Digital circuits – a quick recapitulation

In computers, all numbers are represented with the help of just two symbols, 0 and 1. However, using only two symbols is not the usual practice. We generally use the decimal system and that has ten symbols. More the number of symbols, less is the size of a represented quantity. For example, the number 9 requires only one symbol in decimal system while it requires 4 symbols (1001) in the decimal system.

Naturally the question arises, why should we restrict ourselves to just two symbols for representing the numbers in case of a computer ?
The answer is that "the more the symbols employed for number representation, more is the complexity of the system involved". If there are just two symbols to be represented, it could well be done by the presence and absence of a voltage. Presence of a voltage in a wire would be one symbol while absence of it would be the other. It makes a system extremely simple.

This pair of symbols is sometimes referred to as 1 and 0, true and false, Yes and No, High and low (Hi and Lo), 5 volts and 0 volts etc. One day, the phrase 'Me and You' wouldn't be so romantic because it might also indicate the two states of some very prosaic physical quantity !

Thus, any variable in such a system can take up only two values. Hence, binary logical operations are applicable in such cases. Logical operations are basically conditional relations between an output and a number of inputs. Ultimately, mathematical operations can be expressed through logic operations.

In order to explain the above, let us take the case of a few logical conditions. Let us say that a bulb would glow if **A** *and* **B** (two friends) both attend a party. If **A** *or* **B** went alone, the bulb would not glow. And of course, if both missed the party, the bulb would remain off. Now, the glowing of the bulb could be pictured as a two state (binary) variable whose value conditionally depends on the presence or absence of **A** and **B**. If we represent the presence and absence of **A** and **B** by the variables *A* and *B* respectively, they also become binary variables. Hence, with the following conventions :

A present $\rightarrow A = 1$, **A** absent $\rightarrow A = 0$ ($\rightarrow$ is represented by)

B present $\rightarrow B = 1$, **B** absent $\rightarrow B = 0$

bulb on $\rightarrow C = 1$, bulb off $\rightarrow C = 0$

We could say that *C* would be 1 if and only if *A and B* are both 1. This logical relation between *A*, *B* and *C* is written as

$C = A \,.\, B$

The reader should be cautioned that this '**.**' sign here does not represent the arithmetic multiplication but the logical 'and' operation.

A	*B*	*C = A . B*
1	1	1
1	0	0
0	1	0
0	0	0

Table 3.1 Truth table for *A.B*

If all the possible values of a binary output variable are shown for all possible combinations of the input variables in tabular form, it is referred to as a *Truth table*. In table 3.1, such a truth table has been shown for *A.B.*

In a similar fashion, *another* logical relation which might exist between *A,B* and *C* can be expressed in language as :

C would glow if A, or B, or both attend the party.

This is a very lenient condition and in the truth table, we would observe that C becomes 1 in three out of the four cases.

This is referred to as the logical 'or' operation and is symbolically written as

$A + B = C$

A	*B*	*C = A + B*
1	1	1
1	0	1
0	1	1
0	0	0

Table 3.2 Truth table for $A + B$

Once again, it should be mentioned that this '+' here is not the mathematical addition but the logical OR operation.

Just like these two logical operations, there are others as well which are being listed below

The NOT operation : This operation involves a single variable (unary operation).
A binary variable can take up only two values – say, 1 and 0. These values (1 and 0) are sometimes referred to as opposite or compliment of one another. The result of the NOT operation on a binary variable is its compliment.

For example, if $A = 1$, NOT(A) = A' = Opposite of 1 = 0

The XOR operation

A	B	$A \oplus B$
1	1	0
1	0	1
0	1	1
0	0	0

There can also be logical operations which involve two or more such binary operations like :
NOT + AND = NAND = $(A.B)'$, NOT + OR = NOR = $(A + B)'$

A	B	A'	B'	$(A.B)'$	$(A+B)'$
1	1	0	0	0	0
1	0	0	1	1	0
0	1	1	0	1	0
0	0	1	1	1	1

Table 3.3 Truth table for NOT, NAND and NOR operations

All these relations can be realized with the help of electronic circuits and they are symbolically represented as follows :

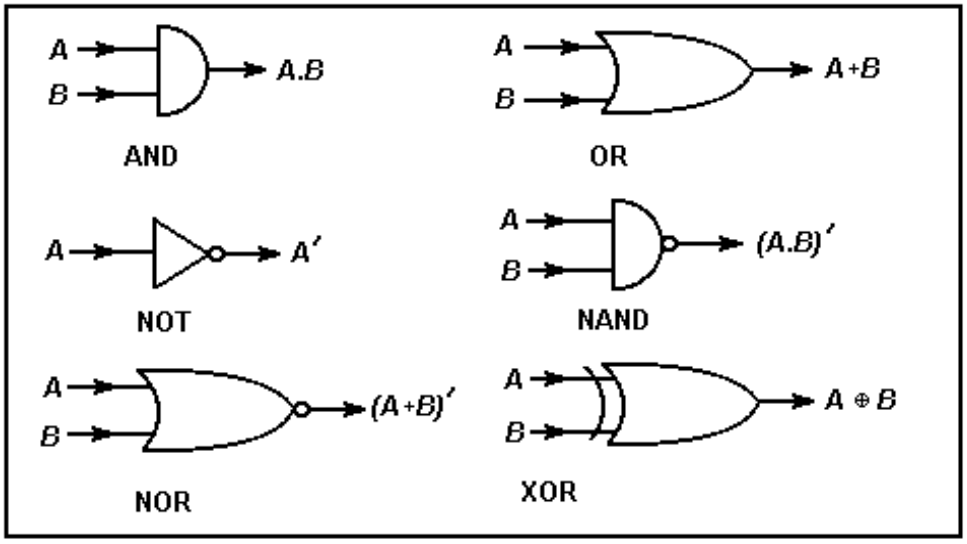

Fig 3.1 Logic Gate Symbols

Digital circuitry can execute logic operations when dealing with voltage signals. As discussed before, a binary variable could be represented by the absence or presence of voltage along a particular cable. Thus, in fig 3.1, *A* is represented by the voltage in the wire corresponding to the symbol *A*. If there happens to be 0 voltage at that wire, *A* is taken to be 0 (i.e., we are adopting the convention here that 0 volts stands for a binary value of 0) and if there is, say, 5 volts available at the wire, *A* is taken to 1.

A number of interesting exercises in binary logic operations could be carried out to gain command over boolean algebra (Boolean algebra is the mathematics involved with the algebra of binary variables). For example, let us suppose that we only have NAND gates at our disposal and with that building block we have to represent all other gates. For example, A NOT gate could be represented in the following manner (fig 3.2) :

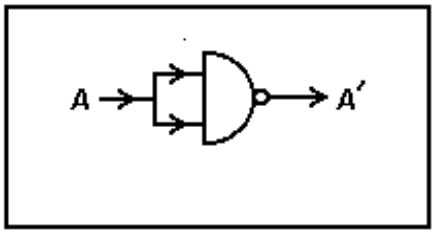

fig 3.2

Such representations for all other common gates (AND, OR, XOR) are given at the end of this chapter. The reader could try out the exercises and compare the results.

It would be relevant to remember the laws of boolean algebra at this juncture. They are provided below :

- law of complements

$$a + a' = 1$$
$$a \,.\, a' = 0$$
$$a + 1 = 1$$
$$a \,.\, 0 = 0$$
$$a \,.\, 1 = a$$
$$a + 0 = a$$

- Commutative law

$$a + b = b + a$$
$$a.b = b.a$$

- Distributive law

$$a.(b + c) = a.b + a.c$$

- associative law

$$a + b.c = (a + b) \,.\, (a + c)$$

- De Morgan's laws

$$(a + b)' = a' \,.\, b'$$
$$(a \,.\, b)' = a' + b'$$

All these operations are logic operations. However, in the CNC machines, apart from logic operations, arithmetic operations like addition, subtraction etc have to be performed as well. All these are performed with the help of logic circuits. This is made possible by simply building circuits which produce the same results as those of addition, subtraction etc. A simple example would make the concept very clear.

Let us suppose that we have to add two numbers which are one bit (bit = binary digit) each in length. Hence, if we add them up, the result would yield a *sum* in the unit's place and a *carry* in the second's place (called the tenth's place in decimal system). The addition would be something like :

a	*b*	*carry*	*sum*
1	1	1	0
1	0	0	1
0	1	0	1
0	0	0	0

Table 3.4 truth table for addition of two bits

A quick look at the values of sum and carry will show us that carry is the same as a . b (a AND b) while sum is nothing but a + b (a XOR b)note – the sign of XOR!!!. Hence, we could carry out the arithmetic addition over two bits with the following logical circuit.

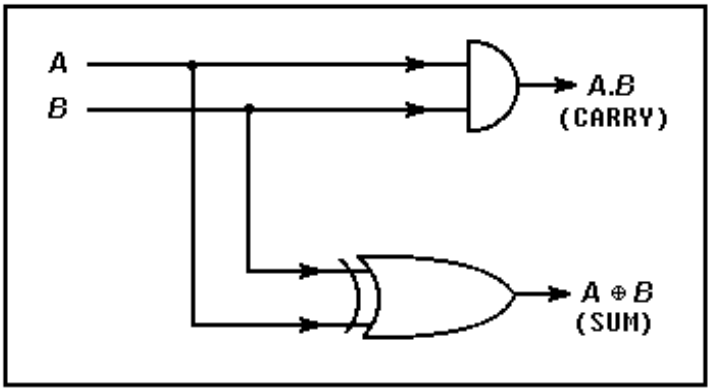

fig 3.3

However, arithmetic addition of two binary *numbers* will not always be restricted to two-bit additions. When we are adding two *numbers* instead of two *bits*, we could possibly have the addition of **three bits**, as shown in the example.

Carry 1011111
11011011
+ 11001111
110101010

Thus, in all the places except the unit's place, we could have the addition of three bits. This requires the use of a logic circuit where we can add three binary digits. Keeping this in mind, let us draw up all the possibilities of such an addition.

a	*b*	*c*	*carry*	*sum*
1	1	1	1	1
1	1	0	1	0
1	0	1	1	0

1	0	0	0	1
0	1	1	1	0
0	1	0	0	1
0	0	1	0	1
0	0	0	0	0

While the logical simulation of single bit addition (i.e., representation of sum and carry by and gate and xor gate) was quite straightforward, the arithmetic addition of three bits cannot be so easily represented. Here, we could employ a method of representation where the arithmetic operation is first transformed into a logic statement and then represented by logic operations.

From the truth table of A, B, C, sum and carry, it can be observed that sum and carry become 1 for certain combinations of A, B and C. Hence, we could say that the sum takes up a value of 1 if and only if,

	A and B and C are all 1
Or	A is 1 and B is 0 and C is 0
Or	A is 0 and B is 1 and C is 0
Or	A is 0 and B is 0 and C is 1

The first conditional statement could be realized by passing A and B and C through an AND gate. Thus, only when A and B and C are all 1, the result would be a 1. This is how the first condition is realized.

However, this is not the only case, there are other conditions also for which the sum assumes a value of 1. If we consider the second of such conditions,

When A is 1 and B is 0 and C is 0 …

If we can construct a circuit which would yield a 1 for this combination only, it would be a physical representation of the condition.

The AND gate, that way, is a very ‘choosy’ or ‘selective’ gate. It yields a result of 1 only for one condition, i.e., when all inputs are 1. The OR gate is not that choosy or strict. It yields a value of 1 for more than one input condition. Hence, the AND gate is an obvious choice for building a representative circuit for a particular combination of inputs with output 1. Now, for the present case under consideration, the inputs A, B and C are not all 1. Hence, a mere feeding of A, B and C to an AND gate would not yield a result of 1. However, our target is well defined, and that is, to find a combination of inputs to an AND gate which would yield a 1 when A = 1, B = 0 and C = 0. A quick inspection of the inputs would reveal that if we invert B and C before feeding them to the AND gate, our problem would be solved. Hence, a circuit of $A.B^{/}.C^{/}$ would yield a result of 1 only if A = 1, B = 0 and C = 0 and be 0 in all other cases.

Following this procedure, we can formulate the full logic circuit which would be able to add three bits.

$Sum = A.B.C + A.B'.C' + A'.B.C' + A'.B'.C$

And

$Carry = A.B.C + A.B.C' + A'.B.C + A.B'.C$

Thus we are now in possession of a powerful method of constructing representative circuits for arithmetic operations. The above expressions can further be simplified by boolean manipulation, as follows,

$Sum = A.B.C + A.B'.C' + A'.B.C' + A'.B'.C = A\,.\,(B.C + B'.C') + A'.\,(B.C' + B'.C)$

Now,

$(B.C + B'.C')' = (B.C)'\,.\,(B'.C')' = (B' + C')\,.\,(B + C) = B.B' + B.C' + B'.C + C.C'$

$= B.C' + B'.C$

Hence,

$A\,.\,(B.C + B'.C') + A'.\,(B.C' + B'.C) = A.(B \oplus C)' + A'\,.\,(B \oplus C) = A \oplus B \oplus C$

In a similar manner,

$Carry = A.B.C + A.B.C' + A'.B.C + A.B'.C = A.B.C + A.B.C' + A.B.C + A'.B.C + A.B.C + A.B'.C$

$= A.B.(C+C') + B.C.(A+A') + A.C.(B+B') = A.B + B.C + A.C$

(in Boolean algebra, a variable can be added any number of times to another without any difference, hence a single addition of A.B.C has been replaced by three additions of the same for ease of algebraic manipulation)