A. LOGIC

George Boole, a ninteenth century British mathematician, made a detailed study of the relationship between certain fundamental logical expressions and their arithmetic counterparts. Boole did not equate mathematics with logic, but he did show how any logical statement can be analyzed with simple arithmetic relationships. In 1847, Boole published a booklet entitled Mathematical Analysis of Logic and in 1854 he published a much more detailed work on the subject. To this day, all practical digital computers and many other electronic circuits are based upon the logic concepts explained by Boole.

Boole's system of logic, which is frequently called Boolean algebra, assumes that a logic condition or statement is either true or false. It cannot be both true and false, and it cannot be partially true or partially false. Fortunately, electronic circuits are admirably suited for this type of dual-state operation. If a circuit in the ON state is said to be true and a circuit in the OFF state is said to be false, an electronic analogy of a logical statement can be readily synthesized.

With this in mind, it is possible to devise electronic equivalents for the three basic logic statements: AND, OR and NOT. The AND statement is true if and only if either or all of its logic conditions are true. A NOT statement merely reverses the meaning of a logic statement so that a true statement is false and a false statement is true.

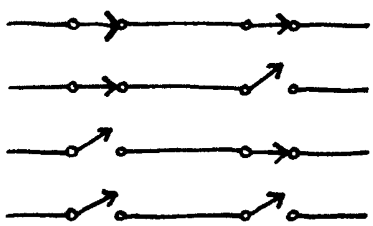
It's easy to generate a simple equivalent of these three logic statements by using on-off switches. A switch which is ON is said to be true while a switch which is OFF is said to be false. Since a switch which is OFF will not pass an electrical current, it can be assigned a numerical value of 0. Similarly, a switch which is ON does pass an electrical current and can be assigned a numerical value of 1.

We can now devise an electronic equivalent of the logical AND statement by examining the various permutations for a two condition AND statement:

CONDITIONS	CONCLUSION
(Inputs)	(Output)
1. True AND True	True
2. True AND False	False
3. False AND True	False
4. False AND False	False

The electronic ON-OFF switch equivalent of these permutations is simply:

CONDITIONS	CONCLUSION
(ON-OFF)	(OUTPUT)
1.	1
2.	0
3.	0
4.	0



Similarly, the numerical equivalents of these permutations is:

CONDITIONS	CONCLUSION
(Inputs)	(Output)
1. 1 AND 1	1
2. 1 AND 0	0
3. 0 AND 1	0
4. 0 AND 0	0

Digital design engineers refer to these table of permutations as truth tables. The truth table for the AND statement with two conditions is usually presented thusly:

A	B	OUT
1	1	1
0	1	0
1	0	0
0	0	0

FIGURE 1-1. AND Function Truth Table

It is now possible to derive the truth tables for the OR and NOT statements, and each is shown in Figures 1-2 and 1-3 respectively.

A	B	OUT
1	1	1
0	1	1
1	0	1
0	0	0

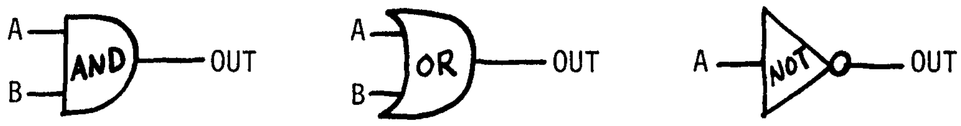
FIGURE 1-2. OR Function Truth Table

A	OUT
1	0
0	1

FIGURE 1-3. NOT Function Truth Table

B. ELECTRONIC LOGIC

All three of the basic logic functions can be implemented by relatively simple transistor circuits. By convention, each circuit has been assigned a symbol to assist in designing logic systems. The three symbols along with their respective truth tables are shown in Figure 1-4.



A	B	OUT
0	0	0
0	1	0
1	0	0
1	1	1

A	B	OUT
0	0	0
0	1	1
1	0	1
1	1	1

A	OUT
0	1
1	0

FIGURE 1-4. The Three Main Logic Symbols

The three basic logic circuits can be combined with one another to produce still more logic statement analogies. Two of these circuit combinations are used so frequently that they are considered basic logic circuits and have been assigned their own logic symbols and truth tables. These circuits are the NAND (NOT-AND) and the NOR (NOT-OR). Figure 1-5 shows the logic symbols and truth tables for these circuits.

A	B	OUT
0	0	1
0	1	1
1	0	1
1	1	0

A	B	OUT
0	0	1
0	1	0
1	0	0
1	1	0

FIGURE 1-5. The NAND and NOR Circuits

Three or more logic circuits make a logic system. One of the most basic logic systems is the EXCLUSIVE-OR circuit shown in Figure 1-6.

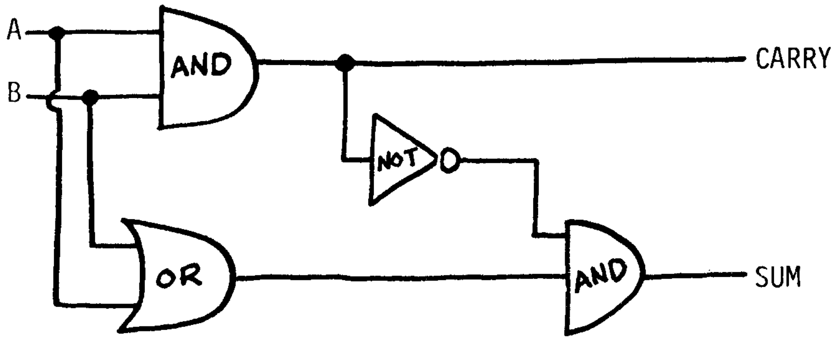


FIGURE 1-6. The EXCLUSIVE-OR Circuit

The EXCLUSIVE-OR circuit can be used to implement logical functions, but it can also be used to add two input conditions. Since electronic logic circuits utilize only two numerical units, 0 and 1, they are compatible with the binary number system, a number system which has only two digits. For this reason, the EXCLUSIVE-OR circuit is often called a binary adder.

Various combinations of logic circuits can be used to implement numerous electronic functions. For example, two NAND circuits can be connected to form a bistable circuit called a flip-flop. Since the flip-flop changes state only when an incoming signal in the form of a pulse arrives, it acts as a short term memory element. Several flip-flops can be cascaded together to form electronic counters and memory registers.

Other logic circuits can be connected together to form monostable and astable circuits. Monostable circuits occupy one of two states unless an incoming pulse is received. They then occupy an opposite state for a brief time and then resume their normal state. Astable circuits continually switch back and forth between two states.

C. NUMBER SYSTEMS

Probably because he found it convenient to count with his fingers, early man devised a number system which consisted of ten digits. Number systems, however, can be based on any number of digits. As we have already seen, dual-state electronic circuits are highly compatible with a two digit number system, and its digits are termed bits (binary digits). Systems based upon eight and sixteen are also compatible with complex electronic logic systems such as computers since they provide a convenient shorthand method for expressing lengthy binary numbers.

D. THE BINARY SYSTEM

Like virtually all digital computers, the *ALTAIR 8800* performs nearly all operations in binary. A typical binary number processed by the computer incorporates 8-bits and may appear as: 10111010. A fixed length binary number such as this is usually called a word or byte, and computers are usually designed to process and store a fixed number of words (or bytes).

A binary word like 10111010 appears totally meaningless to the novice. But since binary utilizes only two digits (bits), it is actually much simpler than the familiar and traditional decimal system. To see why, let's derive the binary equivalents for the decimal numbers from 0 to 20. We will do this by simply adding 1 to each successive number until all the numbers have been derived. Counting in any number system is governed by one basic rule: Record successive digits for each count in a column. When the total number of available digits has been used, begin a new column to the left of the first and resume counting.

Counting from 0 to 20 in binary is very easy since there are only two digits (bits). The binary equivalent of the decimal 0 is 0. Similarly, the binary equivalent of the decimal 1 is 1. Since both available bits have now been used, the binary count must incorporate a new column to form the binary equivalent for the decimal 2. The result is 10. (Incidentally, ignore any resemblance between binary and decimal numbers. Binary 10 is not decimal 10!) The binary equivalent of the decimal number 3 is 11. Both bits have been used again, so a third column must be started to obtain the binary equivalent for the decimal number 4 (100). You should now be able to continue counting and derive all the remaining binary equivalents for the decimal numbers 0 to 20:

DECIMAL	BINARY
0	0
1	1
2	10
3	11

DECIMAL	BINARY
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111
16	10000
17	10001
18	10010
19	10011
20	10100

A simple procedure can be used to convert a binary number into its decimal equivalent. Each bit in a binary number indicates by which power of two the number is to be raised. The sum of the powers of two gives the decimal equivalent for the number. For example, consider the binary number 10011:

$$10011 = [(1\text{x}2^4) + (0\text{x}2^3) + (0\text{x}2^2) + (1\text{x}2^1) + (1\text{x}2^0)]$$

$$= [(16) + (0) + (0) + (2) + (1)]$$

$$= 19$$

```
Microsoft MS-DOS version 2.11
Copyright 1981,82,83 Microsoft Corp.
Rev. 1.01 6/29/84

Command v. 2.11

A>
A>type twoscomp.txt

Two's complement is a clever way of storing integers so that common math p
-roblems are very simple to implement.

To understand, you have to think of the numbers in binary.

It basically says,

  @  for zero, use all 0's.
  @  for positive integers, start counting up, with a maximum of 2^(number
     of bits - 1)-1.
  @  for negative integers, do exactly the same thing, but switch the role
     of 0's and 1's (so instead of starting with 0000, start with 1111 - t
     -hat's the "complement" part).

Let's try it with a mini-byte of 4 bits (we'll call it a nibble - 1/2 a by
-te).

  @  0000 - zero
  @  0001 - one
  @  0010 - two
  @  0011 - three
  @  0100 to 0111 - four to seven

That's as far as we can go in positives. 2^3-1 = 7.

For negatives:

  @  1111 - negative one
  @  1110 - negative two
  @  1101 - negative three
  @  1100 to 1000 - negative four to negative eight

Note that you get one extra value for negatives (1000 = -8) that you don’t
for positives. This is because 0000 is used for zero. This can be consider
-ed as Number Line of computers.

Distinguishing between positive and negative numbers...

Doing this, the first bit gets the role of the "sign" bit, as it can be us
-ed to distinguish between positive and negative decimal values. If the mo
-st significant bit is 1, then the binary can be said to be negative, wher
-e as if the most significant bit (the leftmost) is 0, you can say discern
the decimal value is positive.

EOF.

A>format a:
Insert new diskette for drive A: and strike any key when ready

Formatting...Format Complete

   362496 bytes total disk space
   362496 bytes available on disk

Format another (Y/N)?n
A>dir

 Volume in drive A has no label
 Directory of  A:\

File not found

A>
```