Digital Systems
Section 2

Chapter (6)

Mohammed Khalil

- **Registers** and **counters** are practical examples of clocked **sequential** circuits

- A **register** consists of a group of **flip-flops** each is capable of **storing one bit** of information
  - The group of FFs normally share a **common clock**
  - **n-bit** register consists of a group of **n** FFs capable of **storing n bits** (Each FF store 1-bit)

- **Registers** may have **combinational logic/gates** that affect its **operation** (Loading/Reading Operations)
  - **Flip-flops hold** the data (bits)
  - **Gates determine** how the data (bits) is **transferred** into/from the register

- **Registers** are used in computers, calculators, and almost all **electronic equipment** these days. Any device must have registers to store binary data. A **memory unit** is an **array of registers**

- A **counter** is a **register** that goes through a **predetermined sequence** of binary **states**
  - The **gates** in the counter are **connected** in such a way as to produce the **prescribed sequence** of states.
- **Counters** are a **special type** of **register**

Registers

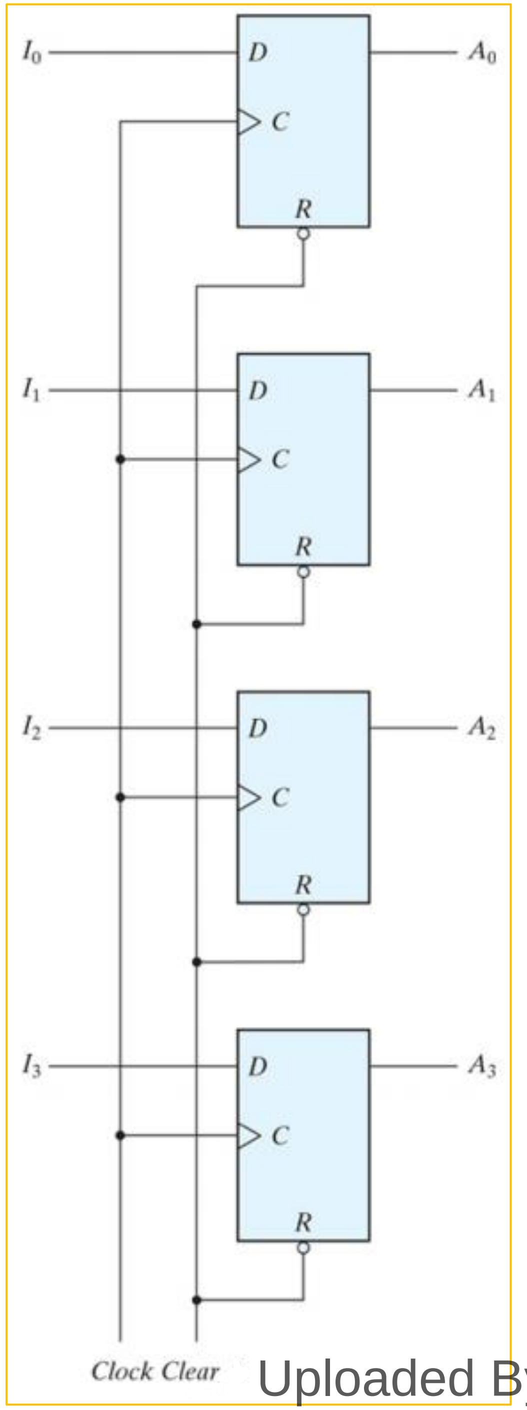
Various types of **registers** are available

The simplest register → **4-bit** Register

- Clear (R) is **0** → **clear/reset** the register to **all 0's asynchronously**
  - Note the **bubble** at R → Active **low** clear/reset
- Clear (R) is **1** → With each **+ve** Edge Clock, the **input** is **stored**
  - The value of (**$I_0I_1I_2I_3$**) immediately **before** the clock edge determines the value of (**$A_0A_1A_2A_3$**) **after** the clock edge

- To **prevent** the **previously stored** value from **changing**:
  1) The Clock has to be **disabled** **Not Practical**
  2) Or the **same** value needs to be **passed back** to the inputs

USE: Parallel Load Approach



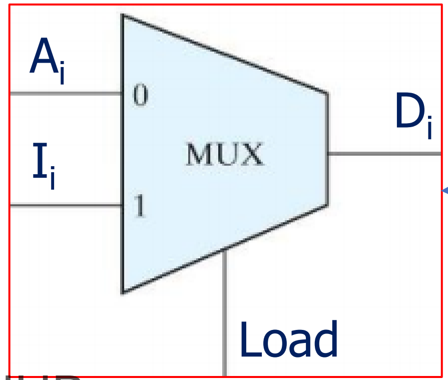
Registers with Parallel Load

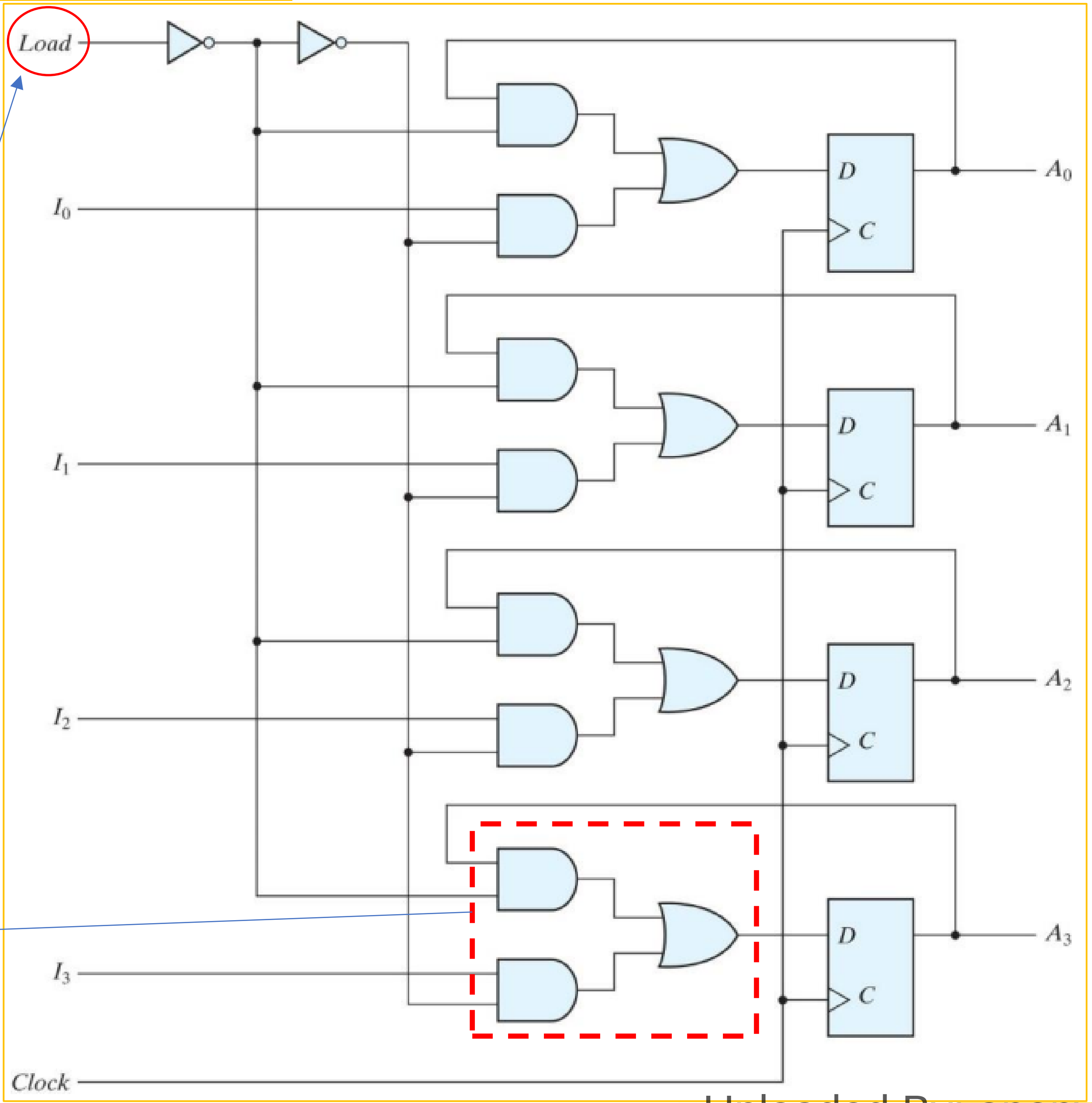
- **Registers** with **parallel load** are a fundamental building block in digital systems
- Synchronous digital systems have a **master clock generator** that supplies a continuous train of clock pulses
- The pulses are applied to **all flip-flops and registers** in the system
- The master clock acts like a drum that supplies a constant beat to all parts of the system
- A **separate control signal** must be used to decide **which** register **operation** will execute at each clock **pulse**
- **loading** or **updating** the register is The **transfer** of **new** information into a register
- The **loading** is in **parallel** if **all** the bits are **loaded simultaneously** at the **same** clock cycle
- **Recall**: To keep the **content** of the register **unchanged**:
  1) The **inputs** must be held **constant**
  2) Or The **clock** must be **disabled** from the circuit

Registers with Parallel Load

- It is advisable to **control** the **operation** of the register **with** the **D inputs**, rather than controlling the clock of the FF's
- Use a **combinational** circuit to either **keep previous** value, or **load** a **new** one:
  - a) Load=**0**➔ **keep previous** value
  - b) Load=**1**➔ **accept (load)** new value from **I**'s

This can be constructed using a **2x1 mux** with each **FF**

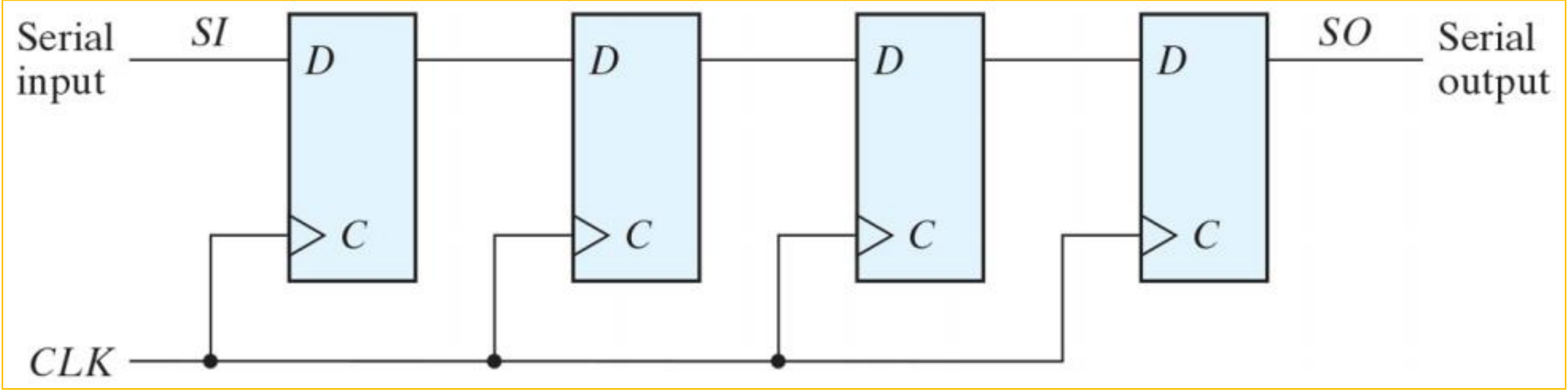




Shift Registers

- A **shift** register is a register that is **capable of shifting (moving)** the binary information held in each cell to its **neighboring** cell, in a **selected direction**
- A **shift** register consists of a **chain** of **flip-flops** in **cascade**, with the **output** of one flip-flop **connected** to the **input** of the next flip-flop
- **All** flip-flops receive **common** clock pulses, which **activate** the **shift** of data from one stage to the next

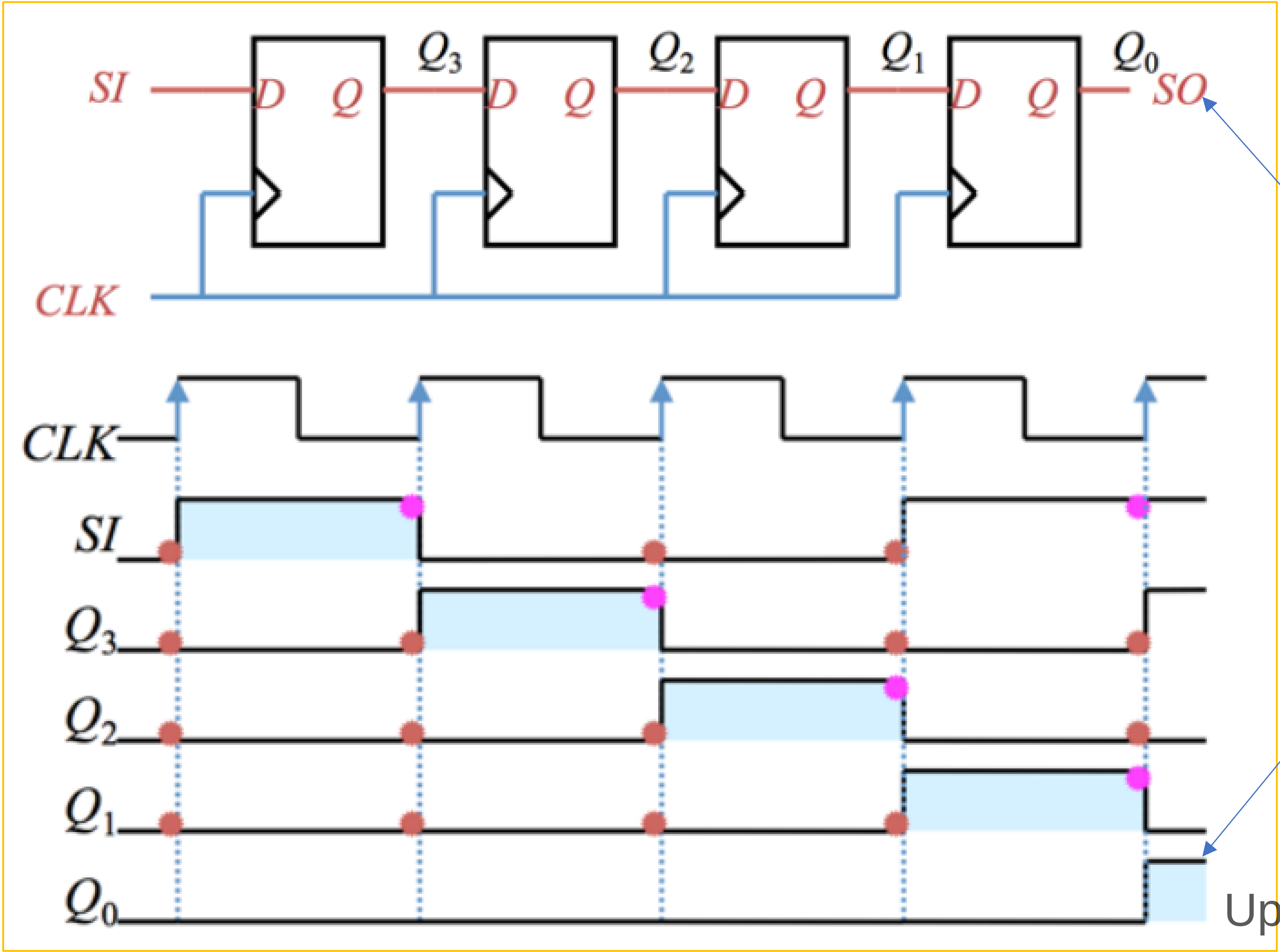
With **each** clock cycle, the data will **move** from one **FF** to its **adjacent** one.
The **SI** will get to the **far-left** FF and the **SO** will get out from the **right-most** FF



Unidirectional (left-to-right) 4-bit Shift register

Shift Registers

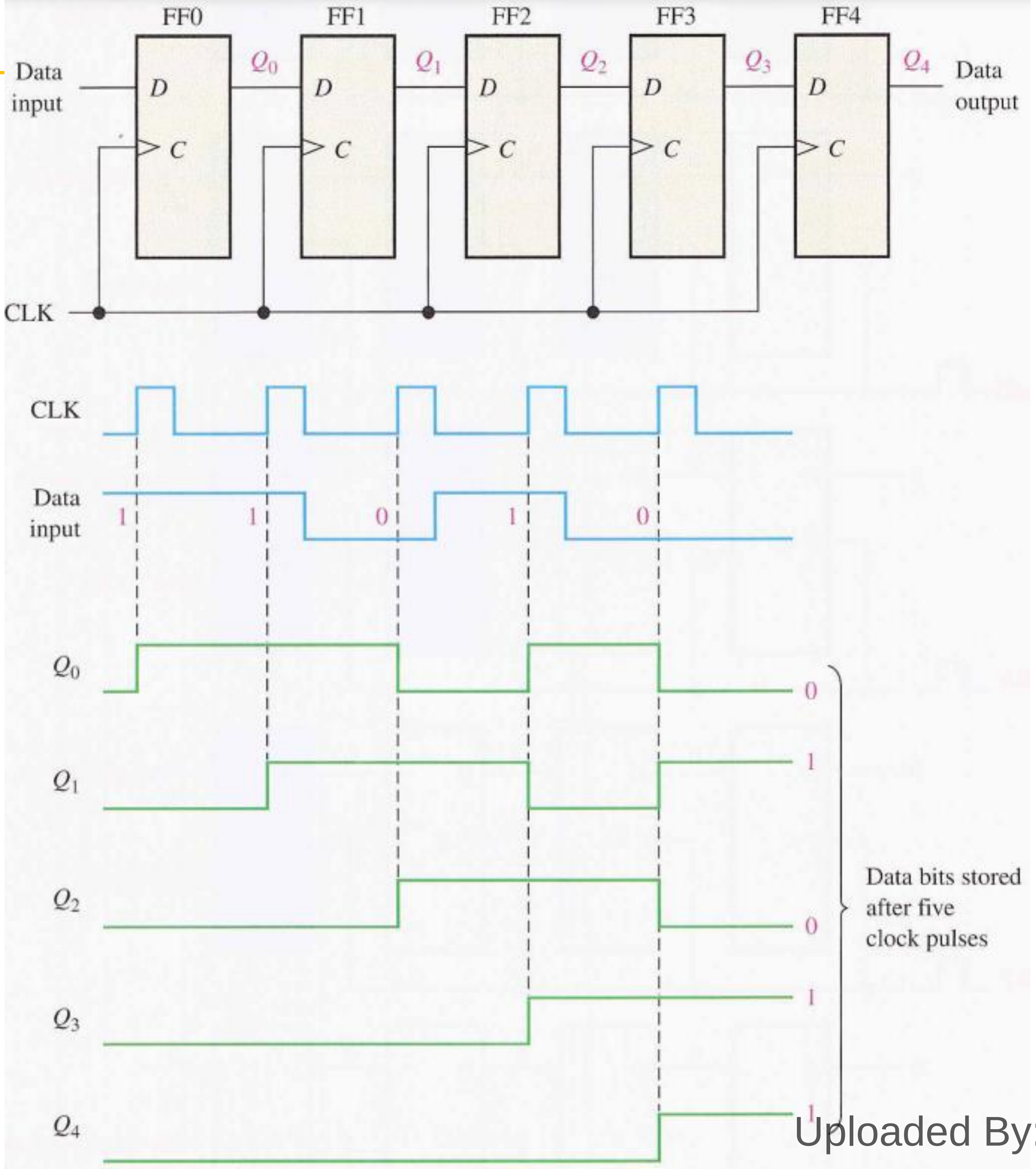
- **This** shift register is **unidirectional** (Does not support a left shift)
- The **serial input** determines what goes into the **leftmost** flip-flop during the shift.
- The **serial output** is taken from the output of the **rightmost** flip-flop
- A **clear/reset** signal is required in practical designs (to reset the whole register to zero)



SO

Shift Registers

- **5-bit** shift-right register
- Initially the register is cleared (all 0s)

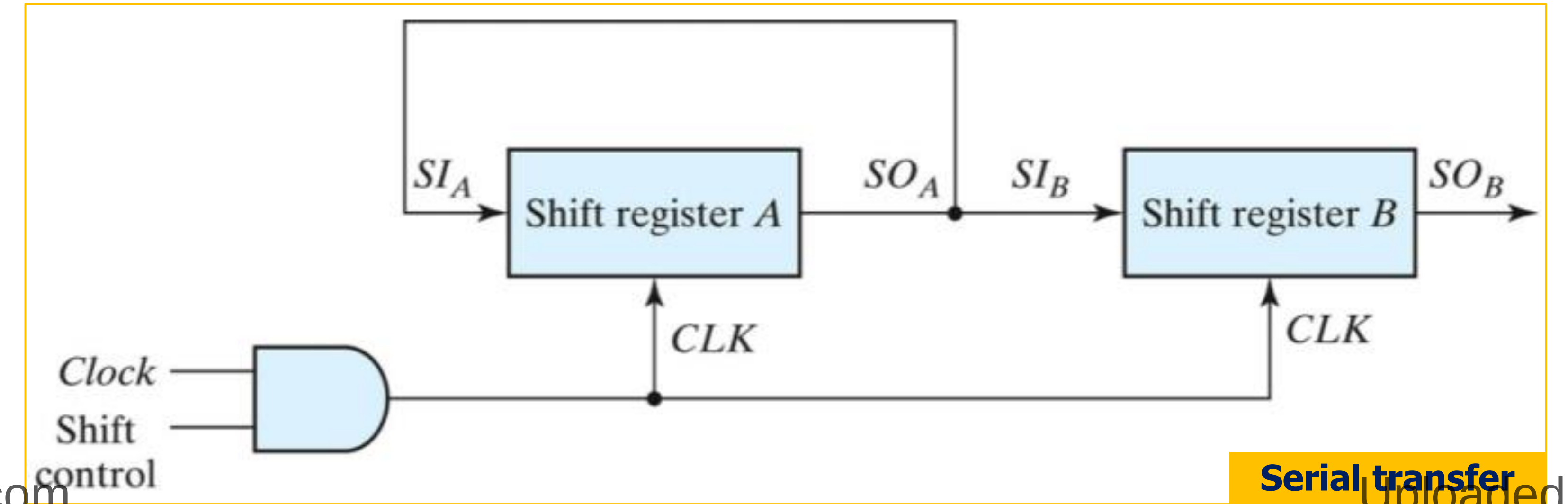


Serial Transfer

- Two main **Operation mode** for any digital system:
  1) **Serial mode:** data is transferred and processed **one bit at a time**
     - Take **more time** to process a collection of data (**Slower**)
     - The hardware needed is **for a single bit** (**Simpler**)
     - Examples: USB cables, RS232

  2) **Parallel mode:** data is transferred and processed **in parallel** (All at once)
     - Take only **1** clock cycle to process a collection of data (**Faster**)
     - The hardware needed is **for a n-bits** (**More Complex**)
     - Examples: Internal Memory Bus

Each Register has **n-bits**. The contents of A are **copied** into B, so that
the contents of A remain **unchanged** (i.e., the contents of A are **restored** to their **original** value)

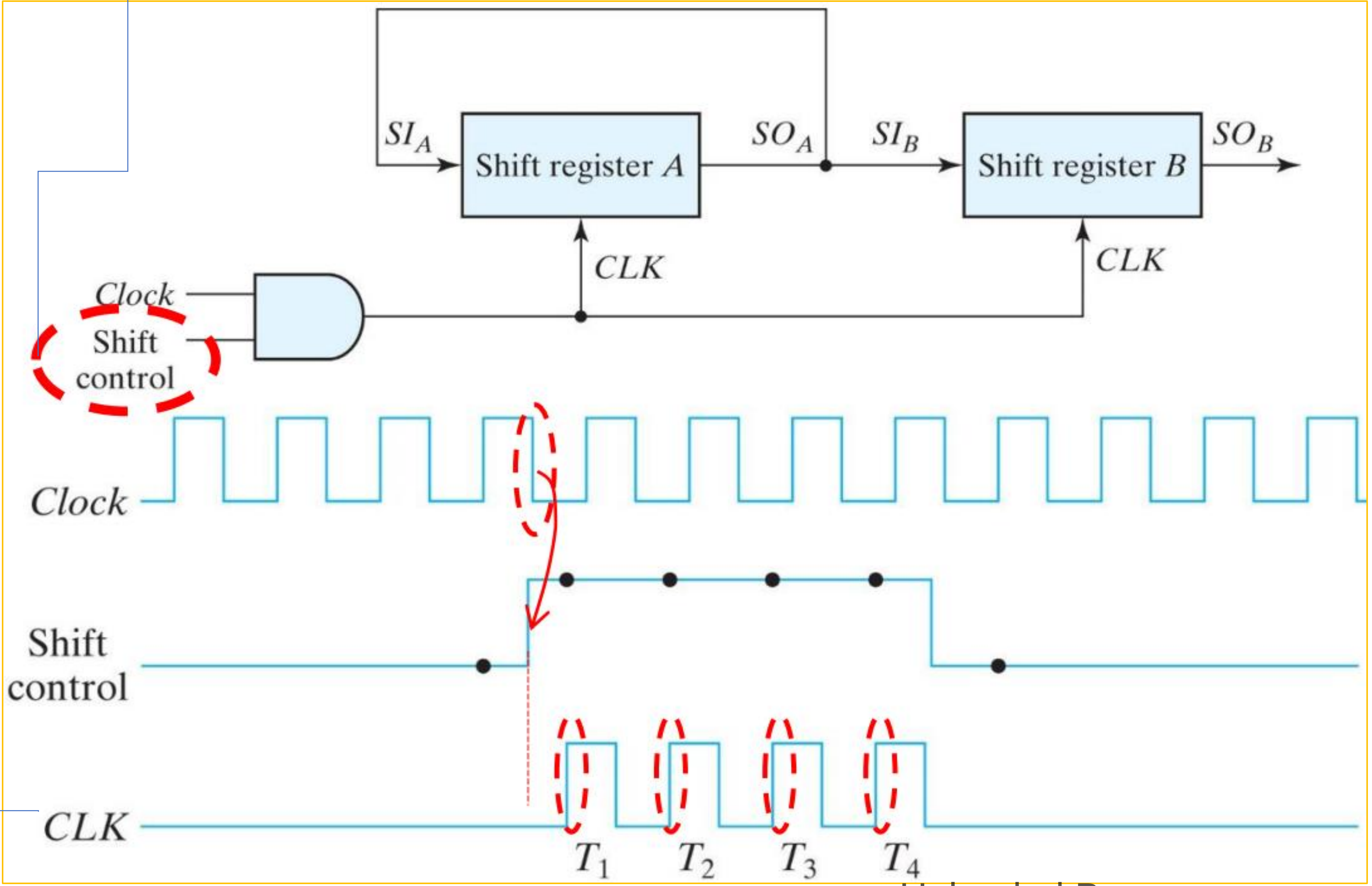


Serial transfer

Serial Transfer

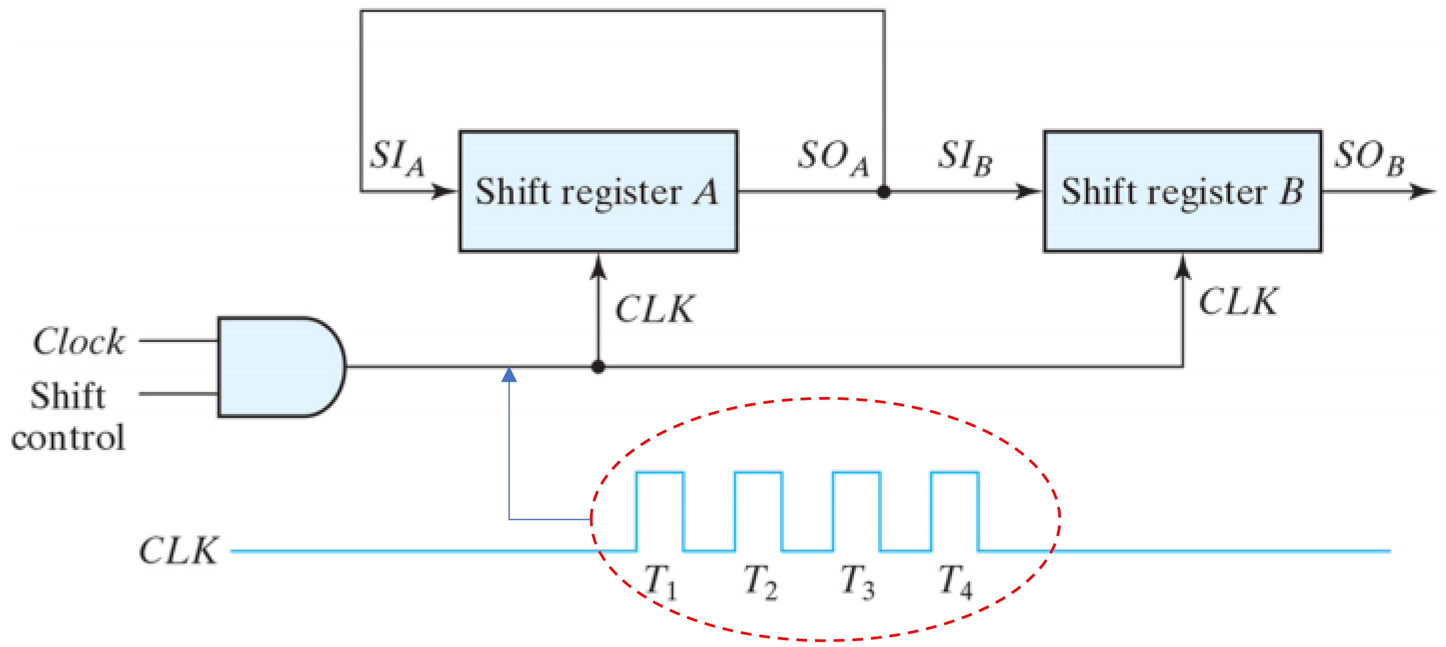
Control the shift so that it occurs only with **certain** pulses, but not with others.

- The **4** pulses perform identical operations, **shifting** the bits of A into B, **one at a time**



Serial Transfer

- The **4** pulses perform identical operations, **shifting** the bits of A into B, **one at a time**
- After the **fourth** shift, the **shift control** goes to **0**, and registers A and B both have the value **1011**.

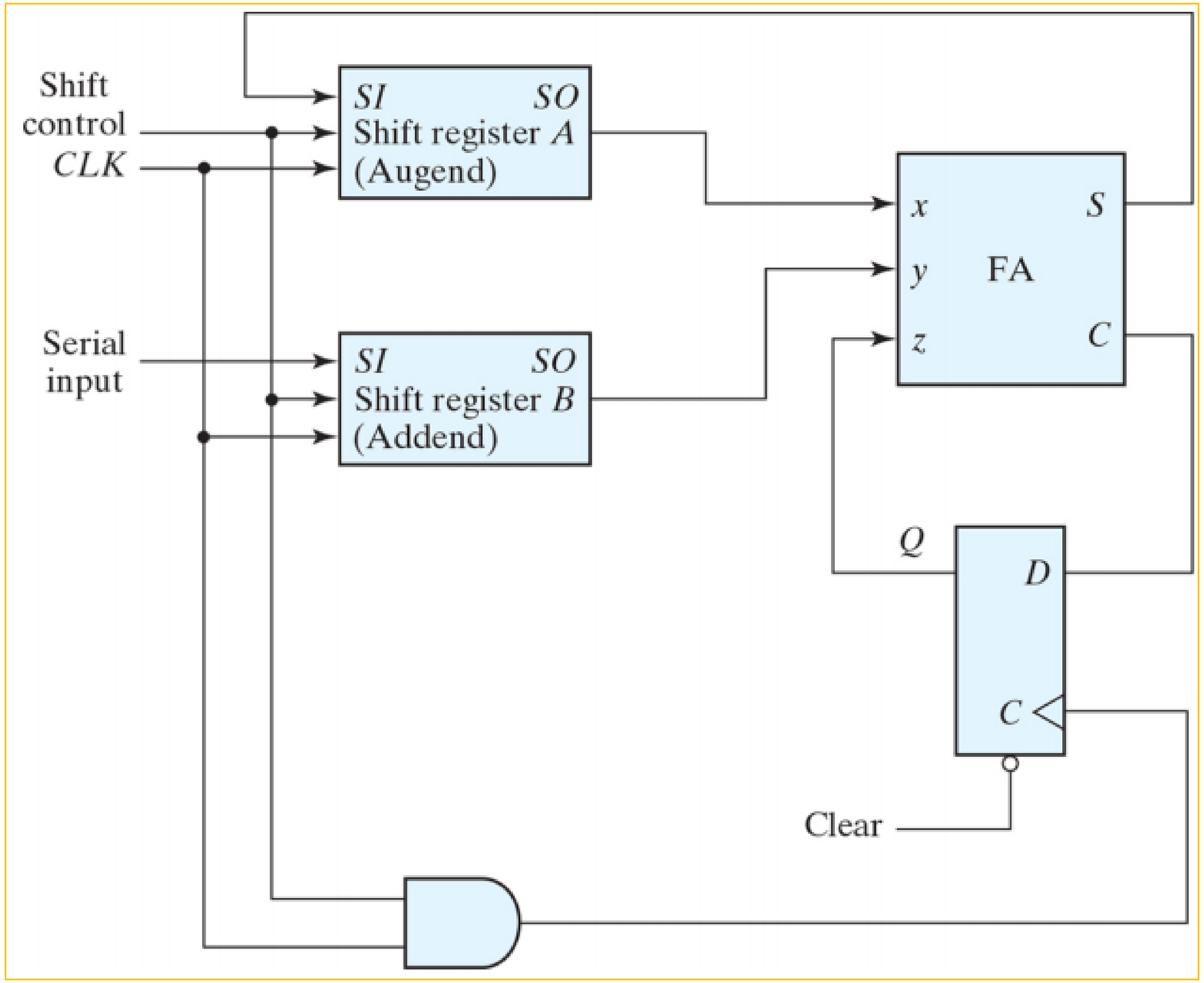


Timing Pulse	Shift Register *A*	Shift Register *B*
Initial value	1 0 1 1	0 0 1 0
After T_1	1 1 0 1	1 0 0 1
After T_2	1 1 1 0	1 1 0 0
After T_3	0 1 1 1	0 1 1 0
After T_4	1 0 1 1	1 0 1 1

Serial Adder: **sequential** circuit that performs **serial addition**

- The **inputs** comes from **two shift registers (A,B)**
- A **single full adder** is used to add **one pair of bits** at a time along with the **carry**
- The **result** is **stored back** into one of the shift registers **(A)**
- The **carry out** is **stored** in a **D-FF** and is then used as the **carry input** for the next pair of significant bits

Only **1-FA** is used compared to **n** FAs in the **n**-bit binary adder

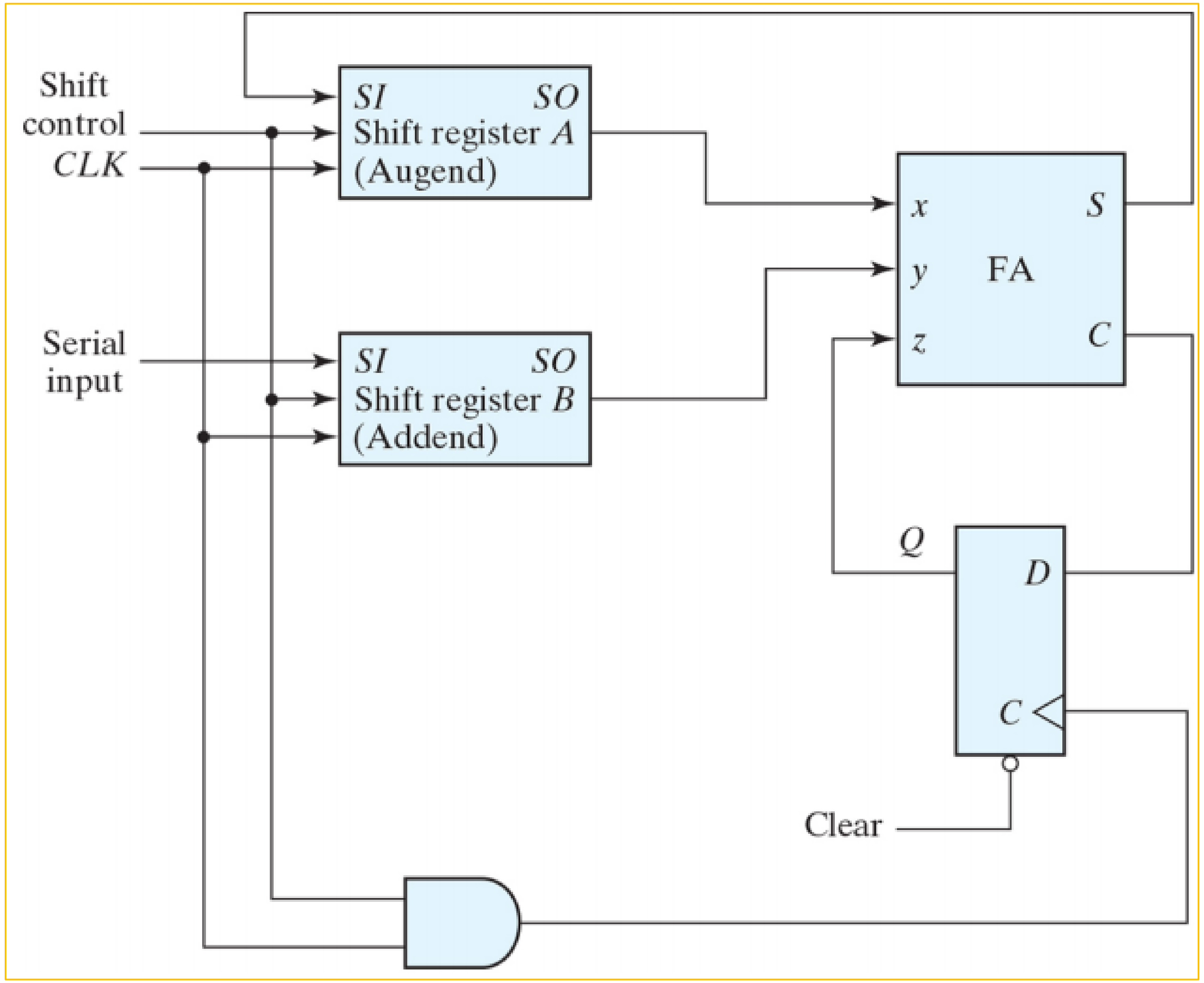


Serial Adder

Shift control signal is used to control when the registers will be shifted → control the next 2-bits addition

- Registers A & B hold the **two** binary numbers to be added
- D-FF is reset to **0** **(**C_{in} is 0**)**
- The **SO** of both registers are **connected** to the **inputs** of the FA
- The **output** of the FF is connected to the **carry-in** input of the FA
- **S** is connected to **SI** of register A
- **C** is connected to the D of the FF

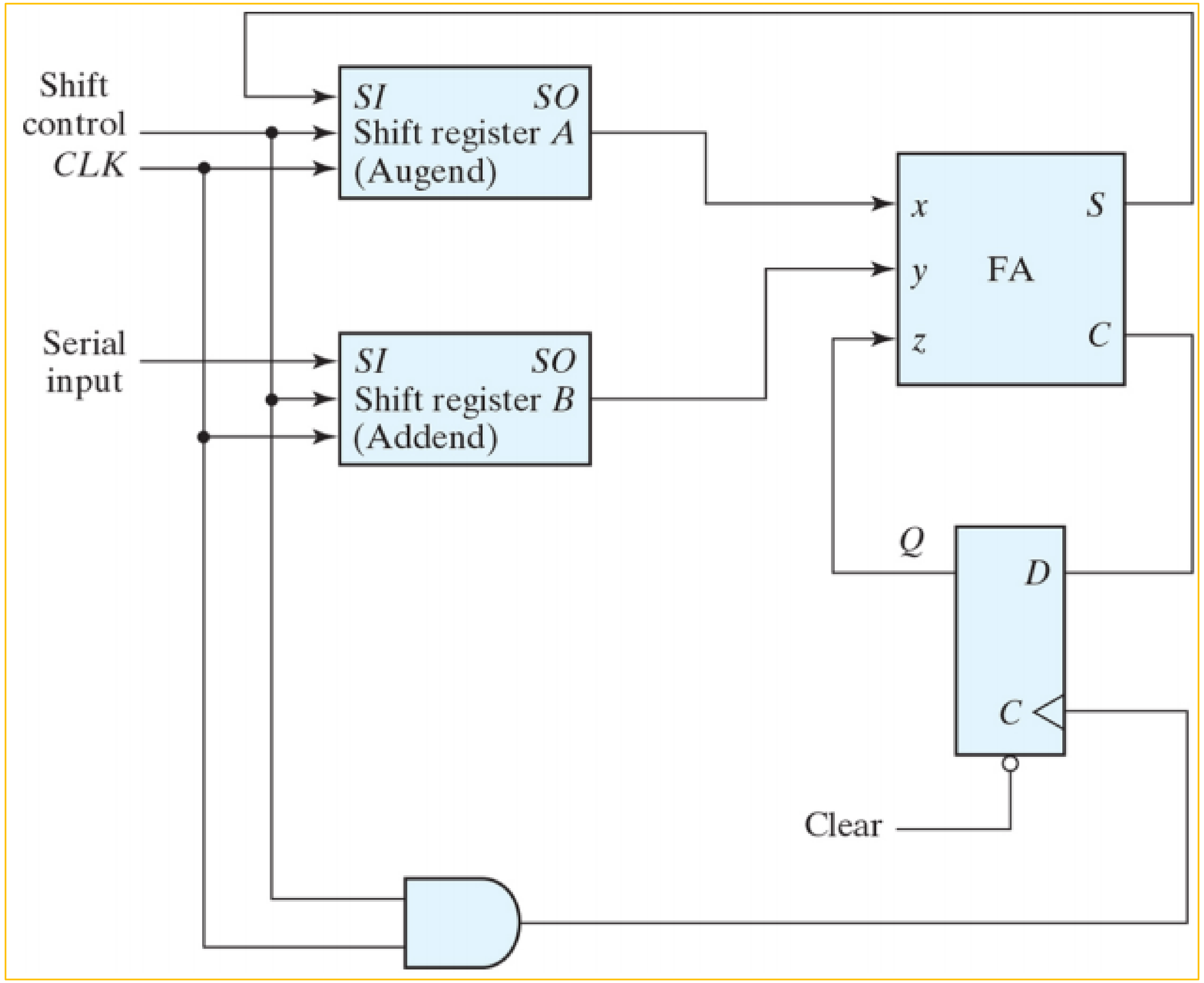
After applying **n** clock pulses the sum **result** is stored in register (**A**)



Example: 4-bit serial adder

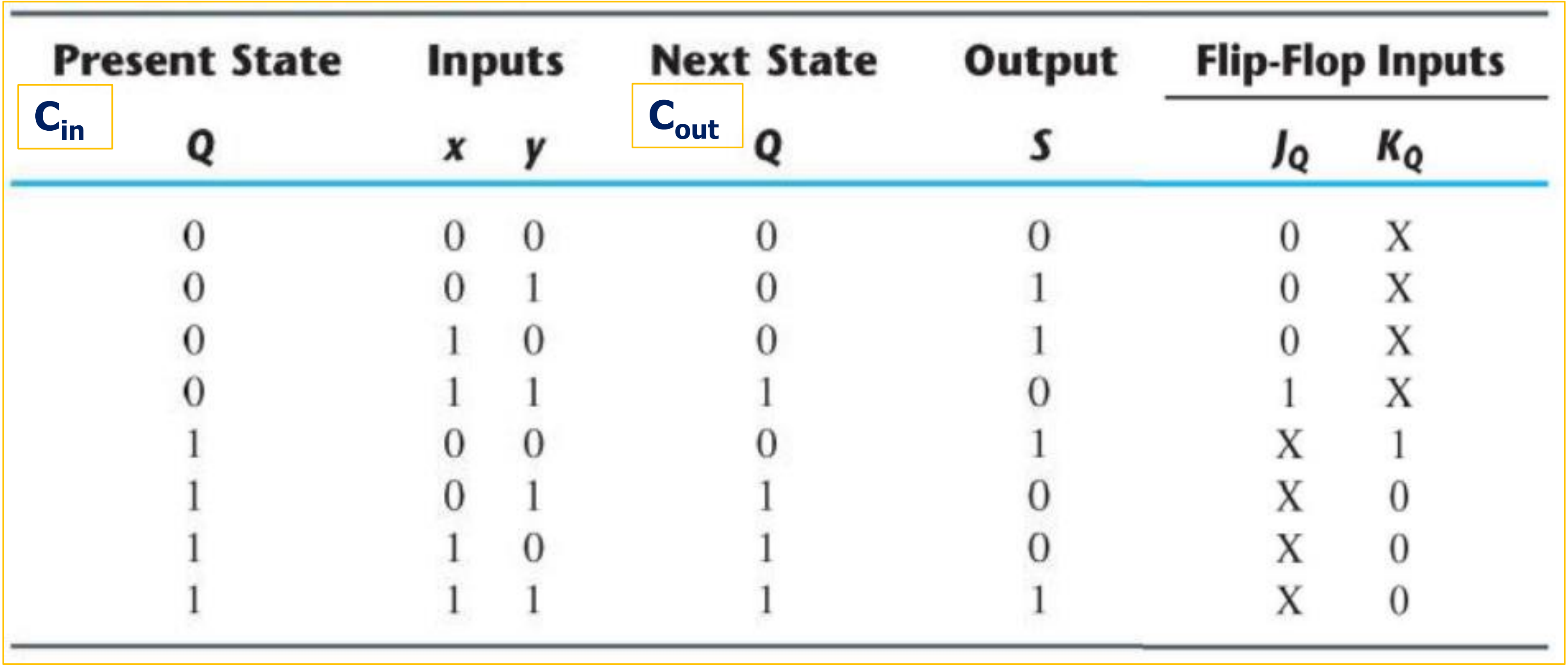
CLK	A	B	S	C	Q
	1001	0111	0	1	0
1	0100	0011	0	1	1
2	0010	0001	0	1	1
	0001	0000	0	1	1
	0000	0000			1

1001+0111 = 10000!


Shift control
CLK
Serial input
SI SO Shift register A (Augend)
SI SO Shift register B (Addend)
x
y
z
FA
S
C
Q
D
C
Clear

Serial Addition: Systematic Design

- Follow a sequential circuit **design approach** and use **JK-FF** to store the **carry**
- Let the **LSB** bits from Reg.A and Reg.B → **x and y**
- **x and y** are **available** for every CLK since the Shift-Reg will provide a **new x and y** for each CLK
- The circuit will have: Two **inputs**, x, y, **Output** S for the sum , Carry that is stored in **Q**

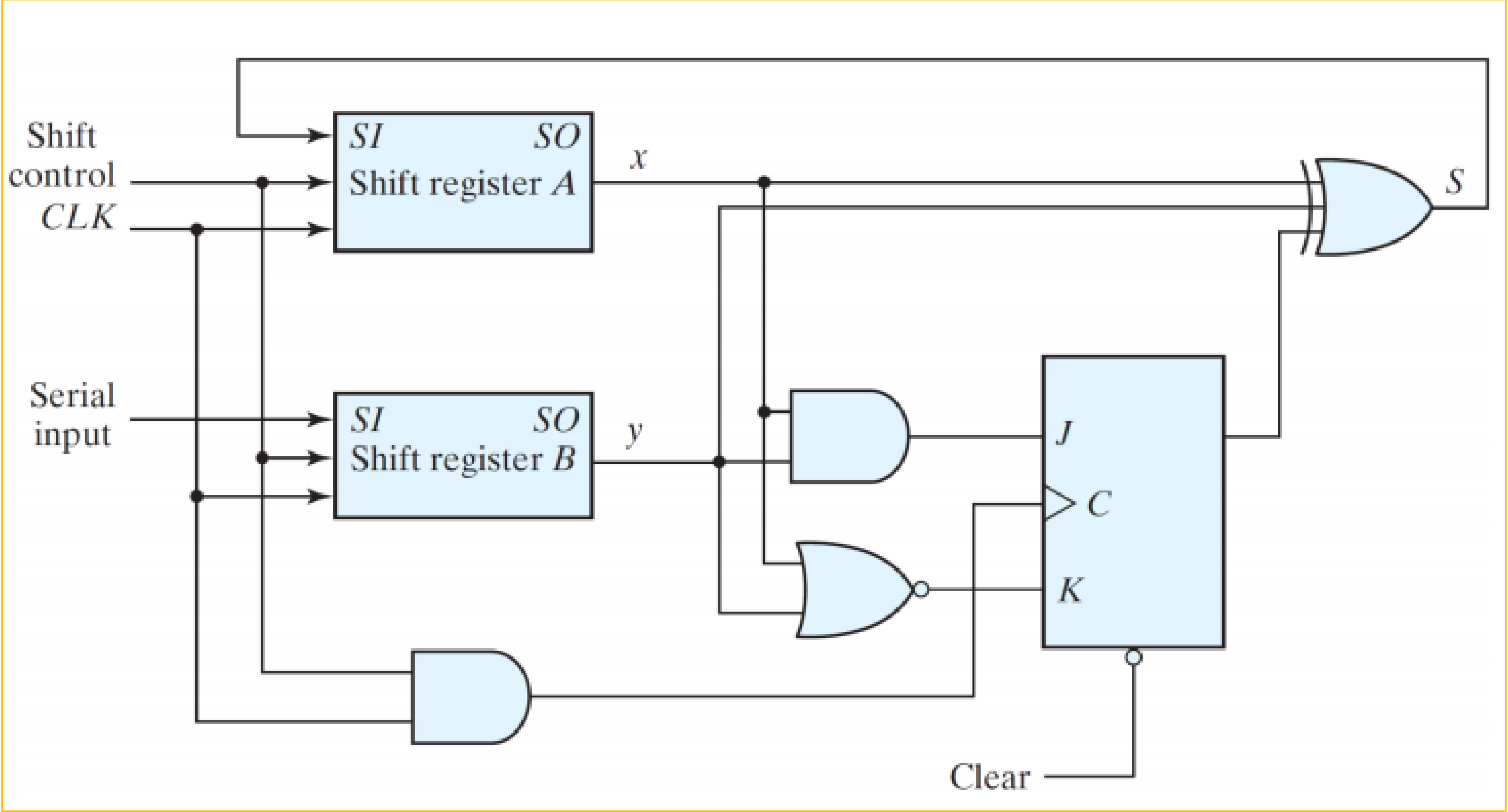


Present State	Inputs		Next State	Output	Flip-Flop Inputs	
C_{in} Q	x	y	C_{out} Q	S	J_Q	K_Q
0	0	0	0	0	0	X
0	0	1	0	1	0	X
0	1	0	0	1	0	X
0	1	1	1	0	1	X
1	0	0	0	1	X	1
1	0	1	1	0	X	0
1	1	0	1	0	X	0
1	1	1	1	1	X	0

$$J_Q = xy,\ K_Q = x'y' = (x+y)',\ S = x \oplus y \oplus Q$$

Serial Addition: Systematic Design

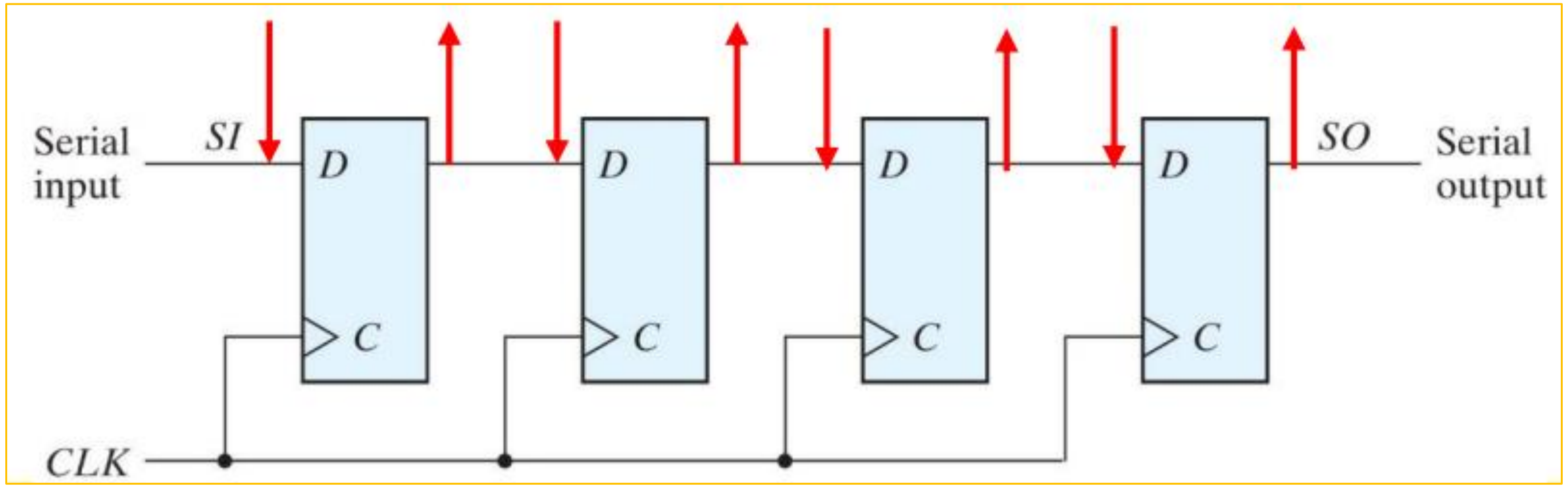
$J_Q = xy,\ K_Q = x'y' = (x+y)',\ S = x \oplus y \oplus Q$



Universal Shift Register

- A register that can shift in **one** direction (left **or** right) is called a **unidirectional** shift register
- A register that can shift in **both** directions is called a **bidirectional** shift register
- A register that can shift in **both** directions and has a **parallel load** capability is called a **Universal Shift Register**

- **In Universal Shift Register:**
  - The data **entered serially** by shifting can be **taken out in parallel** from the outputs of the flip-flops
  - The data **entered in parallel** can be **taken out in serial** fashion by shifting the data stored in the register



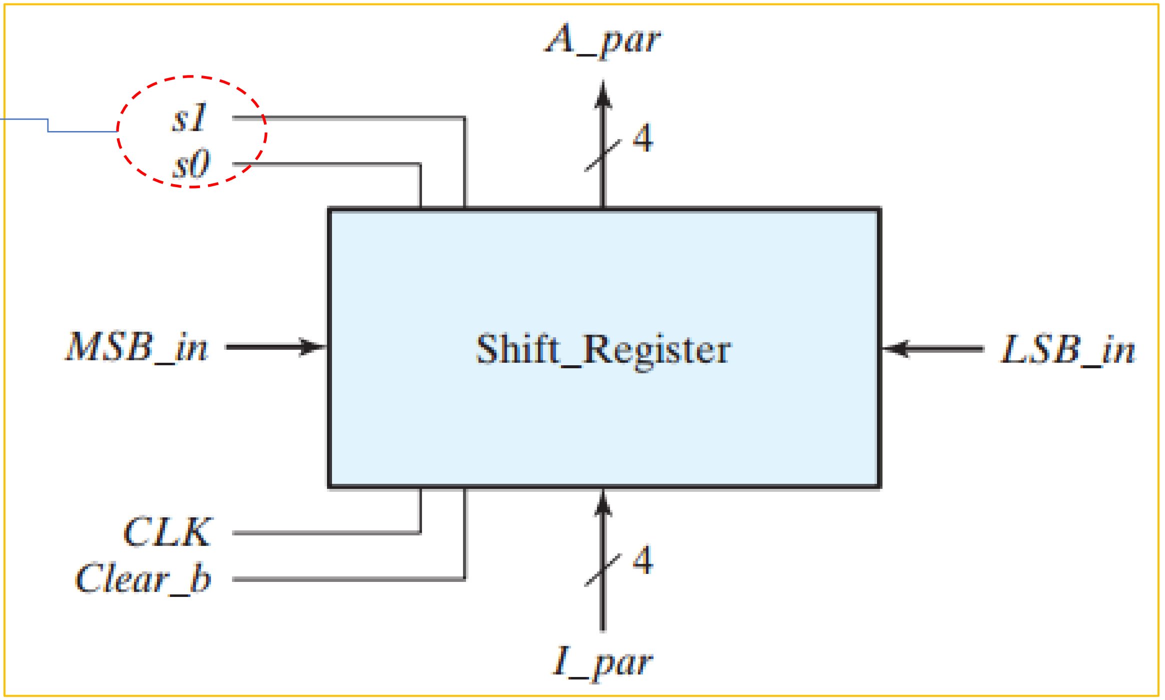
Universal Shift Register

The Universal Shift Register has the following capabilities:

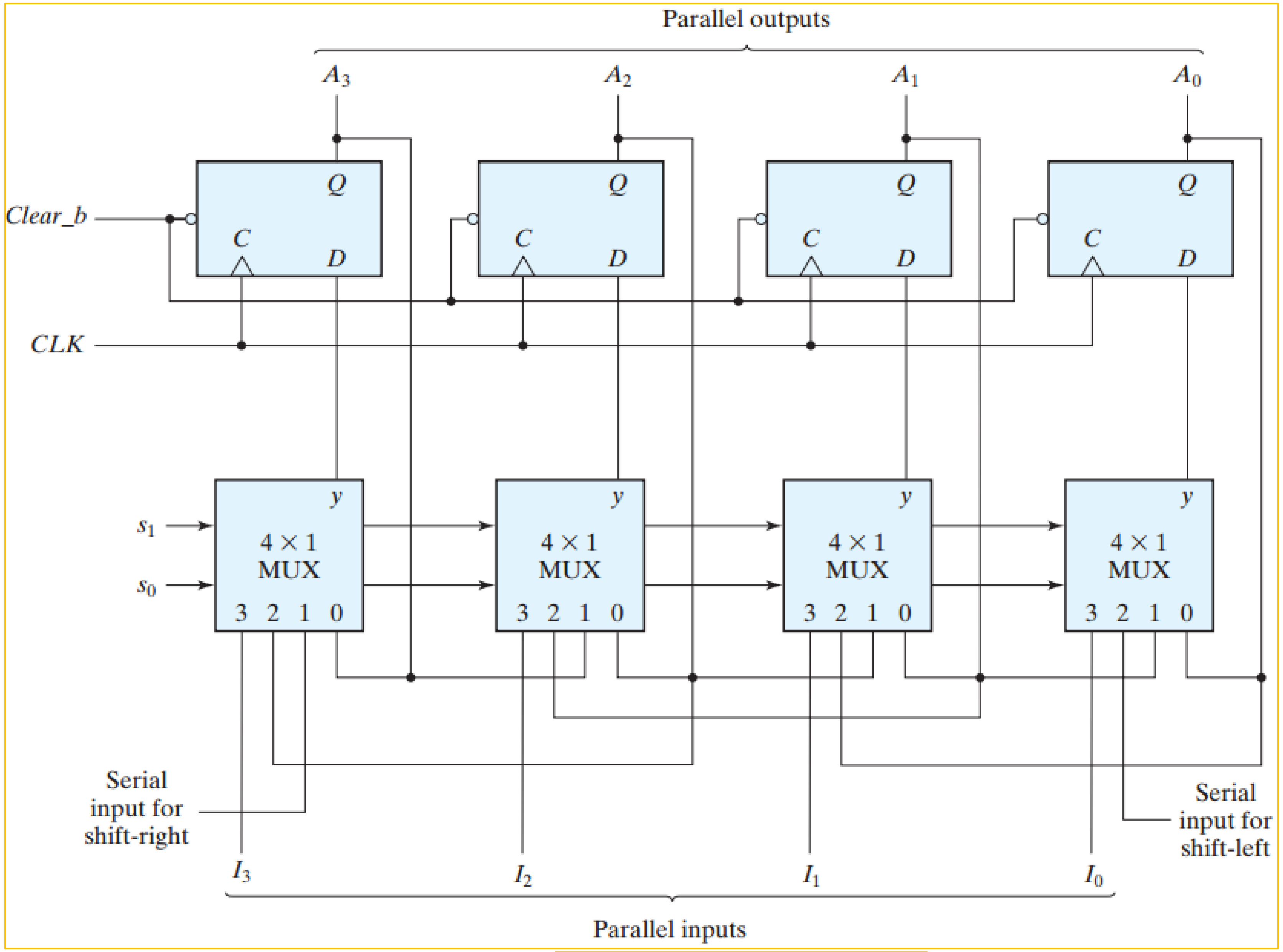
- A **clear** control to clear the register to 0
- A **clock** input to synchronize the operations
- A **shift-right control** to enable the shift-right operation
- A **shift-left control** to enable the shift-left operation
- A **parallel-load** control to enable a parallel transfer
- A **control** that leaves the data in the register **unchanged** in response to the clock
- The **serial input and output** lines
- **n parallel input lines** associated with the parallel transfer
- **n parallel output lines** associated with the parallel transfer

4 Control Signals

2 bits → 4 Combinations → (4 control signals)

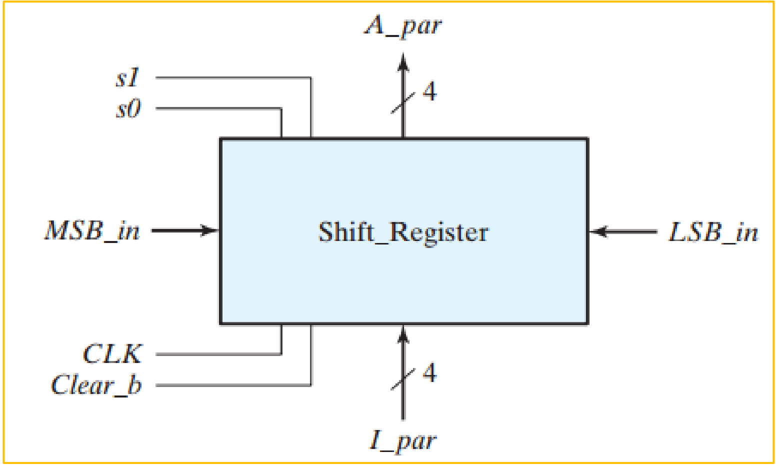


Universal Shift Register



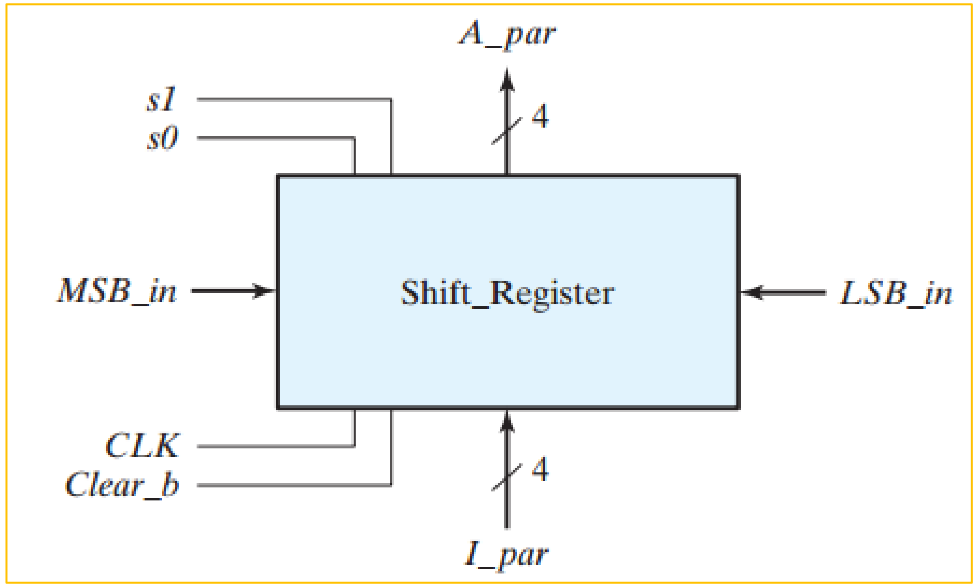
Very Important

Mode Control		
s_1	s_0	**Register Operation**
0	0	No change
0	1	Shift right
1	0	Shift left
1	1	Parallel load



Logic/circuit Diagram

Universal Shift Register



Block Diagram

Mode Control		
s_1	s_0	Register Operation
0	0	No change
0	1	Shift right
1	0	Shift left
1	1	Parallel load

Function Table

- Universal Shift registers are often used to interface digital systems situated **remotely** from each other
- It will be **expensive** to use **n lines** to **transmit** **n bits** in **parallel** for a long distance.
- It is more economical to use a **single** line and **transmit** the information **serially**
- The **transmitter** accepts the **n-bit** **data in parallel** into a **universal shift register** and then **transmits** the data **serially**.
- The **receiver** accepts the data **serially** into a **universal shift register** and then it can be **taken from the outputs** of the register **in parallel**
- The **transmitter** performs a **parallel-to-serial conversion** of data
- The **receiver** does a **serial-to-parallel** conversion

Counters

- In real life, A counter is a device which stores (and sometimes displays) the number of times a particular event or process has occurred.

- A **Counter** is a **register** that goes through a **prescribed sequence of states** upon the application of input pulses
  1) Input pulses may be **clock** pulses (CLK)
  2) Input may originate from **external** sources (Example: other signals like A, B, etc.)

- There are two types of counters:
  1) **Synchronous** Counters
     - Input sequence occurs at a **fixed** interval of time
     - **All flip-flops are triggered by the same clock pulses.**
  2) **Asynchronous** Counters (**Ripple** Counters)
     - Input sequence occurs at **random** (input occurrence time is not pre-defined)
     - **Each flip flop is triggered by the transition of other flip-flops.**

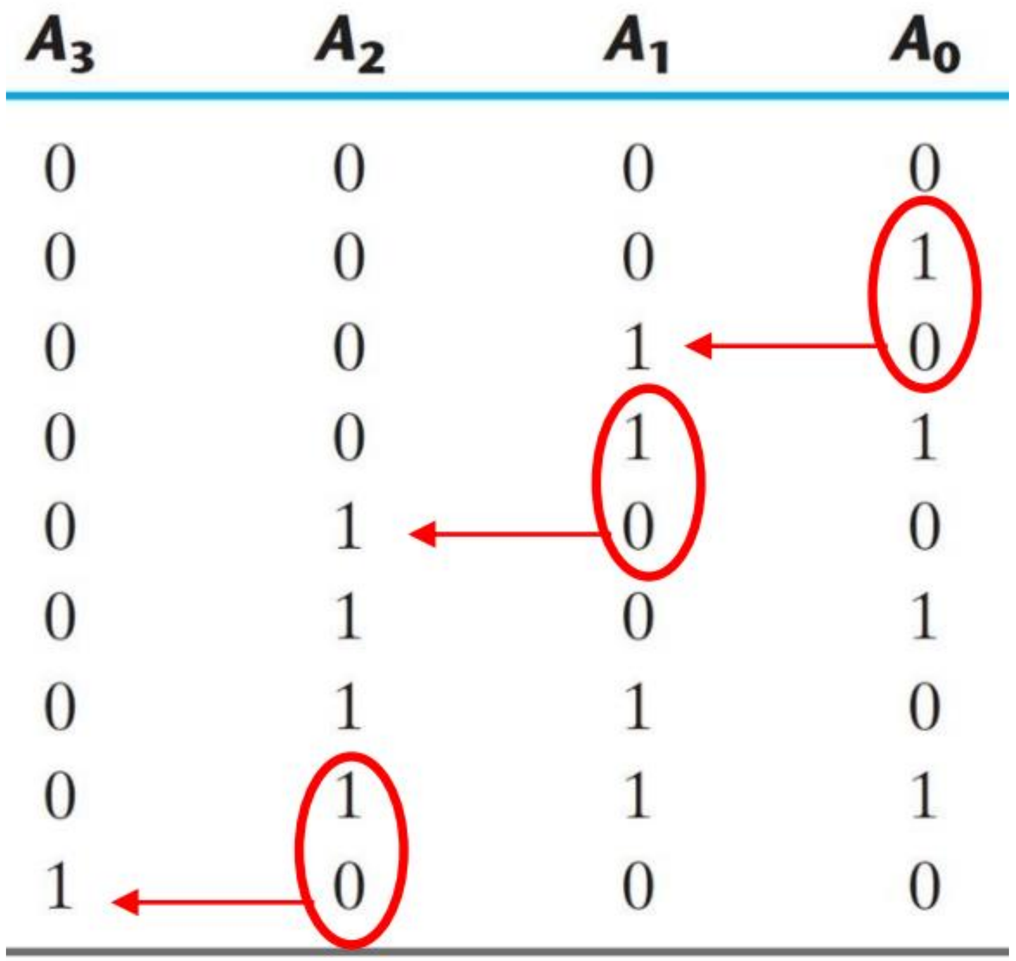
- A counter that follows the **binary** number **sequence** is called a **binary counter**
  - (**n**-bit binary counter: count from **0** to **2^n -1**)

Binary Ripple Counters

- In **Ripple** Counters, a flip-flop output **transition serves as a trigger** for other flip-flops. In other words, the **CLK** input of some or all flip-flops **are triggered by the transition** that occurs in other flip-flop outputs.

- **Binary ripple counter** consists of a series of **complementing flip-flops**.
  - Either **Count-up** or **Count-down**
  - Could be implemented using **D, T,** or **JK** FFs.
  - **Negative** Edge or **Positive** Edge FFs could be used

Binary Count-up

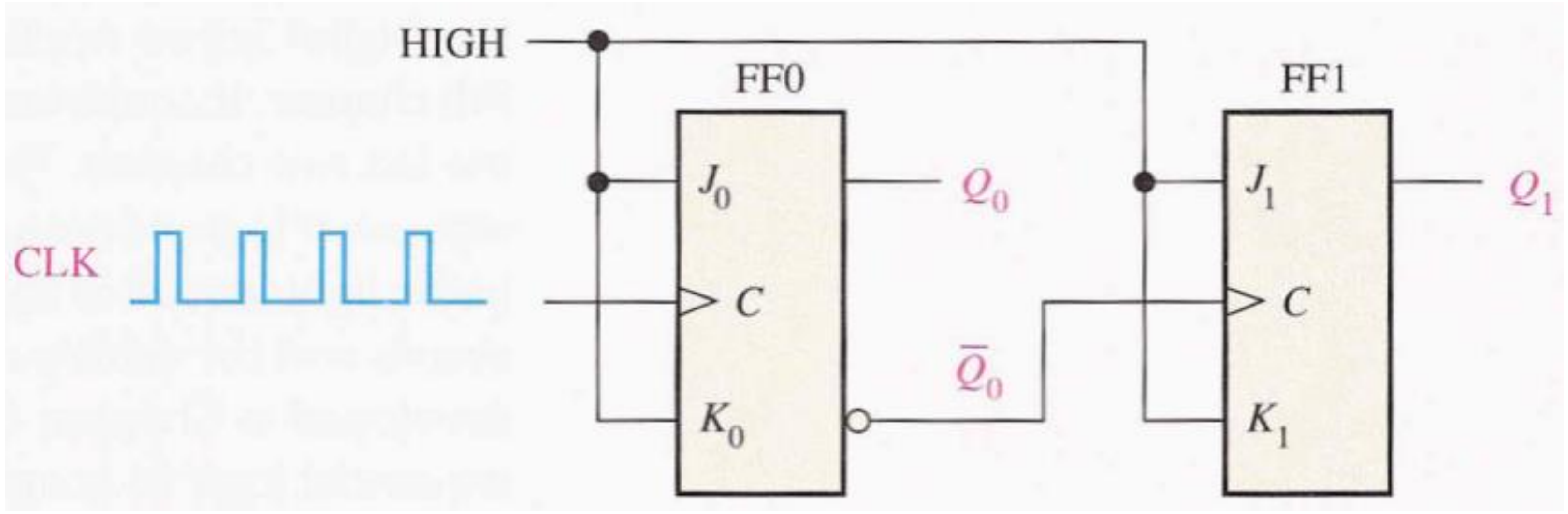


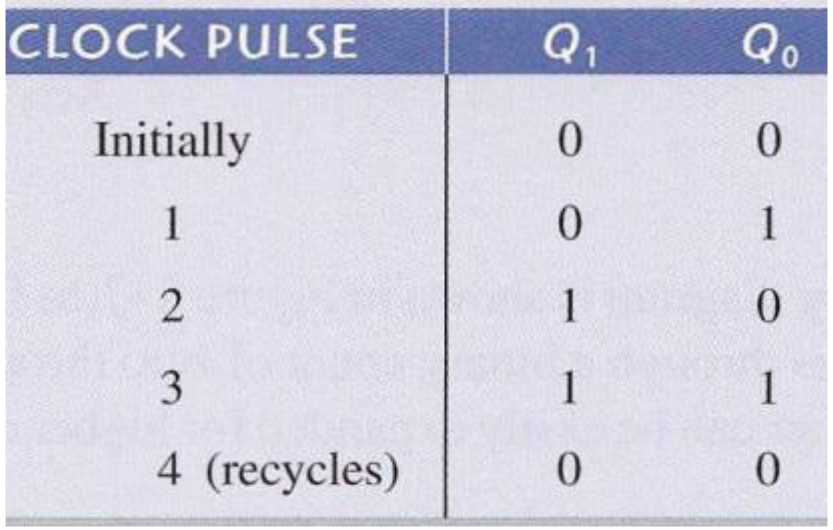
A_3	A_2	A_1	A_0
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0

1. The **first** FF is complemented **every** Clock Edge
2. The **second** FF is complemented when the **first** FF is **changed from 1 → 0**. (The **first** FF **acts** as the **Clock** for the **second** FF).
3. **Same** Applied for **higher** FFs.

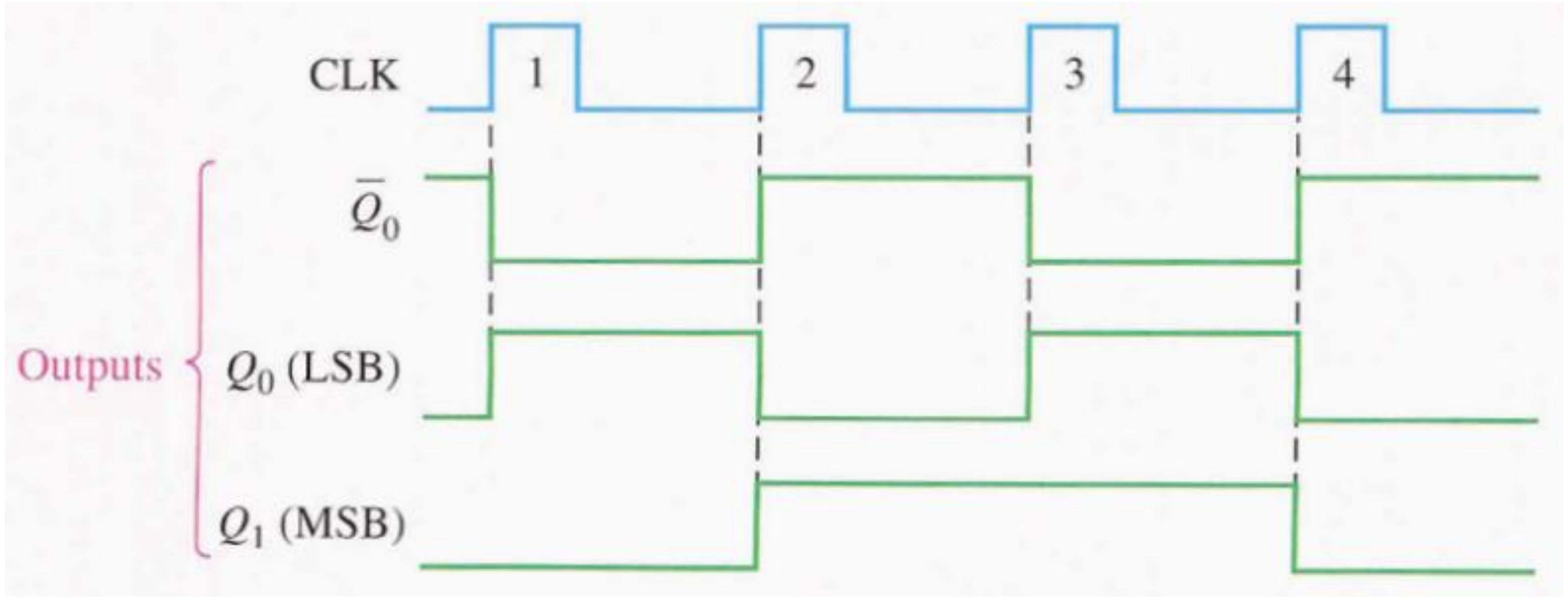
Binary Ripple Counters (Count-up, +Ve Edge)

2-bit Ripple Counter (JK-FF)





CLOCK PULSE	Q_1	Q_0
Initially	0	0
1	0	1
2	1	0
3	1	1
4 (recycles)	0	0



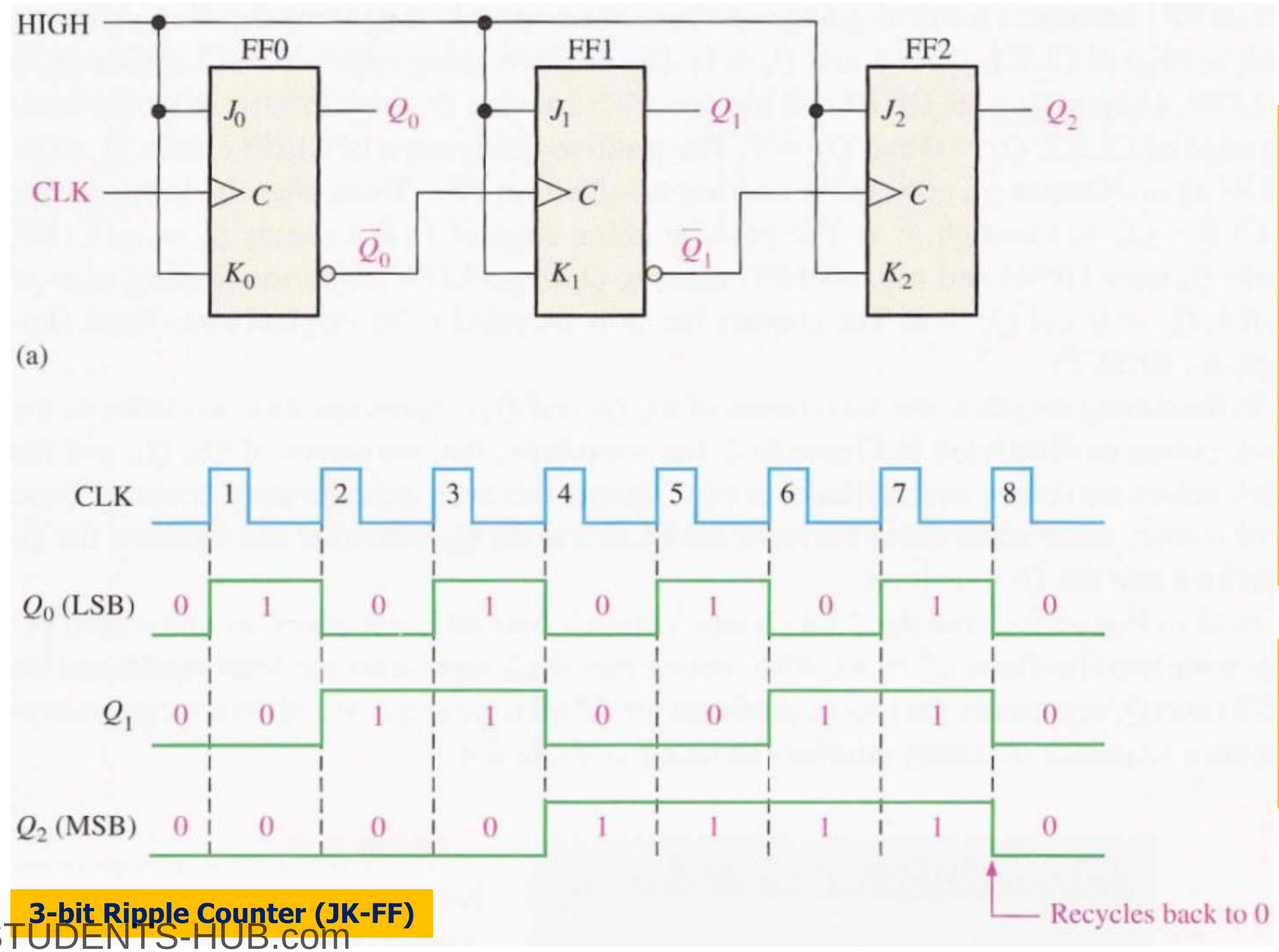
Notice:

When **+Ve** Edge FF is used → Connect **Q′** to the clock of the **next** FF

When **–Ve** Edge FF is used → Connect **Q** to the clock of the **next** FF

Binary Ripple Counters (Count-up, +Ve Edge)

ENCS 2340

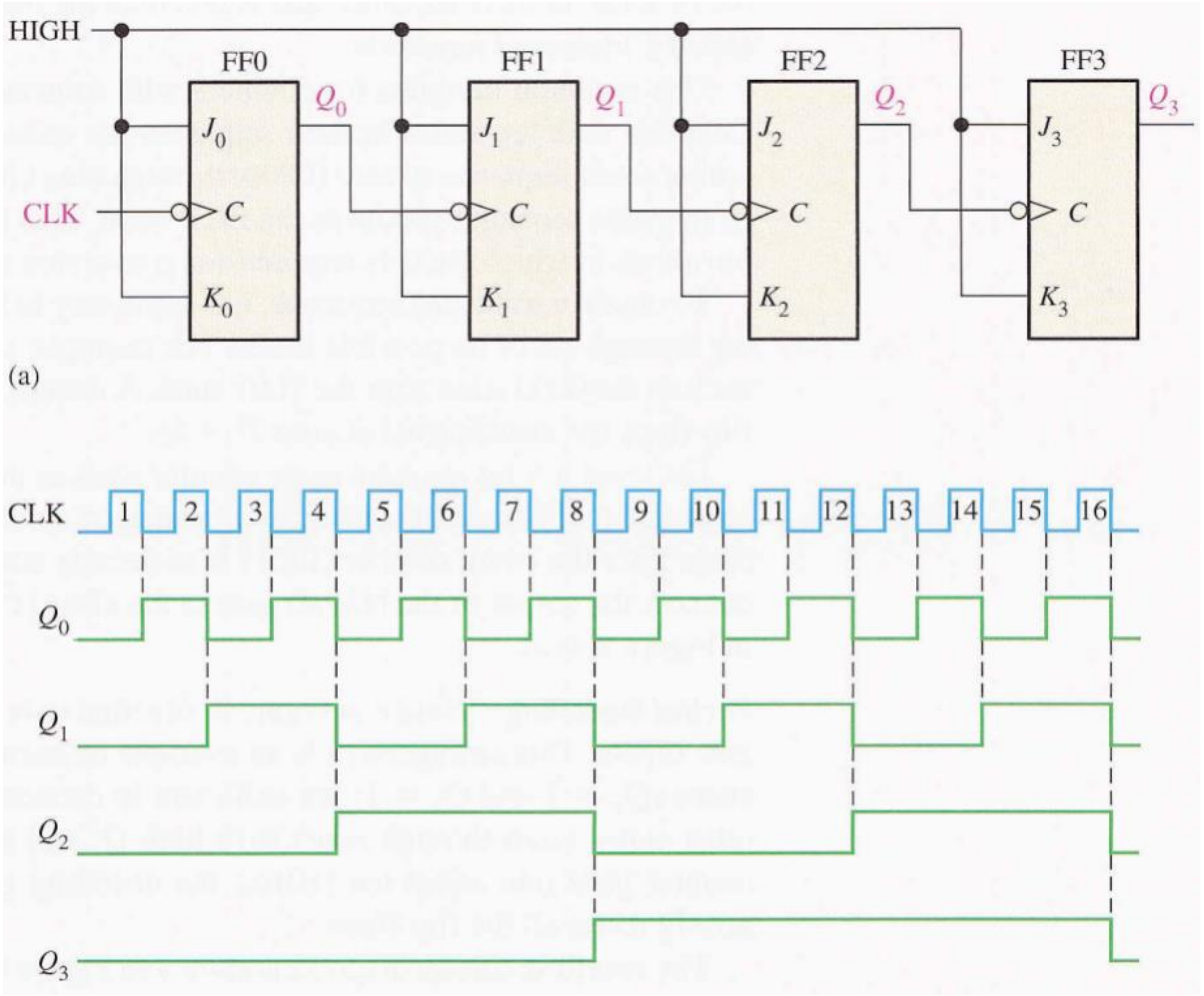


- Transition of Q_0 from **1 to 0** triggers FF1 and it's output is complemented (Q_1 is complemented)
- Similarly, as Q_1 changes from **1 to 0**, FF2 is triggered (Q_2 is complemented)
- This is exactly what happens in binary counting

Notice:
When **+Ve** Edge FF is used → Connect **Q'** to the clock of the **next** FF

3-bit Ripple Counter (JK-FF)

Binary Ripple Counters (Count-up, -Ve Edge)



Notice:
When **–Ve** Edge FF is used → Connect **Q** to the clock of the **next** FF

4-bit Ripple Counter (JK-FF)

Binary Ripple Counters (Count-up, -Ve Edge)

4-bit Ripple Counter (T-FF & D-FF)

Important Points:

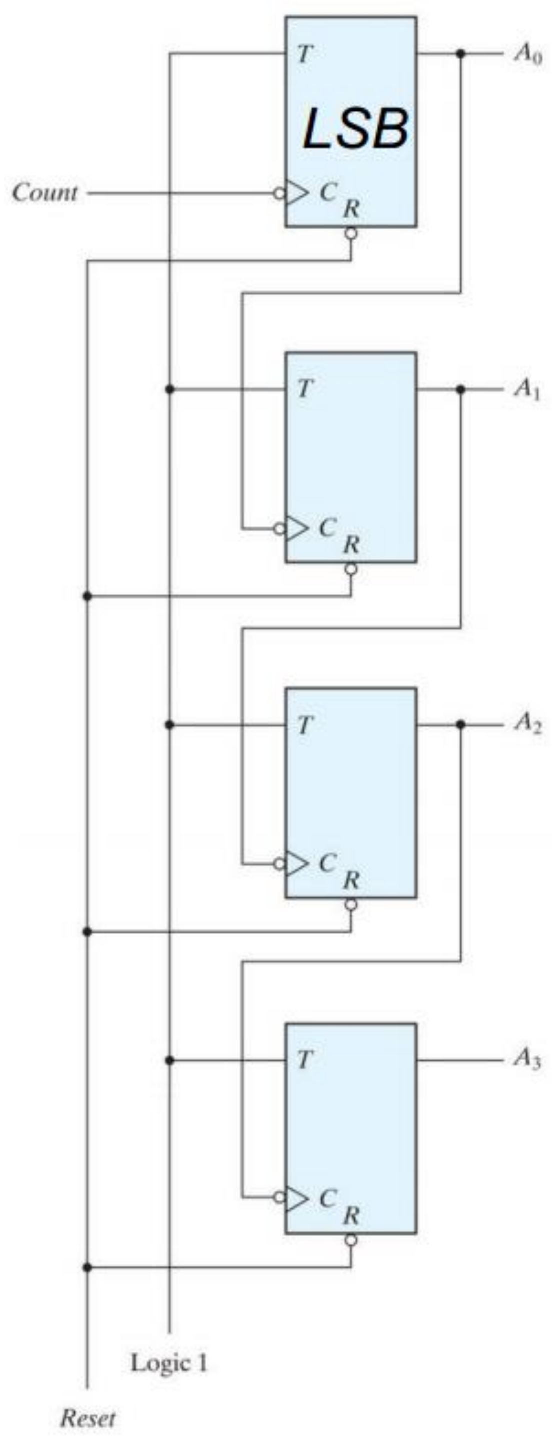
1) To activate the **complement** Mode in **JK/T** FFs → connect Logic **1** (High) to the **JK/T** inputs.

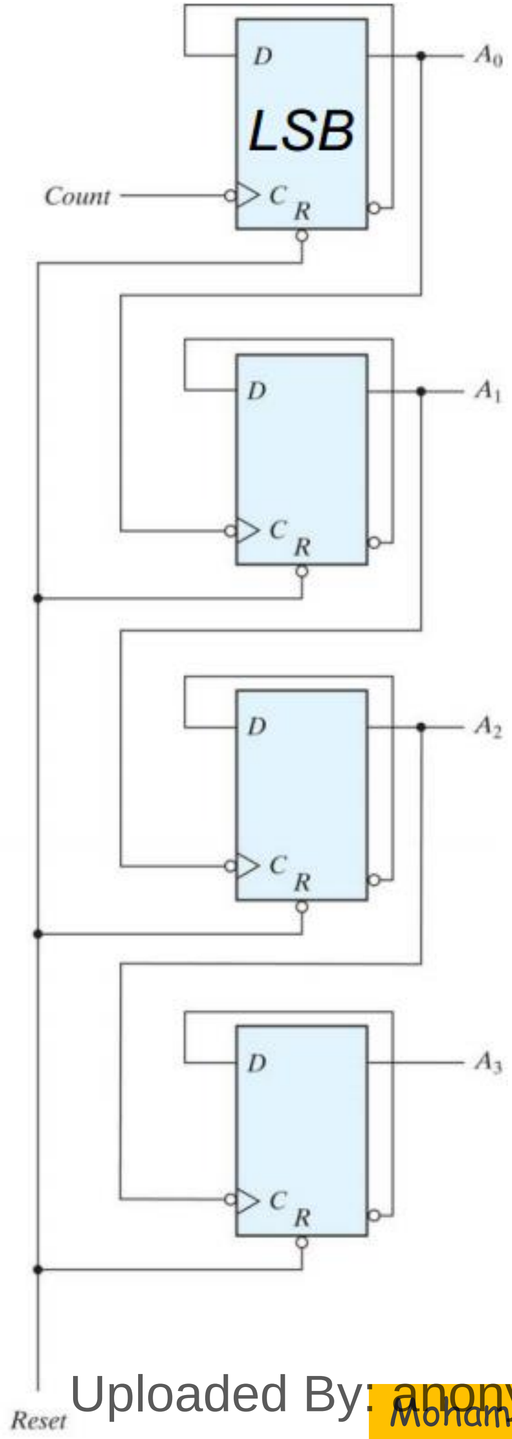
2) To activate the **complement** Mode in **D** FFs → connect the **inverted** FF **output** (**Q'**) back to the **D** input.

3) **Reset** Signal usually used to clear/reset the counter to the **initial** count/state at **any** time.

4) The **CLK** signal connected to the **first** FF is usually named **Count**

Notice:
When **–Ve** Edge FF is used → Connect **Q** to the clock of the **next** FF

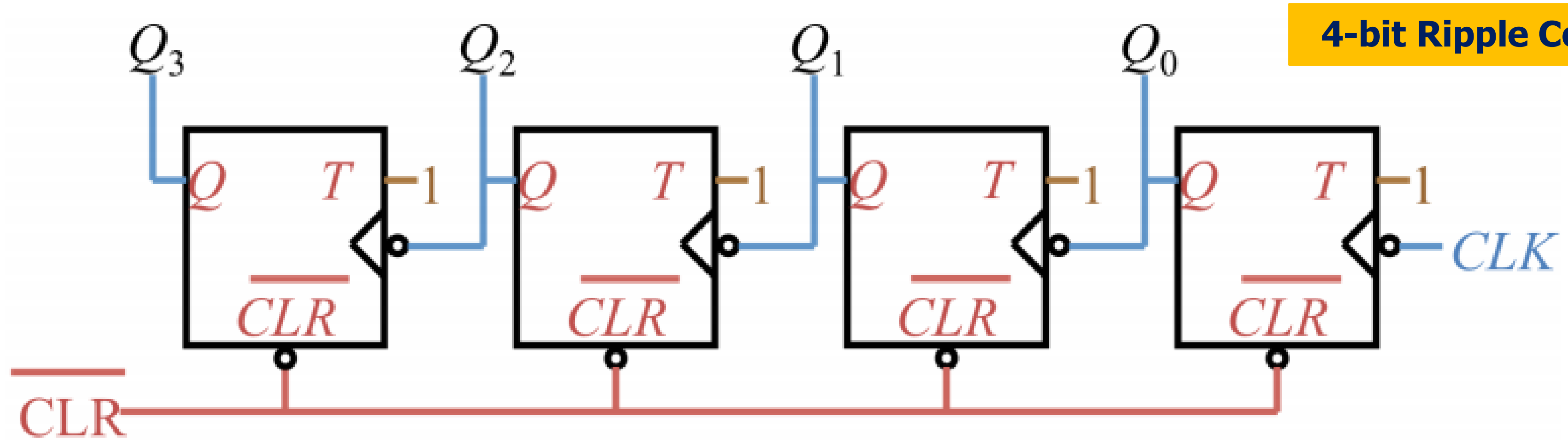




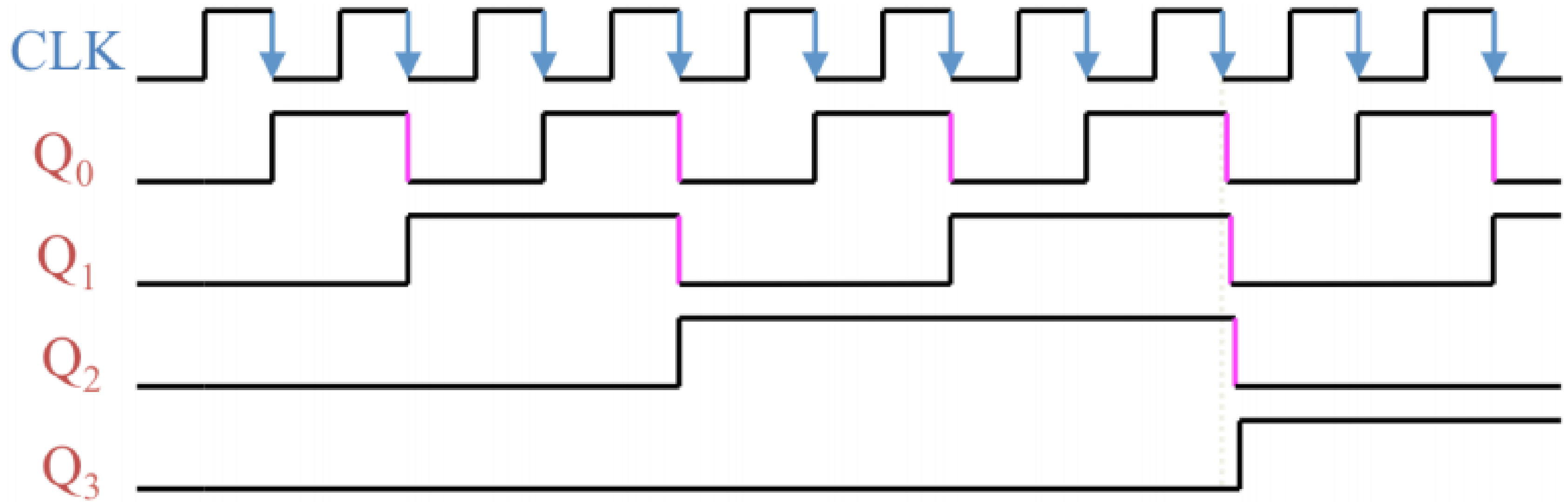
ENCS 2340

Binary Ripple Counters (Count-up, -Ve Edge)

4-bit Ripple Counter (T-FF)

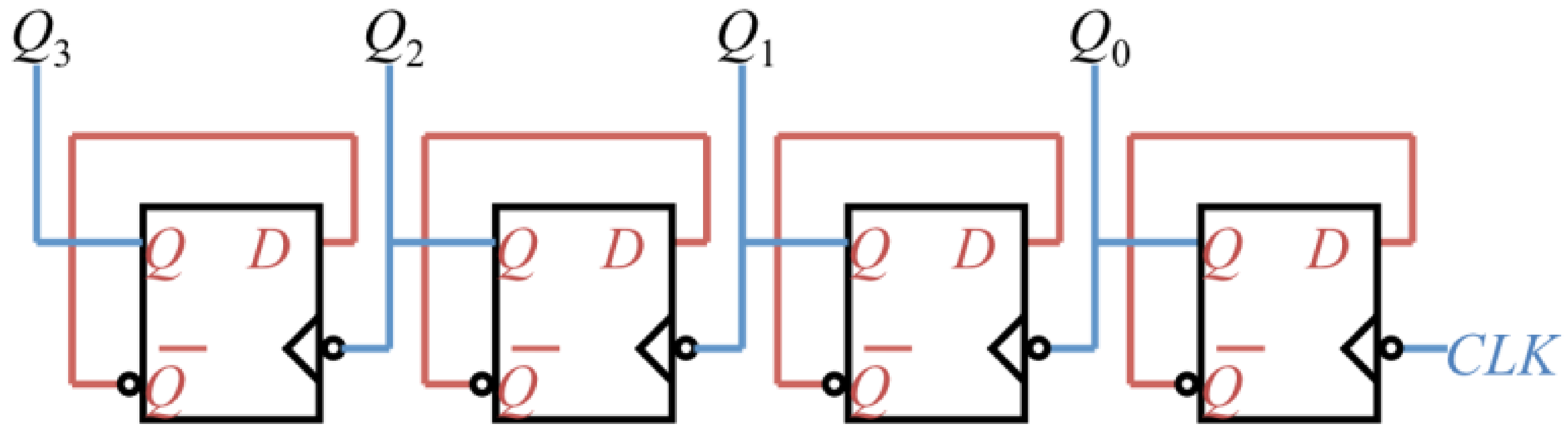


Timing Diagram

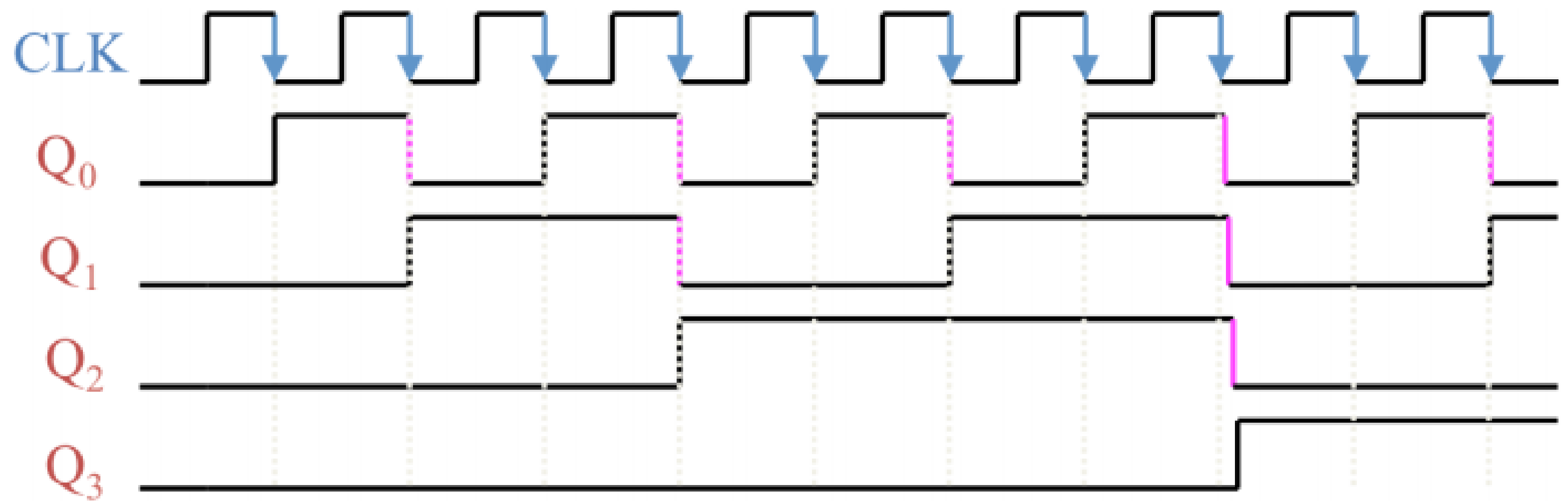


Binary Ripple Counters (Count-up, -Ve Edge)

4-bit Ripple Counter (D-FF)



Timing Diagram



Binary Ripple Counters (Count-down)

- A binary **count-down** counter is a binary counter with a **reverse** count
- **Initial** State is **All 1s** (4-bit counter starts with **1111**)
- Any bit in the sequence is **complemented** if its **previous** least significant bit goes from **0 to 1**

Same design as Count-up **Except**:

When **+Ve** Edge FF is used → Connect **Q** to the clock of the **next** FF

When **–Ve** Edge FF is used → Connect **Q′** to the clock of the **next** FF

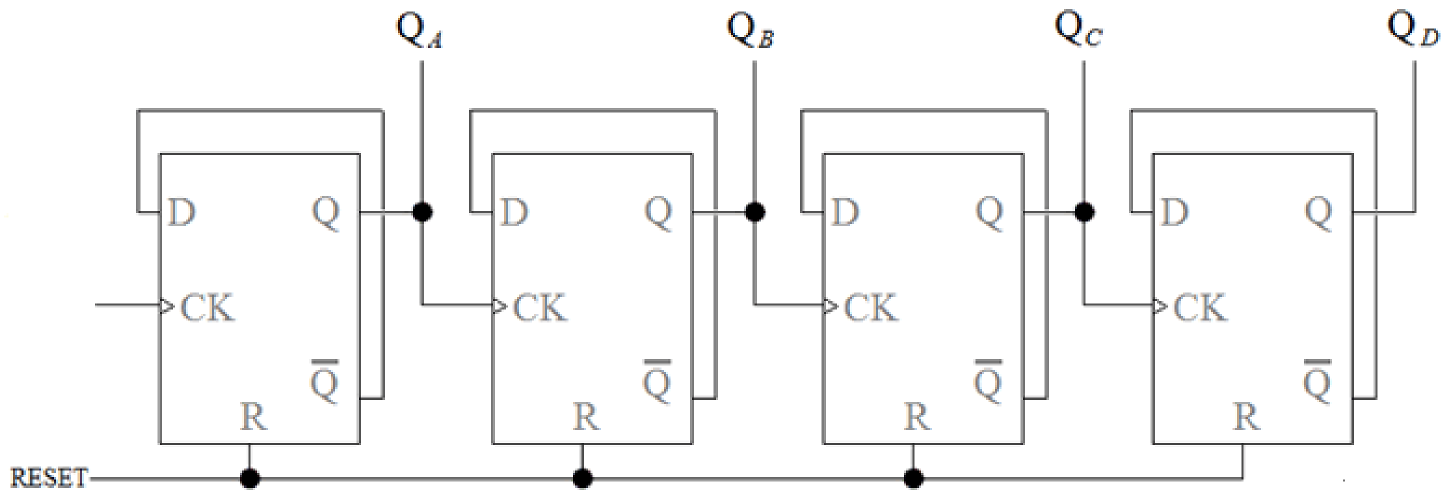
Common Practice:
Count-Up: Use **Negative** Edge
Count-Down: Use **Positive** Edge

Binary Count Down

A_3	A_2	A_1	A_0
1	1	1	1
1	1	1	0
1	1	0	1
1	1	0	0
1	0	1	1
1	0	1	0
1	0	0	1
1	0	0	0
0	1	1	1
0	1	1	0

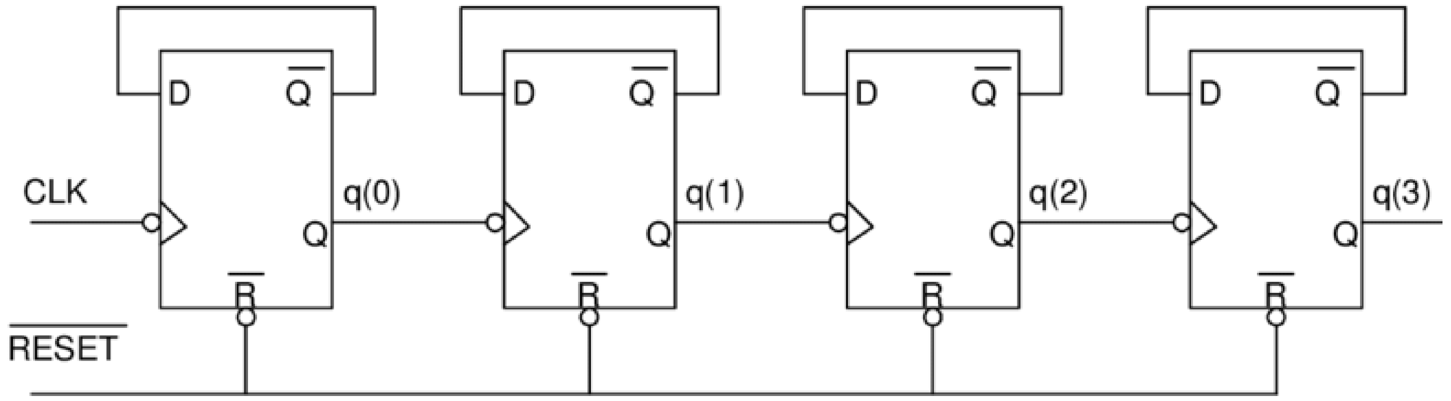
Binary Ripple Counters (Count-down vs Count-up)

Binary Count Down +Ve Edge



Common Practice:
Use **+Ve** Edge FF → Connect **Q** to the clock of the **next** FF

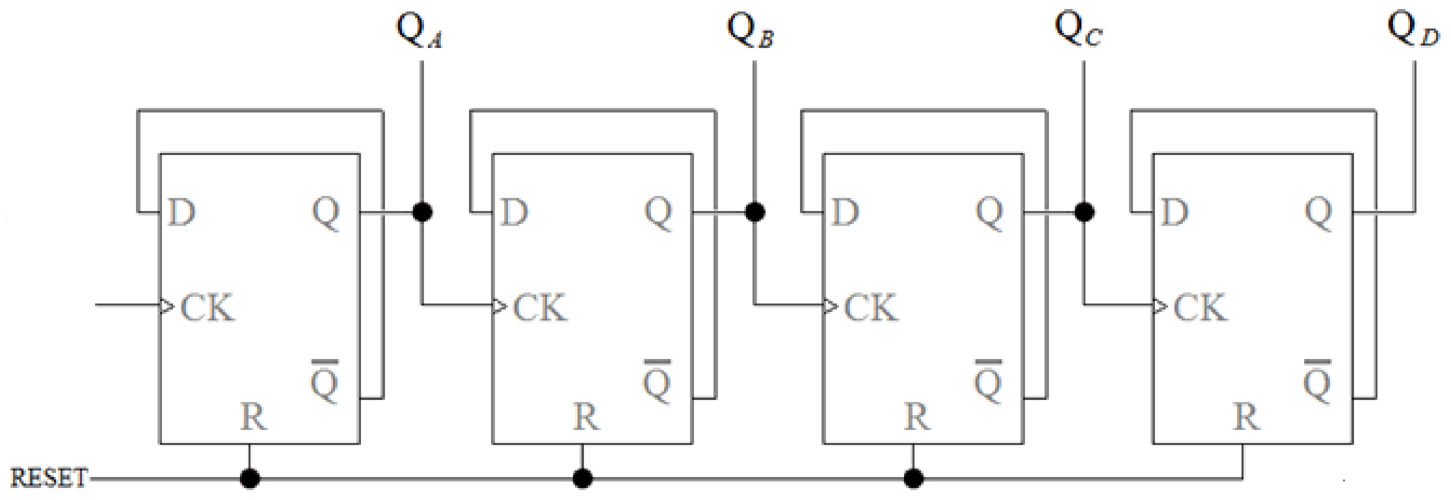
Binary Count UP (-Ve Edge)



Common Practice:
Use **-Ve** Edge FF → Connect **Q** to the clock of the **next** FF

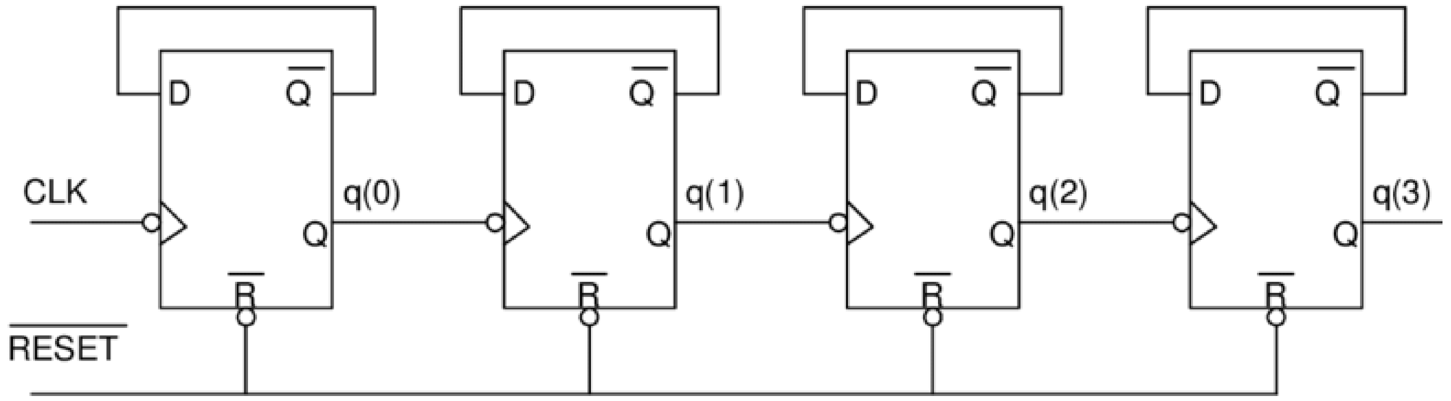
Binary Ripple Counters (Count-down vs Count-up)

Binary Count Down +Ve Edge



Common Practice:
Use **+Ve** Edge FF → Connect **Q** to the clock of the **next** FF

Binary Count UP (-Ve Edge)

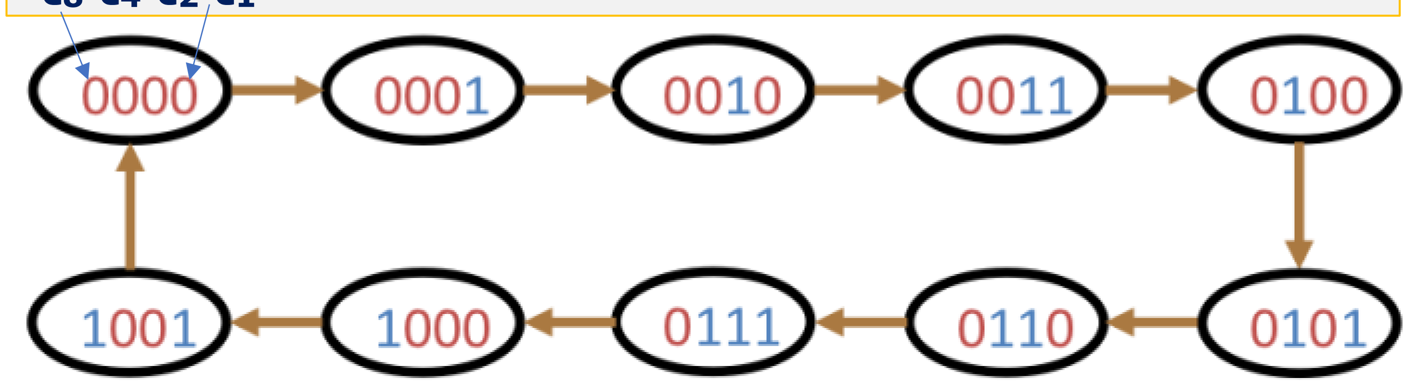


Common Practice:
Use **-Ve** Edge FF → Connect **Q** to the clock of the **next** FF

BCD Ripple Counters

- A **decimal** (**BCD**) counter follows a sequence of **10** states and **returns to 0** after the count of **9**
  - (Count 0→9 then restart to 0)
- **10** States → **4** FFs are needed ($2^3 = 8$, < 10 → 4 is needed)
- **BCD** counter sequence: **0000** → **1001** then restart to **0000**
- **Common** Names: Decimal / BCD/ Decade

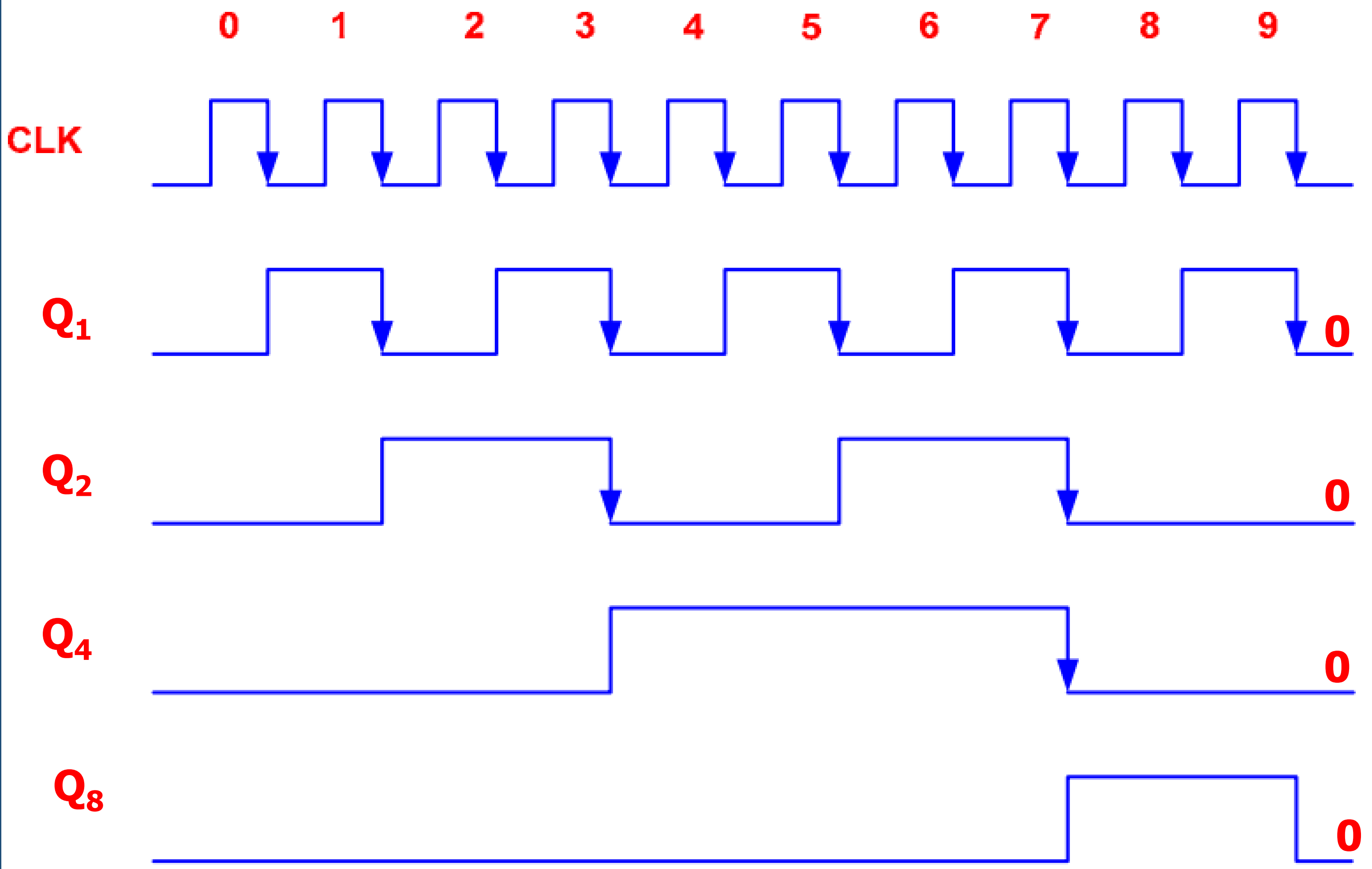
The **4** FFs are named according to their decimal **weight** → **$Q_8Q_4Q_2Q_1$**



BCD Counter

Notice: **Q_8** and **Q_2** FFs **reset** after the **10th** pulse
In Binary Counter: 1001 → 1010 (Both Q3,Q1 = 1)

BCD Ripple Counters



BCD Counter - Timing Diagram

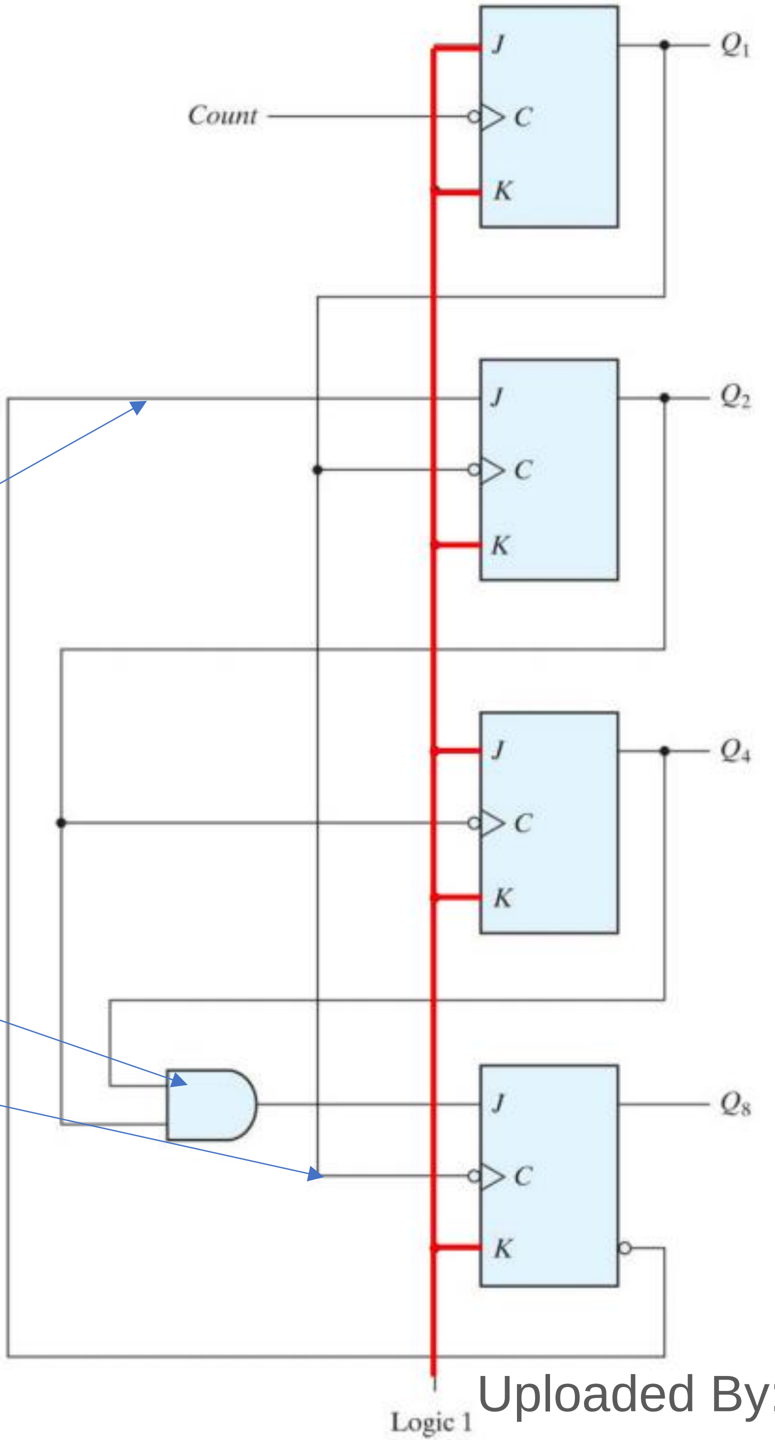
Notice:

1) Q_1 complements after **each** clock pulse.
2) Q_2 complements every time Q_1 goes from **1 to 0**, as long as Q_8 **= 0**.
3) When Q_8 = **1**, Q_2 remains at **0**.
4) Q_4 complements every time Q_2 goes from **1 to 0**.
5) Q_8 remains at **0** as long as Q_2 or Q_4 is **0**
6) When both Q_2 and Q_4 become **1**, Q_8 complements when Q_1 goes from **1 to 0**.
7) Q_8 is cleared on the next transition of Q_1

Point 3: → Connect Q_8' to **J** input of Q_2
Point 6: → Connect **AND**(Q_2,Q_4) to **J** input of Q_8
Point 7: → Connect Q_1 to the Clock of Q_8

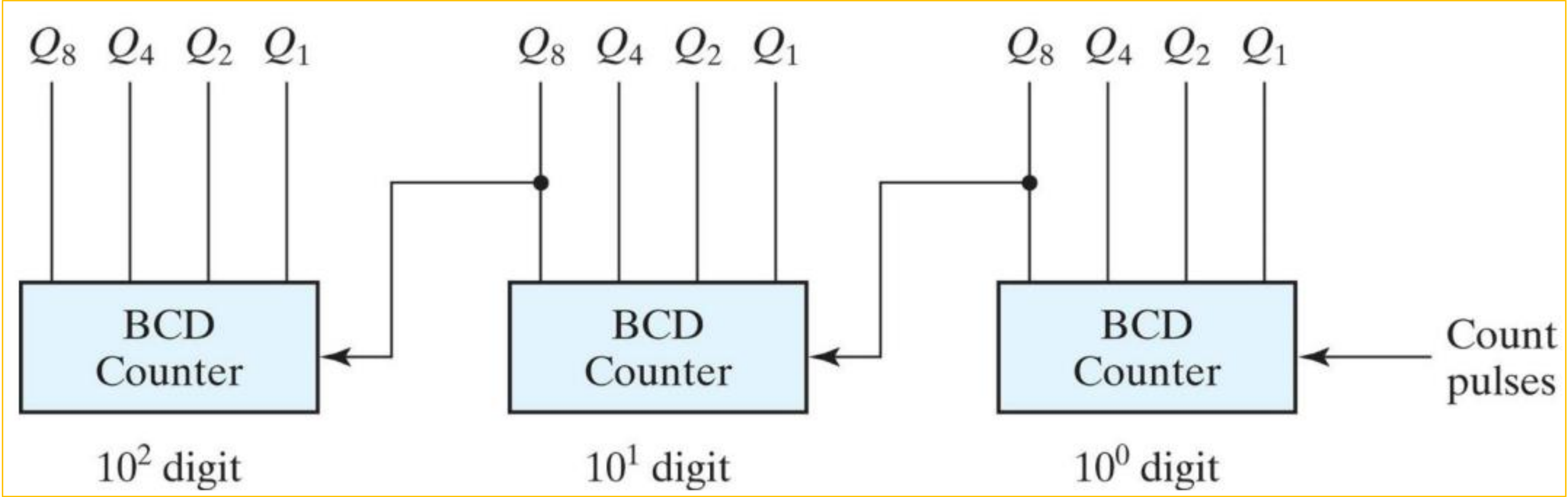
BCD Ripple Counters

- Use **JK** FF, -Ve Edge (As in Binary Ripple Counter)
- Apply the following Modifications:
  1) Connect $\mathbf{Q_8}'$ to **J** input of $\mathbf{Q_2}$
  2) Connect **AND**($\mathbf{Q_2}$,$\mathbf{Q_4}$) to **J** input of $\mathbf{Q_8}$
  3) Connect $\mathbf{Q_1}$ to the Clock of $\mathbf{Q_8}$



BCD Ripple Counters

- We can **cascade** BCD counters (connect in **series**) to obtain **multiple** decimal digits counter
- To count from **0** to **999**
  - **Three** BCD counters is needed (**One** for **Each** Digit)
  - Constructed by **connecting** the BCD counters in **cascade**



Remember: All **ripple** counters are an **asynchronous** sequential circuits.

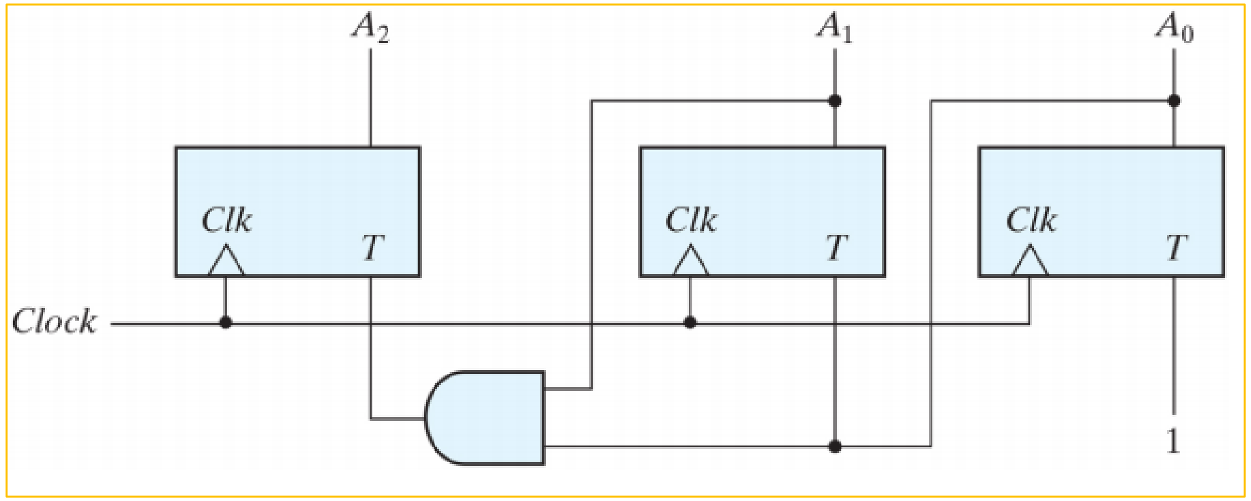
Synchronous Counters

- A **common** clock **triggers all** flip-flops simultaneously
- Typically, their design procedure follows that for Sequential circuits

- The design of a synchronous binary counter is so **simple** that there is **no need** to go through a sequential logic design process
- In a synchronous binary counter, the flip-flop in the **least** significant position (**LSB**) is **complemented** with **every** pulse
- A flip-flop in any **other** position is **complemented** when **all the bits** in the **lower** significant **positions** are **equal to 1**

Binary Count UP

A_3	A_2	A_1	A_0
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0

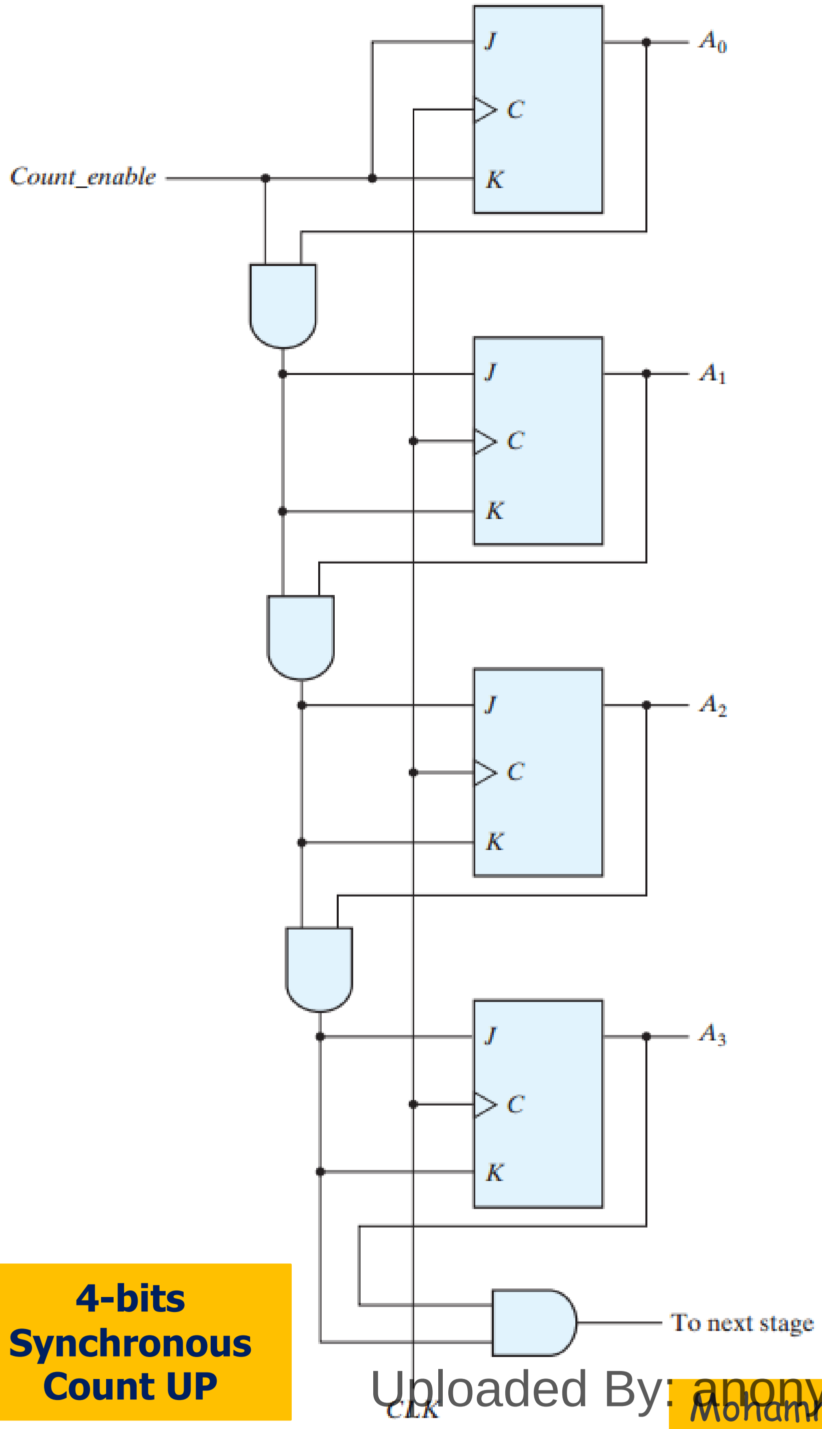


3-bits Synchronous Binary Up Counter using T-FF (Example in **Ch-5**)

- **A_0** complements every time the count pulses go from 1 to 0
- **A_1** complements only when A_0 is 1 and goes to 0
- **A_2** complements only when A_1 and A_0 are 1 and going to 0
- **A_3** complements only when A_2, A_1 and A_0 are all 1 and going to 0

Synchronous UP Counters

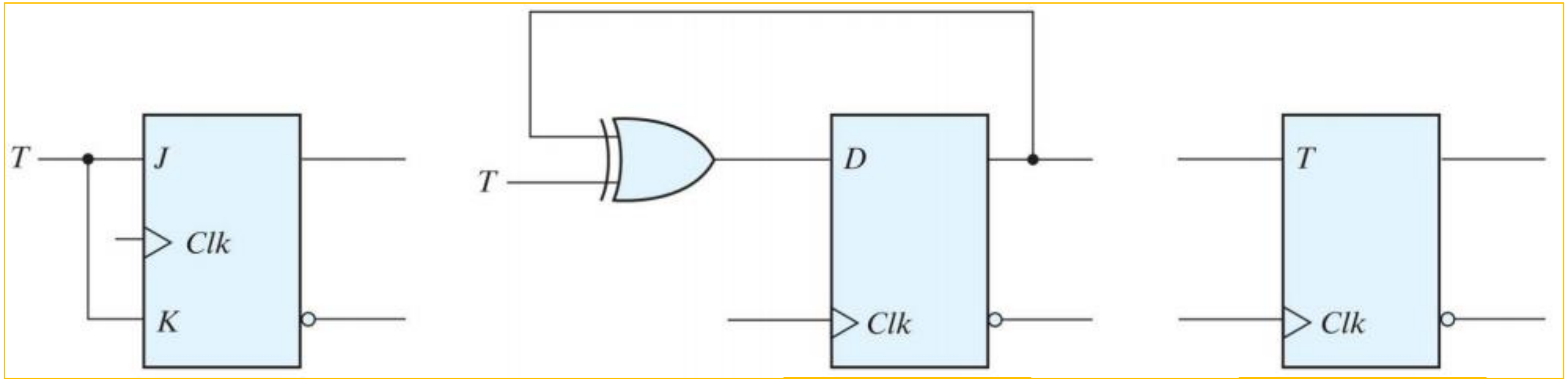
- Synchronous binary counters have a **regular pattern** and can be constructed with **complementing** **flip-flops** and **gates**
- The counter is enabled by **Count_enable**
  - Count_enable =**0** ➔ **No Change**
- The counter can be **extended** to **any** number of stages, with **each** stage having **an additional** **FF** and an **AND** gate
- The synchronous counter can be designed with either the positive or the negative clock edge



4-bits Synchronous Count UP

Complementing Flip-Flops

- As stated earlier, Synchronous counters need to be designed with **complementing** Flip-Flops
- The **complementing** FF in can be of either the **JK** type, the **T** type, or the **D** type with **XOR** gates



Complementing JK-FF

Complementing D-FF

Complementing T-FF

Notice:
T-FF: is a **complementing** FF by its **nature**
JK-FF: can be <u>converted</u> to a **complementing** FF (T-FF), by **connecting** **both J,K** to **same input**
D-FF: can be <u>converted</u> to a **complementing** FF (T-FF), by **XOR** **both** the **Present State** and the **input**

Synchronous Down Counter

- A synchronous **count-down** binary counter goes through the binary states in **reverse** order, from 1111 down to 0000 and back to 1111
- The bit in the **least** significant position(LSB) is **complemented** with **each** pulse
- A bit in **any** other position is **complemented** if **all lower** significant bits are **equal to 0**
- A countdown binary counter can be constructed as up counter, **except** that the **inputs** to the **AND** gates must come from the **complemented outputs** of the **FF (Q')**

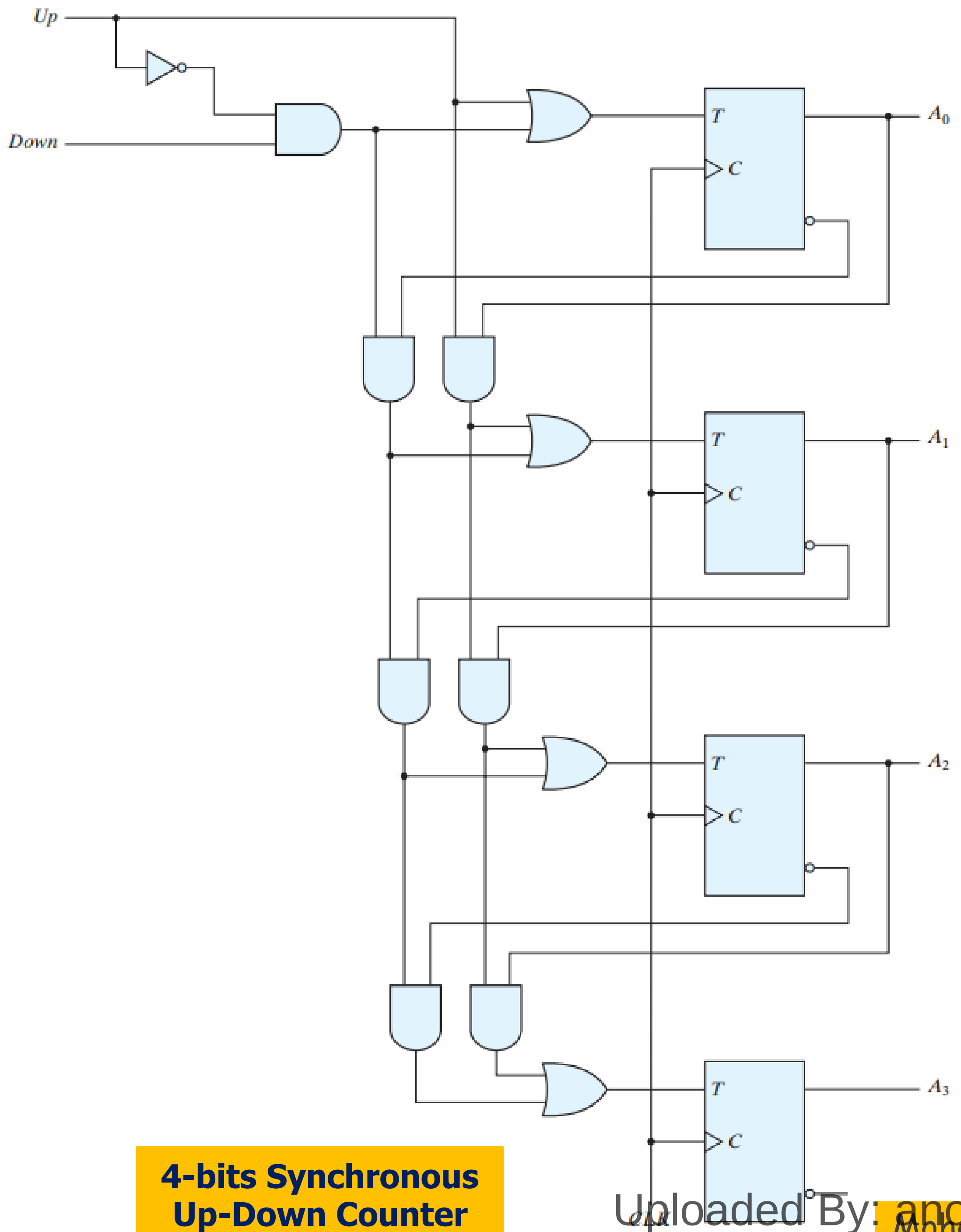
Binary Count Down

A_3	A_2	A_1	A_0
1	1	1	1
1	1	1	0
1	1	0	1
1	1	0	0
1	0	1	1
1	0	1	0
1	0	0	1
1	0	0	0
0	1	1	1
0	1	1	0

Synchronous Up-Down Counter

- The two (Up & Down) counters can be combined in one circuit to form a counter capable of counting either up or down
- It has an Up and Down **control** Inputs
  1. Up = 0 & Down = 0 ➔ **Don't Count**
  2. Up = 0 & Down = 1 ➔ Count **Down**
  3. Up = 1 & Down = X ➔ Count **Up**
- This set of conditions ensures that only **one operation** is **performed** at any given time
- The **Up** input has **priority**

Up	Down	Operation
0	0	Same
0	1	Down
1	x	Up

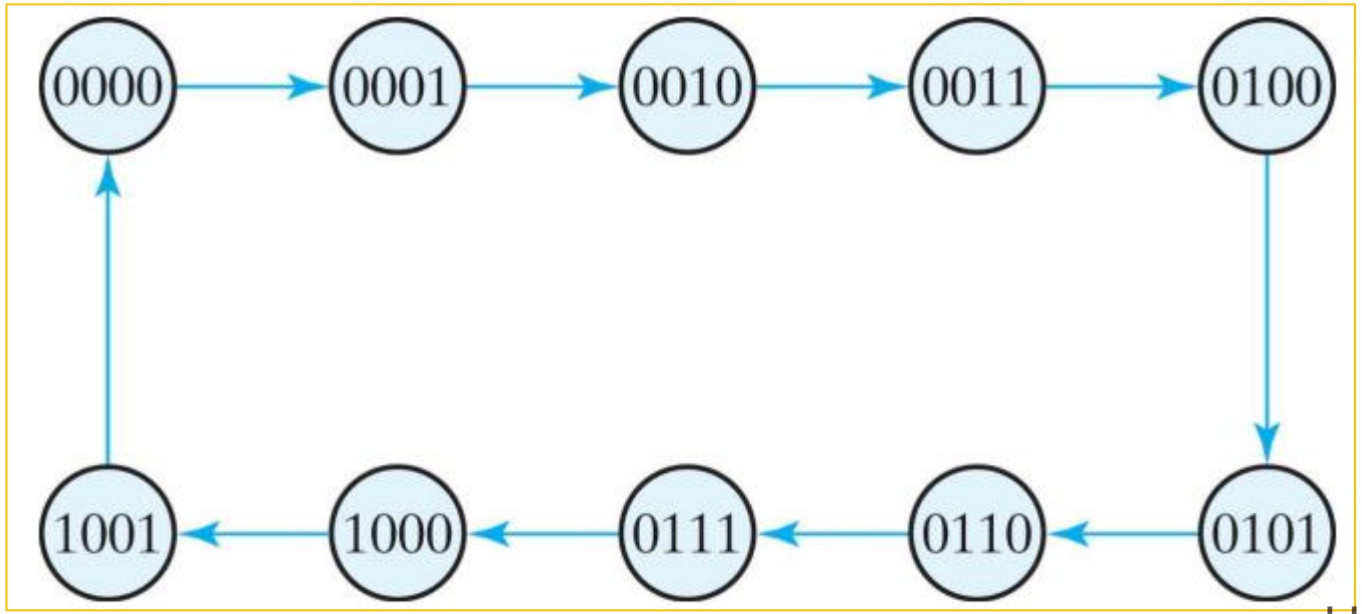


4-bits Synchronous Up-Down Counter

BCD Synchronous Counter

- A **BCD** counter counts in binary-coded decimal from 0000 to 1001 and back to 0000
- Because of the return to 0 after a count of 9, a BCD counter **does not have a regular pattern**, unlike a straight binary count
- To derive the circuit of a BCD synchronous counter, it is necessary to **go through a sequential** circuit **design** procedure
- An **output** is defined, to **enable** the **next** decade counter (stage)

BCD Counter – State Diagram



BCD Synchronous Counter

The **output** y, equal to **1** when the **present** state is **1001**, to **enable** the **count** of the **next-higher** significant decade while the same pulse switches the present decade from 1001 to 0000

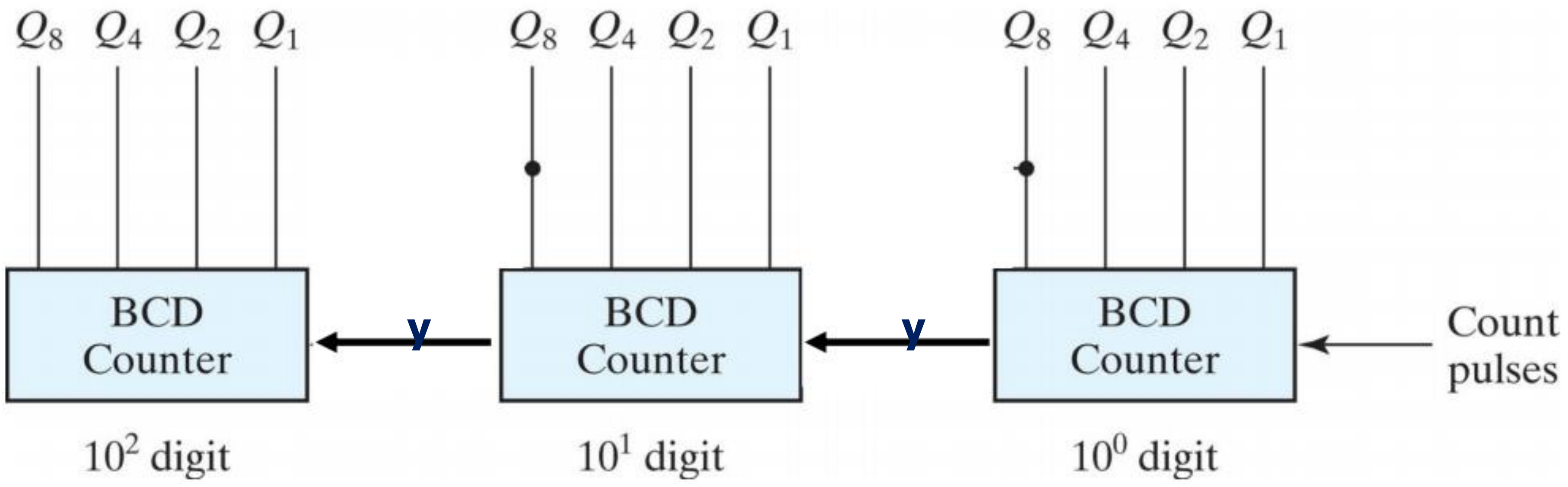
Present State				Next State				Output	Flip-Flop Inputs			
Q_8	Q_4	Q_2	Q_1	Q_8	Q_4	Q_2	Q_1	y	TQ_8	TQ_4	TQ_2	TQ_1
0	0	0	0	0	0	0	1	0	0	0	0	1
0	0	0	1	0	0	1	0	0	0	0	1	1
0	0	1	0	0	0	1	1	0	0	0	0	1
0	0	1	1	0	1	0	0	0	0	1	1	1
0	1	0	0	0	1	0	1	0	0	0	0	1
0	1	0	1	0	1	1	0	0	0	0	1	1
0	1	1	0	0	1	1	1	0	0	0	0	1
0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	0	0	1	0	0	0	0	1
1	0	0	1	0	0	0	0	$\bigcirc\!\!\!\!1$	1	0	0	1

BCD Counter – State Table (Using T-FF)

BCD Synchronous Counter

- The **unused** states (**6** States) for minterms **10 to 15** are taken as **don't-care** terms
- The **simplified T-FF** and **Output** functions (using K-Maps) are:

$$T_{Q1} = 1$$
$$T_{Q2} = Q_8' Q_1$$
$$T_{Q4} = Q_2 Q_1$$
$$T_{Q8} = Q_8 Q_1 + Q_4 Q_2 Q_1$$
$$y = Q_8 Q_1$$

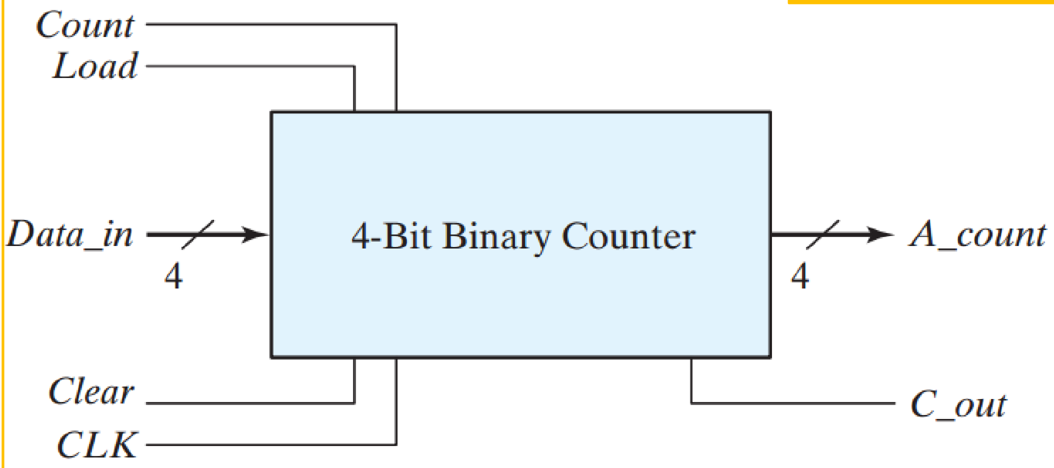


BCD Counter – 3 Stages

Binary Counter with Parallel Load

- **Parallel load** is used to **start** counting from a **defined** state (does not always have to be ZERO)
- **Clear input** is used to **clear** (reset to 0) the counter **asynchronously**
- **Carry bit** is set when the count **reaches all 1's** (**saturate**), to give indication to a **following** stage to **start** the count

4-bit binary counter with parallel load



Block Diagram

Clear	CLK	Load	Count	Function
0	X	X	X	Clear to 0
1	↑	1	X	Load inputs
1	↑	0	1	Count next binary state
1	↑	0	0	No change

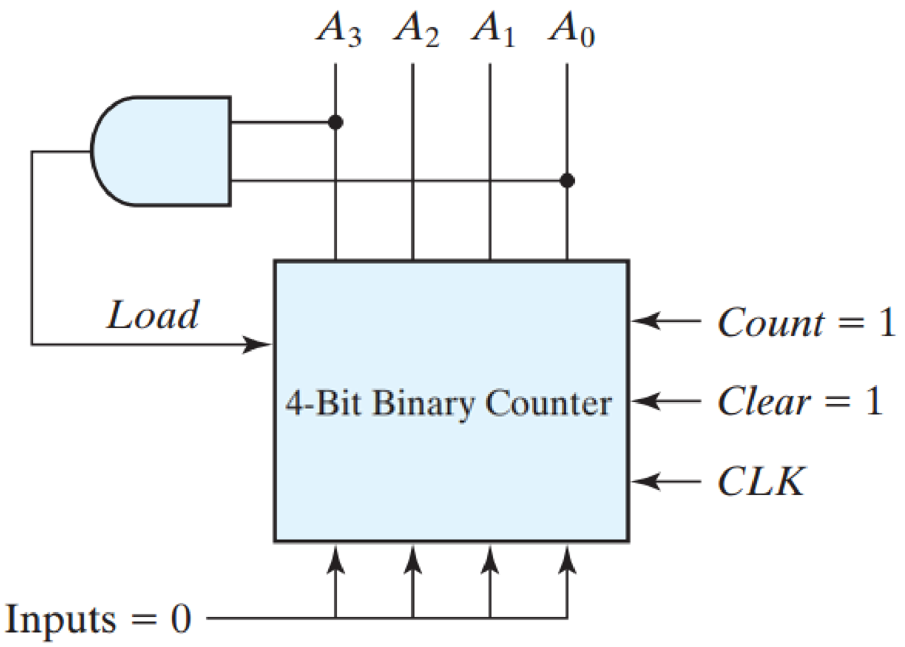
Function Table

3 Control Inputs (With the following Priority: **High → Low**: Clear → Load → Count)
1 Clock input
n-bits **data/load** input, **n**-bits **count value**/output (here n=4)
1-bit **carry** output

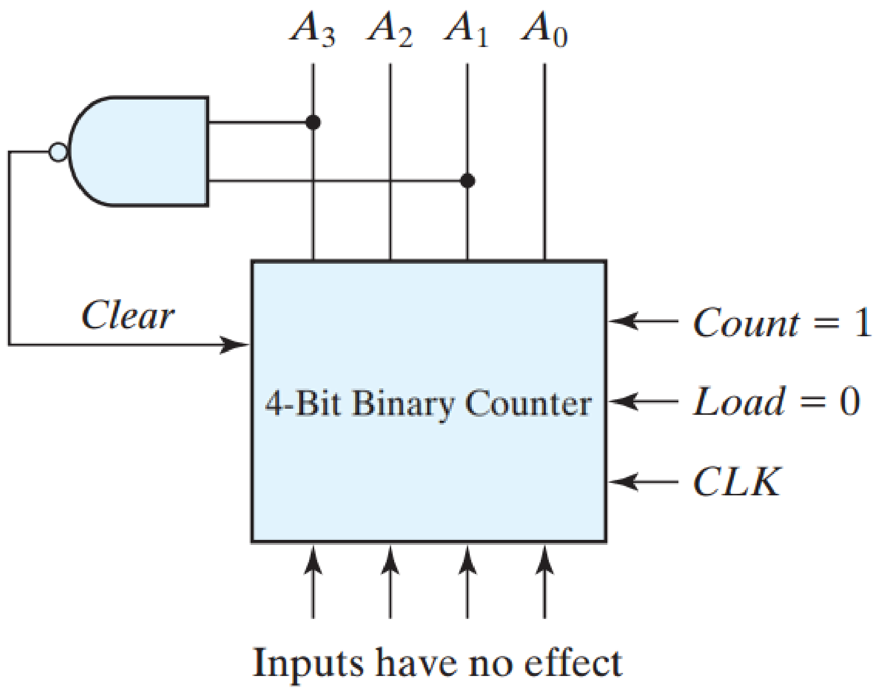
Binary Counter with Parallel Load

A counter with a parallel load can be used to **generate** **any** desired **count** **sequence**

Two ways to achieve a BCD counter using a counter with parallel load



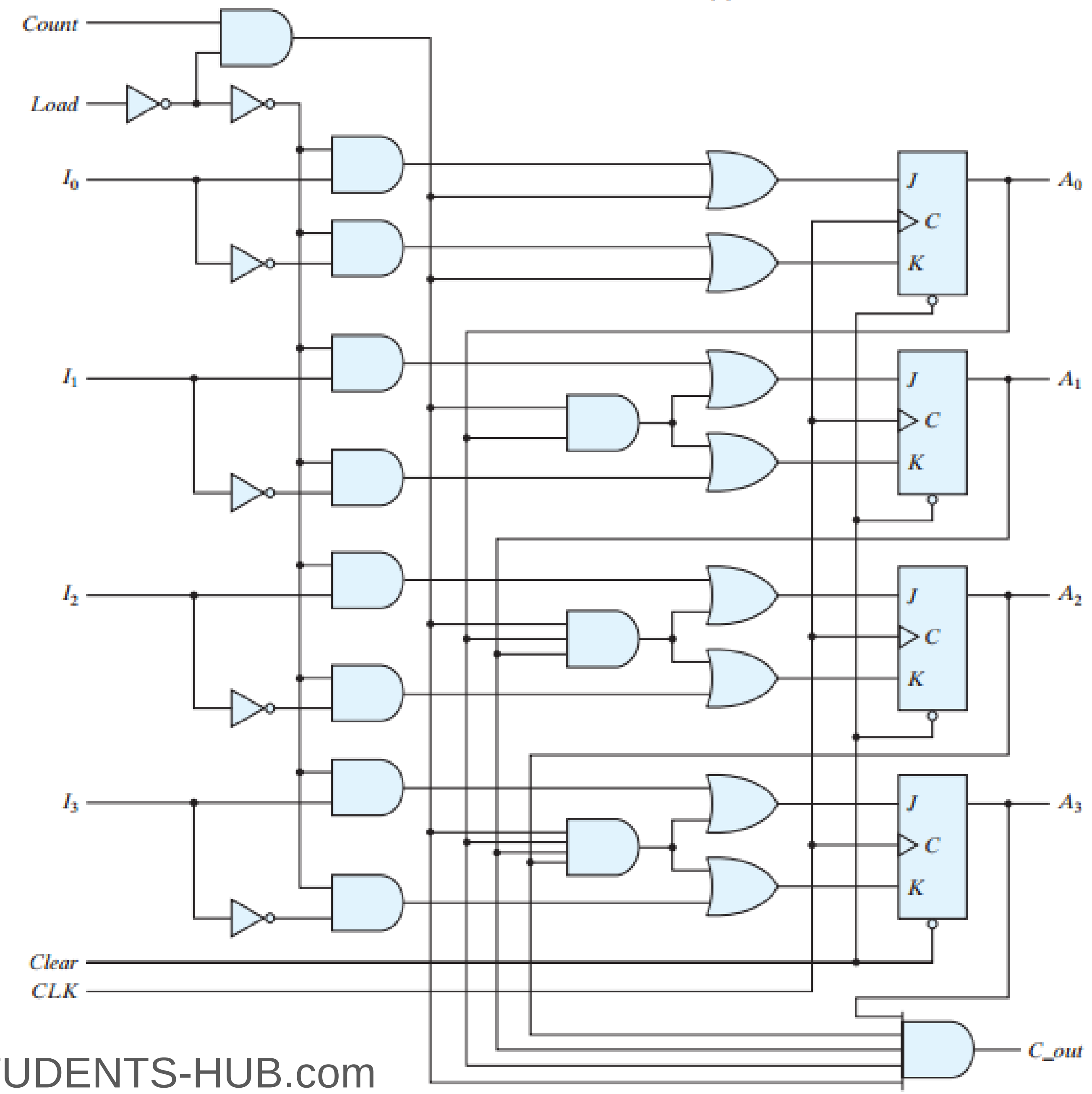
Using Load Control



Using Clear Control

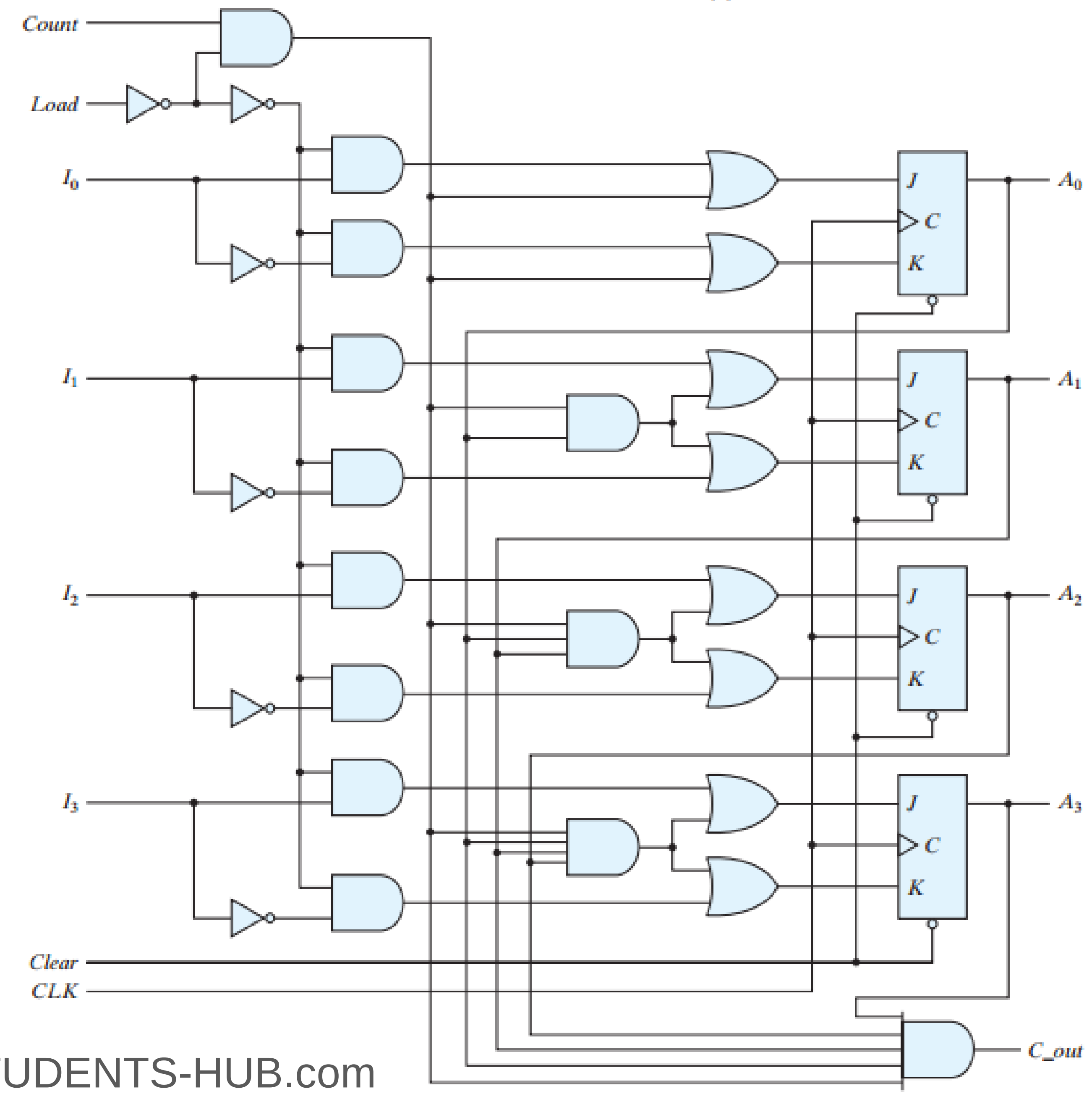
Notice: **Clear** has the **highest** priority

Binary Counter with Parallel Load



Circuit Diagram
4-bit binary counter with parallel load

Binary Counter with Parallel Load



Circuit Diagram
4-bit binary counter with parallel load

Other Counters

- Counters can be designed to generate **any** desired **sequence** of states
- The sequence may follow the **binary** count or may be any other **arbitrary** sequence
- Counters are used to generate **timing signals** to control the sequence of operations in a digital system
- Counters can also be constructed **by** means of **shift registers**
- We will briefly discuss:
  - Counter with **Unused** States
  - **Ring** Counter
  - **Johnson** Counter

Counter with Unused States

Example: Design a counter that repeats (0,1,2,4,5,6) using **JK** flip-flops

2 Unused States (**011,111**). These shall be considered as **Don't Care** in the state table

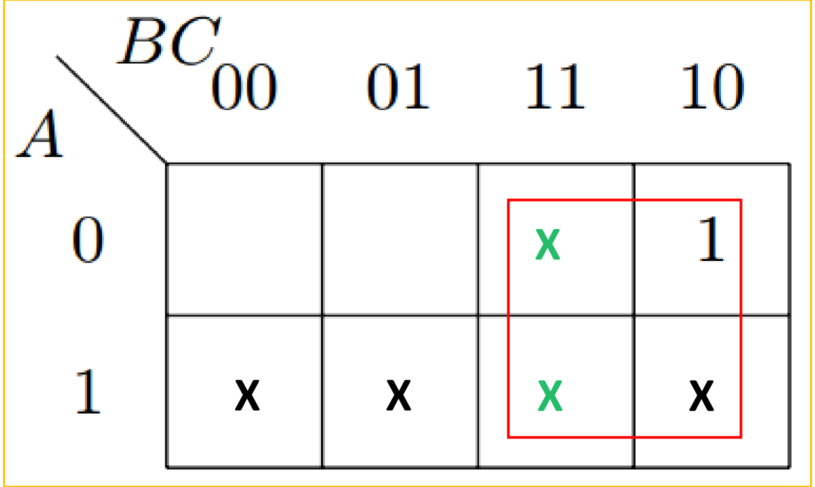
Present State			Next State			Flip-Flop Inputs					
A	B	C	A	B	C	J_A	K_A	J_B	K_B	J_C	K_C
0	0	0	0	0	1	0	X	0	X	1	X
0	0	1	0	1	0	0	X	1	X	X	1
0	1	0	1	0	0	1	X	X	1	0	X
1	0	0	1	0	1	X	0	0	X	1	X
1	0	1	1	1	0	X	0	1	X	X	1
1	1	0	0	0	0	X	1	X	1	0	X

Counter State Table

$$J_A = B \qquad K_A = B$$
$$J_B = C \qquad K_B = 1$$
$$J_C = B' \qquad K_C = 1$$

Simplified JK equations (using k-map)

K-map for J_A (as an example)



X: Original from JK Excitation Table
X: From Don't Care of States: 3,7

Example Continue:

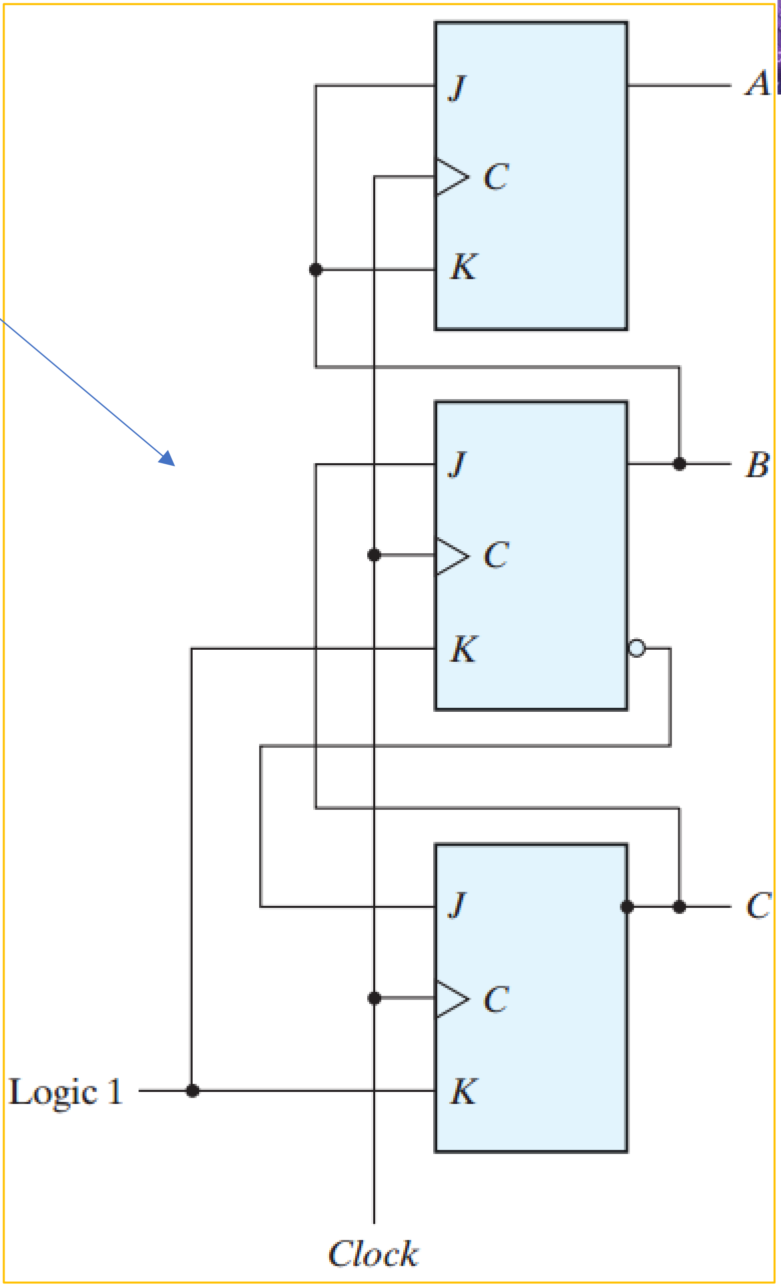
Since there are **two** unused states, we **analyze** the **circuit diagram** to determine their **effect**:

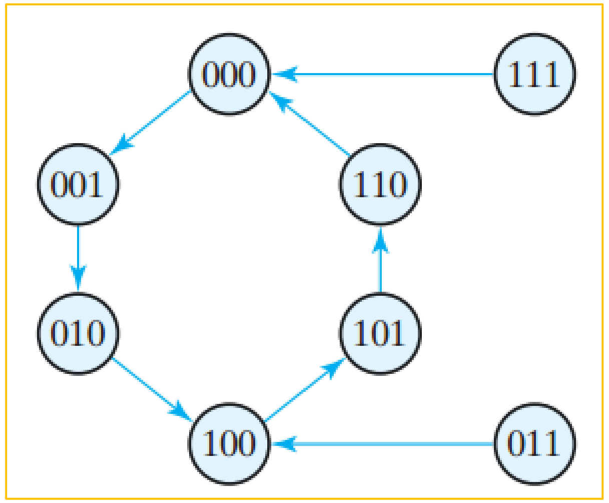
1. If the circuit happens to be in state **011** because of an error signal, the circuit goes to state **100** after the application of a clock pulse
   a) (B=1 → $J_A = K_A = 1$, $J_C = 0$) → A will be complemented(1→0), C=0
   b) (C=1 → $J_B = 1$) → B will be complemented (0→1)

1. If the circuit happens to be in state **111** because of an error signal, the circuit goes to state **000** after the application of a clock pulse
   a) Same as above

Self-Correcting Counter

Counter Circuit Diagram





Counter State Diagram

Mohammed Khalil

Counter with Unused States

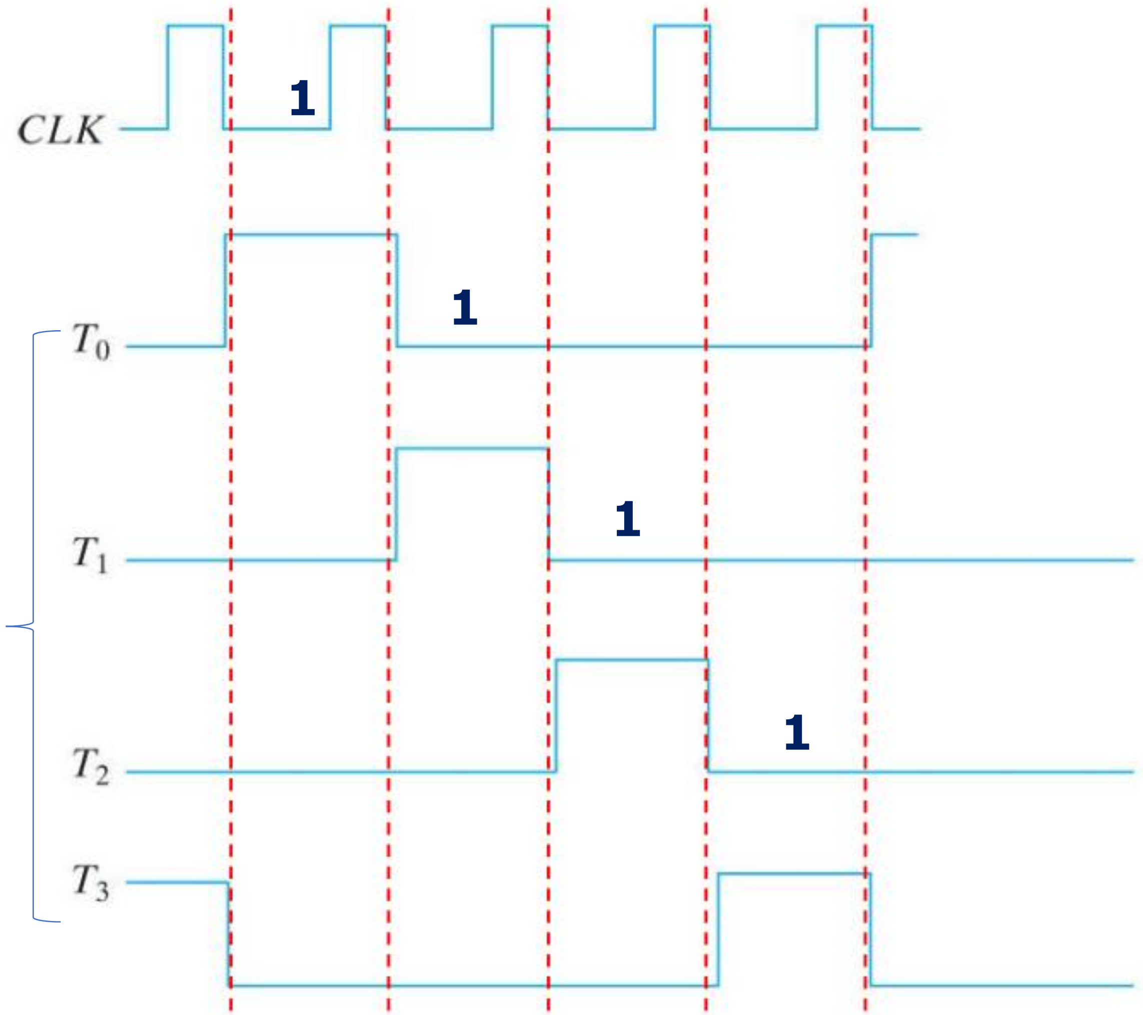
- As shown in the previous example, if the circuit ever goes to one of the **unused** states because of an error, the **next count pulse** **transfers** **it to one of the** **valid** **states** and the circuit **continues** to count **correctly**. This kind of counters is called **Self-Correcting Counter**

- In a **self-correcting counter**, if the counter happens to be in one of the **unused** states, it **eventually** reaches the **normal** count **sequence** after **one or more** clock **pulses**

- An alternative design could use **additional** logic to **direct/enforce** **every unused** state to a **specific** **next** state

Ring Counters

- In digital systems, we sometimes need a control signal that will **trigger** every **2^n-1** cycles
- Such circuit is called a **ring counter**
- A **ring counter** is a circular shift register with **only one** flip -flop being **set** at **any** particular **time**; **all** others are **cleared**
- The **single** bit (which =1) is **shifted** from one flip-flop to the next to produce the sequence of timing signals

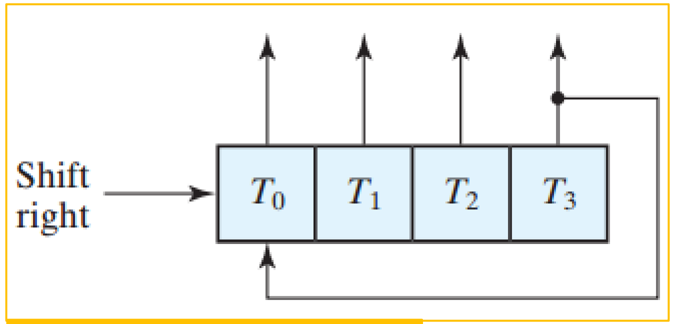


Four different timing signals generated using the **same** CLK

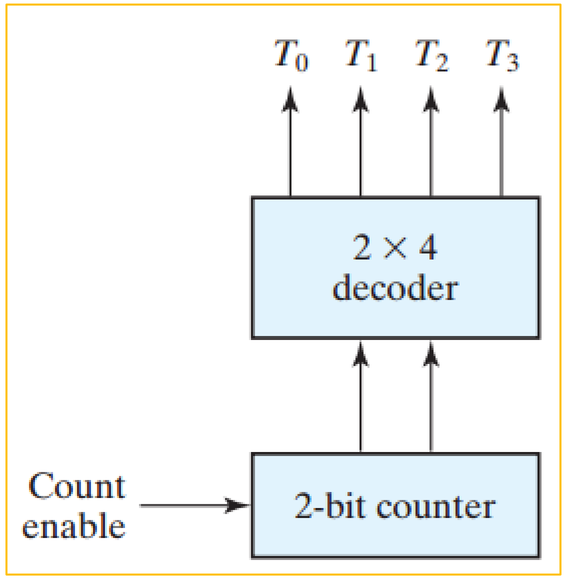
n=2, **Each** signal (T_0 → T_3) is **triggered** (set to 1) after every **3** cycles (2^2-1 = 3)

Ring Counters

- To generate **2^n** timing signals, we can use:
  1. A **shift register** with **2^n flip-flops**
  2. An **n-bit** binary **counter** together with an **n-to-2^n decoder**

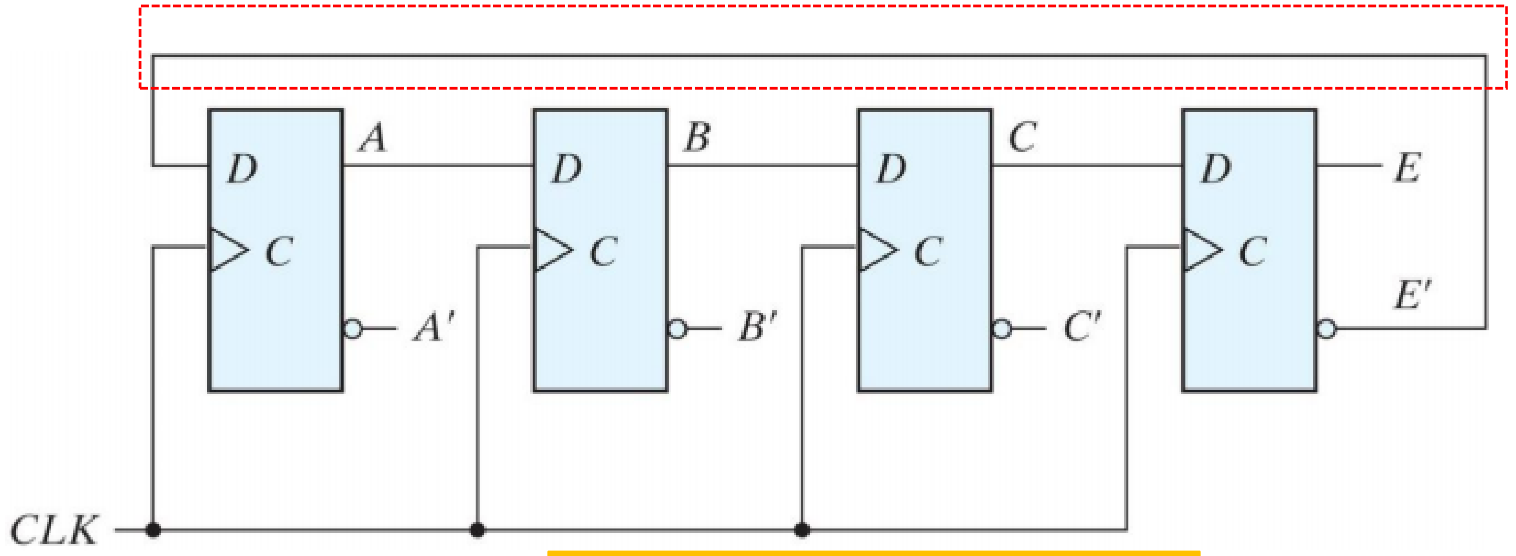


Initial Value : 1000



Johnson Counter

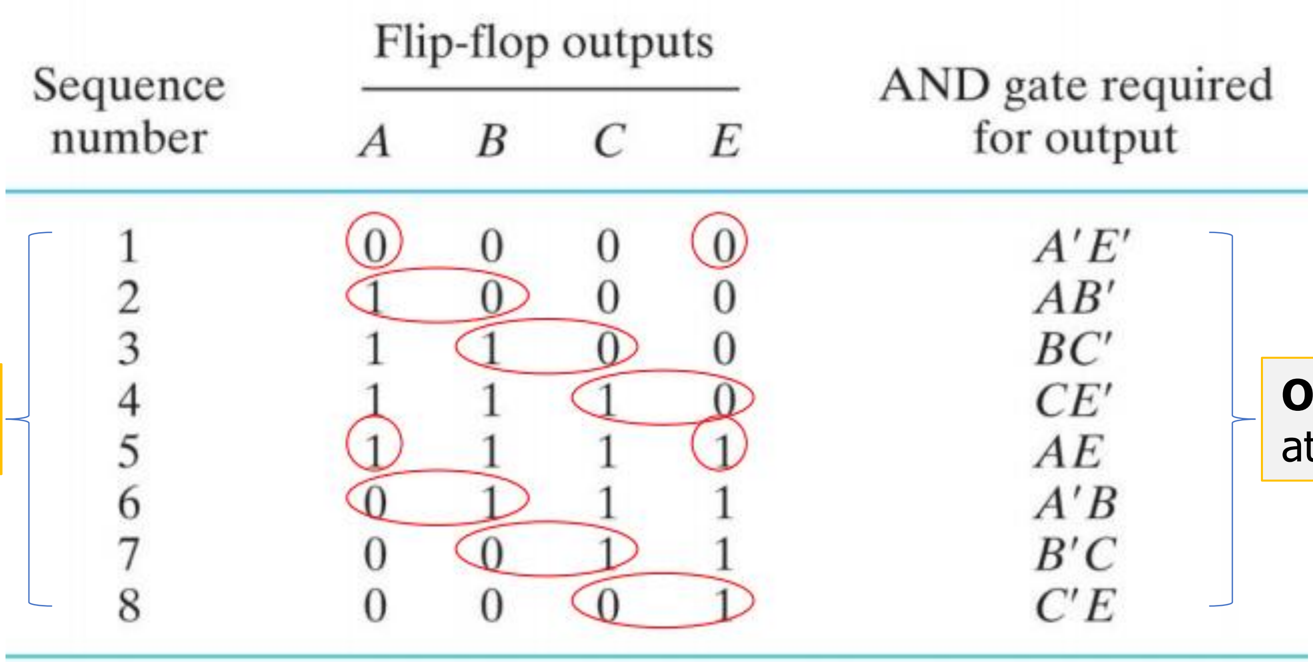
- **k-**bit **ring** counter **circulates** a **single** bit among the flip-flops to **provide k distinguishable states**
- The **number** of states can be **doubled** if the shift **register** is **connected** as a **switch-tail ring counter**
- A **switch-tail** ring counter is a **circular** shift register with the **complemented output** of the **last** flip-flop **connected** to the **input** of the **first** flip-flop



4 stages switch-tail ring counter

Johnson Counter

- A **Johnson** counter is a **k-bit** switch-tail ring counter with **2k** decoding gates to **provide** outputs for **2k timing signals**
- The **eight AND** gates listed in the table below, when **connected** to the circuit, will **complete** the construction of the Johnson counter
- **Each gate** is **enabled** during **one particular state sequence**
  - The **outputs** of the gates **generate eight timing** signals in succession



Sequence number	Flip-flop outputs A	B	C	E	AND gate required for output
1	0	0	0	0	$A'E'$
2	1	0	0	0	AB'
3	1	1	0	0	BC'
4	1	1	1	0	CE'
5	1	1	1	1	AE
6	0	1	1	1	$A'B$
7	0	0	1	1	$B'C$
8	0	0	0	1	$C'E$

K=4 → 2K = 8 → 8 States/Timing Signals

Only one gate will produce 1 at any given state

4-bit Johnson Counter

Johnson Counter

- One **disadvantage** of Johnson counter is that if it finds itself in an unused state, it will keep in moving from one invalid state to another and **never find its way to a valid state** (**NOT Self-Correcting**)

- This can be corrected by modifying the circuit to avoid this undesirable condition

- One possible **correction**: $\mathbf{D_C = (A + C)B}$ (instead of $D_C = B$)

- Johnson counters can be constructed for any number of timing sequences.
  1. The number of **flip-flops** needed is **one-half** the number of **timing** signals.
  2. The number of **decoding gates** is **equal** to the number of **timing** signals, and only **two-input** gates are needed.