Digital Logic Design

Lecture 5

Sequential logic design

The outputs of **sequential circuits** are dependent on not only their current inputs (as in combinational circuits), but also on all their past inputs.

→ sequential circuits must contain **memory elements**

Memory elements are made up of the same basic logic gates as combinational circuits.

In order for a circuit to “remember” its current value, we have to connect the output of a logic gate directly or indirectly back to the input of that same gate.

This is called a **feedback loop circuit**.

Latches and flip-flops

Latches and **flip-flops** are the basic memory elements for storing information.

They are the fundamental building blocks for all sequential circuits.

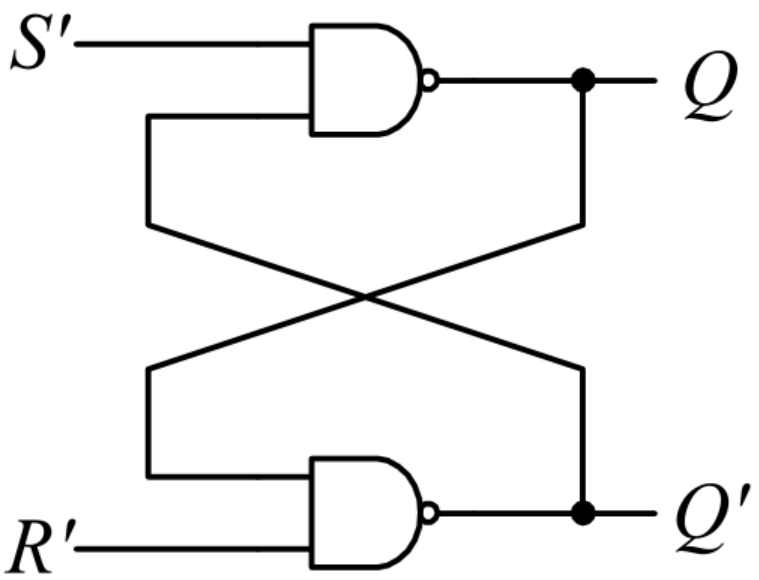
A latch or flip-flop can store only one bit of information which is referred to as the **state** of the latch or flip-flop.

A latch or flip-flop can be in one of two states: 0 or 1.

A latch or a flip-flop changes state when its content changes from a 0 to a 1 or from 1 to 0.

This state value is always available at the output.

SR Latch



Two states:

A **set state**: Q = 1

A **reset state**: Q = 0

The primes in S and R denote that they are **active-low**:

0 asserts them

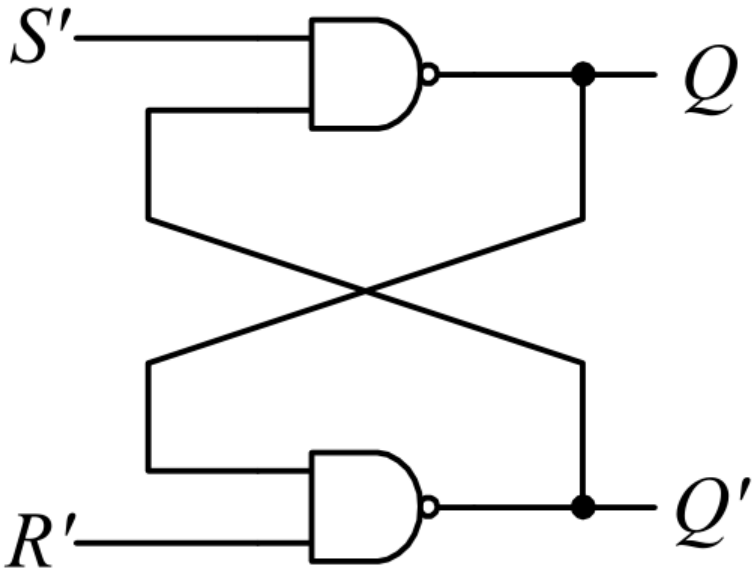
1 de-asserts them

S'	R'	Q	Q_{next}	Q_{next}'
0	0	×	1	1
0	1	×	1	0
1	0	×	0	1
1	1	0	0	1
1	1	1	1	0

To go to the set state:

assert S' by setting it to 0 and de-assert R' by setting it to 1

0 NAND anything = 1, => Q = 1, and the latch is set



If we now de-assert S' so that S' = R' = 1, the latch will remain in the set state because Q' = 0, which will keep Q = 1.

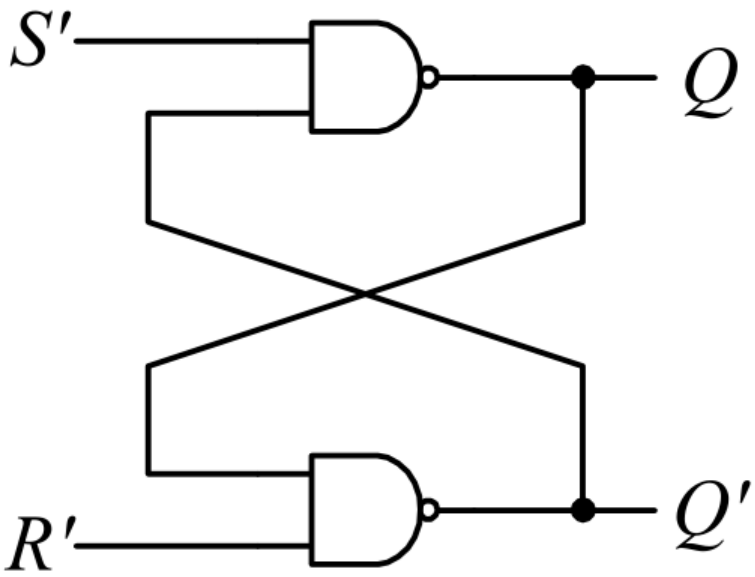
Reset the latch by making R' = 0 and S' = 1

S'	R'	Q	Q_{next}	Q_{next}'
0	0	×	1	1
0	1	×	1	0
1	0	×	0	1
1	1	0	0	1
1	1	1	1	0

With R' being a 0, Q' will go to a 1

At the top NAND gate, 1 NAND 1 = 0, thus forcing Q to go to 0

If we de-assert R' next so that, again, we have S' = R' = 1, this time the latch will remain in the reset state.



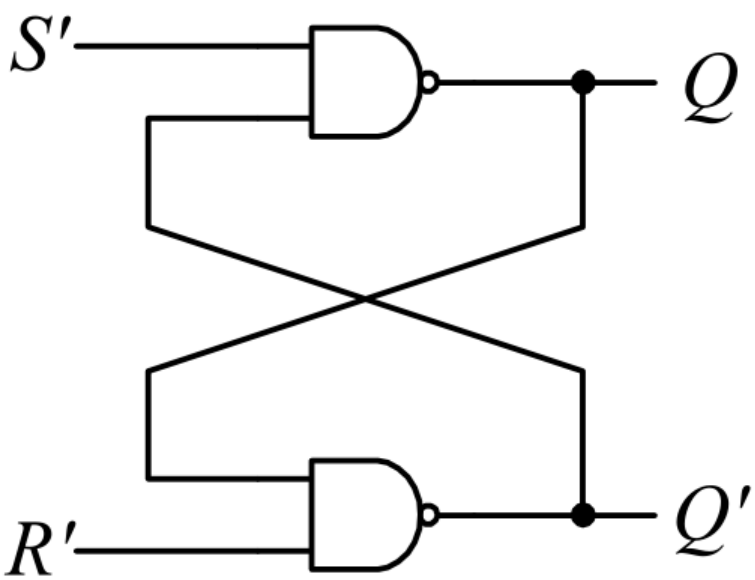
Notice the two cases when both inputs S' and R' are de-asserted (i.e., S' = R' = 1) but these inputs give different outputs.

The difference is in the value of Q immediately before those times.

S'	R'	Q	Q_{next}	Q_{next}'
0	0	×	1	1
0	1	×	1	0
1	0	×	0	1
1	1	0	0	1
1	1	1	1	0

When both inputs are de-asserted, the SR latch remembers its previous state.

SR Latch problem: undefined state

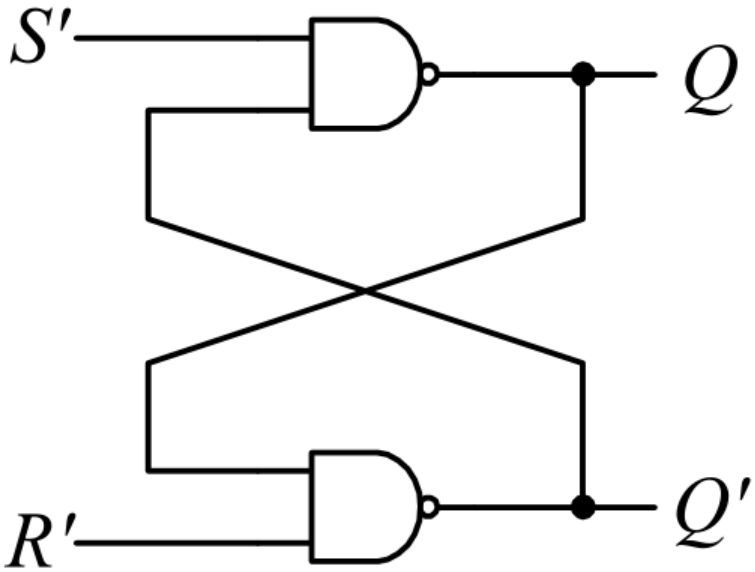


S'	R'	Q	Q_{next}	Q_{next}'
0	0	×	1	1
0	1	×	1	0
1	0	×	0	1
1	1	0	0	1
1	1	1	1	0

If both S' and R' are asserted (i.e., S' = R' = 0), then both Q and Q' are equal to a 1, since 0 NAND anything gives a 1.

If one of the input signals is de-asserted earlier than the other, the latch will end up in the state forced by the signal that is de-asserted later.

A problem exists if both S' and R' are de-asserted at exactly the same time.

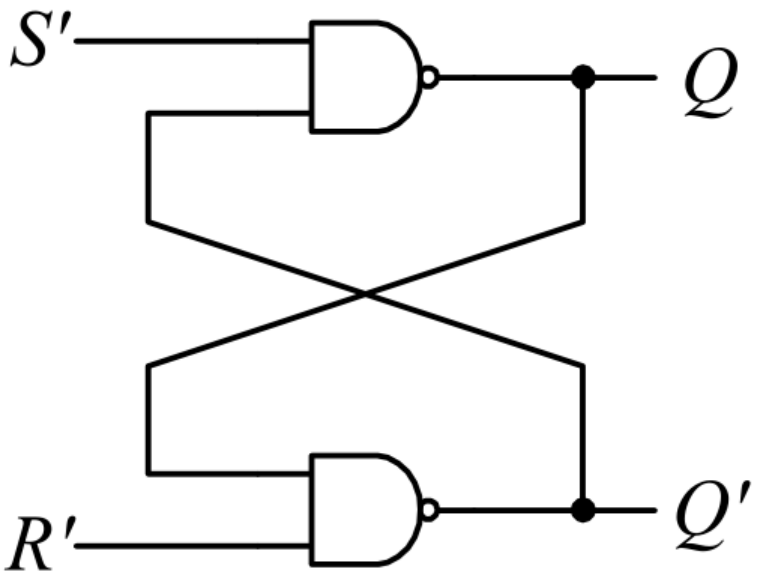


Assume for a moment that both gates have exactly the same delay and that the two wire connections between the output of one gate to the input of the other gate also have exactly the same delay.

Currently, both Q and Q' are at a 1.

If we set S' and R' to a 1 at exactly the same time, then both NAND gates will perform a 1 NAND 1 and will both output a 0 at exactly the same time.

The two 0's will be fed back to the two gate inputs at exactly the same time, because the two wire connections have the same delay.



This time, the two NAND gates will perform a 1 NAND 0 and will both produce a 1 again at exactly the same time.

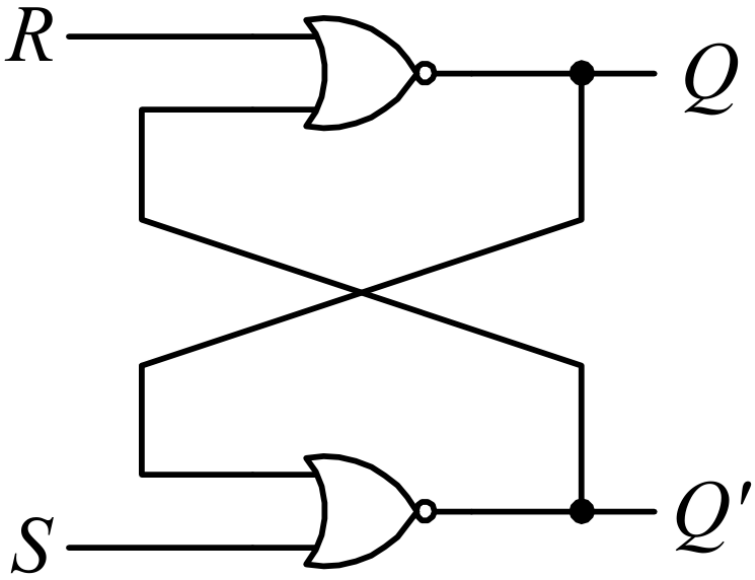
This time, two 1's will be fed back to the inputs, which again will produce a 0 at the outputs, and so on and on.

This oscillating behavior, called the **critical race**, will continue indefinitely until one outpaces the other.

If the two gates do not have exactly the same delay then, the situation is similar to de-asserting one input before the other, and so, the latch will go into one state or the other.

Since we do not know which is the faster gate, therefore, we do not know which state the latch will end up in. Thus, the latch's next state is undefined.

SR latch implemented using NOR gates



S	R	Q	Q_{next}	Q_{next}'
0	0	0	0	1
0	0	1	1	0
0	1	×	0	1
1	0	×	1	0
1	1	×	0	0

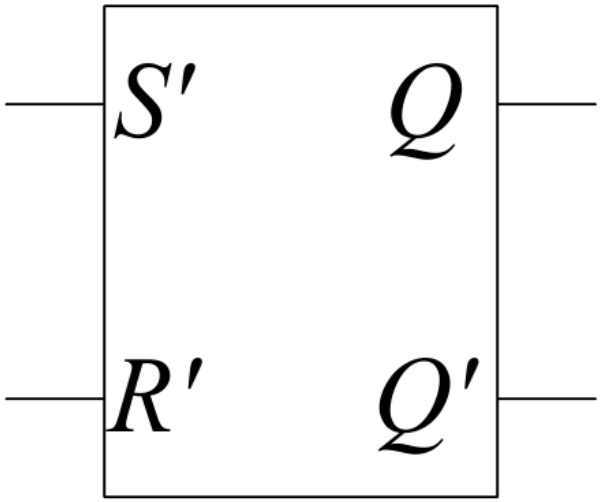
The S and R inputs are active-high, so that setting S to 1 will set the latch, and setting R to 1 will reset the latch.

The latch is set when Q = 1, and reset when Q = 0.

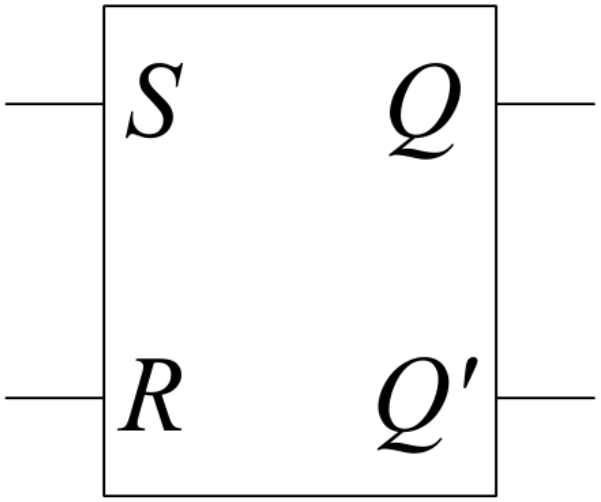
The latch remembers its previous state when S = R = 0.

When S = R = 1, both Q and Q' are 0.

Logic symbols for the SR latches



NAND implementation

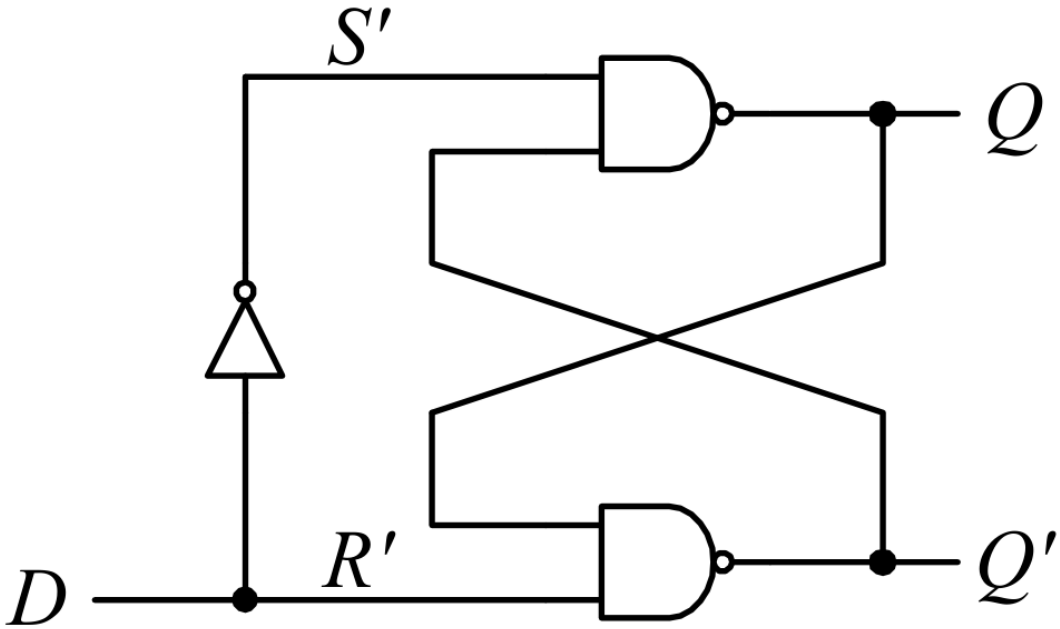


NOR implementation

D latch

We can guarantee that S and R, are never de-asserted at exactly the same time by not having both of them asserted.

This situation is prevented in the D latch by adding an inverter between the original S' and R' inputs.



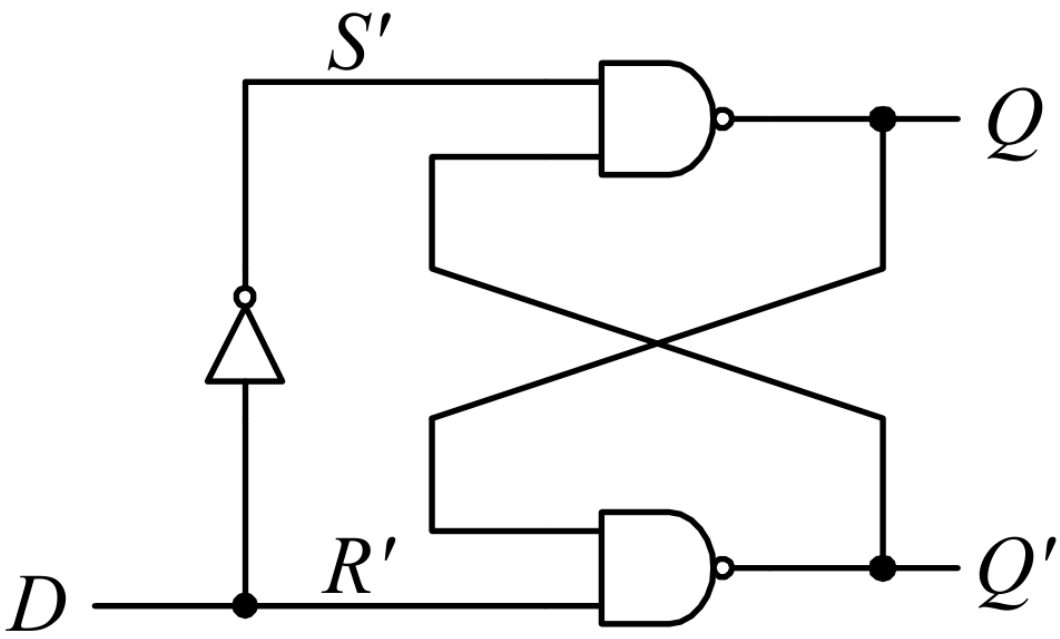
This way, S' and R' will always be inverses of each other, and so, they will never be asserted together.

D latch

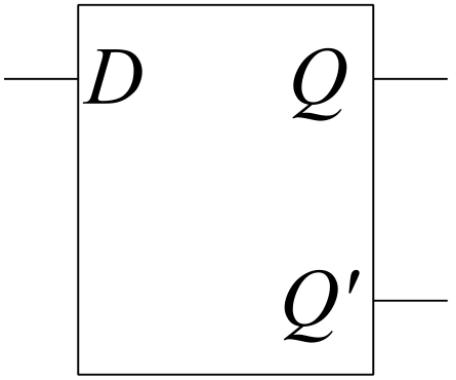
When D (data) = 0, then S' = 1 and R' = 0, so this is similar to resetting the SR latch by making Q = 0.

When D = 1, then S' = 0 and R' = 1, and Q will be set to 1.

Q_{next} always gets the same value as the input D, and is independent of the current value of Q.



D	Q	Q_{next}	Q_{next}'
0	×	0	1
1	×	1	0

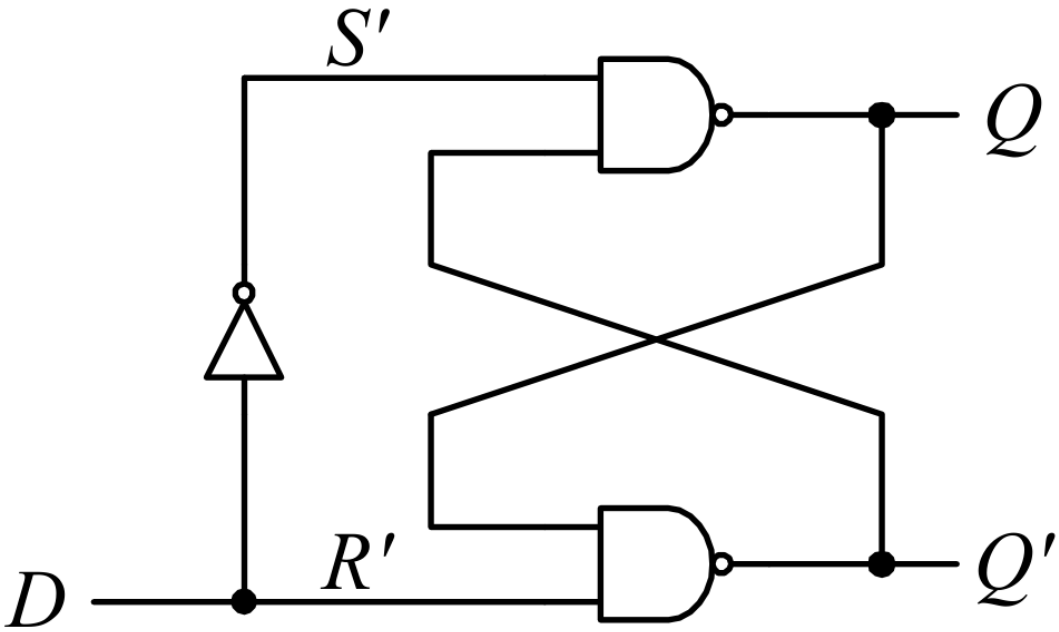


D latch

In this circuit, we have lost the ability to set S' = R' = 1 in order for the circuit to remember its current value.

Q_{next} cannot remember the current value of Q, instead it will always follow D.

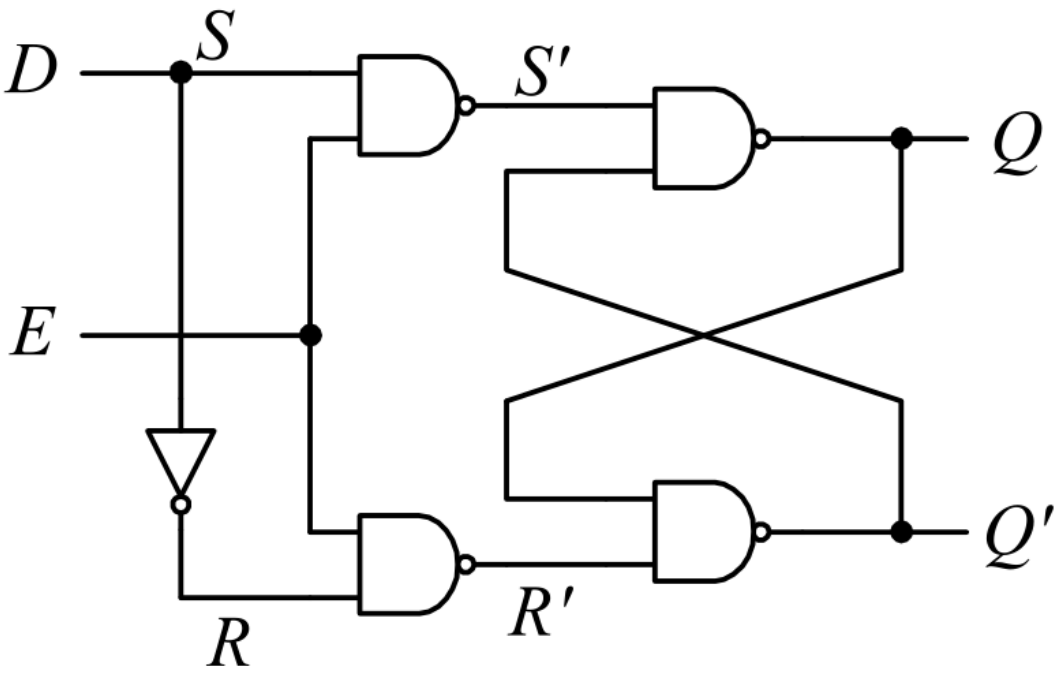
The end result is like having a piece of wire where the output is the same as the input!

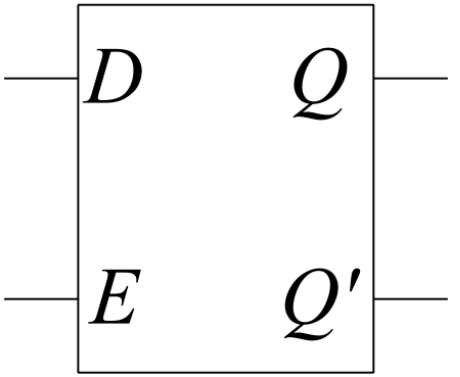


D	Q	Q_{next}	Q_{next}'
0	×	0	1
1	×	1	0

D Latch with enable

Also known as a **gated D latch**





When the enable signal E is asserted (E = 1), Q_{next} follows the D input.

E	D	Q	Q_{next}	Q_{next}'
0	×	0	0	1
0	×	1	1	0
1	0	×	0	1
1	1	×	1	0

When E is de-asserted (E = 0), Q_{next} retains its last value independent of the D input.

D flip-flop

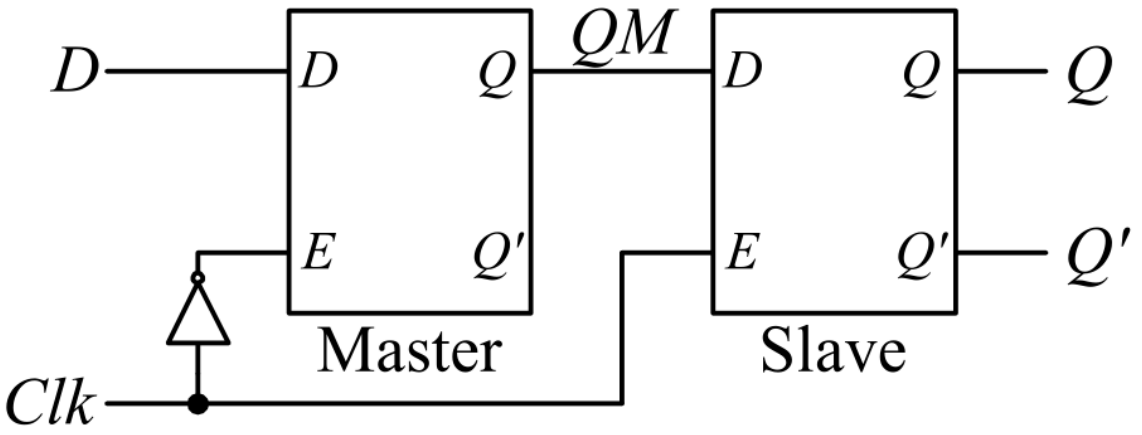
Latches are **level-sensitive** because their outputs are affected by their inputs as long as they are enabled.

Unlike the latch, a flip-flop is not level-sensitive, but rather **edge-triggered**.

Data gets stored into a flip-flop only at the active edge of the clock.

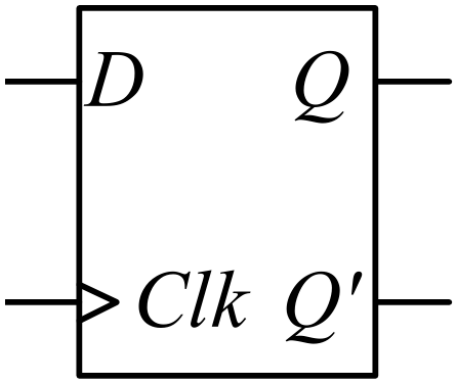
An **edge-triggered D flip-flop** achieves this by combining in series a pair of D latches.

Positive-edge-triggered D flip-flop



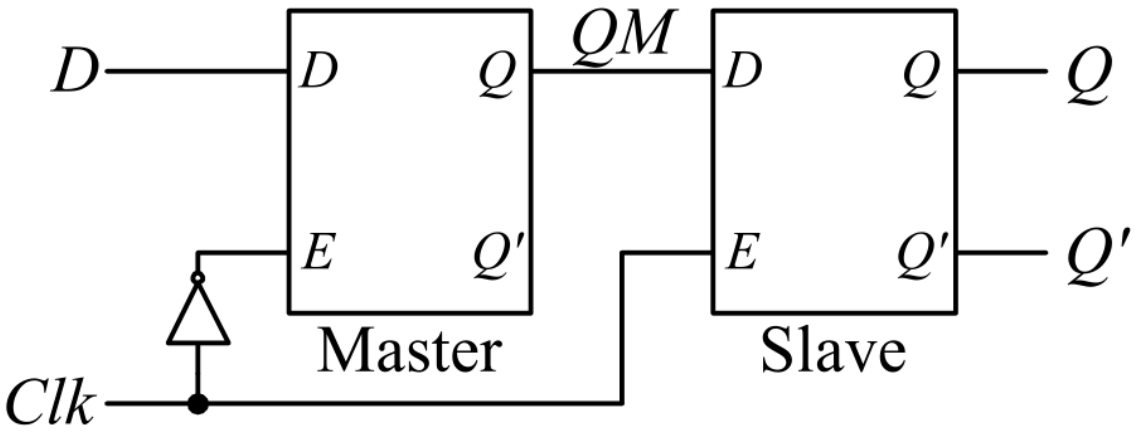
Clk – clock signal

Logic symbol



Operation table

Clk	*D*	*Q*	Q_{next}	Q_{next}'
0	×	0	0	1
0	×	1	1	0
1	×	0	0	1
1	×	1	1	0
↑	0	×	0	1
↑	1	×	1	0

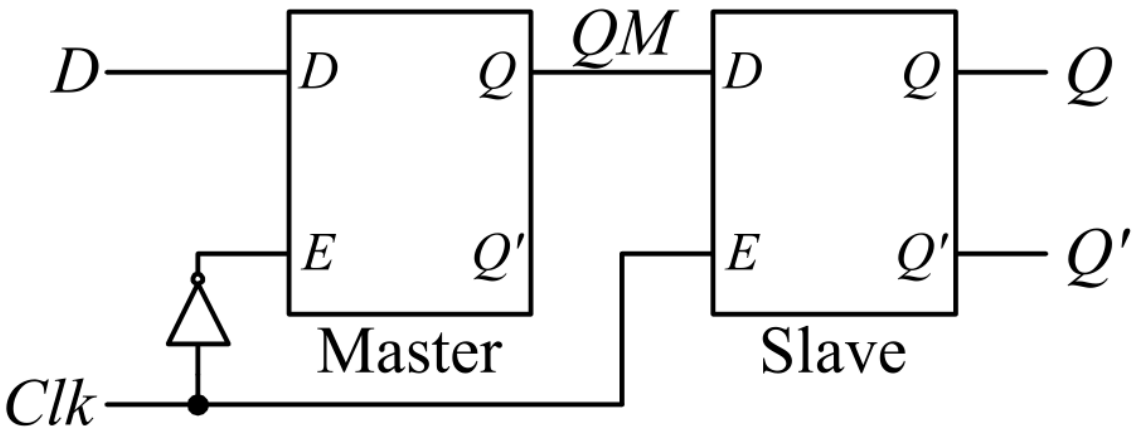


Clk	*D*	*Q*	Q_{next}	Q_{next}'
0	×	0	0	1
0	×	1	1	0
1	×	0	0	1
1	×	1	1	0
↑	0	×	0	1
↑	1	×	1	0

The master latch is enabled when Clk = 0, and so QM follows the primary input D.

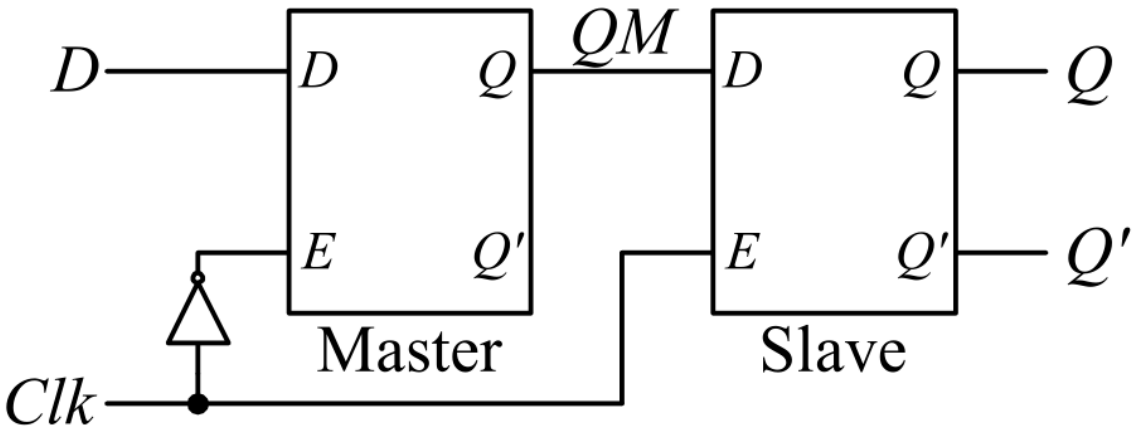
However, the signal at QM cannot pass over to Q, because the slave latch is disabled when Clk = 0.

When Clk = 1, the master latch is disabled, but the slave latch is enabled so that the output from the master latch, QM, is transferred to the primary output Q.



The slave latch is enabled all the while that Clk = 1, but its content changes only at the rising edge of the clock, because once Clk is 1, the master latch is disabled, and the input to the slave latch, QM, will be constant.

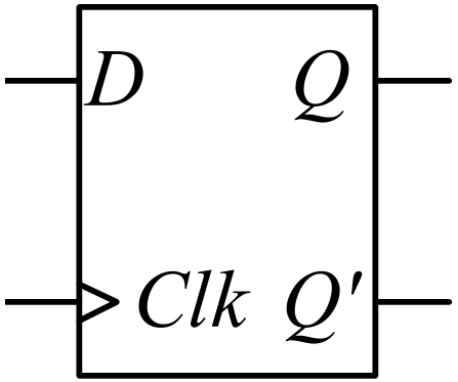
Therefore, when Clk = 1 and the slave latch is enabled, the primary output Q will not change because the input QM is not changing.



Positive-edge-triggered D flip-flop: the primary output Q on the slave latch changes only at the rising edge of the clock.

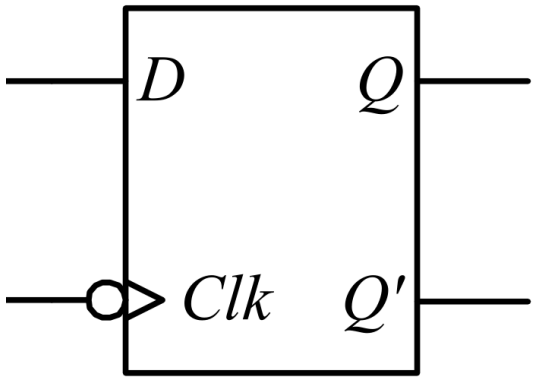
If the slave latch is enabled when the clock is low (i.e., with the inverter output connected to the E of the slave latch), then it is referred to as a **negative-edge-triggered flip-flop**

The circuit is also referred to as a **master-slave D flip-flop** because of the two D latches used in the circuit.



The small triangle at the clock input indicates that the circuit is triggered by the edge of the signal, and so it is a flip-flop.

Without the small triangle, the symbol would be that for a latch.

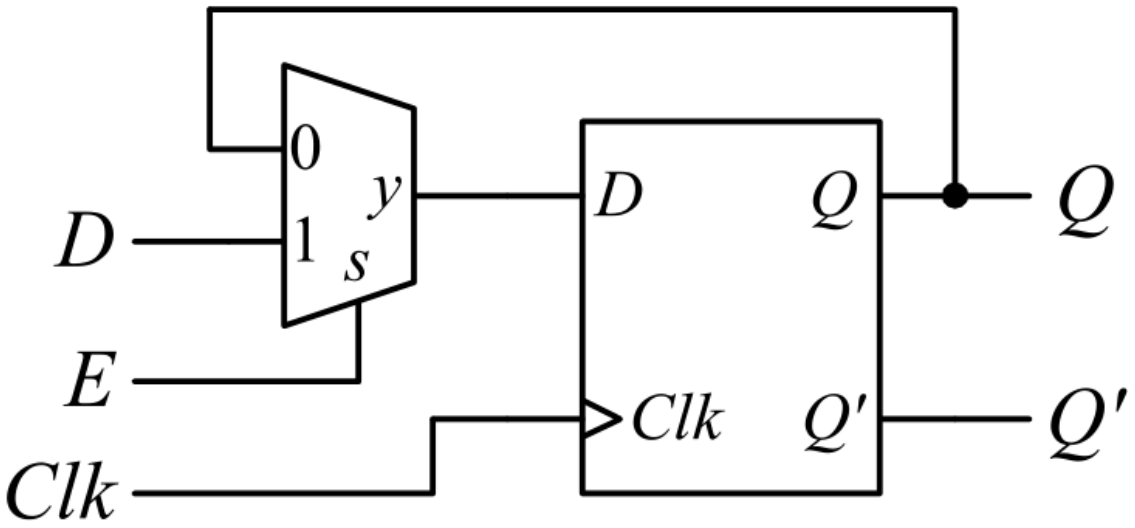


If there is a circle in front of the clock line, then the flip-flop is triggered by the falling edge of the clock, making it a negative-edge-triggered flip-flop.

D Flip-Flop with enable

At every active edge of the clock, the D flip-flop will load in a new value. It cannot remember its current content.

Solution: add an enable input, E, through a 2-input multiplexer

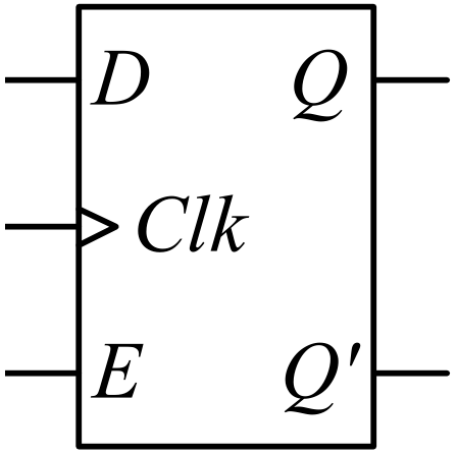


When E = 1, the primary input D signal will pass to the D input of the flip-flop, thus updating the content of the flip-flop at the active edge.

When E = 0, the current content of the flip-flop at Q is passed back to the D input of the flip-flop, thus keeping its current value.

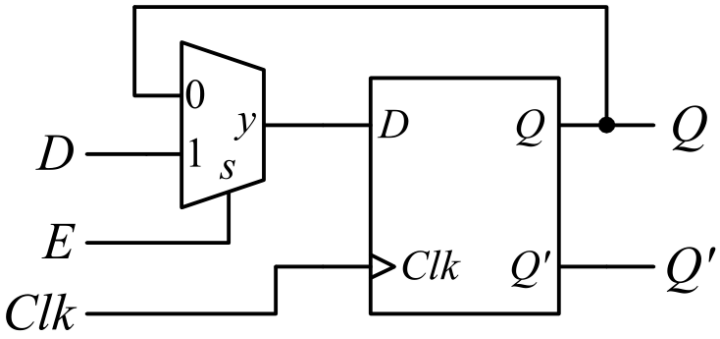
D Flip-Flop with enable

Logic symbol

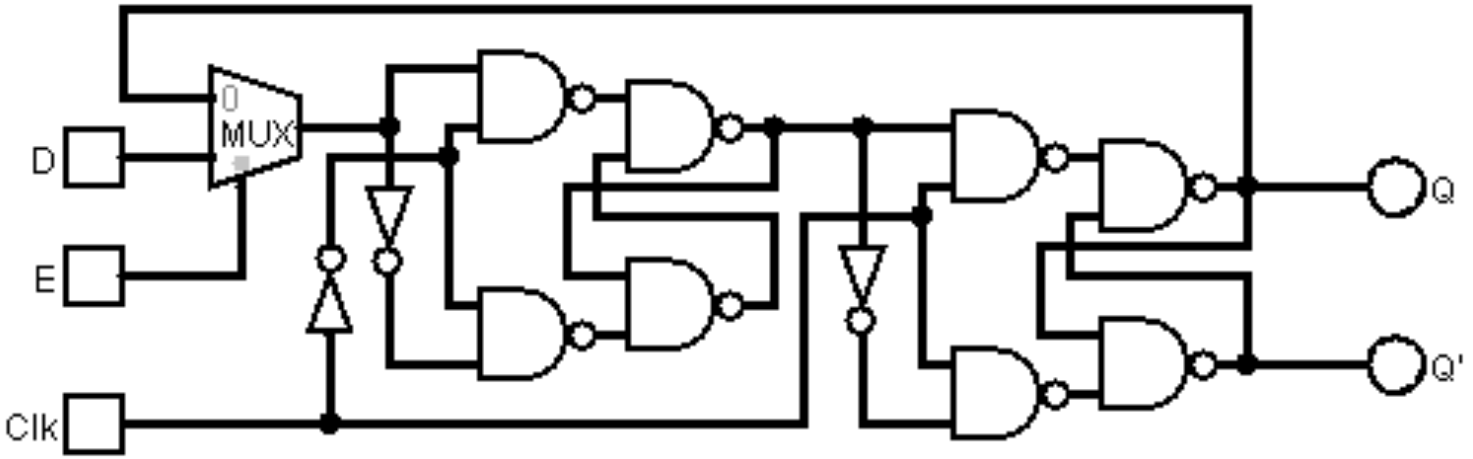


Operation table

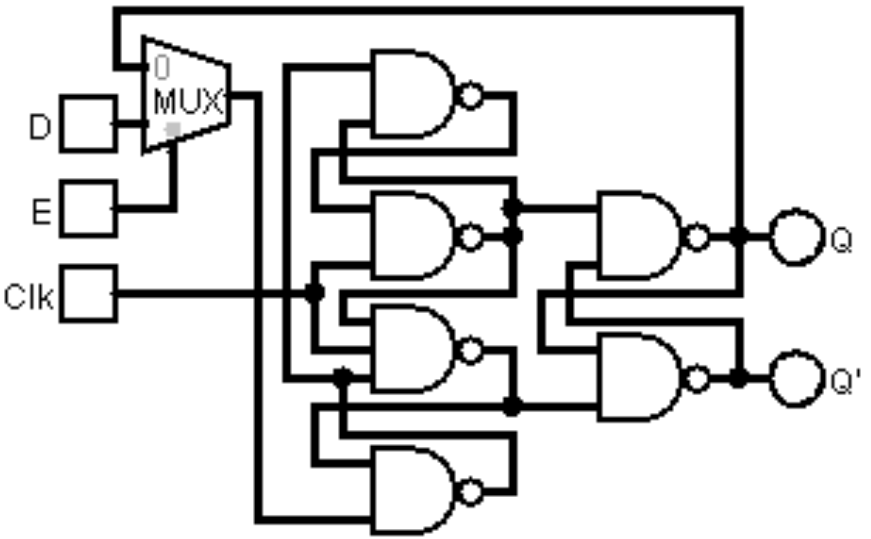
Clk	E	D	Q	Q_{next}	Q_{next}'
0	×	×	0	0	1
0	×	×	1	1	0
1	×	×	0	0	1
1	×	×	1	1	0
↑	0	×	0	0	1
↑	0	×	1	1	0
↑	1	0	×	0	1
↑	1	1	×	1	0



Naive implementation:



A smarter implementation:



Other flip-flop types

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

They differ in the number of inputs they have and how they change states.

Each type can have different variations, such as

– active-high or active-low inputs

– whether they change state at the rising or falling edge of the clock signal

– whether they have the enable input

– whether they have any asynchronous inputs

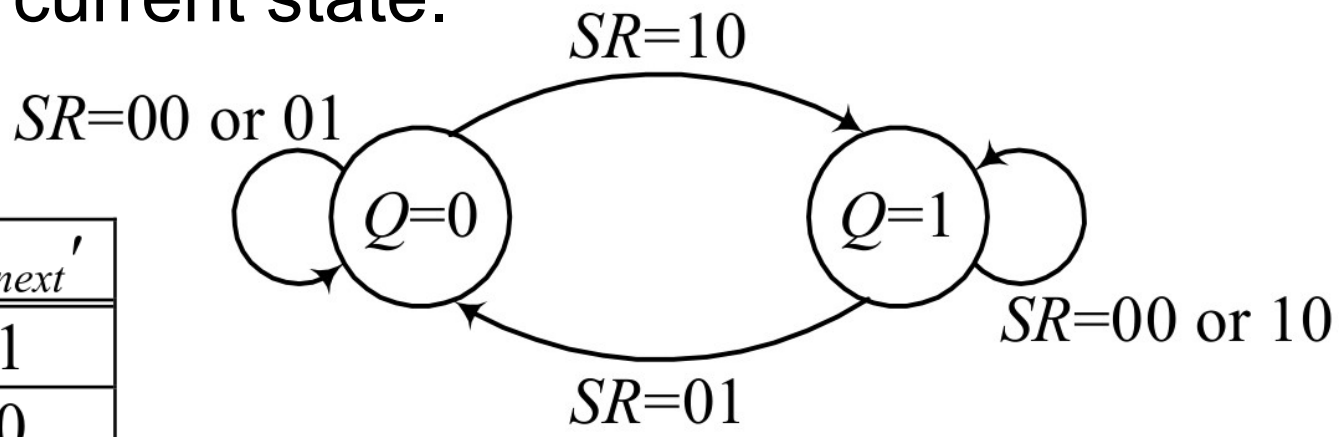
Any given sequential circuit can be built using any of these types of flip-flops or combinations of them.

SR flip-flop

Similar to SR latches, SR flip-flops can enter an undefined state when both inputs are asserted simultaneously.

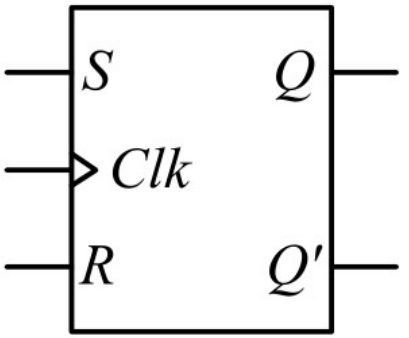
When the two inputs are de-asserted, then the next state is the same as the current state.

Active-high:



S	R	Q	Q_{next}	Q_{next}'
0	0	0	0	1
0	0	1	1	0
0	1	0	0	1
0	1	1	0	1
1	0	0	1	0
1	0	1	1	0
1	1	0	×	×
1	1	1	×	×

Q	Q_{next}	S	R
0	0	0	×
0	1	1	0
1	0	0	1
1	1	×	0

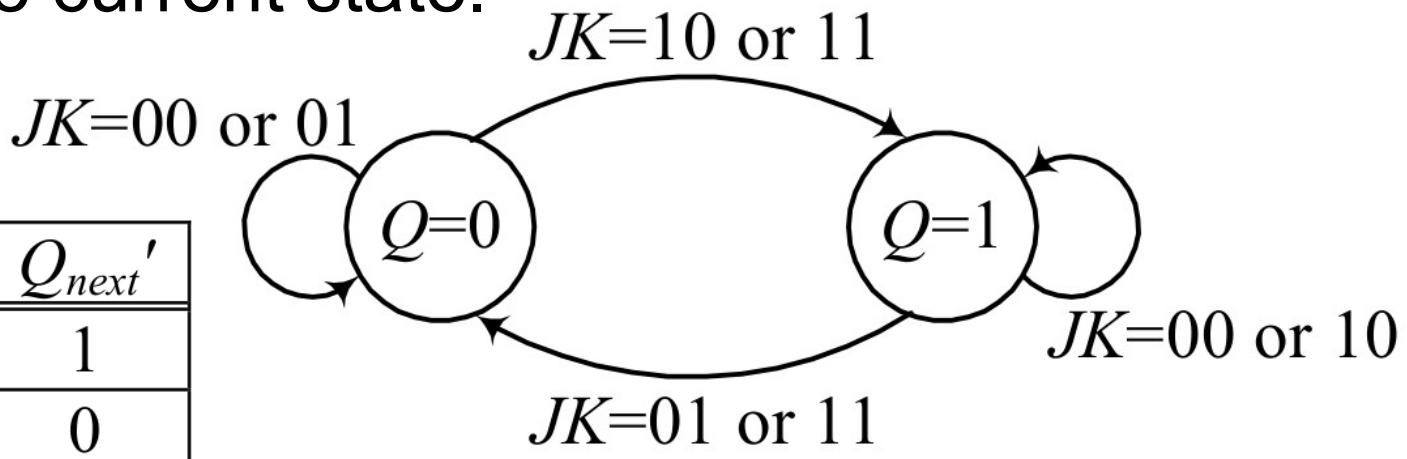


JK flip-flop

The J input sets the flip-flop when asserted.

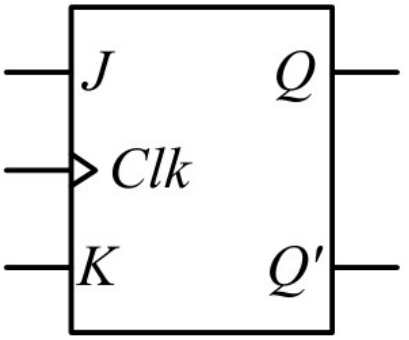
The K input resets the flip-flop when asserted.

When both inputs, J and K, are asserted, the next state is the inverse of the current state.



J	K	Q	Q_{next}	Q_{next}'
0	0	0	0	1
0	0	1	1	0
0	1	0	0	1
0	1	1	0	1
1	0	0	1	0
1	0	1	1	0
1	1	0	1	0
1	1	1	0	1

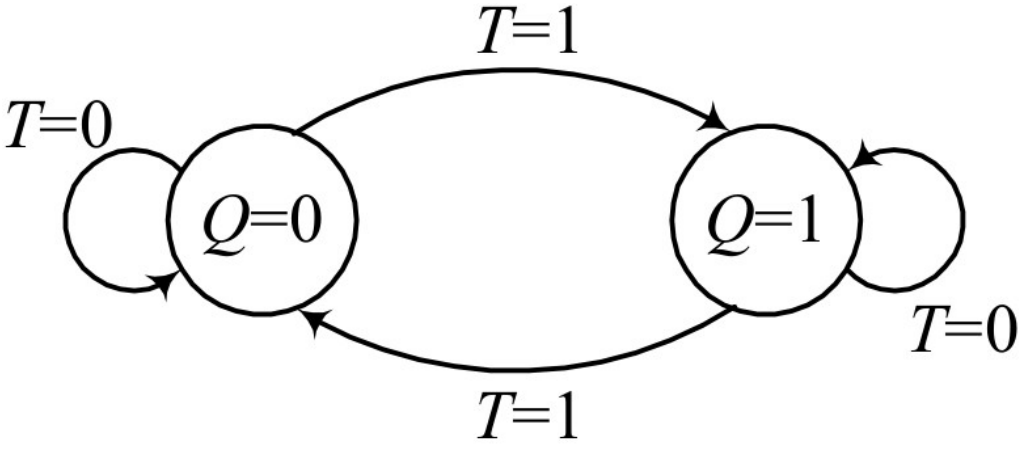
Q	Q_{next}	J	K
0	0	0	×
0	1	1	×
1	0	×	1
1	1	×	0



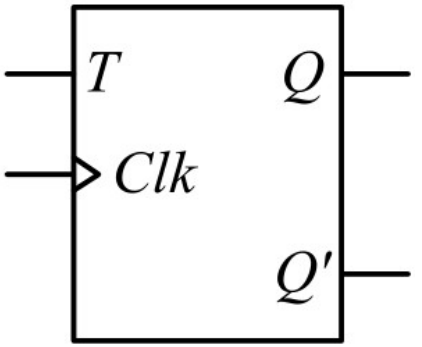
T flip-flop

When T is asserted (T = 1), the flip-flop state toggles back and forth at each active edge of the clock, and when T is de-asserted, the flip-flop keeps its current state.

T	Q	Q_{next}	Q_{next}'
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1



Q	Q_{next}	T
0	0	0
0	1	1
1	0	1
1	1	0



Synchronous and asynchronous inputs

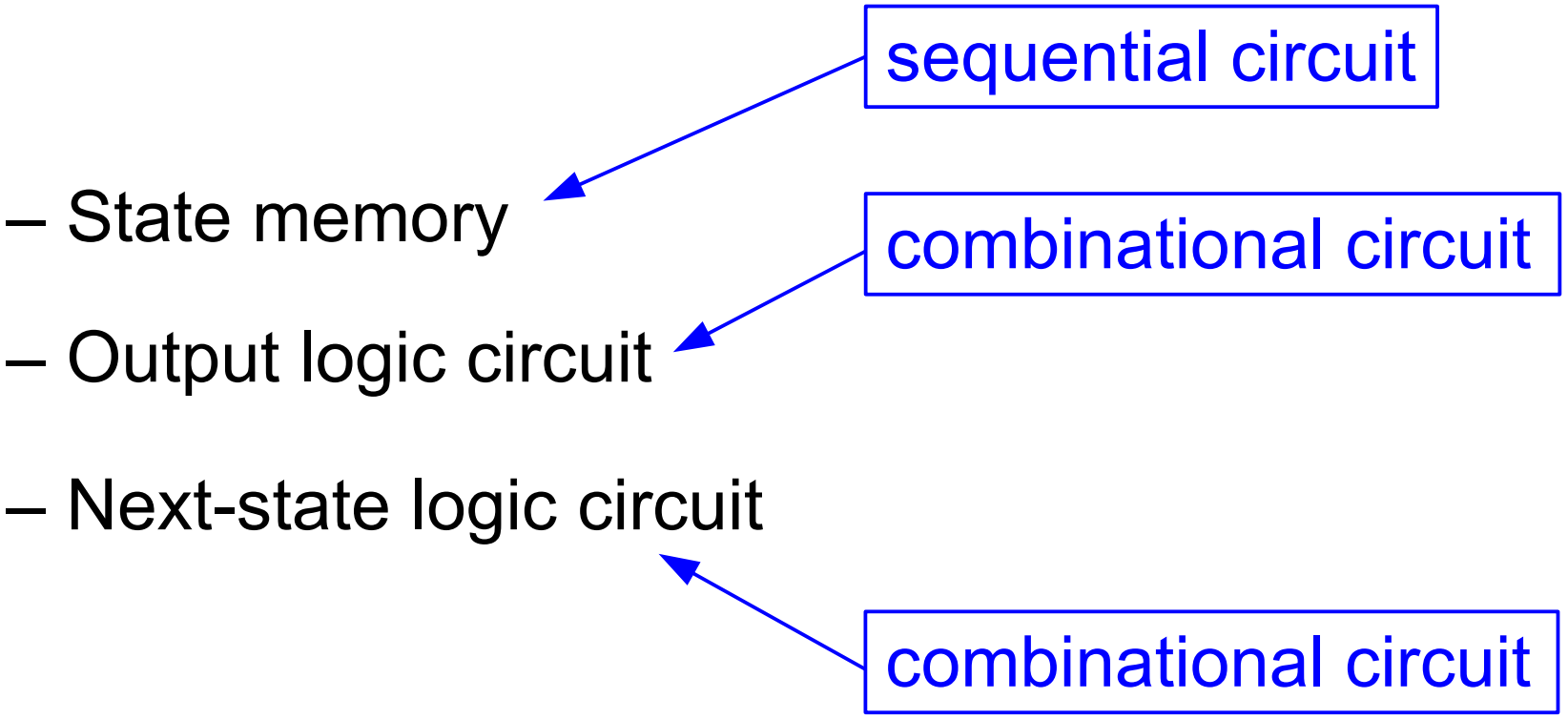
Sequential circuits that change states whenever a change in input values occurs that is independent of the clock are referred to as **asynchronous sequential circuits**

Synchronous sequential circuits change states only at the active edge of the clock signal.

Asynchronous inputs are available for both flip-flops and latches, and they are used to either set or clear the storage element's content that is independent of the clock.

Parts of sequential circuits

- State memory
- Output logic circuit
- Next-state logic circuit



The state and the state memory

A single flip-flop is capable of remembering only one bit of information or one bit of history.

To remember more inputs and a longer history, the circuit must contain more flip-flops.

The collection of (D) flip-flops used to remember the complete history of past inputs is referred to as the **state memory**.

The entire content of the state memory at a particular time forms a binary encoding that represents the complete history of inputs up to that time.

This binary encoding in the state memory at one particular instance of time is called the **state** of the system at that time.

Output logic circuit

The output signals from a sequential circuit are generated by the **output logic circuit**.

Since all the inputs are “remembered” in the form of states in the state memory, the outputs are dependent on the content of the state memory.

The output logic is a combinational circuit that is dependent on the content of the state memory, and may or may not be dependent on the current inputs.

The output signals that the output logic generates constitute the actions or operations that are performed by the sequential circuit.

A sequential circuit can perform different operations in different states by generating different output signals.

Next-state logic circuit

A sequential circuit operates by transitioning from one state to the next, generating different output signals.

The part inside a sequential circuit that is responsible for determining what next state to go to is called the **next-state logic circuit**.

Based on the current state that the system is in (i.e., the past inputs), and the current inputs, the next-state logic will determine what the next state should be.

The next-state logic is a combinational circuit that takes the contents of the state memory flip-flops and the current inputs as its inputs.

The outputs from the next-state logic are used to change the contents of the state memory flip-flops.

Finite-state machine

A sequential circuit is also known as a **finite-state machine (FSM)** because a circuit with k registers can be in one of a finite number = 2^k of unique states.

A sequential circuit is like a machine that operates by stepping through a sequence of states.

The control unit inside the microprocessor is a finite-state machine.

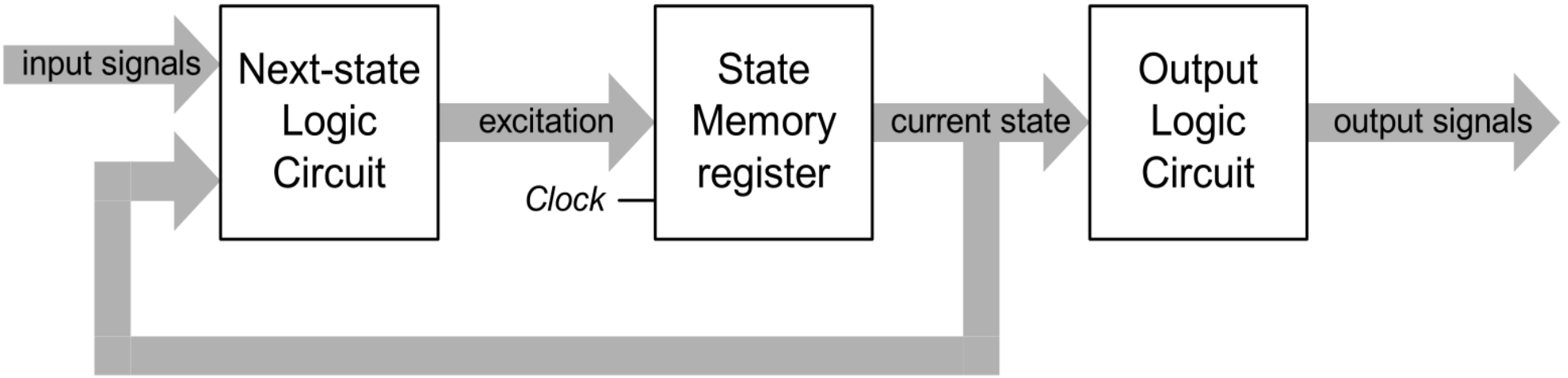
Finite-State-Machine (FSM) models

Two different FSM models:

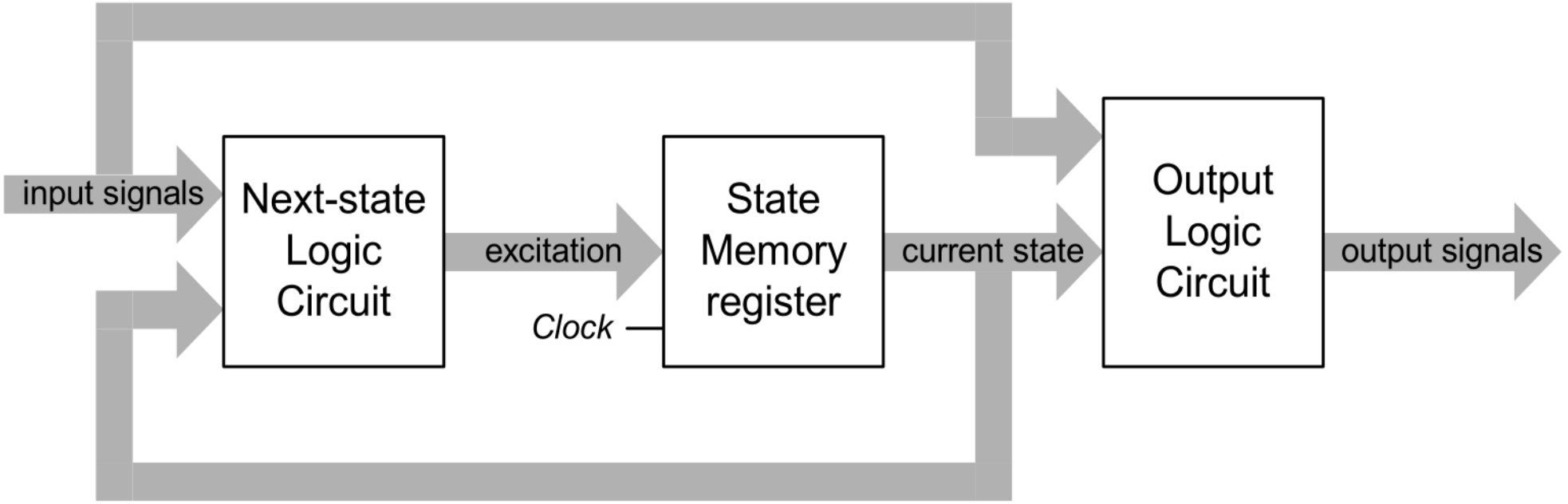
Moore FSM: outputs are dependent only on its current state, i.e. on the content of the state memory

Mealy FSM: outputs are dependent on both the current state of the machine and also the current inputs

Moore FSM



Mealy FSM



The **analysis of sequential circuits**:

given a sequential circuit, and we want to obtain a precise description of the operation of the circuit.

Because a sequential circuit consists of state memory, output logic, and next-state logic circuits, its analysis involves analysing these circuits.

The analysis of sequential circuits is done using

- excitation equations
- next-state equations
- next-state table
- output equations
- output table
- the state diagram

Excitation equations

The excitation equations are the equations for the next-state logic circuit in the FSM.

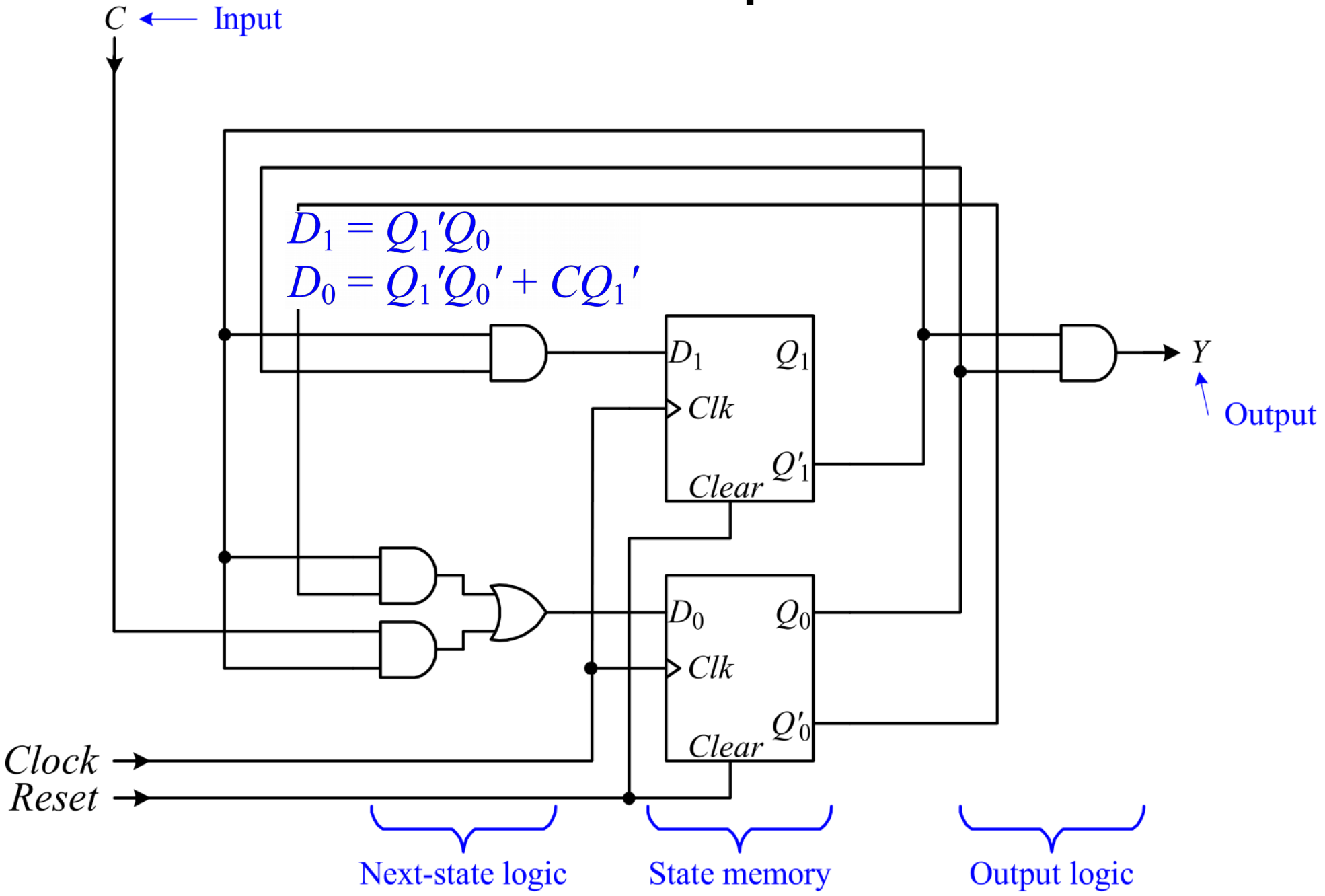
They are the input equations to the state memory flip-flops in the FSM.

Since the next-state logic is a combinational circuit, therefore, deriving the excitation equations is just an analysis of a combinational circuit.

The next-state logic circuit that is derived by these equations “excites” the flip-flops by causing them to change states, hence the name “excitation equation”.

There is one equation for each flip-flop’s input.

Excitation equations



Next-state equation

The next-state equations specify what the flip-flops' next state is going to be depending on two things:

1) the inputs to the flip-flops, and

2) the functional behavior of the flip-flops

The inputs to the flip-flops are provided by the excitation equations.

The functional behavior of a flip-flop is described by its **characteristic equation**.

To derive the next-state equations, we substitute the excitation equations into the corresponding flip-flop's characteristic equations.

Next-state equation

Example: the characteristic equation for the D flip-flop is

$$Q_{next} = D$$

Substituting the two excitation equations into the characteristic equation for the D flip-flop will give us the following next-state equations

$$Q_{1next} = D_1 = Q_1'Q_0$$
$$Q_{0next} = D_0 = Q_1'Q_0' + CQ_1'$$

Next-state table

The **next-state table** is the truth table as derived from the next-state equations.

It lists for every combination of the current state (the Q) values and input values, what the next state (the Q_{next}) values should be.

Example:

$$Q_{1next} = Q_1'Q_0$$
$$Q_{0next} = Q_1'Q_0' + CQ_1'$$

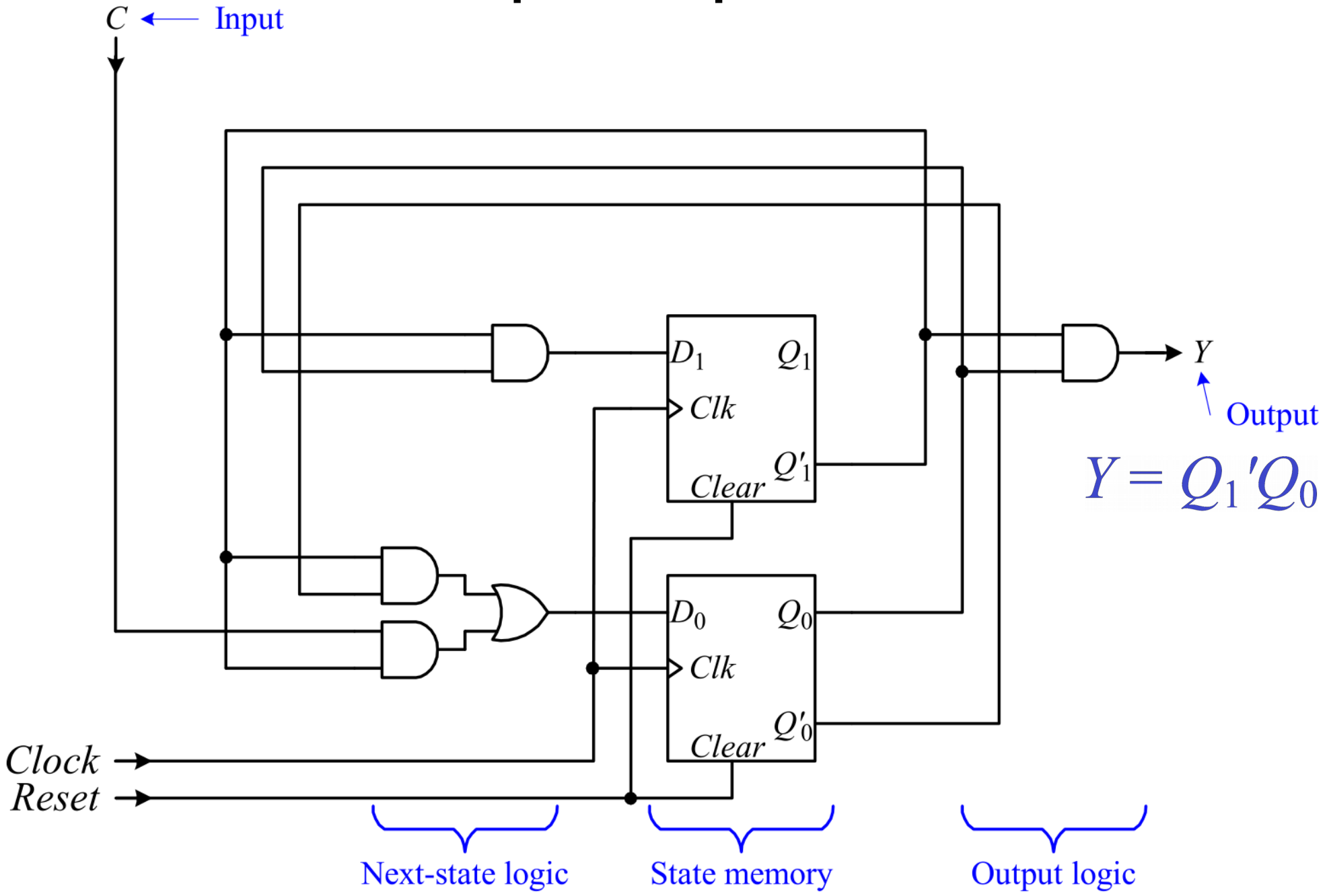
Current State Q_1Q_0	Next State $Q_{1next}\,Q_{0next}$	
	$C = 0$	$C = 1$
00	01	01
01	10	11
10	00	00
11	00	00

Output equation

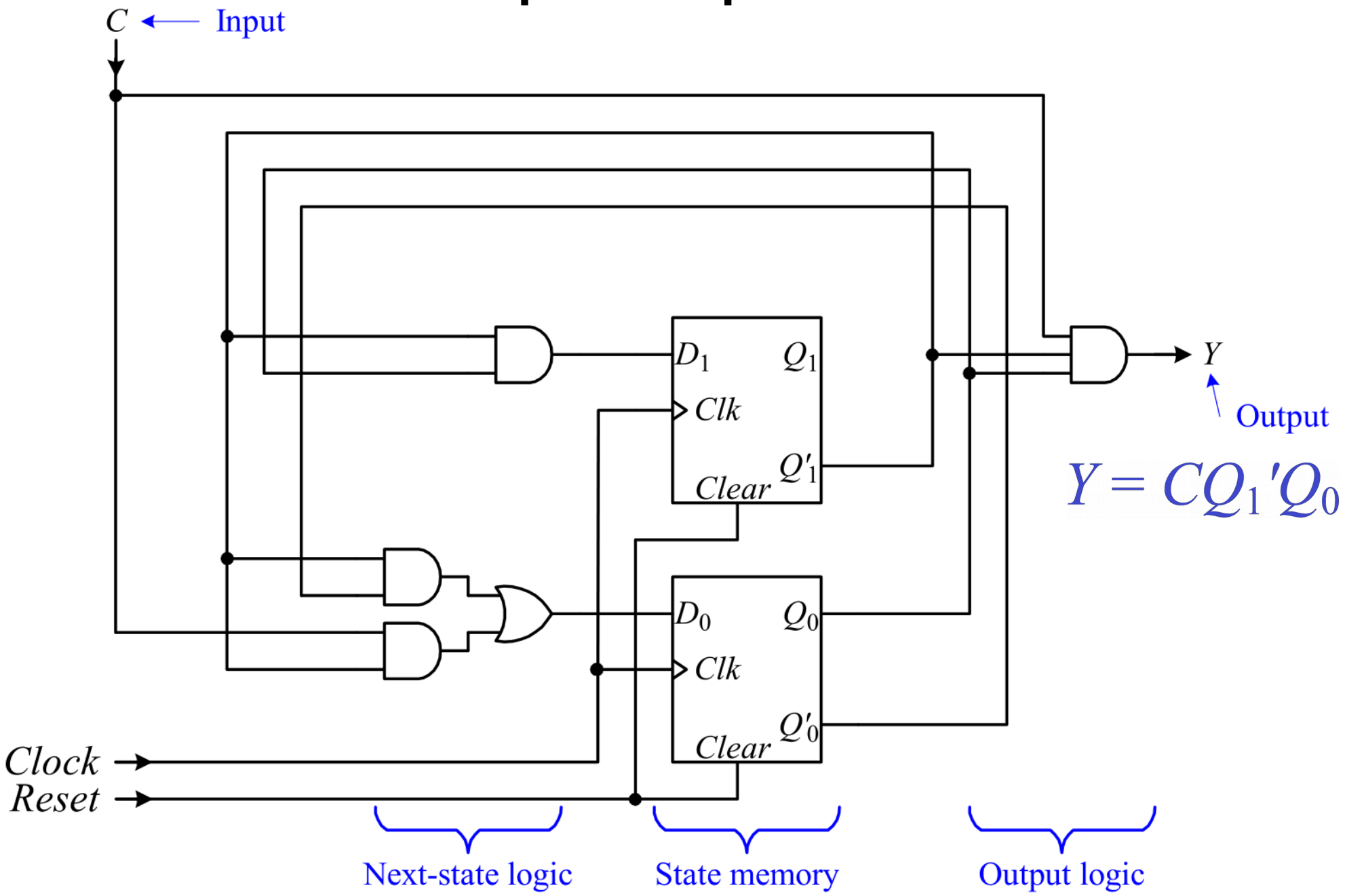
The **output equations** are the equations derived from the combinational output logic circuit in the FSM.

Depending on the type of FSM (Moore or Mealy), the output equations can be dependent on just the current state or on both the current state and the inputs.

Output equation



Output equation



Output table

The output table is the truth table that is derived from the output equations.

Examples:

Current State Q_1Q_0	Output Y
00	0
01	1
10	0
11	0

Moore

Current State Q_1Q_0	Output Y	
	$C = 0$	$C = 1$
00	0	0
01	0	1
10	0	0
11	0	0

Mealy

State diagrams

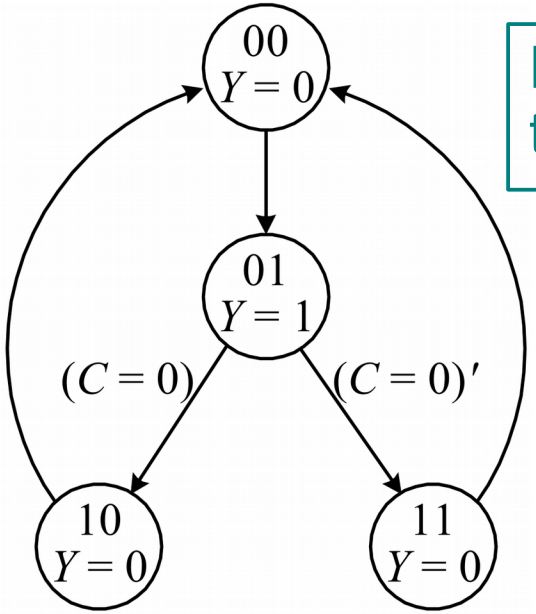
State diagrams are used to describe the operation of finite-state machines.

One node for every state of the FSM

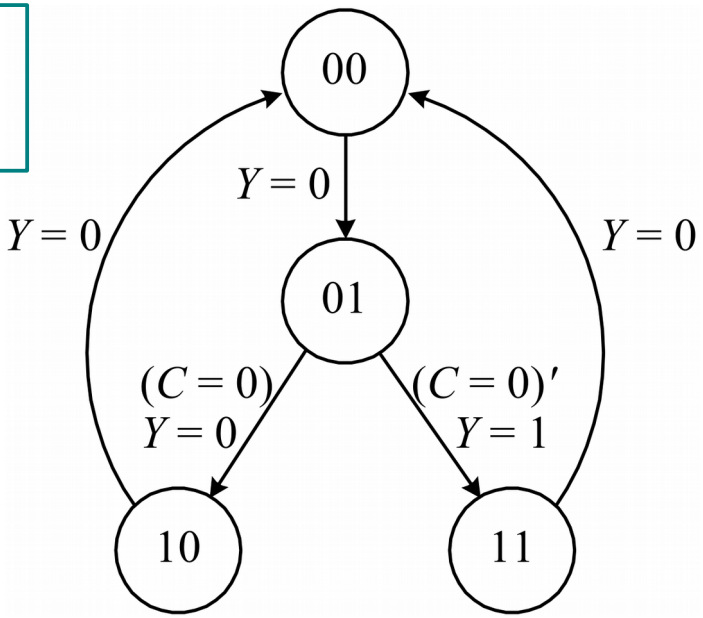
For every state transition there is a directed edge connecting two nodes

Edges for unconditional transitions don't have a label

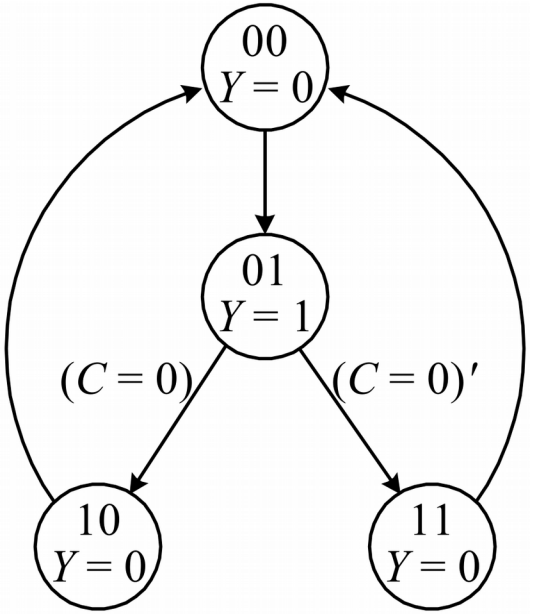
If the output is dependent only on the current state, it is labeled inside the node



A Moore FSM with four states, one input signal C, and one output signal Y



A Mealy FSM with four states, one input signal C, and one output signal Y



After reset, the FSM starts from state 00. When it is in state 00, it outputs a 0 for Y.

At the next clock cycle, the FSM unconditionally transitions to state 01 and outputs a 1 for Y.

Next, the FSM will either go to state 10 or 11 in the next clock cycle depending on the condition (C = 0).

If the condition (C = 0) is true, then the FSM will go to state 10 and output a 0 for Y, otherwise, it will go to state 11 and also output a 0 for Y.

From either state 10 or 11, the FSM will unconditionally transition back to state 00 at the next clock cycle.

The FSM will always go to a new state at the beginning of the next active clock edge.

State diagrams

The state diagram is obtained directly from the next-state table and the output table.

Synthesis of sequential circuits

The synthesis of sequential circuits is just the reverse of the analysis of sequential circuits.

In synthesis, we start with a functional description of the circuit that we want.

From this description, we need to come up with the precise operation of the circuit using a state diagram.

The state diagram allows us to construct the next-state and output tables.

From these two tables, we get the next-state and output equations, and finally the complete FSM circuit.