MODULE-4

Memory and programmable logic

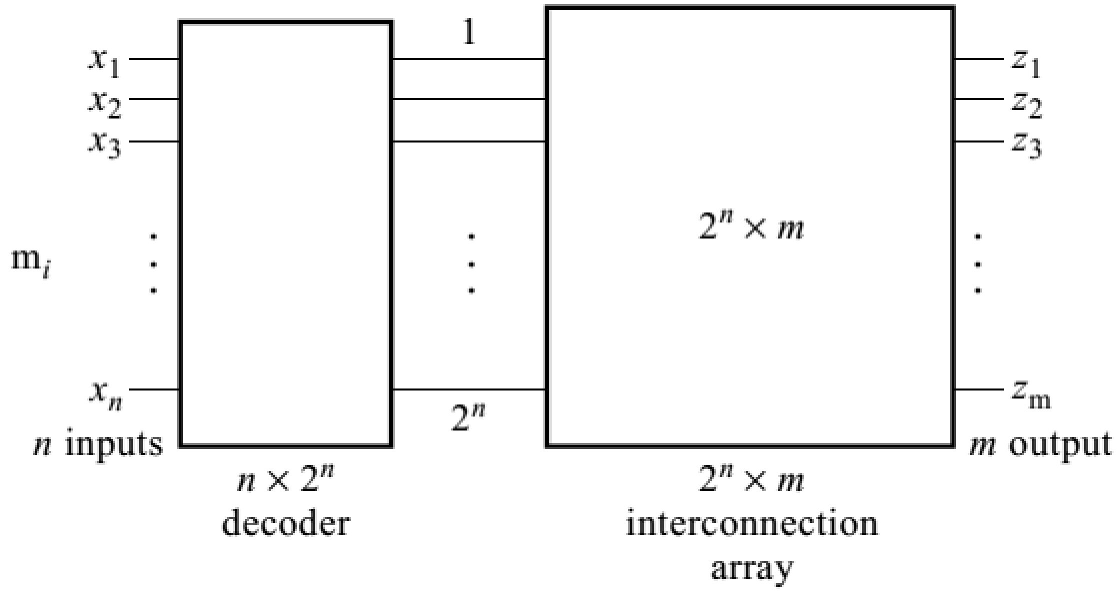
READ-ONLY MEMORY (ROM)

A read-only memory (ROM) is a device that includes both the decoder and the OR gates within a single IC package. The connections between the outputs of the decoder and the inputs of the OR gates can be specified for each particular configuration.

A ROM is essentially a memory (or storage) device in which permanent binary information is stored. The binary information must be specified by the designer and is then embedded in the unit to form the required interconnection pattern. ROMs come with special internal electronic fuses that can be "programmed" for a specific configuration. Once the pattern is established, it stays within the unit even when power is turned off and on again.

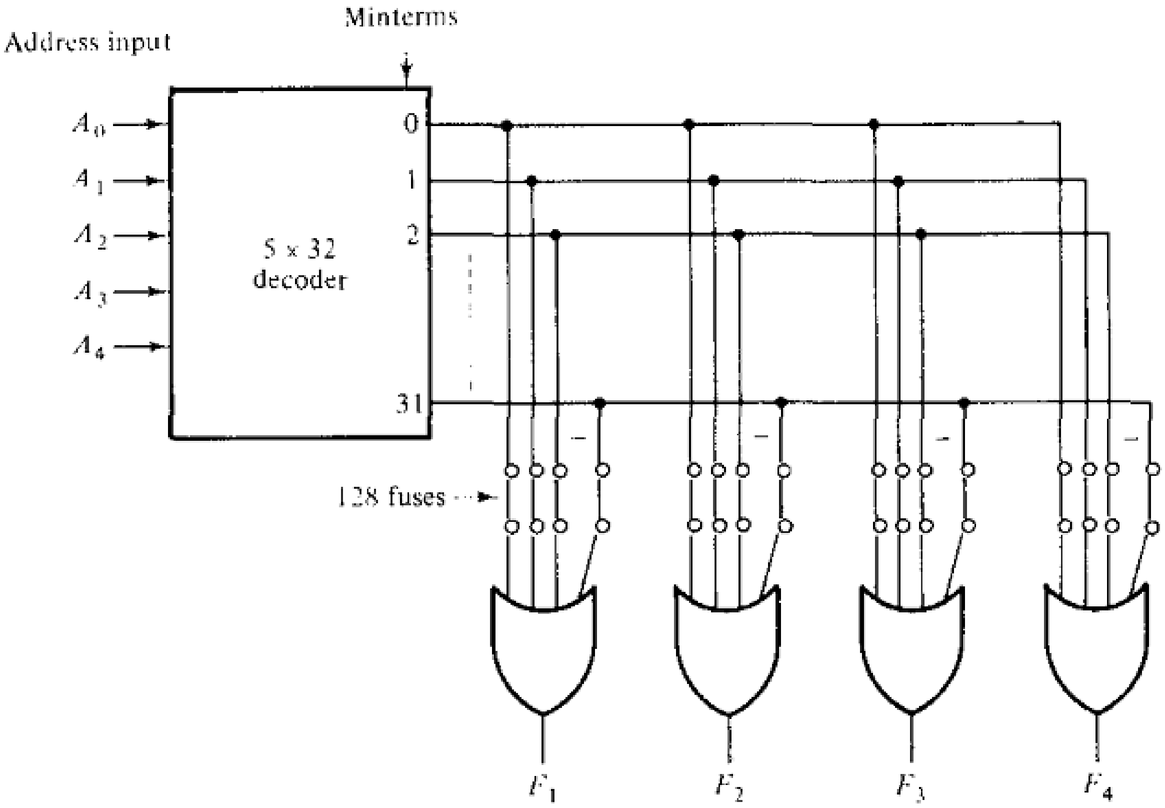
It consists of n input lines and m output lines. Each bit combination of the input variables is called an address. Each bit combination that comes out of the output lines is called a word. The number of bits per word is equal to the number of output lines, m. An address is essentially a binary number that denotes one of the min terms of n variables. The number of distinct addresses possible with n input variables is 2^n. An output word can be selected by a unique address, and since there are 2^n distinct addresses in a ROM, there are 2^n distinct words that are said to be stored in the unit.

The word available on the output lines at any given time depends on the address value applied to the input lines. A ROM is characterized by the number of words 2^n and the number of bits per word m.



Basic structure of ROM

Consider a 32 x 8 ROM. The unit consists of 32 words of 8 bits each. This means that there are eight output lines and that there are 32, distinct words stored in the unit, each of which may be applied to the output lines. The particular word selected that is presently available on the output lines is determined from the five input lines. There are only five inputs in a 32 x 8 ROM because $2^5 = 32$, and with five variables, we can specify 32 addresses or min terms. For each address input, there is a unique selected word. Thus, if the input address is 00000, word number 0 is selected and it appears on the output lines. If the input address is 11111, word number 31 is selected and applied to the output lines. In between, there are 30 other addresses that can select the other 30 words.



ROM Application:-

Preprogrammed toy circuit,

- Preprogrammed robot circuit,
- Standard look up table,
- Arithmetic function table generator,
- User defined code generator,
- Character generator,
- Printable or displayable fonts table

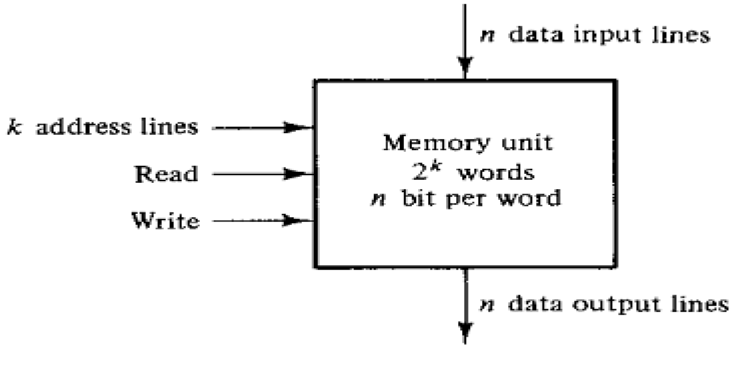
RANDOM-ACCESS MEMORY (RAM):-

A memory unit is a collection of storage cells together with associated circuits needed to transfer information in and out of the device. Memory cells can be accessed for information transfer to or from any desired random location and hence the name random access memory, abbreviated RAM.

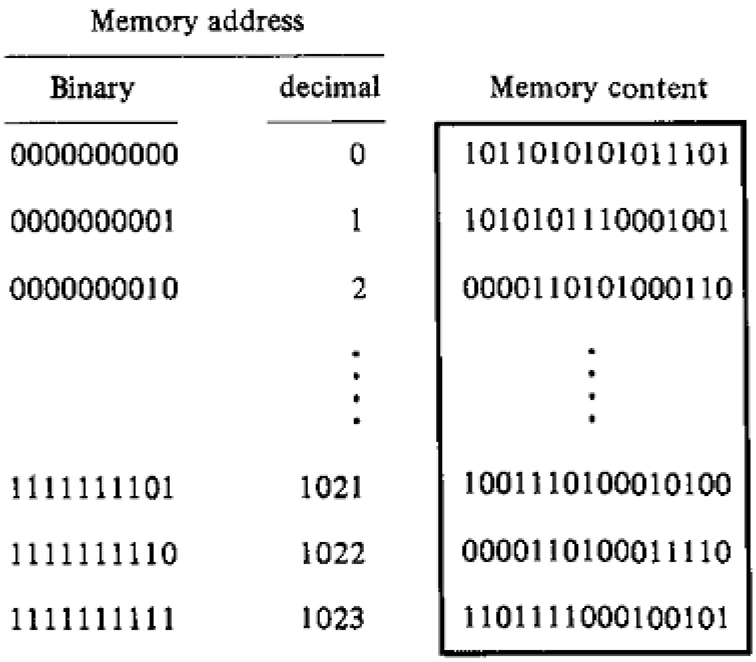
A memory unit stores binary information in groups of bits called words. A word in memory is an entity of bits that move in and out of storage as a unit. A memory word is a group of 1's and 0's and may represent a number, an instruction, one or more alphanumeric characters, or any other binary-coded information. A group of eight bits is called a byte. Most computer memories use words that are multiples of 8 bits in length. Thus, a 16-bit word contains two bytes, and a 32-bit word is made up of four bytes. The capacity of a memory unit is usually stated as the total number of bytes that it can store.

The communication between a memory and its environment is achieved through data input and output lines, address selection lines, and control lines that specify the direction of transfer .The n data input lines provide the information to be stored in memory and the n data output lines supply the information coming out of memory. The k address lines specify the particular word chosen among the many available. The two control inputs specify the direction of transfer desired: The write input causes binary data to be transferred into the memory, and the read input causes binary data to be transferred out of memory.

The memory unit is specified by the number of words it contains and the number of bits in each word. The address lines select one particular word. Each word in memory is assigned an identification number, called an address, starting from 0 and continuing with 1, 2, 3,upto 2^k - 1, where k is the number of address lines. The selection of a specific word inside the memory is done by applying the k-bit binary address to the address lines.



Basic structure of RAM



Memory address		Memory content
Binary	decimal	
0000000000	0	1011010101011101
0000000001	1	1010101110001001
0000000010	2	0000110101000110
	⋮	⋮
1111111101	1021	1001110100010100
1111111110	1022	0000110100011110
1111111111	1023	1101111000100101

Content of a 1024 x 16 memory

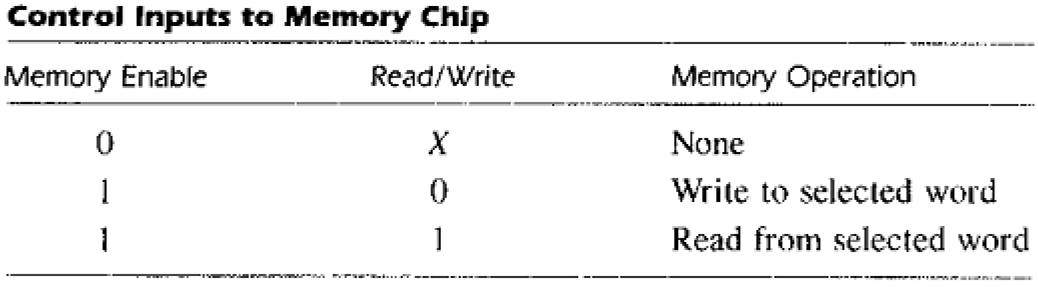
Write and Read Operations

The two operations that a random-access memory can perform are the write and read operations. The write signal specifies a transfer-in operation and the read signal specifies a transfer-out operation. On accepting one of these control signals, the internal circuits inside the memory provide the desired function. The steps that must be taken for the purpose of transferring a new word to be stored into memory are as follows:

1. Transfer the binary address of the desired word to the address lines.

2. Transfer the data bits that must be stored in memory to the data input lines.

3. Activate the write input.



Control Inputs to Memory Chip

Memory Enable	Read/Write	Memory Operation
0	X	None
1	0	Write to selected word
1	1	Read from selected word

The memory unit will then take the bits from the input data lines and store them in the word specified by the address lines. The steps that must be taken for the purpose of transferring a stored word out of memory are as follows:

1. Transfer the binary address of the desired word to the address lines.

2. Activate the read input.

The memory unit will then take the bits from the word that has been selected by the address and apply them to the output data lines. The content of the selected word does not change after reading.

Types of Memories

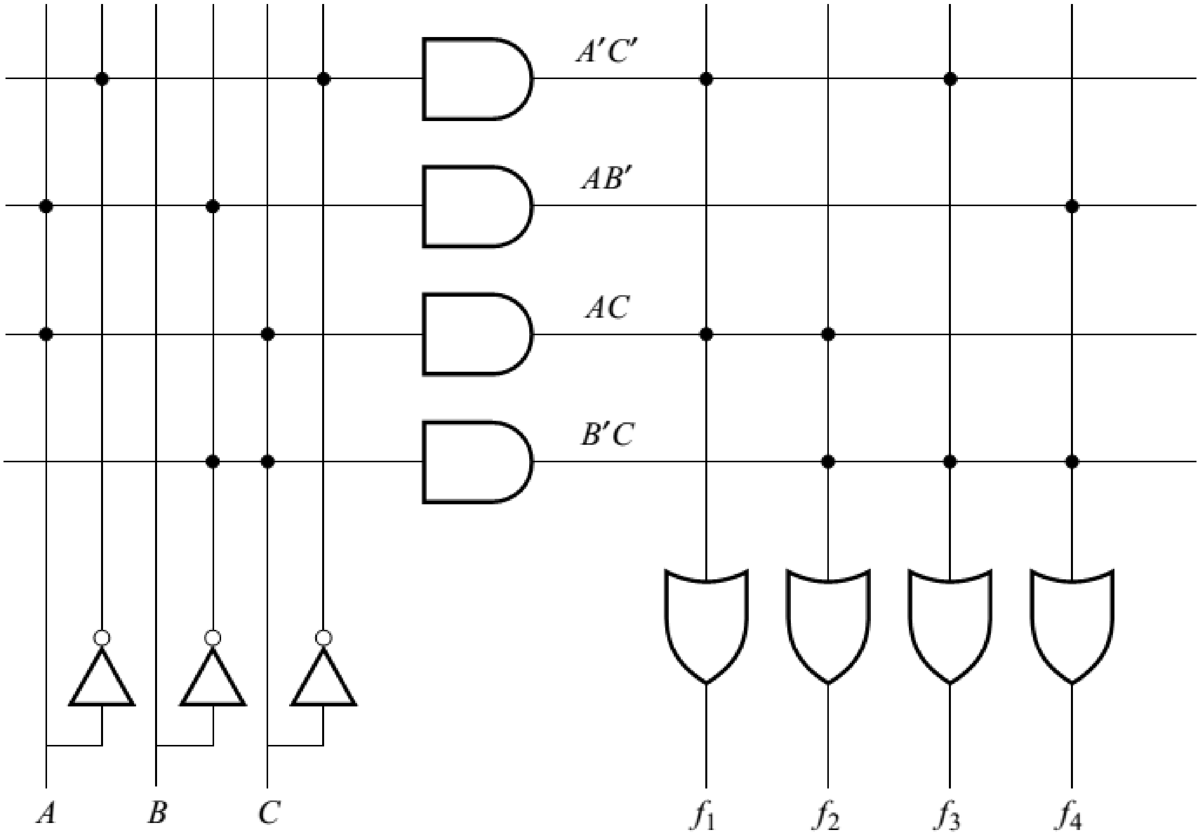
Integrated-circuit RAM units are available in two possible operating modes, static and dynamic. The static RAM consists essentially of internal flip-flops that store the binary information. The stored information remains valid as long as power is applied to the unit. The dynamic RAM stores the binary information in the form of electric charges that are applied to capacitors. The capacitors are provided inside the chip by MOS transistors. The stored charge on the capacitors tends to discharge with time and the capacitors must be periodically recharged by refreshing the dynamic memory. Refreshing is done by cycling through the words every few milliseconds to restore the decaying charge. Dynamic RAM offers reduced power consumption and larger storage capacity in a single memory chip, but static RAM is easier to use and has shorter read and write cycles.

Programmed Logic Array (PLA) :-

The canonic sum-of-products implementation of a logic function is wasteful in two ways: in the number of AND gates used (as many as there are min terms, 2^n) and in the number of inputs to each AND gate n .Suppose we contemplate a reduced (possibly minimal) sum-of-products implementation. Given a logic function of n variables, the largest number of terms in a minimal sum-of-products expression representing this function is 2^{n-1}-just half the number of min terms.

That means a savings of 50 percent in AND gates for the worst single-output case. Since there will be a reduced set of inputs to the AND gates, this saving in gates is paid for by the need to program not only the outputs of the AND gates but their inputs as well. The structure of the circuit that results is called a programmable (or programmed) logic array (PLA).

The diagram in given Figure is not a circuit diagram but a schematic diagram. A single line is shown to represent all inputs to each AND and OR gate. The number of input lines to each AND gate should be 2n, twice the number of inputs, to accommodate the possibility of connecting each variable or its complement to each AND gate. The number of input lines to each OR gate should equal the number of AND gates, say p. (For simplicity and without fear of confusion, even the gate symbols can be omitted.) The programmed connections between the inputs and the AND gates, and between the AND-gate outputs and the OR gates for a specific set of output functions are shown by the heavy dots at the intersections.



Structure of a PLA.

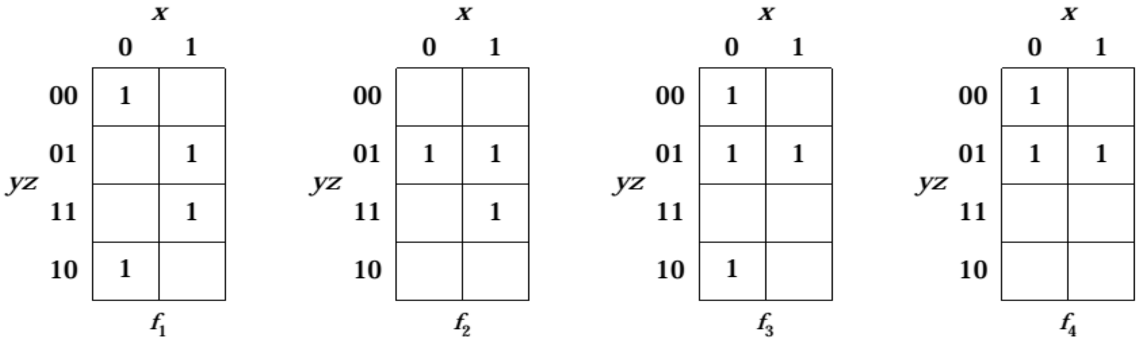
Maps of the four output functions and minimal sum-of-products expressions are shown in below Figure. In this example, a total of only four product terms covers all functions, so only four AND gates are needed in the implementation. Two sets of lines must be programmed: the input lines and the output lines. To do this, we construct a programming table as follows:

- The implicants (product terms) are listed as row headings.
- In one set of columns, the headings are the input variables; this part of the table must provide the information that tells which variables (or their complements) are factors in each implicant.
- In a second set of columns, the headings are the output functions; this part of the table must provide the information that indicates the output gate to which each implicant (AND-gate output) is directed.

In the first set of columns, if a variable (uncomplemented) is present in a particular row, the corresponding entry is 1; if its complement is present, the entry is 0. If neither is present, the entry can be left blank, but it is preferable to show some symbol instead; a dash is often used.

In the second set of columns, corresponding to the output functions, if a particular function covers a particular implicant, then the corresponding entry is 1; otherwise it could be left blank, but it is customary to enter a dot. To illustrate, consider row 4. Since the implicant is y'z, the entry in column z is 1, that in column y is 0, and that in x is a dash. In the output columns, only f_1 does not cover implicant y'z; hence, the entry will be 1 in every column in row 4 except the f_1 column. Confirm the remaining rows.

Once the programming is done, fabricating the links (connection points) in a PLA is carried out in a similar manner as for the ROM. The PLA is either mask programmable or field programmable (FPLA). In the case of the FPLA, with p= the number of AND gates, there will be 2np links at the inputs and mp links at the outputs.



yz \ x	0	1
00	1	
01		1
11		1
10	1	

f_1

yz \ x	0	1
00		
01	1	1
11		1
10		

f_2

yz \ x	0	1
00	1	
01	1	1
11		
10	1	

f_3

yz \ x	0	1
00	1	
01	1	1
11		
10		

f_4

Product	Inputs			Outputs			
Term	x	y	z	f_1	f_2	f_3	f_4
1: $x'z'$	0	–	0	1	•	1	•
2: xy'	1	0	–	•	•	•	1
3: xz	1	–	1	1	1	•	•
4: $y'z$	–	0	1	•	1	1	1

$f_1 = x'z' + xz$
$f_2 = xz + y'z$
$f_3 = x'z' + y'z$
$f_4 = xy' + y'z$

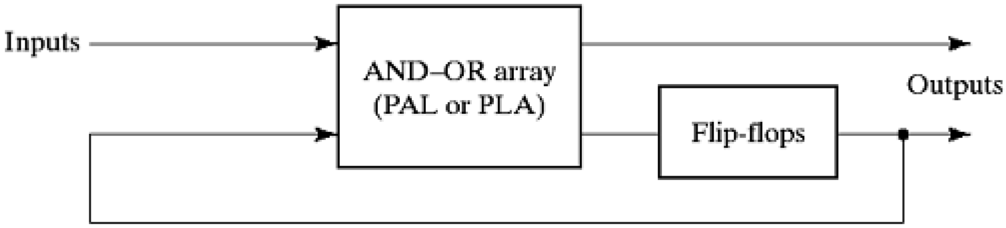
Programming the PLA

In the example, the number of links is 4(6 + 4) = 40. Only 16 of these are to be kept, meaning that, during field programming, 24 links are to be blown out. Typical PLAs have many more inputs, outputs, and AND gates than are shown in the example.

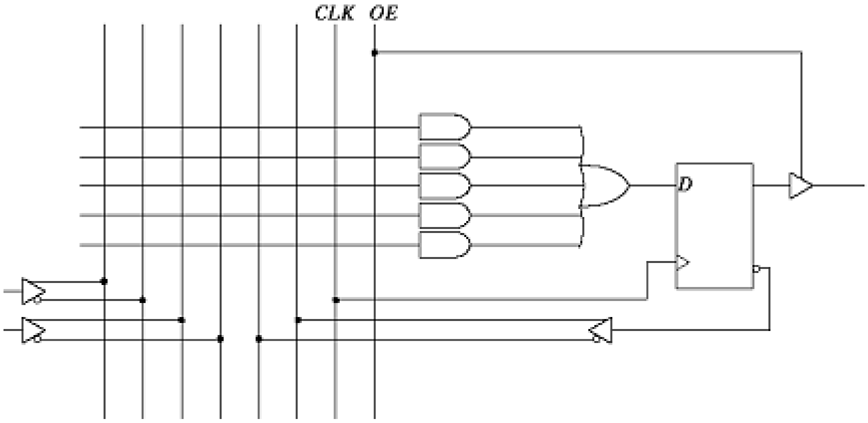
When a set of switching functions is presented for implementation with a PLA, a design goal would be reduction in p(the number of AND gates). The economy achieved is not derived from a reduction in the production cost of gates. (The production cost of an IC is practically the same for one with 40 gates as it is for one with 50 gates.) Rather, the removal of one AND gate eliminates 2n+m links; the main source of savings is the elimination of a substantial number of links due to the elimination of each AND gate. On the other hand, reduction of the number of AND gates to a minimum does not mean that each function should be minimized or that all implicants should be prime implicants. The implicants should be chosen so that as many as possible of them are common to many of the output functions.

Sequential Programmable Devices:-

The most simple sequential PLD = PLA (PAL) + Flip-Flops



The mostly used configuration for SPLD is constructed with 8 to 10 macro cells as shown below.

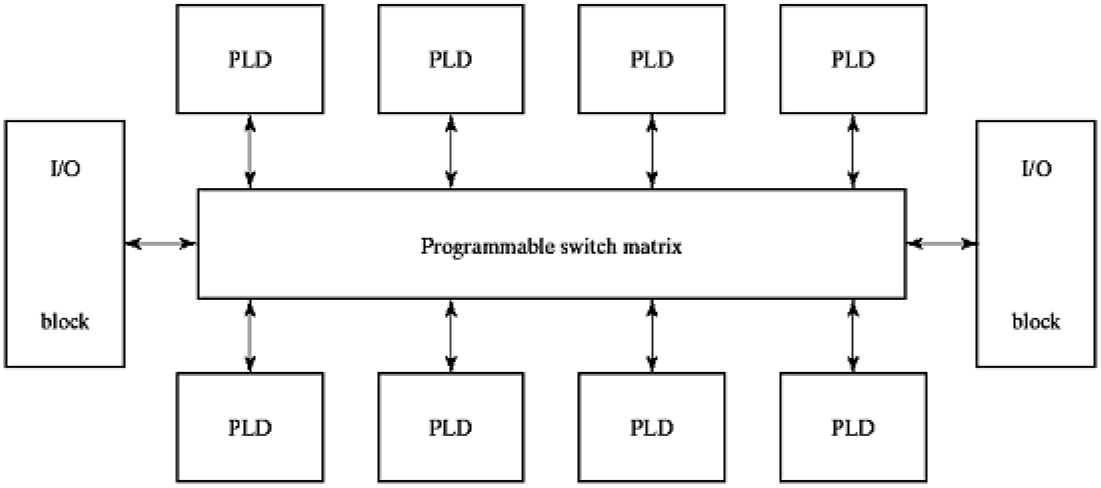


Complex PLD:-

Complex digital systems often require the connection of several devices to produce the complex specification

More economical to use a complex PLD (CPLD)

CPLD is a collection of individual PLDs on a single IC with programmable interconnection structure



Field Programmable Gate Array (FPGA):-

-Gate array: a VLSI circuit with some pre-fabricated gates repeated thousands of times

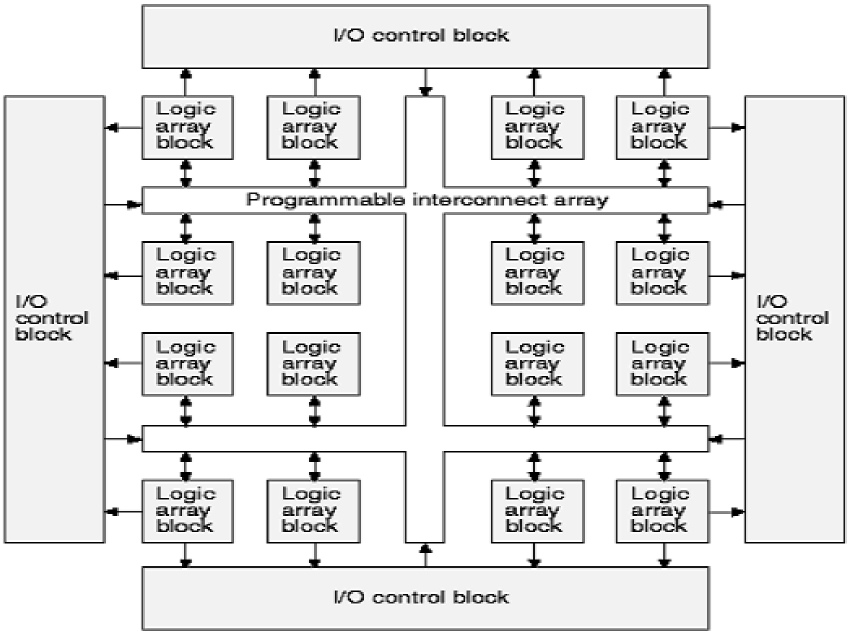
-Designers have to provide the desired interconnection patterns to the manufacturer (factory)

-A field programmable gate array (FPGA) is a VLSI circuit that can be programmed in the user's location

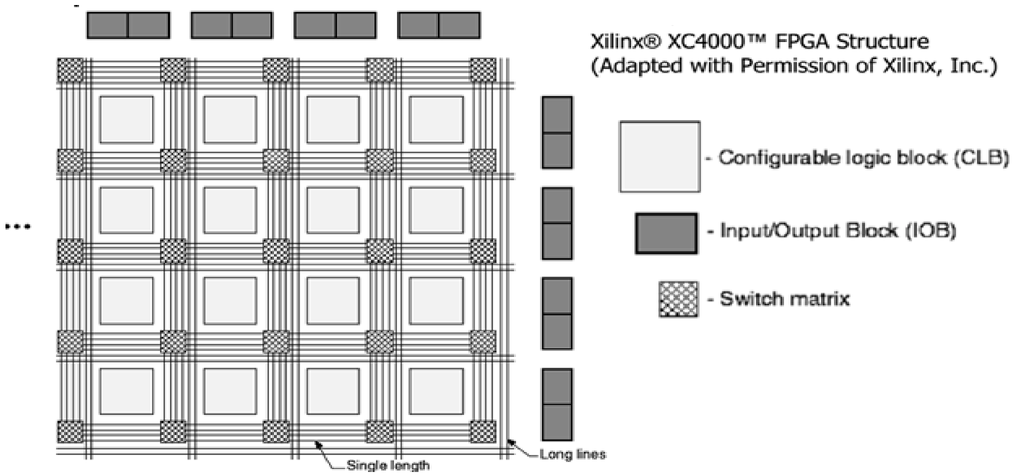
-Easier to use and modify

-Getting popular for fast and reusable prototyping

-There are various implementations for FPGA



FPGA structure (Altera)


Xilinx® XC4000™ FPGA Structure
(Adapted with Permission of Xilinx, Inc.)
- Configurable logic block (CLB)
- Input/Output Block (IOB)
- Switch matrix
Single length
Long lines
...

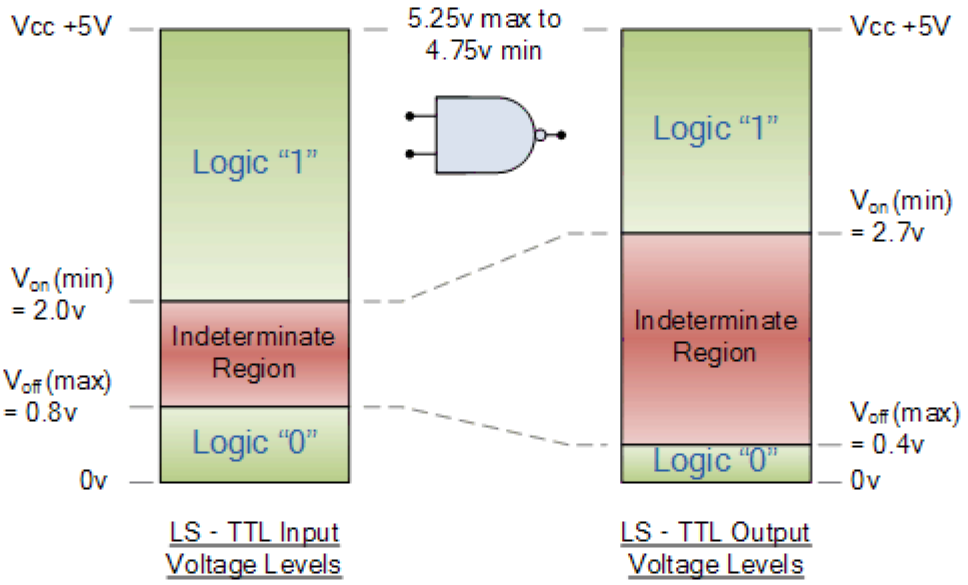
FPGA (Xilinx)

Digital Logic Circuits :-

TTL Circuit:-

TTL logic IC's use NPN and PNP type Bipolar Junction Transistors while **CMOS** logic IC's use complementary MOSFET or JFET type Field Effect Transistors for both their input and output circuitry. As well as TTL and CMOS technology, simple Digital Logic Gates can also be made by connecting together diodes, transistors and resistors to produce **RTL**, Resistor-Transistor logic gates, **DTL**, Diode-Transistor logic gates or **ECL**, Emitter-Coupled logic gates but these are less common now compared to the popular **CMOS** family.

TTL Input & Output Voltage Levels:-



When using a standard +5 volt supply any TTL voltage input between 2.0 V and 5 V is considered to be a logic "1" or "HIGH" while any voltage input below 0.8 V is recognized as a logic "0" or "LOW". The voltage region in between these two voltage levels either as an input or as an output is called the *Indeterminate Region* and operating within this region may cause the logic gate to produce a false output.

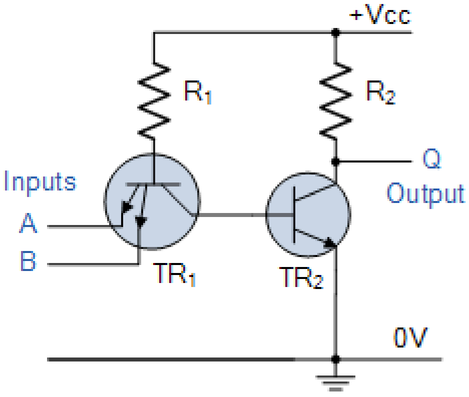
The CMOS 4000 logic family uses different levels of voltages compared to the TTL types as they are designed using field effect transistors, or FET's. In CMOS technology a logic "1" level operates between 3.0 volts and 18 volts and a logic "0" level is below 1.5 volts.

Basic TTL Logic Gates:-

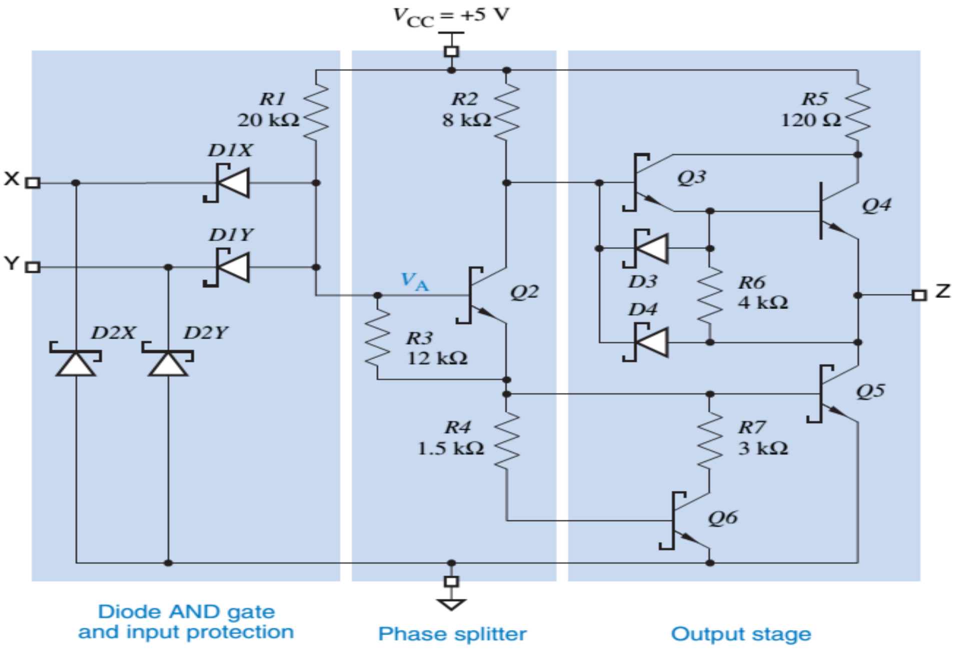
The simple Diode-Resistor AND gate above uses separate diodes for its inputs, one for each input. As a transistor is made up off two diode circuits connected together representing an NPN or a PNP device, the input diodes of the DTL circuit can be replaced by one single NPN transistor with multiple emitter inputs as shown.

As the NAND gate contains a single stage inverting NPN transistor circuit (TR_2) an output logic level "1" at Q is only present when both the emitters of TR_1 are connected to logic level "0" or ground allowing base current to pass through the PN junctions of the emitter and not the collector. The multiple emitters of TR_1 are connected as inputs thus producing a NAND gate function.

In standard TTL logic gates, the transistors operate either completely in the "cut off" region, or else completely in the saturated region, Transistor as a Switch type operation.



2 input NAND Gate



Circuit diagram of 2-input LS-TTL NAND gate.

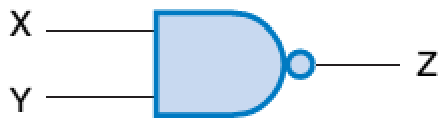
(a)

X	Y	V_A	$Q2$	$Q3$	$Q4$	$Q5$	$Q6$	V_Z	Z
L	L	1.05	off	on	on	off	off	2.7	H
L	H	≤1.05	off	on	on	off	off	2.7	H
H	L	≤1.05	off	on	on	off	off	2.7	H
H	H	≤1.2	on	off	off	on	on	≤0.35	L

(b)

X	Y	Z
0	0	1
0	1	1
1	0	1
1	1	0

(c)



(a)Function Table (b) Truth Table (c) Logic symbol

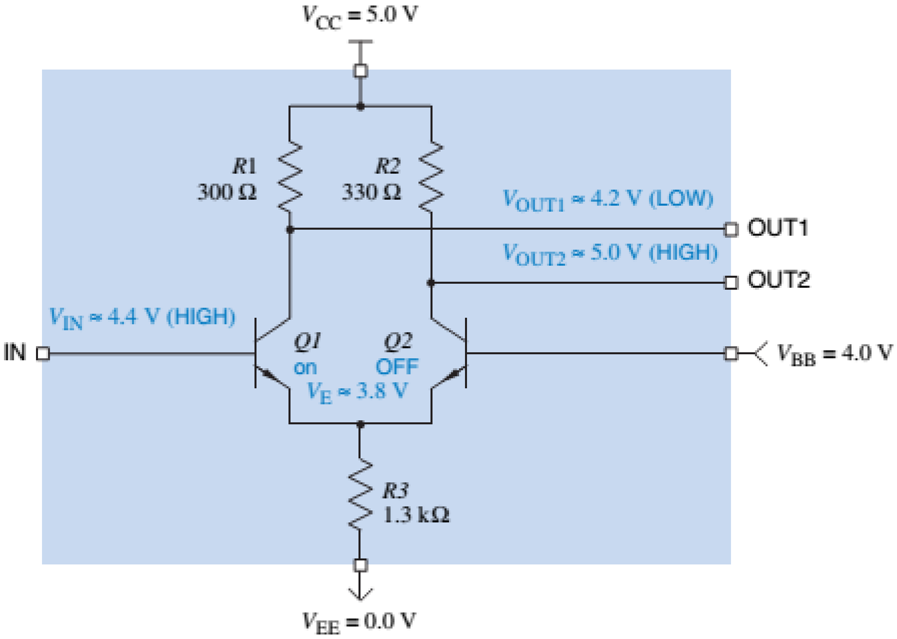
ECL: Emitter-Coupled Logic :-

The key to reducing propagation delay in a bipolar logic family is to prevent a gate's transistors from saturation. However, it is also possible to prevent saturation by using a radically different circuit structure, called current-mode logic (CML) or emitter-coupled logic (ECL).

Unlike the other logic families in this chapter, ECL does not produce a large voltage swing between the LOW and HIGH levels. Instead, it has a small voltage swing, less than a volt, and it internally switches current between two possible paths, depending on the output state.

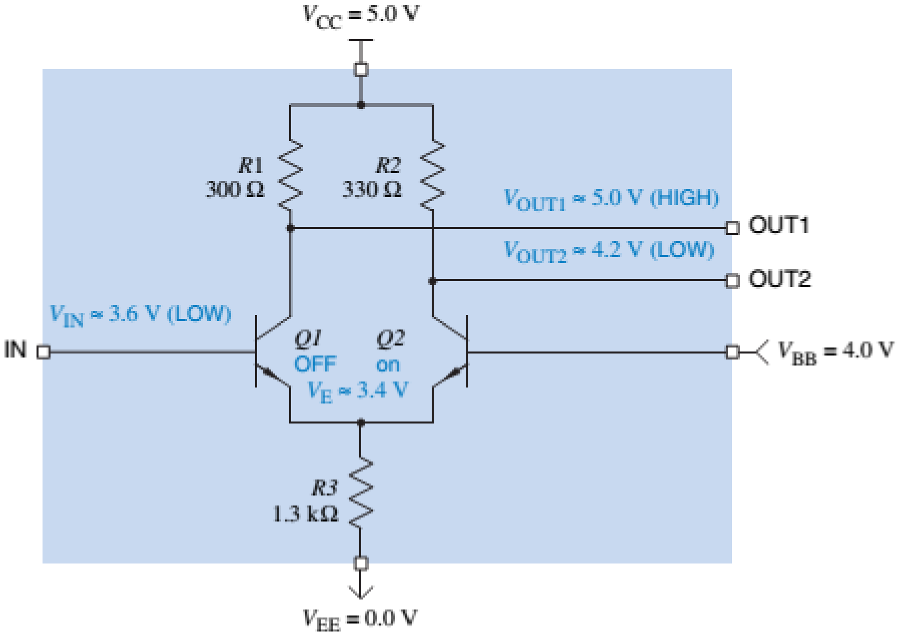
Basic ECL Circuit

The basic idea of current-mode logic is illustrated by the inverter/buffer circuit in given Figure. This circuit has both an inverting output (OUT1) and a non inverting output (OUT2). Two transistors are connected as a differential amplifier with a common emitter resistor. The supply voltages for this example are V_{CC} =5.0, V_{BB} =4.0, and V_{EE}=0 V, and the input LOW and HIGH levels are defined to be 3.6 and 4.4 V. This circuit actually produces output LOW and HIGH levels that are 0.6 V higher (4.2 and 5.0 V), but this is corrected in real ECL circuits.

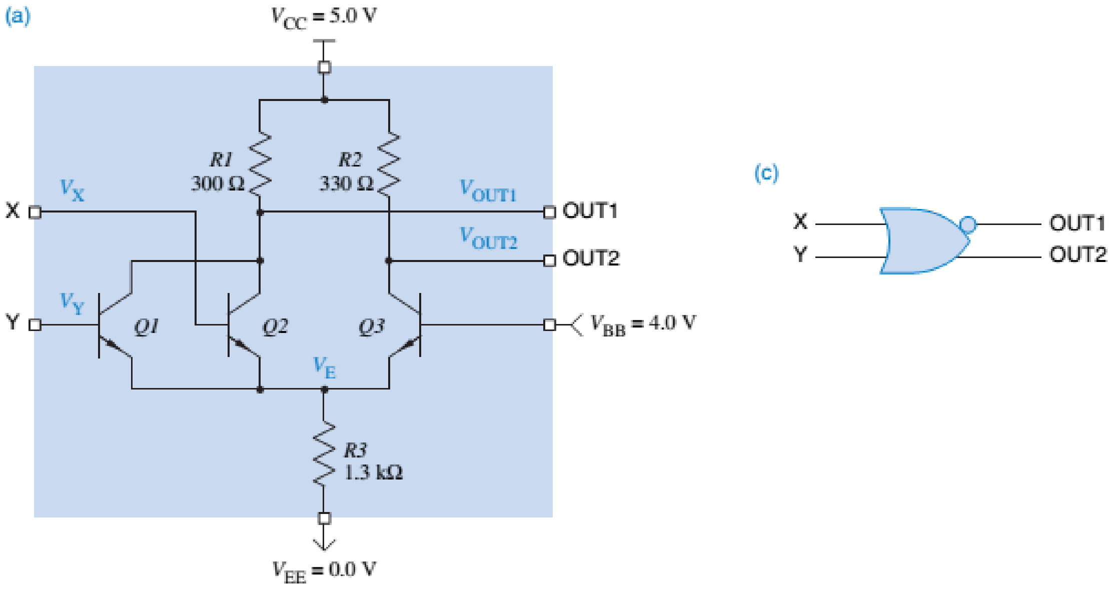


Basic ECL inverter/buffer circuit with input HIGH.

When V_{IN} is HIGH, as shown in the figure, transistor Q_1 is on, but not saturated, and transistor $Q_2$is OFF. This is true because of a careful choice of resistor values and voltage levels. Thus, V_{OUT2} is pulled to 5.0 V (HIGH) through R_2, and it can be shown that the voltage drop across $R_1$is about 0.8 V, so that $V_{OUT1}$is about 4.2 V (LOW).



Basic ECL inverter/ buffer circuit with input LOW.



(b)

X	Y	V_X	V_Y	*Q1*	*Q2*	*Q3*	V_E	V_{OUT1}	V_{OUT2}	OUT1	OUT2
L	L	3.6	3.6	OFF	OFF	on	3.4	5.0	4.2	H	L
L	H	3.6	4.4	OFF	on	OFF	3.8	4.2	5.0	L	H
H	L	4.4	3.6	on	OFF	OFF	3.8	4.2	5.0	L	H
H	H	4.4	4.4	on	on	OFF	3.8	4.2	5.0	L	H

(d)

X	Y	OUT1	OUT2
0	0	1	0
0	1	0	1
1	0	0	1
1	1	0	1

ECL 2-input OR/NORgate: (a) circuit diagram; (b) function table; (c) logic symbol; (d) truth table

DIODE TRANSISTOR LOGIC CIRCUITS :-

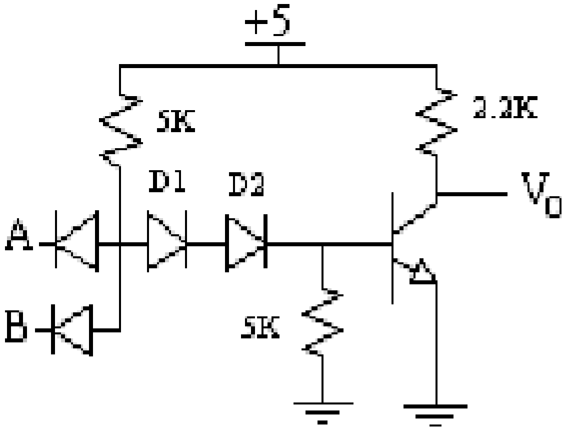
A typical DTL NAND gate is shown in the given Figure. Observe the diode AND function on the front end and the transistor NOT at the output end. The extra resistors and diodes are used to maintain appropriate currents, to maintain proper functioning, and to guarantee certain noise margins.

Analysis of the DTL gate :-

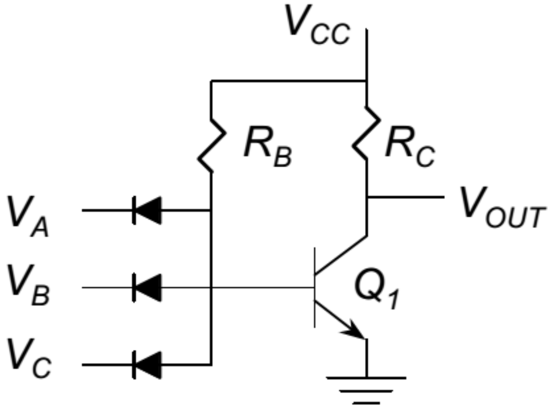
Analysis of the DTL gate is dependent on complete understanding of the currents within the gate under all logic conditions. First let us develop a generic understanding of the operation. Of particular importance will be the direction of currents at the terminals of the gate. As in most logic systems, the transistor will either be cutoff or saturated.

If all inputs are high, (+5 V), no current will come out of the input diodes at the input and current will flow down through the first 5K resistor and through the diodes D_1 and D_2 toward the base of the transistor. Some current will split off and go down through the lower 5K resistor to ground. However, most of the current will go into the base of the transistor causing it to saturate, pulling the output low, V_O=0.2 Volts. We will show this condition quantitatively shortly.

If one or more of the inputs to the gate are held low (0.2 V), then the current down through the 5K resistor will go out the input diode, away from the transistor base. Under this condition, the transistor will be cutoff and the ouput will be high with V_O=5 Volts.



DTL circuit



Improved gate with reversed diodes.

-If all inputs are high, the transistor saturates and V_{OUT} goes low.

-If any input goes low, the base current is diverted out through the input diode. The transistor cuts off and V_{OUT} goes high.

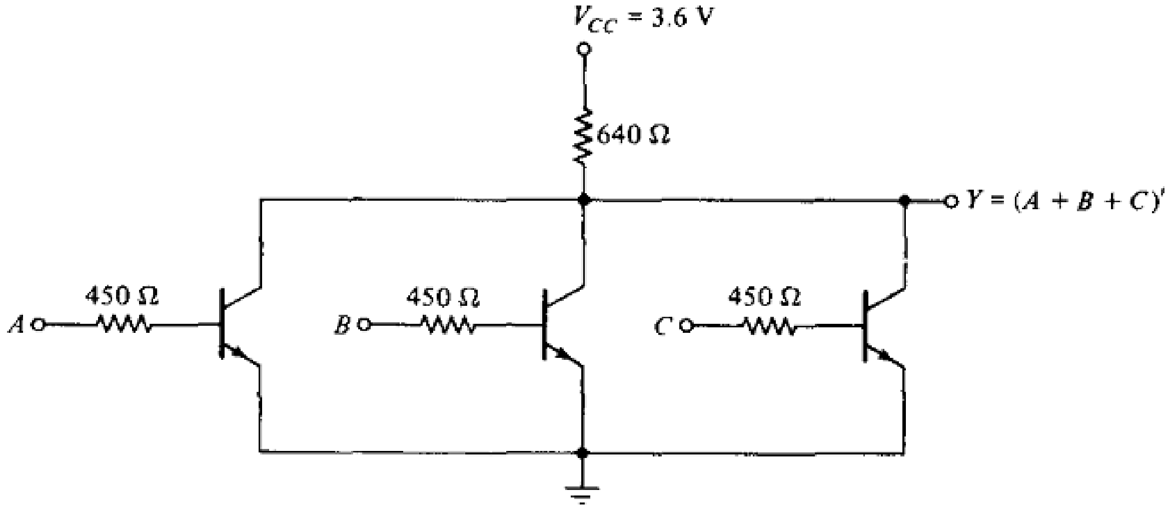
-This is a NAND gate.

RTL Logic:-

The basic circuit of the RTL digital logic family is the NOR gate shown in given Figure. Each input is associated with one resistor and one transistor. The collectors of the transistors are tied together at the output. The voltage levels for the circuit are 0.2 V for the low level and from 1 to 3.6 V for the high level.

The analysis of the RTL gate is very simple and as follows. If any input of the RTL gate is high, the corresponding transistor is driven into saturation. This causes the output to be low, regardless of the states of the other transistors. If all inputs are low at 0.2 V, all transistors are cut off because $V_{BE} < 0.6$ V. This causes the output of the circuit to be high, approaching the value of supply voltage V_{cc}. This confirms the conditions stated in given Fig for the NOR gate. Note that the noise margin for low signal input is 0.6 - 0.2 = 0.4 V.

The fan-out of the RTL gate is limited by the value of the output voltage when high. As the output is loaded with inputs of other gates, more current is consumed by the load. This current must flow through the 640 Ω resistor.

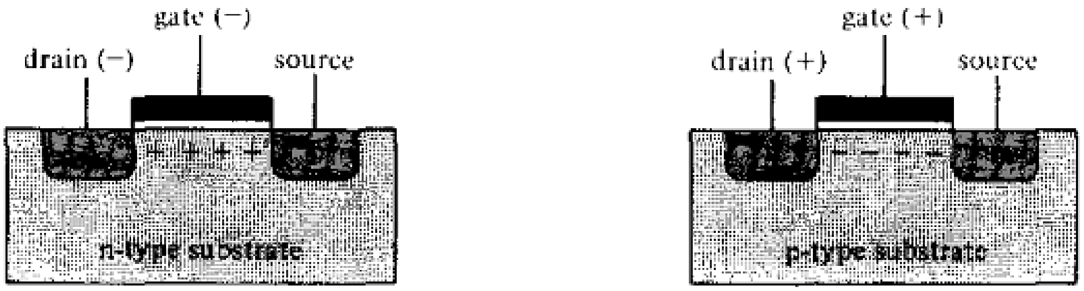


RTL NOR Gate

METAL OXIDE SEMICONDUCTOR (MOS) :-

The field-effect transistor (FET) is a unipolar transistor, since its operation depends on the flow of only one type of carrier. There are two types of field-effect transistors: the junction field-effect transistor (JFET) and the metal-oxide semi-conductor (MOS). The former is used in linear circuits and the latter in digital circuits. MOS transistors can be fabricated in less area than bipolar transistors.

The basic structure of the MOS transistor is shown in given Fig. The p-channel MOS consists of a lightly doped substrate of n-type silicon material. Two regions are heavily doped by diffusion with p-type impurities to form the source and drain. The region between the two p type sections serves as the channel. The gate is a metal plate separated from the channel by an insulated dielectric of silicon dioxide. A negative voltage (with respect to the substrate) at the gate terminal causes an induced electric field in the channel that attracts p-type carriers from the substrate. As the magnitude of the negative voltage on the gate increases, the region below the gate accumulates more positive carriers, the conductivity increases, and current can flow from source to drain provided a voltage difference is maintained between these two terminals.

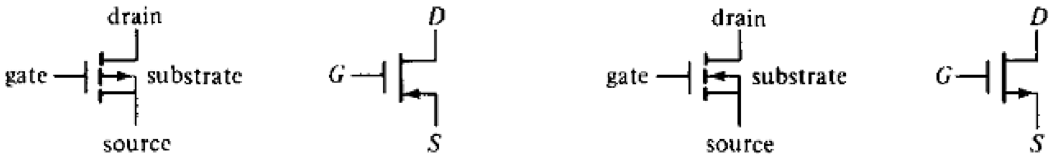


(a) p-channel

(b) n-channel

Basic structure of MOS transistor

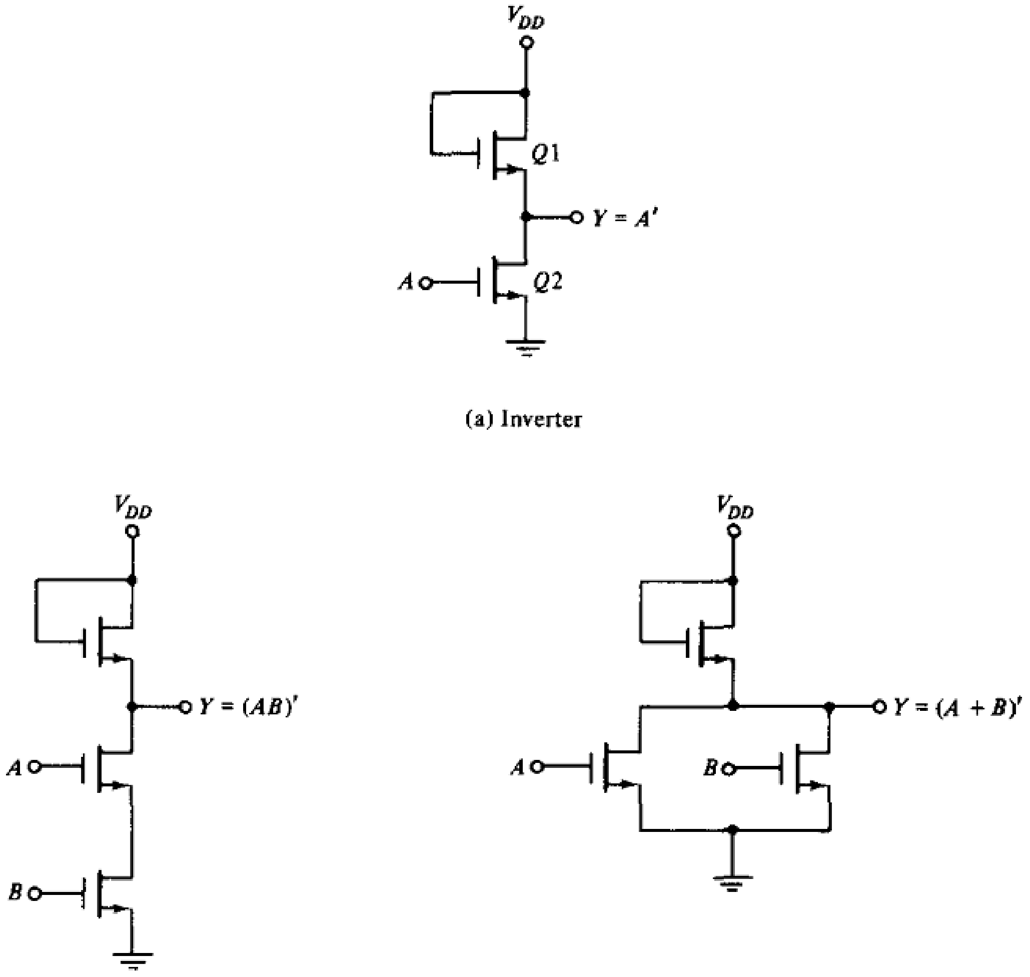
There are four basic types of MOS structures. The channel can be a p-or n-type, depending on whether the majority carriers are holes or electrons. The mode of operation can be enhancement or depletion, depending on the state of the channel region at zero gate voltage. If the channel is initially doped lightly with p-type impurity (diffused channel), a conducting channel exists at zero gate voltage and the device is said to operate in the depletion mode. In this mode, current flows unless the channel is depleted by an applied gate field. If the region beneath the gate is left initially uncharged, a channel must be induced by the gate field before current can flow. Thus, the channel current is enhanced by the gate voltage and such a device is said to operate in the enhancement mode.



(a) p-channel

(b) n-channel

Symbol of MOS Transistor



(a) Inverter

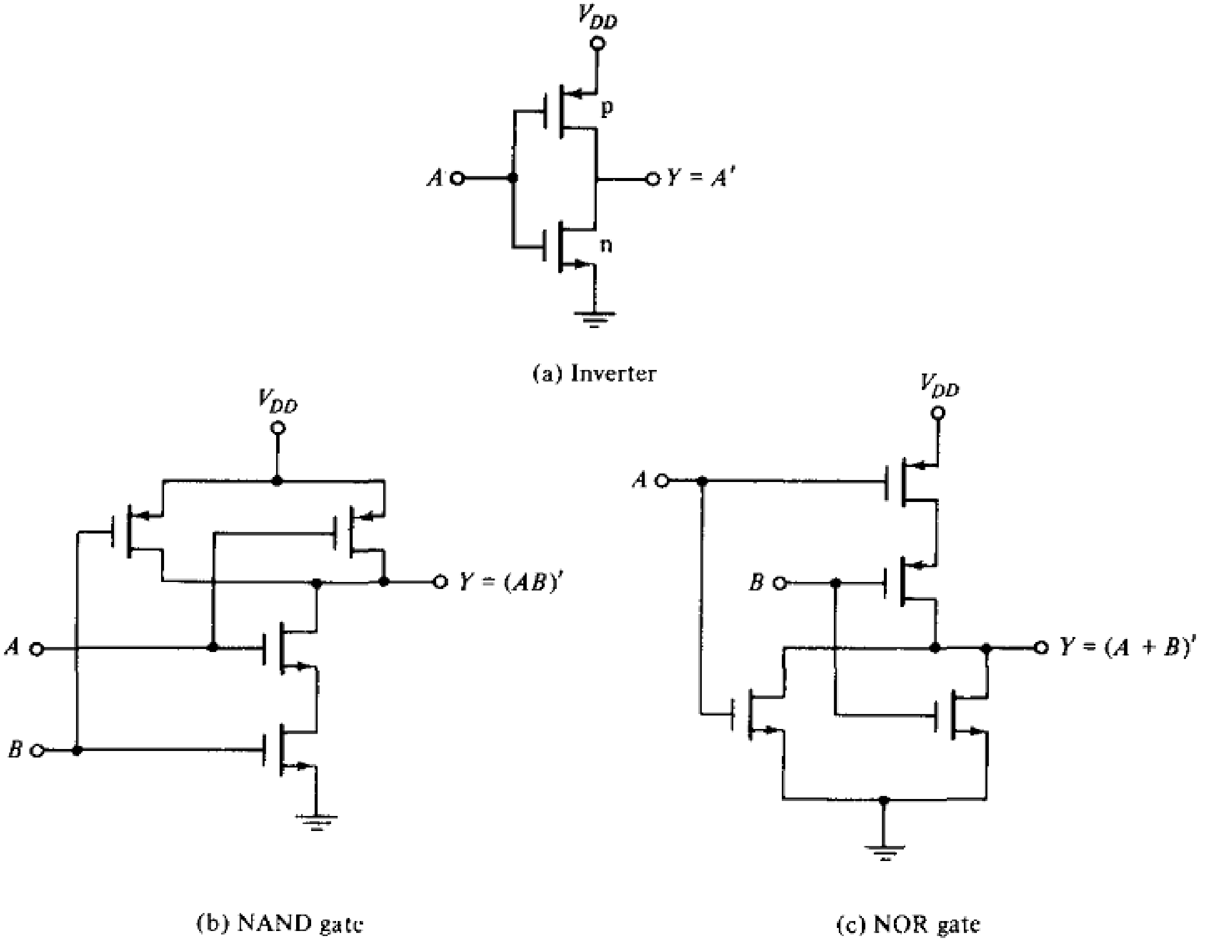
(b) NAND gate

(c) NOR gate

n-channel MOS logic circuits

COMPLEMENTARY MOS (CMOS) :-

Complementary MOS circuits take advantage of the fact that both n-channel and p-channel devices can be fabricated on the same substrate. CMOS circuits consist of both types of MOS devices interconnected to form logic functions. The basic circuit is the inverter, which consists of one p-channel transistor and one n -channel transistor.



(a) Inverter

(b) NAND gatc

(c) NOR gate

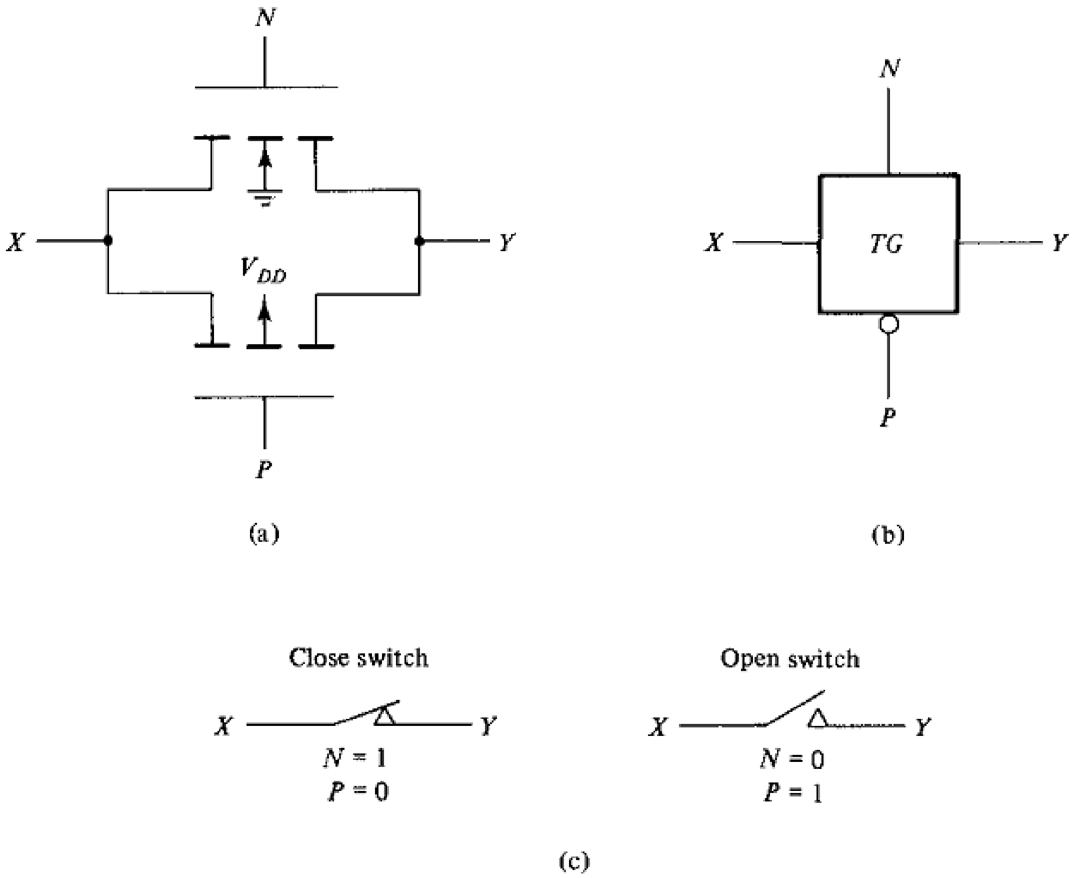
CMOS logic circuits

Now consider the operation of the inverter. When the input is low, both gates arc at zero potential. The input is at -V_{DD} relative to the source of the p-channel device and at 0 V relative to the source of the n-channel device. The result is that the p-channel device is turned on and the n-channel device is turned off. Under these conditions, there is a low-impedance path from V_{DD} to the output and a very high-impedance path from output to ground. Therefore, the output voltage approaches the high level V_{DD} under normal loading conditions. When the input is high, both gates are at V_{DD} and the situation is reversed: The p-channel device is off and the n-channel device is on. The result is that the output approaches the low level of 0 V.

CMOS TRANSMISSION GATE CIRCUITS:-

The transmission gate is essentially an electronic switch that is controlled by an input logic level. It is used for simplifying the construction of various digital components when fabricated with CMOS technology.

It consists of one n-channel and one p-channel MOS transistor connected in parallel. The n-channel substrate is connected to ground and the p-channel substrate is connected to V_{DD}. When the N gate is at V_{DD} and the P gate is at ground, both transistors conduct and there is a closed path between input X and output Y. When the N gate is at ground and the P gate at V_{DD} both transistors are off and there is an open circuit between X and Y.



(a)

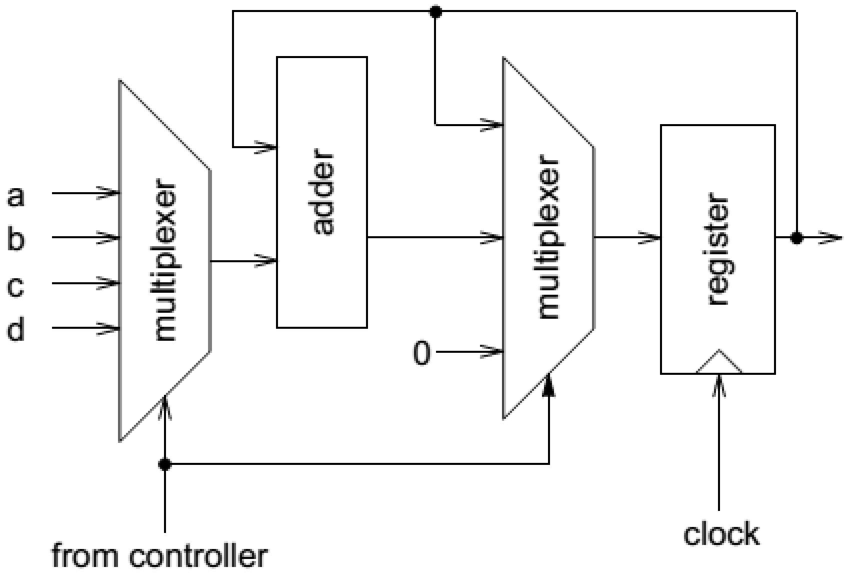
(b)

(c)

Transmission Gate (TG)

RTL Design Example:-

To show how an RTL design is described in VHDL and to clarify the concepts involved, we will design a four-input adder. This design will also demonstrate how to create packages of components that can be re-used. The data path shown below can load the register at the start of each clock cycle with one of: zero, the current value of the register, or the sum of the register and one of the four inputs. It includes one 8-bit register, an 8-bit adder and a multiplexer that selects one of the four possible inputs as the value to be added to the current value of the register.



The first design unit is a package that defines a new type, num, for eight-bit unsigned numbers and an enumerated type, states, with six possible values. nums are defined as a subtype of the unsigned type.

```
-- RTL design of 4-input summer

-- subtype used in design

library ieee ;

use ieee.std_logic_1164.all;

use ieee.std_logic_arith.all;
```

```
package averager_types is
sub type num is unsigned (7 downto 0);
type states is (clr, add_a, add_b, add_c,
add_d, hold);
end averager_types;
```

The first entity defines the data path. In this case the four numbers to be added are available as inputs to the entity and there is one output for the current sum. The inputs to the data path from the controller are a 2-bit selector for the multiplexer and two control signals to load or clear (set to 0) the register.

```
-- data path
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.al;
use work.averager_types.all;
entity datapath is
port (
a, b, c, d: in num;
sum : out num ;
sel : in std_logic_vector (1 downto 0) ;
load, clear, clk : in std_logic
) ;
end datapath ;
architecture rtl of datapath is
signal mux_out, sum_reg, next_sum_reg : num ;
```

```
constant sum_zero : num :=
conv_unsigned(0,next_sum_reg’length) ;
begin
-- mux to select input to add
with sel select mux_out <=
a when "00",
b when "01",
c when "10",
d when others ;
-- mux to select register input
next_sum_reg <=
sum_reg + mux_out when load = ’1’ else
sum_zero when clear = ’1’ else
sum_reg ;
-- register sum
process(clk)
begin
if clk’event and clk = ’1’ then
sum_reg <= next_sum_reg ;
end if ;
end process ;
-- entity output is register output
sum <= sum_reg ;
end rtl;
```

The RTL design’s controller is a state machine whose outputs control the multiplexers in the datapath. The controller’s inputs are signals that control the controller’s state transitions. In this case the only input is an update signal that tells our device to recompute the sum (presumably because one or more of the inputs has changed).

This particular state machine sits at the “hold” state until the update signal is true. It then sequences through the other five states and then stops at the hold state again. The other five states are used to clear the register and to add the four inputs to the current value of the register.

```
-- controller
library ieee ;
use ieee.std_logic_1164.all ;
use work.averager_types.all ;
entity controller is
port (
update : in std_logic ;
sel : out std_logic_vector (1 downto 0) ;
load, clear : out std_logic ;
clk : in std_logic
) ;
end controller ;
architecture rtl of controller is
signal s, holdns, ns : states ;
signal tmp : std_logic_vector (3 downto 0) ;
begin
-- select next state
with s select ns <=
add_a when clr,
add_b when add_a,
add_c when add_b,
add_d when add_c,
hold when add_d,
holdns when others ; -- hold
-- next state if in hold state
holdns <=
clr when update = ’1’ else
hold ;
-- state register
process(clk)
begin
if clk’event and clk = ’1’ then
s <= ns ;
end if ;
end process ;
-- controller outputs
with s select sel <=
"00" when add_a,
"01" when add_b,
"10" when add_c,
"11" when others ;
load <= ’0’ when s = clr or s = hold else ’1’ ;
```

```
clear <= ’1’ when s = clr else ’0’ ;

end rtl ;
```

State machine :-

In general, a state machine is any device that stores the status of something at a given time and can operate on input to change the status and/or cause an action or output to take place for any given change. A computer is basically a state machine and each machine instruction is input that changes one or more states and may cause other actions to take place. Each computer's data register stores a state. The read-only memory from which a boot program is loaded stores a state (the boot program itself is an initial state). The operating system is itself a state and each application that runs begins with some initial state that may change as it begins to handle input. Thus, at any moment in time, a computer system can be seen as a very complex set of states and each program in it as a state machine. In practice, however, state machines are used to develop and describe specific device or program interactions.

To summarize it, a state machine can be described as:

- An initial state or record of something stored someplace
- A set of possible input events
- A set of new states that may result from the input
- A set of possible actions or output events that result from a new state

A finite state machine is one that has a limited or finite number of possible states. (An infinite state machine can be conceived but is not practical.) A finite state machine can be used both as a development tool for approaching and solving problems and as a formal way of describing the solution for later developers and system maintainers. There are a number of ways to show state machines, from simple tables through graphically animated illustrations.

The finite state machine is also a useful approach to many problems in software architecture; only in this case you don’t build one you simulate it.

Essentially a finite state machine consists of a number of states – finite naturally! When a symbol, a character from some alphabet say, is input to the machine it changes state in such a way that the next state depends only on the current state and the input symbol.

Notice that this is more sophisticated than you might think because inputting the same symbol doesn’t always produce the same behaviour or result because of the change of state.

The new state depends on the old state and the input.

What this means that the entire history of the machine is summarized in its current state. All that matters is the state that it is in and not how it reached this state. Before you write off the finite state machine as so feeble as to be not worth considering as a model of computation it is worth pointing out that as you can have as many states as you care to invent the machine can record arbitrarily long histories. All you need is a state for each of the possible past histories and then the state that you find the machine in is an indication of not only its current state but how it arrived in that state.

Because a finite state machine can represent any history and a reaction, by regarding the change of state as a response to the history, it has been argued that it is a sufficient model of human behaviour i.e. humans are finite state machines.

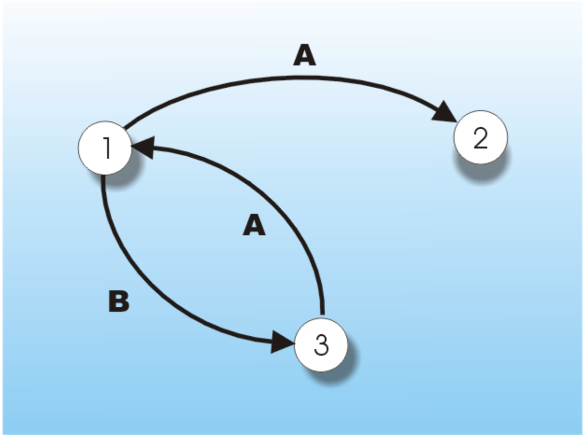
If you know some probability theory you will recognize a connection between finite state machines and Markov chains. A Markov chain sums up the past history in terms of the current state and the probability of transition to the next state only depends on the current state. The Markov chain is a sort of probabilistic version of the finite state machine.

Representing Finite State Machines

You can represent a finite state machine in a form that makes it easier to understand and think about.

All you have to do is draw a circle for every state and arrows that show which state follows for each input symbol.

For example, the finite state machine in the diagram below has three states. If the machine is in state 1 then an A moves it to state 2 and a B moves it to state 3.



A three-state finite state machine

This really does make the finite state machine look very simple and you can imagine how as symbols are applied to it how it jumps around between states.

What is the point of such a simple machine?

There are two good reasons for being interested in finite state machines. The first is practical. As mentioned earlier, there are some practical applications which are best modelled as a finite state machine.

For example, many communications protocols, such as USB can be defined by a finite state machine's diagram showing what happens as different pieces of information are input. You can even write or obtain a compiler that will take a finite state machine's specification and produce code that behaves correctly.

Many programming problems are most easily solved by actually implementing a finite state machine. You set up an array or other data structure which stores the possible states and you implement a pointer to the location that is the current state. Each state contains a lookup table that shows what the next state is given an input symbol. When a symbol is read in your program simply has to look it up in the lookup table and move the pointer to the new state.