Discrete Maths

Btec hnd computing (software elective)

BCAS Campus, Kalmunai

By: Eng. A.L. Jubailah Begum

Boolean algebra

**Boolean algebra**

Boolean Algebra is used to analyze and simplify the digital (logic) circuits. It uses only the binary numbers i.e. 0 and 1. It is also called as **Binary Algebra** or **logical Algebra**. Boolean algebra was invented by **George Boole** in 1854.

## Rule in Boolean algebra

Following are the important rules used in boolean algebra.

* Variable used can have only two values. Binary 1 for HIGH and Binary 0 for LOW.
* Complement of a variable is represented by an overbar (¯). Thus complement of variable B is represented as . Thus if B = 0 then = 1 and B = 1 then = 0.
* ORing of the variables is represented by a plus (+) sign between them. For example ORing of A, B, C is represented as A + B + C.
* Logical ANDing of the two or more variable is represented by writing a dot between them such as A.B.C. Sometime the dot may be omitted like ABC.

## Boolean Laws

There are six types of Boolean Laws.

### Commutative law

Any binary operation which satisfies the following expression is referred to as commutative operation

Commutative Law

Commutative law states that changing the sequence of the variables does not have any effect on the output of a logic circuit.

### Associative law

This law states that the order in which the logic operations are performed is irrelevant as their effect is the same.

Associative Law

### Distributive law

Distributive law states the following condition.

Distributive Law

### AND law

These laws use the AND operation. Therefore they are called as **AND** laws. AND Law

### OR law

These laws use the OR operation. Therefore they are called as **OR** laws. OR Law

### INVERSION law

This law uses the NOT operation. The inversion law states that double inversion of a variable result in the original variable itself.

NOT Law

**Logic Gate Truth Tables**

As well as a standard Boolean Expression, the input and output information of any **Logic Gate** or circuit can be plotted into a standard table to give a visual representation of the switching function of the system.

The table used to represent the boolean expression of a logic gate function is commonly called a **Truth Table**. A logic gate truth table shows each possible input combination to the gate or circuit with the resultant output depending upon the combination of these input(s).

For example, consider a single **2-input** logic circuit with input variables labelled as A and B. There are “four” possible input combinations or 22 of “OFF” and “ON” for the two inputs.

However, when dealing with Boolean expressions and especially logic gate truth tables, we do not generally use “ON” or “OFF” but instead give them bit values which represent a logic level “1” or a logic level “0” respectively.

Then the four possible combinations of A and B for a 2-input logic gate is given as:

* Input Combination 1. – “OFF” – “OFF” or ( 0, 0 )
* Input Combination 2. – “OFF” – “ON” or ( 0, 1 )
* Input Combination 3. – “ON” – “OFF” or ( 1, 0 )
* Input Combination 4. – “ON” – “ON” or ( 1, 1 )

Therefore, a 3-input logic circuit would have 8 possible input combinations or 23 and a 4-input logic circuit would have 16 or 24, and so on as the number of inputs increases.

Then a logic circuit with “n” number of inputs would have 2n possible input combinations of both “OFF” and “ON”. In order to keep things simple to understand, we will only deal with simple **2-input** logic gates, but the principals are still the same for gates with more inputs.

The Truth tables for a 2-input AND Gate, a 2-input OR Gate and a NOT Gate are given as:

**2-input AND Gate**

For a 2-input AND gate, the output Q is true if BOTH input A “AND” input B are both true, giving the Boolean Expression of: (Q = A and B).

|  |  |  |  |
| --- | --- | --- | --- |
| Symbol | Truth Table | | |
|  | A | B | Q |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |
| Boolean Expression Q = A.B | Read as A AND B gives Q | | |

Note that the Boolean Expression for a two input AND gate can be written as: A.B or just simply AB without the decimal point.

**2-input OR (Inclusive OR) Gate**

For a 2-input OR gate, the output Q is true if EITHER input A “OR” input B is true, giving the Boolean Expression of: (Q = A or B).

|  |  |  |  |
| --- | --- | --- | --- |
| Symbol | Truth Table | | |
|  | A | B | Q |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |
| Boolean Expression Q = A+B | Read as A OR B gives Q | | |

**NOT Gate**

For a single input NOT gate, the output Q is ONLY true when the input is “NOT” true, the output is the inverse or complement of the input giving the Boolean Expression of: ( Q = NOT A ).

|  |  |  |
| --- | --- | --- |
| Symbol | Truth Table | |
| boolean algebra NOT gate truth table | A | Q |
| 0 | 1 |
| 1 | 0 |
| Boolean Expression Q = NOT A or | Read as inversion of A gives Q | |

The NAND and the NOR Gates are a combination of the AND and OR Gates with that of a NOT Gate or inverter.

**2-input NAND (Not AND) Gate**

For a 2-input NAND gate, the output Q is true if BOTH input A and input B are NOT true, giving the Boolean Expression of: (Q = not (A and B)).

|  |  |  |  |
| --- | --- | --- | --- |
| Symbol | Truth Table | | |
|  | A | B | Q |
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |
| Boolean Expression Q = | Read as A AND B gives NOT-Q | | |

**2-input NOR (Not OR) Gate**

For a 2-input NOR gate, the output Q is true if BOTH input A and input B are NOT true, giving the Boolean Expression of: (Q = not (A or B)).

|  |  |  |  |
| --- | --- | --- | --- |
| Symbol | Truth Table | | |
|  | A | B | Q |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |
| Boolean Expression Q = | Read as A OR B gives NOT-Q | | |

As well as the standard logic gates there are also two special types of logic gate function called an Exclusive-OR Gate and an Exclusive-NOR Gate. The actions of both of these types of gates can be made using the above standard gates however, as they are widely used functions, they are now available in standard IC form and have been included here as reference.

**2-input EX-OR (Exclusive OR) Gate**

For a 2-input Ex-OR gate, the output Q is true if EITHER input A or if input B is true, but NOT both giving the Boolean Expression of: (Q = (A and NOT B) or (NOT A and B)).

|  |  |  |  |
| --- | --- | --- | --- |
| Symbol | Truth Table | | |
| Ex-OR gate truth table | A | B | Q |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |
| Boolean Expression Q = A  B | Q = A. + .B | | |

**2-input EX-NOR (Exclusive NOR) Gate**

For a 2-input Ex-NOR gate, the output Q is true if BOTH input A and input B are the same, either true or false, giving the Boolean Expression of: ( Q = (A and B) or (NOT A and NOT B)).

|  |  |  |  |
| --- | --- | --- | --- |
| Symbol | Truth Table | | |
| Ex-NOR gate truth table | A | B | Q |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |
| Boolean Expression Q = | Q = A.B +. | | |

**Summary of 2-input Logic Gates**

The following Truth Table compares the logical functions of the 2-input logic gates above.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Inputs | | Truth Table Outputs For Each Gate | | | | | |
| A | B | AND | NAND | OR | NOR | EX-OR | EX-NOR |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |

The table gives a list of the common logic functions and their equivalent Boolean notation.

|  |  |
| --- | --- |
| Logic Function | Boolean Notation |
| AND | A.B |
| OR | A+B |
| NOT |  |
| NAND |  |
| NOR |  |
| EX-OR | (A.) + (.) or A  B |
| EX-NOR | (A.B) + (.) or |

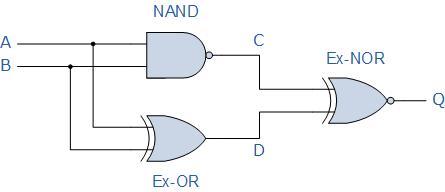
2-input logic gate truth tables are given here as examples of the operation of each logic function, but there are many more logic gates with 3, 4 even 8 individual inputs. The multiple input gates are not different to the simple 2-input gates above, So a 4-input AND gate would still require ALL 4-inputs to be present to produce the required output at Q and its larger truth table would reflect that.

## Boolean Algebra Examples

We have seen throughout this section that digital logic functions can be defined and displayed as either a Boolean Algebra expression or as a logic gate truth table. So here are a few examples of how we can use **Boolean Algebra** to simplify larger digital logic circuits.

## Boolean Algebra Example No1

Construct a Truth Table for the logical functions at points C, D and Q in the following circuit and identify a single logic gate that can be used to replace the whole circuit.



First observations tell us that the circuit consists of a 2-input NAND gate, a 2-input EX-OR gate and finally a 2-input EX-NOR gate at the output. As there are only 2 inputs to the circuit labelled A and B, there can only be 4 possible combinations of the input ( 22 ) and these are: 0-0, 0-1, 1-0 and finally 1-1. Plotting the logical functions from each gate in tabular form will give us the following truth table for the whole of the logic circuit below.

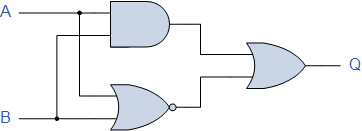
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Inputs | | Output at | | |
| A | B | C | D | Q |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 |

From the truth table above, column C represents the output function generated by the NAND gate, while column D represents the output function from the Ex-OR gate. Both of these two output expressions then become the input condition for the Ex-NOR gate at the output.

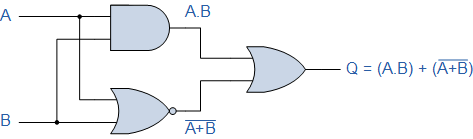
It can be seen from the truth table that an output at Q is present when any of the two inputs A or B are at logic 1. The only truth table that satisfies this condition is that of an OR Gate. Therefore, the whole of the above circuit can be replaced by just one single **2-input** OR Gate.

## Boolean Algebra Example No2

Find the Boolean algebra expression for the following system.



The system consists of an AND Gate, a NOR Gate and finally an OR Gate. The expression for the AND gate is A.B, and the expression for the NOR gate is . Both these expressions are also separate inputs to the OR gate which is defined as A+B. Thus the final output expression is given as:



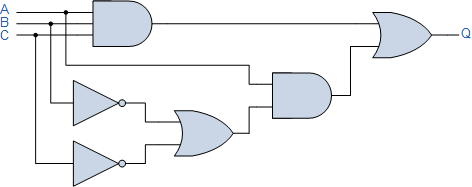
The output of the system is given as Q = (A.B) + (), but the notation () is the same as the De Morgan´s notation ., Then substituting . into the output expression gives us a final output notation of Q = (A.B)+(.), which is the Boolean notation for an Exclusive-NOR Gate as seen in the previous section.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Inputs | | Intermediates | | Output |
| B | A | A.B |  | Q |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 1 |

Then, the whole circuit above can be replaced by just one single Exclusive-NOR Gate and indeed an Exclusive-NOR Gate is made up of these individual gate functions.

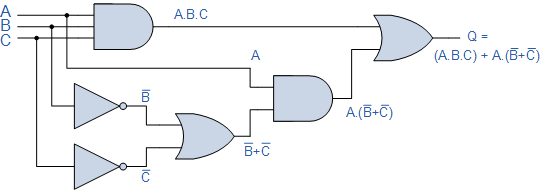
## Boolean Algebra Example No3

Find the Boolean algebra expression for the following system.



This system may look more complicated than the other two to analyse but again, the logic circuit just consists of simple AND, OR and NOT gates connected together.

As with the previous Boolean examples, we can simplify the circuit by writing down the Boolean notation for each logic gate function in turn in order to give us a final expression for the output at Q.



The output from the 3-input AND gate is only at logic “1” when **ALL** the gates inputs are HIGH at logic level “1” (A.B.C). The output from the lower OR gate is only a “1” when one or both inputs B or C are at logic level “0”. The output from the 2-input AND gate is a “1” when input A is a “1” and inputs B or C are at “0”. Then the output at Q is only a “1” when inputs A.B.C equal “1” or A is equal to “1” and both inputs B or C equal “0”, A.(+).

By using “**de Morgan’s theorem**” inputs B and input C cancel out as to produce an output at Q they can be either at logic “1” or at logic “0”. Then this just leaves input A as the only input needed to give an output at Q as shown in the table below.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Inputs | | | Intermediates | | | | | Output |
| C | B | A | A.B.C |  |  | + | A.(+) | Q |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |

Then we can see that the entire logic circuit above can be replaced by just one single input labelled A thereby reducing a circuit of six individual logic gates to just one single piece of wire, (or [Buffer](http://www.electronics-tutorials.ws/logic/logic_9.html)). This type of circuit analysis using [Boolean Algebra](http://amazon.com/s/?field-keywords=Bebop+to+the+Boolean+Boogie%3A+An+Unconventional+Guide+to+Electronics&tag=basicelecttut-20) can be very powerful and quickly identify any unnecessary logic gates within a digital logic design thereby reducing the number of gates required, the power consumption of the circuit and of course the cost.

## The Laws of Boolean

As well as the logic symbols “0” and “1” being used to represent a digital input or output, we can also use them as constants for a permanently “Open” or “Closed” circuit or contact respectively. A set of rules or Laws of Boolean Algebra expressions have been invented to help reduce the number of logic gates needed to perform a particular logic operation resulting in a list of functions or theorems known commonly as the **Laws of Boolean Algebra**.

**Boolean Algebra** is the mathematics we use to analyze digital gates and circuits. We can use these “Laws of Boolean” to both reduce and simplify a complex Boolean expression in an attempt to reduce the number of logic gates required. Boolean Algebra is therefore a system of mathematics based on logic that has its own set of rules or laws which are used to define and reduce Boolean expressions.

The variables used in [Boolean Algebra](http://amazon.com/s/?field-keywords=Bebop+to+the+Boolean+Boogie%3A+An+Unconventional+Guide+to+Electronics&tag=basicelecttut-20) only have one of two possible values, a logic “0” and a logic “1” but an expression can have an infinite number of variables all labelled individually to represent inputs to the expression, For example, variables A, B, C etc, giving us a logical expression of A + B = C, but each variable can ONLY be a 0 or a 1.

Examples of these individual laws of Boolean, rules and theorems for Boolean Algebra are given in the following table.

### Truth Tables for the Laws of Boolean

|  |  |  |  |
| --- | --- | --- | --- |
| Boolean Expression | Description | Equivalent Switching Circuit | Boolean Algebra Law or Rule |
| A + 1 = 1 | A in parallel with closed = "CLOSED" | universal parallel circuit | Annulment |
| A + 0 = A | A in parallel with open = "A" | universal parallel | Identity |
| A . 1 = A | A in series with closed = "A" | universal series circuit | Identity |
| A . 0 = 0 | A in series with open = "OPEN" | universal series | Annulment |
| A + A = A | A in parallel with A = "A" | indempotent parallel circuit | Indempotent |
| A . A = A | A in series with A = "A" | indempotent series circuit | Indempotent |
| NOT A = A | NOT NOT A (double negative) = "A" |  | Double Negation |
| A + A = 1 | A in parallel with not A = "CLOSED" | complement parallel circuit | Complement |
| A . A = 0 | A in series with not A = "OPEN" | complement series circuit | Complement |
| A+B = B+A | A in parallel with B = B in parallel with A | absorption parallel circuit | Commutative |
| A.B = B.A | A in series with B = B in series with A | absorption series circuit | Commutative |
| A+B = A.B | invert and replace OR with AND |  | de Morgan’s Theorem |
| A.B = A+B | invert and replace AND with OR |  | de Morgan’s Theorem |

The basic **Laws of Boolean Algebra** that relate to the **Commutative Law** allowing a change in position for addition and multiplication, the **Associative Law** allowing the removal of brackets for addition and multiplication, as well as the **distributive Law** allowing the factoring of an expression, are the same as in ordinary algebra.

Each of the [Boolean Laws](http://amazon.com/s/?field-keywords=Bebop+to+the+Boolean+Boogie%3A+An+Unconventional+Guide+to+Electronics&tag=basicelecttut-20) above are given with just a single or two variables, but the number of variables defined by a single law is not limited to this as there can be an infinite number of variables as inputs too the expression. These Boolean laws detailed above can be used to prove any given Boolean expression as well as for simplifying complicated digital circuits.

A brief description of the various **Laws of Boolean** are given below with A representing a variable input.

## Description of the Laws of Boolean Algebra

* Annulment Law – A term AND´ed with a “0” equals 0 or OR´ed with a “1” will equal 1.
* + A . 0 = 0,    A variable AND’ed with 0 is always equal to 0.
  + A + 1 = 1,    A variable OR’ed with 1 is always equal to 1.
* Identity Law – A term OR´ed with a “0” or AND´ed with a “1” will always equal that term.
* + A + 0 = A,   A variable OR’ed with 0 is always equal to the variable.
  + A . 1 = A,    A variable AND’ed with 1 is always equal to the variable.
* Indempotent Law – An input AND´ed with itself or OR´ed with itself is equal to that input.
* + A + A = A,    A variable OR’ed with itself is always equal to the variable.
  + A . A = A,    A variable AND’ed with itself is always equal to the variable.
* Complement Law – A term AND´ed with its complement equals “0” and a term OR´ed with its complement equals “1”.
* + A . A = 0,    A variable AND’ed with its complement is always equal to 0.
  + A + A = 1,    A variable OR’ed with its complement is always equal to 1.
* Commutative Law – The order of application of two separate terms is not important.
* + A . B = B . A,    The order in which two variables are AND’ed makes no difference.
  + A + B = B + A,    The order in which two variables are OR’ed makes no difference.
* Double Negation Law – A term that is inverted twice is equal to the original term.
* + A = A,     A double complement of a variable is always equal to the variable.
* de Morgan´s Theorem – There are two “de Morgan´s” rules or theorems,
* (**1**) Two separate terms NOR´ed together is the same as the two terms inverted (Complement) and AND´ed for example, A+B = A. B.
* (**2**) Two separate terms NAND´ed together is the same as the two terms inverted (Complement) and OR´ed for example, A.B = A +B.

Other algebraic laws not detailed above include:

* Distributive Law – This law permits the multiplying or factoring out of an expression.
* Absorptive Law – This law enables a reduction in a complicated expression to a simpler one by absorbing like terms.
* Associative Law – This law allows the removal of brackets from an expression and regrouping of the variables.

## Boolean Algebra Functions

Using the information above, simple 2-input AND, OR and NOT Gates can be represented by 16 possible functions as shown in the following table.

|  |  |  |
| --- | --- | --- |
| Function | Description | Expression |
| 1. | NULL | 0 |
| 2. | IDENTITY | 1 |
| 3. | Input A | A |
| 4. | Input B | B |
| 5. | NOT A | A |
| 6. | NOT B | B |
| 7. | A AND B (AND) | A . B |
| 8. | A AND NOT B | A . B |
| 9. | NOT A AND B | A . B |
| 10. | NOT A AND NOT B (NAND) | A . B |
| 11. | A OR B (OR) | A + B |
| 12. | A OR NOT B | A + B |
| 13. | NOT A OR B | A + B |
| 14. | NOT OR (NOR) | A + B |
| 15. | Exclusive-OR | A.B + A.B |
| 16. | Exclusive-NOR | A.B + A.B |

## Laws of Boolean Algebra Example No1

Using the above laws, simplify the following expression:  (A + B)(A + C)

|  |  |  |
| --- | --- | --- |
| Q = | (A + B)(A + C) |  |
|  | AA + AC + AB + BC | - Distributive law |
|  | A + AC + AB + BC | - Identity AND law (A.A = A) |
|  | A(1 + C) + AB + BC | – Distributive law |
|  | A.1 + AB + BC | - Identity OR law (1 + C = 1) |
|  | A(1 + B) + BC | - Distributive law |
|  | A.1 + BC | - Identity OR law (1 + B = 1) |
| Q = | A + BC | - Identity AND law (A.1 = A) |

Then the expression:  (A + B)(A + C) can be simplified to A + BC

## Introduction To Boolean

In 1854, **George Boole** performed an investigation into the “laws of thought” which were based on a simplified version of the “group” or “set” theory, and from this **Boolean** or “Switching” algebra was developed. **Boolean Algebra** deals mainly with the theory that both logic and set operations are either “TRUE” or “FALSE” but not both at the same time.

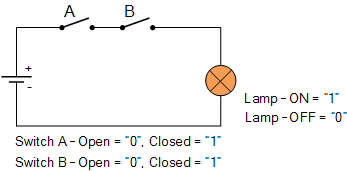
For example, A + A = A and not 2A as it would be in normal algebra. [Boolean Algebra](http://amazon.com/s/?field-keywords=Bebop+to+the+Boolean+Boogie%3A+An+Unconventional+Guide+to+Electronics&tag=basicelecttut-20) is a simple and effective way of representing the switching action of standard Logic Gates and the basic logic statements which concern us here are given by the logic gate operations of the AND, the OR and the NOT gate functions.

## The logic AND Function

The **Logic AND Function** function states that two or more events must occur together and at the same time for an output action to occur. The order in which these actions occur is unimportant as it does not affect the final result. For example, A & B = B & A. In Boolean algebra the Logic AND Function follows the **Commutative Law** which allows a change in position of either variable.

The AND function is represented in electronics by the dot or full stop symbol ( . ) Thus a 2-input (A B) AND Gate has an output term represented by the Boolean expression A**.**B or just AB.

### Switch Representation of the AND Function



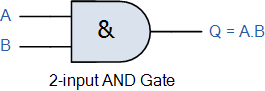
Here the two switches, A and B are connected together to form a series circuit. Therefore, in the circuit above, both switch A **AND** switch B must be closed (Logic “1”) in order to put the lamp on. In other words, both switches must be closed, or at logic “1” for the lamp to be “ON”.

Then this type of logic gate ( an AND Gate ) only produces an output when “ALL” of its inputs are present. In **Boolean Algebra** terms the output will be TRUE only when all of its inputs are TRUE. In electrical terms, the logic AND function is equal to a series circuit as shown above.

As there are only two Switches, each with two possible states “open” or “closed”. Defining a Logic “0” as being when the switch is open and a Logic “1” when the switch is closed, there are then four different ways or combinations of arranging the two switches together as shown.

### AND Function Truth Table

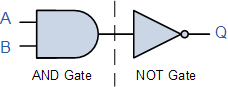
|  |  |  |  |
| --- | --- | --- | --- |
| Switch A | Switch B | Output | Description |
| 0 | 0 | 0 | A and B are both open, lamp OFF |
| 0 | 1 | 0 | A is open and B is closed, lamp OFF |
| 1 | 0 | 0 | A is closed and B is open, lamp OFF |
| 1 | 1 | 1 | A is closed and B is closed, lamp ON |
| Boolean Expression (A AND B) | | | A . B |

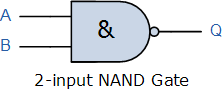


[Logic AND Gates](http://amazon.com/s/?field-keywords=Digital+Logic+Design&tag=basicelecttut-20) are available as standard i.c. packages such as the common TTL 74LS08 Quadruple 2-input Positive AND Gates, (or the 4081 CMOS equivalent) the TTL 74LS11 Triple 3-input Positive AND Gates or the 74LS21 Dual 4-input Positive AND Gates. AND Gates can also be “cascaded” together to produce circuits with more than just 4 inputs.

## The Logic NAND Function

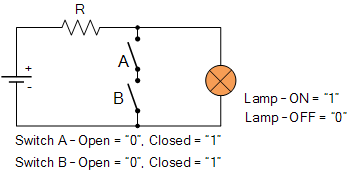
The NAND or “Not AND” function is a combination of the two separate logical functions, the AND function and the NOT function connected together in series. The logic NAND function can be expressed by the Boolean expression of, A.B





The **Logic NAND Function** only produces an output when “ANY” of its inputs are not present and in Boolean Algebra terms the output will be TRUE only when any of its inputs are FALSE.

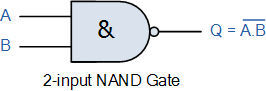
### Switch Representation of the NAND Function



The truth table for the NAND function is the opposite of that for the previous AND function because the NAND gate performs the reverse operation of the AND gate. In other words, the NAND gate is the complement of the basic AND gate.

### NAND Function Truth Table

|  |  |  |  |
| --- | --- | --- | --- |
| Switch A | Switch B | Output | Description |
| 0 | 0 | 1 | A and B are both open, lamp ON |
| 0 | 1 | 1 | A is open and B is closed, lamp ON |
| 1 | 0 | 1 | A is closed and B is open, lamp ON |
| 1 | 1 | 0 | A is closed and B is closed, lamp OFF |
| Boolean Expression (A AND B) | | | A . B |

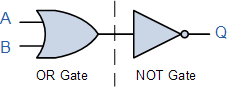


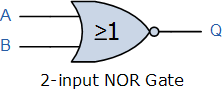
The **NAND Function** is sometimes known as the **Sheffer Stroke Function** and is denoted by a vertical bar or upwards arrow operator, for example, A NAND B = **A|B** or **A↑B**.

[Logic NAND Gates](http://amazon.com/s/?field-keywords=Digital+Logic+Design&tag=basicelecttut-20) are used as the basic “building blocks” to construct other logic gate functions and are available in standard i.c. packages such as the very common TTL 74LS00 Quadruple 2-input NAND Gates, the TTL 74LS10 Triple 3-input NAND Gates or the 74LS20 Dual 4-input NAND Gates. There is even a single chip 74LS30 8-input NAND Gate.

## The Logic NOR Function

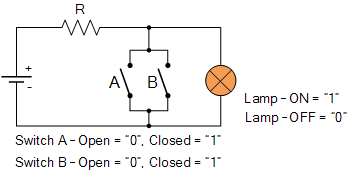
Like the previous NAND Gate, the NOR or “Not OR” Gate is also a combination of two separate functions connected together to form a single logic gate function. The OR function and the NOT function are connected together in series with its operation given by the Boolean expression as, A + B





The **Logic NOR Function** only produces and output when “ALL” of its inputs are not present and in Boolean Algebra terms the output will be TRUE only when all of its inputs are FALSE.

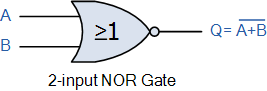
### Switch Representation of the NOR Function



The truth table for the NOR function is the opposite of that for the previous OR function because the NOR gate performs the reverse operation of the OR gate. Then we can see that the NOR gate is the complement of the OR gate.

### NOR Function Truth Table

|  |  |  |  |
| --- | --- | --- | --- |
| Switch A | Switch B | Output | Description |
| 0 | 0 | 1 | Both A and B are open, lamp ON |
| 0 | 1 | 0 | A is open and B is closed, lamp OFF |
| 1 | 0 | 0 | A is closed and B is open, lamp OFF |
| 1 | 1 | 0 | A is closed and B is closed, lamp OFF |
| Boolean Expression (A OR B) | | | A + B |



The **NOR Function** is sometimes known as the **Pierce Function** and is denoted by a downwards arrow operator as shown, A NOR B = **A↓B**.

[Logic NOR Gates](http://amazon.com/s/?field-keywords=Digital+Logic+Design&tag=basicelecttut-20) are available as standard i.c. packages such as the TTL 74LS02 Quadruple 2-input NOR Gate, the TTL 74LS27 Triple 3-input NOR Gate or the 74LS260 Dual 5-input NOR Gate.

## The Logic NOT Function

The **Logic NOT Function** is simply a single input inverter that changes the input of a logic level “1” to an output of logic level “0” and vice versa. The logic NOT function is so called because its output state is **NOT** the same as its input state.

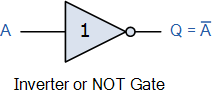
The “logical NOT function” is generally denoted by a bar or overline ( ¯ ) over its input symbol which denotes the inversion operation, (hence its name as an inverter). As NOT gates perform the logic **INVERT** or **COMPLEMENTATION** function they are more commonly known as Inverters because they invert the signal. In logic circuits this negation can be represented by a normally closed switch.

### Switch Representation of the NOT Function

If A means that the switch is closed, then NOT A or simply A says that the switch is **NOT** closed or in other words, it is open. The logic NOT function has a single input and a single output as shown.

### NOT Function Truth Table

|  |  |
| --- | --- |
| Switch | Output |
| 1 | 0 |
| 0 | 1 |
| Boolean Expression | not-A or A |



The inversion indicator for a logic NOT function is a “bubble”, ( O ) symbol on the output (or input) of the logic elements symbol. In Boolean algebra the Logic NOT Function follows the **Complementation Law** producing inversion.

complementation law

[Logic NOT Gates](http://amazon.com/s/?field-keywords=Digital+Logic+Design&tag=basicelecttut-20) or “Inverters” as they are more commonly called, can be connected with standard AND and OR gates to produce NAND and NOR gates respectively. Logic NOT gates can also be used to produce “Complementary” signals in more complex decoder/logic circuits for example, the complement of logic A is A and two Inverters connected together in series will give a double inversion which produces at its output the original value of A.

When designing logic circuits and you may only need one or two logic NOT gates within your design, but do not have the space or the money for a dedicated Inverter chip such as the 74LS04. Then you can easily make a logic NOT function easily by using any spare NAND or NOR gates by simply connecting their inputs together as shown below.

### NOT Function Equivalents

logic not gate equivalent

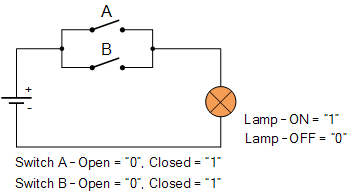
## The Logic OR Function

The **Logic OR Function** function states that an output action will occur or become TRUE if either one “OR” more events are TRUE, but the order at which they occur is unimportant as it does not affect the final result. For example, A + B = B + A. In Boolean algebra the Logic OR Function follows the **Commutative Law** the same as for the logic AND function, allowing a change in position of either variable.

The OR function is sometimes called by its full name of “Inclusive OR” in contrast to the [Exclusive-OR](http://www.electronics-tutorials.ws/boolean/bool_6.html) function we will look at later in tutorial six.

The logic or Boolean expression given for a logic OR gate is that for Logical Addition which is denoted by a plus sign, (+). Thus a 2-input (A B) **Logic OR Gate** has an output term represented by the Boolean expression of:  A+B = Q.

### Switch Representation of the OR Function



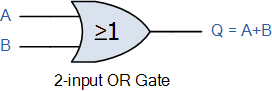
Here the two switches A and B are connected in parallel and either Switch A **OR** Switch B can be closed in order to put the lamp on. In other words, either switch can be closed, or at logic “1” for the lamp to be “ON”.

Then this type of logic gate only produces and output when “ANY” of its inputs are present and in Boolean Algebra terms the output will be TRUE when any of its inputs are TRUE. In electrical terms, the logic OR function is equal to a parallel circuit.

Again as with the AND function there are two switches, each with two possible positions open or closed so therefore there will be 4 different ways of arranging the switches.

### OR Function Truth Table

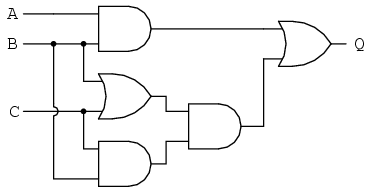
|  |  |  |  |
| --- | --- | --- | --- |
| Switch A | Switch B | Output | Description |
| 0 | 0 | 0 | A and B are both open, lamp OFF |
| 0 | 1 | 1 | A is open and B is closed, lamp ON |
| 1 | 0 | 1 | A is closed and B is open, lamp ON |
| 1 | 1 | 1 | A is closed and B is closed, lamp ON |
| Boolean Expression (A OR B) | | | A + B |



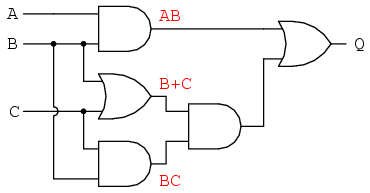
[Logic OR Gates](http://amazon.com/s/?field-keywords=Digital+Logic+Design&tag=basicelecttut-20) are available as standard i.c. packages such as the common TTL 74LS32 Quadruple 2-input Positive OR Gates. As with the previous AND Gate, OR can also be “cascaded” together to produce circuits with more inputs such as in security alarm systems (Zone A or Zone B or Zone C,etc).

# Circuit simplification examples

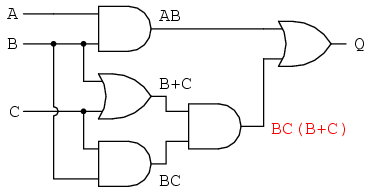
Let's begin with a semiconductor gate circuit in need of simplification. The "A," "B," and "C" input signals are assumed to be provided from switches, sensors, or perhaps other gate circuits. Where these signals originate is of no concern in the task of gate reduction.



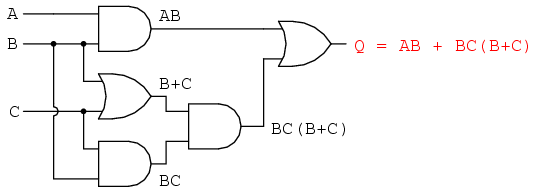
Our first step in simplification must be to write a Boolean expression for this circuit. This task is easily performed step by step if we start by writing sub-expressions at the output of each gate, corresponding to the respective input signals for each gate. Remember that OR gates are equivalent to Boolean addition, while AND gates are equivalent to Boolean multiplication. For example, I'll write sub-expressions at the outputs of the first three gates:



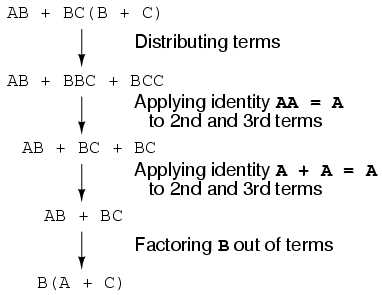
. . . then another sub-expression for the next gate:



Finally, the output ("Q") is seen to be equal to the expression AB + BC(B + C):

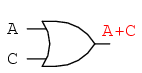


Now that we have a Boolean expression to work with, we need to apply the rules of Boolean algebra to reduce the expression to its simplest form (simplest defined as requiring the fewest gates to implement):

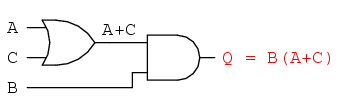


The final expression, B(A + C), is much simpler than the original, yet performs the same function. If you would like to verify this, you may generate a truth table for both expressions and determine Q's status (the circuits' output) for all eight logic-state combinations of A, B, and C, for both circuits. The two truth tables should be identical.

Now, we must generate a schematic diagram from this Boolean expression. To do this, evaluate the expression, following proper mathematical order of operations (multiplication before addition, operations inside parentheses before anything else), and draw gates for each step. Remember again that OR gates are equivalent to Boolean addition, while AND gates are equivalent to Boolean multiplication. In this case, we would begin with the sub-expression "A + C", which is an OR gate:

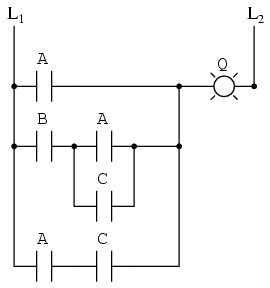


The next step in evaluating the expression "B(A + C)" is to multiply (AND gate) the signal B by the output of the previous gate (A + C):

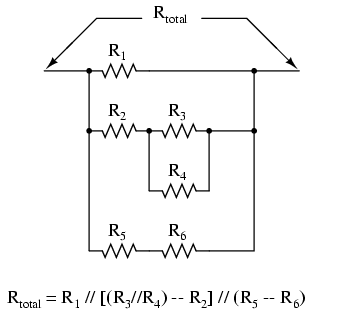


Obviously, this circuit is much simpler than the original, having only two logic gates instead of five. Such component reduction results in higher operating speed (less delay time from input signal transition to output signal transition), less power consumption, less cost, and greater reliability.

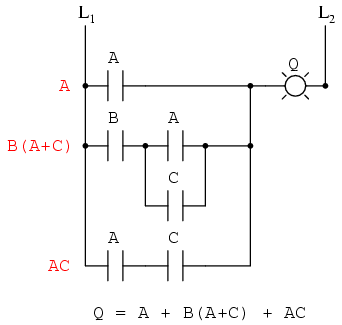
Electromechanical relay circuits, typically being slower, consuming more electrical power to operate, costing more, and having a shorter average life than their semiconductor counterparts, benefit dramatically from Boolean simplification. Let's consider an example circuit:



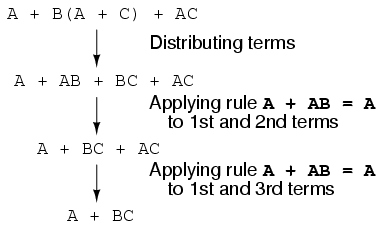
As before, our first step in reducing this circuit to its simplest form must be to develop a Boolean expression from the schematic. The easiest way I've found to do this is to follow the same steps I'd normally follow to reduce a series-parallel resistor network to a single, total resistance. For example, examine the following resistor network with its resistors arranged in the same connection pattern as the relay contacts in the former circuit, and corresponding total resistance formula:



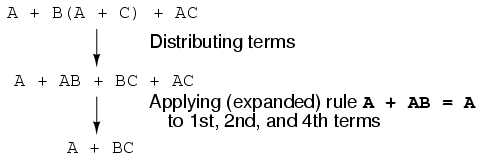
Remember that parallel contacts are equivalent to Boolean addition, while series contacts are equivalent to Boolean multiplication. Write a Boolean expression for this relay contact circuit, following the same order of precedence that you would follow in reducing a series-parallel resistor network to a total resistance. It may be helpful to write a Boolean sub-expression to the left of each ladder "rung," to help organize your expression-writing:



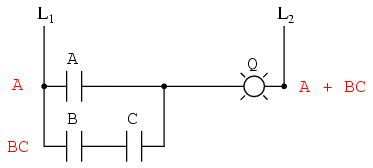
Now that we have a Boolean expression to work with, we need to apply the rules of Boolean algebra to reduce the expression to its simplest form (simplest defined as requiring the fewest relay contacts to implement):



The more mathematically inclined should be able to see that the two steps employing the rule "A + AB = A" may be combined into a single step, the rule being expandable to: "A + AB + AC + AD + . . . = A"



As you can see, the reduced circuit is much simpler than the original, yet performs the same logical function:

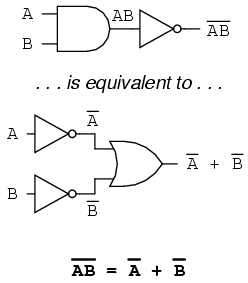


* **REVIEW:**
* To convert a gate circuit to a Boolean expression, label each gate output with a Boolean sub-expression corresponding to the gates' input signals, until a final expression is reached at the last gate.
* To convert a Boolean expression to a gate circuit, evaluate the expression using standard order of operations: multiplication before addition, and operations within parentheses before anything else.
* To convert a ladder logic circuit to a Boolean expression, label each rung with a Boolean sub-expression corresponding to the contacts' input signals, until a final expression is reached at the last coil or light. To determine proper order of evaluation, treat the contacts as though they were resistors, and as if you were determining total resistance of the series-parallel network formed by them. In other words, look for contacts that are either *directly* in series or *directly* in parallel with each other first, then "collapse" them into equivalent Boolean sub-expressions before proceeding to other contacts.
* To convert a Boolean expression to a ladder logic circuit, evaluate the expression using standard order of operations: multiplication before addition, and operations within parentheses before anything else.

# DeMorgan's Theorems

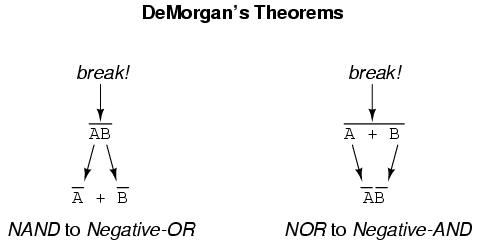
A mathematician named DeMorgan developed a pair of important rules regarding group complementation in Boolean algebra. By *group* complementation, I'm referring to the complement of a group of terms, represented by a long bar over more than one variable.

You should recall from the chapter on logic gates that inverting all inputs to a gate reverses that gate's essential function from AND to OR, or vice versa, and also inverts the output. So, an OR gate with all inputs inverted (a Negative-OR gate) behaves the same as a NAND gate, and an AND gate with all inputs inverted (a Negative-AND gate) behaves the same as a NOR gate. DeMorgan's theorems state the same equivalence in "backward" form: that inverting the output of any gate results in the same function as the opposite type of gate (AND vs. OR) with inverted inputs:

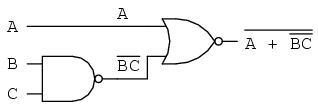


A long bar extending over the term AB acts as a grouping symbol, and as such is entirely different from the product of A and B independently inverted. In other words, (AB)' is not equal to A'B'. Because the "prime" symbol (') cannot be stretched over two variables like a bar can, we are forced to use parentheses to make it apply to the whole term AB in the previous sentence. A bar, however, acts as its own grouping symbol when stretched over more than one variable. This has profound impact on how Boolean expressions are evaluated and reduced, as we shall see.

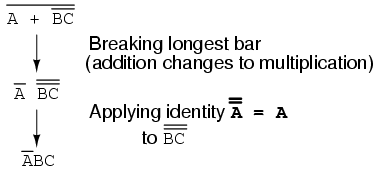
DeMorgan's theorem may be thought of in terms of *breaking* a long bar symbol. When a long bar is broken, the operation directly underneath the break changes from addition to multiplication, or vice versa, and the broken bar pieces remain over the individual variables. To illustrate:



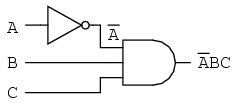
When multiple "layers" of bars exist in an expression, you may only break *one bar at a time*, and it is generally easier to begin simplification by breaking the longest (uppermost) bar first. To illustrate, let's take the expression (A + (BC)')' and reduce it using DeMorgan's Theorems:



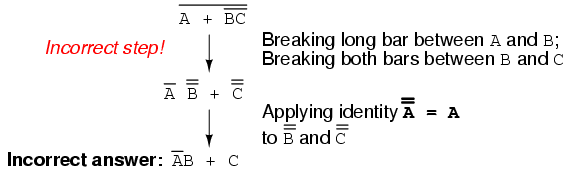
Following the advice of breaking the longest (uppermost) bar first, I'll begin by breaking the bar covering the entire expression as a first step:



As a result, the original circuit is reduced to a three-input AND gate with the A input inverted:

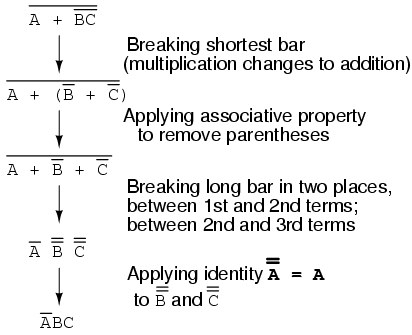


You should *never* break more than one bar in a single step, as illustrated here:



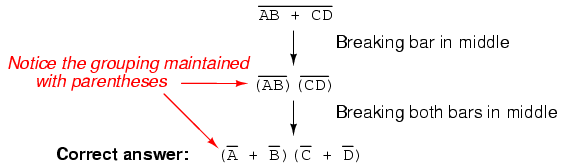
As tempting as it may be to conserve steps and break more than one bar at a time, it often leads to an incorrect result, so don't do it!

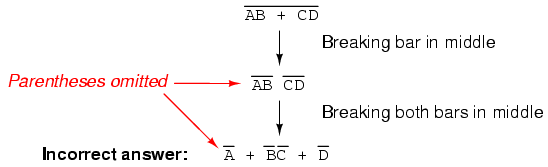
It is possible to properly reduce this expression by breaking the short bar first, rather than the long bar first:



The end result is the same, but more steps are required compared to using the first method, where the longest bar was broken first. Note how in the third step we broke the long bar in two places. This is a legitimate mathematical operation, and not the same as breaking two bars in one step! The prohibition against breaking more than one bar in one step is *not* a prohibition against breaking a bar in more than one place. Breaking in more than one *place* in a single step is okay; breaking more than one *bar* in a single step is not.

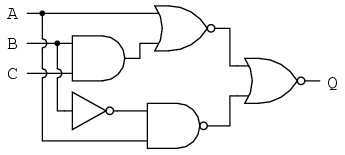
You might be wondering why parentheses were placed around the sub-expression B' + C', considering the fact that I just removed them in the next step. I did this to emphasize an important but easily neglected aspect of DeMorgan's theorem. Since a long bar functions as a grouping symbol, the variables formerly grouped by a broken bar must remain grouped lest proper precedence (order of operation) be lost. In this example, it really wouldn't matter if I forgot to put parentheses in after breaking the short bar, but in other cases it might. Consider this example, starting with a different expression:



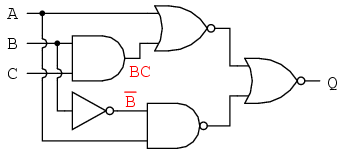


As you can see, maintaining the grouping implied by the complementation bars for this expression is crucial to obtaining the correct answer.

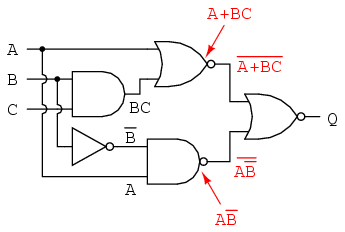
Let's apply the principles of DeMorgan's theorems to the simplification of a gate circuit:



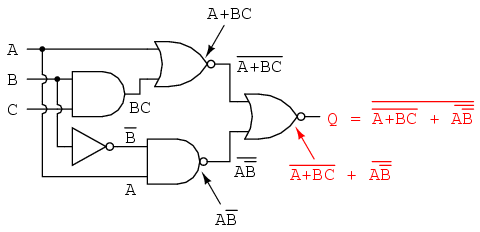
As always, our first step in simplifying this circuit must be to generate an equivalent Boolean expression. We can do this by placing a sub-expression label at the output of each gate, as the inputs become known. Here's the first step in this process:



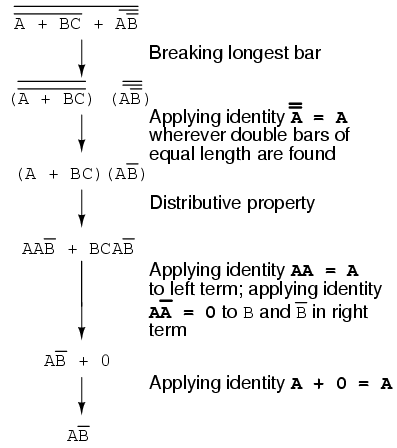
Next, we can label the outputs of the first NOR gate and the NAND gate. When dealing with inverted-output gates, I find it easier to write an expression for the gate's output *without* the final inversion, with an arrow pointing to just before the inversion bubble. Then, at the wire leading out of the gate (after the bubble), I write the full, complemented expression. This helps ensure I don't forget a complementing bar in the sub-expression, by forcing myself to split the expression-writing task into two steps:



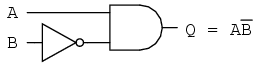
Finally, we write an expression (or pair of expressions) for the last NOR gate:



Now, we reduce this expression using the identities, properties, rules, and theorems (DeMorgan's) of Boolean algebra:



The equivalent gate circuit for this much-simplified expression is as follows:

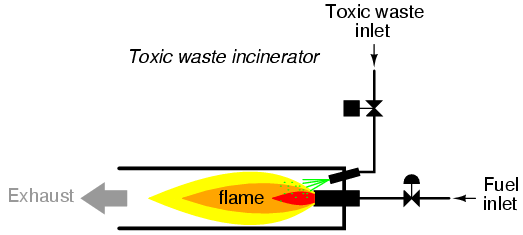


* **REVIEW**
* DeMorgan's Theorems describe the equivalence between gates with inverted inputs and gates with inverted outputs. Simply put, a NAND gate is equivalent to a Negative-OR gate, and a NOR gate is equivalent to a Negative-AND gate.
* When "breaking" a complementation bar in a Boolean expression, the operation directly underneath the break (addition or multiplication) reverses, and the broken bar pieces remain over the respective terms.
* It is often easier to approach a problem by breaking the longest (uppermost) bar before breaking any bars under it. You must *never* attempt to break two bars in one step!
* Complementation bars function as grouping symbols. Therefore, when a bar is broken, the terms underneath it must remain grouped. Parentheses may be placed around these grouped terms as a help to avoid changing precedence.

# Converting truth tables into Boolean expressions

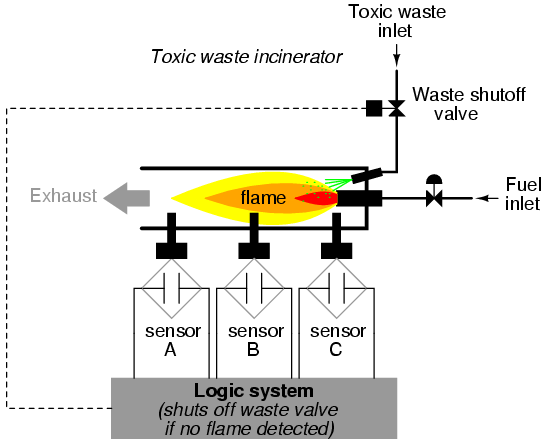
In designing digital circuits, the designer often begins with a truth table describing what the circuit should do. The design task is largely to determine what type of circuit will perform the function described in the truth table. While some people seem to have a natural ability to look at a truth table and immediately envision the necessary logic gate or relay logic circuitry for the task, there are procedural techniques available for the rest of us. Here, Boolean algebra proves its utility in a most dramatic way.

To illustrate this procedural method, we should begin with a realistic design problem. Suppose we were given the task of designing a flame detection circuit for a toxic waste incinerator. The intense heat of the fire is intended to neutralize the toxicity of the waste introduced into the incinerator. Such combustion-based techniques are commonly used to neutralize medical waste, which may be infected with deadly viruses or bacteria:



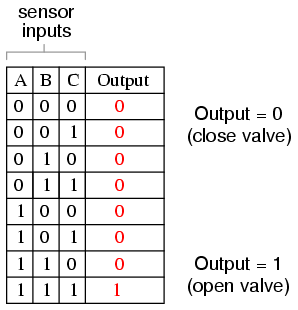
So long as a flame is maintained in the incinerator, it is safe to inject waste into it to be neutralized. If the flame were to be extinguished, however, it would be unsafe to continue to inject waste into the combustion chamber, as it would exit the exhaust un-neutralized, and pose a health threat to anyone in close proximity to the exhaust. What we need in this system is a sure way of detecting the presence of a flame, and permitting waste to be injected only if a flame is "proven" by the flame detection system.

Several different flame-detection technologies exist: optical (detection of light), thermal (detection of high temperature), and electrical conduction (detection of ionized particles in the flame path), each one with its unique advantages and disadvantages. Suppose that due to the high degree of hazard involved with potentially passing un-neutralized waste out the exhaust of this incinerator, it is decided that the flame detection system be made redundant (multiple sensors), so that failure of a single sensor does not lead to an emission of toxins out the exhaust. Each sensor comes equipped with a normally-open contact (open if no flame, closed if flame detected) which we will use to activate the inputs of a logic system:

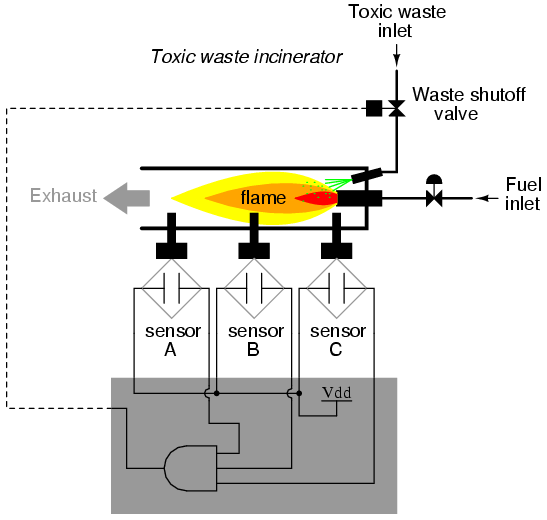


Our task, now, is to design the circuitry of the logic system to open the waste valve if and only if there is good flame proven by the sensors. First, though, we must decide what the logical behavior of this control system should be. Do we want the valve to be opened if only one out of the three sensors detects flame? Probably not, because this would defeat the purpose of having multiple sensors. If any one of the sensors were to fail in such a way as to falsely indicate the presence of flame when there was none, a logic system based on the principle of "any one out of three sensors showing flame" would give the same output that a single-sensor system would with the same failure. A far better solution would be to design the system so that the valve is commanded to open if and only if *all three sensors* detect a good flame. This way, any single, failed sensor falsely showing flame could not keep the valve in the open position; rather, it would require all three sensors to be failed in the same manner -- a highly improbable scenario -- for this dangerous condition to occur.

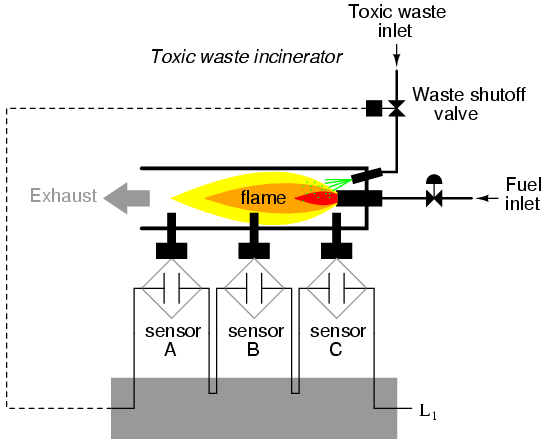
Thus, our truth table would look like this:



It does not require much insight to realize that this functionality could be generated with a three-input AND gate: the output of the circuit will be "high" if and only if input A *AND* input B *AND* input C are all "high:"

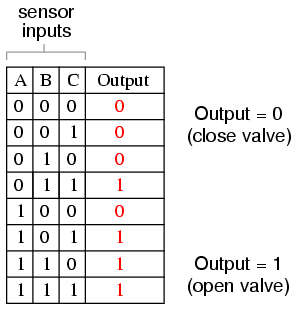


If using relay circuitry, we could create this AND function by wiring three relay contacts in series, or simply by wiring the three sensor contacts in series, so that the only way electrical power could be sent to open the waste valve is if all three sensors indicate flame:



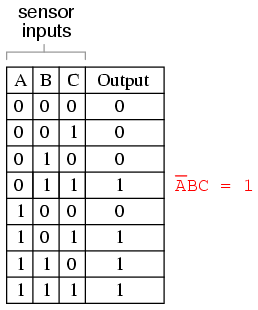
While this design strategy maximizes safety, it makes the system very susceptible to sensor failures of the opposite kind. Suppose that one of the three sensors were to fail in such a way that it indicated no flame when there really was a good flame in the incinerator's combustion chamber. That single failure would shut off the waste valve unnecessarily, resulting in lost production time and wasted fuel (feeding a fire that wasn't being used to incinerate waste).

It would be nice to have a logic system that allowed for this kind of failure without shutting the system down unnecessarily, yet still provide sensor redundancy so as to maintain safety in the event that any single sensor failed "high" (showing flame at all times, whether or not there was one to detect). A strategy that would meet both needs would be a "two out of three" sensor logic, whereby the waste valve is opened if at least two out of the three sensors show good flame. The truth table for such a system would look like this:

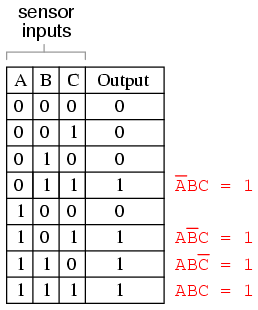


Here, it is not necessarily obvious what kind of logic circuit would satisfy the truth table. However, a simple method for designing such a circuit is found in a standard form of Boolean expression called the *Sum-Of-Products*, or *SOP*, form. As you might suspect, a Sum-Of-Products Boolean expression is literally a set of Boolean terms added (*summed*) together, each term being a multiplicative (*product*) combination of Boolean variables. An example of an SOP expression would be something like this: ABC + BC + DF, the sum of products "ABC," "BC," and "DF."

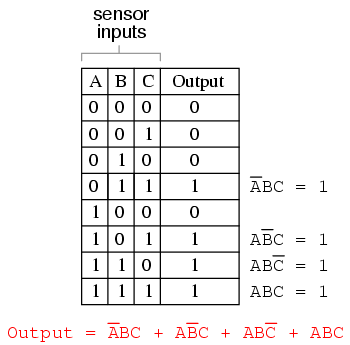
Sum-Of-Products expressions are easy to generate from truth tables. All we have to do is examine the truth table for any rows where the output is "high" (1), and write a Boolean product term that would equal a value of 1 given those input conditions. For instance, in the fourth row down in the truth table for our two-out-of-three logic system, where A=0, B=1, and C=1, the product term would be A'BC, since that term would have a value of 1 if and only if A=0, B=1, and C=1:



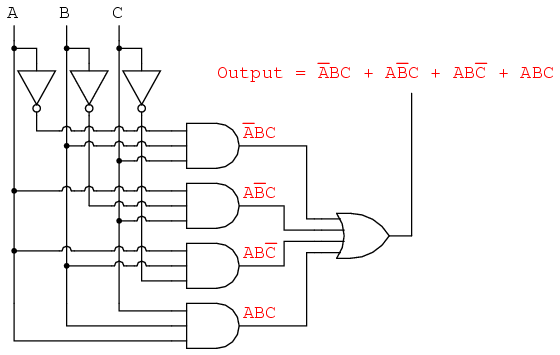
Three other rows of the truth table have an output value of 1, so those rows also need Boolean product expressions to represent them:

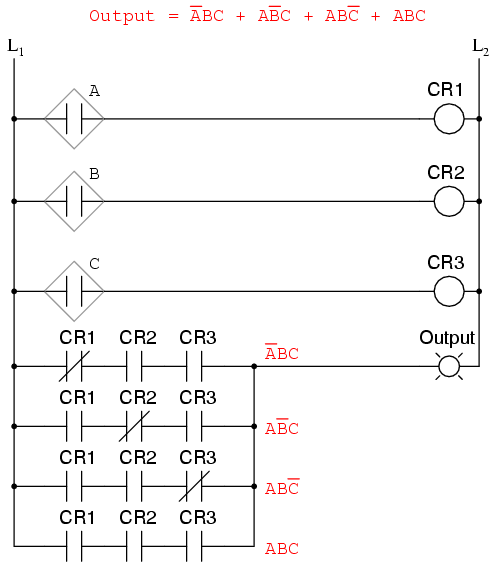


Finally, we join these four Boolean product expressions together by addition, to create a single Boolean expression describing the truth table as a whole:

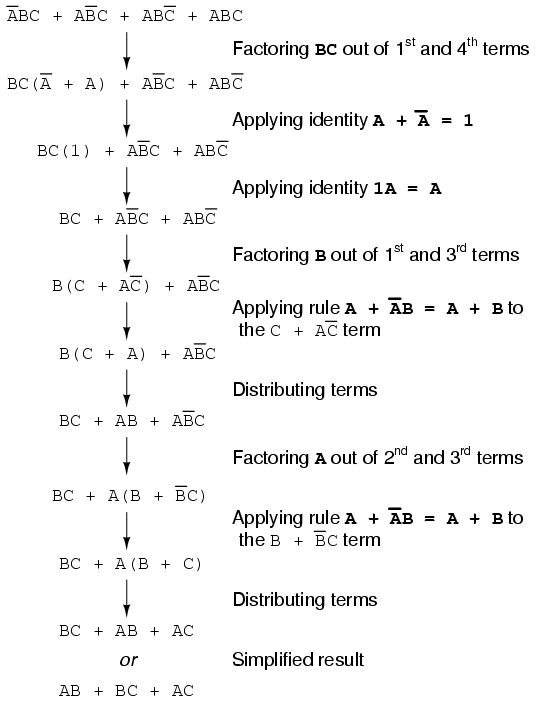


Now that we have a Boolean Sum-Of-Products expression for the truth table's function, we can easily design a logic gate or relay logic circuit based on that expression:

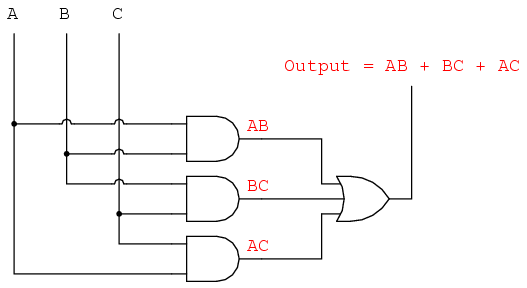


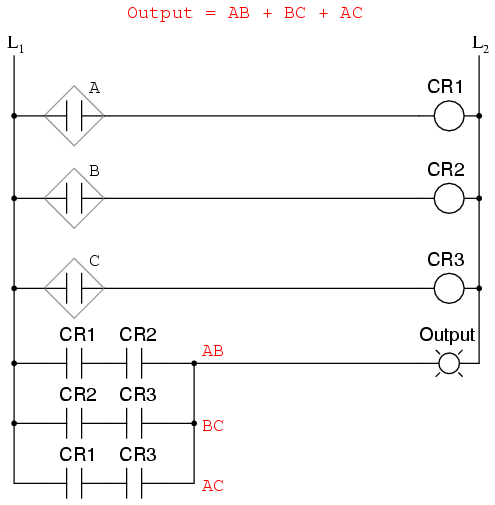


Unfortunately, both of these circuits are quite complex, and could benefit from simplification. Using Boolean algebra techniques, the expression may be significantly simplified:



As a result of the simplification, we can now build much simpler logic circuits performing the same function, in either gate or relay form:

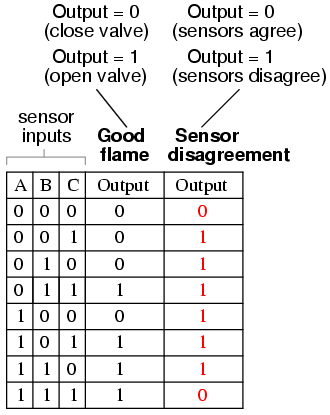




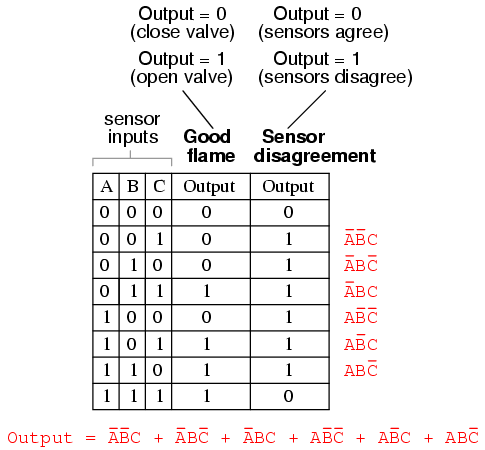
Either one of these circuits will adequately perform the task of operating the incinerator waste valve based on a flame verification from two out of the three flame sensors. At minimum, this is what we need to have a safe incinerator system. We can, however, extend the functionality of the system by adding to it logic circuitry designed to detect if any one of the sensors does not agree with the other two.

If all three sensors are operating properly, they should detect flame with equal accuracy. Thus, they should either all register "low" (000: no flame) or all register "high" (111: good flame). Any other output combination (001, 010, 011, 100, 101, or 110) constitutes a disagreement between sensors, and may therefore serve as an indicator of a potential sensor failure. If we added circuitry to detect any one of the six "sensor disagreement" conditions, we could use the output of that circuitry to activate an alarm. Whoever is monitoring the incinerator would then exercise judgment in either continuing to operate with a possible failed sensor (inputs: 011, 101, or 110), or shut the incinerator down to be absolutely safe. Also, if the incinerator is shut down (no flame), and one or more of the sensors still indicates flame (001, 010, 011, 100, 101, or 110) while the other(s) indicate(s) no flame, it will be known that a definite sensor problem exists.

The first step in designing this "sensor disagreement" detection circuit is to write a truth table describing its behavior. Since we already have a truth table describing the output of the "good flame" logic circuit, we can simply add another output column to the table to represent the second circuit, and make a table representing the entire logic system:

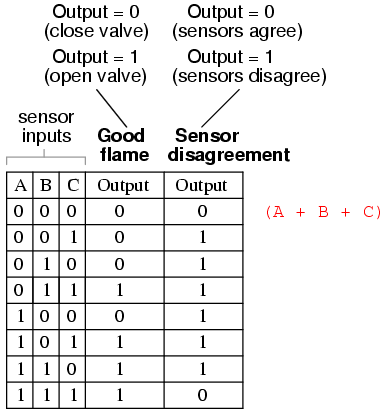


While it is possible to generate a Sum-Of-Products expression for this new truth table column, it would require six terms, of three variables each! Such a Boolean expression would require many steps to simplify, with a large potential for making algebraic errors:

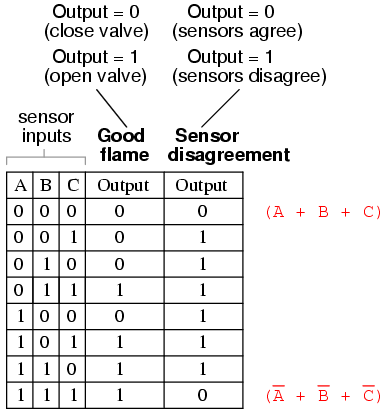


An alternative to generating a Sum-Of-Products expression to account for all the "high" (1) output conditions in the truth table is to generate a *Product-Of-Sums*, or *POS*, expression, to account for all the "low" (0) output conditions instead. Being that there are much fewer instances of a "low" output in the last truth table column, the resulting Product-Of-Sums expression should contain fewer terms. As its name suggests, a Product-Of-Sums expression is a set of added terms (*sums*), which are multiplied (*product*) together. An example of a POS expression would be (A + B)(C + D), the product of the sums "A + B" and "C + D".

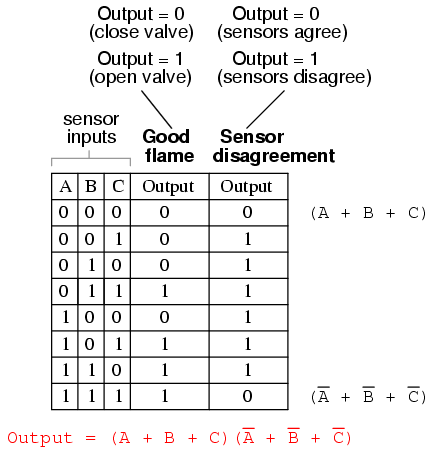
To begin, we identify which rows in the last truth table column have "low" (0) outputs, and write a Boolean sum term that would equal 0 for that row's input conditions. For instance, in the first row of the truth table, where A=0, B=0, and C=0, the sum term would be (A + B + C), since that term would have a value of 0 if and only if A=0, B=0, and C=0:



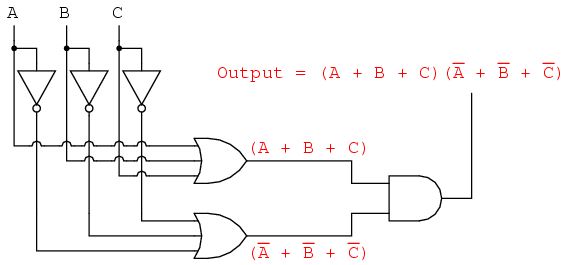
Only one other row in the last truth table column has a "low" (0) output, so all we need is one more sum term to complete our Product-Of-Sums expression. This last sum term represents a 0 output for an input condition of A=1, B=1 and C=1. Therefore, the term must be written as (A' + B'+ C'), because only the sum of the *complemented* input variables would equal 0 for that condition only:



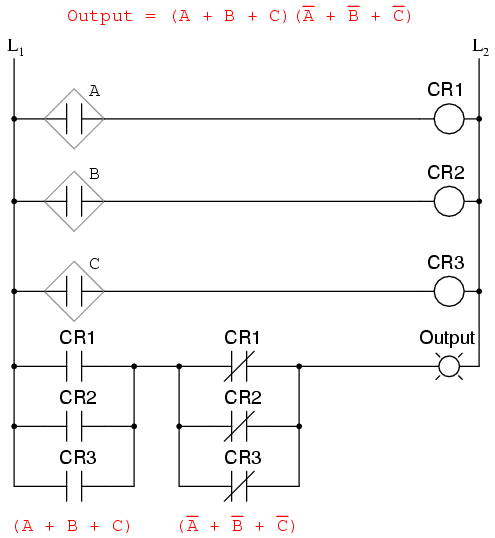
The completed Product-Of-Sums expression, of course, is the multiplicative combination of these two sum terms:



Whereas a Sum-Of-Products expression could be implemented in the form of a set of AND gates with their outputs connecting to a single OR gate, a Product-Of-Sums expression can be implemented as a set of OR gates feeding into a single AND gate:

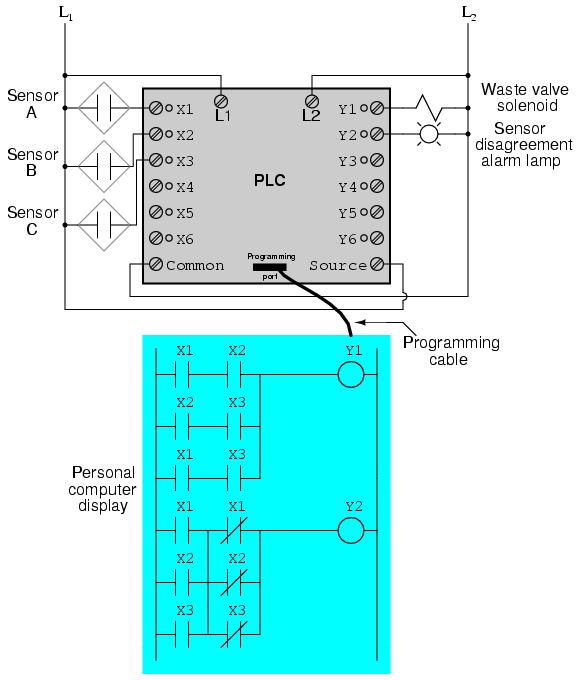


Correspondingly, whereas a Sum-Of-Products expression could be implemented as a parallel collection of series-connected relay contacts, a Product-Of-Sums expression can be implemented as a series collection of parallel-connected relay contacts:



The previous two circuits represent different versions of the "sensor disagreement" logic circuit only, not the "good flame" detection circuit(s). The entire logic system would be the combination of both "good flame" and "sensor disagreement" circuits, shown on the same diagram.

Implemented in a Programmable Logic Controller (PLC), the entire logic system might resemble something like this:



As you can see, both the Sum-Of-Products and Products-Of-Sums standard Boolean forms are powerful tools when applied to truth tables. They allow us to derive a Boolean expression -- and ultimately, an actual logic circuit -- from nothing but a truth table, which is a written specification for what we want a logic circuit to do. To be able to go from a written specification to an actual circuit using simple, deterministic procedures means that it is possible to automate the design process for a digital circuit. In other words, a computer could be programmed to design a custom logic circuit from a truth table specification! The steps to take from a truth table to the final circuit are so unambiguous and direct that it requires little, if any, creativity or other original thought to execute them.

* **REVIEW:**
* *Sum-Of-Products*, or *SOP*, Boolean expressions may be generated from truth tables quite easily, by determining which rows of the table have an output of 1, writing one product term for each row, and finally summing all the product terms. This creates a Boolean expression representing the truth table as a whole.
* Sum-Of-Products expressions lend themselves well to implementation as a set of AND gates (products) feeding into a single OR gate (sum).
* *Product-Of-Sums*, or *POS*, Boolean expressions may also be generated from truth tables quite easily, by determining which rows of the table have an output of 0, writing one sum term for each row, and finally multiplying all the sum terms. This creates a Boolean expression representing the truth table as a whole.
* Product-Of-Sums expressions lend themselves well to implementation as a set of OR gates (sums) feeding into a single AND gate (product).

Here are some examples of converting binary directly into decimal. We simply add up the place values of each 1 digit in the binary number.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| * 1001012 = 3710:  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | Exponents | 25 |  | 24 |  | 23 |  | 22 |  | 21 |  | 20 | | Place Values | 32 |  | 16 |  | 8 |  | 4 |  | 2 |  | 1 | | Bits | 1 |  | 0 |  | 0 |  | 1 |  | 0 |  | 1 | | Value | 32 |  |  |  |  | + | 4 |  |  | + | 1 | = | 37 |  * 100011102 = 14210:  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | Exponents | 27 |  | 26 |  | 25 |  | 24 |  | 23 |  | 22 |  | 21 |  | 20 | | Place Values | 128 |  | 64 |  | 32 |  | 16 |  | 8 |  | 4 |  | 2 |  | 1 | | Bits | 1 |  | 0 |  | 0 |  | 0 |  | 1 |  | 1 |  | 1 |  | 0 | | Value | 128 |  |  |  |  |  |  | + | 8 | + | 4 | + | 2 |  |  | = | 142 | | * 1111010002 = 48810:  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | Exponents | 28 |  | 27 |  | 26 |  | 25 |  | 24 |  | 23 |  | 22 |  | 21 |  | 20 | | Place Values | 256 |  | 128 |  | 64 |  | 32 |  | 16 |  | 8 |  | 4 |  | 2 |  | 1 | | Bits | 1 |  | 1 |  | 1 |  | 1 |  | 0 |  | 1 |  | 0 |  | 0 |  | 0 | | Value | 256 | + | 128 | + | 64 | + | 32 |  |  | + | 8 |  |  |  |  |  |  | = | 488 |  * 101101012 = 18110:  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | Exponents | 27 |  | 26 |  | 25 |  | 24 |  | 23 |  | 22 |  | 21 |  | 20 | | Place Values | 128 |  | 64 |  | 32 |  | 16 |  | 8 |  | 4 |  | 2 |  | 1 | | Bits | 1 |  | 0 |  | 1 |  | 1 |  | 0 |  | 1 |  | 0 |  | 1 | | Value | 128 |  |  | + | 32 | + | 16 |  |  | + | 4 |  |  | + | 1 | = | 181 | |

Here are several examples conversions from decimal to binary using the left-to-right method. To convert the decimal number *n*, do this:

1. Choose the largest power of two less than or equal to *n*. Call it *p*.
2. Generate the initial one of your result.
3. Subtract *p* from *n*.
4. Divide *p* by 2.
5. Repeat as indicated:
   1. if *n* is less than *p*, append a 0 to your solution.
   2. if *n* is greater or eqaul to *p*, append a 1 to your solution and subtract *p* from *n*.
   3. If *p* is 1, stop. Otherwise, divide *p* by two and return to step a.

Just like this:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| * 89210 = 11011111002:  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | 29 | = | 512 | 892 | − | 512 | = | 380 | ⇒ | 1 | | 28 | = | 256 | 380 | − | 256 | = | 124 | ⇒ | 1 | | 27 | = | 128 |  |  |  |  |  | ⇒ | 0 | | 26 | = | 64 | 124 | − | 64 | = | 60 | ⇒ | 1 | | 25 | = | 32 | 60 | − | 32 | = | 28 | ⇒ | 1 | | 24 | = | 16 | 28 | − | 16 | = | 12 | ⇒ | 1 | | 23 | = | 8 | 12 | − | 8 | = | 4 | ⇒ | 1 | | 22 | = | 4 | 4 | − | 4 | = | 0 | ⇒ | 1 | | 21 | = | 2 |  |  |  |  |  | ⇒ | 0 | | 20 | = | 1 |  |  |  |  |  | ⇒ | 0 |  * 7110 = 10001112:  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | 26 | = | 64 | 71 | − | 64 | = | 7 | ⇒ | 1 | | 25 | = | 32 |  |  |  |  |  | ⇒ | 0 | | 24 | = | 16 |  |  |  |  |  | ⇒ | 0 | | 23 | = | 8 |  |  |  |  |  | ⇒ | 0 | | 22 | = | 4 | 7 | − | 4 | = | 3 | ⇒ | 1 | | 21 | = | 2 | 3 | − | 2 | = | 1 | ⇒ | 1 | | 20 | = | 1 | 1 | − | 1 | = | 0 | ⇒ | 1 | | * 3310 = 1000012:  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | 25 | = | 32 | 33 | − | 32 | = | 1 | ⇒ | 1 | | 24 | = | 16 |  |  |  |  |  | ⇒ | 0 | | 23 | = | 8 |  |  |  |  |  | ⇒ | 0 | | 22 | = | 4 |  |  |  |  |  | ⇒ | 0 | | 21 | = | 2 |  |  |  |  |  | ⇒ | 0 | | 20 | = | 1 | 1 | − | 1 | = | 0 | ⇒ | 1 |  * 19610 = 110001002:  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | 27 | = | 128 | 196 | − | 128 | = | 68 | ⇒ | 1 | | 26 | = | 64 | 68 | − | 64 | = | 4 | ⇒ | 1 | | 25 | = | 32 |  |  |  |  |  | ⇒ | 0 | | 24 | = | 16 |  |  |  |  |  | ⇒ | 0 | | 23 | = | 8 |  |  |  |  |  | ⇒ | 0 | | 22 | = | 4 | 4 | − | 4 | = | 0 | ⇒ | 1 | | 21 | = | 2 |  |  |  |  |  | ⇒ | 0 | | 20 | = | 1 |  |  |  |  |  | ⇒ | 0 | |

Here are some examples of converting hexadecimal into decimal. For each place, we multiply the digit value by the place value, and total the products.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| * 5a816 = 144810:  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | Exponent: | 162 |  | 161 |  | 160 | | Place Value: | 256 |  | 16 |  | 1 | | Digit Value: | 5 |  | 10 |  | 8 | | Product: | 1280 | + | 160 | + | 8 | = | 1448 |  * 2c0316 = 1126710:  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | Exponent: | 163 |  | 162 |  | 161 |  | 160 | | Place Value: | 4096 |  | 256 |  | 16 |  | 1 | | Digit Value: | 2 |  | 12 |  | 0 |  | 3 | | Product: | 8192 | + | 3072 | + | 0 | + | 3 | = | 11267 | | * 9516 = 14910:  |  |  |  |  | | --- | --- | --- | --- | | Exponent: | 161 |  | 160 | | Place Value: | 16 |  | 1 | | Digit Value: | 9 |  | 5 | | Product: | 144 | + | 5 | = | 149 |  * b1716 = 283910:  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | Exponent: | 162 |  | 161 |  | 160 | | Place Value: | 256 |  | 16 |  | 1 | | Digit Value: | 11 |  | 1 |  | 7 | | Product: | 2816 | + | 16 | + | 7 | = | 2839 | |

Here are some examples of binary addition. These are computed without regard to the word size, hence there can be no sense of "overflow." Work through the columns right to left, add up the ones and express the answer in *binary*. The low bit goes in the sum, and the high bit carries to the next column left.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| • 10001 + 11101 = 101110:   |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | |  | 1 |  |  |  | 1 |  | |  |  | 1 | 0 | 0 | 0 | 1 | | + |  | 1 | 1 | 1 | 0 | 1 | |  | 1 | 0 | 1 | 1 | 1 | 0 | | • 101101 + 11001 = 1000110:   |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | |  | 1 | 1 | 1 |  |  | 1 |  | |  |  | 1 | 0 | 1 | 1 | 0 | 1 | | + |  |  | 1 | 1 | 0 | 0 | 1 | |  | 1 | 0 | 0 | 0 | 1 | 1 | 0 | | • 1011001 + 111010 = 10010011:   |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | |  | 1 | 1 | 1 | 1 |  |  |  |  | |  |  | 1 | 0 | 1 | 1 | 0 | 0 | 1 | | + |  |  | 1 | 1 | 1 | 0 | 1 | 0 | |  | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | |
| • 1110 + 1111 = 11101:   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  | 1 | 1 | 1 |  |  | |  |  | 1 | 1 | 1 | 0 | | + |  | 1 | 1 | 1 | 1 | |  | 1 | 1 | 1 | 0 | 1 | | • 10111 + 110101 = 1001100:   |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | |  | 1 | 1 |  | 1 | 1 | 1 |  | |  |  |  | 1 | 0 | 1 | 1 | 1 | | + |  | 1 | 1 | 0 | 1 | 0 | 1 | |  | 1 | 0 | 0 | 1 | 1 | 0 | 0 | | • 11011 + 1001010 = 1100101:   |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | |  |  | 1 | 1 |  | 1 |  |  | |  |  |  | 1 | 1 | 0 | 1 | 1 | | + | 1 | 0 | 0 | 1 | 0 | 1 | 0 | |  | 1 | 1 | 0 | 0 | 1 | 0 | 1 | |

Here are some examples of binary subtraction. These are computed without regard to the word size, hence there can be no sense of "overflow" or "underflow". Work the columns right to left subtracting in each column. If you must subtract a one from a zero, you need to “borrow” from the left, just as in decimal subtraction.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| • 1011011 − 10010 = 1001001:   |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | |  |  |  |  |  |  |  |  | |  | 1 | 0 | 1 | 1 | 0 | 1 | 1 | | − |  |  | 1 | 0 | 0 | 1 | 0 | |  | 1 | 0 | 0 | 1 | 0 | 0 | 1 | | • 1010110 − 101010 = 101100:   |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | |  | 0 |  | 0 |  |  |  |  | |  | ×1 | 10 | ×1 | 10 | 1 | 1 | 0 | | − |  | 1 | 0 | 1 | 0 | 1 | 0 | |  |  | 1 | 0 | 1 | 1 | 0 | 0 | | • 1000101 − 101100 = 11001:   |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | |  | 0 | 1 | 1 |  |  |  |  | |  | ×1 | ×10 | ×10 | 10 | 1 | 0 | 1 | | − |  | 1 | 0 | 1 | 1 | 0 | 0 | |  |  |  | 1 | 1 | 0 | 0 | 1 | |
| • 100010110 − 1111010 = 10011100:   |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | |  | 0 | 1 | 1 | 1 | 10 |  |  |  |  | |  | ×1 | ×10 | ×10 | ×10 | ×1 | 10 | 1 | 1 | 0 | | − |  |  | 1 | 1 | 1 | 1 | 0 | 1 | 0 | |  |  | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | | • 101101 − 100111 = 110:   |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | |  |  |  | 0 | 10 |  |  | |  | 1 | 0 | ×1 | ×1 | 10 | 1 | | − | 1 | 0 | 0 | 1 | 1 | 1 | |  |  |  |  | 1 | 1 | 0 | | • 1110110 − 1010111 = 11111:   |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | |  |  | 0 | 10 | 1 | 10 | 10 |  | |  | 1 | ×1 | ×1 | ×10 | ×1 | ×1 | 10 | | − | 1 | 0 | 1 | 0 | 1 | 1 | 1 | |  |  |  | 1 | 1 | 1 | 1 | 1 | |

**he Conversion Procedure**

The rules for converting a decimal number into floating point are as follows:

1. Convert the absolute value of the number to binary, perhaps with a fractional part after the binary point. This can be done by converting the integral and fractional parts separately. The integral part is converted with the techniques examined previously. The fractional part can be converted by multiplication. This is basically the inverse of the division method: we repeatedly multiply by 2, and harvest each one bit as it appears left of the decimal.
2. Append × 20 to the end of the binary number (which does not change its value).
3. Normalize the number. Move the binary point so that it is one bit from the left. Adjust the exponent of two so that the value does not change.
4. Place the mantissa into the mantissa field of the number. Omit the leading one, and fill with zeros on the right.
5. Add the bias to the exponent of two, and place it in the exponent field. The bias is 2*k*−1 − 1, where *k* is the number of bits in the exponent field. For the eight-bit format, *k* = 3, so the bias is 23−1 − 1 = 3. For IEEE 32-bit, *k* = 8, so the bias is 28−1 − 1 = 127.
6. Set the sign bit, 1 for negative, 0 for positive, according to the sign of the original number.

**Using The Conversion Procedure**

* Convert 2.625 to our 8-bit floating point format.
  1. The integral part is easy, 210 = 102. For the fractional part:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0.625 | × 2 = | 1.25 | 1 | Generate 1 and continue with the rest. |
| 0.25 | × 2 = | 0.5 | 0 | Generate 0 and continue. |
| 0.5 | × 2 = | 1.0 | 1 | Generate 1 and nothing remains. |

* 1. So 0.62510 = 0.1012, and 2.62510 = 10.1012.
  2. Add an exponent part: 10.1012 = 10.1012 × 20.
  3. Normalize: 10.1012 × 20 = 1.01012 × 21.
  4. Mantissa: 0101
  5. Exponent: 1 + 3 = 4 = 1002.
  6. Sign bit is 0.

The result is 01000101. Represented as hex, that is 4516.

* Convert -4.75 to our 8-bit floating point format.
  1. The integral part is 410 = 1002. The fractional:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0.75 | × 2 = | 1.5 | 1 | Generate 1 and continue with the rest. |
| 0.5 | × 2 = | 1.0 | 1 | Generate 1 and nothing remains. |

* 1. So 4.7510 = 100.112.
  2. Normalize: 100.112 = 1.00112 × 22.
  3. Mantissa is 0011, exponent is 2 + 3 = 5 = 1012, sign bit is 1.

So -4.75 is 11010011 = d316

* Convert 0.40625 to our 8-bit floating point format.
  1. Converting:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0.40625 | × 2 = | 0.8125 | 0 | Generate 0 and continue. |
| 0.8125 | × 2 = | 1.625 | 1 | Generate 1 and continue with the rest. |
| 0.625 | × 2 = | 1.25 | 1 | Generate 1 and continue with the rest. |
| 0.25 | × 2 = | 0.5 | 0 | Generate 0 and continue. |
| 0.5 | × 2 = | 1.0 | 1 | Generate 1 and nothing remains. |

* 1. So 0.4062510 = 0.011012.
  2. Normalize: 0.011012 = 1.1012 × 2-2.
  3. Mantissa is 1010, exponent is -2 + 3 = 1 = 0012, sign bit is 0.

So 0.40625 is 00011010 = 1a16

* Convert -12.0 to our 8-bit floating point format.
  1. 1210 = 11002.
  2. Normalize: 1100.02 = 1.12 × 23.
  3. Mantissa is 1000, exponent is 3 + 3 = 6 = 1102, sign bit is 1.

So -12.0 is 11101000 = e816

* Convert decimal 1.7 to our 8-bit floating point format.
  1. The integral part is easy, 110 = 12. For the fractional part:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0.7 | × 2 = | 1.4 | 1 | Generate 1 and continue with the rest. |
| 0.4 | × 2 = | 0.8 | 0 | Generate 0 and continue. |
| 0.8 | × 2 = | 1.6 | 1 | Generate 1 and continue with the rest. |
| 0.6 | × 2 = | 1.2 | 1 | Generate 1 and continue with the rest. |
| 0.2 | × 2 = | 0.4 | 0 | Generate 0 and continue. |
| 0.4 | × 2 = | 0.8 | 0 | Generate 0 and continue. |
| 0.8 | × 2 = | 1.6 | 1 | Generate 1 and continue with the rest. |
| 0.6 | × 2 = | 1.2 | 1 | Generate 1 and continue with the rest. |
| … | | | |  |

* 1. The reason why the process seems to continue endlessly is that it does. The number 7/10, which makes a perfectly reasonable decimal fraction, is a repeating fraction in binary, just as the faction 1/3 is a repeating fraction in decimal. (It repeats in binary as well.) We cannot represent this exactly as a floating point number. The closest we can come in four bits is .1011. Since we already have a leading 1, the best eight-bit number we can make is 1.1011.
  2. Already normalized: 1.10112 = 1.10112 × 20.
  3. Mantissa is 1011, exponent is 0 + 3 = 3 = 0112, sign bit is 0.

The result is 00111011 = 3b16. This is not exact, of course. If you convert it back to decimal, you get 1.6875.

* Convert -1313.3125 to IEEE 32-bit floating point format.
  1. The integral part is 131310 = 101001000012. The fractional:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0.3125 | × 2 = | 0.625 | 0 | Generate 0 and continue. |
| 0.625 | × 2 = | 1.25 | 1 | Generate 1 and continue with the rest. |
| 0.25 | × 2 = | 0.5 | 0 | Generate 0 and continue. |
| 0.5 | × 2 = | 1.0 | 1 | Generate 1 and nothing remains. |

* 1. So 1313.312510 = 10100100001.01012.
  2. Normalize: 10100100001.01012 = 1.010010000101012 × 210.
  3. Mantissa is 01001000010101000000000, exponent is 10 + 127 = 137 = 100010012, sign bit is 1.

So -1313.3125 is 11000100101001000010101000000000 = c4a42a0016

* Convert 0.1015625 to IEEE 32-bit floating point format.
  1. Converting:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0.1015625 | × 2 = | 0.203125 | 0 | Generate 0 and continue. |
| 0.203125 | × 2 = | 0.40625 | 0 | Generate 0 and continue. |
| 0.40625 | × 2 = | 0.8125 | 0 | Generate 0 and continue. |
| 0.8125 | × 2 = | 1.625 | 1 | Generate 1 and continue with the rest. |
| 0.625 | × 2 = | 1.25 | 1 | Generate 1 and continue with the rest. |
| 0.25 | × 2 = | 0.5 | 0 | Generate 0 and continue. |
| 0.5 | × 2 = | 1.0 | 1 | Generate 1 and nothing remains. |

* 1. So 0.101562510 = 0.00011012.
  2. Normalize: 0.00011012 = 1.1012 × 2-4.
  3. Mantissa is 10100000000000000000000, exponent is -4 + 127 = 123 = 011110112, sign bit is 0.

So 0.1015625 is 00111101110100000000000000000000 = 3dd0000016

* Convert 39887.5625 to IEEE 32-bit floating point format.
  1. The integral part is 3988710 = 10011011110011112. The fractional:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0.5625 | × 2 = | 1.125 | 1 | Generate 1 and continue with the rest. |
| 0.125 | × 2 = | 0.25 | 0 | Generate 0 and continue. |
| 0.25 | × 2 = | 0.5 | 0 | Generate 0 and continue. |
| 0.5 | × 2 = | 1.0 | 1 | Generate 1 and nothing remains. |

* 1. So 39887.562510 = 1001101111001111.10012.
  2. Normalize: 1001101111001111.10012 = 1.00110111100111110012 × 215.
  3. Mantissa is 00110111100111110010000, exponent is 15 + 127 = 142 = 100011102, sign bit is 0.

So 39887.5625 is 01000111000110111100111110010000 = 471bcf9016

**The Conversion Procedure**

The rules for converting a floating point number into decimal are simply to reverse of the decimal to floating point conversion:

1. If the original number is in hex, convert it to binary.
2. Separate into the sign, exponent, and mantissa fields.
3. Extract the mantissa from the mantissa field, and restore the leading one. You may also omit the trailing zeros.
4. Extract the exponent from the exponent field, and subtract the bias to recover the actual exponent of two. As before, the bias is 2*k*−1 − 1, where *k* is the number of bits in the exponent field, giving 3 for the 8-bit format and 127 for the 32-bit.
5. De-normalize the number: move the binary point so the exponent is 0, and the value of the number remains unchanged.
6. Convert the binary value to decimal. This is done just as with binary integers, but the place values right of the binary point are fractions.
7. Set the sign of the decimal number according to the sign bit of the original floating point number: make it negative for 1; leave positive for 0.

If the binary exponent is very large or small, you can convert the mantissa directly to decimal without de-normalizing. Then use a calculator to raise two to the exponent, and perform the multiplication. This will give an approximate answer, but is sufficient in most cases.

**Examples Using The Conversion Procedure**

* Convert the 8-bit floating point number e7 (in hex) to decimal.
  1. Convert: e716 = 111001112.
  2. Seprate: 11100111
  3. Mantissa: 1.0111
  4. Exponent: 1102 = 610; 6 − 3 = 3.
  5. De-normalize: 1.01112 × 23 = 1011.1
  6. Convert:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Exponents | 23 |  | 22 |  | 21 |  | 20 |  | 2-1 |
| Place Values | 8 |  | 4 |  | 2 |  | 1 |  | 0.5 |
| Bits | 1 |  | 0 |  | 1 |  | 1 | . | 1 |
| Value | 8 |  |  | + | 2 | + | 1 | + | 0.5 | = | 11.5 |

* 1. Sign: negative.

Result: e7 is -11.5

* Convert the 8-bit floating point number 26 (in hex) to decimal.
  1. Convert and separate: 2616 = 00100110 2
  2. Exponent: 0102 = 210; 2 − 3 = -1.
  3. Denormalize: 1.0112 × 2-1 = 0.1011.
  4. Convert:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Exponents | 20 |  | 2-1 |  | 2-2 |  | 2-3 |  | 2-4 |
| Place Values | 1 |  | 0.5 |  | 0.25 |  | 0.125 |  | 0.0625 |
| Bits | 0 | . | 1 |  | 0 |  | 1 |  | 1 |
| Value |  |  | 0.5 |  |  | + | 0.125 | + | 0.0625 | = | 0.6875 |

* 1. Sign: positive

Result: 26 is 0.6875.

* Convert the 8-bit floating point number d3 (in hex) to decimal.
  1. Convert and separate: d316 = 11010011 2
  2. Exponent: 1012 = 510; 5 − 3 = 2.
  3. Denormalize: 1.00112 × 22 = 100.11.
  4. Convert:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Exponents | 22 |  | 21 |  | 20 |  | 2-1 |  | 2-2 |
| Place Values | 4 |  | 2 |  | 1 |  | 0.5 |  | 0.25 |
| Bits | 1 |  | 0 |  | 0 | . | 1 |  | 1 |
| Value | 4 |  |  |  |  | + | 0.5 | + | 0.25 | = | 4.75 |

* 1. Sign: negative

Result: d3 is -4.75.

* Convert the 32-bit floating point number 44361000 (in hex) to decimal.
  1. Convert and separate: 4436100016 = 01000100001101100001000000000000 2
  2. Exponent: 100010002 = 13610; 136 − 127 = 9.
  3. Denormalize: 1.011011000012 × 29 = 1011011000.01.
  4. Convert:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Exponents | 29 |  | 28 |  | 27 |  | 26 |  | 25 |  | 24 |  | 23 |  | 22 |  | 21 |  | 20 |  | 2-1 |  | 2-2 |
| Place Values | 512 |  | 256 |  | 128 |  | 64 |  | 32 |  | 16 |  | 8 |  | 4 |  | 2 |  | 1 |  | 0.5 |  | 0.25 |
| Bits | 1 |  | 0 |  | 1 |  | 1 |  | 0 |  | 1 |  | 1 |  | 0 |  | 0 |  | 0 | . | 0 |  | 1 |
| Value | 512 |  |  | + | 128 | + | 64 |  |  | + | 16 | + | 8 |  |  |  |  |  |  |  |  | + | 0.25 | = | 728.25 |

* 1. Sign: positive

Result: 44361000 is 728.25.

* Convert the 32-bit floating point number be580000 (in hex) to decimal.
  1. Convert and separate: be58000016 = 10111110010110000000000000000000 2
  2. Exponent: 011111002 = 12410; 124 − 127 = -3.
  3. Denormalize: 1.10112 × 2-3 = 0.0011011.
  4. Convert:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Exponents | 20 |  | 2-1 |  | 2-2 |  | 2-3 |  | 2-4 |  | 2-5 |  | 2-6 |  | 2-7 |
| Place Values | 1 |  | 0.5 |  | 0.25 |  | 0.125 |  | 0.0625 |  | 0.03125 |  | 0.015625 |  | 0.0078125 |
| Bits | 0 | . | 0 |  | 0 |  | 1 |  | 1 |  | 0 |  | 1 |  | 1 |
| Value |  |  |  |  |  |  | 0.125 | + | 0.0625 |  |  | + | 0.015625 | + | 0.0078125 | = | 0.2109375 |

* 1. Sign: negative

Result: be580000 is -0.2109375.

* Convert the 32-bit floating point number a3358000 (in hex) to decimal.
  1. Convert and separate: a335800016 = 10100011001101011000000000000000 2
  2. Exponent: 010001102 = 7010; 70 − 127 = -57.
  3. Since the exponent is far from zero, convert the original (normalized) mantissa:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Exponents | 20 |  | 2-1 |  | 2-2 |  | 2-3 |  | 2-4 |  | 2-5 |  | 2-6 |  | 2-7 |  | 2-8 |
| Place Values | 1 |  | 0.5 |  | 0.25 |  | 0.125 |  | 0.0625 |  | 0.03125 |  | 0.015625 |  | 0.0078125 |  | 0.00390625 |
| Bits | 1 | . | 0 |  | 1 |  | 1 |  | 0 |  | 1 |  | 0 |  | 1 |  | 1 |
| Value | 1 |  |  | + | 0.25 | + | 0.125 |  |  | + | 0.03125 |  |  | + | 0.0078125 | + | 0.00390625 | = | 1.41796875 |

* 1. Use calculator to find 1.41796875 × 2-57. You should get something like 9.83913471531 × 10-18 .
  2. Sign: negative

Result: a3358000 is about -9.83913471531 × 10-18 .

* Convert the 32-bit floating point number 76650000 (in hex) to decimal.
  1. Convert and separate: 7665000016 = 01110110011001010000000000000000 2
  2. Exponent: 111011002 = 23610; 236 − 127 = 109.
  3. Since the exponent is far from zero, convert the original (normalized) mantissa:

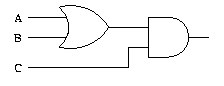
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Exponents | 20 |  | 2-1 |  | 2-2 |  | 2-3 |  | 2-4 |  | 2-5 |  | 2-6 |  | 2-7 |
| Place Values | 1 |  | 0.5 |  | 0.25 |  | 0.125 |  | 0.0625 |  | 0.03125 |  | 0.015625 |  | 0.0078125 |
| Bits | 1 | . | 1 |  | 1 |  | 0 |  | 0 |  | 1 |  | 0 |  | 1 |
| Value | 1 | + | 0.5 | + | 0.25 |  |  |  |  | + | 0.03125 |  |  | + | 0.0078125 | = | 1.7890625 |

* 1. Use calculator to find 1.7890625 × 2109. You should get something like 1.16116794981 × 1033 .
  2. Sign: positive

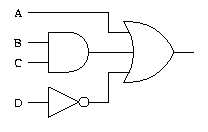
Result: 76650000 is about 1.16116794981 × 1033 .

Here are some logic gate circuit problems:

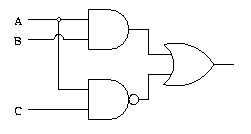
* Draw a logic circuit for (A + B)C.



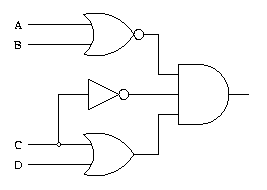
* Draw a logic circuit for A + BC + D.



* Draw a logic circuit for AB + AC.



* Draw a logic circuit for (A + B)(C + D)C.



Here are some examples of Boolean algebra simplifications. Each line gives a form of the expression, and the rule or rules used to derive it from the previous one. Generally, there are several ways to reach the result. [Here is the list of simplification rules](http://sandbox.mc.edu/%7Ebennet/cs110/boolalg/rules.html).

* Simplify: C + BC:

|  |  |
| --- | --- |
| Expression | Rule(s) Used |
| C + BC | Original Expression |
|  |  |
| C + (B + C) | DeMorgan's Law. |
| (C + C) + B | Commutative, Associative Laws. |
| *T* + B | Complement Law. |
| *T* | Identity Law. |

* Simplify: AB(A + B)(B + B):

|  |  |
| --- | --- |
| Expression | Rule(s) Used |
| AB(A + B)(B + B) | Original Expression |
|  |  |
| AB(A + B) | Complement law, Identity law. |
| (A + B)(A + B) | DeMorgan's Law |
| A + BB | Distributive law. This step uses the fact that or distributes over and. It can look a bit strange since addition does not distribute over multiplication. |
| A | Complement, Identity. |

* Simplify: (A + C)(AD + AD) + AC + C:

|  |  |
| --- | --- |
| Expression | Rule(s) Used |
| (A + C)(AD + AD) + AC + C | Original Expression |
|  |  |
| (A + C)A(D + D) + AC + C | Distributive. |
| (A + C)A + AC + C | Complement, Identity. |
| A((A + C) + C) + C | Commutative, Distributive. |
| A(A + C) + C | Associative, Idempotent. |
| AA + AC + C | Distributive. |
| A + (A + *T*)C | Idempotent, Identity, Distributive. |
| A + C | Identity, twice. |

* You can also use distribution of or over and starting from A(A+C)+C to reach the same result by another route.
* Simplify: A(A + B) + (B + AA)(A + B):

|  |  |
| --- | --- |
| Expression | Rule(s) Used |
| A(A + B) + (B + AA)(A + B) | Original Expression |
|  |  |
| AA + AB + (B + A)A + (B + A)B | Idempotent (AA to A), then Distributive, used twice. |
| AB + (B + A)A + (B + A)B | Complement, then Identity. (Strictly speaking, we also used the Commutative Law for each of these applications.) |
| AB + BA + AA + BB + AB | Distributive, two places. |
| AB + BA + A + AB | Idempotent (for the A's), then Complement and Identity to remove BB. |
| AB + AB + A*T* + AB | Commutative, Identity; setting up for the next step. |
| AB + A(B + *T* + B) | Distributive. |
| AB + A | Identity, twice (depending how you count it). |
| A + AB | Commutative. |
| (A + A)(A + B) | Distributive. |
| A + B | Complement, Identity. |