UNIT – V

MEMORY AND PROGRAMMABLE LOGIC & HARDWARE DESCRIPTION LANGUAGE

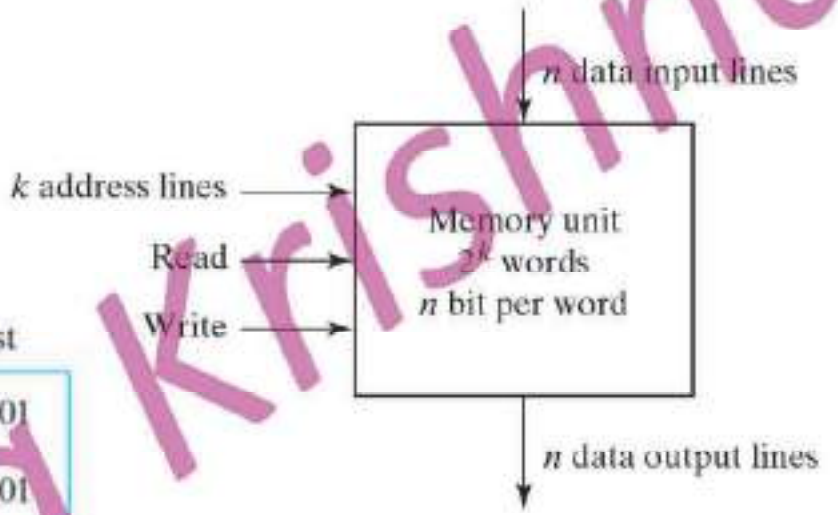
Introduction

- A memory unit is a device to which binary information is transferred for storage and from which information is available when needed for processing.
- There are two types of memories:
  - Random Access Memory
  - Read Only Memory

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.
- The communication between memory and its environment is achieved through data input and output lines, address selection lines, and control lines that specify the direction of transfer.

Block Diagram of a Memory Unit



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

Write and Read Operations

- The two operations that a RAM can perform are the Write and Read operations.
- On accepting one of the signals, the internal circuits inside the memory provide the desired operation.
- Write Operation:
  - Apply the binary address of the desired word to the address lines.
  - Apply the data bits that must be stored in memory to the data input lines.
  - Active the write input.
- Read Operation:
  - Apply the binary address of the desired word to the address lines.
  - Active the read input.

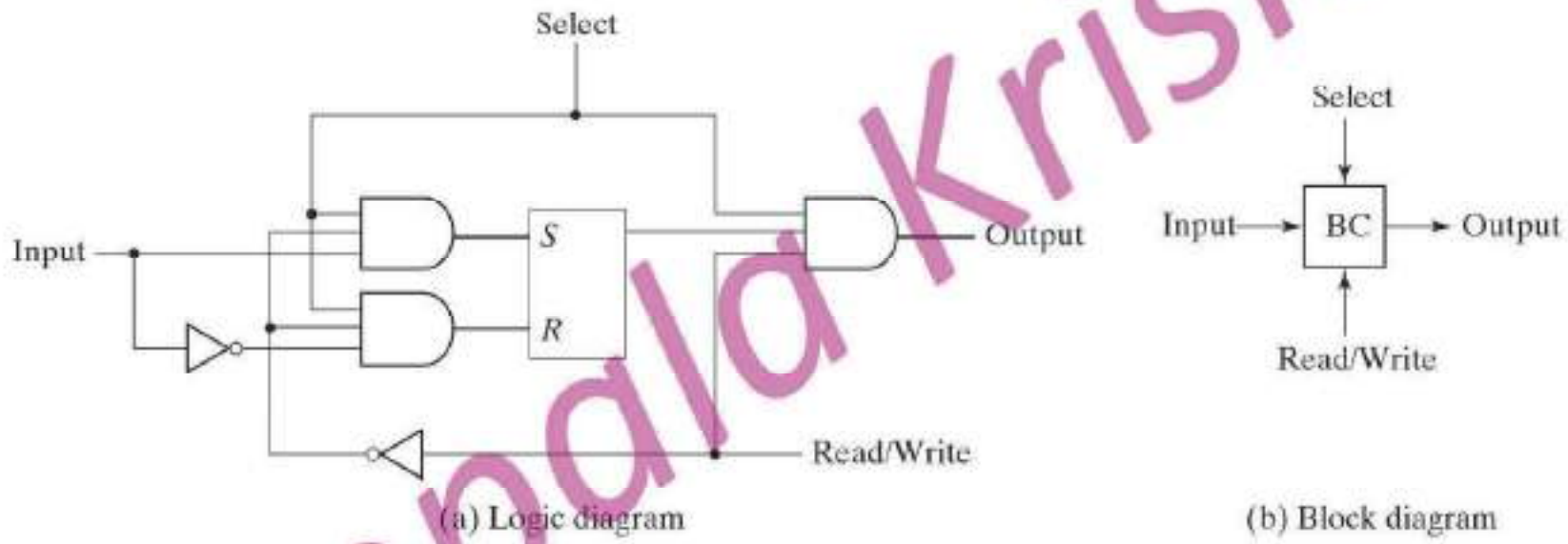
Types of Memories

- There are two types of RAM's:
  - Static RAM
  - Dynamic RAM
- The Static RAM consists essentially of internal latches that store the binary information. The stored information is valid as long as the power is applied to the unit.
- The Dynamic RAM stores the binary information in the form of electric charges on capacitors. The capacitors are provided inside the chip by MOS transistors.

Memory Decoding

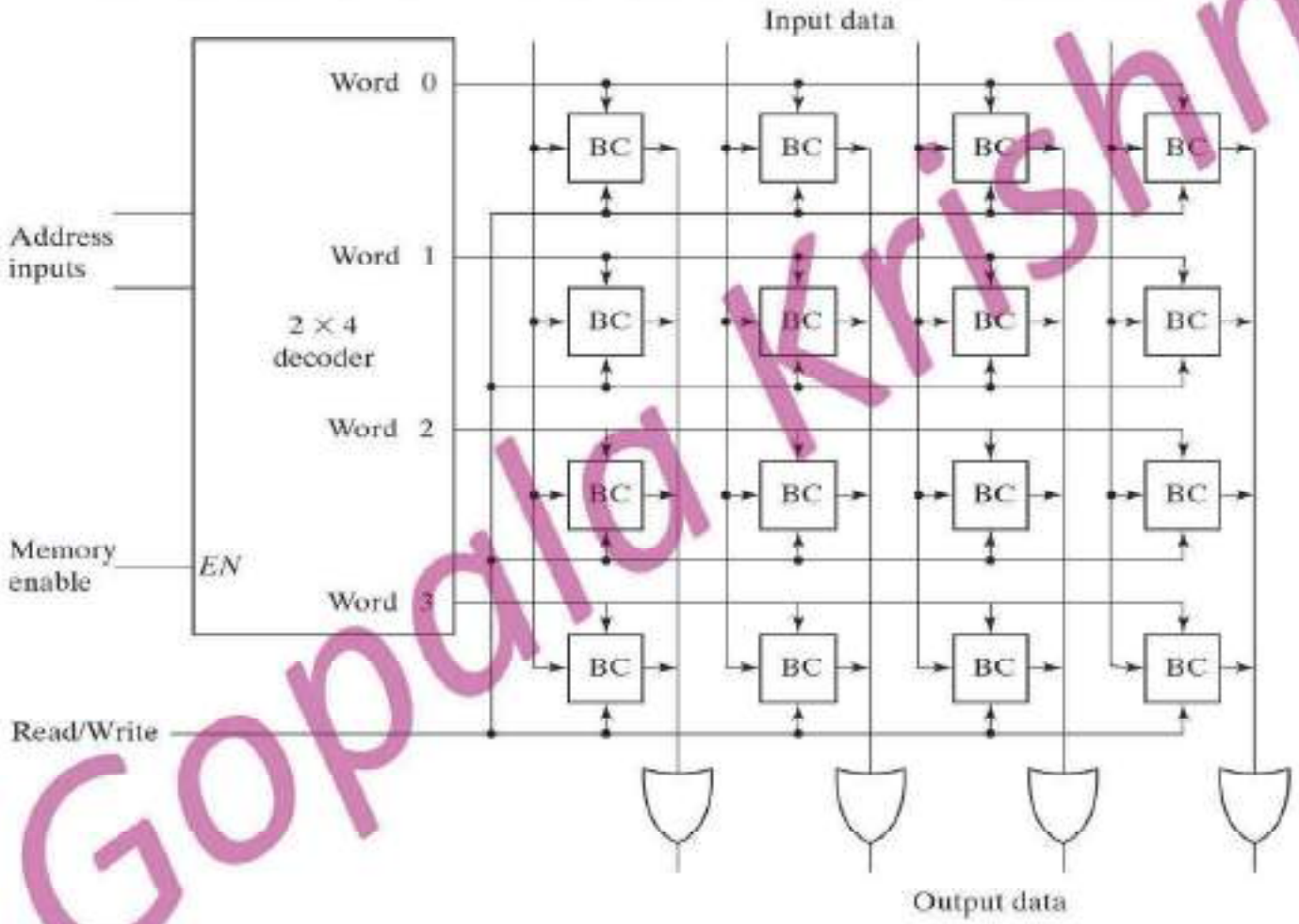
- In addition to the storage components in a memory unit, there is a need for decoding circuits to select the memory word specified by the input address.
- The internal construction of a random access memory of m words and n bits per word consists of mXn binary storage cells and associated decoding circuits for selecting individual words.
- The binary storage cell is the basic building block of the memory unit.

Binary Storage Cell



(a) Logic diagram (b) Block diagram

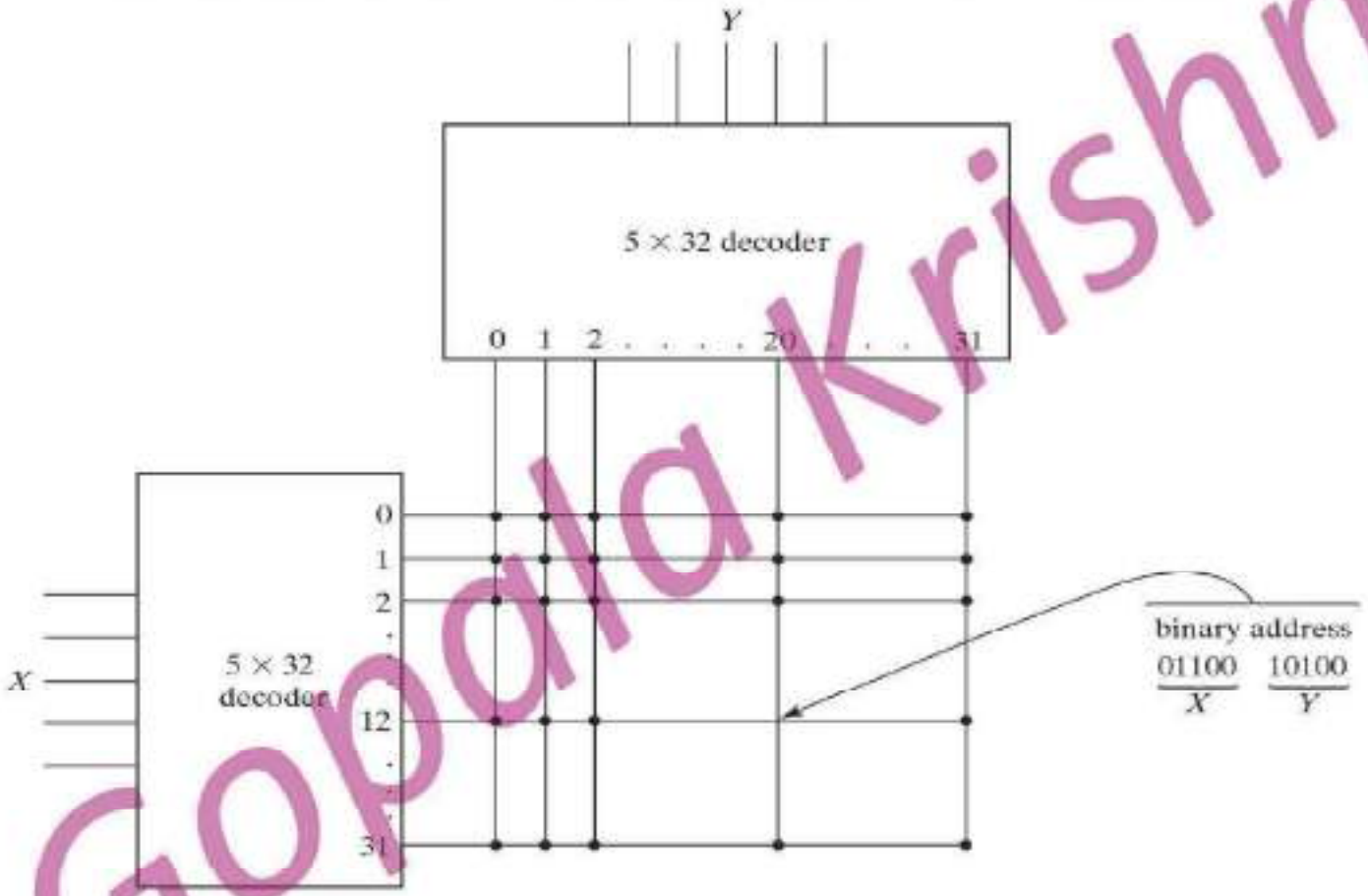
4 X 4 RAM



Coincident Decoding

- A decoder with k inputs and 2^k outputs require 2^k AND gates with k inputs per gate.
- The total number of gates and the number of inputs per gate can be reduced by employing two decoders in a two dimensional selection scheme.
- The basic idea in two dimensional decoding is to arrange the memory cell in an array that is close as possible to square.
- In this configuration, two k/2 input decoders are used instead of one k-input decoder.

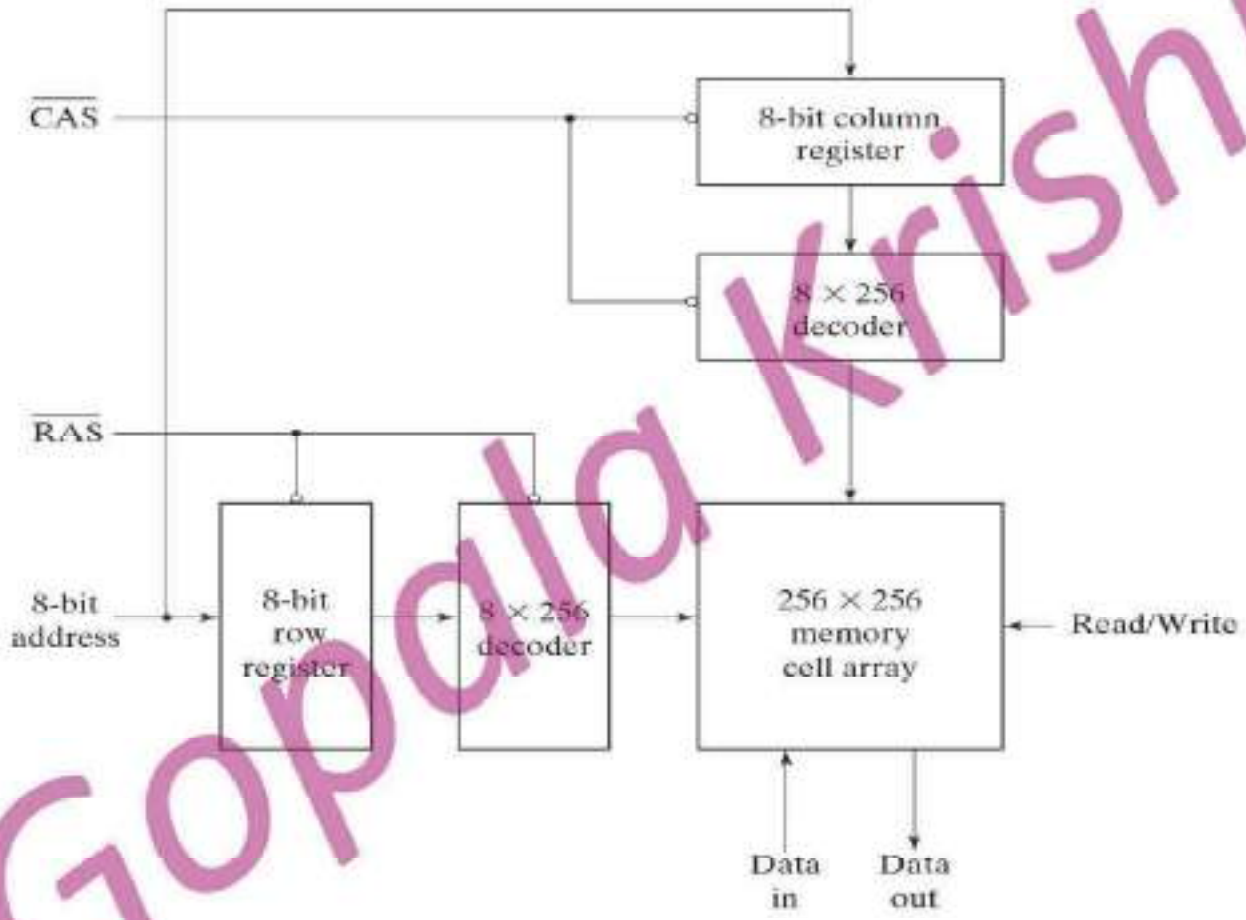
Two Dimensional Decoding Structure



Address Multiplexing

- Normally, the SRAM memory cells contains six transistors.
- Because of the simple structure of the DRAM cells they have four times the density of SRAM.
- Because of their large capacity, the address decoding of DRAMs is arranged in two dimensional array and large memories often have multiple arrays.

Address Multiplexing for DRAM

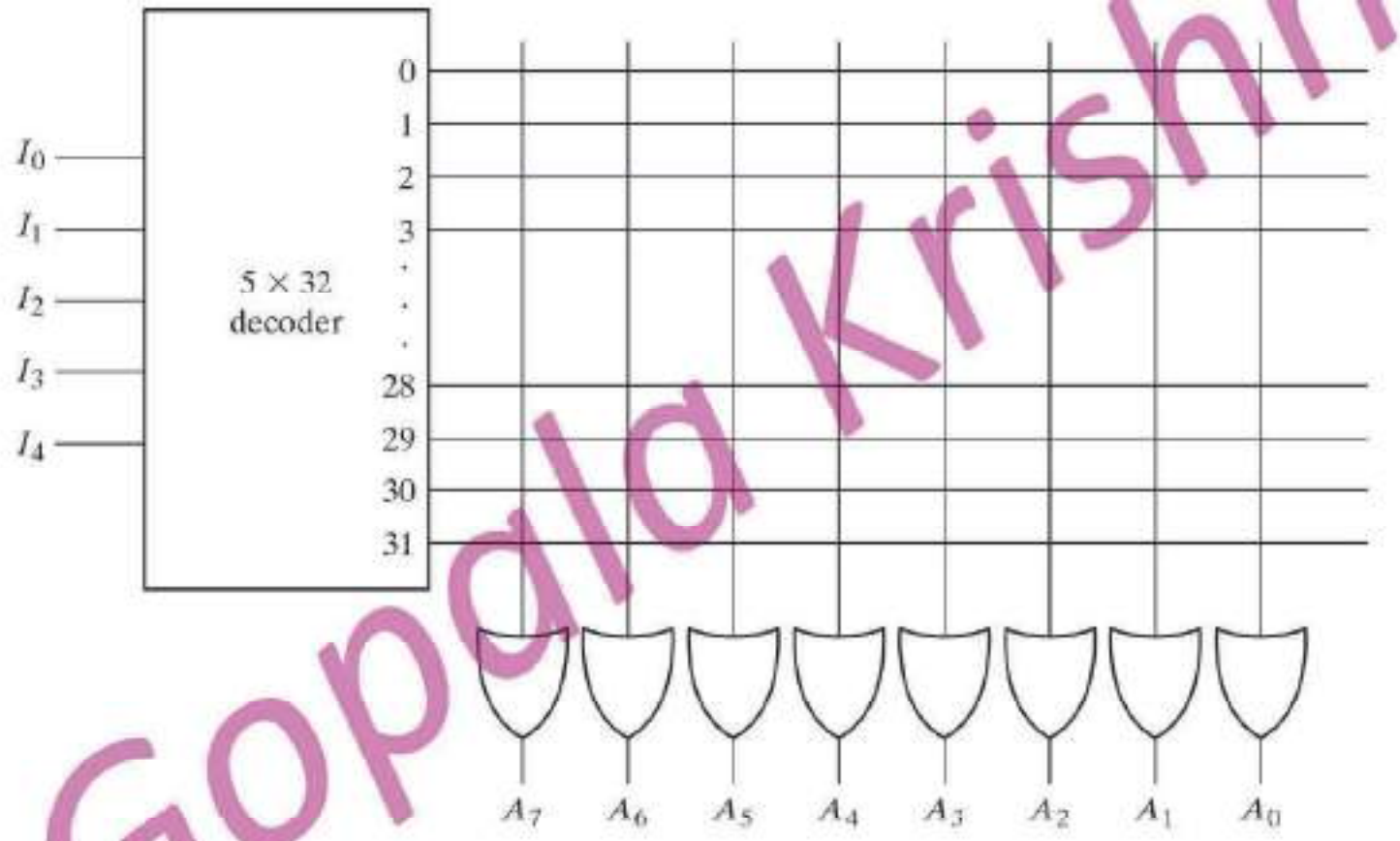


Read Only Memory (ROM)

- A ROM is essentially a memory 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.
- Once the pattern is established, it stays within the unit even when power is turned off and on again.

Block Diagram of ROM

Internal Logic of 32X8 ROM



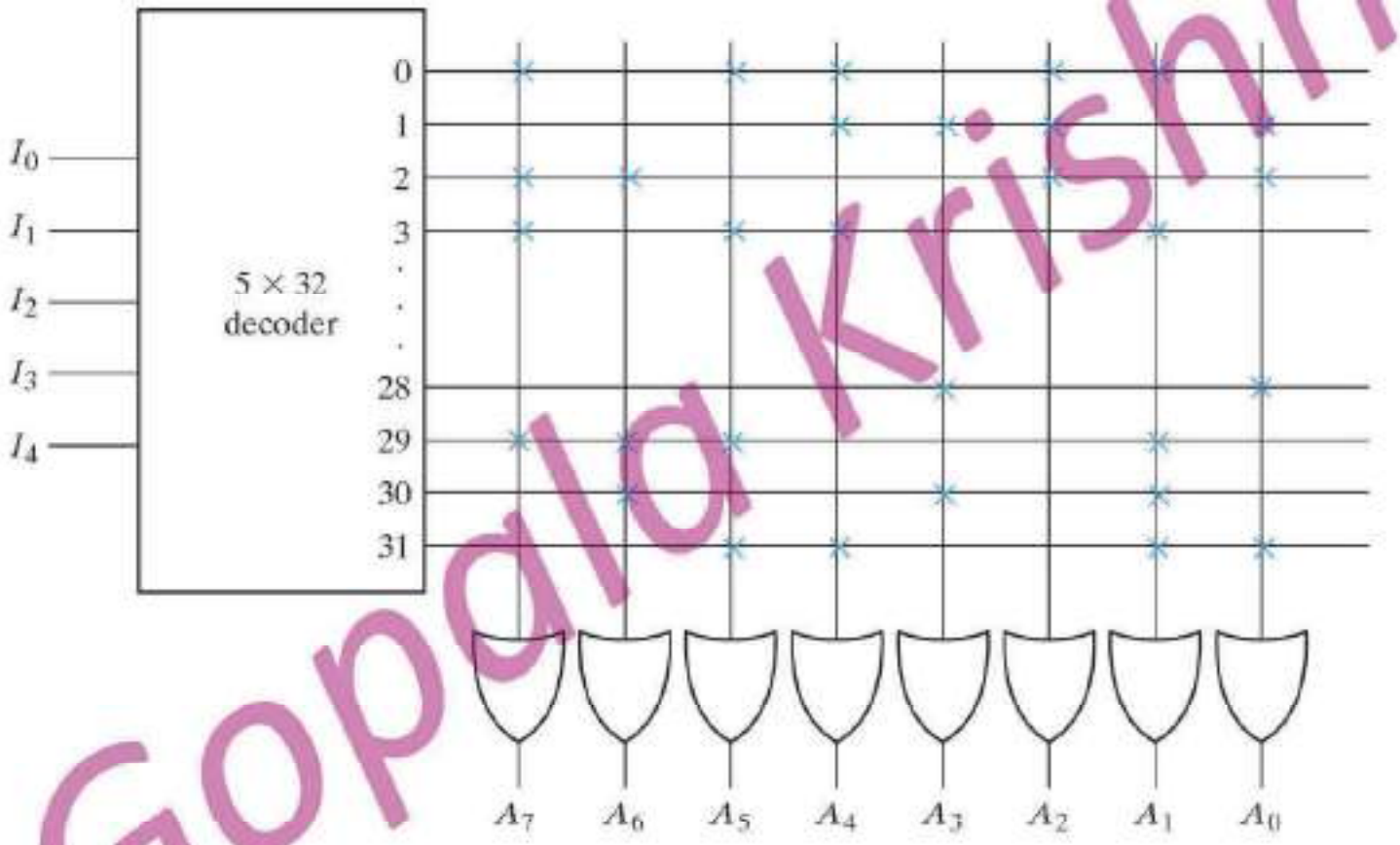
Programming ROM

- The intersections of ROM are programmable.
- A programmable connection between two lines is logically equivalent to a switch that can be altered to be either close or open.
- The programmable intersection between two points is called cross point.
- Various physical devices are used to implement crosspoint switches. One of the simplest technology employs a fuse that normally connects two points, but is opened or blown by applying a high voltage pulse into the fuse.

ROM Truth Table

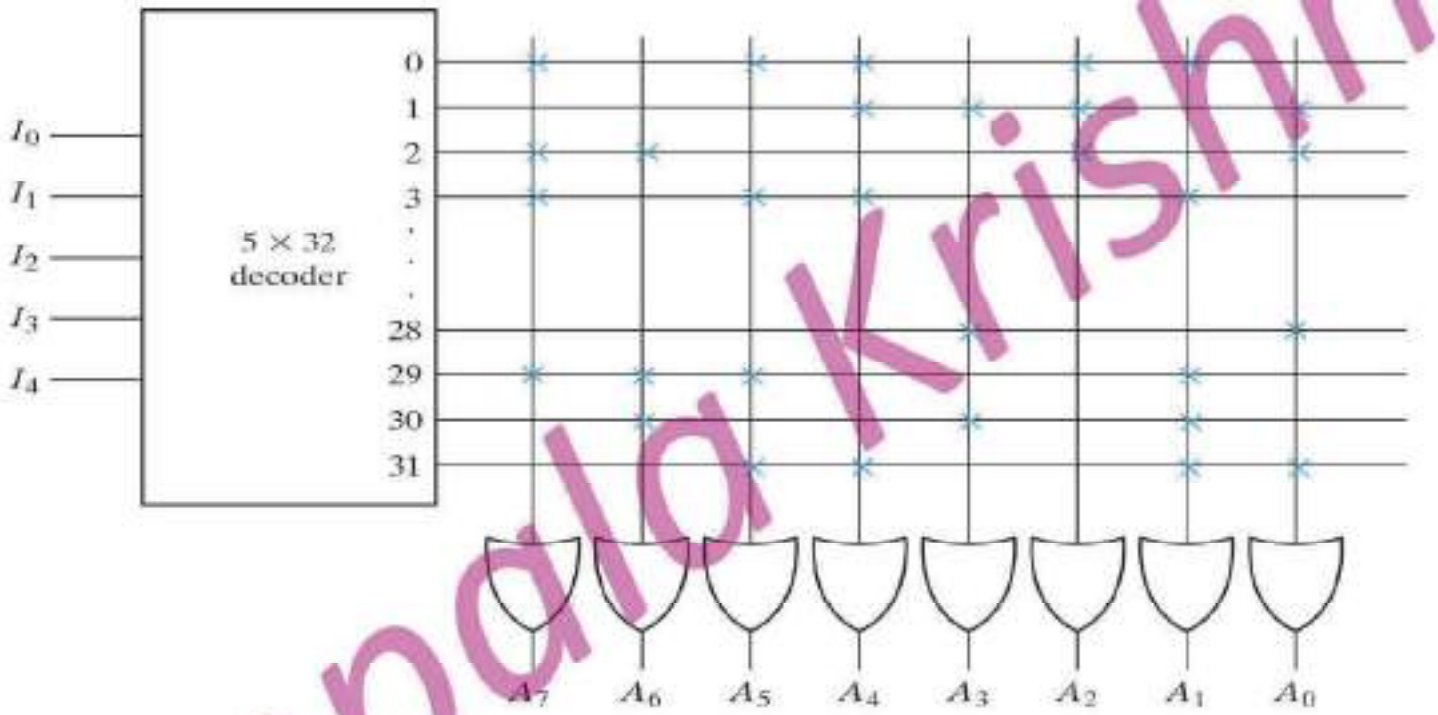
Inputs					Outputs							
I4	I3	I2	I1	I0	A7	A6	A5	A4	A3	A2	A1	A0
0	0	0	0	0	1	0	1	1	0	1	1	0
0	0	0	0	1	0	0	0	1	1	1	0	1
0	0	0	1	0	1	1	0	0	0	1	0	1
0	0	0	1	1	1	0	1	1	0	0	1	0
..	..	..	..	..	..	..	..	..	..	..	..	..
1	1	1	0	0	0	0	0	0	1	0	0	1
1	1	1	0	1	1	1	1	0	0	0	1	0
1	1	1	1	0	0	1	0	0	1	0	1	0
1	1	1	1	1	0	0	1	1	0	0	1	1

Programming ROM (32*8)



Combinational Circuit Implementation

- The internal operation of a ROM can be interpreted in two ways.
- The first interpretation is that of a memory unit that contains a fixed pattern of stored words.
- The second interpretation is of a unit that implements a combinational circuit. From this point of view, each output terminal is considered separately as the output of a Boolean function expressed as a sum of minterms.
- For example,



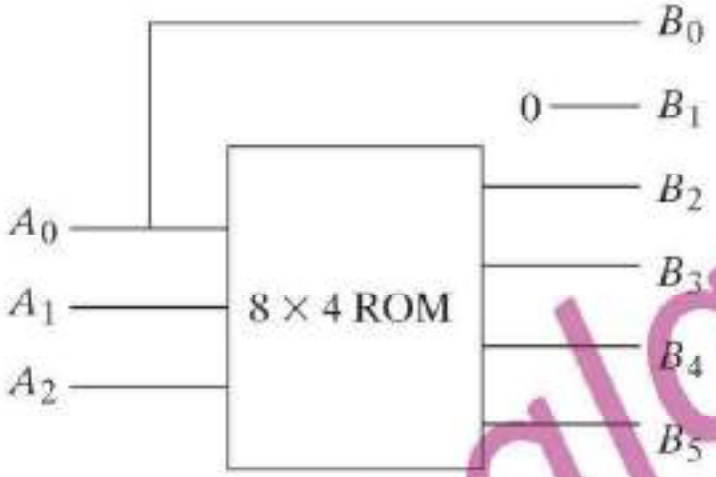
- A7(i4,i3,i2,i1,i0)=(0,2,3,…,29)

Example

- Design a combinational circuit using a ROM that accepts a 3 bit number and generates an output binary number equal to the square of the input number.

Inputs			Outputs						decimal
a2	a1	a0	b5	b4	b3	b2	b1	b0	
0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	1	1
0	1	0	0	0	0	1	0	0	4
0	1	1	0	0	1	0	0	1	9
1	0	0	0	1	0	0	0	0	16
1	0	1	0	1	1	0	0	1	25
1	1	0	1	0	0	1	0	0	36
1	1	1	1	1	0	0	0	1	49

ROM Implementation



(a) Block diagram

A_2	A_1	A_0	B_5	B_4	B_3	B_2
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	1
0	1	1	0	0	1	0
1	0	0	0	1	0	0
1	0	1	0	1	1	0
1	1	0	1	0	0	1
1	1	1	1	1	0	0

(b) ROM truth table

Types of ROM's

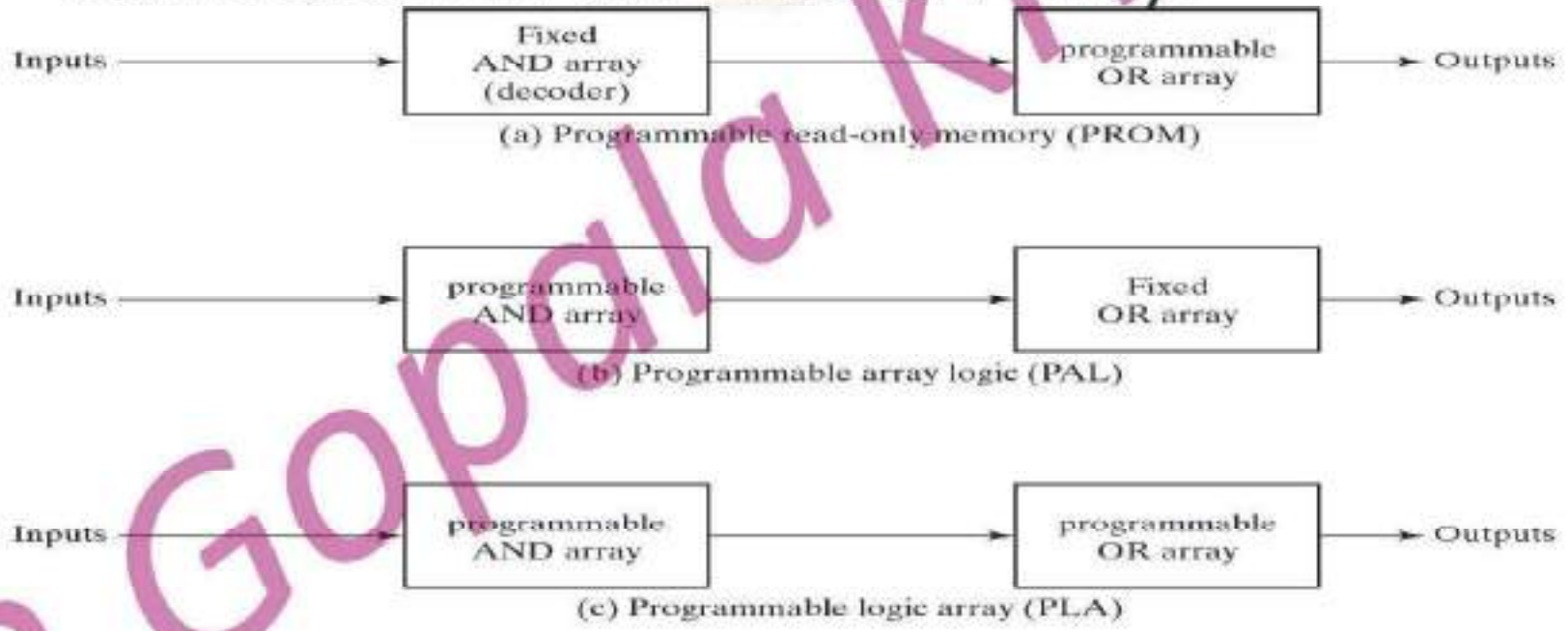
- PROM: PROM contains all the fuses intact giving all 1's in the bits of the words. The fuses in the PROM are blown out by application of high voltage pulse to the device through a special pin. The PROM is irreversible.
- EPROM: it stands for erasable PROM. When the EPROM is placed under ultraviolet radiation for some time, the short wave radiation discharges the internal floating gates.
- EEPROM: stands for electrically erasable PROM. Here the previously programmed connections can be erased with an electric signal instead of ultra violet.

Combinational PLDs

- The PROM is a combinational programmable logic device (PLD).
- A Combinational PLD is an integrated circuit with programmable gates divided into an AND array and an OR array to provide an AND-OR sum of product implementation.

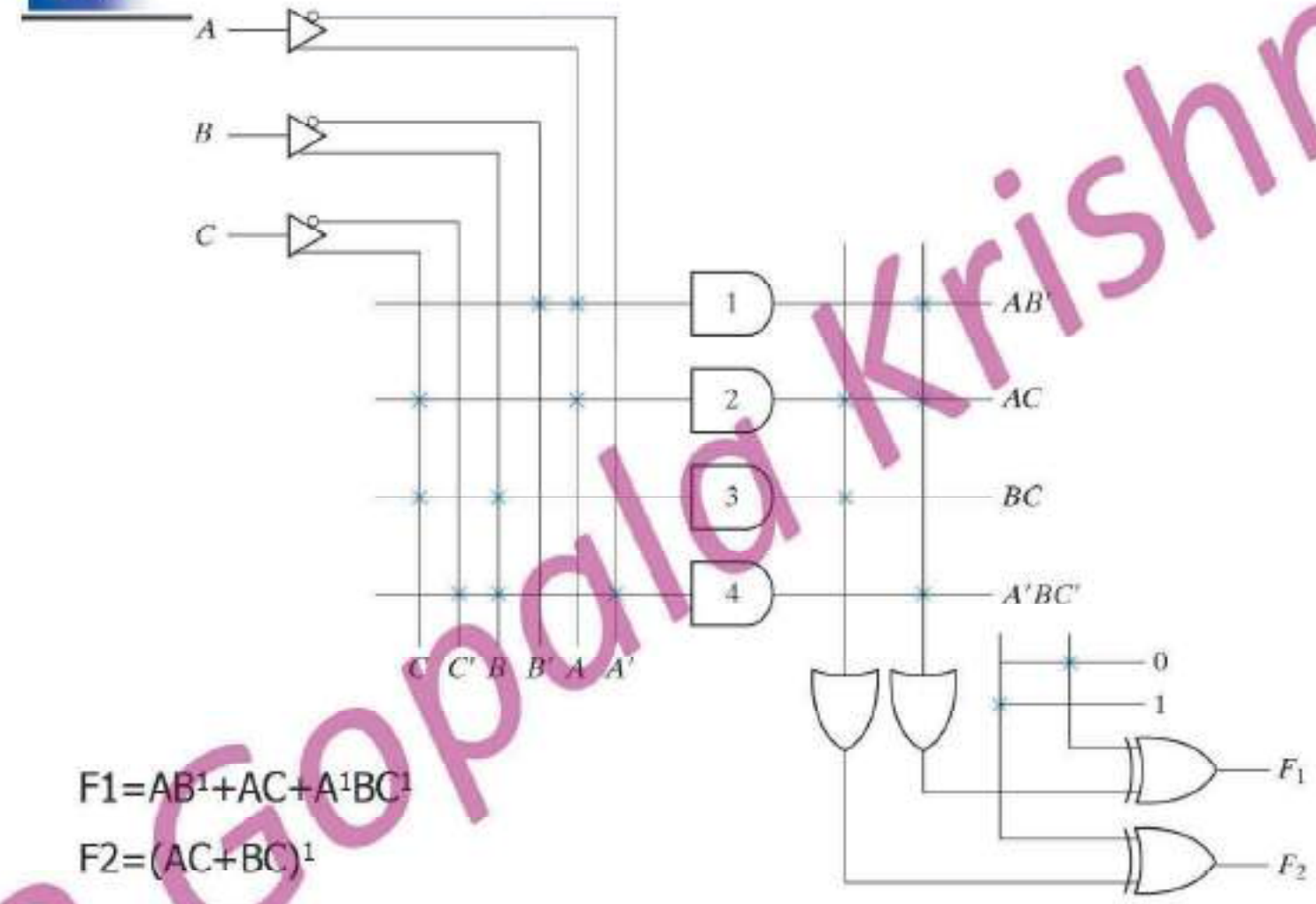
Types of combinational PLDs

- There are three major types of combinational PLDs and they differ in the placement of the programmable connections in the AND-OR array.



Programmable Logic Array

- The programmable logic array is similar to the PROM in concept except that the PLA does not provide full decoding of the variables and does not generate all the minterms.
- The decoder is replaced by an array of AND gates that can be programmed to generate any product term of the input variables.
- The product terms are the connected to the OR gates to provide the sum of products for the required boolean functions.



$F1=AB^1+AC+A^1BC^1$

$F2=(AC+BC)^1$

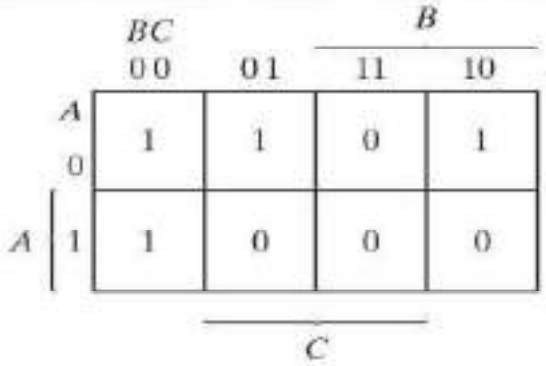
PLA Programming Table

	PRODUCT TERM	INPUTS A	B	C	OUPUTS (T) F1	(C) F2
AB^1	1	1	0	--	1	--
AC	2	1	--	1	1	1
BC	3	--	1	1	--	1
A^1BC^1	4	0	1	0	1	--

Example

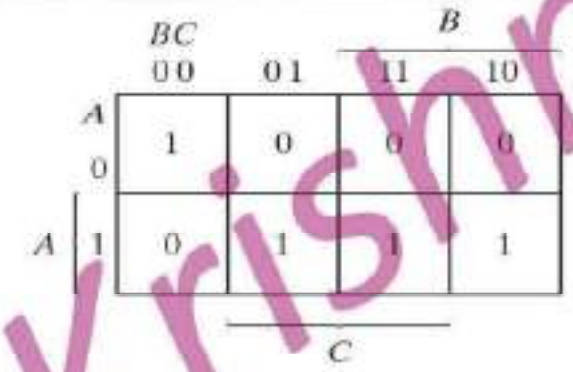
- Implement the following two boolean functions with a PLA:
  - F1(A,B,C)=Σ(0,1,2,4)
  - F2(A,B,C)=Σ(0,5,6,7)

Example



$F_1 = A'B' + A'C' + B'C'$

$F_1 = (AB + AC + BC)'$

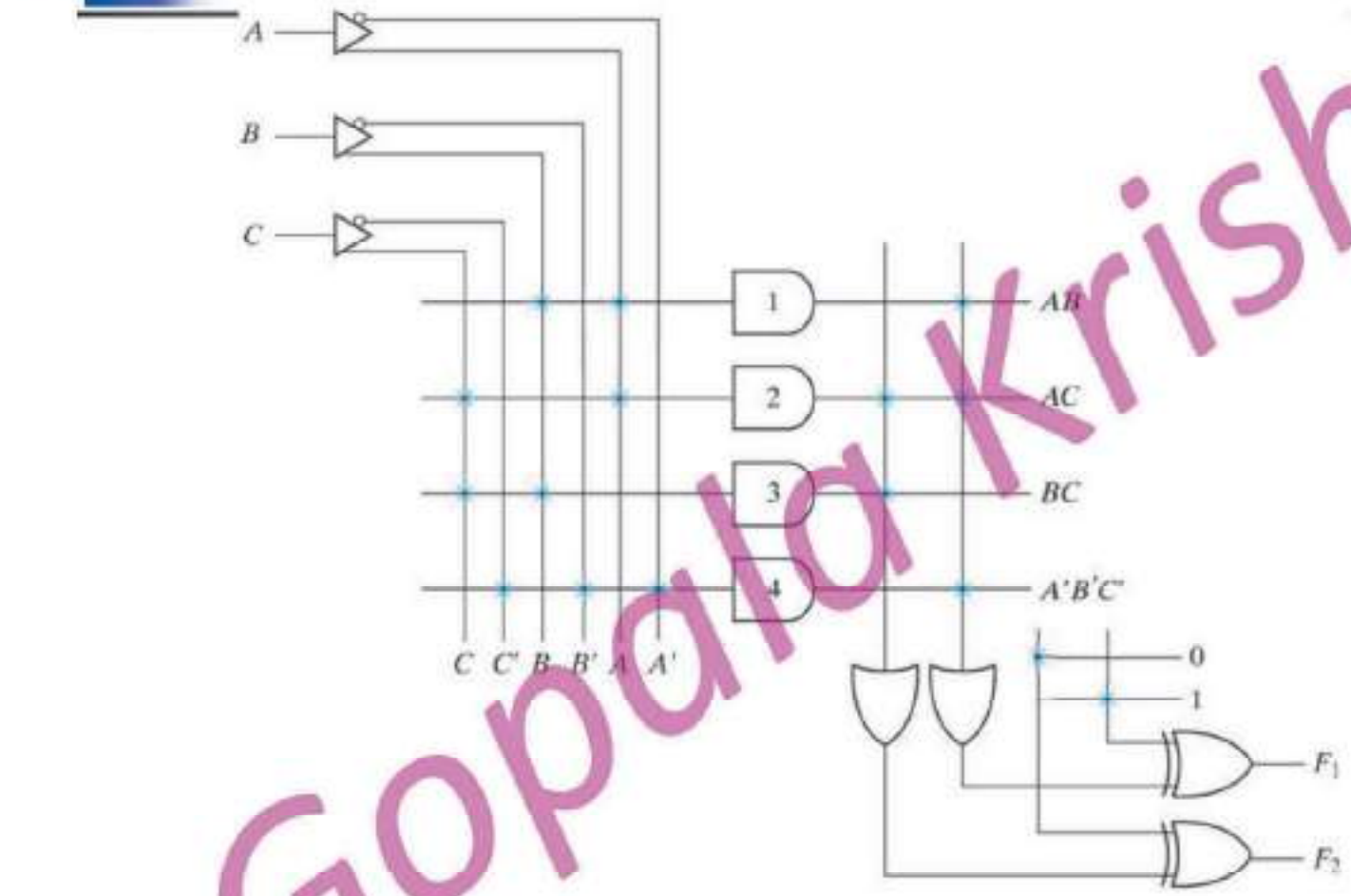


$F_2 = AB + AC + A'B'C'$

$F_2 = (A'C + A'B + AB'C')'$

PLA programming table

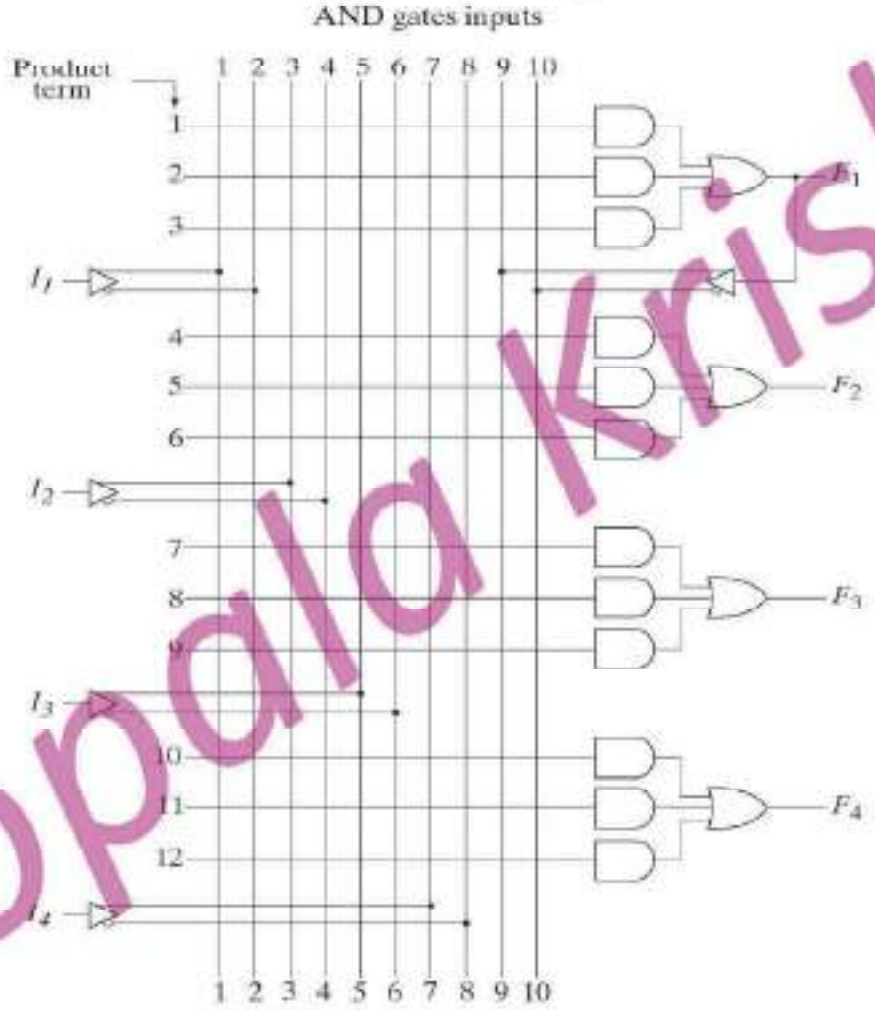
	Product term	Inputs A	B	C	Outputs (C) F_1	(T) F_2
AB	1	1	1	–	1	1
AC	2	1	–	1	1	1
BC	3	–	1	1	1	–
$A'B'C'$	4	0	0	0	–	1



Programmable Array Logic

- The programmable logic is a programmable logic device with a fixed OR array and a programmable AND array.
- Because only the AND gates are programmable, the PAL is easier to program, but is not as flexible as the PLA.

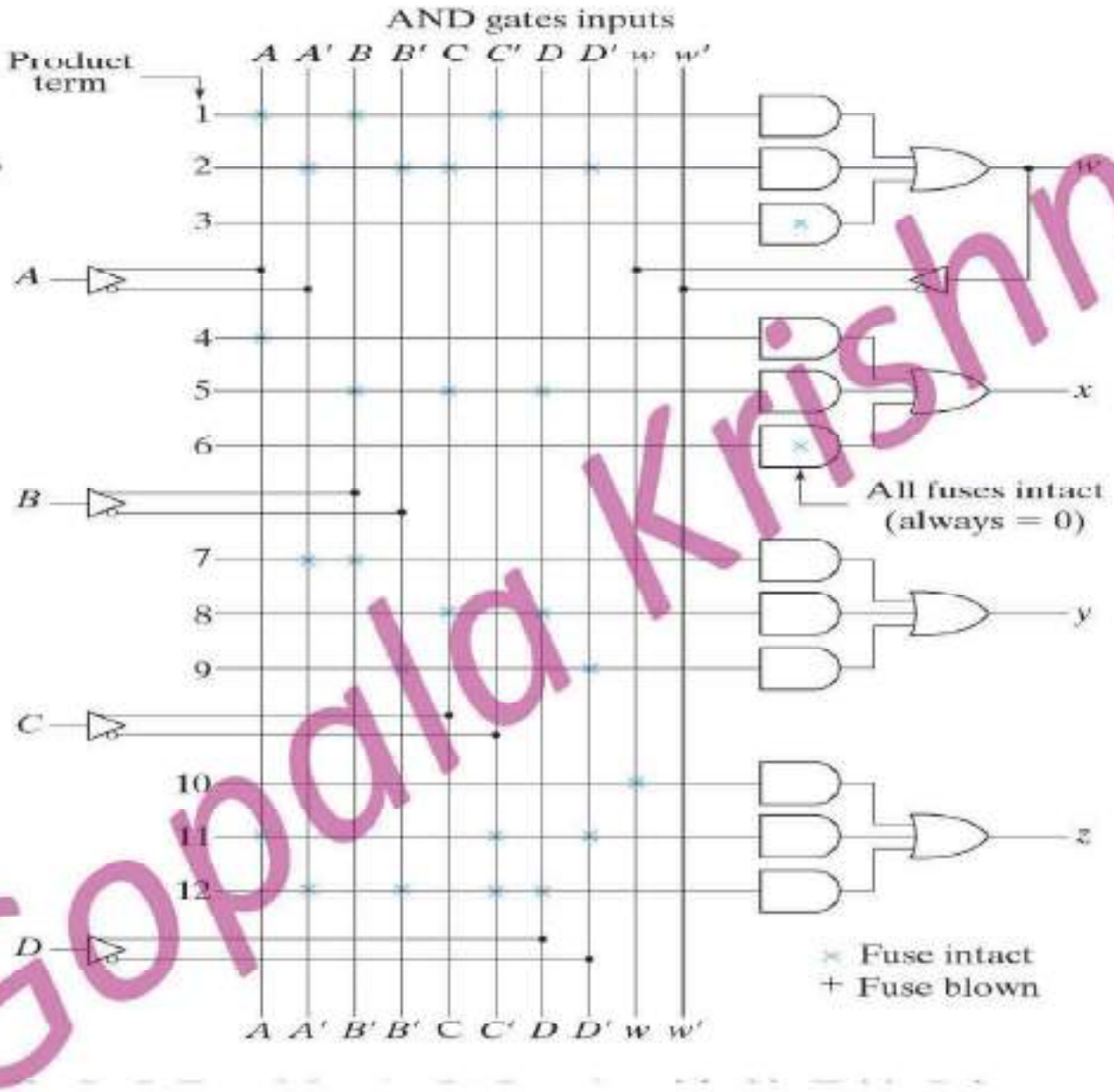
PAL Diagram



Example

- w(A,B,C,D) =Σ(2,12,13)
- X(A,B,C,D)= Σ(7,8,9,10,11,12,13,14,15)
- Y(A,B,C,D)= Σ(0,2,34,5,6,7,8,10,11,15)
- z(A,B,C,D)= Σ(1,2,8,12,13)

- $w=ABC^1+A^1B^1CD^1$
- $X=A+BCD$
- $Y=A^1B+CD+B^1D^1$
- $Z=ABC^1+A^1B^1CD^1+AC^1D^1+A^1B^1C^1D$
  $=W+AC^1D^1+A^1B^1C^1D$



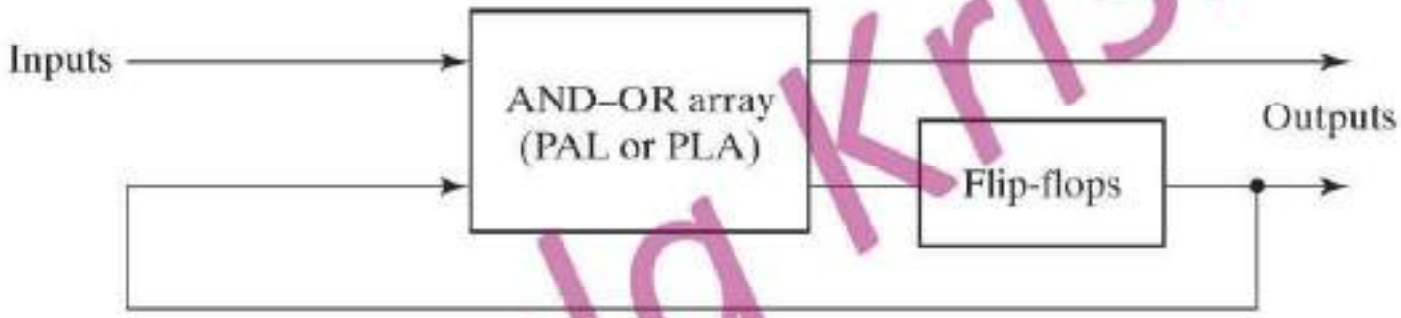
PAL Programming Table

Product term	AND Inputs					Outputs
	A	B	C	D	W	
1	1	1	0	--	--	
2	0	0	1	0	--	$w=ABC^1+A^1B^1CD^1$
3	--	--	--	--	--	
4	1	--	--	--	--	
5	--	1	1	1	--	$X=A+BCD$
6	--	--	--	--	--	
7	0	1	--	--	--	
8	--	--	1	1	--	$Y=A^1B+CD+B^1D^1$
9	--	0	--	0	--	
10	--	--	--	--	1	
11	1	--	0	0	--	$Z=W+AC^1D^1+A^1B^1C^1D$
12	0	0	0	1	--	

Sequential Programmable Devices

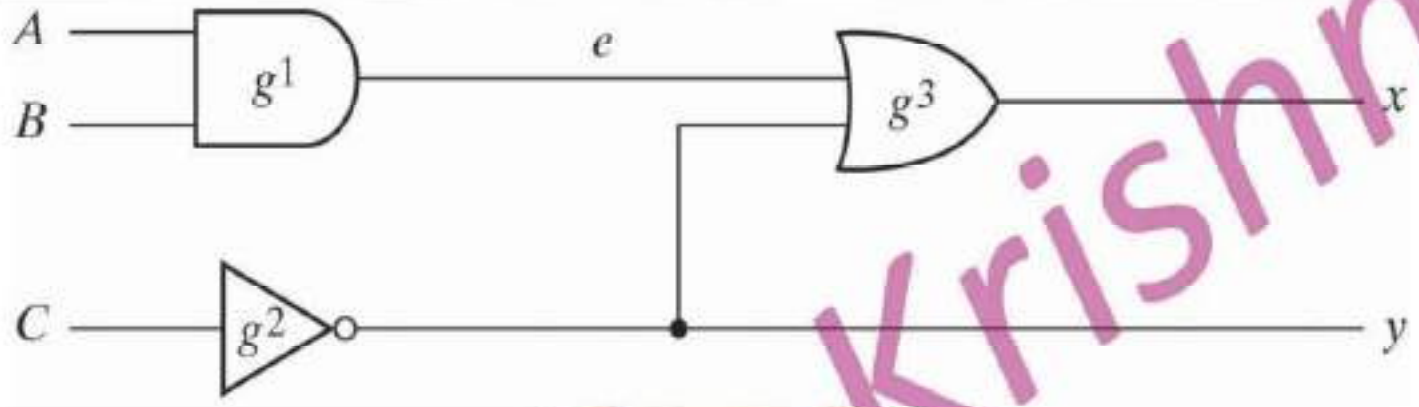
- Digital systems are designed using flip flops and gates.
- Since the combinational PLD consists of only gates, it is necessary to include external flip flops when they are used in the design.
- Sequential programmable devices include both gates and flip flops.
- These devices can be programmed to perform various sequential circuit functions.
- The major types of sequential programmable devices are:
  - Sequential (or simple) programmable logic device (SPLD)
  - Complex programmable logic device (CPLD)
  - Field programmable gate array (FPGA)

Sequential Programmable Logic Device



Hardware Description Language

- HDL resembles a programming Language, but it is specifically oriented to describe digital hardware.
- It represents logic diagrams and other digital information in textual form.
- It is used to simulate the system before its construction to check the functionality and verify its operation before it is submitted for fabrication.




```
//Description of simple circuit in the above figure
Module smpl_circuit(A,B,C,x,y);
    input A,B,C;
    output x,y;
    wire e;
    and g1(e,A,B);
    not g2(y,C);
    or g3(X,e,Y);
endmodule
```

Hardware Description Language

- A hardware description language is a language that describes the hardware of digital system in a textual form.
- It resembles a programming language, but is specifically oriented to describing hardware structures and behavior.
- As a documentation language, HDL is used to represent and document digital systems in a form that can be read by both human and computers and is suitable as an exchange language between designers.
- There are two applications of HDL Processing:
  - Simulation
  - Synthesis

Logic Simulation

- Logic Simulation is the representation of the structure and behavior of a digital logic system through the use of a computer.
- A simulator interprets HDL description and produces readable output, such as a timing diagram that predicts how the hardware will behave before it is actually fabricated.
- Simulation allows the detection of functional errors in a design without having to physically create the circuit.
- Errors that are detected during the simulation can be corrected by modifying appropriate HDL statements.
- The stimulus that tests the functionality of the design is called a test bench.
- Thus to simulate a digital system, the design is first described in HDL and then verified by simulating the design and checking it with a test bench which is also written in HDL.

Logic Synthesis

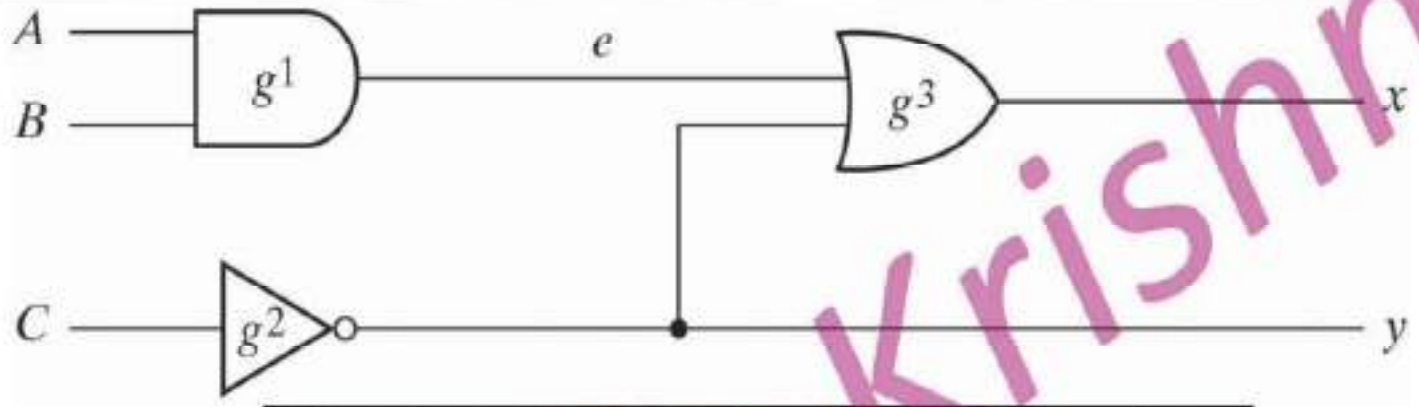
- Logic synthesis is the process of deriving a list of components and their interconnections (called netlist) from the model of a digital system described in HDL.
- The gate level netlist can be used to fabricate an integrated circuit or to layout a printed circuit board.
- The logic synthesis produces a database with instructions on how to fabricate a physical piece of digital hardware that implements the statements described by the HDL.
- Logic synthesis is based on formal exact procedures that implement digital circuits and consists of that part of a digital design that can be automated with computer software.

Proprietary HDL'S

- There are many Proprietary HDL's in industry developed by companies that design or help in design of integrated circuits.
- There are two standard HDL's supported by IEEE:
  - HDL
  - Verilog HDL

Module Representation

- Verilog HDL has a syntax describes precisely the legal constructs that can be used in the language.
- In particular, Verilog uses about 100 keywords predefined, lowercase, identifiers that define the language constructs. For example, module, endmodule, input, output, wire etc.,
- A module is the building block in Verilog. It is declared by the keyword module and is always terminated by the keyword endmodule.




```
//Description of simple circuit in the above figure
Module smpl_circuit(A,B,C,x,y);
    input A,B,C;
    output x,y;
    wire e;
    and g1(e,A,B);
    not g2(y,C);
    or g3(X,e,Y);
endmodule
```

Gate Delays

- When HDL is used during simulation, it is sometimes necessary to specify the amount of delay from the input to the output of gates. In verilog, the delay is specified in terms of time units and the symbol #. For example,

```
//Description of simple circuit with delay
Module circuit_with_delay(A,B,C,x,y);
    input A,B,C;
    output x,y;
    wire e;
    and #(30)  g1(e,A,B);
    or   #(20)   g3(X,e,Y);
    not  #(10)  g2(y,C);
Endmodule
```

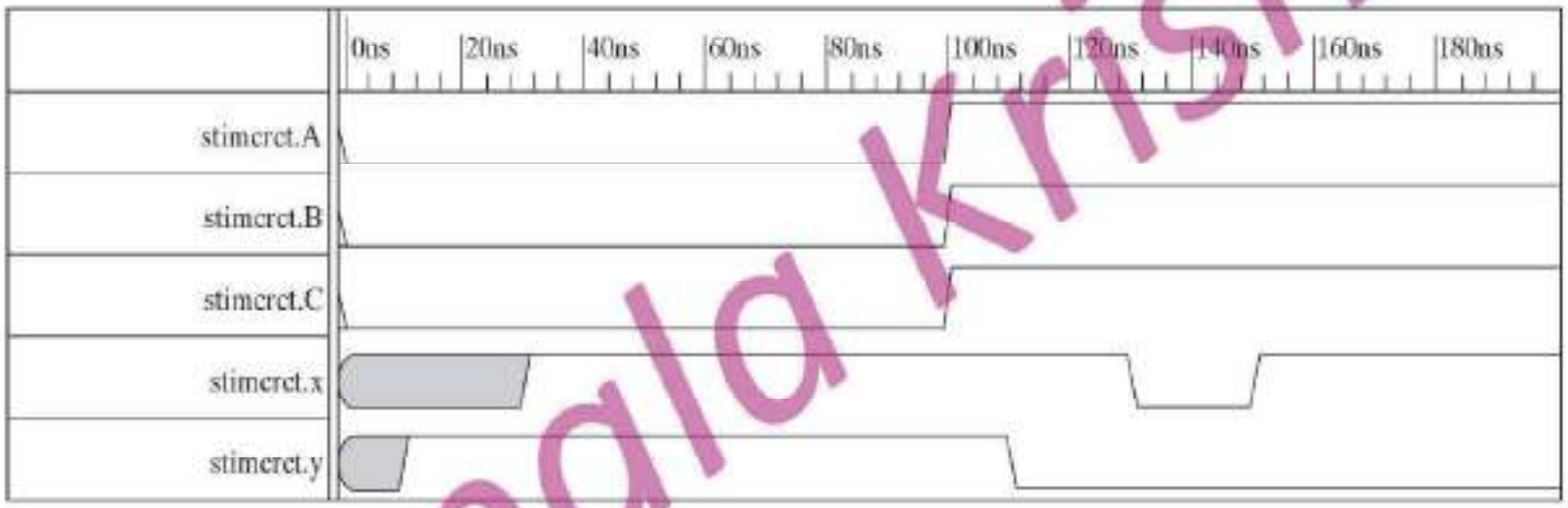
	Time in units	Inputs			Output		
	(ns)	A	B	C	y	e	x
Initial	--	0	0	0	1	0	1
Change	--	1	1	1	1	0	1
	10	1	1	1	0	0	1
	20	1	1	1	0	0	1
	30	1	1	1	0	1	0
	40	1	1	1	0	1	0
	50	1	1	1	0	1	1

Test Bench Example

```
// stimulus for simple circuit
Module stimcrct;
reg A,B,C;
Wire x,y;
circuit_with_delay cwd(A,B,C,x,y);
initial
    begin
        A=1 'b0; B=1 'b0; C=1 'b0;
      #100
        A=1 'b1; B=1 'b1; C=1 'b1;
      #100 $finish;
    end
endmodule
```

```
//Description of simple circuit with
delay
Module circuit_with_delay(A,B,C,x,y);
    input A,B,C;
    output x,y;
    wire e;
    and #(30)  g1(e,A,B);
    not #(20)  g2(y,C);
    or  #(10)  g3(X,e,Y);
Endmodule
```

Timing Diagram



Boolean Expression in HDL

- Boolean expressions are specified in Verilog HDL with a continuous assignment statement consisting of the keyword assign followed by a Boolean expression.
- To distinguish the arithmetic plus from logical OR, Verilog HDL uses the symbols (&),(|), and (~) for AND, OR, and NOT, respectively.

- The following example shows the description of a circuit that is specified with the Boolean expressions;
  - $x = A+BC+B^1D$
  - $y=B^1C+BC^1D^1$

```
// Circuit specified with Boolean expression
Module circuit_bin(x,y,A,B,C,D);
  input A,B,C,D;
  output x,y;
  assign x=A | (B & C) | (~B & D)
  assign y=(~B & C) | (B & ~C & ~D)
endmodule
```

User Defined Primitives

- The logic gates used in HDL descriptions with keywords and, or etc., are defined by the system and are referred to as system primitives.
- The user can create additional primitives by defining them in a tabular form.
- These types of circuits are referred to as user defined primitives.
- The user defined primitives are declared with the keyword primitive.

Rules for writing UDP

- It is declared with the keyword primitive followed by a name and port list.
- There can be only one output and it must be listed first in the port list and declared with an output keyword.
- There can be any number of inputs. The order in which they are listed in the input declaration must conform to the order in which they are given values in the table that follows.
- The truth table is enclosed within the keywords table and endtable.
- The values of the inputs are listed in order ending with a colon(:). The output is always the last entry in a row followed by a semicolon(;).
- It ends with the keyword endprimitive.

```
// user defined primitive (UDP)
Primitive crctp(x,A,B,C);
    output x;
    input A,B,C;
//Truth table for x(A,B,C)=(0,2,4,6,7)
   table
//A      B      C      :      X
   0      0      0      :      1;
   0      0      1      :      0;
   0      1      0      :      1;
   0      1      1      :      0;
   1      0      0      :      1;
   1      0      1      :      0;
   1      1      0      :      1;
   1      1      1      :      1;
   endtable
endprimitive
```

```
// instantiate primitive
Module declare_crctp;
reg x,y,z;
wire w;
crctp(w,X,Y,Z);
endmodule
```

HDL for Combinational Circuits

- A hardware description language is a language that describes the hardware of digital system in a textual form.
- The module is the basic building block of Verilog HDL.
- A module can be described in any one of the following modeling techniques:
  - Gate Level Modeling instantiation of primitive gates and user defined modules.
  - Dataflow Modeling using continuous assignment statements with keyword assign.
  - Behavioral Modeling using procedural assignment statements with keyword always.

- Gate Level modeling describes the circuit by specifying the gates and how they are connected with each other.
- Dataflow modeling is used for combinational circuits.
- Behavioral modeling is used to describe digital systems at a higher level of abstraction.

Gate - Level Modeling

- In this type of representation, a circuit is specified by its logic and their interconnection. It provides a textual description of a schematic diagram.
- Verilog recognizes 12 basic gates as predefined primitives.
- Four primitive gates are of the three state type.
- The other eight are represened with lower case letters and, nand, or, nor, xor, xnor, not, buf.
- When the gates are simulated, the system assigns a four valued logic set to each gate. In addition to two logic values of 0 and 1, there are two other values: unknown (x) and high impedance (z).

- An unknown value is considered during simulation for the case when input or output is ambiguous, for instance if it has not yet been assigned a value of 0 or 1.
- A high impedance condition occurs in the output of three state gates or if a wire is inadvertently left unconnected.

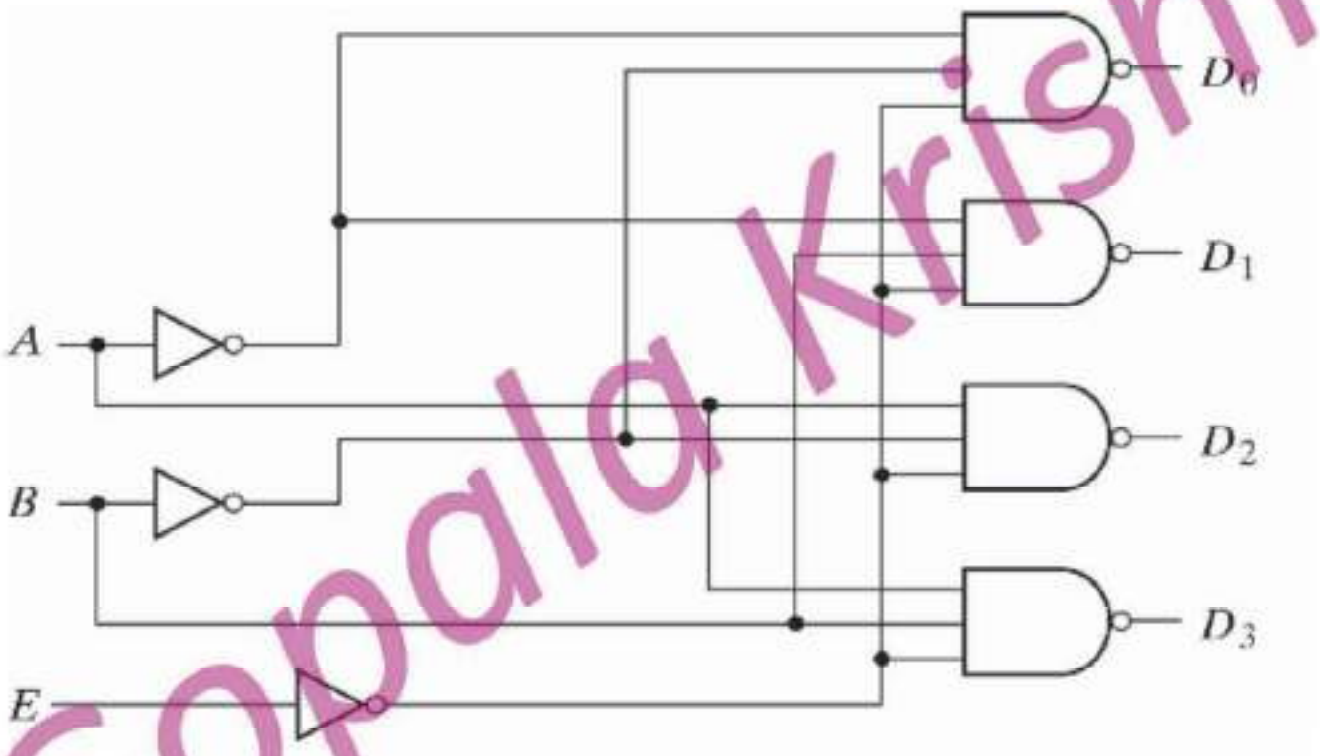
AND	0	1	X	Z
0	0	0	0	0
1	0	1	X	X
X	0	X	X	X
Z	0	X	X	X

OR	0	1	X	Z
0	0	1	X	X
1	1	1	1	1
X	X	1	X	X
Z	X	1	X	X

XOR	0	1	X	Z
0	0	1	X	X
1	1	0	X	X
X	X	X	X	X
Z	X	X	X	X

NOT	INPUT	OUTPUT
0	0	1
1	1	0
X	X	X
Z	Z	X

HDL Example

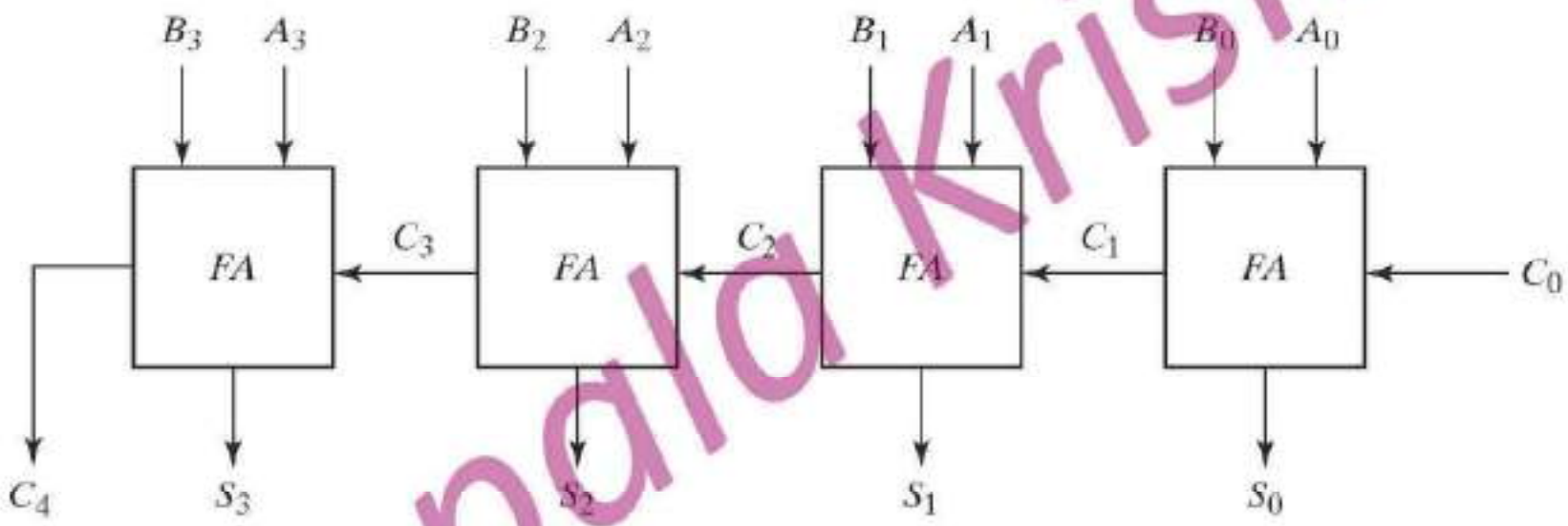


2-to-4 Line Decoder

HDL Example

```
//Gate Level description of a 2 to 4 line decoder
module decoder_g1(A,B,E,D);
        input   A,B,E;
       output   [0:3] D;
         wire   Anot, Bnot, Enot;
          not
           n1   (Anot,A);
           N2   (Bnot,B);
           N3   (Enot,E);
         Nand
           N4   (D[0],Anot,Bnot,Enot);
           N5   (D[1],Anot,B,Enot);
           N6   (D[2],A,Bnot,Enot);
           N7   (D[3],A,B,Enot);
  Endmodule
```

HDL Example



4-bit Adder

```
//Hierarchial description of 4-bit Adder
    Module Halfadder(S,C,x,y);
        Input   x,y;
        Output  S,C;
//Instantiate Primitive gates
        Xor     (S,x,y);
        And     (C,x,y);
    Endmodule

//description of Full Adder
    Module fulladder(S,C,x,y,z);
        Input   x,y,z;
        Ouput   S,C;
        Wire    S1,D1,D2;
    Halfadder   HA1(S1,D1,x,y);
                HA2(S,D2,S1,z);
        Or      g1(C,D2,D1);
    endmodule
```

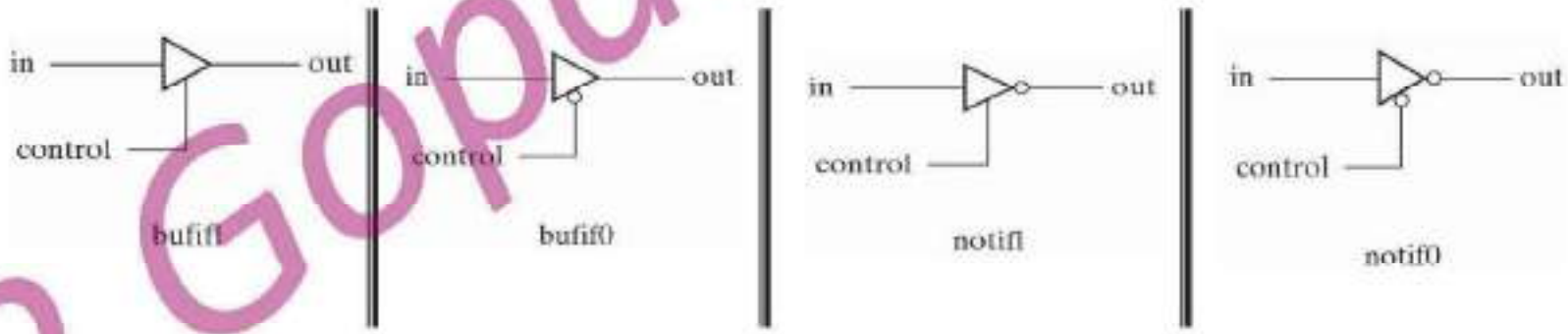
```
//description of 4 bit adder
Module _4bitadder(S,C4,A,B,C0);
    Input   [3:0] A,B;
    input   C0;
    output  [3:0] S;
    output  C4;
    wire    C1,C2,C3;
fulladder   FA0(S[0],C1,A[0],B[0],C0);
            FA1(S[1],C2,A[1],B[1],C1);
            FA2(S[2],C3,A[2],B[2],C2);
            FA3(S[3],C4,A[3],B[3],C3);

endmodule
```

Three State Gates

- The three state gate have a control input that can place the gate into a high impedance state.
- The high impedance state is symbolized by z in HDL.
- There are four types of three state gates.

- The bufif1 gate behaves like a normal buffer if control=1. the output goes to a high impedance state z when control=0.
- The bufif0 gate behaves like a normal buffer if control=0. the output goes to a high impedance state z when control=1.
- The two not gates operate in a similar manner except that output is the complement of the input.



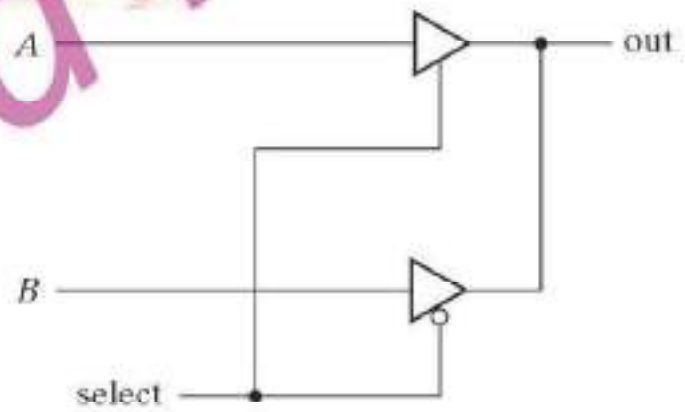
- The three state gates can be instantiated with the statement

  gatename (ouput,input,control)

- The gatename can be any one of the four three state gates. The output can result in 0,1,z. for example, bufif1(OUT,A,control)…
- The outputs of the three state gates can be connected together to form a common output line.
- To identify such a connection, HDL uses the keyword tri to indicate that the output has multiple drivers.

2-to-1 Line Multiplexer using three state buffers

```
Module muxtri (A, B, select, OUT);
        input A, B, select;
        output OUT;
        tri OUT;
        bufif1(OUT, A, select);
        bufif0(OUT, B, select);
endmodule
```



Dataflow Modeling

- Dataflow modeling uses a number of operators that act on operands to produce desired results.
- Verilog HDL provides about 30 operator types.

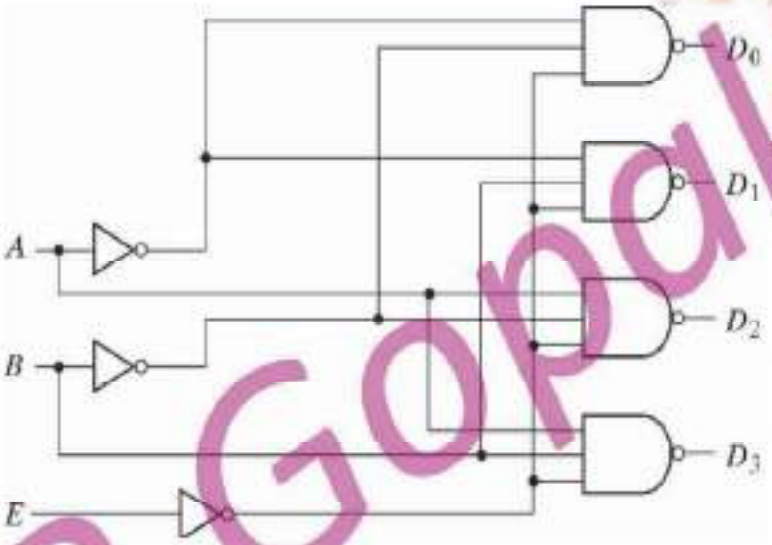
Symbol	Operation
+	Binary Addition
-	Binary Subtraction
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Bitwise NOT
==	Equality
>	Greater Than
<	Less Than
{ }	Concatenation
?:	Conditional

- Dataflow Modeling uses continuous assignments and the keyword **assign**.
- A continuous assignment is a statement that assigns a value to a net. The data type net is used in Verilog HDL to represent a physical connection between circuit elements.
- For example,

assign Y=(A & S) | (B & ~S)

Decoder- A Dataflow Model Example

```
//Dataflow Description of a 2-to4 Line Decoder
module  decoder_df(A,B,E,D);
 input  A,B,E;
output  [0:3] D;
assign  D[0]=~(~A & ~B & ~E);
        D[1]=~(~A & B & ~E);
        D[2]=~(A & ~B & ~E);
        D[3]=~(A & B & ~E);
endmodule
```



Dataflow Description of a binary adder

```
//Dataflow Description of a 4 bit adder

module  binary adder(A,B,Cin,SUM,Cout);

            Input   [3:0] A,B;

            Input   Cin;

           Output   [3:0] SUM;

           Output   Cout;

           Assign   {Cout,SUM}=A+B+Cin;

Endmodule
```

Dataflow Description of a comparator

```
//Dataflow Description of a 4 bit comparator

module      magcomp(A,B,ALSB,AGTB,AEQB);

     Input  [3:0] A,B;

    Output  ALTB,AGTB,AEQB;

    assign  ALTB=(A<B);

    assign  AGTB=(A>B);

    assign  AEQB=(A==B)

endmodule
```

2-to-1 Line Multiplexer with conditional operator

- The conditional operator takes three operands:

  condition ? True expression : false expression;

- The condition is evaluated, if the logic is 1, the true expression is evaluated. If the logic is 0, the false expression evaluated.

```
Module mux2x1_df (A, B, select, OUT);
        input A, B, select;
        output OUT;
        assign OUT=select ?A:B;
endmodule
```

Behavioral Modeling

- Behavioral modeling represents digital circuits at a functional and algorithmic level.
- It is mostly used to describe sequential circuits, but can be used also to describe combinational circuits.
- Behavioral description use the keyword **always** followed by a list of procedural assignment statements. The target output of procedural assignment statement must be of the **reg** type.
- Contrary to the **wire** data type, where the target output of an assignment may be continuously updated, a reg data type retains its value until a new value is assigned.

2-to-1 Line multiplexer- A Behavioral Model Example

```
//Behavioral Description of a 2-to-1 Line Multiplexer
Module   Mux2x1_bh(A,B,select,OUT);
Input    A,B,select;
Output   OUT;
Reg      OUT;
Always   @ (select or A or B)
         If (select==1) OUT=A;
         Else OUT=B;
Endmodule
```

4-to-1 Line multiplexer- A Behavioral Model Example

```
//Behavioral Description of a 4-to-1 Line Multiplexer
Module  Mux4x1_bh(i0,i1,i2,i3,select,y);
Input   i0,i1,i2,i3;
Input   [1:0] select;
Output  Y;
Reg     y;
Always  @ (i0 or i1 or i2 or i3 or select)
        Case (select)
        2'b00:  y=i0;
        2'b01:  y=i1;
        2'b10:  y=i2;
        2'b11:  y=i3;
Endmodule
```

Writing A Simple Test Bench

- A test bench is an HDL program used for applying stimulus to an HDL design in order to test it and observe its response during simulation.
- Test benches can be quite complex and lengthy and may take longer to develop than the design that is tested.
- In addition to the always statement, test benches use the **initial** statement to provide stimulus to the circuit under test.
- The always statement executes repeatedly in a loop. The initial statement executes only once starting from simulation time=0 and may continue with any operation that are delayed by a given number of time units as specified by # symbol.

- For example, consider the initial block:

```
initial
        begin
                A=0;B=0;
        #10     A=1;
        #20     A=0;B=1;
        end
```

- The block is enclosed between the keywords begin and end. At time=0, A and B are set to 0. 10 time units later, A is changed to 1. 20 time units later A is changed to 0 and B to 1.
- Another example of producing inputs to a 3-bit truth table can be generated with the following initial block:

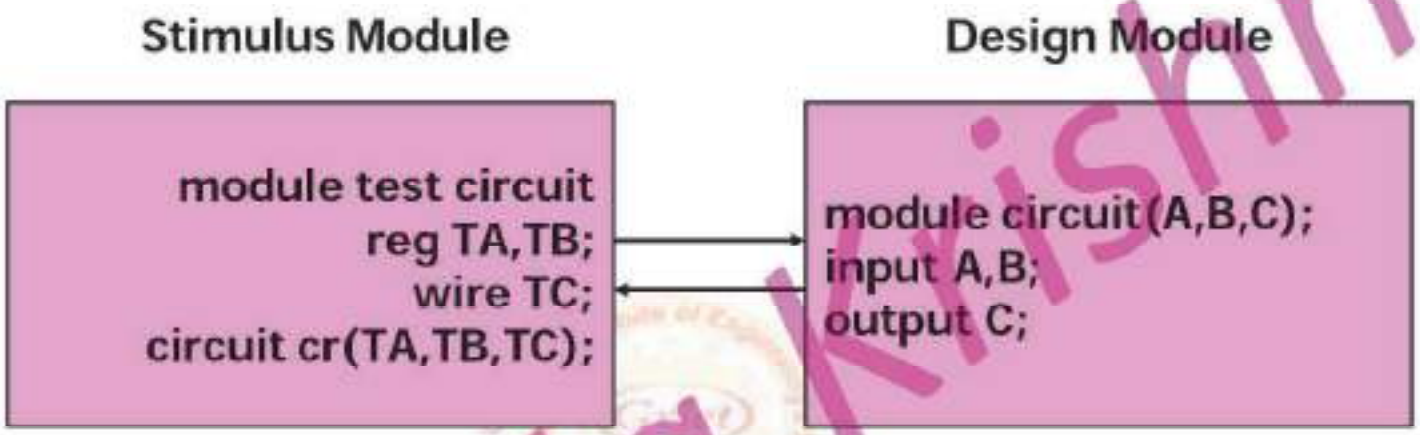
```
initial
        begin
                D=3'b000;
                repeat(7)
        #10     D=D+3'b001;
        end
```

- A stimulus module is an HDL program that has the following form:

  Module testname.
  Declare local **reg** and **wire** identifiers
  Instantiate the design module under test
  Generate stimulus using **initial** and **always** statements
  Display the output response
  endmodule

- A test module has no inputs or outputs. The signals that are applied as inputs to the design module for simulation are declared in the stimulus module as local and reg data type.
- The outputs of the design module that are displayed for testing are declared in the stimulus module as wire data type.
- The module under test is then instantiated using the local idenifiers.

Stimulus and design modules interaction



- The response to the stimulus generated by the initial and always blocks will appear at the output of the simulator as timing diagrams. It is also possible to display numeric outputs using Verilog system tasks.

Verilog System Tasks

- These are the system built-in functions that are recognized by keywords that begin with the symbol $.
- Some of the system tasks useful for display are:
  - $display- display one time value of variables or strings with end of line return.
  - $write- same as $display but without going to next line.
  - $monitor- displays variables whenever a value changes during simulation run.
  - $time- displays simulation time.
  - $finish- terminates the simulation.
- The syntax for $display, $write, and $monitor is of the form:

Taskname (format specification, argument list);

Test Bench for 2X1 Multiplexer

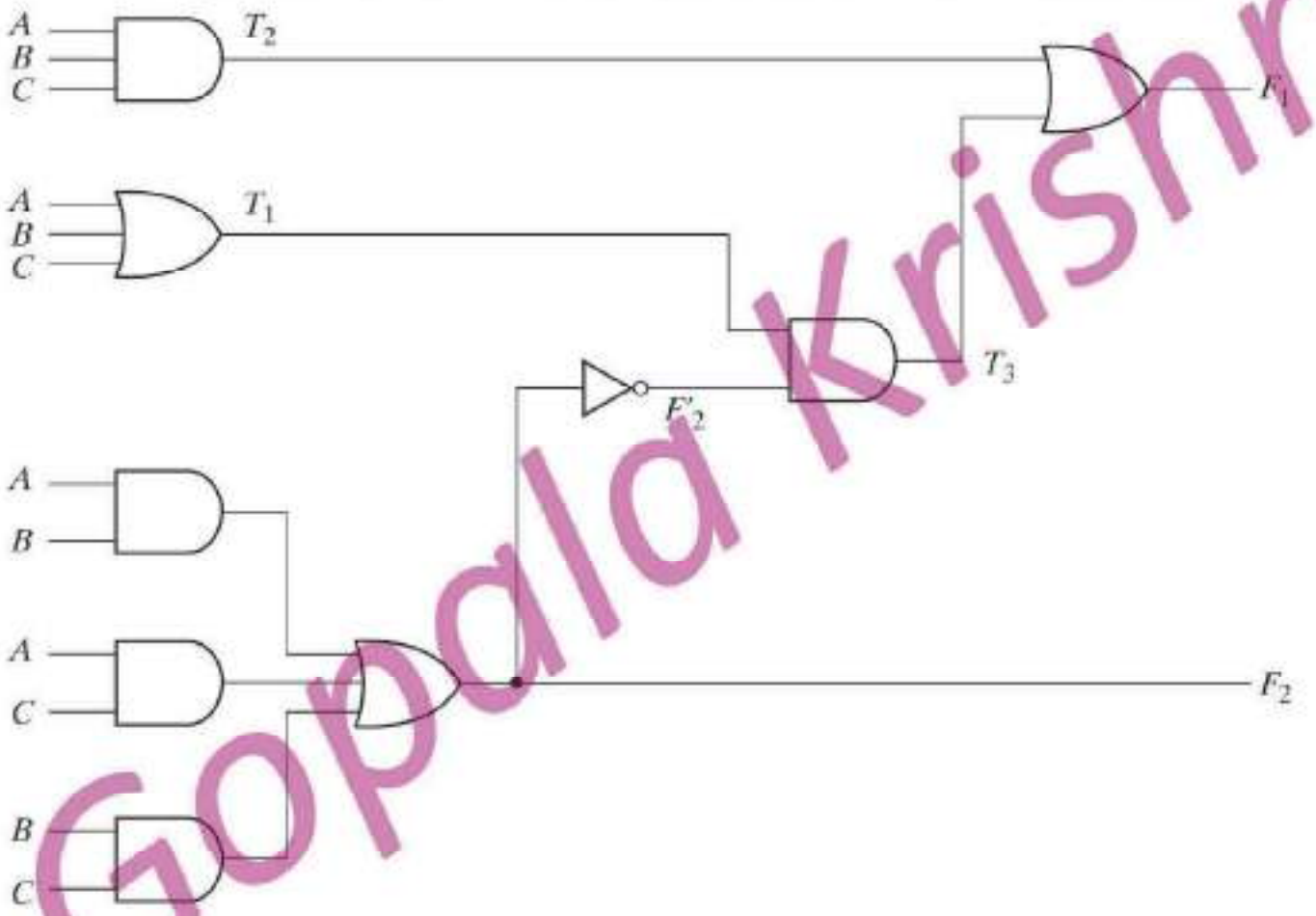
```
// stimulus for mux2x1_df
    module   testmux;
       reg   TA,TB,TS;  //inputs for mux
      wire   Y;  //outputs for mux
Mux2x1_df    Mx(TA,TB,TS,Y);
             //instantiate mux
   initial
   Begin
             TS=1;TA=0;TB=1;
       #10   TA=1;TB=0;
       #10   TS=0;
       #10   TA=0;TB=1;
       end
   initial
$monitor     ("select=%b   A=%b   B=%b
             OUT=%b          TIME=%d",
             TS,TA,TB,Y,$time);
```

```
//data flow description of 2 to 1 line
multiplexer
    Module   Mux2x1_df(A,B,select,OUT);
     input   A,B,select;
    Output   OUT;
    assign   OUT=select ? A:B;
endmodule
```

Simulation Log:

```
Select=1 A=0 B=1 OUT=0 time=0
Select=1 A=1 B=0 OUT=1 time=10
Select=0 A=1 B=0 OUT=0 time=20
Select=0 A=0 B=1 OUT=1 time=30
```

Test Bench for Gate Level Description



Test Bench for Gate Level Description

```
// gate level description of the circuit
        Module  Analysis(A,B,C,F1,F2);
         Input  A,B,C;
        Output  F1,F2;
          Wire  T1,T2,T3,F2NOT,E1,E2,E3;
            Or  g1(T1,A,B,C);
           And  g2(T2,A,B,C);
           And  g3(e1,A,B);
           And  g4(e2,A,C);
           And  g5(e3,B,C);
            Or  g6(F2,E1,E2,E3);
           Not  g7(F2NOT,F2);
           And  g8(T3,T1,F2NOT);
            Or  g9(F1,T2,T3);
     Endmodule
```

```
//stimulus to analyze the circuit
Module  Test_circuit;
Reg     [2:0]D;
Wire    F1,F2;
Analysis Fig42(D[2],D[1],D[0],F1,F
        2);
Initial
        Begin
        D=3'b000;
        Repeat(7)
        #10  D=D+b'001;
        End
Initial
$monitor ("ABC=%b          F1=%b
        F2=%b ",D,F1,F2);
```

```
//SCANNING LOG:
ABC=000     F1=0     F2=0
ABC=001     F1=1     F2=0
ABC=010     F1=1     F2=0
ABC=011     F1=0     F2=1
ABC=100     F1=1     F2=0
ABC=101     F1=0     F2=1
ABC=110     F1=0     F2=1
ABC=111     F1=1     F2=1
```

HDL for Sequential Circuits

- Behavioral Modeling: There are two kinds of Behavioral Statements in Verilog HDL: initial and always.
  - Initial: the initial behavior executes once beginning at time=0.
  - Always: the always behavior executes repeatedly and re executes until the simulation terminates.
- The behavior is declared with in a module by using the keywords initial or always followed by a set of statements enclosed by the keywords begin and end.

Initial and always - example

```
initial
begin
  clock=1'b0;
  repeat(30)
  #10 clock=~clock;
end
```

```
initial
begin
  clock=1'b0;
  #300 $finish;
end
always
    #10 clock=~clock;
```

Procedural assignment statements

- The always statement can be controlled by delays that wait for a certain time or by certain conditions to become true or by events to occur.
- **Always @(event control expression) procedural assignment statements.**
- For example, always @(A or B or Reset), always @(posedge clock or negedge reset)
- There are two types of procedural assignment statements: blocking and non blocking.
- Blocking: blocking assignment statements executed in the sequential order as in the block. These statements use the symbol (=).
- Non Blocking: these statements evaluate the expression on the right hand side, but do not make the assignment to the left hand side until all expression are evaluated. These statements use the symbol (<=).
- B=A, C=B+1 (Blocking statements)
- B<=A, C<=B+1 (Non Blocking statements)

HDL for D Latch/Flip Flop

```
// Description of a D Latch
Module D_Latch(Q,D,control)
  input D,control;
  output Q;
  reg Q;
  always @(control or D)
  if (control==1)   Q=D;
endmodule
```

```
// Description of a D Flip Flop
Module D_FF(Q,D,CLK)
  input D,CLK;
  output Q;
  reg Q;
  always @(posedge CLK)   Q=D;
endmodule
```

```
//D Flip Flop with asynchronous reset
Module DFF(Q,D,CLK,RST);
  input D,CLK,RST;
  output Q;   reg Q;
  always @(posedge CLK or negedge RST)
If(~RST) Q=1'b0;
  else Q=D;
endmodule
```

HDL for T & JK Flip Flops using D Flip Flop

```
//T Flip Flop from D flip flop and gates
module TFF(Q,T,CLK,RST);
        Input   T,C,CLK,RST;
       Output   Q;
         Wire   DT;
       Assign   DT=Q^T;
          DFF   TF1(Q,DT,CLK,RST);
    Endmodule
//JK Flip Flop from D Flip Flop and gates
Module JKFF(Q,J,K,CLK,RST);
        Input   J,K,CLK,RST;
       Output   Q;
         Wire   JK;
       Assign   JK=(J&~Q) | (~K&Q);
          DFF   JK1(Q,JK,CLK,RST);
  endmodule
```

```
//D Flip Flop
module DFF(Q,D,CLK,RST);
        Input   D,CLK,RST;
       Output   Q;
          Reg   Q;
Always @ (posedge CLK or NEGEDGE RST)
           If   (~RST) Q=1'b0;
         Else   Q=D;
  endmodule
```

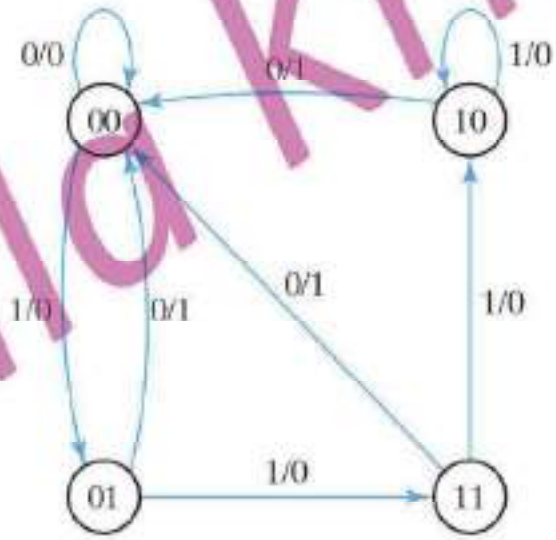
HDL for JK Flip Flop

```
// Functional description of JK Flip Flop
        Module  JK_FF(J,K,CLK,Q,QNOT);
        Output  Q,QNOT;
        Input   J,K,CLK;
        Reg     Q;
        Assign  QNOT=~Q;
        Always  @(posedge CLK)
                Case({J,K})
                2'b00:Q=Q;
                2'b01:Q=0;
                2'b10:Q=1;
                2'b11:Q=~Q;
        Endcase
Endmodule
```

J	K	Q(t+1)	
0	0	Q(t)	No change
0	1	0	Clear to 0
1	0	1	Set to 1
1	1	$Q^1(t)$	Complement

HDL for State Diagram

- The operation of sequential circuit is described in HDL in the same format as a state diagram.

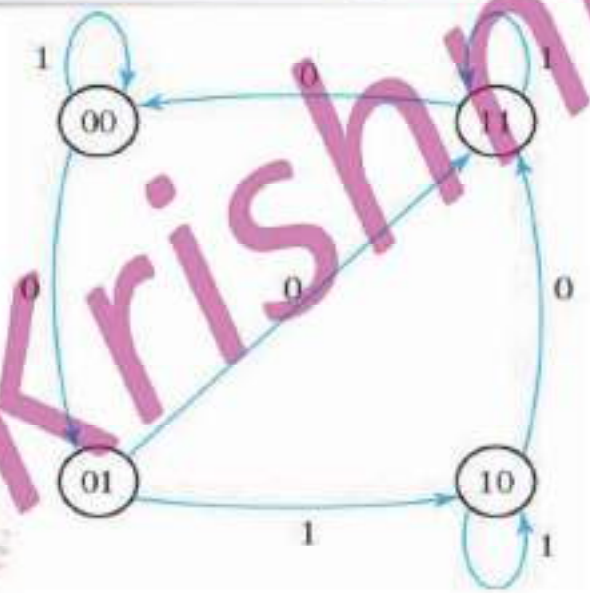


HDL for a Mealy State Diagram

```
//Mealy State Diagram
module mealy_md1(X,Y,CLK,RST);
        Input   X,CLK,RST;
        Output  Y;
        Reg     Y;
        Reg     [0:1] prstate,nxtstate;
        Parameter  S0=2'b00;
                   S1=2'b01;
                   S2=2'b10;
                   S3=2'b11;
Always @(posedge CLK or negedge RST)
        If(~RST)  Prstate=S0;
             Else  Prstate=nxtstate;
        Always  @(prstate or X)
        Case    (prstate)
        S0:     If(x) nxtstate=S1;
             Else  Nxtstate=S0;
        S1:     If(x) nxtstate=S3;
             Else  Nxtstate=S0;
        S2:     If(~x) nxtstate=S0;
             Else  Nxtstate=S2;
        S3:     If(x) nxtstate=S2;
             Else  Nxtstate=S0;
        Endcase
        Always  @(prstate or x)
        Case    (prstate)
        S0:     Y=0;
        S1:     If(x) y=1'b0;else y=1'b1
        S2:     If(x) y=1'b0;else y=1'b1
        S3:     If(x) y=1'b0;else y=1'b1
        Endcase
Endmodule
```

HDL for a Moore State Diagram

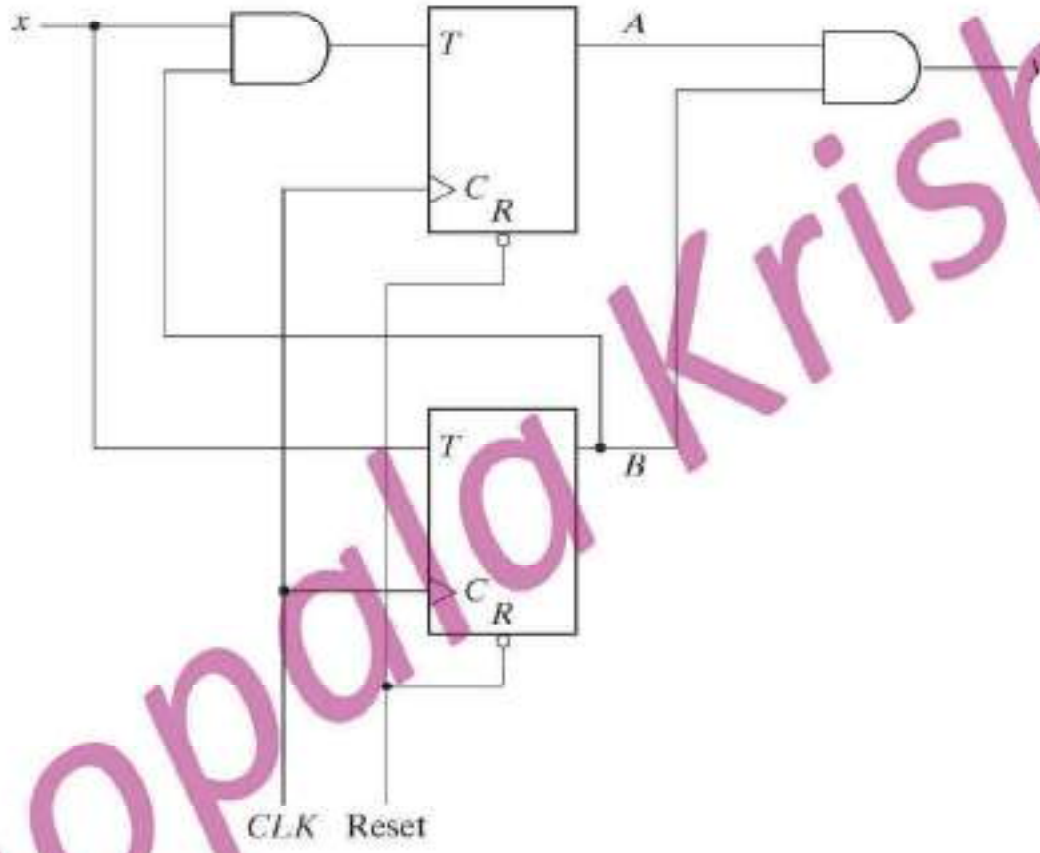
```
//Moore State Diagram
module moore_md1(X,A,B,CLK,RST);
        Input   X,CLK,RST;
        Output  [1:0]AB;
        Reg     [1:0]state;
        Parameter  S0=2'b00;
                   S1=2'b01;
                   S2=2'b10;
                   S3=2'b11;
Always @(posedge CLK or negedge RST)
        If(~RST)  state=S0;
            Else
        Case  (state)
            S0:  If(~x) state=S1;else state=S0;
            S1:  If(x) state=S2;else state=S3;
            S2:  If(~x) state=S3;else state=S2;
            S3:  If(~x) state=s0;else state=S3;
        Endcase
        Asssign  AB=state;
endmodule
```



Structural Description

- Combinational circuits can be described in HDL by using gate level or data flow statements.
- Sequential circuits use behavioral statements to describe the flip flop operation.
- Since sequential circuits made up of flip flops and gates, its structure can be described by a combination of dataflow and behavioral statements.
- The flip flops are described with an always statement. The combinational part can be described with assign statements and Boolean equations.
- The separate modules can be combined by instantiation.

Structural Description-Example

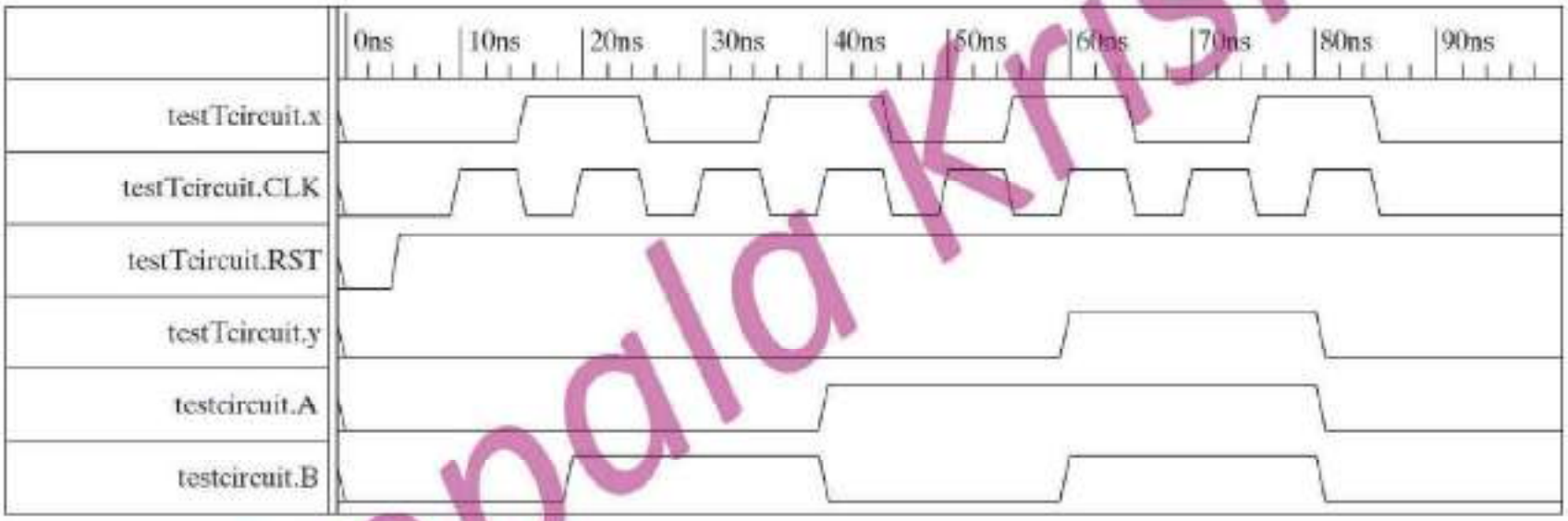


HDL for Structural Description

```
//Structural Description of sequential circuit
module Tcircuit (X,Y,A,B,CLK,RST);
        Input   X,CLK,RST;
        Output  Y,A,B;
        Wire    TA,TB;
        Assign  TB=X;
                TA=X&B;
        Assign  Y=A&B;
        T_FF    BF(B,TB,CLK,RST);
        T_FF    AF(A,TA,CLK,RST);
Endmodule
//T Flip Flop
        Module  T_FF(Q,T,CLK,RST);
        Output  Q;
        Input   T,CLK,RST;
Always@(posedge CLK or negedge RST)
        If(~RST)  Q=1'b0;
        Else      Q=Q^T;
endmodule
```

```
//stimulus for testing circuit
        Module   Testcircuit;
        Reg      X,CLK,RST;
        Wire     Y,A,B;
        Tcircuit TC(x,y,A,B,CLK,RST);
        Initial
                 RST=0;
                 CLK=0;
        #5       RST=1; Repeat(16)
        #5       CLK=~CLK;
        End
        Initial
        Begin
                 X=0;
        #15      X=1;
                 Repeat(8);
        #10      X=~x;
        End
endmodule
```

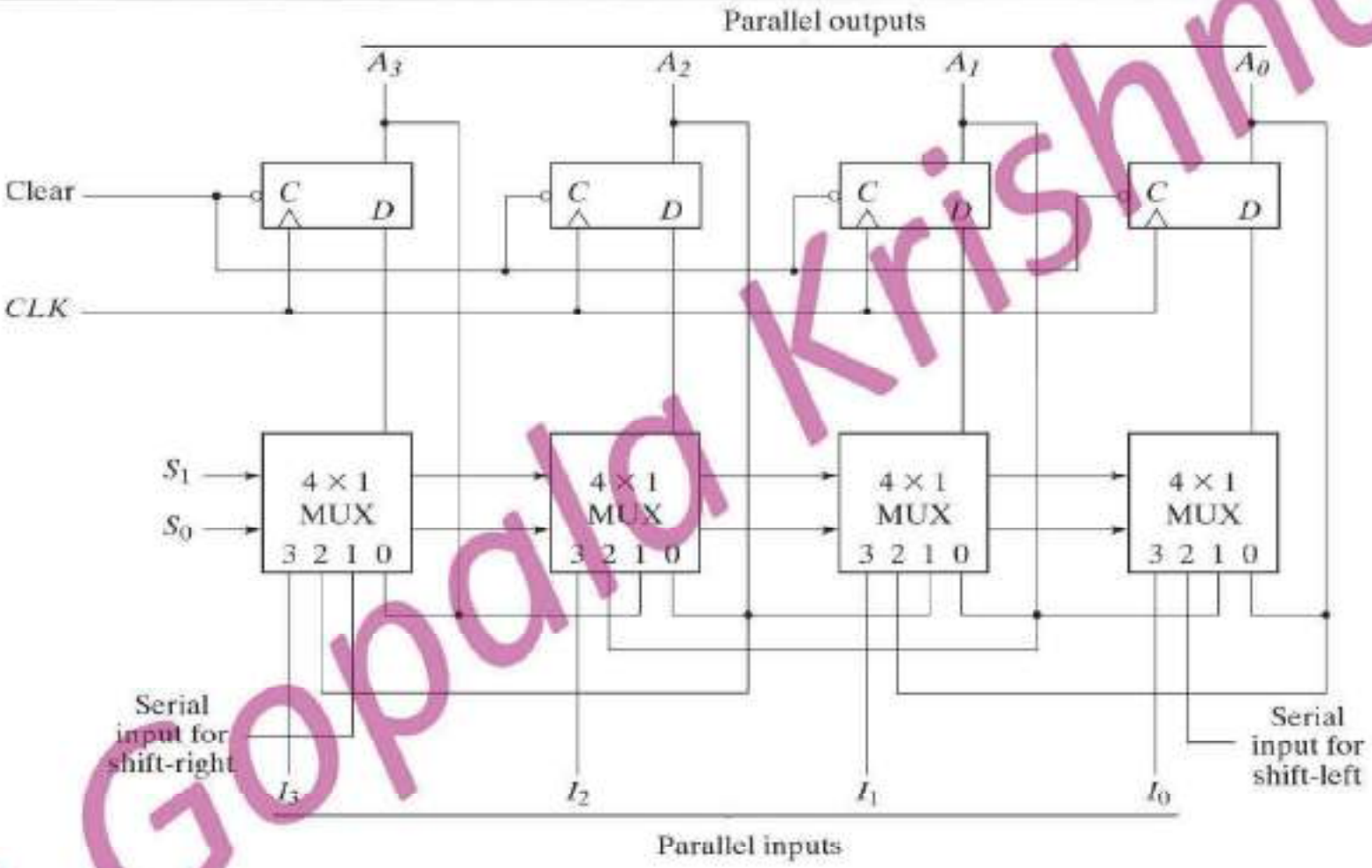
Simulation Output



HDL for Registers and Counters

- Registers and counters can be described in HDL at either behavioral or the structural level.
- In the behavioral level, the register is specified by a description of the various operations that it performs similar to a function table.
- A structural level description shows the circuit in terms of collection of components such as gates, flip flops, and multiplexers.

HDL for Universal Shift Register



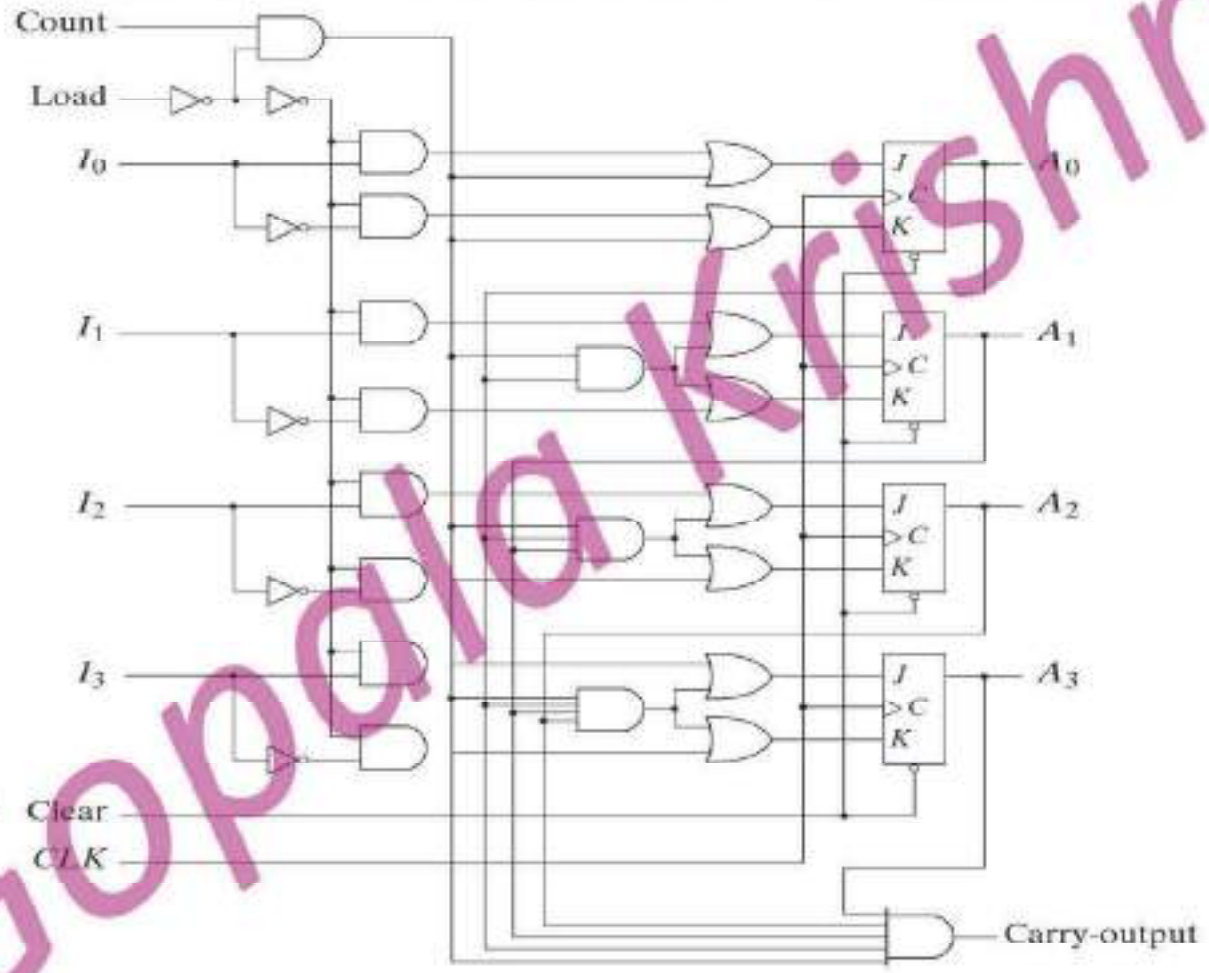
HDL for Universal Shift Register

```
// Behavioral Description of Universal Shift Register
Module   shftreg(s1,s0,pin,lfin,rtin,A,clk, Clr);
Input    s1,s0;
Input    lfin,rtin;
Input    clk,Clr;
Input    [3:0] pin;
Output   [3:0] A;
Reg      [3:0] A;
Always   @(posedge clk or negedge Clr)
         If(~clr) A=4'b0000;
         Case({s1,s0})
         2'b00:  A=A;
         2'b00:  A={rtin,A[3:1]};
         2'b00:  A={A[2:0],lfin};
         2'b11:  A=Pin;
         Endcase
endmodule
```

Mode Control

S1	S0	Register Operation
0	0	No change
0	1	Shift right
1	0	Shift left
1	1	Parallel load

HDL for Binary Counter with Parallel Load



HDL for Binary Counter with Parallel Load

```
//Binary Counter with Parallel Load
        Module  Counter(count,load,in,clk,clr,a,co);
         Input  Count,load,clk,clr;
         Input  [3:0]in;
        Output  Co;
        Output  [3:0]a;
           Reg  [3:0]a;
        Always  @(posedge clk or negedge clr)
                If(~clr) a=4'b0000;
                Else if(load) a=in;
                Else if(count) a=a+1'b1;
                Else a=a;
     endmodule
```

HDL for Read and Write Operations of Memory

```
//Read and Write operations of Memory and its size is 64 words of 4 bits each.
        Module   Memory(enable,readwrite,address,datain,dataout);
         Input   enable,readwrite;
         Input   [3:0]datain;
         Input   [5:0]address;
        Output   [3:0] dataout;
           Reg   [3:0] dataout;
           Reg   [3:0] mem [0:63];
        Always   @(enable or readwrite)
            If   (enable)
            If   (readwrite)
                 Dataout=mem[address];
          Else
                 Mem[address]=datain;
          Else   Dataout=4'bz;
     endmodule
```