13. Lecture Notes

Unit-I

BINARYSYSTEMS

Philosophy of Number Systems

Numbering System:

Many number systems are in use in digital technology. The most common are the decimal, binary, octal, and hexadecimal systems. The decimal system is clearly the most familiar to us because it is a tool that we use every day. Examining some of its characteristics will help us to better understand the other systems. In the next few pages we shall introduce four numerical representation systems that are used in the digital system. There are other systems, which we will look at briefly.

- Decimal
- Binary
- Octal
- Hexadecimal

Decimal System:

The decimal system is composed of 10 numerals or symbols. These 10 symbols are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Using these symbols as digits of a number, we can express any quantity. The decimal system is also called the base-10 system because it has 10digits.

10^3	10^2	10^1	10^0		10^{-1}	10^{-2}	10^{-3}
=1000	=100	=10	=1	.	=0.1	=0.01	=0.001
Most Significant Digit				Decimal point			Least Significant Digit

Even though the decimal system has only 10 symbols, any number of any magnitude can be expressed by using our system of positional weighting.

Decimal Examples

- 3.14_{10}

- 52_{10}
- 1024_{10}
- 64000_{10}

Binary System

In the binary system, there are only two symbols or possible digit values, 0 and 1. This base-2 system can be used to represent any quantity that can be represented in decimal or other base system.

2^3	2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}
=8	=4	=2	=1	.	=0.5	=0.25	=0.125
Most Significant Digit				Binary point			Least Significant Digit

Binary Counting

The Binary counting sequence is shown in the table:

2^3	2^2	2^1	2^0	Decimal
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	10
1	0	1	1	11
1	1	0	0	12

1	1	0	1	13
1	1	1	0	14
1	1	1	1	15

In additional to binary and decimal, two other number systems find wide-spread applications in digital systems. The octal (base-8) and hexadecimal (base-16) number systems are both used for the same purpose- to provide an efficient means for representing large binary system.

Octal System

The octal number system has a base of eight, meaning that it has eight possible digits: 0, 1,2,3,4,5,6,7.

8^3	8^2	8^1	8^0		8^{-1}	8^{-2}	8^{-3}
=512	=64	=8	=1	.	=1/8	=1/64	=1/512
Most Significant Digit				Octal point			Least Significant Digit

Octal to Decimal Conversion

- $237_8 = 2 \times (8^2) + 3 \times (8^1) + 7 \times (8^0) = 159_{10}$
- $24.6_8 = 2 \times (8^1) + 4 \times (8^0) + 6 \times (8^{-1}) = 20.75_{10}$
- $11.1_8 = 1 \times (8^1) + 1 \times (8^0) + 1 \times (8^{-1}) = 9.125_{10}$
- $12.3_8 = 1 \times (8^1) + 2 \times (8^0) + 3 \times (8^{-1}) = 10.375_{10}$

Hexadecimal System

The hexadecimal system uses base 16. Thus, it has 16 possible digit symbols. It uses the digits 0 through 9 plus the letters A, B, C, D, E, and F as the 16 digitsymbols.

16^3	16^2	16^1	16^0			16^{-1}	16^{-2}	16^{-3}
=4096	=256	=16	=1	.		=1/16	=1/256	=1/4096
Most Significant Digit				Hexa point	Decimal			Least Significant Digit

Hexadecimal to Decimal Conversion

- $24.6_{16} = 2 \times (16^1) + 4 \times (16^0) + 6 \times (16^{-1}) = 36.375_{10}$
- $11.1_{16} = 1 \times (16^1) + 1 \times (16^0) + 1 \times (16^{-1}) = 17.0625_{10}$
- $12.3_{16} = 1 \times (16^1) + 2 \times (16^0) + 3 \times (16^{-1}) = 18.1875_{10}$

Code Conversion

Converting from one code form to another code form is called code conversion, like converting from binary to decimal or converting from hexadecimal to decimal.

Binary-To-Decimal Conversion

Any binary number can be converted to its decimal equivalent simply by summing together the weights of the various positions in the binary number which contain a1.

Binary	Decimal
11011_2	
$2^4+2^3+0^1+2^1+2^0$	$=16+8+0+2+1$
Result	27_{10}

and

Binary	Decimal
10110101_2	
$2^7+0^6+2^5+2^4+0^3+2^2+0^1+2^0$	$=128+0+32+16+0+4+0+1$
Result	181_{10}

You should have noticed that the method is to find the weights (i.e., powers of 2) for each bit position that contains a 1, and then to add them up.

Decimal-To-Binary Conversion

There are 2methods:

- Reverse of Binary-To-Decimal Method
- Repeat Division

Reverse of Binary-To-Decimal Method

Decimal	Binary
45_{10}	$=32 + 0 + 8 + 4 + 0 + 1$
	$=2^5+0+2^3+2^2+0+2^0$
Result	$=101101_2$

Repeat Division-Convert decimal to binary

This method uses repeated division by2. Convert

25_{10} to binary

Division	Remainder	Binary
25/2	= 12+ remainder of1	1 (Least Significant Bit)
12/2	= 6 + remainder of0	0
6/2	= 3 + remainder of0	0
3/2	= 1 + remainder of1	1
½	= 0 + remainder of1	1 (Most Significant Bit)
Result	25_{10}	$=11001_2$

The Flow chart for repeated-division method is as follows:

Binary-To-Octal / Octal-To-Binary Conversion

OctalDigit	0	1	2	3	4	5	6	7
Binary Equivalent	000	001	010	011	100	101	110	111

Each Octal digit is represented by three binary digits.

Example:

$100\ 111\ 010_2 = (100)\ (111)\ (010)_2 = 4\ 72_8$

Repeat Division-Convert decimal to octal

This method uses repeated division by8.

Example: Convert 177_{10} to octal and binary

Division	Result	Binary
177/8	= 22+ remainder of1	1 (Least Significant Bit)
22/8	= 2 + remainder of6	6
2 /8	= 0 + remainder of2	2 (Most Significant Bit)
Result	177_{10}	$=261_8$
Binary		$=010110001_2$

Hexadecimal to Decimal/Decimal to Hexadecimal Conversion

Example: $2AF_{16} = 2 \times (16^2) + 10 \times (16^1) + 15 \times (16^0) = 687_{10}$

Repeat Division- Convert decimal tohexadecimal

This method uses repeated division by16.

Example: convert 378_{10} to hexadecimal and binary:

Division	Result	Hexadecimal
378/16	= 23+ remainder of10	A (Least Significant Bit)23
23/16	= 1 + remainder of7	7
1/16	= 0 + remainder of1	1 (Most Significant Bit)
Result	378_{10}	$=17A_{16}$
Binary		$= 0001\ 01111010_2$

Binary-To-Hexadecimal /Hexadecimal-To-Binary Conversion

Hexadecimal Digit	0	1	2	3	4	5	6	7
Binary Equivalent	0000	0001	0010	0011	0100	0101	0110	0111

Hexadecimal Digit	8	9	A	B	C	D	E	F
Binary Equivalent	1000	1001	1010	1011	1100	1101	1110	1111

Each Hexadecimal digit is represented by **four** bits of binary digit.

Example:

$1011\ 0010\ 1111_2 = (1011)\ (0010)\ (1111)_2 = B\ 2F_1$

Octal-To-Hexadecimal Hexadecimal-To-Octal Conversion

- Convert Octal (Hexadecimal) to Binary first.
- Regroup the binary number by three bits per group **starting from LSB** if Octal is required.
- Regroup the binary number by four bits per group starting from LSB if Hexadecimal is required.
- Example: Convert $5A8_1$
-
-

.

Hexadecimal	Binary/Octal
5A816	= **0101** 1010 **1000**(Binary)
	= **010** 110 **101** 000(Binary)
Result	= 2 6 5 0(Octal)

Complement representation of negative numbers

Signed-Magnitude representation**:**

- This is the simplest method.
- Write the magnitude of the number in binary. Then add a 1 to the front of it if the number is negative and a 0 if it is positive.
  Examples: +7 would be 111 and then a 0 in front so 00000111 for an 8-bit representation.
  -9 would be 1001 (+9) and then a 1 so 10001001 for an 8-bitrepresentation.
- It is not the best method or representation because it makes computation awkward.

2's complement representation:

- Most widely used method of representation.
- Positive numbers are represented as they are(simple binary).
- To get a negative number, write the positive number in binary, then change all0's to 1's and 1's to 0's. Then add 1 to the number.
- Example:+7wouldbe0111in4-bit2'scomplement.
  To represent –5 we take +5 (0101) and then invert the digits (1010) and add1 (1011). –5 is thus1011.
- Suppose you already have a number that is in two's complement representation and want to find its value in binary.

If the number starts with a 1 it is a negative number. If it starts with a 0 it is a positive number. If it is a negative number, take the 2's complement of that number. You will get the number in ordinary binary. The sign you already know. Let's take1101.
Take the 2's complement and you get 0011. Since it started with a 1, it was negative and the value is 0011 which is 3. The number represented by 1101 is –3 in 2'scomplement.
Let's see how this system is better:
If we add +5 and -5 in decimal we get0.
Let's add them in 4-bit signed-magnitude. +5 is 0101 and –5 is 1101. On adding we get 10010. That is nonzero.
Let's do the same thing in 2's complement. Adding 0101 (+5) and 1011 (-5) gives 10000. If we discard the carry of 1 we get 0000 – i.e.0.
Thus addition works out ok for negative numbers in 2's complement whereas it doesn't in sign magnitude.
Similarly, you can show that multiplication and subtraction all work in 2's complement but do not in other representations. The other number systems require much more complicated hardware to implement basic mathematical functions. i.e. add/subtract/multiply.

Sign-and-magnitude

8 bit signed magnitude

Binary	Signed	Unsigned
00000000	+0	0
00000001	1	1
...	...	...
01111111	127	127
10000000	−0	128
10000001	−1	129
...	...	...
11111111	−127	255

One may first approach the problem of representing a number's sign by allocating one sign bit to represent the sign: set that bit (often the most significant bit) to 0 for a positive number, and set to 1 for a negative number. The remaining bits in the number indicate the magnitude (or absolute value). Hence in a byte with only 7 bits (apart from the sign bit), the magnitude can range from 0000000 (0) to 1111111 (127). Thus you can represent numbers from −12710 to +12710 once you add the sign bit (the eighth bit). A consequence of this representation is that there are two ways to represent zero, 00000000 (0) and 10000000 (−0). Decimal −43 encoded in an eight-bit byte this way is0101011.

This approach is directly comparable to the common way of showing a sign (placing a "+" or "−" next to the number's magnitude). Some early binary computers (e.g. IBM7090) used this representation, perhaps because of its natural relation to common
usage. Sign-and-magnitude is the most common way of representing the significant and in floating point values.

8 bit ones'complement

Binary value	Ones' complement interpretation	Unsigned interpretation
00000000	+0	0
00000001	1	1
...	...	...
01111101	125	125
01111110	126	126
01111111	127	127
10000000	−127	128
10000001	−126	129
10000010	−125	130
...	...	...
11111110	**−1**	**254**
11111111	**−0**	**255**

Alternatively, a system known as ones' complement can be used to represent negative numbers. The ones' complement form of a negative binary number is the bitwise NOT applied to it — the complement of its positive counterpart. Like sign-and-magnitude representation, ones' complement has two representations of 0: 00000000 (+0) and 11111111(−0).

As an example, the ones' complement form of 00101011 (43) becomes 11010100 (−43). The range of signed numbers using ones' complement is represented by $-(2^{N-1}-1)$ to $(2^{N-1}-1)$ and +/−0. A conventional eight-bit byte is -127_{10} to $+127_{10}$ with zero being either 00000000 (+0) or 11111111(−0).

To add two numbers represented in this system, one does a conventional binary addition, but it is then necessary to add any resulting carry back into the resulting sum. To see why this is necessary, consider the following example showing the case of the addition of −1 (11111110) to +2(00000010).

```
    '"binary    decimal"' 11111110
                      -1
    +   00000010   +2
    .............    ...
     1   00000000    0   <-- not the correct answer
           1     +1   <-- add carry
    .............    ...
       00000001      1   <-- correct answer
```

In the previous example, the binary addition alone gives 00000000, which is incorrect. Only when the carry is added back in does the correct result (00000001)appear.

This numeric representation system was common in older computers; the PDP-1, CDC 160A and UNIVAC 1100/2200 series, among many others, used ones'-complement arithmetic.

A remark on orthography: The system is referred to as "ones' complement" because the negation of a positive value x (represented as the bitwise NOT of x) can also be formed by subtracting x from the ones' complement representation of zero that is a long sequence of ones (-0). Two's complement arithmetic, on the other hand, forms the negation of x by subtracting x from a single large power of two that is congruent to
+0.[1] Therefore, ones' complement and two's complement representations of the same negative value will differ by one.

The Internet protocols IPv4, ICMP, UDP and TCP all use the same 16-bit ones' complement checksum algorithm. Although most computers lack "end-around carry" hardware, the extra complexity is accepted because "it is equally sensitive to errors in all bit positions".[2] In UDP, the all 0s representation of zero indicates that the optional checksum feature has been omitted. The other representation, FFFF, indicates a checksum value of 0.[3] (Checksums are mandatory in IPv4, TCP and ICMP; they were omitted fromIPv6).

Note that the ones' complement representation of a negative number can be obtained from the sign-magnitude representation merely by bitwise complementing the magnitude.

Binary Arithmetic: Binary addition

Adding binary numbers is a very simple task, and very similar to the longhand addition of decimal numbers. As with decimal numbers, you start by adding the bits (digits) one column, or place weight, at a time, from right to left. Unlike decimal addition, there is little to memorize in the way of rules for the addition of binary bits:

0 + 0 =0
1 + 0 =1
0 + 1 =1
1 + 1 =10
1 + 1 + 1 =11

Just as with decimal addition, when the sum in one column is a two-bit (two-digit) number, the least significant figure is written as part of the total sum and the most significant figure is "carried" to the next left column. Consider the following examples

```
.                    11  1 <--- Carry bits ----->11
.    1001101             1001001                1000111
.   +0010010            +0011001                +0010110
.   ---------          ---------              ---------
.    1011111             1100010                1011101
```

The addition problem on the left did not require any bits to be carried, since the sum of bits in each column was either 1 or 0, not 10 or 11. In the other two problems, there definitely were bits to be carried, but the process of addition is still quite simple.

As we'll see later, there are ways that electronic circuits can be built to perform this very task of addition, by representing each bit of each binary number as a voltage signal (either "high," for a 1; or "low" for a 0). This is the very foundation of all the arithmetic which modern digital computers perform.

Negative binary numbers

With addition being easily accomplished, we can perform the operation of subtraction with the same technique simply by making one of the numbers negative. For example, the subtraction problem of 7 - 5 is essentially the same as the addition problem 7 + (-5). Since we already know how to represent positive numbers in binary, all we need to know now is how to represent their negative counterparts and we'll be able to subtract.

Usually we represent a negative decimal number by placing a minus sign directly to the left of the most significant digit, just as in the example above, with -5. However, the whole purpose of using binary notation is for constructing on/off circuits that can represent bit values in terms of voltage (2 alternative values: either "high" or "low"). In this context, we don't have the luxury of a third symbol such as a "minus" sign, since these circuits can only be on or off (two possible states). One solution is to reserve a bit (circuit) that does nothing but represent the mathematical sign:

$101_2 = 5_{10}$ (positive)

Extra bit, representing sign (0=positive,1=negative)

$0101_2 = 5_{10}$ (positive)

Extra bit, representing sign (0=positive,1=negative)

$1101_2 = -5_{10}$ (negative)

As you can see, we have to be careful when we start using bits for any purpose other than standard place-weighted values. Otherwise, 1101_2 could be misinterpreted as the number thirteen when in fact we mean to represent negative five. To keep thingsstraight here, we must first decide how many bits are going to be needed to represent the largest numbers we'll be dealing with, and then be sure not to exceed that bit field length in our arithmetic operations. For the above example, I've limited myself to the representation of numbers from negative seven (1111_2) to positive seven (0111_2), and no more, by making the fourth bit the "sign" bit. Only by first establishing these limits can I avoid confusion of a negative number with a larger, positive number.

Representing negative five as 1101_2 is an example of the *sign-magnitude* system of negative binary numeration. By using the leftmost bit as a sign indicator and not a place-weighted value, I am sacrificing the "pure" form of binary notation for something that gives me a practical advantage: the representation of negative numbers. The leftmost bit is read as the sign, either positive or negative, and the remaining bits are interpreted according to the standard binary notation: left to right, place weights in multiples of two.

As simple as the sign-magnitude approach is, it is not very practical for arithmetic purposes. For instance, how do I add a negative five (1101_2) to any other number, using the standard technique for binary addition? I'd have to invent a new way of doing addition in order for it to work, and if I do that, I might as well just do the job with longhand subtraction; there's no arithmetical advantage to using negative numbers to perform subtraction through addition if we have to do it with sign-magnitude numeration, and that was our goal!

There's another method for representing negative numbers which works with our familiar technique of longhand addition, and also happens to make more sense from a place-weighted numeration point of view, called *complementation*. With this strategy, we assign the leftmost bit to serve a special purpose, just as we did with the sign- magnitude approach, defining our number limits just as before. However, this time, the leftmost bit is more than just a sign bit; rather, it possesses a negative place-weight value. For example, a value of negative five would be represented as such:

Extra bit, place weight = negative eight
.
. $1011_2 = 5_{10}$ (negative)
.
. $(1 \times -8_{10}) + (0 \times 4_{10}) + (1 \times 2_{10}) + (1 \times 1_{10}) = -5_{10}$

With the right three bits being able to represent a magnitude from zero through seven, and the leftmost bit representing either zero or negative eight, we can successfully represent any integer number from negative seven ($1001_2 = -8_{10} + 1_{10} = -7_{10}$) to positive seven ($0111_2 = 0_{10} + 7_{10} = 7_{10}$).

Representing positive numbers in this scheme (with the fourth bit designated as the negative weight) is no different from that of ordinary binary notation. However, representing negative numbers is not quite as straightforward:

zero	0000		
positive one	0001	negative one	1111
positive two	0010	negative two	1110
positive three	0011	negative three	1101
positive four	0100	negative four	1100
positive five	0101	negative five	1011
positive six	0110	negative six	1010
positive seven	0111	negative seven	1001
.		negative eight	1000

Note that the negative binary numbers in the right column, being the sum of the right three bits' total plus the negative eight of the leftmost bit, don't "count" in the same progression as the positive binary numbers in the left column. Rather, the right three bits have to be set at the proper value to equal the desired (negative) total when summed with the negative eight place value of the leftmost bit.

Those right three bits are referred to as the *two's complement* of the corresponding positive number. Consider the following comparison:

positivenumber	two'scomplement
001	111
010	110
011	101
100	100
101	011
110	010
111	001

In this case, with the negative weight bit being the fourth bit (place value of negative eight), the two's complement for any positive number will be whatever value is needed to add to negative eight to make that positive value's negative equivalent. Thankfully, there's an easy way to figure out the two's complement for any binary number: simply invert all the bits of that number, changing all 1's to 0's and vice versa (to arrive at what

is called the *one's complement*) and then add one! For example, to obtain the two's complement of five (101_2), we would first invert all the bits to obtain 010_2 (the "one's complement"), then add one to obtain 011_2, or -5_{10} in three-bit, two's complement form.

Interestingly enough, generating the two's complement of a binary number works the same if you manipulate *all* the bits, including the leftmost (sign) bit at the same time as the magnitude bits. Let's try this with the former example, converting a positive five to a negative five, but performing the complementation process on all four bits. We must be sure to include the 0 (positive) sign bit on the original number, five (0101_2). First, inverting all bits to obtain the one's complement: 1010_2. Then, adding one, we obtain the final answer: 1011_2, or -5_{10} expressed in four-bit, two's complement form.

It is critically important to remember that the place of the negative-weight bit must be already determined before any two's complement conversions can be done. If our binary numeration field were such that the eighth bit was designated as the negative- weight bit (10000000_2), we'd have to determine the two's complement based on all seven of the other bits. Here, the two's complement of five (0000101_2) would be 1111011_2. A positive five in this system would be represented as 00000101_2, and a negative five as11111011_2.

Binary Subtraction

We can subtract one binary number from another by using the standard techniques adapted for decimal numbers (subtraction of each bit pair, right to left, "borrowing" as needed from bits to the left). However, if we can leverage the already familiar (and easier) technique of binary addition to subtract, that would be better. As we just learned, we can represent negative binary numbers by using the "two's complement" method and a negative place-weight bit. Here, we'll use those negative binary numbers to subtract through addition. Here's a sample problem:

Subtraction: $7_{10} - 5_{10}$ Addition equivalent: $7_{10} + (-5_{10})$

If all we need to do is represent seven and negative five in binary (two's complemented) form, all we need is three bits plus the negative-weight bit:

positive seven = 0111_2 negative five =1011_2

Now, let's add them together:

```
.                1111  <--- Carry bits
.                 0111
.                +1011
.                ------
.                 10010
.                 |
.          Discard extra bit
.
.         Answer =0010₂
```

Since we've already defined our number bit field as three bits plus the negative-weight bit, the fifth bit in the answer (1) will be discarded to give us a result of 0010_2, or positive two, which is the correct answer.

Another way to understand why we discard that extra bit is to remember that the leftmost bit of the lower number possesses a negative weight, in this case equal to negative eight. When we add these two binary numbers together, what we're actually doing with the MSBs is subtracting the lower number's MSB from the upper number's MSB. In subtraction, one never "carries" a digit or bit on to the next left place-weight.

Let's try another example, this time with larger numbers. If we want to add -25_{10} to 18_{10}, we must first decide how large our binary bit field must be. To represent the largest (absolute value) number in our problem, which is twenty-five, we need at least five bits, plus a sixth bit for the negative-weight bit. Let's start by representing positive twenty- five, then finding the two's complement and putting it all together into on enumeration:

$+25_{10} = 011001_2$ (showing all six bits) One's complement of $11001_2 = 100110_2$
One's complement + 1 = two's complement $= 100111_2$
$-25_{10} = 100111_2$

Essentially, we're representing negative twenty-five by using the negative-weight (sixth) bit with a value of negative thirty-two, plus positive seven (binary111_2).

Now, let's represent positive eighteen in binary form, showing all six bits:

$18_{10} = 010010_2$

```
.
.      Now, let's add them together and see what we get:
.
.                  11   <--- Carry bits
.                 100111
.                +010010
.
```

. 111001

Since there were no "extra" bits on the left, there are no bits to discard. The leftmost bit on the answer is a 1, which means that the answer is negative, in two's complement form, as it should be. Converting the answer to decimal form by summing all the bits times their respective weight values, we get:

$(1 \times -32_{10}) + (1 \times 16_{10}) + (1 \times 8_{10}) + (1 \times 1_{10}) = -7_{10}$

Indeed -7_{10} is the proper sum of -25_{10} and 18_{10}.

Binary Codes

Binary codes are codes which are represented in binary system with modification from the original ones. Below we will be seeing the following:

- Weighted Binary Systems
- Non Weighted Codes

Weighted Binary Systems

Weighted binary codes are those which obey the positional weighting principles, each position of the number represents a specific weight. The binary counting sequence is an example.

Decimal	8421	2421	5211	Excess-3
0	0000	0000	0000	0011
1	0001	0001	0001	0100
2	0010	0010	0011	0101
3	0011	0011	0101	0110
4	0100	0100	0111	0111
5	0101	1011	1000	1000
6	0110	1100	1010	1001
7	0111	1101	1100	1010
8	1000	1110	1110	1011
9	1001	1111	1111	1100

8421 Code/BCDCode

The BCD (Binary Coded Decimal) is a straight assignment of the binary equivalent. It is possible to assign weights to the binary bits according to their positions. The weights in the BCD code are8,4,2,1.

Example: The bit assignment 1001, can be seen by its weights to represent the decimal 9because:

1x8+0x4+0x2+1x1 =9

2421Code

This is a weighted code, its weights are 2, 4, 2 and 1. A decimal number is represented in 4-bit form and the total four bits weight is 2 + 4 + 2 + 1 = 9. Hence the 2421 code represents the decimal numbers from 0 to9.

5211Code

This is a weighted code, its weights are 5, 2, 1 and 1. A decimal number is represented in 4-bit form and the total four bits weight is 5 + 2 + 1 + 1 = 9. Hence the 5211 code represents the decimal numbers from 0 to9.

Reflective Code

A code is said to be reflective when code for 9 is complement for the code for 0, and so is for 8 and 1 codes, 7 and 2, 6 and 3, 5 and 4. Codes 2421, 5211, and excess-3 are reflective, whereas the 8421 code isnot.

Sequential Codes

A code is said to be sequential when two subsequent codes, seen as numbers in binary representation, differ by one. This greatly aids mathematical manipulation of data. The 8421 and Excess-3 codes are sequential, whereas the 2421 and 5211 codes are not.

Non Weighted Codes

Non weighted codes are codes that are not positional weighted. That is, each position within the binary number is not assigned a fixed value.

Excess-3Code

Excess-3 is a non weighted code used to express decimal numbers. The code derives its name from the fact that each binary code is the corresponding 8421 code plus0011(3).

Example: 1000 of 8421 = 1011 inExcess-3

Gray Code

The gray code belongs to a class of codes called minimum change codes, in which only one bit in the code changes when moving from one code to the next. The Gray code is non-weighted code, as the position of bit does not contain any weight. The gray code is a reflective digital code which has the special property that any two subsequent numbers codes differ by only one bit. This is also called a unit-distance code. In digital Gray code has got a specialplace.

DecimalNumber	BinaryCode	GrayCode
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110

12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

Binary to Gray Conversion

- Gray Code MSB is binary code MSB.
- Gray Code MSB-1 is the XOR of binary code MSB andMSB-1.
- MSB-2 bit of gray code is XOR of MSB-1 and MSB-2 bit of binary code.
- MSB-N bit of gray code is XOR of MSB-N-1 and MSB-N bit of binary code.

Error Detecting and Correction Codes

For reliable transmission and storage of digital data, error detection and correction is required. Below are a few examples of codes which permit error detection and error correction after detection.

Error Detecting Codes

When data is transmitted from one point to another, like in wireless transmission, or it is just stored, like in hard disks and memories, there are chances that data may get corrupted. To detect these data errors, we use special codes, which are error detection codes.

Parity

In parity codes, every data byte, or nibble (according to how user wants to use it) is checked if they have even number of ones or even number of zeros. Based on this information an additional bit is appended to the original data. Thus if we consider 8-bit data, adding the parity bit will make it 9 bit long.

At the receiver side, once again parity is calculated and matched with the received parity (bit 9), and if they match, data is ok, otherwise data is corrupt.

There are two types of parity:

- Even parity: Checks if there is an even number of ones; if so, parity bit is zero. When the number of ones is odd then parity bit is set to1.
- Odd Parity: Checks if there is an odd number of ones; if so, parity bit is zero. When number of ones is even then parity bit is set to1.

Checksums

The parity method is calculated over byte, word or double word. But when errors need to be checked over 128 bytes or more (basically blocks of data), then calculating parity is not the right way. So we have checksum, which allows checking for errors on block of data. There are many variations of checksum.

- Adding all bytes
- CRC
- Fletcher's checksum
- Adler-32

The simplest form of checksum, which simply adds up the asserted bits in the data, cannot detect a number of types of errors. In particular, such a checksum is not changed by:

- Reordering of the bytes in the message
- Inserting or deleting zero-valued bytes
- Multiple errors which sum to zero

Example of Checksum: Given 4 bytes of data (can be done with any number of bytes): 25h, 62h, 3Fh,52h

- Adding all bytes together gives118h.
- Drop the Carry Nibble to give you18h.

Get the two's complement of the 18h to get E8h. This is the checksum byte

To Test the Checksum byte simply adds it to the original group of bytes. This should give you 200h.

Drop the carry nibble again giving 00h. Since it is 00h this means the checksum means the bytes were probably not changed.

Error-Correcting Codes

Error-correcting codes not only detect errors, but also correct them. This is used normally in Satellite communication, where turn-around delay is very high as is the probability of data getting corrupt.

ECC (Error correcting codes) are used also in memories, networking, Hard disk, CDROM, DVD etc. Normally in networking chips (ASIC), we have 2 Error detection bits and 1 Error correction bit.

HammingCode

Hamming code adds a minimum number of bits to the data transmitted in a noisy channel, to be able to correct every possible one-bit error. It can detect (not correct) two-bits errors and cannot distinguish between 1-bit and 2-bits inconsistencies. It can't - in general - detect 3(or more)-bits errors.

The idea is that the failed bit position in an n-bit string (which we'll call X) can be represented in binary with log2(n) bits, hence we'll try to get it adding just log2(n)bits.

First, we set m = n + log2(n) to the encoded string length and we number each bit position starting from 1 through m. Then we place these additional bits at power-of-two positions, that is 1, 2, 4, 8..., while remaining ones (3, 5, 6, 7...) hold the bit string in the originalorder.

Now we set each added bit to the parity of a group of bits. We group bits this way: we form a group for every parity bit, where the following relationholds:

position(bit) AND position(parity) =position(parity)
(Note that: AND is the bit-wise boolean AND; parity bits are included in the groups; each bit can belong to one or moregroups.)

So bit 1 groups bits 1, 3, 5, 7... while bit 2 groups bits 2, 3, 6, 7, 10... , bit 4 groups bits 4, 5, 6,7, 12, 13... and soon.

Thus, by definition, X (the failed bit position defined above) is the sum of the incorrect parity bits positions (0 for noerrors).

To understand why it is so, let's call Xn the nth bit of X in binary representation. Now consider that each parity bit is tied to a bit of X: parity1 -> X1, parity2 -> X2, parity4 -> X3, parity8 -> X4 and so on - for programmers: they are the respective AND masks -. By construction, the failed bit makes fail only the parity bits which correspond to the 1s in X, so each bit of X is 1 if the corresponding parity is wrong and 0 if it iscorrect.

Note that the longer the string, the higher the throughput n/m and the lower the probability that no more than one bit fails. So the string to be sent should be broken into blocks whose length depends on the transmission channel quality (the cleaner the channel, the bigger the block). Also, unless it's guaranteed that at most one bit per block fails, a checksum or some other form of data integrity check should beadded.

AlphanumericCodes

The binary codes that can be used to represent all the letters of the alphabet, numbers and mathematical symbols, punctuation marks, are known as alphanumeric codes or character codes. These codes enable us to interface the input-output devices like the keyboard, printers, video displays with thecomputer.

ASCIICode

ASCII stands for American Standard Code for Information Interchange. It has become a world standard alphanumeric code for microcomputers and computers. It is a 7-bit code representing 27 = 128 different characters. These characters represent 26 upper case letters (A to Z), 26 lowercase letters (a to z), 10 numbers (0 to 9), 33 special characters and symbols and 33 controlcharacters.

The 7-bit code is divided into two portions, The leftmost 3 bits portion is called zone bits and the 4-bit portion on the right is called numericbits.

An 8-bit version of ASCII code is known as USACC-II 8 or ASCII-8. The 8-bit version can represent a maximum of 256characters.

EBCDICCode

EBCDIC stands for Extended Binary Coded Decimal Interchange. It is mainly used with large computer systems like mainframes. EBCDIC is an 8-bit code and thus accomodates up to 256 characters. An EBCDIC code is divided into two portions: 4 zone bits (on the left) and 4 numeric bits (on theright).

Boolean algebra: derives its name from the mathematician George Boole. Symbolic Logic uses values, variables and operations:

- True is represented by the value1.
- False is represented by the value0.
- Variables are represented by letters and can have one of two values, either 0 or 1. Operations are functions of one or morevariables.
- AND is represented byX.Y
- OR is represented by X +Y
- NOT is represented by X' . Throughout this tutorial the X' form will be used and sometime !X will beused.

These basic operations can be combined to giveexpressions.

Example:

- X
- X.Y
- W.X.Y +Z

Precedence

As with any other branch of mathematics, these operators have an order of precedence. NOT operations have the highest precedence, followed by AND operations, followed by OR operations. Brackets can be used as with other forms of algebra. e.g. X.Y + Z and X.(Y + Z) are not the samefunction.

FunctionDefinitions

The logic operations given previously are defined as follows: Define f(X,Y) to be

some function of the variables X andY.

f(X,Y) =X.Y

- 1 if X = 1 and Y =1
- 0Otherwise

f(X,Y) = X +Y

- 1 if X = 1 or Y =1
- 0Otherwise

f(X) =X'

- 1 if X = 0
- 0Otherwise

TruthTables

Truth tables are a means of representing the results of a logic function using a table. They are constructed by defining all possible combinations of the inputs to a function, and then calculating the output for each combination in turn. For the three functions we have just defined, the truth tables are asfollows.

AND

X	Y	F(X,Y)
0	0	0
0	1	0
1	0	0
1	1	1

OR

X	Y	F(X,Y)
0	0	0
0	1	1
1	0	1
1	1	1

NOT

X	F(X)
0	1
1	0

Truth tables may contain as many input variables asdesired.

F(X,Y,Z) = X.Y +Z

X	Y	Z	F(X,Y,Z)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Boolean SwitchingAlgebras

A Boolean Switching Algebra is one which deals only with two-valued variables. Boole's general theory covers algebras which deal with variables which can hold n values.

Consider a set **S = { 0.1}**

Consider two binary operations, + and . , and one unary operation, -- , that act on these elements. **[S, ., +, --, 0, 1]** is called a switching algebra that satisfies the following axiomsS

If X S and Y S thenX.Y S If X ∈S
and Y ∈S then X+Y∈S

an identity 0 for + such that X + 0=X ∃an identity 1
for . such that X . 1 =X

X + Y = Y + X X . Y = Y
.X

X.(Y + Z) = X.Y +X.Z
X + Y.Z = (X + Y) . (X +Z)

X S a complementX'suchthat X + X' =1

X . X' =0

The complement X' isunique.

A number of theorems may be proved for switchingalgebras

IdempotentLaw

X + X=X X . X =X

DeMorgan'sLaw

(X + Y)' = X' . Y', These can be proved by the use of truthtables.

Proof of (X + Y)' = X' .Y'

X	Y	X+Y	(X+Y)'
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

X	Y	X'	Y'	X'.Y'
0	0	1	1	1
0	1	1	0	0
1	0	0	1	0
1	1	0	0	0

The two truth tables are identical, and so the two expressions areidentical.

(X.Y) = X' + Y', These can be proved by the use of truthtables.

Proof of (X.Y) = X' +Y'

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

X	Y	X'	Y'	X'+Y'
0	0	1	1	1
0	1	1	0	1
1	0	0	1	1
1	1	0	0	0

Note : DeMorgans Laws are applicable for any number ofvariables.

X + 1 =1
X . 0 =0

X + (X . Y) =X
X . (X + Y) =X

X + (X' . Y) = X +Y
X.(X' + Y) =X.Y

If X + Y = 1 and X.Y = 0 then X =Y'

X'' = X 0' =1

X + (Y + Z) = (X + Y) +Z
X . (Y . Z) = (X . Y) .Z

DualityPrinciple

InBoolean algebrastheDualityPrinciplecanbeisobtainedbyinterchangingAND and OR operators and replacing 0's by 1's and 1's by 0's. Compare the identities on the left side with the identities on theright.

Example

X.Y+Z' =(X'+Y').Z

Consensustheorem

X.Y + X'.Z + Y.Z = X.Y +X'.Z or dual
form asbelow
(X + Y).(X' + Z).(Y + Z) = (X + Y).(X' +Z) Proof of

X.Y + X'.Z + Y.Z = X.Y +X'.Z:

X.Y + X'.Z +Y.Z	**= X.Y +X'.Z**
X.Y + X'.Z +(X+X').Y.Z	= X.Y +X'.Z
X.Y.(1+Z) +X'.Z.(1+Y)	= X.Y +X'.Z

X.Y +X'.Z = X.Y +X'.Z

(X.Y'+Z).(X+Y).Z = X.Z+Y.Z instead ofX.Z+Y'.Z
X.Y'Z+X.Z+Y.Z
(X.Y'+X+Y).Z (X+Y).Z
X.Z+Y.Z
The term which is left out is called the consensusterm.

Given a pair of terms for which a variable appears in one term, and its complement in the other, then the consensus term is formed by ANDing the original terms together, leaving out the selected variable and itscomplement.

Example:
The consensus of X.Y and X'.Z isY.Z

The consensus of X.Y.Z and Y'.Z'.W' is(X.Z).(Z.W')

Shannon ExpansionTheorem

The Shannon Expansion Theorem is used to expand a Boolean logic function (F)intermsof(orwithrespectto)aBooleanvariable(X),asinthefollowing forms.

F = X . F (X = 1) + X' . F (X =0)

where F (X = 1) represents the function F evaluated with X set equal to 1; F (X = 0) represents the function F evaluated with X set equal to 0. Also the following

function F can be expanded with respect toX, F = X' . Y + X . Y . Z' + X' . Y' .Z

= X . (Y . Z') + X' . (Y + Y' .Z)

Thus, the function F can be split into two smallerfunctions. F (X = '1') = Y

.Z'

This is known as the cofactor of F with respect to X in the previous logic equation. The cofactor of F with respect to X may also be represented as F X (the cofactor of F with respect to X' is F X'). Using the Shannon Expansion Theorem, a Boolean function may be expanded with respect to any of its variables. For example, if we expand F with respect to Y instead ofX,

F = X' . Y + X . Y . Z' + X' . Y' .Z

= Y . (X' + X . Z') + Y' . (X' .Z)

A function may be expanded as many times as the number of variables it contains until the canonical form is reached. The canonical form is a unique representation for any Boolean function that uses only minterms. A minterm is a product term that contains all the variables of F¿such as X . Y' .Z).

Any Boolean function can be implemented using multiplexer blocks by representing it as a series

of terms derived using the Shannon Expansion Theorem.

Identity	Dual
Operations with 0 and1	
X + 0 = X(identity)	X.1 =X
X + 1 = 1 (nullelement)	X.0 =0
Idempotencytheorem	
X + X =X	X.X =X
Complementarity	
X + X' =1	X.X' =0
Involutiontheorem	
(X')' =X	
Cummutativelaw	
X + Y = Y +X	X.Y = YX
Associativelaw	
(X + Y) + Z = X + (Y + Z) = X + Y +Z	(XY)Z = X(YZ) =XYZ
Distributivelaw	
X(Y + Z) = XY +XZ	X + (YZ) = (X + Y)(X +Z)
DeMorgan's theorem	
(X + Y + Z + ...)' = X'Y'Z'... or { f (X1,X2,...,Xn,0,1,+,.) } = { f (X1',X2',...,Xn',1,0,.,+)}	(XYZ...)' = X' + Y' + Z' +
Simplificationtheorems	
XY + XY' = X(uniting)	(X + Y)(X + Y') =X

X + XY = X(absorption)
(X + Y')Y = XY(adsorption)
Consensustheorem
XY + X'Z + YZ = XY +X'Z
Duality
$(X + Y + Z + ...)^D$ = XYZ... or$\{f(X1,X2,...,Xn,0,1,+,.)\}^D$ = f(X1,X2,...,Xn,1,0,.,+)
Shannon ExpansionTheorem
f(X . X . X)
$f(X_1.....X_k....X_n)$

AlgebraicManipulation

Minterms andMaxterms

Any boolean expression may be expressed in terms of either minterms or maxterms. To do this we must first define the concept of a literal. A literal is a single variable within a term which may or may not be complemented. For an expression with N variables, minterms and maxterms are defined as follows:

- A minterm is the product of N distinct literals where each literal occurs exactly once.
- A maxterm is the sum of N

distinct literals where each literal occurs

exactly once For a two-variable

expression, the minterms and maxterms

are asfollows

1	0	X.	X'+
1	1	X.	X'+

For a three-variable expression, the minterms and maxterms are asfollows

X	Y	Z	Minterm	Maxterm
0	0	0	X'.Y'.Z'	X+Y+Z
0	0	1	X'.Y'.Z	X+Y+Z'
0	1	0	X'.Y.Z'	X+Y'+Z
0	1	1	X'.Y.Z	X+Y'+Z'
1	0	0	X.Y'.Z'	X'+Y+Z
1	0	1	X.Y'.Z	X'+Y+Z'
1	1	0	X.Y.Z'	X'+Y'+Z
1	1	1	X.Y.Z	X'+Y'+Z'

This allows us to represent expressions in either Sum of Products or Product of Sums forms

Sum Of Products(SOP)

The Sum of Products form represents an expression as a sum ofminterms.

$F(X, Y, ...) = Sum(a_k.m_k)$

where a_k is 0 or 1 and m_k is aminterm.

To derive the Sum of Products form from a truth table, OR together all of the minterms which give a value of1.

Example -SOP

Consider the truthtable

X	Y	F	Minterm
0	0	0	X'.Y'
0	1	0	X'Y
1	0	1	X.Y'
1	1	1	X.Y

Here SOP is f(X.Y) = X.Y' +X.Y

Product Of Sum(POS)

The Product of Sums form represents an expression as a product of maxterms. F(X, Y,

.....) = Product (b_k + M_k), where b_k is 0 or 1 and M_k is amaxterm.

To derive the Product of Sums form from a truth table, AND together all of themaxterms which give a value of0.

Example -POS

Consider the truth table from the previousexample.

X	Y	F	Maxterm
0	0	1	X+Y
0	1	0	X+Y'
1	0	1	X'+Y
1	1	1	X'+Y'

Here POS is F(X,Y) =(X+Y')

Exercise

Give the expression represented by the following truth table in both Sum of Products and Product of Sumsforms.

X	Y	Z	F(X,Y,X)
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Conversion between POS andSOP

Conversion between the two forms is done by application of DeMorgansLaws.

Simplification

As with any other form of algebra you have encountered, simplification of expressions can be performed with Booleanalgebra.

Example

Show that X.Y.Z' + X'.Y.Z' + Y.Z = Y

X.Y.Z' + X'.Y.Z' + Y.Z = Y.Z' + Y.Z =Y

Example

Show that (X.Y' + Z).(X + Y).Z = X.Z +Y.Z

(X.Y' + Z).(X +Y).Z
= (X.Y' + Z.X +Y'.Z).Z
= X.Y'Z + Z.X +Y'.Z
= Z.(X.Y' + X +Y')
=Z.(X+Y')

Digital LogicGates

LogicGates

A logic gate is an electronic circuit/device which makes the logical decisions. To arrive at this decisions, the most common logic gates used are OR, AND, NOT, NAND, and NOR gates. The NAND and NOR gates are called universal gates. The exclusive-OR gate is another logic gate which can be constructed using AND, OR and NOTgate.

Logic gates have one or more inputs and only one output. The output is active only for certain input

combinations. Logic gates are the building blocks of any digital circuit. Logic gates arealso

called switches. With the advent of integrated circuits, switches have been replaced by TTL (Transistor Transistor Logic) circuits and CMOS circuits. Here I give example circuits on how to construct simplesgates.

SymbolicLogic

Boolean algebra derives its name from the mathematician George Boole. Symbolic Logic uses values, variables andoperations.

Inversion

A small circle on an input or an output indicates inversion. See the NOT, NAND and NOR gates given below forexamples.

Multiple InputGates

Given commutative and associative laws, many logic gates can be implemented with more than two inputs, and for reasons of space in circuits, usually multiple input, complex gates are made. You will encounter such gates in real world (maybe you could analyze an ASIC lib to findthis).

GatesTypes

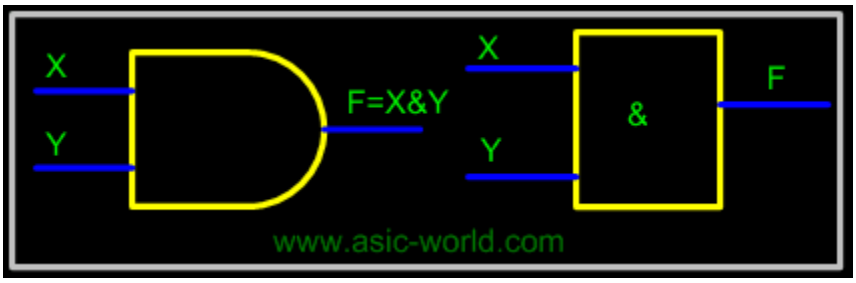
- AND
- OR
- NOT
- BUF
- NAND
- NOR
- XOR
- XNOR

ANDGate

The AND gate performs logical multiplication, commonly known as AND function. The AND gate has two or more inputs and single output. The output of AND gate is HIGH only when all its inputs are HIGH (i.e. even if one input is LOW, Output will beLOW).

If X and Y are two inputs, then output F can be represented mathematically as F = X.Y, Here dot (.) denotes the AND operation. Truth table and symbol of the AND gate is shown in the figurebelow.

Symbol

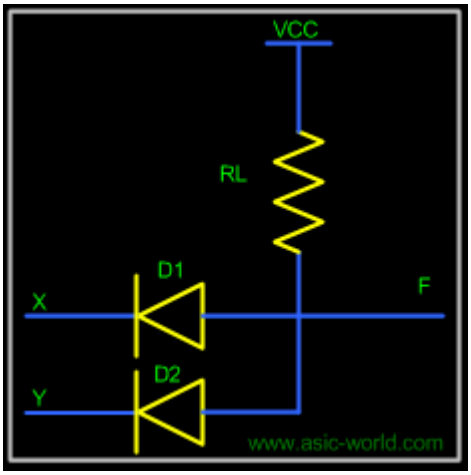


TruthTable

X	Y	F=(X.Y)
0	0	0
0	1	0
1	0	0
1	1	1

Two input AND gate using "diode-resistor" logic is shown in figure below, where X, Y are inputs and F is theoutput.

Circuit



If X = 0 and Y = 0, then both diodes D1 and D2 are forward biased and thus both diodes conduct and pull Flow.

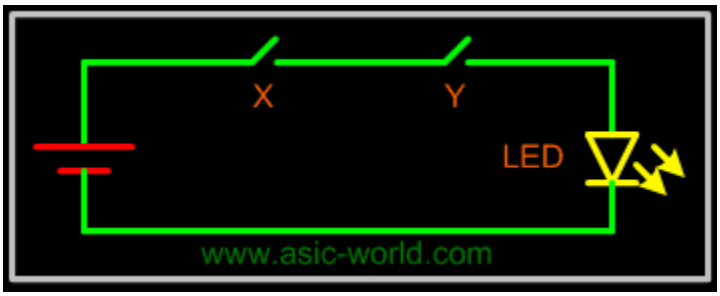
If X = 0 and Y = 1, D2 is reverse biased, thus does not conduct. But D1 is forward biased, thus conducts and thus pulls Flow.

If X = 1 and Y = 0, D1 is reverse biased, thus does not conduct. But D2 is forward biased, thus conducts and thus pulls Flow.

If X = 1 and Y = 1, then both diodes D1 and D2 are reverse biased and thus both the diodes are in cut-off and thus there is no drop in voltage at F. Thus F isHIGH.

Switch Representation of ANDGate

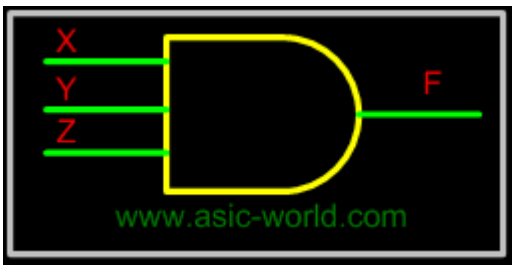
In the figure below, X and Y are two switches which have been connected in series (or just cascaded) with the load LED and source battery. When both switches are closed, current flows toLED.



Three Input ANDgate

Since we have already seen how a AND gate works and I will just list the truth table of a 3 input AND gate. The figure below shows its symbol and truthtable.

Circuit



TruthTable

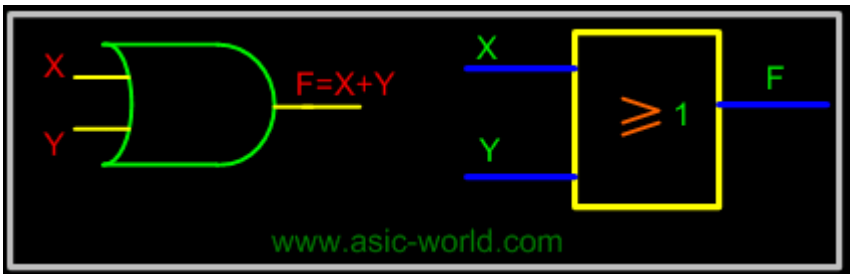
X	Y	Z	F=X.Y.Z
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

ORGate

The OR gate performs logical addition, commonly known as OR function. The OR gate has two or more inputs and single output. The output of OR gate is HIGH only when any one of its inputs are HIGH (i.e. even if one input is HIGH, Output will beHIGH).

If X and Y are two inputs, then output F can be represented mathematically as F = X+Y. Here plus sign (+) denotes the OR operation. Truth table and symbol of the OR gate is shown in the figurebelow.

Symbol

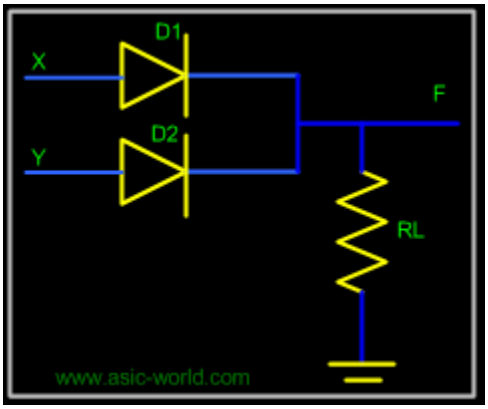


TruthTable

X	Y	F=(X+Y)
0	0	0
0	1	1
1	0	1
1	1	1

Two input OR gate using "diode-resistor" logic is shown in figure below, where X, Y are inputs and F is theoutput.

Circuit



If X = 0 and Y = 0, then both diodes D1 and D2 are reverse biased and thus both the diodes are in cut-off and thus F islow.

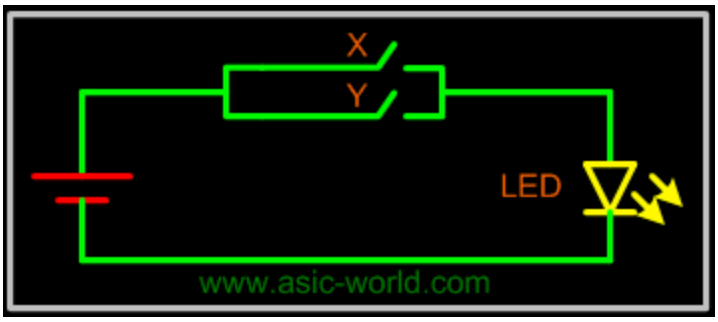
If X = 0 and Y = 1, D1 is reverse biased, thus does not conduct. But D2 is forward biased, thus conducts and thus pulling F toHIGH.

If X = 1 and Y = 0, D2 is reverse biased, thus does not conduct. But D1 is forward biased, thus conducts and thus pulling F toHIGH.

If X = 1 and Y = 1, then both diodes D1 and D2 are forward biased and thus both the diodes conduct and thus F isHIGH.

Switch Representation of ORGate

In the figure, X and Y are two switches which have been connected in parallel, and this is connected in series with the load LED and source battery. When both switches are open, current does not flow to LED, but when any switch is closed then currentflows.



Three Input ORgate

Since we have already seen how an OR gate works, I will just list the truth table of a 3-input OR gate. The figure below shows its circuit and truthtable.

TruthTable

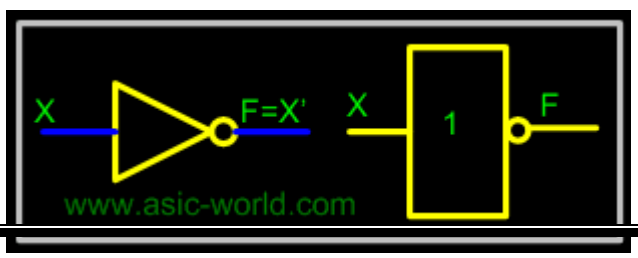
X	Y	Z	F=X+Y+Z
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

NOTGate

The NOT gate performs the basic logical function called inversion or complementation. NOT gate is also called inverter. The purpose of this gate is to convert one logic level into the opposite logic level. It has one input and one output. When a HIGH level is applied to an inverter, a LOW level appears on its output and viceversa.

If X is the input, then output F can be represented mathematically as F = X', Here apostrophe (') denotes the NOT (inversion) operation. There are a couple of other ways to represent inversion, F= !X, here ! represents inversion. Truth table and NOT gate symbol is shown in the figurebelow.

Symbol

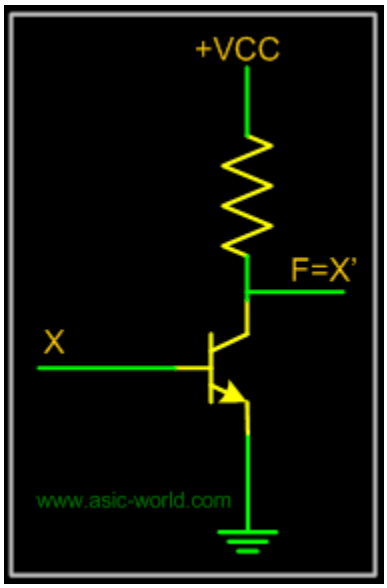


TruthTable

X	Y=X'
0	1
1	0

NOT gate using "transistor-resistor" logic is shown in the figure below, where X is the input and F is theoutput.

Circuit



When X = 1, The transistor input pin 1 is HIGH, this produces the forward bias across the emitter base junction and so the transistor conducts. As the collector current flows, the voltage drop across RL increases and hence F isLOW.

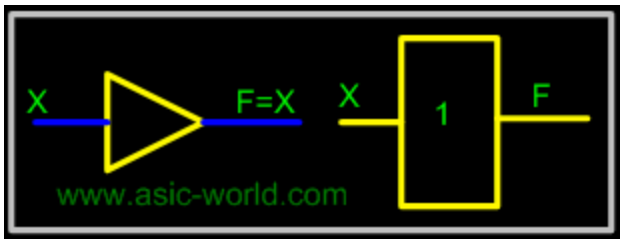
When X = 0, the transistor input pin 2 is LOW: this produces no bias voltage across the transistor base emitter junction. Thus Voltage at F isHIGH.

BUFGate

Buffer or BUF is also a gate with the exception that it does not perform any logical operation on its input. Buffers just pass input to output. Buffers are used to increase the drive strength or sometime just to introduce delay. We will look at this in detaillater.

If X is the input, then output F can be represented mathematically as F = X. Truth table and symbol of the Buffer gate is shown in the figurebelow.

Symbol



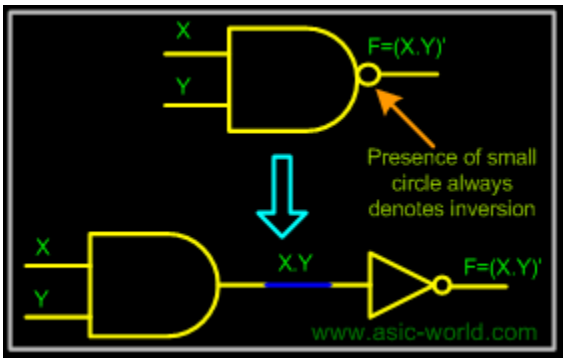
TruthTable

X	Y=X
0	0
1	1

NANDGate

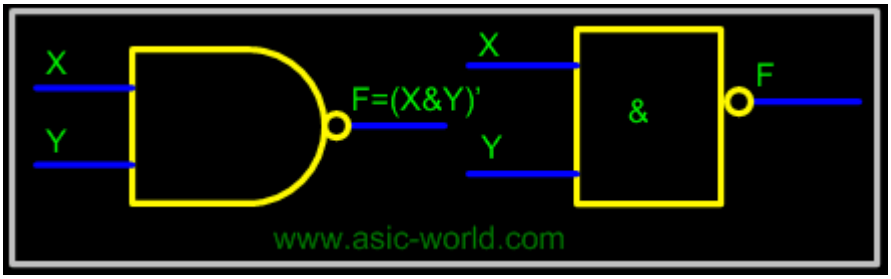
NAND gate is a cascade of AND gate and NOT gate, as shown in the figure below. It has two or more inputs and only one output. The output of NAND gate is HIGH when any one of its input is LOW (i.e. even if one input is LOW, Output will beHIGH).

NAND From AND andNOT



If X and Y are two inputs, then output F can be represented mathematically as F = (X.Y)', Here dot (.) denotes the AND operation and (') denotes inversion. Truth table and symbol of the N AND gate is shown in the figurebelow.

Symbol



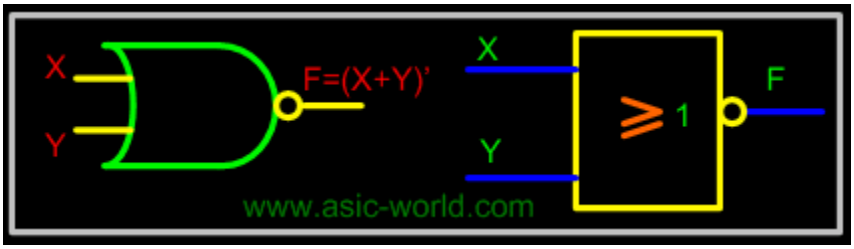
TruthTable

X	Y	F=(X.Y)'
0	0	1
0	1	1
1	0	1
1	1	0

NOR Gate

NOR gate is a cascade of OR gate and NOT gate, as shown in the figure below. It has two or more inputs and only one output. The output of NOR gate is HIGH when any all its inputs are LOW (i.e. even if one input is HIGH, output will beLOW).

Symbol



If X and Y are two inputs, then output F can be represented mathematically as F = (X+Y)'; here plus (+) denotes the OR operation and (') denotes inversion. Truth table and symbol of the NOR gate is shown in the figurebelow.

TruthTable

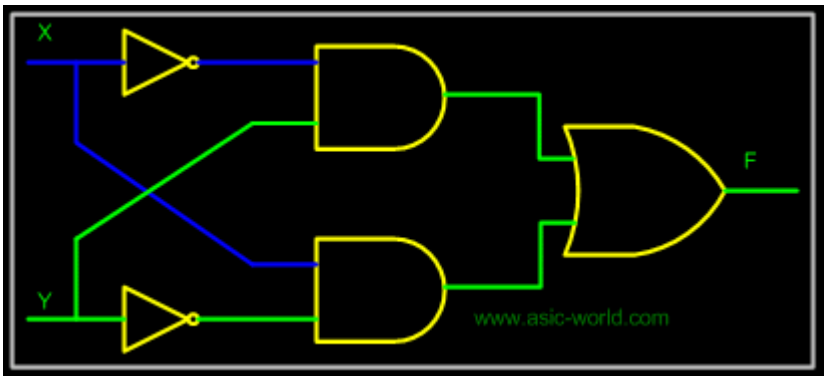
X	Y	F=(X+Y)'
0	0	1
0	1	0
1	0	0
1	1	0

XORGate

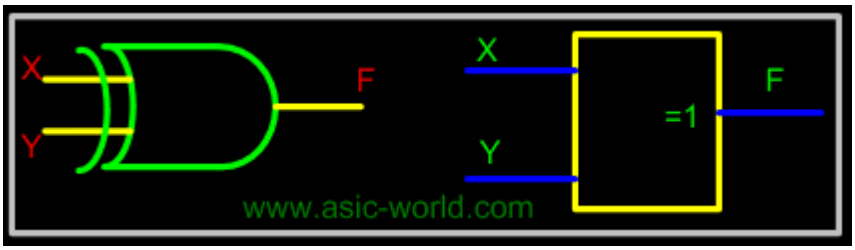
An Exclusive-OR (XOR) gate is gate with two or three or more inputs and one output. The output of a two-input XOR gate assumes a HIGH state if one and only one input assumes a HIGH state. This is equivalent to saying that the output is HIGH if either input X or input Y is HIGH exclusively, and LOW when both are 1 or 0simultaneously.

If X and Y are two inputs, then output F can be represented mathematically asF= X⊕Y, Here ⊕ denotes the XOR operation. ⊕Y and is equivalent to X.Y' + X'.Y. Truth table and symbol of the XOR gate is shown in the figurebelow.

XOR From Simplegates



Symbol



TruthTable

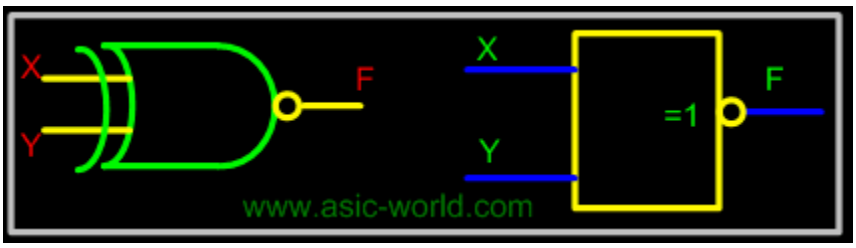
X	Y	F=(X Y)
0	0	0
0	1	1
1	0	1
1	1	0

XNOR Gate

An Exclusive-NOR (XNOR) gate is gate with two or three or more inputs and one output. The output of a two-input XNOR gate assumes a HIGH state if all the inputs assumes same state. This is equivalent to saying that the output is HIGH if both input X and input Y is HIGH exclusively or same as input X and input Y is LOW exclusively, and LOW when both are notsame.

If X and Y are two inputs, then output F can be represented mathematically asF= X⊕Y, Here ⊕ denotes the XNOR operation. ⊕Y and is equivalent to X.Y + X'.Y'. Truth table and symbol of the XNOR gate is shown in the figurebelow.

Symbol



TruthTable

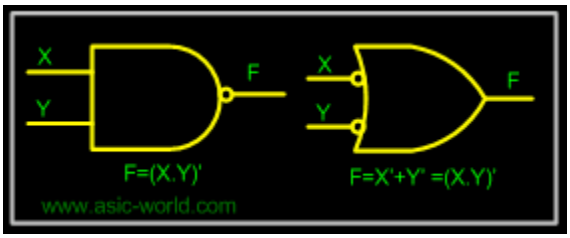
X	Y	F=(X Y)'
0	0	1
0	1	0
1	0	0
1	1	1

UniversalGates

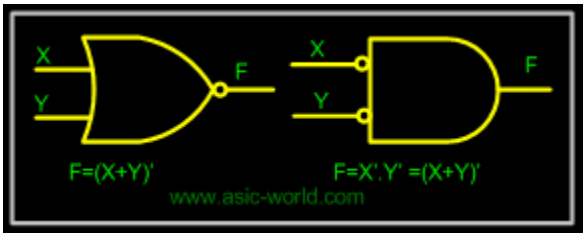
Universal gates are the ones which can be used for implementing any gate like AND, OR and NOT, or any combination of these basic gates; NAND and NOR gates are universal gates. But there are some rules that need to be followed when implementing NAND or NOR basedgates.

To facilitate the conversion to NAND and NOR logic, we have two new graphic symbols for thesegates.

NANDGate



NOR Gate



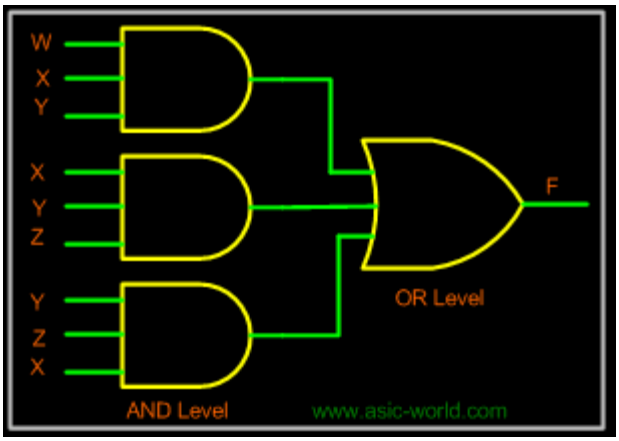
Realization of logic function usingNANDgates

Anylogicfunctioncanbeimplementedusing NAND gates. To achieve this, first the logic function has to be written in Sum of Product (SOP) form. Once logic function is converted to SOP, then is very easy to implement using NAND gate. In other words any logic circuit with AND gates in first level and OR gates in second level can be converted into a NAND-NAND gatecircuit.

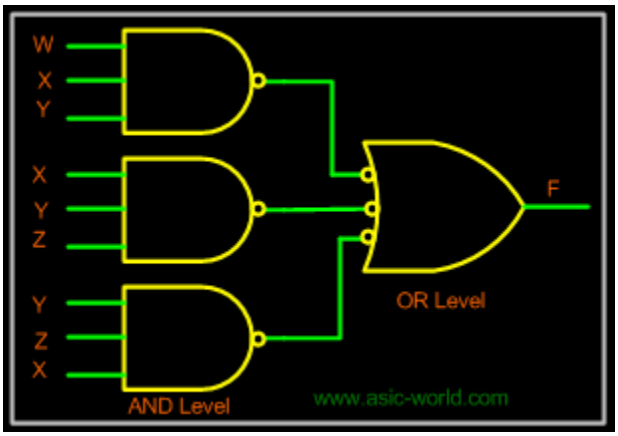
Consider the following SOPexpression

F = W.X.Y + X.Y.Z +Y.Z.W

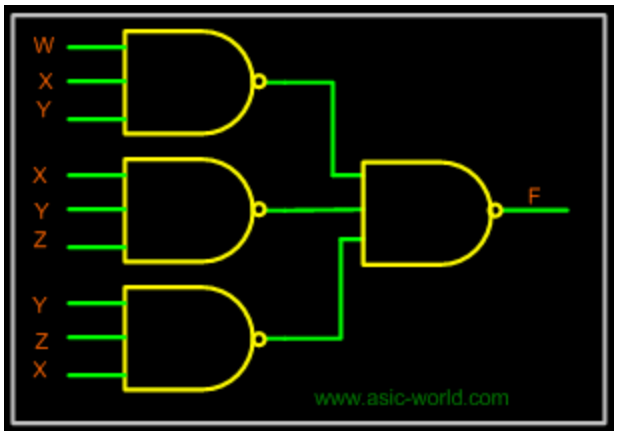
The above expression can be implemented with three AND gates in first stage and one OR gate in second stage as shown infigure.



If bubbles are introduced at AND gates output and OR gates inputs (the same for NOR gates), the above circuit becomes as shown infigure.



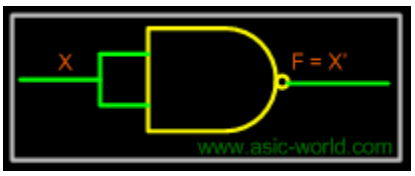
Now replace OR gate with input bubble with the NAND gate. Now we have circuit which is fully implemented with just NANDgates.



Realization of logic gates usingNANDgates

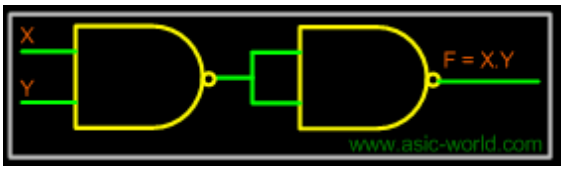
Implementing an inverter using NANDgate

Input	Output	Rule
(X.X)'	=X'	Idempotent



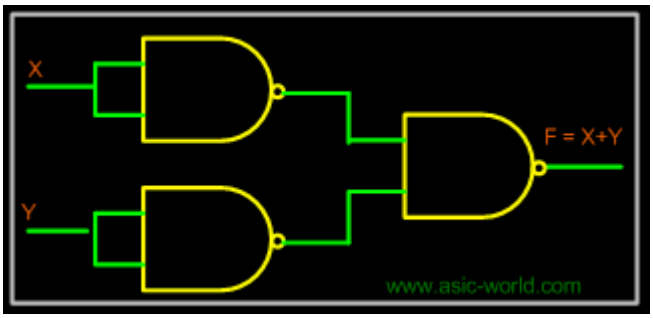
Implementing AND using NANDgates

Input	Output	Rule
((XY)'(XY)')'	=((XY)')'	Idempotent
	=(XY)	Involution



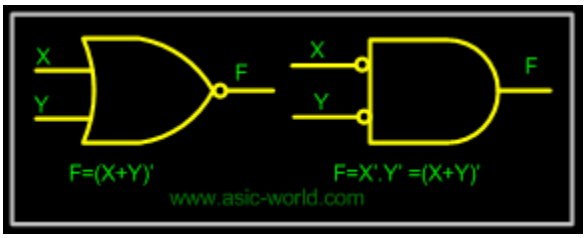
Implementing OR using NANDgates

Input	Output	Rule
((XX)'(YY)')'	=(X'Y')'	Idempotent
	=X''+Y''	DeMorgan
	=X+Y	Involution



Implementing NOR using NANDgates

Input	Output	Rule
((XX)'(YY)')'	=(X'Y')'	Idempotent
	=X''+Y''	DeMorgan
	=X+Y	Involution
	=(X+Y)'	Idempotent



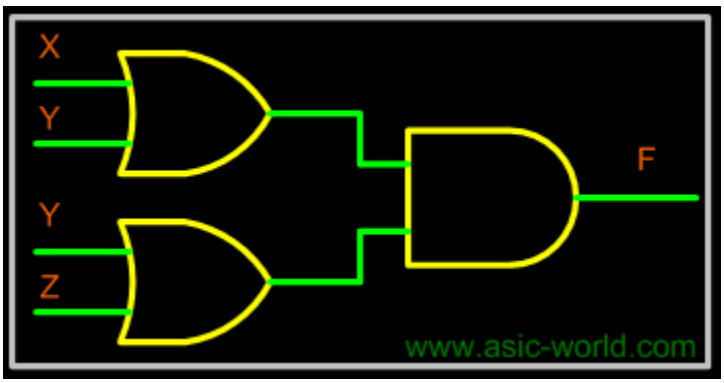
Realization of logic function using NORgates

Any logic function can be implemented using NOR gates. To achieve this, first the logic function has to be written in Product of Sum (POS) form. Once it is converted to POS, then it's very easy to implement using NOR gate. In other words any logic circuit with OR gates in first level and AND gates in second level can be converted into a NOR- NOR gatecircuit.

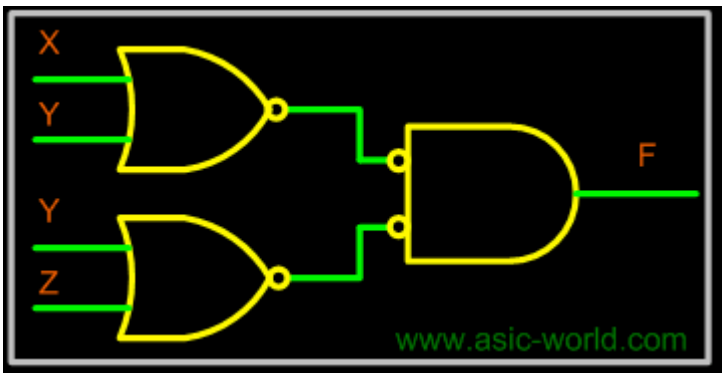
Consider the following POS expression F =

(X+Y) .(Y+Z)

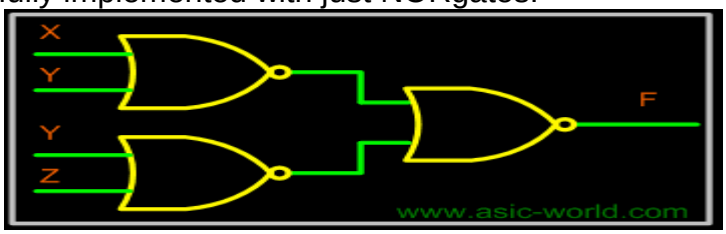
The above expression can be implemented with three OR gates in first stage and one AND gate in second stage as shown infigure.



If bubble are introduced at the output of the OR gates and the inputs of AND gate, the above circuit becomes as shown infigure.



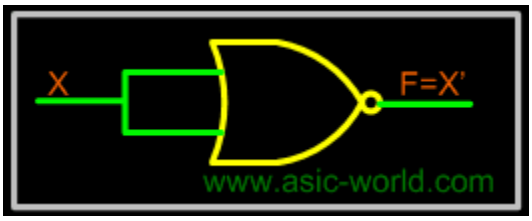
Now replace AND gate with input bubble with the NOR gate. Now we have circuit which is fully implemented with just NORgates.



Realization of logic gates usingNORgates

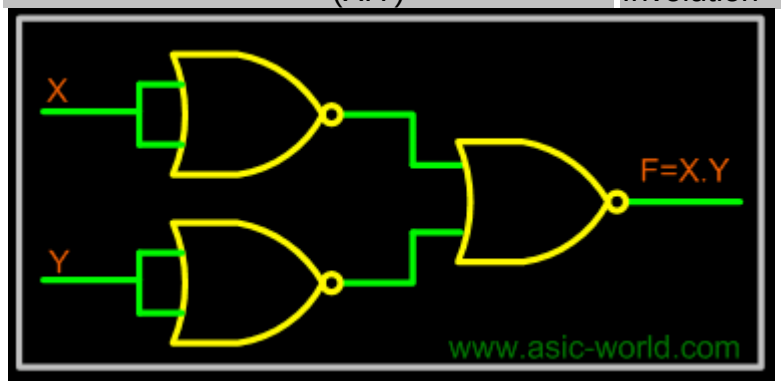
Implementing an inverter using NORgate

Input	Output	Rule
(X+X)'	=X'	Idempotent



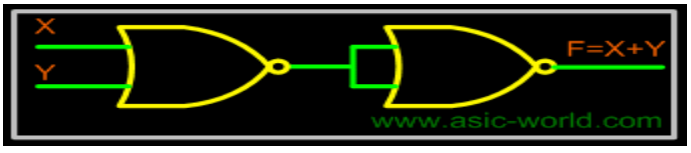
Implementing AND using NORgates

Input	Output	Rule
((X+X)'+(Y+Y)')'	=(X'+Y')'	Idempotent
	=X''.Y''	DeMorgan
	=(X.Y)	Involution



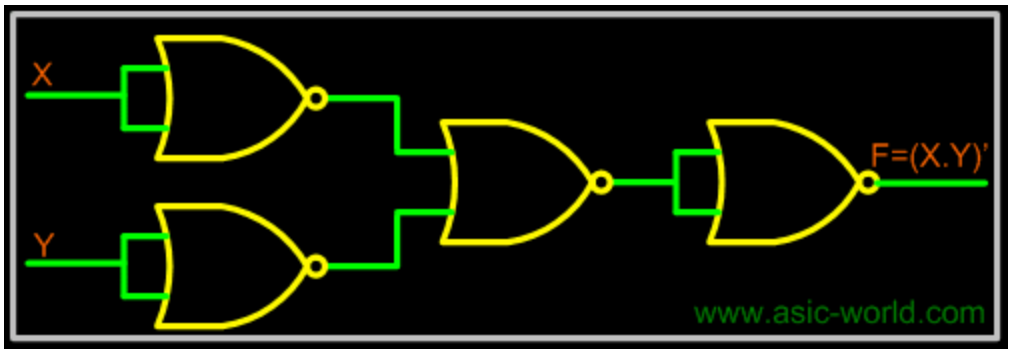
Implementing OR using NORgates

Input	Output	Rule
((X+Y)'+(X+Y)')'	=((X+Y)')'	Idempotent
	=X+Y	Involution



Implementing NAND using NORgates

Input	Output	Rule
((X+Y)'+(X+Y)')'	=((X+Y)')'	Idempotent
	=X+Y	Involution
	=(X+Y)'	Idempotent



CANONICAL AND STANDARDFORMS

A binary variable may appear either in its normal form (*x*) or in its complement form(*x*_).Now consider two binary variables *x* and *y* combined with an AND operation.Sinceeachvariable may appear in either form, there are four possible combinations: *x*_*y*_, *x*_*y*,*xy*_,and *xy*. Each of these four AND terms is called a *minterm*, or a *standard product*.Inasimilar manner, *n* variables can be combined to form 2*n* minterms. The 2*n* different mintermsmay be determined by a method similar to the one shown in Table 2.3 for three variables.Thebinarynumbersfrom0to2*n*-1arelistedunderthe*n*variables.Eachminterm is obtained from an AND term of the *n* variables, with each variable being primed if the corresponding bit of the binary number is a 0 and unprimed if a 1. A symbolfor each minterm is also shown in the table and is of the form *mj*, where the subscript*j*denotes the decimal equivalent of the binary number of the mintermdesignated.In a similar fashion, *n* variables forming an OR term, with each variable being primedor unprimed, provide 2*n* possible combinations, called *maxterms*, or *standard sums*.The eight maxterms for three variables, together with their symbolic designations, are listedin Table 2.3 . Any 2*n* maxterms for *n* variables may be determined similarly. It is importantto note that (1) each maxterm is obtained from an OR term of the *n* variables, with each variable being unprimed if the corresponding bit is a 0 and primed if a 1, and (2)each maxterm is the complement of its corresponding minterm and viceversa.**A Boolean function can be expressed algebraically from a given truth table by forminga minterm for each combination of the variables that produces a 1 in the functionand then taking the OR of all those terms.** For example, the function *f*1 in Tableisdetermined by expressing the combinations 001, 100, and 111 as *x*_*y*_*z*, *xy*_*z*_,and*xyz*,respectively. Since each one of these minterms results in f1 = 1, wehave $f_1 = x\_y\_z + xy\_z\_ + xyz = m_1 + m_4 + m_7$

Minterms and Maxterms for Three Binary Variables

			Minterms		Maxterms	
x	*y*	*z*	Term	Designation	Term	Designation
0	0	0	$x'y'z'$	m_0	$x + y + z$	M_0
0	0	1	$x'y'z$	m_1	$x + y + z'$	M_1
0	1	0	$x'yz'$	m_2	$x + y' + z$	M_2
0	1	1	$x'yz$	m_3	$x + y' + z'$	M_3
1	0	0	$xy'z'$	m_4	$x' + y + z$	M_4
1	0	1	$xy'z$	m_5	$x' + y + z'$	M_5
1	1	0	xyz'	m_6	$x' + y' + z$	M_6
1	1	1	xyz	m_7	$x' + y' + z'$	M_7

ERROR DETECTION ANDCORRECTION

The dynamic physical interaction of the electrical signals affecting the data path of amemory unit may cause occasional errors in storing and retrieving the binary information.The reliability of a memory unit may be improved by employing error-detecting and error-correcting codes. The most common error detection scheme is the parity bit.(See Section 3.9.) A parity bit is generated and stored along with the data word in memory. The parityof the word is checked after reading it from memory. The data wordis accepted if the parity of the bits read out is correct. If the parity checked results in an inversion, an error is detected, but it cannot becorrected.An error-correcting code generates multiple parity check bits that are stored with the data word in memory. Each check bit is a parity over a group of bits in the data word. When the word is read back from memory, the associated parity bits are also read from memory and compared with a new set of check bits generated from the data that have been read. If the check bits are correct, no error has occurred. If the check bits do not match the stored parity, they generate a unique pattern, called a *syndrome*,that can be used to identify the bit that is in error. A single error occurs when a bit changes in value from 1 to 0 or from 0 to 1 during the write or read operation. If thespecific bit in error is identified, then the error can be corrected bycomplementing the erroneousbit.

HammingCode

One of the most common error-correcting codes used in RAMs was devised by R. W.Hamming. In the Hamming code, k parity bits are added to an n -bit data word, forminga new word of $n + k$ bits. The bit positions are numbered in sequence from 1 to $n+k$.Those positions numbered as a power of 2 are reserved for the parity bits. The remainingbits are the data bits. The code can be used with words of any length. Before giving the general characteristics of the code, we will illustrate its operation with a data wordof eightbits.Consider, for example:the 8-bit data word 11000100. We include 4 parity bits with the 8-bit word and arrange the 12 bits asfollows:

Bitposition:	1	2	3	4	5	6	7	8	9	10	11	12
	*P*1	*P*2	1	*P*4	1	0	0	*P*8	0	1	0	0

2, 3, 6, 7, and so on. Comparing these numbers with the bit positions used in generating and checking parity bits in the Hamming code, we note the relationship between the bit groupings in the code and the position of the 1-bits in the binary count sequence. Note that each group of bits starts with a number that is a power of 2: 1, 2, 4, 8, 16, etc. These numbers are also the position numbers for the parity bits.

Single-Error Correction, Double-Error Detection

The Hamming code can detect and correct only a single error. By adding another parity bit to the coded word, the Hamming code can be used to correct a single error and detect double errors. If we include this additional parity bit, then the previous 12-bit coded word becomes 001110010100P_{13}, where P_{13} is evaluated from the exclusive-OR of the other 12 bits. This produces the 13-bit word 0011100101001 (even parity). When the 13-bit word is read from memory, the check bits are evaluated, as is the parity P over the entire 13 bits. If $P = 0$, the parity is correct (even parity), but if $P = 1$, then the parity over the 13 bits is incorrect (odd parity). The following four cases can arise:

If $C = 0$ and $P = 0$, no error occurred.

If $C \neq 0$ and $P = 1$, a single error occurred that can be corrected.

If $C \neq 0$ and $P = 0$, a double error occurred that is detected, but that cannot be corrected.

If $C = 0$ and $P = 1$, an error occurred in the P_{13} bit.

This scheme may detect more than two errors, but is not guaranteed to detect all such errors.

Integrated circuits use a modified Hamming code to generate and check parity bits for single-error correction and double-error detection. The modified Hamming code uses a more efficient parity configuration that balances the number of bits used to calculate the XOR operation. A typical integrated circuit that uses an 8-bit data word and a 5-bit check word is IC type 74637. Other integrated circuits are available for data words of 16 and 32 bits. These circuits can be used in conjunction with a memory unit to correct a single error or detect double errors during write and read operations.

KarnaughMaps

Karnaugh maps provide a systematic method to obtain simplified sum-of-products (SOPs) Boolean expressions. This is a compact way of representing a truth table and is a technique that is used to simplify logic expressions. It is ideally suited for four or less variables, becoming cumbersome for five or more variables. Each square represents either a minterm or maxterm. A K-map of n variables will have 2 squares. For a Boolean expression, product terms are denoted by 1's, while sum terms are denoted by 0's - but 0's are often leftblank.

A K-map consists of a grid of squares, each square representing one canonical minterm combination of the variables or their inverse. The map is arranged so that squares representing minterms which differ by only one variable are adjacent both vertically and horizontally. Therefore XY'Z' would be adjacent to X'Y'Z' and would also adjacent to XY'Z andXYZ'.

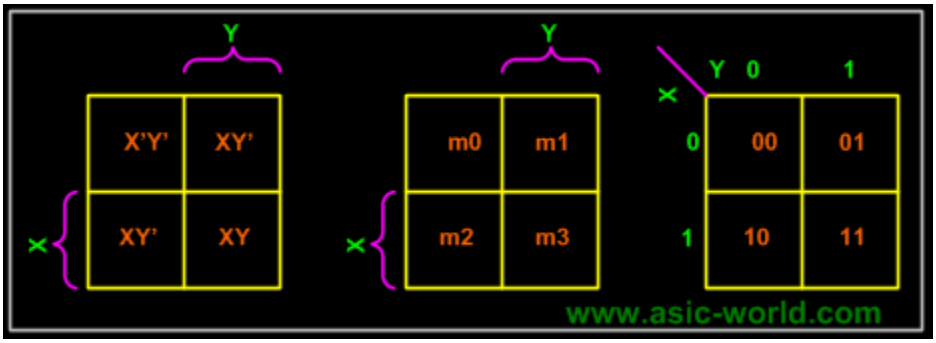
MinimizationTechnique

- Based on the Unifying Theorem: X + X' =1
- The expression to be minimized should generally be in sum-of-product form (If necessary, the conversion process is applied to create the sum-of-productform).
- The function is mapped onto the K-map by marking a 1 in those squares corresponding to the terms in the expression to be simplified (The other squares may be filled with0's).
- Pairs of 1's on the map which are adjacent are combined using the theorem Y(X+X') = Y where Y is any Boolean expression (If two pairs are also adjacent, then these can also be combined using the sametheorem).
- The minimization procedure consists of recognizing those pairs and multiple pairs.
    - These are circled indicating reducedterms.
    - Groups which can be circled are those which have two (2^1) 1's, four (2^2) 1's, eight (2^3) 1's, and soon.
    - Note that because squares on one edge of the map are considered adjacent to those on the opposite edge, group can be formed with these squares.
    - Groups are allowed tooverlap.
- The objective is to cover all the 1's on the map in the fewest number of groups and to create the largest groups to dothis.
- Once all possible groups have been formed, the corresponding terms are identified.
    - A group of two 1's eliminates one variable from the originalminterm.
    - A group of four 1's eliminates two variables from the originalminterm.
    - A group of eight 1's eliminates three variables from the original minterm, and soon.
    - The variables eliminated are those which are different in the original minterms of thegroup.

2-VariableK-Map

In any K-Map, each square represents a minterm. Adjacent squares always differ by just one literal (So that the unifying theorem may apply: X + X' = 1). For the 2-variable case (e.g.:

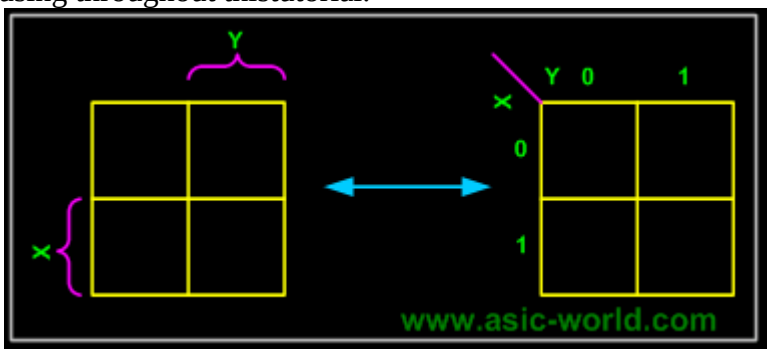
variables X, Y), the map can be drawn as below. Two variable map is the one which has got only two variables asinput.



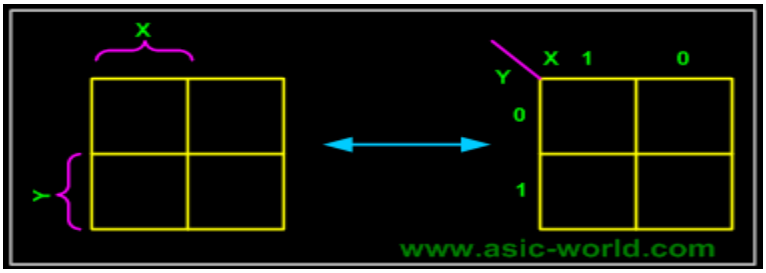
Equivalentlabeling

K- map needs not follow the ordering as shown in the figure above. What this means is that we can change the position of m0, m1, m2, m3 of the above figure as shown in the two figuresbelow.

Position assignment is the same as the default k-maps positions. This is the one which we will be using throughout thistutorial.



This figure is with changed position of m0, m1, m2,m3.



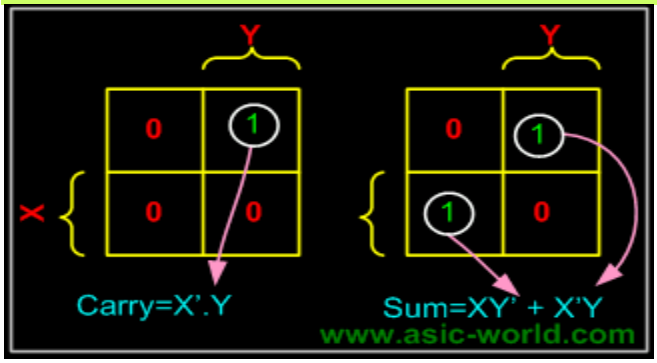
The K-map for a function is specified by putting a '1' in the square corresponding to a minterm, a '0'otherwise.

Example- Carry and Sum of a halfadder

In this example we have the truth table as input, and we have two output functions. Generally we may have n output functions for m input variables. Since we have two output functions, we need

to draw two k-maps (i.e. one for each function). Truth table of 1 bit adder is shown below. Draw the k-map for Carry and Sum as shownbelow.

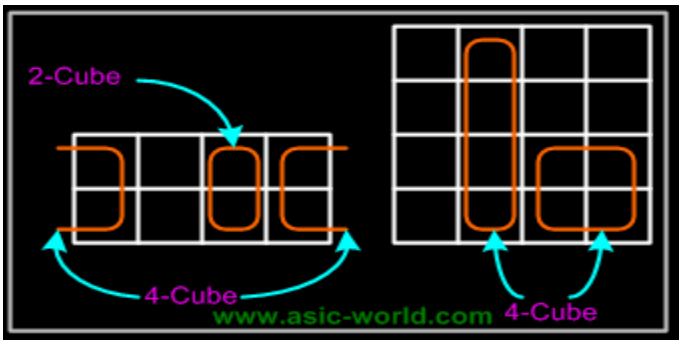
X	Y	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



Grouping/CirclingK-maps

The power of K-maps is in minimizing the terms, K-maps can be minimized with the help of grouping the terms to form single terms. When forming groups of squares, observe/consider thefollowing:

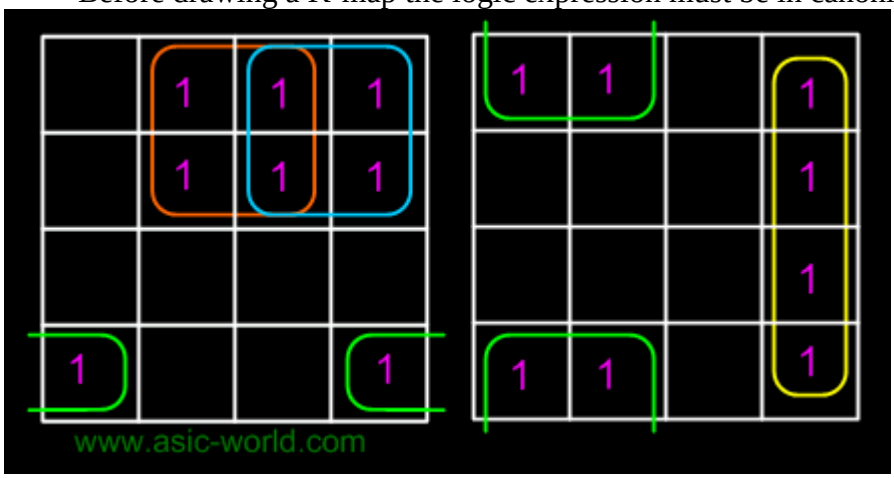
- Every square containing 1 must be considered at leastonce.
- A square containing 1 can be included in as many groups asdesired.
- A group must be as large aspossible.



- If a square containing 1 cannot be placed in a group, then leave it out to include in finalexpression.
- The number of squares in a group must be equal to2
- , i.e.2,4,8,.

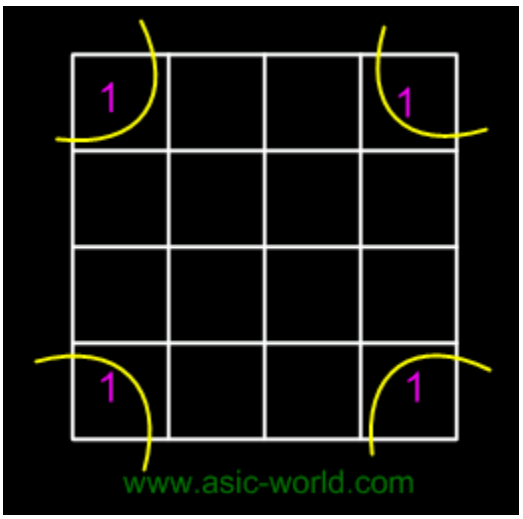
- The map is considered to be folded or spherical, therefore squares at the end of a row or column are treated as adjacentsquares.
- The simplified logic expression obtained from a K-map is not always unique. Groupings can be made in differentways.

- Before drawing a K-map the logic expression must be in canonicalform.



L-

M-



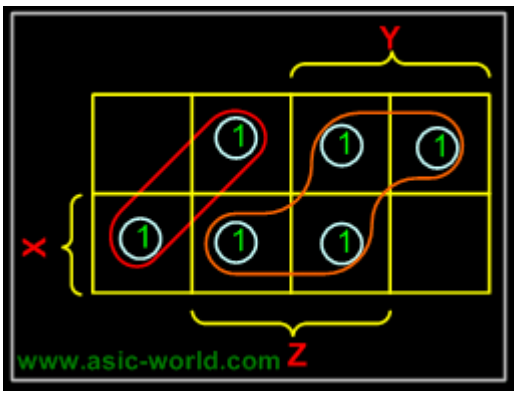
N-

O- In the next few pages we will see some examples ongrouping.

P-

Q- **Example of invalidgroups**

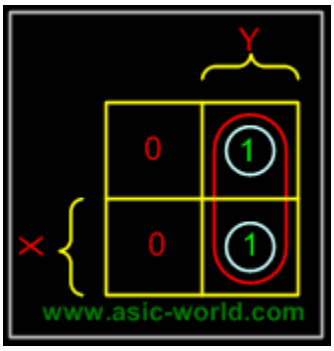
R-



✦ S-**Example -X'Y+XY**

In this example we have the equation as input, and we have one output function. Draw the k-map for function F with marking 1 for X'Y and XY position. Now combine two 1's as shown in figure to form the single term. As you can see X and X' get canceled and only Y remains.

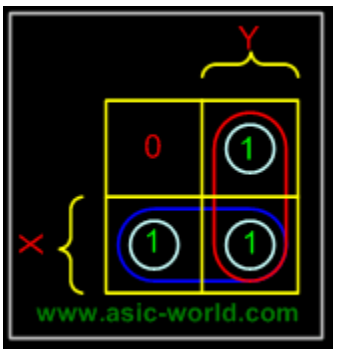
F =Y



Example -X'Y+XY+XY'

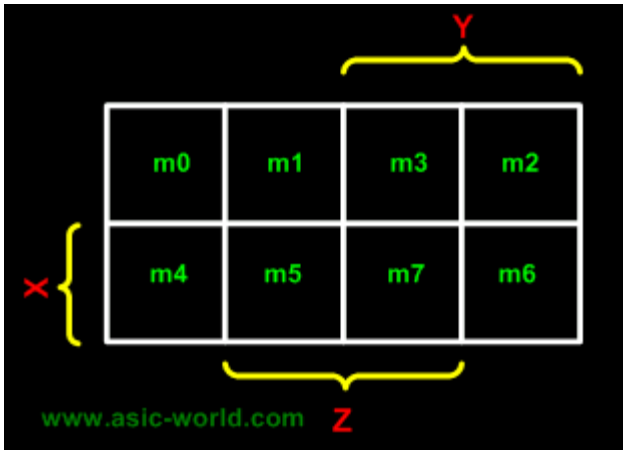
In this example we have the equation as input, and we have one output function. Draw the k-map for function F with marking 1 for X'Y, XY and XY position. Now combine two 1's as shown in figure to form the two singleterms.

F = X + Y



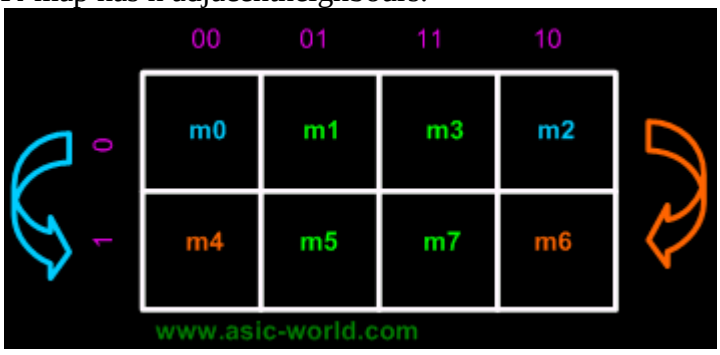
3-VariableK-Map

There are 8 minterms for 3 variables (X, Y, Z). Therefore, there are 8 cells in a 3- variable K-map. One important thing to note is that K-maps follow the gray code sequence, not the binaryone.



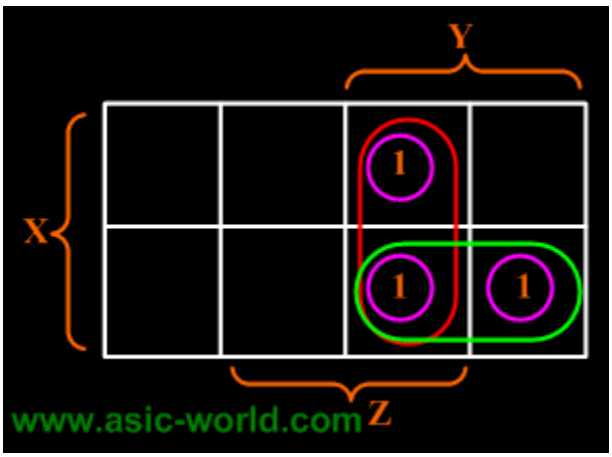
Using gray code arrangement ensures that minterms of adjacent cells differ by only ONE literal. (Other arrangements which satisfy this criterion may also beused.)

Each cell in a 3-variable K-map has 3 adjacent neighbours. In general, each cell in an n- variable K-map has n adjacentneighbours.



Example

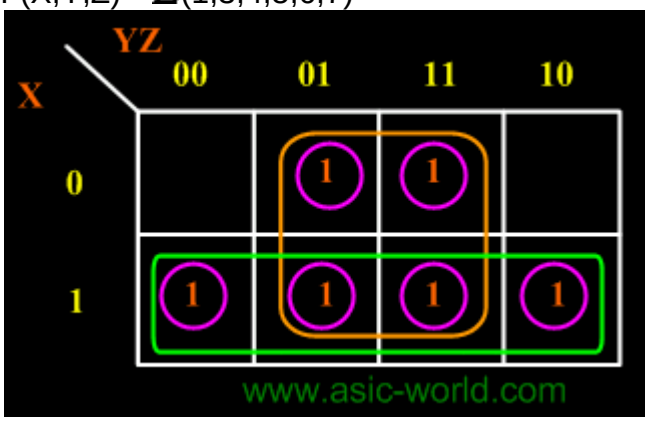
F =XYZ'+XYZ+X'YZ



F = XY +YZ

Example

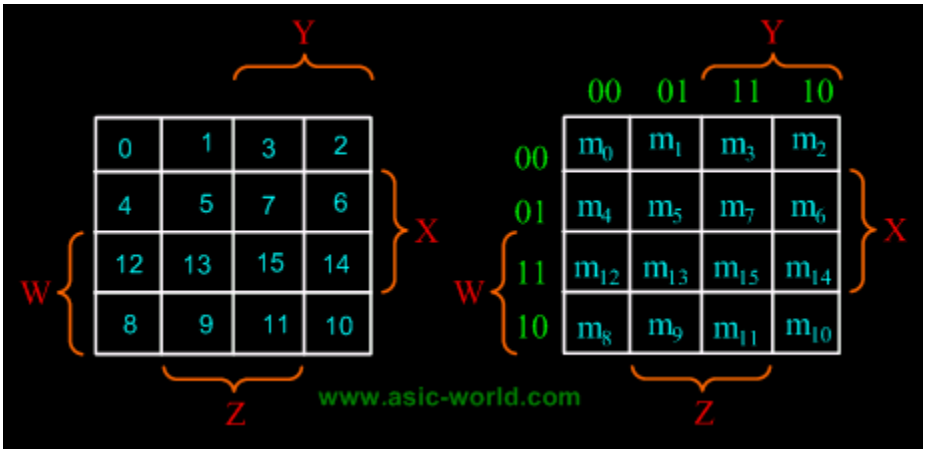
F(X,Y,Z) =Σ(1,3,4,5,6,7)



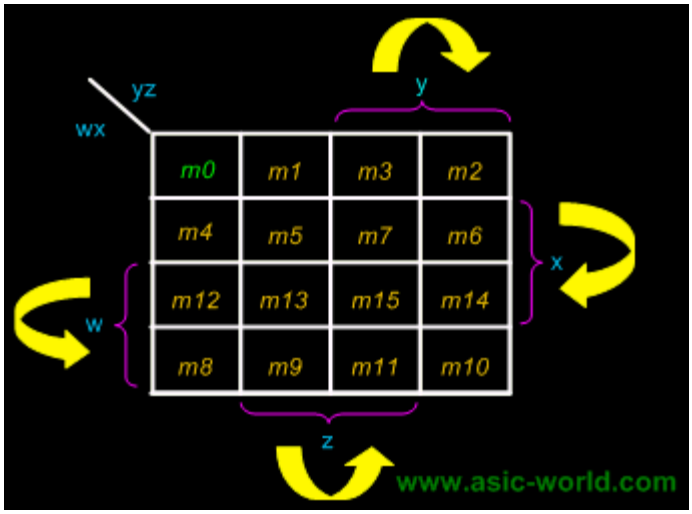
F = X +Z

3-VariableK-Map

There are 16 cells in a 4-variable (W, X, Y, Z); K-map as shown in the figurebelow.

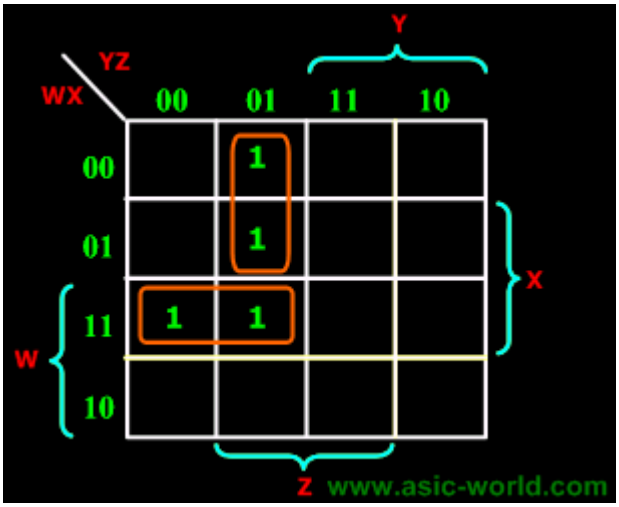


There are 2 wrap-around: a horizontal wrap-around and a vertical wrap-around. Every cell thus has 4 neighbours. For example, the cell corresponding to minterm m0 has neighbours m1, m2, m4 andm8.



Example

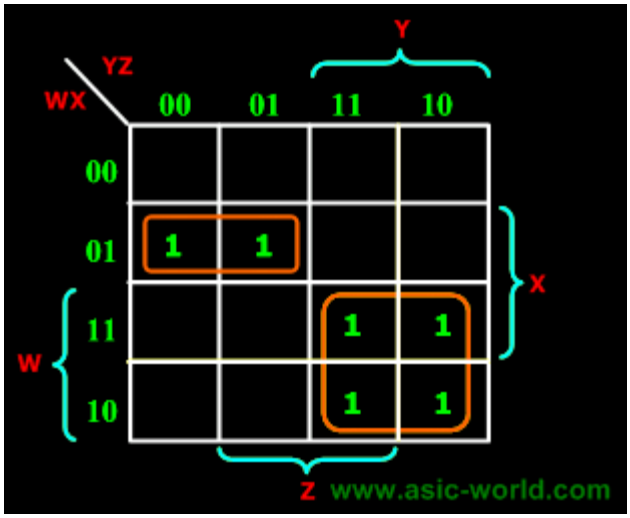
F(W,X,Y,Z) =(1,5,12,13)



F = WY'Z +W'Y'Z

Example

F(W,X,Y,Z) = (4, 5, 10, 11, 14,15)



F = W'XY' +WY

MinimizationTechnique

- The expression is represented in the canonical SOP form if not already in that form.
- The function is converted into numericnotation.
- The numbers are converted into binaryform.
- The minterms are arranged in a column divided intogroups.
- Begin with the minimizationprocedure.
    - Each minterm of one group is compared with each minterm in the group immediatelybelow.
    - Each time a number is found in one group which is the same as a number in the group below except for one digit, the numbers pair is ticked and a new composite iscreated.
    - This composite number has the same number of digits as the numbers in the pair except the digit different which is replaced by an"x".
- The above procedure is repeated on the second column to generate a third column.
- The next step is to identify the essential prime implicants, which can be done using a prime implicantchart.
    - Where a prime implicant covers a minterm, the intersection of the corresponding row and column is marked with across.
    - Those columns with only one cross identify the essential primeimplicants.
      -> These prime implicants must be in the finalanswer.
    - The single crosses on a column are circled and all the crosses on the same row are also circled, indicating that these crosses are covered by the prime implicantsselected.
    - Once one cross on a column is circled, all the crosses on that column can be circled since the minterm is nowcovered.

- If any non-essential prime implicant has all its crosses circled, the prime implicant is redundant and need not be consideredfurther.

- Next, a selection must be made from the remaining nonessential prime implicants, by considering how the non-circled crosses can be coveredbest.
  - One generally would take those prime implicants which cover the greatest number of crosses on theirrow.
  - If all the crosses in one row also occur on another row which includes further crosses, then the latter is said to dominate the former and can be selected.
  - The dominated prime implicant can then bedeleted.

Example

Find the minimal sum of products for theBoolean expression, f=Σ (1,2,3,7,8,9,10,11,14,15), using Quine-McCluskeymethod.

Firstly these minterms are represented in the binary form as shown in the table below. The above binary representations are grouped into a number of sections in terms of the number of 1's as shown in the tablebelow.

Binary representation ofminterms

Minterms	U	V	W	X
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
14	1	1	1	0
15	1	1	1	1

Group of minterms for different number of1's

No of1's	Minterms	U	V	W	X
1	1	0	0	0	1
1	2	0	0	1	0

1	8	1	0	0	0
2	3	0	0	1	1
2	9	1	0	0	1
2	10	1	0	1	0
3	7	0	1	1	1
3	11	1	0	1	1
3	14	1	1	1	0
4	15	1	1	1	1

Any two numbers in these groups which differ from each other by only one variable can be chosen and combined, to get 2-cell combination, as shown in the tablebelow.

2-Cellcombinations

Combinations	U	V	W	X
(1,3)	0	0	-	1
(1,9)	-	0	0	1
(2,3)	0	0	1	-
(2,10)	-	0	1	0
(8,9)	1	0	0	-
(8,10)	1	0	-	0
(3,7)	0	-	1	1
(3,11)	-	0	1	1
(9,11)	1	0	-	1
(10,11)	1	0	1	-
(10,14)	1	-	1	0
(7,15)	-	1	1	1
(11,15)	1	-	1	1
(14,15)	1	1	1	-

From the 2-cell combinations, one variable and dash in the same position can be combined to form 4-cell combinations as shown in the figurebelow.

4-Cellcombinations

Combinations	U	V	W	X
(1,3,9,11)	-	0	-	1
(2,3,10,11)	-	0	1	-

(8,9,10,11)	1	0	-	-
(3,7,11,15)	-	-	1	1
(10,11,14,15)	1	-	1	-

The cells (1,3) and (9,11) form the same 4-cell combination as the cells (1,9) and (3,11). The order in which the cells are placed in a combination does not have any effect. Thus the (1,3,9,11) combination could be written as(1,9,3,11).

From above 4-cell combination table, the prime implicants table can be plotted as shown in tablebelow.

Prime ImplicantsTable

Prime Implicants	1	2	3	7	8	9	10	11	14	15
(1,3,9,11)	X	-	X	-	-	X	-	X	-	-
(2,3,10,11)	-	X	X	-	-	-	X	X	-	-
(8,9,10,11)	-	-	-	-	X	X	X	X	-	-
(3,7,11,15)	-	-	-	-	-	-	X	X	X	X
-	X	X	-	X	X	-	-	-	X	-

The columns having only one cross mark correspond to essential prime implicants. A yellow cross is used against every essential prime implicant. The prime implicants sum gives the function in its minimal SOPform.

Y = V'X + V'W + UV' + WX +UW

AlgebraicManipulation

Minterms andMaxterms

Any boolean expression may be expressed in terms of either minterms or maxterms. To do this we must first define the concept of a literal. A literal is a single variable within a term which may or may not be complemented. For an expression with N variables, minterms and maxterms are defined as follows:

- A minterm is the product of N distinct literals where each literal occurs exactly once.
- A maxterm is the sum of N distinct literals where each literal occurs exactlyonce

5-For a two-variable expression, the minterms and maxterms are asfollows

6-

X	Y	Minterm	Maxterm

0	0	X'.Y'	X+Y
0	1	X'.Y	X+Y'
1	0	X.Y'	X'+Y
1	1	X.Y	X'+Y'

7-
8-
9-For a three-variable expression, the minterms and maxterms are asfollows
10-

X	Y	Z	Minterm	Maxterm
0	0	0	X'.Y'.Z'	X+Y+Z
0	0	1	X'.Y'.Z	X+Y+Z'
0	1	0	X'.Y.Z'	X+Y'+Z
0	1	1	X'.Y.Z	X+Y'+Z'
1	0	0	X.Y'.Z'	X'+Y+Z
1	0	1	X.Y'.Z	X'+Y+Z'
1	1	0	X.Y.Z'	X'+Y'+Z
1	1	1	X.Y.Z	X'+Y'+Z'

11-
12-
13- This allows us to represent expressions in either Sum of Products or Product of Sums forms
14-
15- **Sum Of Products(SOP)**
16- The Sum of Products form represents an expression as a sum ofminterms.
17-
18- **$F(X, Y, ...) = Sum(a_k.m_k)$**
19-
20- where a_k is 0 or 1 and m_k is aminterm.
21-
22- To derive the Sum of Products form from a truth table, OR together all of the minterms which give a value of1.
23-
24- **Example -SOP**
25-
26- Consider the truthtable
27-

X	Y	F	Minterm
0	0	0	X'.Y'

0	1	0	X'Y
1	0	1	X.Y'
1	1	1	X.Y

Here SOP is f(X.Y) = X.Y' +X.Y

Product Of Sum(POS)

The Product of Sums form represents an expression as a product ofmaxterms.

F(X, Y, ...) = Product ($b_k + M_k$), where b_k is 0 or 1 and M_k is amaxterm.

To derive the Product of Sums form from a truth table, AND together all of themaxterms which give a value of0.

Example -POS

Consider the truth table from the previousexample.

X	Y	F	Maxterm
0	0	1	X+Y
0	1	0	X+Y'
1	0	1	X'+Y
1	1	1	X'+Y'

Here POS is F(X,Y) =(X+Y')

Exercise

Give the expression represented by the following truth table in both Sum of Products and Product of Sumsforms.

X	Y	Z	F(X,Y,X)
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0

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

Conversion between POS andSOP

Conversion between the two forms is done by application of DeMorgansLaws.

Simplification

As with any other form of algebra you have encountered, simplification of expressions can be performed with Booleanalgebra.

Example

Show that X.Y.Z' + X'.Y.Z' + Y.Z = Y

X.Y.Z' + X'.Y.Z' + Y.Z = Y.Z' + Y.Z =Y

Example

Show that (X.Y' + Z).(X + Y).Z = X.Z +Y.Z

(X.Y' + Z).(X +Y).Z
= (X.Y' + Z.X +Y'.Z).Z
= X.Y'Z + Z.X +Y'.Z
= Z.(X.Y' + X +Y')
=Z.(X+Y')

DON'T-CARECONDITIONS

The logical sum of the minterms associated with a Boolean function specifies the conditions under which the function is equal to 1. The function is equal to 0 for the restof the minterms. This pair of conditions assumes that all the combinations of the values

for the variables of the function are valid. In practice, in some applications the function
is not specified for certain combinations of the variables. As an example, the four-bit
binary code for the decimal digits has six combinations that are not used and consequently
are considered to be unspecified. Functions that have unspecified outputsfor some input combinations are called *incompletely specified functions* . In most applications,

we simply don"t care what value is assumed by the function for the unspecified minterms. For this reason, it is customary to call the unspecified minterms of a function

don't-care conditions . These don"t-care conditions can be used on a map to provide

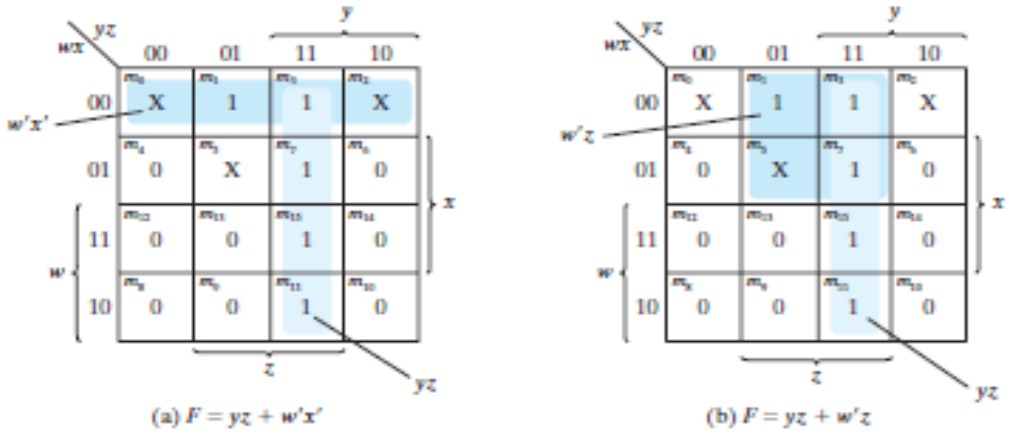
further simplification of the Booleanexpression.

A don"t-care minterm is a combination of variables whose logical value is not specified.
Such a minterm cannot be marked with a 1 in the map, because it would require that the function always be a 1 for such a combination. Likewise, putting a 0 on the square requires the function to be 0. To distinguish the don"t-care condition from 1"sand
0"s, an X is used. Thus, an X inside a square in the map indicates that we don"t care
whether the value of 0 or 1 is assigned to *F* for the particularminterm.
In choosing adjacent squares to simplify the function in a map, the don"t-care minterms may be assumed to be either 0 or 1. When simplifying the function, we can choose to include each don"t-care minterm with either the 1"s or the 0"s, depending on which combination gives the simplestexpression.

Simplify the Booleanfunction

$F(w, x, y, z) = \sum(1, 3, 7, 11, 15)$ which has the don"t-care conditions $d(w, x, y, z) = \sum(0, 2, 5)$.
The min terms of *F* are the variable combinations that make the function equal to 1. The Min terms of *d* are the don"t-care min terms that may be assigned either 0or 1. The map simplification is shown in Fig. 3.15 . The min terms of *F* are marked by 1"s, those of *d* are marked by X"s, and the remainingsquares are filled with 0"s.
To get the simplified expression in sum-of-products form, we must include allfive

1"s in the map, but we may or may not include any of the X"s, depending on the way the function is simplified. The term yz *covers the four min terms in the third column. The remaining min term,* m*1, can becombined.*



with min term *m*3 to give the three-literal term *w_x_z*. However, by including one or two adjacent X‟s we can combine four adjacent squares to give a two-literal term. In Fig. 3.15(a), don‟t-care min terms 0 and 2 are included with the 1‟s, resulting in the simplifiedfunction
*F = yz + w_x_* In Fig. 3.15(b), don‟t-care minterm 5 is included with the 1‟s, and the simplified function is now *F = yz +w_z*
Either one of the preceding two expressions satisfies the conditions stated for this example.

The previous example has shown that the don‟t-care minterms in the map are initially
marked with X‟s and are considered as being either 0 or 1. The choice between 0 and 1 is made depending on the way the incompletely specified function is simplified.
Once the choice is made, the simplified function obtained will consist of a sum of minterms
that includes those minterms which were initially unspecified and have been chosen to be included with the 1‟s. Consider the two simplified expressions
in Example 3.8 : $F(w, x, y, z) = yz + w\_x\_ = \_(0, 1, 2, 3, 7, 11, 15)$
$F(w, x, y, z) = yz + w\_z = \_(1, 3, 5, 7, 11, 15)$

Both expressions include minterms 1, 3, 7, 11, and 15 that make the function F equal to 1. The don‟t-care minterms 0, 2, and 5 are treated differently in each expression. The first expression includes minterms 0 and 2 with the 1‟s and leaves minterm5withthe0‟s.Thesecondexpressionincludesminterm5withthe1‟sand leaves minterms 0 and 2 with the 0‟s. The two expressions represent two functions that are not algebraically equal. Both cover the specified minterms of the function, but each covers different don‟t-care minterms. As far as the incompletely specified function is concerned, either expression is acceptable because the only difference is in the value of F for the don‟t-care minterms. It is also possible to obtain a simplified product-of-sums expression for the function of Fig. 3.15 . In this case, the only way to combine the 0‟s is to include don‟t-care minterms 0 and 2 with the 0‟s to give a simplified complemented function: F_ = z_ + wy_ Taking the complement of F_ gives the simplified expression in product-of-sumsform:

F (w, x, y, z) = z(w_ + y) = _(1, 3, 5, 7, 11,15)

Inthiscase,weincludeminterms0and2withthe0‟sandminterm5withthe1‟s.

3.6 NAND AND NOR IMPLEMENTATION

Digital circuits are frequently constructed with NAND or NOR gates rather than with AND and OR gates. NAND and NOR gates are easier to fabricate with electronic components and are the basic gates used in all IC digital logic families. Because of the prominence of NAND and NOR gates in the design of digital circuits, rules and procedures have been developed for the conversion from Boolean functions given in terms of AND, OR, and NOT into equivalent NAND and NOR logic diagrams.

NAND Circuits

The NAND gate is said to be a *universal* gate because any logic circuit can be implemented with it. To show that any Boolean function can be implemented with NAND gates, we need only show that the logical operations of AND, OR, and complement can be obtained with NAND gates alone. This is indeed shown in Fig. 3.16. The complement operation is obtained from a one-input NAND gate that behaves exactly like an inverter. The AND operation requires two NAND gates. The first produces the NAND operation and the second inverts the logical sense of the signal. The OR operation is achieved through a NAND gate with additional inverters in each input.

A convenient way to implement a Boolean function with NAND gates is to obtain the simplified Boolean function in terms of Boolean operators and then convert the function to NAND logic. The conversion of an algebraic expression from AND, OR, and complement to NAND can be done by simple circuit manipulation techniques that change AND–OR diagrams to NAND diagrams.

To facilitate the conversion to NAND logic, it is convenient to define an alternative graphic symbol for the gate. Two equivalent graphic symbols for the NAND gate are shown in Fig. 3.17. The AND-invert symbol has been defined previously and consists

Two-Level Implementation

The implementation of Boolean functions with NAND gates requires that the functions be in sum-of-products form. To see the relationship between a sum-of-products expression and its equivalent NAND implementation, consider the logic diagrams drawn in Fig. 3.18. All three diagrams are equivalent and implement the function

$$F = AB + CD$$

The function is implemented in Fig. 3.18(a) with AND and OR gates. In Fig. 3.18(b), the AND gates are replaced by NAND gates and the OR gate is replaced by a NAND gate with an OR-invert graphic symbol. Remember that a bubble denotes complementation and two bubbles along the same line represent double complementation, so both can be removed. Removing the bubbles on the gates of (b) produces the circuit of (a). Therefore, the two diagrams implement the same function and are equivalent.

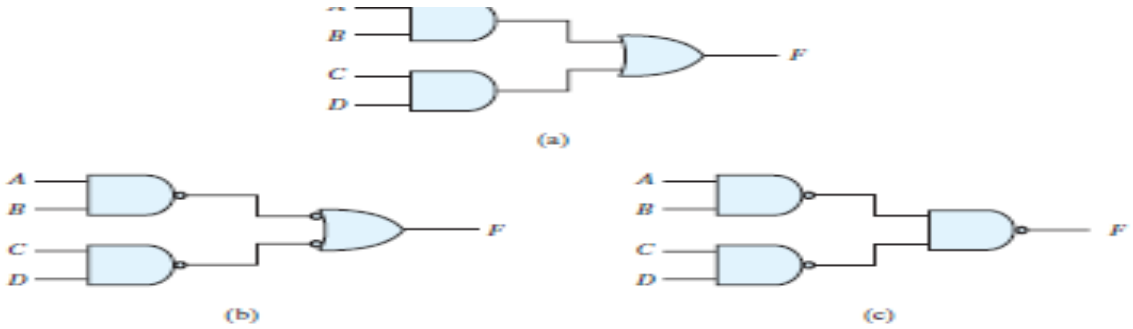


FIGURE 3.18
Three ways to implement $F = AB + CD$

In Fig. 3.18(c), the output NAND gate is redrawn with the AND-invert graphic symbol. In drawing NAND logic diagrams, the circuit shown in either Fig. 3.18(b) or (c) is acceptable. The one in Fig. 3.18(b) is in mixed notation and represents a more direct relationship to the Boolean expression it implements. The NAND implementation in Fig. 3.18(c) can be verified algebraically. The function it implements can easily be converted to sum-of-products form by DeMorgan's theorem:

$$F = ((AB)'(CD)')' = AB + CD$$

EXAMPLE 3.9

Implement the following Boolean function with NAND gates:

$$F(x, y, z) = (1, 2, 3, 4, 5, 7)$$

The first step is to simplify the function into sum-of-products form. This is done by means of the map of Fig. 3.19(a), from which the simplified function is obtained:

$$F = xy' + x'y + z$$

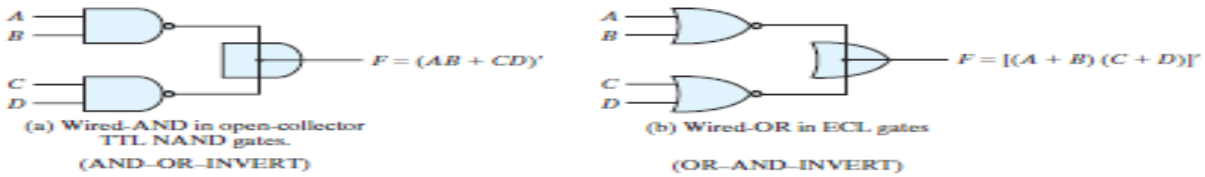
The two-level NAND implementation is shown in Fig. 3.19(b) in mixed notation. Note that input z must have a one-input NAND gate (an inverter) to compensate for the bubble in the second-level gate. An alternative way of drawing the logic diagram is given in Fig. 3.19(c). Here, all the NAND gates are drawn with the same graphic symbol. The inverter with input z has been removed, but the input variable is complemented and denoted by z'.

OTHER TWO-LEVEL IMPLEMENTATIONS

The types of gates most often found in integrated circuits are NAND and NOR gates. For this reason, NAND and NOR logic implementations are the most important from a practical point of view. Some (but not all) NAND or NOR gates allow the possibility of a wire connection between the outputs of two gates to provide a specific logic function. This type of logic is called *wired logic*. For example, open-collector TTL NAND gates, when tied together, perform wired-AND logic. The wired-AND logic performed with two NAND gates is depicted in Fig. 3.26(a). The AND gate is drawn with the lines going through the center of the gate to distinguish it from a conventional gate. The wired-AND gate is not a physical gate, but only a symbol to designate the function obtained from the indicated wired connection. The logic function implemented by the circuit of Fig. 3.26(a) is

$$F = (AB)' \cdots (CD)' = (AB + CD)' = (A' + B')(C' + D')$$

and is called an AND–OR–INVERT function.



(a) Wired-AND in open-collector TTL NAND gates. (AND–OR–INVERT)

(b) Wired-OR in ECL gates (OR–AND–INVERT)

FIGURE 3.26
Wired logic
(a) Wired-AND logic with two NAND gates
(b) Wired-OR in emitter-coupled logic (ECL) gates

Similarly, the NOR outputs of ECL gates can be tied together to perform a wired-OR function. The logic function implemented by the circuit of Fig. 3.26(b) is

$$F = (A + B)' + (C + D)' = [(A + B)(C + D)]'$$

and is called an OR–AND–INVERT function.

A wired-logic gate does not produce a physical second-level gate, since it is just a wire connection. Nevertheless, for discussion purposes, we will consider the circuits of Fig. 3.26 as two-level implementations. The first level consists of NAND (or NOR) gates and the second level has a single AND (or OR) gate. The wired connection in the graphic symbol will be omitted in subsequent discussions.

Nondegenerate Forms

It will be instructive from a theoretical point of view to find out how many two-level combinations of gates are possible. We consider four types of gates: AND, OR, NAND, and NOR. If we assign one type of gate for the first level and one type for the second level, we find that there are 16 possible combinations of two-level forms. (The same type of gate can be in the first and second levels, as in a NAND–NAND implementation.) Eight of these combinations are said to be *degenerate* forms because they degenerate to a single operation. This can be seen from a circuit with AND gates in the first level and an AND gate in the second level. The output of the circuit is merely the AND function of all input variables. The remaining eight *nondegenerate* forms produce an implementation in sum-of-products form or product-of-sums form. The eight *nondegenerate* forms are as follows:

AND–OR	OR–AND
NAND–NAND	NOR–NOR
NOR–OR	NAND–AND
OR–NAND	AND–NOR

(a) OR–NAND (b) OR–NAND (c) NOR–OR

FIGURE 3.28
OR–AND–INVERT circuits, $F = [(A + B)(C + D)E]'$

Implementation with Other Two-Level Forms

Equivalent Nondegenerate Form		Implements the Function	Simplify F' into	To Get an Output of
(a)	(b)*			
AND–NOR	NAND–AND	AND–OR–INVERT	Sum-of-products form by combining 0's in the map.	F
OR–NAND	NOR–OR	OR–AND–INVERT	Product-of-sums form by combining 1's in the map and then complementing.	F

*Form (b) requires an inverter for a single literal term.

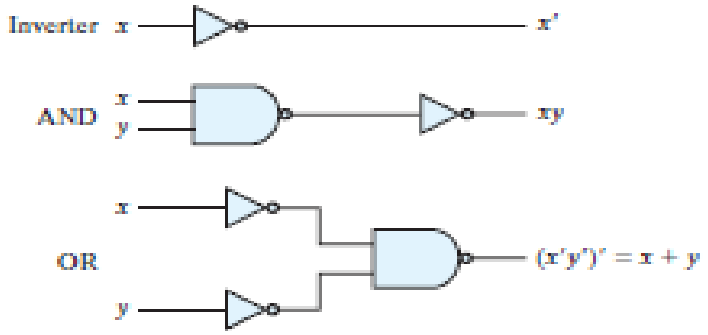


FIGURE 3.16
Logic operations with NAND gates

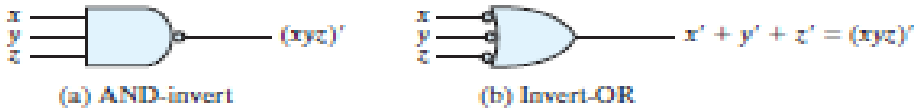


FIGURE 3.17
Two graphic symbols for a three-input NAND gate

of an AND graphic symbol followed by a small circle negation indicator referred to as a bubble. Alternatively, it is possible to represent a NAND gate by an OR graphic symbol that is preceded by a bubble in each input. The invert-OR symbol for the NAND gate follows DeMorgan's theorem and the convention that the negation indicator (bubble) denotes complementation. The two graphic symbols' representations are useful in the analysis and design of NAND circuits. When both symbols are mixed in the same diagram, the circuit is said to be in mixed notation.