APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

KTU SPECIAL

We can together dream the B.tech

APJ ABDUL KALAM
TECHNOLOGICAL UNIVERSITY
സാങ്കേതിക ശാസ്ത്ര സർവകലാശാല

• KTU STUDY MATERIALS	• SYLLABUS
• KTU LIVE NOTIFICATION	• SOLVED QUESTION PAPERS

JOIN WITH US

WWW.KTUSPECIAL.IN KTUSPECIAL t.me/ktuspecial1

SYLLABUS

Module 1: Introduction to digital Systems ;-Digital abstraction: Number Systems – Binary, Hexadecimal, grouping bits, Base conversion; Binary Arithmetic – Addition and subtraction, Unsigned and Signed numbers; Fixed-Point Number Systems; Floating-Point Number Systems Basic gates- Operation of a Logic circuit; Buffer; Gates - Inverter, AND gate, OR gate, NOR gate, NAND gate, XOR gate, XNOR gate; Digital circuit operation - logic levels, output dc specifications, input dc specifications, noise margins, power supplies; Driving loads - driving other gates, resistive loads and LEDs.

Module 2: Combinational Logic Design: –Boolean Algebra - Operations, Axioms, Theorems; Combinational logic analysis - Canonical SOP and POS, Minterm and Maxterm equivalence; Logic minimization - Algebraic minimization, K-map minimization, Dont cares, Code convertors.
Modeling concurrent functionality in Verilog:- Continuous assignment - Continuous Assignment with logical operators, Continuous assignment with conditional operators, Continuous assignment with delay

Module 3: MSI Logic and Digital Building Blocks
MSI logic - Decoders (One-Hot decoder, 7 segment display decoder), Encoders, Multiplexers, Demultiplexers; Digital Building Blocks - Arithmetic Circuits - Half adder, Full adder, half subtractor, full subtractor;Comparators.
Structural design and hierarchy - lower level module instantiation, gate level primitives, user defined primitives, adding delay to primitives.

Module 4: Sequential Logic Design :- Latches and Flip-Flops- SR latch, SR latch with enable, JK flipflop, D flipflop, Register Enabled Flip-Flop, Resettable Flip-Flop. Sequential logic timing considerations; Common circuits based on sequential storage devices - toggle flop clock divider, asynchronous ripple couter, shift register. **Finite State Machines** :-Finite State Machines - logic synthesis for an FSM,FSM design process and design examples; Synchronous Sequential Circuits -Counters;**Verilog (Part 2) :** -Procedural assignment; Conditional Programming constructs; Test benches; Modeling a D flipflop in Verilog; Modeling an FSM in Verilog.

MODULE 4

Sequential Logic Design :- Latches and Flip-Flops- SR latch, SR latch with enable, JK flipflop, D flipflop, Register Enabled Flip-Flop, Resettable Flip-Flop. Sequential logic timing considerations; Common circuits based on sequential storage devices - toggle flop clock divider, asynchronous ripple counter, shift register.

Finite State Machines :-Finite State Machines - logic synthesis for an FSM, FSM design process and design examples; Synchronous Sequential Circuits - Counters;

Verilog (Part 2) : -

Procedural assignment; Conditional Programming constructs; Test benches; Modeling a D flipflop in Verilog; Modeling an FSM in Verilog.

Sequential Logic Design: Latches and Flip-Flops

- Sequential logic circuits are a class of digital circuits whose output depends not only on the current input but also on the past history of inputs (i.e., their previous states).
- This means they have memory, unlike combinational logic circuits. Latches and flip-flops are basic components used to implement sequential =logic.

FLIP FLOPS

- The flip-flop is a circuit that maintains a state until directed by input to change the state.
- A basic flip-flop can be constructed using four-NAND or four-NOR gates. Flip-flop is popularly known as the basic digital memory circuit. It has its two states as logic 1(High) and logic 0(low) states.
- A flip flop is a sequential circuit which consist of single binary state of information or data.
- The digital circuit is a flip flop which has two outputs and are of opposite

states. It is also known as a Bistable Multivibrator.

Latches

A latch is a simple memory device that can store one bit of data. It is level-sensitive, meaning it responds to input signals as long as a control signal (usually called Enable or Clock) is active.

SR Latch

- S-R latches i.e., Set-Reset latches are the simplest form of latches and are implemented using two inputs: S (Set) and R (Reset).
- The S input sets the output to 1, while the R input resets the output to 0.
- When both S and R inputs are at 1, the latch is said to be in an "undefined" state. They are also known as preset and clear states.
- The SR latch forms the basic building blocks of all other types of flip-flops.

Truth Table of SR Latch

S	R	Q	Q'
0	0	Latch	Latch

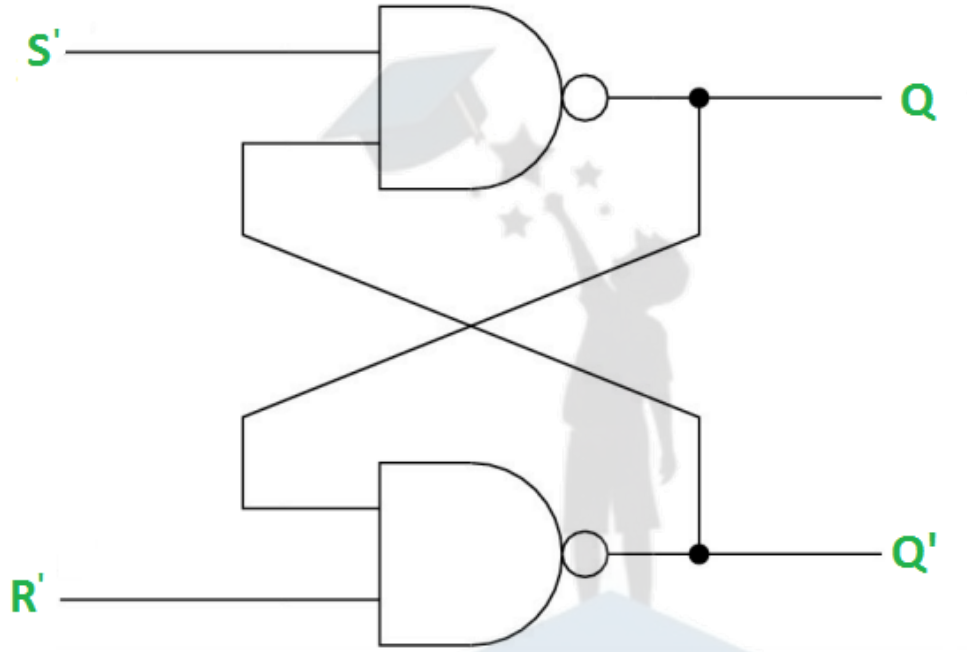
0	1	0	1
1	0	1	0
1	1	0	0

Logic Diagram of SR Latch

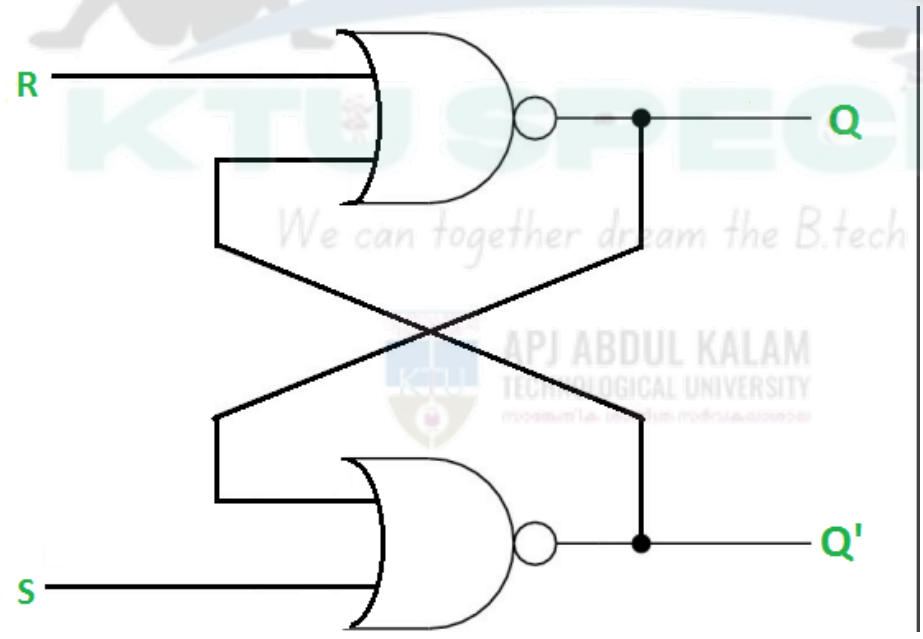
SR Latch is a logic circuit with:

- 2 cross-coupled NOR gate or 2 cross-coupled NAND gate.
- 2 input S for SET and R for RESET
- 2 output Q, Q'.

The below logic diagram represents the SR latch using NAND gate.



The below logic diagram represents SR latch using NOR Gate.



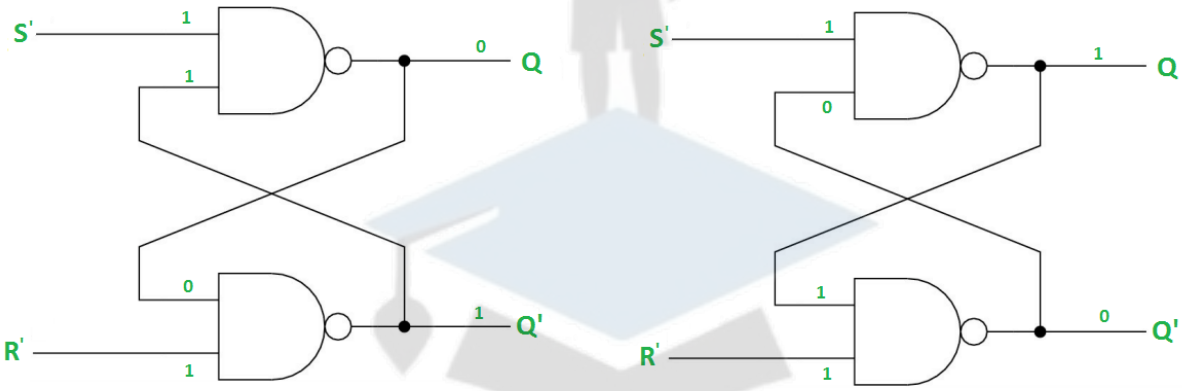
Different Cases of SR Latch

The different cases of SR latch are discussed below.

Case 1: S' = R' = 1 (S = R = 0)

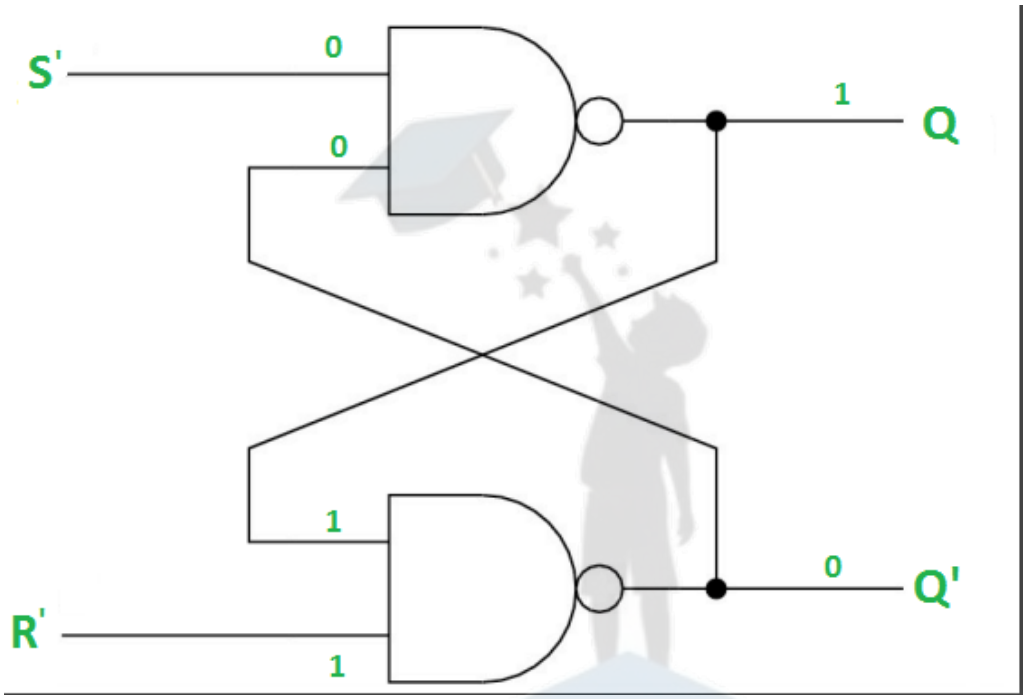
If Q = 1, Q and R' inputs for 2nd NAND gate are both 1.

If Q = 0, Q and R' inputs for 2nd NAND gate are 0 and 1 respectively



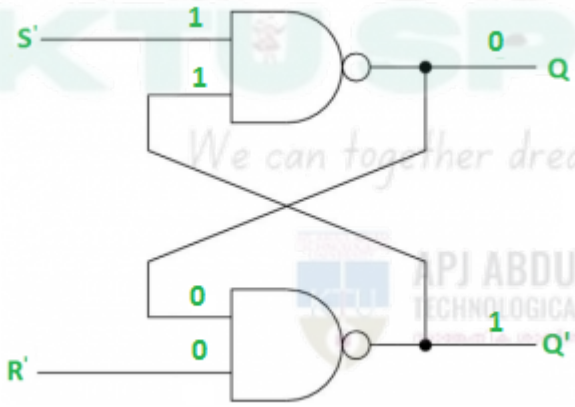
Case 2: S' = 0, R' = 1 (S = 1, R = 0)

- As S' = 0, the output of 1st NAND gate, Q = 1 (SET state).
- In second NAND gate, as Q and R' inputs are 1, Q'=0.



Case 3: S' = 1, R' = 0 (S = 0, R = 1)

- As R'=0, the output of 2nd NAND gate, Q' = 1.
- In first NAND gate, as Q and S' inputs are 1, Q = 0 (**RESET state**)



Case 4: S' = R' = 0 (S = R = 1)

When S = R = 1, both Q and Q' becomes 1 which is not allowed. So, the input condition is prohibited.

SR Latch with Enable

- The SR Latch with Enable (also known as an SR Latch with Clock/Control) is an enhancement of the basic SR latch where an additional Enable (or Clock) input is added.
- The Enable input controls whether the SR latch can change its state or not. When Enable is active (usually Enable = 1), the latch behaves like a normal SR latch.
- When Enable is inactive (usually Enable = 0), the latch holds its state, and the inputs S and R are ignored**.**

- **When Enable = 0 (inactive):**
    - The latch holds its previous state (i.e., ignores changes to S and R).
- When Enable = 1 (active):
    - The latch behaves like a normal SR latch and responds to S and R inputs.

SR Latch with Enable Truth Table

Enable (E)	S (Set)	R (Reset)	Q	Q'
0	0	0	Q	Q'
0	1	0	Q	Q'
0	0	1	Q	Q'
1	0	0	Q	Q'
1	1	0	1	0
1	0	1	0	1
1	1	1	Undefined	Undefined

Enable = 0 (Inactive):

- When Enable is low (0), the latch holds its current state. Regardless of the values of S and R, the output Q and Q' remain unchanged. This is because the latch is disabled from updating its state when Enable is 0.

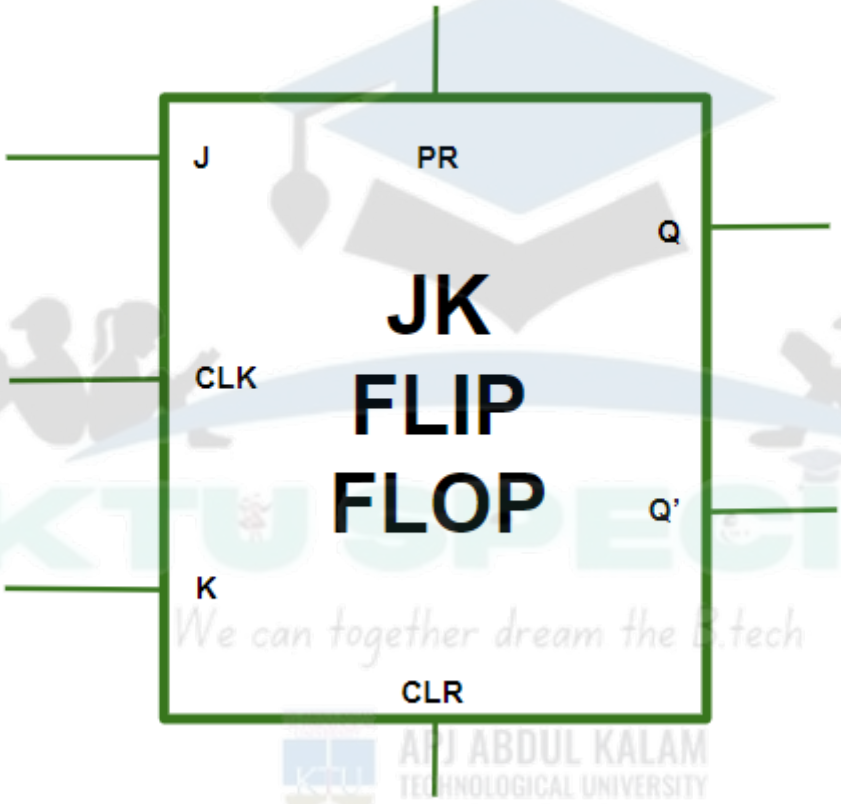
Enable = 1 (Active):

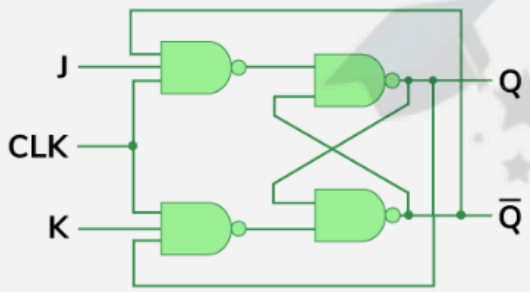
- When Enable is high (1), the latch is active and can respond to the inputs S and R:
    - S = 1, R = 0: The latch sets the output Q to 1 and Q' to 0.
    - S = 0, R = 1: The latch resets the output Q to 0 and Q' to 1.
    - S = 0, R = 0: The output Q and Q' retain their previous state (no change).
    - S = 1, R = 1: This is an invalid state for an SR latch because both outputs would be 0, which violates the principle that Q and Q'

should always be complements of each other. It leads to an undefined state.

JK flipflop

- It is one kind of sequential logic circuit which stores binary information in bitwise manner.
- It consists of two inputs and two outputs.
- Inputs are Set(J) & Reset(K) and their corresponding outputs are Q and Q'.
- JK flipflop has two modes of operation which are synchronous mode and asynchronous mode.
- In synchronous mode, the state will be changed with the clock(clk) signal.
- In asynchronous mode, the change of state is independent from its clock signal.





Truth Table

J	K	Q_N	Q_{N+1}
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Two 3-input NAND gates are used in place of the original two 2-input AND gates. The outputs at Q and Q' are coupled to each gate's third input. Since the two inputs are now interlocked, the SR flip-flop's cross-coupling enables the previously invalid condition of (S = "1", R = "1") to be employed to perform the "toggle action".

In a circuit "set", the bottom NAND gate interrupts the J input coming from the "0" position of Q'. In the "RESET" state, the top NAND gate interrupts the K input coming from the 0 positions of Q. We can use Q and Q' to control the input because they are always different. The flip flop is toggled according to the truth table when both inputs "J" and "K" are set to 1.

1. J = 0, K = 0 (No Change): Here, the output Q(n+1) stays the same. This means the next state (Q(n+1)) is just like the current one (Qn).
2. J = 0, K = 1 (Reset State): In this case, the next state is reset to 0 (Q(n+1) = 0), no matter what the current state is (Qn).
3. J = 1, K = 0 (Set State): Now, the next state gets set to 1 (Q(n+1) = 1), again, no matter what's happening in the current state (Qn).
4. J = 1, K = 1 (Toggle State): In this state, the output Q(n+1) toggles. So if the current state is set (Qn = 1), it will change to 0 (Q(n+1) = 0). If

it’s reset (Qn = 0), then it flips to 1 (Q(n+1) = 1).

Excitation Table

The excitation table shows us which input combinations we should use for a JK flip-flop to get the output we want.

- X - Don't Care
- Qn - Current State
- Q(n+1) - Next State
- J and K - Two input values

Here, X for Don't Care. This means it can be either 0 or 1, & it won't change how the flip-flop works.

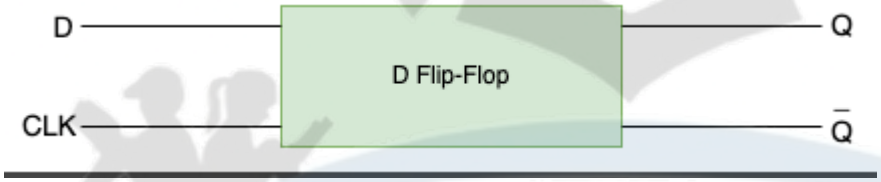
INPUT		OUTPUT	
Qn	Q(n+1)	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

This table guides us on what input values to use so we can move from the current (Qn) to the next state (Q(n+1)).

D Flip Flop

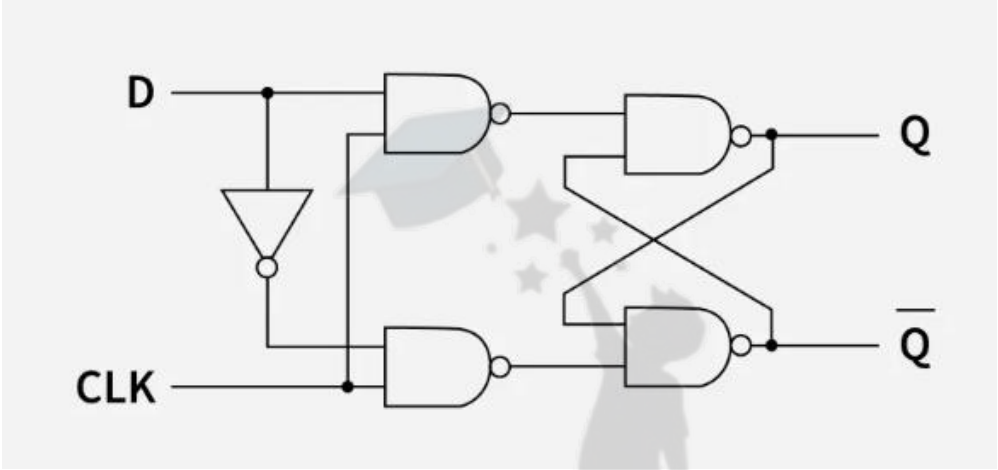
D flip flop is an electronic devices that is known as "delay flip flop" or "data flip flop" which is used to store single bit of data.

- D flip flops can be either synchronous or asynchronous.
- The clock signal is required for the synchronous version of D flip flops but not for the asynchronous one.
- The D flip flop has two inputs, data and clock input which controls the flip flop.
- When clock input is high, the data is transferred to the output of the flip flop.
- When the clock input is low, the output of the flip flop is held in its previous state.



- A D flip-flop is created by modifying an SR flip-flop.
- The S input is connected to the D input, and the R input is connected to the inverted D input.
- As a result, a D flip-flop functions similarly to an SR flip-flop, but with complementary inputs, preventing any possibility of an invalid intermediate state.
- One major issue with the SR flip-flop is the race around condition, which is eliminated in the D flip-flop due to the inverted inputs.

The circuit diagram of the D flip-flop is shown in the figure below:



Working of D Flip Flop

D flip flop consist of a single input D and two outputs (Q and Q'). The basic working of D Flip Flop is as follows:

- When the clock signal is low, the flip flop holds its current state and ignores the D input.
- When the clock signal is high, the flip flop samples and stores D input.
- The value that was previously fed into the D input is reflected at the flip flop's Q output.
    - If D = 0 then Q will be 0.
    - If D = 1 then Q will be 1.
- The Q' output of the flip flop is complemented by the Q output.
    - If Q = 0 then Q' will be 1.
    - If Q = 1 then Q' will be 0.

Characteristic Table of D Flip Flop

D	Q(Current)	Q(n+1) (Next)
0	0	0
0	1	0
1	0	1
1	1	1

- D is the input, and Q is current state, Q(n+1) is the next state outputs.
- Q(n+1) will always be zero when D is 0, irrespective of current state of flip flop.
- When the input of the flip flop is 1, next state of flip flop will always be 1, regardless of the current state of flip flop.

Advantages of D Flip Flop

- D flip flop is very simple to design.
- The computation speed of D flip flop is very fast compared to other flip flops.
- D flip flop requires very few components to design which makes it simple to understand.

Register enabled flip-flop

An **enabled flip-flop** is a type of sequential logic circuit that functions as a storage element, similar to a regular flip-flop, but with an additional control input, often called an **Enable (EN)** signal. The flip-flop changes its state (or stores data) based on the combination of the enable signal and the clock input.

- **Enable (EN) Input**: The flip-flop only changes its state when the Enable input is active (usually high or 1).
- **Clock (CLK) Input**: The clock still plays a role in determining when the flip-flop can update its state, but it is only active when the Enable input is active.
- **D or T Flip-Flop**: An enabled flip-flop can be implemented with different types of flip-flops like **D-type** or **T-type**, but the enabling mechanism typically affects when the flip-flop responds to the clock signal.

Example of Behavior:

- **When EN = 1**:
  - The flip-flop operates normally (i.e., on the rising or falling edge of the clock, it updates or stores data).
- **When EN = 0**:
  - The flip-flop ignores the clock input and does not change its state, regardless of the clock signal.

Example Circuit:

Let's take a **D-type enabled flip-flop** as an example:

- **Inputs**:
  - **D (Data)**: The data input.
  - **EN (Enable)**: The enable signal.
  - **CLK (Clock)**: The clock input.
- **Outputs**:
  - **Q**: The output.
  - **Q'**: The complementary output.

The behavior of a D-type enabled flip-flop:

- When **EN = 0**, the output **Q** holds its current value regardless of the clock or the data input.
- When **EN = 1**, the flip-flop behaves like a standard D flip-flop, updating its output **Q** with the value of the data input **D** on the clock edge.

EN CLK D Q (Next State)

0 ↑ X Q (holds value)

1 ↑ 0 0

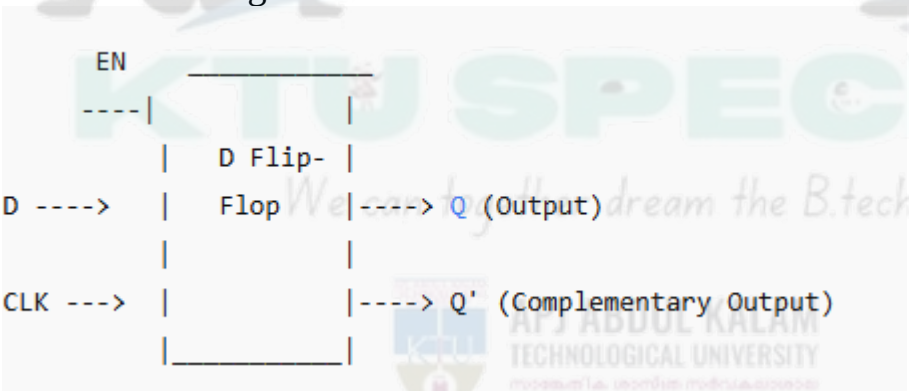
1 ↑ 1 1

EN	CLK	D	Q (next state)
0	↑	X	Q (holds value)
1	↑	0	0
1	↑	1	1

Applications:

Enabled flip-flops are useful in designs where you need to control when a flip-flop stores or updates its data, such as in:

- **Shift Registers**: Controlling when data should be shifted in or out.
- **Memory Elements**: Selectively enabling certain memory cells in a larger storage system.
- **Timing Control**: Managing when state changes should occur based on control logic.



Resettable flip flop

A **resettable flip-flop** is a type of flip-flop that has an additional input (often labeled **RESET** or **R**) which, when activated, forces the flip-flop to set its output to a predefined state (usually **0**). This is useful in circuits where you

need to reset the state of a flip-flop to a known value, regardless of the clock or data inputs.
There are a few common types of resettable flip-flops, including **D Flip-Flops** with an asynchronous reset, **JK Flip-Flops**, and **T Flip-Flops** with reset functionality.

- **RESET Input**: The reset input directly controls the flip-flop's output.
- **Asynchronous Operation**: The reset is typically asynchronous, meaning it can force the output to reset at any time, regardless of the clock input.

Common Types:

1. **D Flip-Flop with Reset**:
   - When the **RESET** input is activated (usually **RESET = 1**), the output **Q** is set to **0** immediately.
   - When **RESET = 0**, the flip-flop operates normally based on the clock input.
2. **JK Flip-Flop with Reset**:
   - A JK flip-flop with a reset input can reset its state to **0** when the **RESET** input is active.
3. **T Flip-Flop with Reset**:
   - A T flip-flop with a reset input can also reset its output to **0** when **RESET** is asserted.
   -

Truth Table for a D Flip-Flop with Reset:

RESET	CLK	D	Q (Next State)
1	--	--	0
0	↑	0	0
0	↑	1	1

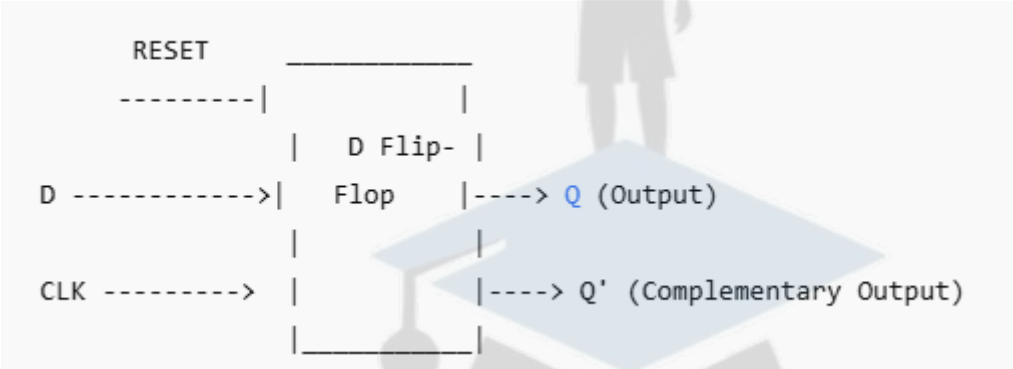
Circuit Diagram for a Resettable D Flip-Flop

Steps to Draw the Circuit:

1. **D Flip-Flop Symbol**:
   - Draw the standard D flip-flop rectangle and label it with **D**, **Q**,

and **Q'** for data input and outputs.

2. **Add the Reset Input**:
   - Add an additional input, labeled **RESET**.
   - Connect **RESET** directly to the **SET/RESET** control pin of the flip-flop, which forces the output **Q** to **0** when **RESET = 1**.
3. **Clock Input (CLK)**:
   - Add the **CLK** input that triggers the flip-flop on the rising edge of the clock signal when **RESET = 0**.
4. **Output**:
   - On the right side, draw the outputs **Q** and **Q'**.



Sequential logic timing considerations

1. Clock Signal

- A **clock signal** is used to synchronize the timing of state changes in sequential circuits. It ensures that all flip-flops or registers change states at the same time, preventing race conditions and maintaining synchronized operations.

2. Setup Time (Tsu)

- **Setup time** is the minimum time before the clock edge during which the data input (D or J/K inputs, etc.) must be stable and valid. If the data changes too close to the clock edge, the flip-flop might not correctly latch the value.

3. Hold Time (Th)

- **Hold time** is the minimum amount of time after the clock edge that the data input must remain stable in order for the flip-flop to correctly latch the value.

4. Clock Skew and Jitter

- **Clock skew** is the difference in arrival times of the clock signal at different parts of the circuit. Ideally, all flip-flops should receive the clock signal at the same time, but clock distribution networks can introduce slight timing differences.

5. Propagation Delay (Tp)

- **Propagation delay** is the time taken for a signal to travel from the input to the output of a logic gate or flip-flop. This delay impacts the overall timing of the circuit and must be accounted for to avoid violations of setup or hold times.

6. Timing Diagram

- A **timing diagram** is a graphical representation of the behavior of signals over time, showing how data and control signals change in relation to the clock.

7. Maximum Clock Frequency (f_max)

- The **maximum clock frequency** is the fastest clock speed at which a circuit can operate reliably. It's determined by the timing constraints of the sequential elements in the circuit.

8. Race Conditions

- A **race condition** occurs when the timing of signals causes unexpected behavior, particularly when there is uncertainty about the order of state changes. Race conditions can happen if the setup and hold times are violated, causing the output to change unpredictably.

9. State Machine Design

- In **sequential circuits** like **Finite State Machines (FSMs)**, the timing of transitions between states is critical. Incorrect timing can lead to:
  - **Unintended state transitions**
  - **Incorrect outputs**
  - **Glitches in the state machine**

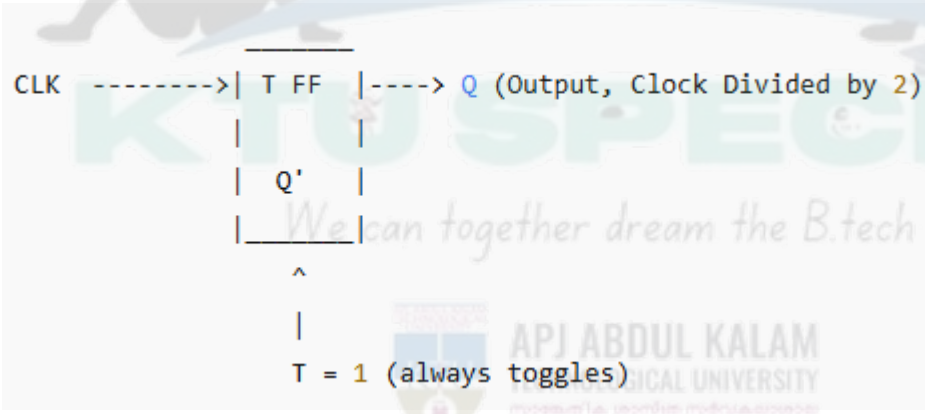
Toggle flop clock divider

A **Toggle Flip-Flop Clock Divider** (often called a **T Flip-Flop Clock Divider**) is a simple and effective way to divide the frequency of a clock signal by a factor of **2**. The T flip-flop (toggle flip-flop) is ideal for this task because of its ability to toggle its output state on each clock cycle, effectively halving the frequency of the input clock signal.

How a Toggle Flip-Flop Works as a Clock Divider:

1. **T Flip-Flop**: A T flip-flop has a single **T** (toggle) input, a **CLK** (clock) input, and an output **Q**. The flip-flop toggles its state (from 0 to 1, or from 1 to 0) on every rising edge of the clock if **T = 1**. When **T = 0**, the flip-flop holds its current state.
2. **Clock Division**: By connecting the clock signal directly to the **CLK** input of the T flip-flop and setting the **T input** to **1** (so the flip-flop always toggles), each rising edge of the input clock will toggle the output **Q**. Therefore, the output **Q** will have a frequency **half** that of the input clock.

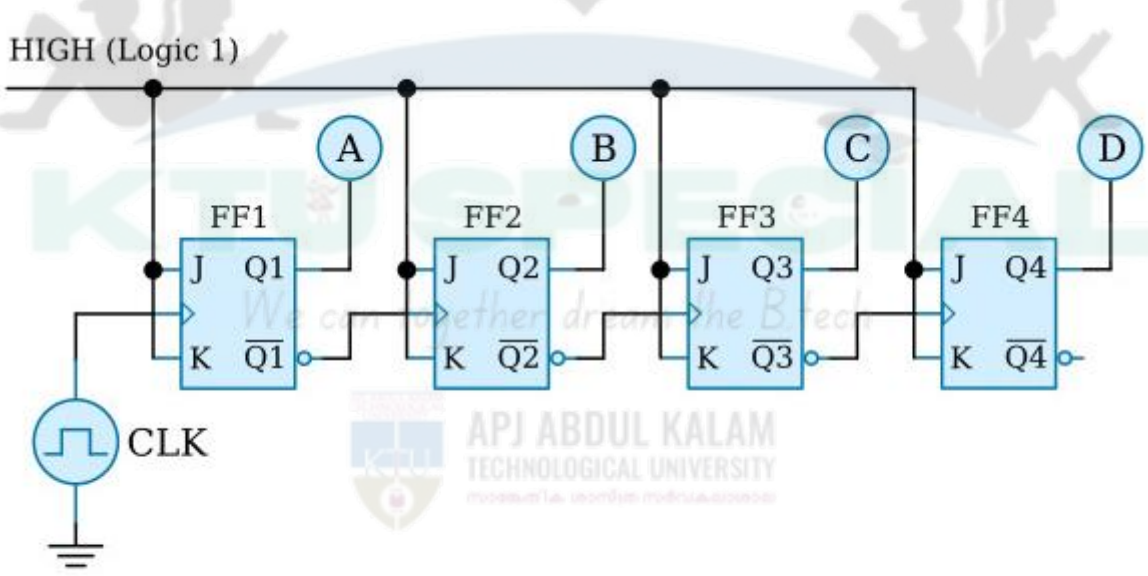
Simple Diagram of a T Flip-Flop Clock Divider:



- **CLK**: The input clock signal.
- **T**: Set to **1** to enable the toggling action.
- **Q**: The divided output clock, which has a frequency that is half the frequency of the input **CLK**.

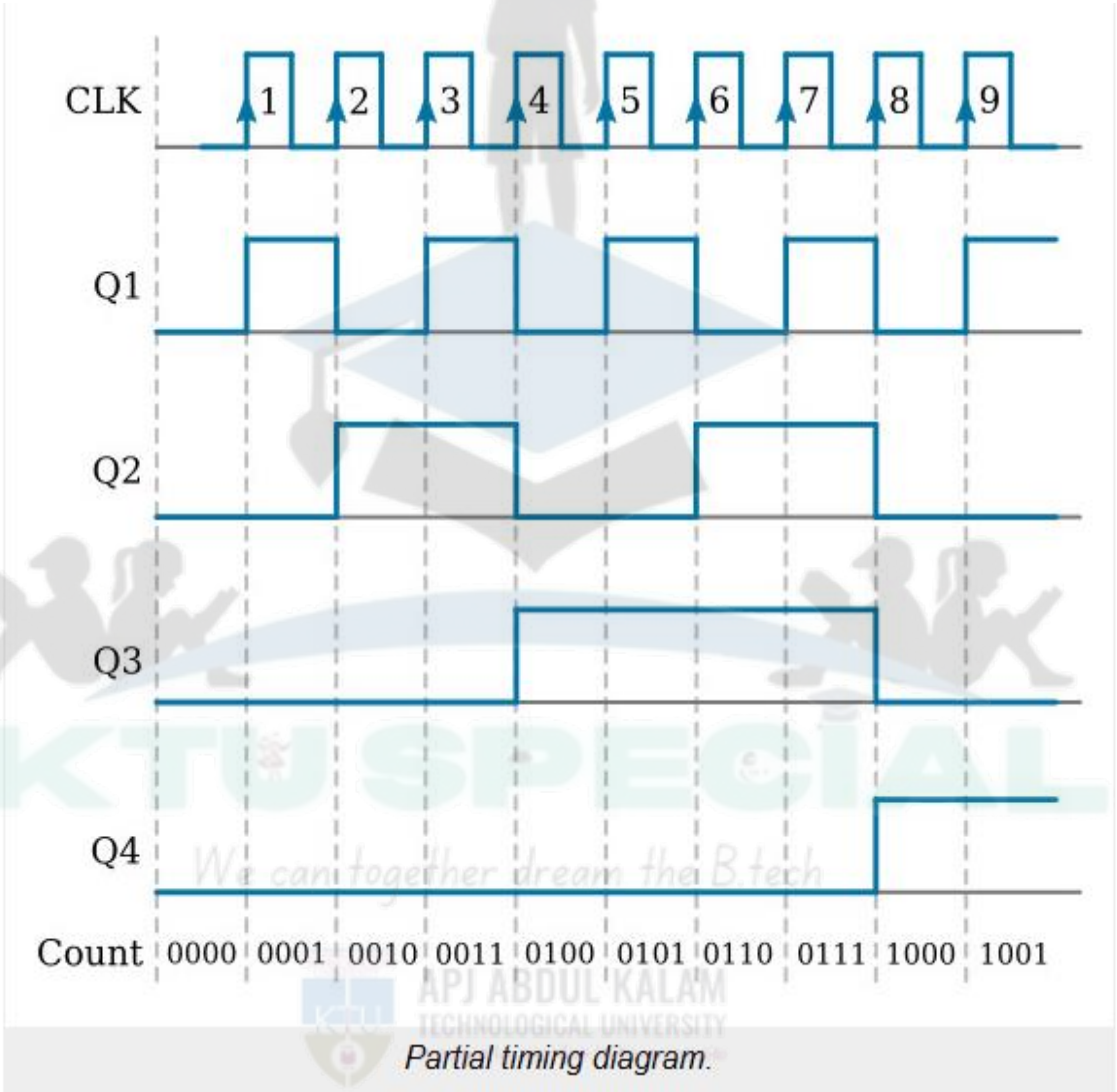
Asynchronous (Ripple) Counter

- Ripple counters are so named because the count is like a chain reaction that ripples through the counter because of the time involved.
- This effect will become more evident with the explanation of the following circuit. The ripple counter is also called an **asynchronous** counter.
- Asynchronous means that the events (setting and resetting of flip-flops) occur one after the other rather than all at once.
- Because the ripple count is asynchronous, it can produce erroneous indications when the clock speed is high.
- A high-speed clock can cause the lower stage flip-flops to change state before the upper stages have reacted to the previous clock pulse.
- The errors are produced by the flip-flops' inability to keep up with the clock.



Four-stage (4-bit) asynchronous counter.

- The figure above shows a basic four-stage, or modulo-16, asynchronous counter. The inputs and outputs are shown in the figure below.
- The four J-K flip-flops are connected to perform a toggle function; which divides the input frequency by 2.
- The HIGHs on the J and K inputs enable the flip-flops to toggle.
- The flip-flops change state on the positive-going pulse.



Partial timing diagram.

- Assume that A, B, C, and D are lamps and that all the flip-flops are reset.
- The lamps will all be out, and the count indicated will be 0000_2.
- The positive-going pulse of clock pulse 1 causes flip-flop FF1 to set.
- This lights lamp A, and we have a count of 0001_2. The positive-going

pulse of clock pulse 2 toggles FF1, causing it to reset.

- The positive-going input to FF2 (from Q1) causes it to set and causes B to light. The count after two clock pulses is 0010_2, or 2_{10}.
- Clock pulse 3 causes FF1 to set and lights lamp A.
- The setting of FF1 does not affect FF2, and lamp B stays lit. After three clock pulses, the indicated count is 0011_2.
- Clock pulse 4 causes FF1 to reset, which causes FF2 to reset, which causes FF3 to set, giving us a count of 0100_2. This step shows the ripple effect.

Shift Registers

- Shift Register is a group of flip flops used to store multiple bits of data.
- The bits stored in such registers can be made to move within the registers and in/out of the registers by applying clock pulses.
- An n-bit shift register can be formed by connecting n flip-flops where each flip-flop stores a single bit of data.
- The registers which will shift the bits to the left are called "Shift left registers".
- The registers which will shift the bits to the right are called "Shift right registers".

Shift registers are basically of following types.

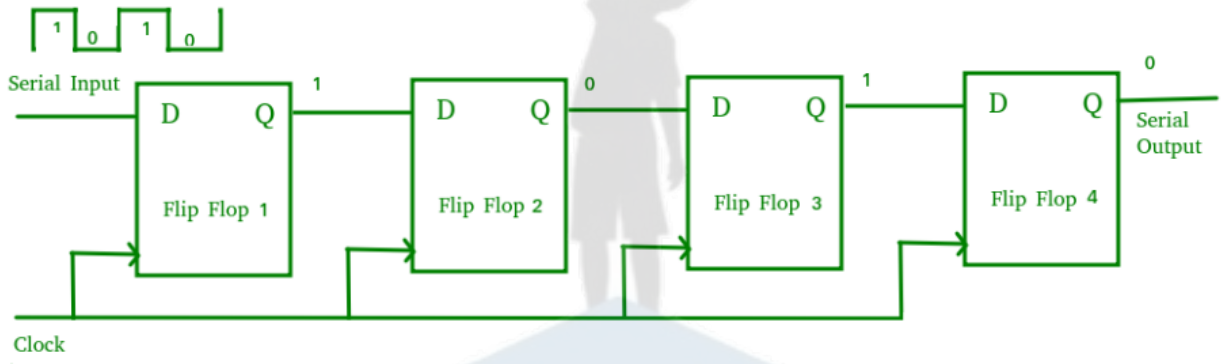
- Serial In Serial Out shift register
- Serial In parallel Out shift register
- Parallel In Serial Out shift register
- Parallel In parallel Out shift register

Serial-In Serial-Out Shift Register (SISO)

- The shift register, which allows serial input (one bit after the other through a single data line) and produces a serial output is known as a Serial-In Serial-Out shift register.
- Since there is only one output, the data leaves the shift register one bit at a

time in a serial pattern, thus the name Serial-In Serial-Out Shift Register.

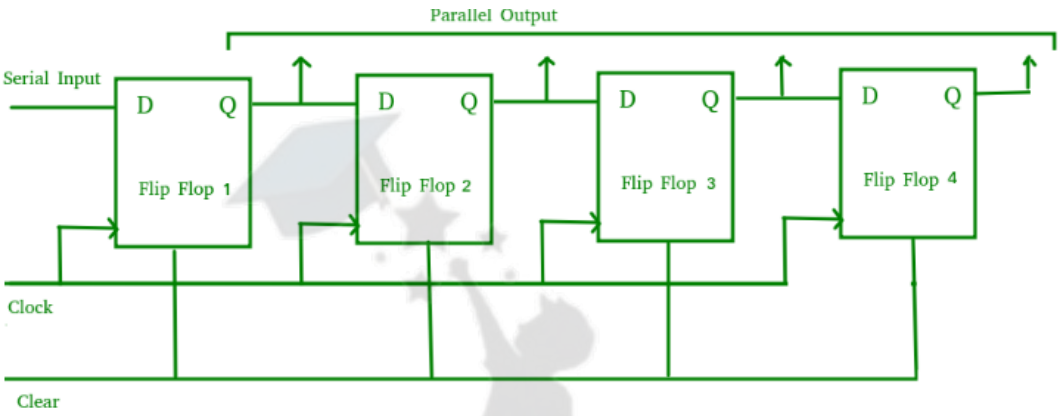
- The logic circuit given below shows a serial-in serial-out shift register.
- The circuit consists of four D flip-flops which are connected in a serial manner.
- All these flip-flops are synchronous with each other since the same clock signal is applied to each flip-flop.



Serial-In Serial-Out Shift Register (SISO)

Serial-In Parallel-Out Shift Register (SIPO)

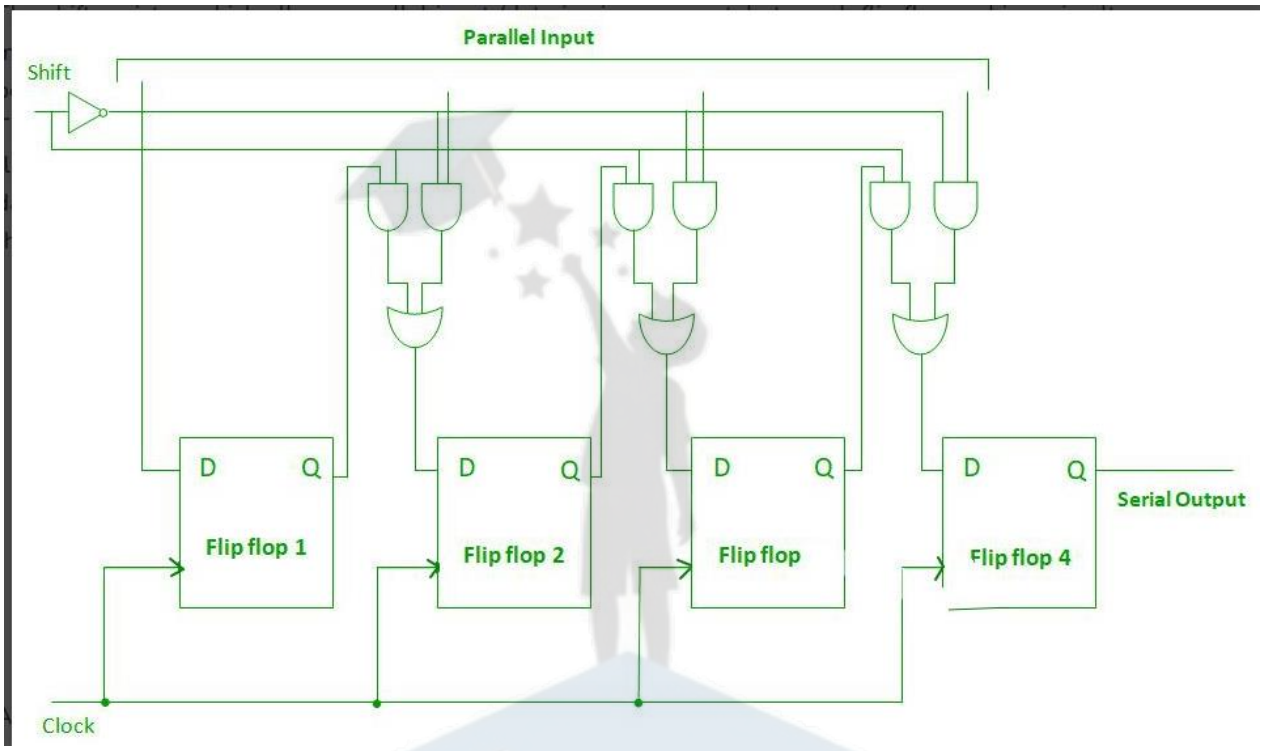
- The shift register, which allows serial input (one bit after the other through a single data line) and produces a parallel output is known as the Serial-In Parallel-Out shift register.
- The logic circuit given below shows a serial-in-parallel-out shift register.
- The circuit consists of four D flip-flops which are connected.
- The clear (CLR) signal is connected in addition to the clock signal to all 4 flip flops in order to RESET them.
- The output of the first flip-flop is connected to the input of the next flip flop and so on.
- All these flip-flops are synchronous with each other since the same clock signal is applied to each flip-flop.



Serial-In Parallel-Out shift Register (SIPO)

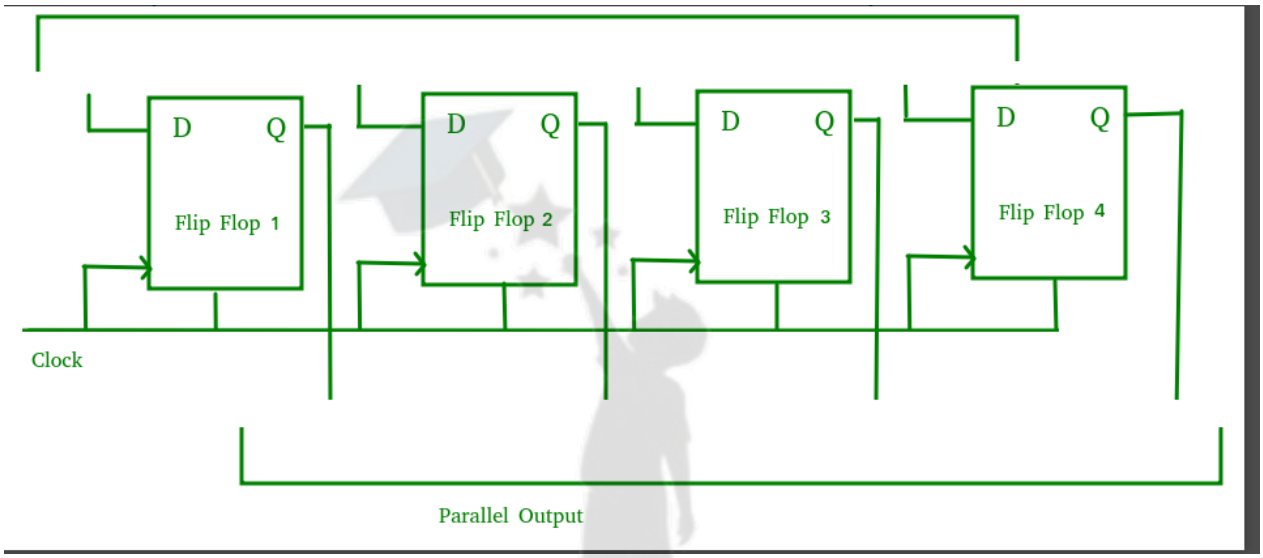
Parallel-In Serial-Out Shift Register (PISO)

- The shift register, which allows parallel input (data is given separately to each flip flop and in a simultaneous manner) and produces a serial output is known as a Parallel-In Serial-Out shift register.
- The logic circuit given below shows a parallel-in-serial-out shift register.
- The circuit consists of four D flip-flops which are connected.
- The clock input is directly connected to all the flip-flops but the input data is connected individually to each flip-flop through a multiplexer at the input of every flip-flop.
- The output of the previous flip-flop and parallel data input are connected to the input of the MUX and the output of MUX is connected to the next flip-flop.
- All these flip-flops are synchronous with each other since the same clock signal is applied to each flip-flop.



Parallel-In Parallel-Out Shift Register (PIPO)

- The shift register, which allows parallel input (data is given separately to each flip flop and in a simultaneous manner) and also produces a parallel output is known as Parallel-In parallel-Out shift register.
- The logic circuit given below shows a parallel-in-parallel-out shift register.
- The circuit consists of four D flip-flops which are connected.
- The clear (CLR) signal and clock signals are connected to all 4 flip-flops.
- In this type of register, there are no interconnections between the individual flip-flops since no serial shifting of the data is required.
- Data is given as input separately for each flip flop and in the same way, output is also collected individually from each flip flop.



Finite State Machine

- A **Finite State Machine (FSM)** is a mathematical model that is used to explain and understand the behavior of a digital system.
- More specifically, it is a structured and systematic model that helps to understand the behavior of a sequential circuit that exists in a finite number of states at a given point of time.
- In more simple words, a synchronous sequential circuit is also called as Finite State Machine FSM, if it has a finite number of states.

Components of a Finite State Machine

Finite States

The finite states are nothing but the distinct modes or conditions in the given system. Each of these finite states represents a specific behavior. In digital system representation, these finite states are generally represented through symbols or labels.

State Transitions

In terms of finite state machines, the state transition can be defined as the change from one state to another. This change in state or state transition takes placed based on some specific inputs or conditions. These state transitions are generally triggered by events which are associated with some rules or conditions and determine the next state of the system.

State Diagram

The state transition and the behavior of a finite state machine can be represented in a graphical form that is known as state diagram of the finite state machine.

Inputs

The inputs to the finite state machines are the external signals that trigger the state transitions in the system. These inputs are to be entered into the finite state machine by using sensors, user input devices like mic, keyboard, etc.

Outputs

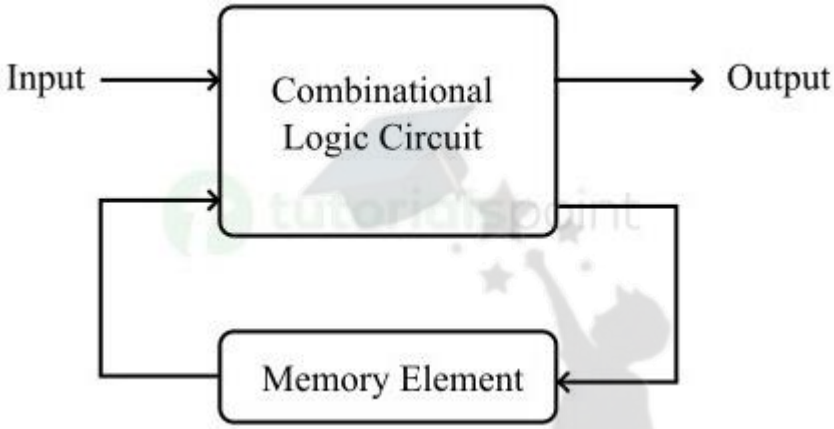
The results produced by the system as per the inputs and current states are known as outputs. These outputs of the system can be used to trigger events, control actuators, or to provide feedback to the external environment.

Types of Finite State Machine

- Mealy State Machine
- Moore State Machine

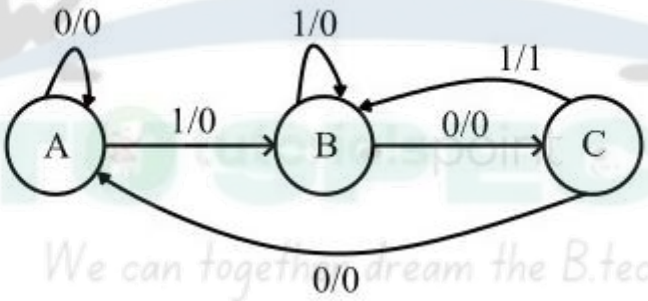
Mealy State Machine

A Finite State Machine is said to be a Mealy state machine, if its outputs depend on both present inputs & present states. The **block diagram of the Mealy state machine** is shown in the following figure –



As shown in the figure, there are two main parts presents in the Mealy state machine. Those are combinational logic circuit and memory element. The memory element is useful to provide some part of previous outputs and present states as inputs to the combinational logic circuit.
Based on the present inputs and present states, the Mealy state machine produces outputs. Therefore, the outputs will be valid only at positive or negative transition of the clock signal.

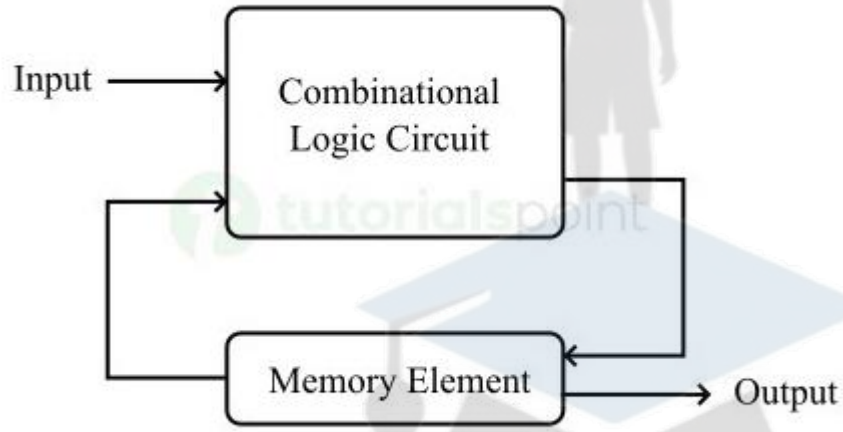
State Diagram of Mealy State Machine



In the above figure, there are three states, namely A, B and C. These states are labelled inside the circles and each circle corresponds to one state. State transitions between these states are represented with directed lines. Here, 0 / 0, 1 / 0 and 1 / 1 denote the input / output. In the above figure, there are two state transitions from each state based on the value of input.

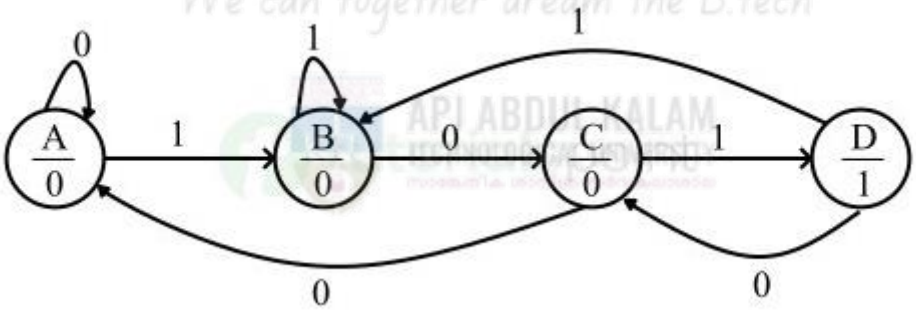
Moore State Machine

A Finite State Machine is said to be a Moore state machine, if its outputs depend only on the present states.
The **block diagram of the Moore state machine** is shown in the following figure



As shown in above figure, there are two parts presents in a Moore state machine. Those are combinational logic and memory. In this case, the present inputs and present states determine the next states. So, based on next states, Moore state machine produces the outputs. Therefore, the outputs will be valid only after transition of the state.

State Diagram of Moore State Machine



In the above figure, there are four states, namely A, B, C, and D. These states and the respective outputs are labelled inside the circles. Here, only the input value is

labeled on each transition. In the above figure, there are two transitions from each state based on the value of input.
In general, the number of states required in Moore state machine is more than or equal to the number of states required in Mealy state machine. There is an equivalent Mealy state machine for each Moore state machine. So, based on the requirement we can use one of them.

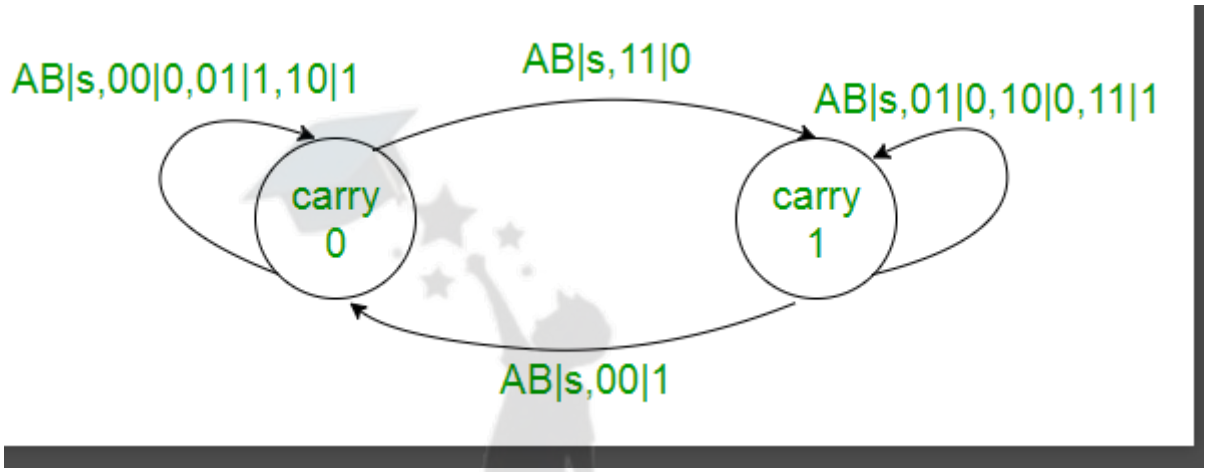
Synchronous Sequential Circuits

- A sequential circuit is a digital circuit, whose output depends not only on the current inputs but also on the history of past inputs.
- This places sequential circuits in a different category than the combinational circuits, whose outputs depend merely on the current inputs, and which have no "memory" of what happened previously.
- Sequential circuits use memory elements like flip-flops or latches that retain their past state;
- In a synchronous sequential circuit, the state of the circuit changes only on the rising or falling edge of the clock signal, and all changes in the circuit are synchronized to this clock.
- Synchronous sequential circuits can be implemented using flip-flops, which are circuits that store binary values and maintain their state even when the inputs change.

Steps to solve a problem

1. Draw the state diagram from the problem statement or from the given state table.

Example: Serial Adder. The functioning of serial adder can be depicted by the following state diagram. X1 and X2 are inputs, A and B are states representing carry.



2. Draw the state table. If there is any redundant state then reduce the state table.

Present state y	00	X1,X2 External inputs 01	(Next state, Output) 11	10
A	A,0	A,1	B,0	A,1
B	A,1	B,0	B,1	B,0

3. Select **state assignment** i.e. assign binary numbers to the states according to total number states. Also decide the memory element (flip-flops) for the circuit. A -> 0 B -> 1

4. Replace the assignments in the state table to obtain Transition table:

Present state y	00	01	11	10
0	0,0	0,1	1,0	0,1
1	0,1	1,0	1,1	1,0

5. Separate the output table from the transition table.

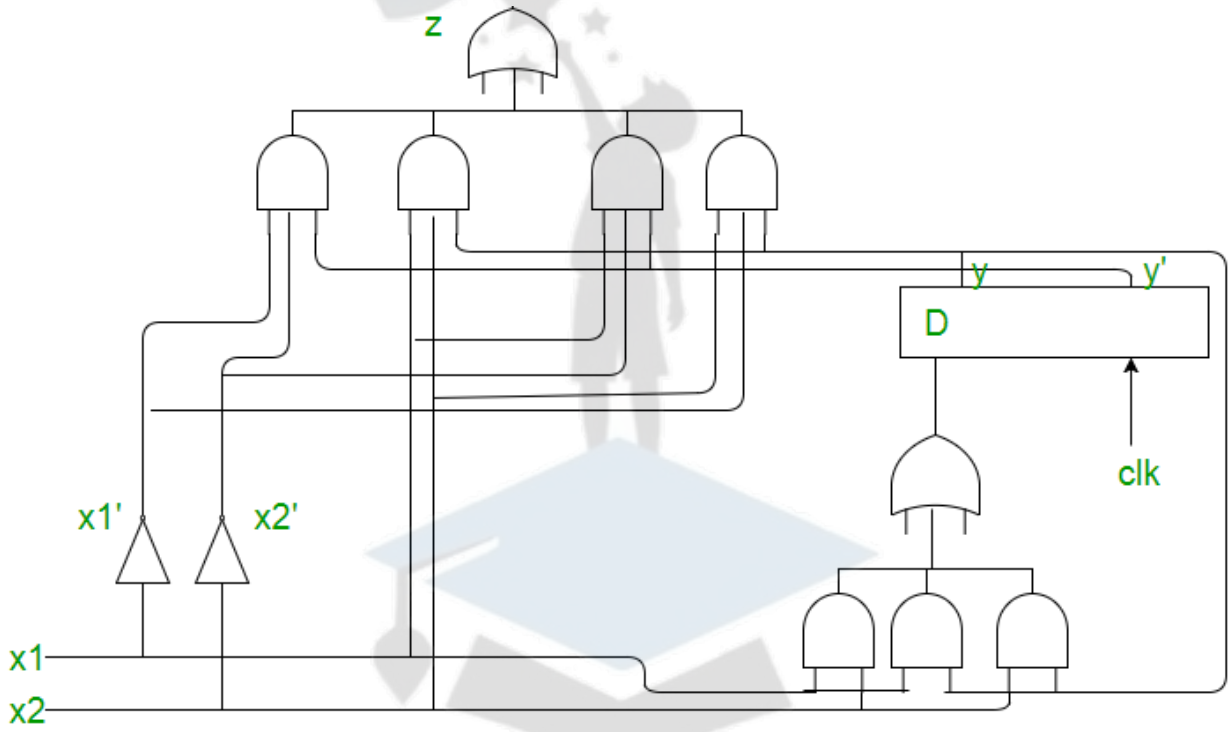
Present state y	00	01	11	10
0	0	1	0	1
1	1	0	1	0

6. Excitation table for the flip-flop is obtained from the transition table using the output of flip-flop. **Excitation table for D flip-flop:**

Present state y	00	01	11	10
0	0	0	1	0
1	0	1	1	1

D = x1 x2 +x1 y+x2 y

7. Draw the circuit diagram using gates and flip-flops.



Advantages of Synchronous Sequential Circuits

- **Predictable behavior**: The use of a clock signal makes the behavior of a synchronous sequential circuit predictable and deterministic, which is important for real-time control applications.
- **Synchronization**: Synchronous sequential circuits ensure that all elements of the circuit change at the same time, preventing race conditions and making the circuit easier to design and debug.

Disadvantages of Synchronous Sequential Circuits

1. **Clock skew**: Clock skew is a timing error that occurs when the clock signal arrives at different flip-flops at different times. This can cause errors in the operation of the circuit.
2. **Timing jitter**: Timing jitter is a variation in the arrival time of the clock signal that can cause errors in the operation of the circuit.

Counters

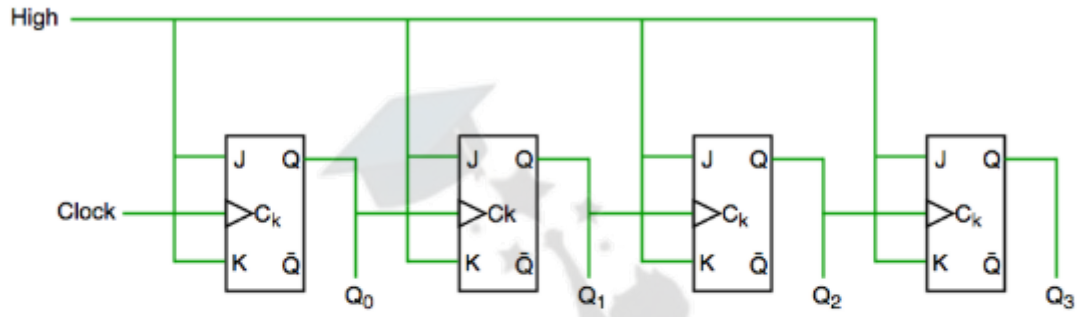
- A **Counter** is a device which stores (and sometimes displays) the number of times a particular event or process has occurred, often in relationship to a clock signal.
- Counters are used in digital electronics for counting purpose, they can count specific event happening in the circuit.
- Counters are sequential circuit that count the number of pulses can be either in binary code or BCD form.
- The main properties of a counter are timing , sequencing , and counting. Counter works in two modes.

counters are broadly divided into two categories

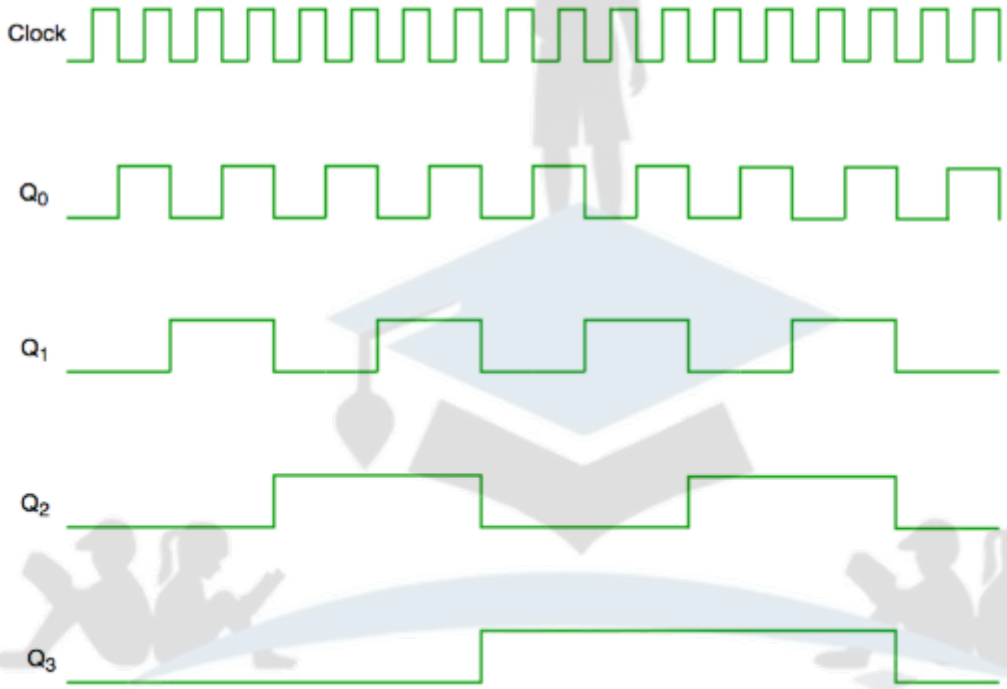
1. Asynchronous counter
2. Synchronous counter

1. Asynchronous Counter

In asynchronous counter we don't use universal clock, only first flip flop is driven by main clock and the clock input of rest of the following flip flop is driven by output of previous flip flops. We can understand it by following diagram-



(a) Asynchronous counter

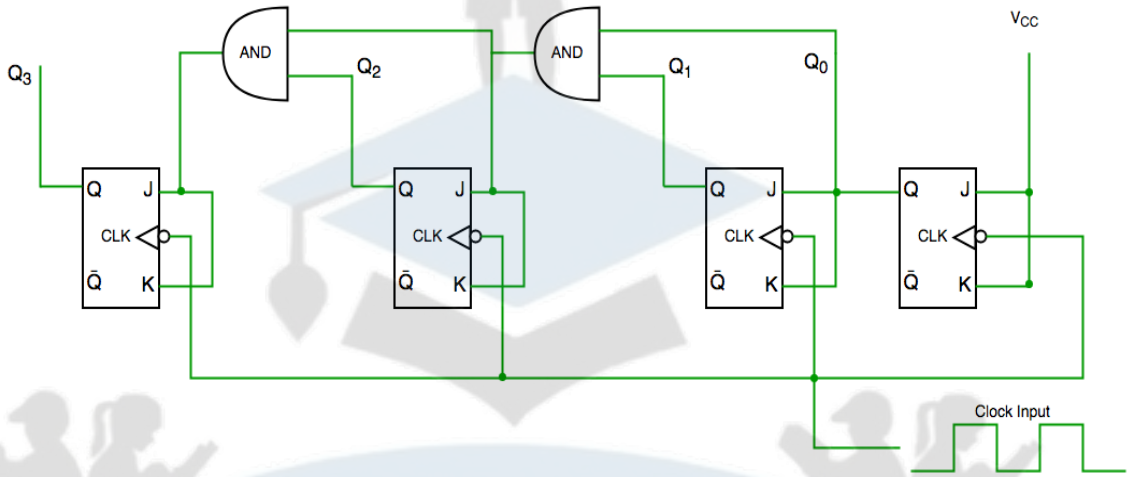


(b) Timing Diagram

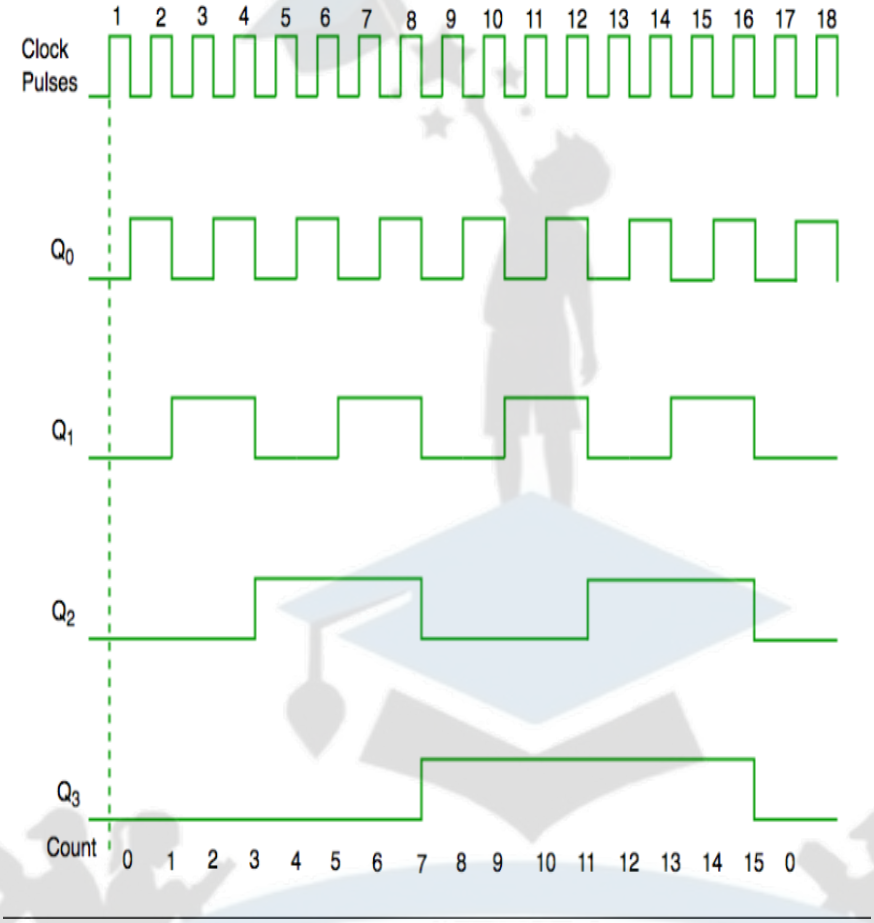
- It is evident from timing diagram that Q0 is changing as soon as the rising edge of clock pulse is encountered, Q1 is changing when rising edge of Q0 is encountered(because Q0 is like clock pulse for second flip flop) and so on.
- In this way ripples are generated through Q0,Q1,Q2,Q3 hence it is also called **RIPPLE counter and serial counter.**
- A ripple counter is a cascaded arrangement of flip flops where the output of one flip flop drives the clock input of the following flip flop

2. **Synchronous Counter**

Unlike the asynchronous counter, synchronous counter has one global clock which drives each flip flop so output changes in parallel. The one advantage of synchronous counter over asynchronous counter is, it can operate on higher frequency than asynchronous counter as it does not have cumulative delay because of same clock is given to each flip flop. It is also called as parallel counter.



Synchronous counter circuit



Procedural assignment in verilog

A **procedural assignment** in Verilog is typically done inside **procedural blocks**, such as always or initial, and is used to assign values to variables. The two main types of procedural assignments are:

- Blocking assignment (=)
- Non-blocking assignment (<=)

1. Blocking Assignment (=)

Blocking assignments execute sequentially. The next statement is executed only after the current assignment completes

```
module blocking_example;
   reg [3:0] a, b, c;

   initial begin
      a = 4'b0001;    // Blocking assignment
      b = 4'b0010;    // This will happen after 'a' is assigned
      c = a + b;      // This will happen after 'b' is assigned
   end
endmodule
```

In this example, each statement executes in the order it's written. The variable `c` gets the value of `a + b` after `b` is assigned.

2. **Non-blocking Assignment (<=)**

Non-blocking assignments allow multiple statements in a procedural block to run in parallel. It schedules the assignments but does not wait for them to complete before moving to the next statement.

```
module nonblocking_example;
   reg [3:0] a, b, c;

   initial begin
      a <= 4'b0001;    // Non-blocking assignment
      b <= 4'b0010;    // This happens in parallel with the assignment to 'a'
      c <= a + b;      // 'c' will be assigned after both 'a' and 'b' are updated
   end
endmodule
```

Here, a and b are updated at the same time, and then c is updated after both a and b have their values updated.

- **Blocking assignments** are typically used for combinatorial logic or when sequential operations are needed (like in procedural flow).
- **Non-blocking assignments** are preferred for **sequential logic** (like flip-flops or registers), especially in clocked processes. This helps in designing hardware that accurately reflects hardware behavior in real-world circuits.

Conditional Programming Constructs

Conditional statements in a programming language help to branch the execution of code depending on some condition. In simpler words, using these statements, the execution of specific code lines can be controlled using some conditions. Verilog provides two types of conditional statements, namely

1. If-else
2. Case statement

If-else

If-else is the most straightforward conditional statement construct. For the compiler, this statement means that "If some condition is true, execute code inside if or else execute codes inside else." The condition evaluates to be true if the argument is a non-zero number. In if-else, it is not necessary to always have an else block.

Syntax:

```
if(<condition>) begin
<code>
end
else begin
<code>
end
```


If-else-if

If-else-if construct is used when more than one condition needs to be checked. All the conditions are checked serially, and if none of the conditions is true, then the `else` block is executed.

```
if(<condition>) begin
  <code>
end
else if(<condition>) begin
  <code>
end
else begin
  <code>
end
```

Case Statements

Case statements are generally used when the condition is an equivalence check, which means "==" or "===". Case statement makes the code more readable wrt to if-else if there are many conditions to check. As the name indicates, this statement will switch a particular case based on some arguments.

Syntax:

```
case(<argument>)
<case-1>: <single-line code>
<case-2>: begin
<code>
end

default: <code>
endcase
```

There are three case statement variants: case, casez, and casex

Case

In the case statement, exact matching is done. If even a single bit does not match the cases defined, the default case will be executed.

```
module case_demo;
   reg [1:0] op_code;
   initial begin
      op_code = 2'd1;
      $display("Op code is %0d", op_code);
      #10;
      op_code = 2'd2;
      $display("Op code is %0d", op_code);
      #10;
      op_code = 2'bx;
      $display("Op code is %0d", op_code);
      #10;
      op_code = 2'bz1;
      $display("Op code is %0d", op_code);
   end

   always @(op_code) begin
      case(op_code)
         2'b00: begin
            $display("Case 0 selected");
         end
         2'b01: begin
            $display("Case 1 selected");
         end
         2'bz1: begin
            $display("Case z1 selected");
```

```
        end
        default: begin
            $display("No operation for this case");
        end
    endcase
  end
endmodule
```

Output

Op code is 1

Case 1 selected

Op code is 2

No operation for this case

Op code is x

No operation for this case

Op code is Z

Case z1 selected

Casez

Consider a simple example where the master uses a select signal to select a particular slave amongst many (say, four slaves). Thus, the select line would be a 4-bit signal. A single bit needs to be high to select any slave, and the rest of the bits can be anything. Thus, instead of writing multiple cases for a slave, we can use casez. In casez, the z or '?' is does not care, which means it will not be considered while comparing the case with the argument.

```
module casez_demo;
   reg [1:0] op_code;

   initial begin
      op_code = 2'd1;
      $display("Op code is %0d", op_code);
      #10;
      op_code = 2'd2;
      $display("Op code is %0b", op_code);
      #10;
      op_code = 2'b1z;
      $display("Op code is %0b", op_code);
      #10;
      op_code = 2'b11;
      $display("Op code is %0b", op_code);
      #10;
      op_code = 2'b0z;
      $display("Op code is %0b", op_code);
   end

   always @(op_code) begin
      casez(op_code)
         // 2'b01: begin
         //     $display("Case **01** selected");
         // end
         2'b00: begin
            $display("Case **00** selected");
         end
         2'b01: begin
            $display("Case **01** selected");
         end
         2'b1?: begin
            $display("Case **1?** selected");
         end
         default: begin
            $display("No operation for this case");
         end
      endcase
   end
```

```
endmodule
```

Output

```
# Op code is 1
# Case **01** selected
# Op code is 10
# Case **1?** selected
# Op code is 1z
# Case **1?** selected
# Op code is 11
# Case **1?** selected
# Op code is 0z
# Case **00** selected
```

Casex

Casex is a bit more flexible than casez. In casex, even X is not considered during case comparison. Thus, X, Z, ? all are considered as do not care in casex.

In the below example, the same operations are performed in different always block but using a different case statement. In the output, it can be seen that, if the value of the argument, i.e., op_code is X, then default case is executed in the case and casez construct, but for casex, the X is considered as do not care, and thus the first case is executed.

When the op_code is Z, then only in the case construct the default case is executed, and for the casex and casez, Z is considered as do not care, and the 1st case is executed.

```
module case_diff_demo;
   reg [7:0] a, b, op, op_x, op_z;
   reg [1:0] op_code;

   always @(a or b or op_code) begin
      case(op_code)
         2'b00: begin
            op = a + b;
         end
         2'b01: begin
            op = a - b;
         end
         default
            op = 8'd0;
      endcase
      $display("Output of case = %0d", op);
   end

   always @(a or b or op_code) begin
      casez(op_code)
         2'b00: begin
            op_z = a + b;
         end
         2'b01: begin
            op_z = a - b;
         end
         default
            op_z = 8'd0;
      endcase
      $display("Output of casez = %0d", op_z);
   end

   always @(a or b or op_code) begin
      casex(op_code)
         2'b00: begin
            op_x = a + b;
```

```
        end
        2'b01: begin
            op_x = a - b;
        end
        default
            op_x = 8'd0;
    endcase
    $display("Output of casex = %0d", op_x);
  end

  initial begin
    a = 8'd12;
    b = 8'd4;
    op_code = 0;
    $display("op_code = %0d", op_code);
    #10;
    op_code = 1;
    $display("op_code = %0d", op_code);
    #10;
    op_code = 3;
    $display("op_code = %0d", op_code);
    #10;
    op_code = 2'bx;
    $display("op_code = %0d", op_code);
    #10;
    op_code = 2'bz;
    $display("op_code = %0d", op_code);
  end
endmodule
```


Output

op_code = 0

Output of case = 16

Output of casez = 16
Output of casex = 16
op_code = 1
Output of case = 8
Output of casez = 8
Output of casex = 8
op_code = 3
Output of case = 0
Output of casez = 0
Output of casex = 0
op_code = x
Output of case = 0
Output of casez = 0
Output of casex = 16
op_code = z
Output of case = 0
Output of casez = 16
Output of casex = 16

Test benches

In Verilog, a **testbench** is used to simulate and verify the functionality of a design (module) in a controlled environment. Testbenches are essential for verifying the correctness of your design before hardware implementation.

A **testbench** generally includes:

1. **Instantiation of the module under test (MUT)**

2. **Stimulus generation** (providing input values)
3. **Monitoring and checking outputs**
4. **Simulation control** (like initial and always blocks for timing control)

Basic Structure of a Testbench

```
module tb_example;
    // Declare signals for inputs and outputs
    reg clk;
    reg reset;
    reg [3:0] a, b;
    wire [3:0] sum;

    // Instantiate the module under test (MUT)
    adder uut (
        .a(a),
        .b(b),
        .sum(sum)
    );

    // Generate clock signal
    always begin
        #5 clk = ~clk; // Toggle clock every 5 time units
    end

    // Stimulus generation and test cases
    initial begin
        // Initialize signals
        clk = 0;
        reset = 0;
        a = 4'b0000;
        b = 4'b0000;

        // Apply test stimulus
```

```
    #10 a = 4'b0101; b = 4'b0011; // Apply inputs and wait 10 time units
    #10 a = 4'b1111; b = 4'b1001;
    #10 a = 4'b1010; b = 4'b1100;
    #10 $stop; // Stop the simulation
  end

  // Monitor outputs
  initial begin
    $monitor("At time %t, a = %b, b = %b, sum = %b", $time, a, b, sum);
  end
endmodule
```

Components of a Testbench

Clock Generation

A clock signal is essential for synchronous circuits. It is usually generated with an `always` block that toggles the clock after a set delay.

```
always begin
  #5 clk = ~clk; // Toggle every 5 time units
end
```

Reset Signal

Typically, designs start with an active-low or active-high reset. The testbench often initializes the reset signal and later deasserts it.

```
initial begin
    reset = 1;
    #10 reset = 0;
end
```

Stimulus (Input Signals)

This is the part where you apply various input combinations to test your module. You can use the initial block to apply stimulus to your design and wait for specific time intervals.

```
initial begin
    a = 4'b0000;
    b = 4'b0000;
    #10 a = 4'b0101; b = 4'b0011;
    #10 a = 4'b1111; b = 4'b1001;
end
```

Monitoring Outputs

The $monitor command can be used to display outputs in the simulation console whenever any of the monitored signals change.

```
initial begin
    $monitor("At time %t, a = %b, b = %b, sum = %b", $time, a, b, sum);
end
```

Test Control (e.g., $stop, $finish)

- $stop: Pauses the simulation at a specific point.
- $finish: Ends the simulation.

```
initial begin
   #100 $finish; // End simulation after 100 time units
end
```

Example: 4-bit Adder

DUT (Device Under Test) - 4-bit Adder:

```
module adder (
   input [3:0] a, b,
   output [3:0] sum
);
   assign sum = a + b;
endmodule
```

Testbench for 4-bit Adder:

```
module tb_adder;

   // Declare testbench signals
   reg [3:0] a, b;
```

```
    wire [3:0] sum;

    // Instantiate the adder module
    adder uut (
        .a(a),
        .b(b),
        .sum(sum)
    );

    // Apply test vectors
    initial begin
        // Initialize values
        a = 4'b0000; b = 4'b0000;
        #10 a = 4'b0001; b = 4'b0011;  // 1 + 3 = 4
        #10 a = 4'b1111; b = 4'b0001;  // 15 + 1 = 16
        #10 a = 4'b0110; b = 4'b1010;  // 6 + 10 = 16
        #10 a = 4'b1010; b = 4'b1101;  // 10 + 13 = 23
        #10 $stop;  // End the simulation
    end

    // Monitor the outputs
    initial begin
        $monitor("Time=%0t | a=%b, b=%b, sum=%b", $time, a, b, sum);
    end

endmodule
```

Test Vectors: Various values for `a` and `b` are applied, and the expected sum is computed manually. This helps to verify that the adder works as expected.

$monitor: The output is displayed at each time step where the values of `a`, `b`, and `sum` change.

Simulation Control: The `$stop` command halts the simulation after a few test cases, so you can check the results.

Modeling a D flipflop in verilog

A D Flip-Flop is a basic digital storage element that captures the value of the data input (D) on the rising edge of the clock signal and holds that value until the next clock edge. It's commonly used in synchronous designs.

1. Basic D Flip-Flop (Positive Edge Triggered)

The most basic form of a D Flip-Flop triggers on the positive edge of the clock. The value at the input (`D`) is stored in the output (`Q`) when the clock rises (i.e., `posedge clk`).

```
module d_flipflop (
    input clk,   // Clock input
    input reset, // Asynchronous reset
    input D,     // Data input
    output reg Q // Output
);

    // Always block triggered on positive edge of clk or reset
    always @(posedge clk or posedge reset) begin
        if (reset)
            Q <= 1'b0;  // Reset Q to 0 on reset
        else
            Q <= D;     // On clock edge, latch the value of D
    end

endmodule
```

Inputs:

- `clk`: Clock signal (used to trigger the flip-flop).
- `reset`: Asynchronous reset that resets `Q` to `0`.

- `D`: Data input.

Output:

- `Q`: The output, which holds the value of `D` at the rising edge of the clock.

2. **D Flip-Flop with Active Low Reset**

Another common variation is when the reset is active low (i.e., `reset` is asserted with `0`). This is often used in designs where a low signal is used to reset the flip-flop.

```
module d_flipflop (
    input clk,   // Clock input
    input nreset, // Active-low reset
    input D,     // Data input
    output reg Q // Output
);

    // Always block triggered on positive edge of clk or active-low reset
    always @(posedge clk or negedge nreset) begin
        if (~nreset)
            Q <= 1'b0;  // Reset Q to 0 when nreset is 0
        else
            Q <= D;     // On clock edge, latch the value of D
    end

endmodule
```

Inputs:

- `clk`: Clock signal.
- `nreset`: Active-low reset signal.
- `D`: Data input.

Output:

- `Q`: The output signal that holds the value of `D`.

Modeling an FSM in verilog

Finite State Machines (FSMs) are widely used in digital design to model systems with a limited number of states. FSMs are classified into two types:

- **Moore FSM**: The output depends only on the state.
- **Mealy FSM**: The output depends on both the state and the inputs.

1. Moore FSM Example

This Moore FSM has two states:

- **IDLE**: Output is `0`.
- **ACTIVE**: Output is `1`.

```
module moore_fsm (
    input clk,        // Clock input
    input reset,      // Reset input
    output reg out    // Output
);
```

```
    // State encoding
    reg [1:0] state;  // 2-bit state variable

    // State definition
    parameter IDLE = 2'b00, ACTIVE = 2'b01;

    // Sequential state transition
    always @(posedge clk or posedge reset) begin
        if (reset)
            state <= IDLE;  // Reset to IDLE state
        else
            state <= (state == IDLE) ? ACTIVE : IDLE; // Toggle between IDLE
and ACTIVE
    end

    // Output logic
    always @(state) begin
        case(state)
            IDLE: out = 0;
            ACTIVE: out = 1;
            default: out = 0;
        endcase
    end

endmodule
```

- The FSM toggles between `IDLE` and `ACTIVE` states on each clock cycle.
- The output (`out`) is `0` in the `IDLE` state and `1` in the `ACTIVE` state.
- A reset signal forces the FSM to the `IDLE` state.

2.Mealy FSM

This Mealy FSM has two states:

- **IDLE**: Output depends on the input (input_signal).
- **ACTIVE**: Output depends on the input (input_signal).

```
module mealy_fsm (
   input clk,           // Clock input
   input reset,         // Reset input
   input input_signal,   // Input signal
   output reg out       // Output
);

   // State encoding
   reg [1:0] state;  // 2-bit state variable

   // State definition
   parameter IDLE = 2'b00, ACTIVE = 2'b01;

   // Sequential state transition
   always @(posedge clk or posedge reset) begin
      if (reset)
         state <= IDLE;  // Reset to IDLE state
      else
         state <= (state == IDLE) ? ACTIVE : IDLE; // Toggle between IDLE and
ACTIVE
   end

   // Output logic (Mealy FSM)
   always @(state or input_signal) begin
      case(state)
         IDLE: out = input_signal;  // Output depends on input_signal
         ACTIVE: out = input_signal;  // Output depends on input_signal
         default: out = 0;
```

```
    endcase
  end

endmodule
```

- In the `IDLE` and `ACTIVE` states, the output depends on the input signal (`input_signal`).
- The FSM toggles between `IDLE` and `ACTIVE` on each clock cycle.
- The output is 1 if `input_signal` is 1, otherwise `out = 0`.

Question Bank

1. What is the difference between an SR latch and an SR latch with enable? Explain with circuit diagrams.
2. Describe the operation of a JK flip-flop. How does it differ from a D flip-flop?
3. What is the purpose of a Register Enabled Flip-Flop? Give a Verilog example.
4. Explain the function of a Resettable Flip-Flop. How is asynchronous reset implemented?
5. Discuss the significance of timing considerations in sequential logic circuits. What are setup time and hold time?
6. Design a toggle flip-flop using a JK flip-flop. Explain its working with the help of a truth table and timing diagram.
7. Draw and explain the working of an asynchronous ripple counter. How is it different from a synchronous counter?
8. What is a shift register? Explain its types with examples.
9. What is an FSM? Differentiate between Mealy and Moore machines with examples.
10. Explain the process of FSM design. Create an FSM to detect the sequence '101' in a serial input stream.
11. Design and explain a 3-bit synchronous up-counter using FSM concept.
12. What are the steps involved in logic synthesis of an FSM? Illustrate with an example.
13. Write a Verilog code to implement a D flip-flop with asynchronous reset.
14. Explain procedural assignments in Verilog with suitable examples.

15. Differentiate between blocking and non-blocking assignments in Verilog with examples.

16. What are conditional programming constructs in Verilog? Write a code snippet to implement a 4:1 multiplexer using conditional operators.

17. What is the purpose of a testbench in Verilog? Write a testbench to verify the working of a D flip-flop.

18. Write a Verilog code to model a FSM that detects consecutive '11' in an input bitstream.

KTU SPECIAL
CONNECT WITH US

WHATSAPP GROUP

FOLLOW US NOW

TELEGRAM CHANNEL

SUBSCRIBE NOW

www.ktuspecial.in

SUBSCRIBE

FOLLOW US

Visit Website