PHY307
Lecture-1

Department of Physics

Shiv Nadar University, Gautam Buddha Nagar

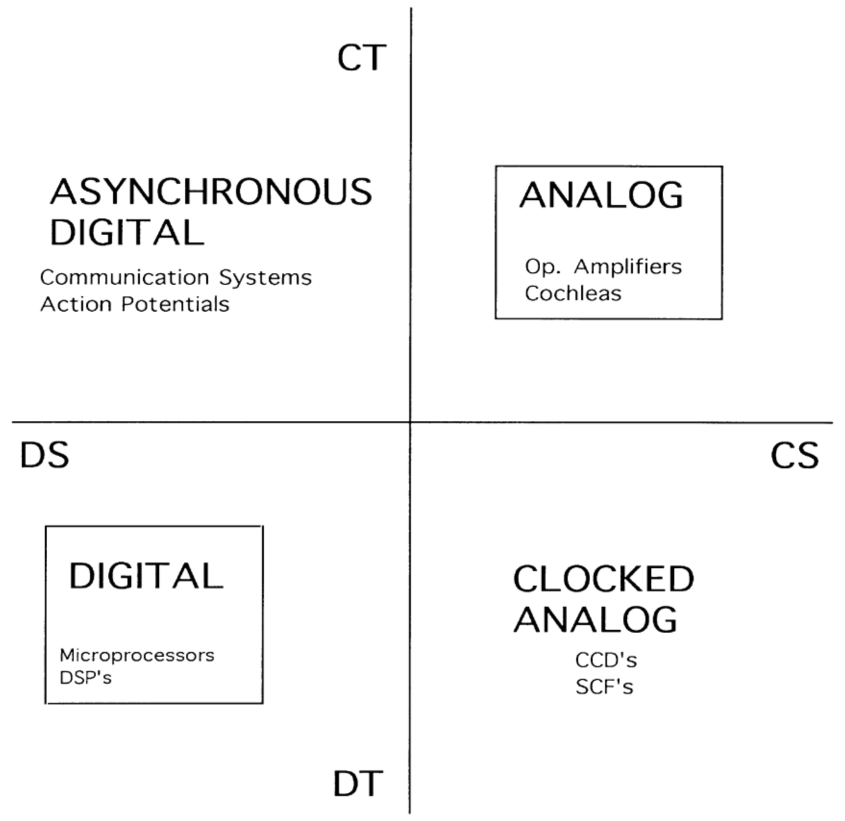
Analog electronics, voltage is a continuous variable. This is useful because most physical quantities we encounter in Physics are continuous: sound levels, light intensity, temperature, pressure, etc.

Digital electronics, is characterized by only two distinguishable voltages. Various names of two states: on/off, true/false, high/low, and 1/0.

These two states are defined by the circuit voltage being above or below a certain value. For example, in TTL logic circuits, a high state corresponds to a voltage above 2.0 V, while a low state is defined as a voltage below 0.8 V.

Analog *vs* Digital:

Electronic systems operate with continuous signals (CS) or discrete signals (DS), and in continuous time (CT) or discrete time (DT). Thus, there are four classes of systems: CSCT, CSDT, DSCT, DSDT (Hosticka, 1985).



The four types of systems, shown in figure, depicts the electronic and biological systems that operate with continuous or discrete signals (CS Or DS) and in continuous or discrete time (CT or DT).

*SCF stands for switched capacitor filter; CCD stands for charge coupled device.

Analog systems that are continuous in both the signal and time domains (CSCT), and digital systems that are discrete in both the signal and time domains (DSDT) have been boxed in the figure.

Comparison: Analog vs Digital

Sr. #	Analog	Digital
1.	Compute with continuous values of physical variables in some range, typically voltages between the lower and upper power-supply voltages.	Compute with discrete values of physical variables, typically the lower and upper power supply voltages, denoted by 0 and 1, respectively.
2.	Primitives of computation arise from the physics of the computing devices: physical relations of transistors, capacitors, resistors, floating-gate devices, Kirchoff's current and voltage laws and so forth. The use of these primitives is an art form and does not lend itself easily to automation. The amount of computation squeezed out of a single transistor is high.	Primitives of computation arise from the mathematics of Boolean logic: logical relations like AND, OR,NOT, NAND, and XOR. The use of these primitives is a science and lends itself easily to automation. The transistor is used as a switch, and the amount of computation squeezed out of a single transistor is low.
3.	One wire represents many bits of information at a given time.	One wire represents 1 bit of information at a given time.
4.	Computation is offset prone since it is sensitive to mismatches in the parameters of the physical devices. The degradation in performance is graceful.	Computation is not offset prone since it is insensitive to mismatches in the parameters of the physical devices. However, a single bit error can result in catastrophic failure.
5.	Noise is due to thermal fluctuations in physical devices.	Noise is due to round-off error.
6.	Signal is not restored at each stage of the computation.	Signal is restored to 1/0 at each stage of the computation.
7.	In a cascade of analog stages, noise starts to accumulate. Thus, complex systems with many stages are difficult to build.	Round-off error does not accumulate significantly for many computations. Thus, complex systems with many stages are easy to build.

…exploiting physics to do computation can be powerful.

The advantage of an analog machine over a digital machine is especially great when there is a straightforward mapping between the operations needed in the computation and the primitives of the technology.

For large-scale systems, as in the implementation of silicon cochlea (Sarpeshkar, Lyon, & Mead, 1998), depending on the nature of the digital implementation, the advantage can range from a factor of 300 to 10^5 in power consumption.

Because the number of devices required to perform a computation is greater in digital systems, there is more wiring and communication overhead.

The presence of more devices and more communication overhead causes digital circuits to have typically higher area consumption than that of analog circuits. The switching energy dissipation due to the large number of devices and the communication overhead also causes the power consumption to be higher in digital circuits.

Unlike digital CMOS circuits, whose power dissipation occurs only during switching and is entirely dynamic, many analog circuits have standby or static power dissipation and little or no dynamic power dissipation.

The inefficiency of ignoring the physical computational primitives inherent in the technology and the inefficiency of encoding only 1 bit per wire is always present in digital computation. Consequently, analog computation still retains its advantage.

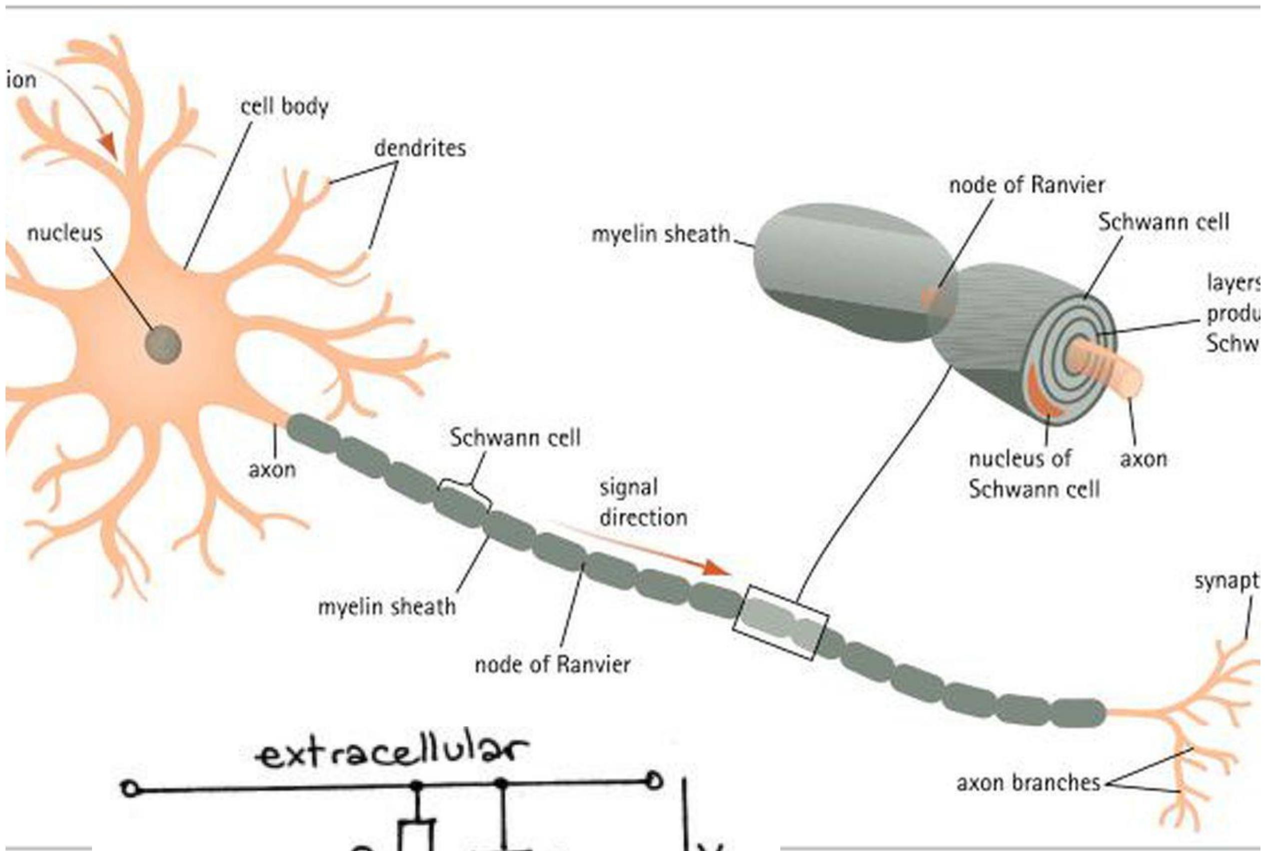
The human brain performs on the order of 3.6×10^{15} synaptic operations per second (Sarpeshkar, 1997).

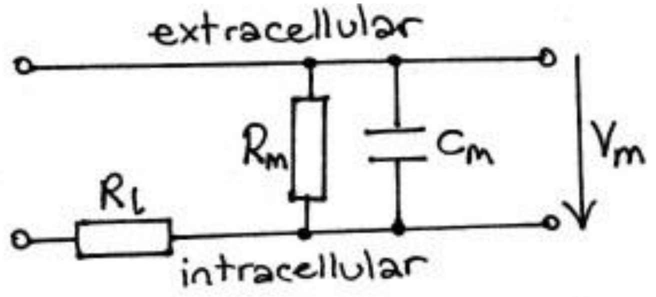
* From measurements of cerebral blood flow and oxygen consumption, it is known that the brain consumes only **12W**.
* Its efficiency of computation is thus about 3×10^{14} operations per joule. The human brain is capable of doing tremendously complex computation in real time despite the slow and noisy components in our heads and bodies.

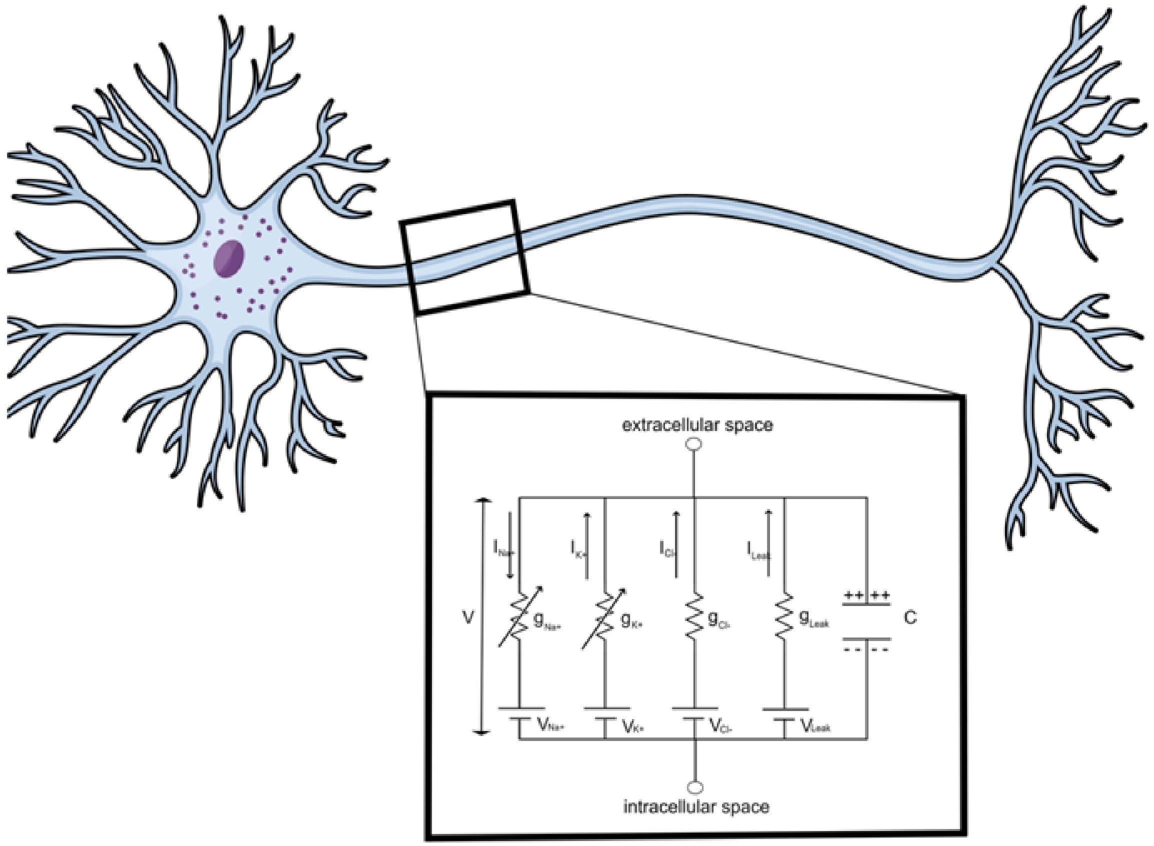
An extremely fast microprocessor such as the DEC Alpha 21164 performs about 255×10^{6} floating-point operations per second and consumes 40W.

Its efficiency is thus about 6.25×10^{6} operations per joule. It is incapable of solving even relatively simple behavioral tasks in real time in spite of its blazingly fast and precise transistors.

Circuit as models of complex systems: Myelinated Neurons



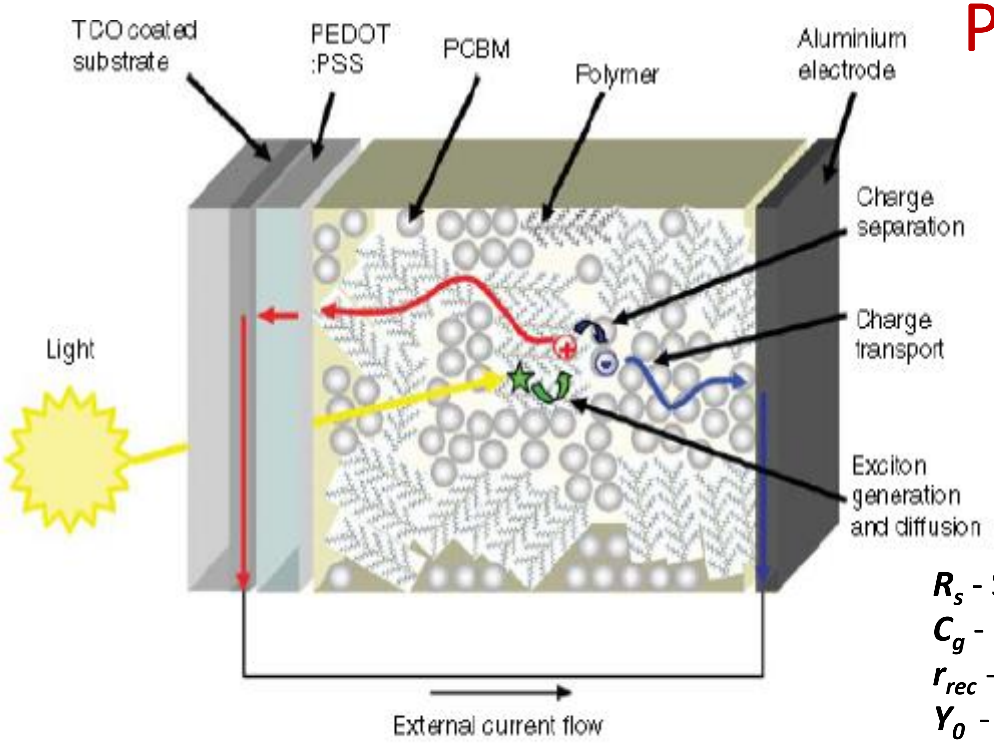


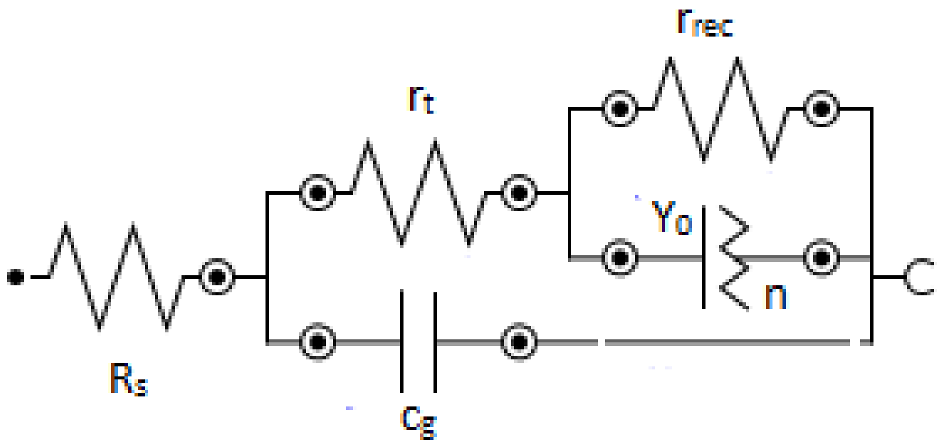


•R_l: longitudinal resistance per unit length,
•R_m: membrane resistance per unit length,
•C_m: membrane capacitance per unit length.

Elementary formulae for the circumference and area of a circle mean that:
•R_l α πr^2 (fatter axons have lower longitudinal resistance),
•R_m is inversely proportional to $1/2\pi r$ (fatter axons have a larger surface area to leak charge),
•C_p α $2\pi r$ (fatter axons have a larger surface area to hold charge),
where r is the radius of the axon.

Physics of Bulk heterojunction Organic solar cells:

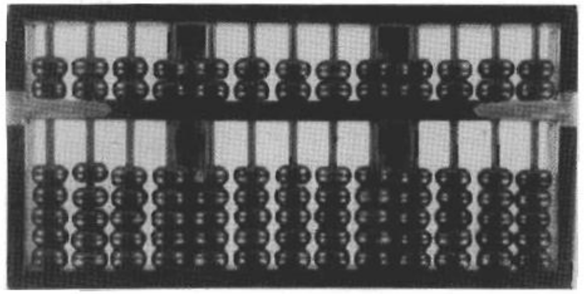


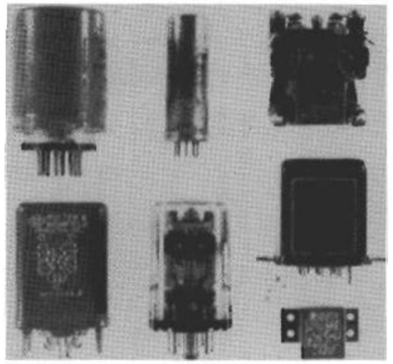


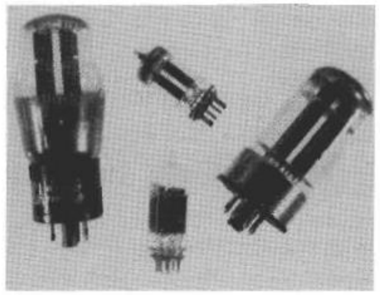
R_s - Series resistance from contact layers and electrodes; ***r_t*** - Transport resistance
C_g - Device capacitance associated with the depletion region within the active layer
r_{rec} - Recombination resistance r_{rec} in parallel with a constant phase element (CPE)
Y_0 - The coefficient of the CPE; ***n*** - An "ideality" factor characteristic of the distribution of relaxation times;
C_n - Stored excess minority carriers gives rise to distributed chemical capacitances

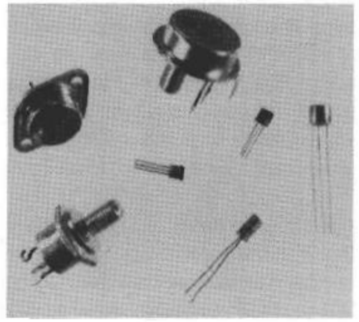
Extraction of characteristic parameters associated with various physical mechanisms in devices

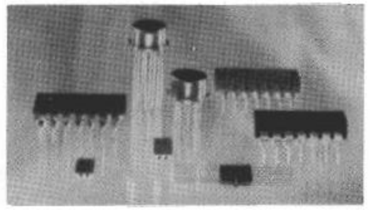
Transit time :	$\tau_d = r_t c_n$	Electron mobility:	$\mu_n = \frac{eD_n}{k_B T}$
Effective lifetime :	$\tau_n = r_{rec} c_n$	Impedance :	$Z = Y_0^{-1}(j\omega)^{-n}$
Electron diffusivity:	$D_n = \frac{L^2}{\tau_d}$	Chemical Potential :	$C_\mu = \frac{(Y_0 r_{rec})^{\frac{1}{n}}}{r_{rec}}$

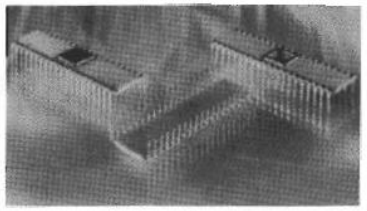

(a)

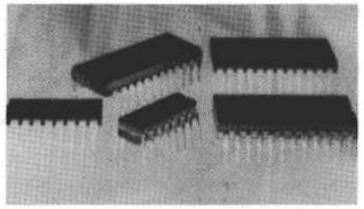

(b)


(c)


(d)


(e)


(g)


(f)

Abacus

Relays

Electron tubes

Transistors

Small scale IC

Medium scale IC

Large scale IC

Basic Digital Concepts:

- Converting continuous analog signals into a finite number of discrete states, a process called digitization.
- The extent the states are sufficiently well separated so that noise does not create errors.
- The resulting digital signals allow the following (slightly idealized):
    - storage over arbitrary periods of time
    - flawless retrieval and reproduction of the stored information
    - flawless transmission of the information

Some information is intrinsically digital, so it is natural to process and manipulate it using purely digital techniques.

Examples: Numbers and Words.

The drawback to digitization:

A single analog signal (e.g. a voltage which is a function of time, like a stereo signal) needs many discrete states, or bits, in order to give a satisfactory reproduction. For example, it requires a minimum of 10 bits to determine a voltage at any given time to an accuracy of ≈ 0.1%.

For transmission, one requires 10 lines instead of the one original analog line. The explosion in digital techniques and technology has been made possible by the incredible increase in the density of digital circuitry, its <u>robust performance</u>, its relatively <u>low cost</u>, and its <u>speed</u>.

The requirement of using many bits in reproduction is no longer an issue: The more the better. This circuitry is based upon the transistor, which can be operated as a switch with two states. Hence, the digital information is intrinsically binary. So in practice, the terms digital and binary are used interchangeably.

Decimal and Binary number systems:

The decimal number system is said to be of *base,* or *radix,* 10 because it uses 10 digits and the coefficients are multiplied by powers of 10.

The convention is to write only the numeric coefficients and, from their position, deduce the necessary powers of 10 with powers increasing from right to left. In general, a number with a decimal point is represented by a series of coefficients:

$$a_5a_4a_3a_2a_1a_0.\ a_{-1}a_{-2}a_{-3}$$

The coefficients a_j are any of the 10 digits (0, 1, 2…. 9), and the subscript value j gives the place value and, hence, the power of 10 by which the coefficient must be multiplied.

Thus, the preceding decimal number can be expressed as

$$10^5 a_5 + 10^4 a_4 + 10^3 a_3 + 10^2 a_2 + 10^1 a_1 + 10^0 a_0 + 10^{-1} a_{-1} + 10^{-2} a_{-2} + 10^{-3} a_{-3}$$

with $a_3 = 7$, $a_2 = 3$, $a_1 = 9$, and $a_0 = 2$.

The *binary* system is a different number system. The coefficients of the binary number system have only two possible values: 0 and 1.

Number systems provide the basis for all operations in information processing systems.

* In a number system the information is divided into a group of symbols; for example, 26 English letters, 10 decimal digits etc. In conventional arithmetic, a number system based upon ten units (0 to 9) is used.

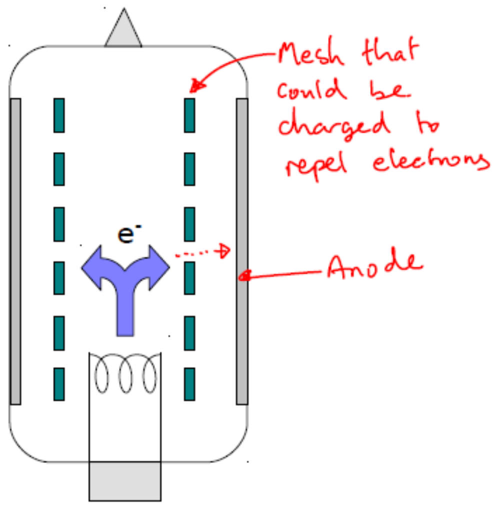
Decimal	Binary	Octal	Hexadecimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

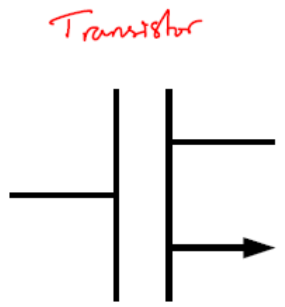
* The number system with the basic symbols 0 and 1 is called binary. Arithmetic and logic circuits used in digital systems operate with only 0's and 1's (Binary digits). The binary digit (either 0 or 1) is called **a bit**.

Name	Decimal	Binary	Octal	Hexadecimal
Radix	10	2	8	16
Digits	0, 1, 2, 3, 4, 5, 6, 7, 8, 9	0, 1	0, 1, 2, 3, 4, 5, 6, 7	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, *A, B, C, D, E, F*
First	0	0	0	0
seventeen	1	1	1	1
positive	2	10	2	2
integers	3	11	3	3
	4	100	4	4
	5	101	5	5
	6	110	6	6
	7	111	7	7
	8	1000	10	8
	9	1001	11	9
	10	1010	12	*A*
	11	1011	13	*B*
	12	1100	14	*C*
	13	1101	15	*D*
	14	1110	16	*E*
	15	1111	17	*F*
	16	10000	20	10

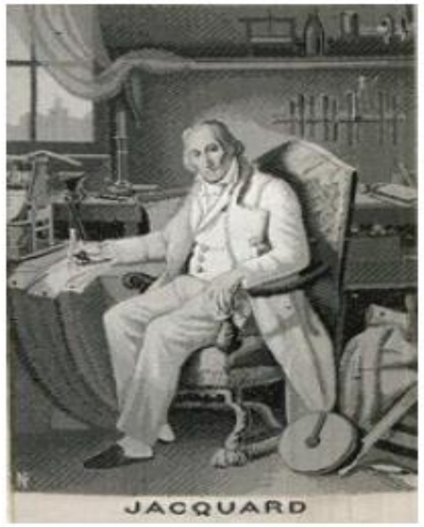
Bit and Digital Electronics:

- The nice thing about a bit is that, with only two states, it is easy to embody in physical machinery
- Each bit is simply a switch and computers moved from vacuum tubes to transistors for this





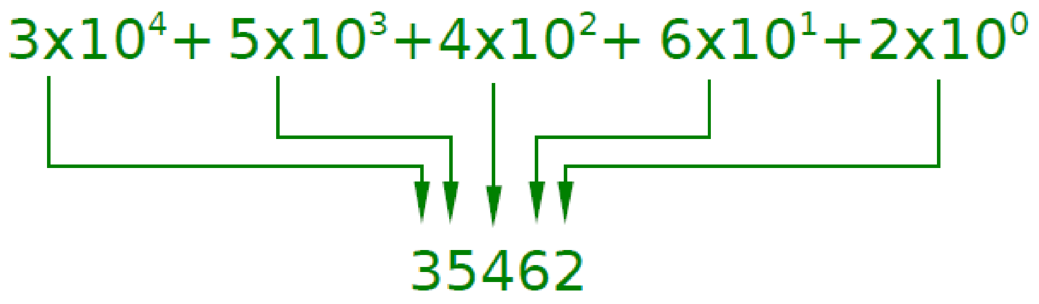
- A bit (Binary digIT) is merely 0 or 1
- It is a unit of *information* since you cannot communicate with anything less than two states
- The use of binary encoding dates back to the 1600s with Jacquard's loom, which created textiles using card templates with holes that allowed needles through



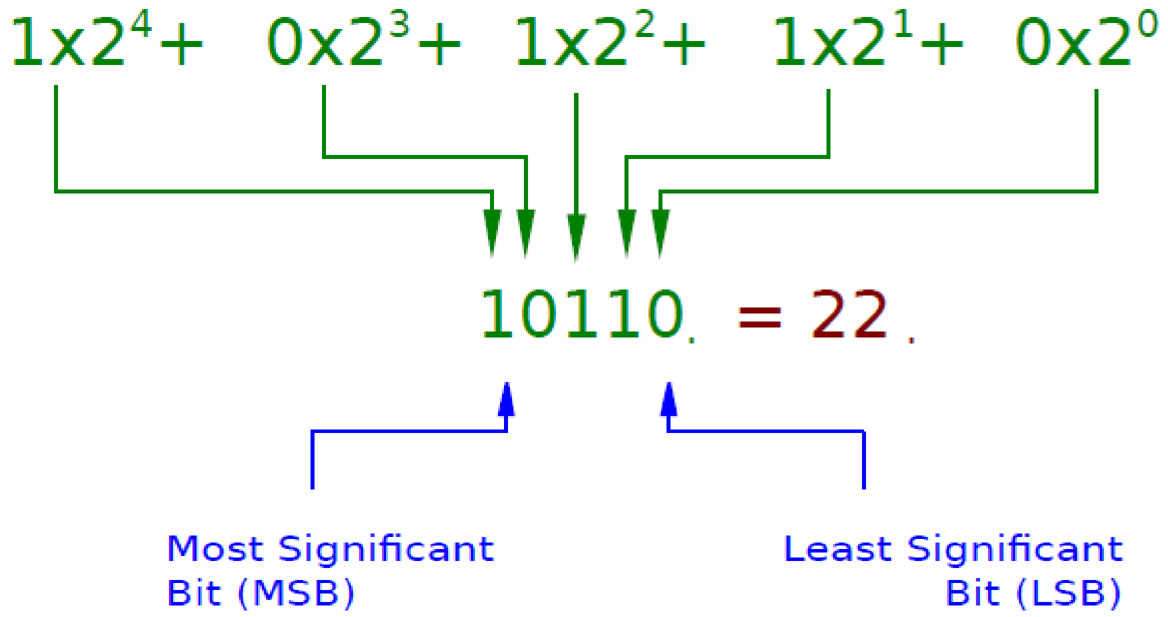
2 10

Decimal and Binary number systems:

Decimal number system:

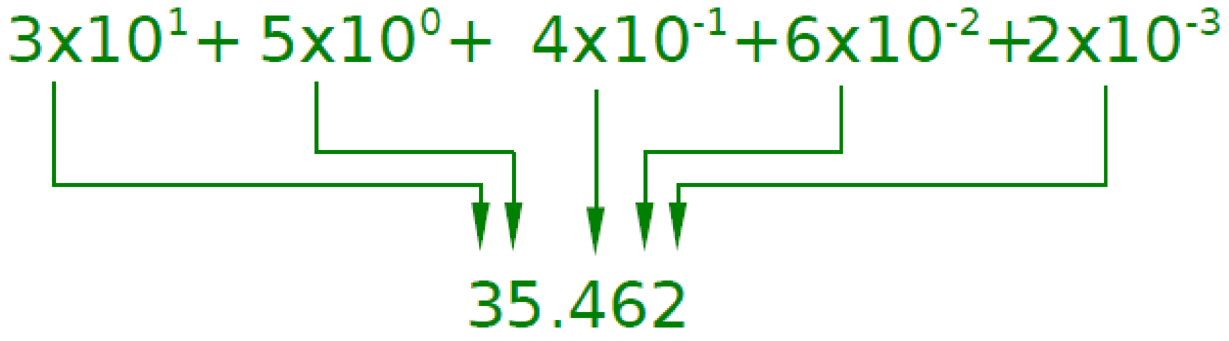


Binary number system:


$1x2^4 + 0x2^3 + 1x2^2 + 1x2^1 + 0x2^0$
10110 = 22
Most Significant Bit (MSB)
Least Significant Bit (LSB)

Dealing with Fractional numbers:

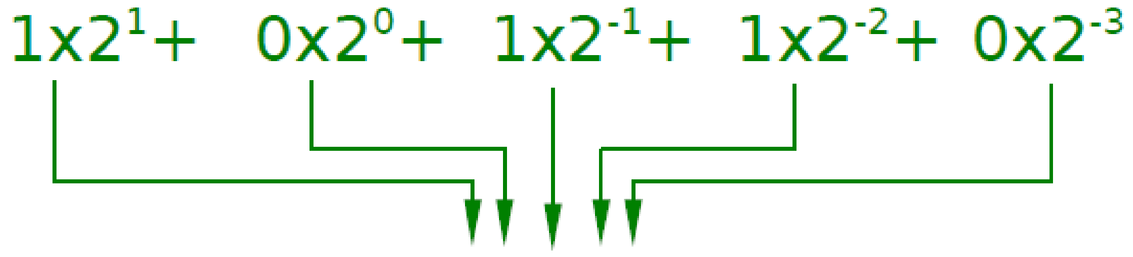
Decimal number system:



$3\times10^{1}+5\times10^{0}+4\times10^{-1}+6\times10^{-2}+2\times10^{-3}$

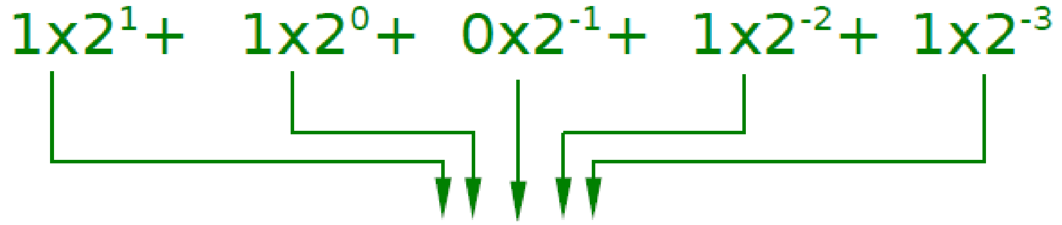
35.462

Binary number system:



$1\times2^{1}+0\times2^{0}+1\times2^{-1}+1\times2^{-2}+0\times2^{-3}$

$10.110_{2} = 2.75_{10}$



$1\times2^{1}+1\times2^{0}+0\times2^{-1}+1\times2^{-2}+1\times2^{-3}$

$11.011_{2} = 3.375_{10}$

- With d decimal digits, we can represent 10^d different values, usually the numbers 0 to (10^d-1) inclusive
- In binary with n bits this becomes 2^n values, usually the range 0 to (2^n-1)

- Computers usually assign a set number of bits (physical switches) to an instance of a type.
  - An integer is often 32 bits, so can represent positive integers from 0 to 4,294,967,295 incl.
  - Or a range of negative and positive integers...

Bytes:

- A byte was traditionally the number of bits needed to store a character of text
- A de-facto standard of 8 bits has now emerged
  - 256 values
  - 0 to 255 incl.
  - Two hex digits to describe
    - 0x00=0, 0xFF=255

Other common bases:

- Higher bases make for shorter numbers that are easier for humans to manipulate. e.g. $6654733_d = 11001011000101100001101_b$
- We traditionally choose powers-of-2 bases because this corresponds to whole chunks of binary
- **Octal** is base-8 ($8=2^3$ digits, which means 3 bits per digit)
  - 6654733_d=011-001-011-000-101-100-001-101_b= 31305415_o
- **Hexadecimal** is base-16 ($16=2^4$ digits so 4 bits per digit)
  - Our ten decimal digits aren't enough, so we add 6 new ones: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
  - 6654733_d=0110-0101-1000-1011-0000-1101_b=$658B0D_h$

- Check: what does 0xBD represent?
  - B $\rightarrow$ 11 or 1011
  - D $\rightarrow$ 13 or 1101
  - Result is $11 \times 16^1 + 13 \times 16^0 =$ **189** or 10111101

Units of Bytes:

- Strictly the SI units since 1998 are:
- Kibibyte (KiB)
  - 1024 bytes (closest power of 2 to 1000)
- Mebibyte (MiB)
  - 1,048,576 bytes
- Gibibyte (GiB)
  - 1,073,741,824 bytes
- but these haven't really caught on so we tend to still use the SI Kilobyte, Megabyte, Gigabyte. This leads to lots of confusion since technically these are multiples of 1,000.

Unassigned Integers Addition:

- Addition of unsigned integers works the same way as addition of decimal (only simpler!)
  - 0 + 0 = 0
  - 0 + 1 = 1
  - 1 + 0 = 1
  - 1 + 1 = 0, carry 1

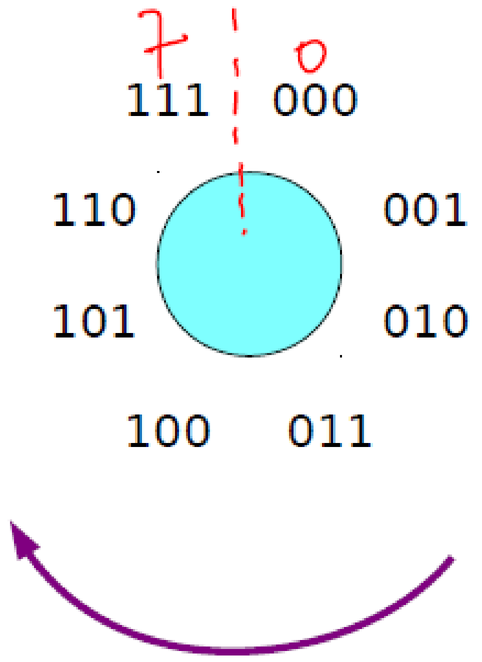
- Only issue is that computers have fixed sized types so we can't go on adding forever...

$$\begin{array}{r} \boxed{0}\;0\;1 \\ 001 \\ +\ 001 \\ \hline 010 \end{array} \qquad\qquad \begin{array}{r} \boxed{1}\;1\;1 \\ 111 \\ +\ 001 \\ \hline 000 \end{array}$$

Carry flag: 0   Carry flag: 1

Clock arithmetic:

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

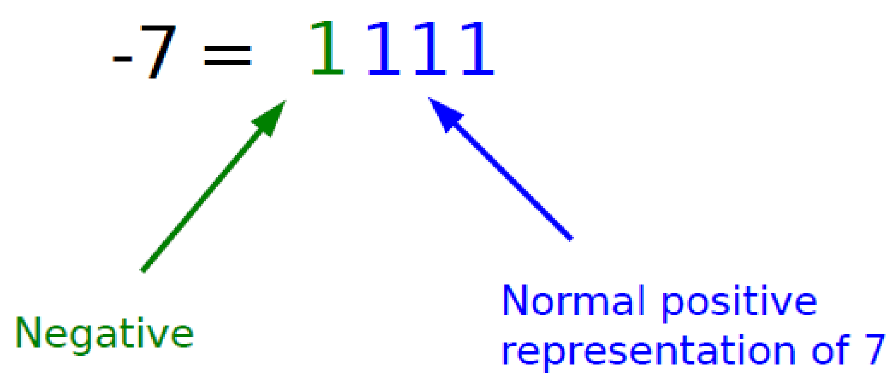


- Overflow takes us across the dotted boundary
  - So 7+1=0 (overflow)
  - We say this is (7+1) mod 8

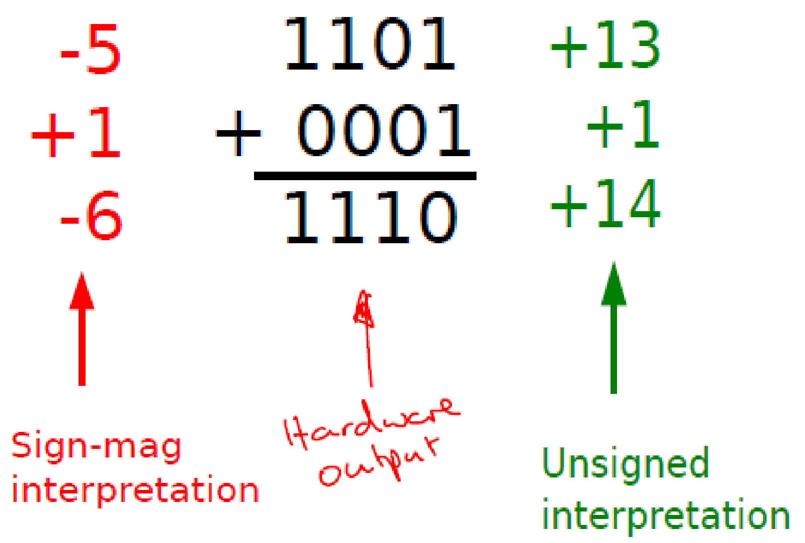
Negative numbers: **<u>sign-magnitude</u>** technique

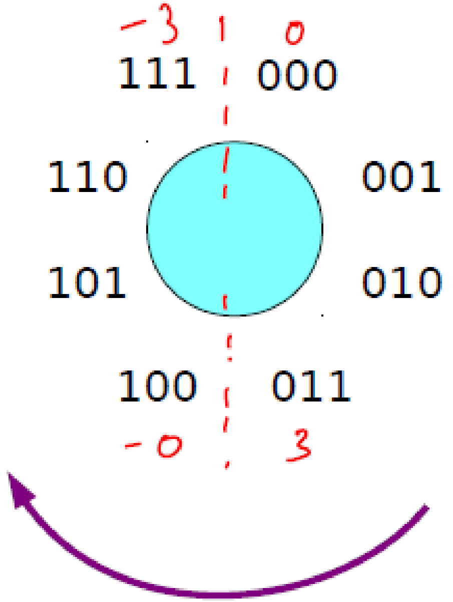
- All of this skipped over the need to represent negatives.
- The naïve choice is to use the MSB to indicate +/-
  - 1 in the MSB → negative
  - 0 in the MSB → positive

$-7 = 1111$



Problems with the Sign-Magnitude:





000	0
001	1
010	2
011	3
100	-0
101	-1
110	-2
111	-3

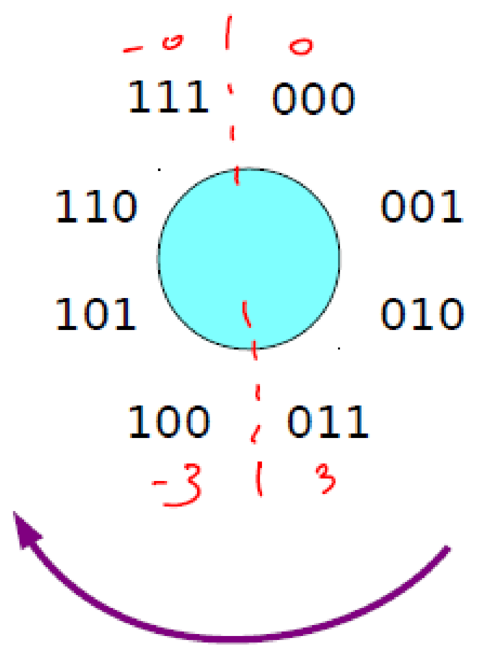
Increasing

Increasing

- Gives us two discontinuities and a reversal of direction using normal addition circuitry!!

Ones' compliment:

- The negative is the positive with all the bits flipped
- 7 → 0111 so -7 → 1000
- Still the MSB is the sign

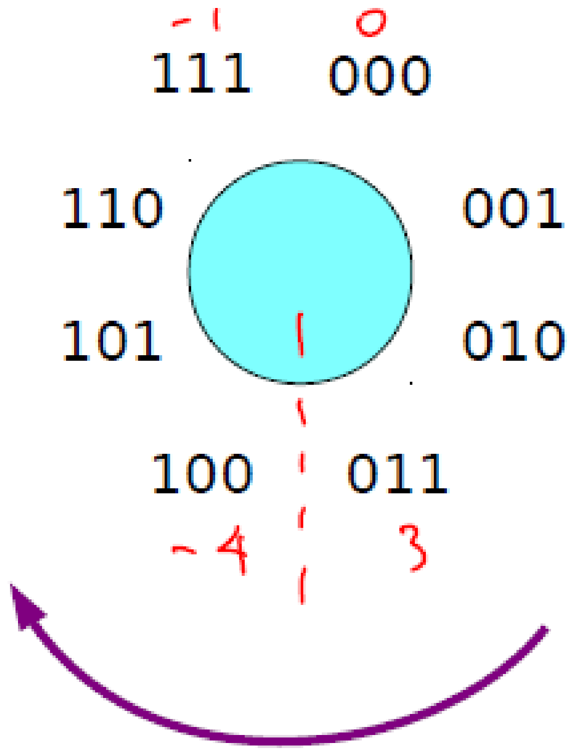


000	0	Increasing
001	1	
010	2	
011	3	
100	-3	Increasing
101	-2	
110	-1	
111	-0	

- One discontinuity but still -0

Two’s compliment:

- The negative is the positive with all the bits flipped and 1 added (the same procedure for the inverse)
- 7 → 0111 so -7 → 1000+0001 → 1001



000	0
001	1
010	2
011	3
100	-4
101	-3
110	-2
111	-1

- Still the MSB is the sign
- One discontinuity and proper ordering

Two’s compliment:

- Positive to negative: Invert all the bits and add 1

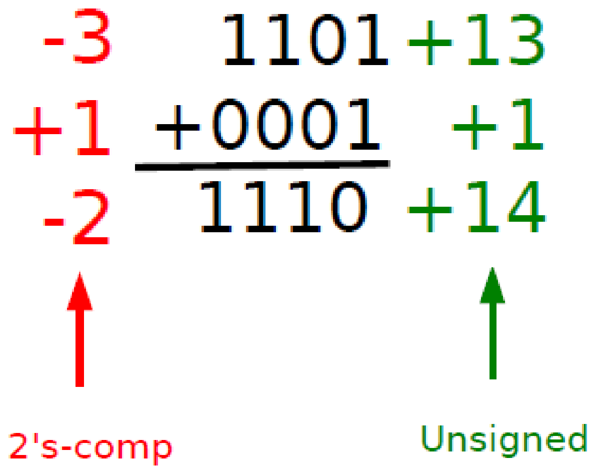
  0101 (+5) → 1010 → 1011 (-5)

- Negative to positive: Same procedure!!

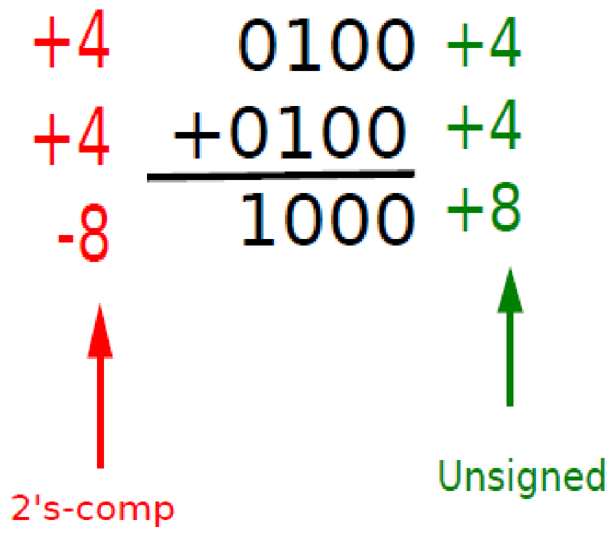
  1011 (-5) → 0100 → 0101 (+5)

Signed addition:

- ...it just works with our addition algorithm!



- So we can use the same circuitry for unsigned and 2s-complement addition :-)



Carry flag: 0

- The problem is our MSB is now signifying the sign and our carry should really be testing the bit to its right :-(
- So we introduce an overflow flag that indicates this problem

- When adding/subtracting
  - **Carry** flag → overflow for **unsigned** integer
  - **Overflow** flag → overflow for **signed** integer

Signals in the digital systems have,

* two distinct values,
* the circuit elements having have two stable states.

There is a direct analogy among binary signals, binary circuit elements, and binary digits. A binary number of *n* digits, may be represented by *n* binary circuit elements, each having an output signal equivalent to 0 or 1.

Digital systems represent and manipulate not only binary numbers, but also many other discrete elements of information.

A group of bits which is used to represent the discrete elements of information is a symbol. The mapping of symbols to a binary value is known a binary code. This mapping must be unique. For example, the decimal digits 0 through 9 are represented in a digital system with a code of four bits.

A digital system is a system that manipulates discrete elements of information that is represented internally in binary form.

* A discrete element of information that is distinct among a group of quantities can be represented with a binary code (i.e., a pattern of 0's and 1's).
* Binary codes merely change the symbols, not the meaning of the elements of information that they represent.
* A bits of a computer represent some type of coded information rather than binary numbers.

An n-bit binary code is a group of n bits that assumes up to 2^n distinct combinations of 1's and 0's, with each combination representing one element of the set that is being coded. Examples,

1. A set of four elements can be coded with two bits, with each element assigned one of the following bit combinations: 00, 01, 10, 11.
2. A set of eight elements requires a three-bit code and a set of 16 elements requires a four-bit code.

Each element must be assigned a unique binary bit combination, and no two elements can have the same value.

Although the *minimum* number of bits required to code $2n$ distinct quantities is n, there is no *maximum* number of bits that may be used for a binary code. For example, the 10 decimal digits can be coded with 10 bits, and each decimal digit can be assigned a bit combination of nine 0's and a 1.

Examples of Signals in a digital system:

A **binary code** represents text, computer processor instructions, or any other data using a two-symbol system. The two-symbol system used is often "0" and "1" from the binary number system. The binary code assigns a pattern of binary digits, also known as bits, to each character, instruction, etc. For example, a binary string of eight bits can represent any of 256 possible values and can, therefore, represent a wide variety of different items.

The American Standard Code for Information Interchange (ASCII), uses a 7-bit binary code to represent text and other characters within computers, communications equipment, and other devices. Each letter or symbol is assigned a number from 0 to 127.

For example, lowercase "a" is represented by `1100001` as a bit string (which is "97" in decimal).

```
01010111  01101001  01101011
01101001  01110000  01100101
01100100  01101001  01100001
```

Exercise: Write your name in Binary code.

Current uses of binary:

Most modern computers use binary encoding for instructions and data. CDs, DVDs, and Blu-ray Discs represent sound and video digitally in binary form. Telephone calls are carried digitally on long-distance and mobile phone networks using pulse-code modulation, and on voice over IP networks.

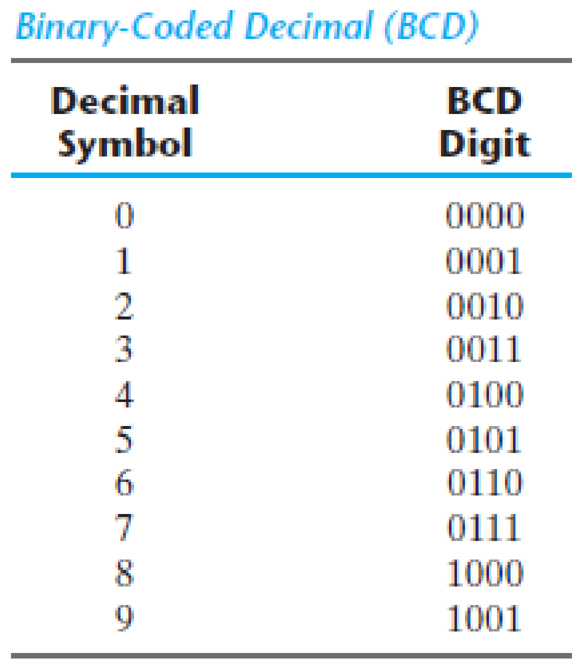
American Standard Code for Information Interchange (ASCII)

	$b_7b_6b_5$							
$b_4b_3b_2b_1$	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	“	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	‘	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	–	=	M	]	m	}
1110	SO	RS	.	>	N	∧	n	~
1111	SI	US	/	?	O	—	o	DEL

Binary-Coded Decimal codes:

The binary number system is the most natural system for a computer because it is readily represented in today’s electronic technology, most people are more accustomed to the decimal system.

The code most commonly used for the decimal digits is the straight binary assignment listed in Table. This scheme is called *binary-coded decimal* (BCD).



Binary-Coded Decimal (BCD)

Decimal Symbol	BCD Digit
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

The binary combinations 1010 through 1111 are not used and have no meaning in BCD.

Example: Consider decimal 185 and its corresponding value in BCD and binary:

$(185)_{10} = (0001\ 1000\ 0101)_{BCD} = (10111001)_2$

Binary logic and logic gates:

Binary logic deals with variables that take on two discrete values and with operations that assume logical meaning. The two values the variables assume may be called by different names (*true* and *false, yes* and *no*, 0 and 1 etc.).

LOGIC STATES The voltage in a digital circuit is allowed to be in only one of two states: HIGH or LOW.

In the TTL logic family, any voltage in the range 2.8 to 5.0 V is HI, and any voltage in the range 0 to 0.8 V is LO.
Any voltage outside this range is undefined, and therefore illegal, except briefly during transitions. If a TTL circuit is given a voltage in this undefined range, it might, unpredictably, interpret it as either a "1" or a "0."
We will sometimes refer to HI as the "5 volt" level, and LO as the "0 volt" level.

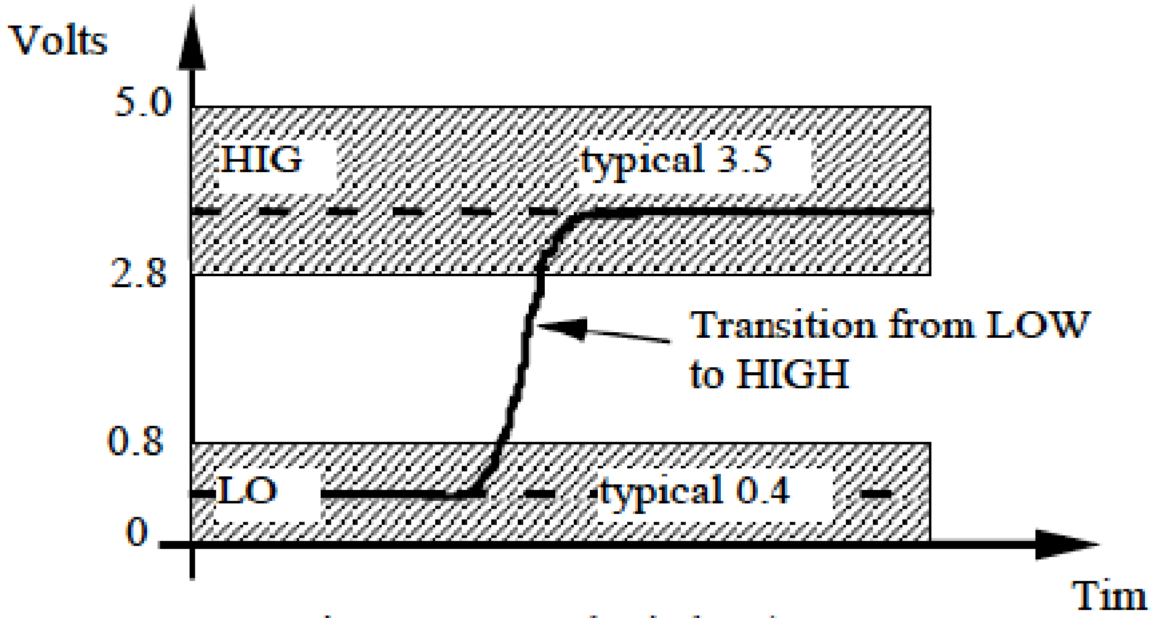


Figure 9.1 TTL logic levels

Binary logic:

Binary logic deals with variables that take on two discrete values and with operations that assume logical meaning.

The two values the variables assume may be called by different names (*true* and *false*, *yes* and *no*, etc.), but for our purpose, it is convenient to think in terms of bits and assign the values 1 and 0.

The binary logic is equivalent to an algebra called Boolean algebra.

Binary logic should not be confused with binary arithmetic.

One should realize that an arithmetic variable designates a number that may consist of many digits.

A logic variable is always either 1 or 0. For example, in binary arithmetic, we have 1 + 1 = 10 (read "one plus one is equal to 2"),

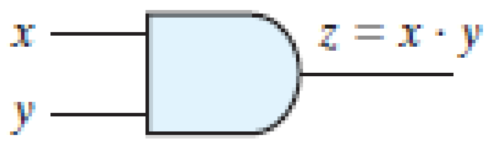
whereas in binary logic, we have 1 + 1 = 1 (read "one OR one is equal to one").

For each combination of the values of *x* and *y*, there is a value of *z* specified by the definition of the logical operation. Definitions of logical operations may be listed in a compact form called *truth tables*.

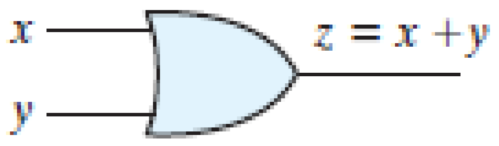
A truth table is a table of all possible combinations of the variables, showing the relation between the values that the variables may take and the result of the operation.

Logic gates:

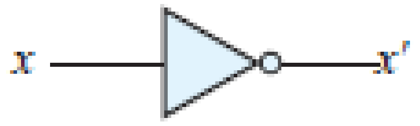
Logic gates are electronic circuits that operate on one or more input signals to produce an output signal.



(a) Two-input AND gate



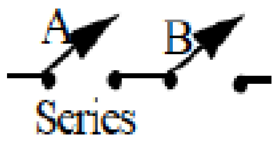
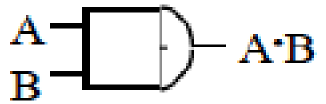
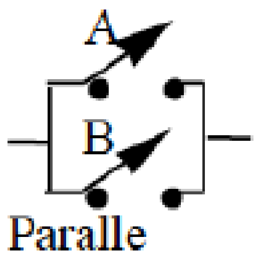
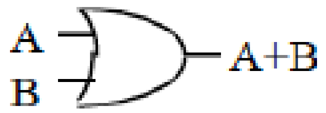
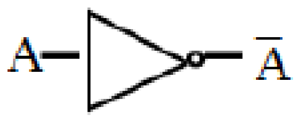
(b) Two-input OR gate



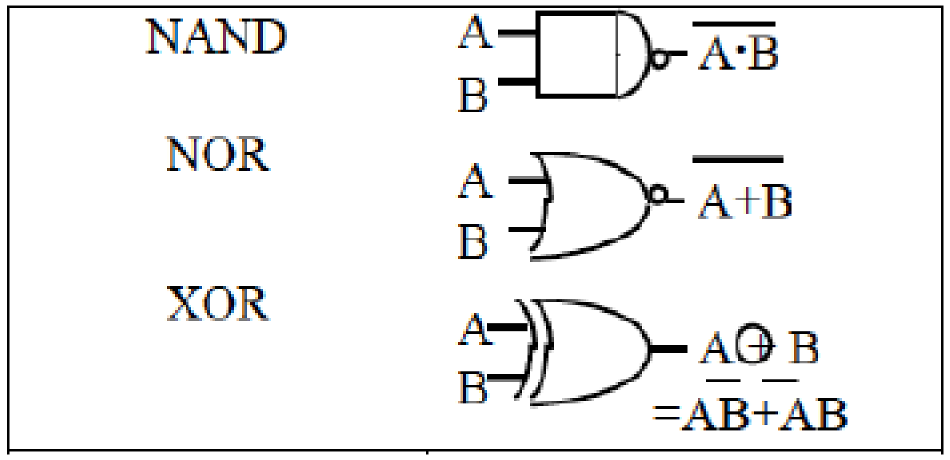
(c) NOT gate or inverter

Logic gates:

All combinational tasks in digital systems are performed with devices called gates, which perform the operations of Boolean algebra applied to two-state (binary) systems.

Operation	Switches	Condition that circuit is closed	Boolean Notation	Symbol	Truth Table
AND		(A AND B are closed)	$A \bullet B$ or AB		A B | A·B 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1
OR		(A OR B is closed)	$A + B$		A B | A+B 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1
NOT (same as invert)	Different kind of switch	1 means open 0 means closed	NOT $A \equiv \bar{A}$		A | Ā 0 | 1 1 | 0

Compound Gates



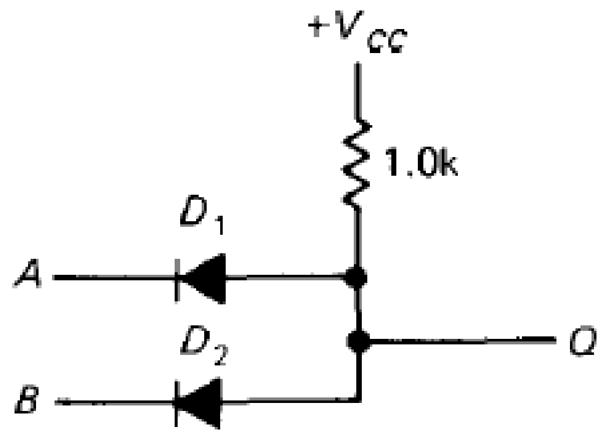
The INVERT function can be combined with gates, forming NAND and NOR.

Exclusive-OR is an interesting function, although less fundamental AND and OR.

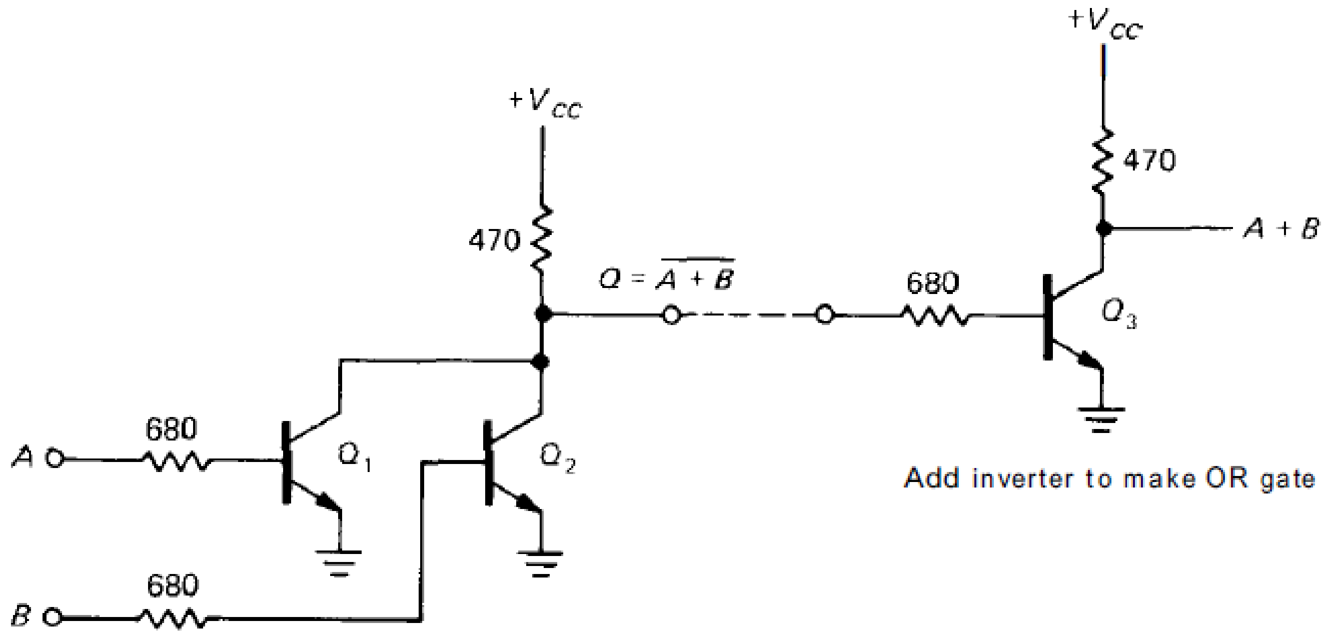
Boolean algebra:

OR	AND	NOT
$A+0=A$	$A \bullet 0=0$	$A+\bar{A}=1$
$A+1=1$	$A \bullet 1=A$	$A \bullet \bar{A}=0$
$A+A=A$	$A \bullet A=A$	$\bar{\bar{A}}=A$
$A+\bar{A}=1$	$A \bullet \bar{A}=0$	

Discrete circuits for gates:



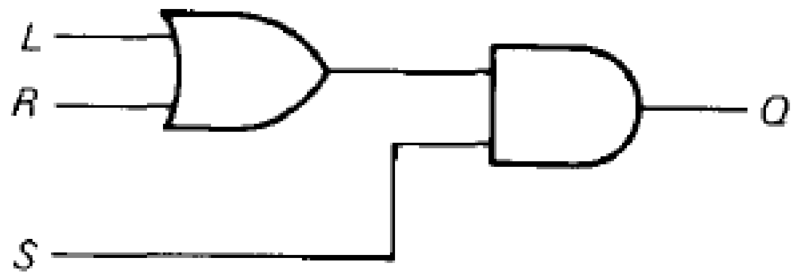
A diode AND gate.



A transistor NOR gate.

<u>Disadvantages:</u> (a) Its LOW output is a diode drop above the signal holding the input LOW. Obviously you couldn't use very many of these in a row. (b) There is no ability of one output to drive several inputs, since any load at the output is seen by the signal at the input.
(c) It is slow, because of resistive pullup.

Problem: To sound a buzzer if either car door is open and the driver is seated. Discuss.



In practice, the switches the inputs will probably close a circuit to ground, to save extra wiring. This means that the inputs will go LOW when a door is opened. In other words, we have "negative-true" inputs. Let's rework the example with this in mind, calling the inputs L’, R', and S’.

Lift doors should close (*Z*) if:

(a) the master switch (*A*) is on and either

(b) a call (*B*) is received from any other floor, or

(c) the doors (*C*) have been open for more than 10 seconds, or

(d) the selector push within the lift (*D*) is pressed for another floor.

Devise a logic circuit to meet these requirements.

A water tank feeds three separate processes. When any two of the processes are in operation at the same time, a signal is required to start a pump to maintain the head of water in the tank.
Devise a logic circuit using **nor**-gates only to give the required signal.

Boolean Algebra:

Because binary logic is used in all of today’s digital computers and devices, the cost of the circuits that implement it is an important factor addressed by designers.

Finding simpler and cheaper, but equivalent, realizations of a circuit can reap huge payoffs in reducing the overall cost of the design. Mathematical methods that simplify circuits rely primarily on Boolean algebra.

Boolean algebra, defined with a set of elements, a set of operators, and a number of unproved axioms or postulates.

A *set* of elements is any collection of objects, usually having a common property.

A *binary operator* defined on a set *S* of elements is a rule that assigns, to each pair of elements from *S*, a unique element from *S*.

Example, consider the relation $a * b = c$. We say that * is a binary operator if it specifies a rule for finding c from the pair (a, b) and also if $a, b, c \in S$. However, * is not a binary operator if a, b H S, and if $c \notin S$.

THE ALGEBRA OF LOGIC

Simple Relations

$0 + 0 = 0$	$0 \cdot 1 = 0$	$A + A = A$	$0 + A = A$
$0 \cdot 0 = 0$	$0 + 1 = 1$	$A \cdot A = A$	$0 \cdot A = 0$
$1 + 1 = 1$	$\overline{0} = 1$	$\overline{A} \cdot A = 0$	$1 + A = 1$
$1 \cdot 1 = 1$	$\overline{1} = 0$	$\overline{A} + A = 1$	$1 \cdot A = A$

Commutative Law	Associative Law	Distributive Law
$A + B = B + A$	$(A + B) + C = A + (B + C)$	$(A \cdot B) \cdot C = A \cdot (B \cdot C)$
$A \cdot B = B \cdot A$	$A \cdot (B + C) = A \cdot B + A \cdot C$	

Simplification Rules

$A + A \cdot B = A$ $\qquad$ $A + \overline{A} \cdot B = A + B$ $\qquad$ $A \cdot (A + B) = A$

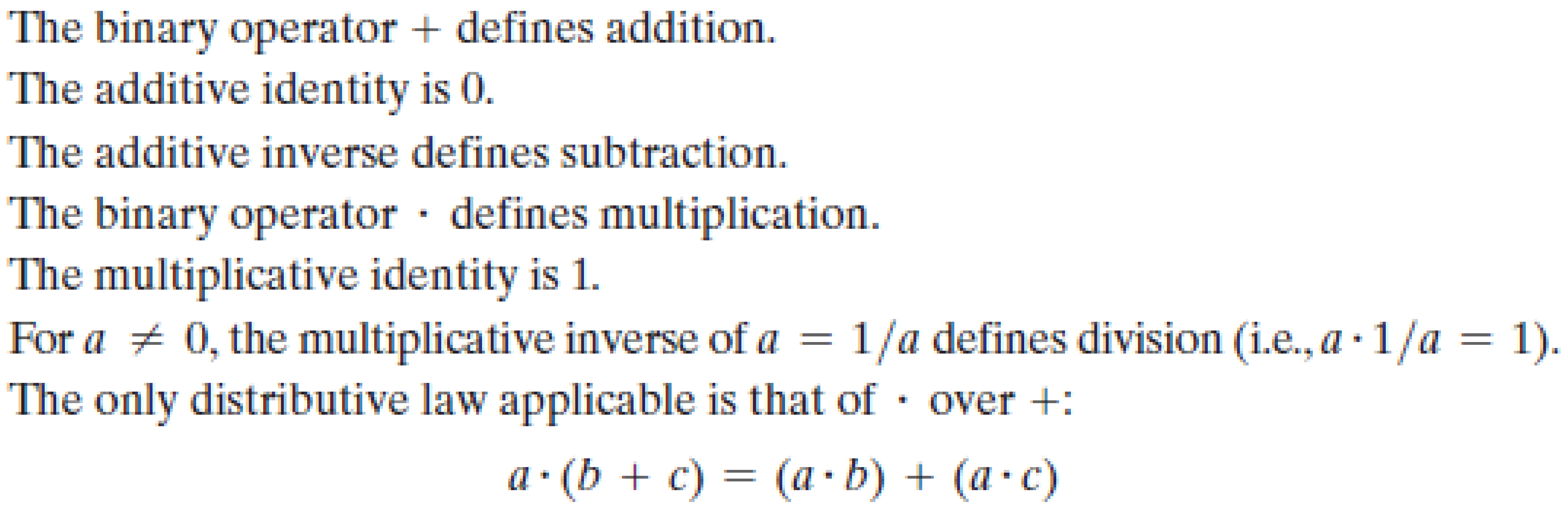
DeMorgan's Theorem

$\overline{A \cdot B} = \overline{A} + \overline{B}$ $\qquad$ $\overline{A + B} = \overline{A} \cdot \overline{B}$

Postulates and Theorems of Boolean Algebra

Postulate 2	(a)	$x + 0 = x$	(b)	$x \cdot 1 = x$
Postulate 5	(a)	$x + x' = 1$	(b)	$x \cdot x' = 0$
Theorem 1	(a)	$x + x = x$	(b)	$x \cdot x = x$
Theorem 2	(a)	$x + 1 = 1$	(b)	$x \cdot 0 = 0$
Theorem 3, involution		$(x')' = x$		
Postulate 3, commutative	(a)	$x + y = y + x$	(b)	$xy = yx$
Theorem 4, associative	(a)	$x + (y + z) = (x + y) + z$	(b)	$x(yz) = (xy)z$
Postulate 4, distributive	(a)	$x(y + z) = xy + xz$	(b)	$x + yz = (x + y)(x + z)$
Theorem 5, DeMorgan	(a)	$(x + y)' = x'y'$	(b)	$(xy)' = x' + y'$
Theorem 6, absorption	(a)	$x + xy = x$	(b)	$x(x + y) = x$

The operators and postulates have the following meanings:



The binary operator + defines addition.

The additive identity is 0.

The additive inverse defines subtraction.

The binary operator $\cdot$ defines multiplication.

The multiplicative identity is 1.

For $a \neq 0$, the multiplicative inverse of $a = 1/a$ defines division (i.e., $a \cdot 1/a = 1$).

The only distributive law applicable is that of $\cdot$ over +:

$$a \cdot (b + c) = (a \cdot b) + (a \cdot c)$$

Two-Valued Boolean Algebra

A two-valued Boolean algebra is defined on a set of two elements, *B* = {0, 1}, with rules for the two binary operators + and **.** as shown in the following operator tables:

x	y	$x \cdot y$
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$x + y$
0	0	0
0	1	1
1	0	1
1	1	1

x	x'
0	1
1	0

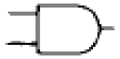
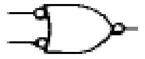
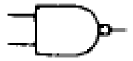
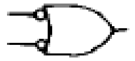
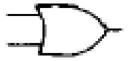
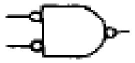
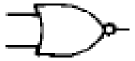
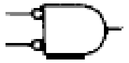
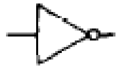
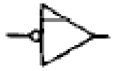
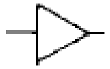
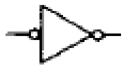
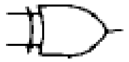
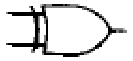
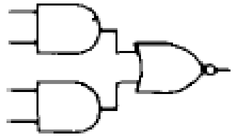
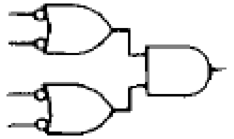
Logic Gate interchangeability:

It is possible to form one kind of gate from another.

important property of Boolean algebra is called the *duality principle* and states that every algebraic expression deducible from the postulates of Boolean algebra remains valid if the operators and identity elements are interchanged.

An AND gate has a HIGH output if both inputs are HIGH. If HIGH means "true," you get a true output only if all inputs are true. In other words, with positive-true logic, an AND gate performs the AND function. The same holds for OR.

What happens if LOW means "true," as in the last example? An AND gate gives a LOW if either input is true (LOW): It's an OR function! Similarly an OR gate gives a LOW only if both inputs are true (LOW). It's an AND function!

Name	Expression	Symbol	Negative true symbol
AND	AB		
NAND	$\overline{AB}$		
OR	$A + B$		
NOR	$\overline{A + B}$		
INVERT	$\overline{A}$		
BUFFER	A		
XOR	$A \oplus B$		
XNOR	$\overline{A \oplus B}$		
AOI			

Simplify and design the logic (digital) circuit for the following functions:

(i). $A(A'+B)$; (ii). $A+A'B$; (iii). $(A+B)(A+B')$; (iv). $AB+A'C+BC$; (v). $(A+B)(A'+C)(B+C)$.

Boolean function:

A Boolean function described by an algebraic expression consists of binary variables, the constants 0 and 1, and the logic operation symbols.

Complement of a function:

The complement of a function *F* is *F’* and is obtained from an interchange of 0’s for 1’s and 1’s for 0’s in the value of *F*.

The complement of a function may be derived algebraically through DeMorgan’s theorems and it can be extended to three or more variables.

Find the complement of the functions $F_1 = A'BC' + A'B'C$ and $F_2 = A(B'C' + BC)$.

Exclusive-OR gate:

Exclusive-OR is an interesting function, although less fundamental AND and OR.

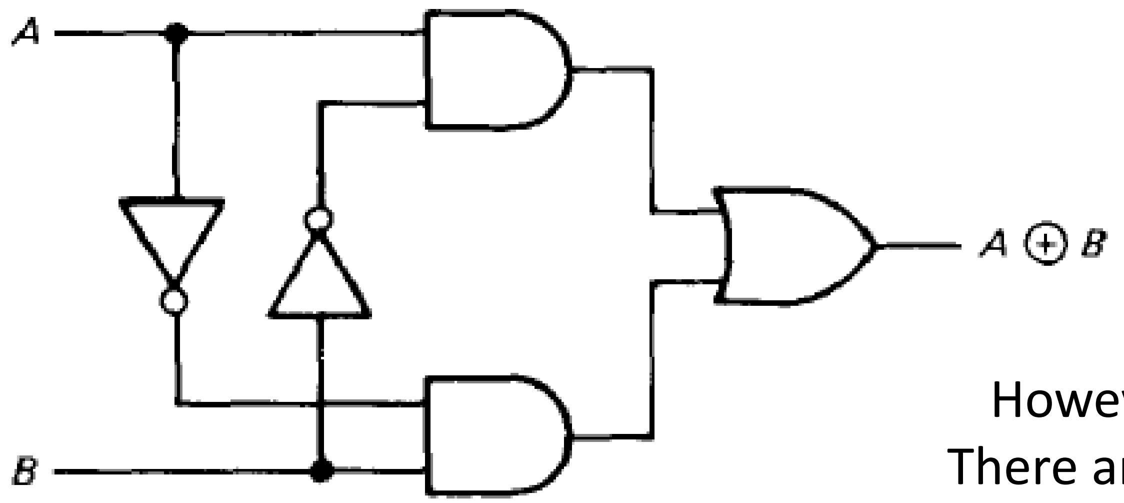
From truth table, we realize that the output is 1 only when (A,B)=(0,1) or (1,0).

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

In functional form, we can express it as,

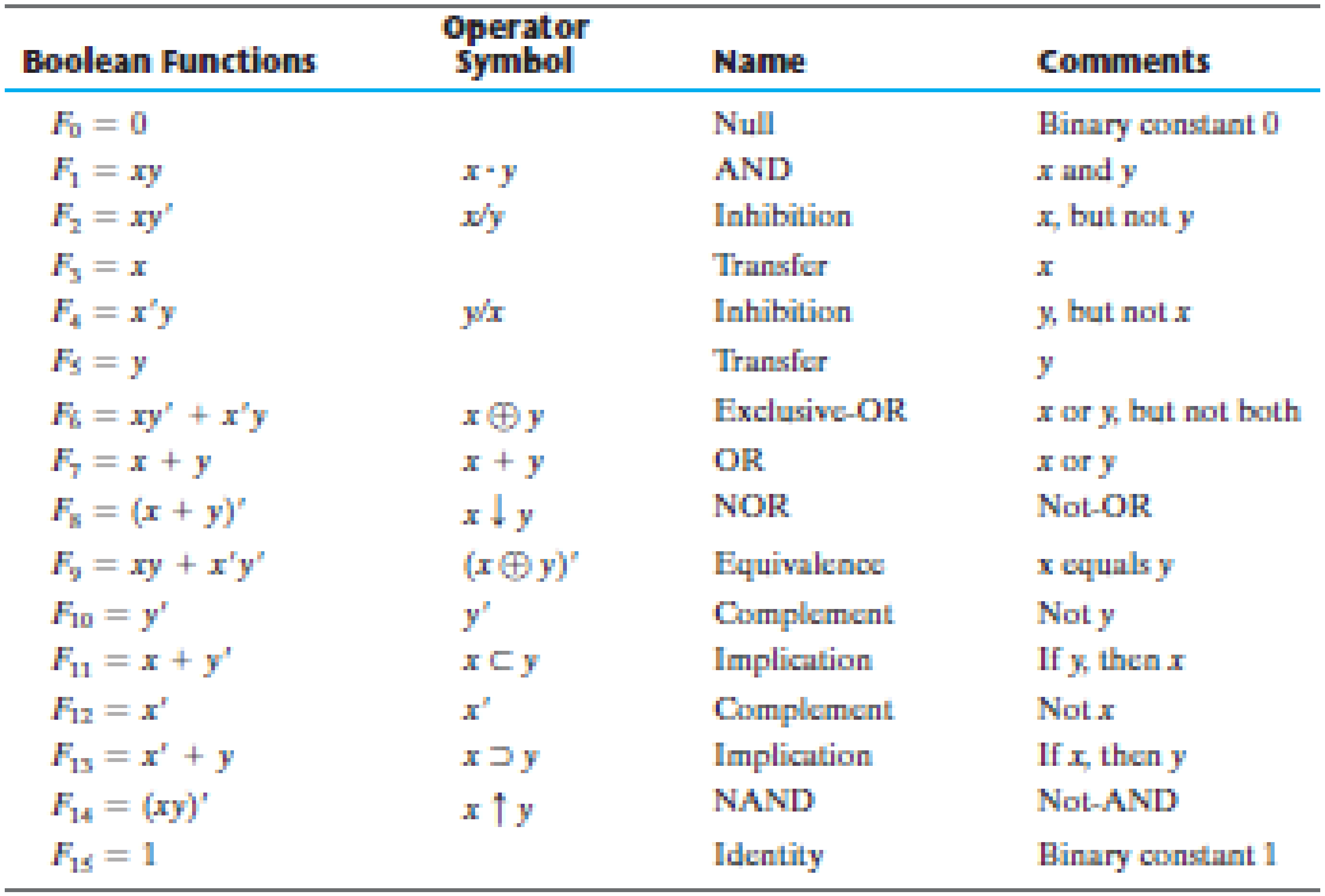
$$A \oplus B = \overline{A}B + A\overline{B}$$

Exclusive-OR gate can be realized as shown below,



However, this realization is not unique. There are still other ways to construct XOR.

Boolean Expressions for the 16 Functions of Two Variables:



Boolean Functions	Operator Symbol	Name	Comments
$F_0 = 0$		Null	Binary constant 0
$F_1 = xy$	$x \cdot y$	AND	x and y
$F_2 = xy'$	x/y	Inhibition	x, but not y
$F_3 = x$		Transfer	x
$F_4 = x'y$	y/x	Inhibition	y, but not x
$F_5 = y$		Transfer	y
$F_6 = xy' + x'y$	$x \oplus y$	Exclusive-OR	x or y, but not both
$F_7 = x + y$	$x + y$	OR	x or y
$F_8 = (x + y)'$	$x \downarrow y$	NOR	Not-OR
$F_9 = xy + x'y'$	$(x \oplus y)'$	Equivalence	x equals y
$F_{10} = y'$	y'	Complement	Not y
$F_{11} = x + y'$	$x \subset y$	Implication	If y, then x
$F_{12} = x'$	x'	Complement	Not x
$F_{13} = x' + y$	$x \supset y$	Implication	If x, then y
$F_{14} = (xy)'$	$x \uparrow y$	NAND	Not-AND
$F_{15} = 1$		Identity	Binary constant 1

Types of logic gates:

The possibility of constructing gates for the other logic operations is of practical interest. Factors to be weighed in considering the construction of other types of logic gates are,

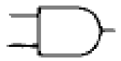
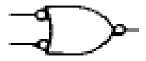
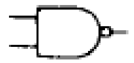
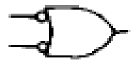
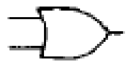
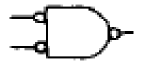
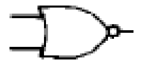
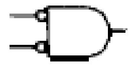
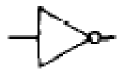
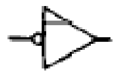
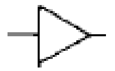
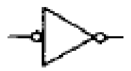
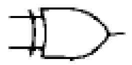
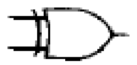
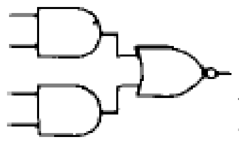
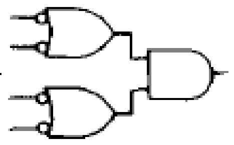
(1) the feasibility and economy of producing the gate with physical components,
(2) the possibility of extending the gate to more than two inputs,
(3) the basic properties of the binary operator, such as commutativity and associativity, and
(4) the ability of the gate to implement Boolean functions alone or in conjunction with other gates.

Of the 16 functions defined in Table, two are equal to a constant and four are repeated. There are only 10 functions left to be considered as candidates for logic gates.
Two—inhibition and implication—are not commutative or associative and thus are impractical to use as standard logic gates.

The other eight—complement, transfer, AND, OR, NAND, NOR, exclusive-OR, and equivalence—are used as standard gates in digital design.

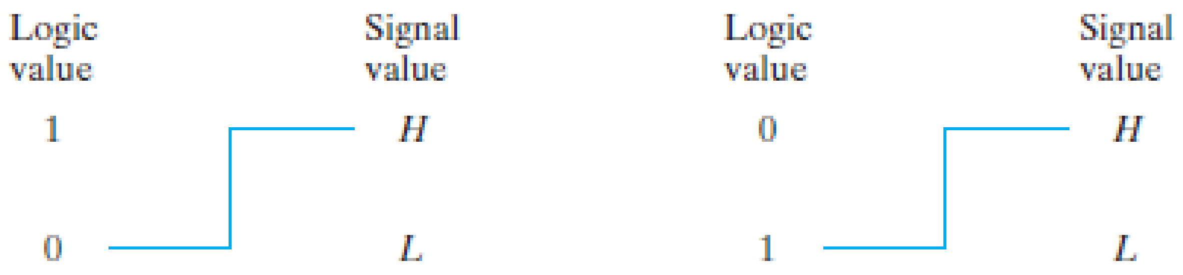
Catalog of common gates:

The last entry in the table is an AND-OR-INVERT gate (AOI).

Name	Expression	Symbol	Negative true symbol
AND	AB		
NAND	$\overline{AB}$		
OR	$A + B$		
NOR	$\overline{A + B}$		
INVERT	$\bar{A}$		
BUFFER	A		
XOR	$A \oplus B$		
XNOR	$\overline{A \oplus B}$		
AOI			

Positive and Negative Logic

The binary signal at the inputs and outputs of any gate has one of two values, except during transition.
One signal value represents logic 1 and the other logic 0. Since two
signal values are assigned to two logic values, there exist two different assignments of signal level to logic value.
The high-level *H* to represent logic 1 defines a positive logic system. Choosing the low-level *L* to represent logic 1 defines a negative logic system.
The terms *positive* and *negative* are somewhat misleading, since both signals
may be positive or both may be negative.



(a) Positive logic (b) Negative logic

Integrated circuits:

An integrated circuit (IC) is fabricated on a die of a silicon semiconductor crystal, called a *chip,* containing the electronic components for constructing digital gates.

Levels of Integration:

Digital ICs are often categorized according to the complexity of their circuits, as measured by the number of logic gates in a single package. The differentiation between those chips which have a few internal gates and those having hundreds of thousands of gates is made by customary reference to a package as being either a small-, medium-, large-, or very large-scale integration device.

Small-scale integration (SSI) devices contain several independent gates in a single package. The inputs and outputs of the gates are connected directly to the pins in the package. The number of gates is usually fewer than 10 and is limited by the number of pins available in the IC.

Medium-scale integration (MSI) devices have a complexity of approximately 10 to 1,000 gates in a single package. They usually perform specific elementary digital operations.

Large-scale integration (LSI) devices contain thousands of gates in a single package. They include digital systems such as processors, memory chips, and programmable logic devices.

Very large-scale integration (VLSI) devices now contain millions of gates within a single package. Examples are large memory arrays and complex microcomputer chips.

Because of their small size and low cost, VLSI devices have revolutionized the computer system design technology, giving the designer the capability to create structures that were previously uneconomical to build.

Digital integrated circuits are classified not only by their complexity or logical operation, but also by the specific circuit technology to which they belong.

Each logic family has its own basic electronic circuit upon which more complex digital circuits and components are developed. The basic circuit in each technology is a NAND, NOR, or inverter gate. The electronic components employed in the construction of the basic circuit are usually used to name the technology. Many different logic families of digital integrated circuits have been introduced commercially. The following are the most popular:

TTL transistor–transistor logic;
ECL emitter-coupled logic;
MOS metal-oxide semiconductor;
CMOS complementary metal-oxide semiconductor.

TTL AND CMOS:

Two most popular logic families: TTL (Transistor-transistor logic) and CMOS (complimentary MOS).

TTL is a logic family that has been in use for 50 years and is considered to be standard.
ECL has an advantage in systems requiring high-speed operation.
MOS is suitable for circuits that need high component density.
CMOS is preferable in systems requiring low power consumption, such as digital cameras, personal media players, and other handheld portable devices.

Low power consumption is essential for VLSI design; therefore, CMOS has become the dominant logic family, while TTL and ECL continue to decline in use.

The most important parameters distinguishing logic families,

Fan-out specifies the number of standard loads that the output of a typical gate can drive without impairing its normal operation. A standard load is usually defined as the amount of current needed by an input of another similar gate in the same family.

Fan-in is the number of inputs available in a gate.

Power dissipation is the power consumed by the gate that must be available from the power supply.

Propagation delay is the average transition delay time for a signal to propagate from input to output. For example, if the input of an inverter switches from 0 to 1, the output will switch from 1 to 0, but after a time determined by the propagation delay of the device. The operating speed is inversely proportional to the propagation delay.

Noise margin is the maximum external noise voltage added to an input signal that does not cause an undesirable change in the circuit output.

Computer-Aided Design of VLSI Circuits

Integrated circuits (having submicron geometric features)…. manufactured by optically projecting patterns of light onto silicon wafers (Photolithography).

Prior to exposure, the wafers are coated with a photoresistive material that either hardens or softens when exposed to light.

Removing extraneous photoresist leaves patterns of exposed silicon. The exposed regions are then implanted with dopant atoms to create a semiconductor material having the electrical properties of transistors and the logical properties of gates.

The design process translates a functional specification or description of the circuit (i.e., what it must do) into a physical specification or description (how it must be implemented in silicon).

Electronic design automation (EDA)

covers all phases of the design of integrated circuits.

A typical design flow for creating VLSI circuits consists of a sequence of steps beginning with design entry (e.g., entering a schematic) and culminating with the generation of the database that contains the photomask used to fabricate the IC.

There are a variety of options available for creating the physical realization of a digital circuit in silicon. The designer can choose between an application-specific integrated circuit (ASIC), a field-programmable gate array (FPGA), a programmable logic device (PLD), and a full-custom IC.

With each of these devices comes a set of CAD tools that provide the necessary software to facilitate the hardware fabrication of the unit.

Photolithography for IC manufacturing

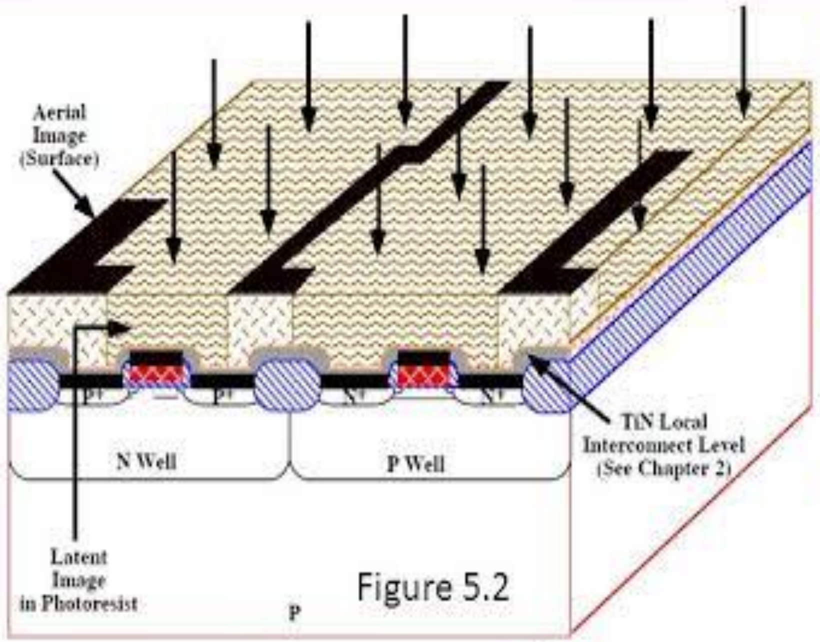

Aerial Image (Surface)
TiN Local Interconnect Level (See Chapter 2)
P+
N+
N Well
P Well
Latent Image in Photoresist
P

Figure 5.2

- In IC manufacturing, lithography is the single most important technology.
- ~35% of wafer manufacturing costs comes from lithography.
- The SIA roadmap is driven by the desire to continue scaling device feature sizes.
- 0.7× linear dimension shrink every 3 yr.
- Placement/alignment accuracy ~1/3 of feature size.

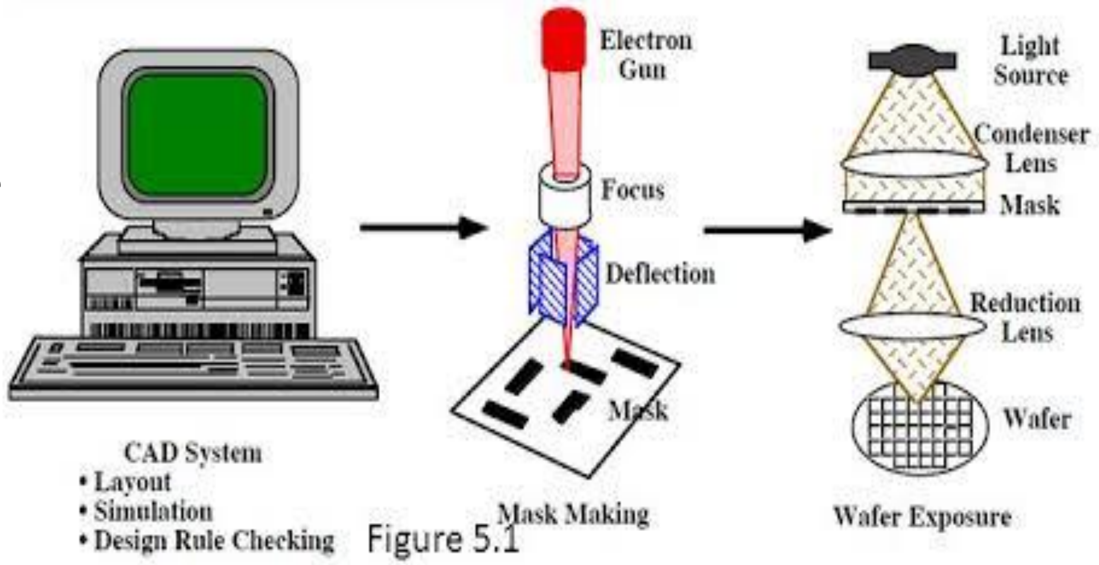


Figure 5.1

Patterning process consists of:
  Mask design
  Mask fabrication
  Wafer exposure

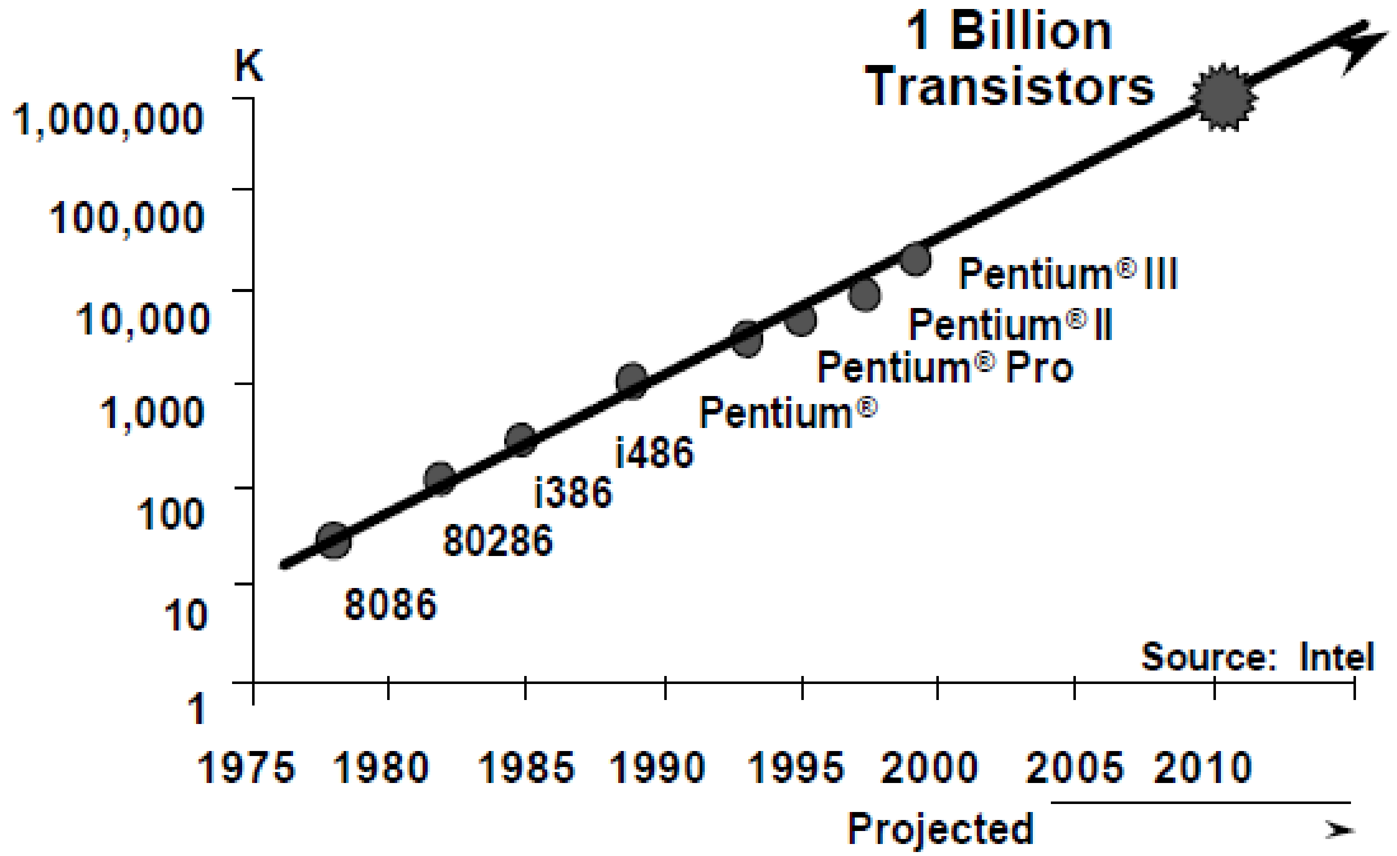
An important development in the design of digital systems is the use of a **hardware description language (HDL)**.

HDL-based models of a circuit or system are simulated to check and verify its functionality before it is submitted to fabrication, thereby reducing the risk and waste of manu-facturing a circuit that fails to operate correctly.

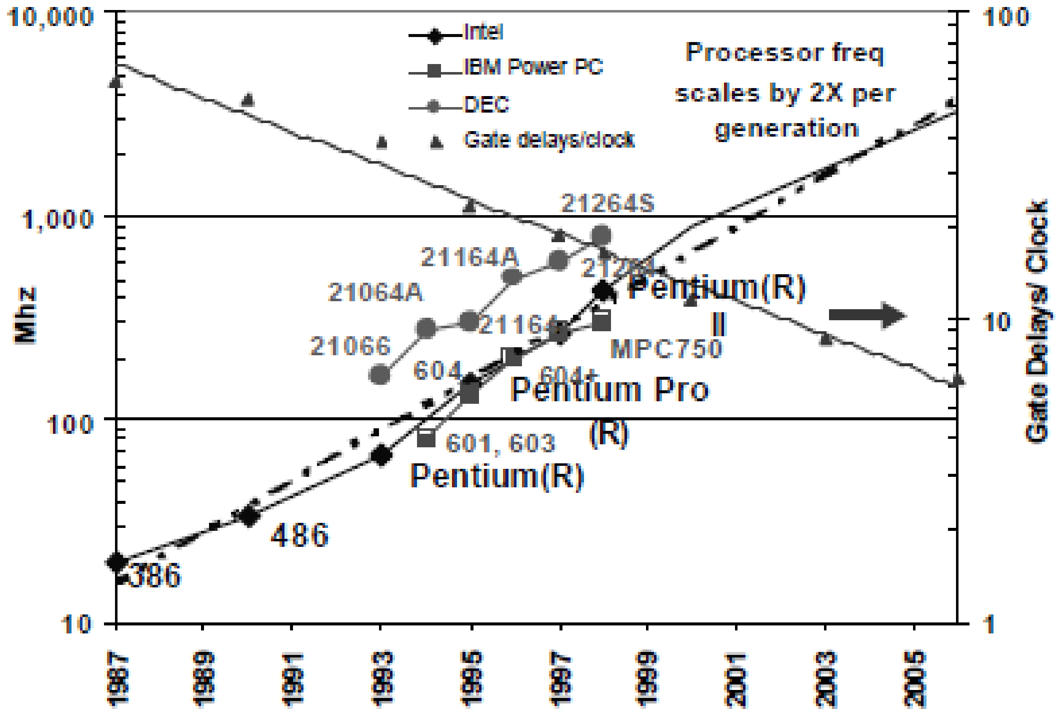
These two advances in technology have led to an **almost total reliance by industry on HDL-based synthesis tools and methodologies for the design of the circuits of complex digital systems.** Two HDLs—Verilog and VHDL—have been approved as standards by the Institute of Electronics and Electrical Engineers (IEEE) and are in use by design teams worldwide.

Moore’ law:

In 1965, Gordon Moore noted that the number of transistors on a chip doubled every 18 to 24 months.
He made a prediction that semi-conductor technology will double its effectiveness every 18 months



Processor frequency trend:



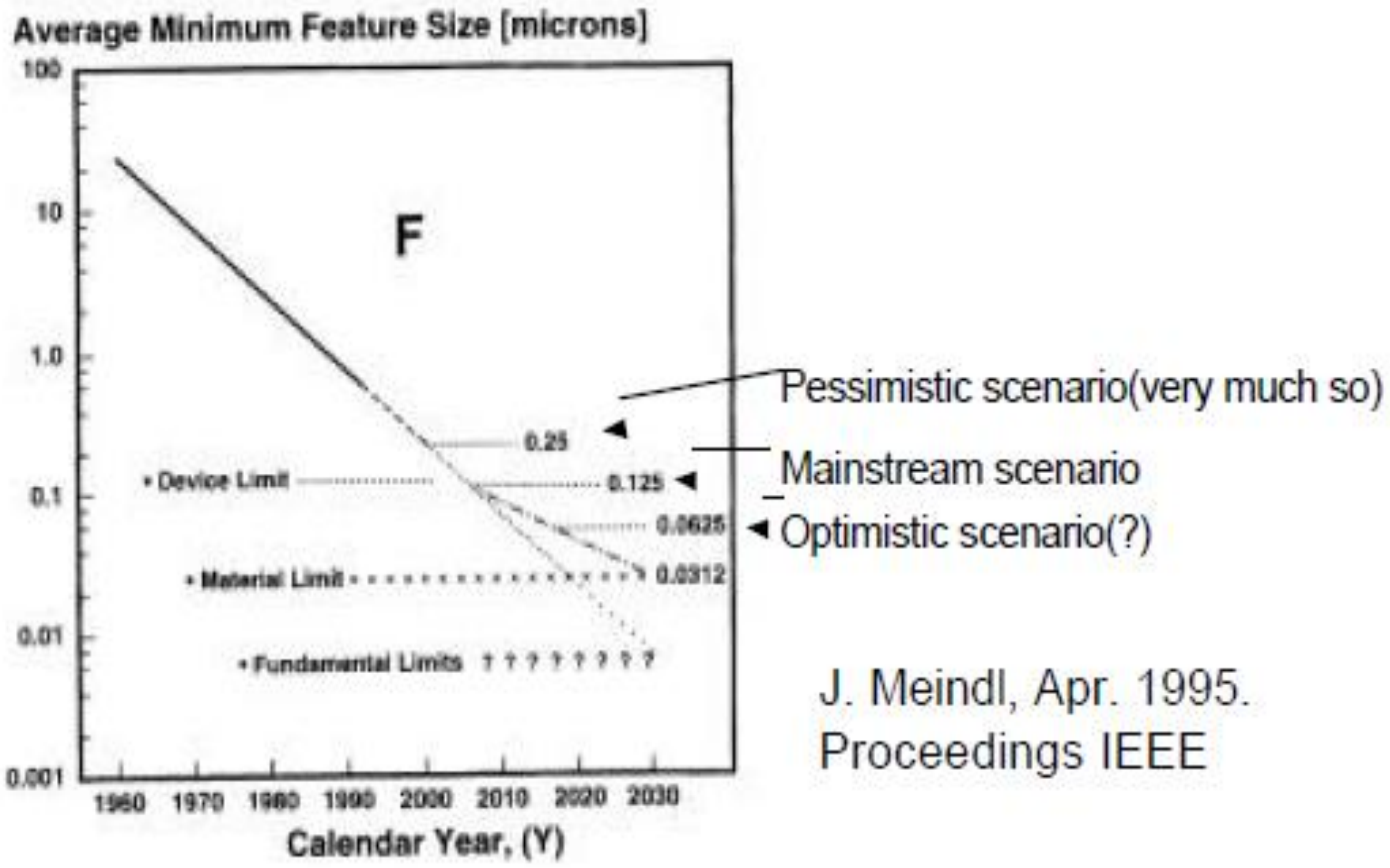
V.De, S. Borkar
ISLPED'99

❶ **Frequency doubles each generation**
❷ **Number of gates/clock reduce by 25%**

Technology scaling:

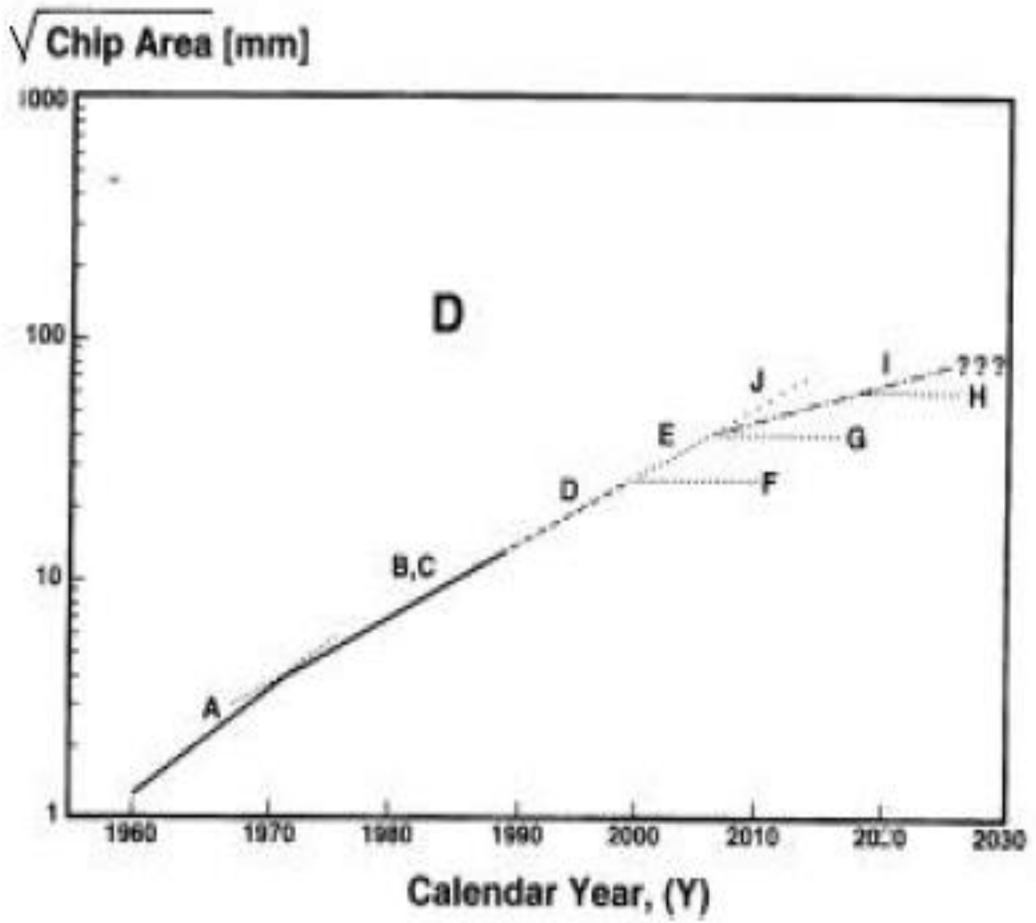
- Goals of scaling the dimensions by 30%:
  - » Reduce gate delay by 30% (increase operating frequency by 43%)
  - » Double transistor density
  - » Reduce energy per transition by 65% (50% power savings @ 43% increase in frequency
- Technology generation spans 2-3 years, but µP speed doubles every generation (not increased only by 43%)

Future size:



J. Meindl, Apr. 1995.
Proceedings IEEE

Future size:



1960: 1.2 mm
1980: 6.5-7mm
2000: 25-30mm
Driven by economy

Boolean functions:

A Boolean function described by an algebraic expression consists of binary variables, the constants 0 and 1, and the logic operation symbols.

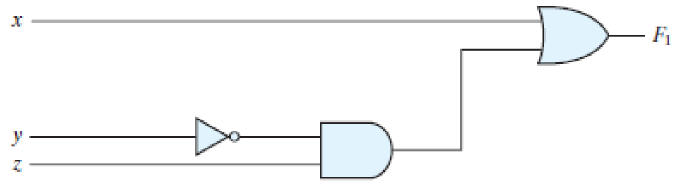
Example: $F_1 = x + y'.z$

A Boolean function can be represented in a truth table.

The number of rows in the truth table is 2^n, where *n* is the number of variables in the function. The binary combinations for the truth table are obtained from the binary numbers by counting from 0 through $2^n - 1$.

x	y	z	F_1
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

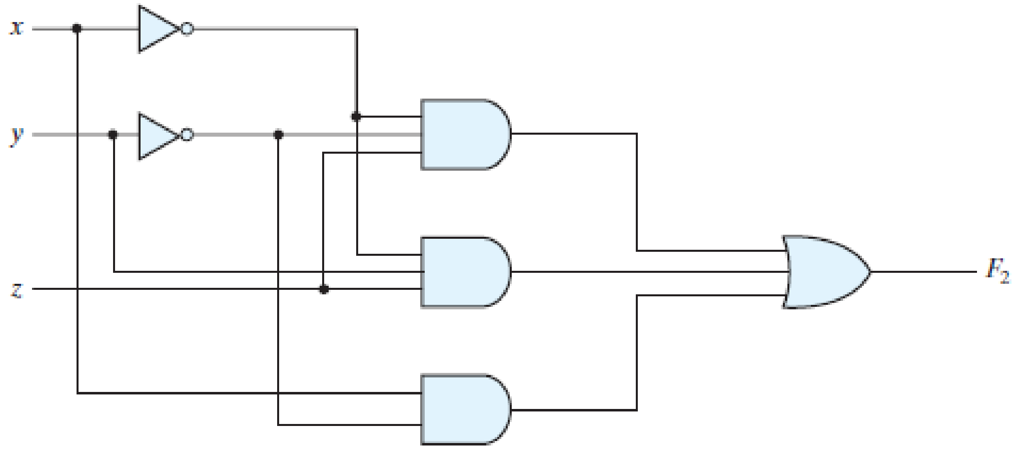
A Boolean function can be transformed from an algebraic expression into a circuit diagram composed of logic gates connected in a particular structure.



There is only one way that a Boolean function can be represented in a truth table. However, when the function is in algebraic form, it can be expressed in a variety of ways, all of which have equivalent logic.

Example: $F_2 = x'.y'.z + x'.y.z + x.y'$

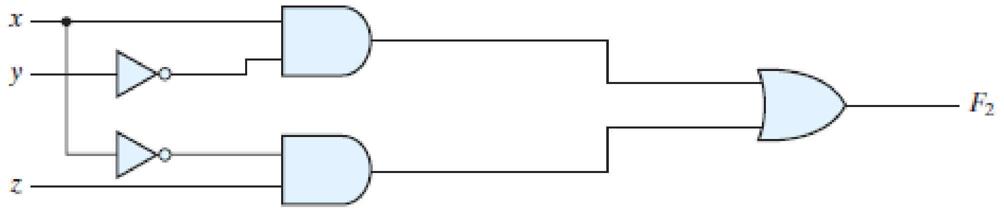
x	y	z	F_2
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0



By manipulating a Boolean expression, we can simplify the expression for the same function and thus reduce the number of gates in the circuit and the number of inputs to the gate.
Designers are motivated to reduce the complexity and number of gates because their effort can significantly reduce the cost of a circuit.

$$F_2 = x'.y'.z + x'.y.z + x.y' = x'.z + x.y'$$

Since both functions produce the same truth table, they are equivalent.



There are many equivalent representations of a logic function.
Finding the most economic representation of the logic is an important design task.

Algebraic manipulations:

When a Boolean expression is implemented with logic gates, each term requires a gate and each variable within the term designates an input to the gate. We define a *literal* to be a single variable within a term, in complemented or un-complemented form.

The function F_2 has three terms and eight literals. In its simplified form, it has two terms and four literals.

By reducing the number of terms, the number of literals, or both in a Boolean expression, it is often possible to obtain a simpler circuit. The manipulation of Boolean algebra consists mostly of reducing an expression for the purpose of obtaining a simpler circuit.

Boolean functions and many different outputs, designers of digital circuits use computer minimization programs that are capable of producing optimal circuits with millions of logic gates.

Forms of Boolean Expressions

- Sum-of-products form (SOP) – first the product (AND) terms are formed then these are summed (OR) – example, ABC + DEF + GHI
- Product-of-sum form (POS) – first the sum (OR) terms are formed then the products are taken (AND) – example, (A+B+C) (D+E+F) (G+H+I)
- It is possible to convert between these two forms using Boolean algebra (DeMorgan’s)

Canonical Form

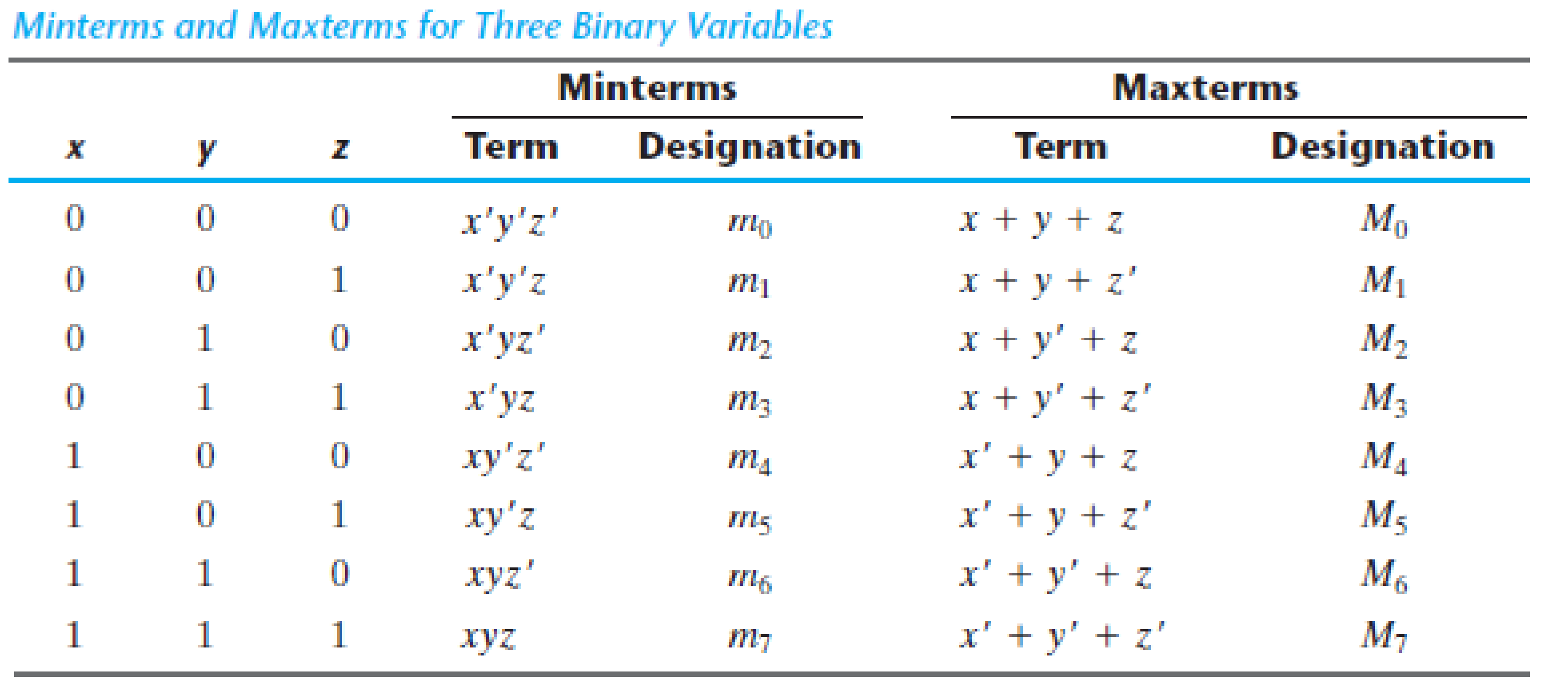
- Canonical form is not efficient but sometimes useful in analysis and design.
- In an expression in canonical form, every variable appears in every term – note that the dot (meaning AND) is often omitted,

$$f(A,B,C,D) = ABC'D + AB'CD + A'BCD'$$

Minterms and Maxterms:

n variables forming an AND term, with each variable being primed or unprimed, provide 2^n possible combinations, called *minterms*, or *standard product.*

n variables forming an OR term, with each variable being primed or unprimed, provide $2n$ possible combinations, called *maxterms*, or *standard sums.*



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

1. Simplify $(x + y).(x + y')$.

2. Simplify to a minimum number of literals,

(i). $(x.y + x'.z + y.z)$

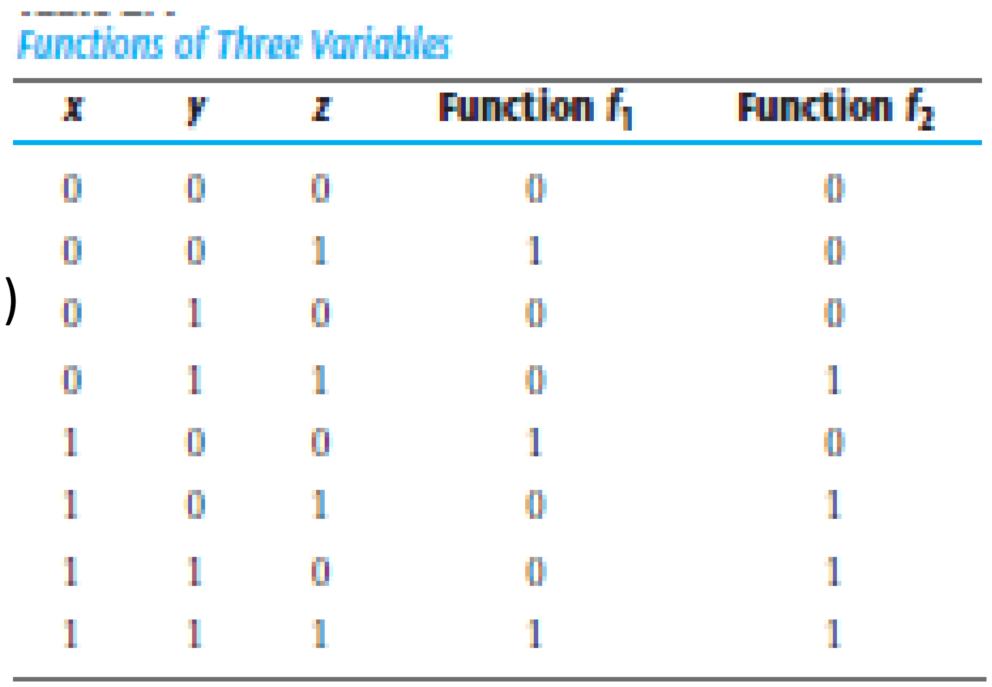
(ii). $(x+y). (x'+z). (y+z)$

3. $F_1 = x'.y.z' + x'.y'.z$, *Find the complement of* F_1.

A Boolean function can be expressed algebraically from a given truth table by forming a minterm for each combination of the variables that produces a 1 in the function and then taking the OR of all those terms.

$F_1(x,y,z)=x'y'z+xy'z'+xyz=m_1+m_4+m_7=\Sigma(1, 4, 7)$

$F_2(x,y,z)=x'yz+xy'z+xyz'+xyz=m_3+m_5+m_6+m_7=\Sigma(3, 5, 6, 7)$


Functions of Three Variables

x	y	z	Function f_1	Function f_2
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

An important property of Boolean algebra:

Any Boolean function can be expressed as a sum of minterms (with "sum" meaning the ORing of terms).

$F_1'=x'y'z'+x'yz'+x'yz+xy'z+xyz'$

$(F_1')'=(x+y+z).(x+y'+z).(x+y'+z').(x'+y+z').(x'+y'+z)=M_o.M_2.M_3.M_5.M_6$
$=\Pi\,(0, 2, 3, 5, 6)$

Similarly, do for F_2.

Boolean functions expressed as a sum of minterms or product of maxterms are said to be in *canonical form* .

The minterms whose sum defines the Boolean function are those which give the 1’s of the function in a truth table.

Conversion between Canonical Forms:

The maxterm with subscript *j* is a complement of the minterm with the same subscript *j* and vice versa.

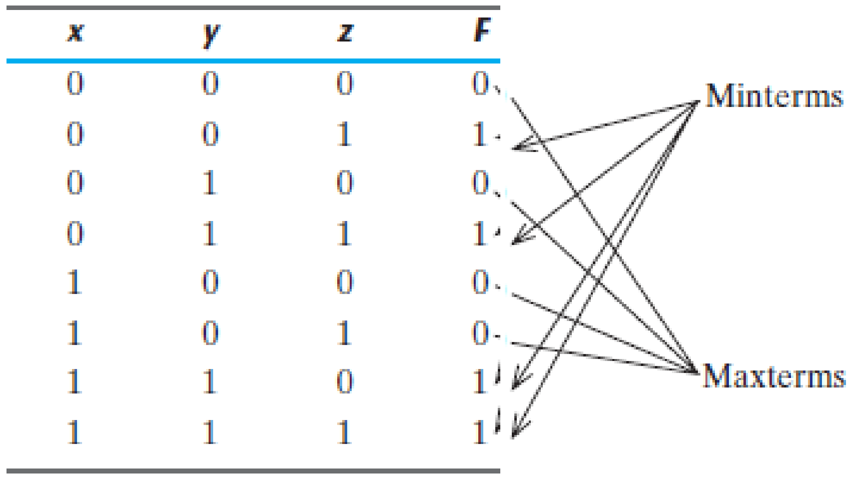
$F(A, B, C) = \Sigma(1, 4, 5, 6, 7)$

$F'(A,B,C) = \Sigma(0, 2, 3) = m0 + m2 + m3$

$(F')' = F = (m0 + m2 + m3)' = m_0'.\ m_2'.\ m_3' = M_0 M_2 M_3 = \Pi(0, 2, 3)$

Example:

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


Minterms
Maxterms

x	y	z	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

$$F(x, y, z) = \Sigma(1, 3, 6, 7)$$

$$F(x, y, z) = \Pi\ (0, 2, 4, 5)$$

- An SOP expression can be forced into canonical form by ANDing the incomplete terms with terms of the form (A+A’) where A is the name of the missing variable.

  Example: $f(A,B,C)= AB+BC=AB(C+C')+(A+A')BC$

  $= ABC+ABC'+ABC+A'BC$

  $=ABC+ABC'+A'BC$

  The product term in a canonical SOP expression is called a 'minterm‘.

Another way to express Boolean functions is in *standard* form. In this configuration, the terms that form the function may contain one, two, or any number of literals.

There are two types of standard forms: the sum of products and products of sums.The *sum of products* is a Boolean expression containing AND terms, called *product terms,* with one or more literals each.

The *sum* denotes the ORing of these terms.

Example,

$F_1 = y' + x.y + x'.y.z'$

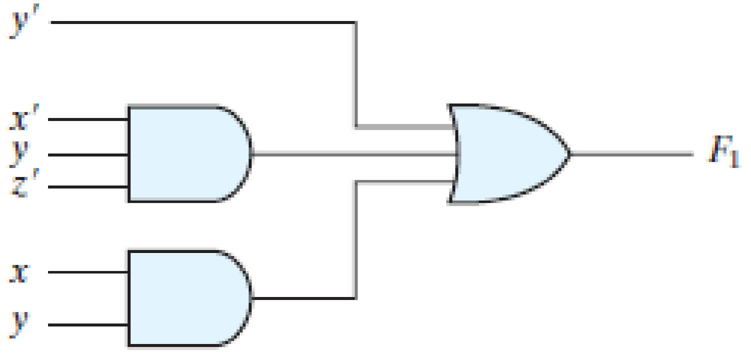
The expression has three product terms, with one, two, and three literals. Their sum is an OR operation.

A *product of sums* is a Boolean expression containing OR terms, called *sum terms*.
Each term may have any number of literals. The *product* denotes the ANDing of these terms. An example of a function expressed as a product of sums is

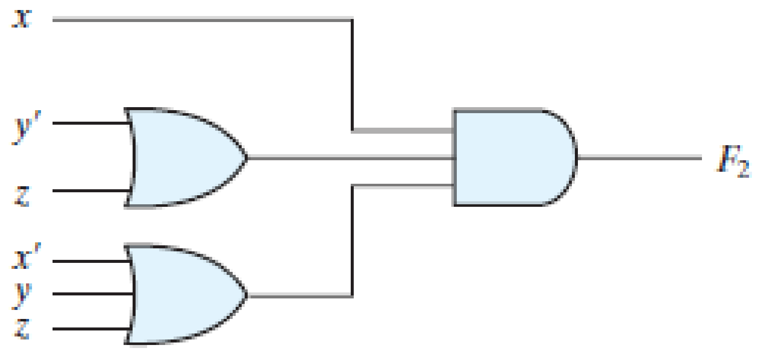
$$F_2 = x.(y' + z).(x' + y + z')$$

This expression has three sum terms, with one, two, and three literals. The product is an AND operation.

Two-level implementation:

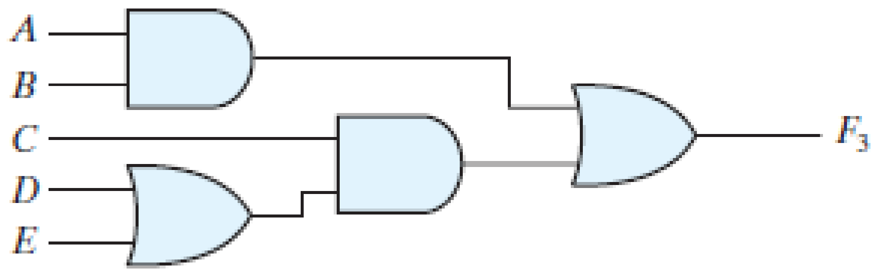


(a) Sum of Products

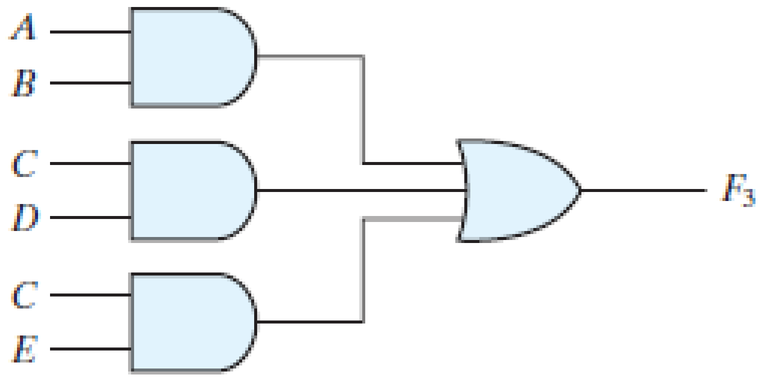


(b) Product of Sums

Three- and two-level implementation:



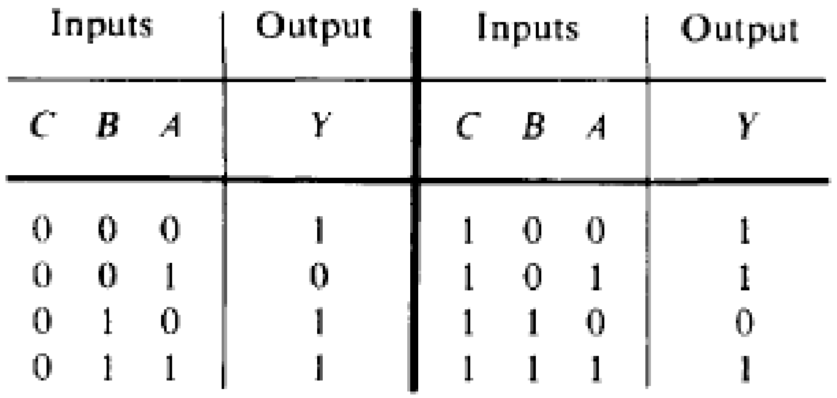
(a) $AB + C(D + E)$



(b) $AB + CD + CE$

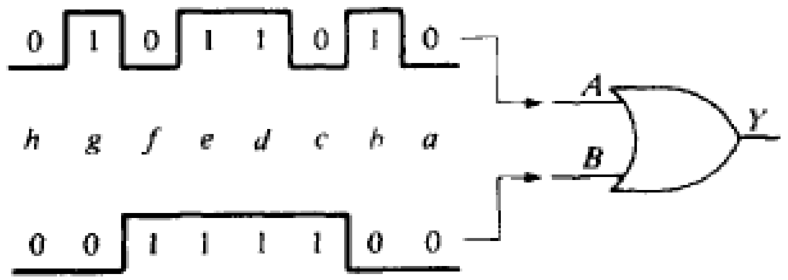
Examples:

Write a max-term representation,



Inputs			Output	Inputs			Output
C	B	A	Y	C	B	A	Y
0	0	0	1	1	0	0	1
0	0	1	0	1	0	1	1
0	1	0	1	1	1	0	0
0	1	1	1	1	1	1	1

Draw the output pulse train,



Gate level Minimization:

Since a realization of a logic function (even one as simple as exclusive-OR) is not unique, it is often desirable to find the simplest, or perhaps most conveniently constructed, circuit for a given function. Many good minds have worked on this problem, and there are several methods available, including algebraic techniques that can be coded to run on a computer.

1. Minimization by Boolean Algebra:
   - Make use of relationships and theorems to simplify Boolean expressions – perform algebraic manipulation resulting in a complexity reduction – this method relies on your algebraic skill – 3 things to try

2. Minimization by Karnaugh Maps:

For problems with four or fewer inputs, a Karnaugh map provides one of the nicest methods; it also enables you to find a logic expression (if you don't know it) once you can write down the truth table.

Karnaugh maps (K-map):

The K-map method provides a simple, straightforward procedure for minimizing Boolean functions.

“A Karnaugh map is a two-dimensional truth-table. Unlike ordinary (i.e., one-dimensional) truth tables”.

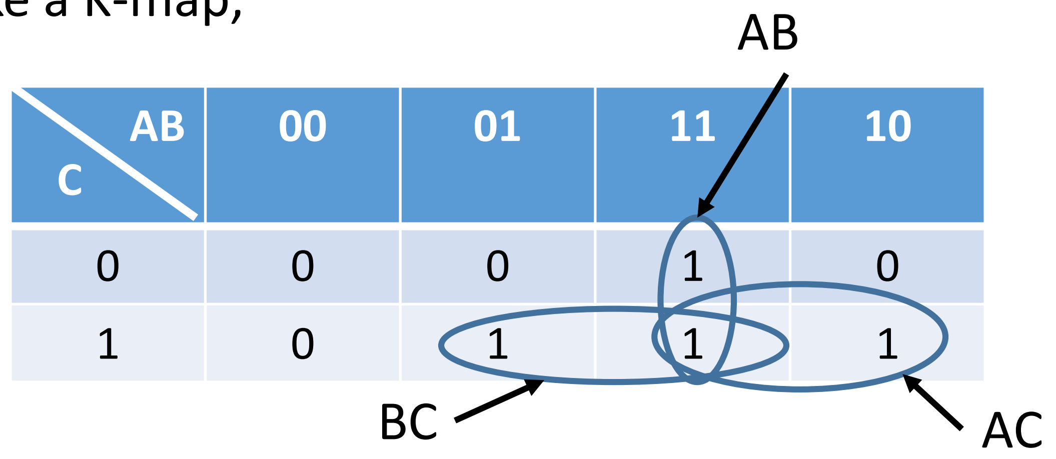
Example: Suppose you want to generate a logic circuit to count votes and you have three positive-true inputs, each ether (1 or 0) and output (0 or 1). The output is to be 1 if at least two of the inputs are 1.

Step 1: Make a truth table,

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

Write an X (= "don't care") if either output state is OK.

Step 2: Make a K-map,



C \ AB	00	01	11	10
0	0	0	1	0
1	0	1	1	1

…represent the variables along two axes in such a way that only one input bit changes in going from one square to an adjacent square.

Step 3: Identify on the map groups of 1’s,*

**alternatively, you could use groups of 0’s which will result,*
Q’= A’.B’+B’.C’+A’.C’

The three blobs enclose the logic expressions *AB,AC,* and *BC.* Finally, read off the required function,

Q= A.B+B.C+A.C

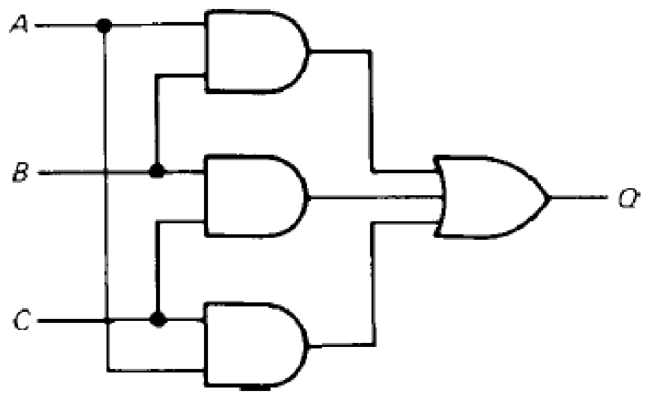
AB

AB / C	00	01	11	10
0	0	0	1	0
1	0	1	1	1

BC

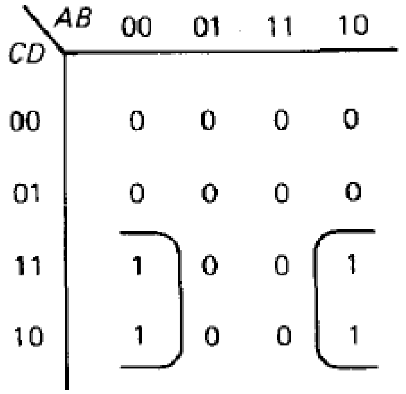
AC

Which can be realized as,

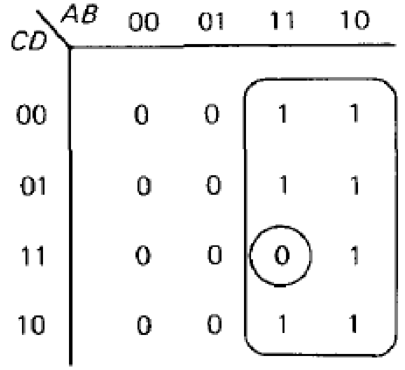


More comments on K-map:

1. Look for groups of 2, 4, 8, etc., squares; they have the simple logic expressions.
2. The larger the block you describe, the simpler the logic.
3. The edges of the Karnaugh map connect up. For instance, the map (shown below) is described by Q = *B′C.*



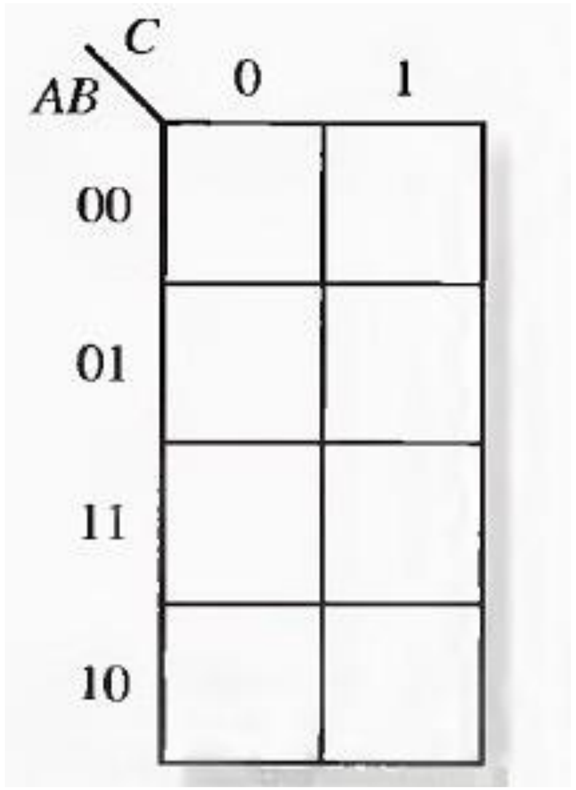
4. A block of with only one or two may be best described by the grouping shown below, which corresponds to the logic expression *Q* = A(BCD)′.
5. Xs (don’t care) are "wild cards." Use them as 1’s or 0’s to generate the simplest logic.
6. A K-map may not lead directly to the best solution: A more complicated logic expression may sometimes have a simpler realization in gates, if some of its terms already exist as logic in your circuit, and you can exploit intermediate outputs (from other terms) as inputs.



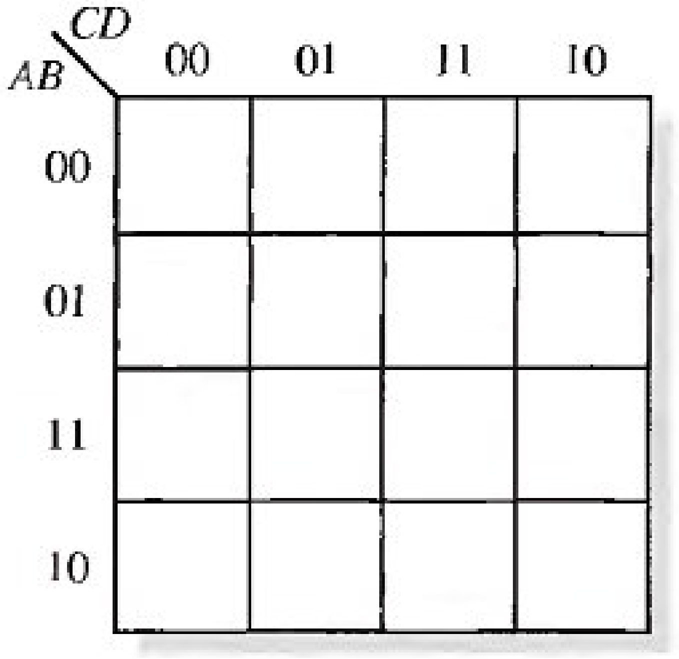
Furthermore, exclusive-OR realizations are not always obvious from K-maps. Finally, package constraints (e.g, the fact that four 2-input gates come in a single (IC) also figure into the choice of logic used in the final circuit realization.

Cell Adjacency

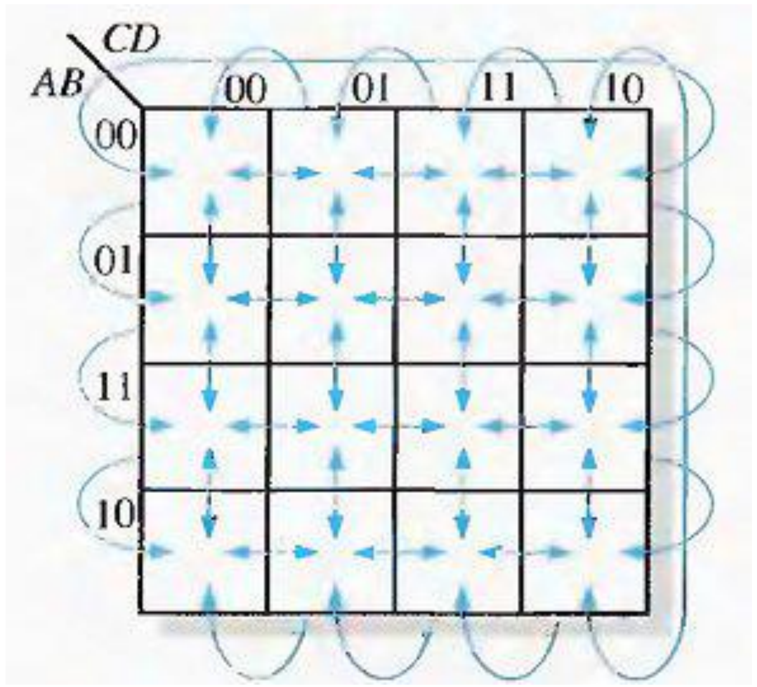
K-map with 3 variables:



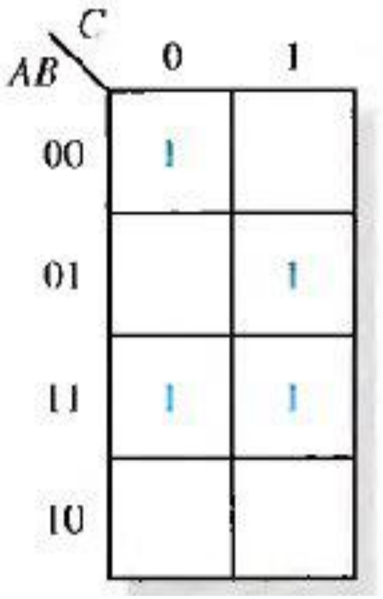
K-map with 4 variables:



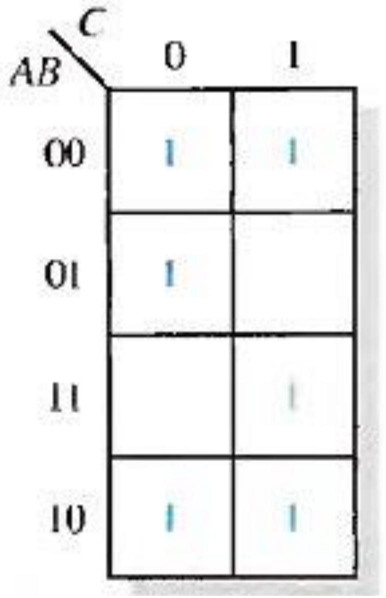
Adjacency is defined by a single variable change.



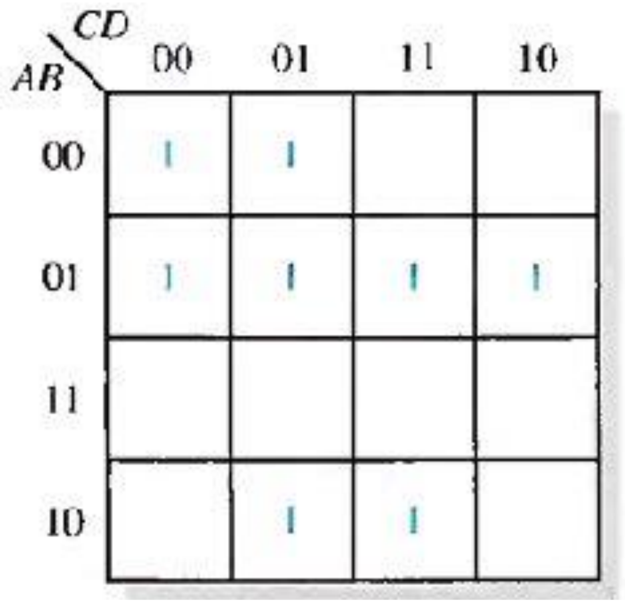
Group the 1’s:



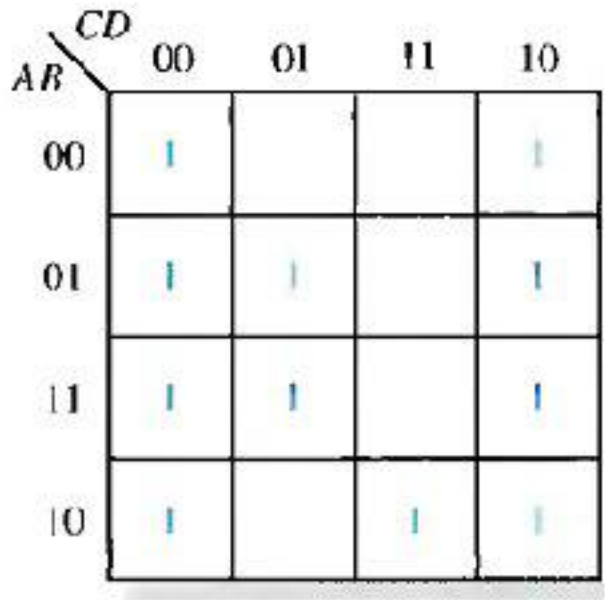
AB \ C	0	1
00	1	
01		1
11	1	1
10		



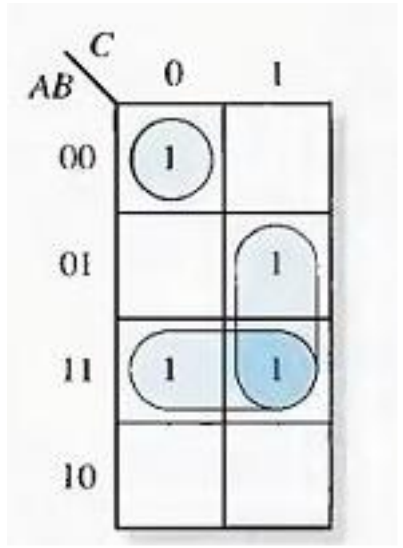
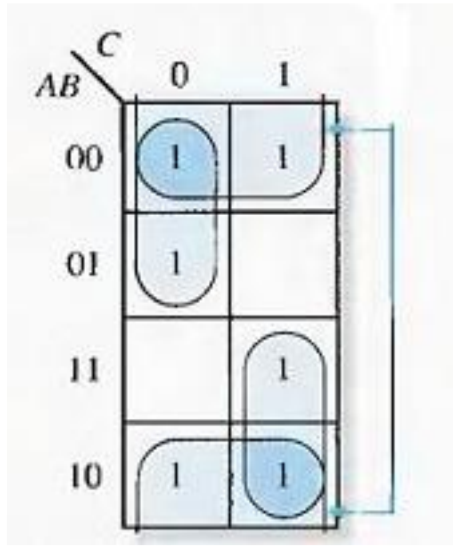
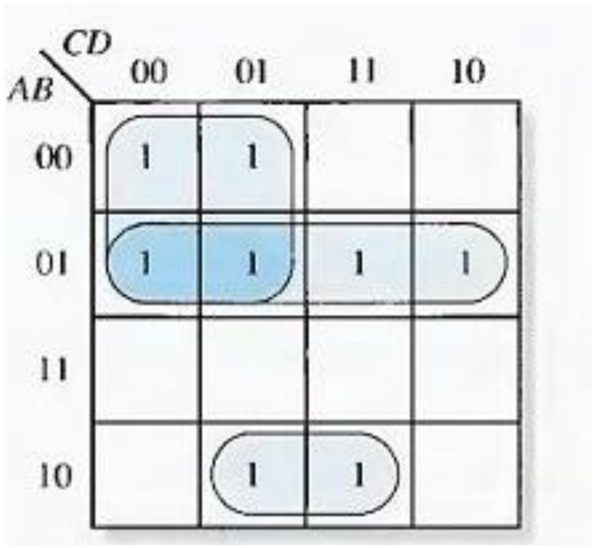
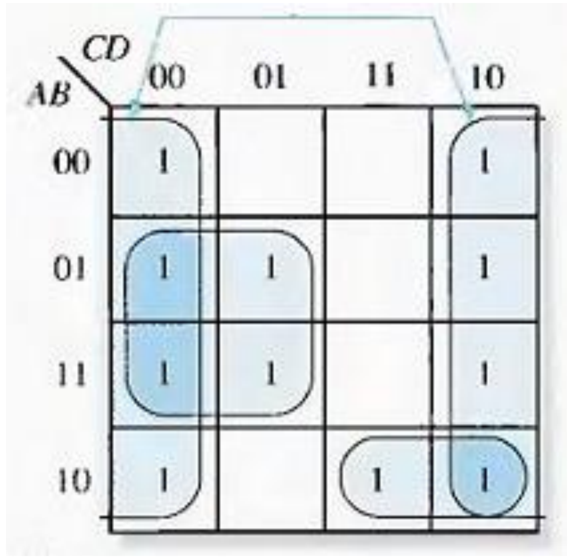
AB \ C	0	1
00	1	1
01	1	
11		1
10	1	1



AB \ CD	00	01	11	10
00	1	1		
01	1	1	1	1
11				
10		1	1	



AB \ CD	00	01	11	10
00	1			1
01	1	1		1
11	1	1		1
10	1		1	1

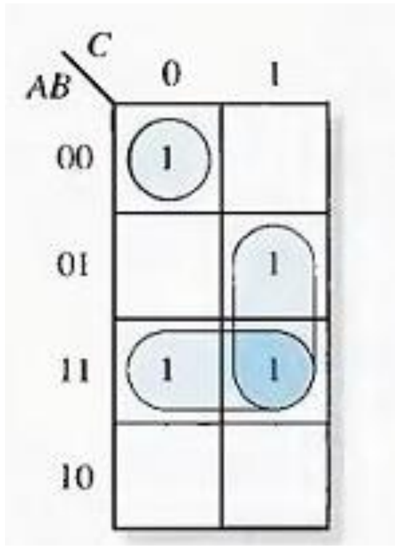





For a 3-variable map:

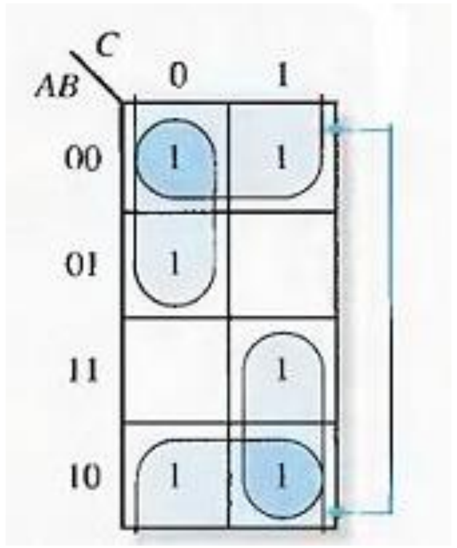
(1) A 1-cell group yields a 3-variable product term
(2) A 2-cell group yields a 2-variable product term
(3) A 4-cell group yields a 1-variable term
(4) An 8-cell group yields a value of 1 for the expression

For a 4-variable map:

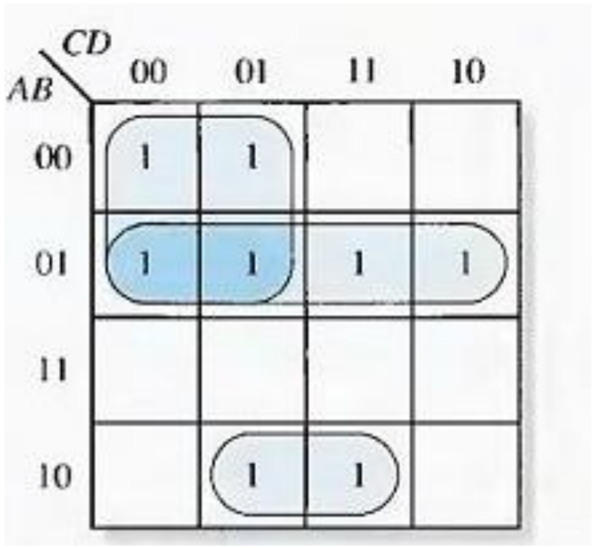
(1) A 1-cell group yields a 4-variable product term
(2) A 2-cell group yields a 3-variable product term
(3) A 4-cell group yields a 2-variable product term
(4) An 8-cell group yields a 1-variable term
(5) A 16-cell group yields a value of 1 for the expression



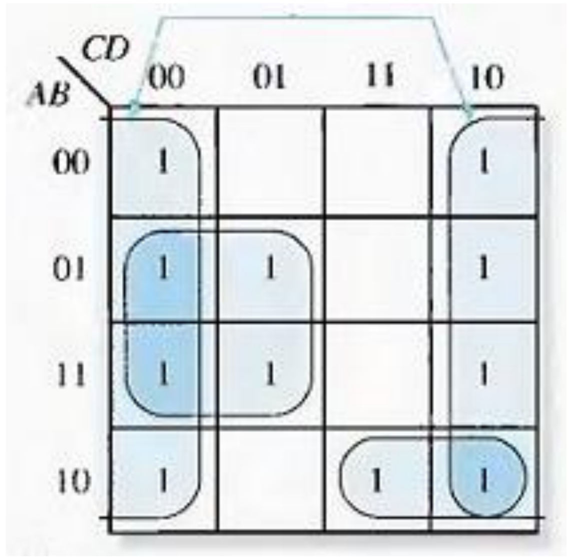
AB+BC+A’B’C’



B’ + AC + A’C’



A’B + A’C’ + AB’D



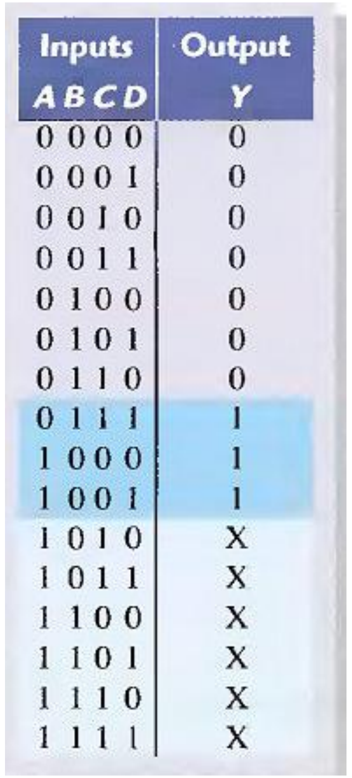
D’ + AB’C + BC’

Sometimes a situation arises in which some input variable combinations are not allowed. For example, in the BCD code there are six invalid combinations: 1010, 1011, 1100, 1101, 1110, and 1111. Since these unallowed states will never occur in an application involving the BCD code, they can be treated as "don't care" terms with respect to their effect on the output.

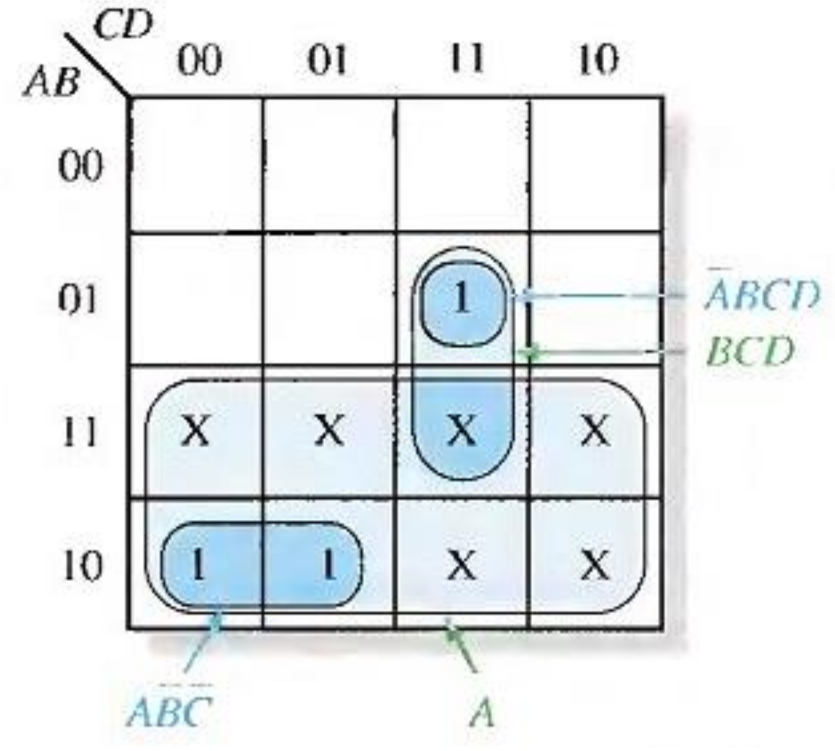
…for these "don't care" terms either a 1 or a 0 may be assigned to the output: it really does not matter since they will never occur.

The "don't care" terms can be used to advantage on the Karnaugh map.

When grouping the 1 s, the Xs can be treated as 1s to make a larger grouping or as 0s if they cannot be used to advantage. The larger a group, the simpler the resulting term will be.



Inputs	Output
A B C D	*Y*
0 0 0 0	0
0 0 0 1	0
0 0 1 0	0
0 0 1 1	0
0 1 0 0	0
0 1 0 1	0
0 1 1 0	0
0 1 1 1	1
1 0 0 0	1
1 0 0 1	1
1 0 1 0	X
1 0 1 1	X
1 1 0 0	X
1 1 0 1	X
1 1 1 0	X
1 1 1 1	X

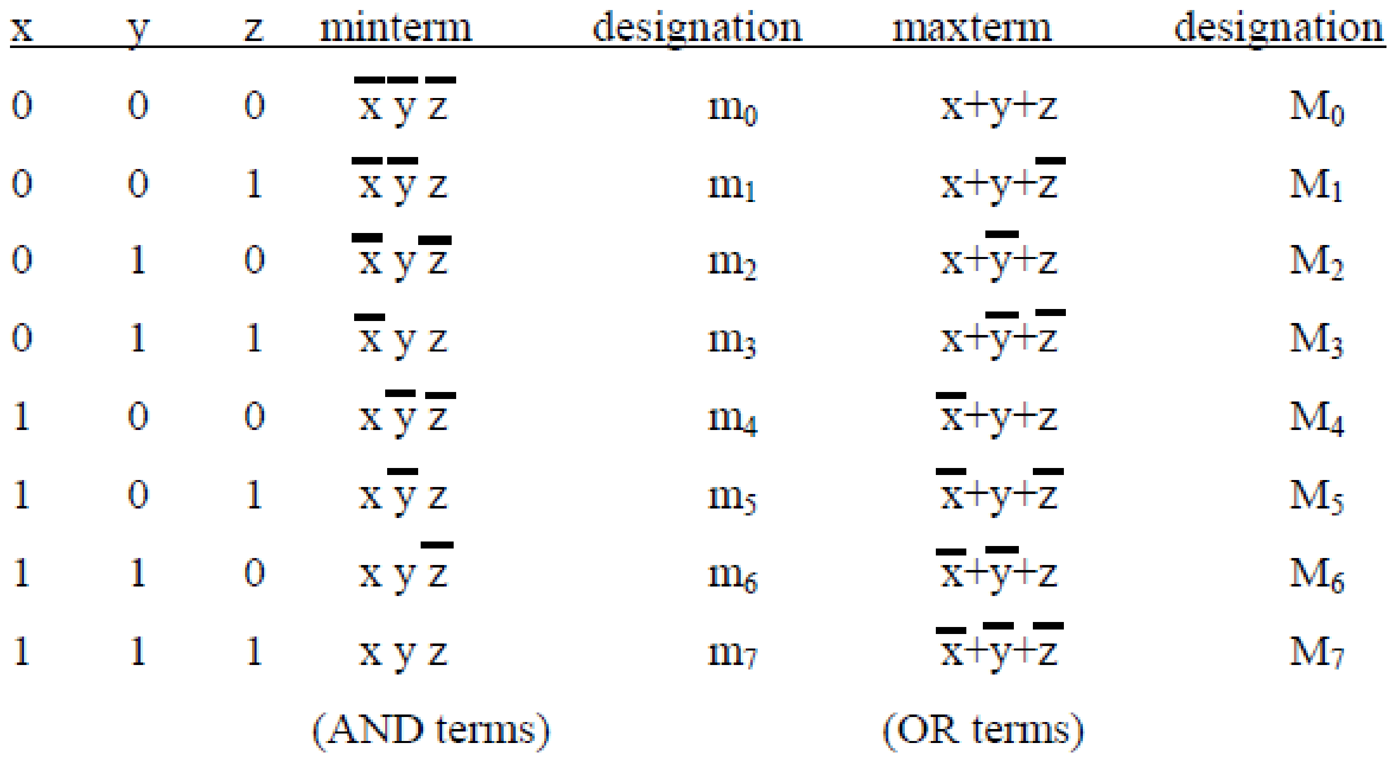


Without don’t cares: F=AB’C’+A’BCD
With don’t cares: F=A+BCD

Minterm and Maxterm:

With one variable, x & x'.
With two variables, x' y', x y', x' y and x y.
With three variables, x' y' z', x' y' z, x' y z', x y' z', x y z', x y' z, x' y z & x y z.

These eight AND terms are called minterms.
n variables can be combined to form 2^n minterms.



x	y	z	minterm	designation	maxterm	designation
0	0	0	$\bar{x}\,\bar{y}\,\bar{z}$	m_0	$x+y+z$	M_0
0	0	1	$\bar{x}\,\bar{y}\,z$	m_1	$x+y+\bar{z}$	M_1
0	1	0	$\bar{x}\,y\,\bar{z}$	m_2	$x+\bar{y}+z$	M_2
0	1	1	$\bar{x}\,y\,z$	m_3	$x+\bar{y}+\bar{z}$	M_3
1	0	0	$x\,\bar{y}\,\bar{z}$	m_4	$\bar{x}+y+z$	M_4
1	0	1	$x\,\bar{y}\,z$	m_5	$\bar{x}+y+\bar{z}$	M_5
1	1	0	$x\,y\,\bar{z}$	m_6	$\bar{x}+\bar{y}+z$	M_6
1	1	1	$x\,y\,z$	m_7	$\bar{x}+\bar{y}+\bar{z}$	M_7
			(AND terms)		(OR terms)	

Each maxterm is the complement of its corresponding minterm and vice versa.

Any Boolean function can be expressed as a sum of minterms (sum of products **SOP**) or product of maxterms (product of sums **POS**).

Example, A function F such that,

x	y	z	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

$F = x\,y\,z + x'\,y'\,z + x\,y'\,z'$

$F = m_1 + m_4 + m_7$

Express the Boolean function F = A + B’C in a sum of minterms (SOP).

The complement of a function expressed as the sum of minterms equal to the sum of minterms missing from the original function.

Express F = xy + x’z in a product of maxterms form.

The complement of a function expressed as the product of Maxterms equal to the product of maxterms missing from the original function.

To convert from one canonical form to another, interchange the symbols Σ, Π and list those numbers missing from the original form.

$F = M_4 M_5 M_0 M_2 = m_1 + m_3 + m_6 + m_7$

$F(x, y, z) = \Pi(0, 2, 4, 5) = \Sigma(1, 3, 6, 7)$

K-map and minterms

Three variable K-map,

m_0	m_1	m_3	m_2
m_4	m_5	m_7	m_6

One square represents one minterm, giving a term with three literals.
Two adjacent squares represent a term with two literals.
Four adjacent squares represent a term with one literal.
Eight adjacent squares encompass the entire map and produce a function that is always equal to 1.

Four variable K-map,

m_0	m_1	m_3	m_2
m_4	m_5	m_7	m_6
m_{12}	m_{13}	m_{15}	m_{14}
m_8	m_9	m_{11}	m_{10}

One square represents one minterm, giving a term with four literals.
Two adjacent squares represent a term with three literals.
Four adjacent squares represent a term with two literals.
Eight adjacent squares represent a term with one literal.
Sixteen adjacent squares produce a function that is always equal to 1.

Simplify the Boolean function,

$$F(A,B,C)=\Sigma(2,3,4,5)$$
$$F(A,B,C)=\Sigma(3,4,6,7)$$
$$F(A, B, C, D) = \Sigma(0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14)$$

References:

- Art of Electronics, Paul Horowitz.
- Digital Design, Morris Mano, M. D. Ciletti
- Web sources (like NPTEL etc).