Digital Systems

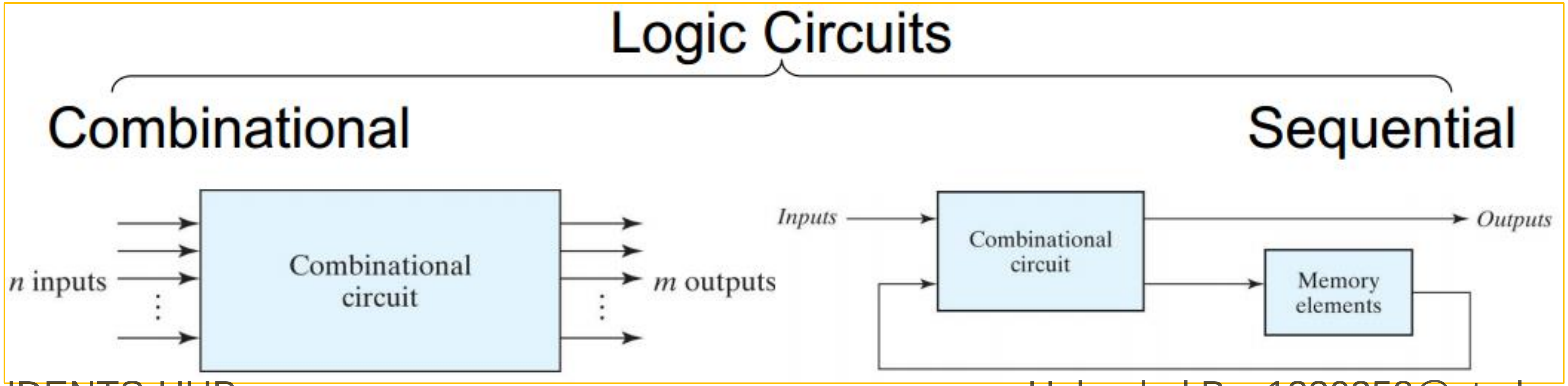
Section 2

Chapter (5)

Mohammed Khalil

Logic Circuits

- In Digital Systems, Logic circuits can be categorized as **combinational** or **sequential**
- **Combinational** Circuit
  - Circuit made of **logic gates only** and perform an operation that can be specified logically by a set of Boolean functions.
  - Circuit output at any time are determined **only by the current combination (current state/value)** of inputs.
- **Sequential** Circuit
  - Circuit is made of **storage/memory** elements and **logic gates**.
  - Circuit output depend on the current combination of inputs and **previously stored values.**



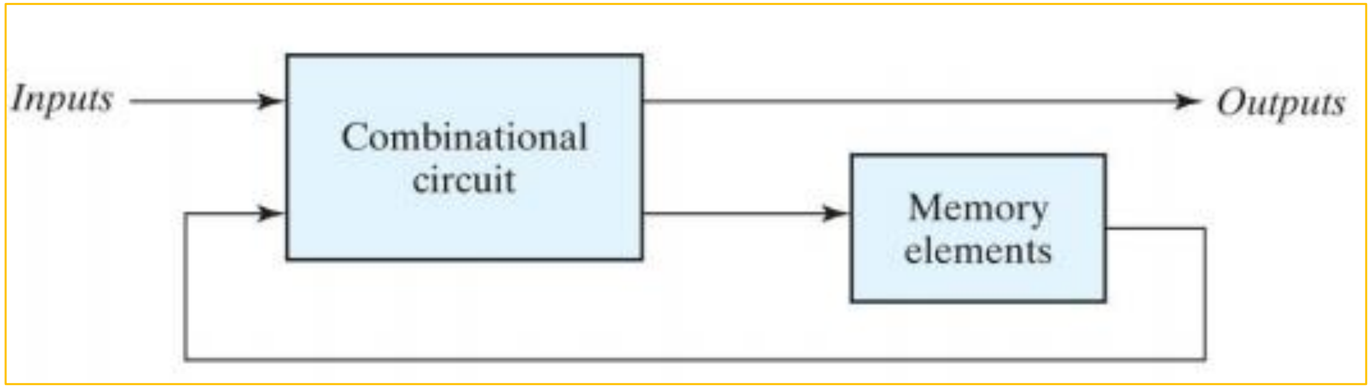
Sequential Circuits

- A **sequential** circuit consists of a **combinational** circuit to which **storage** elements are connected to form a **feedback** path.
  - It can **store**, **retain** and then **retrieve** this information

- The **storage/memory** elements are devices capable of **storing** binary information
  - The **stored binary** information define the **current/present state** (**Q(t)**) of the sequential circuit at any given *time*.

- A sequential circuit receives information from an **external** inputs as well as the **current/present** state of the **memory** elements to determine the **output** and **next state (Q(t+1))**.

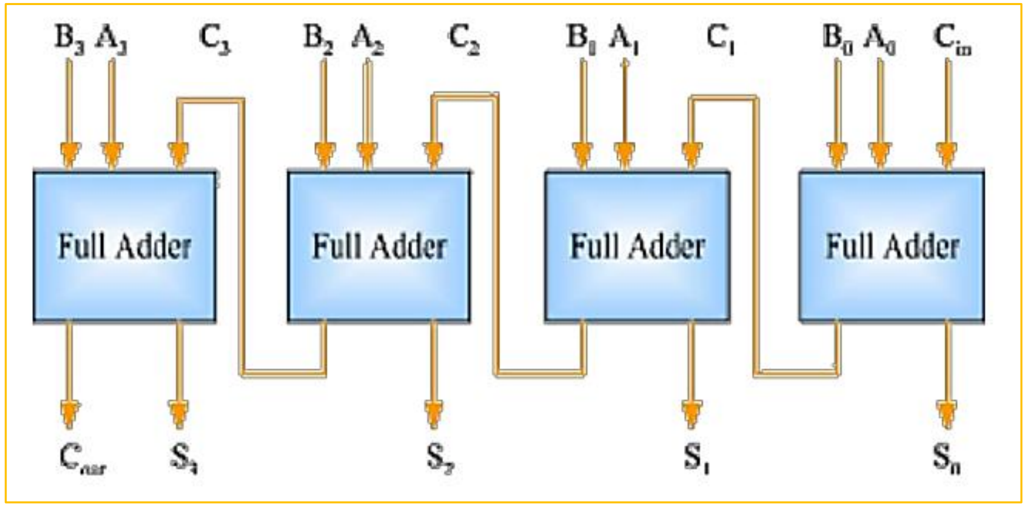
The **outputs** of the sequential circuit are **functions** of the **inputs** and the **present state** of the **memory** elements.



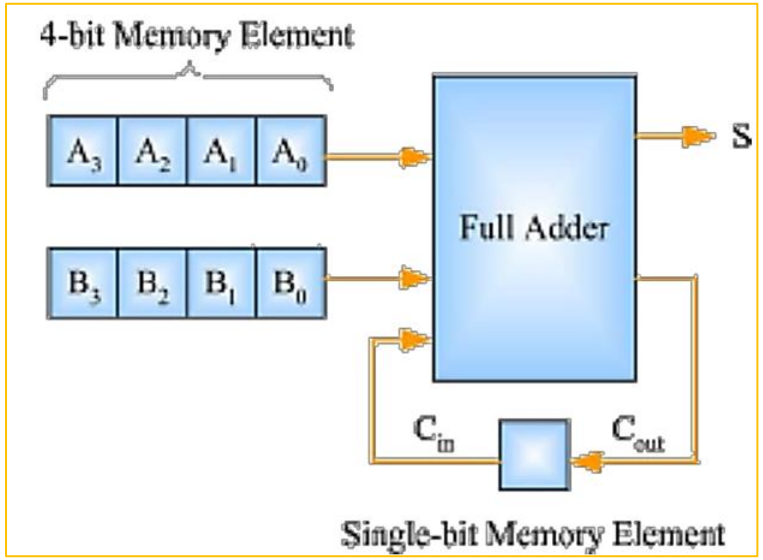
Block Diagram of **Sequential** Circuits

Sequential vs. Combinational Circuits

- The **combinational** circuit of a 4-bit binary adder comprises 4 full-adders



- The **sequential** circuit of a 4-bit binary adder comprises a full-adder and **memory**



Sequential Circuits

- A **sequential** circuit is specified by a **time sequence** of inputs, outputs, and **internal** states.

- **Two** main **types** of sequential circuits based on the **timing** of their signals
  - **Asynchronous** Sequential circuits: behavior depends on the **order** of **change** of **input** signals and can be affected at **any instant** of time [**Out** of this Course Scope]
  - **Synchronous** Sequential Circuits - behavior is defined from the knowledge of signals at **discrete** instants of time.

- **Synchronous** Sequential Circuit
  - Uses a **clock** signal as an additional **input**
  - Changes in the memory elements are **controlled** by the **clock**
  - Changes happen at **discrete** instances of time

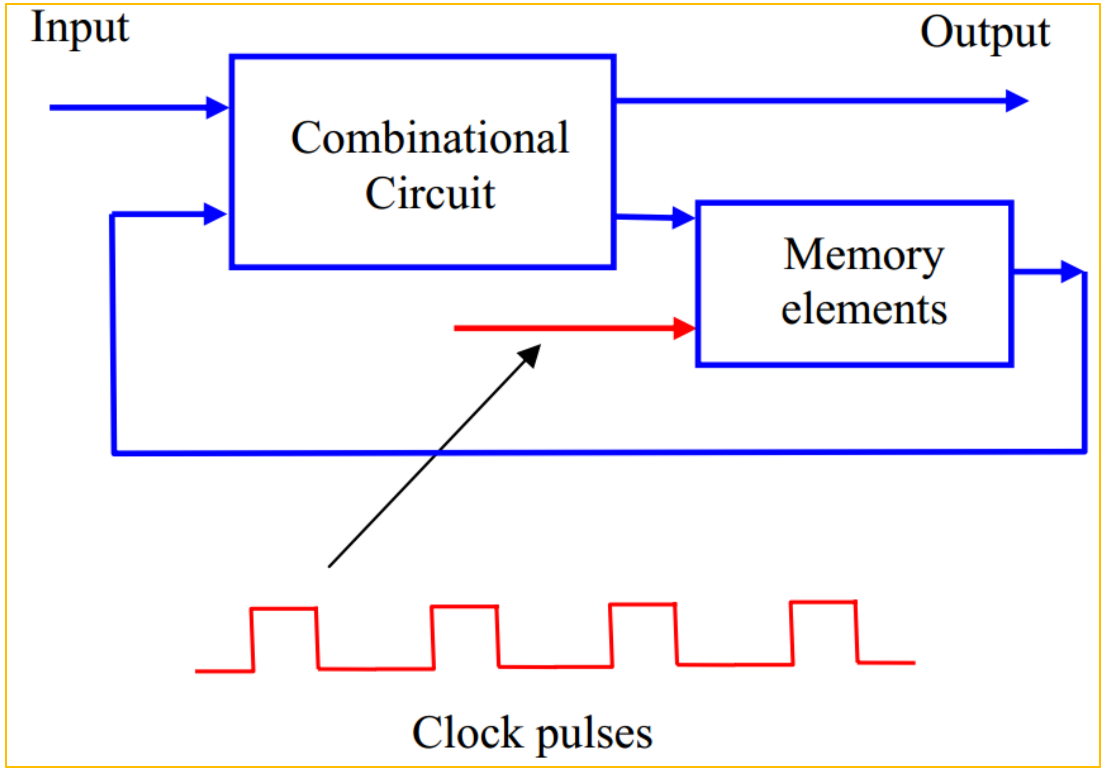
- **Asynchronous** Sequential Circuit
  - **No** clock signal
  - Changes in the memory elements can happen at **any** instance of time

- Our focus will be on **Synchronous** Sequential Circuits
  - **Easier** to design and analyze compared to asynchronous sequential circuits

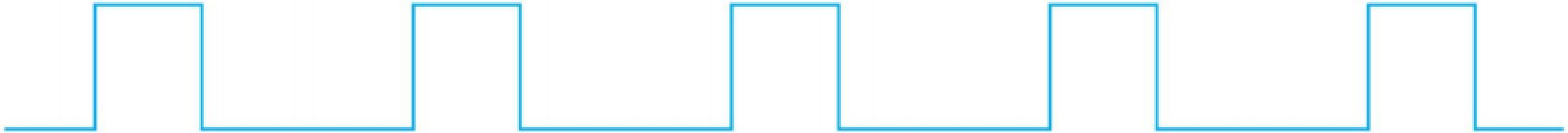
Synchronous Sequential Circuits

- Synchronous sequential circuits use a **clock** signal
- The clock signal is an **input** to the **memory** elements
- The clock **determines** when the **memory** should be **updated**
- The **present state** = **output** value of memory (**stored**)
- The **next state** = **input** value to memory (**NOT stored** yet)
- **Q(t) → present state**
- **Q(t+1) → next state**

Block Diagram of **Synchronous** Sequential Circuits

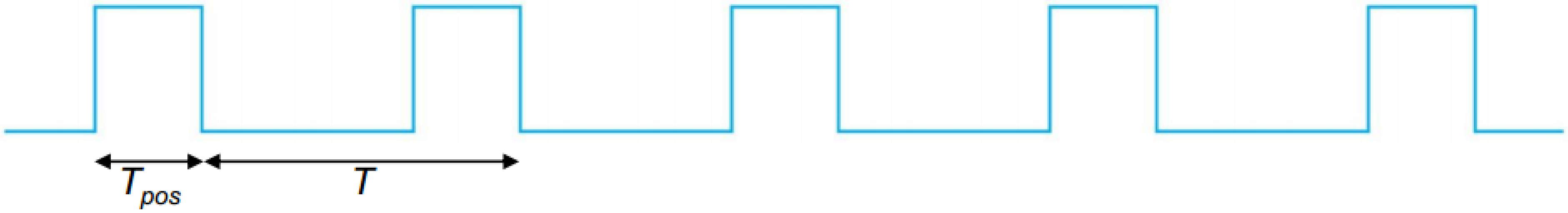


Clock (CLK)



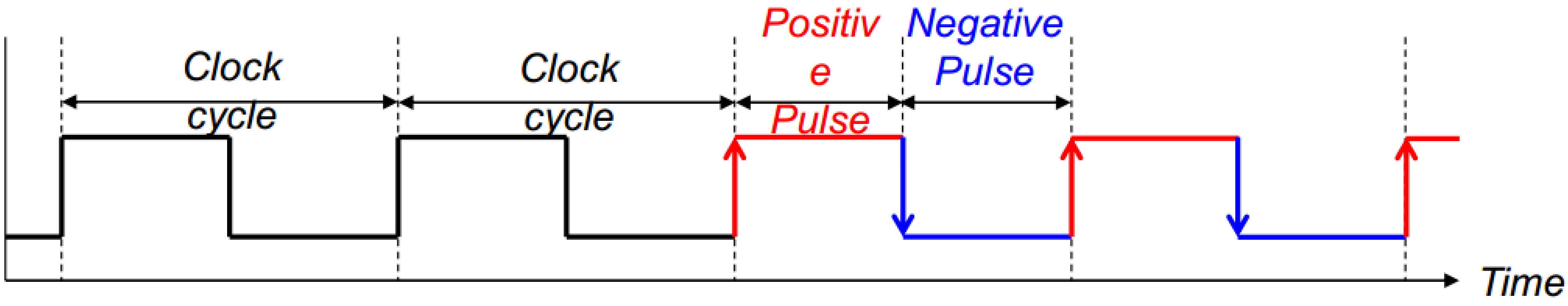
- Clock generator provides a clock signal having the form of a **periodic** train of clock **pulses**
- The clock determine **when** **computational activity will occur** in the circuit
  - (Transition from 0→1, or 1→0)
- The other signals (inputs and current state) determine **what** **changes** will take place affecting the **storage** elements and the **outputs**
- **Synchronous** sequential circuits that **use** **clock pulses** to control storage elements are called **clocked sequential circuits**

Clock (CLK)



- **T_{pos}** = Time of the **positive** portion of the clock
- **T** = **Duration** of a **complete** cycle
- **Duty** Cycle = **T_{pos} /T**
  - 50% **duty** cycle divides the clock period into **half positive** and **half** zero (**negative**)
- The **frequency** of the clock is **F=1/T**
  - Example: clock period (T) is **1ns** → frequency (F) is **1GHz**

Clock (CLK)



- Clock is a **periodic** signal = Train of **pulses** (1's and 0's)
  - The **same** clock **cycle** **repeats** **indefinitely** over time
- **Positive** Pulse: when the level of the clock is **1**
- **Negative** Pulse: when the level of the clock is **0**
- **Rising** Edge (**Positive** Edge): when the clock **goes** from **0** to **1**
- **Falling** Edge (**Negative** Edge): when the clock **goes** from **1** down to **0**

Memory/Storage Elements

- Memory elements can **store and maintain** binary states (0's or 1's)
  - Until directed by an **input** signal to **change state** Or the **power** source is lost

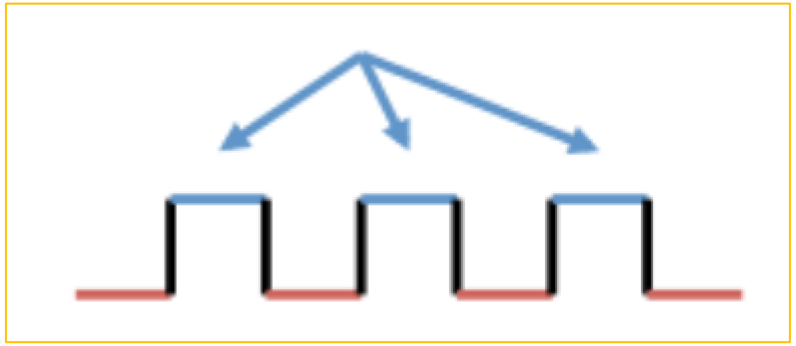
- Main differences between memory elements are the **number** of **inputs** and **how** they **change** state

- **Two** main **types**:
  - **Latches** are **level-sensitive** (sensitive to the **level** of the clock)
  - **Flip-Flops (FF)** are **edge-sensitive** (sensitive to the **edge** of the clock – **The Transition**)

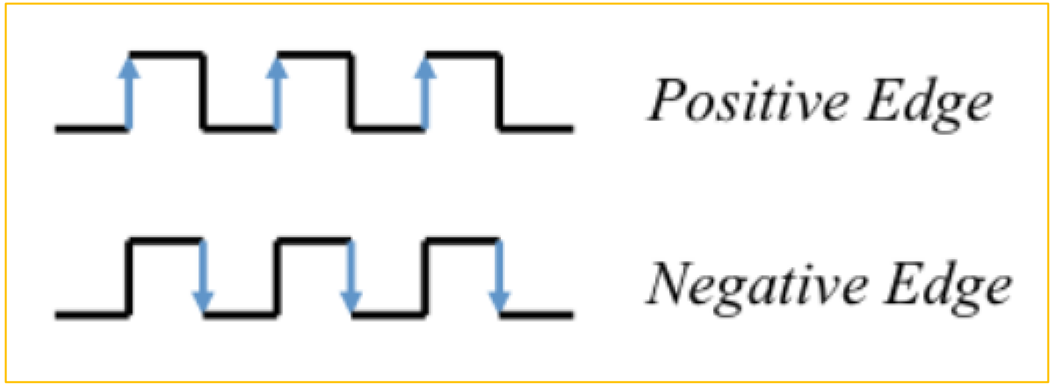
- **Flip-Flips (FF)** are the basic storage elements in **synchronous** sequential circuits
- **A Flip-Flop (FF)** is made up of a **latch/latches** with **clock** control

One latch or flip-flop can **store one** bit of information

Four main types of latches and flip-flops: **SR**, **D**, **JK**, and **T**



Latches



Flip Flops

Latches

- A **latch** is a temporary binary storage element that can store 0 or 1
  - Two **stable** states – **Bi-stable**
  - Can "**remember**" **one** bit of information
- **Latches** are the **basic** building blocks of more practical types of flip-flops
- **Feedback** connection – **outputs** are **connected back** to the **inputs**
- Easily **constructed** from a **pair** of **NOR** gates or a **pair** of **NAND** gates
- Two types: **SR** Latch, **D** Latch

SR Latches

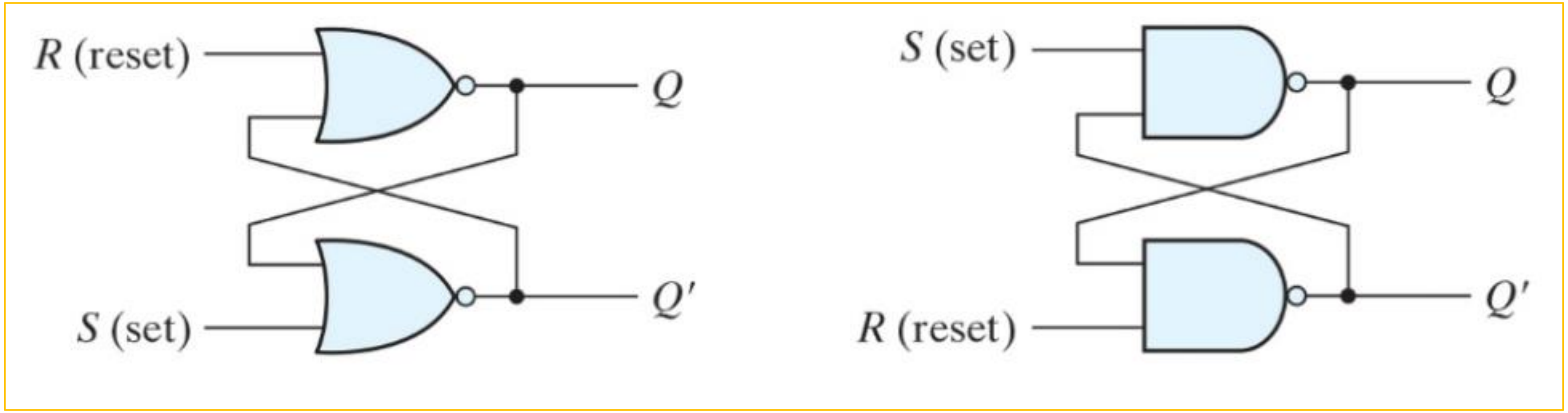
Two inputs: S (Set) and R (Reset)

Two outputs: Q and $\overline{Q}$

- **SR** stands for **Set-Reset**
- Two types of SR latches:
  - Active-**HIGH** input SR latch (two cross-coupled **NOR** gates)
  - Active-**LOW** input SR latch (two cross-coupled **NAND** gates)
- **Output** of each gate is connected to an **input** of the **opposite** gate

R → Q

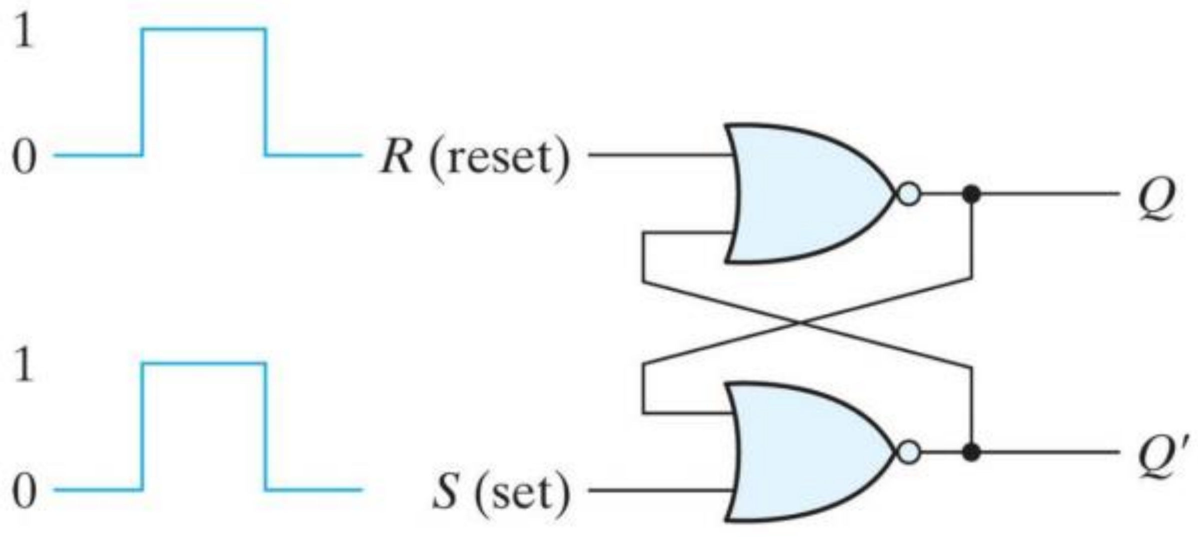
S → Q



NOR SR

NAND SR

NOR SR Latch



(a) Logic diagram

S	R	Q	Q'	
1	0	1	0	**SET**
0	0	1	0	(after $S = 1, R = 0$)
0	1	0	1	**RESET**
0	0	0	1	(after $S = 0, R = 1$)
1	1	0	0	(forbidden)

(b) Function table

If S=1 and R=0 then Set ($Q = 1,\ \overline{Q} = 0$)

If S=0 and R=1 then Reset ($Q = 0,\ \overline{Q} = 1$)

When $S = R = 0$, Q and $\overline{Q}$ are unchanged

The latch stores its outputs Q and $\overline{Q}$ as long as S=R=0

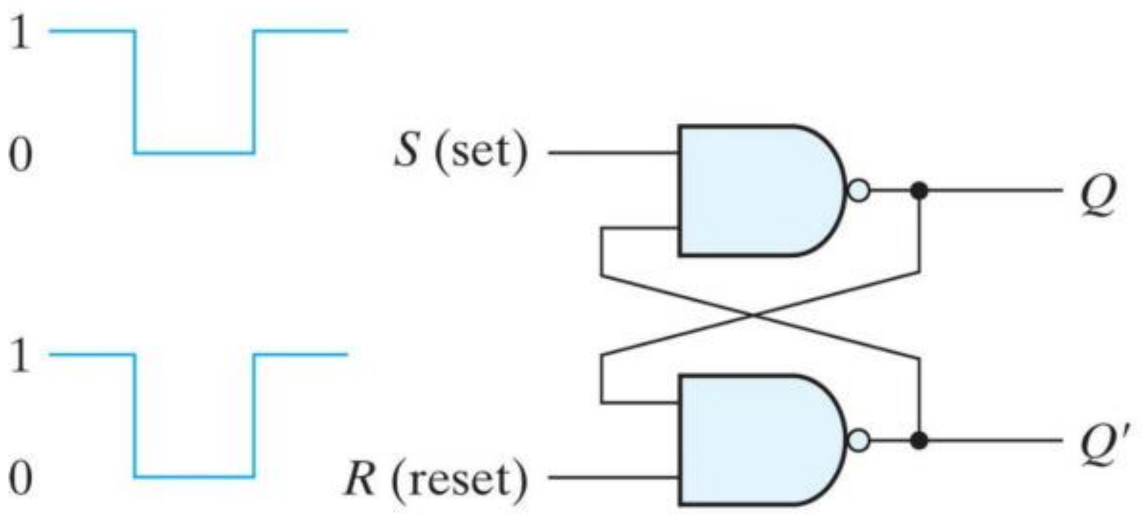
When S=R=1, Q and $\overline{Q}$ are undefined (should never be used)

Active-High → **Set** when **S=1**
Reset when **R=1**

Set State → Q = **1**
Reset State → Q = **0**

In **normal** operations, the outputs Q and Q′ are always **complement** of each other

NAND SR Latch



(a) Logic diagram

S	R	Q	Q′	
1	0	0	1	**RESET**
1	1	0	1	(after $S = 1, R = 0$)
0	1	1	0	**SET**
1	1	1	0	(after $S = 0, R = 1$)
0	0	1	1	(forbidden)

(b) Function table

If $S = 0$ and $R = 1$ then Set ($Q = 1,\ \overline{Q} = 0$)

If $S = 1$ and $R = 0$ then Reset ($Q = 0,\ \overline{Q} = 1$)

When $S = R = 1$, Q and $\overline{Q}$ are unchanged (remain the same)

The latch stores its outputs Q and $\overline{Q}$ as long as $\bar{S} = \bar{R} = 1$

When $S = R = 0$, Q and $\overline{Q}$ are undefined (should never be used)

Active-Low → **Set** when **S=0**
Reset when **R=0**

Set State → Q = **1**
Reset State → Q = **0**

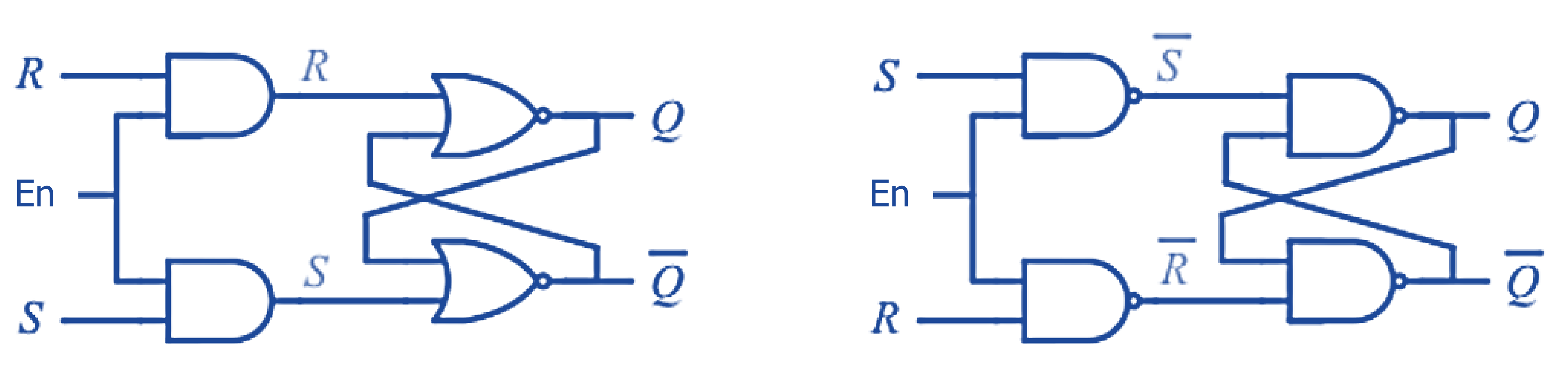
In **normal** operations, the outputs Q and Q′ are always **complement** of each other

SR Latch with Enable Input – Gated SR Latch

- An additional **Enable/Control** input signal **En** is used

- Enable **controls** **when** the state of the latch can be **changed**
  1) En=**0** → S and R inputs have **NO** effect on the latch [**Latch is Disabled**]
     - ✪ The latch will remain in the same state, regardless of S and R
  2) En=**1** → **normal** SR latch operation [**Latch is Enabled**]

- **NOR** SR → **AND** Gates are used with En
- **NAND** SR → **NAND** Gates are used with En [**Inverted Behavior**]

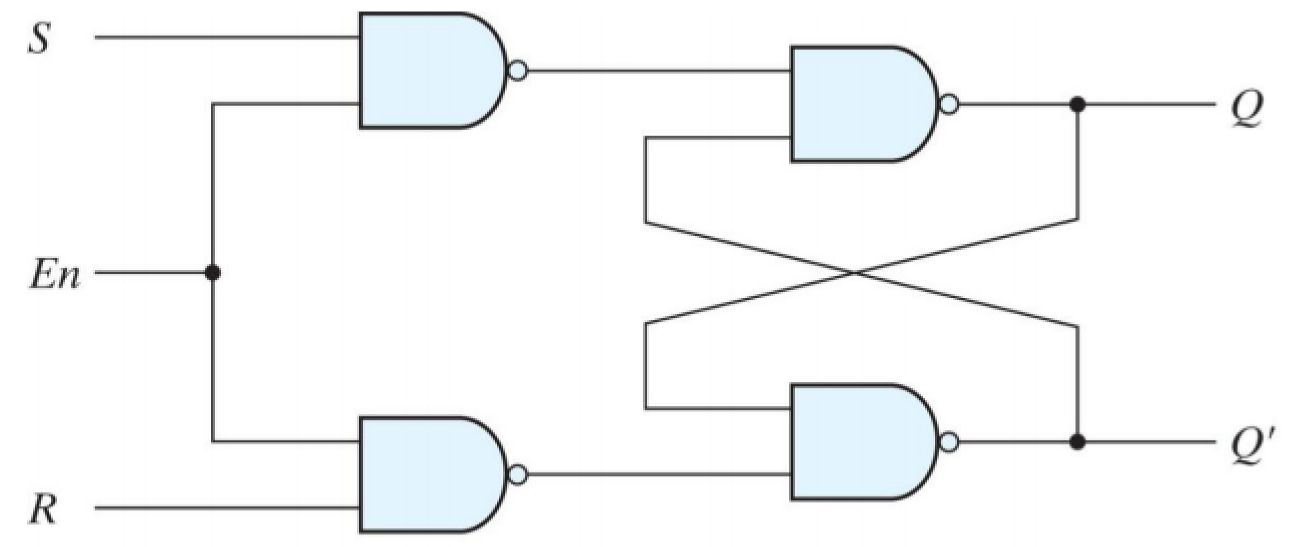


En = 0 → Inputs to the NOR SR = 00 → NO Change

En = 0 → Inputs to the NAND SR (Right NAND) = 11 → NO Change

SR Latch with Enable Input – Gated SR Latch

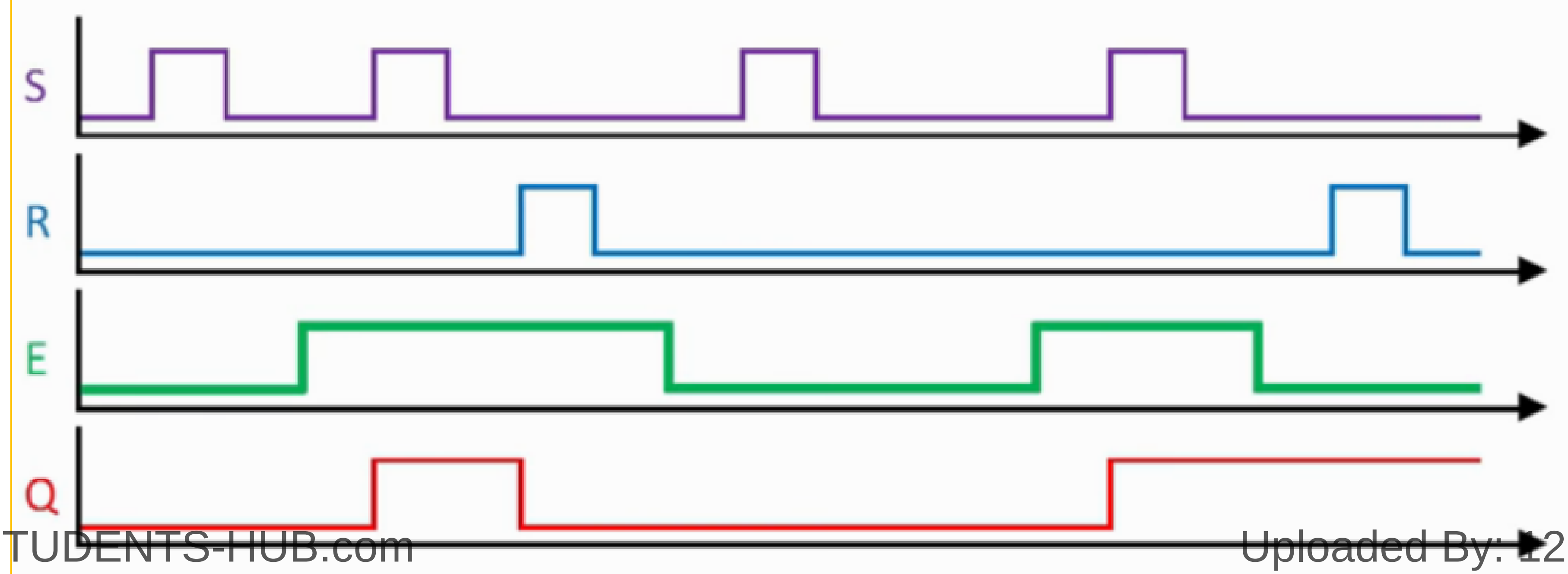
Be Careful:
S & R is **Complemented** (Left NAND) → **Inverted** Behavior for the latch



(a) Logic diagram

En	*S*	*R*	Next state of *Q*
0	X	X	No change
1	0	0	No change
1	0	1	$Q = 0$; reset state
1	1	0	$Q = 1$; set state
1	1	1	Indeterminate

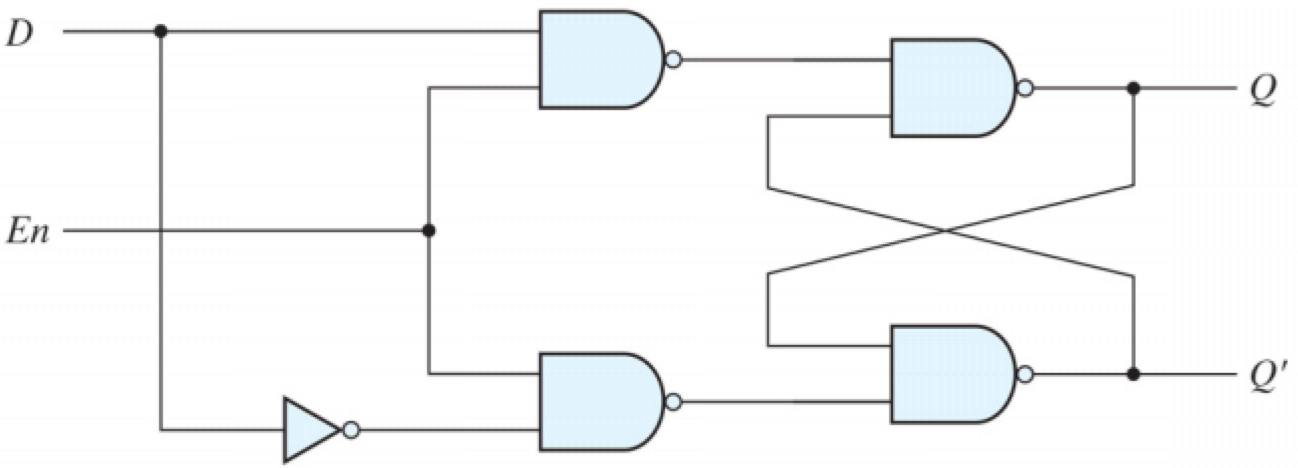
(b) Function table



Timing Diagram Example

D Latch

- All SR Latches have One Major **Issue** → Undefined/Indeterminant/Forbidden **State**
- This **undefined** state only exists when **both** inputs (S,R) are equal to 1/0 (depends on the used gates)
- A solution is to make sure that **both** inputs are **NOT** 1/0 at the **same time**, is to have a **single input only (D)** → **D Latch**
  - Only one **data input** D
  - An **inverter** is added: $\boldsymbol{S=D}$ and $\boldsymbol{R=D'}$
  - S and R can **never** be 11/00 **simultaneously** → **No undefined state**
  - When En=0, Q remains the same (No change in state), When En=1, $Q=D$ and $Q'=D'$
  - It is also called a **transparent** latch because whatever appears at the D input follows at the Q output (**Q=D**)

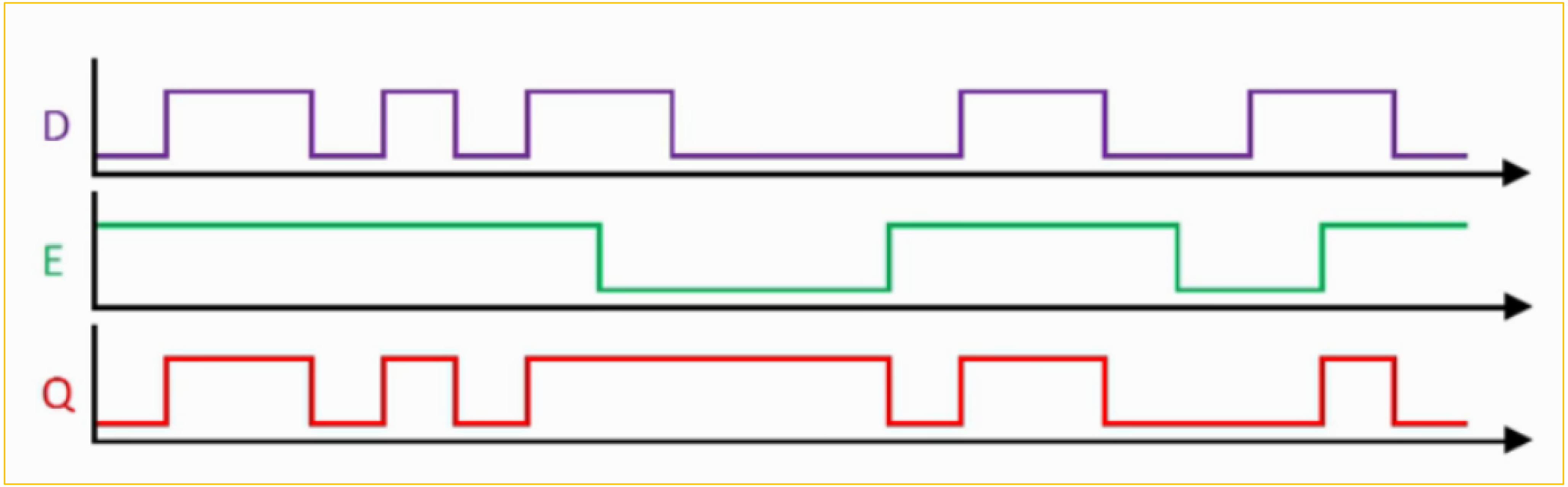


(a) Logic diagram

En	D	Next state of Q
0	X	No change
1	0	$Q = 0$; reset state
1	1	$Q = 1$; set state

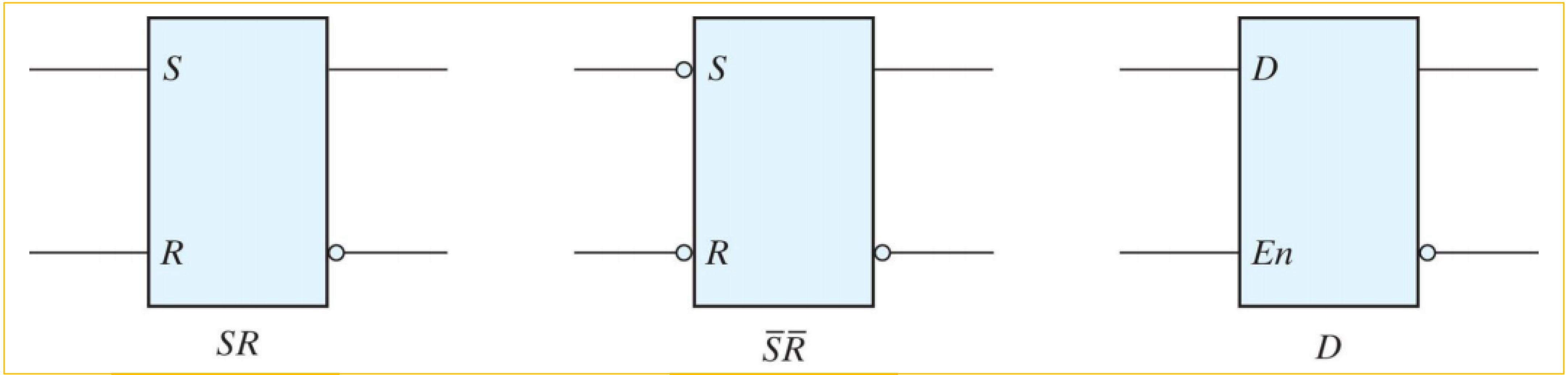
(b) Function table

D Latch



- ✪ Q **follows** D
- ✪ When D = 1, Q becomes 1 and so on
- ✪ However, if the enable input E is 0, **Q will retain its state** (Q will not change, it will not follow D)
- ✪ In short, if a D latch is **disabled** → it can **store/retain** one **data** bit (0 or 1)

Graphic Symbols for Latches



Active-**HIGH**

Active-**Low**

Be Careful:
Bubble @ **Output** → **Inverter** → Q'
Bubble @ **Input** → Active-**Low**

Latches Summary

- SR Latch
  - SR latch with NOR gates (active **HIGH**)
  - SR latch with NAND gates (active **LOW**)
  - SR latch with enable/control input (**Gated** SR Latch)
- D Latch (**Transparent** Latch)

Problem with Latches

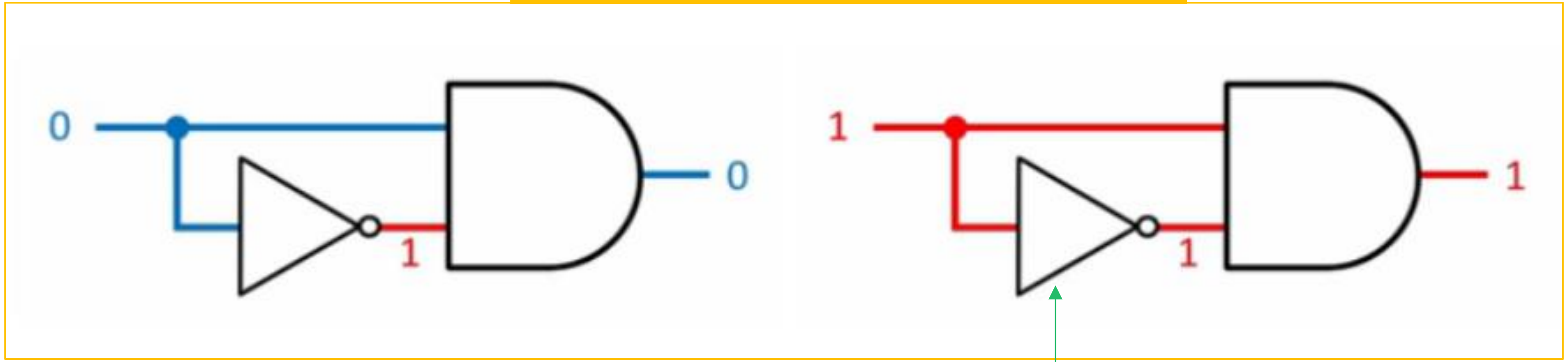
- A latch is **level-sensitive** (sensitive to the **level** of the **En**)
- As long as the **enable** signal (control) is **high** then
  - **Any change** in the value of **input** D **appears** in the **output** Q
- Output Q **keeps changing** its **value** as long as the enable is **activated (high)**
  - **Final** value of output Q is **uncertain**
- Due to this **uncertainty**, latches are **NOT** used as memory elements in **synchronous** circuits
  - Used mainly in **Asynchronous** circuits

- To overcome these issues, a **synchronized** latch is needed **Flip-Flop (FF)**

1) **A Clock** Signal is connected to the **En** input → **Clocked** Latch (Synchronized Level-Triggering)
   - The data might **change** while the clock is **HIGH**
2) **Construct** the clock in a way that **enable** the latch/FF for a **very small amount** of time
   - At signal **transition only** [clock edge] → **Edge-Triggering** (Positive or Negative)
   - Edge **detection** circuit is needed

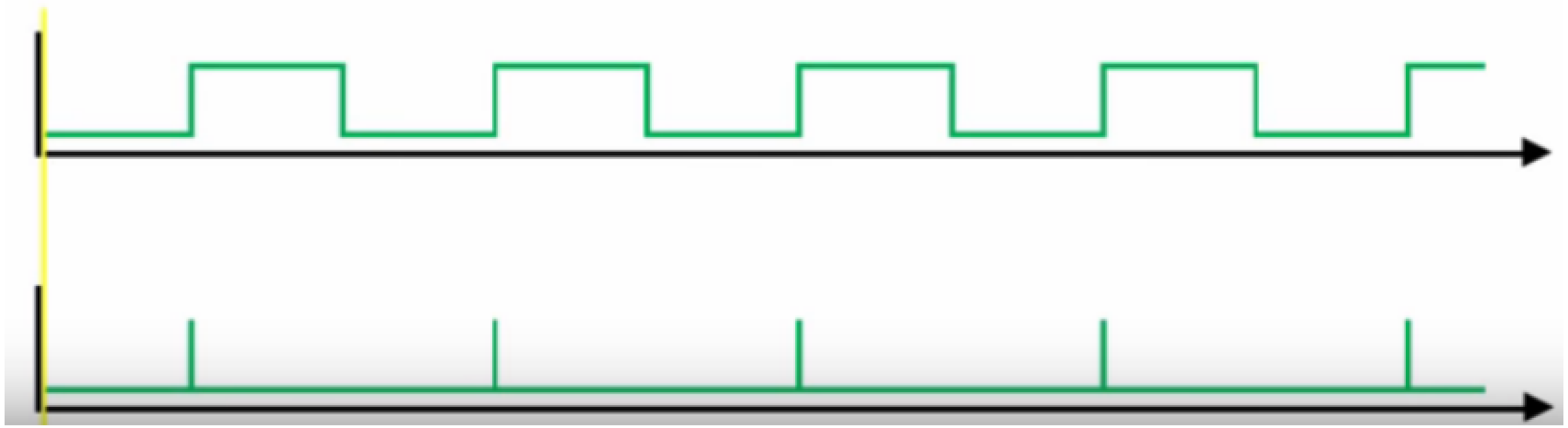
Flip-Flop is a synchronized **Edge Triggered** Latch with Edge detection circuit **connected** to its Enable input

Positive (+ve) Edge Detector



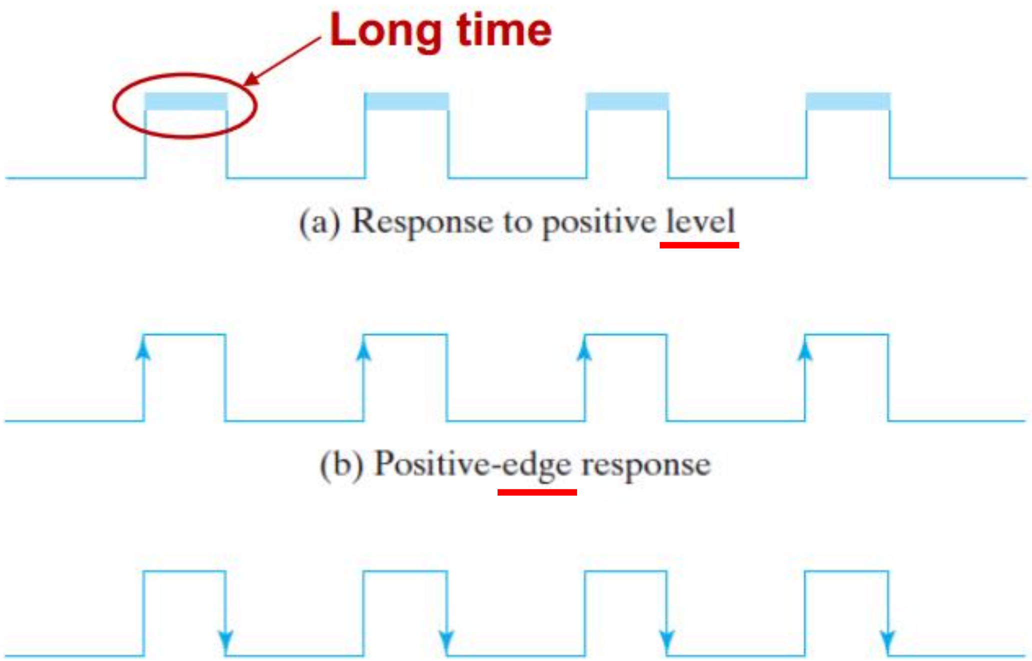
Delay Gate

Due to the **Delay** of the **inverter** gate → the output is **1** for a very **small** time → **Edge detection**



Latches to Flip-Flops (FF): Level-Trigger to Edge-Trigger

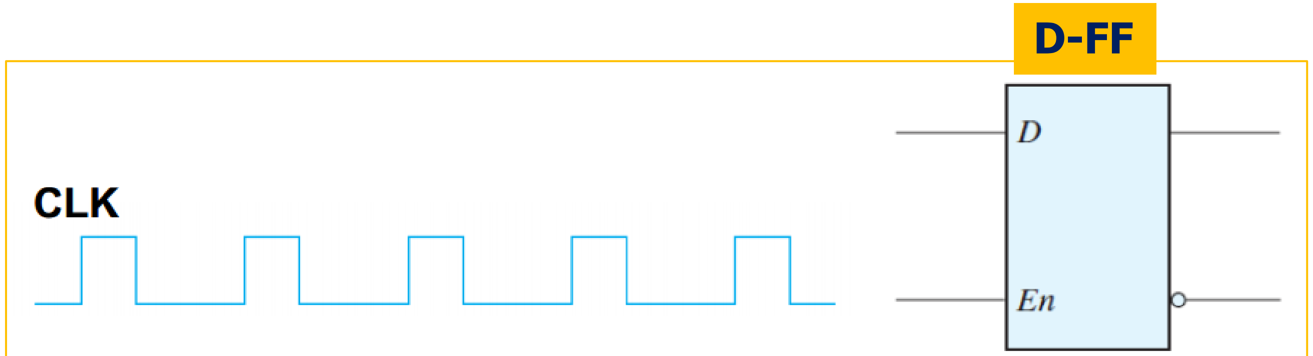
- **Recall**: In **Level-Trigger** elements (latches), As long as the control (En) is **high** → output **may** change state.
- To overcome this, we use **Edge Trigger** Control → output will change **only** when a 0→1 or 1→0 **transition** occurs
- To achieve **Edge Trigger** → a clock (CLK) signal is used to drive the control (enable) input of a latch
    - This clock will **trigger** the flip-flop to **change** state only at the **transition**



(a) Response to positive level

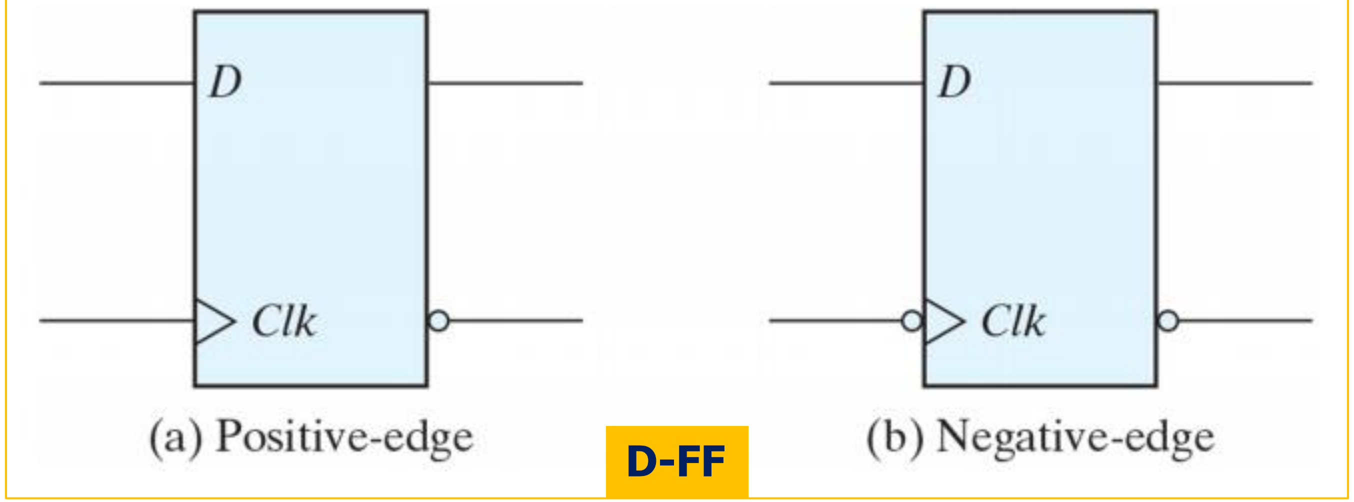
(b) Positive-edge response

(c) Negative-edge response



Edge-Triggered D Latch (D Flip-Flop)

- We connect the **edge-detection** circuit at the **enable** input of the D latch
  - This converts it to an **edge-triggered** D latch
  - This new circuit is also called an Edge-Triggered **D Flip Flop (D-FF)**
- We usually specify it as
  - a **positive-edge** triggered flip-flop
  - OR as a **negative-edge** triggered flip-flop



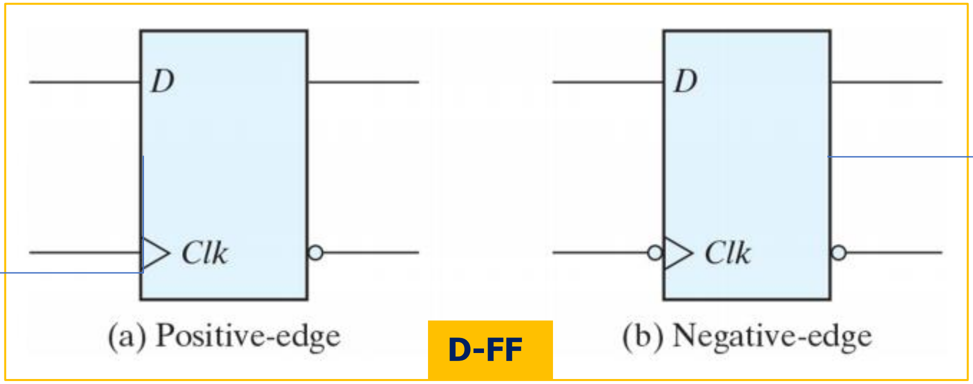
(a) Positive-edge

(b) Negative-edge

D-FF

- The **dynamic** indicator (**>**) denotes that flip-flop responds to the edge **transition** of the clock
- A **bubble** adjacent to the dynamic indicator denotes that it is a **negative-edge** triggered flip-flop

Edge-Triggered D Latch (D Flip-Flop)



- When the input **clock (Clk)** makes a **positive** transition (i.e., moves from 0 to 1), the value of D is **transferred** to Q
- A **negative** edge of the clock does **NOT affect** the output
- The output is **NOT affected** by changes in D when the **Clk** is in the **steady** logic-1 **level** or the logic-0 **level**
- This type of flip-flop responds to the **transition** from **0 to 1** and **nothing else**

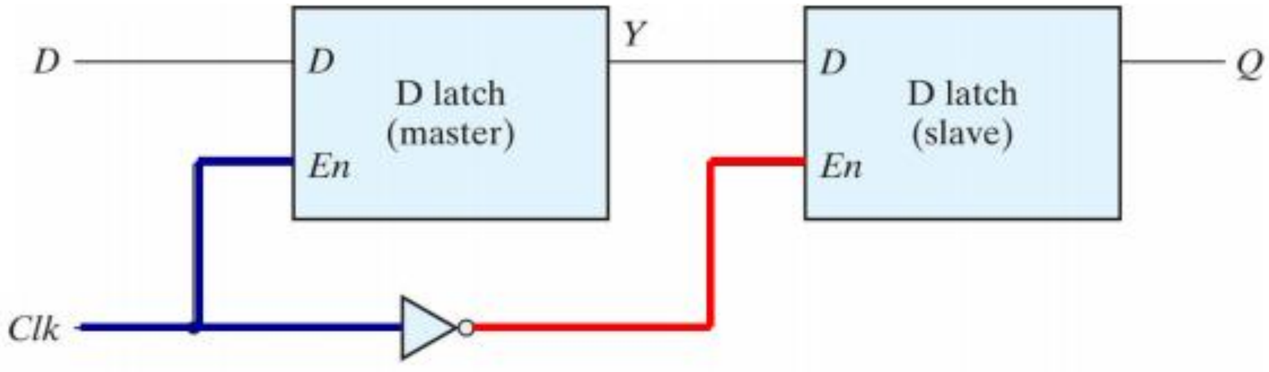
- When the input **clock (Clk)** makes a **negative** transition (i.e., moves from 1 to 0), the value of D is **transferred** to Q
- A **positive** edge of the clock does **NOT affect** the output
- The output is **NOT affected** by changes in D when the **Clk** is in the **steady** logic-1 **level** or the logic-0 **level**
- This type of flip-flop responds to the **transition** from **1 to 0** and **nothing else**

Master-Slave D-FF (Negative Edge-Triggered)

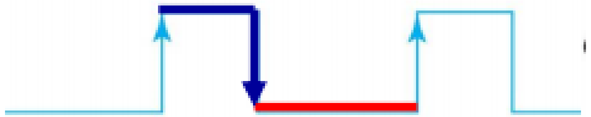
- Common Design/Configuration of the Negative Edge-Triggered D Flip-Flop

- Built using **two** latches in a **master-slave** configuration
  - A **master** latch (D-type) receives **external** inputs
  - A **slave** latch (D-type) receives **inputs** from the **master** latch

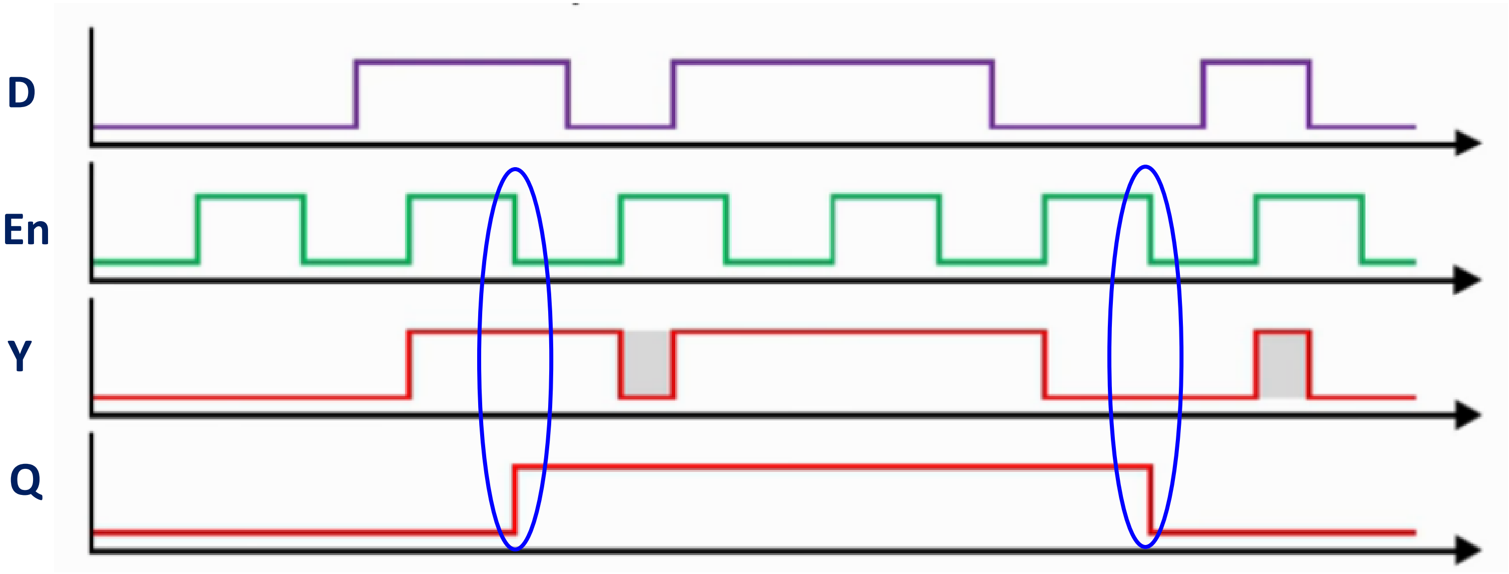
- Only **ONE** latch is **enabled** at **any** given time
  - When **Clk=1**, the **master is enabled** and the D input is latched (**slave is disabled**)
  - When **Clk=0**, the **slave is enabled** to generate the outputs (**master is disabled**)



Change in the FF output can be triggered only by the transition of the clock from 1 to 0



Master-Slave D-FF (Negative Edge-Triggered)



- ✓ **Master** is enabled during positive CLK period.
- ✓ **Slave** is enabled during negative CLK period.
- ✓ D → Y during + ve CLK period, while Q is NOT changed.
- ✓ Y → Q during – ve CLK period, while Y is NOT changed

- The behavior of the master–slave flip-flop dictates that
  - The output may change only **once**
  - A change in the output is **triggered** by the **edge** of the clock
  - The change may occur only during the clock's **negative** level
- The value that is **produced** at the **output** of the flip -flop is the value that was **stored** in the **master** stage **immediately** before the **negative** edge occurred

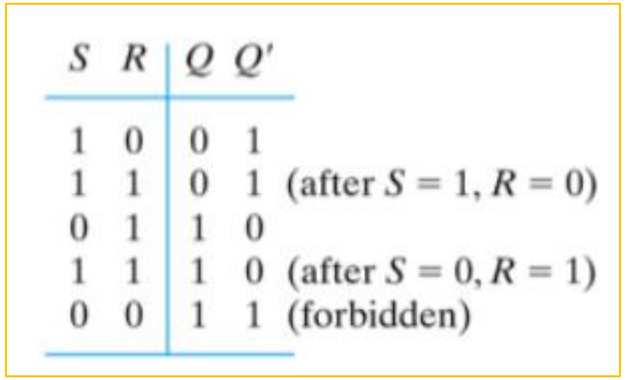
Positive Edge-Triggered D-FF (3 SR Latches)

- **Two** of the three SR latches responds to **CLK** and **D** inputs, the **third** provides **output**

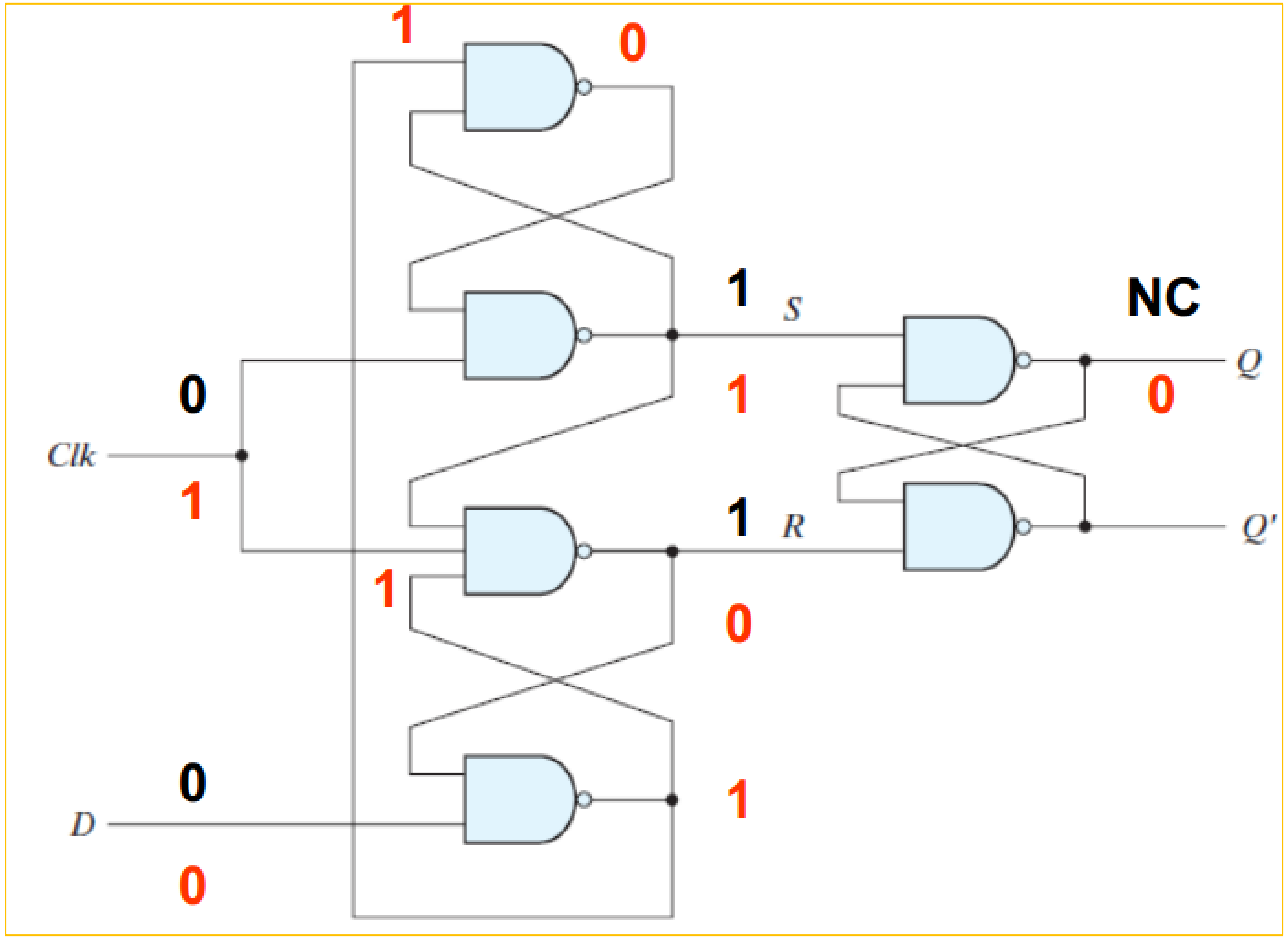
- CLK=0 → Q(t+1)=Q(t) (**NC**: NO Change)
- CLK=1 & D=0 → Q=0
- CLK=1 & D=1 → Q=1

- If D changes after CLK becomes 1, the output is NOT changed

- Change only on CLK transition from 0 → 1, and no other change



S	R	Q	Q'	
1	0	0	1	
1	1	0	1	(after $S = 1, R = 0$)
0	1	1	0	
1	1	1	0	(after $S = 0, R = 1$)
0	0	1	1	(forbidden)

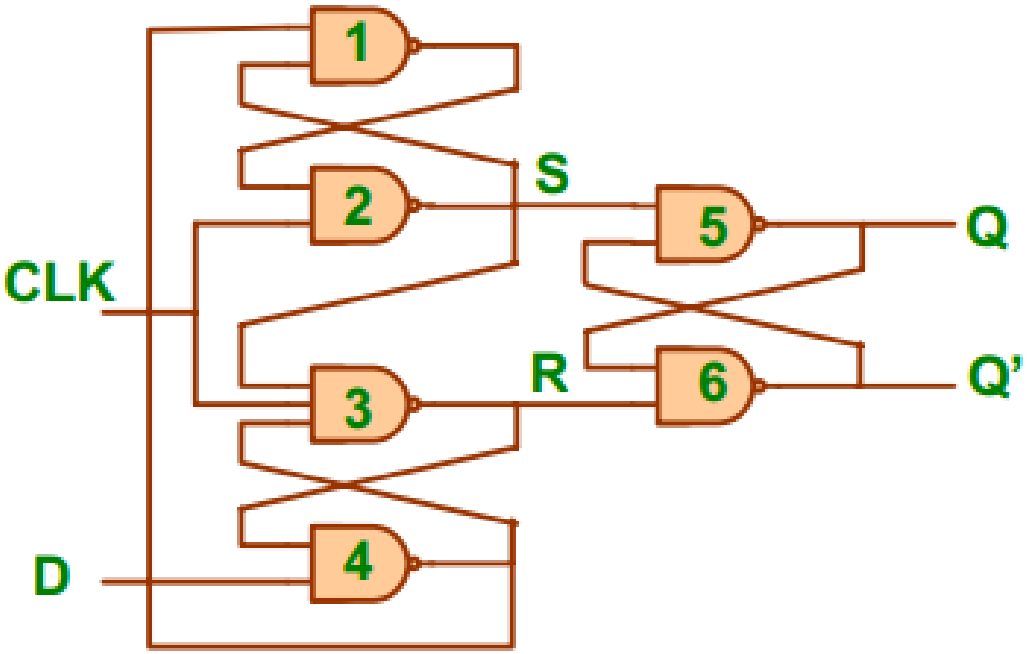


Positive Edge-Triggered D-FF (3 SR Latches)

- When CLK=0 and D=0, then the outputs of gates 1, 2, 3, and 4 are going to be 0111.
- When CLK=0 and D=1, then the outputs of these gates are going to be 1110 instead.
- In both cases, S=R=1 → No Change

- Suppose that CLK becomes 1 while D=0. The output of gate **3** (which is **R**) becomes **0**. this will **reset** the output flip-flop.
- Once R is 0, then **D can change to 1 and R remains 0**.
- This means that Q remains 0 while CLK is 1. No change will occur to Q until the clock returns to 0 and then goes to 1 on the next clock pulse. This scenario shows that the flip-flop is a positive edge triggered.

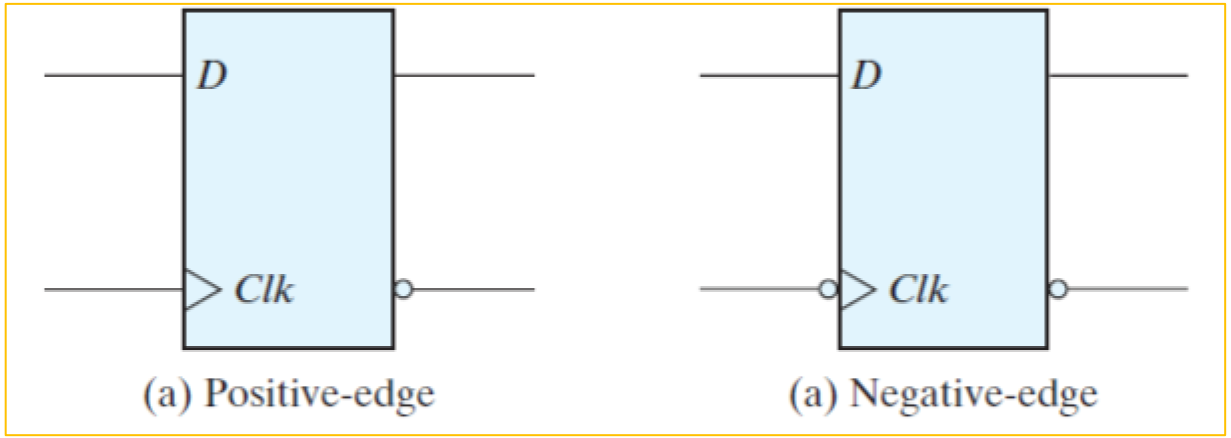
- Similar procedure occurs if D=1 while the clock goes from 0 to 1. In this case, S=0 and Q=1.



D-FF Summary

$$Q(t+1) = D$$

Characteristic Equation



(a) Positive-edge (a) Negative-edge

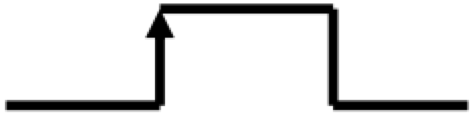
Graphic Symbol

D Flip-Flop

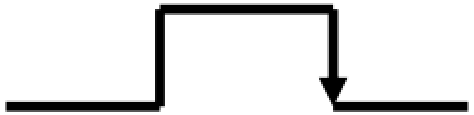
D	Q(t + 1)	
0	0	Reset
1	1	Set

Characteristic Table

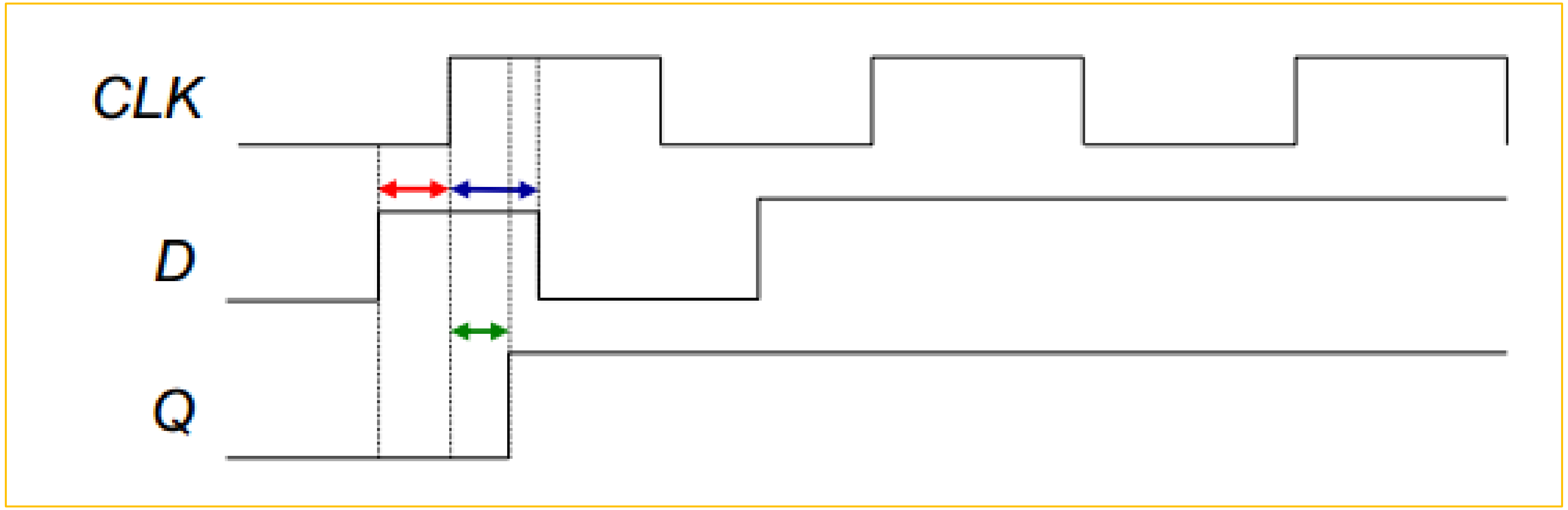
- In a positive edge triggered FF, output changes on CLK transition from 0 → 1



- In a negative edge triggered FF, output changes on CLK transition from 1 → 0



Timing Considerations



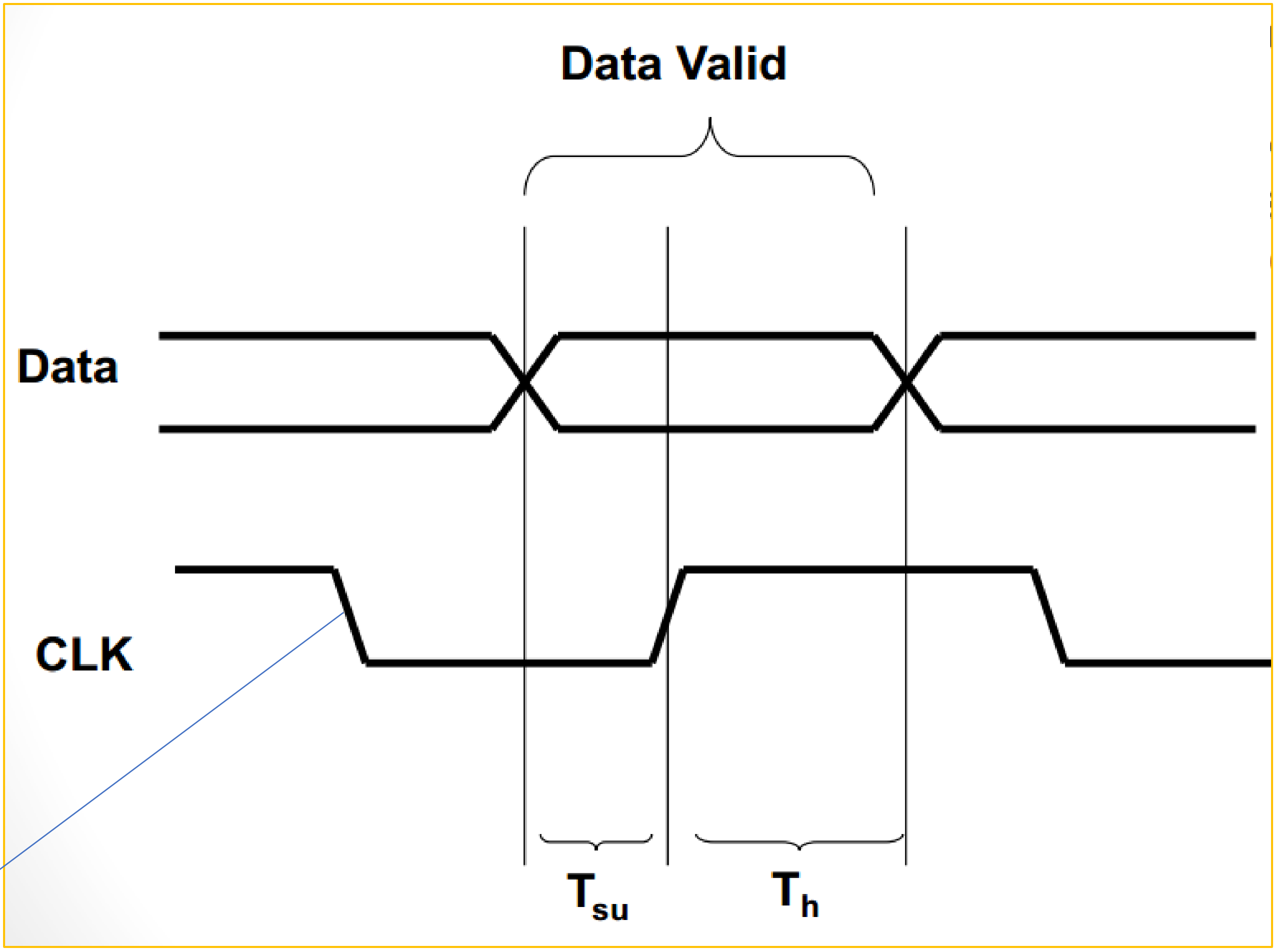
- **Setup Time (Ts):** There is a **minimum** time during which the D must be **valid** and **stable before** the clock edge.

- **Hold Time (Th):** there is a **minimum** time during which D must **not change after** the clock edge.

- **Propagation Delay:** the **interval** between the **trigger** edge and the **stabilization** of the **output** to a new state.

Timing Considerations

- If the setup and hold times are **violated**, a gate may produce an **unknown** logic signal at its output.
- This condition is called as **meta-stability**.



- In practical applications, the CLK (clock) signal, like all digital signals, requires a **finite** amount of **time** to transition between states and does **NOT** change **instantaneously**.
- For simplicity in theoretical studies, we assume that the clock signal transitions instantly, **neglecting** any transition **delays**.

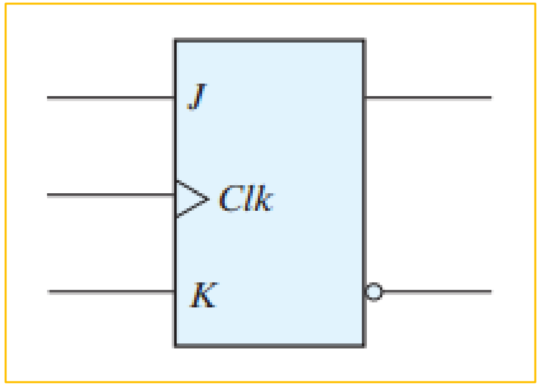
Other Flip-Flops

- The most **economical** and **efficient** FF is **D-FF** because it requires the least number of gates
- Other types of FF can be **constructed using** the **D-FF** and **external logic**
- Other used flip-flops are **JK-FF** and **T-FF**

JK Flip-Flop

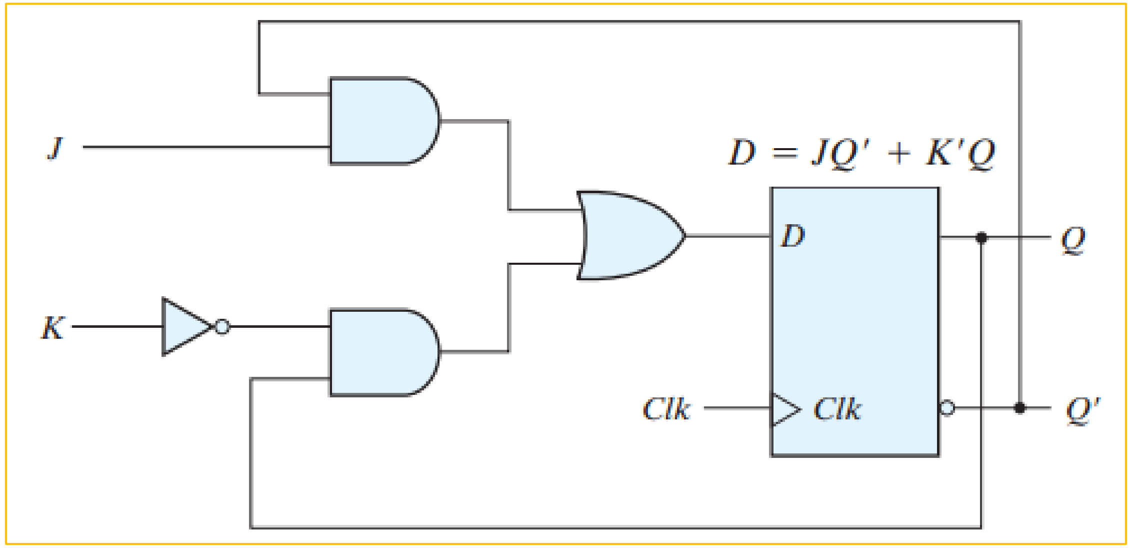
- The JK is another type of Flip-Flop with inputs: **J, K**, and Clk
- Typically constructed using the **D-FF** and **external gates**
- **Four** operations: **set** to 1, **reset** to 0, **no change**, & **invert** output
  1) When JK = 10 → Set
  2) When JK = 01 → Reset/Clear
  3) When JK = 11 → Invert/Complement output
  4) When JK = 00 → No change
- JK can be implemented using two Clocked SR latches and gates

Graphic Symbol

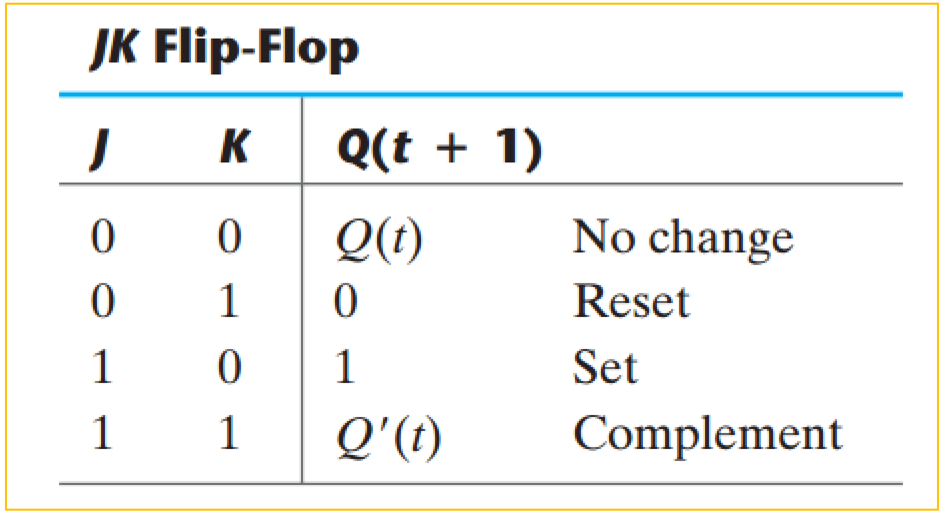


$$Q(t+1) = JQ' + K'Q$$

Characteristic Equation



Circuit Diagram (using D-FF)



JK Flip-Flop

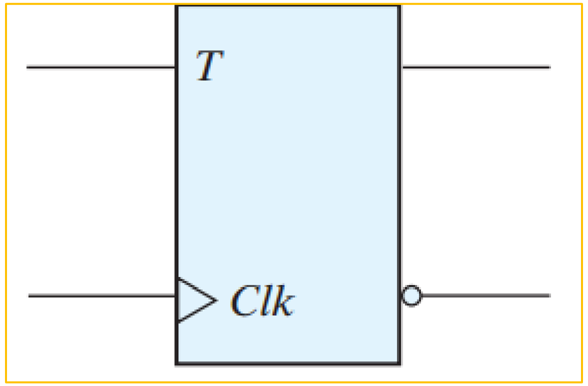
J	*K*	$Q(t+1)$	
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	$Q'(t)$	Complement

Characteristic Table

T (Toggle) Flip-Flop

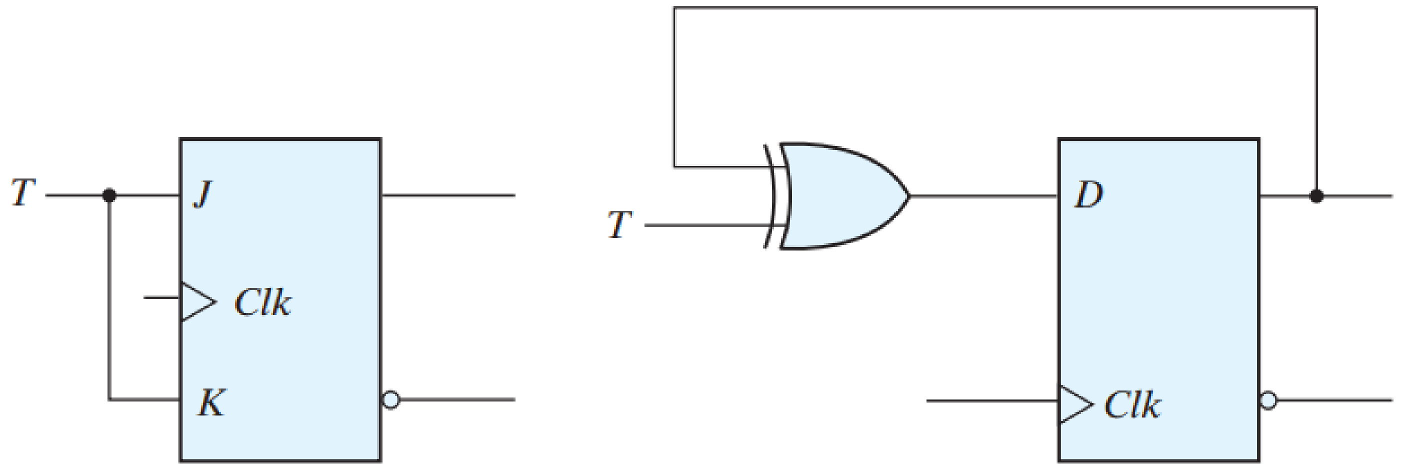
- The T (**Toggle**) flip-flop has inputs: **T** and Clk
- **Two** operations: **no change** and **invert** output
  1) When T = 1 → Invert/Complement output
  2) When T = 0 → No change
- Can be implemented using
  - JK-FF
  - Or D-FF and XOR gate

Graphic Symbol



$$Q(t+1) = T \oplus Q = TQ' + T'Q$$

Characteristic Equation



(a) From *JK* flip-flop

(b) From *D* flip-flop

Circuit Diagram

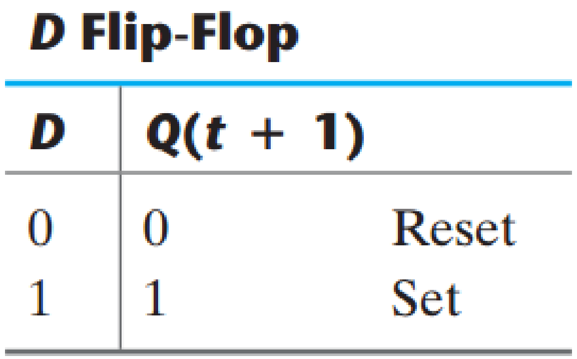
***T* Flip-Flop**

T	Q(*t* + 1)	
0	$Q(t)$	No change
1	$Q'(t)$	Complement

Characteristic Table

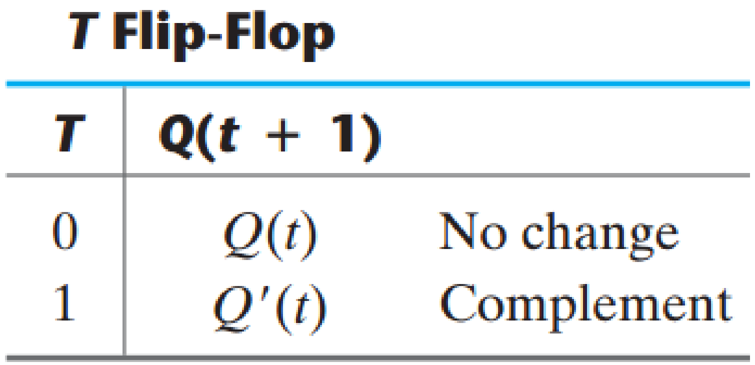
Flip-Flop Characteristic Tables/Equations

- **Characteristic Tables**: Defines the **operation** of a flip-flop in a tabular form
- **Next state** is defined in terms of the **current** state and the **inputs**
- $Q(t)$ refers to current state **before** the clock edge arrives
- $Q(t+1)$ refers to **next state** **after** the clock edge arrives



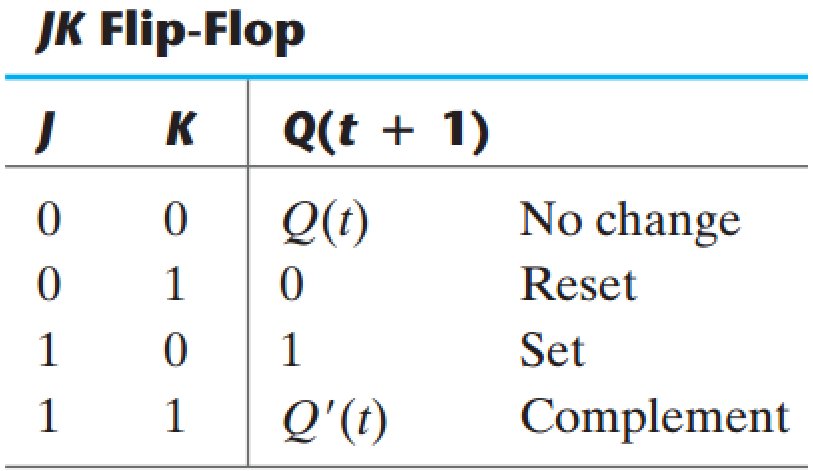
D Flip-Flop

D	$Q(t+1)$	
0	0	Reset
1	1	Set



T Flip-Flop

T	$Q(t+1)$	
0	$Q(t)$	No change
1	$Q'(t)$	Complement



JK Flip-Flop

J	K	$Q(t+1)$	
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	$Q'(t)$	Complement

- **The characteristic equation defines the operation of a flip-flop**

D-FF

$$Q(t+1) = D$$

T-FF

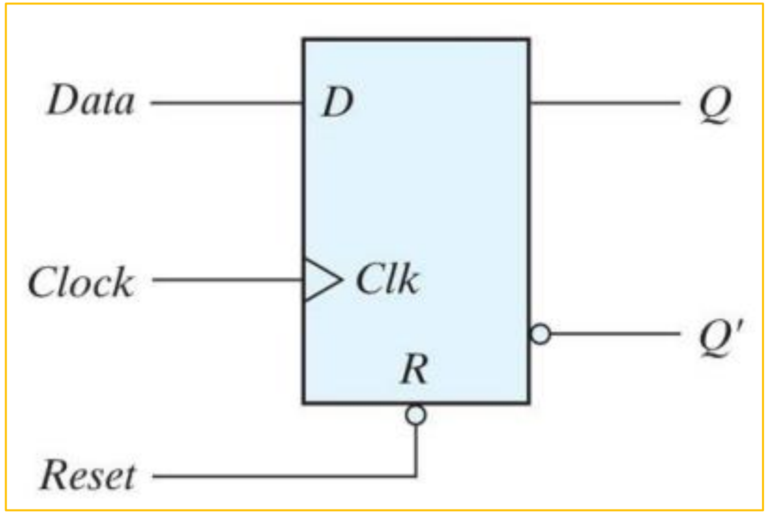
$$Q(t+1) = T \oplus Q = TQ' + T'Q$$

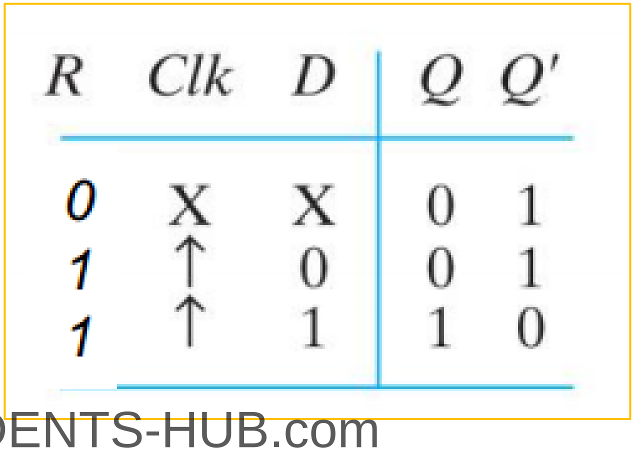
JK-FF

$$Q(t+1) = JQ' + K'Q$$

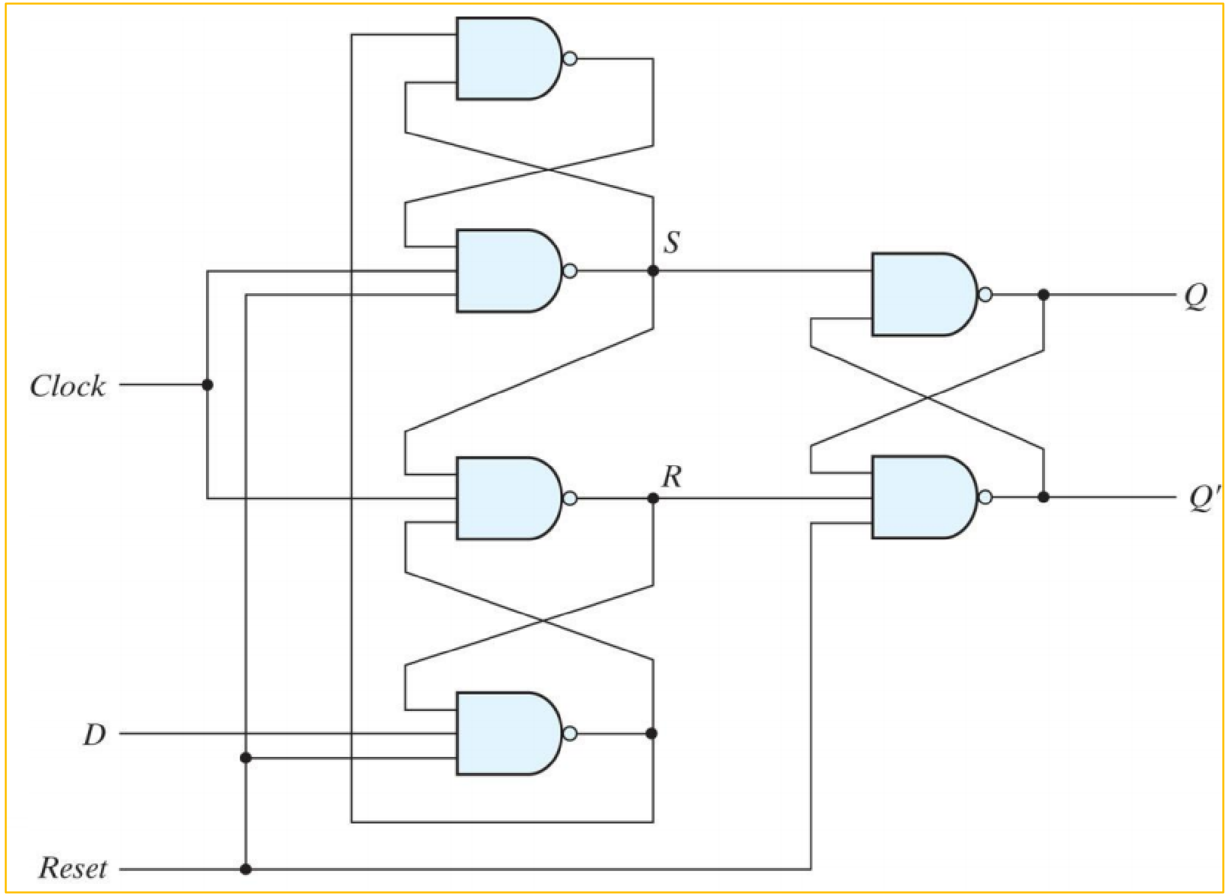
Direct Inputs (Asynchronous Inputs)

- **Direct** Inputs are **Asynchronous** inputs that force the FF to a particular state regardless of the **clock**.
- The Input (**S/PR**) that sets the Flip-Flop to 1 (Q=1) is called **preset** or **direct set**.
- The Input (**R/CLR**) that sets the Flip-Flop to 0 (Q=0) is called **clear** or **direct reset**.
- When Flip-Flops are powered, their **initial** state is **unknown**





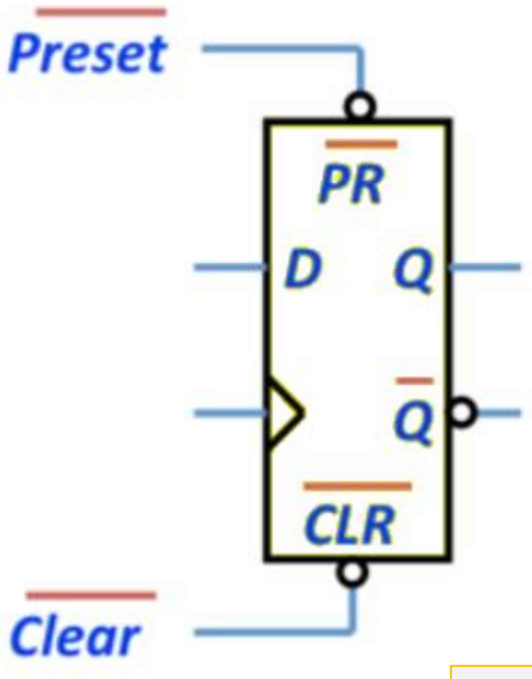
R	Clk	D	Q	Q'
0	X	X	0	1
1	↑	0	0	1
1	↑	1	1	0



Direct Inputs (Asynchronous Inputs)

- **Direct** Inputs are **Asynchronous** inputs that force the FF to a particular state regardless of the **clock**.
- The Input (**S/PR**) that sets the Flip-Flop to 1 (Q=1) is called **preset** or **direct set**.
- The Input (**R/CLR**) that sets the Flip-Flop to 0 (Q=0) is called **clear** or **direct reset**.
- When Flip-Flops are powered, their **initial** state is **unknown**

Notice: As the **Bubble** Indicates Both (PR, CLR) Are **Active Low**



PR'	CLR'	D	CLK	Q(t+1)
1	0	x	x	0
0	1	x	x	1
1	1	0	↑	0
1	1	1	↑	1

Clear

Set

Notice: Both (PR, CLR) **Can't Be 0** (Activated) at the **same time**

Analysis of Clocked Sequential Circuits

- **Recall**: **Analysis** is describing **what a given circuit do**
- A logic diagram is recognized as a **clocked sequential circuit** if it includes **flip-flops with clock input**
- The **behavior** of a clocked sequential circuit is determined from the **inputs**, the **outputs**, and the **state of its flip –flops**
- The **outputs** and the **next state (NS)** are both a function of the **inputs** and the **present** state (**PS**)
- The analysis of sequential circuits consists of:
  1) Determine the Circuit **Type** from the **Circuit/Logic Diagram** (**Sequential** or Combinational)
  2) Deriving **State Equations** for the next states of memory elements
  3) Obtaining a **State Table** for the time sequence of inputs, internal states and outputs of the circuit
  4) Obtaining a **State Diagram**

State Equations

- A **state equation** describes the **next state** as a function of the **present state and inputs**
  - Also called a **transition equation**

$$A(t+1) = A(t)x(t) + B(t)x(t)$$
$$B(t+1) = A'(t)x(t)$$

State Equations

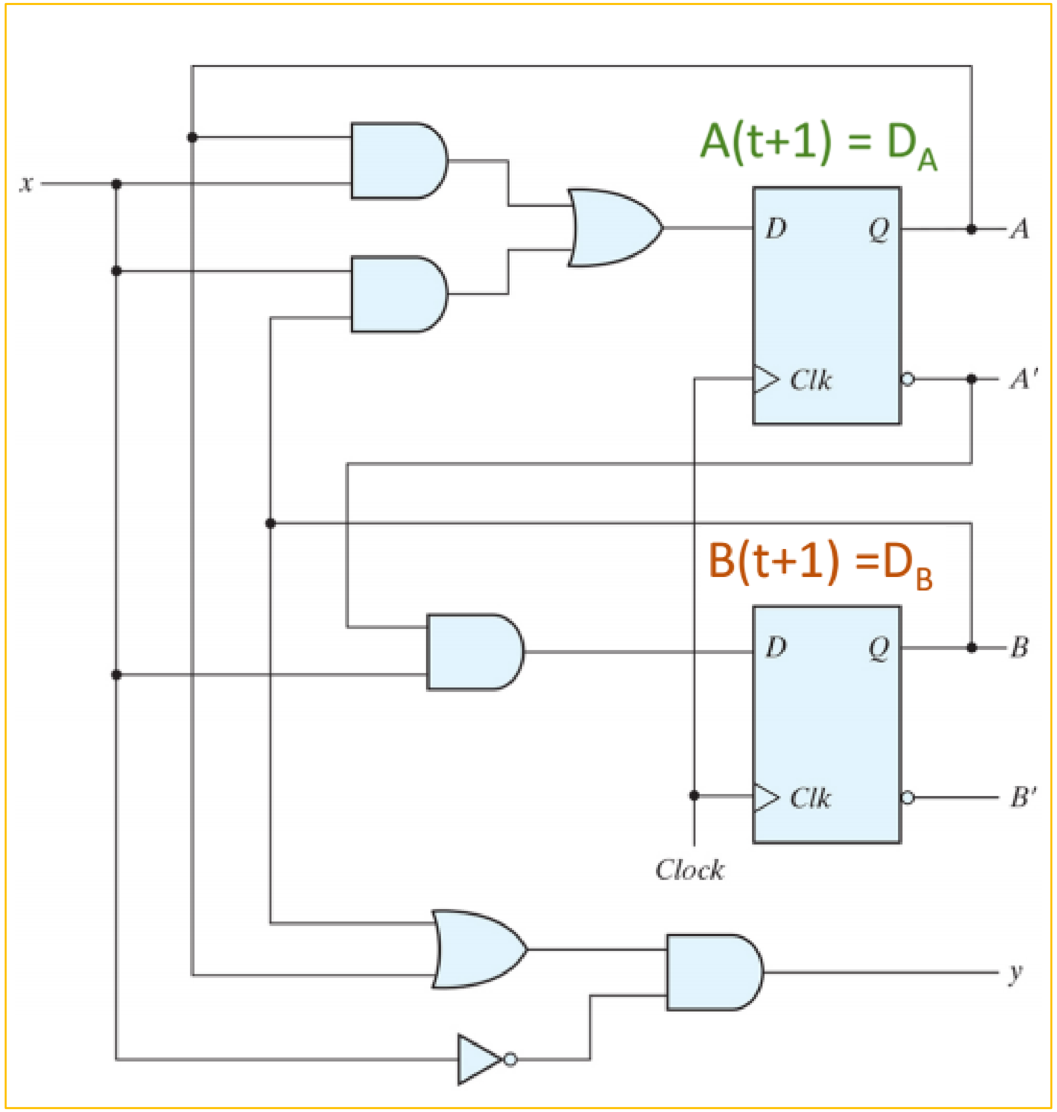
$$A(t+1) = Ax + Bx$$
$$B(t+1) = A'x$$

2 Flip-Flops → **2** State Equations

$$y(t) = [A(t) + B(t)]x'(t)$$

Output Expression

$$y(t) = (A + B)x'$$



Circuit Diagram

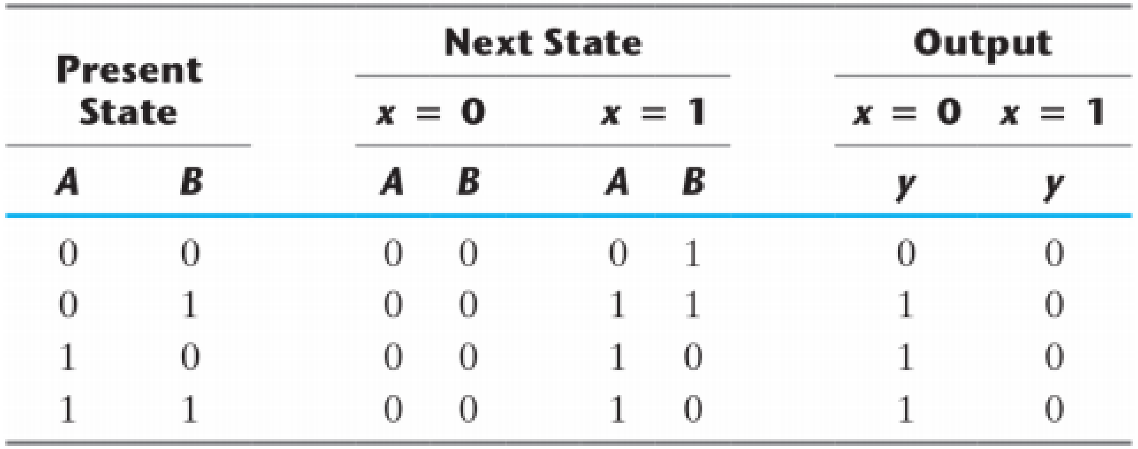
State Table

- The time sequence of inputs, outputs and states can be described using **state tables**
- **State tables** contain **four** sections:
  Present state, Input, Next State, Output
- Write all possible binary **combinations** for **Present** state and **Input** together
- **Next state** values are determined from either the **logic diagram** or the **state equations**

- A sequential circuit with **m** flipflops and **n** inputs →
  A. **Normal** Form: 2^{m+n} rows
  B. **Compact** Form: 2^{m} rows
- The **next-state** section has **m** columns, **one** for each flip-flop
- The **output** section has as many columns as there are **output variables**

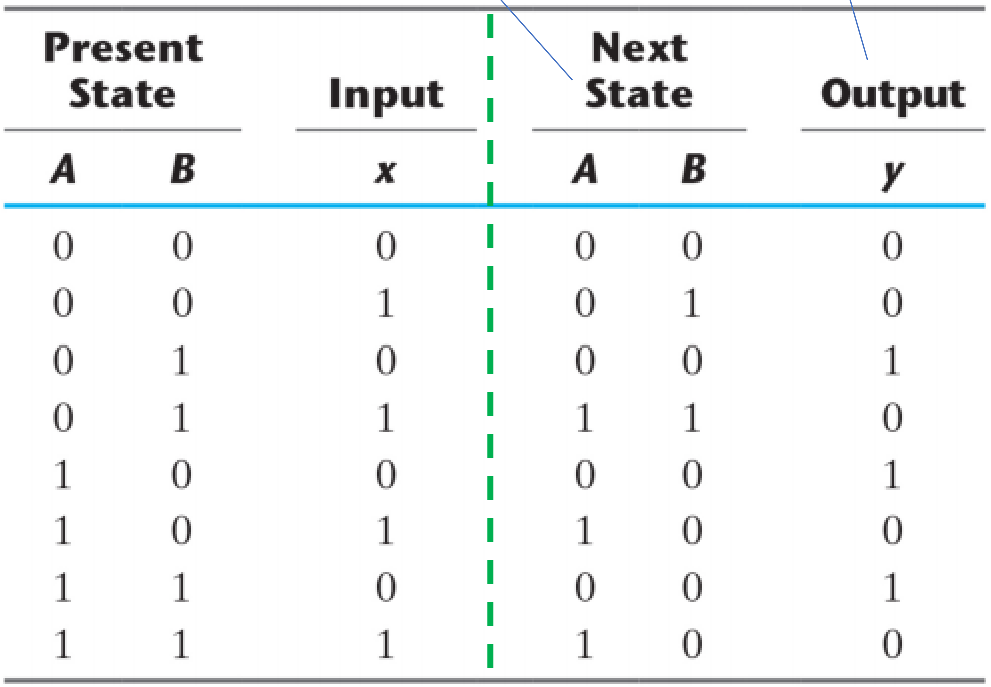
$$A(t+1) = Ax + Bx$$
$$B(t+1) = A'x$$

$$y(t) = (A+B)x'$$



Present State		Next State				Output	
		x = 0		x = 1		x = 0	x = 1
A	B	A	B	A	B	y	y
0	0	0	0	0	1	0	0
0	1	0	0	1	1	1	0
1	0	0	0	1	0	1	0
1	1	0	0	1	0	1	0

Compact Form (**2-D**) : State Table



Present State		Input	Next State		Output
A	B	x	A	B	y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

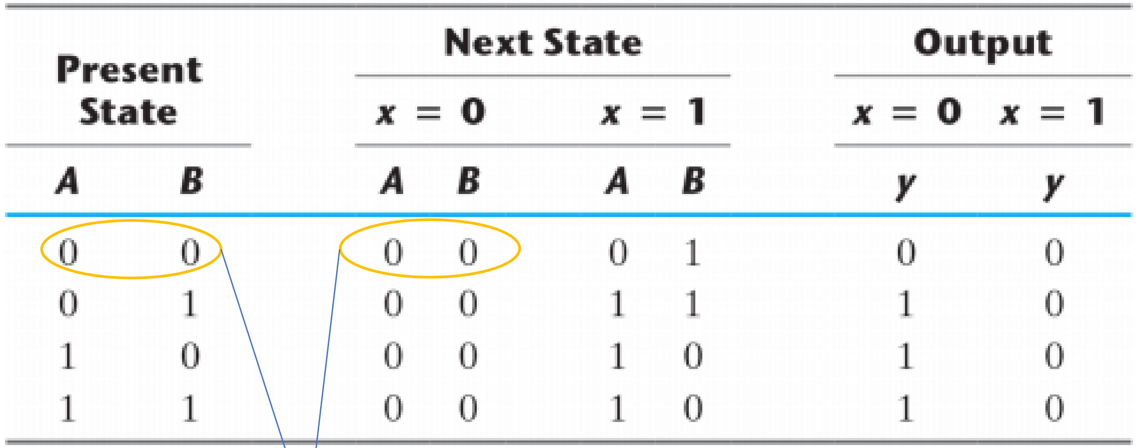
Normal Form (**1-D**): State Table

State Diagram

- **State diagram** is a **graphical** way of showing the **state table**
  - More convenient to understand the behavior of a circuit
- In this diagram, **states** are represented by **circles**
- The **transition** from **one** state to another is represented by a **line (Arrow)** between the **circles**
- Lines/Arrows are labeled as: (**Inputs** / **Outputs)**

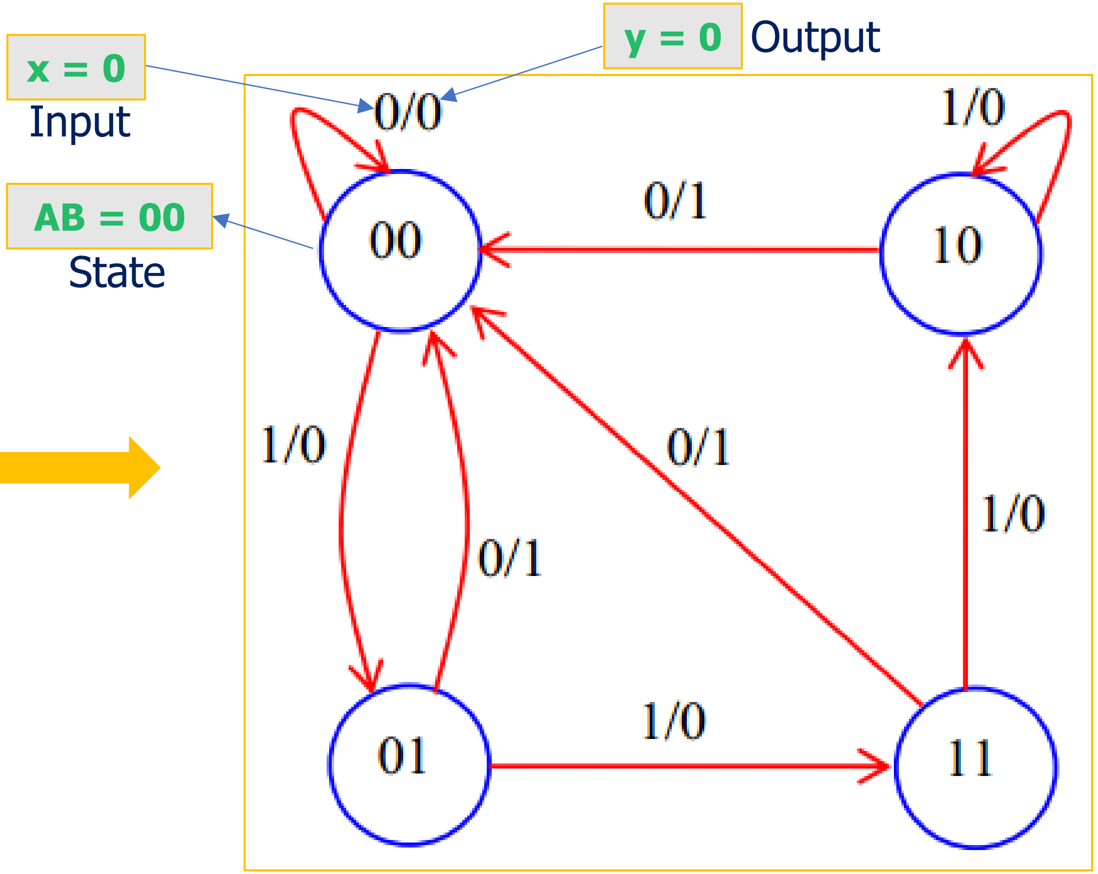
- **m** flipflops → m state **variables** → 2^m states **(circles)**

m=2 → 2^2 → **4** States: 00,01,10,11 (All **A,B** Combinations)



Present State		Next State				Output	
		$x = 0$		$x = 1$		$x = 0$	$x = 1$
A	B	A	B	A	B	y	y
0	0	0	0	0	1	0	0
0	1	0	0	1	1	1	0
1	0	0	0	1	0	1	0
1	1	0	0	1	0	1	0

AB = 00
State



State Diagram

General Flip-Flop Analysis Procedure

- Sequential circuits → combinational circuits + flip-flops

- We need **two** types of equations for drawing the logic diagram of the sequential circuit:
  1) Set of Boolean functions that describes algebraically the **combinational** circuit that generates **outputs**
     - **Output Equations**
  2) Set of Boolean functions that describes algebraically the circuit that generates the **inputs to flip-flop**
     - **Input Equations** or **Excitation Equations**

- Consider the previous example:

$$\left.\begin{aligned} D_A &= A\,x + B\,x \\ D_B &= A'\,x \end{aligned}\right\} \textit{input equations}$$

$$\left. y = x'\,(A+B) \right\} \textit{output equations}$$

Flip-Flops Analysis Steps (Mainly for JK & T Flip-Flops)

1) **Determine** the flip-flop **input equations** in terms of the **current state** and circuit **inputs**

2) **List** the **binary values** of each **input equation**

3) **Use** the flip-flop **characteristic table** to determine the **next state** values in the state table

Analysis with D-FF

- **One** D flip-flop (A) → **Two** States
- **NO** external output
- **Two** inputs **x, y**

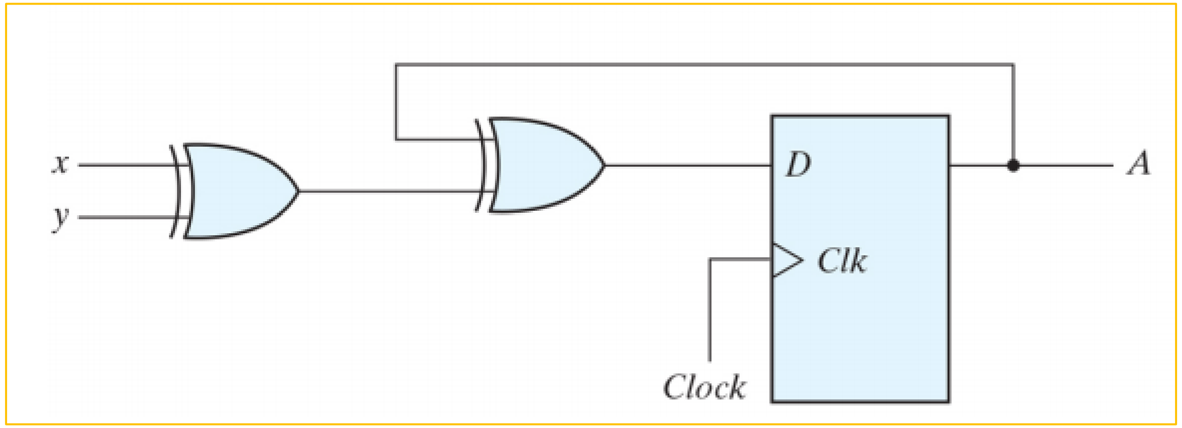
Input equation: $D_A = A \oplus x \oplus y$

D-FF → Next State = D

State equation: $A(t+1) = A \oplus x \oplus y$

2

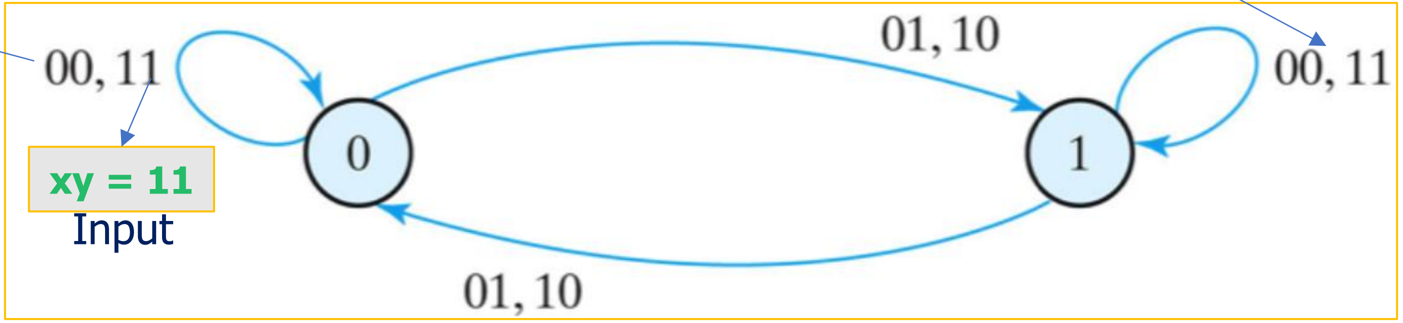
1 **Circuit Diagram**



PS Present state	Inputs		NS Next state
A	x	y	A
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Notice: Since NS = D,
No need for a column D

Notice: When **Multiple** Input combinations →
Same Transition : We use (**,**) to separate them



xy = 00
Input

xy = 11
Input

Notice: **No** External Output in this circuit →
No (**/**) exist over the arrow

State Diagram 4

State Table 3

Recap: Different Types of Equations

- **Characteristic Equations:**

$$\text{D Flip-Flop: } Q(t+1) = D$$
$$\text{JK Flip-Flop: } Q(t+1) = JQ' + K'Q$$
$$\text{T Flip-Flop: } Q(t+1) = TQ' + T'Q$$

- **State Equations:**

$$A(t+1) = Ax + Bx$$
$$B(t+1) = A'x$$

Examples

- **Input/Output Equations:**

Input

$$D_A = Ax + Bx$$
$$D_B = A'x$$

Output

$$y = (A+B)'$$

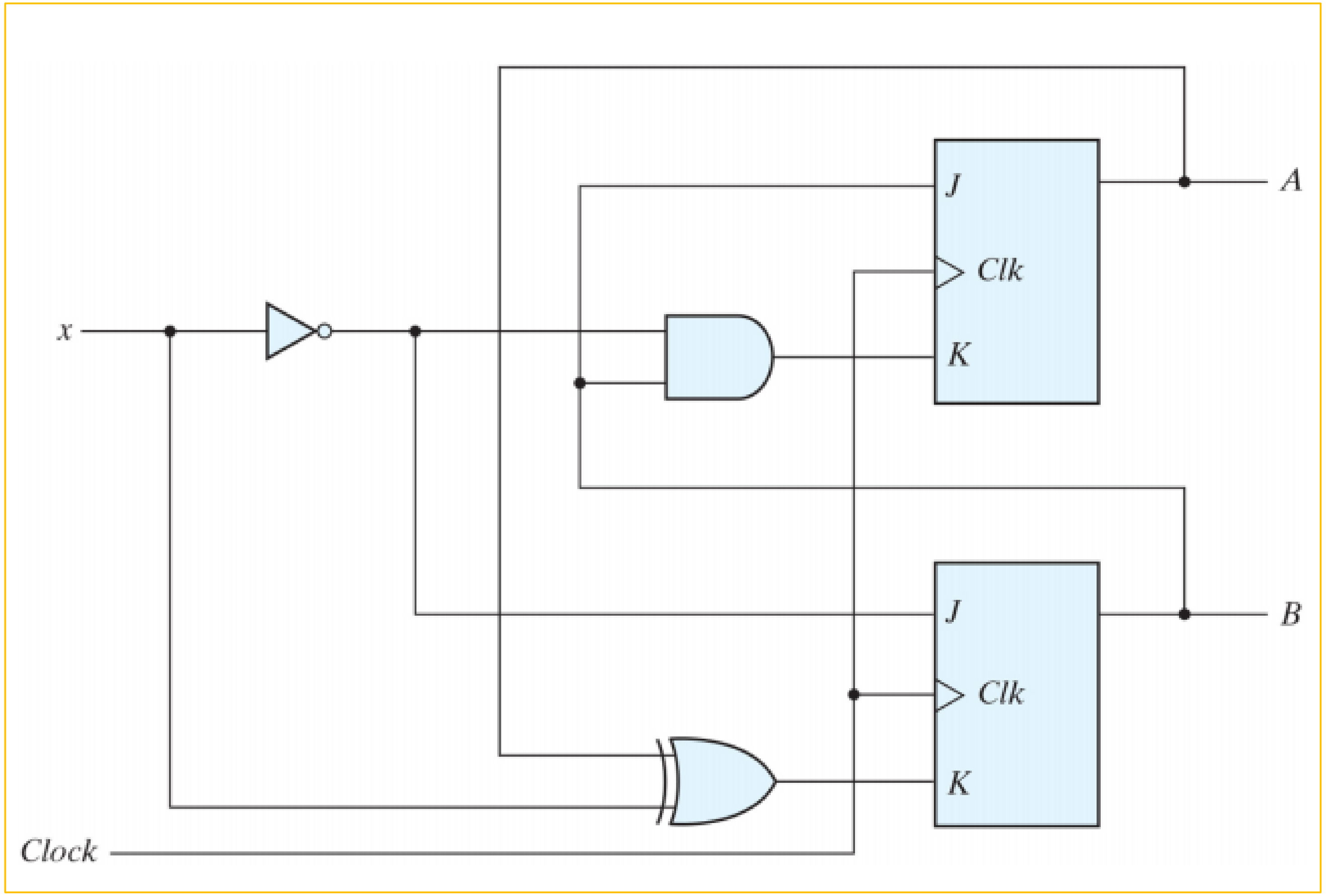
Examples

Analysis with JK-FF

- **Two** JK flip-flops (A,B) → **Four** States
- **NO** external output
- **One** input **x**

Input equation: $J_A = B$, $K_A = B\,x'$

$J_B = x'$, $K_B = A \oplus x$

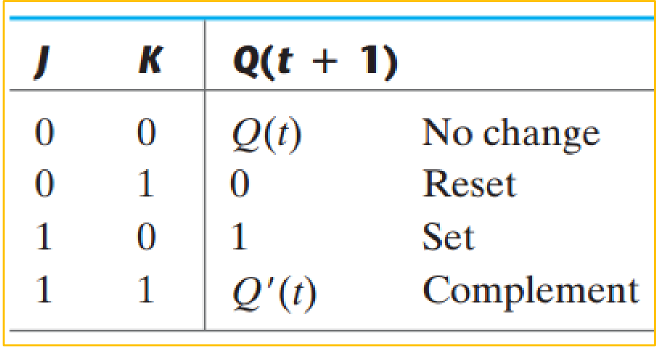


Circuit Diagram

Analysis with JK-FF: Using Characteristic Table

Input equation: $J_A = B$, $K_A = B\,x'$

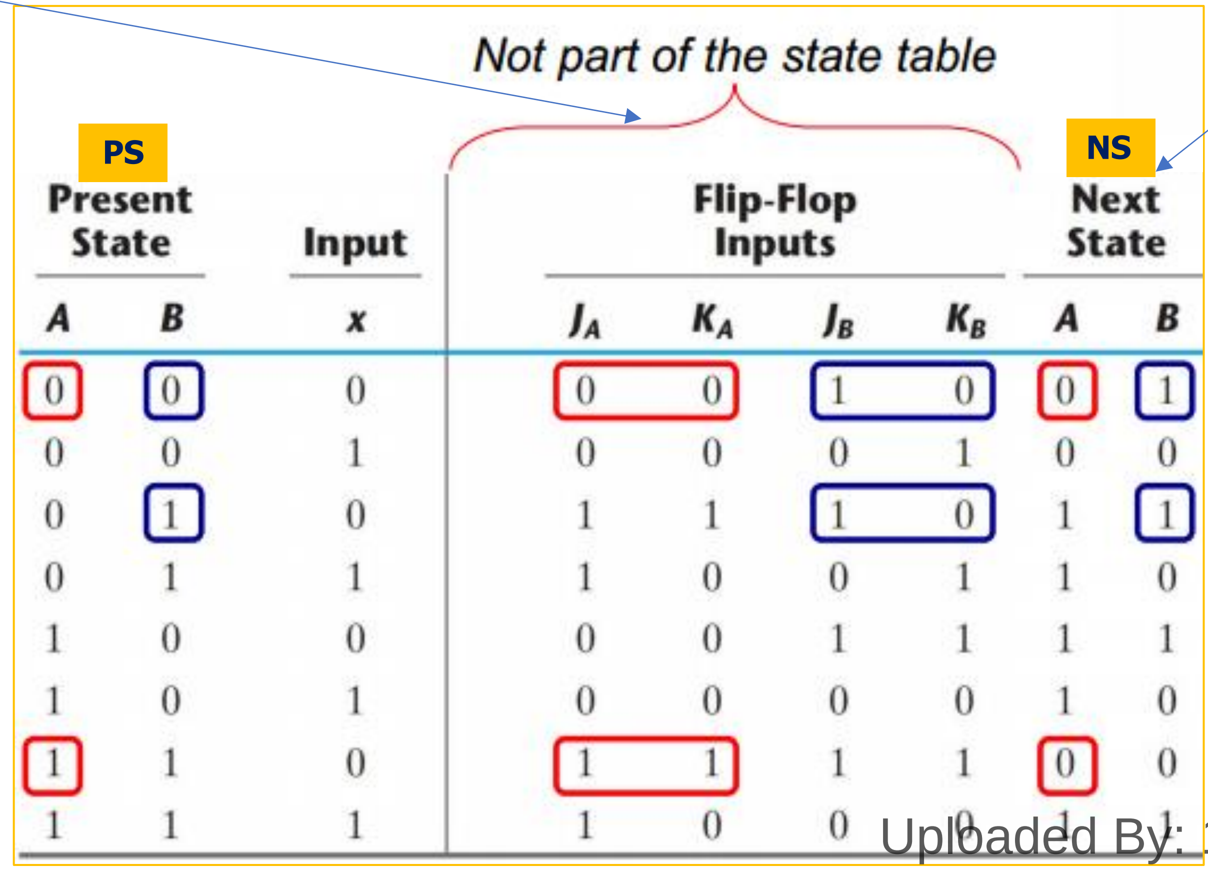
$J_B = x'$, $K_B = A \oplus x$



J	*K*	*Q*(*t* + 1)	
0	0	*Q*(*t*)	No change
0	1	0	Reset
1	0	1	Set
1	1	*Q*′(*t*)	Complement

Derive the **JK columns** by substituting **PS/Input** combinations into the **Input equations** 1

Use JK-FF **Characteristic** Table to fill the **next** state **columns** 2



Not part of the state table

PS

NS

Present State		Input	Flip-Flop Inputs				Next State	
A	*B*	*x*	J_A	K_A	J_B	K_B	*A*	*B*
0	0	0	0	0	1	0	0	1
0	0	1	0	0	0	1	0	0
0	1	0	1	1	1	0	1	1
0	1	1	1	0	0	1	1	0
1	0	0	0	0	1	1	1	1
1	0	1	0	0	0	0	1	0
1	1	0	1	1	1	1	0	0
1	1	1	1	0	0	0	1	1

Notice: **No** need to find the **state equations.** The **Characteristic** Table is directly used to find **NS** values in the state table

Analysis with JK-FF: Using Characteristic Equation

Alternative Method using **Characteristic** **Equations**

- The next-state values can also be obtained by **evaluating** the **state equations** from the **characteristic equation**. This is done as:
  1) **Determine** the FF **input** equations
  2) **Substitute** the **input** equations into the FF **characteristic** **equation** to get the **state** **equation**
  3) **Use** corresponding **state** equations to determine **next state** values in the state table

JK-Characteristic equation: $Q(t+1) = JQ' + K'Q$

Characteristic equation:

$$A(t+1) = J_A A' + K'_A A$$

$$B(t+1) = J_B B' + K'_B B$$

Input equation: $J_A = B$, $K_A = B x'$

$J_B = x'$, $K_B = A \oplus x$

Substitute

$$A(t+1) = (B)A' + (Bx')'A$$
$$= A'B + AB' + Ax$$

$$B(t+1) = (x')B' + (A \oplus x)'B$$
$$= B'x' + ABx + A'Bx'$$

State equations

Analysis with JK-FF: Using Characteristic Equation

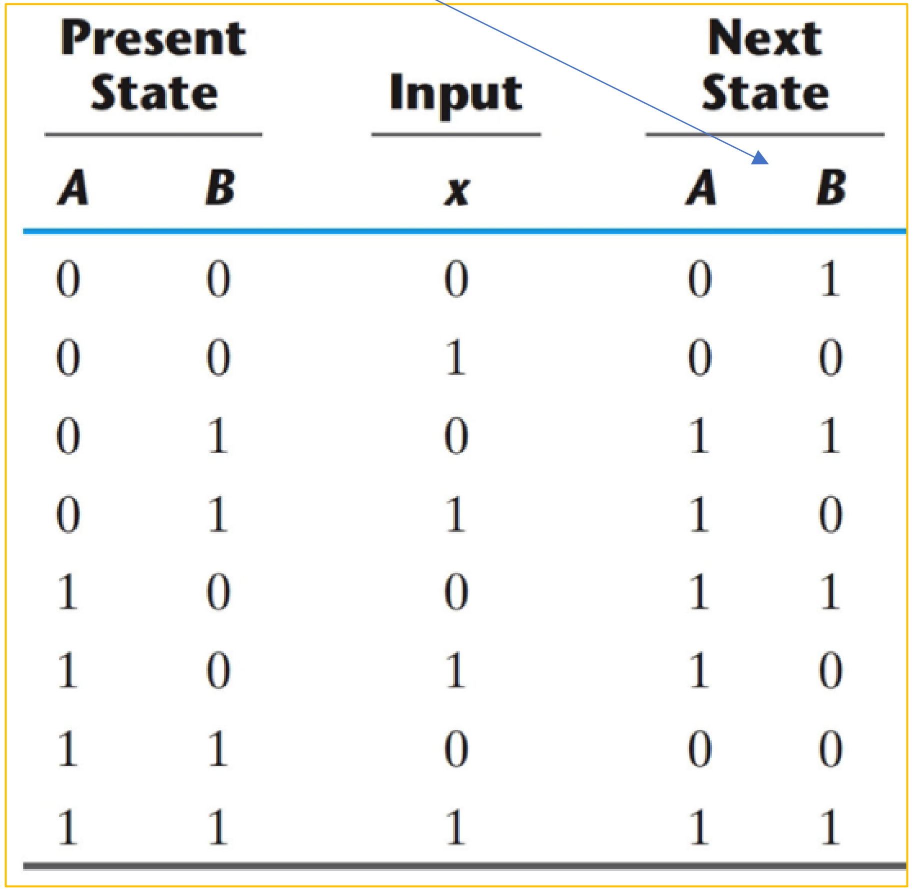
Construct the **State table** directly from the **State equations**

Notice: It is common practice to label the states with **alphanumeric labels** such as S0,S1.. Where the numeric value is the **decimal equivalence** of the state

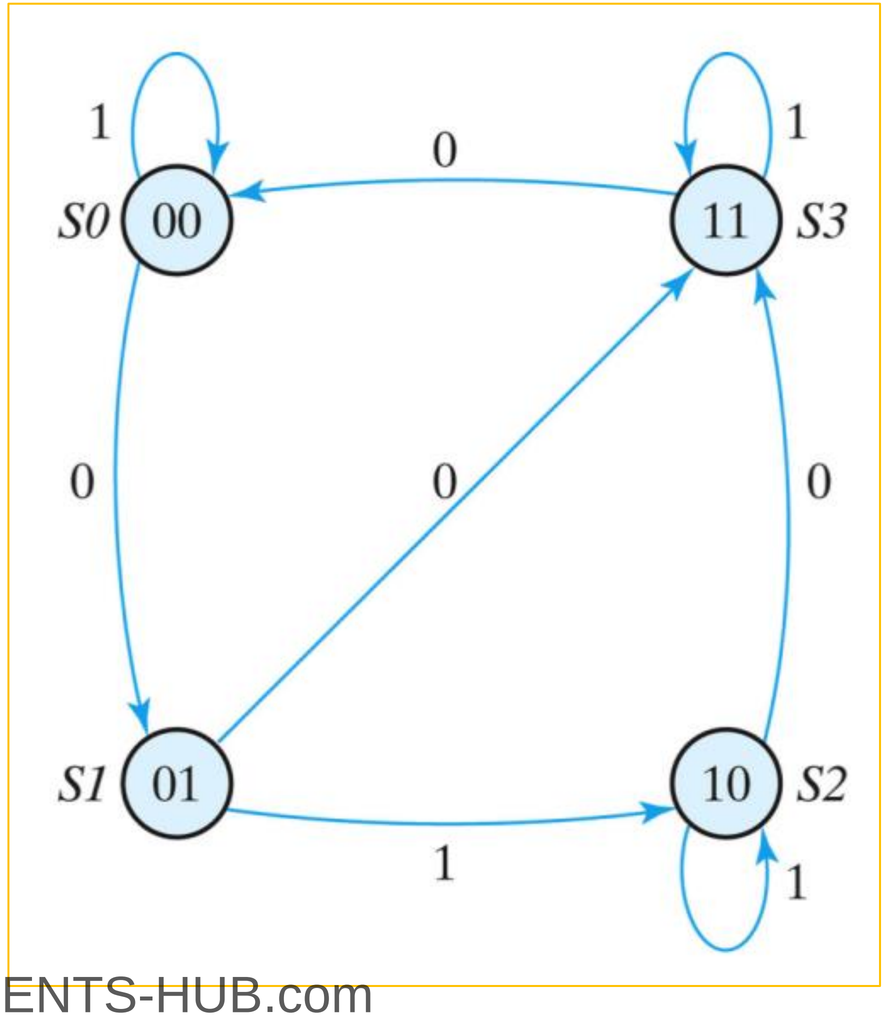
State equations

$$A(t+1) = A'B + AB' + Ax$$

$$B(t+1) = B'x' + ABx + A'Bx'$$



Present State		Input	Next State	
A	B	x	A	B
0	0	0	0	1
0	0	1	0	0
0	1	0	1	1
0	1	1	1	0
1	0	0	1	1
1	0	1	1	0
1	1	0	0	0
1	1	1	1	1



Analysis with T-FF

- **Two** T flip-flops (A,B) → **Four** States
- **One** external output
- **One** input **x**

Output equation: $y = AB$

Input equation: $T_A = Bx$
$T_B = x$

Substitute

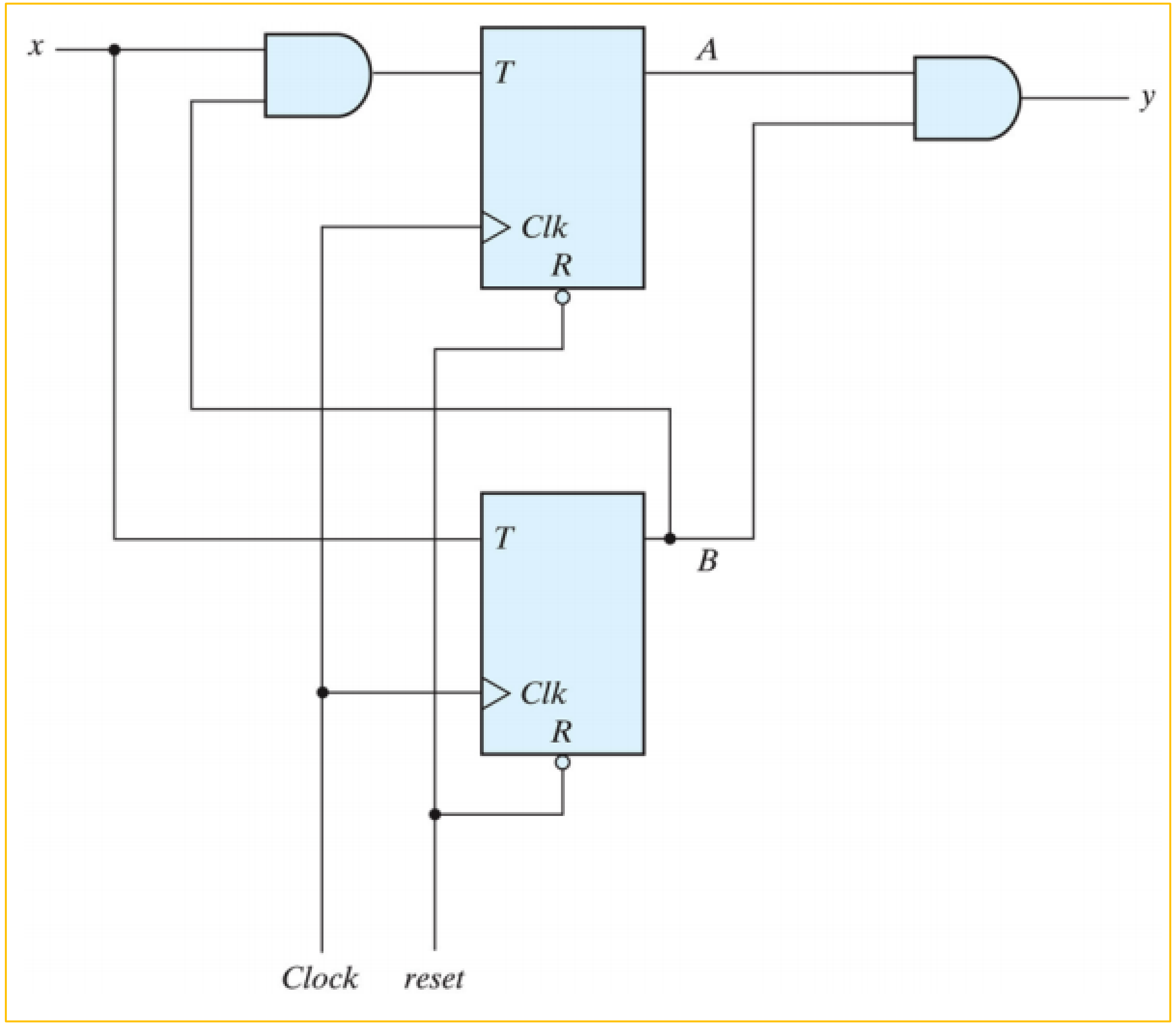
$$Q(t+1) = T \oplus Q = TQ' + T'Q$$

T-FF - Characteristic equation

$A(t+1) = (xB)'A + xBA' = x'A + AB' + xA'B$

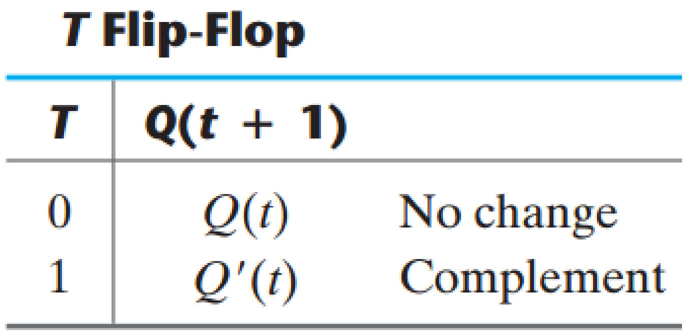
$B(t+1) = x'B + xB'$

State equations



Circuit Diagram

Analysis with T-FF: Using Characteristic Table



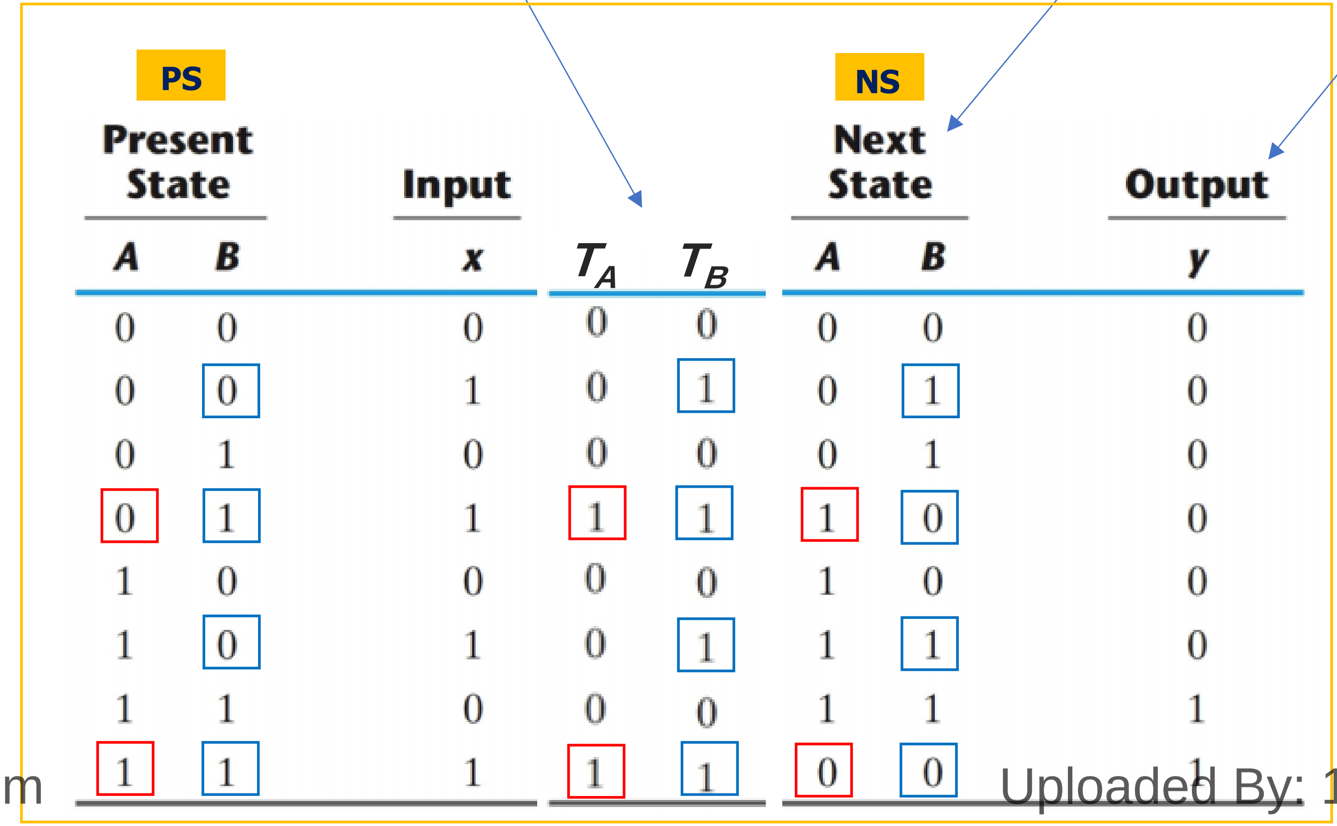
T Flip-Flop

T	$Q(t+1)$	
0	$Q(t)$	No change
1	$Q'(t)$	Complement

Input equation: $T_A = Bx$
$T_B = x$

Derive the **T columns** by substituting **PS/Input** combinations into the **Input equations** **1**

Use T-FF **Characteristic** Table to fill the **next** state **columns** **2**



PS Present State		Input			NS Next State		Output
A	B	x	T_A	T_B	A	B	y
0	0	0	0	0	0	0	0
0	$\boxed{0}$	1	0	$\boxed{1}$	0	$\boxed{1}$	0
0	1	0	0	0	0	1	0
$\boxed{0}$	$\boxed{1}$	1	$\boxed{1}$	$\boxed{1}$	$\boxed{1}$	$\boxed{0}$	0
1	0	0	0	0	1	0	0
1	$\boxed{0}$	1	0	$\boxed{1}$	1	$\boxed{1}$	0
1	1	0	0	0	1	1	1
$\boxed{1}$	$\boxed{1}$	1	$\boxed{1}$	$\boxed{1}$	$\boxed{0}$	$\boxed{0}$	1

Output equation: $y = AB$

Output Column always constructed from the **Output Equation**

Notice: **No** need to find the **state equations**. The **Characteristic** Table is directly used to find **NS** values in the state table

Analysis with T-FF: Using State Equations

Construct the **State table** directly from the **State equations**

Output Column always constructed from the **Output Equation**

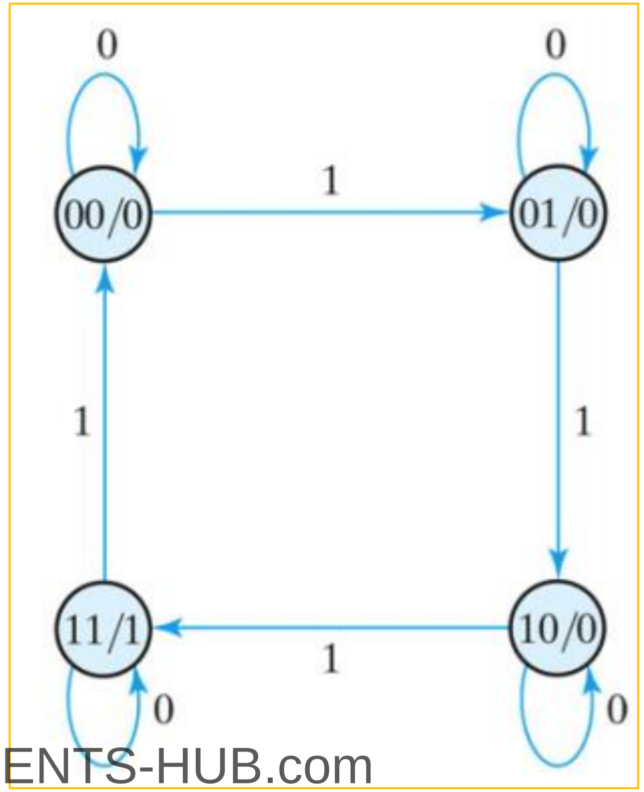
State equations

$$A(t+1) = (xB)'A + xBA' = \boxed{x'A + AB' + xA'B}$$

$$B(t+1) = \boxed{x'B + xB'}$$

$$y = AB$$

Notice: We put the **output inside** the state **circle** because it does **Not depend** on the Input

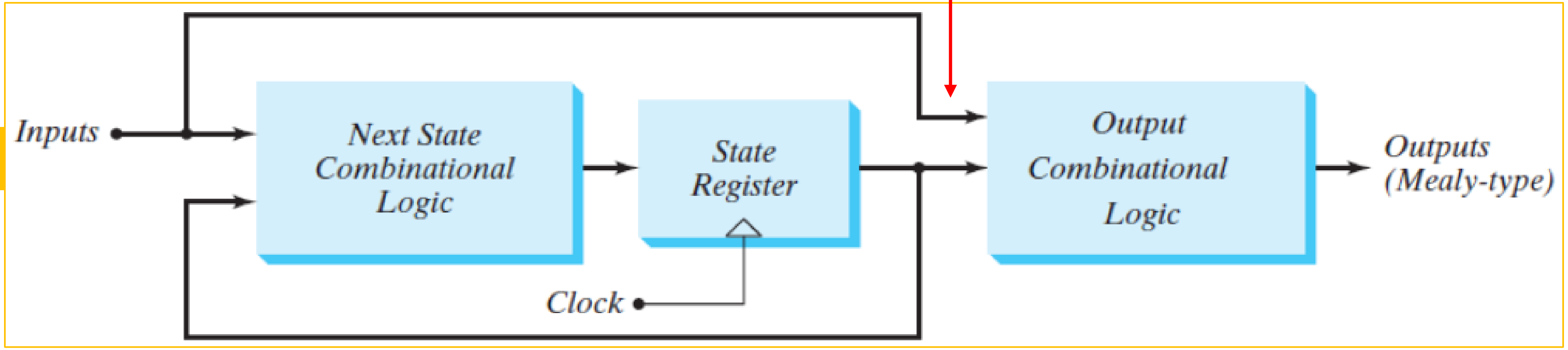


Present State		Input	Next State		Output
A	**B**	**x**	**A**	**B**	**y**
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	1

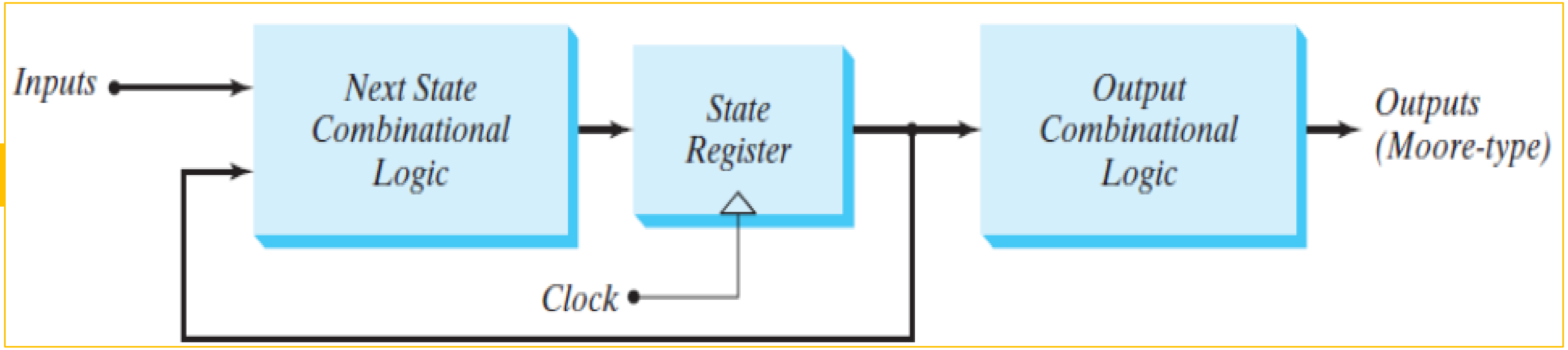
Finite State Machine Models (FSMs)

- There are **two** ways to design a clocked sequential circuit (differ only in the way **output** is **generated**)
  1) **Mealy FSM**: **Outputs** depend on **present state and inputs**
  2) **Moore FSM**: Outputs depend on **present state only**
- A circuit may have **both** types of **outputs**

Mealy FSM

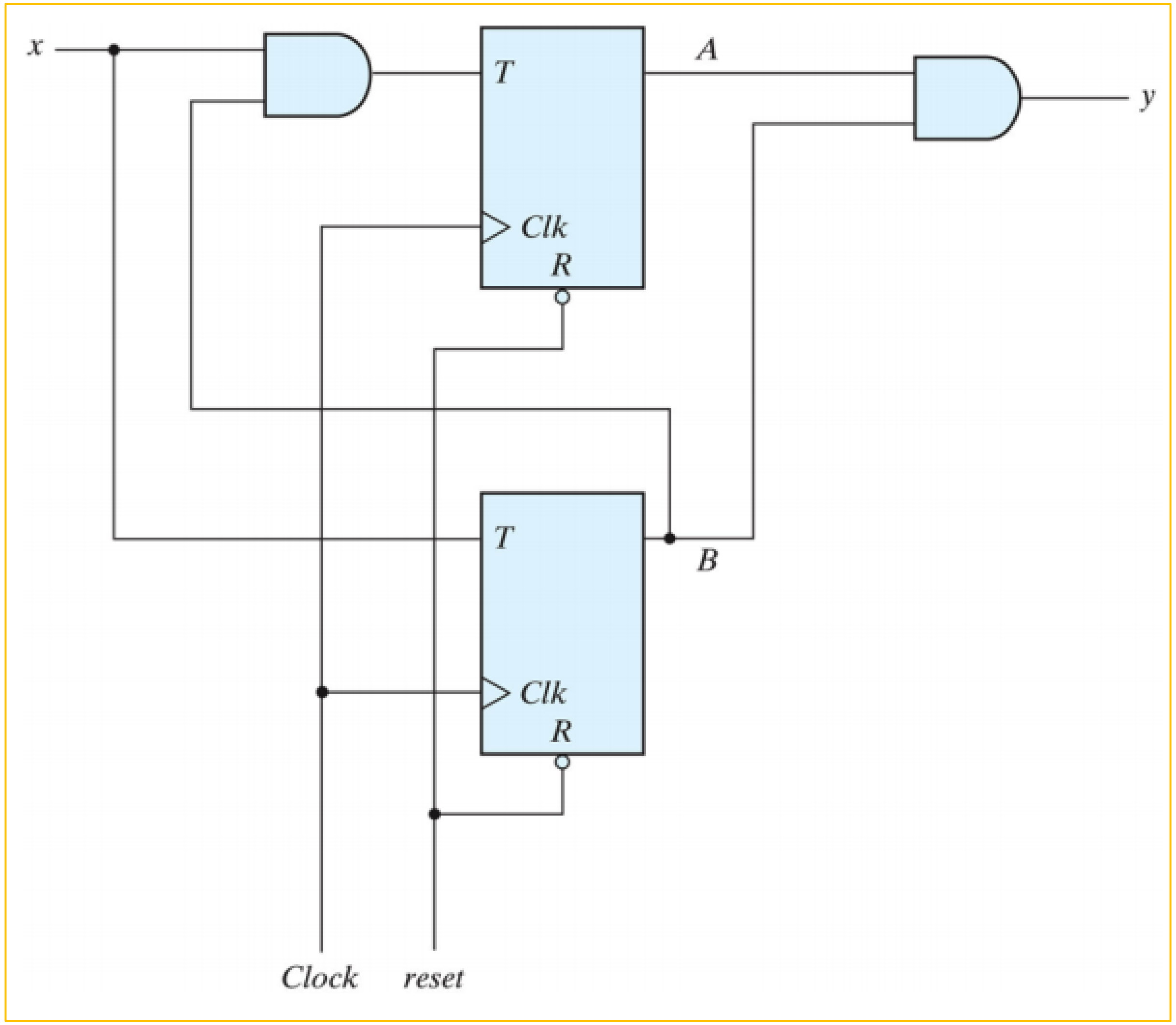


Moore FSM

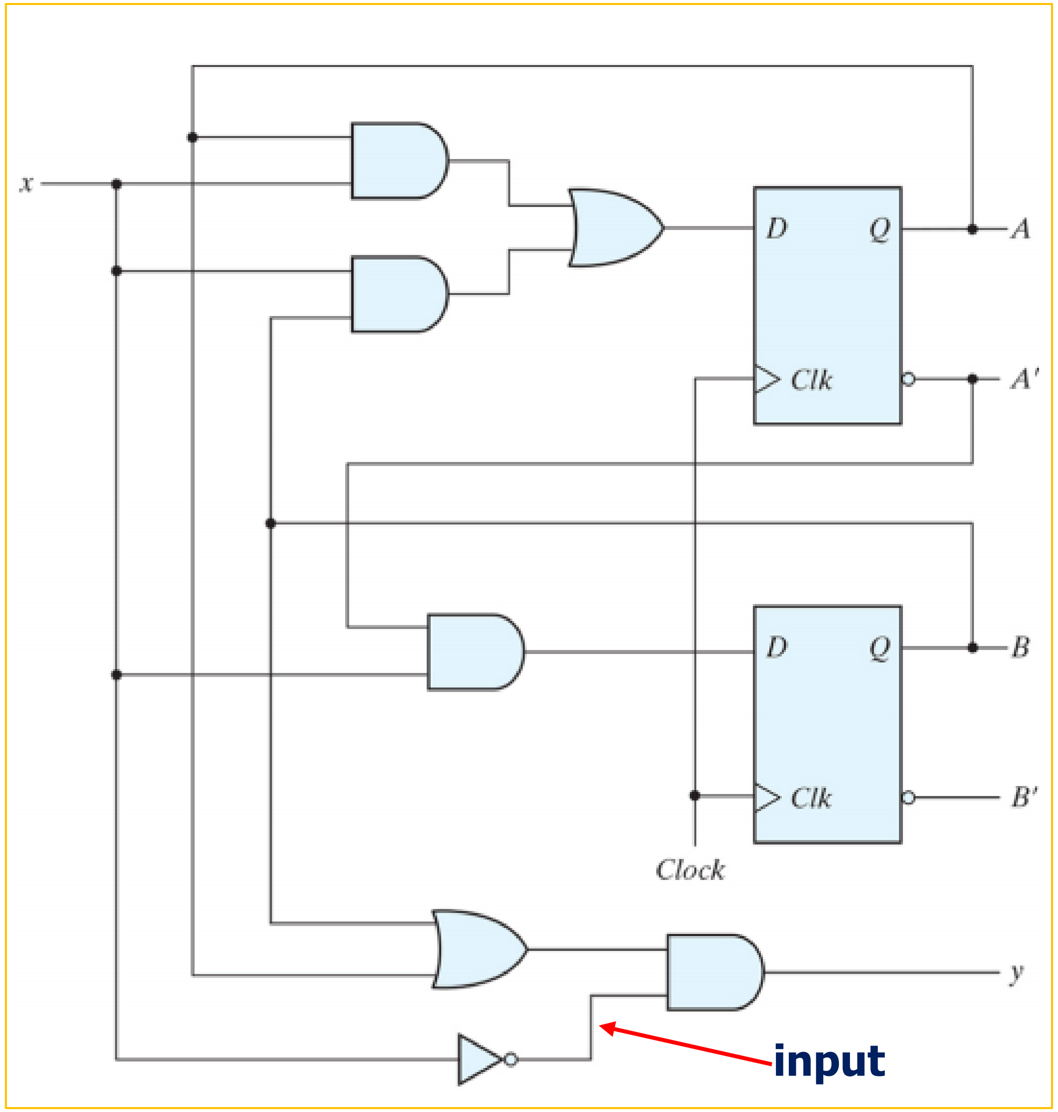


Finite State Machine Models (FSMs)

Moore FSM



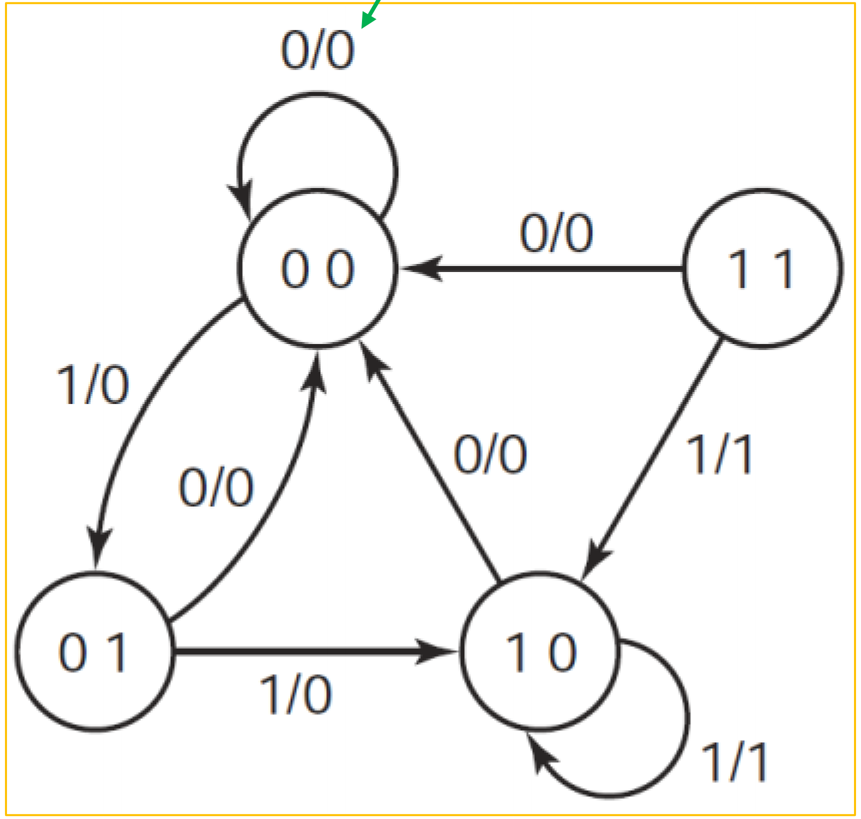
Mealy FSM



Mealy Machine (Mealy FSM)

- The **outputs** are a function of the **present state and Inputs**
- The **outputs** are **NOT synchronized** with the clock
- The **outputs** may **change** if inputs change **during** the clock cycle
- The **outputs** may have momentary **false values** (called **glitches**)
- The **correct outputs** are present **just before** the **edge** of the clock

Notice: We indicate the **output over the line/arrow** because it **depend** on the **Input**

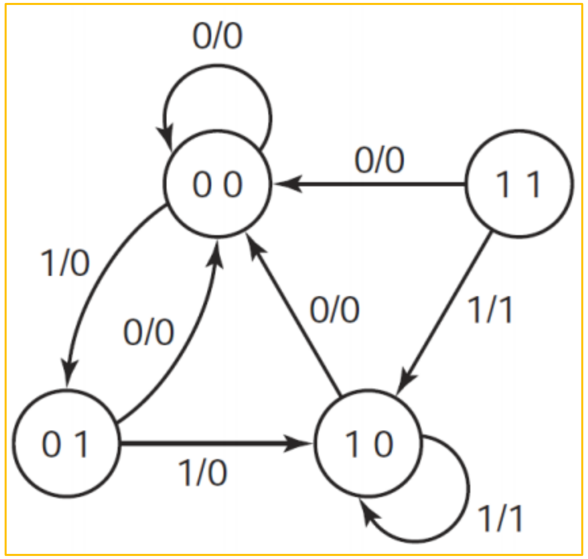


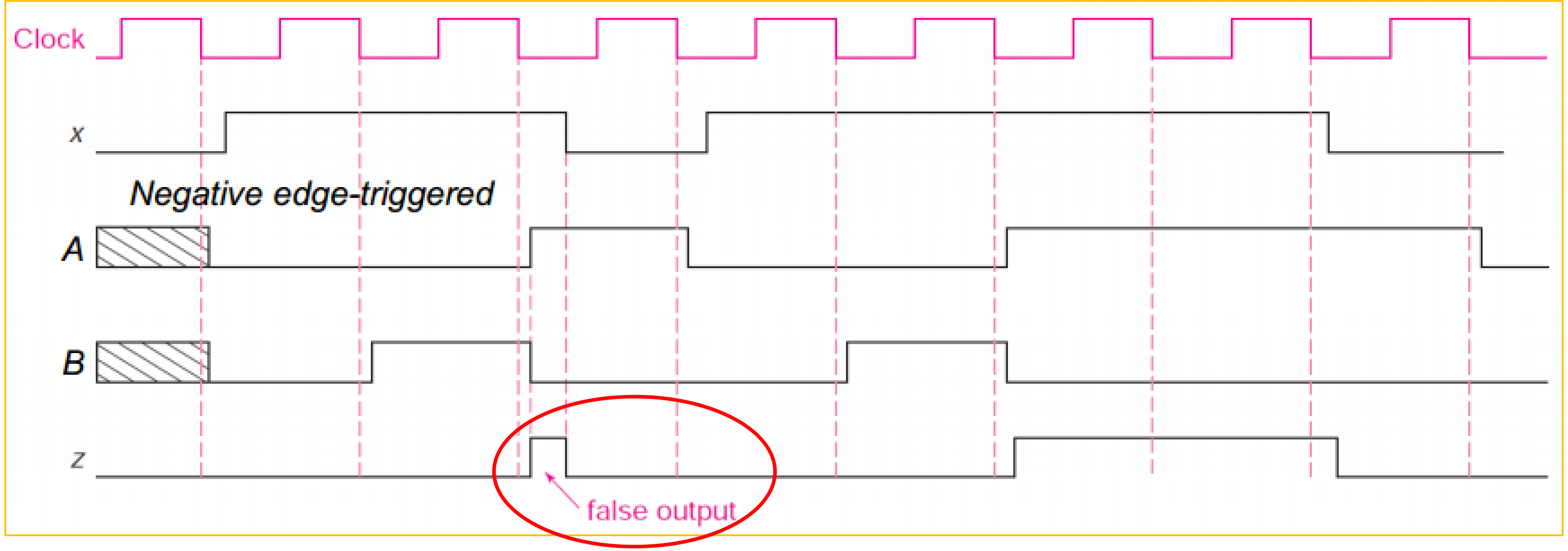
Mealy FSM

- Each line/arrow is **labeled** with: **Input / Output**
- The **output** is shown on the line/arrow of the state diagram

Tracing a Mealy FSM: Timing Diagram

Cycle	0	1	2	3	4	5	6	7	8
Input *x*	0	1	1	0	1	1	1	1	0
Present State *A B*	?	0	0	1	0	0	1	1	1
	?	0	1	0	0	1	0	0	0
Output *z*	0	0	0	0	0	0	1	1	0



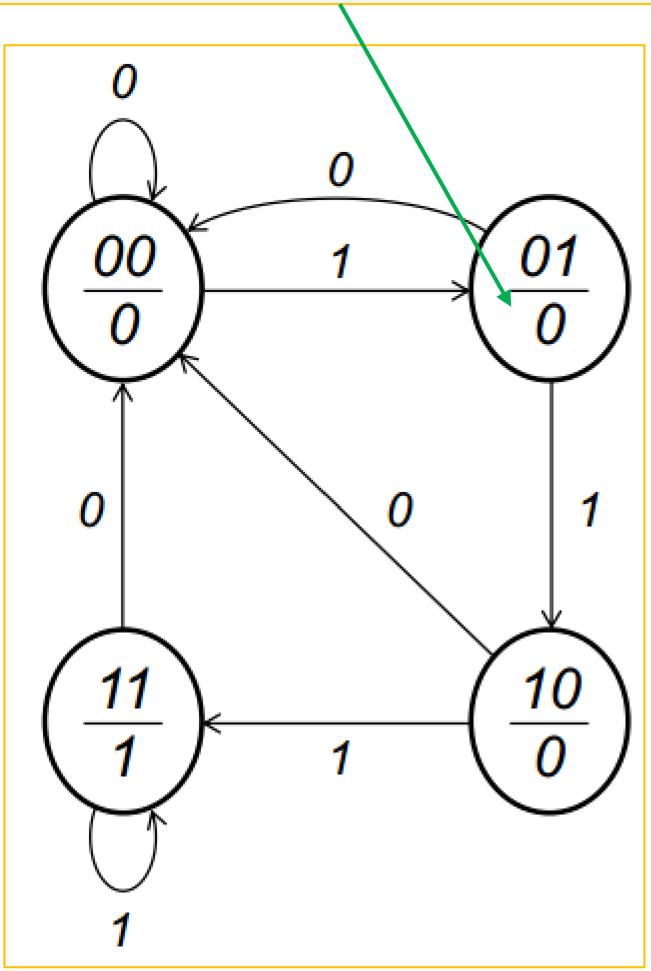


- When the circuit is **powered**, the initial state (AB) is **unknown**
- Even though the initial state is unknown, the **input x = 0 forces a transition to state AB = 00**, regardless of the present state
- Sometimes, a **reset input** is used to initialize the state to 00

Moore Machine (Mealy FSM)

- The **outputs** are a function of the **Flip-Flop** outputs **only**
- The **outputs** depend on the **current state only**
- The **outputs** are **synchronized** with the clock
- **Glitches cannot** appear in the outputs (even if inputs change)

Notice: We indicate the **output inside the state circle** because it **depend** on the **state only**



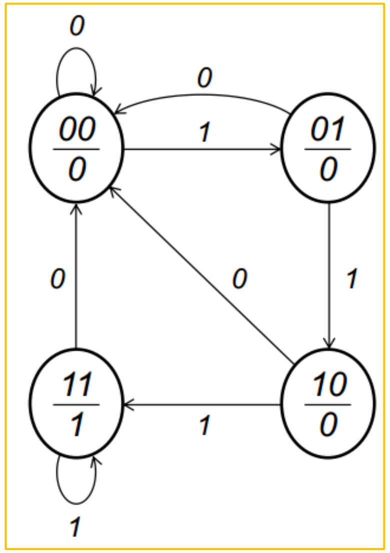
Moore FSM

- Each line/arrow is **labeled** with **Input** only
- The **output** is shown **inside** the state: (**State** / **Output**)
  - ✓ The output depends on the current state only

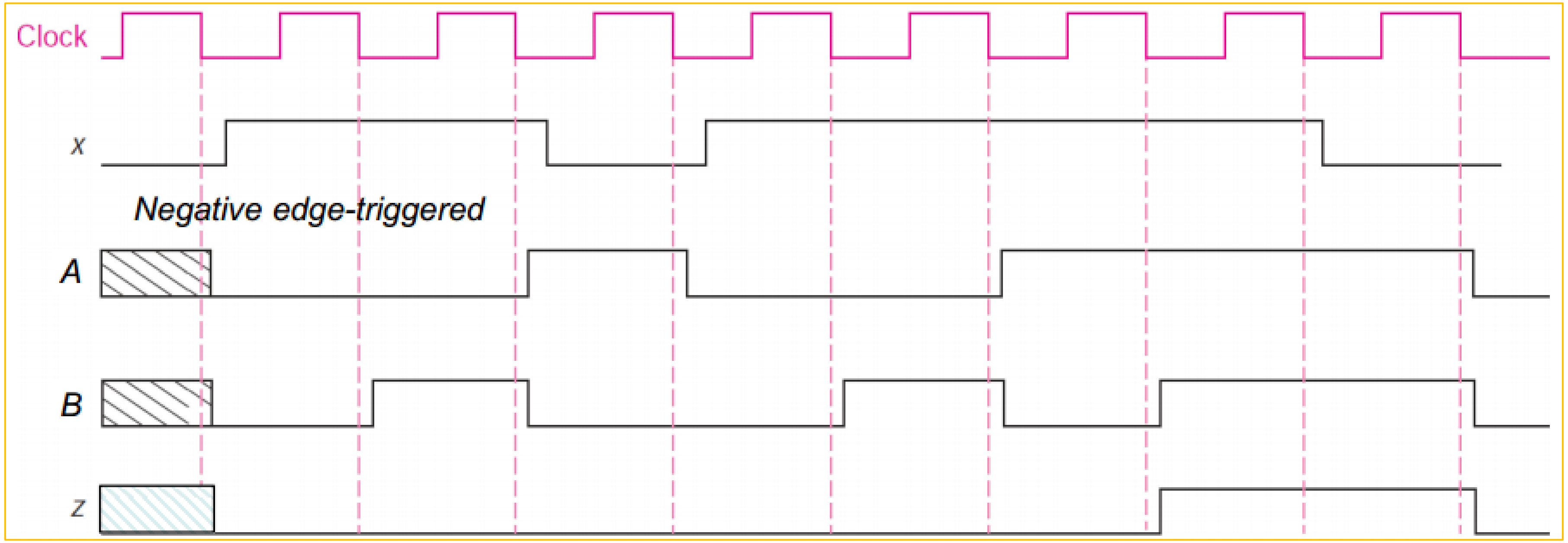
A given design might **mix** between Mealy and Moore

Tracing a Mealy FSM: Timing Diagram

Cycle	0	1	2	3	4	5	6	7	8
Input *x*	0	1	1	0	1	1	1	1	0
Present State *A B*	?	0	0	1	0	0	1	1	1
	?	0	1	0	0	1	0	1	1
Output *z*	?	0	0	0	0	0	0	1	1



- When the circuit is **powered**, the initial state (AB) and output are **unknown**
- **Input x = 0 resets the state AB to 00**. Can also be done with a reset signal.



The output is **synchronized** with the clock. **No false** output (or **glitch**) may appear.

State Reduction

- **Recall:** The **number of states** in the system determines the **number of FFs** needed
  - 2^m **states** → **m FF**
- Two sequential circuits may have **same input-output** behavior but **different number of internal** states
- Certain properties of sequential circuits allow us to **reduce their number of states**
- **State-Reduction** is referred to the reduction of the **number of states** in a sequential circuit to **reduce the number of FFs** (Less FFs → Cost reduction)
  - **Reducing** the **number of states** while **preserving** the input/output relationship

State **Reduction** ≡ State Minimization ≡ Eliminate **Redundant** States

Notice: Reduction of states does **NOT** always result in reduction of flip-flops

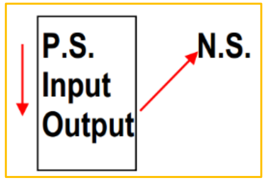
Sometimes, reduction in flip-flops result in a **bigger combinational circuit** to realize the next state and the outputs

State Reduction

Example: Consider the input sequence: **01010110100** starting from the **initial** state a

State	a	a	b	c	d	e	f	f	g	f	g	a
Input	**0**	**1**	**0**	**1**	**0**	**1**	**1**	**0**	**1**	**0**	**0**	
Output	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**0**	**1**	**0**	**0**	

Pattern

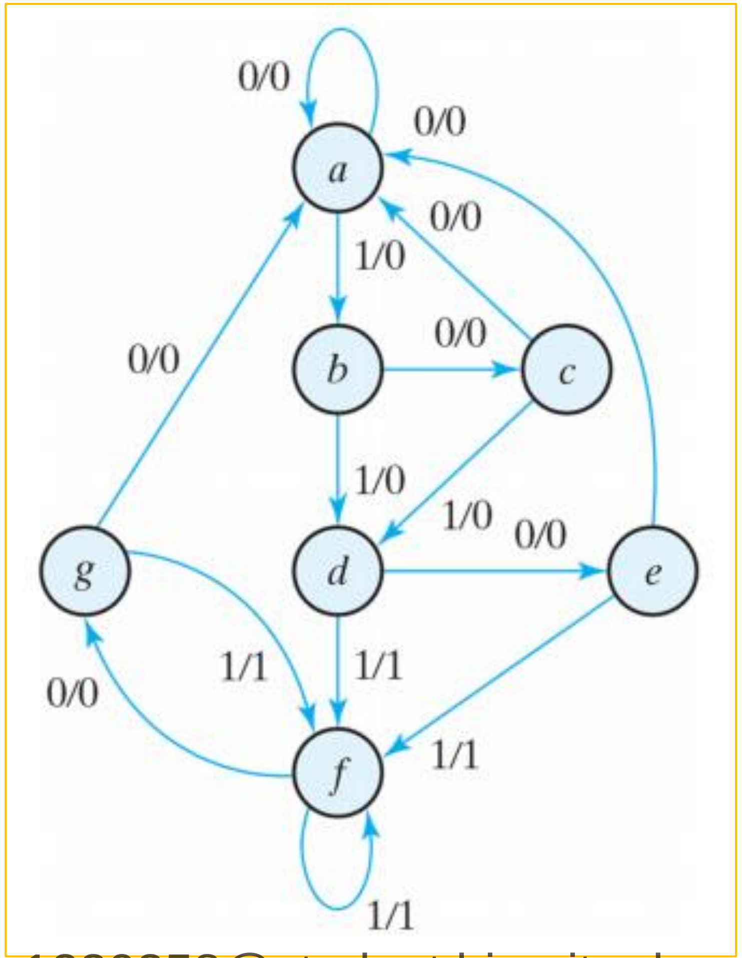


P.S.
Input
Output
N.S.

Only interested in **input-output relationship**, states are of secondary importance

There are an **infinite** number of input sequences that may be applied to the circuit; each results in a **unique** output sequence

Only **input-output sequences are important**, the internal states are used to provide required sequences

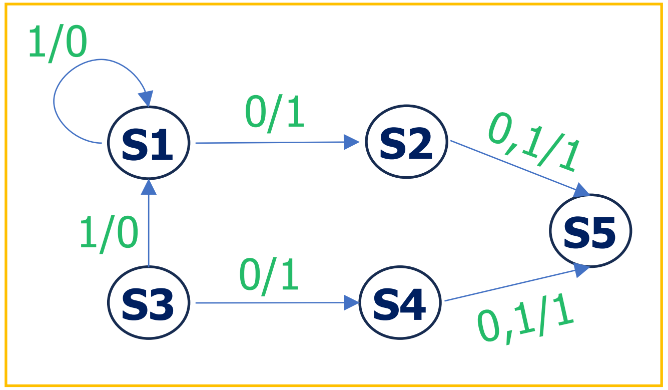


State Reduction

- Two circuits are said to be **identical** if the **same** applied **input sequence** gives the **same output sequence**
- The challenge with state-reduction is to find ways to reduce the number of states **without affecting** the input-output relationships

Two states are said to be **equivalent**, if, for each set of inputs, they **give exactly** the **same output** and **send** the circuit to the **same state** or to an **equivalent state**

When two states are **equivalent**, one of them can be **removed** without **altering** the input–output relationships



S2 & S4 are **equivalent**

⇩

S1 & S3 are **equivalent**

It is easier to work with **state tables**

State Reduction: Implication Table Method

- **Systematically** determines which **states** can be considered **equivalent**
- **Steps** (Let **n** = Number of States in the original Diagram/Table)

1) Start with the **state** table

2) Construct the Implication table
   a) **Stair Case** Structure
   b) Shall contains a **square** for each **pair** of states
   c) Rows (2 to n) | Columns (1 to n-1)

3) For every square, compare each pair of rows in the state table:
   a) If any of the **outputs** for the rows being compared **differ**, place an **X** in the square.
   b) If the **outputs** are the **same**, list the **implied pairs** in the square.
      - Any implied pair that is **identical** or the states **themselves** is **omitted**.
   c) If the **outputs** are the **same** and if both the **implied pairs** are **identical** and/or the states **themselves**, then place a ✓ in the square.

4) For **all** squares in the table with **implied pairs**, examine the square of each implied pair. If **any** of the implied pair squares has an X, then put an **X** in this square

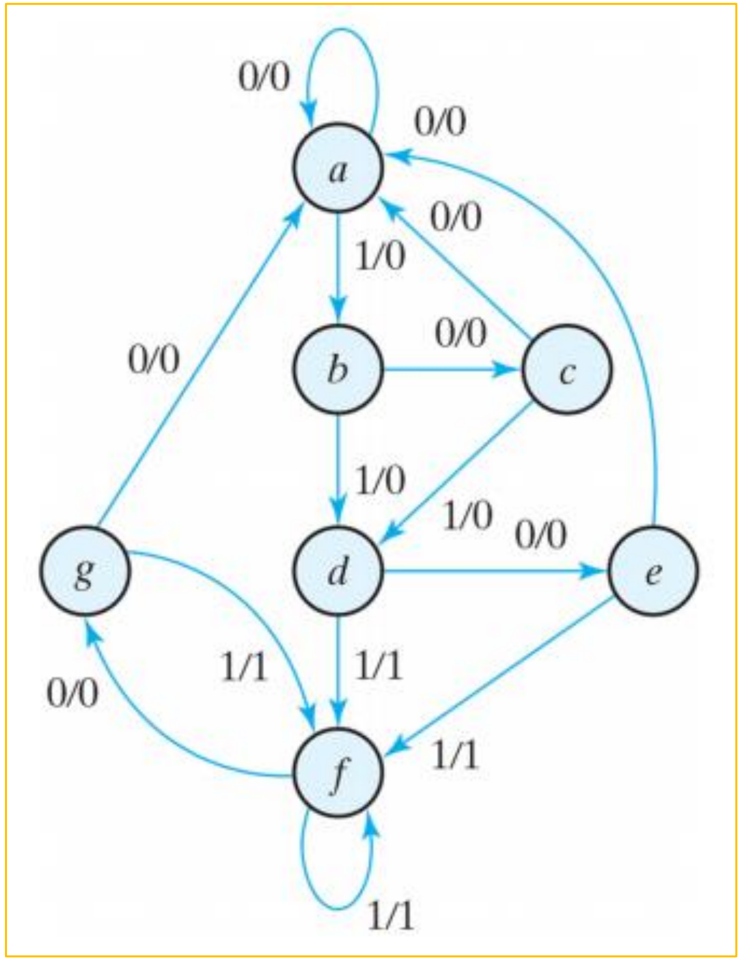
5) Repeat the step 4 until **NO** more **Xs** are added.

6) Upon completion of the previous step, squares **without X's** indicate **equivalent states**.

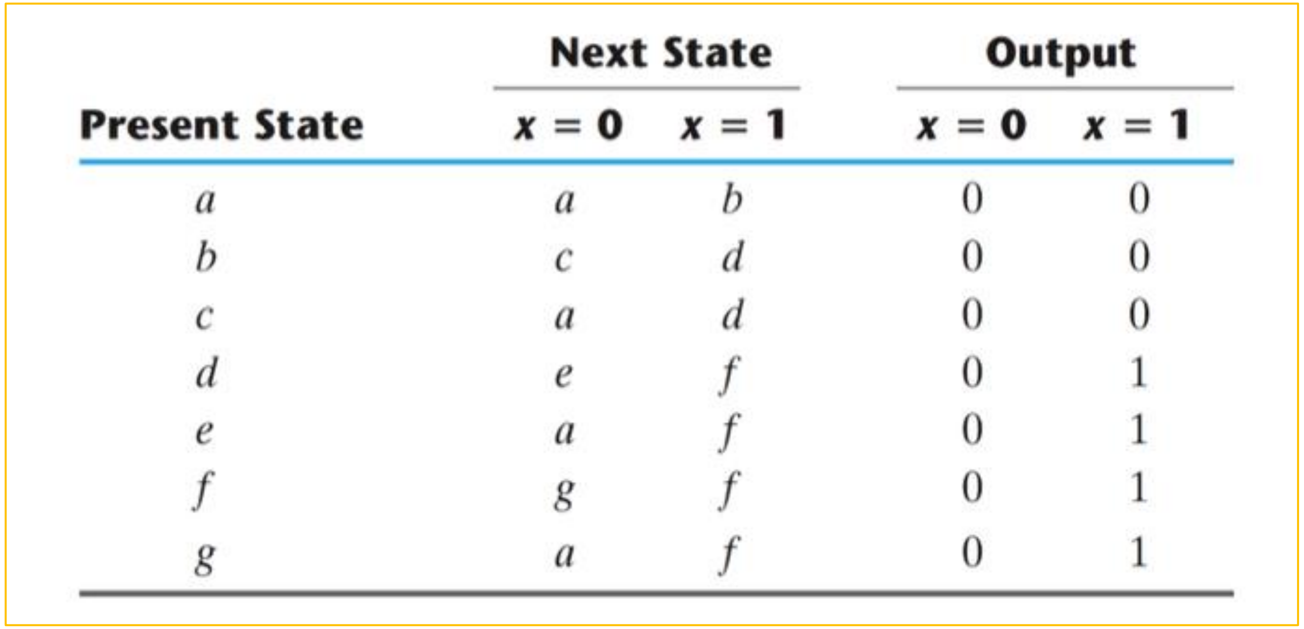
7) **Remove** redundant states, and **replace** them with equivalent states.

State Reduction: Implication Table Method

Example:



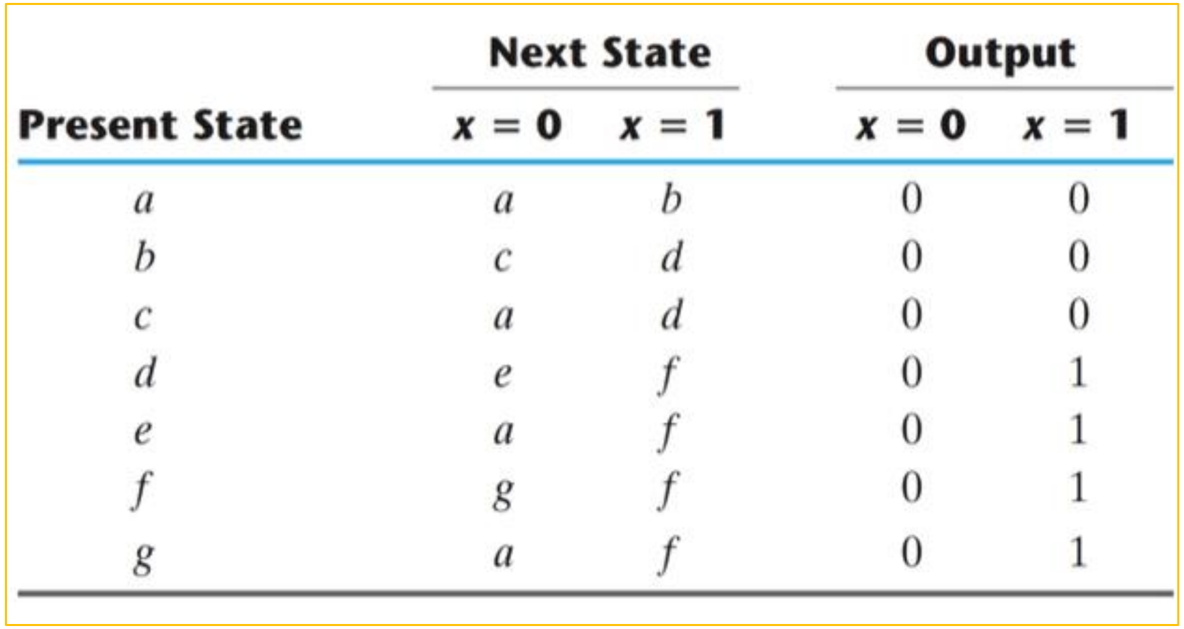
Step 1



Present State	Next State		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	*a*	*b*	0	0
b	*c*	*d*	0	0
c	*a*	*d*	0	0
d	*e*	*f*	0	1
e	*a*	*f*	0	1
f	*g*	*f*	0	1
g	*a*	*f*	0	1

State Reduction: Implication Table Method

Example:



	Next State		Output	
Present State	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1
e	a	f	0	1
f	g	f	0	1
g	a	f	0	1

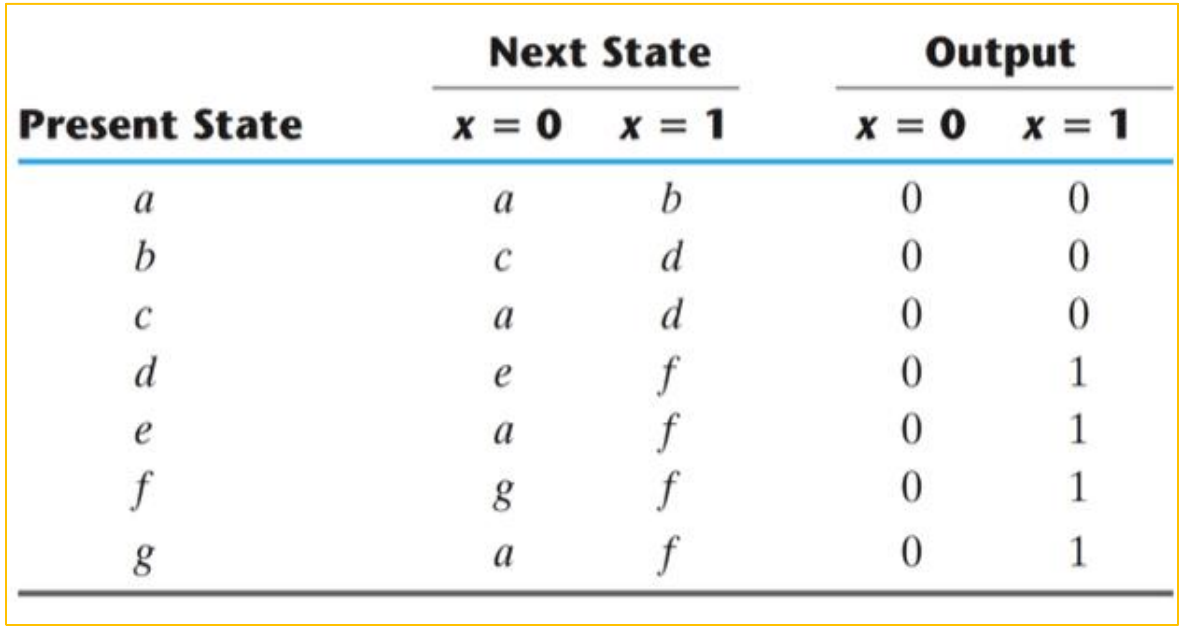
Step 2

b						
c						
d						
e						
f						
g						
	a	**b**	**c**	**d**	**e**	**f**

n = 7 → 6x6 Table

State Reduction: Implication Table Method

Example:



Present State	Next State $x = 0$	Next State $x = 1$	Output $x = 0$	Output $x = 1$
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1
e	a	f	0	1
f	g	f	0	1
g	a	f	0	1

Step 3

b	**(a=c) (b=d)**					
c	**(b=d)**	**(a=c)**				
d	X	X	X			
e	X	X	X	(a=e)		
f	X	X	X	(e=g)	(a=g)	
g	X	X	X	(a=e)	✓	(a=g)
	a	**b**	**c**	**d**	**e**	**f**

n = 7 → 6x6 Table

State Reduction: Implication Table Method

Example:

b	**(a,c)** **(b,d)**					
c	**(b,d)**	**(a,c)**				
d	X	X	X			
e	X	X	X	(a,e)		
f	X	X	X	(e,g)	(a,g)	
g	X	X	X	(a,e)	✓	(a,g)
	a	**b**	**c**	**d**	**e**	**f**

Steps 4-5

b	**(a=c)** **(b=d) X**					
c	**(b=d)** X	**(a=c)** X				
d	X	X	X			
e	X	X	X	X		
f	X	X	X	✓	X	
g	X	X	X	X	✓	X
	a	**b**	**c**	**d**	**e**	**f**

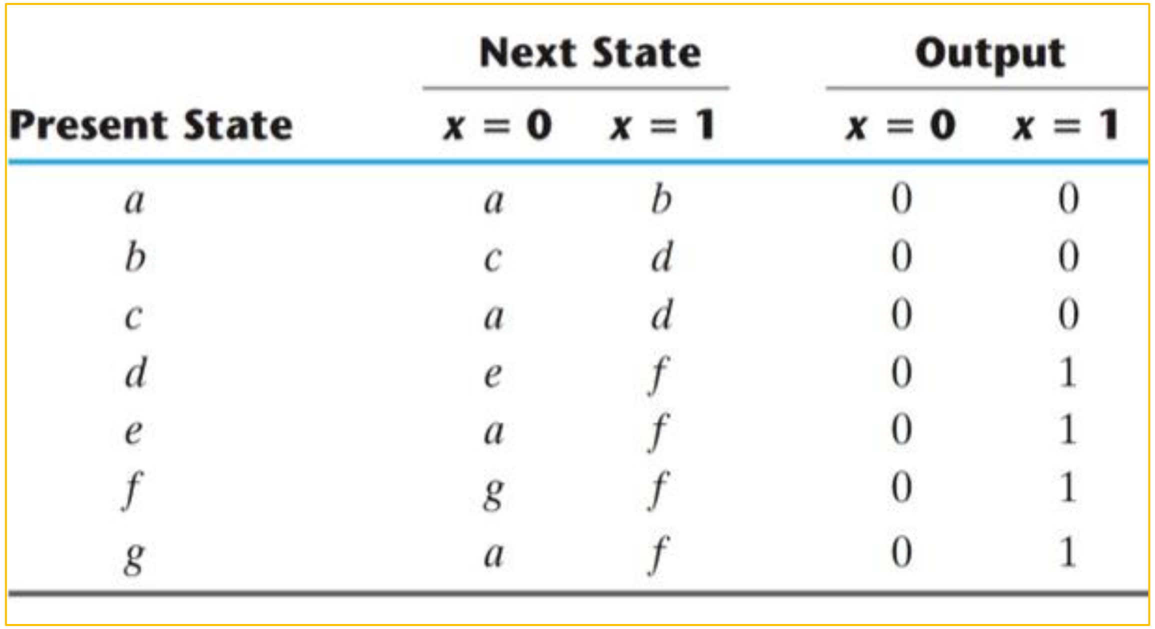
Steps 6

$State\ \boldsymbol{e} \equiv State\ \boldsymbol{g}$

$State\ \boldsymbol{d} \equiv State\ \boldsymbol{f}$

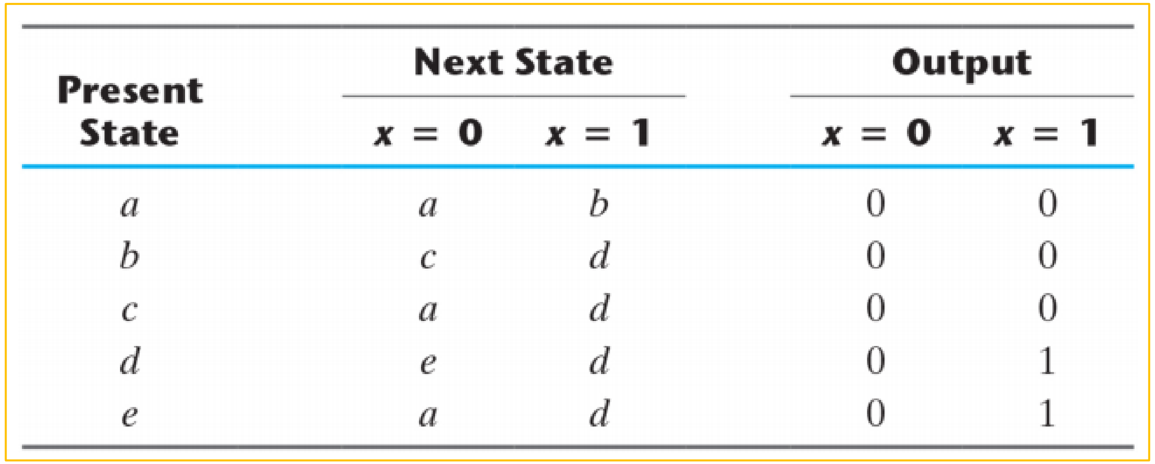
State Reduction: Implication Table Method

Example:



Present State	Next State		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1
e	a	f	0	1
f	g	f	0	1
g	a	f	0	1

Steps 7



Present State	Next State		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	d	0	1
e	a	d	0	1

$State\ \boldsymbol{e} \equiv State\ \boldsymbol{g}$

e and **g** are equivalent →

1) **Remove g** row
2) **Replace** g with **e** whenever it occurs as next State

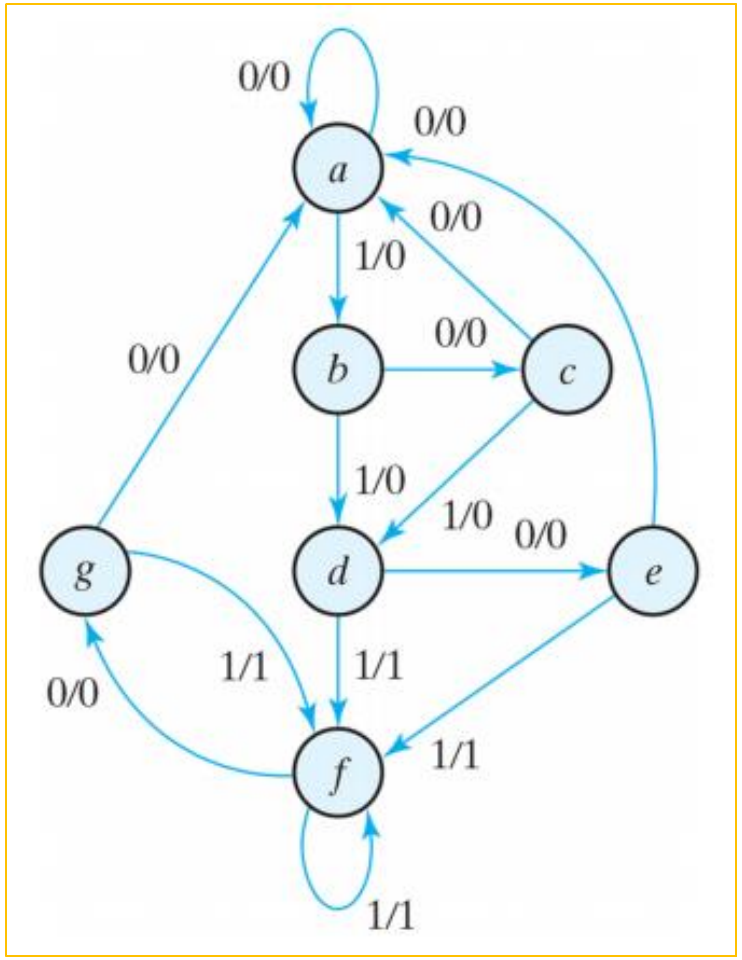
$State\ \boldsymbol{d} \equiv State\ \boldsymbol{f}$

d and **f** are equivalent →

1) **Remove f** row
2) **Replace** f with **d** whenever it occurs as next State

State Reduction – Alternative Method (Row Matching Method)

Example:



Present State	Next State		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1
e	a	f	0	1
f	g	f	0	1
g	a	f	0	1

State ***e*** $\equiv$ *State* ***g***

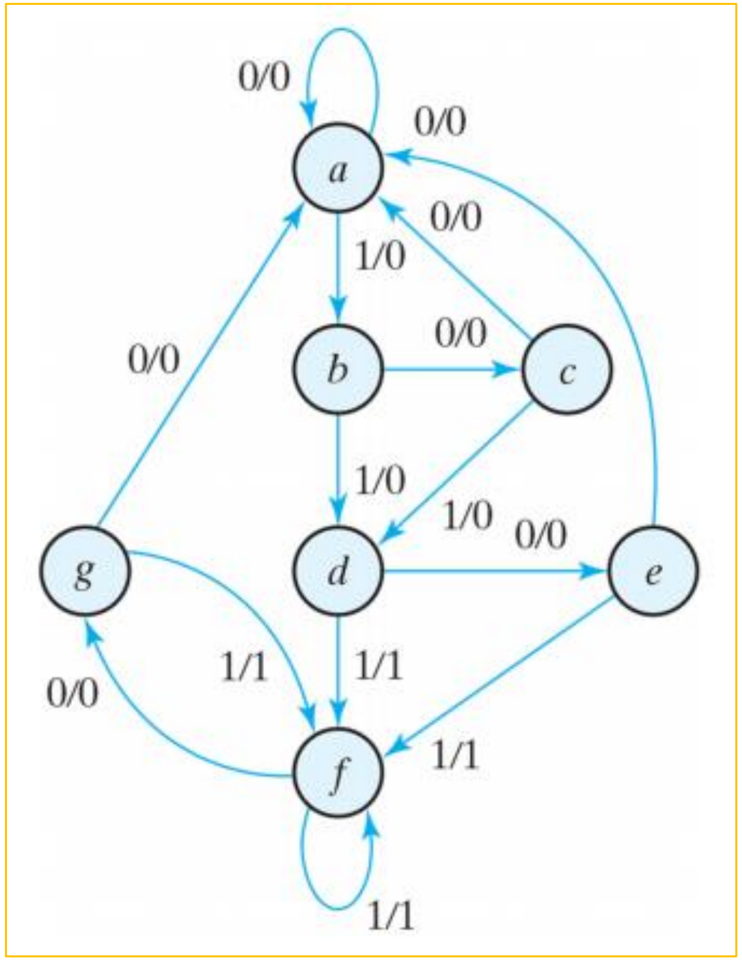
e and **g** are equivalent →

1) **Remove g** row
2) **Replace** g with **e** whenever it occurs as next State

Present State	Next State		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1
e	a	f	0	1
f	e	f	0	1

State Reduction – Alternative Method (Row Matching Method)

Example:



Present State	Next State $x = 0$	Next State $x = 1$	Output $x = 0$	Output $x = 1$
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1
e	a	f	0	1
f	e	f	0	1

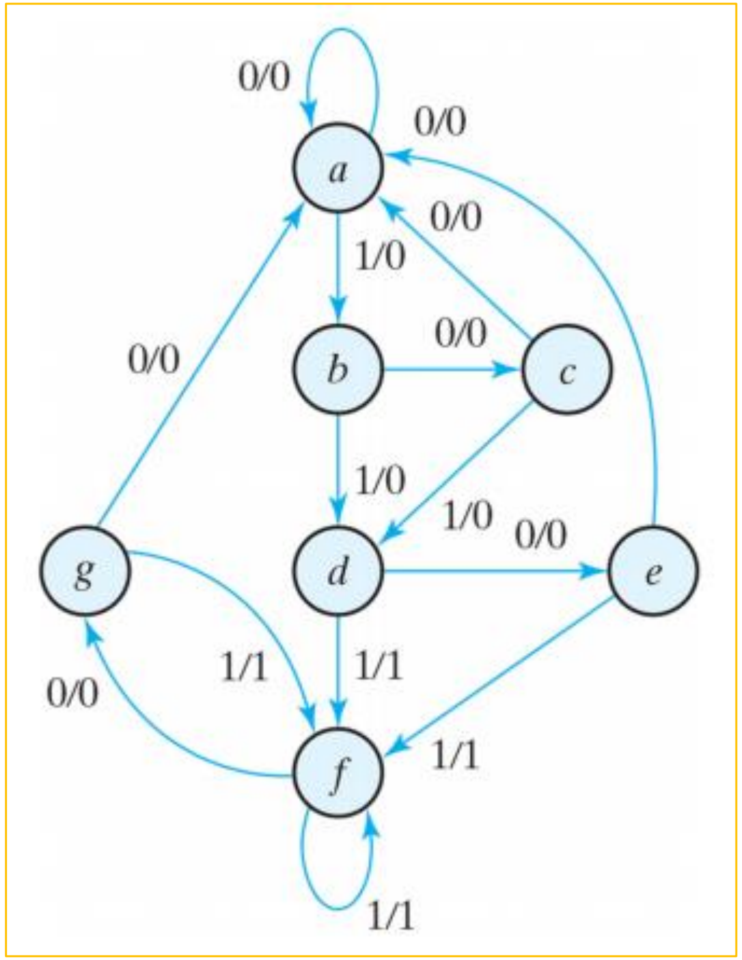
*State **d** ≡ State **f***

d and **f** are equivalent →

1) **Remove f** row
2) **Replace** f with **d** whenever it occurs as next State

Present State	Next State $x = 0$	Next State $x = 1$	Output $x = 0$	Output $x = 1$
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	d	0	1
e	a	d	0	1

Example:

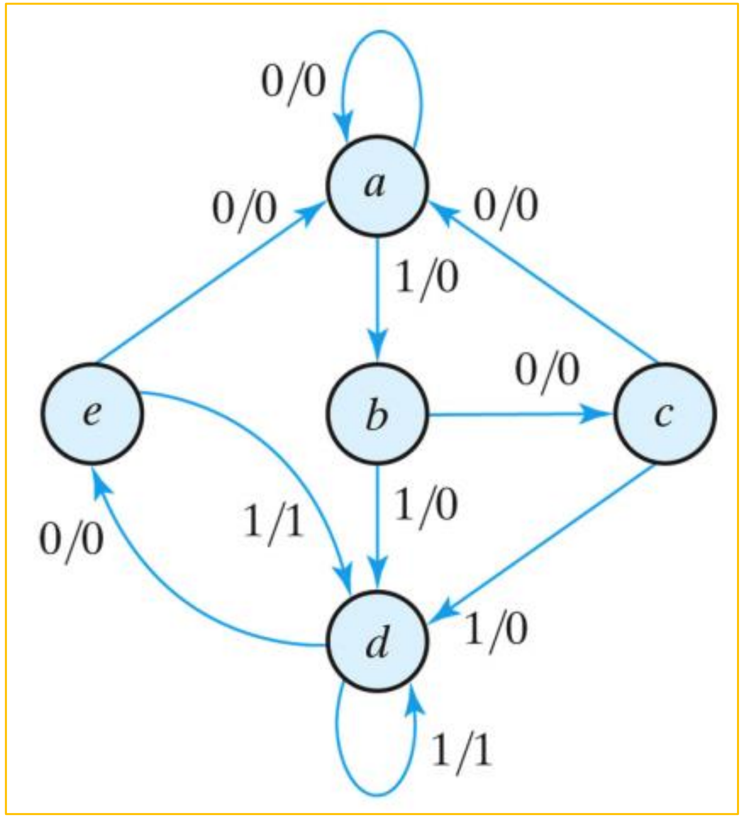


Reduced from 7 – 5 states

To cover 7 states → 3 FFs is needed
To cover 5 states → 3 FFs is needed

Notice: No. FFs (m) $\geq \lceil log_2(No. States) \rceil$

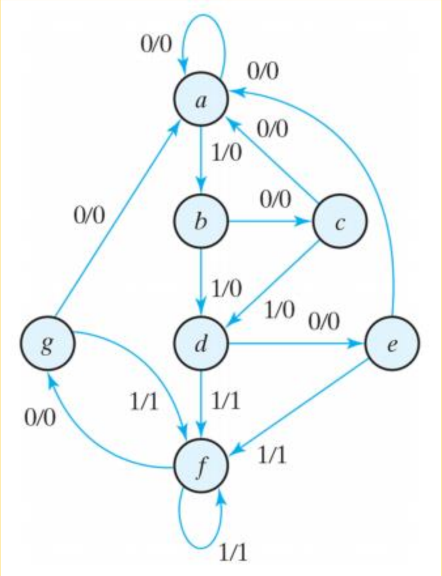
Ceiling Function

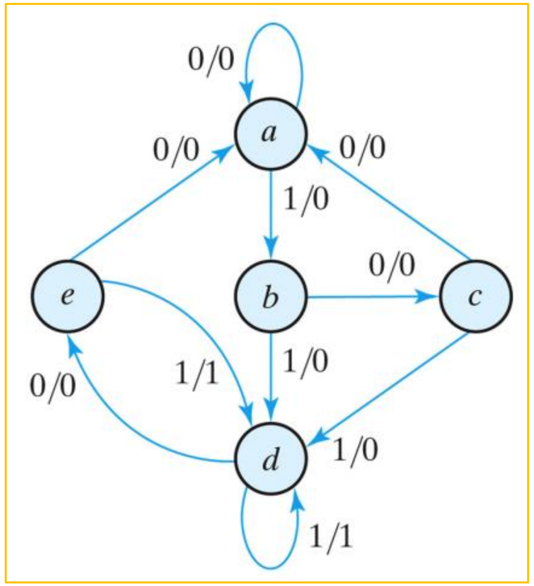


We did **NOT** save any **FF** in this design, but we have **more Unused states** that we can substitute for as **don't cares during design**, thus **simplifying** some combinational design logic.

Notice:
No. **Unused** Sates $= \boldsymbol{Max}\ No. States - No. \boldsymbol{Used}\ States$
$= 2^3 - 5 = \mathbf{3}$

Example: *Verify Reduction*





State	a	a	b	c	d	e	f	f	g	f	g	a
Input	0	1	0	1	0	1	1	0	1	0	0	
Output	0	0	0	0	0	1	1	0	1	0	0	

Identical Output Sequence

State	a	a	b	c	d	e	d	d	e	d	e	a
Input	0	1	0	1	0	1	1	0	1	0	0	
Output	0	0	0	0	0	1	1	0	1	0	0	

Extra Example: (Moore FSM)

n = 8 → 7x7 Table

Present State	Next State		Present Output
	X=0	X=1	
a	g	c	0
b	f	h	0
c	e	d	1
d	a	c	0
e	c	a	1
f	f	b	1
g	a	c	0
h	c	g	1

b	(g=f) (c=h)✗						
c	X	X					
d	(g=a)✓	(f=a)✗ (h=c)	X				
e	X	X	(d=a)✓	X			
f	X	X	(e=f) (d=b)✗	X	(c=f) (a=b)✗		
g	✓	(f=a)✗ (h=c)	X	✓	X	X	
h	X	X	(e=c)✓ (d=g)✓	X	(a=g)✓	(f=c) (b=g)✗	X
	a	b	c	d	e	f	g

*State **a** ≡ State **d** ≡ State **g***

*State **c** ≡ State **e** ≡ State **h***

→ ***Four Sates*** **(a , b, c, f)**

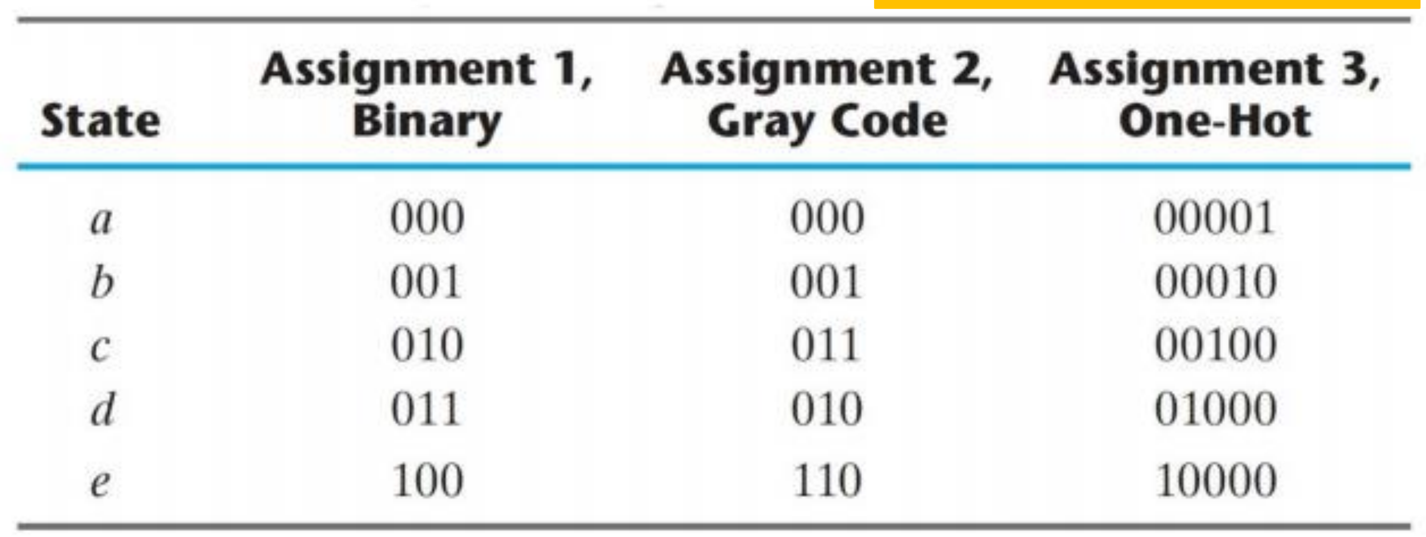
→ 8 states → 4 states
3 FFs → 2 FFs

State Assignment

- it is necessary to assign **unique** coded **binary values** to the **states**
- For a circuit with **m** states, the codes must contain at least **n** bits, where $2^n \geq m$
  - This may generate $(2^n - m)$ **unused** states

- A **different** assignment will result in a state table with different binary values for the states.
- The **complexity** of the combinational circuit **depends** on the binary state **assignment chosen.**

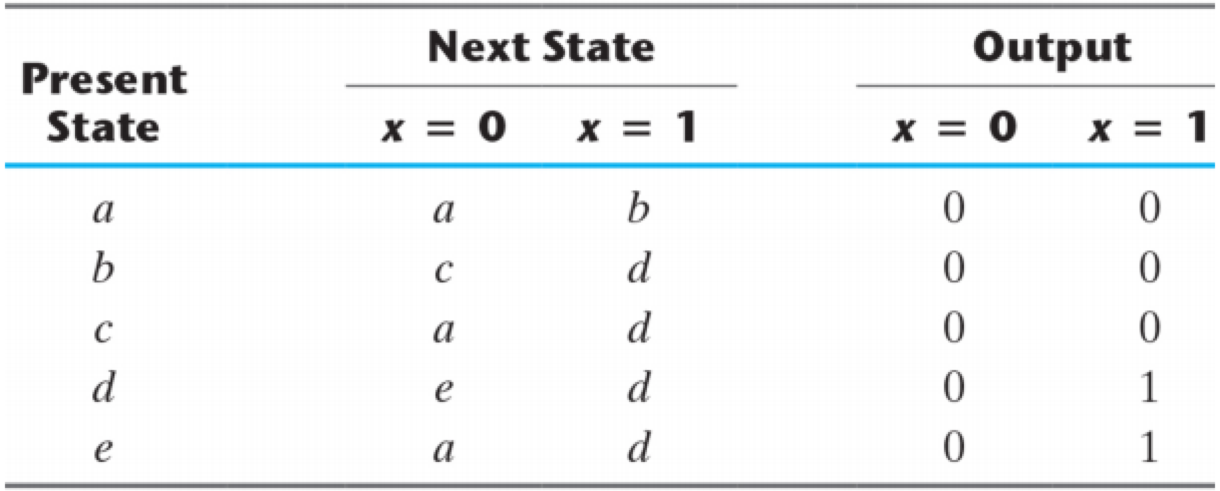
State Assignment Methods



State	Assignment 1, Binary	Assignment 2, Gray Code	Assignment 3, One-Hot
a	000	000	00001
b	001	001	00010
c	010	011	00100
d	011	010	01000
e	100	110	10000

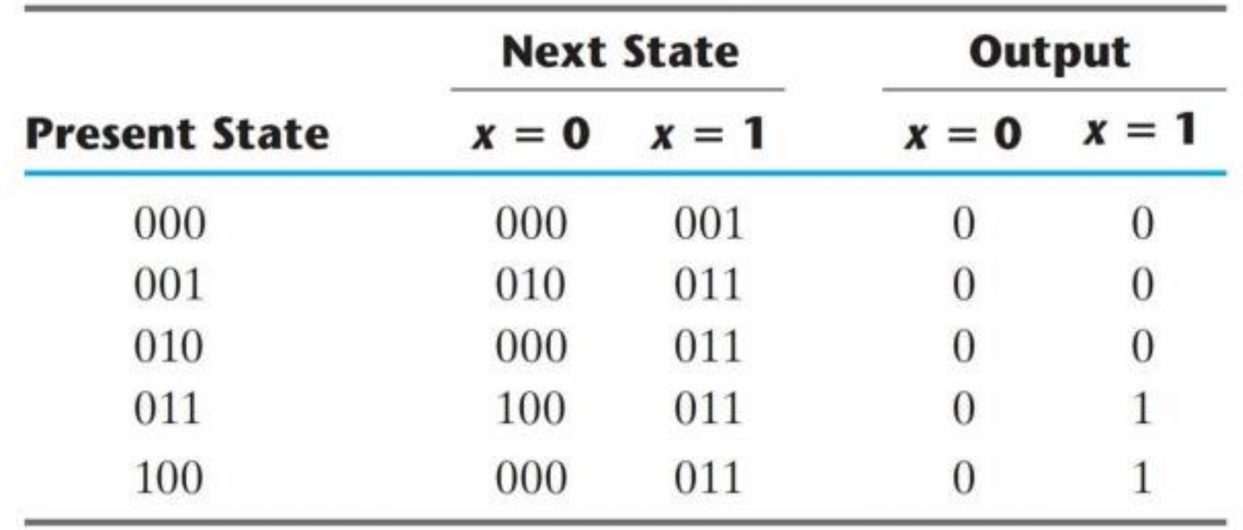
Notice: **One-hot** requires extra FFS, but usually leads to **simpler** logic for the next state and output

- A **state table** with a binary assignment is called a **Transition table**



Present State	Next State		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	d	0	1
e	a	d	0	1

State Table



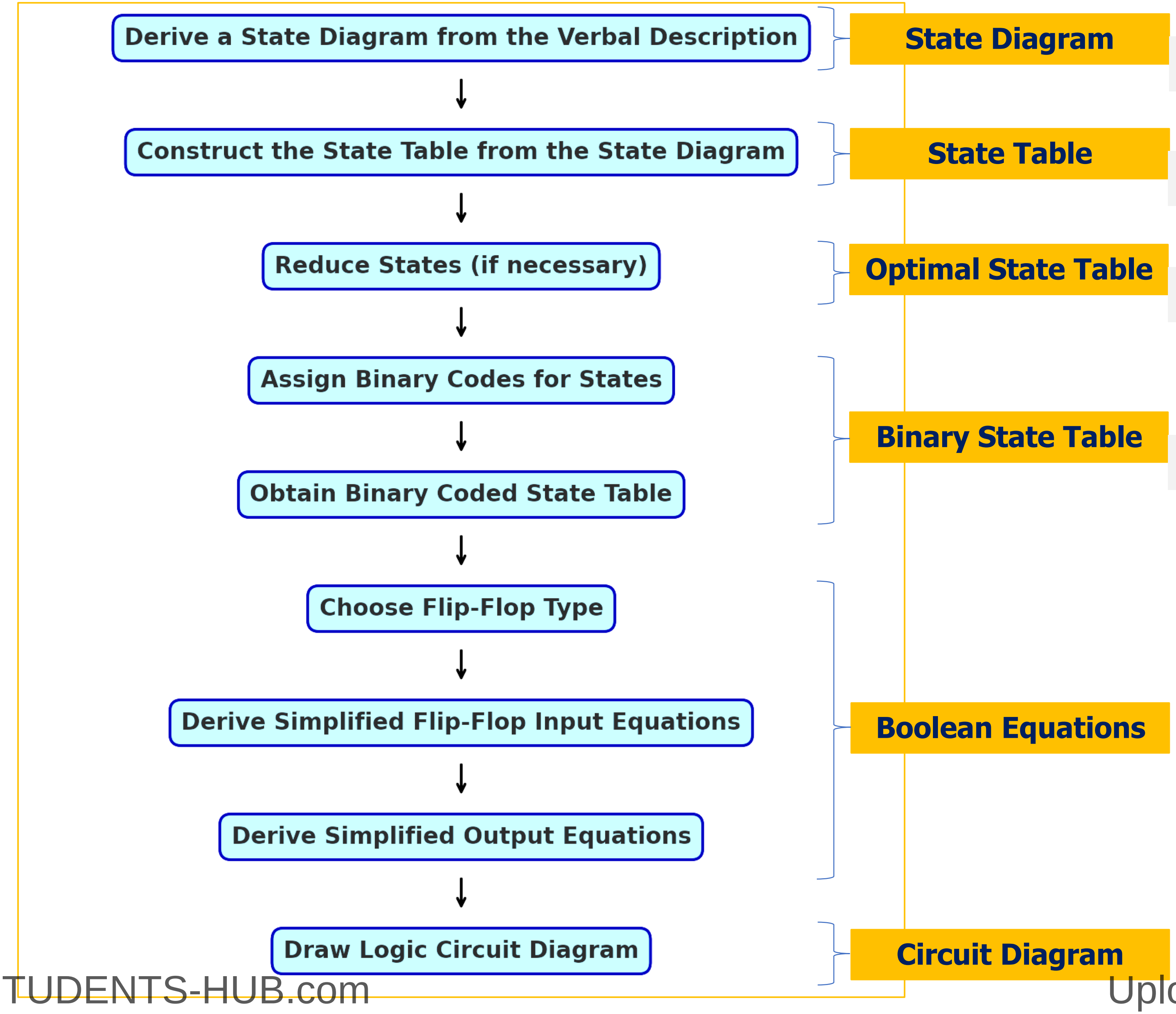
Present State	Next State		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
000	000	001	0	0
001	010	011	0	0
010	000	011	0	0
011	100	011	0	1
100	000	011	0	1

Notice: 101, 110, 111 → **Unused** States

Transition Table

- **Recall**: **Design** goal is to specify the **hardware/circuit** that will **implement** a desired behavior
  - **Starts** from a **set of specifications** and **conclude** with a **logic diagram**
- **Recall**: Sequential circuits are made up of flip-flops and combinational logic that may influence the flip-flops inputs and/or the circuit outputs
- Once the type and number of flip-flops are determined, the design problem turns to a combinational circuit design problem
- The **D-FF** is the **basic** storage element from which all others are derived using additional combinational logic
- **Remember**, number of **FF depends** on number of **states**

Design Procedure of Sequential Circuits



Derive a State Diagram from the Verbal Description → **State Diagram**

Decide on Moore or Mealy FSM

Construct the State Table from the State Diagram → **State Table**

Use 2D or 1D Form

Reduce States (if necessary) → **Optimal State Table**

Use Implication Table Method

Assign Binary Codes for States

Obtain Binary Coded State Table → **Binary State Table**

Use Suitable State Assignment Method

Choose Flip-Flop Type

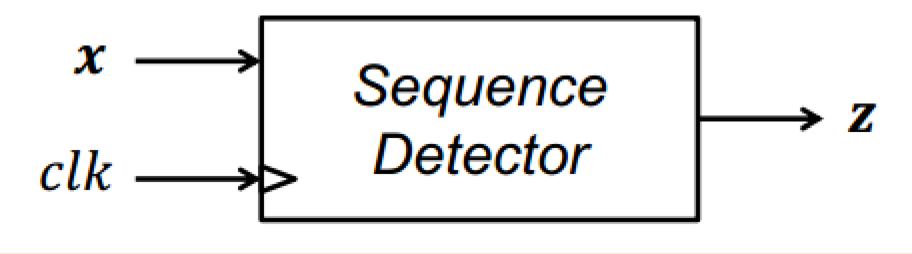
Derive Simplified Flip-Flop Input Equations

Derive Simplified Output Equations → **Boolean Equations**

Draw Logic Circuit Diagram → **Circuit Diagram**

Example: Sequence Detector

- This kind of sequential circuits is commonly known as A **sequence detector**
- It **Detects** a specific **sequence of bits** in the **input**
- The input is a serial bit stream: **One** input bit x is fed to the sequence detector **each cycle**
- The output is also a bit stream: **One** output bit z each cycle **indicates** whether a given sequence is **detected or NOT**

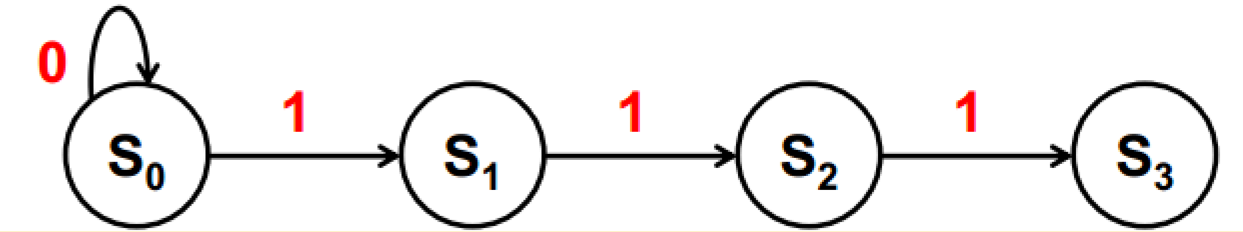


Verbal Description

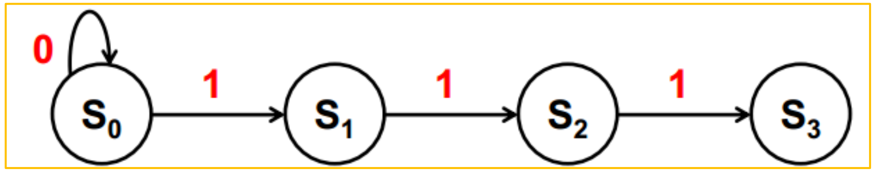
Example: Design a **sequence detector** that will **detect three or more consecutive 1's**. When the three **consecutive 1's** (**111**) are detected, the **output** is **set** to **1** for **one clock cycle**. If any **further consecutive 1** is detected the **output remains** to be **1**. If a **zero** occurs in between the 1′ s, the **output** become **0** and the detector goes to its **initial state** and starts all over again.

State Diagram

1) Begin in an **initial** state: call it $\mathbf{S_0}$
2) $\mathbf{S_0}$ indicates that a **1** is **NOT detected** yet
3) As long as the input x is 0, remain in the initial state $\mathbf{S_0}$
4) Add a state (call it $\mathbf{S_1}$) that detects the **first** "**1**" in the input
5) Add a state (call it $\mathbf{S_2}$) that detects the input sequence "**11**"
6) Add a state (call it $\mathbf{S_3}$) that detects the input sequence "**111**"



Example: Sequence Detector **State Diagram**



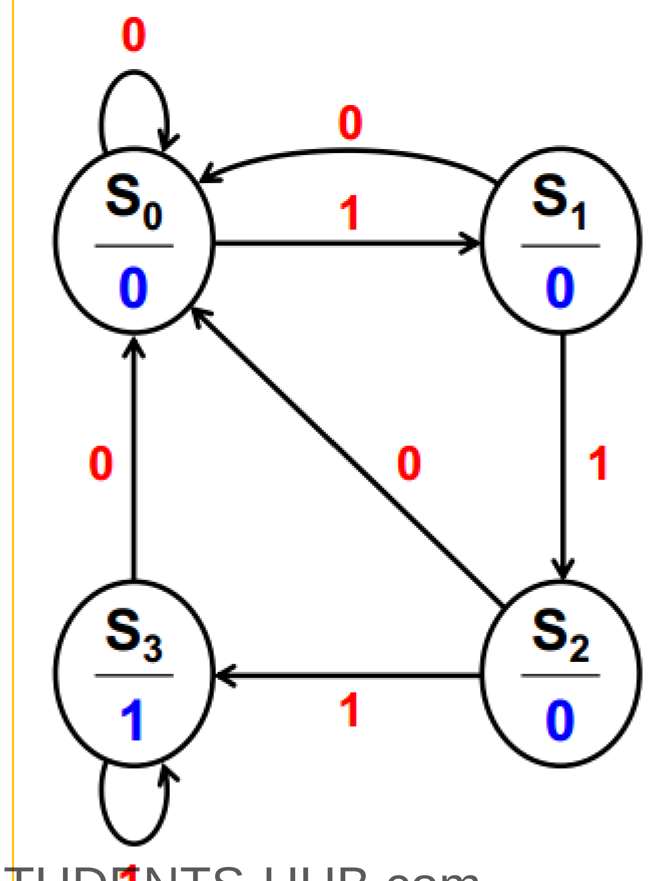
Complete the State Diagram

- When the **input** is **0**: Add transitions from S_1, S_2, and S_3 **back to S_0**
- When the **input** is **1**: Add transition from S3 **to itself** to detect sequences longer than **three 1's**

Consider *Moore FSM*

Assign Output to States

- The output in S_0, S_1, and S_2 should be **0**
- The output in S_3 should be **1**



State Reduction is NOT necessary

State Table

Present State	Next State $x = 0$	Next State $x = 1$	Output z
S_0	S_0	S_1	0
S_1	S_0	S_2	0
S_2	S_0	S_3	0
S_3	S_0	S_3	1

Example: **Sequence Detector** **State Assignment**

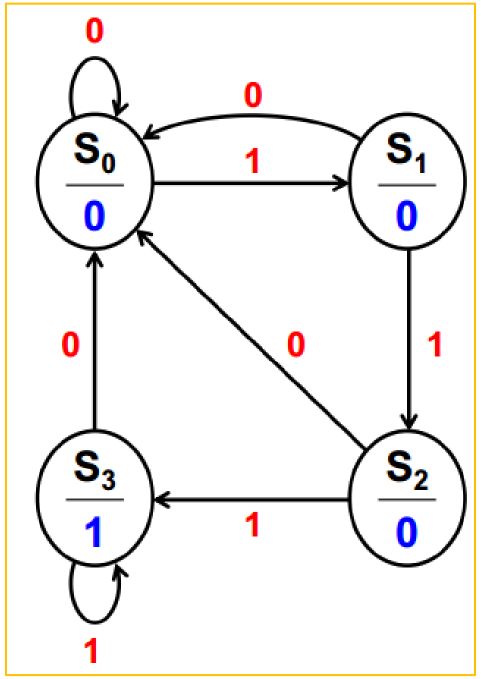
Recall: for m states

The minimum number of state bits: $n = \lceil \log_2 m \rceil$

$\lceil x \rceil$ is the smallest integer ≥ x *(ceiling function)*

- **Each** state must be assigned a **unique** binary code
- In this example:
  - **Four** states: S_0, S_1, S_2, and S_3 ($m=4$) → Minimum number of state bits (FFs) is ($n=2$)
- State assignment: $\mathbf{S_0 = 00, S_1 = 01, S_2 = 10,}$ and $\mathbf{S_3 = 11}$
- **Recall**: If n bits are **used**, the number of unused states = $(2^n - m)$
  - In this example, there are **NO unused** states

Example: Sequence Detector



State Assignment

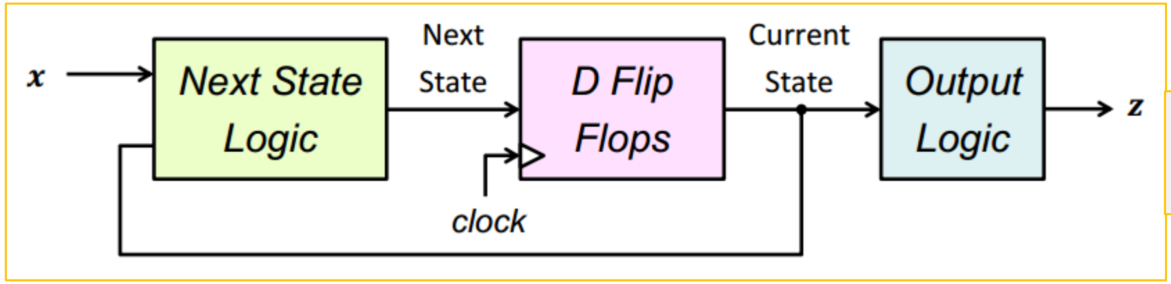
S_0 = 00, S_1 = 01

S_2 = 10, S_3 = 11

Binary-Coded State Table

Present State	Next State		Output
	x = 0	x = 1	z
0 0	0 0	0 1	0
0 1	0 0	1 0	0
1 0	0 0	1 1	0
1 1	0 0	1 1	1

Recall: Moore FSM Structure



In this Design, Only **D-FF** will be used

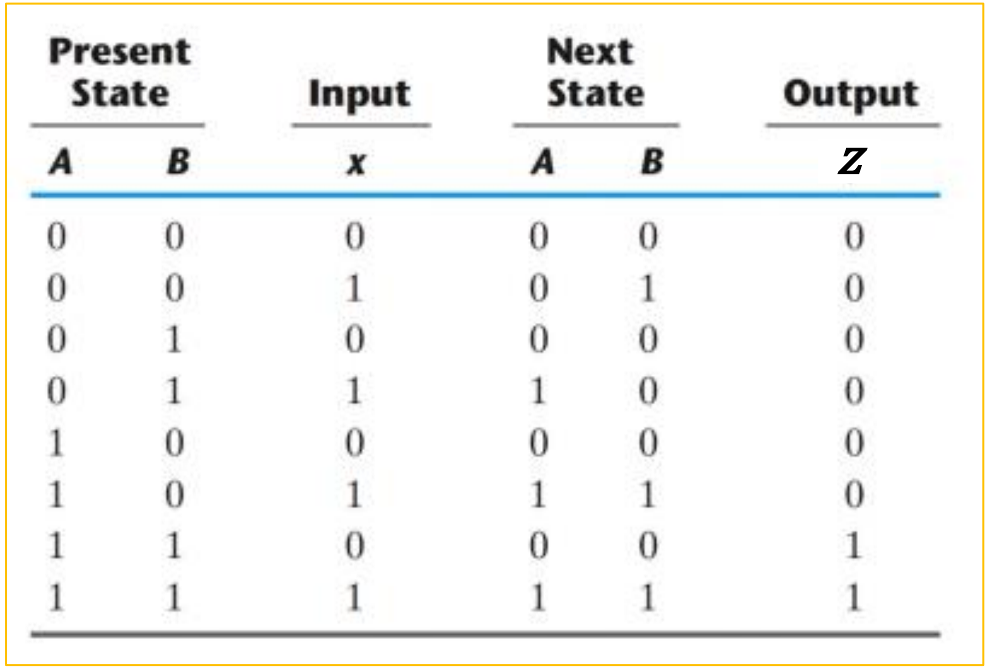
Example: Sequence Detector **Boolean Equations (D-FF)**

Present State	Next State		Output
	x = 0	x = 1	z
0 0	0 0	0 1	0
0 1	0 0	1 0	0
1 0	0 0	1 1	0
1 1	0 0	1 1	1

2 State Bits → **2** Flip-Flops

Label: State Bits (FFs) → **A,B**

1-D form is easier to derive Equations



Present State		Input	Next State		Output
A	*B*	*x*	*A*	*B*	*Z*
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	1	1	1	0
1	1	0	0	0	1
1	1	1	1	1	1

Recall: When Using **D-FF**

Next State = Flip-Flop Inputs D_1 and D_0

Example: Sequence Detector — Boolean Equations (D-FF)

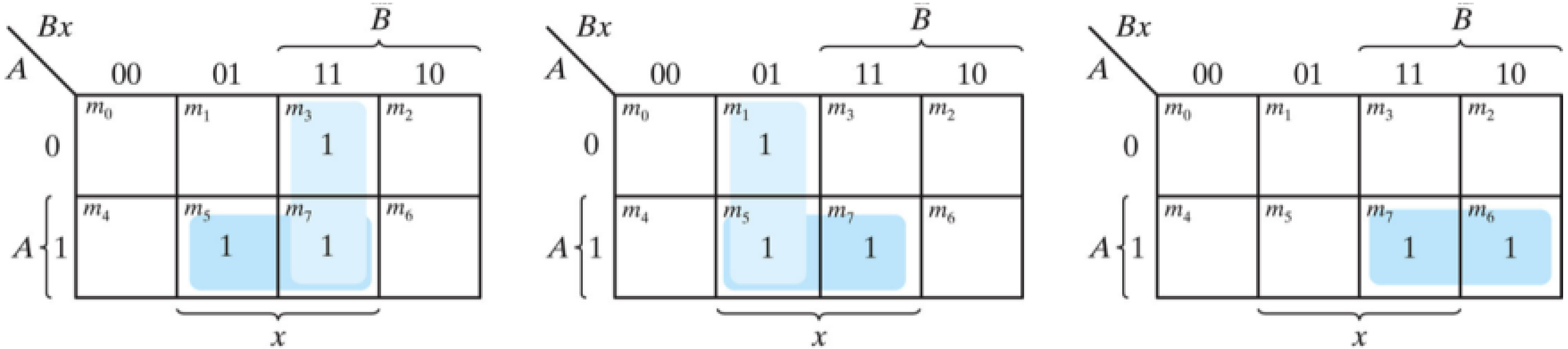
Present State		Input	Next State		Output
A	B	x	A	B	Z
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	1	1	1	0
1	1	0	0	0	1
1	1	1	1	1	1

$A(t+1) = D_A(A,B,x) = \Sigma(3, 5, 7)$

$B(t+1) = D_B(A,B,x) = \Sigma(1, 5, 7)$

$Z(A,B,x) = \Sigma(6,7)$

FF-Inputs/Output Simplification

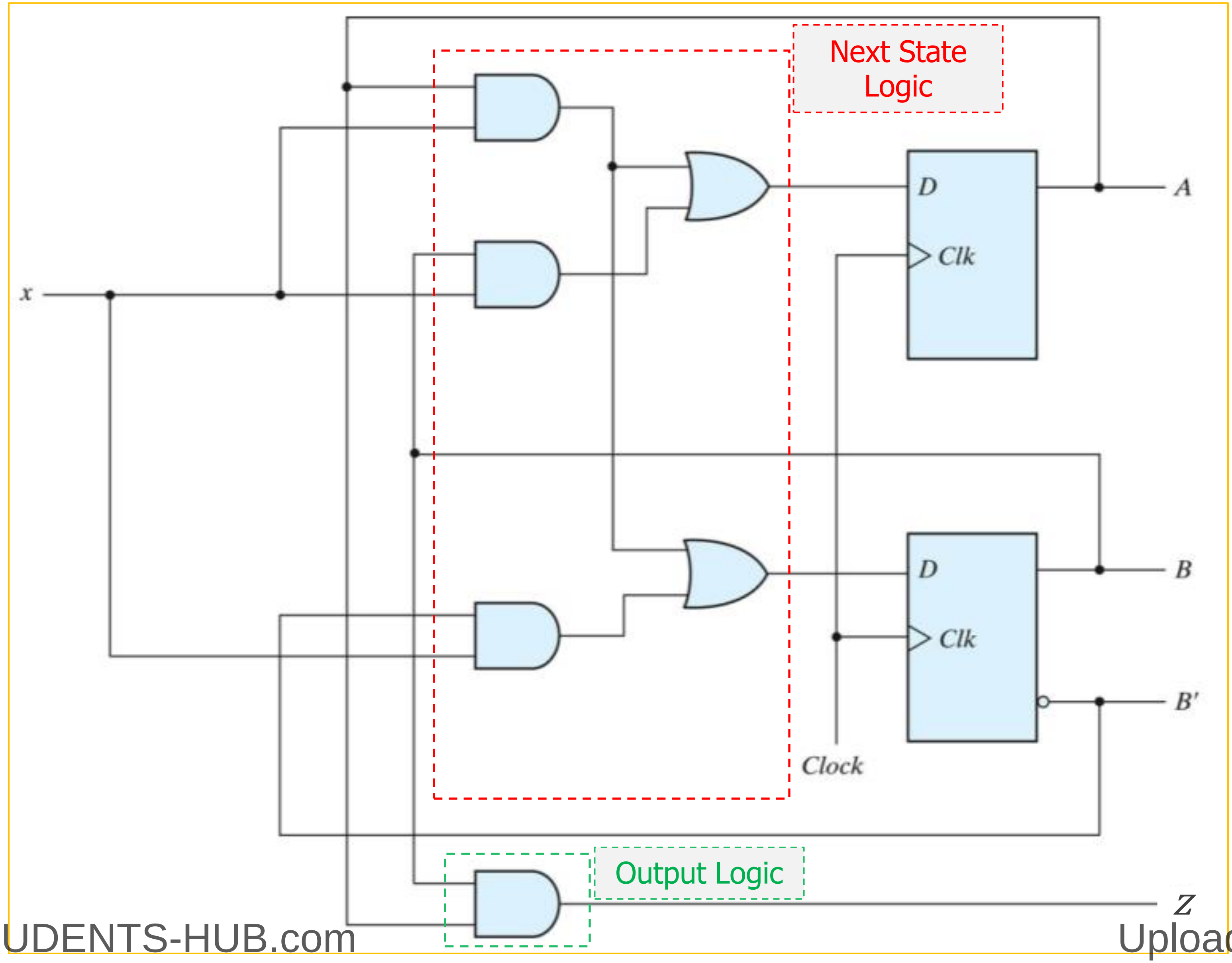


$D_A = Ax + Bx$

$D_B = Ax + B'x$

$Z = AB$

Example: Sequence Detector **Logic Circuit Diagram**



$D_A = Ax + Bx$

$D_B = Ax + B'x$

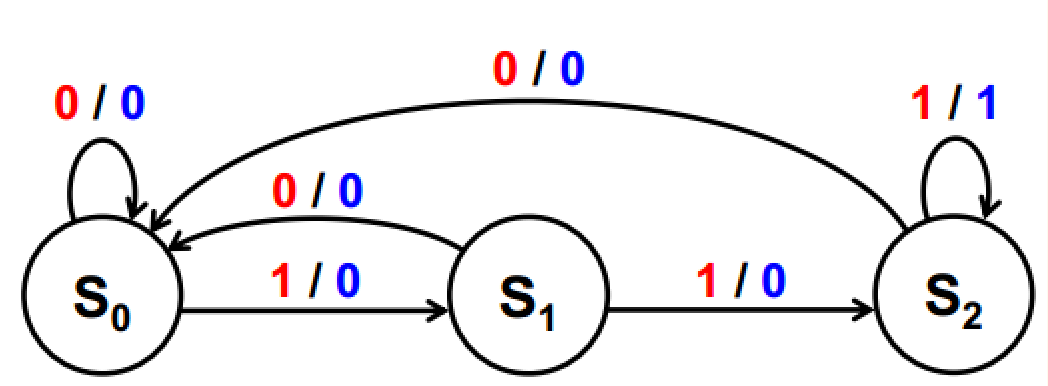
$Z = AB$

Example: Sequence Detector – Mealy FSM

Recall: in **Mealy** FSM

Each line/arrow is **labeled** with: **Input** / **Output**
The **output** is shown on the line/arrow of the state diagram

Change in State Diagram
At S_2, when the next **input** is 1 → the **output** should be 1
Make a **transition** from S_2 **back to itself** labeled **1 / 1**
No need for state S_3, because output is on the line/arrow



State Diagram

Mealy FSM
typically use
less states than
Moore FSM

Present State	Next State		Output z	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
S_0	S_0	S_1	0	0
S_1	S_0	S_2	0	0
S_2	S_0	S_2	0	1

State Table

State Assignment
$S_0 = 00$, $S_1 = 01$
$S_2 = 10$

State 11 → Unused

Binary-Coded State Table

Present State	Next State		Output z	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
0 0	0 0	0 1	0	0
0 1	0 0	1 0	0	0
1 0	0 0	1 0	0	1

Example: Sequence Detector – Mealy FSM

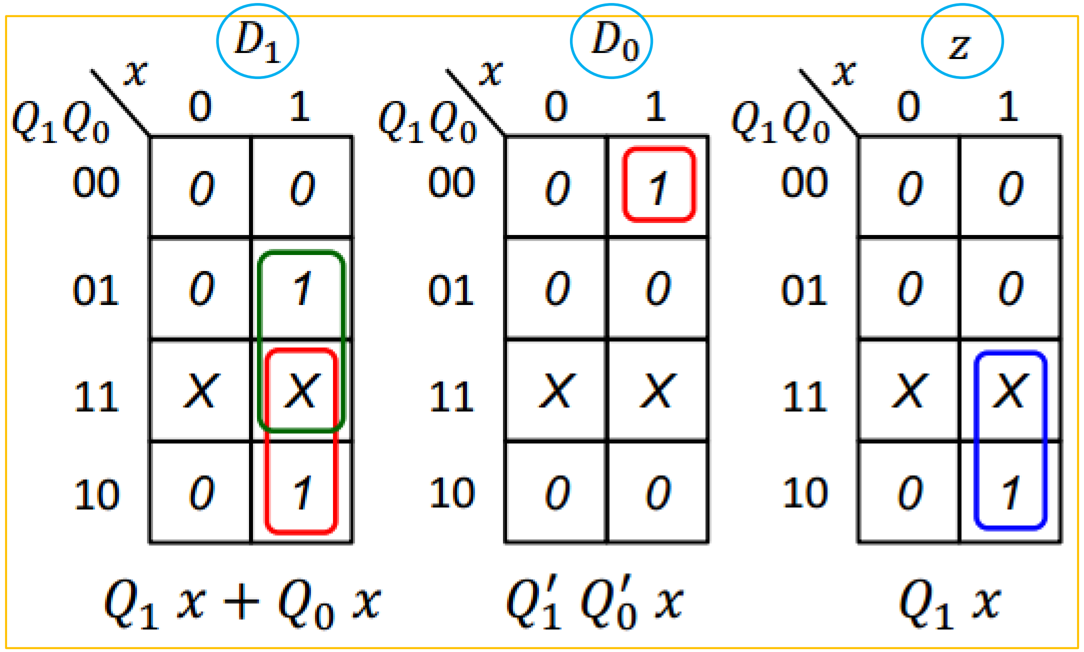
Binary-Coded State Table

Present State	Next State		Output z	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
0 0	0 0	0 1	0	0
0 1	0 0	1 0	0	0
1 0	0 0	1 0	0	1
1 1	X X	X X	X	X

Unused States → **Don't Care Conditions in Next States & Output**

2 State Bits → **2** Flip-Flops

Label: State Bits (FFs) → **Q_1, Q_0**



D_1

$Q_1Q_0 \backslash x$	0	1
00	0	0
01	0	1
11	X	X
10	0	1

$Q_1\,x + Q_0\,x$

D_0

$Q_1Q_0 \backslash x$	0	1
00	0	1
01	0	0
11	X	X
10	0	0

$Q_1'\,Q_0'\,x$

z

$Q_1Q_0 \backslash x$	0	1
00	0	0
01	0	0
11	X	X
10	0	1

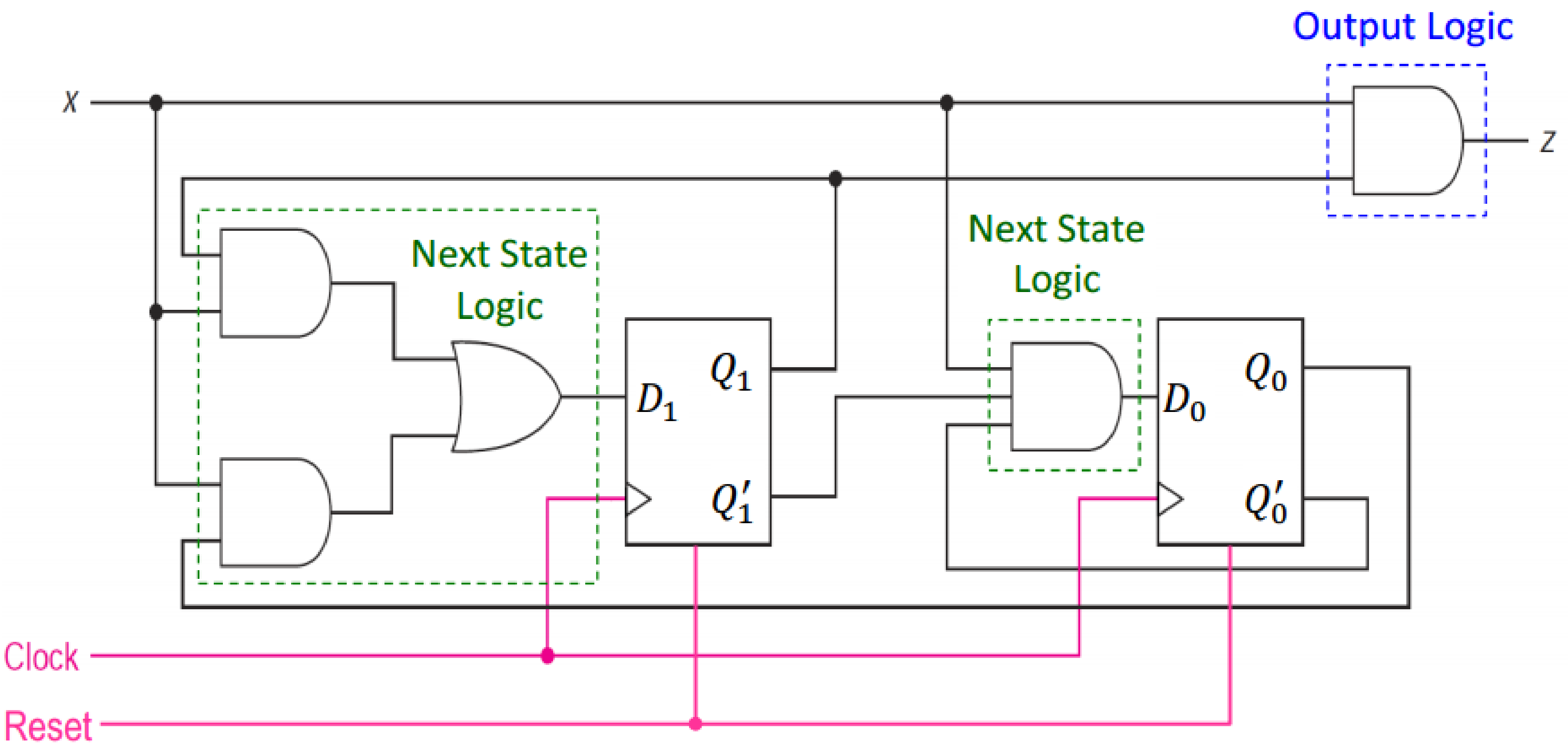
$Q_1\,x$

Boolean Equations (D-FF)

Example: Sequence Detector – Mealy FSM

Logic Circuit Diagram

$$D_1 = Q_1\,x + Q_0\,x \qquad D_0 = Q_1'\,Q_0'\,x \qquad z = Q_1\,x$$



Design/Circuit Verification

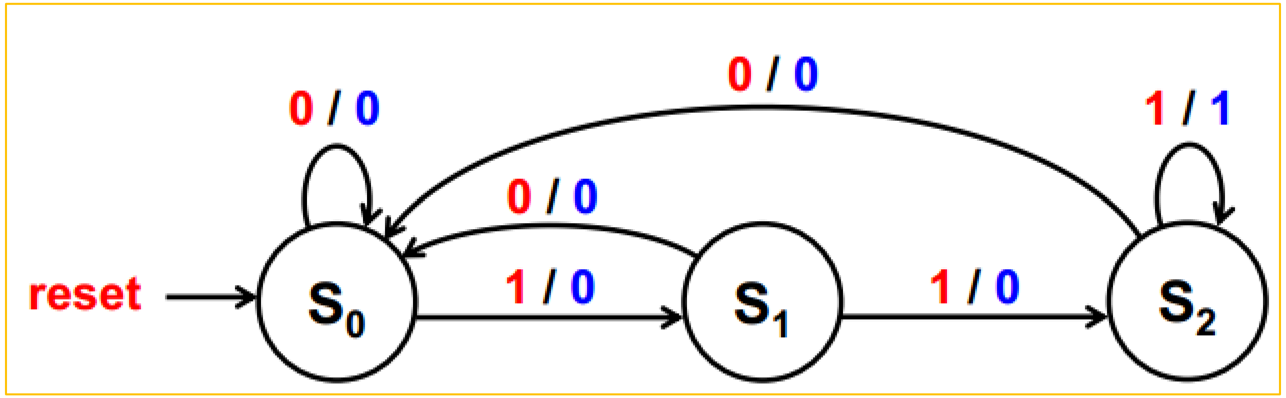
- Sequential circuits should be **verified** by showing that the circuit **produces** the **original** state diagram
- Verification can be done **manually**, or with the help of a **simulation** program
- **All** possible input **combinations** are applied at **each** state and the **state variables and outputs** are **observed**
- A **reset** input is used to **reset** the circuit to its **initial** state
- Apply a **sequence** of **inputs** to **test** **all the state-input combinations**, i.e., **all transitions** in the state diagram
- **Observe** the **output and the next state** that appears **after each clock edge** in the timing diagram

Input Test Sequence

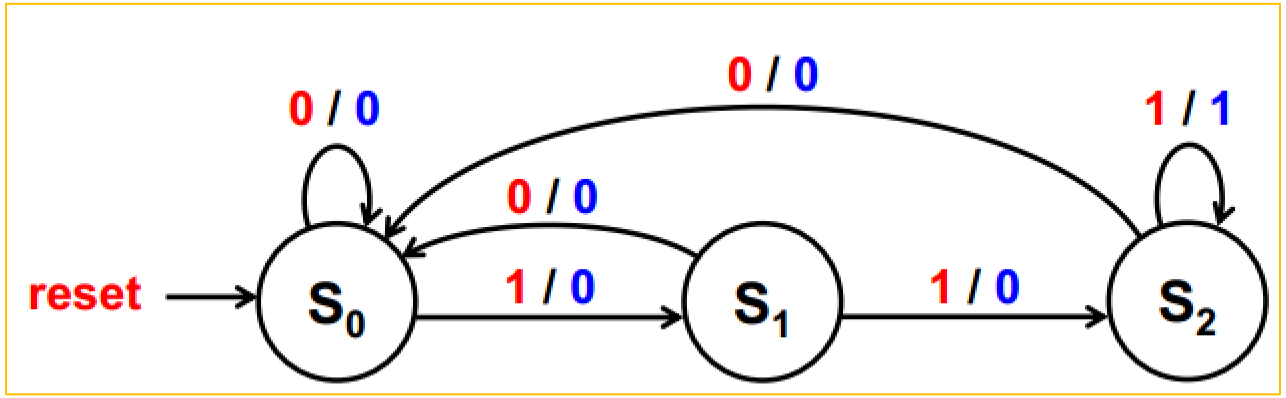
- Required to **verify** **the correct operation** of a sequential circuit
- It should **test** **each state transition** of the state diagram
- Test sequences can be **produced from the state diagram**
- Consider the **Mealy** sequence detector, starting at S_0 (reset), we can use the following input test sequence to verify all state transitions:

Input test sequence: **reset** then x = **0, 1, 0, 1, 1, 0, 1, 1, 1, 1**

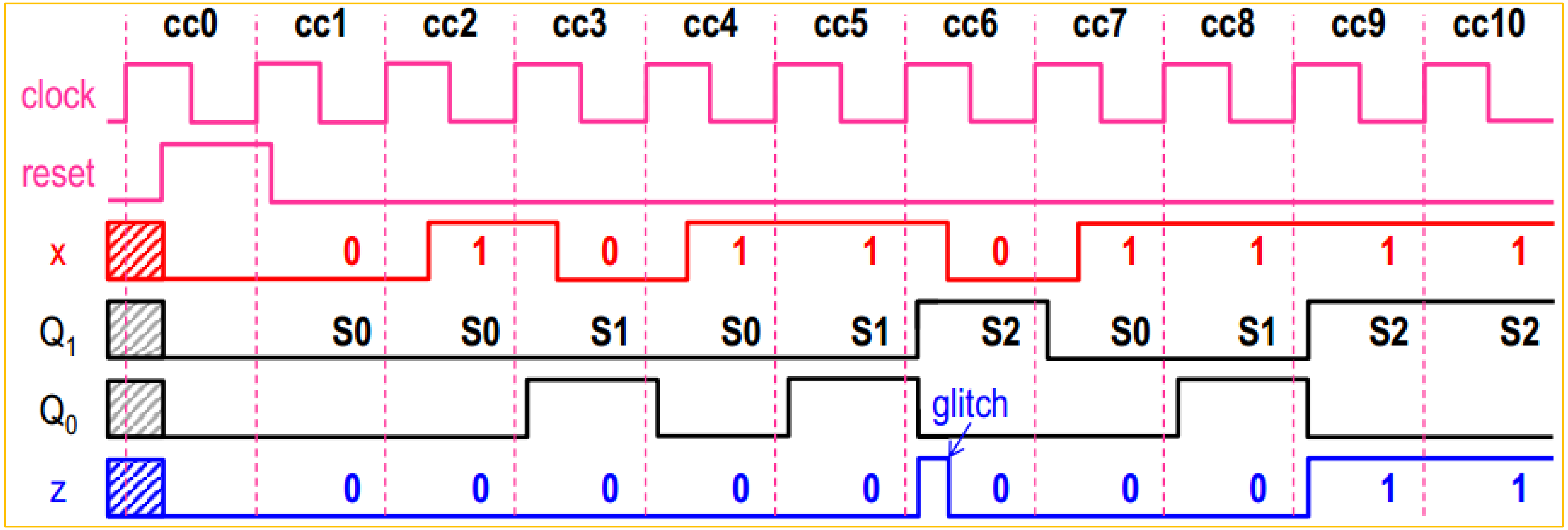
Reset input forces initial state to be $\boldsymbol{S_0}$



Sequence Detector – Mealy FSM Timing Diagram

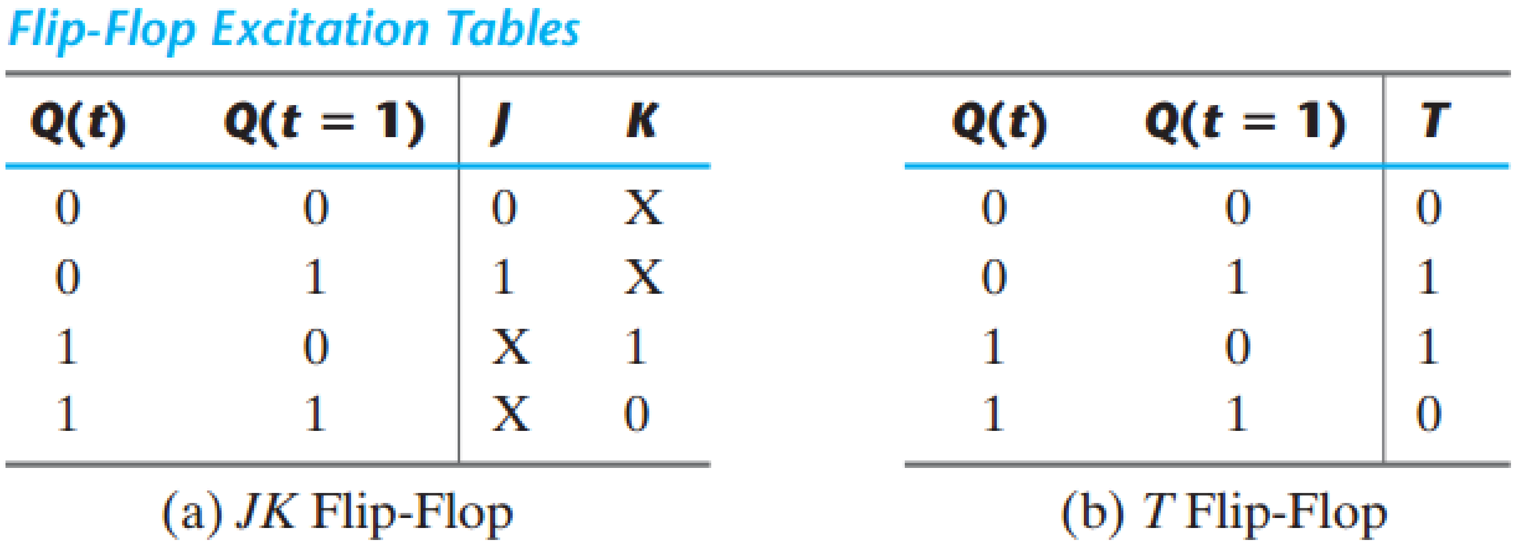


Input test sequence: **reset** then x = **0, 1, 0, 1, 1, 0, 1, 1, 1, 1**



Recall: The drawback of Mealy is that **glitches** can appear in the output if the input is not synchronized with the clock

- **Recall**: In the design of clocked sequential circuits, we **know** the **present** state and **next** state of the flip-flops
- We need to find the flip-flop **input functions/equations** and the **output functions /equations** in order to design the combinational circuit part of the circuit
- These **functions/equations** can be easily obtained using the **flip flop excitation table**
- **Excitation Tables** give us the **required inputs** that will **achieve** a given **transition** from preset state (PS) to next state (NS) [i.e. Q_t to Q_{t+1}]



Flip-Flop Excitation Tables

Q(t)	Q(t = 1)	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

(a) *JK* Flip-Flop

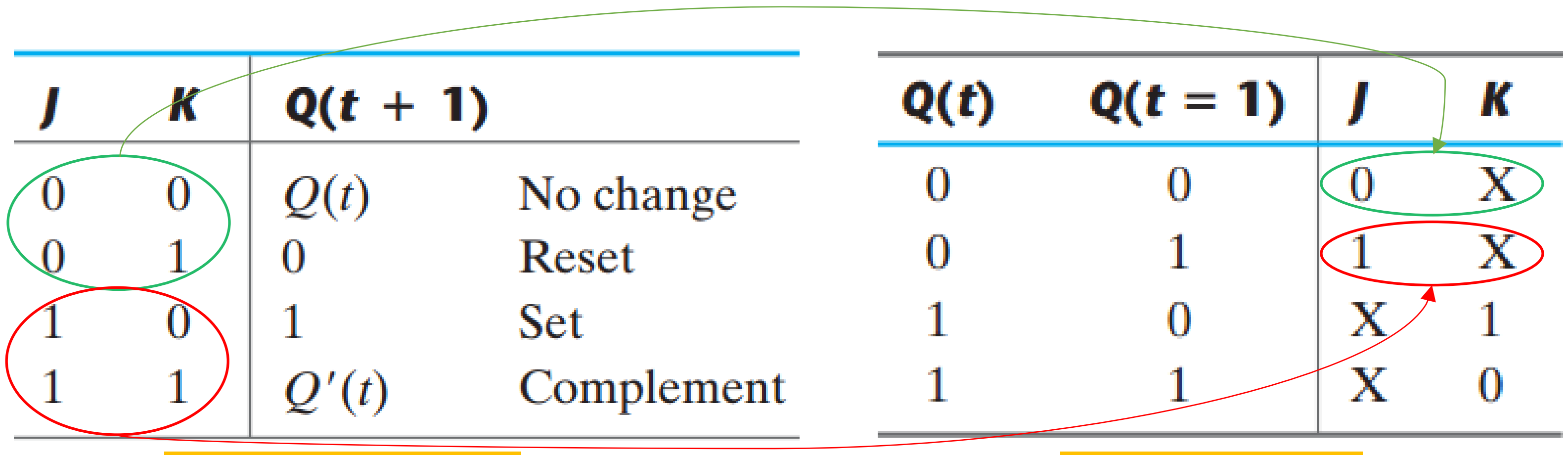
Q(t)	Q(t = 1)	T
0	0	0
0	1	1
1	0	1
1	1	0

(b) *T* Flip-Flop

What about D Flip-Flop Excitation Table?

List the inputs that will **cause** the **state change** in the **state table**

Design of Sequential Circuits: JK-FF



J	K	Q(t + 1)	
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	$Q'(t)$	Complement

Characteristic Table

Q(t)	Q(t = 1)	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

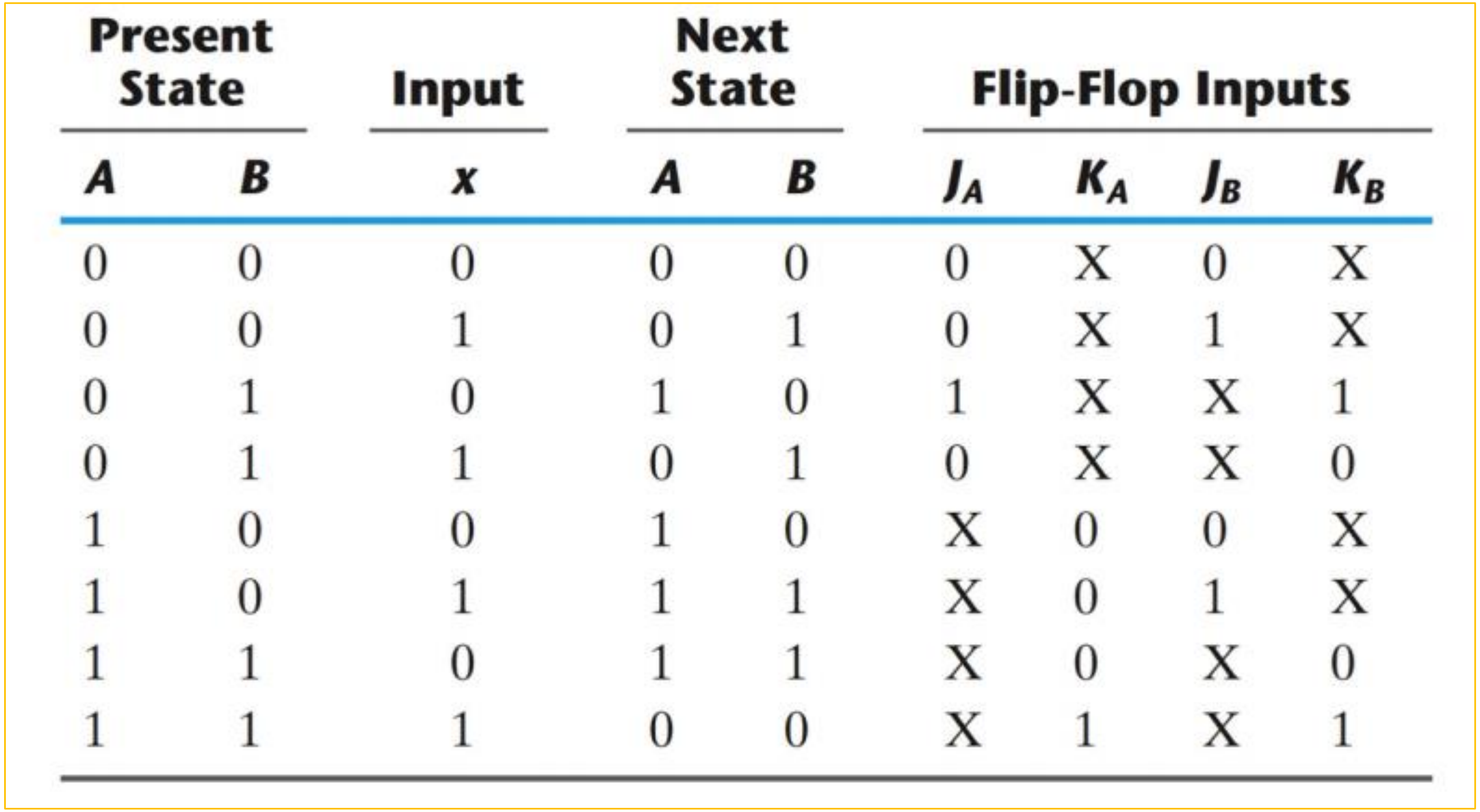
Excitation Table

Recall: If **present** state is **0**, and **next** state is **1**, then you either had a **Toggle** (J=1, K=1) or you had a **set** (J=1, K=0)

Recall: If **present** state is **0**, and **next** state is **0**, then you either had a **NC** (J=0, K=0) or you had a **reset** (J=0, K=1)

Side Note: **Synthesis** is a design procedure that follows some **predefined** steps.

Sequence Detector – Using JK-FF

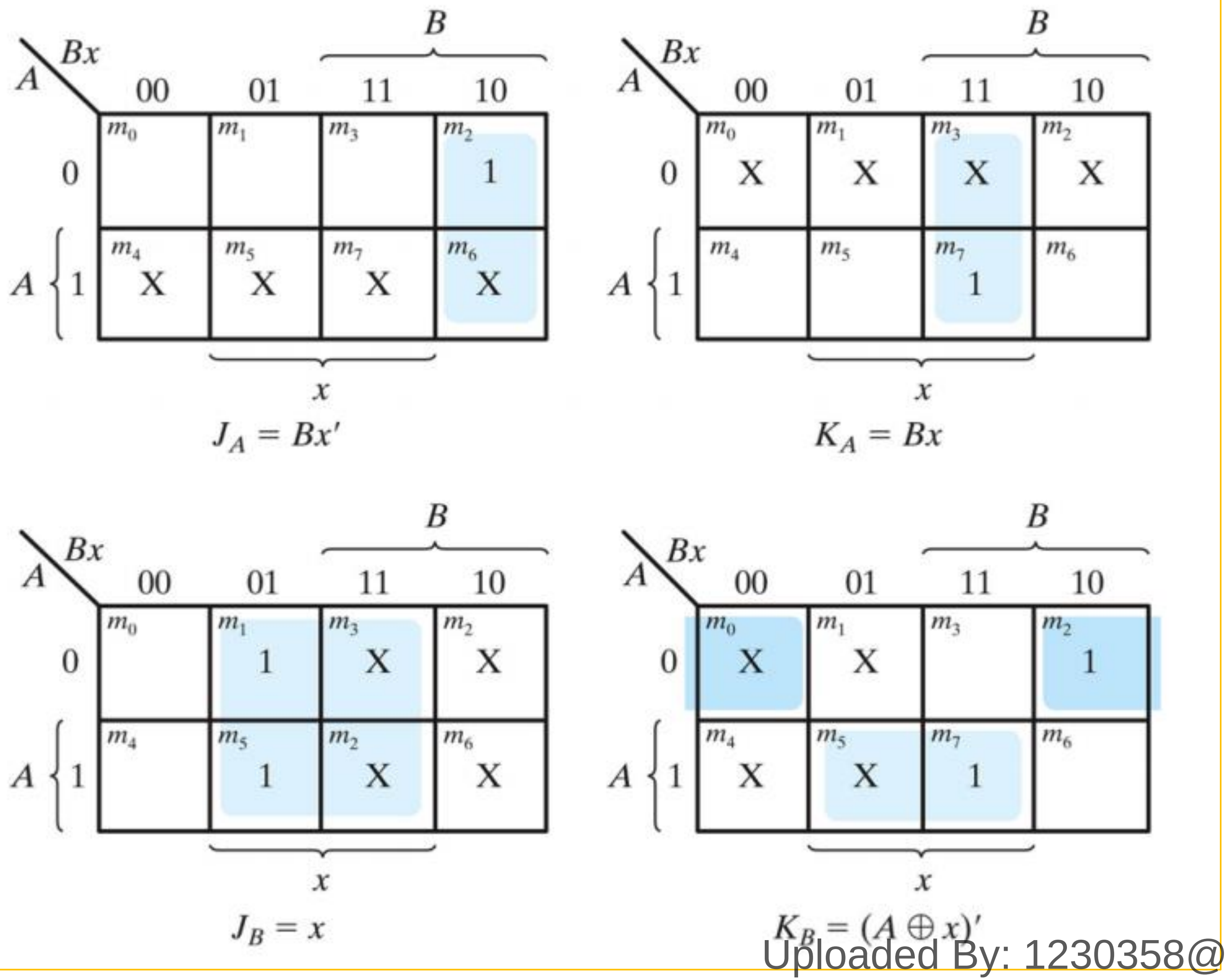


Present State		Input	Next State		Flip-Flop Inputs			
A	*B*	*x*	*A*	*B*	J_A	K_A	J_B	K_B
0	0	0	0	0	0	X	0	X
0	0	1	0	1	0	X	1	X
0	1	0	1	0	1	X	X	1
0	1	1	0	1	0	X	X	0
1	0	0	1	0	X	0	0	X
1	0	1	1	1	X	0	1	X
1	1	0	1	1	X	0	X	0
1	1	1	0	0	X	1	X	1

From **State transitions**, and using the **excitation table** of a JK-FF, derive the JK-FF inputs

Sequence Detector – Using JK-FF

Boolean Equations (JK-FF)



A \ Bx	00	01	11	10
0	m_0	m_1	m_3	m_2 1
1	m_4 X	m_5 X	m_7 X	m_6 X

$J_A = Bx'$

A \ Bx	00	01	11	10
0	m_0 X	m_1 X	m_3 X	m_2 X
1	m_4	m_5	m_7 1	m_6

$K_A = Bx$

A \ Bx	00	01	11	10
0	m_0	m_1 1	m_3 X	m_2 X
1	m_4	m_5 1	m_7 X	m_6 X

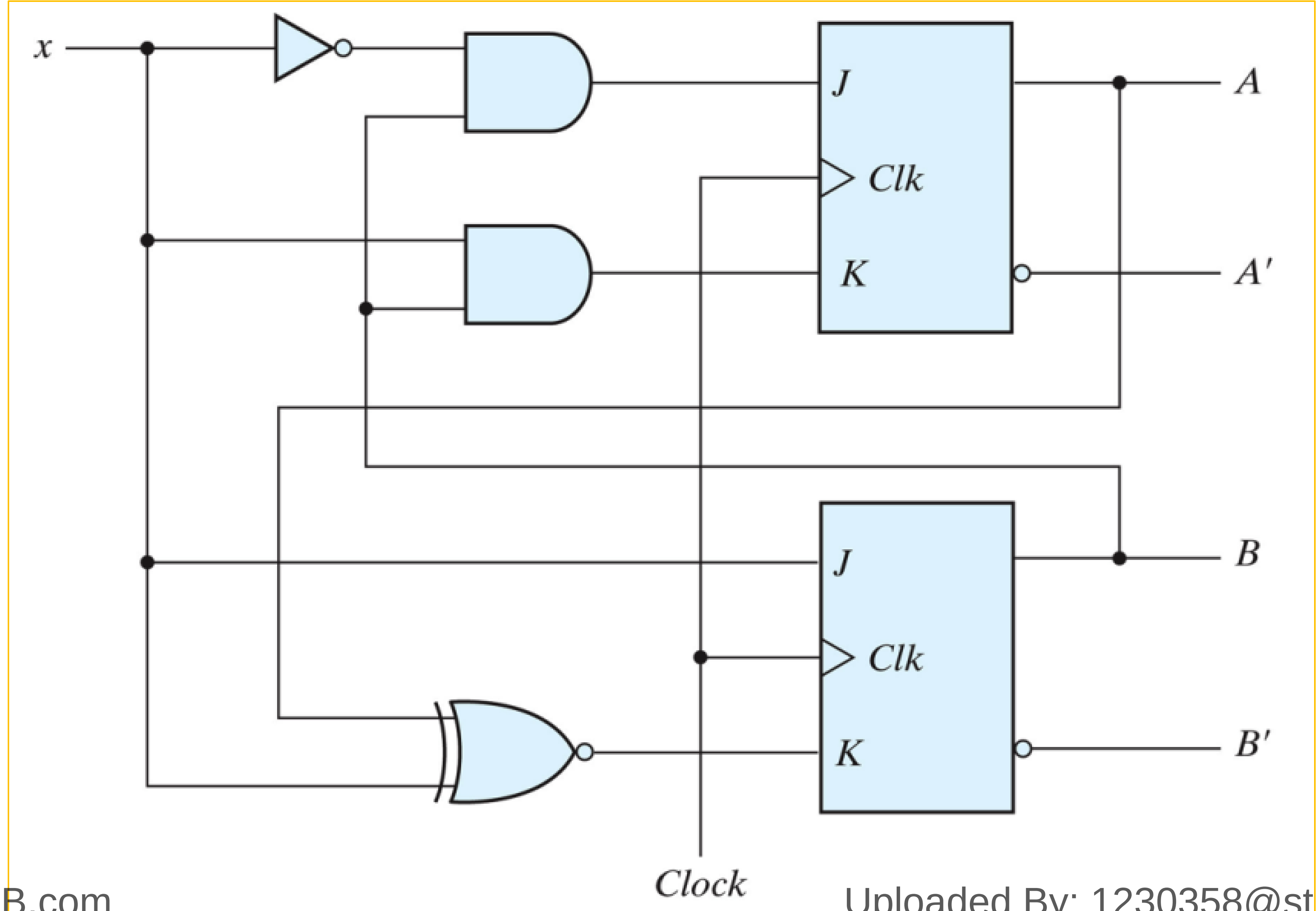
$J_B = x$

A \ Bx	00	01	11	10
0	m_0 X	m_1 X	m_3	m_2 1
1	m_4 X	m_5 X	m_7 1	m_6

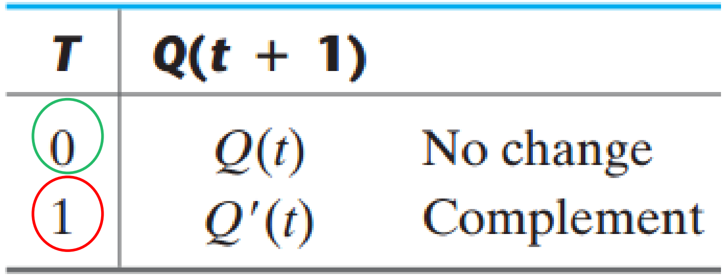
$K_B = (A \oplus x)'$

Sequence Detector – Using JK-FF

Circuit Diagram (JK-FF)

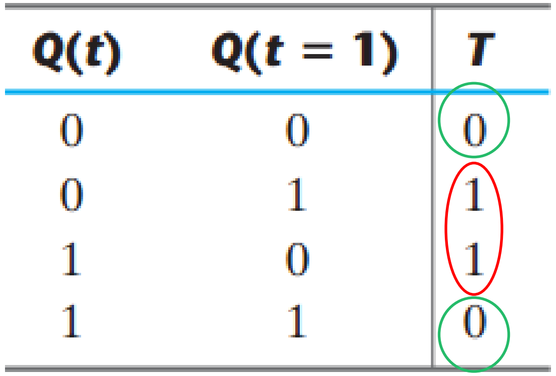


Design of Sequential Circuits: T-FF



T	$Q(t+1)$	
0	$Q(t)$	No change
1	$Q'(t)$	Complement

Characteristic Table



$Q(t)$	$Q(t = 1)$	T
0	0	0
0	1	1
1	0	1
1	1	0

Excitation Table

Recall: If **present** state and **next** state are **different** → **Toggle** (T=1)

Recall: If **present** state and **next** state are **same** → NC (T=0)

Design of Sequential Circuits: T-FF

Verbal Description

Example: Design a circuit that **counts up** from **0 to 7 then back to 0**

- ✓ When reaching **7**, the counter **goes back** to **0** then **repeat**
- ✓ There is **no input** to the circuit
- ✓ The counter is **incremented each cycle**
- ✓ The **output** of the circuit is the **present** state (count value)
- ✓ The circuit should be designed using **T-type** Flip-Flops

State Diagram

1) **Eight** states are needed to store the count values 0 to 7
2) **No input**, state **transition** happens at the **edge** of each cycle

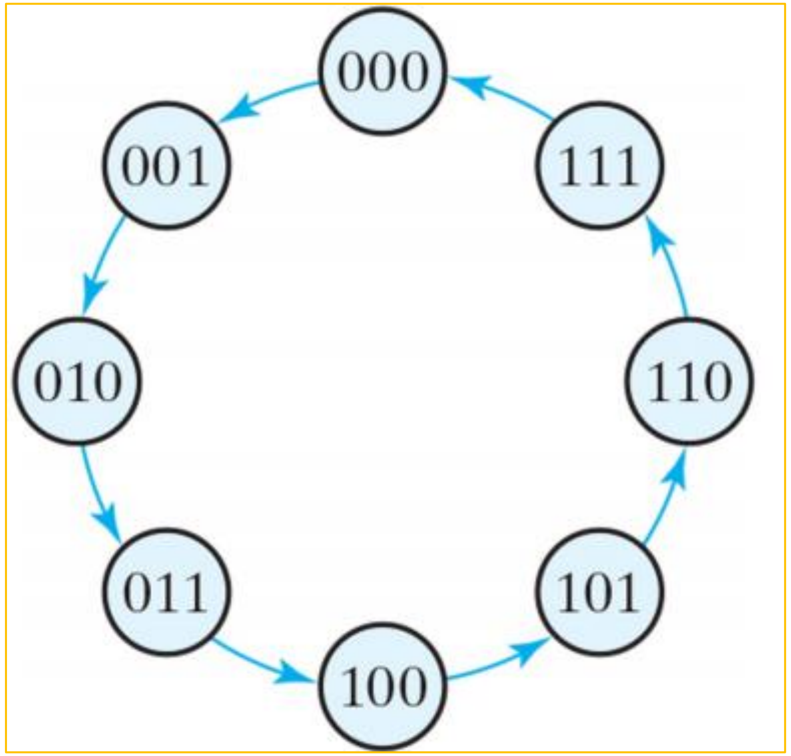
State Assignment

1) **Eight** states → 3 State Bits → **3** FFs
2) **State Assignment (000 – 111)**

State Reduction is NOT necessary

Design of Sequential Circuits: T-FF

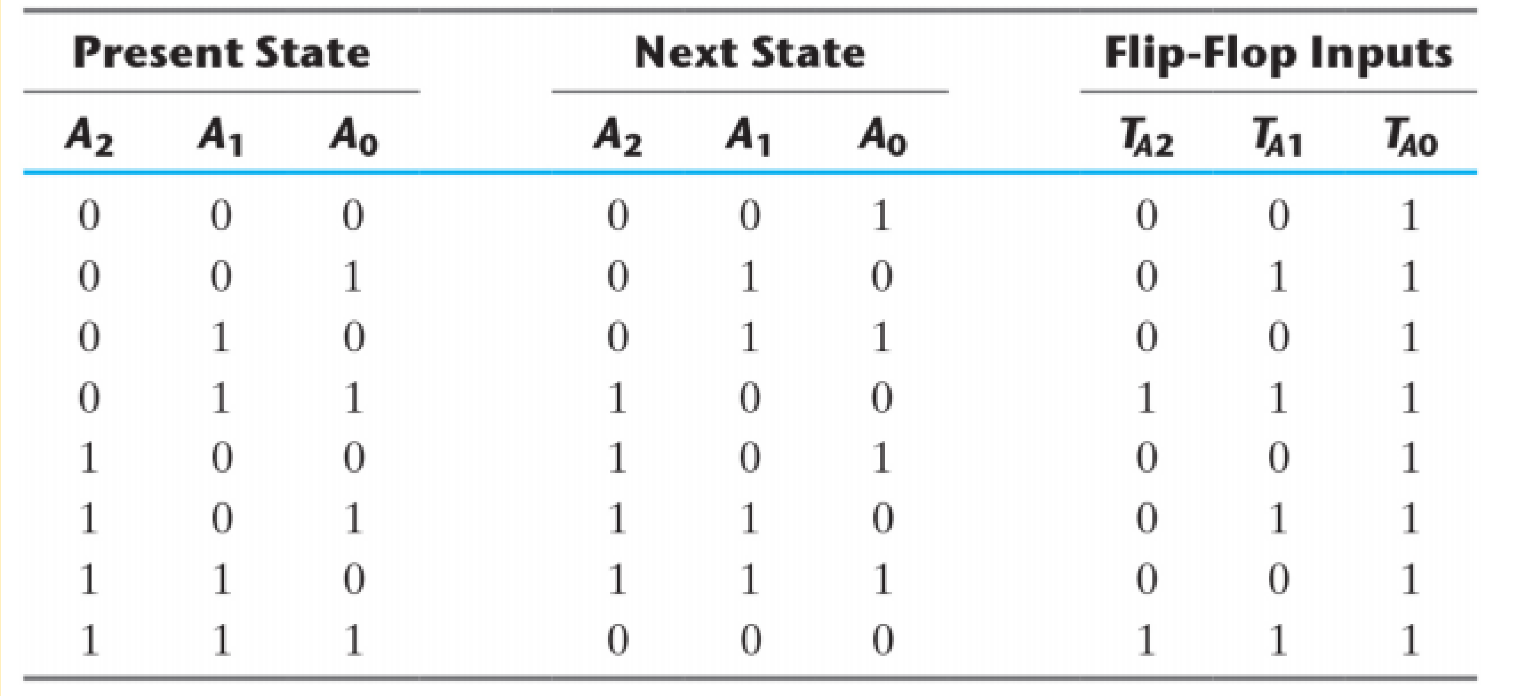
Example: Up Counter



State Diagram

3 State Bits → **3** Flip-Flops

Label: State Bits (FFs) → **A_2, A_1, A_0**



Present State			Next State			Flip-Flop Inputs		
A_2	A_1	A_0	A_2	A_1	A_0	T_{A2}	T_{A1}	T_{A0}
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	1
0	1	0	0	1	1	0	0	1
0	1	1	1	0	0	1	1	1
1	0	0	1	0	1	0	0	1
1	0	1	1	1	0	0	1	1
1	1	0	1	1	1	0	0	1
1	1	1	0	0	0	1	1	1

From **State transitions**, and using the **excitation table** of a T-FF, derive the T-FFs inputs

Design of Sequential Circuits: T-FF

Example: Up Counter

Boolean Equations (T-FF)

$A_2 \backslash A_1A_0$	00	01	11	10
0	m_0	m_1	m_3 1	m_2
1	m_4	m_5	m_7 1	m_6

A_1 over columns 11, 10; A_0 under columns 01, 11; A_2 beside row 1.

$T_{A2} = A_1A_0$

$A_2 \backslash A_1A_0$	00	01	11	10
0	m_0	m_1 1	m_3 1	m_2
1	m_4	m_5 1	m_7 1	m_6

A_1 over columns 11, 10; A_0 under columns 01, 11; A_2 beside row 1.

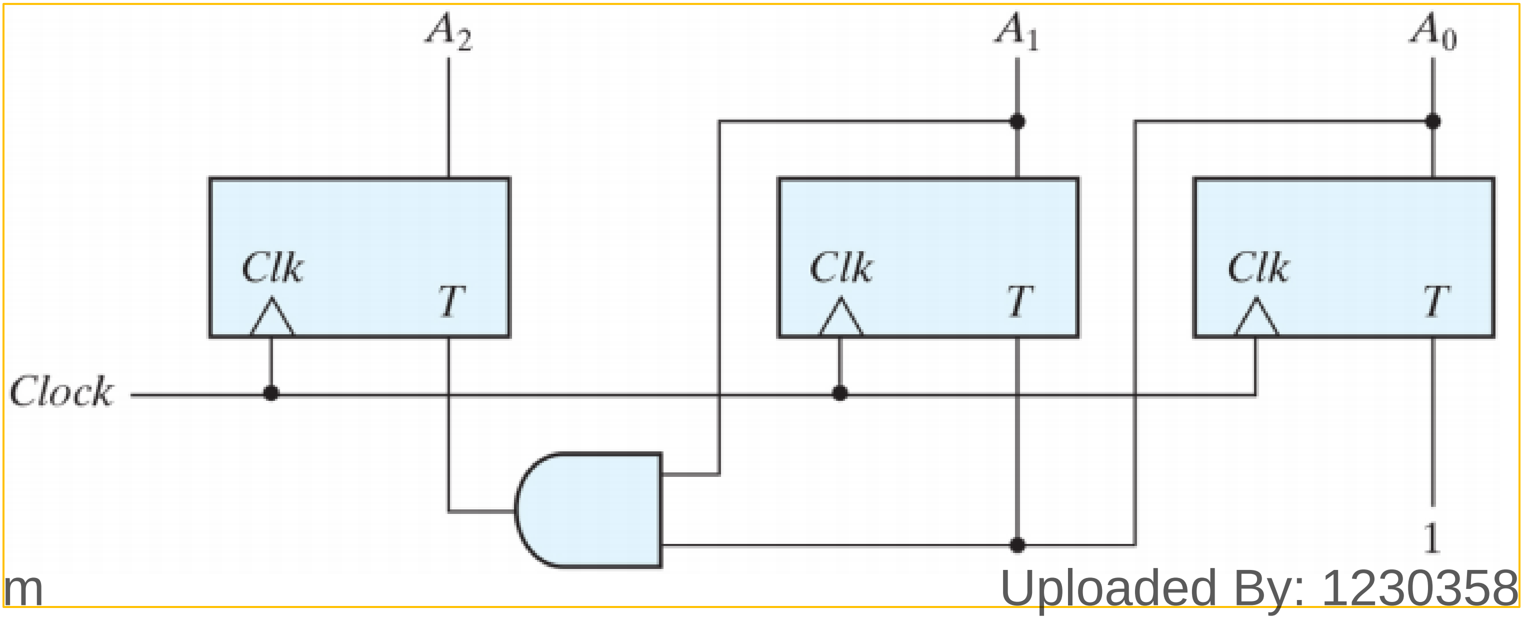
$T_{A1} = A_0$

$A_2 \backslash A_1A_0$	00	01	11	10
0	m_0 1	m_1 1	m_3 1	m_2 1
1	m_4 1	m_5 1	m_7 1	m_6 1

A_1 over columns 11, 10; x under columns 01, 11; A_2 beside row 1.

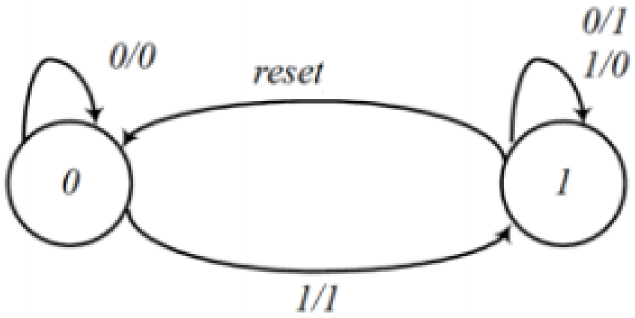
$T_{A0} = 1$

Circuit Diagram

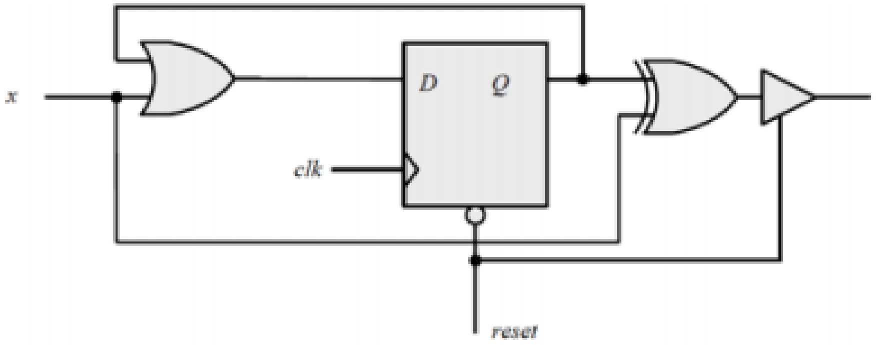


Practice Problem:

Design a one-input(x), one-output(y) serial 2’s complement with asynchronous reset



Present state	Input	Next state	Output
A	*x*	*A*	*y*
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	0


x
D
Q
clk
reset