

CSL 202 DIGITAL LAB

CSL 202	DIGITAL LAB	CATEGORY	L	T	P	CREDIT
		PCC	0	0	3	2

Preamble: This course helps the learners to get familiarized with (i) Digital Logic Design through the implementation of Logic Circuits using ICs of basic logic gates & flip-flops and (ii) Hardware Description Language based Digital Design. This course helps the learners to design and implement hardware systems in areas such as games, music, digital filters, wireless communications and graphical displays.

Prerequisite: Topics covered under the course Logic System Design (CST 203)

Course Outcomes: After the completion of the course the student will be able to

CO 1	Design and implement combinational logic circuits using Logic Gates (Cognitive Knowledge Level: Apply)
CO 2	Design and implement sequential logic circuits using Integrated Circuits (Cognitive Knowledge Level: Apply)
CO 3	Simulate functioning of digital circuits using programs written in a Hardware Description Language (Cognitive Knowledge Level: Apply)
CO 4	Function effectively as an individual and in a team to accomplish a given task of designing and implementing digital circuits (Cognitive Knowledge Level: Apply)

Mapping of course outcomes with program outcomes

	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO10	PO11	PO12
CO 1	✓	✓	✓	✓				✓				✓
CO 2	✓	✓	✓	✓				✓				✓
CO 3	✓	✓	✓	✓	✓			✓				✓
CO 4	✓	✓	✓	✓				✓	✓			✓

Assessment Pattern

Bloom's Category	Continuous Assessment Test (Internal Exam) (Percentage)	End Semester Examination (Percentage)
Remember	20	20
Understand	20	20
Apply	60	60
Analyse		
Evaluate		
Create		

Mark Distribution

Total Marks	CIE Marks	ESE Marks	ESE Duration
150	75	75	3 hours

Continuous Internal Evaluation Pattern:

Attendance : 15 marks

Continuous Evaluation in Lab : 30 marks

Continuous Assessment Test : 15 marks

Viva-voce : 15 marks

Internal Examination Pattern: The marks will be distributed as Design/Algorithm 30 marks, Implementation/Program 20 marks, Output 20 marks and Viva 30 marks. Total 100 marks which will be converted out of 15 while calculating Internal Evaluation marks.

End Semester Examination Pattern: The marks will be distributed as Design/Algorithm 30 marks, Implementation/Program 20 marks, Output 20 marks and Viva 30 marks. Total 100 marks will be converted out of 75 for End Semester Examination.

Fair Lab Record:

All Students attending the Digital Lab should have a Fair Record. The fair record should be produced in the University Lab Examination. Every experiment conducted in the lab should be noted in the fair record. For every experiment in the fair record, the right hand page should contain Experiment Heading, Experiment Number, Date of Experiment, and Aim of Experiment. The left hand page should contain components used, circuit design or a print out of the code used for the experiment and sample output obtained.

SYLLABUS

Conduct a minimum of **8** experiments from **Part A** and a minimum of **4** experiments from **Part B**. The starred experiments in Part A are mandatory. The lab work should be conducted in groups (maximum group size being 4). The performance of a student in the group should be assessed based on teamwork, integrity and cooperation.

Part A (Any 8 Experiments)

- A 2 hour session should be spent to make the students comfortable with the use of trainer kit/breadboard and ICs.
 - The following experiments can be conducted on breadboard or trainer kits.
 - Out of the 15 experiments listed below, a minimum of 8 experiments should be completed by a student, including the mandatory experiments (5).
1. Realization of functions using basic and universal gates (SOP and POS forms).
 2. Design and realization of half adder, full adder, half subtractor and full subtractor using:
a) basic gates (b) universal gates. *
 3. Code converters: Design and implement BCD to Excess 3 and Binary to Gray code converters.
 4. Design and implement 4 bit adder/subtractor circuit and BCD adder using IC7483.
 5. Implementation of Flip Flops: SR, D, T, JK and Master Slave JK Flip Flops using basic gates.*
 6. Asynchronous Counter: Design and implement 3 bit up/down counter.
 7. Asynchronous Counter: Realization of Mod N counters (At least one up counter and one down counter to be implemented). *
 8. Synchronous Counter: Realization of 4-bit up/down counter.
 9. Synchronous Counter: Realization of Mod-N counters and sequence generators. (At least one mod N counter and one sequence generator to be implemented) *
 10. Realization of Shift Register (Serial input left/right shift register), Ring counter and Johnson Counter using flipflops. *
 11. Realization of counters using IC's (7490, 7492, 7493).
 12. Design and implement BCD to Seven Segment Decoder.
 13. Realization of Multiplexers and De-multiplexers using gates.
 14. Realization of combinational circuits using MUX & DEMUX ICs (74150, 74154).
 15. To design and set up a 2-bit magnitude comparator using basic gates.

PART B (Any 4 Experiments)

- The following experiments aim at training the students in digital circuit design with *Verilog*. The experiments will lay a foundation for digital design with Hardware Description Languages.
- A 3 hour introductory session shall be spent to make the students aware of the fundamentals of development using Verilog
- Out of the 8 experiments listed below, a minimum of 4 experiments should be completed by a student

Experiment 1. Realization of Logic Gates and Familiarization of Verilog

- (a) Familiarization of the basic syntax of Verilog
- (b) Development of Verilog modules for basic gates and to verify truth tables.
- (c) Design and simulate the HDL code to realize three and four variable Boolean functions

Experiment 2: Half adder and full adder

- (a) Development of Verilog modules for half adder in 3 modeling styles (dataflow/structural/behavioural).
- (b) Development of Verilog modules for full adder in structural modeling using half adder.

Experiment 3: Design of code converters

Design and simulate the HDL code for

- (a) 4- bit binary to gray code converter
- (b) 4- bit gray to binary code converter

Experiment 4: Mux and Demux in Verilog

- (a) Development of Verilog modules for a 4x1 MUX.
- (b) Development of Verilog modules for a 1x4 DEMUX.

Experiment 5: Adder/Subtractor

- (a) Write the Verilog modules for a 4-bit adder/subtractor
- (b) Development of Verilog modules for a BCD adder

Experiment 6: Magnitude Comparator

Development of Verilog modules for a 4 bit magnitude comparator

Experiment 7: Flipflops and shiftregisters

- (a) Development of Verilog modules for SR, JK, T and D flip flops.
- (b) Development of Verilog modules for a Johnson/Ring counter

Experiment 8: Counters

- (a) Development of Verilog modules for an asynchronous decade counter.
- (b) Development of Verilog modules for a 3 bit synchronous up-down counter.

Practice Questions

PART A

1. Design a two bit parallel adder using gates and implement it using ICs of basic gates
2. A combinatorial circuit has 4 inputs and one output. The output is equal to 1 when (a) all inputs are 1, (b) none of the inputs are 1, (c) an odd number of inputs are equal to 1. Obtain the truth table and output function for this circuit and implement the same.
3. Design and implement a parallel subtractor.
4. Design and implement a digital circuit that converts Gray code to Binary.
5. Design a combinational logic circuit that will output the 1's compliment of a 4-bit input number.
6. Implement and test the logic function $f(A, B, C) = \sum m(0,1,3,6)$ using an 8:1 MUX IC
7. Design a circuit that will work as a ring counter or a Johnson counter based on a mode bit, M.
8. Design a 4-bit synchronous down counter.
9. Design a Counter to generate the binary sequence 0,1,3,7,6,4
10. Design an asynchronous mod 10 down counter
11. Design and implement a synchronous counter using JK flip flop ICs to generate the sequence: 0 - 1 - 3 - 5 - 7 - 0.

PART B

1. Develop Verilog modules for a full subtractor in structural modeling using half subtractors.
2. Design a 4 bit parallel adder using Verilog.
3. Develop Verilog modules for a 4 bit synchronous down counter.
4. Write Verilog code for implementing a 8:1 multiplexer.
5. Develop Verilog modules for a circuit that converts Excess 3 code to binary.
6. Write the Verilog code for a JK Flip flop, and its test-bench. Use all possible combinations of inputs to test its working
7. Write the hardware description in Verilog of a 8-bit register with shift left and shift right modes of operations and test its functioning.
8. Write the hardware description in Verilog of a mod-N ($N > 9$) counter and test it.

LIST OF EXPERIMENTS

CYCLE 1

1. Familiarization of Logic Gates
2. Realization of functions using basic and universal gates (SOP and POS forms).
3. Design and realization of half adder, full adder, half subtractor and full subtractor using: a) basic gates (b) universal gates.
4. Implementation of Flip Flops: SR, D, T, JK and Master Slave JK Flip Flops using basic gates
5. Asynchronous Counter: Realization of Mod N counters
6. Synchronous Counter: Realization of Mod-N counters and sequence generators
7. Realization of Shift Register (Serial input left/right shift register), Ring counter and Johnson Counter using flip flops.
8. Realization of Multiplexers and De-multiplexers using gates.
9. Realization of combinational circuits using MUX & DEMUX ICs (74150, 74154).

EXPT NO: 1_

DATE: __/__/__

FAMILIARIZATION OF LOGIC GATES**OBJECTIVE:**

To familiarize logic gates.

HARDWARE REQUIRED:

SLNo.	Components/Equipments	Specification	Quantity
1.	Digital Trainer Kit		
2.	IC		

INTRODUCTION:

A **logic gate** is an idealized or physical device implementing a Boolean function, that is, it performs a logical operation on one or more logic inputs and produces a single logic. There are seven basic logic gates: AND, OR, XOR, NOT, NAND, NOR, and XNOR.

The **AND gate** is so named because, if 0 is called "false" and 1 is called "true," the gate acts in the same way as the logical "and" operator. The output is "true" when both inputs are "true." Otherwise, the output is "false."

The **OR gate** gets its name from the fact that it behaves after the fashion of the logical inclusive "or." The output is "true" if either or both of the inputs are "true." If both inputs are "false," then the output is "false."

A **NOT gate** sometimes called a logical inverter to differentiate it from other types of electronic inverter devices, has only one input. It reverses the logic state

The **NAND gate** operates as an AND gate followed by a NOT gate. It acts in the manner of the logical operation "and" followed by negation. The output is "false" if both inputs are "true." Otherwise, the output is "true."

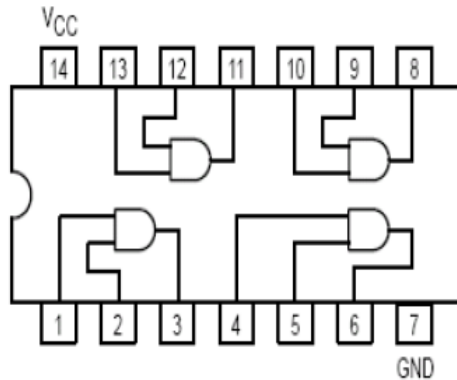
The **NOR gate** is a combination OR gate followed by an inverter. Its output is "true" if both inputs are "false." Otherwise, the output is "false."

The **XOR(exclusive-OR) gate** acts in the same way as the logical "either/or." The output is "true" if either, but not both, of the inputs are "true." The output is "false" if both inputs are "false" or if both inputs are "true." Another way of looking at this circuit is to observe that the output is 1 if the inputs are different, but 0 if the inputs are the same.

The **XNOR** (exclusive-NOR) **gate** is a combination XOR gate followed by an inverter. Its output is "true" if the inputs are the same, and "false" if the inputs are different

CIRCUIT DIAGRAM AND OBSERVATIONS:

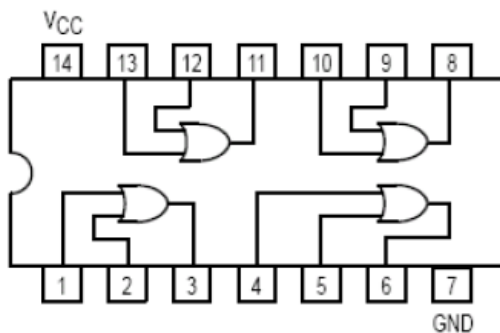
1) AND Gate-7408LS



Truth table

A	B	O/P
0	0	0
0	1	0
1	0	0
1	1	1

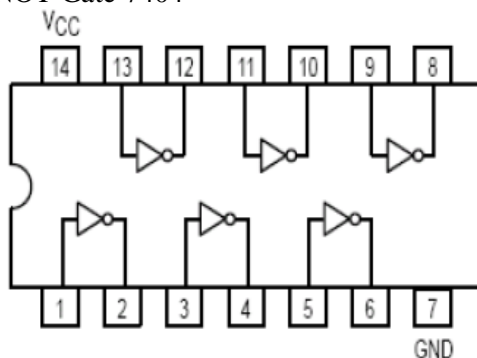
2) OR Gate-7432LS



Truth table

A	B	O/P
0	0	0
0	1	1
1	0	1
1	1	1

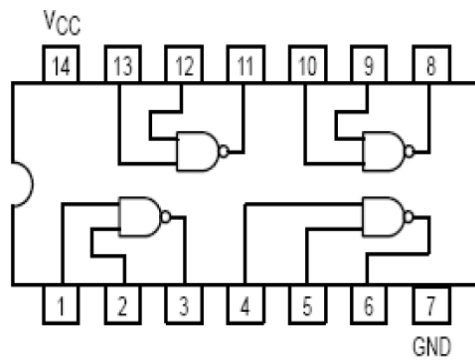
3) NOT Gate-7404



Truth table

A	O/P
0	1
1	0

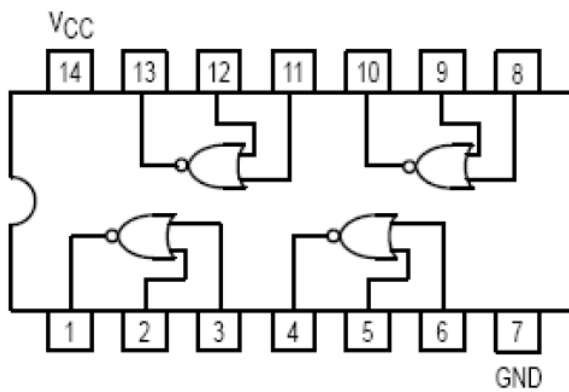
4) NAND Gate-7400LS



Truth table

A	B	O/P
0	0	1
0	1	1
1	0	1
1	1	0

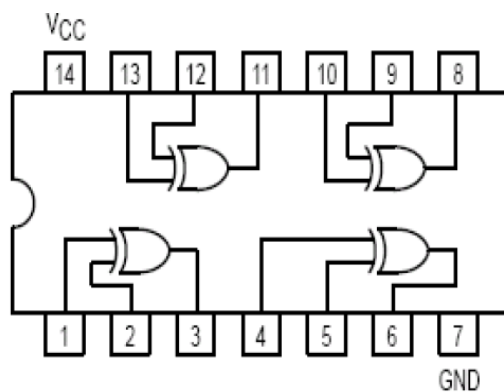
5) NOR Gate-7402LS



Truth table

A	B	O/P
0	0	1
0	1	0
1	0	0
1	1	0

6) EX-OR Gate-7486LS

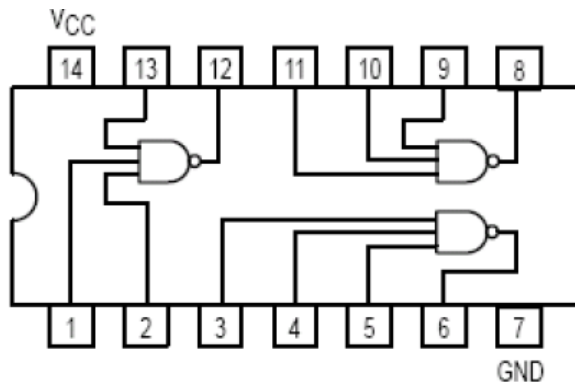


Truth table

A	B	O/P
0	0	0
0	1	1
1	0	1
1	1	0

7) 3 INPUT NAND Gate-7410LS

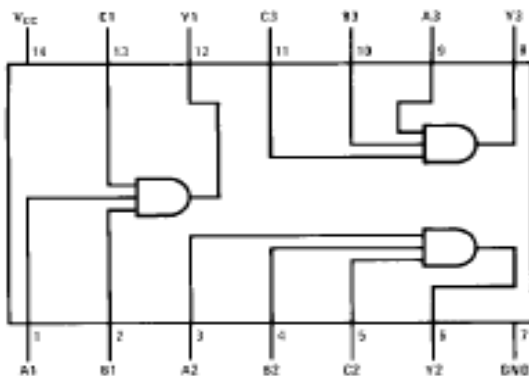
Truth table



A	B	C	O/P
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

8) 3 INPUT AND Gate-7411LS

Truth table



A	B	C	O/P
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

EXPERIMENT:**Procedure**

- 1) Place the IC on IC Trainer Kit.
- 2) Connect V_{CC} and ground to respective pins of IC Trainer Kit.
- 3) Connect the inputs to the input switches provided in the IC Trainer Kit.
- 4) Connect the outputs to the switches of output LEDs
- 5) Apply various combinations of inputs according to the truth table and observe condition of LEDs
- 6) Disconnect output from the LEDs and note down the corresponding multimeter voltage readings for various combinations of inputs.

CONCLUSION**REVIEW QUESTIONS:**

- 1) Which are the universal logic gates? Why they are called so?
- 2) What is a logic gate?
- 3) Implement AND and OR gate using NAND gate
- 4) Implement OR using NOR gate
- 5) What is the difference between 54 series and 74 series?

EXPT NO: 2

DATE: __/__/__

IMPLEMENTATION OF GIVEN BOOLEAN EXPRESSION USING LOGIC GATES IN SOP AND POS FORMS

OBJECTIVE:

To design and implement the given Boolean function using logic gates in sum of product and product of sum forms.

HARDWARE REQUIRED:

SLNo.	Components/Equipments	Specification	Quantity
1.	Digital Trainer Kit		1
2.	IC	7400	1
		7402	1
		7404	1
		7408LS	1

INTRODUCTION:

Boolean algebra provides a concise way to express the operation of a logic circuit formed by a combination of logic gates so that the output can be determined for various combinations of input values.

All Boolean expressions regardless of their form can be converted into either of two standard forms: the sum of products form or the product of sum form. Standardization makes the evaluation, simplification and implementation of Boolean expressions much more systematic and easier.

THE SUM OF PRODUCTS (SOP) FORM

When two or more product terms are summed by Boolean addition, the resulting expression is a sum of products (sop).

Example:

$$\overline{A}B + ABC$$

$$\overline{A}\overline{B}C + CD + A\overline{C}$$

$$AB + ABC + AC$$

Implementation of SOP Expression

Implementing SOP expression simply requires ORing the outputs of two or more AND gates. A product term is produced by an AND operation and the sum (addition) of two or more product terms is produced by an OR operation. Therefore, an SOP expression can be implemented by AND-OR logic in which the outputs of a number (equal to the number of product terms in the expression) of AND gates connect to the inputs of an OR gate. The output X of the OR gate equals the SOP expression. A SOP expression can always be implemented with one OR gate and two or more AND gates.

The standard POS form:

Any logic expression can be changed in to SOP form by applying Boolean algebra techniques.

The expression $A(B+CD)$ can be converted to SOP form by applying the distributive law:

Example:

$$A(B+CD) = AB + ACD$$

A standard SOP expression is one in which all the variables in the domain appear in each product term in the expression. Any nonstandard SOP expression can be converted to standard form using Boolean algebra.

THE PRODUCT-OF-SUM (POS) FORM

When two or more sum terms are multiplied, the resulting expression is a product-of-sums (POS).

Examples:

$$(A + \bar{B} + C)(C + D)$$

$$(A + B)(\bar{A} + B + C)(C + \bar{D})$$

Implementation of POS expression:

Implementing a POS expression simply requires ANDing the outputs of two or more OR gates. A sum term is produced by an OR operation and the product of two or more sum term is produced by an AND operation. Therefore a POS expression can be implemented by a logic in which the outputs of number of OR gates connect to the input of an AND gate.

The standard POS form:

A standard POS expression is one in which all the variables in the domain appear in each sum term in the expression.

Example:

$$(A+B+C+D)(A+B+C+D)$$

Any nonstandard POS expression can be converted to the standard form using Boolean algebra.

CIRCUIT DIAGRAM AND OBSERVATIONS:

Questions

- 1) $F = \sum m(0,1,2,3,4,5,6)$
- 2) $F = \sum m(1,2,3,6,8,12,14,15)$
- 3) $F = \prod M(0,3,5)$
- 4) $F = \prod M(1,2,3,6,8,12,14,15)$

EXPERIMENT:**Procedure**

- 7) Place the IC on IC Trainer Kit.
- 8) Connect V_{CC} and ground to respective pins of IC Trainer Kit .
- 9) Set up the circuit obtained from the expression
- 10) Connect the inputs to the input switches provided in the IC Trainer Kit.
- 11) Connect the outputs to the switches of output LEDs
- 12) Apply various combinations of inputs according to the truth table and observe condition of LEDs
- 13) Disconnect output from the LEDs and note down the corresponding multimeter voltage readings for various combinations of inputs

CONCLUSION**REVIEW QUESTIONS:**

- 1) Map the given expression

$$F = \overline{A}\overline{B}C + \overline{A}B\overline{C} + \overline{A}BC + A\overline{B}\overline{C} + ABC$$

- 2) Map the given expression

$$F = (A + B + C)(\overline{A} + B + \overline{C})(\overline{A} + \overline{B} + \overline{C})$$

- 3) Minimize the following functions using k-map

$$F(X_1 X_2 X_3 X_4) = \sum m(1, 2, 3, 5, 7, 8, 9) + d(12, 14)$$

- 4) Convert the expression into standard form

$$ABCD + \overline{B}C + AD + \overline{A}C$$

- 5) Explain the general procedure to simplify the Boolean expression

EXPT NO: 3

DATE: __/__/__

DESIGN AND IMPLEMENTATION OF ARITHMETIC CIRCUITS**OBJECTIVE:**

To design and implement the Arithmetic Circuits

HARDWARE REQUIRED:

SLNo.	Components/Equipments	Specification	Quantity
1.	Digital Trainer Kit		
2.	IC		

INTRODUCTION:**Half adder**

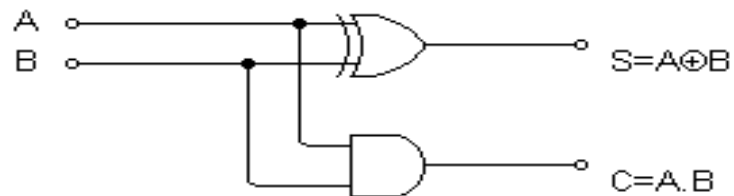
With the help of half adder, we can design circuits that are capable of performing simple addition with the help of logic gates. The truth table of half adder is shown below. Sum, s is the normal output and carry, c is the carry-out.

Full adder

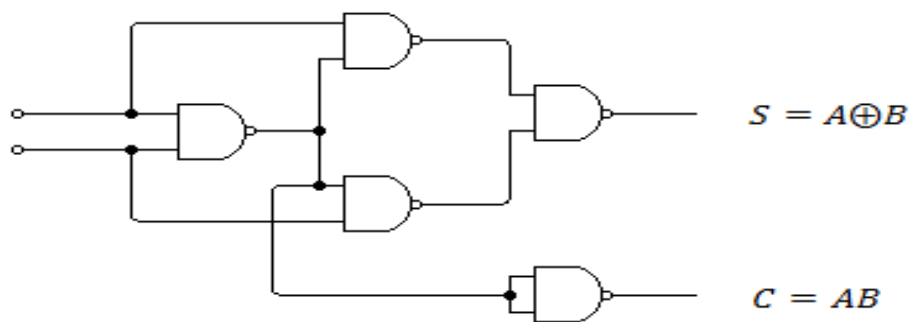
This type of adder is a little more difficult to implement than a half-adder. The main difference between a half-adder and a full-adder is that the full-adder has three inputs and two outputs. The first two inputs are A and B and the third input is an input carry designated as CIN. When a full adder logic is designed we will be able to string eight of them together to create a byte-wide adder and cascade the carry bit from one adder to the next. The output carry is designated as C and the normal output is designated as S.

CIRCUIT DIAGRAM:

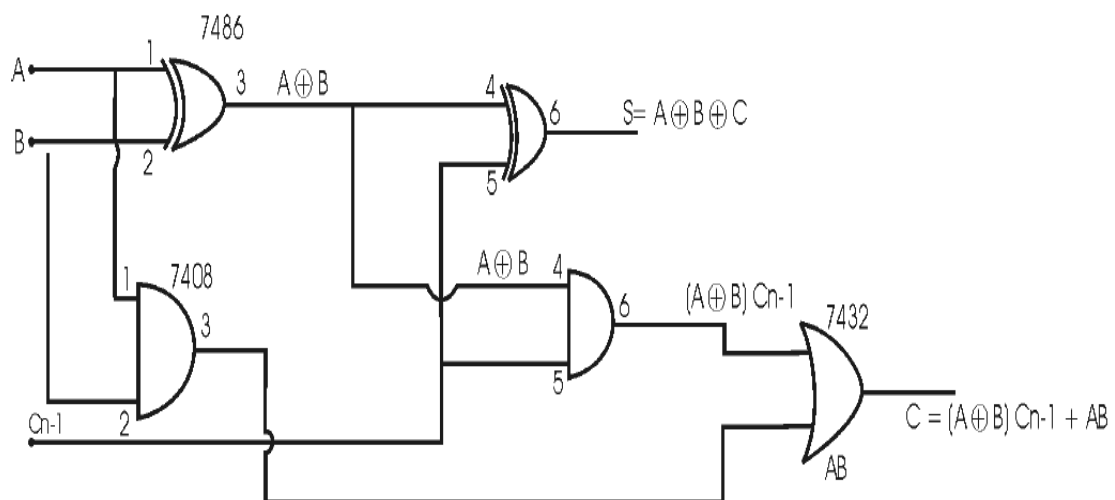
1. Half-Adder using basic gates



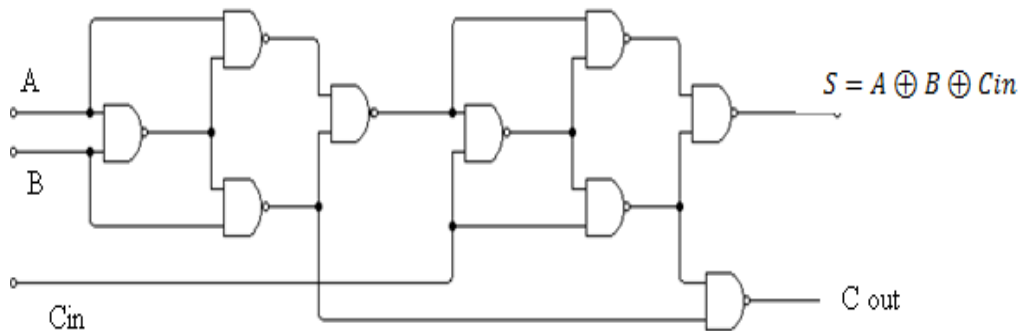
2. Half-Adder using NAND gates only



3. Full-Adder using basic gates

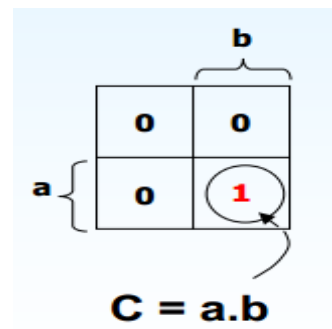
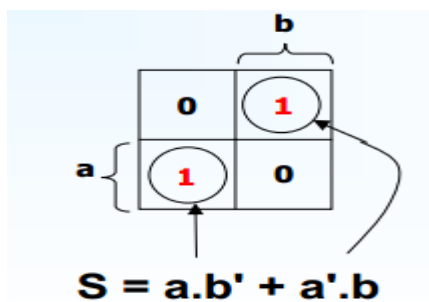


4. Full-Adder using NAND gates only

**DESIGN:**

1) Half adder

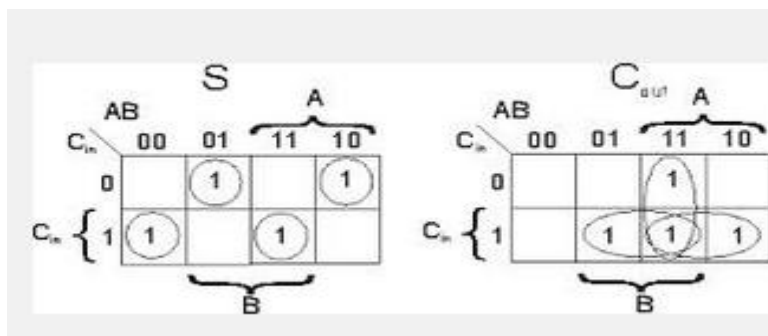
Inputs		Outputs	
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Using basic gatesUsing NAND Logic

$$\begin{aligned}
 S &= A\bar{B} + \bar{A}B = A\bar{B} + A\bar{A} + \bar{A}B + B\bar{B} \\
 &= A(\bar{A} + \bar{B}) + B(\bar{A} + \bar{B}) \\
 &= A \cdot \overline{AB} + B \cdot \overline{AB} \\
 &= \overline{A \cdot AB \cdot B \cdot AB} \\
 C &= AB = \overline{\overline{AB}}
 \end{aligned}$$

2) Full adder

Inputs			Sum	Carry
A	B	C _{in}	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



Using basic gates

$$\begin{aligned}
 S &= \bar{A}\bar{B}C_{in} + \bar{A}B\bar{C}_{in} + A\bar{B}\bar{C}_{in} + ABC_{in} \\
 &= (\bar{A}\bar{B} + \bar{A}B)\bar{C}_{in} + (A\bar{B} + \bar{A}B)C_{in} = (A \oplus B)\bar{C}_{in} + (\overline{A \oplus B})C_{in} = A \oplus B \oplus C_{in}
 \end{aligned}$$

$$C_{out} = \bar{A}BC_{in} + A\bar{B}C_{in} + AB\bar{C}_{in} + ABC_{in} = AB + (A \oplus B)C_{in} = AB + AC_{in} + BC_{in}$$

Using NAND Logic

$$S = A \oplus B \oplus C_{in} = \overline{(A \oplus B) \cdot (A \oplus B)C_{in} \cdot C_{in} \cdot (A \oplus B)C_{in}}$$

$$C_{out} = C_{in}(A \oplus B) + AB = \overline{\overline{C_{in}(A \oplus B)} \cdot \overline{AB}}$$

EXPERIMENT:**Procedure**

- 1) Place the IC on IC Trainer Kit.
- 2) Connect V_{CC} and ground to respective pins of IC Trainer Kit.
- 3) Connect the inputs to the input switches provided in the IC Trainer Kit.
- 4) Connect the outputs to the switches of output LEDs
- 5) Apply various combinations of inputs according to the truth table and observe condition of LEDs
- 6) Disconnect output from the LEDs and note down the corresponding multimeter voltage readings for various combinations of inputs

OBSERVATIONS:**Truthtable****Half-Adder**

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Full Adder

A	B	C_{IN}	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

CONCLUSION**REVIEW QUESTIONS:**

1. Realize the half adder and full adder using NOR gates
2. Realize the subtractor using (a) basic gates (b) NAND gates (c) NOR gates
3. Design adder/subtractor circuit using 7483 IC
4. Design a BCD adder circuit
5. What are the applications of half adder and full adder?
6. What is a half subtractor and full subtractor?
7. What is a BCD adder?
8. What is a half adder subtractor?

FLIP FLOPS**OBJECTIVE:**

To implement various flip flops using NAND gates and to familiarize the ICs 7474 and 7476

HARDWARE REQUIRED:

SLNo.	Components/Equipments	Specification	Quantity
1.	Digital Trainer Kit		
2.	IC		

INTRODUCTION:

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

CLOCKED SR FLIP FLOP

S and R stands for set and reset. There are 4 input combinations possible. But $S=R=1$ is forbidden, since the output will be invalid. When the flip flop is switched on, its output state will be uncertain. When an initial state is to be assigned, two separate inputs called preset and clear are used. They are active low inputs.

JK FLIP FLOPS

The invalid output state of S-R flip flop, when $S=R=1$ is avoided by converting it into a J-K flip flop.

MASTER SLAVE JK FLIP FLOPS

The race around condition of JK flip flop is rectified in Master Slave JK flip flop. Racing is the toggling of the output more than once during a positive clock edge. Master Slave JK flip flop is created by cascading two JK flip flops. The clock fed to the first stage (master) is inverted and fed to the second stage (slave). This ensures that the slave follows the master eliminates the chance of racing.

D FLIP FLOP

It has only one input referred to as D input or data input. The input data is transferred to the output after a clock pulse applied. D flip flop can be derived JK flip flops by using J inputs as D input and J is inverted and fed to K input.

T FLIP FLOP

T stands for toggle. The output toggles when a clock pulse is applied. That is the output of the flip flop changes state for an input pulse. T flip flop can be derived from JK flip flop by shorting J and K inputs.

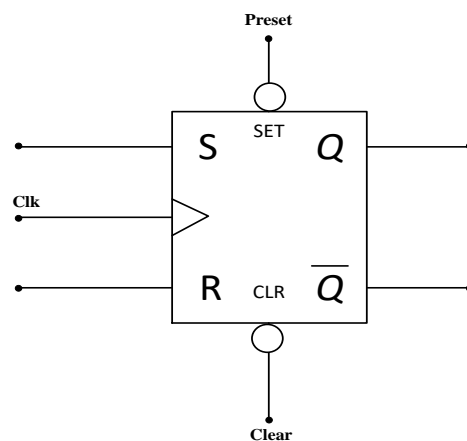
FLIP FLOP ICS

IC 7476 is dual negative edge triggered MS JK flip flop with preset and clear facility. It has a 16 pin DIP chip. IC 7473 is a dual negative edge triggering Master Slave JK flip flop with clear in 14 pin DIP. It does not have preset input. IC 7474 is positive edge triggered dual D flip flop with preset and clear in 14 pin DIP.

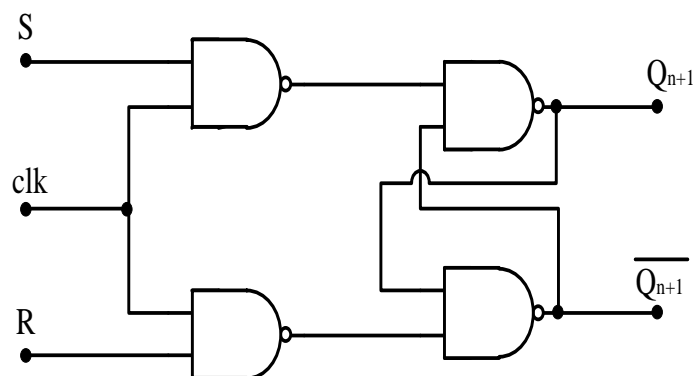
CIRCUIT DIAGRAM AND OBSERVATIONS:

1) SR FLIPFLOP

LOGIC SYMBOL

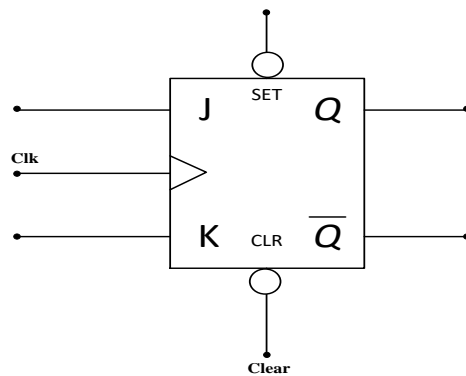


SR FLIP FLOP USING GATES

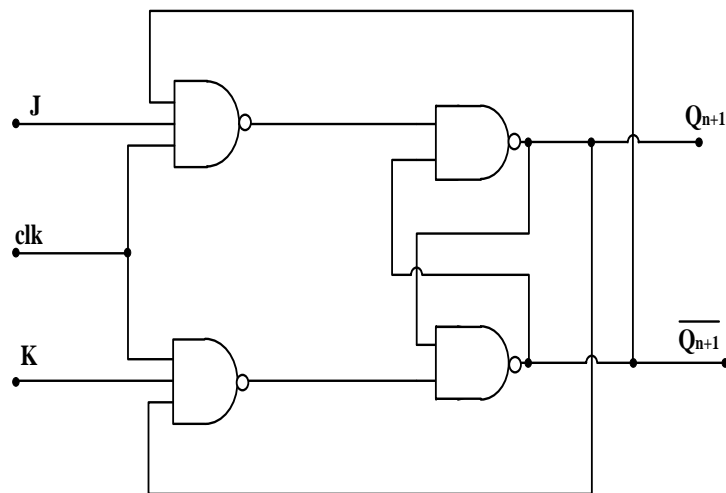


2) JK FLIP FLOP

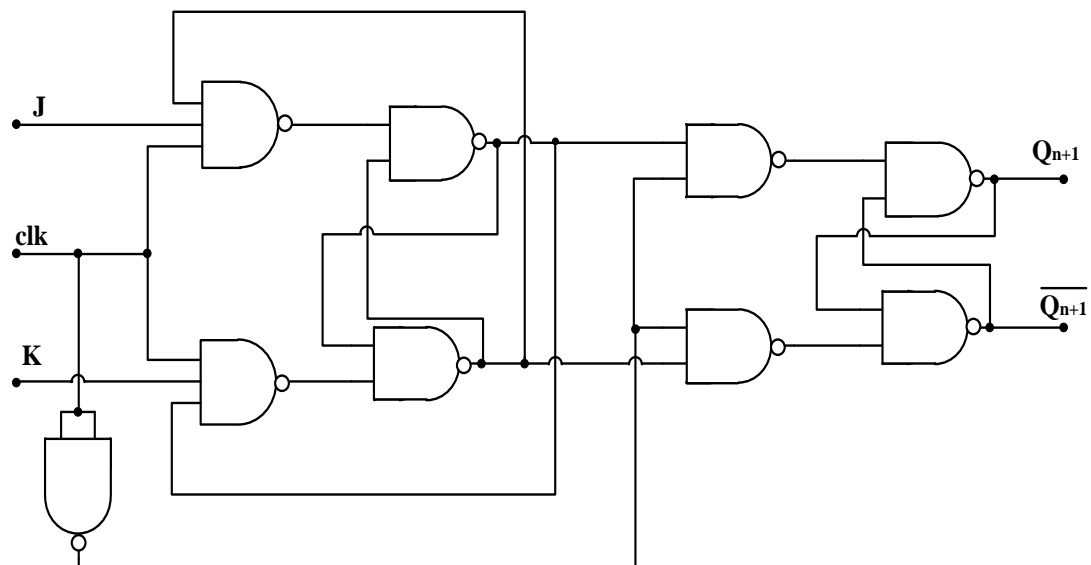
LOGIC SYMBOL



JK FLIP FLOP USING GATES

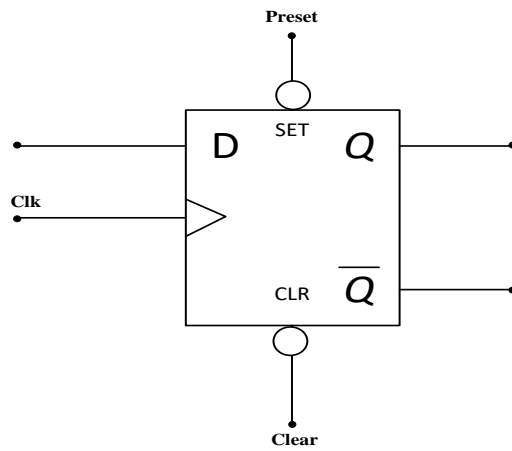


3) MASTER SLAVE JK FLIP FLOP NAND GATES

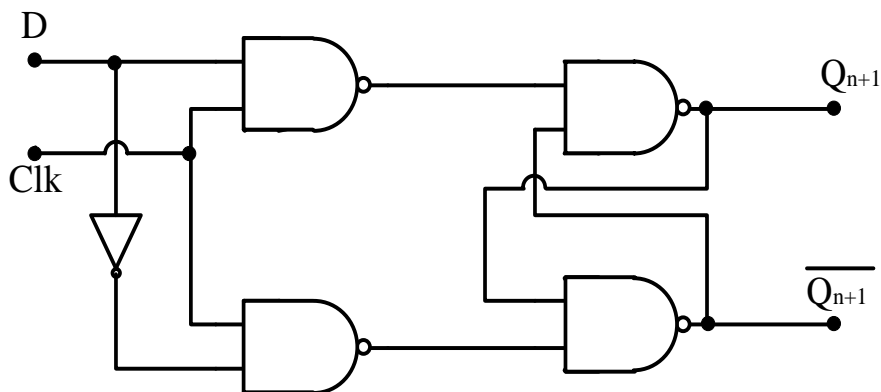


4) D FLIPFLOP

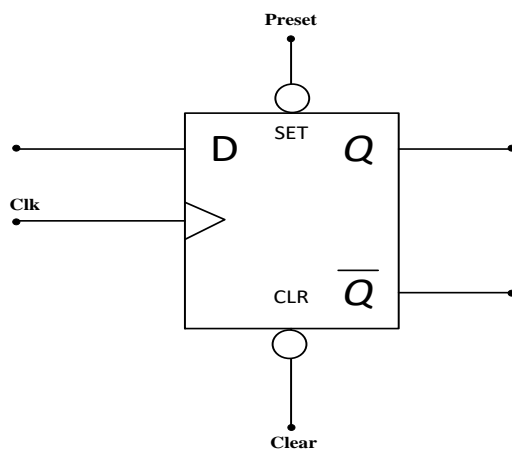
LOGIC SYMBOL



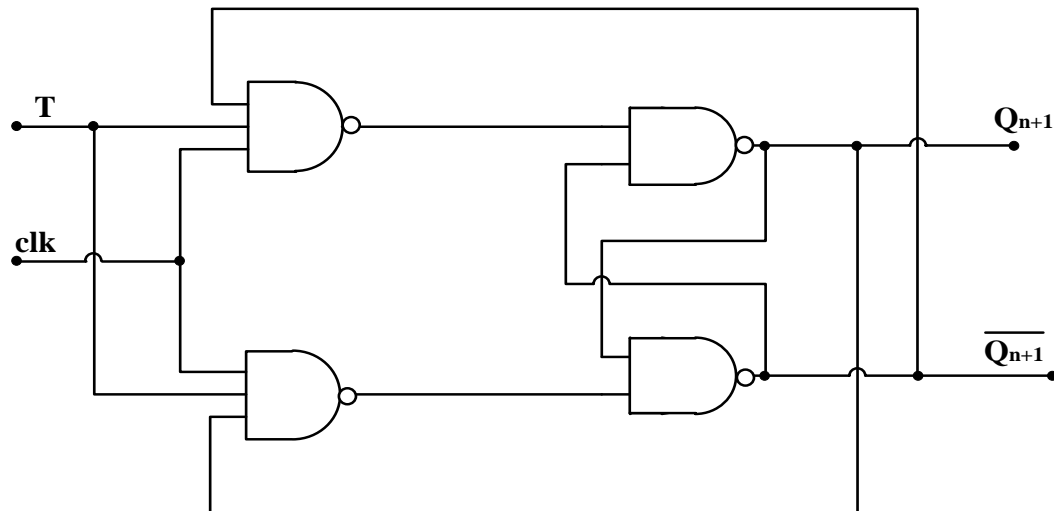
D FLIP FLOP USING GATES

**5) T FLIP FLOP**

LOGIC SYMBOL

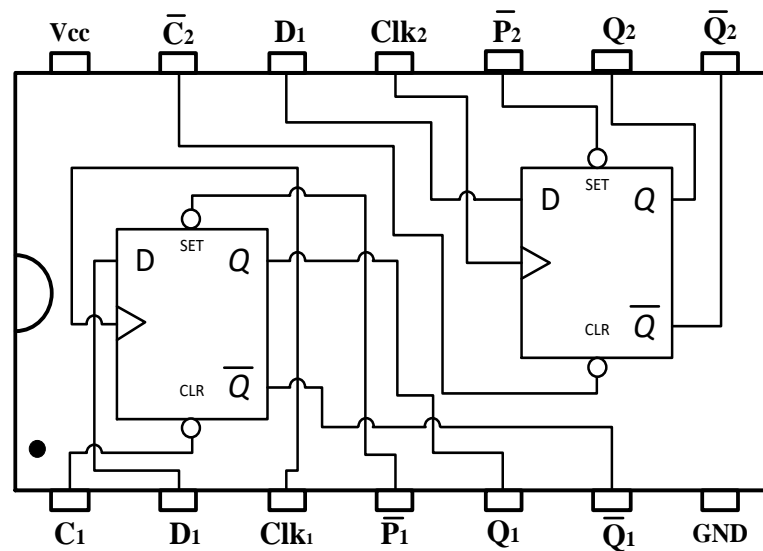


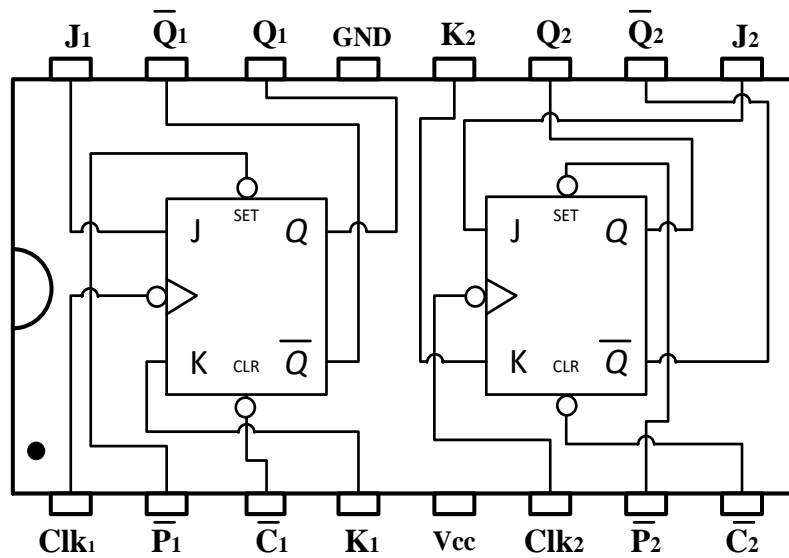
T FLIP FLOP USING GATES



FLIP FLOP ICS

D FLIP FLOP IC- 7474



JK FLIPFLOP IC-7476**EXPERIMENT:****Procedure**

- 1) Place the IC on IC Trainer Kit.
- 2) Connect V_{CC} and ground to respective pins of IC Trainer Kit.
- 3) Test all the components and IC packages using multimeter or digital IC tester. Set up the flip flops using gates and verify their truth table.
- 4) Verify the truth tables of 7474, 7476

OBSERVATIONS:**Truth table****1) SR flipflop**

Inputs		Output
S_n	R_n	Q_{n+1}
0	0	Q_n
1	0	1
0	1	0
1	1	?

2) JK flipflop

Inputs			Outputs		
J	K	C	Q	Q'	Comments
0	0	↑	Q	Q'	No change
0	1	↑	0	1	RESET
1	0	↑	1	0	SET
1	1	↑	Q'	Q	Toggle

3) Master slave JK flip-flop

J	K	$Q_{(t+1)}$
0	0	$Q_{(t)}$ unchanged
0	1	0 reset
1	0	1 set
1	1	$\bar{Q}_{(t)}$ output inversion

4) D flip flop

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

5) T flip flop

T	Q	\bar{Q}
0	Q	\bar{Q}
1	\bar{Q}	Q
0	\bar{Q}	Q
1	Q	\bar{Q}

CONCLUSION**REVIEW QUESTIONS:**

- 1) What is a latch?
- 2) What is edge triggered flipflop?
- 3) Explain single bit memory element.
- 4) Explain some applications of flipflops.
- 5) Explain race around condition?
- 6) Differentiate combinational and sequential circuits
- 7) Draw all flipflop IC's?

EXPT NO: 5

DATE: __/__/__

ASYNCHRONOUS COUNTERS**OBJECTIVE:**

To realize asynchronous counters

HARDWARE REQUIRED:

SLNo.	Components/Equipments	Specification	Quantity
1.	Digital Trainer Kit		1
2.	IC	7476, 7486	1
		7400	1
		7473	1

INTRODUCTION:**Asynchronous counter**

The term asynchronous refers to events that do not occur at the same time. With respect to counter operation, asynchronous means that the flip-flops within the counter are not connected in a way to cause all flip-flops at exactly the same time. They are wired in a way that links the clock of the next flip-flop to the Q of the current device. This causes the output count states to ripple through the counter. A counter is a circuit that produces a set of unique output combinations corresponding to the number of applied input pulses. The number of unique outputs of a counter is known as its modulus or mod number. Each flip flop is triggered by the output from the previous flip flop except for the first flip flop (LSB) which receives an external clock. Asynchronous counters are commonly referred to as ripple counters because the effect of the input clock pulse is first felt by FF0. This effect cannot get to FF1 immediately because of the propagation delay through FF0.

Asynchronous decade counter

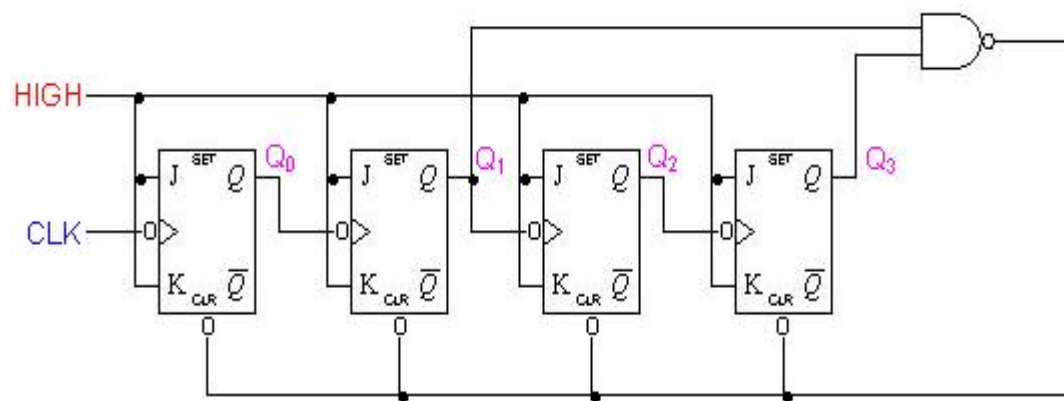
The modulus for counters is the number of unique states through which the counter will sequence. The maximum possible number of states of a counter is 2^n , where n is the number of flip flops in the counter. A decade counter with a count sequence of zero (0000) through nine (1001) is a BCD decade counter because its ten-state sequence produces the BCD code. This type of counter is useful in display applications in which BCD is required for conversion to a decimal readout.

3-bit up/down counter

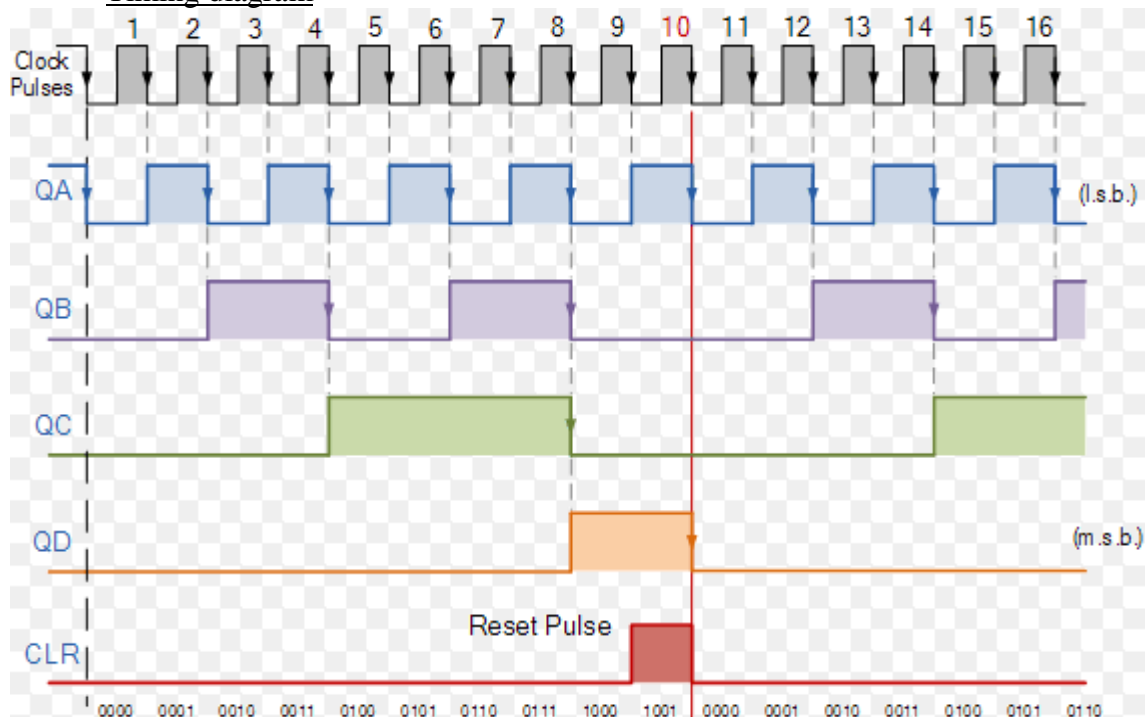
The direction of counting is decided by a mode control input M. An XOR gate between flip flop functioning as a controlled inverter connects either of Q or Q bar to the clock input of the succeeding flip flop as decided by the logic state at M. When mode control is 0, Q outputs get connected to the clock inputs of the succeeding flip flop and counter counts up. When mode control is 1, Qbar outputs are connected to the clock inputs and counter counts down.

CIRCUIT DIAGRAM AND DESIGN:

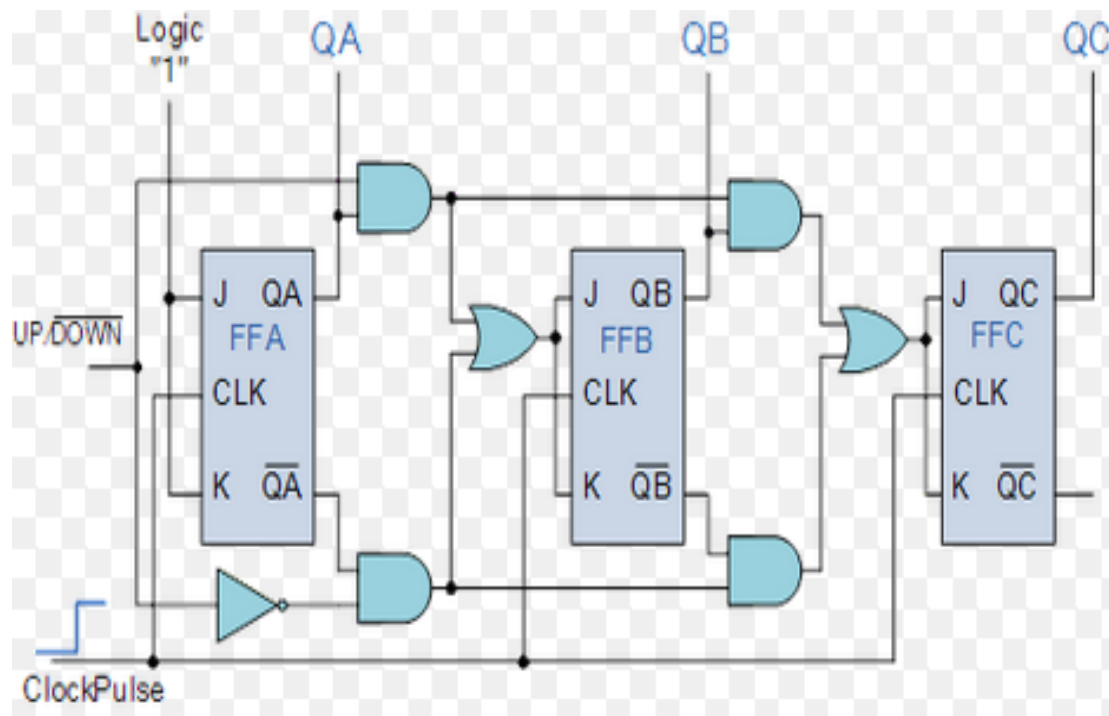
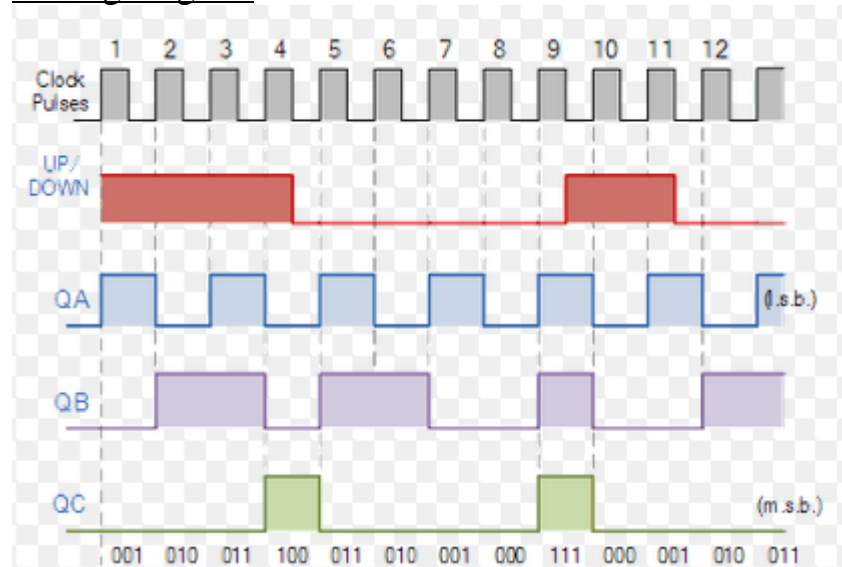
1) Decade counter



Timing diagram



2) 3-bit up/down counter using mode control

Timing diagram**EXPERIMENT:****Procedure**

- 1) Place the IC on IC Trainer Kit.
- 2) Connect V_{CC} and ground to respective pins of IC Trainer Kit.


- 3) Connect the inputs to the input switches provided in the IC Trainer Kit.
- 4) Connect the outputs to the switches of output LEDs
- 5) Apply various combinations of inputs according to the truth table and observe condition of LEDs
- 6) Disconnect output from the LEDs and note down the corresponding multimeter voltage readings for various combinations of inputs

OBSERVATIONS:

Truth table

- 1) Decade counter

Clock Pulse	Q3	Q2	Q1	Q0
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



- 2) 3-bit up/down counter using mode control

CLK	M	Q ₂	Q ₁	Q ₀
0	1	0	0	0
1	1	1	1	1
2	1	1	1	0
3	1	1	0	1
4	1	1	0	0
5	1	0	1	1
6	1	0	1	0
7	1	0	0	1
8	1	0	0	0
9	0	0	0	1
10	0	0	1	0
11	0	0	1	1
12	0	1	0	0
13	0	1	0	1
14	0	1	1	0
15	0	1	1	1

CONCLUSION**REVIEW QUESTIONS:**

1. Give some applications of all the above circuits.
2. What is the limitation of asynchronous counters?
3. How many flip flops will be complemented in a 10-bit binary ripple counter to reach the next count after the following counts?
 - (a) 1001100111
 - (b) 0011111111
4. Design a circuit which simultaneously divide the input frequency by a factor of 2,4,8.
5. Set up a 4-bit decade down counter to count from 9 to 0.

EXPI NO: 6

DATE: __/__/__

SYNCHRONOUS COUNTERS**OBJECTIVE:**

Realization of synchronous counters

HARDWARE REQUIRED:

SLNo.	Components/Equipments	Specification	Quantity
1.	Digital Trainer Kit		1
2.	IC	7476	1
		7400	1
		7408	1

INTRODUCTION:**Synchronous counter**

A synchronous counter, in contrast to an asynchronous counter, is one whose output bits change state simultaneously, with no ripple. The only way we can build such a counter circuit from J-K flip-flops is to connect all the clock inputs together, so that each and every flip-flop receives the exact same clock pulse at the exact same.

Asynchronous counter

The term asynchronous refers to events that do not occur at the same time. With respect to counter operation, asynchronous means that the flip-flops within the counter are not connected in a way to cause all flip-flops at exactly the same time. They are wired in a way that links the clock of the next flip-flop to the Q of the current device. This causes the output count states to ripple through the counter.

Mod -N Counter

A mod-n counter has n possible states. That means it counts from 0 to n and rolls over it. There must be a way to force the counter stops counting at n and roll over to 0. This is where the asynchronous inputs come into play. The asynchronous inputs can override the synchronous inputs and force the output either at high or low.

DESIGN and CIRCUIT DIAGRAM :

1) 4 bit synchronous up counter

Present state	Next state	J3	K3	J2	K2	J1	K1	J0	K0
0000	0001	0	X	0	X	0	X	1	X
0001	0010	0	X	0	X	1	X	X	1
0010	0011	0	X	0	X	X	0	1	X
0011	0100	0	X	1	X	X	1	X	1
0100	0101	0	X	X	0	0	X	1	X
0101	0110	0	X	X	0	1	X	X	1
0110	0111	0	X	X	0	X	0	1	X
0111	1000	1	X	X	1	X	1	X	1
1000	1001	X	0	0	X	0	X	1	X
1001	1010	X	0	0	X	1	X	X	1
1010	1011	X	0	0	X	X	0	1	X
1011	1100	X	0	1	X	X	1	X	1
1100	1101	X	0	X	0	0	X	1	X
1101	1110	X	0	X	0	1	X	X	1
1110	1111	X	0	X	0	X	0	1	X
1111	0000	X	1	X	1	X	1	X	1

K-maps:

$Q_1 Q_0$ $Q_3 Q_2$	00	01	11	10
00	0	0	0	0
01	0	0	1	0
11	x	x	x	X
10	x	x	x	x

$$J_3 = Q_2 Q_1 Q_0$$

$Q_1 Q_0$ $Q_3 Q_2$	00	01	11	10
00	x	x	x	X
01	x	X	x	x
11	0	0	1	0
10	0	0	0	0

$$K_3 = Q_2 Q_1 Q_0$$

$Q_1 Q_0$ $Q_3 Q_2$	00	01	11	10
00	0	0	1	0
01	X	X	X	X
11	X	X	X	X
10	0	0	1	0

$$J_2 = Q_1 Q_0$$

$Q_1 Q_0$ $Q_3 Q_2$	00	01	11	10
00	x	x	x	X
01	0	0	1	0
11	0	0	1	0
10	x	x	x	X

$$K_2 = Q_1 Q_0$$

$Q_1 Q_0$ $Q_3 Q_2$	00	01	11	10
00	0	1	x	X
01	0	1	x	X
11	0	1	x	x
10	0	1	x	x

$$J_1 = Q_0$$

$Q_1 Q_0$ $Q_3 Q_2$	00	01	11	10
00	x	x	1	0
01	x	x	1	0
11	x	x	1	0
10	x	x	1	0

$$K_1 = Q_0$$

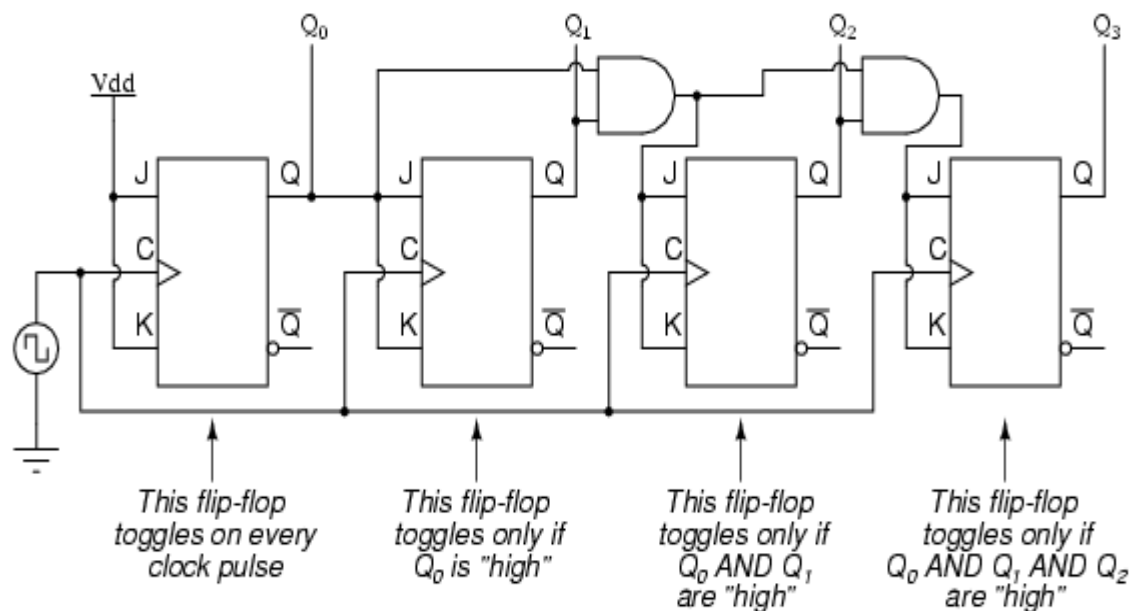
$Q_1 Q_0$ $Q_3 Q_2$	00	01	11	10
00	1	1	1	1
01	x	x	x	x
11	x	x	x	X
10	1	1	1	1

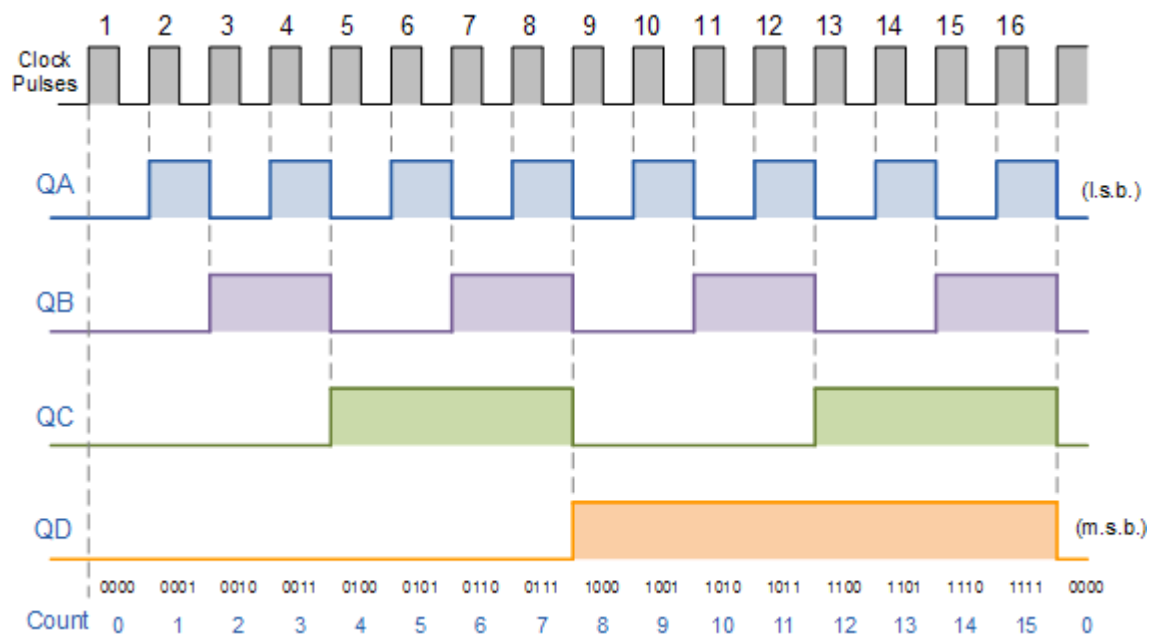
$$J_0 = 1$$

$Q_1 Q_0$ $Q_3 Q_2$	00	01	11	10
00	x	x	x	X
01	1	1	1	1
11	1	1	1	1
10	x	x	x	x

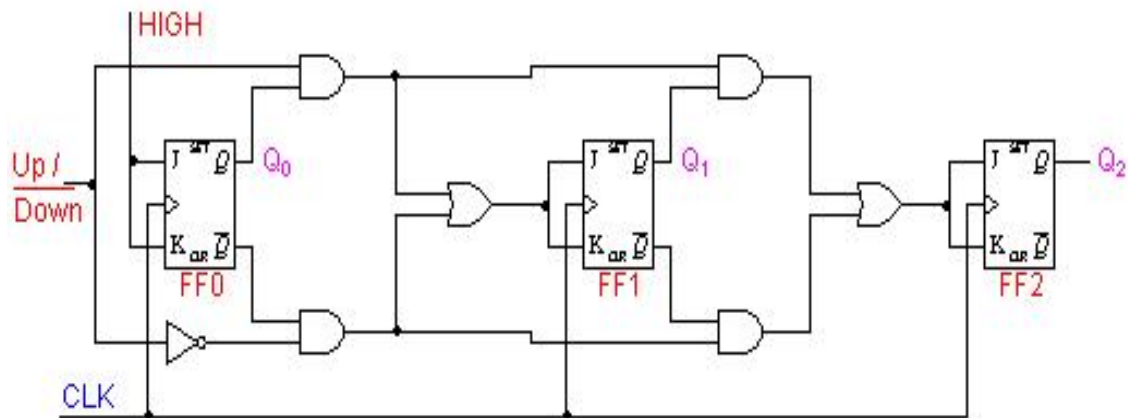
$$K_0 = 1$$

Circuit daigram

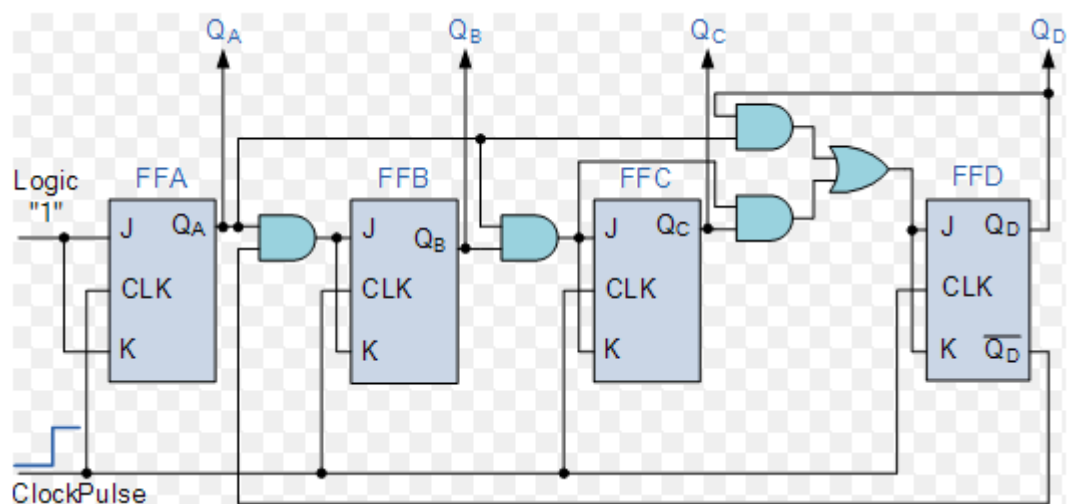


Timing diagram**2) 3 bit synchronous up/down counter**Truth Table

Clock pulse	Present state				Next state		
	M						
0	0	0	0	0	0	0	1
1	0	0	0	1	0	1	0
2	0	0	1	0	0	1	1
3	0	0	1	1	1	0	0
4	0	1	0	0	1	0	1
5	0	1	0	1	1	1	0
6	0	1	1	0	1	1	1
7	0	1	1	1	0	0	0
8	1	0	0	0	1	1	1
9	1	0	0	1	0	0	0
10	1	0	1	0	0	0	1
11	1	0	1	1	0	1	0
12	1	1	0	0	0	1	1
13	1	1	0	1	1	0	0
14	1	1	1	0	1	0	1
15	1	1	1	1	1	1	0

Circuit daigram**3) Mod 10 synchronous counter****Truth Table**

Clock Pulse	Q3	Q2	Q1	Q0
Initially	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
10 (recycles)	0	0	0	0

Circuit daigram

- 4) Random sequence generator to generate the sequence

EXPERIMENT:**Procedure**

1. Place the IC on IC Trainer Kit.
2. Connect V_{CC} and ground to respective pins of IC Trainer Kit.
3. Connect the inputs to the input switches provided in the IC Trainer Kit.
4. Connect the outputs to the switches of output LEDs
5. Apply various combinations of inputs according to the truth table and observe condition of LEDs
6. Disconnect output from the LEDs and note down the corresponding multimeter voltage readings for various combinations of inputs.

CONCLUSION

The synchronous, asynchronous and mod n counters are designed and implemented.

REVIEW QUESTIONS:

1. Give some applications of all the above circuits.
2. With a neat circuit diagram explain the working of a precision bridge rectifier?

SHIFT REGISTERS**OBJECTIVE:**

To design and Implement the Shift Registers.

HARDWARE REQUIRED:

SLNo.	Components/Equipments	Specification	Quantity
1.	Digital Trainer Kit		1
2.	IC	7474	1
		7473	1

INTRODUCTION:**Shift register**

In digital circuits, a shift register is a cascade of flip flops, sharing the same clock, which has the output of anyone but the last flip-flop connected to the "data" input of the next one in the chain, resulting in a circuit that shifts by one position the one-dimensional "bit array" stored in it, shifting in the data present at its input and shifting out the last bit in the array, when enabled to do so by a transition of the clock input. More generally, a shift register may be multidimensional; such that its "data in" input and stage outputs are themselves bit arrays: this is implemented simply by running several shift registers of the same bit-length in parallel. Shift registers can have both parallel and serial inputs and outputs. These are often configured as serial-in, parallel-out (SIPO) or as parallel-in, serial-out (PISO). There are also types that have both serial and parallel input and types with serial and parallel output.

Serial in serial out shift registers

These are the simplest kind of shift registers. The data string is presented at 'Data In', and is shifted right one stage each time 'Data Advance' is brought high. At each advance, the bit on the far left (i.e. 'Data In') is shifted into the first flip-flop's output. The bit on the far right (i.e. 'Data Out') is shifted out and lost.

Serial in parallel out shift registers

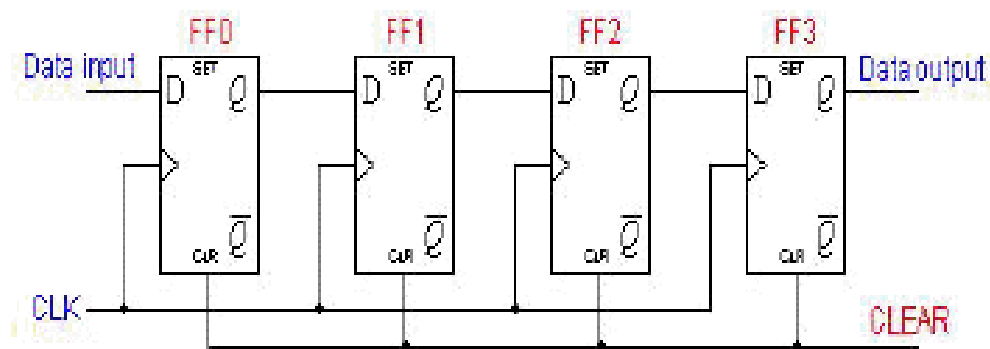
This configuration allows conversion from serial to parallel format. Data is input serially, as described in the SISO section above. Once the data has been input, it may be either read off at each output simultaneously, or it can be shifted out and replaced.

Parallel in serial out shift registers

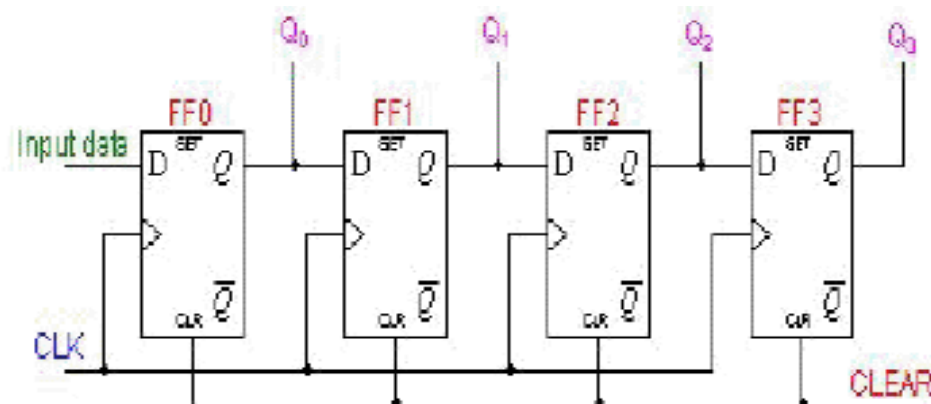
This configuration has the data input on lines D1 through D4 in parallel format. To write the data to the register, the Write/Shift control line must be held LOW. To shift the data, the W/S control line is brought HIGH and the registers are clocked. The arrangement now acts as a SISO shift register, with D1 as the Data Input. However, as long as the number of clock cycles is not more than the length of the data-string, the Data Output, Q, will be the parallel data read off in order.

CIRCUIT DIAGRAM :

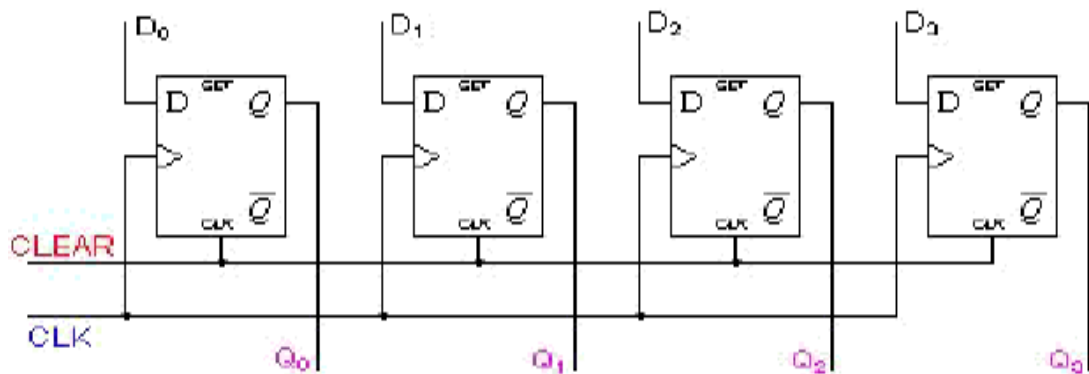
1) Serial in serial out shift register



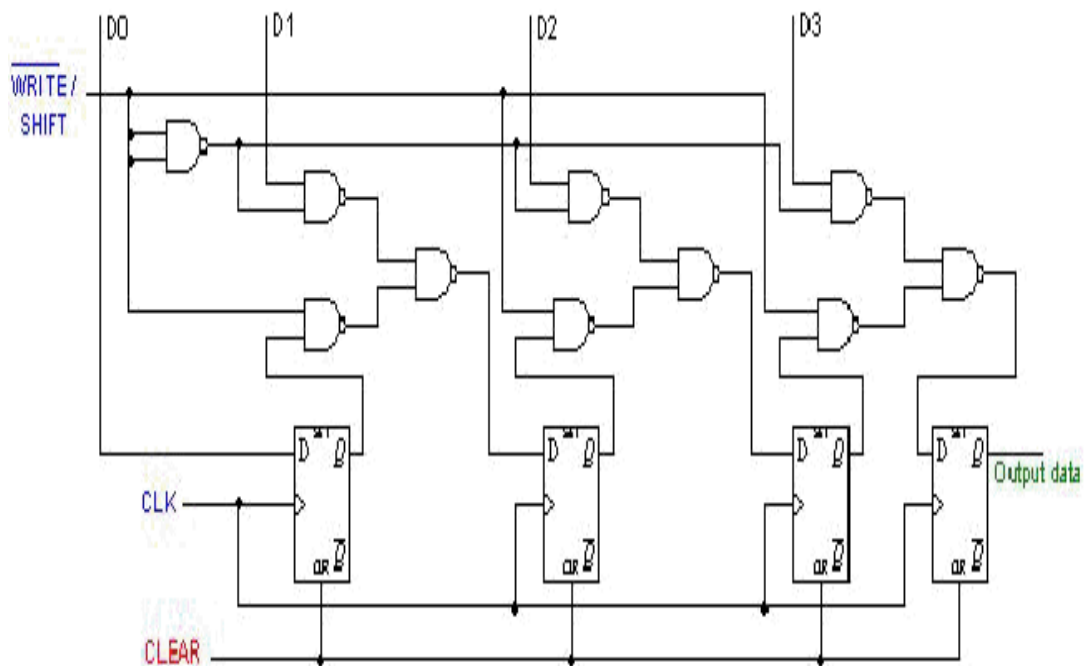
2) Serial in parallel out shift register



3) Parallel in parallel out shift register



4) Parallel in serial out shift register



EXPERIMENT:

Procedure

- 1) Place the IC on IC Trainer Kit.
- 2) Connect V_{CC} and ground to respective pins of IC Trainer Kit.

- 3) Connect the inputs to the input switches provided in the IC Trainer Kit.
- 4) Connect the outputs to the switches of output LEDs
- 5) Apply various combinations of inputs according to the truth table and observe condition of LEDs
- 6) Disconnect output from the LEDs and note down the corresponding multimeter voltage readings for various combinations of inputs

OBSERVATIONS:

Truthtable

Serial in serial out shift register

Clock	Serial i/p	QA	QB	QC	QD
1	do=0	0	X	X	X
2	d1=1	1	0	X	X
3	d2=1	1	1	0	X
4	d3=1	1	1	1	0=do
5	X	X	1	1	1=d1
6	X	X	X	1	1=d2
7	X	X	X	X	1=d3

Serial in parallel out shift register

Clock	Serial i/p	QA	QB	QC	QD
1	0	0	X	X	X
2	1	1	0	X	X
3	1	1	1	0	X
4	1	1	1	1	0

Parallel in parallel out shift register

Clock	Parallel i/p				Parallel o/p			
	A	B	C	D	QA	QB	QC	QD
1	1	0	1	1	1	0	1	1

Parallel in serial out shift register

Mode	Clock	Parallel i/p				Parallel o/p			
		A	B	C	D	QA	QB	QC	QD
1	1	1	0	1	1	1	0	1	1
0	2	X	X	X	X	X	1	0	1
0	3	X	X	X	X	X	X	1	0
0	4	X	X	X	X	X	X	X	1

CONCLUSION

.

REVIEW QUESTIONS:

- 1) Explain how to multiply or divide a binary number by using shift registers?
- 2) Distinguish between shift registers and counters?

RING COUNTER AND JOHNSON COUNTER

OBJECTIVE:

To design and Implement counters using shift registers – Ring Counter and Johnson Counter.

HARDWARE REQUIRED:

SLNo.	Components/Equipments	Specification	Quantity
1.	Digital Trainer Kit		1
2.	IC	7473	1
		7474	1

INTRODUCTION:

Ring Counter

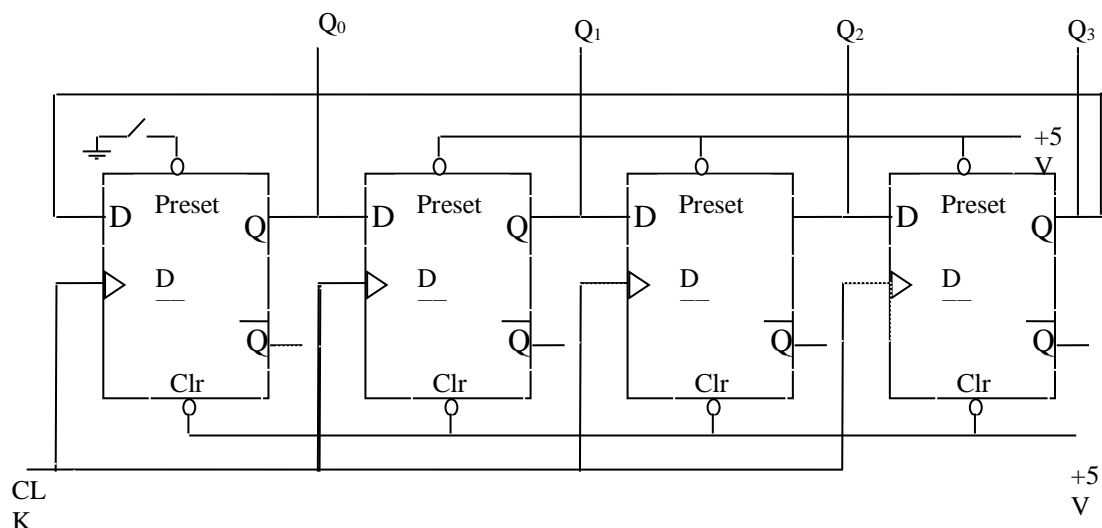
A ring counter is basically a circulating shift register in which the output of the MSB is fed back to the input of the LSB. Here the 4 bit ring counter is constructed using D flip-flops. The output of each stage is shifted into the next stage on the positive edge of a clock pulse. Here the clear signal is high, all the flip-flops except first one FF are reset to 0. FF0 is preset to 1 instead.

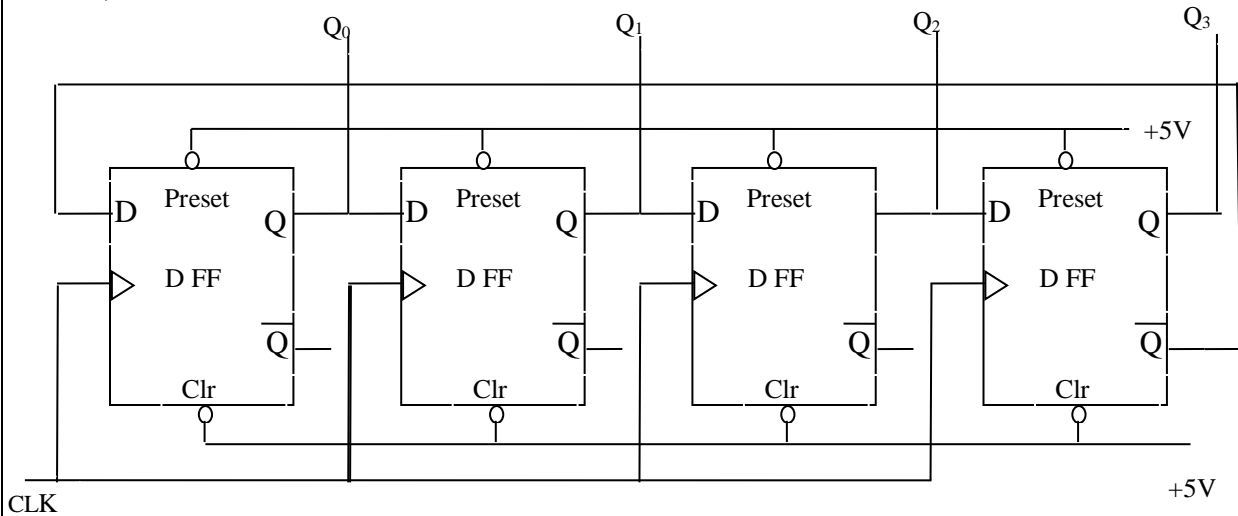
Johnson Counter

They are a variation of standard ring counters, with the inverted output of the last stage fed back to the first stage. They are also known as twisted ring counters. An n-stage Johnson counter yields a count sequence of length $2n$, so it may be considered to be mod $2n$ counter.

CIRCUIT DIAGRAM

1) Ring Counter



2) Johnson Counter**EXPERIMENT:****Procedure**

- 1) Place the IC on IC Trainer Kit.
- 2) Connect V_{CC} and ground to respective pins of IC Trainer Kit.
- 3) Connect the inputs to the input switches provided in the IC Trainer Kit.
- 4) Connect the outputs to the switches of output LEDs
- 5) Apply various combinations of inputs according to the truth table and observe condition of LEDs

OBSERVATIONS:**Truthtable****Ring Counter**

Clock Pulse	Q_3	Q_2	Q_1	Q_0
0	0	0	0	1
1	0	0	1	0
2	0	1	0	0
3	1	0	0	0

Johnson Counter

Clock Pulse	Q_3	Q_2	Q_1	Q_0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	1
3	0	1	1	1
4	1	1	1	1
5	1	1	1	0
6	1	1	0	0
7	1	0	0	0
8	0	0	0	0

CONCLUSION:

REVIEW QUESTIONS:

- 1) With an example explain self starting counter?
- 2) What are shift register counters?
- 3) What do you meant by lock-out of a counter?
- 4) Differentiate Ring counter and Johnson counter?

EXPT NO: 8

DATE: __/__/__

DESIGN AND IMPLEMENTATION OF MULTIPLEXER AND DEMULTIPLEXER**OBJECTIVE:**

To design and implement multiplexer and demultiplexer.

HARDWARE REQUIRED:

SLNo.	Components/Equipments	Specification	Quantity
1.	Digital Trainer Kit		
2.	IC	74151	1
		74154	1

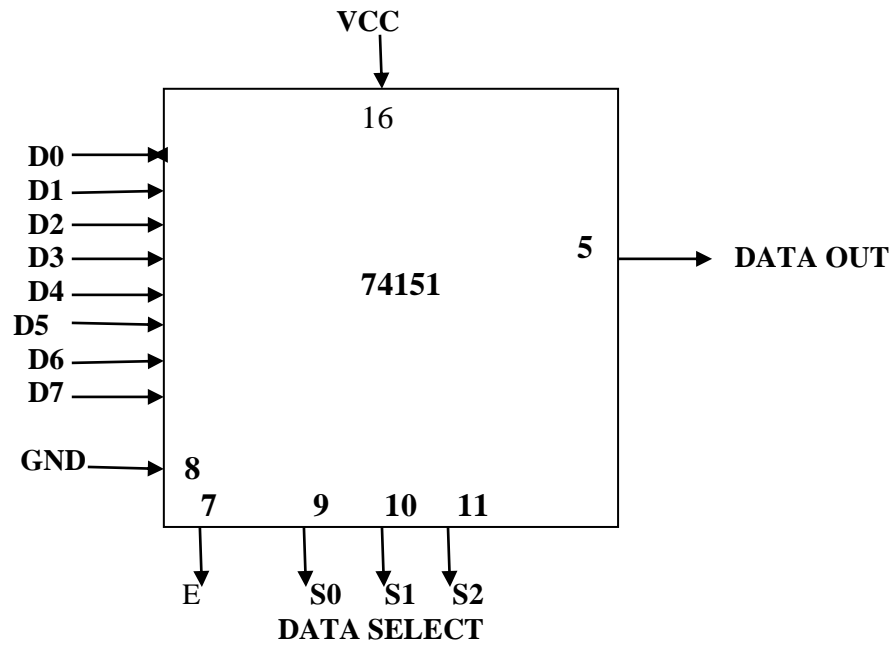
INTRODUCTION:**MULTIPLEXER:**

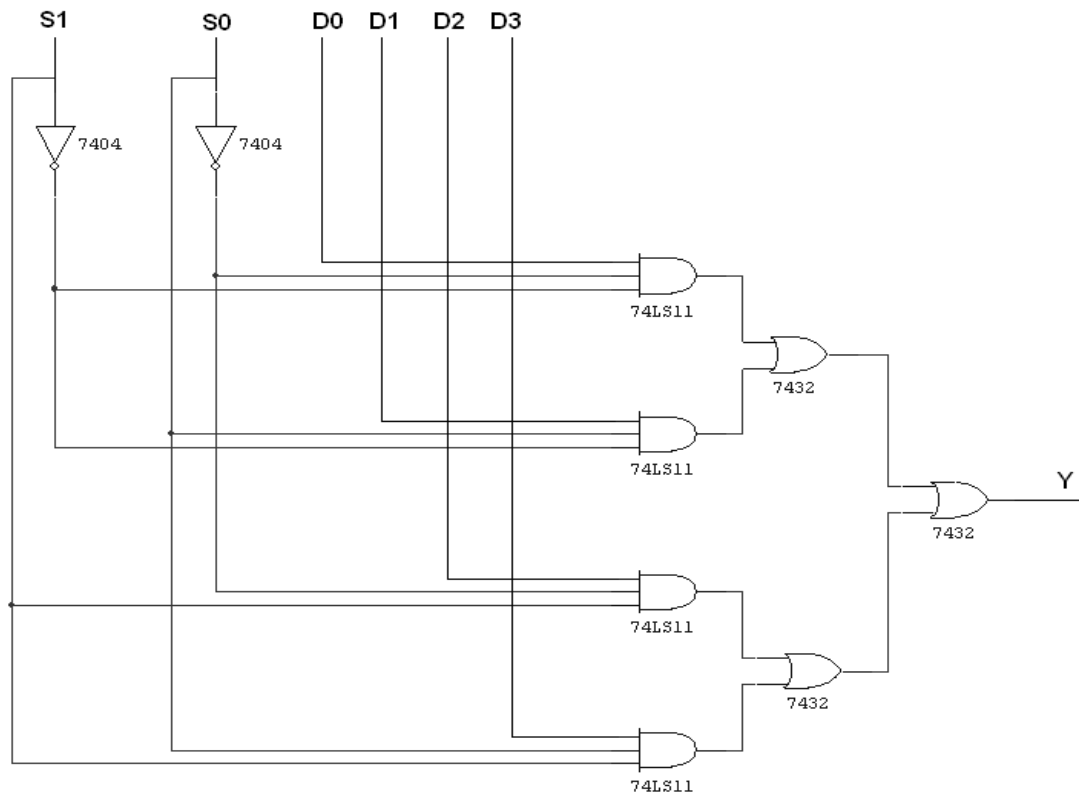
Multiplexer means transmitting a large number of information units over a smaller number of channels or lines. A digital multiplexer is a combinational circuit that selects binary information from one of many input lines and directs it to a single output line. The selection of a particular input line is controlled by a set of select lines. Normally there are 2^n input lines and n selection lines whose bit combination determines which input is selected.

DEMULTIPLEXER:

The function of Demultiplexer is in contrast to multiplexer function. It takes information from one line and distributes it to a given number of output lines. For this reason, the demultiplexer is also known as a data distributor. Decoder can also be used as demultiplexer.

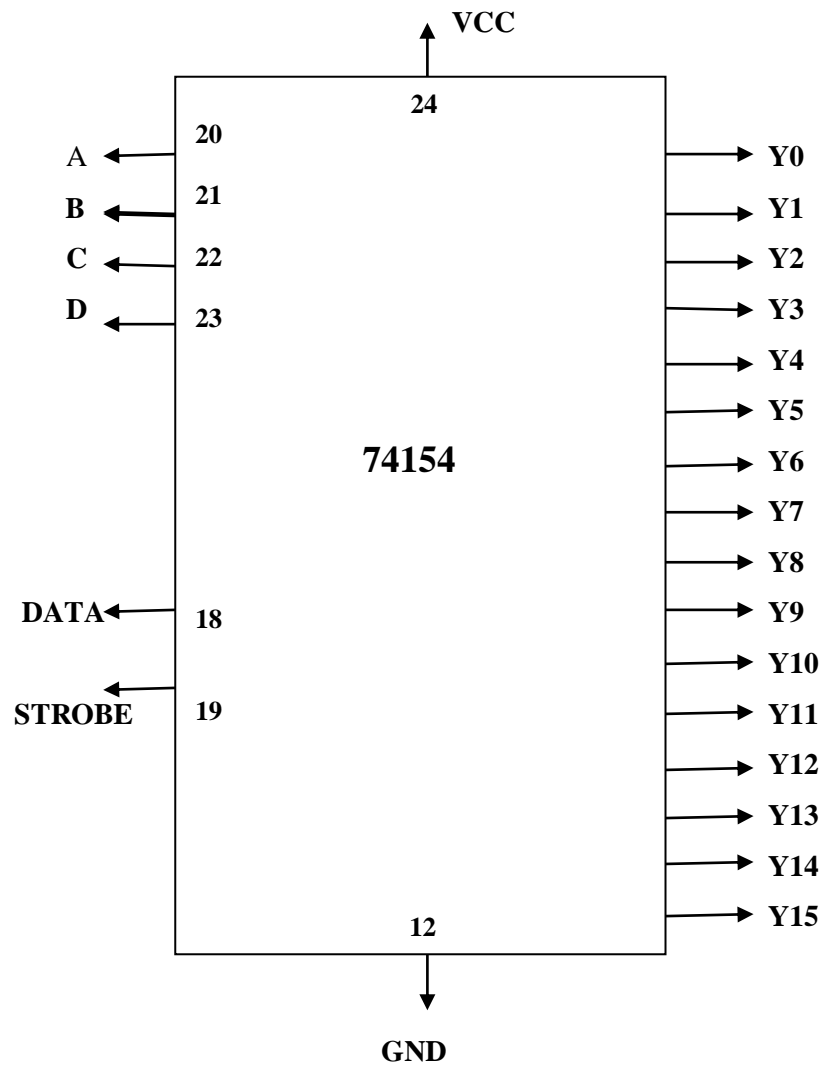
In the 1: 4 demultiplexer circuit, the data input line goes to all of the AND gates. The data select lines enable only one gate at a time and the data on the data input line will pass through the selected gate to the associated data output line.

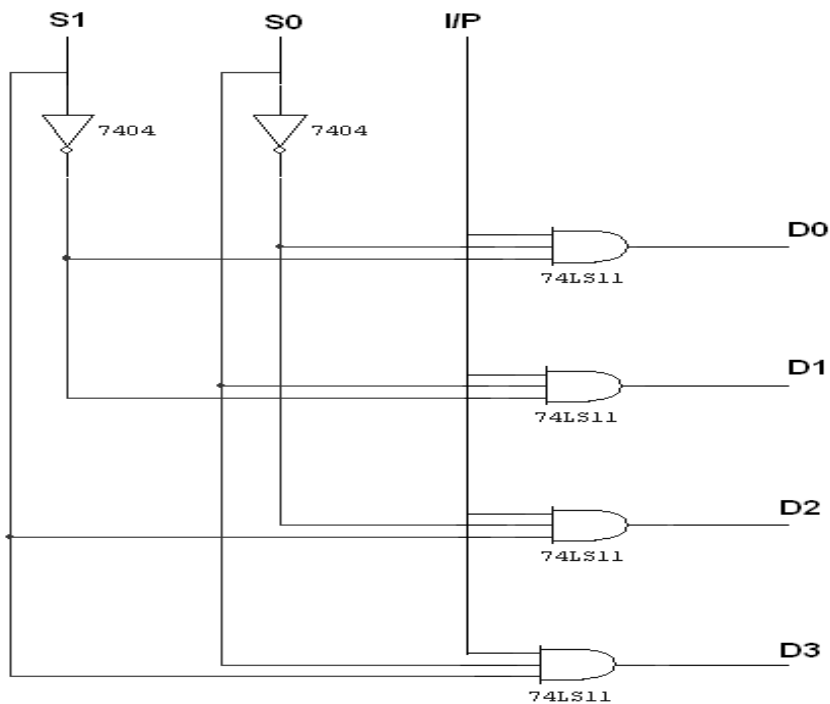
CIRCUIT DIAGRAM :**8:1 MULTIPLEXER:****4:1 MULTIPLEXER:**

**FUNCTION TABLE:**

S1	S0	INPUTS Y
0	0	$D0 \rightarrow D0 S1' S0'$
0	1	$D1 \rightarrow D1 S1' S0$
1	0	$D2 \rightarrow D2 S1 S0'$
1	1	$D3 \rightarrow D3 S1 S0$

$$Y = D0 S1' S0' + D1 S1' S0 + D2 S1 S0' + D3 S1 S0$$

1:16 DECODER / DEMULTIPLEXER:

1:4 DEMULTIPLEXER:**FUNCTION TABLE:**

S1	S0	INPUT
0	0	$X \rightarrow D0 = X S1' S0'$
0	1	$X \rightarrow D1 = X S1' S0$
1	0	$X \rightarrow D2 = X S1 S0'$
1	1	$X \rightarrow D3 = X S1 S0$

$$Y = X S1' S0' + X S1' S0 + X S1 S0' + X S1 S0$$

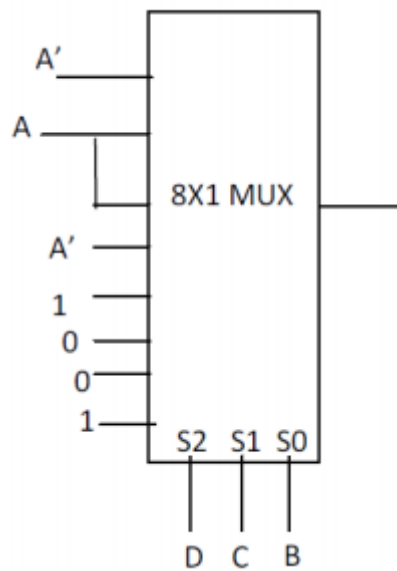
EXPERIMENT:**Procedure**

- 1) Place the IC on IC Trainer Kit.
- 2) Connect V_{CC} and ground to respective pins of IC Trainer Kit.
- 3) Connect the inputs to the input switches provided in the IC Trainer Kit.
- 4) Connect the outputs to the switches of output LEDs
- 5) Apply various combinations of inputs according to the truth table and observe condition of LEDs

DESIGN:

1. Implement $F(A,B,C,D) = \Sigma(0,3,5,6,8,9,14,15)$ using IC 74151

D	C	B	A	F	
0	0	0	0	1	A'
0	0	0	1	0	
0	0	1	0	0	A
0	0	1	1	1	
0	1	0	0	0	A
0	1	0	1	1	
0	1	1	0	1	A'
0	1	1	1	0	
1	0	0	0	1	1
1	0	0	1	1	
1	0	1	0	0	0
1	0	1	1	0	
1	1	0	0	0	0
1	1	0	1	0	
1	1	1	0	1	1
1	1	1	1	1	



2. Implement $F = A'B'CD' + A'BC'D' + A'BCD + AB'C'D'$ using IC 74154.

OBSERVATIONS:

Truth table for multiplexer using gates.

S1	S0	Y = OUTPUT
0	0	D0
0	1	D1
1	0	D2
1	1	D3

Truth table for Demultiplexer using gates.

INPUT			OUTPUT			
S1	S0	I/P	D0	D1	D2	D3
0	0	0	0	0	0	0
0	0	1	1	0	0	0
0	1	0	0	0	0	0
0	1	1	0	1	0	0
1	0	0	0	0	0	0
1	0	1	0	0	1	0
1	1	0	0	0	0	0
1	1	1	0	0	0	1

Truth table for Multiplexer and Demultiplexer using ICs.

CONCLUSION:**REVIEW QUESTIONS:**

1. What is meant by multiplexer?
2. What does De- multiplexer mean?
3. Design a full adder using 8X1 multiplexer?
4. Role of Mux in digital Circuit?
5. Write the applications of multiplexer and demultiplexer?

CYCLE-II

LIST OF EXPERIMENTS

1. Realization of logic gates and familiarization of verilog.
2. Half Adder & Full Adder in verilog.
3. Mux and Demux in verilog.
4. Flipflops and shift registers .
5. Counters

FAMILIARIZATION OF VERILOG

Digital systems are highly complex. At their most detailed level, they may consist of millions of elements, such as a collection of logic gates or transistors. Hardware description languages have evolved to aid in the design of systems with this large number of elements and wide range of electronic and logical abstractions. Hardware description languages have been developed for modeling and simulating hardware functions. Difference between standard programming languages and hardware description languages:

- Standard programming languages: sequential
- HDLs: describe parallel and concurrent behavior

There are two standard HDLs that are supported by IEEE: VHDL and Verilog HDL. The HDL used in our lab will be Verilog.

INTRODUCTION TO VERILOG HDL

Verilog is a Hardware Description Language (HDL) which is used to model electronic systems. It provides the designer entry into the world of large, complex digital systems design. The Verilog language provides the digital system designer with a means of describing a digital system at a wide range of levels of abstraction, and, at the same time, provides access to computer-aided design tools to aid in the design process at these levels. It also fulfils the need for verifying the design for functionality and timing constraints like propagation delay setup and hold times. The components of the target design can be described at different levels with the help of constructs in Verilog.

1. Circuit Level/ Switch Level :MOS switch is the basic element which can be used to build basic circuits like inverters, logic gates, 1-bit dynamic and static memories.
2. Gate Level or structural level: Design is carried out in terms of basic gates. All basic gates are available as ready modules called “primitives”. Primitives can be incorporated into design descriptions directly.
3. Data Flow Level: All possible operations on signals and variables are represented in terms of assignments.
4. Behavioral Level: This level describes a system by concurrent algorithms and the design description looks like a C program. Compactness and the comprehensive nature of the design description make the development process fast and efficient. Functions, Tasks and always blocks are the main elements.

LANGUAGE CONSTRUCTS AND CONVENTIONS IN VERILOG

Verilog has its own constructs and conventions. Any source file in Verilog is made up of number of ASCII characters. These characters are grouped into sets - referred to as “lexical tokens”. Verilog has 7 types of lexical tokens - operators, keywords, identifiers, white spaces, comments, numbers and strings. Verilog is case sensitive, so all keywords are in lower case.

White Space Characters

Blanks(\b), tabs (\t), newlines (\n) and form feed form the white space characters in Verilog. In any design description the white space characters are included to improve readability. White space characters have significance only when they appear in string.

Comments

Comments are incorporated in two ways.

// begins a single line comment, terminated by a newline.

/* begins a multi-line block comment, terminated by a */.

For Example;

Module D_FF (Q,DP,CLK); //This is the design of a D flip-flop.

/* Here Q is the output.DP is the input and CLK is the clock.*/

Keywords

The keywords define the language constructs. All keywords in Verilog are in small letters and require to be used as such.

Some of the examples are:

module ---→ signifies the beginning of a module definition.

endmodule ---→ signifies the end of module definition.

begin signifies---→ the beginning of block of statements.

end -→ signifies the end of a block of statements.

if → signifies a conditional activity to be checked.

assign -→ assigns a value or an expression to a net or variable.

Identifiers

Identifiers are user-defined words for variables, function names, module names, block names and instance names. Identifiers begin with a letter or underscore and can include any number of letters, digits and underscores. Identifiers in Verilog are case sensitive.

Numbers

The numbers can be of integer type or real type.

Integer Numbers

Number storage is defined as a number of bits, but values can be specified in binary, octal, decimal or hexadecimal. The representation has three tokens with an optional sign preceding it. Numbers may be sized or unsized. Unsized integers default to at least 32 bits.

Syntax: size'base value

Examples:

Integer	stored As	Description
1	00000000000000000000000000000001	Unsize 32 bits
8'h AA	10101010	Sized hexadecimal
3'b001	001	3 bit number
9'o123	001 010 011	9 bit octal number
9'h?A	zzzzz1010	5 bit Hex number.

String

A string is a sequence of characters enclosed by double quotes. It must be contained on a single line. Verilog treats a string as a sequence of ASCII characters.

Logical Values

Verilog uses a 4 value logic system for modelling.

Logical Value	Description
0	Zero , low or false
1	One, high or true
Z or z	High impedance (tri-state or floating)
X or x	Unknown or uninitialized (don't care)

Data Types

Verilog has two major data type classes:

- 1) Net Data type
- 2) Variable data type.

Nets: Net data types are used to make connections between parts of a design. Nets reflect the value and strength level of the drivers of the net or the capacitance of the net, and do not have a value of their own. A net can be specified in different ways.

Wire

A wire (or net) represents a physical wire in a circuit and is used to connect gates or modules. The value of a wire can be read, but not assigned to, in a function or block. A wire does not store its value but must be driven by a continuous assignment statement or by connecting it to the output of a gate or module.

Syntax: wire [msb: lsb] wire_variable_list;

Example

```
wire c; //declare a wire c
```

Variable data types

Variable data types are used as temporary storage of programming data. Variables can only be assigned a value from within an initial procedure, an always procedure, a task or a function. Variables can only store logic values; they cannot store logic strength. Variables are un-initialized at the start of simulation, and will contain logic X until a value is assigned. Variables can be declared through a keyword reg.

Reg

A reg(register) is a data object that holds its value from one procedural assignment to the next. They are used only in functions and procedural blocks. A reg is a Verilog variable type and does not necessarily imply a physical register.

Syntax: reg [msb: lsb] reg_variable_list;

Example

```
reg a; // single 1-bit register variable
```

```
reg [7:0] r_vector; // an 8-bit vector; a bank of 8 registers.
```

Scalars and Vectors

Entities representing single bits are called “scalars”. Often multiple lines carry signals in a cluster like data bus or address bus. The group of regs is treated as “vector”.

Verilog Module

Verilog language uses a hierarchical, functional unit based design approach. The whole design consists of several smaller modules. The complexity of the modules is decided by the designer

Verilog module:

- Definition of the input and output ports
- Definition of the logical relationship between the input and output ports

Port declaration syntax:

```
<direction> <data_type> <size> <port_name>;
```

• Direction:

- Input port: input
- Output port: output
- Bi-directional: inout

Verilog language uses a hierarchical, functional unit based design approach:

- The whole design consists of several smaller modules
- The complexity of the modules is decided by the designer

Module Syntax:

```
module module_name (port_list);  
    input [msb:lsb] input_port_list;  
    output [msb:lsb] output_port_list;  
    inout [msb:lsb] inout_port_list;  
    ... statements ...  
endmodule
```

Operators

Arithmetic Operators

These perform arithmetic operations.

- + (addition)
- (subtraction)
- * (multiplication)
- / (division)
- %(modulus)

Relational Operators

Relational operators compare two operands and return a single bit 1 or 0. These operators synthesize into comparators.

- < (less than)
- <= (less than or equal to)
- > (greater than)
- >= (greater than or equal to)
- == (equal to)
- != (not equal to)

Bit-wise Operators

Bit-wise operators do a bit-by-bit comparison between two operands

- ~ (bitwise NOT)
- & (bitwise AND)
- | (bitwise OR)
- ^ (bitwise XOR)
- ~^ or ^~ (bitwise XNOR)

Logical Operators

Logical operators return a single bit 1 or 0. They are the same as bit-wise operators only for single bit operands. They can work on expressions, integers or groups of bits, and treat all values that are nonzero as “1”. Logical operators are typically used in conditional (if ... else) statements since they work with expressions.

! (logical NOT)

&& (logical AND)

|| (logical OR)

Concatenation Operator

The concatenation operator combines two or more operands to form a larger vector.

{ }(concatenation)

EXPERIMENT NO: 01

DATE:

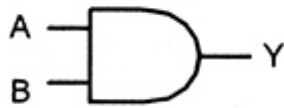
BASIC GATES/UNIVERSAL GATES

AIM: To write a verilog description for: a)AND Gate b)OR Gate c)NOT Gate
d)NAND Gate e)NOR Gate f)XOR Gate g)XNOR Gate

SOFTWARE REQUIRED: ModelSIM

LOGIC DIAGRAM and TRUTH TABLE:

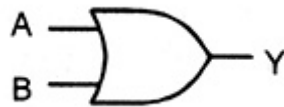
a) **AND Gate:**



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

$$Y = A \bullet B$$

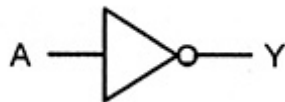
b) OR Gate:



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

$$Y = A + B$$

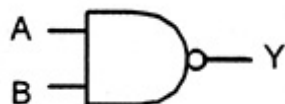
c) NOT Gate:



A	Y
0	1
1	0

$$Y = \bar{A}$$

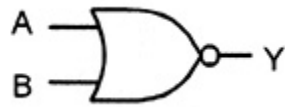
d) **NAND Gate:**



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

$$Y = \overline{A \cdot B}$$

e) NOR Gate:



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

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

f) XOR Gate:



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

$$Y = A \oplus B$$

g) XNOR Gate:



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

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

PROGRAM:

a) AND Gate:

```
module and_gate(Y,A,B);
input A,B;
output Y;
assign Y=A&B;
endmodule
```

b) OR Gate:

```
module or_gate(A,B,Y);
input A,B;
output Y;
assign Y=A|B;
endmodule
```

c) NOT Gate:

```
module not_gate(A,Y);
input A;
```



```

output Y;
assign Y=~A;
endmodule

```

d) **NAND Gate:**

```
module nand_gate(A,B,Y);
  input A,B;
  output Y;
  assign Y=~(A&B);
endmodule
```

e) NOR Gate:

```

module nor_gate(A,B,Y);
  input A,B;
  output Y;
  assign Y=~(A|B);
endmodule

```

f) XOR Gate:

```

module xor_gate(A,B,Y);
  input A,B;
  output Y;
  assign Y=A^B;
endmodule

```

g) XNOR Gate:

```

module xnor_gate(A,B,Y);
  input A,B;
  output Y;
  assign Y=A~^B;
endmodule

```

OUTPUT:

a) AND Gate:

/and_gate/A	S0
/and_gate/B	S1
/and_gate/Y	S0

◆ /or_gate/A	St1								
◆ /or_gate/B	St1								
◆ /or_gate/Y	St1								

◆ /not_gate/A	St1				
◆ /not_gate/Y	St0				

◆ /nand_gate/A	St1							
◆ /nand_gate/B	St0							
◆ /nand_gate/Y	St1							

◆ /nor_gate/A	St1								
◆ /nor_gate/B	St1								
◆ /nor_gate/Y	St0								

◆ /xor_gate/A	St1								
◆ /xor_gate/B	St1								
◆ /xor_gate/Y	St0								

	/xnor_gate/A	St1								
	/xnor_gate/B	St1								
	/xnor_gate/Y	St1								

Basic gates are realized using verilog and simulation completed successfully.

EXPERIMENT NO: 02 (A)**DATE:****COMBINATIONAL CIRCUITS-HALF ADDER/HALF SUBTRACTOR**

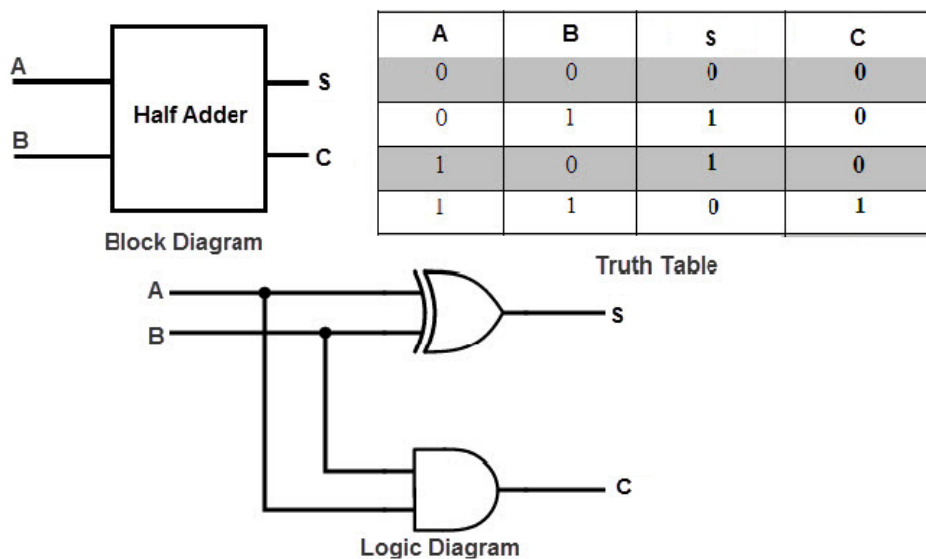
AIM: To write a verilog description for the combinational circuits: a)Half adder b)half subtractor.

SOFTWARE REQUIRED: ModelSIM

LOGIC DIAGRAM and TRUTH TABLE:

a) Half adder:

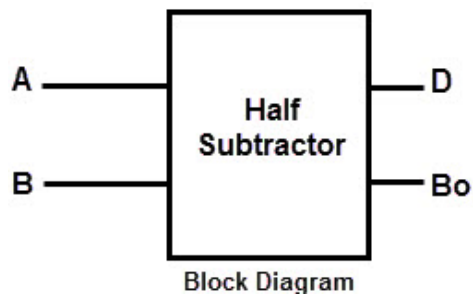
Half adder is a combinational arithmetic circuit that adds two numbers and produces a sum bit (S) and carry bit (C) as the output. If A and B are the input bits, then sum bit (S) is the X-OR of A and B and the carry bit (C) will be the AND of A and B. From this it is clear that a half adder circuit can be easily constructed using one X-OR gate and one AND gate. Half adder is the simplest of all adder circuit, but it has a major disadvantage. The half adder can add only two input bits (A and B) and has nothing to do with the carry if there is any in the input. So if the input to a half adder have a carry, then it will be neglected it and adds only the A and B bits. That means the binary addition process is not complete and that's why it is called a half adder.



$$S = A \oplus B \quad C = A \cdot B$$

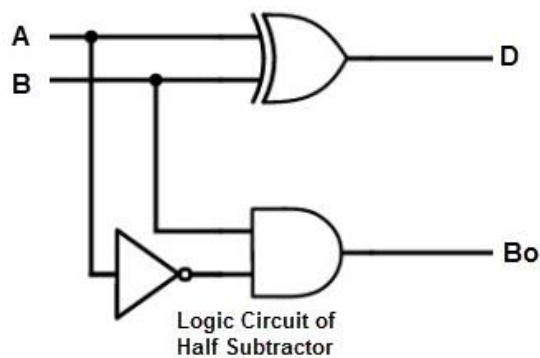
b) Half subtractor:

The half-subtractor is a combinational circuit which is used to perform subtraction of two bits. It has two inputs, X (minuend) and Y (subtrahend) and two outputs D (difference) and B (borrow).



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

Truth Table



$$D = A \oplus B$$

$$B_0 = \bar{A} B$$

PROGRAM:

a) Half adder:

```

module halfadder(A,B,S,C);
  input A,B;
  output S,C;
  assign S=A^B;
  assign C=A&B;
endmodule

```

b) Half subtractor:





```

module halfsubtractor(A,B,D,B0);
  input A,B;
  output D,B0;
  assign D=A^B;
  assign B0=~A&B;
endmodule





```

OUTPUT:

a) Half adder:

 /halfadder/A	St1								
 /halfadder/B	St1								
 /halfadder/S	St0								
 /halfadder/C	St1								

b) Half subtractor:

 /halfsubtractor/A	St1								
 /halfsubtractor/B	St1								
 /halfsubtractor/D	St0								
 /halfsubtractor/B0	St0								

RESULT:

Half adder and half subtractor are realized using verilog and simulation completed successfully.

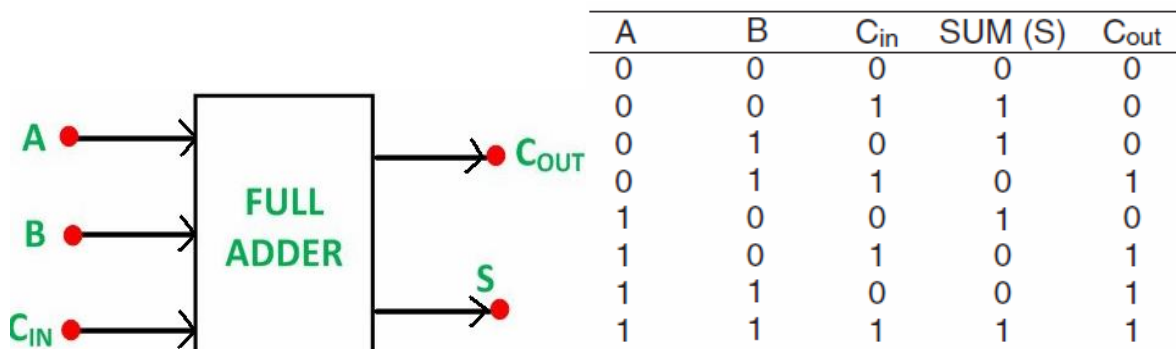
(B) FULL ADDER IN 3 MODELLING STYLES

AIM: Write a verilog description for full adder in: a)Dataflow modeling b)Structural modeling c)Behavioural modeling.

SOFTWARE REQUIRED: ModelSIM

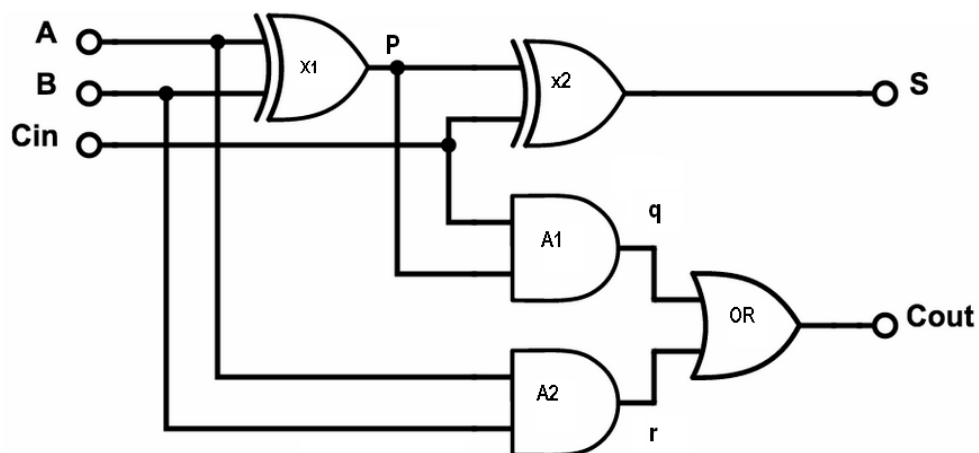
LOGIC DIAGRAM and TRUTH TABLE:

Full adder is a logic circuit that adds two input operand bits plus a Carry in bit and outputs a Carry out bit and a sum bit.. The Sum out (Sout) of a full adder is the XOR of input operand bits A, B and the Carry in (Cin) bit.



Block Diagram

Truth Table



Logic Diagram

$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = A \cdot B + A \cdot C_{in} + B \cdot C_{in}$$

PROGRAM:

a) Dataflow modeling:

```
module fulladder_dataflow(A,B,Cin,S,Cout);
    output S,Cout;
    input A,B,Cin;
```



```

assign S=A^B^Cin;
assign Cout=(A&B)|((A^B)&Cin);
endmodule

```

b) Structural modeling:

```

module fulladder_struct(A,B,Cin,S,Cout);
output S,Cout;
input A,B,Cin;
wire p,q,r;
xor x1(p,A,B);
xor x2(S,p,Cin);
and A1(q,p,Cin);
and A2(r,A,B);
or OR(Cout,q,r);
endmodule

```

c) Behavioural modeling:

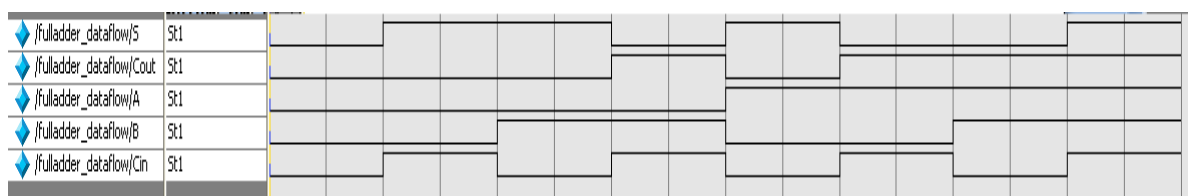
```

module fulladder_behavioural(A,B,Cin,S,Cout);
output reg S,Cout;
input A,B,Cin;
always@(A,B,Cin)
begin
    {Cout,S}=A+B+Cin;
end
endmodule

```

OUTPUT:

a) Dataflow modelling:



b) Structural modelling:

c) Behavioural modelling:

RESULT:

70

EXPERIMENT NO: 03**DATE:****MULTIPLEXER /De-MULTIPLEXER**

AIM: Write a verilog description for: a) Multiplexer b) De-Multiplexer using dataflow modelling.

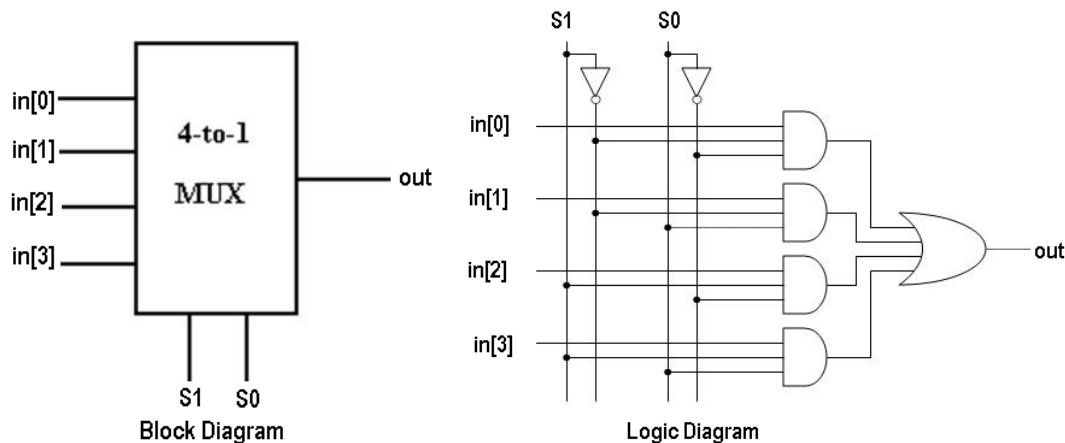
SOFTWARE REQUIRED: ModelSIM

LOGIC DIAGRAM and TRUTH TABLE:

a) Multiplexer:

Multiplexer means many into one. A multiplexer is a circuit used to select and route any one of the several input signals to a signal output. An simple example of an non electronic circuit of a multiplexer is a single pole multiposition switch. A multiplexer is a circuit that accept many input but give only one output.

A multiplexer has 2^n data inputs, n control inputs and 1 output.

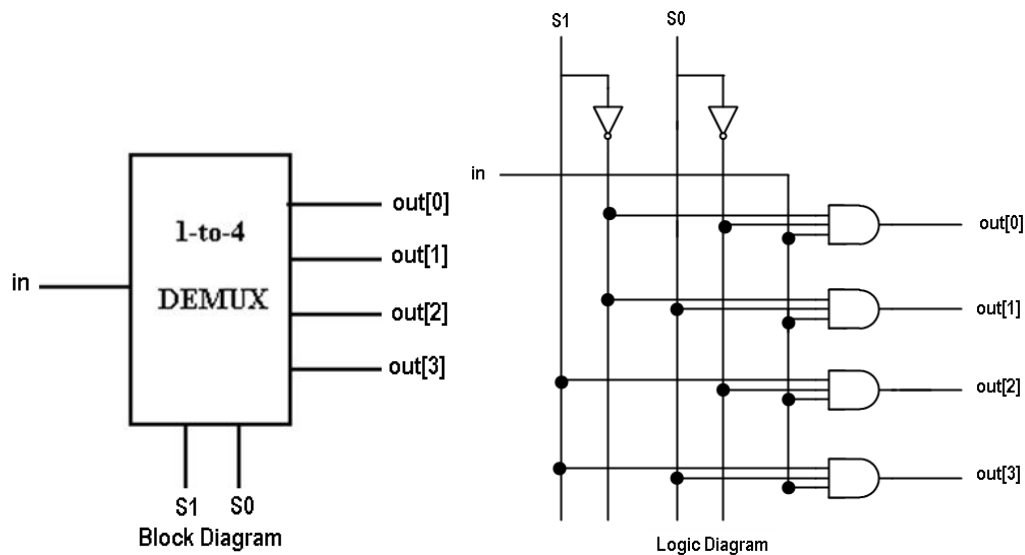


S1	S0	out
0	0	in[0]
0	1	in[1]
1	0	in[2]
1	1	in[3]

Truth Table

b) De-Multiplexer:

De-multiplexer is also a device with one input and multiple output lines. It is used to send a signal to one of the many devices. A de-multiplexer has 2^n data outputs, n control inputs and 1 input.



in	S1	S0	out[0]	out [1]	out [2]	out [3]
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

Truth Table**PROGRAM:**

a) Multiplexer:

```

module mux4_1(in,s0,s1,out);
    input s0,s1;
    input [3:0]in;
    output out;
    assign out=(~s1&~s0&in[0])|(~s1&s0&in[1])|(s1&~s0&in[2])|(s1&s0&in[0]);
endmodule

```

b) De-Multiplexer:









```

module demux1_4(in,s0,s1,out);
    input s0,s1,in;
    output [3:0]out;
    assign out[0]=~s1&~s0&in;
    assign out[1]=~s1&s0&in;
    assign out[2]=s1&~s0&in;
    assign out[3]=s1&s0&in;
endmodule










```


OUTPUT:

a) Multiplexer:

 /mux4_1/s0	St1								
 /mux4_1/s1	St1								
  /mux4_1/in	1011	1011							
 [3]	St1								
 [2]	St0								
 [1]	St1								
 [0]	St1								
 /mux4_1/out	St1								

b) De-Multiplexer:

 /demux1_4/s0	St1								
 /demux1_4/s1	St1								
 /demux1_4/in	St1								
  /demux1_4/out	1000	0001	0010	0100	1000				
 [3]	St1								
 [2]	St0								
 [1]	St0								
 [0]	St0								

RESULT:

Verilog code for 4:1 multiplexer and 1:4 demultiplexer were written using dataflow modelling and simulated successfully.

EXPERIMENT NO: 04**DATE:****FLIPFLOPS(SR,JK,T,D)**

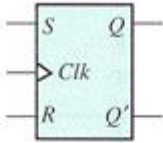
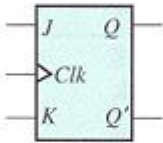
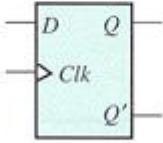
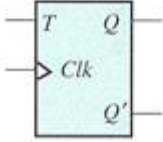
AIM: Write a verilog description for: a)SR b)JK c)T d)D flipflops using behavioral modelling.

SOFTWARE REQUIRED: ModelSIM

LOGIC DIAGRAM and TRUTH TABLE:

In [electronics](#), a flip-flop or latch is a [circuit](#) that has two stable states and can be used to store state information. A flip-flop is a [bistable multivibrator](#). The circuit can be made to change state by signals applied to one or more control inputs and will have one or two outputs. It is the basic storage element in [sequential logic](#). Flip-flops and latches are fundamental building blocks of [digital electronics](#) systems used in computers, communications, and many other types of systems.

A flip-flop is a special type of gated latch. The difference between a flip-flop and a gated latch is that in a flip-flop, the inputs aren't enabled merely by the presence of a HIGH signal on the CLOCK input. Instead, the inputs are enabled by the transition of the CLOCK input. Thus, at the moment that the clock input transitions from low to high, the inputs are briefly enabled. Once the clock stabilizes at the HIGH setting, the output state of the flip-flop is latched until the next clock pulse.

FLIP-FLOP NAME	FLIP-FLOP SYMBOL	CHARACTERISTIC TABLE	CHARACTERISTIC EQUATION	EXCITATION TABLE																																			
SR		<table><tr><th>S</th><th>R</th><th>Q(next)</th></tr><tr><td>0</td><td>0</td><td>Q</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>NA</td></tr></table>	S	R	Q(next)	0	0	Q	0	1	0	1	0	1	1	1	NA	$Q(next) = S + R'Q$ $SR = 0$	<table><tr><th>Q</th><th>Q(next)</th><th>S</th><th>R</th></tr><tr><td>0</td><td>0</td><td>0</td><td>X</td></tr><tr><td>0</td><td>1</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>X</td><td>0</td></tr></table>	Q	Q(next)	S	R	0	0	0	X	0	1	1	0	1	0	0	1	1	1	X	0
S	R	Q(next)																																					
0	0	Q																																					
0	1	0																																					
1	0	1																																					
1	1	NA																																					
Q	Q(next)	S	R																																				
0	0	0	X																																				
0	1	1	0																																				
1	0	0	1																																				
1	1	X	0																																				
JK		<table><tr><th>J</th><th>K</th><th>Q(next)</th></tr><tr><td>0</td><td>0</td><td>Q</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>Q'</td></tr></table>	J	K	Q(next)	0	0	Q	0	1	0	1	0	1	1	1	Q'	$Q(next) = JQ' + K'Q$	<table><tr><th>Q</th><th>Q(next)</th><th>J</th><th>K</th></tr><tr><td>0</td><td>0</td><td>0</td><td>X</td></tr><tr><td>0</td><td>1</td><td>1</td><td>X</td></tr><tr><td>1</td><td>0</td><td>X</td><td>1</td></tr><tr><td>1</td><td>1</td><td>X</td><td>0</td></tr></table>	Q	Q(next)	J	K	0	0	0	X	0	1	1	X	1	0	X	1	1	1	X	0
J	K	Q(next)																																					
0	0	Q																																					
0	1	0																																					
1	0	1																																					
1	1	Q'																																					
Q	Q(next)	J	K																																				
0	0	0	X																																				
0	1	1	X																																				
1	0	X	1																																				
1	1	X	0																																				
D		<table><tr><th>D</th><th>Q(next)</th></tr><tr><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td></tr></table>	D	Q(next)	0	0	1	1	$Q(next) = D$	<table><tr><th>Q</th><th>Q(next)</th><th>D</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	Q	Q(next)	D	0	0	0	0	1	1	1	0	0	1	1	1														
D	Q(next)																																						
0	0																																						
1	1																																						
Q	Q(next)	D																																					
0	0	0																																					
0	1	1																																					
1	0	0																																					
1	1	1																																					
T		<table><tr><th>T</th><th>Q(next)</th></tr><tr><td>0</td><td>Q</td></tr><tr><td>1</td><td>Q'</td></tr></table>	T	Q(next)	0	Q	1	Q'	$Q(next) = TQ' + T'Q$	<table><tr><th>Q</th><th>Q(next)</th><th>T</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	Q	Q(next)	T	0	0	0	0	1	1	1	0	1	1	1	0														
T	Q(next)																																						
0	Q																																						
1	Q'																																						
Q	Q(next)	T																																					
0	0	0																																					
0	1	1																																					
1	0	1																																					
1	1	0																																					

PROGRAMME:

a) SR flipflop:

```

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; q1=1'bx; end
    endcase
end
endmodule

```

b) JK flipflop:

```

module jkff(q,q1,j,k,clk);
    output q,q1;
    input j,k,clk;
    reg q,q1;
    initial
    begin
        q=1'b0;
        q1=1'b1;
    end
    always @(posedge clk)
    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

```

c) T flipflop:

```

module tff(q,q1,t,clk);
    output q,q1;
    input t,clk;

```



```
reg q,q1;
initial
    begin
        q=1'b0;
        q1=1'b1;
    end
always@(posedge clk)
begin
    if(t==0)
    begin
        q<=q;
        q1<=q1;
    end
    else
    begin
        q<=~q;
        q1<=~q1;
    end
end
endmodule
```

d) D flipflop:

```
module dff(q,q1,d,clk);
output q,q1;
input d,clk;
reg q,q1;
initial
    begin
        q=1'b0;
        q1=1'b1;
    end
always@(posedge clk)
begin
    q<=d;
```








```

q1<=~d;
end
endmodule

```

OUTPUT:

























a) SR flipflop

 /srff/q	x								
 /srff/q1	x								
 /srff/r	St1								
 /srff/s	St1								
 /srff/clk	St0								









b) JK flipflop:

/jkff/q	0
/jkff/q1	1
/jkff/j	St1
/jkff/k	St1
/jkff/clk	St1

c) T flipflop:

 /tff/q	0							
 /tff/q1	1							
/tff/t	St0							
 /tff/clk	St0							

d) D flipflop:

 /dff/q	1	
 /dff/q1	0	
 /dff/d	St1	
 /dff/clk	St0	

RESULT:

Verilog description for SR, JK,T and D flipflops were written using behavioral modeling and simulated successfully.

EXPERIMENT NO: 05**DATE:****BINARY COUNTERS**

AIM: Write a verilog description for binary counters : a) synchronous 4-bit up counter b) asynchronous decade counter c) ring counter d) Johnson counter using behavioral modelling.

SOFTWARE REQUIRED: ModelSIM

LOGIC DIAGRAM and TRUTH TABLE:

A counter is a [sequential circuit](#) that counts in a cyclic sequence. It is essentially a [register](#) that goes through a predetermined sequence of states upon the application of input pulses. There are two types of counters – Synchronous Counter & Asynchronous Counter.

Synchronous Counter

In a synchronous counter, the input pulses are applied to all clock pulse inputs of all flip flops simultaneously (directly). Synchronous counter is also known as parallel sequential circuit. Examples of Synchronous Counters are as below:

- [Ring Counter](#)
- [Johnson Counter \(Switch Tail or Twisted Ring Counter\)](#)

Asynchronous Counter

In an asynchronous counter, the [flip flop](#) output transition serves as a source for triggering other flip flops. In other words, the clock pulse inputs of all flip flops, except the first, are triggered not by the incoming pulses, but rather by the transition that occurs in previous flip flop's output.. Asynchronous counter is also known as serial sequential circuit. Example of Asynchronous Counters are as below:

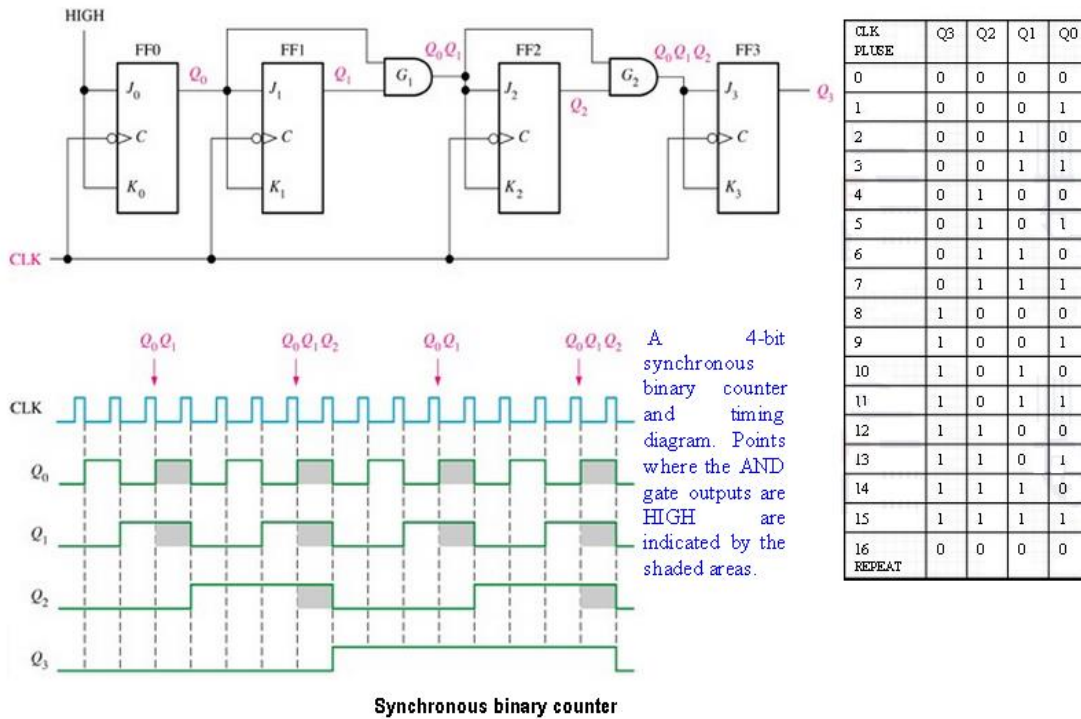
- [Binary Ripple Counter](#)
- [Up Down Counter](#)

Synchronous counters are faster than asynchronous counter because in synchronous counter all flip flops are clocked simultaneously.

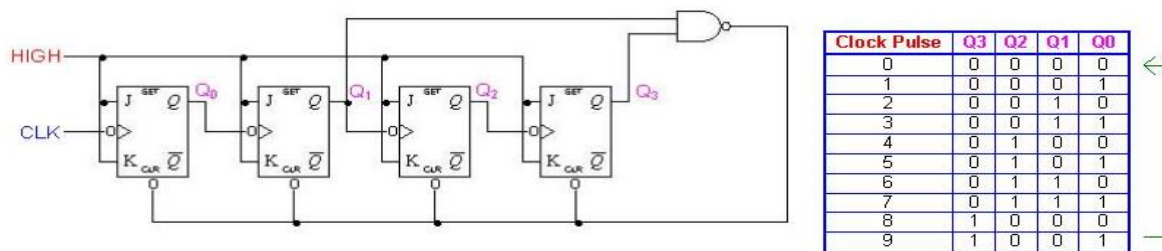
In synchronous counters, the clock input is connected to all of the flip-flop so that they are clocked simultaneously. An asynchronous counter is one in which the flip-flop

within the counter do not change states at exactly the same time because they do not have a common clock pulse.

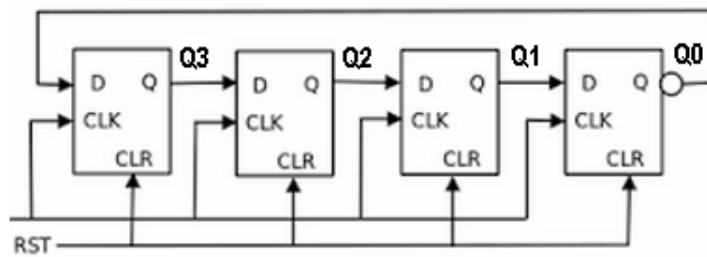
a) Synchronous 4-bit upcounter:



b) Asynchronous decade counter:



c) Ring counter:

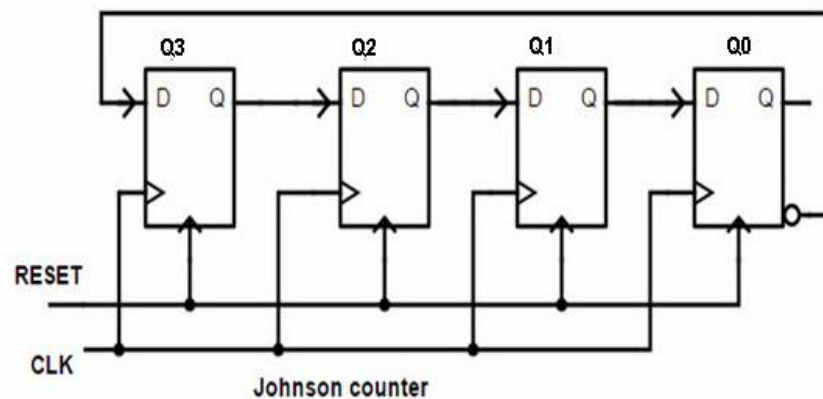


Ring counter

Q3	Q2	Q1	Q0
1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1
1	0	0	0
0	1	0	0
0	0	1	0

d) Johnson counter:

Q3	Q2	Q1	Q0
0	0	0	0
1	0	0	0
1	1	0	0
1	1	1	0
1	1	1	1
0	1	1	1
0	0	1	1
0	0	0	1
repeat			



Johnson counter

PROGRAMME:

a) Synchronous 4-bit upcounter:

```

module upcount_4bit(q,clk,res);
    input clk,res;
    output [3:0]q;
    reg [3:0]q;
    always@(posedge clk)
    begin
        if(res==1)
            begin
                q<=4'b0000;
            end
        else
    
```



```
begin
    q<=q+1;
end
end
endmodule
```

b) Asynchronous decade counter:

```
module decadecounter(q,clk,res);
    input clk,res;
    output [3:0]q;
    reg [3:0]q;
    always@(posedge clk)
    begin
        if(res==1)
            begin
                q<=4'b0000;
            end
        else if(q>=4'b1001)
            begin
                q<=4'b0000;
            end
        else
            begin
                q<=q+4'b0001;
            end
        end
    end
endmodule
```

c) Ring counter:

```
module ringcounter(q,clk,res);
    input clk,res;
    output [3:0]q;
    reg [3:0]q;
    always@(posedge clk)
    begin
```



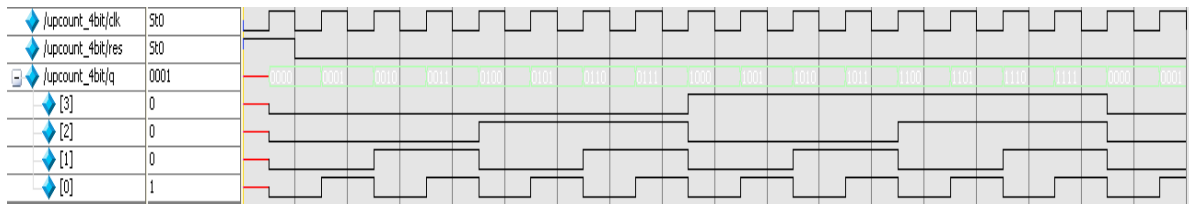
```
        if(res==1)
            begin
                q<=4'b1000;
            end
        else
            begin
                q[3]<=q[0];
                q[2]<=q[3];
                q[1]<=q[2];
                q[0]<=q[1];
            end
        end
    endmodule
```

d) Johnson counter:

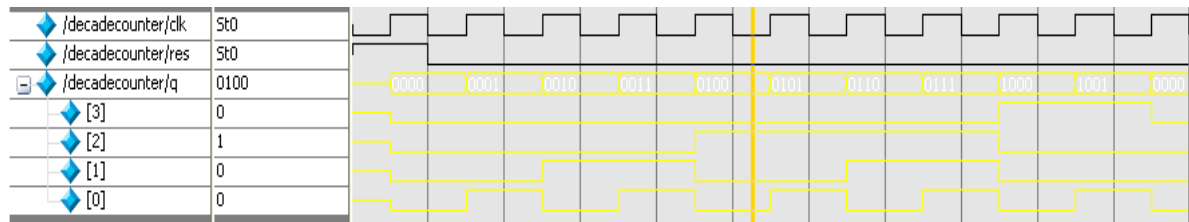
```
module johnsoncounter(q,clk,res);
    input clk,res;
    output [3:0]q;
    reg [3:0]q;
    always@(posedge clk)
    begin
        if(res==1)
            begin
                q<=4'b0000;
            end
        else
            begin
                q[3]<=~q[0];
                q[2]<=q[3];
                q[1]<=q[2];
                q[0]<=q[1];
            end
        end
    end
endmodule
```


OUTPUT:

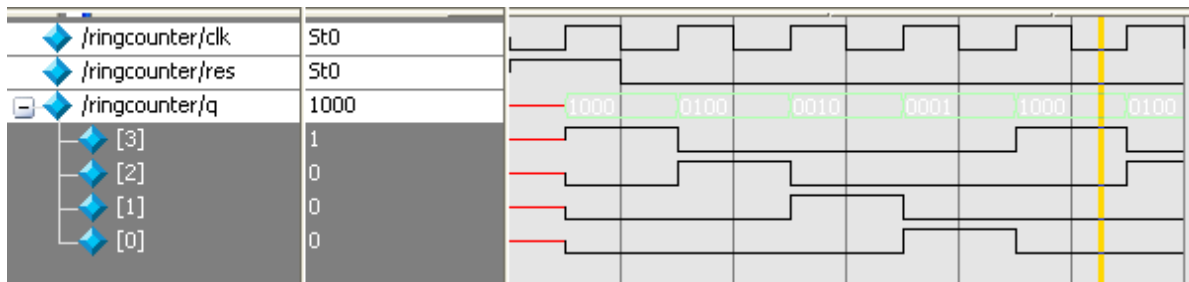
a) Synchronous 4-bit upcounter:



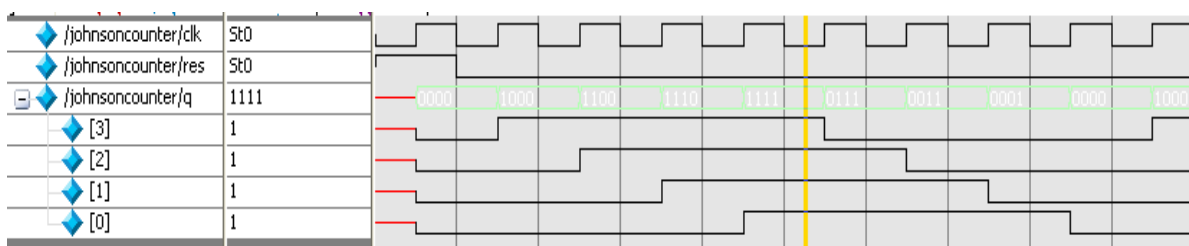
b) Asynchronous decade counter:



c) Ring counter:



e) Johnson counter:

**RESULT:**

Verilog description for synchronous 4-bit up counter, asynchronous decade counter, ring counter and Johnson counter were written using behavioral modeling and simulated successfully.