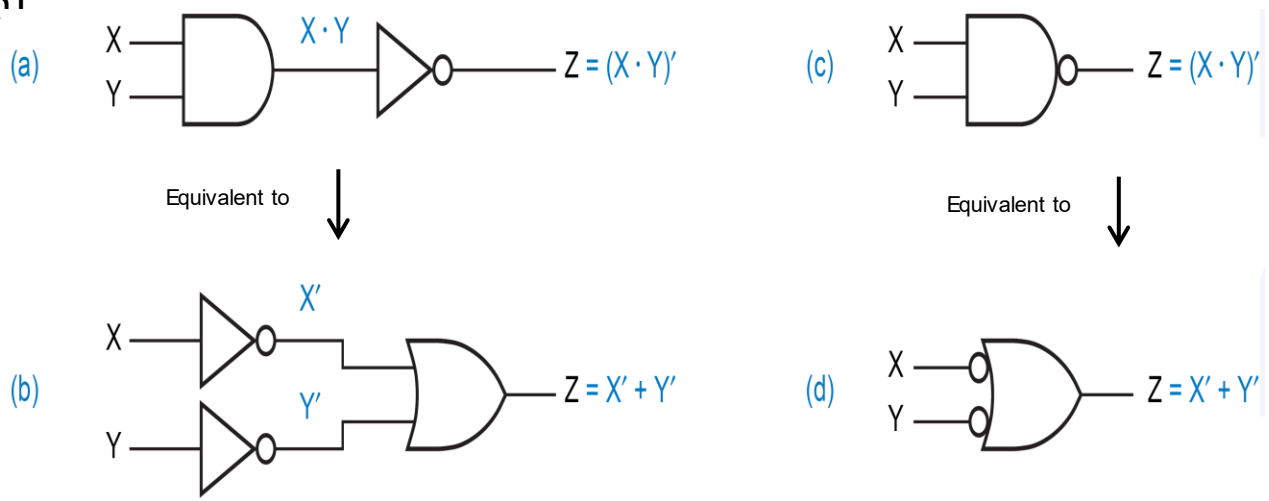
Theorem T8 is identical to the distributive law for integers and reals—that is, logical multiplication distributes over logical addition. Hence, we can "**multiply out**" an expression to obtain a **sum-of-products** form, as in the example below:

$$V \cdot (W + X) \cdot (Y + Z) = V \cdot W \cdot Y + V \cdot W \cdot Z + V \cdot X \cdot Y + V \cdot X \cdot Z$$

However, Boolean algebra also has the unfamiliar property that the reverse is true—logical addition distributes over logical multiplication as demonstrated by theorem T8'. Thus, we can also "**add out**" an expression to obtain a **product of-sums** form:

$$(V \cdot W \cdot X) + (Y \cdot Z) = (V + Y) \cdot (V + Z) \cdot (W + Y) \cdot (W + Z) \cdot (X + Y) \cdot (X + Z)$$

DeMorgan's Laws (T9 and T9′) are probably the most commonly used of all the theorems of Boolean algebra. These theorems apply to any number of inputs. Theorem T9′ simply says that an *n*-input AND gate whose output is inverted is equivalent to an *n*-input OR gate whose inputs are inverted. That is, the circuits of the figure (a) and (b) or (c) and (d) are equivalent



Equivalent circuits according to DeMorgan's theorem T9′.

Theorems (T10, T10′) and (T11, T11′) are used extensively in the minimization of logic functions. For example, if the subexpression X + X·Y appears in a logic expression, the *absorption theorem* T10 says that we need only include X in the expression.

Order Of Operation

The order of priority in Boolean expression is **NOT first**, **AND second**, and **OR last**, unless otherwise indicated by grouping signs, such as parentheses, brackets, or braces. According to these rules, in the expression A + B · C, B is ANDed with C first then the result is ORed with A.

Duality

We stated all of the axioms of Boolean algebra in pairs (e.g., (A1) and (A1′)). The primed version (′) of each axiom is obtained from the unprimed version by simply swapping (0) and (1) and, if present, (·) and (+). As a result, we can state the following ***metatheorem***, a theorem about theorems:

Principle of Duality **Any theorem in Boolean algebra remains true if (0) and (1) are swapped and (·) and (+) are swapped throughout**.

The foregoing axioms and theorems of the Boolean algebra are used in analysis and synthesis of digital circuits. It can be also used to simplify the logic expressions. The following example illustrates this:

Examples

Simplify the following Boolean functions to a minimum number of literals.

1. $Z = X + X \cdot Y$

Ans.

$$Z = X \cdot 1 + X \cdot Y$$
$$= X \cdot (1 + Y)$$
$$= X \cdot 1$$
$$= X$$

2. $Z = X \cdot (X' + Y)$

Ans.

$$Z = X \cdot X' + X \cdot Y$$
$$= 0 + X \cdot Y$$
$$= X \cdot Y$$

3. $W = X' \cdot Y' \cdot Z + X' \cdot Y \cdot Z + X \cdot Y'$

Ans.

$$W = X' \cdot Z \cdot (Y' + Y) + X \cdot Y'$$
$$= X' \cdot Z + X \cdot Y'$$

4. $W = X \cdot Y + X' \cdot Z + Y \cdot Z$

Ans.

$$W = X \cdot Y + X' \cdot Z + Y \cdot Z \cdot (X + X')$$
$$= X \cdot Y + X' \cdot Z + X \cdot Y \cdot Z + X' \cdot Y \cdot Z$$
$$= X \cdot Y \cdot (1 + Z) + X' \cdot Z \cdot (1 + Y)$$
$$= X \cdot Y + X' \cdot Z$$

5. $Z = X + X' \cdot Y$

Ans.

$$Z = (X + X')(X + Y)$$
$$= 1 \cdot (X + Y)$$
$$= X + Y$$

Standard Representations of Logic Functions

1. Truth table

The most basic representation of a logic function is the ***truth table***. This representation simply lists the output of the circuit for every possible input combination. Traditionally, the input combinations are arranged in rows in ascending binary counting order, and the corresponding output values are written in a column next to the rows. The general structure of a 3-variable truth table is shown in table below.

Row	X	Y	Z	F
0	0	0	0	F(0,0,0)
1	0	0	1	F(0,0,1)
2	0	1	0	F(0,1,0)
3	0	1	1	F(0,1,1)
4	1	0	0	F(1,0,0)
5	1	0	1	F(1,0,1)
6	1	1	0	F(1,1,0)
7	1	1	1	F(1,1,1)

Table 4 General truth table structure for a 3-varible logic function, F(X,Y,Z)

The rows are numbered 0–7 corresponding to the binary input combinations, but this numbering is not an essential part of the truth table. The truth table for a particular 3-variable logic function is shown in table 5. Each distinct pattern of 0s and 1s in the output column yields a different logic function; there are 28 such patterns. Thus, the logic function shown in the table is one of **28** different logic functions of three variables.

Row	X	Y	Z	F
0	0	0	0	1
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	1
5	1	0	1	0
6	1	1	0	1
7	1	1	1	1

Table 5 Truth table for a particular 3-varible logic function, F(X,Y,Z)

The truth table for an n-variable logic function has **2^n rows**. Obviously, truth tables are practical to write only for logic functions with a small number of variables.

The information contained in a truth table can also be conveyed algebraically. To do so, we first need some definitions:

- A **literal** is a variable or the complement of a variable. Examples: X, Y, X′, Y′.
- A **product term** is a single literal or a logical product of two or more literals. Examples: Z′, W·X·Y, X·Y′·Z, W′·Y′·Z.

- A **sum-of-products** expression is a logical sum of product terms. Example:
- $Z' + W \cdot X \cdot Y + X \cdot Y' \cdot Z + W' \cdot Y' \cdot Z$.
- A **sum term** is a single literal or a logical sum of two or more literals. Examples: Z', $W + X + Y$, $X + Y' + Z$, $W' + Y' + Z$.
- A **product-of-sums** expression is a logical product of sum terms. Example:
- $Z' \cdot (W + X + Y) \cdot (X + Y' + Z) \cdot (W' + Y' + Z)$.
- A **normal term** is a product or sum term in which no variable appears more than once. A nonnormal term can always be simplified to a constant or a normal term using one of Boolean algebra theorems. Examples of nonnormal terms: $W \cdot X \cdot X \cdot Y'$, $W + W + X' + Y$, $X \cdot X' \cdot Y$. Examples of normal terms: $W \cdot X \cdot Y'$, $W + X' + Y$.
- An n-variable **minterm** is a normal product term with n literals. There are 2^n such product terms. Examples of 4-variable minterms: $W' \cdot X' \cdot Y' \cdot Z'$, $W \cdot X \cdot Y' \cdot Z$, $W' \cdot X' \cdot Y \cdot Z'$.
- An n-variable **maxterm** is a normal sum term with n literals. There are 2^n such sum terms. Examples of 4-variable maxterms: $W' + X' + Y' + Z'$, $W + X' + Y' + Z$, $W' + X' + Y + Z'$.

There is a close correspondence between the truth table and minterms and maxterms. A **minterm** can be defined as a product term that is 1 in exactly one row of the truth table. Similarly, a **maxterm** can be defined as a sum term that is 0 in exactly one row of the truth table. The next table shows this correspondence for a 3-variable truth table.

Table 6

Row	X	Y	Z	F	*Minterm*	*Maxterm*
0	0	0	0	F(0,0,0)	X′·Y′·Z′	X + Y + Z
1	0	0	1	F(0,0,1)	X′·Y′·Z	X + Y + Z′
2	0	1	0	F(0,1,0)	X′·Y·Z′	X + Y′ + Z
3	0	1	1	F(0,1,1)	X′·Y·Z	X + Y′ + Z′
4	1	0	0	F(1,0,0)	X·Y′·Z′	X′ + Y + Z
5	1	0	1	F(1,0,1)	X·Y′·Z	X′ + Y + Z′
6	1	1	0	F(1,1,0)	X·Y·Z′	X′ + Y′ + Z
7	1	1	1	F(1,1,1)	X·Y·Z	X′ + Y′ + Z′

An n-variable minterm can be represented by an n-bit integer, the minterm number. We'll use the name minterm i to denote the minterm corresponding to row i of the truth table. In minterm i, a particular variable appears inverted (′) if the corresponding bit in the binary representation of i is 0; otherwise, it is not

inverted. For example, row 5 has binary representation 101 and the corresponding minterm is $X \cdot Y' \cdot Z$. As you might expect, the correspondence for maxterms is just the opposite: in maxterm i, a variable appears inverted if the corresponding bit in the binary representation of i is 1. Thus, maxterm 5 (101) is $X' + Y + Z'$.

Based on the correspondence between the truth table and minterms, we can easily create an algebraic representation of a logic function from its truth table. The ***canonical sum*** of a logic function is a sum of the minterms corresponding to truth-table rows (input combinations) for which the function produces a 1 output. For example, the canonical sum for the logic function in Table 5 is

$$F = \sum_{X,Y,Z}(0,3,4,6,7) = X' \cdot Y' \cdot Z' + X' \cdot Y \cdot Z + X \cdot Y' \cdot Z' + X \cdot Y \cdot Z' + X \cdot Y \cdot Z$$

Here, the notation $\sum_{X,Y,Z}(0,3,4,6,7)$ is a ***minterm list*** and means "the sum of minterms 0, 3, 4, 6, and 7 with variables X, Y, and Z." The minterm list is also known as the *on-set* for the logic function. You can visualize that each minterm turns on the output for exactly one input combination. Any logic function can be written as a canonical sum.

The **canonical product** of a logic function is a product of the maxterms corresponding to input combinations for which the function produces a 0 output. For example, the canonical product for the logic function in Table 5 is

$$F = \prod_{X,Y,Z}(1,2,5) = (X + Y + Z') \cdot (X + Y' + Z) \cdot (X' + Y + Z')$$

Here, the notation $\prod_{X,Y,Z}(1,2,5)$ is a ***maxterm list*** and means "the product of maxterms 1, 2, and 5 with variables X, Y, and Z." The maxterm list is also known as the *off-set* for the logic function. You can visualize that each maxterm turns off the output for exactly one input combination. Any logic function can be written as a canonical product.

It's easy to **convert between a minterm list and a maxterm list**. For a function of n variables, the possible minterm and maxterm numbers are in the set $\{0, 1, \ldots, 2^{n-1}\}$; a minterm or maxterm list contains a subset of these numbers. To switch between list types, take the set complement, for example,

$$\sum_{A,B,C}(0,1,2,3) = \prod_{A,B,C}(4,5,6,7)$$
$$\sum_{X,Y}(1) = \prod_{X,Y}(0,2,3)$$
$$\sum_{W,X,Y,Z}(0,1,2,3,5,7,11,13) = \prod_{W,X,Y,Z}(4,6,8,9,10,12,14,15)$$

We have now learned five possible representations for a logic function:

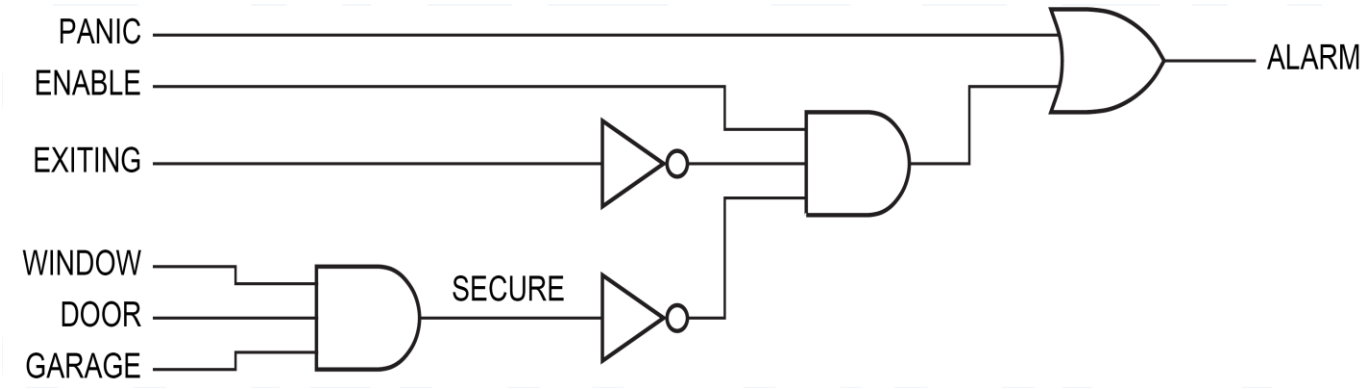
1. A truth table.
2. An algebraic sum of minterms, the canonical sum.
3. A minterm list using the S notation.
4. An algebraic product of maxterms, the canonical product.
5. A maxterm list using the P notation.

Each one of these representations specifies exactly the same information; given any one of them, we can derive the other four.

More often, we describe a logic function using the **English-language** connectives "and," "or," and "not." For example, we might describe an alarm circuit by saying, "The ALARM output is 1 if the PANIC input is 1, or if the ENABLE input is 1, the EXITING input is 0, and the house is not secure; the house is secure if the WINDOW, DOOR, and GARAGE inputs are all 1." Such a description can be translated directly into algebraic expressions:

$$\begin{aligned} ALARM &= PANIC + ENABLE \cdot EXITING' \cdot SECURE' \\ SECURE &= WINDOW \cdot DOOR \cdot GARAGE \\ ALARM &= PANIC + ENABLE \cdot EXITING' \cdot (WINDOW \cdot DOOR \cdot GARAGE)' \end{aligned}$$

We can easily draw a circuit using AND, OR, and NOT gates that realizes the final expression, as shown in figure below.

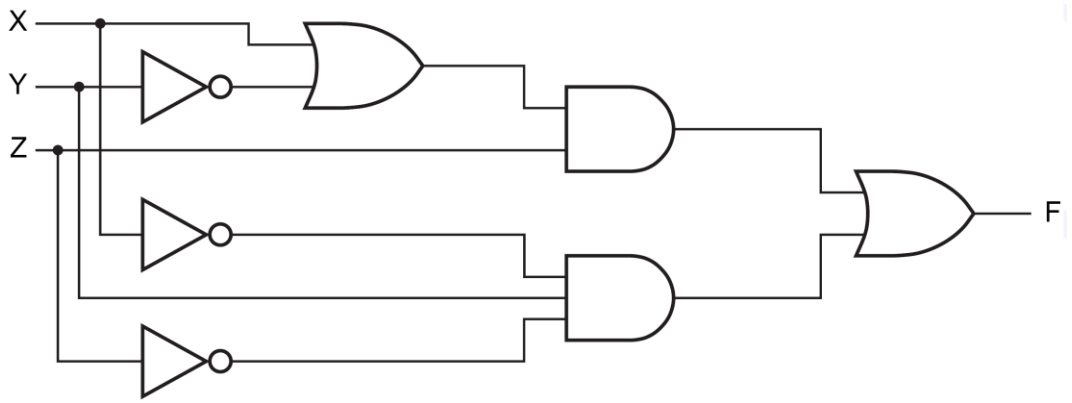


Logic Circuit Analysis

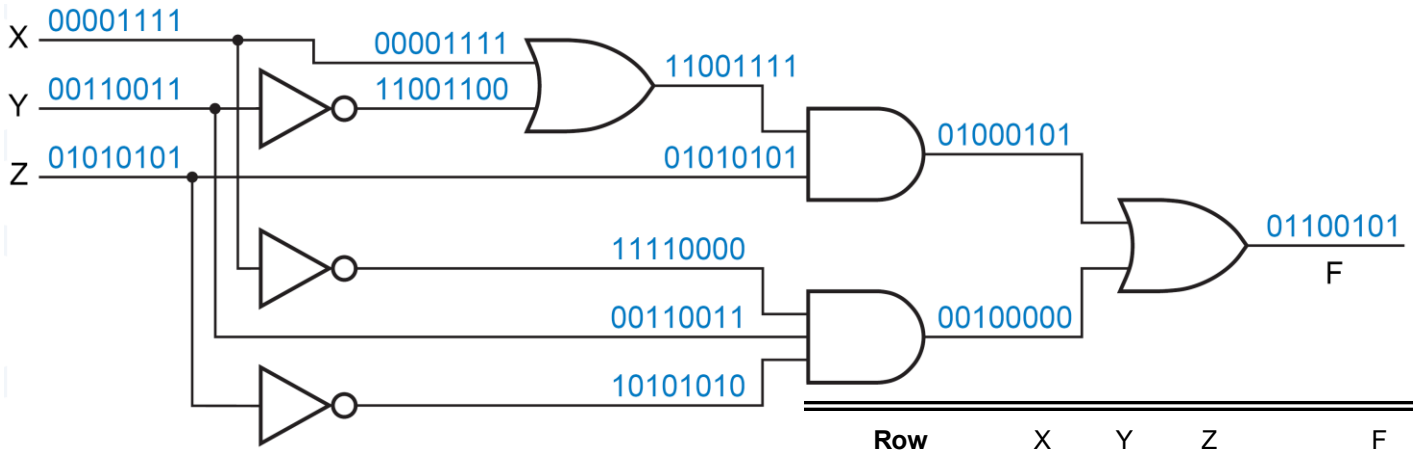
We analyze a logic circuit by obtaining a formal description of its logic function. Once we have a description of the logic function, a number of other operations are possible:

- We can determine the behavior of the circuit for various input combinations.
- We can manipulate an algebraic description to suggest different circuit structures for the logic function.
- We can use an algebraic description of the circuit's functional behavior in the analysis of a larger system that includes the circuit.

Given a logic diagram for a circuit, such as in the shown figure, there are a number of ways to obtain a formal description of the circuit's function. The most primitive functional description is the **truth table**.



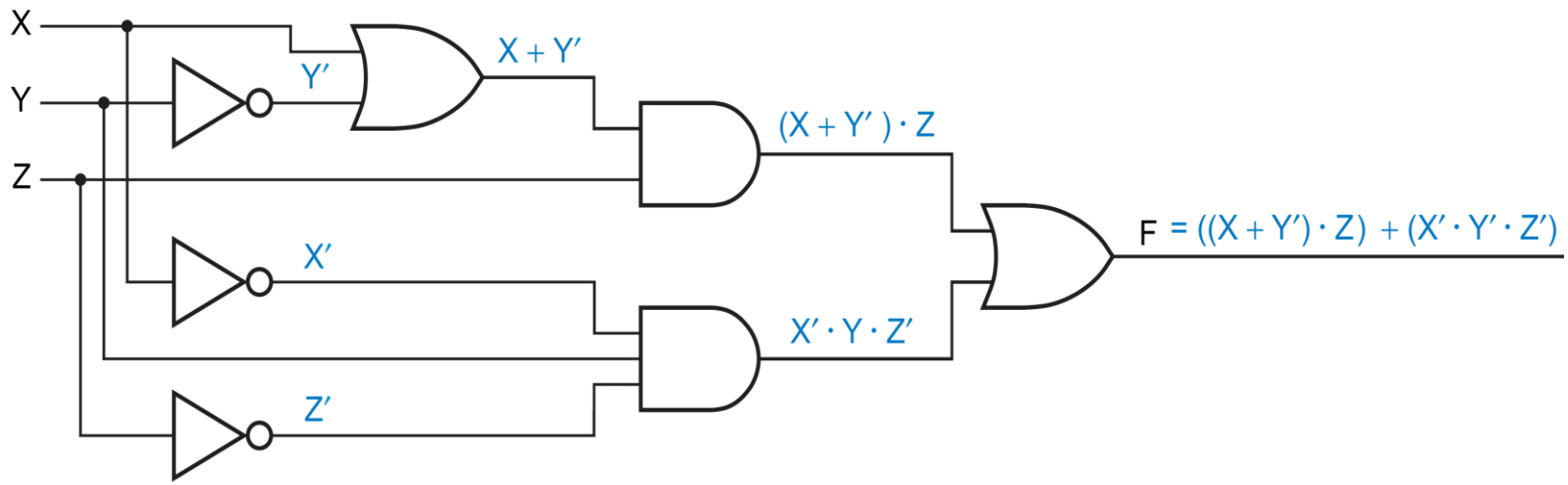
Using only the basic axioms of Boolean algebra, we can obtain the truth table of an n-input circuit by working our way through all 2^n input combinations. For each input combination, we determine all of the gate outputs produced by that input, propagating information from the circuit inputs to the circuit outputs. The figure below applies this "exhaustive" technique to our example circuit. Written on each signal line in the circuit is a sequence of eight logic values.



The truth table can be written by transcribing the output sequence of the final OR gate, as shown in the next table. Once we have the truth table for the circuit, we can also directly write a logic expression—the canonical sum or product—if we wish.

Row	X	Y	Z	F
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	0
4	1	0	0	0
5	1	0	1	1
6	1	1	0	0
7	1	1	1	1

The number of input combinations of a logic circuit grows exponentially with the number of inputs, 2^n, so the exhaustive approach can quickly become exhausting. Instead, we normally use an algebraic approach whose complexity is more linearly proportional to the size of the circuit. The method is simple—we build up a **parenthesized logic expression** corresponding to the logic operators and structure of the circuit. We start at the circuit inputs and propagate expressions through gates toward the output. Using the theorems of Boolean algebra, we may simplify the expressions as we go, or we may defer all algebraic manipulations until an output expression is obtained.



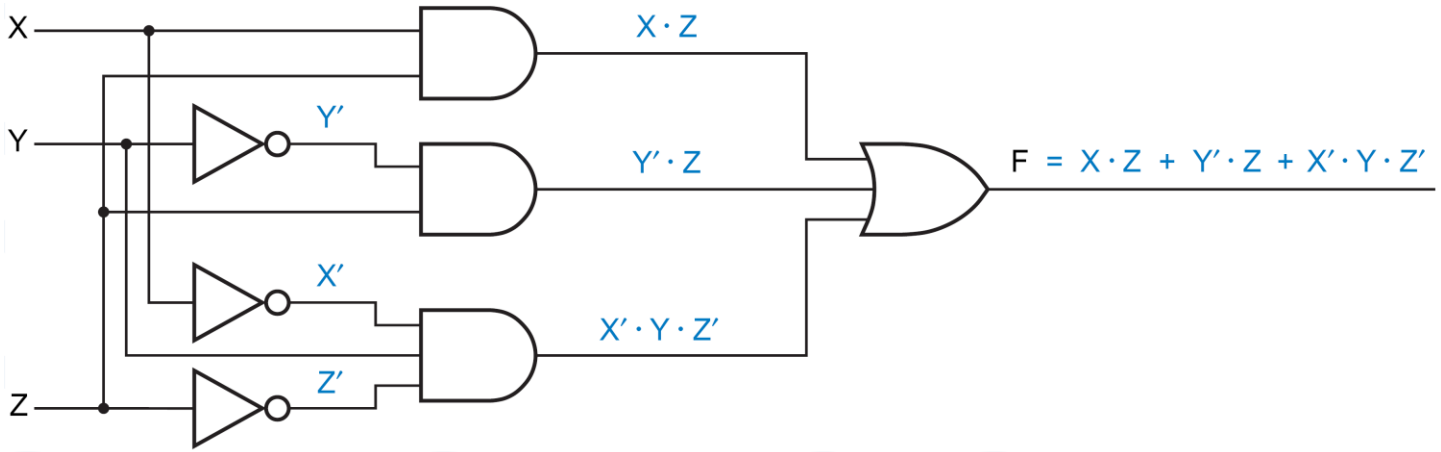
The figure above applies the algebraic technique to our example circuit. The output function is given on the output of the final OR gate:

$$F = ((X + Y')\cdot Z) + (X'\cdot Y\cdot Z')$$

No Boolean-algebra theorems were used in obtaining this expression. However, we can use theorems to transform this expression into another form. For example, a ***sum of products*** can be obtained by "**multiplying out**" (using theorem T8) :

$$F = X{\cdot}Z + Y'{\cdot}Z + X'{\cdot}Y{\cdot}Z'$$

The new expression corresponds to a different circuit for the same logic function, as shown in figure below.



Similarly, we can "**add out**" (using theorem T8′) the original expression to obtain a **product of sums**:

$$F = ((X + Y')\cdot Z) + (X'\cdot Y\cdot Z')$$
$$= (X + Y' + X')\cdot(X + Y' + Y)\cdot(X + Y' + Z')\cdot(Z + X')\cdot(Z + Y)\cdot(Z + Z')$$
$$= 1 \cdot 1 \cdot (X + Y' + Z')\cdot(X' + Z)\cdot(Y + Z) \cdot 1$$
$$= (X + Y' + Z')\cdot(X' + Z)\cdot(Y + Z)$$

The corresponding logic circuit is shown in figure below. Note that the circuits that synthesized from **sum of product logic** expressions are often called ***AND-OR circuits***, while those which are synthesized from **product of sum** logic expressions are called ***OR-AND circuits*****.**