DSCE

Dept. of Information Science & Engineering

DAYANANDA SAGAR COLLEGE OF ENGINEERING
Department of Information Science & Engineering

Shavige Malleshwara Hills, Kumaraswamy Layout, Bangalore-560078

LAB MANUAL
for

DIGITAL SYSTEMS AND LOGIC DESIGN LABORATORY
[18IS3DLLDL]

ACADEMIC YEAR 2019-20

FOR III Semester B. E. ISE

Faculty in Charge

Dr.. Sharon Christa, Ms. Bindu Bhargavi S M
Asst. Professor, Dept. of ISE
DSCE, Bangalore-78

DIGITAL SYSTEMS AND LOGIC DESIGN LABORATORY

Course code: 18IS3DLLDL **Credits: 02**
L: P: T: S: 0:3:0: 0 **CIE Marks: 50**
Exam Hours:03 **SEE Marks: 50**
Total Hours:24

Course objectives: Students undergoing this course are expected to:

1. Gain knowledge of various basic gates as well as its application in data processing circuits.
2. Will learn the designing and working of various data processing circuits.
3. Will learn the working of flip-flops and verify the working of various registers.
4. Will learn the design, implementation and working of synchronous as well as asynchronous circuits.
5. Will learn the Verilog implementation of digital circuits.

Course Outcomes: At the end of the course, student will be able to:

CO1	stand the working of different components used in logic design lab
CO2	Familiarize with the IC's and its layout
CO3	Analyze requirements and design combinational and sequential circuits from it.
CO4	Verify the working of flip-flops and registers
CO5	and verify the working of synchronous and asynchronous circuits.
CO6	the Verilog implementation of various digital circuits.

Mapping of Course outcomes to Program outcomes:

	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3
CO1	3	2	2	-	1	-	-	-	-	-	-	-	2	1	-
CO2	3	2	2	-	1	-	-	-	-	-	-	-	2	1	-
CO3	3	2	2	-	1	-	-	-	-	-	-	-	2	1	-
CO4	3	2	2	-	1	-	-	-	-	-	-	-	2	1	-
CO5	3	2	2	-	1	-	-	-	-	-	-	-	2	1	-
CO6	3	2	2	-	1	-	-	-	-	-	-	-	2	1	-

I. Familiarization of digital IC's and digital IC trainer kit by verifying the truth tables

II. Study of Combinational Circuits:

1. Universal Gates
    a. Simplify the given expression and realize it with universal gates
    b. For the given expression, design and develop a Verilog code after simplifying it. Simulate and verify the working of the same.
2. Adder Subtractor Circuit
    a. Design and verify the truth table of full adder and half subtractor circuit
    b. Design and develop the Verilog code for full adder. Simulate and verify the working of the same.
3. Code Converter and Magnitude Comparator Circuit
    a. To design and set up the BCD to Excess-3 converter circuit
    b. Design and develop the Verilog code for a single bit magnitude comparator circuit. Simulate and verify the working of the same.
4. Data Processing Circuit
    a. Given a four variable expression, simplify using Entered Variable Map (EVM) and realize the simplified logic using 8:1 MUX.
    b. Design and develop the Verilog code for 8:1 MUX. Simulate and verify the working of the same.

III. Flip Flops Using Gates and Familiarization of IC's:

5. SR and D flip flop
    a. To Setup SR and D flip flops using gates and verify the truth table also familiarize the flip flop ICs
    b. Design and develop the Verilog code for SR flip flop with positive edge triggering.
6. JK and T flip flop
    a. To Setup JK and T flip flops using gates and verify the truth table also familiarize the flip flop ICs
    b. Design and develop the Verilog code for T flip flop with positive edge triggering.
7. J-K Master/Slave FF
    a. Realize a J-K Master/Slave flip flop using only NAND gates and verify its truth table.
    b. Design and develop the Verilog code for JK flip flop with positive edge triggering.

IV. Shift Registers and Counters:

8. Shift Register
    a. To set up and verify the performance of 4 bit SISO shift register and 2 bit PISO shift register using D flip flop
    b. Design and develop the Verilog code to represent any give type of Register and verify the working of the same.

9. Ring and Johnson counter
    a. Design and implement ring counter using 4-bit shift register IC and Johnson counter using D flip flop demonstrate its working.
    b. Design and develop the Verilog code for ring counter. Simulate and verify the working of the same.
10. Sequence Generator
    a. Design and implement a sequence generator counter using D flip flop IC's and demonstrate its working.
    b. Design and develop the Verilog code for switched tail counter. Simulate and verify the working of the same.
11. Asynchronous Counter
    a. Design and implement asynchronous counter of the given modulus using decade counter IC and demonstrate its working.
    b. Design and develop the Verilog code for 3 bit counter. Simulate and verify the working of the same.

Practical Examination Procedure:

- All laboratory experiments are to be included for practical examination.
- Students should pick one experiment and execute both Part A and Part B.
- Strictly follow the instructions as printed on the cover page of answer script for breakup of marks

PART A
I. FAMILIARIZATION OF DIGITAL ICS AND DIGITAL IC TRAINER KIT BY VERIFYING THE TRUTH TABLES

AIM: To familiarize with logic gate IC packages and to verify the truth tables of logic gates. Also familiarize with digital IC trainer kit

COMPONENTS REQUIRED

DIGITAL IC TRAINER KIT: The equipment mainly used to test and set up digital circuits. Integrated circuits can be fitted in sockets or bread board. There are built in voltage sources and clock signals. In order to feed monopulses manually, a debouncer switch is also provided. A number of select switches are provided to obtain '0' or '1' state voltages as digital inputs. Green and Red LEDs are provided to represent low and high states respectively to visualize the digital outputs.

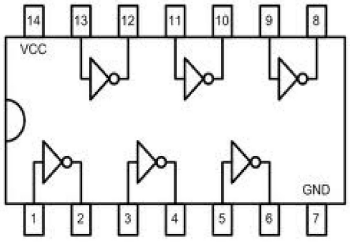
IC PACKAGES: 7404-Hex inverter gates
7400-Quad two input NAND gates
7402-Quad two input NOR gates
7408-Quad two input AND gates
7432-Quad two input OR gates
7486-Quad two input XOR gates

PROCEDURE

1) Test the IC using digital IC tester before conducting the experiment
2) Verify the dual in line package pin out the IC before feeding the input
3) Set up the circuit and observe the output. Enter the input and output stages in truth table corresponding to the input combinations

PIN CONFIGURATION AND TRUTH TABLE

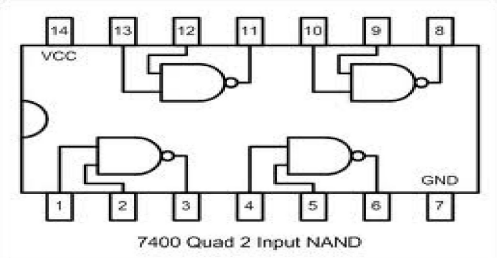
1. 7404-Hex inverter gates



7404 Hex Inverter

NOT	Input	Outputs					
	A	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6
Pins	1,3,5,9,11,13	2	4	6	8	10	12
	0						
	1						

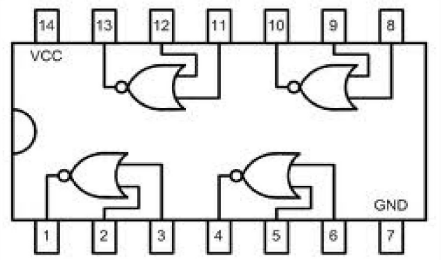
2. 7400-Quad two input NAND gates



7400 Quad 2 Input NAND

NAND	Inputs		Outputs			
	A	B	Y_1	Y_2	Y_3	Y_4
Pins	1,4,9,12	2,5,10,13	3	6	8	11
	0	0				
	0	1				
	1	0				
	1	1				

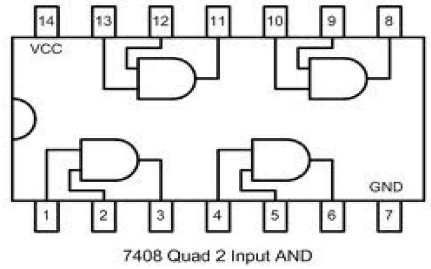
3. 7402-Quad two input NOR gates



7402 Quad 2 Input NOR

NOR	Inputs		Outputs			
	A	B	Y_1	Y_2	Y_3	Y_4
Pins	2,5,8,11	3,6,9,12	1	4	10	13
	0	0				
	0	1				
	1	0				
	1	1				

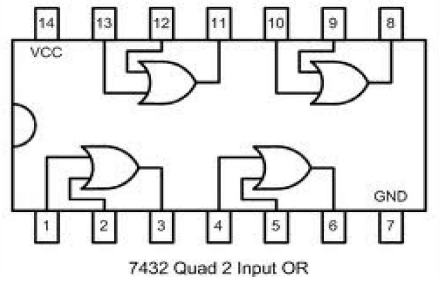
4. 7408-Quad two input AND gates



7408 Quad 2 Input AND

AND	Inputs		Outputs			
	A	B	Y_1	Y_2	Y_3	Y_4
Pins	1,4,9,12	2,5,10,13	3	6	8	11
	0	0				
	0	1				
	1	0				
	1	1				

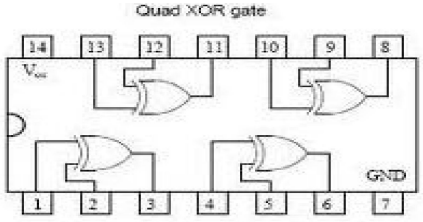
5. 7432-Quad two input OR gates



7432 Quad 2 Input OR

OR	Inputs		Outputs			
	A	B	Y_1	Y_2	Y_3	Y_4
Pins	1,4,9,12	2,5,10,13	3	6	8	11
	0	0				
	0	1				
	1	0				
	1	1				

6. 7486-Quad two input XOR gates



XOR	Inputs		Outputs			
	A	B	Y_1	Y_2	Y_3	Y_4
Pins	1,4,9,12	2,5,10,13	3	6	8	11
	0	0				
	0	1				
	1	0				
	1	1				

RESULT: Familiarized the digital IC trainer kit & logic gate IC packages and verified the truth tables of logic gates.

FAMILIARIZATION OF VERILOG PROGRAMMING

Hardware description language (HDL) is a computer language used to describe the structure and behavior of mostly digital logic circuits. A hardware description language enables a precise, formal description of an electronic circuit that allows for the automated analysis and simulation of an electronic circuit.

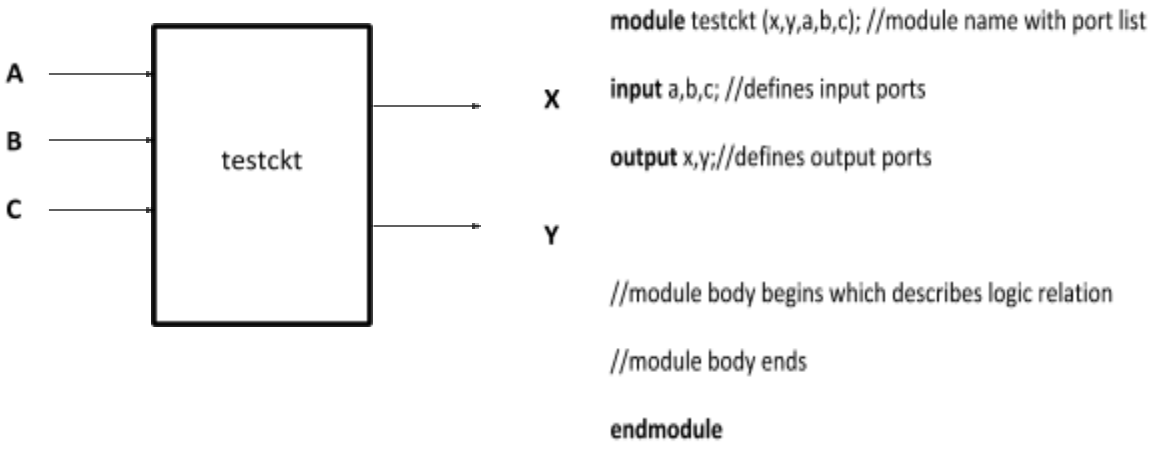
VHDL and Verilog are the two commonly used HDL

Verilog, standardized as IEEE 1364, is a hardware description language (HDL) used to model electronic systems. Syntax similar to the C programming language

Its control flow keywords (if/else, for, while, case, etc.) are equivalent, and its operator precedence is compatible with C

Sequential statements are placed inside a begin/end block and executed in sequential order within the block. However, the blocks themselves are executed concurrently, making Verilog a dataflow language.

Structure of Verilog Code



Circuit representation: 2 fundamental ways

Structural Representation: Use circuit elements such as logic gates or how the elements are connected together is used to write code

Behavioral Representation: circuit is described using logic expressions and programming constructs which represents the behaviors of the circuit.

Verilog is case sensitive.

Name can be anything but should start with letter and can contain only letters, numbers and '_'

The comment starts with // and continues till the end of the line. All Verilog statements end with a semicolon.

Structural Specification

Gate level primitives

and(o/p, i/p1, i/p2)

or(o/p, i/p1, i/p2)

nand(o/p, i/p1, i/p2)

nor(o/p, i/p1, i/p2)

not(o/p, i/p)

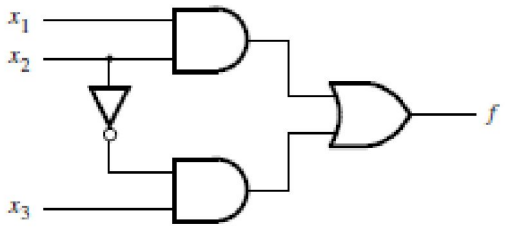


Figure 2.30 A simple logic function.

```
module example1 (x1, x2, x3, f);
   input x1, x2, x3;
   output f;

   and (g, x1, x2);
   not (k, x2);
   and (h, k, x3);
   or (f, g, h);

endmodule
```

```
module example3 (x1, x2, x3, f);
   input x1, x2, x3;
   output f;

   assign f = (x1 & x2) | (~x2 & x3);

endmodule
```

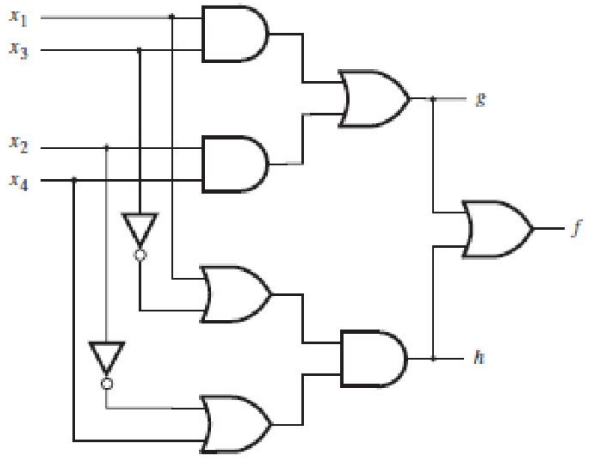
$$f = x_1x_2 + \bar{x}_2x_3$$

```
module example2 (x1, x2, x3, x4, f, g, h);
   input x1, x2, x3, x4;
   output f, g, h;

   and (z1, x1, x3);
   and (z2, x2, x4);
   or (g, z1, z2);
   or (z3, x1, ~x3);
   or (z4, ~x2, x4);
   and (h, z3, z4);
   or (f, g, h);

endmodule
```

Figure 2.32 Verilog code for a four-input circuit.




```
module example4 (x1, x2, x3, x4, f, g, h);
   input x1, x2, x3, x4;
   output f, g, h;

   assign g = (x1 & x3) | (x2 & x4);
   assign h = (x1 | ~x3) & (~x2 | x4);
   assign f = g | h;

endmodule
```

The AND and OR operations are indicated by the "&" and "|" signs, respectively. The assign keyword is used for continuous assignment. The word continuous stems from the use of Verilog for simulation; whenever any signal on the right-hand side changes its state, the value of f will be re-evaluated.

Verilog procedural statement: behavior can be defined with the **if-else** statement

```
if (x2 == 1)
f = x1;
else
```

f = x3;

Verilog syntax requires that procedural statements be contained inside a construct called an **always** block. An **always** block can contain a single statement, or it can contain many statements. The part of the **always** block after the @ symbol, in parentheses, is called the sensitivity list. The statements inside an **always** block are executed by the simulator only when one or more of the signals in the sensitivity list changes value. So that it is not necessary to execute every statement in the code at all times.
If a signal is assigned a value using procedural statements, then Verilog syntax requires that it be declared as a variable; this is accomplished by using the keyword **reg.** value is assigned with a procedural statement, the simulator "registers" this value and it will not change until the **always** block is executed again

```
// Behavioral specification
module example5 (x1, x2, x3, f);
input x1, x2, x3;
output f;
reg f;
always @(x1 or x2 or x3)
if (x2 == 1)
f = x1;
else
f = x3;
endmodule
```

Row number	x_1	x_2	x_3	f
0	0	0	0	1
1	0	0	1	0
2	0	1	0	1
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

Figure 4.1 The function $f(x_1, x_2, x_3) = \sum m(0, 2, 4, 5, 6)$.

```
module example4_21 (x1, x2, x3, f);
  input x1, x2, x3;
  output f;

  assign f = (~x1 & ~x2 & ~x3) | (~x1 & x2 & ~x3) |
             (x1 & ~x2 & ~x3) | (x1 & ~x2 & x3) | (x1 & x2 & ~x3);

endmodule
```

A more convenient approach is to use multibit signals, called vectors, to represent the numbers. Just as a number is represented in a logic circuit as signals on multiple wires, it can be represented in Verilog code as a multibit vector.
input [3:0] W;

This statement defines W to be a four-bit vector. Its individual bits can be referred to by using an index value in square brackets. Thus, the most-significant bit (MSB) is referred to as W[3] and the least-significant bit (LSB) is W[0].

For specifying signals that are neither inputs nor outputs of a module, which are used only for internal connections within the module, Verilog provides the **wire** type. In the adder4 module we need three internal carry signals, which are defined as a three-bit vector in the statement
wire [3:1] C;

W[n−1:0]
parameter n = 4;

Verilog **for** statement.
the **for** statement is a procedural statement that must be placed inside an **always** block,
for loop consists of two statements delineated by **begin** and **end**.
Syntax: for (initial_index; terminal_index; increment) statement;

```
module addern (carryin, X, Y, S, carryout);
parameter n = 32;
input carryin;
input X, Y;
output S;
output carryout;
reg [n 1:0]
[n 1:0]
[n 1:0]
S;
reg carryout;
reg [n:0] C;
integer k;
always @(X or Y or carryin)
begin
C[0] = carryin;
for (k = 0; k < n; k = k+1)
begin
        S[k] = X[k] ∧ Y[k] ∧ C[k];
        C[k+1] = (X[k] & Y[k]) | (X[k] & C[k]) | (Y[k] & C[k]);
end
carryout = C[n];
end
endmodule
```

Verilog provides variables to allow a circuit to be described in terms of its behavior. A variable can be assigned a value in one Verilog statement, and it retains this value until it is overwritten by a subsequent assignment statement. There are two types of variables: **reg** and **integer**. As mentioned above, all signals that are assigned a value using procedural statements must be declared as variables by using the **reg** or **integer** keywords. The scalar carryout and the vectors S and C are examples of the **reg** type. The loop variable k in the same figure illustrates the **integer** type. It serves as a loop index. Such variables are useful for describing a circuit's behavior; they do not usually correspond directly to signals in the resulting circuit.

Verilog Function

A function is declared by the keyword **function** and it comprises a block of statements that ends with the keyword **endfunction**. The function must have at least one input and it returns a single value that is placed where the function is invoked.

```
module mux16to1 (W, S16, f);
input [0:15] W;
input [3:0] S16;
output f;
reg f;
// Function that specifies a 4-to-1 multiplexer
function mux4to1;
input [0:3] X;
input [1:0] S4;
case (S4)
0: mux4to1 = X[0];
1: mux4to1 = X[1];
2: mux4to1 = X[2];
3: mux4to1 = X[3];
endcase
endfunction
always @(W or S16)
case (S16[3:2])
0: f = mux4to1 (W[0:3], S16[1:0]);
1: f = mux4to1 (W[4:7], S16[1:0]);
2: f = mux4to1 (W[8:11], S16[1:0]);
3: f = mux4to1 (W[12:15], S16[1:0]);
endcase
endmodule
```

Using Verilog Constructs for Storage Elements

Asimple way of specifying a storage element is by using the **if-else** statement to describe the desired behavior responding to changes in the levels of data and clock inputs. Consider the **always** block

```
always @(Control or B)
if (Control)
```

A= B;
where A is a variable of **reg** type. This code specifies that the value of A should be made equal to the value of B when Control = 1. But the statement does not indicate an action that should occur when Control = 0. In the absence of an assigned value, the Verilog compiler assumes that the value of A caused by the **if** statement must be maintained until the next time
this **if** statement is evaluated.

module named D_latch, which has the inputs D and Clk and the output Q. The **if** clause defines that the Q output
must take the value of D when Clk = 1. Since no **else** clause is given, a latch will be synthesized to maintain the value of Q when Clk = 0. Therefore, the code describes a gated D latch. The sensitivity list includes Clk and D because both of these signals can cause a change in the value of the Q output.

module D latch (D, Clk, Q);
input D, Clk;
output Q;
reg Q;
always @(D or Clk)
if (Clk)
Q = D;
Endmodule

An **always** construct is used to define a circuit that responds to changes in the signals that appear in the sensitivity list. **always** blocks are sensitive to the levels of signals, it is also possible to specify that a response should take
place only at a particular edge of a signal. The desired edge is specified by using theVerilog keywords **posedge** and **negedge**, which are used to implement edge-triggered circuits.

C =A+ B; is blocking assignment

Verilog also provides a non-blocking assignment, denoted with <=. All non-blocking assignment statements in an **always** block are evaluated using the values that the variables have when the **always** block is entered. Thus, a given variable has the same value for all statements in the block. The meaning of non-blocking is that the result of each assignment is not seen until the end of the **always** block.

Q1 <= D;
Q2 <= Q1;
the variables Q1 and Q2 have some value at the start of evaluating the **always** block, and then they change to a new value concurrently at the end of the **always** block.

```
module example7_4 (D, Clock, Q1, Q2);
   input D, Clock;
   output Q1, Q2;
   reg Q1, Q2;

   always @(posedge Clock)
   begin
      Q1 <= D;
      Q2 <= Q1;
   end

endmodule
```

Figure 7.39 Code for two cascaded flip-flops.

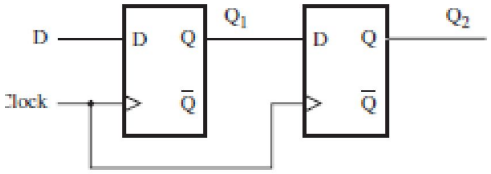


Table 6.2 Verilog operators.

Operator type	Operator symbols	Operation performed	Number of operands
Bitwise	~	1's complement	1
	&	Bitwise AND	2
	|	Bitwise OR	2
	^	Bitwise XOR	2
	~^ or ^~	Bitwise XNOR	2
Logical	!	NOT	1
	&&	AND	2
	||	OR	2
Reduction	&	Reduction AND	1
	~&	Reduction NAND	1
	|	Reduction OR	1
	~|	Reduction NOR	1
	^	Reduction XOR	1
	~^ or ^~	Reduction XNOR	1
Arithmetic	+	Addition	2
	−	Subtraction	2
	−	2's complement	1
	*	Multiplication	2
	/	Division	2
Relational	>	Greater than	2
	<	Less than	2
	>=	Greater than or equal to	2
	<=	Less than or equal to	2
Equality	==	Logical equality	2
	!=	Logical inequality	2
Shift	>>	Right shift	2
	<<	Left shift	2
Concatenation	{,}	Concatenation	Any number
Replication	{{}}	Replication	Any number
Conditional	?:	Conditional	3

Table 6.3 Precedence of Verilog operators.

Operator type	Operator symbols	Precedence
Complement	! ~ –	Highest procedence
Arithmetic	* / + –	
Shift	<< >>	
Relational	< <= > >=	
Equality	== !=	
Reduction	& ~& ^ ~^ | ~|	
Logical	&& ||	
Conditional	?:	Lowest precedence

conditional_expression ? true_expression : false_expression

A= (B < C) ? (D + 5) : (D + 2);

case (expression)
alternative1: statement;
alternative2: statement;
.
.
.
alternativej: statement;
[**default**: statement;]
Endcase

The controlling expression and each alternative are compared bit by bit. When there is one or more matching alternative, the statement(s) associated with the first match (only) is executed. When the specified alternatives do not cover all possible valuations of the controlling expression, the optional **default** clause should be included. Otherwise, the Verilog compiler will synthesize memory elements to deal with the unspecified possibilities;

module mux4to1 (W, S, f);
input [0:3] W;
input [1:0] S;
output f;
reg f;
always @(W or S)

```
case (S)
0: f = W[0];
1: f = W[1];
2: f = W[2];
3: f = W[3];
endcase
endmodule

always @(W or En)
begin
if (En == 0)
Y = 4'b0000;
else
case (W)
0: Y = 4'b1000;
1: Y = 4'b0100;
2: Y = 4'b0010;
3: Y = 4'b0001;
endcase
end
```

II. STUDY OF COMBINATIONAL CIRCUITS

EXPT NO 1: Universal Gates

A. Expression Simplification and Realization

AIM

1. Represent the given expression as truth table, minimize it using K Map (Example: F = A'B' + AB' + A'B)
2. The minimized sum of product (SOP) and product of sum (POS) expressions should be realized using any one universal gates

COMPONENTS REQUIRED

IC Trainer kit, IC 7400, IC 7402

PRINCIPLE

1. The given function can be written as F=A'B' + AB' + A'B + A'B'

i. e. F = B'(A' + A) + A'(B + B') = B' + A'

For the given expression, simplify it using any methods

Write the truth table

Derive the SOP and POS Expression

2. Gates NAND and NOR are known as universal gates, because any logic gates or Boolean expression can be realized by either NAND or NOR gate alone. Each product term in the SOP expression is called minterm and each sum term in the POS expression is called maxterm. SOP expression can be economically realized using NAND gates and POS expression can be economically realized using NOR gates

Realization of SOP expression

Using NAND gates

i) Use NAND gates for each minterm

ii) Use one NAND gate for whole summation

Using NOR gates

i) Invert all the variables in each minterm

ii) Use NOR gates for each minterm having inverted variables

iii) Use NOR gate for whole summation

iv) Use another NOR gate at the output for inverting.

Realization of POS expression

Using NOR gates

i) Use NOR gates for each maxterm

ii) Use one NOR gate for whole multiplication

Using NAND gates

i) Invert all the variables in each maxterm

ii) Use NAND gates for each maxterm having inverted variables

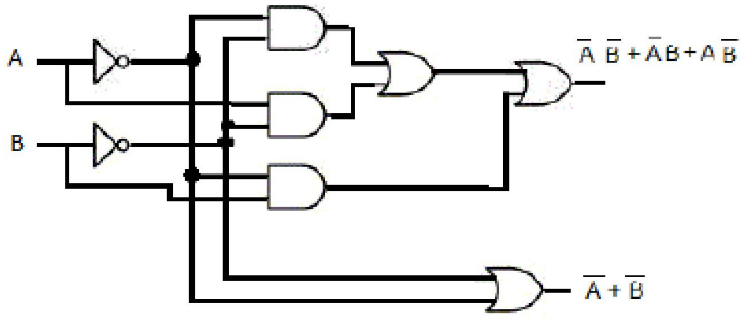
iii) Use NAND gate for whole multiplication

iv) Use another NAND gate at the output for inverting

1) Example f(a, b, c) = $\Sigma_m(1,2,4,5,6)$
Truth Table:

a	b	c	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

2. Minimization



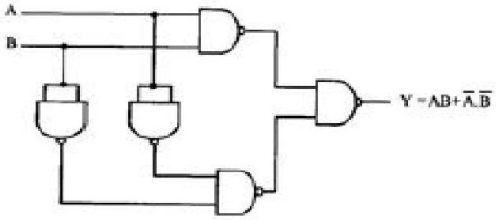
2) Example

3. a)

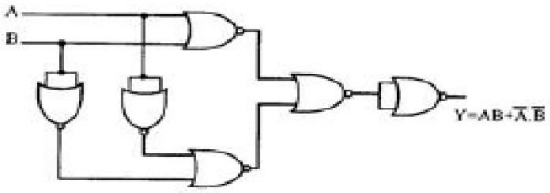
Realisation of SOP expression $Y = AB + \overline{A} \cdot \overline{B}$

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

Truth table



Using NAND gates only



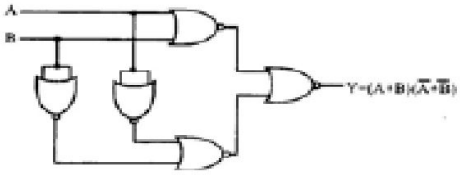
Using NOR gates only

b)

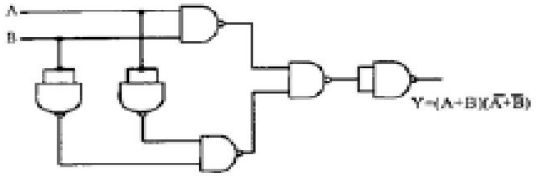
Realisation of POS expression $Y = (A + B)(\overline{A} + \overline{B})$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

Truth table



Using NOR gates only



NAND gates only

PROCEDURE

1) Test all the components and IC packages using multimeter and digital IC tester
2) Set up the circuit one by one and verify their truth table
3) Observe the output corresponding to input combinations and enter it in truth table

RESULT

Sum of products and product of sum expressions were realized using universal gates

EXPT NO 1: Universal Gates

B. For the given expression, design and develop a Verilog code after simplifying it. Simulate and verify the working of the same.

AIM

1. Represent the given expression as truth table, minimize it using K Map
2. The verilog code for either minimized sum of product (SOP) or product of sum (POS) expression has to be realized

Sample expression and code:

For the given SOP expression, write the truth table and minimize it using K Map, then write the code. $f = x_1x_2 + \bar{x}_2x_3$

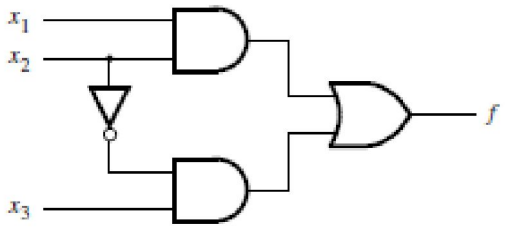


Figure 2.30 A simple logic function.

```
module example1 (x1, x2, x3, f);
   input x1, x2, x3;
   output f;

   and (g, x1, x2);
   not (k, x2);
   and (h, k, x3);
   or (f, g, h);

endmodule
```

or

```
module example3 (x1, x2, x3, f);
   input x1, x2, x3;
   output f;

   assign f = (x1 & x2) | (~x2 & x3);

endmodule
```

EXPT NO 2: Adder Subtractor Circuit

a. Design and verify the truth table of full adder and half subtractor circuit

AIM

1. Derive the truth table of full adder and half subtractor.
2. Based on the truth table derive the expression for full adder and half subtractor
3. Represent the expression as circuit.

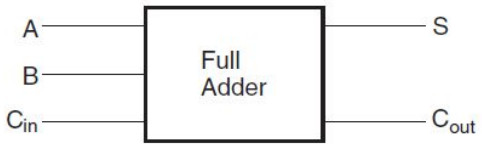
COMPONENTS REQUIRED

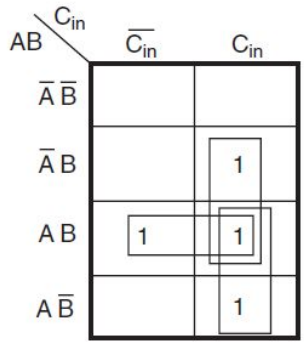
IC Trainer kit, IC 7486, IC 7408, IC 7404, IC 7432

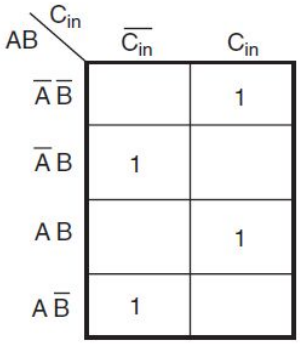
PRINCIPLE

A **full adder** adds binary numbers and accounts for values carried in as well as out. A one-bit full-adder adds three one-bit numbers, often written as A, B, and C_{in}; A and B are the operands, and C_{in} is a bit carried in from the previous less-significant stage. The circuit produces a two-bit output. Output carry and sum typically represented by the signals C_{out} and S, where the sum equals to C_{out} + S.

A	B	C_{in}	SUM (S)	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	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

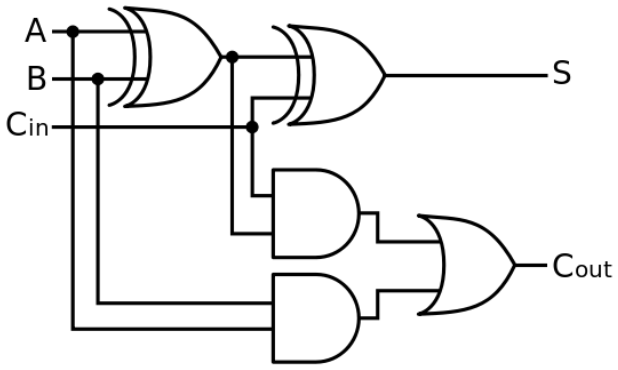




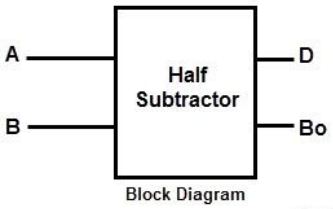


$$C_{out} = \overline{A}.B.C_{in} + A.\overline{B}.C_{in} + A.B.\overline{C}_{in} + A.B.C_{in} \qquad S = \overline{A}.\overline{B}.C_{in} + \overline{A}.B.\overline{C}_{in} + A.\overline{B}.\overline{C}_{in} + A.B.C_{in}$$

Boolean expression of sum can be implemented by using two-input EX-OR gate in which one of the input is Carry in and the other input is the output of another two-input EX-OR gate with A and B as its inputs. On the other hand Carry output can implemented by ORing the outputs of AND gates.



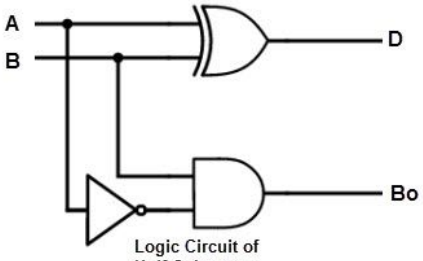
The half subtractor is a combinational circuit which is used to perform subtraction of two bits. It has two inputs, the minuend X and subtrahend Y and two outputs the difference D and borrow out Bout. The borrow out signal is set when the subtractor needs to borrow from the next digit in a multi-digit subtraction. That is, Bout = 1 when X<Y. Since X and Y are bits, Bout = 1 iff X = 0 and Y = 1.



Block Diagram

A	B	D	B_o
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Truth Table



Logic Circuit of Half Subtractor

PROCEDURE

1. Test all the components using digital IC tester
2. Verify the truth table of the circuit by feeding input bit combinations

EXPT NO 2: Adder Subtractor Circuit

b. Design and develop the Verilog code for full adder. Simulate and verify the working of the same.

Verilog Code:

```
module fulladder
(
 input x,
```

```
 input y,
 input cin,

 output A,
 output cout
 );
assign {cout,A} =  cin + y + x;
endmodule
```

EXPT NO 3: Code Converter and Magnitude Comparator Circuit

A. BCD to Excess-3 converter

AIM

1. Represent the BCD to Excess-3 conversion as truth table
2. Simplify the same using K Map and set up the circuit of BCD to Excess-3 converter

COMPONENTS REQUIRED

IC Trainer kit, IC 7486, IC 7408, IC 7404, IC 7432

PRINCIPLE

The Excess-3code for a decimal digit is the binary combination corresponding to the decimal digit plus 3. For example, the excess-3 code for decimal digit 5 is the binary combination for 5+3=8, which is 1000.Each BCD digit four bits with the bits with the bits, from most significant to least significant, labeled A,B,C,D. As the same for excess-3 labeled W, X, Y, Z.

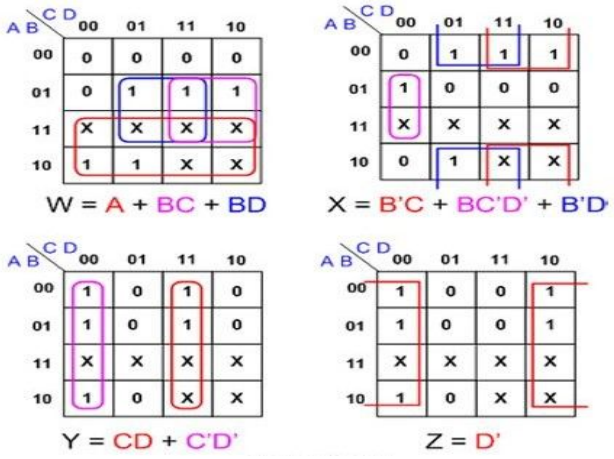
PROCEDURE

1. Test all the components using digital IC tester
2. Verify the truth table of the circuit by feeding input bit combinations

Truth Table - BCD to Excess-3 code

Decimal Digit	Input BCD				Output Excess-3			
	A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	1
9	1	0	0	1	1	1	0	0

Simplification Using K Map



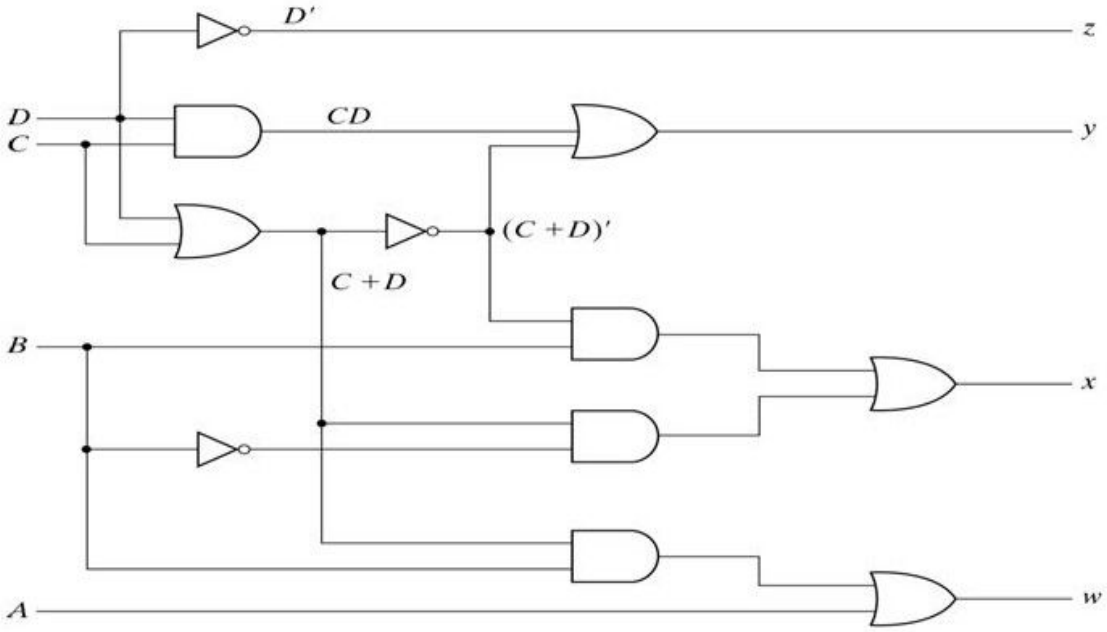
Simplified Expression

$z = D'$; $y = CD + C'D' = CD + (C + D)'$

$x = B'C + B'D + BC'D' = B'(C + D) + B(C + D)'$

$w = A + BC + BD = A + B(C + D)$

Circuit Diagram



RESULT

The circuit of BCD to Excess-3 converter has set up and verified the result.

EXPT NO 3: Code Converter and Magnitude Comparator Circuit

B. Design and develop the Verilog code for a single bit magnitude comparator circuit. Simulate and verify the working of the same.

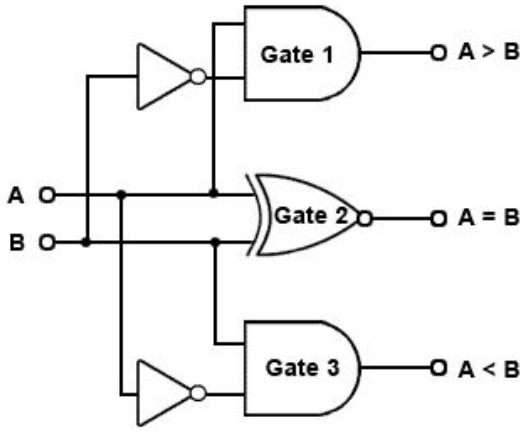
Aim

1. Write the truth table for magnitude comparator
2. Develop the verilog code for the same and verify the working of the same

Truth Table:

A	B	A<B	A=B	A>B
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0

Circuit Diagram:



A>B -> G
A<B -> L
A=B -> E

Verilog Code:

```
module b_comp1 (a, b, L, E,G);
input a, b; output L, E, G;
wire s1, s2;
not X1(s1, a);
not X2 (s2, b);
and X3 (L,s1, b);
and X4 (G,s2, a);
xnor X5 (E, a, b);
```

endmodule

Settings to view the output for One Bit Magnitude Comparator:

1. Set the values of a and b:
    a. Case 1: a=b=0
        i. Run the code. Output E = 1
    b. Case 2: a = 1; b = 0
        i. Run the code. Output G = 1
    c. Case 3: a = 0; b = 1
        i. Run the code. Output L = 1

EXPT NO 4: Data Processing Circuit

MULTIPLEXER

AIM

Given a four variable expression, simplify using Entered Variable Map (EVM) and realize the simplified logic using 8:1 MUX.

COMPONENTS REQUIRED

IC Trainer kit, IC74151, 7404IC

PRINCIPLE

A multiplexer routes one among the many input 2n to the output based on addressing or selection it is called many to one logic circuit. An 8:1 multiplexer has 8 inputs and one output, for addressing it has three selection lines a, b, c pin number(9,10,11), active low enable lines which can be used independently output lines(5,6)

Entered Variable Map (Map Entered Variable (MEV)): It is an alternative to K-maps where a variable is placed as output or one of the input variable is placed inside K-map. This is done separately how it is related with output. This reduces the K-map size by one degree i. e four variable problem that requires 24 = 16 cells in the k-maps will require 2(4-1) = 23 =8 cell in entered variable map. This technique is particularly useful for mapping problems with more input variables. In MEV method there are nine rules as shown below. Entry in MEV map is performed by considering MEV and functional value of the logical expression.

Theory:

Map Entered Variable Method:

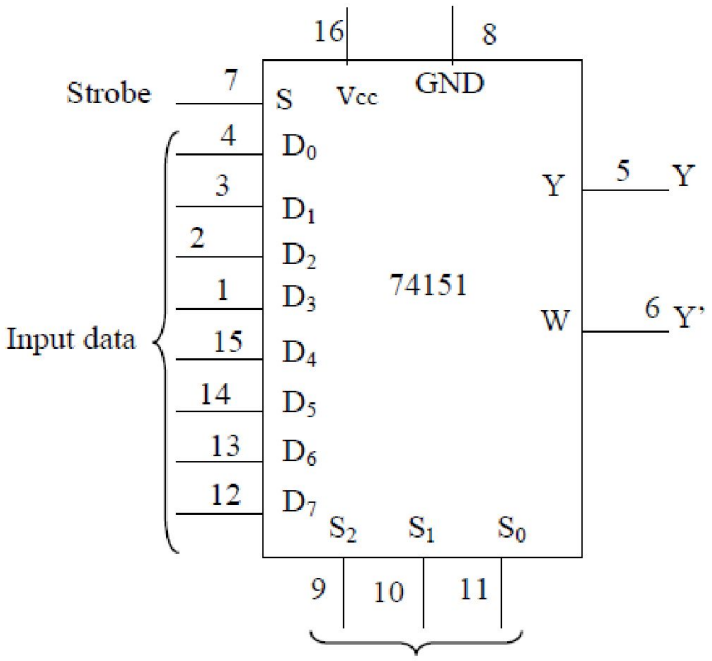
Rules for entering values in a MEV K Map:

Rule No.	MEV f		Entry in MEV Map	Comments
1.	0 1	0 0	0	If function equals 0 for both values of MEV, enter 0 in appropriate cell of MEV Map
2	0 1	1 1	1	If function equals 1 for both values of MEV, enter 1.
3.	0 1	0 1	MEV	If function equals MEV enter MEV
4.	0 1	1 0	------ MEV	If the function is compliment of MEV enter MEV.
5.	0 1	- -	-	If function equals don't care for both values of MEV, enter -
6.	0 1	- 0	0	If f=0 for MEV=0 and f=0 for MEV=1, enter 0.
7.	0 1	0 -	0	If f=0 for MEV=0 and f=- for MEV=1, enter 0.
8.	0 1	- 1	1	If f=-for MEV=0 and f=1 for MEV=1, enter 1.
9.	0 1	1 -	1	If f=1 for MEV=0 and f=- for MEV=-, enter -.

Example: Simplify the function f(a, b, c, d)= Σm(2,3,4,5,13,15) + dc(8,9,10,11)

Decimal		LSB	f	MEV map entry
0}	0	0000	0	0------Do
	1	0001	0	
1}	2	0010	1	1------D1
	3	0011	1	
2}	4	0100	1	1-----D2
	5	0101	1	
3}	6	0110	0	0-----D3
	7	0111	0	
4}	8	1000	X	X-----D4
	9	1001	X	
5}	10	1010	X	X-----D5
	11	1011	X	
6}	12	1100	0	d----D6
	13	1101	1	
7}	14	1110	0	d----D7
	15	1111	1	

IC 74151 Pinout Diagram



Strobe	Select Lines			Output
S	S_2	S_1	S_0	Y
1	X	X	X	0
0	0	0	0	D0
0	0	0	1	D1
0	0	1	0	D2
0	0	1	1	D3
0	1	0	0	D4
0	1	0	1	D5
0	1	1	0	D6
0	1	1	1	D7

PROCEDURE

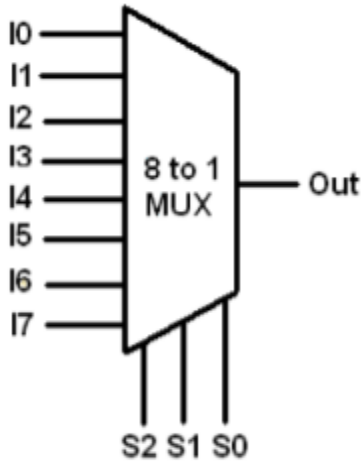
1. Verify all the components and patch chords
2. Make connections as shown in the circuit diagram
3. Give supply to the trainer kit
4. Provide input data verify the truth table and observe the outputs

RESULT AND DISCUSSION

Multiplexer is constructed and verified the truth tables.

Exp No. 4: Data Processing Circuit

B. Design and develop the Verilog code for 8:1 MUX. Simulate and verify the working of the same.



8:1 MUX Block Diagram

Select Lines			Output
S2	**S1**	**S0**	**Out**
0	0	0	I0
0	0	1	I1
0	1	0	I2
0	1	1	I3
1	0	0	I4
1	0	1	I5
1	1	0	I6
1	1	1	I7

VERILOG CODE

```
module mux
(
input [2:0]select,
input [7:0]input,
output reg out
);

always@ (select)
begin
        out = input[select];
end
endmodule

or
module mux1( select, d, q );

input[1:0] select;
input[3:0] d;
output  q;

wire    q;
wire[1:0] select;
wire[3:0] d;

assign q = d[select];

endmodule

or

module mux_tb;

reg[3:0] d;
reg[1:0] select;
wire    q;

integer  i;

mux1 my_mux( select, d, q );

initial
begin
  #1 $monitor("d = %b", d, "  |  select = ", select, "  |  q = ", q );
```

```
  for( i = 0; i <= 15; i = i + 1)
  begin
    d = i;
    select = 0;  #1;
    select = 1;  #1;
    select = 2;  #1;
    select = 3;  #1;
    $display("----------------------------------------");
   end

end
endmodule
```

Settings to view the output for Multiplexer:

1. Set the value of the input line as 0 or 1
2. Set the value of the select line (Any value between 000 to 111)
3. Run the code and see the output. (If input line 2 has 1 as its value and select line is set to 001, the output will be 1)

III. Flip Flops Using Gates and Familiarization of IC's

EXPT NO 5: SR and D flip flop

AIM

A. To Setup SR and D flip flops using gates and verify the truth table also familiarize the flip flop ICs

1. Gated RS flip flop
2. D flip flop

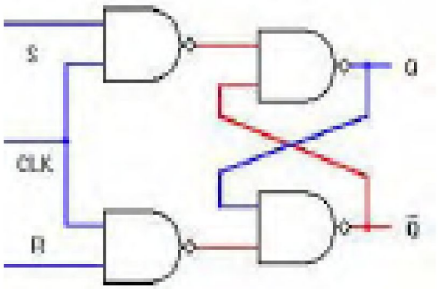
COMPONENTS REQUIRED

Digital IC trainer kit, IC 7400, IC 7410

PRINCIPLE

Flip flops are the basic building blocks in any memory systems since its output will remain in its state until it is forced to change it by some means

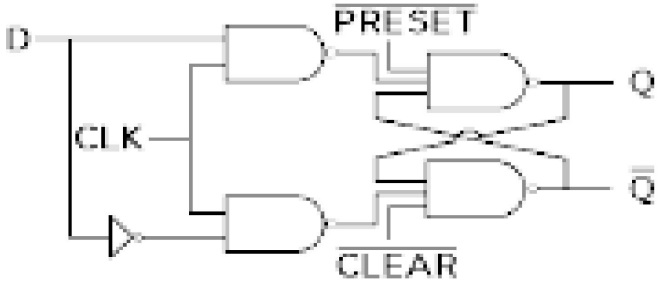
GATED SR FLIP FLOP S and R stands for Set and Reset. There are four input combination possible at the inputs. But S=R=1 is forbidden since the output will be indeterminate.



Input		Output
S	R	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	X

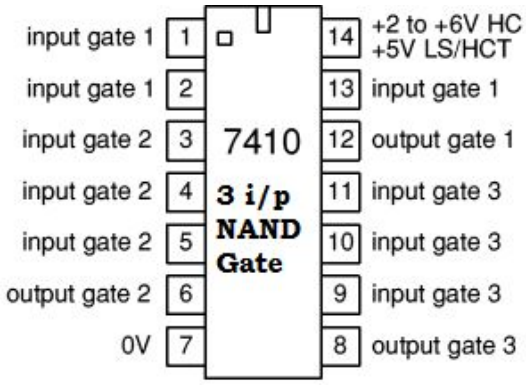
X: Forbidden state

D FF It has only one input called as D input or Data input. The input data is transferred to the output after a clock is applied. D FF can be derived from JK FF by using J input as D input and J is inverted and fed to K input



Input	Output
D	Q_{n+1}
0	0
1	1

7410 Pinout Diagram:



RESULT AND DISCUSSION The flip flops Gated RS flip flop and D flip flop were set up and the output is verified.

B. Design and develop the Verilog code for SR flip flop with positive edge triggering.

SR Flip Flop

```
module srff(q,q1,r,s,clk);
        output q,q1;
        input r,s,clk;
        reg q,q1;
        initial
        begin
                q=1'b0;
                q1=1'b1;
        end
        always @(posedge clk)
          begin
          case({s,r})
                  {1'b0,1'b0}: begin q=q; q1=q1; end
                  {1'b0,1'b1}: begin q=1'b0; q1=1'b1; end
                  {1'b1,1'b0}: begin q=1'b1; q1=1'b0; end
                  {1'b1,1'b1}: begin q=1'bx; q=1'bx; end
        endcase
        end
endmodule
```

Settings to view the output for SR Flip Flop:

1. Set the Clock
2. Set S and R values; Run the code

EXPT NO 6: JK and T flip flop

a. To Setup JK and T flip flops using gates and verify the truth table also familiarize the flip flop ICs

1. JK flip flop
2. T flip flop

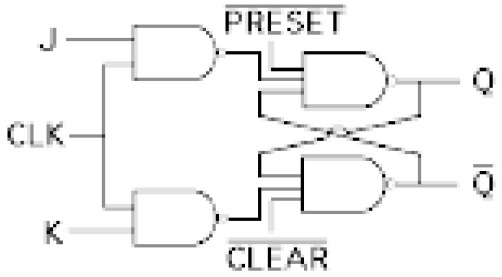
COMPONENTS REQUIRED

Digital IC trainer kit, IC 7400, IC 7410

PRINCIPLE

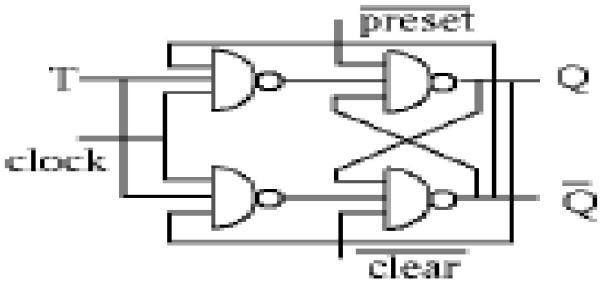
Flip flops are the basic building blocks in any memory systems since its output will remain in its state until it is forced to change it by some means

J K FLIP FLOP The indeterminate output state of SR FF when S=R=1 is avoided by converting it to a JK FF. When flip flop is switched on its output state is uncertain. When an initial state is to be assigned two separate inputs called preset and clear are used. They are active low inputs



Input		Output
J	K	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	$\bar{Q}_n$

T FF T stands for Toggle. The output toggles when a clock pulse is applied. T FF can be derived from JK FF by shorting J and K input



T	Q_{n+1}
0	Q_n
1	$\bar{Q}_n$

RESULT AND DISCUSSION The flip flops Gated JK flip flop and T flip flops were set up and the output is verified.

b. Design and develop the Verilog code for T flip flop with positive edge triggering.

T Flip Flop

```
module mytff(t,q,qb,clk);
input   t,clk;
output q,qb;
reg q,qb;
initial q=0;

always@(posedge clk)
begin
      if (t==1)
      begin
            q=~q;
          end
     else
      begin
     q=q;
      end
      qb=~q;
   end
endmodule
```

Settings to view the output for T Flip Flop:

1. Set the Clock
2. Set T value; Run the code

EXPT NO 7: J-K Master/Slave FF

MASTER SLAVE JK FLIP FLOP

AIM

To realize a JK Master/Slave flip flop using NAND gates and verify its truth table.

COMPONENTS USED

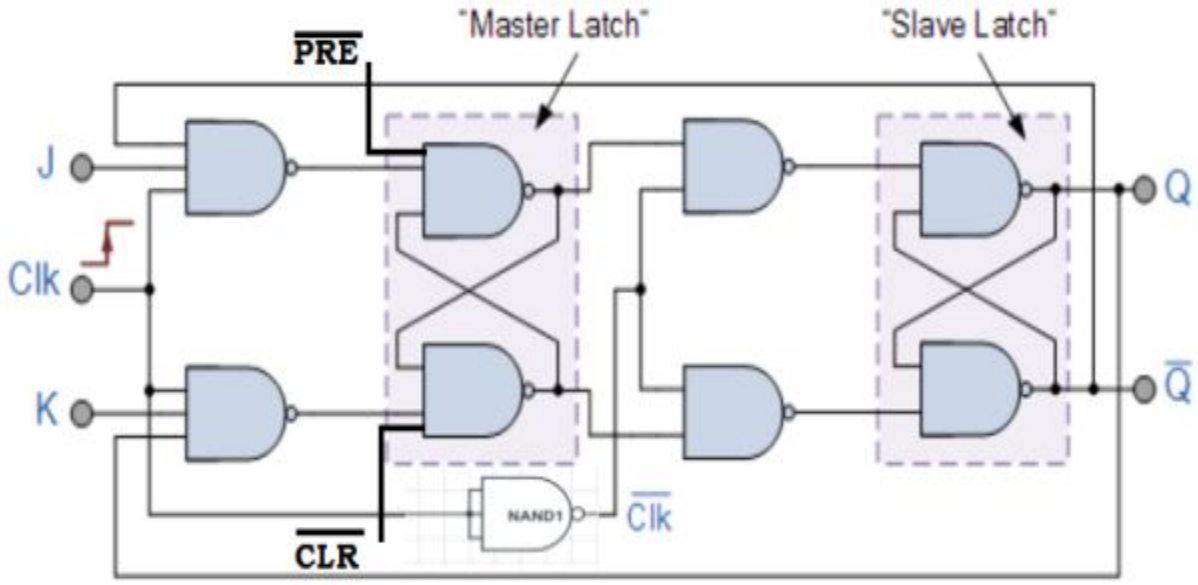
IC 7400, IC 7410, Patch chords, and Trainer kit.

PRINCIPLE

The race around condition of JK FF is rectified in master slave JK FF. Racing is toggling of output more than ones during the positive clock edge. MSJK FF is created by cascading two JK FF. The clock is fed to the first stage (Master) and is inverted and fed to the second stage (slave). This ensure that the master is always followed by the slave and eliminate the chance of racing

CLK	J	K	Q	Q'	Comment
⊓	0	0	Q_0	Q_0'	No Change
⊓	0	1	0	1	Reset

⊓	1	0	1	0	Set
⊓	1	1	Q_0	Q_0	Toggle



PROCEDURE

1. Verify all the components and patch chords.
2. Make the connection as shown in the circuit diagram
3. Give supply to trainer kit
4. Provide input data, verify the truth table and observe outputs.

RESULT

The flip flop MS JK flip flops were set up using NAND gates and verified.

B. Design and develop the Verilog code for JK flip flop with positive edge triggering.

JK Flip Flop

```
module jk(q,q1,j,k,c);
output q,q1;
input j,k,c;
reg q,q1;
initial begin q=1'b0; q1=1'b1; end
always @ (posedge c)
  begin
        case({j,k})
                {1'b0,1'b0}:begin q=q; q1=q1; end
                {1'b0,1'b1}: begin q=1'b0; q1=1'b1; end
                {1'b1,1'b0}:begin q=1'b1; q1=1'b0; end
                {1'b1,1'b1}: begin q=~q; q1=~q1; end
        endcase
```

```
  end
endmodule
```

Settings to view the output for JK Flip Flop:

1. Set the Clock
2. Set J and K values; Run the code

IV. SHIFT REGISTERS AND COUNTERS

EXPT NO 7: SHIFT REGISTERS

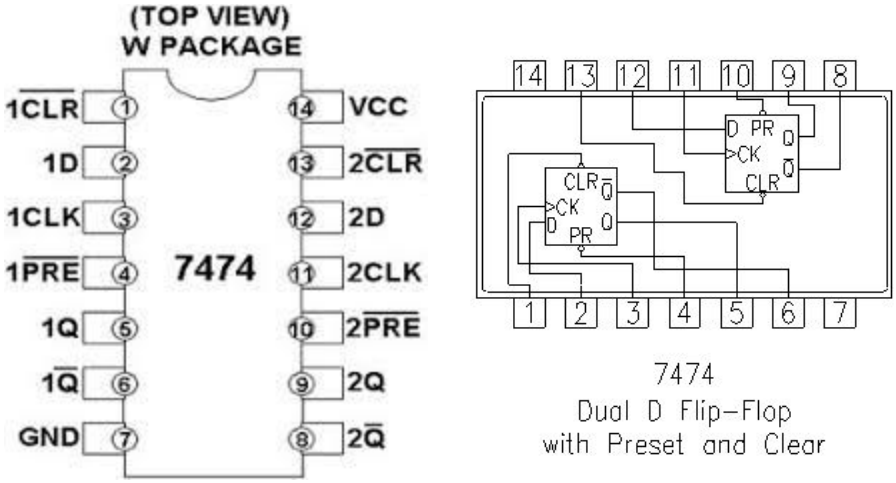
AIM

a. To set up and verify the performance of 4 bit SISO shift register and 2 bit PISO shift register using D flip flop

COMPONENTS REQUIRED

Digital IC trainer kit, IC 7474, IC 7408, IC 7432

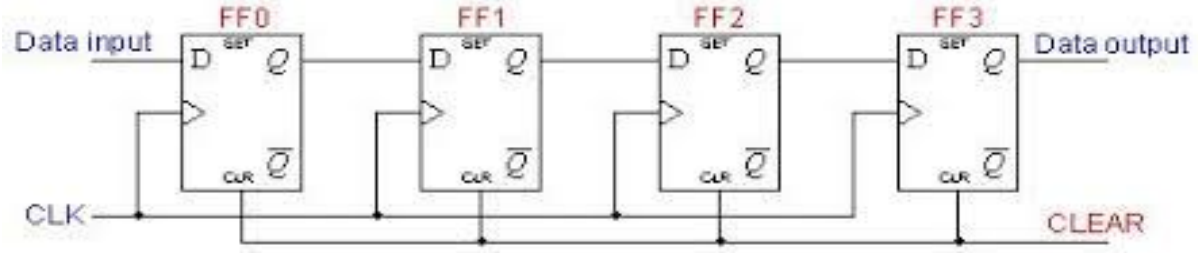
IC 7474 Pinout Diagram



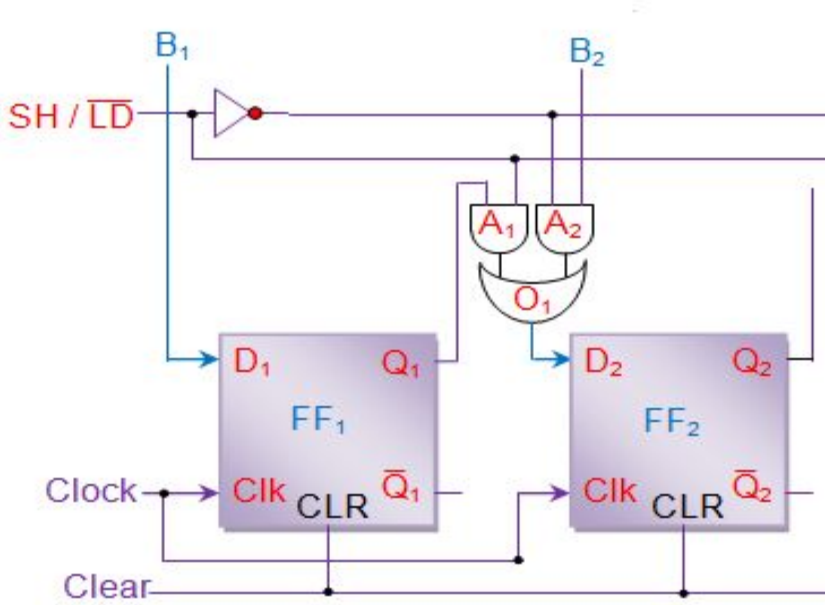
PRINCIPLE

Registers are simply a group of flip flops that can be used to store a binary number. A shift register is nothing but a register which can accept binary number and shift it. The data can be entered in the shift register either in serial or in parallel. The output can be taken either in serial or in parallel. Since there are two ways to shift data in to a register and two ways to shift data out of the register four types of registers can be constructed.

1. 4 Bit Serial In Serial Out (SISO)



2. 2 Bit Parallel In Serial Out (PISO)



PROCEDURE

1. Test all the components using digital IC tester
2. Set up serial input shift register using D FF. Clear all FF using clear pin. Feed 1011 to the serial input starting from LSB using the PRESET and CLEAR pins.

RESULT AND DISCUSSION

The performance of shift registers using D FF are set up and studied

b. Design and develop the Verilog code to represent any give type of Register and verify the working of the same.

```
module shift4mod(R, L, w, Clock, Q);
input [3:0] R;
input L, w, Clock;
output [3:0] Q;
reg [3:0] Q;
always @(posedge Clock)
if (L)
Q <= R;
else
begin
Q[0] <= Q[1];
Q[1] <= Q[2];
Q[2] <= Q[3];
Q[3] <= w;
end
endmodule
```

Settings to view the output:

1. Set the Clock
2. **PIPO**
   a. **Parallel In Parallel Out:** Set L=1; Give 4 bit input to R; Run the Code
3. **PISO**
   a. **Parallel In:** Set L=1; Give 4 bit input to R; Run the Code
   b. **Serial Out:** Set L=0; Run the Code 3 times
4. **SIPO**
   a. **Serial In:**
      Set L=0; Give 4 bit input to w bit by bit; Eg: Data:1010
      i. w=1; Run the Code
      ii. w=0; Run the Code
      iii. w=1; Run the Code
      iv. w=0; Run the Code
   b. **Parallel Out:**
      Set L=1; Run the Code
5. **SISO**
   a. **Serial In:**
      Set L=0; Give 4 bit input to w bit by bit; Eg: Data:1010
      i. w=1; Run the Code
      ii. w=0; Run the Code
      iii. w=1; Run the Code
      iv. w=0; Run the Code
   b. **Serial Out:**
      Set L=0; Run the Code 3 more times

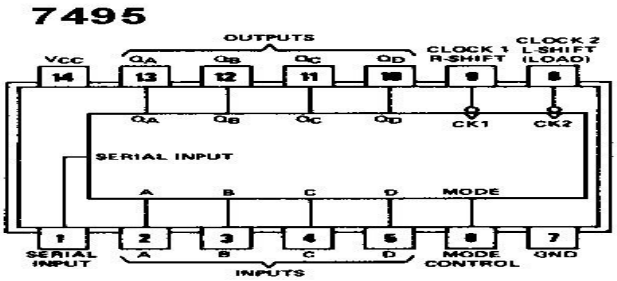
EXPT NO 9: RING COUNTER AND JOHNSON COUNTER

AIM

a. Design and implement ring counter using 4-bit shift register IC and Johnson counter using D flip flop demonstrate its working.

COMPONENTS REQUIRED:

IC 7495 for Ring Counter and IC 7474 for Johnson Counter



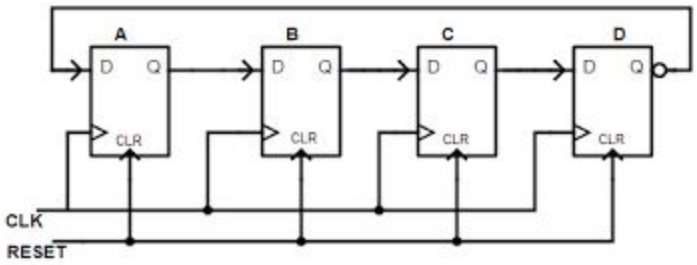
PRINCIPLE

Ring counter and Johnson counters are basically shift registers

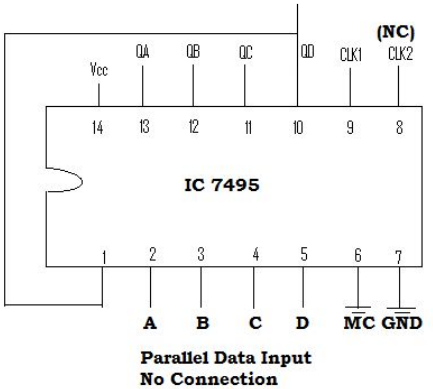
Ring counter

It is made by connecting Q & Q' output of one D FF to D input of next FF respectively. The output of final FF is connected to the input of first FF. To start the counter the first FF is set by using preset facility and the remaining FF are reset input. When the clock arrives the set condition continues to shift around the ring

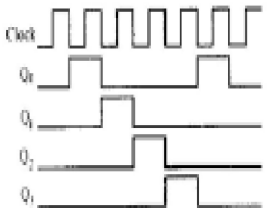
Circuit Diagram of Ring Counter using D FF **for reference**



Circuit Diagram of Ring Counter using Shift Register IC **for implementation**



Waveforms for ring counter

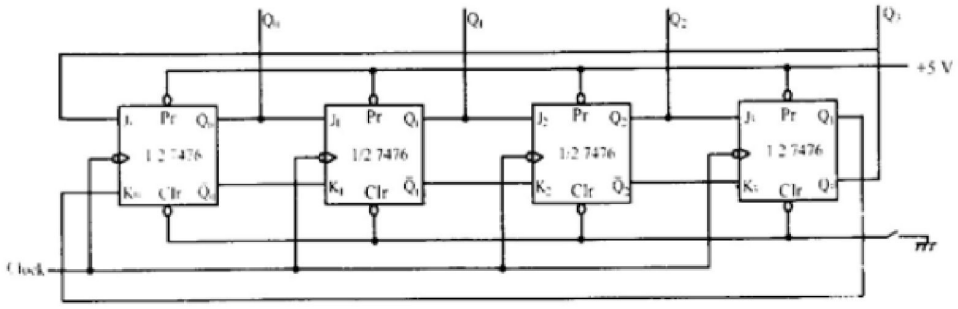


Clk	Q_0 Q_1 Q_2 Q_3
1	1 0 0 0
2	0 1 0 0
3	0 0 1 0
4	0 0 0 1

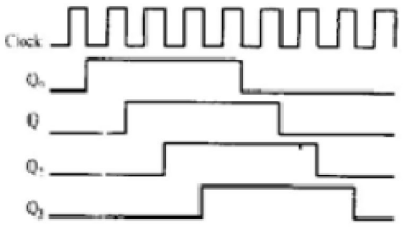
Johnson counter (Twisted ring counter)

The modulus value of a ring counter can be doubled by making a small change in the ring counter circuit. The Q' and Q of the last FFS are connected to the D input of the first FF respectively. This is the Johnson counter.

Johnson counter



Waveforms for Johnson counter



Clk	Q_0 Q_1 Q_2 Q_3
0	0 0 0 0
1	1 0 0 0
2	1 1 0 0
3	1 1 1 0
4	1 1 1 1
5	0 1 1 1
6	0 0 1 1
7	0 0 0 1

PROCEDURE

1. Set up the counter circuit and set clear Q outputs using PRESET and apply mono pulse.
2. Note down the state of the ring counter on the truth table for successive clock 0.
3. Repeat the steps for Johnson counter

RESULT AND DISCUSSION

Four bit ring counter using shift register and the Johnson counter using D FF were set up and its working is verified

EXPT NO 9: Ring and Johnson counter

a. Design and develop the Verilog code for ring counter. Simulate and verify the working of the same.

```
module ring_counter (
    input clock,
    input reset,
output [3:0] q
  );
reg[3:0] a;
always @(posedge clock)
    if (reset)
      a = 4'b0001;
     else
      begin
      a <=  a<<1;
      a[0]<=a[3];
      end
   assign q = a;
endmodule
```

Settings to view the output for Ring Counter:

1. Set the Clock
2. Set reset = 1; Run the code
3. Set reset = 0; Run the code 4 times

EXPT NO 10: SEQUENCE GENERATOR

AIM

a. Design and implement a sequence generator counter using D flip flop IC's and demonstrate its working.

COMPONENTS REQUIRED

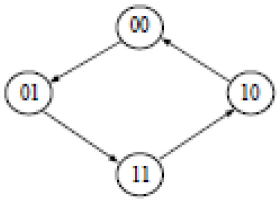
IC 7474, IC 7404, IC 7408 and IC 7432

PRINCIPLE

A Counter is a circuit that produces a set of unique output combinations in relation to the number of applied input pulses. The number of unique outputs of a counter is known as modulus and the counter having N number of unique output is called mod N counter. Normally up or down counter have N=2n where n is the number of flip flops. A counter driven by a clock can be used to count number of clock cycles.

In synchronous counter every flip-flop is triggered in synchronous with the clock i.e. clock inputs are applied simultaneously to all flip-flops. The up-counter counts the numbers from 000 to 111 for mod 8

It can also be used as a sequence generator. Given a sequence, Example (0-1-3-2-0), a state diagram for the counter should be drawn:



Complete the excitation table for the counter and obtain logic expression for the JK flip-flop input functions.

Present state		Next state		Flip-flop input functions			
A	B	A	B	JA	KA	JB	KB
0	0						
0	1						
1	1						
1	0						

Flip-Flop input functions are:

J_A= K_A=

J_B= K_B=

The circuit diagram for the same should be drawn according to the values J_A, K_A, J_B, K_B and the same should be implemented to verify the output.

PROCEDURE

1. Set up the counter circuit and set clear Q outputs using PRESET and apply mono pulse.
2. Give the clock input and verify the output.

RESULT AND DISCUSSION

The sequence generated for the circuit is verified.

b. Design and develop the Verilog code for switched tail counter. Simulate and verify the working of the same.

```
module johnson_counter( out,reset,clk);
input clk,reset;
output [3:0] out;
reg [3:0] q;
always @(posedge clk)
begin
if(reset)
q=4'd0;
else
        begin
                q[3]<=q[2];
                q[2]<=q[1];
                q[1]<=q[0];
                q[0]<=(~q[3]);
        end
end
assign out=q;
endmodule
```

Settings to view the output for Johnson Counter:

1. Set the Clock
2. Set reset = 1; Run the code
3. Run the code 8 times

EXPT NO 10:

ASYNCHRONOUS COUNTER

AIM

a. Design and implement asynchronous counter of the given modulus using decade counter IC and demonstrate its working.

COMPONENTS REQUIRED

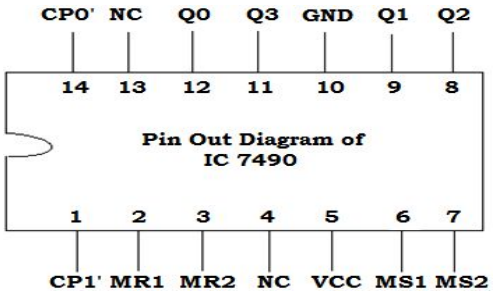
IC 7490, Patch cords and Trainer kit

PRINCIPLE

A Counter is a circuit that produces a set of unique output combinations in relation to the number of applied input pulses. The number of unique outputs of a counter is known as modulus and the counter having N number of unique output is called mod N counter. Normally up or down counter have N=2n where n is the number of flip flops.

In Asynchronous counters the FFs are not given the clock pulse simultaneously. There for the propagation delay increases simultaneously. There for the propagation delay increases with the number of FFs used. FF must be used in toggle mode to count states.

Decade Counter The circuit of the decade up counter is similar to 4 bit ripple up counter but with the aid of logic circuit the count is limited to 0 to 9. As soon as the count reaches 10_D .i.e. 1010, the counting restarts from zero.

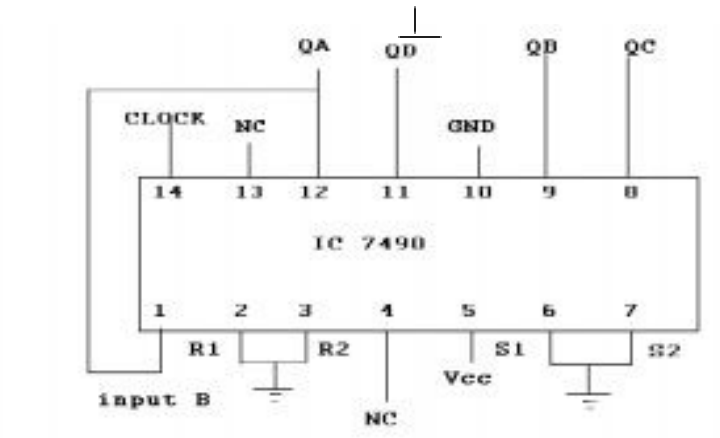


Q0 -> Qa; Q1 -> Qb; Q2 -> Qc; Q3 -> Qd

Circuit Diagram of a decade counter

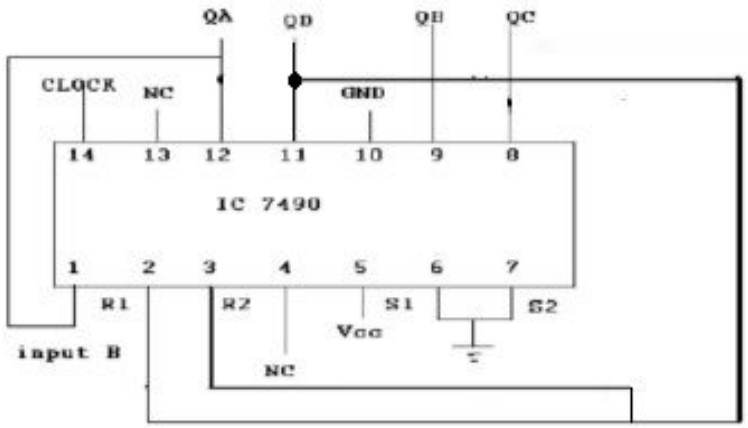
Function Table

CLK	Q_d	Q_c	Q_b	Q_a
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1



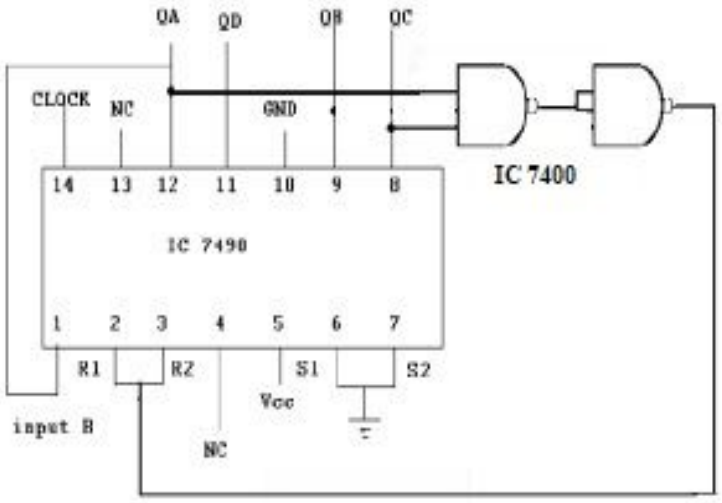
Circuit Diagram of a Mod 8 counter

CLK	Q_d	Q_c	Q_b	Q_a
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0



Circuit Diagram of a Mod 5 counter

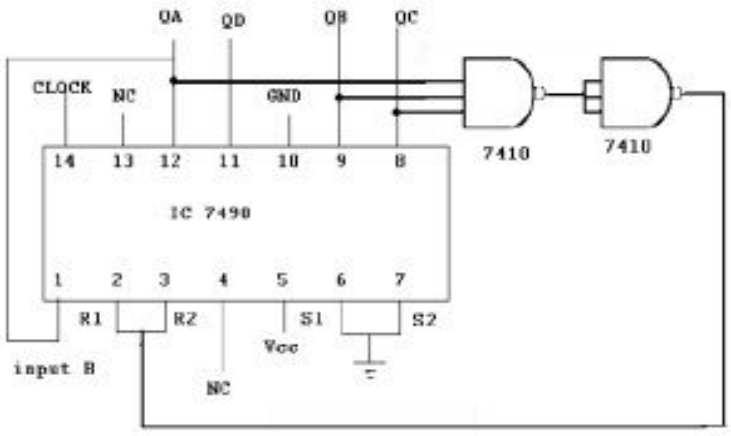
CLK	Q_d	Q_c	Q_b	Q_a
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1



Circuit Diagram of a Mod 7 counter

Function Table

CLK	Q_d	Q_c	Q_b	Q_a
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1



PROCEDURE

1) Set up the circuit using decade counter IC .i.e. IC 7490 for the given modulus
2) Give the proper power supply and ground connection
3) Verify the output

RESULT AND DISCUSSION

The output for the asynchronous counter is observed for the given modulus.

b. Design and develop the Verilog code for 3 bit counter. Simulate and verify the working of the same.

```
module up_counter(input clk, reset, output[3:0] count);
reg [3:0] up;
always @(posedge clk or posedge reset)
begin
if(reset)
 up <= 4'd0;
else
 up <= up + 4'd1;
end
assign count = up;
endmodule
```

Settings to view the output:

4. Set the Clock
5. Set reset = 1; Run the code
6. Set reset = 0; Run the code 8 times

MODEL QUESTIONS

1	Why NAND & NOR gates are called universal gates?
2	Realize the EX – OR gates using minimum number of NAND gates?
3	Give the truth table for EX-NOR and realize using NAND gates
4	Implement the function F (A, B, C) =Σm (0, 1, 4).

5	Design a circuit with four inputs and one output, such that the output goes to '1' whenever two or more of inputs are '1'. For other cases the output remains at '0'.
6	Design half adder cum Subtractor using mode control.
7	Difference between half adder and a full adder
8	Design full adder cum Subtractor using mode control
9	Design a full adder using two half adder and suitable gate
10	Design a two bit digital comparator
11	Which logic is used in single bit digital comparator
12	Design 2-bit odd parity generator and checker
13	Need of parity generator and checker
14	Difference between odd parity and even parity
15	Significance of excess-3 code
16	Significance of Gray code
17	Design 3-bit binary to gray code converter
18	Design 5-bit gray code to binary
19	List four Basic Flip-flop applications?
20	What advantage does a J-K Flip-flop have over an S-R?
21	What is meant by Race around condition?
22	Difference between Decoder and Demultiplexer
23	Difference between encoder and multiplexer
24	Implement the function F (A, B, C) $=\Sigma_m$ (0, 1, 3, 4) using multiplexer.
25	Design an 8:1 multiplexer
26	Design an 1 to 16 demultiplexer
27	Design a full adder using multiplexer
28	Design a bidirectional shift register using mode control
29	Draw the waveforms for each shift register
30	Design a ring counter using D flip flop
31	Design a Johnson counter using D flip flop
32	Design a ring counter/Johnson counter with mode control using JK flip flop
33	Design an up counter using T flip flop
34	Design an up counter using D flip flop
35	Design an asynchronous UP/DOWN counter using mode control
36	Design a Modulo 12 (MOD 12) counter
37	Design an asynchronous decade down counter
38	Design BCD up counter