Digital electronics

Digital electronics, **digital technology** or **digital (electronic) circuits** are electronics that operate on digital signals. In contrast, analog circuits manipulate analog signals whose performance is more subject to manufacturing tolerance, signal attenuation and noise. Digital techniques are helpful because it is much easier to get an electronic device to switch into one of a number of known states than to accurately reproduce a continuous range of values.

Digital electronic circuits are usually made from large assemblies of logic gates (often printed on integrated circuits), simple electronic representations of Boolean logic functions

Binary number

In mathematics and digital electronics, a **binary number** is a number expressed in the **base-2 numeral system** or **binary numeral system**, which uses only two symbols: typically "0" (zero) and "1" (one).

The base-2 numeral system is a positional notation with a radix of 2. Each digit is referred to as a bit. Because of its straightforward implementation in digital electronic circuitry using logic gates, the binary system is used by almost all modern computers and computer-based devices.

Decimal counting

Decimal counting uses the ten symbols *0* through *9*. Counting begins with the incremental substitution of the least significant digit (rightmost digit) which is often called the *first digit*. When the available symbols for this position are exhausted, the least significant digit is reset to *0*, and the next digit of higher significance (one position to the left) is incremented (*overflow*), and incremental substitution of the low-order digit resumes. This method of reset and overflow is repeated for each digit of significance. Counting progresses as follows:

> 000, 001, 002, ... 007, 008, 009, (rightmost digit is reset to zero, and the digit to its left is incremented)
>
> 0**1**0, 011, 012, ...
>
> ...
>
> 090, 091, 092, ... 097, 098, 099, (rightmost two digits are reset to zeroes, and next digit is incremented)
>
> **1**00, 101, 102, ...

Binary counting

Binary counting follows the same procedure, except that only the two symbols *0* and *1* are available. Thus, after a digit reaches 1 in binary, an increment resets it to 0 but also causes an increment of the next digit to the left:

> 0000,
>
> 000**1**, (rightmost digit starts over, and next digit is incremented)
>
> 00**1**0, 0011, (rightmost two digits start over, and next digit is incremented)

Addition

Adder

The simplest arithmetic operation in binary is addition. Adding two single-digit binary numbers is relatively simple, using a form of carrying:

0 + 0 → 0

0 + 1 → 1

1 + 0 → 1

1 + 1 → 0, carry 1 (since 1 + 1 = 2 = 0 + (1 × 2^1))

Adding two "1" digits produces a digit "0", while 1 will have to be added to the next column. This is similar to what happens in decimal when certain single-digit numbers are added together; if the result equals or exceeds the value of the radix (10), the digit to the left is incremented:

5 + 5 → 0, carry 1 (since 5 + 5 = 10 = 0 + (1 × 10^1))

7 + 9 → 6, carry 1 (since 7 + 9 = 16 = 6 + (1 × 10^1))

This is known as *carrying*. When the result of an addition exceeds the value of a digit, the procedure is to "carry" the excess amount divided by the radix (that is, 10/10) to the left, adding it to the next positional value. This is correct since the next position has a weight that is higher by a factor equal to the radix. Carrying works the same way in binary:

0**1**00, 0101, 0110, 0111, (rightmost three digits start over, and the next digit is incremented)

1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111 ...

In the binary system, each digit represents an increasing power of 2, with the rightmost digit representing 2^0, the next representing 2^1, then 2^2, and so on. The value of a binary number is the sum of the powers of 2 represented by each "1" digit. For example, the binary number 100101 is converted to decimal form as follows:

100101_2 = [(**1**) × 2^5] + [(**0**) × 2^4] + [(**0**) × 2^3] + [(**1**) × 2^2] + [(**0**) × 2^1] + [(**1**) × 2^0]

100101_2 = [**1** × 32] + [**0** × 16] + [**0** × 8] + [**1** × 4] + [**0** × 2] + [**1** × 1]

100101_2 = 37_{10}

Subtraction

Further information: signed number representations and two's complement

Subtraction works in much the same way:

0 − 0 → 0

0 − 1 → 1, borrow 1

1 − 0 → 1

1 − 1 → 0

Subtracting a "1" digit from a "0" digit produces the digit "1", while 1 will have to be subtracted from the next column. This is known as *borrowing*. The principle is the same as for carrying. When the result of a subtraction is less than 0, the least possible value of a digit, the procedure is to "borrow" the deficit divided by the radix (that is, 10/10) from the left, subtracting it from the next positional value.

Multiplication

Multiplication in binary is similar to its decimal counterpart. Two numbers A and B can be multiplied by partial products: for each digit in B, the product of that digit in A is calculated and written on a new line, shifted leftward so that its rightmost digit lines up with the digit in B that was used. The sum of all these partial products gives the final result.

Since there are only two digits in binary, there are only two possible outcomes of each partial multiplication:

- If the digit in *B* is 0, the partial product is also 0
- If the digit in *B* is 1, the partial product is equal to *A*

Division

Long division in binary is again similar to its decimal counterpart.

In the example below, the divisor is 101_2, or 5 in decimal, while the dividend is 11011_2, or 27 in decimal. The procedure is the same as that of decimal long division; here, the divisor 101_2 goes into the first three digits 110_2 of the dividend one time, so a "1" is written on the top line. This result is multiplied by the divisor, and subtracted from the first three digits of the dividend; the next digit (a "1") is included to obtain a new three-digit sequence:

Conversion to and from other numeral systems

Decimal

To convert from a base-10 integer to its base-2 (binary) equivalent, the number is divided by two. The remainder is the least-significant bit. The quotient is again divided by two; its remainder becomes the next least significant bit. This process repeats until a quotient of one is reached. The sequence of remainders (including the final quotient of one) forms the binary value, as each remainder must be either zero or one when dividing by two. For example, $(357)_{10}$ is expressed as $(101100101)_{2.}$

Conversion from base-2 to base-10 simply inverts the preceding algorithm. The bits of the binary number are used one by one, starting with the most significant (leftmost) bit. Beginning with the value 0, the prior value is doubled, and the next bit is then added to produce the next value. This can be organized in a multi-column table. For example, to convert 10010101101_2 to decimal:

Binary	1	0	0	1	0	1	0	1	1	0	1

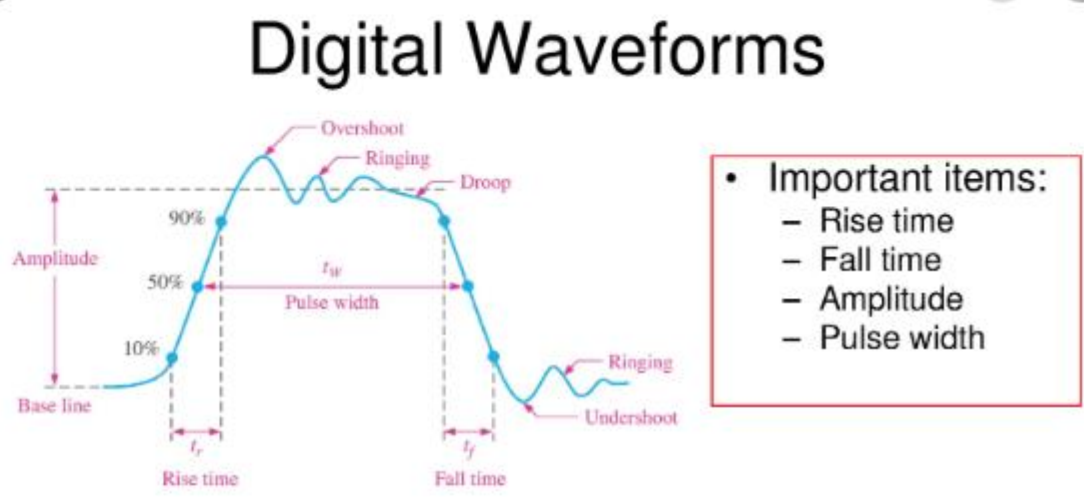
Decimal	1×2^{10} +	0×2^{9} +	0×2^{8} +	1×2^{7} +	0×2^{6} +	1×2^{5} +	0×2^{4} +	1×2^{3} +	1×2^{2} +	0×2^{1} +	1×2^{0} =	1197

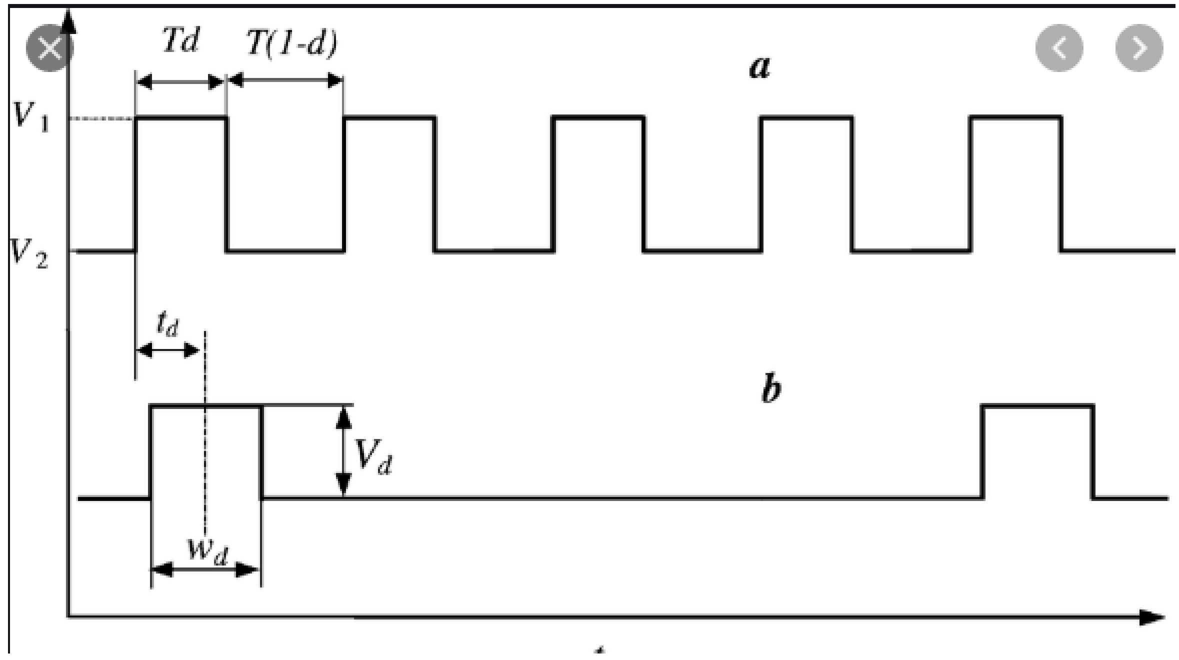
The fractional parts of a number are converted with similar methods. They are again based on the equivalence of shifting with doubling or halving.

In a fractional binary number such as 0.11010110101_2, the first digit is 1/2, the second (1/2)^2, etc. So if there is a 1 in the first place after the decimal, then the number is at least (1/2), and vice versa. Double that number is at least 1. This suggests the algorithm: Repeatedly double the number to be converted, record if the result is at least 1, and then throw away the integer part.

Digital signal

A **digital signal** is a signal that is being used to represent data as a sequence of discrete values; at any given time it can only take on one of a finite number of values.This contrasts with an analog signal, which represents continuous values; at any given time it represents a real number within a continuous range of values.





Logic gate

A **logic gate** is an idealized or physical electronic device implementing a Boolean function, a logical operation performed on one or more binary inputs that produces a single binary output. Depending on the context, the term may refer to an **ideal logic gate**, one that has for instance zero rise time and unlimited fan-out, or it may refer to a non-ideal physical device

Logic gates are primarily implemented using diodes or transistors acting as electronic switches, but can also be constructed using vacuum tubes, electromagnetic relays (relay logic), fluidic logic, pneumatic logic, optics, molecules, or even mechanical elements. With amplification, logic gates can be cascaded in the same way that Boolean functions can be composed, allowing the construction of a physical model of all of Boolean logic, and therefore, all of the algorithms and mathematics that can be described with Boolean logic.

Logic circuits include such devices as multiplexers, registers, arithmetic logic units (ALUs), and computer memory, all the way up through complete microprocessors, which may contain more than 100 million gates. In modern practice, most gates are made from MOSFETs (metal–oxide–semiconductor field-effect transistors).

Compound logic gates AND-OR-Invert (AOI) and OR-AND-Invert (OAI) are often employed in circuit design because their construction using MOSFETs is simpler and more efficient than the sum of the individual gates

Type	Distinctive shape	Rectangular shape	Boolean algebra between A & B	Truth table
AND	A, B — Q	&	$A \cdot B$ or $A \land B$	INPUT A B / OUTPUT Q 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1
OR	A, B — Q	≥1	$A + B$ or $A \lor B$	INPUT A B / OUTPUT Q 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1
NAND	A, B — Q	&	$\overline{A \cdot B}$ or $A \uparrow B$	INPUT A B / OUTPUT Q 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0
NOR	A, B — Q	≥1	$\overline{A + B}$ or $A \downarrow B$	INPUT A B / OUTPUT Q 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0
Exclusive or and Biconditional				
XOR	A, B — Q	=1	$A \oplus B$ or $A \veebar B$	INPUT A B / OUTPUT Q 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0
XNOR	A, B — Q	=1	$\overline{A \oplus B}$ or $A \odot B$	INPUT A B / OUTPUT Q 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 1

De Morgan equivalent symbols

By use of De Morgan's laws, an *AND* function is identical to an *OR* function with negated inputs and outputs. Likewise, an *OR* function is identical to an *AND* function with negated

inputs and outputs. A NAND gate is equivalent to an OR gate with negated inputs, and a NOR gate is equivalent to an AND gate with negated inputs.

This leads to an alternative set of symbols for basic gates that use the opposite core symbol (*AND* or *OR*) but with the inputs and outputs negated. Use of these alternative symbols can make logic circuit diagrams much clearer and help to show accidental connection of an active high output to an active low input or vice versa. Any connection that has logic negations at both ends can be replaced by a negationless connection and a suitable change of gate or vice versa. Any connection that has a negation at one end and no negation at the other can be made easier to interpret by instead using the De Morgan equivalent symbol at either of the two ends. When negation or polarity indicators on both ends of a connection match, there is no logic negation in that path (effectively, bubbles "cancel"), making it easier to follow logic states from one symbol to the next. This is commonly seen in real logic diagrams – thus the reader must not get into the habit of associating the shapes exclusively as OR or AND shapes, but also take into account the bubbles at both inputs and outputs in order to determine the "true" logic function indicated.

A De Morgan symbol can show more clearly a gate's primary logical purpose and the polarity of its nodes that are considered in the "signaled" (active, on) state. Consider the simplified case where a two-input NAND gate is used to drive a motor when either of its inputs are brought low by a switch. The "signaled" state (motor on) occurs when either one OR the other switch is on. Unlike a regular NAND symbol, which suggests AND logic, the De Morgan version, a two negative-input OR gate, correctly shows that OR is of interest. The regular NAND symbol has a bubble at the output and none at the inputs (the opposite of the states that will turn the motor on), but the De Morgan symbol shows both inputs and output in the polarity that will drive the motor.

Boolean algebra

In mathematics and mathematical logic, **Boolean algebra** is the branch of algebra in which the values of the variables are the truth values *true* and *false*, usually denoted 1 and 0 respectively. Instead of elementary algebra where the values of the variables are numbers, and the prime operations are addition and multiplication, the main operations of Boolean algebra are the conjunction (*and)* denoted as ∧, the disjunction (*or)* denoted as ∨, and the negation (*not)* denoted as ¬. It is thus a formalism for describing logical operations in the same way that elementary algebra describes numerical operations.

Basic operations

The basic operations of Boolean algebra are as follows:

- **AND** (conjunction), denoted $x \wedge y$ (sometimes x AND y or Kxy), satisfies $x \wedge y = 1$ if $x = y = 1$, and $x \wedge y = 0$ otherwise.
- **OR** (disjunction), denoted $x \vee y$ (sometimes x OR y or Axy), satisfies $x \vee y = 0$ if $x = y = 0$, and $x \vee y = 1$ otherwise.
- **NOT** (negation), denoted $\neg x$ (sometimes NOT x, Nx or $!x$), satisfies $\neg x = 0$ if $x = 1$ and $\neg x = 1$ if $x = 0$.

Alternatively the values of $x \wedge y$, $x \vee y$, and $\neg x$ can be expressed by tabulating their values with truth tables as follows:

0	**0**	0	0	**0**	1
1	**0**	0	1	**1**	0
0	**1**	0	1		
1	**1**	1	1		

If the truth values 0 and 1 are interpreted as integers, these operations may be expressed with the ordinary operations of arithmetic (where $x + y$ uses addition and xy uses multiplication), or by the minimum/maximum functions:

One might consider that only negation and one of the two other operations are basic, because of the following identities that allow one to define conjunction in terms of negation and the disjunction, and vice versa (De Morgan's laws):

Secondary operations

The three Boolean operations described above are referred to as basic, meaning that they can be taken as a basis for other Boolean operations that can be built up from them by **composition,** the manner in which operations are combined or compounded.

These definitions give rise to the following truth tables giving the values of these operations for all four possible inputs.

Secondary operations. Table 1

0	**0**	1	0	1
1	**0**	0	1	0
0	**1**	1	1	0
1	**1**	1	0	1

The first operation, $x \rightarrow y$, or C*xy*, is called **material implication**. If *x* is true then the value of $x \rightarrow y$ is taken to be that of *y* (e.g. if *x* is true and *y* is false, then $x \rightarrow y$ is also false). But if *x* is false then the value of *y* can be ignored; however the operation must return *some* boolean value and there are only two choices. So by definition, $x \rightarrow y$ is *true* when x is false. (Relevance logic suggests this definition by viewing an implication with a false premise as something other than either true or false.)

The second operation, $x \oplus y$, or J*xy*, is called **exclusive or** (often abbreviated as XOR) to distinguish it from disjunction as the inclusive kind. It excludes the possibility of both *x* and *y* *being* true (e.g. see table): if both are true then result is false. Defined in terms of arithmetic it is addition mod 2 where 1 + 1 = 0.

The third operation, the complement of exclusive or, is **equivalence** or Boolean equality: $x \equiv y$, or E*xy*, is true just when *x* and *y* have the same value. Hence $x \oplus y$ as its complement can be understood as $x \neq y$, being true just when *x* and *y* are different. Thus, its counterpart in arithmetic mod 2 is $x + y$. Equivalence's counterpart in arithmetic mod 2 is $x + y + 1$.

Given two operands, each with two possible values, there are $2^2 = 4$ possible combinations of inputs. Because each output can have two possible values, there are a total of $2^4 = 16$ possible binary Boolean operations. Any such operation or function (as well as any Boolean function with more inputs) can be expressed with the basic operations from above. Hence the basic operations are functionally complete.

Boolean algebra satisfies many of the same laws as ordinary algebra when one matches up ∨ with addition and ∧ with multiplication. In particular the following laws are common to both kinds of algebra

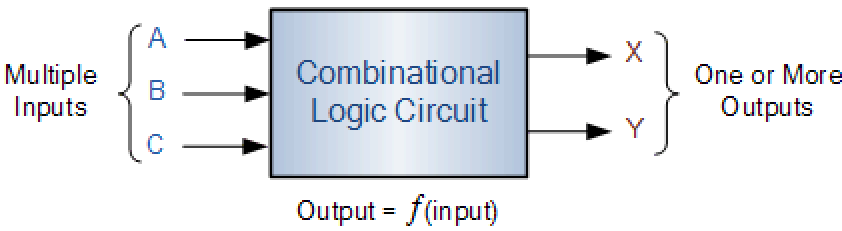
Associativity of ∨:	$x \vee (y \vee z) = (x \vee y) \vee z$
Associativity of ∧:	$x \wedge (y \wedge z) = (x \wedge y) \wedge z$
Commutativity of ∨:	$x \vee y = y \vee x$
Commutativity of ∧:	$x \wedge y = y \wedge x$
Distributivity of ∧ over ∨:	$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$
Identity for ∨:	$x \vee 0 = x$
Identity for ∧:	$x \wedge 1 = x$
Annihilator for ∧:	$x \wedge 0 = 0$

The following laws hold in Boolean Algebra, but not in ordinary algebra:

Annihilator for ∨:	$x \vee 1 = 1$
Idempotence of ∨:	$x \vee x = x$
Idempotence of ∧:	$x \wedge x = x$
Absorption 1:	$x \wedge (x \vee y) = x$
Absorption 2:	$x \vee (x \wedge y) = x$
Distributivity of ∨ over ∧:	$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$

Combinational Logic Circuits

Combinational Logic Circuits are memoryless digital logic circuits whose output at any instant in time depends only on the combination of its inputs



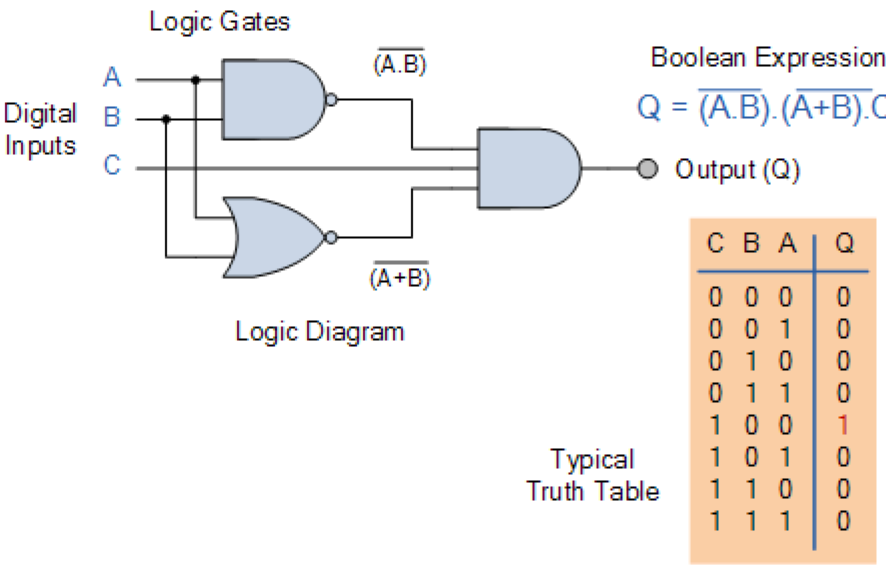
Combinational Logic Circuits are made up from basic logic NAND, NOR or NOT gates that are "combined" or connected together to produce more complicated switching circuits. These logic gates are

the building blocks of combinational logic circuits. An example of a combinational circuit is a decoder, which converts the binary code data present at its input into a number of different output lines, one at a time producing an equivalent decimal code at its output.

Combinational logic circuits can be very simple or very complicated and any combinational circuit can be implemented with only NAND and NOR gates as these are classed as “universal” gates.

The three main ways of specifying the function of a combinational logic circuit are:

- 1. Boolean Algebra – This forms the algebraic expression showing the operation of the logic circuit for each input variable either True or False that results in a logic “1” output.
- 2. Truth Table – A truth table defines the function of a logic gate by providing a concise list that shows all the output states in tabular form for each possible combination of input variable that the gate could encounter.
- 3. Logic Diagram – This is a graphical representation of a logic circuit that shows the wiring and connections of each individual logic gate, represented by a specific graphical symbol, that implements the logic circuit.

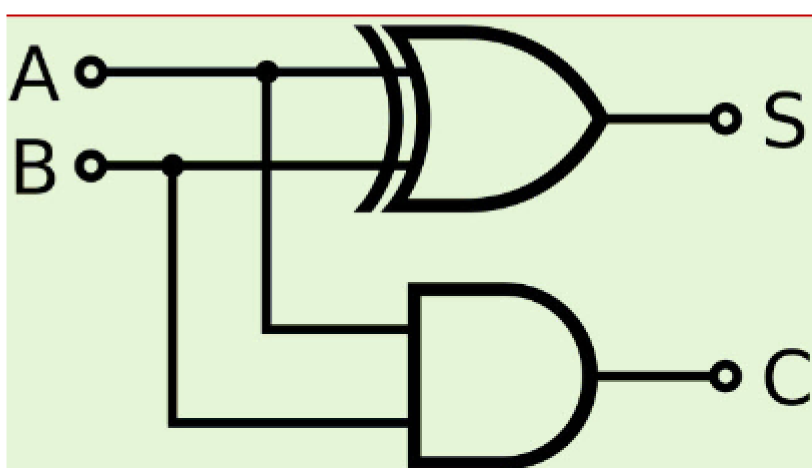


C	B	A	Q
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Half Adder Truth Table

INPUTS		OUTPUTS	
A	B	SUM	CARRY
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Now it has been cleared that 1-bit adder can be easily implemented with the help of the XOR Gate for the output 'SUM' and an AND Gate for the 'Carry'. When we need to add, two 8-bit bytes together, we can be done with the help of a full-adder logic. The half-adder is useful when you want to add one binary digit quantities. A way to develop a two-binary digit adders would be to make a truth table and reduce it. When you want to make a three binary digit adder, do it again. When you decide to make a four digit adder, do it again. The circuits would be fast, but development time is slow.



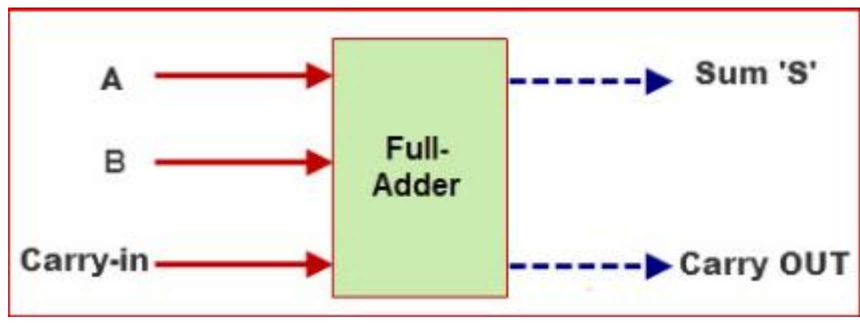
VHDL Code For half Adder

```
entity ha is
Port (a: in  STD_LOGIC;
b : in  STD_LOGIC;
sha : out  STD_LOGIC;
cha : out  STD_LOGIC);
end ha;


architecture Behavioral of ha is

begin
sha <= a xor b ;
cha <= a and b ;
end Behavioral
```

Full Adder



This adder is difficult to implement than a half-adder. The difference between a half-adder and a full-adder is that the full-adder has three inputs and two outputs, whereas half adder has only two inputs and two outputs. The first two inputs are A and B and the third input is an input carry as C-IN. When a full-adder logic is designed, you string eight of them together to create a byte-wide adder and cascade the carry bit from one adder to the next.

INPUTS			OUTPUT	
A	B	C-IN	C-OUT	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1