Synchronous Sequential Logic

Ch.5

Outline

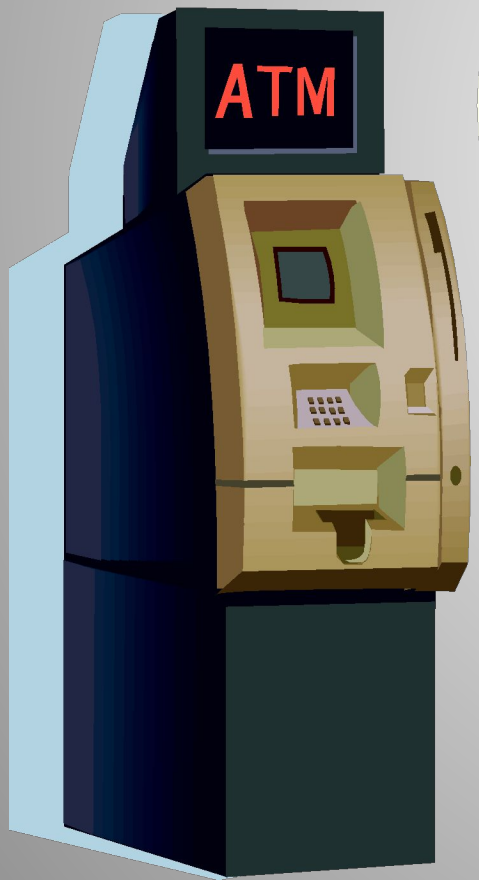
- Synchronous vs asynchronous sequential circuits
- Storage elements
  - Latches
  - Flip-flops
- Mealy and Moore models
- Analysis of synchronous sequential circuit
- State diagram and state table
- Design of synchronous sequential circuit
- State reduction and assignment

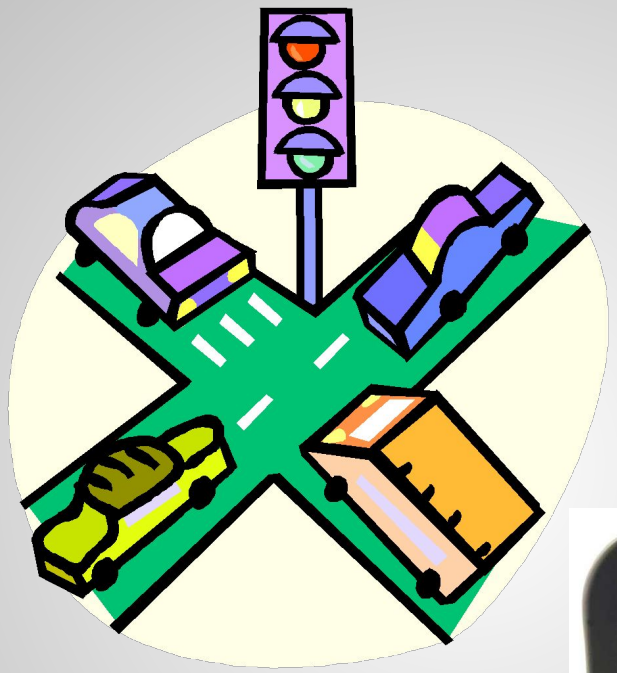
Introduction

. Combinational circuit

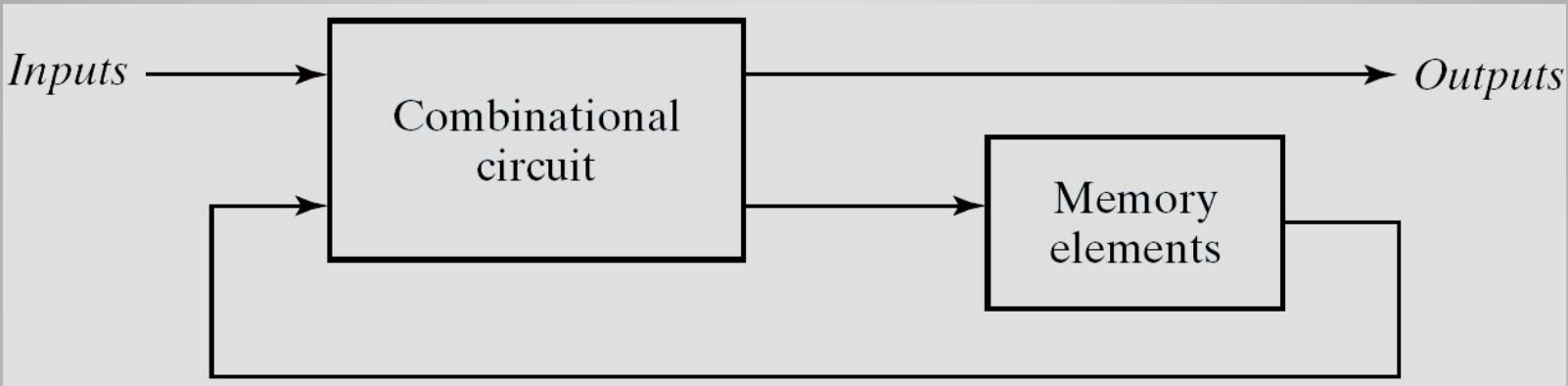
- contains no memory element
- outputs determined solely by present inputs
- does not meet the need of all applications

State Machine Examples





Sequential Circuit



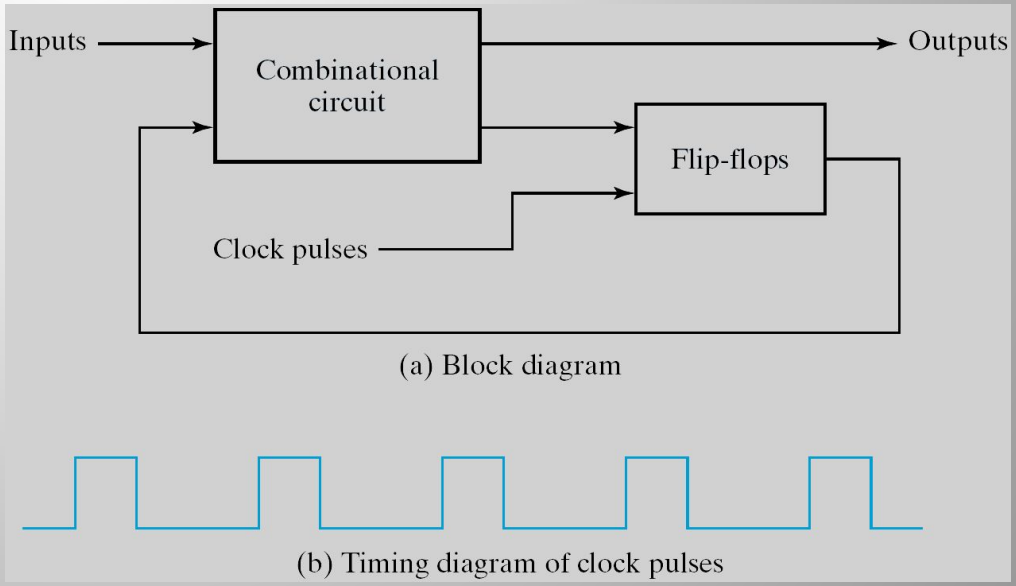
- Has feedback path
- *State* of sequential circuit: content of memory elements
- (inputs, current state) $\Rightarrow$ (outputs, next state)

Synchronous vs Asynchronous

- Types of sequential circuit
  - *Synchronous*: synchronized by a *clock*, the state of the circuit is updated upon receiving a clock pulse
  - *Asynchronous*: the state of the circuit can be updated at any time when an input change occurs

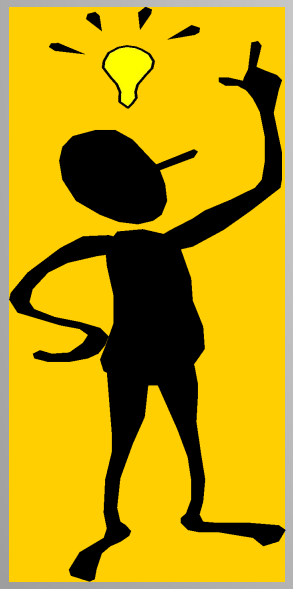
Synchronous Sequential Circuit

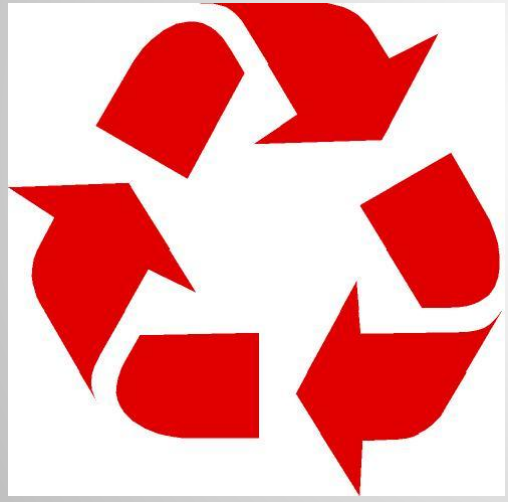
- Synchronous(clocked) sequential circuit
  - a master-clock generator generates a periodic train of clock pulses
  - the clock pulses are distributed throughout the system to the storage elements
  - most popular



Storage

- Storage? How?





Logic Based Storage

. We may keep a 0-bit or a 1-bit by re-generating it in a logic loop!

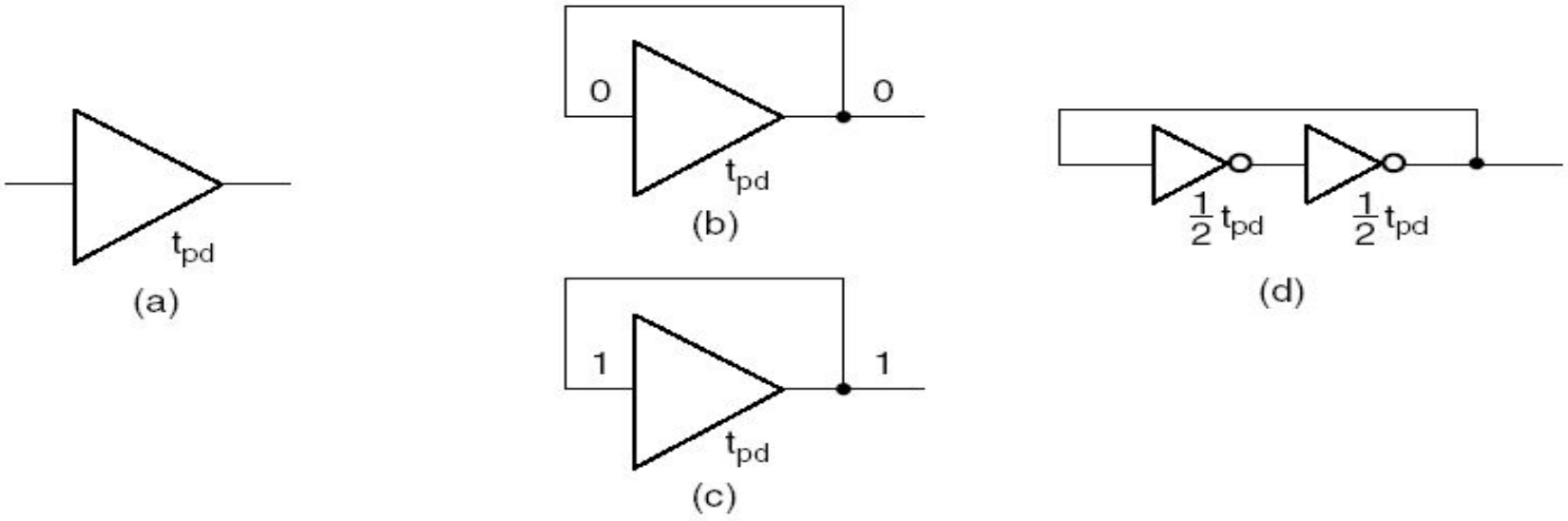
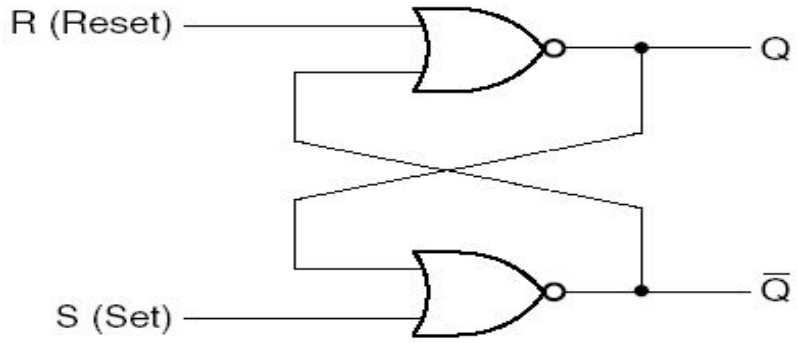


Fig. 4-2 Logic Structures for Storing Information

Latches

. Latches
  - the most basic storage elements
  - *asynchronous* circuits (have no clock input)
  - flip-flops can be constructed from latches

. e.g. *SR* latch



(a) Logic diagram

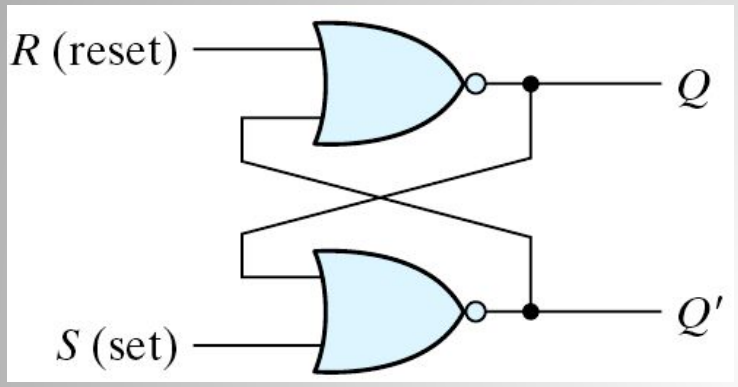
S	R	Q	$\overline{Q}$	
1	0	1	0	Set state
0	0	1	0	
0	1	0	1	Reset state
0	0	0	1	
1	1	0	0	Undefined

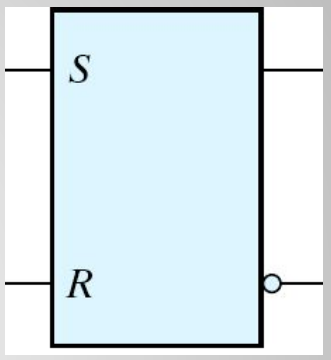
(b) Function table

Fig. 4-4 *SR* Latch with NOR Gates

SR Latch

. Cross coupling two NOR gates



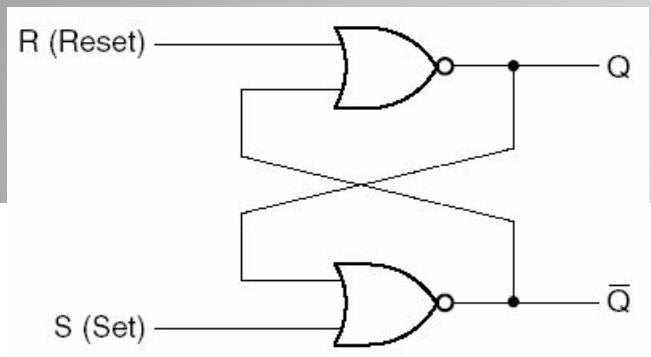


. Operations

- (*S*,*R*)=(0,0): no operation
- (*S*,*R*)=(0,1): reset (*Q* ←0, the clear state)
- (*S*,*R*)=(1,0): set (*Q* ←1, the set state)
- (*S*,*R*)=(1,1): indeterminate state (*Q*=*Q* '=0)

S R	Q^+
0 0	Q
0 1	0
1 0	1
1 1	Not allowed

Logic Simulation of *SR* Latch



R changes; then Q; then Q_B

S changes; then Q_B; then Q

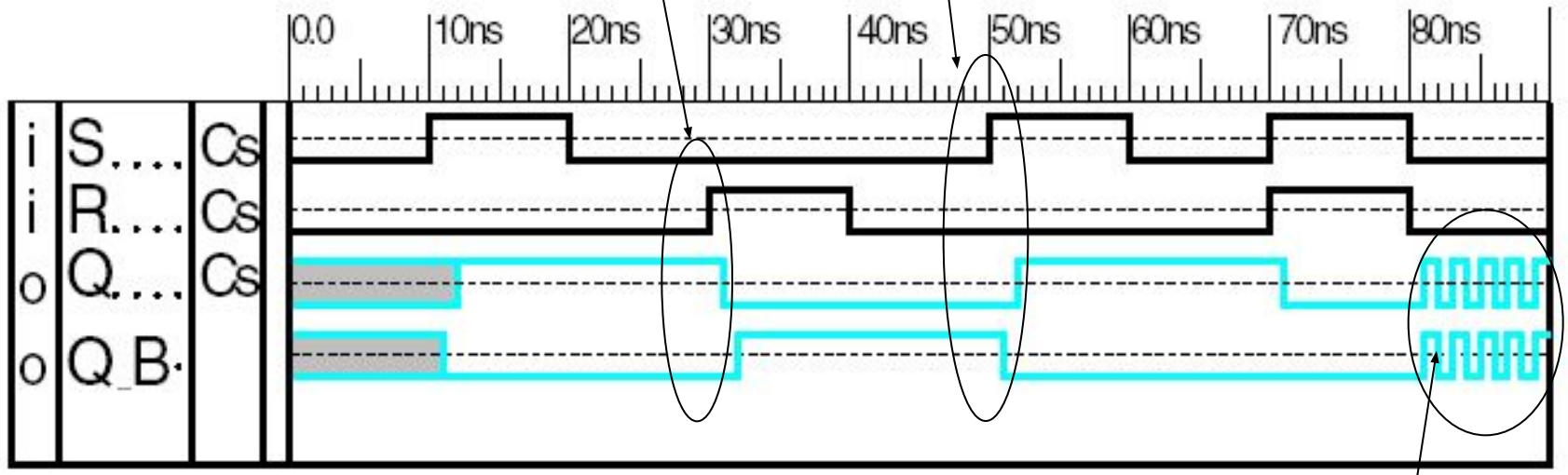
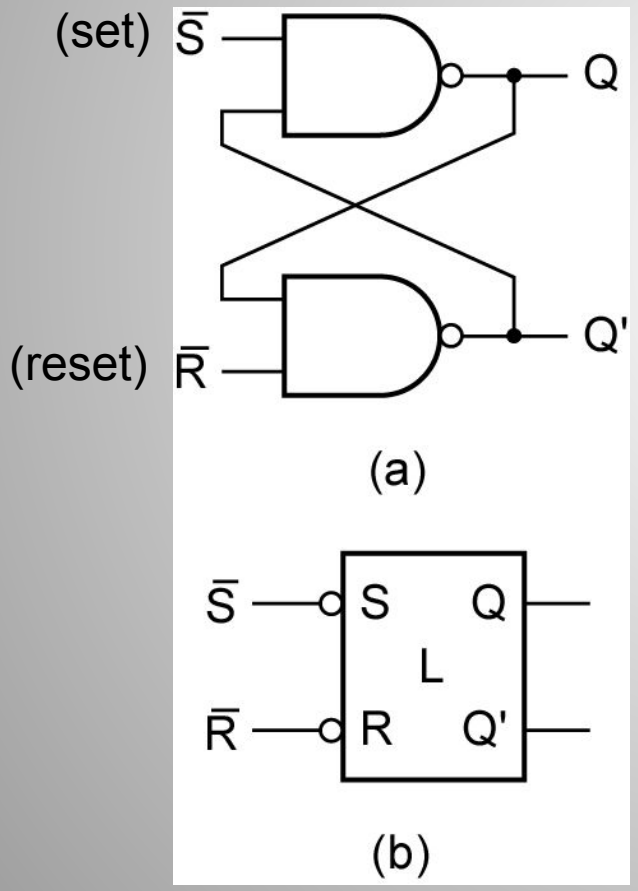


Fig. 4-5 Logic Simulation of *SR* Latch Behavior

indeterminate behavior

$\overline{S}\,\overline{R}$ latch

- use NAND gates
- set and reset occur with a logic-0 signal

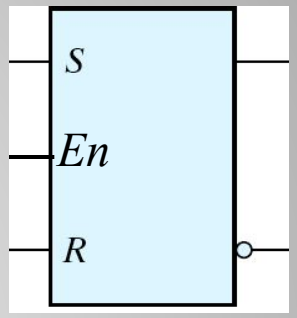


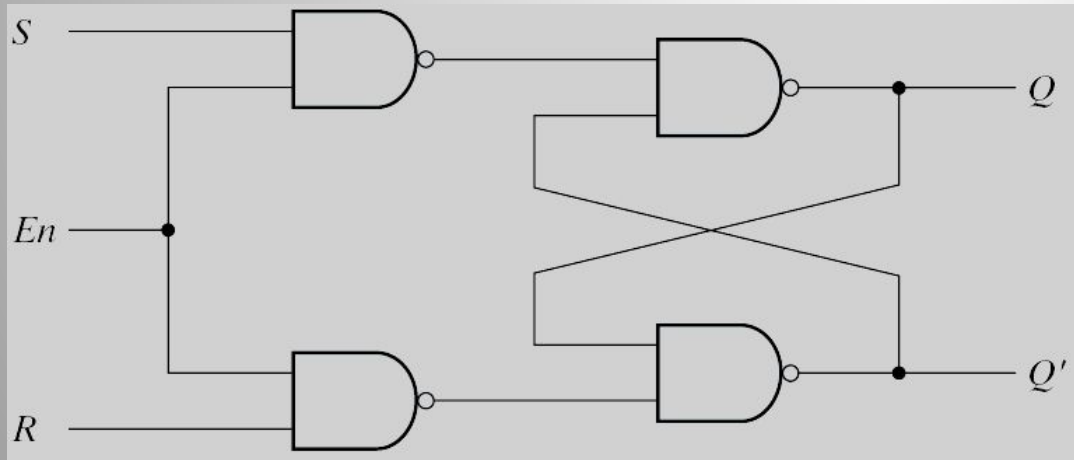
set	reset	Q	Q^+	
1	1	0	0	
1	1	1	1	
1	0	0	0	
1	0	1	0	
0	1	0	1	
0	1	1	1	
0	0	0	–	inputs not allowed
0	0	1	–	

SR Latch with Control Input

SR latch with control input

- *En* = 1: enabled
- *En* = 0: disabled



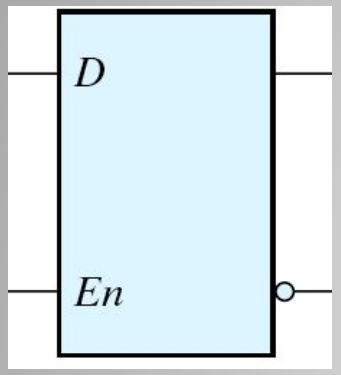


(a) Logic diagram

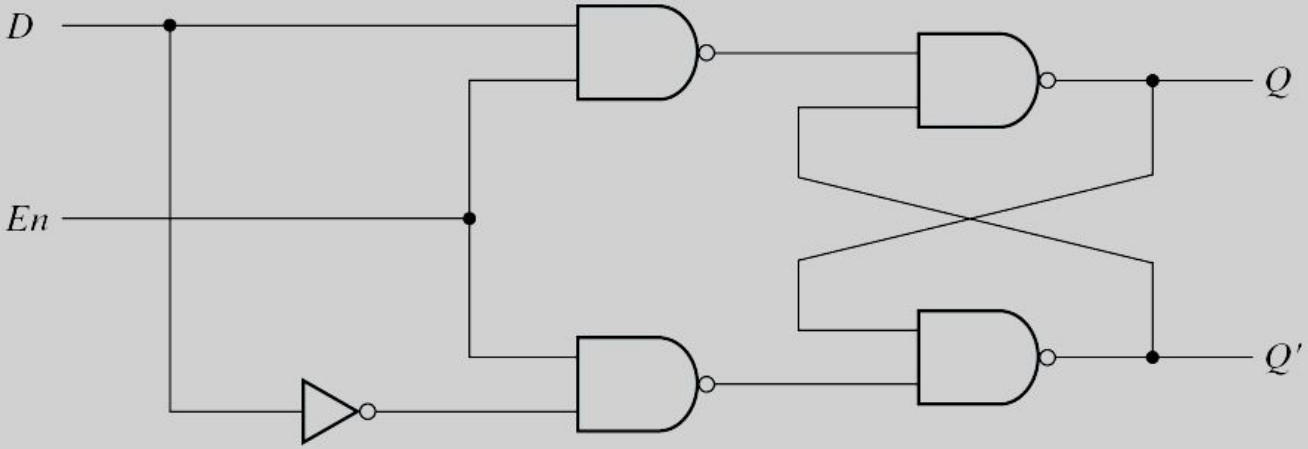
En	*S*	*R*	Next state of *Q*
0	X	X	No change
1	0	0	No change
1	0	1	$Q = 0$; reset state
1	1	0	$Q = 1$; set state
1	1	1	Indeterminate

(b) Function table

D Latch



- Use only a single input *D* to eliminate the undesirable state
- $Q \leftarrow D$ when $En = 1$
- no change when $En = 0$



(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

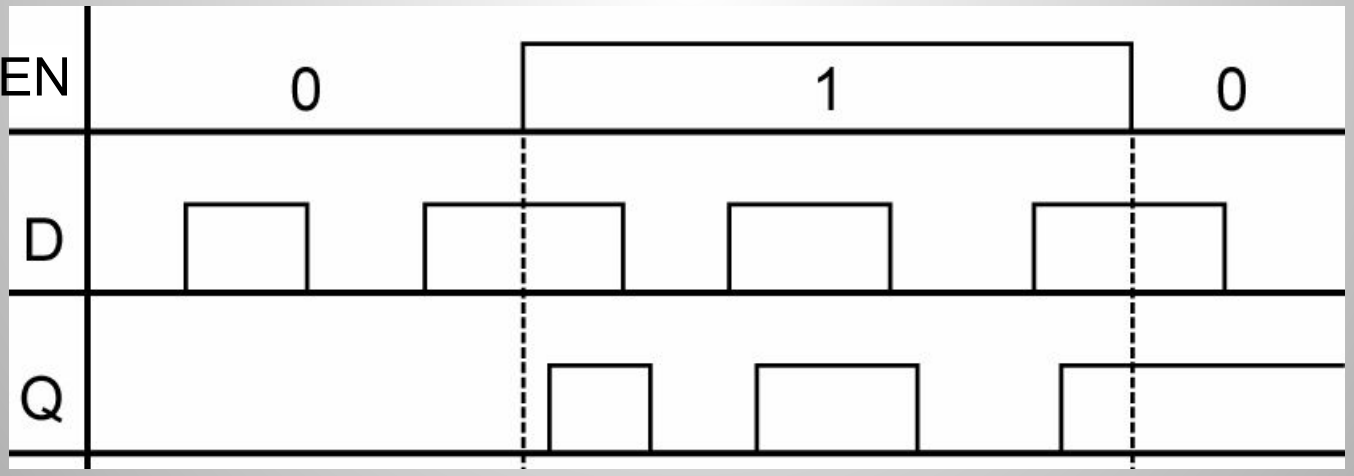
(b) Function table

D Latch in Verilog

```
always @(En, D) begin
  if (En == 1'b1) begin
    Q = D;
  end
end
```

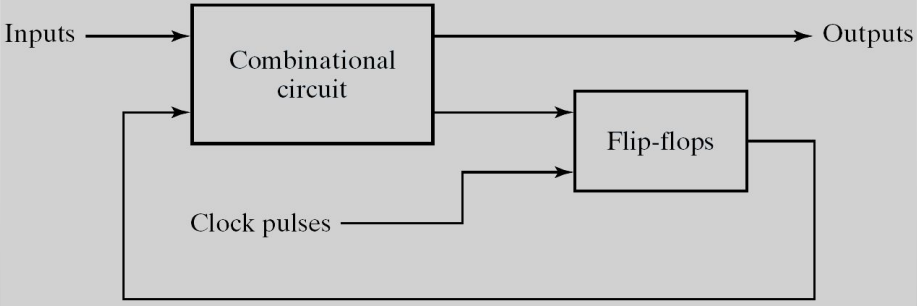
Latch vs Flip-Flop

- Latch
  - *level triggered*
  - "transparent" as long as *EN* = 1
    - state will keep changing if input keeps changing



Latch vs Flip-Flop

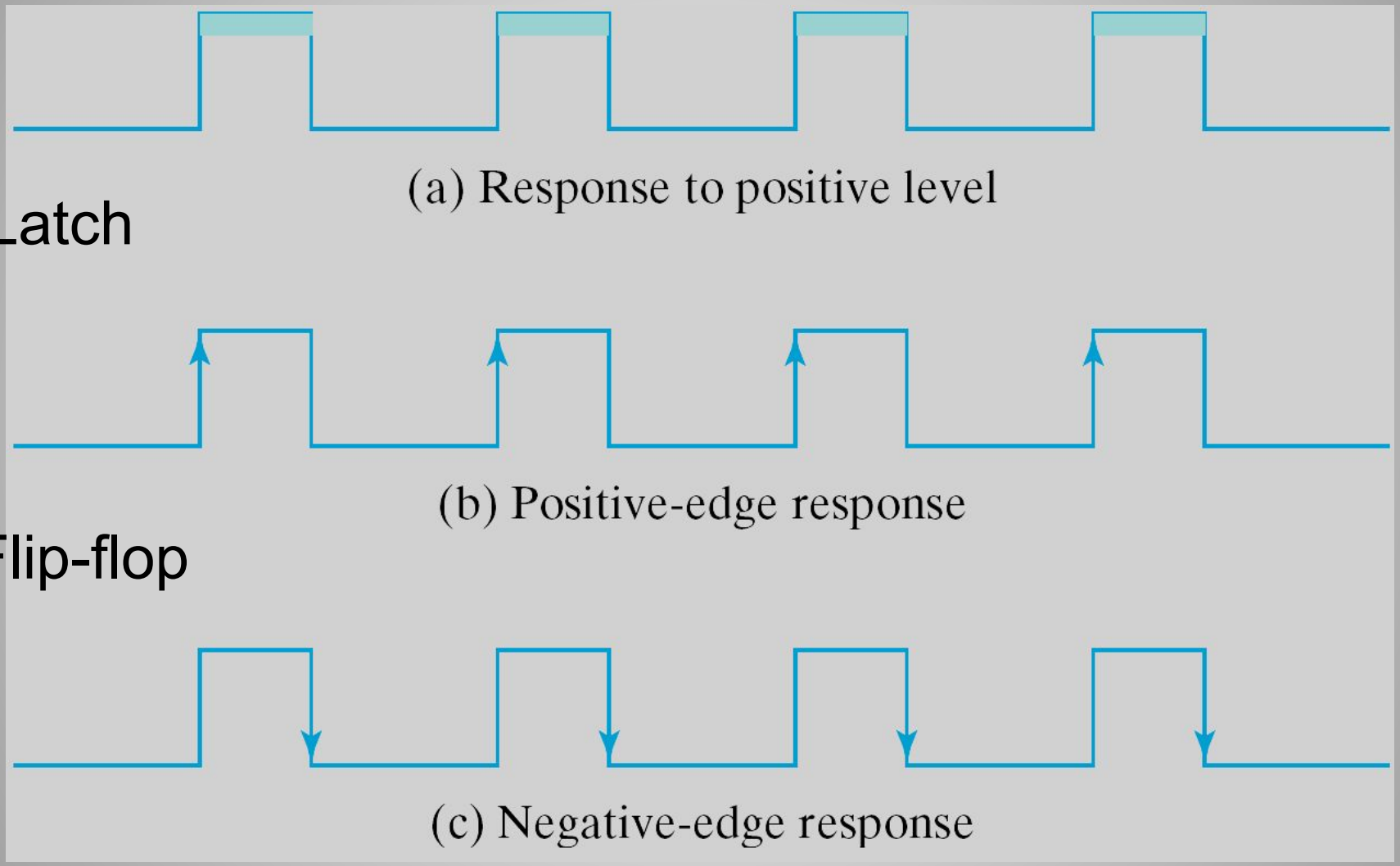
- Flip-flop
  - storage element whose state cannot change more than once in a clock cycle
  - *edge triggered*
    - state transition happens only on active clock edge
    - eliminate the multiple-transition problem



(a) Block diagram

(b) Timing diagram of clock pulses

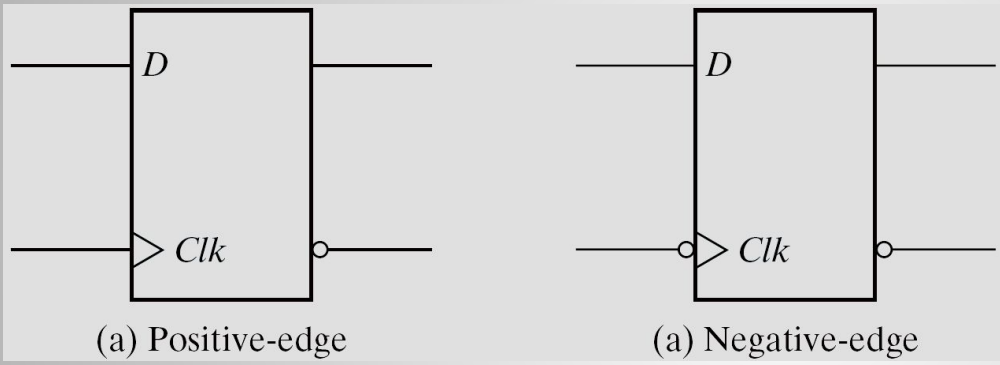
Clock Response in Latch and Flip-Flop



D Flip-Flop

. *D* FF's output changes only at active clock edge

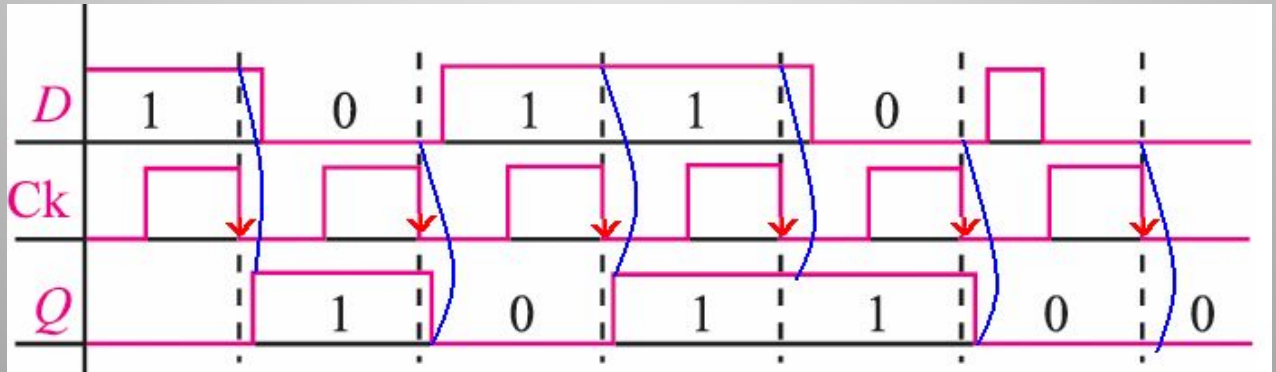
. Either *positive-* or *negative-edge triggered*



D Flip-Flop

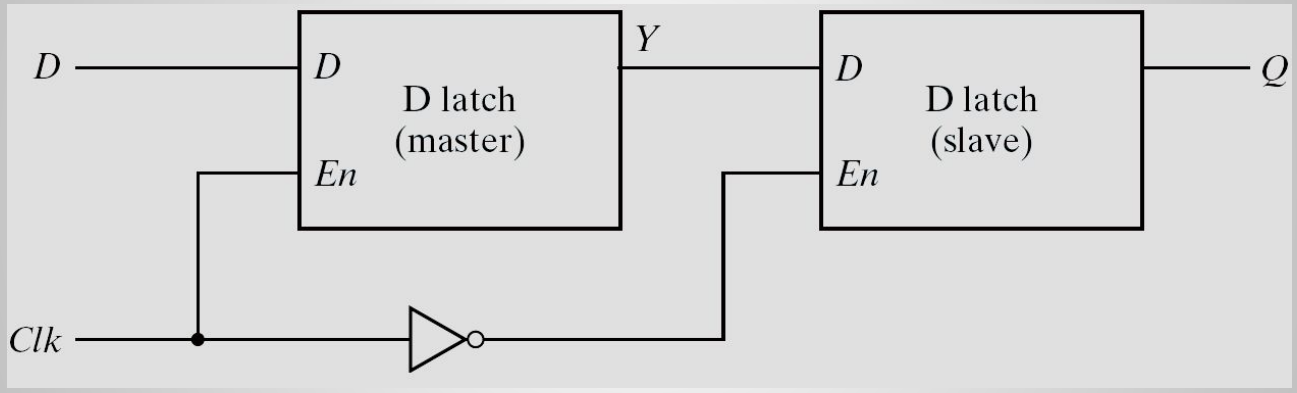
D	Q(t + 1)
0	0
1	1

. E.g. negative-edge triggered version

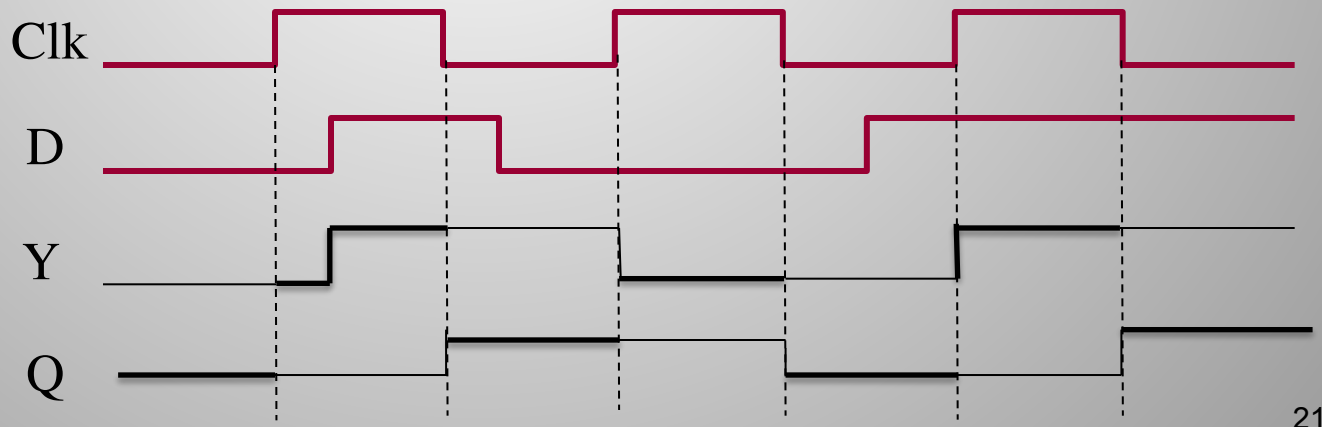


Negative-edge Triggered *D* Flip-Flop

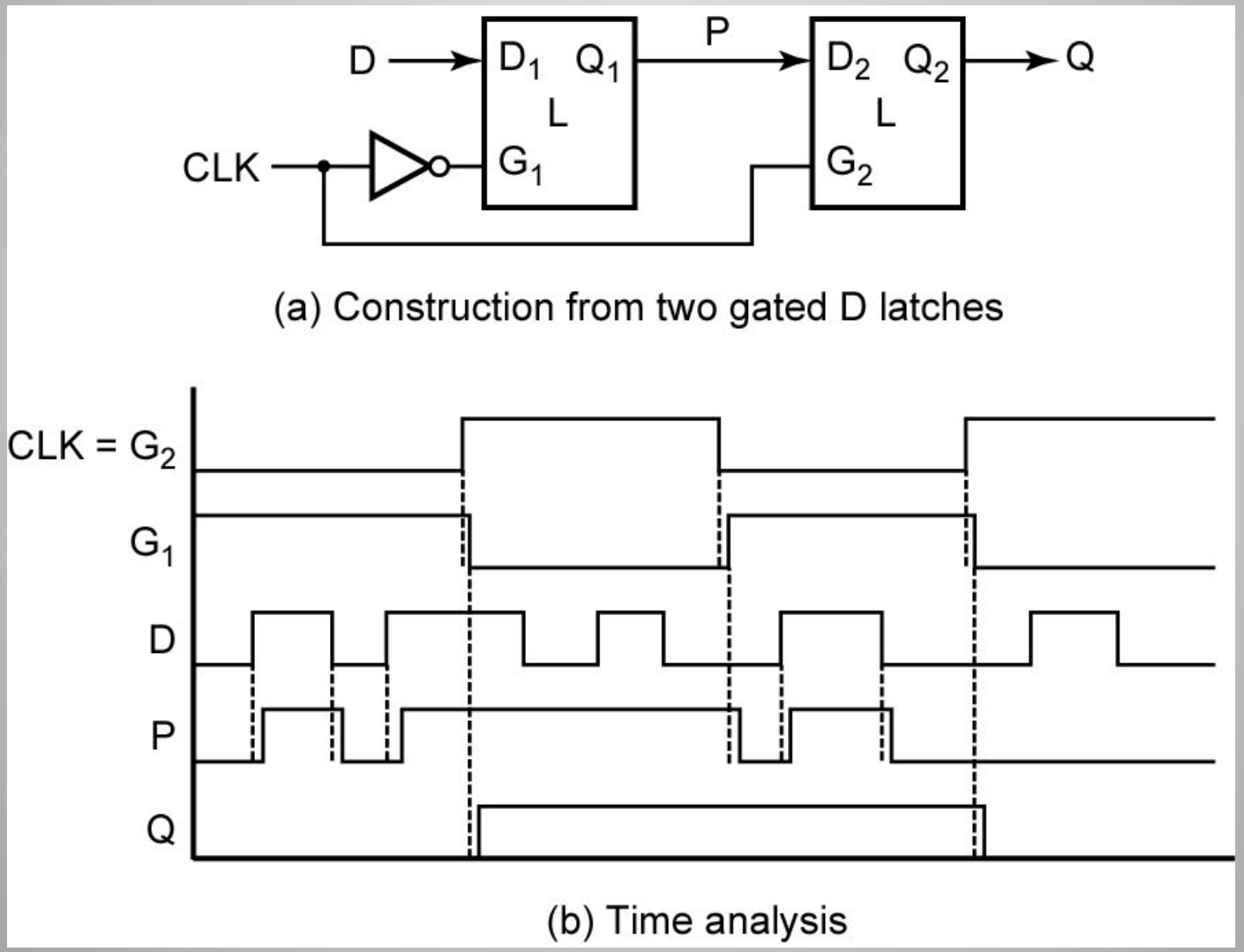
- Can be formed by two latches + an inverter
  - master latch and slave latch



Operation:



Positive-edge Triggered *D* Flip-Flop



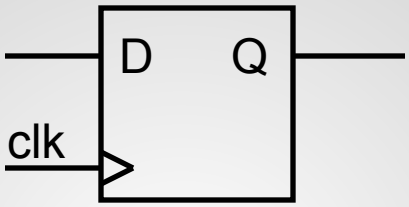
Flip-Flop Timing Parameters

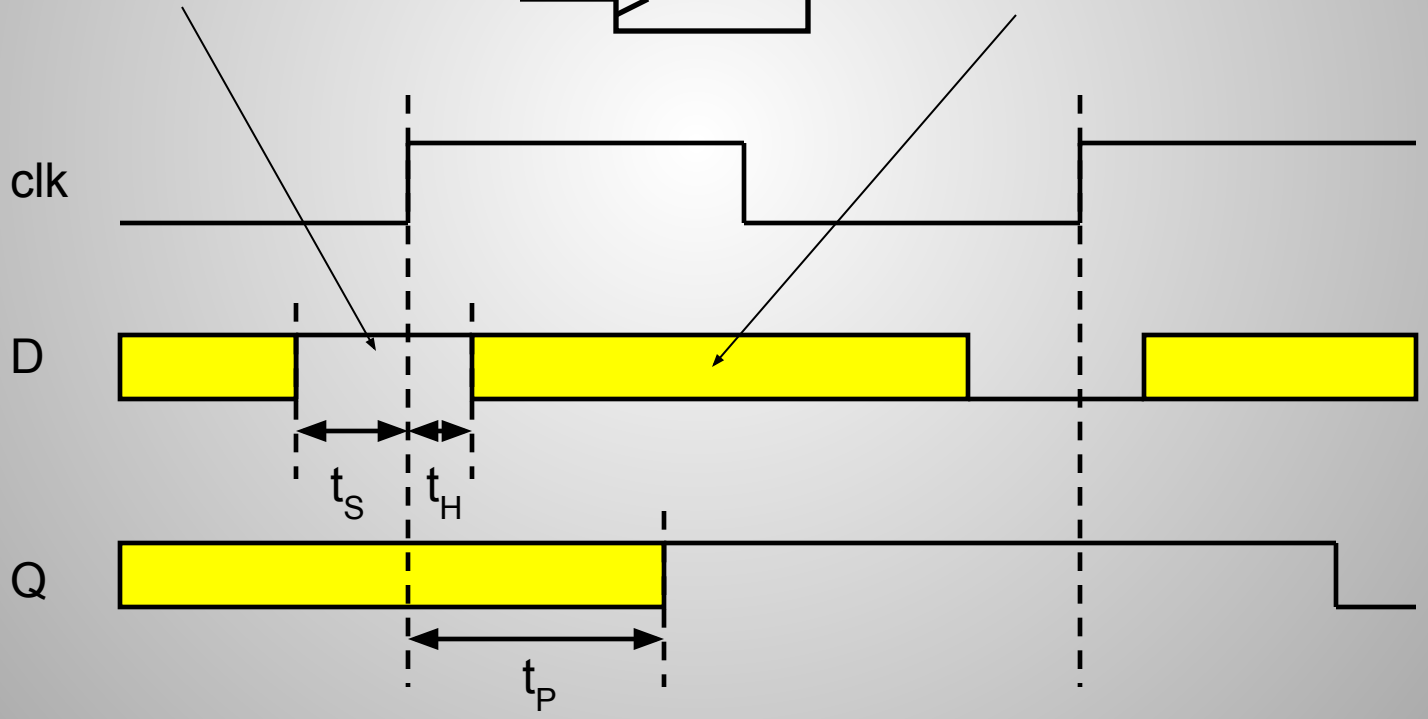
- *Setup time* t_S
  - minimum time for which the input data must be stable before the trigger edge
- *Hold time* t_H
  - minimum time for which the input data must remain stable after the trigger edge
- *Propagation delay* t_P
  - interval between the trigger edge and the stabilization of the output to its new value

Timing Characteristics of Sequential Circuits

Input D must remain stable during this interval

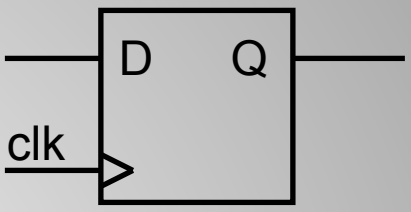
Input D can freely change during this interval





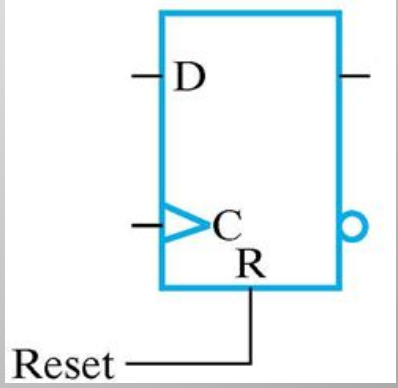
Positive-edge Triggered DFF in Verilog

```
reg Q;
always @(posedge clk) begin
  Q <= D;
end
```

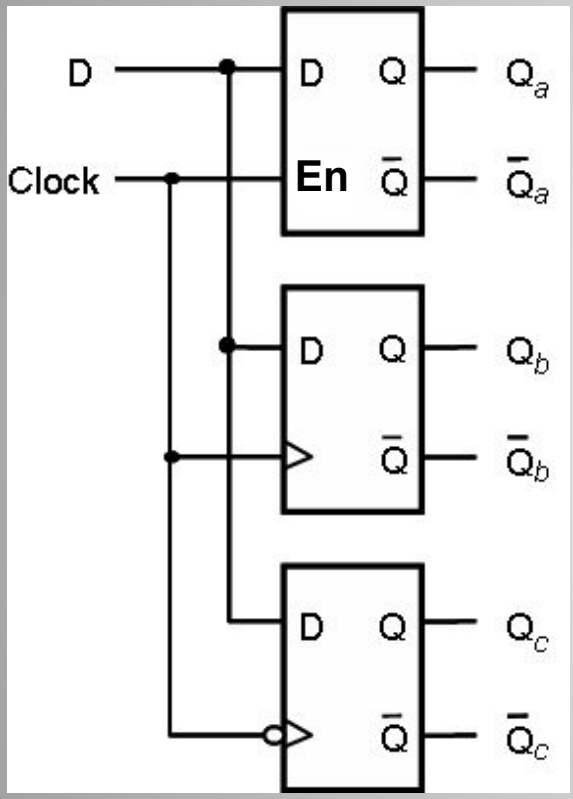


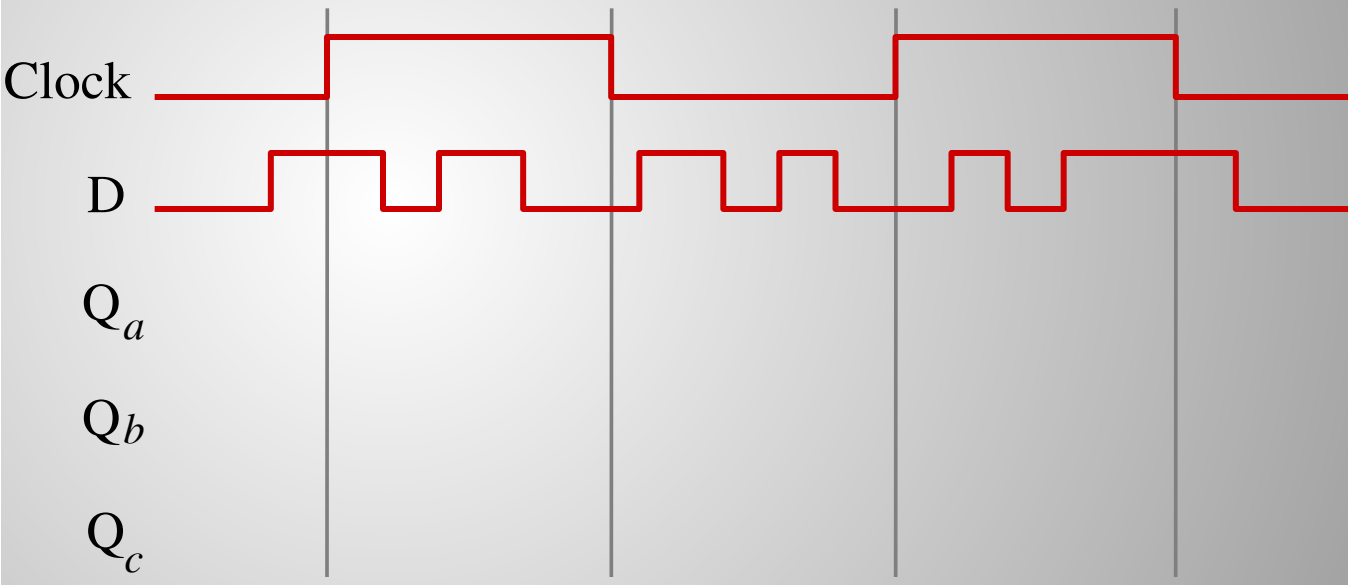
- Or we usually have DFF with a reset control

```
reg Q;
always @(posedge clk, posedge rst) begin
  if (rst == 1)
    Q <= 0;
  else
    Q <= D;
end
```



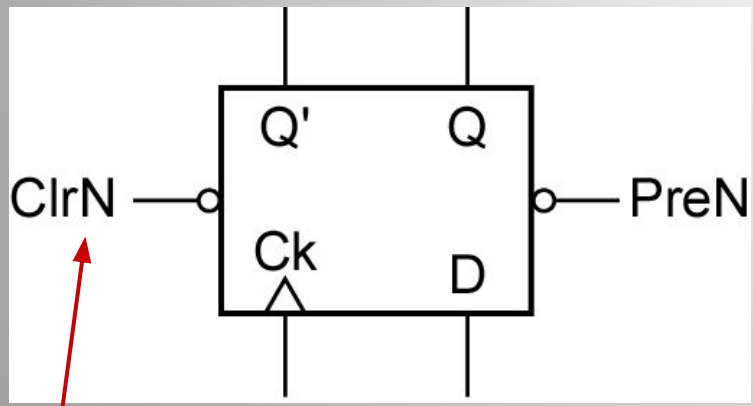
Exercise


D
Clock
D
Q
En
Q̄
Q_a
$\bar{Q}_a$
D
Q
Q̄
Q_b
$\bar{Q}_b$
D
Q
Q̄
Q_c
$\bar{Q}_c$


Clock
D
Q_a
Q_b
Q_c

Flip-Flops with Inputs: Asynchronous Set/Reset

- FFs often have additional inputs to set to an initial state asynchronously
- E.g. *D* flip-flop with *active-low* preset and clear

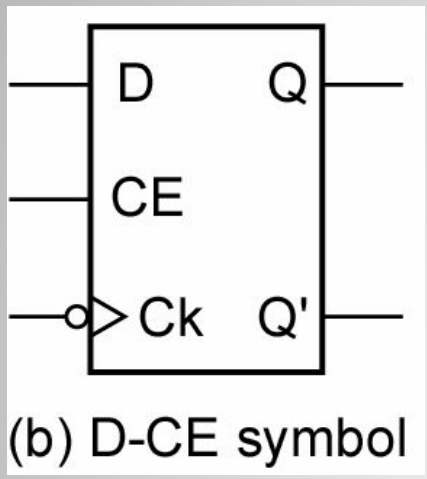


clear at logic-0

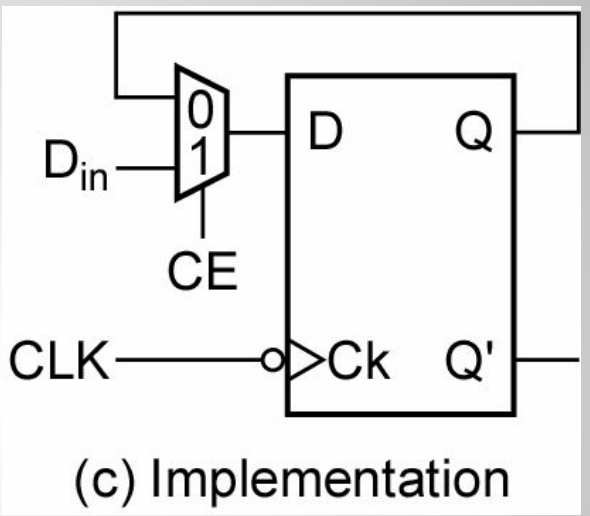
Ck	*D*	PreN	ClrN	Q^+
x	x	0	0	(not allowed)
x	x	0	1	1
x	x	1	0	0
↑	0	1	1	0
↑	1	1	1	1
0,1,↓	x	1	1	*Q* (no change)

Flip-Flops with Additional Inputs: Enable Input

. E.g. *D* flip-flop with clock enable



(b) D-CE symbol



(c) Implementation

- If *CE*=0, disabled, $Q(t+1) = Q$
- If *CE*=1, like a normal *D* flip-flop, $Q(t+1) = D$
- $Q(t+1) = Q \cdot CE' + D \cdot CE$

Importance of the Sensitivity List

- *D-Register with synchronous clear*

```
module dff_sync_clear(
input d, clearb, clock,
output reg q
);
 always @(posedge clock) begin
   if (!clearb) q <= 1'b0;
   else q <= d;
 end
endmodule
```

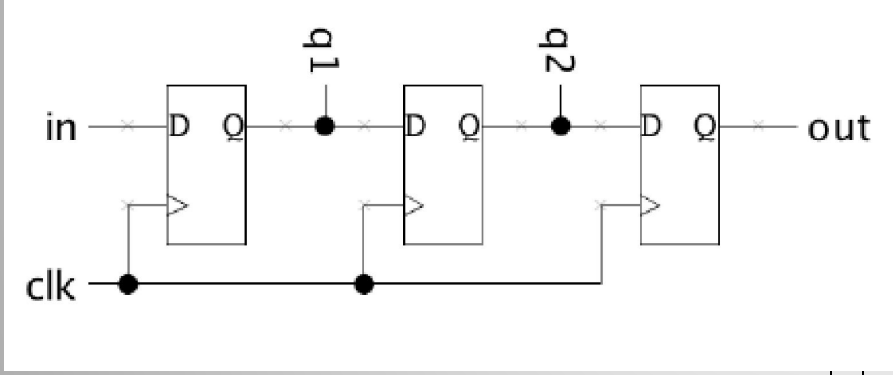
- *D-Register with asynchronous clear*

```
module dff_sync_clear(
input d, clearb, clock,
output reg q
);
  always @(negedge clearb or
    posedge clock) begin
     if (!clearb) q <= 1'b0;
     else q <= d;
  end
endmodule
```

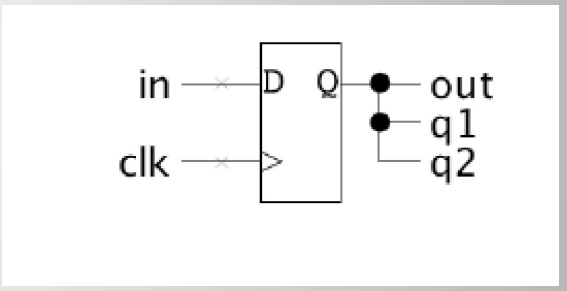
If one signal in the sensitivity list uses posedge/negedge, then all signals must.

Use Nonblocking (<=) for Sequential Logic

```
always @(posedge clk) begin
      q1 <= in;
      q2 <= q1; // uses old q1
      out <= q2; // uses old q2
end
```




```
always @(posedge clk) begin
      q1 = in;
      q2 = q1; // uses new q1
      out = q2; // uses new q2
end
```



Sequential always block style

```
reg [3:0] count1, next_count1;

always @(posedge clk)
    count1 <= next_count1;

always @* begin
  if (reset) next_count1 = 0;
  else next_count1 =
    (count1 == 4) ? 0 : count1 + 1;
end

assign enable1 = (count1 == 4);
```

```
reg [3:0] count2;

always @(posedge clk) begin
 if (reset) count2 <= 0;
 else count2 <=
  (count2 == 4) ? 0 : count2 + 1;
end

assign enable2 = (count2 == 4);
```

always block

- Sequential always block: always @(posedge clock) use “<=“
- Combinatorial always block: always @ * use “=“
- Results of operators (LHS) inside always block (sequential and combinatorial) must be declared as “reg”
- Equivalent Verilog

```
reg z;
always @ *
z = x || y
```

```
Wire z;
assign z = x && y
// z not a “reg”
```

Coding Guidelines

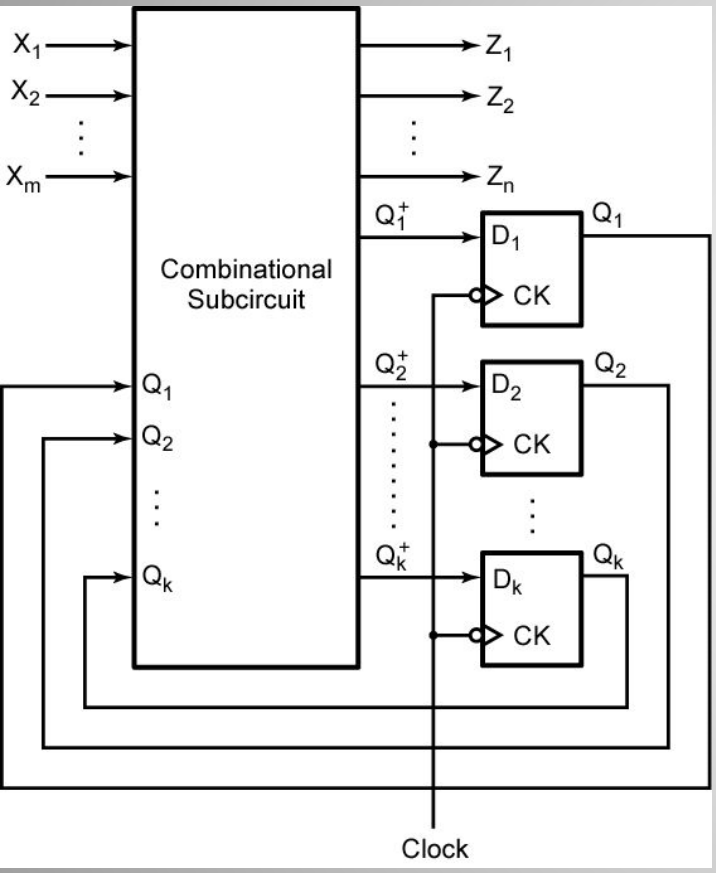
- Ensure your simulation results will match what they synthesized hardware will do:
  - When modeling sequential logic, use nonblocking assignments.
  - When modeling combinational logic with an always block, use blocking assignments.
  - When modeling both sequential and “combinational” logic within the same always block, use nonblocking assignments.
  - Do not mix blocking and nonblocking assignments in the same always block.
  - Do not make assignments to the same variable from more than one always block.

Synchronous Sequential Circuit

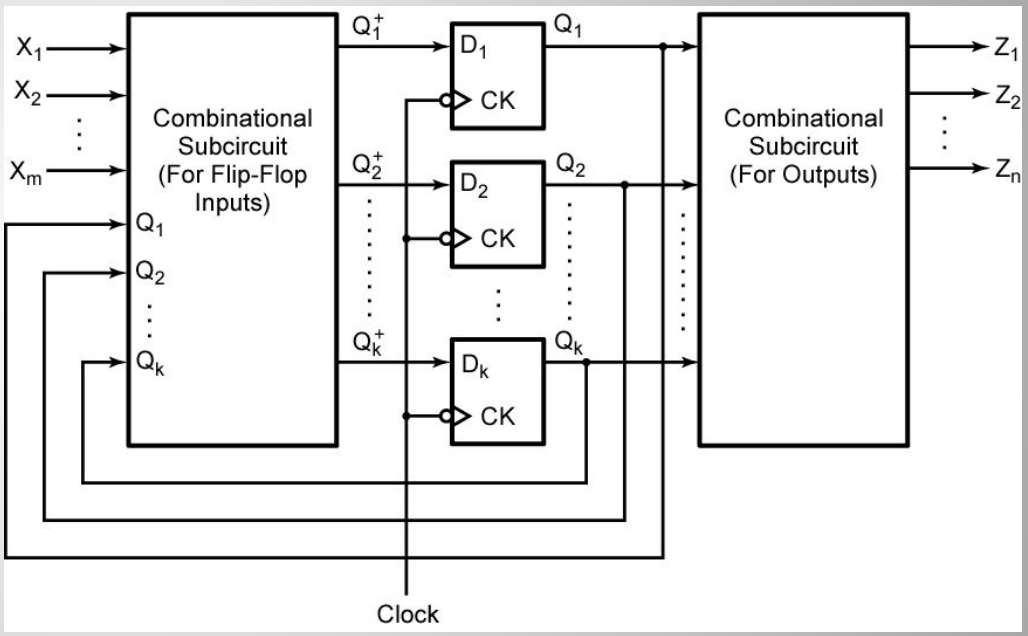
- Also known as *clocked sequential circuit*, or *finite-state machine* (*FSM*)
- Current state at time *t* is stored in a set of flip-flops
- (input, current state) $\Rightarrow$ (output, next state)
  - next state at time *t+1* (one clock period after time *t*) is a Boolean function of the present state and input
  - output is a Boolean function of present state and input

Mealy and Moore Models

- *Mealy circuit*: sequential circuit for which the output is a function of both present state and input
- *Moore circuit*: sequential circuit for which the output is a function of present state only



Mealy machine



Moore machine

Analysis Example with *D* Flip-Flops

- With *D* flip-flops, the next state can be obtained immediately from the flip-flop input equations:

$D_A = AX + BX$ (so $A(t+1) = A(t)X(t) + B(t)X(t)$)

$D_B = A'X$ (so $B(t+1) = A'(t)X(t)$)

$Y = (A+B)X'$

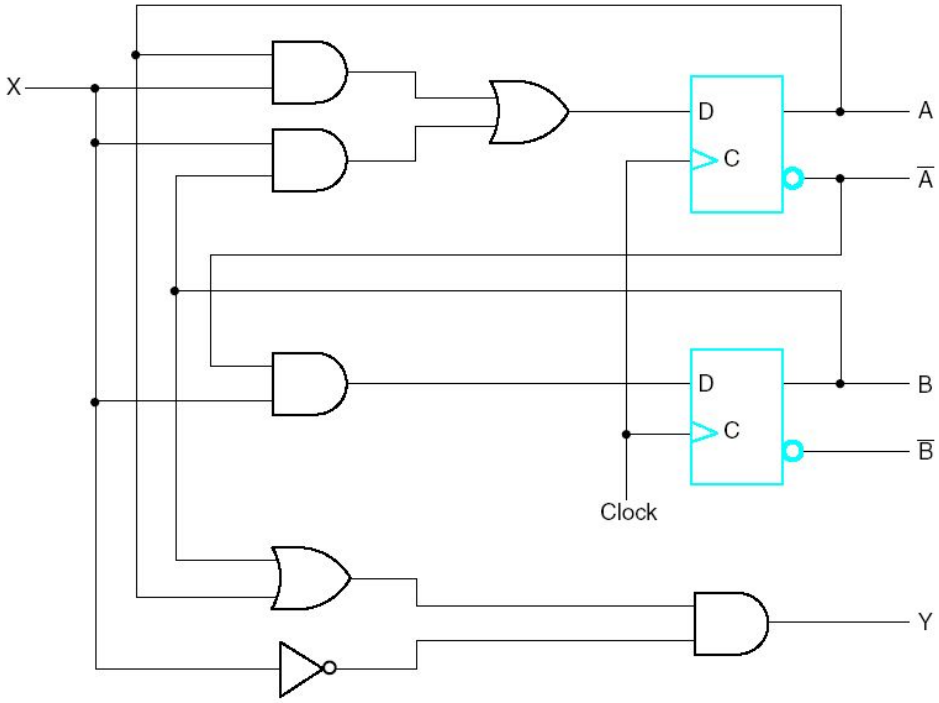


Fig. 4-18 Example of a Sequential Circuit

State Table

- A state table shows the state transitions and outputs in tabular form like below.

TABLE 4-2
State Table for Circuit of Figure 4-18

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

Table 4-2 State Table for Circuit of Figure 4-18

State Table

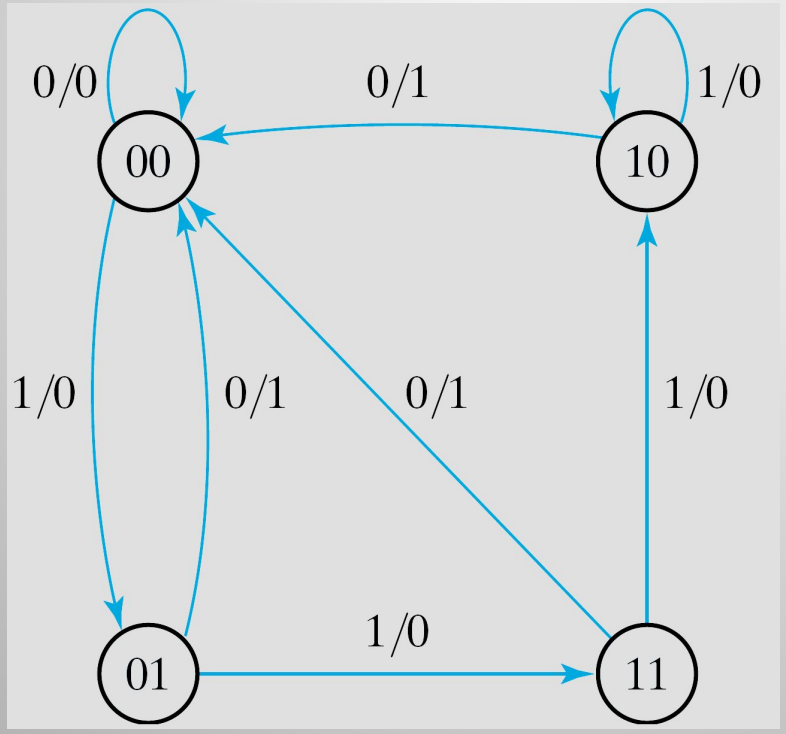
- An alternative way to draw a state table:

Table 5.3
Second Form of the State Table

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

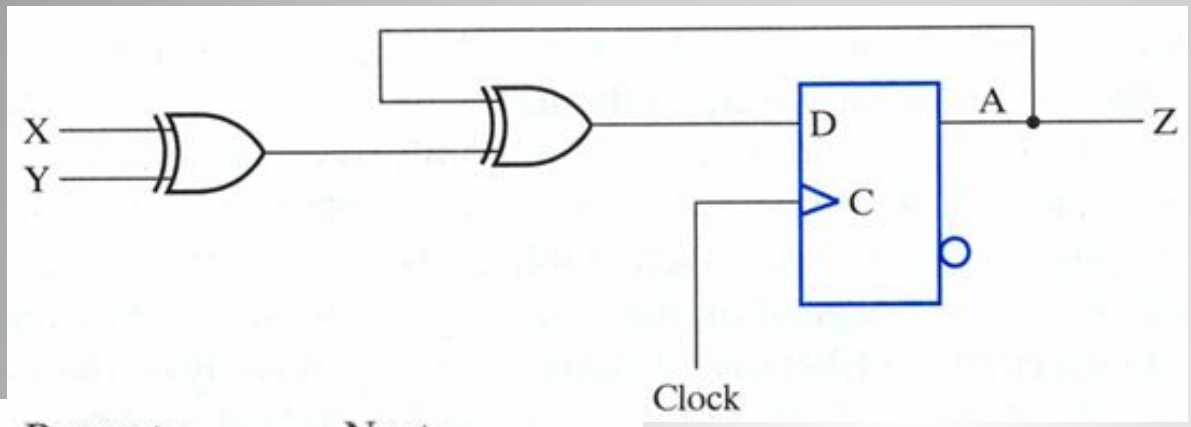
State Diagram

- State transition diagram
  - a circle: a state
  - a directed line: transition between the states



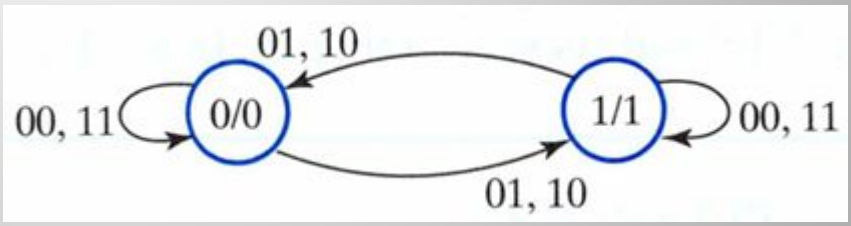
input/output

Another Example



Present State	Inputs		Next State	Output
A	X	Y	A	Z
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	0
1	0	0	1	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

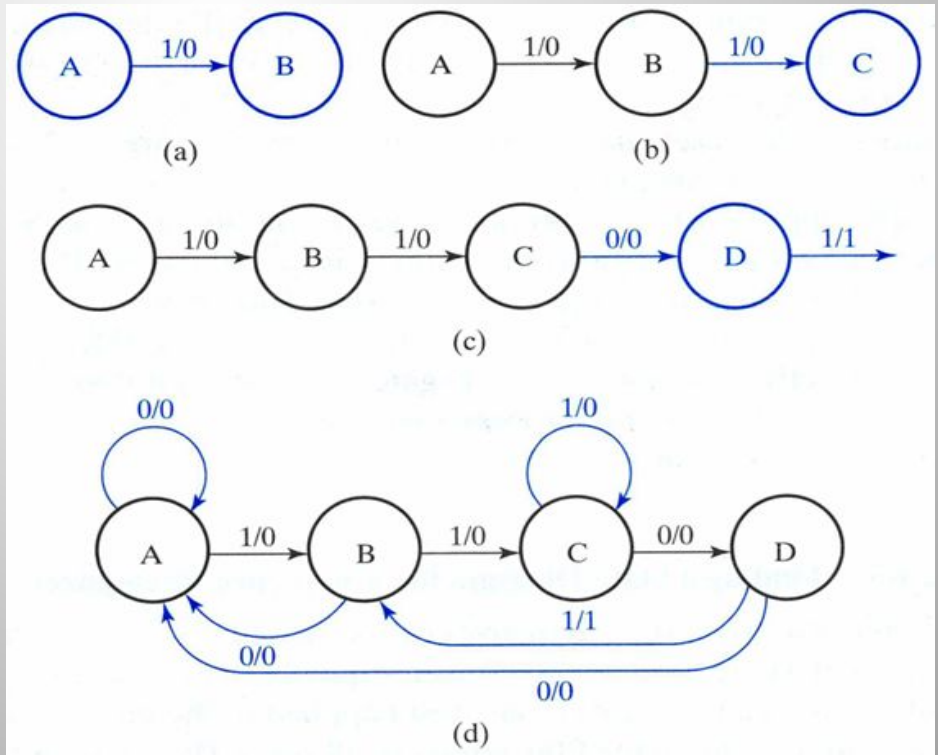
- Input equation
  - $D_A = A \oplus x \oplus y$
- State equation
  - $A(t+1) = A \oplus x \oplus y$
- Output equation
  - $Z = A$



Finding State Diagrams and State Tables

Example. Construct a recognizer that takes one bit at a time and generates an output Z=1 when and only when the sequence “1101” is detected at the input. (A typical input sequence and the corresponding output sequence are: X=0011011100110101101
Z=0000010000000100 1)

A possible design:



Corresponding state table:

TABLE 4-5
State Table for State Diagram in Figure 4-21

Present State	Next State		Output Z	
	X = 0	X = 1	X = 0	X = 1
A	A	B	0	0
B	A	C	0	0
C	D	C	0	0
D	A	B	0	1

Table 4-5 State Table for State Diagram in Figure 4-21

State names replaced by binary codes:

TABLE 4-7
Table 4-5 with Names Replaced by Binary Codes

	Next State		Output Z	
Present State	$X = 0$	$X = 1$	$X = 0$	$X = 1$
00	00	01	0	0
01	00	11	0	0
11	10	11	0	0
10	00	01	0	1

Table 4-7 Table 4-5 with Names Replaced by Binary Codes

Remark: There are many different possible binary code assignments which result in final circuit with different complexities.

(Exercise: (i) Complete the design using the above binary code assignment. (ii) Complete the design using another binary code assignment such as 10, 01, 00, 11 for states *A*, *B*, *C*, and *D*, resp.)

State Reduction

- Two states of a FSM are **equivalent** if the outputs produced for each possible input value are identical and the next states for each possible input value are the same or equivalent
- Equivalent states may be merged into a single state
- State reduction through state equivalence can reduce number of flip-flops used

State Reduction Example

Example

- 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

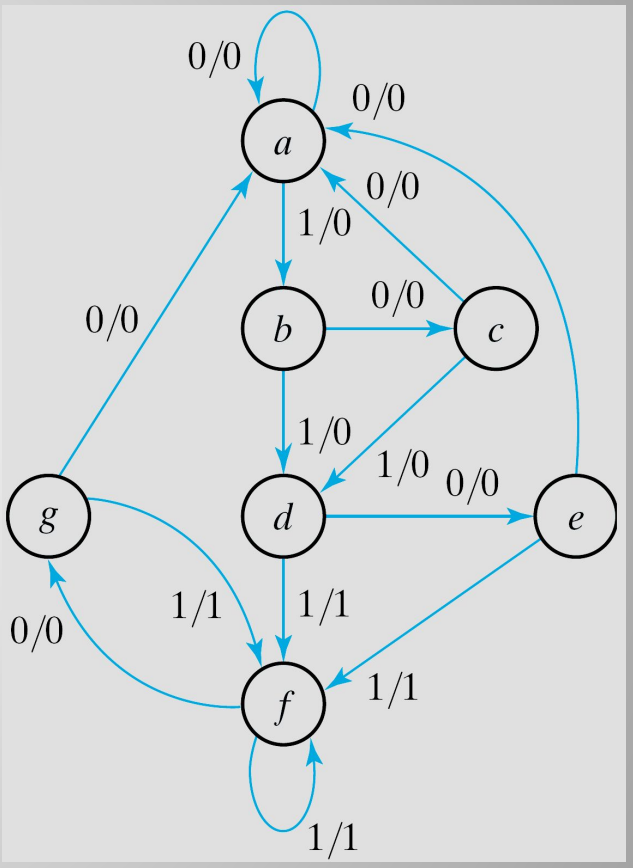


Table 5.6
State Table

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

Equivalent states

- Equivalent states
  - $g = e$
  - one of them can be removed
- Reduced state table

Table 5.7
Reducing the State Table

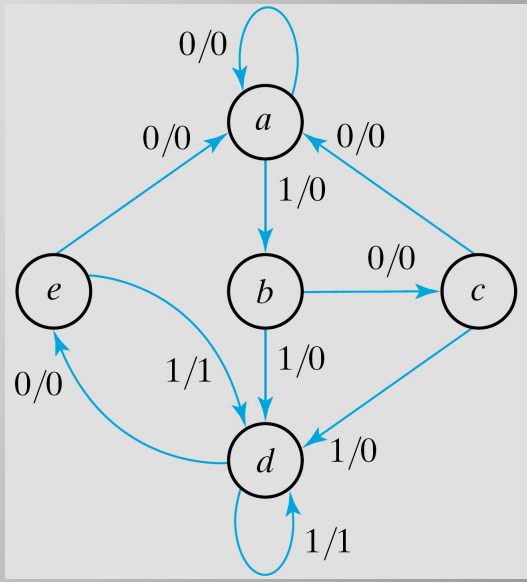
	Next State		Output	
Present State	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1
e	a	f	0	1
f	e	f	0	1

Equivalent states

- New equivalent states
  - $d = f$
  - one of them can be removed
- State table after further reduction

Table 5.8
Reduced State Table

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



State Assignment

- Cost of final circuit depends on state assignment
- Many possible binary state assignments e.g.

Table 5.9
Three Possible Binary State Assignments

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

- *One-hot encoding* uses more FFs but may lead to simpler decoding logic for next state and output

Design Procedure for Sequential Circuit

1. Determine a state diagram or state table
2. State reduction if necessary
3. Assign binary values to the states
   - obtain binary-coded state table
4. Choose the type of flip-flops
   - derive the simplified flip-flop input equations and output equations
5. Obtain the logic diagram

Design Example with D flip-flops

Complete the following design with the given state diagram and the given states to binary codes mapping using *D* flip-flops.

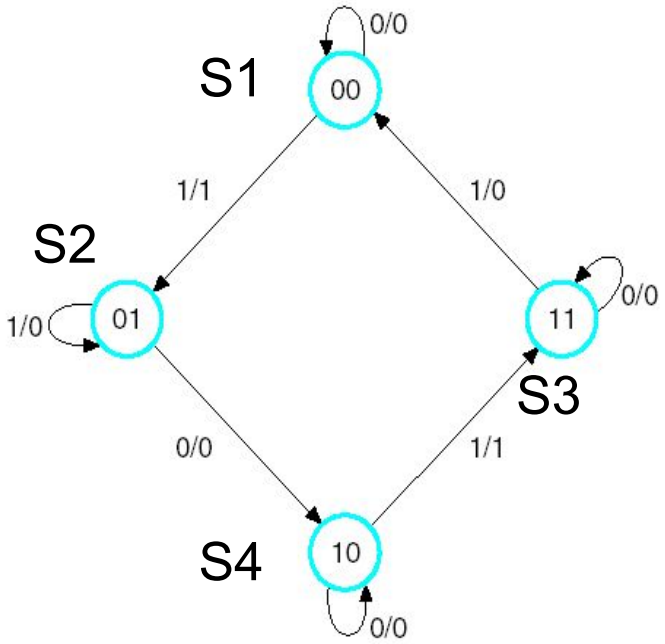


Fig. 4-23 State Diagram for Design Example

TABLE 4-8
State Table for Design Example

Present State		Input	Next State		Output
A	B	X	A	B	Y
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	1	0	0
0	1	1	0	1	0
1	0	0	1	0	0
1	0	1	1	1	1
1	1	0	1	1	0
1	1	1	0	0	0

Table 4-8 State Table for Design Example

Deriving the flip-flop input equations and output function:

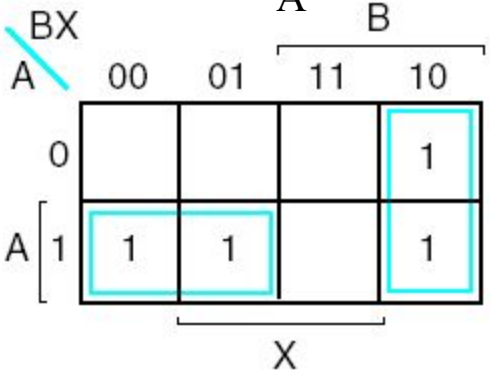
Output $Y(A,B,X) = \Sigma m(1,5)$

Since D flip-flops are used,

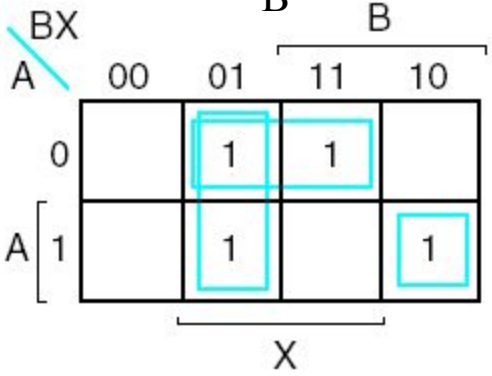
$D_A(A,B,X) = A(t+1) = \Sigma m(2,4,5,6)$

$D_B(A,B,X) = B(t+1) = \Sigma m(1,3,5,6)$

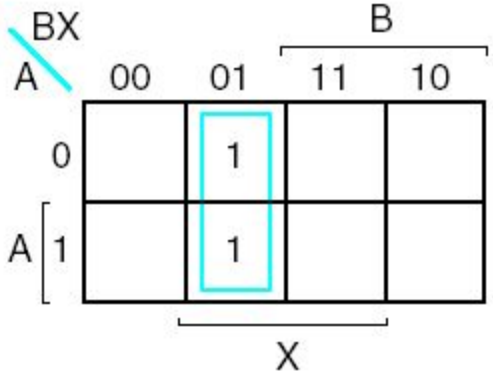
D_A



D_B



Y



Hence, $D_A = AB' + BX'$ $D_B = A'X + B'X + ABX'$

$Y = B'X$

Final logic diagram:

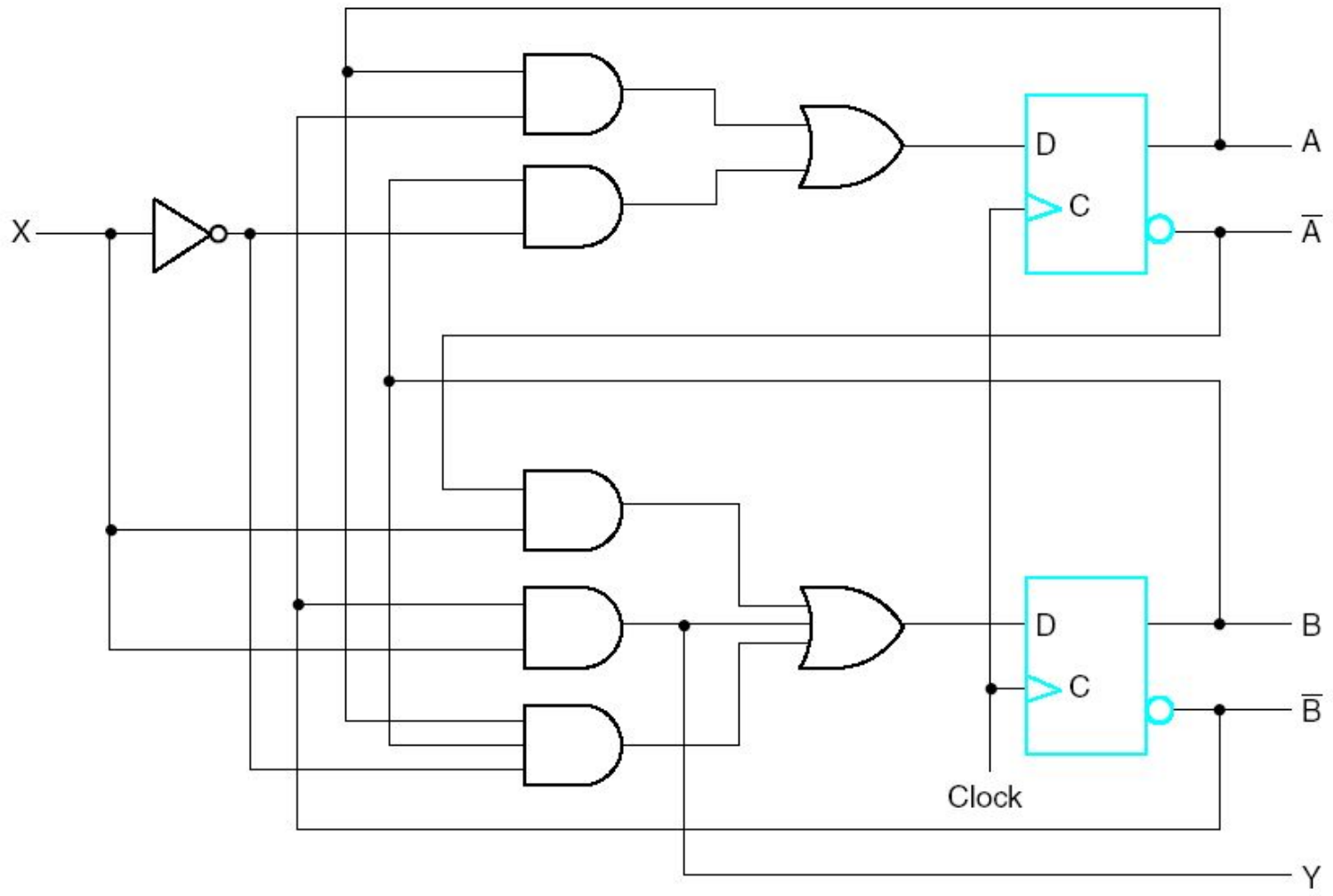


Fig. 4-25 Logic Diagram for Sequential Circuit with *D* Flip-Flops

Remarks

1. Heuristic State Assignment Guidelines

   i. States having the same next state for a given input should be given adjacent assignments.

   ii. States that are next states of a single state should be given adjacent assignments.

   iii. States having the same output for a given input should be given adjacent assignments.

   (2 states are adjacent if they differ in one variable)

Remarks

2. Unused States

- Unused states can be treated as don't care conditions to simplify the circuit
- But, for more robust design it may be desirable to specify the next states and/or output values for the unused states upon a malfunction

3. Start State/Reset State

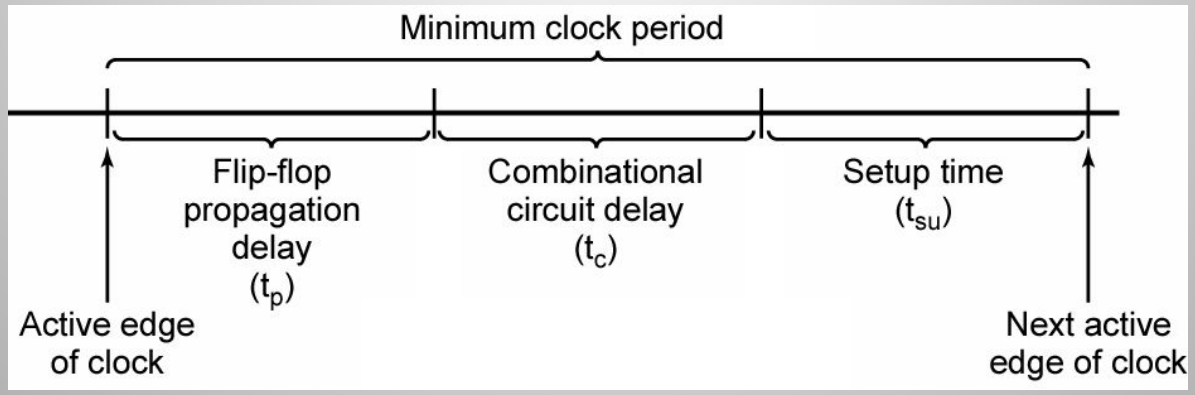
- A master reset signal is usually provided for initializing the flip-flop states

Remarks

4. Timing Issues

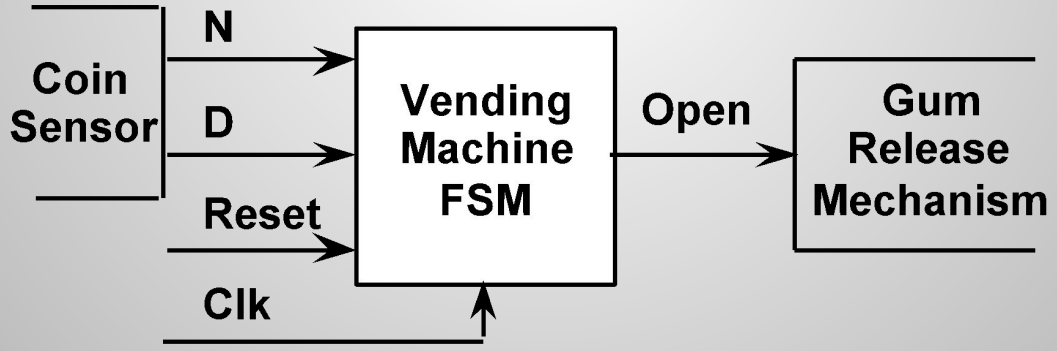
. External input changes must be properly timed with respect to active edge to ensure correct operation

. Must give enough time for signals to be transmitted to flip-flop inputs before we sample the flip-flop inputs



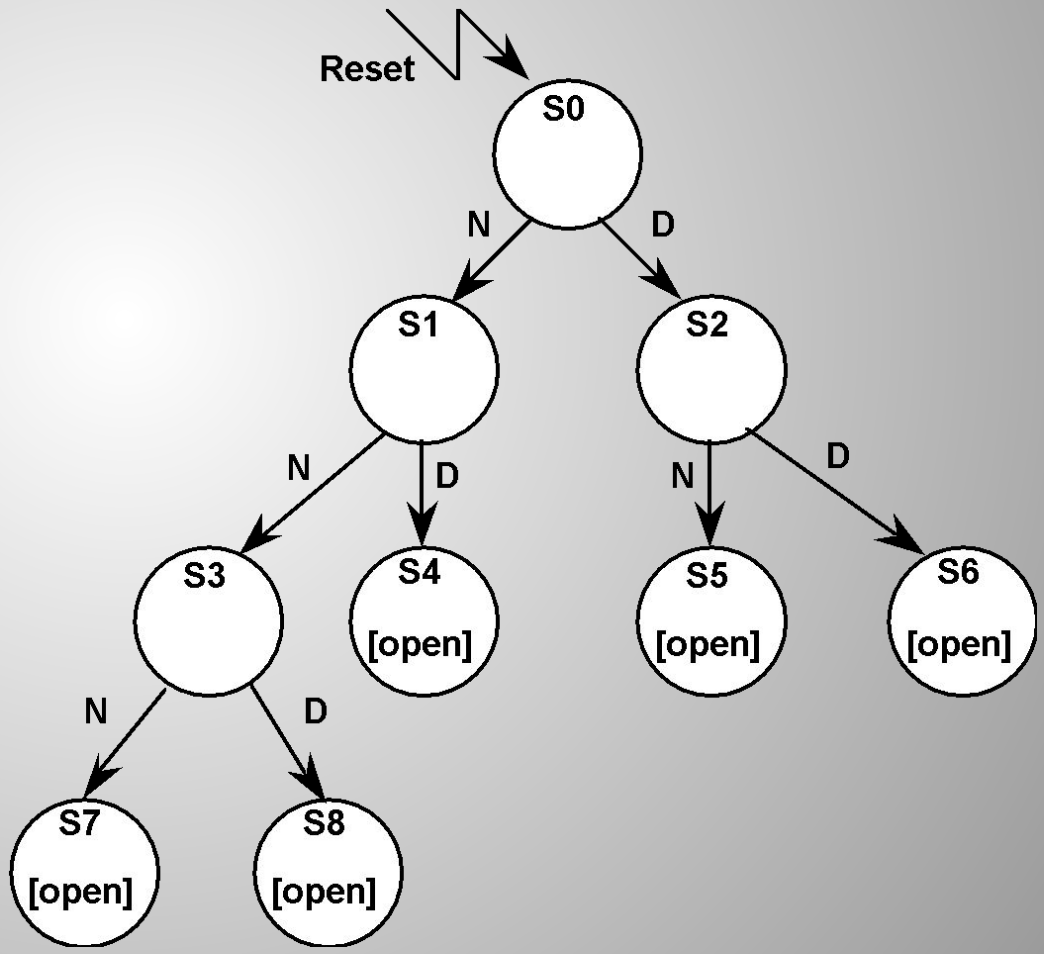
Example: Vending Machine FSM

- Assumptions
  - deliver package of gum after 15 cents deposited
  - single coin slot for dime(10¢), nickel(5¢)
  - no change provided



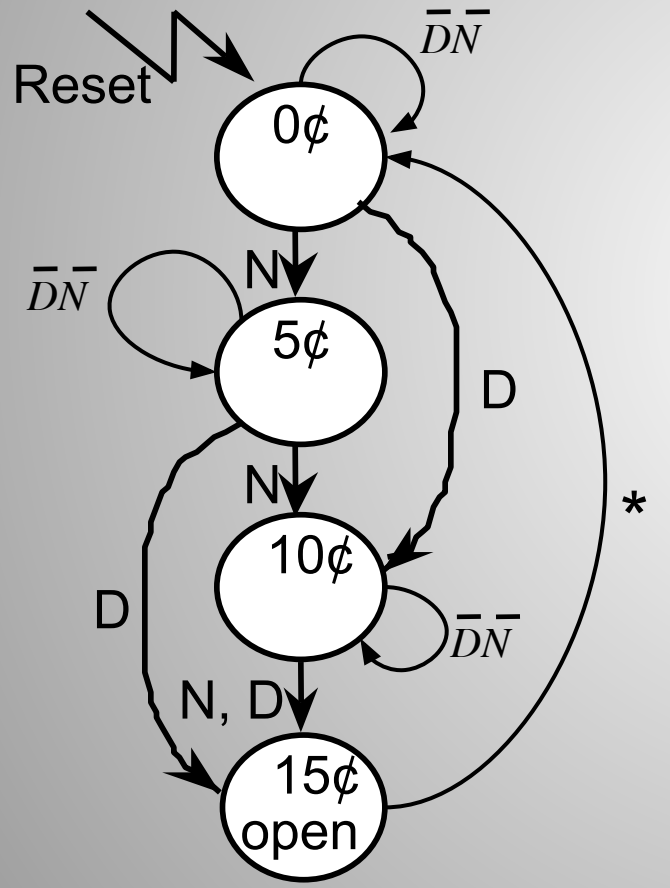
Vending Machine FSM (cont)

- Possible input sequences
  - 3 nickels
  - nickel, then dime
  - dime, then nickel
  - 2 dimes
  - 2 nickels, then dime
- Inputs: N, D, reset
- Output: open



Vending Machine FSM (cont)

State reduction



Present State	Inputs D	Inputs N	Next State	Output Open
0¢	0	0	0¢	0
	0	1	5¢	0
	1	0	10¢	0
	1	1	X	X
5¢	0	0	5¢	0
	0	1	10¢	0
	1	0	15¢	0
	1	1	X	X
10¢	0	0	10¢	0
	0	1	15¢	0
	1	0	15¢	0
	1	1	X	X
15¢	X	X	0¢	1

reuse states whenever possible

Verilog example

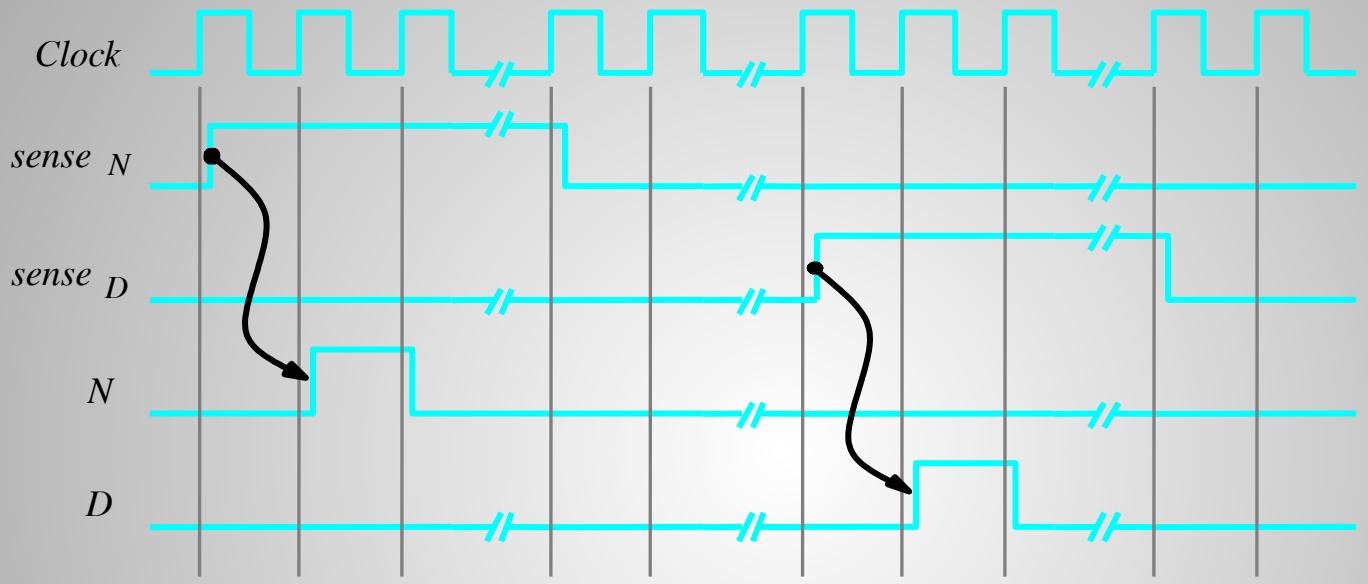
Supplementary Materials

Vending Machine FSM (cont)

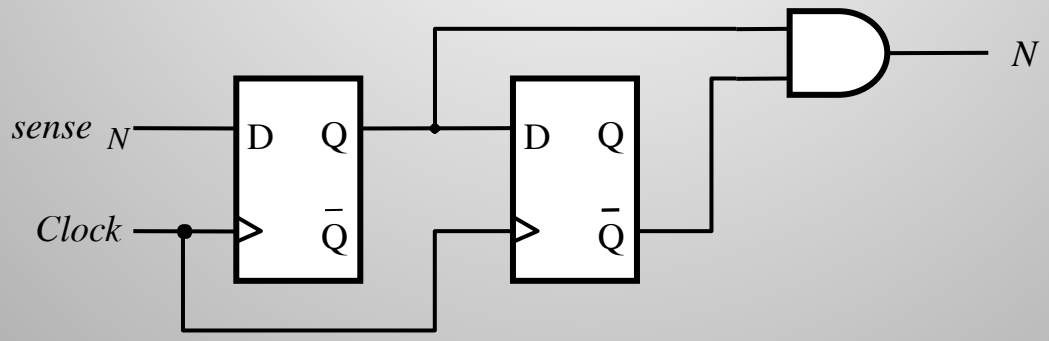
Remarks

- In practice, a vending machine's coin-receptor is a mechanical device and very slow compared to an electronic circuit
- Inserting a dime (nickel) would cause signal $sense_D$ ($sense_N$) to be asserted for a large number of clock cycles. And there may be an arbitrarily long time between insertion of two consecutive coins (see next page)
- So, we need to generate a signal *D(N)* that will be asserted for 1 clock cycle after $sense_D$ ($sense_N$) becomes 1 (see next page)

Vending Machine FSM (cont)



(a) Timing diagram



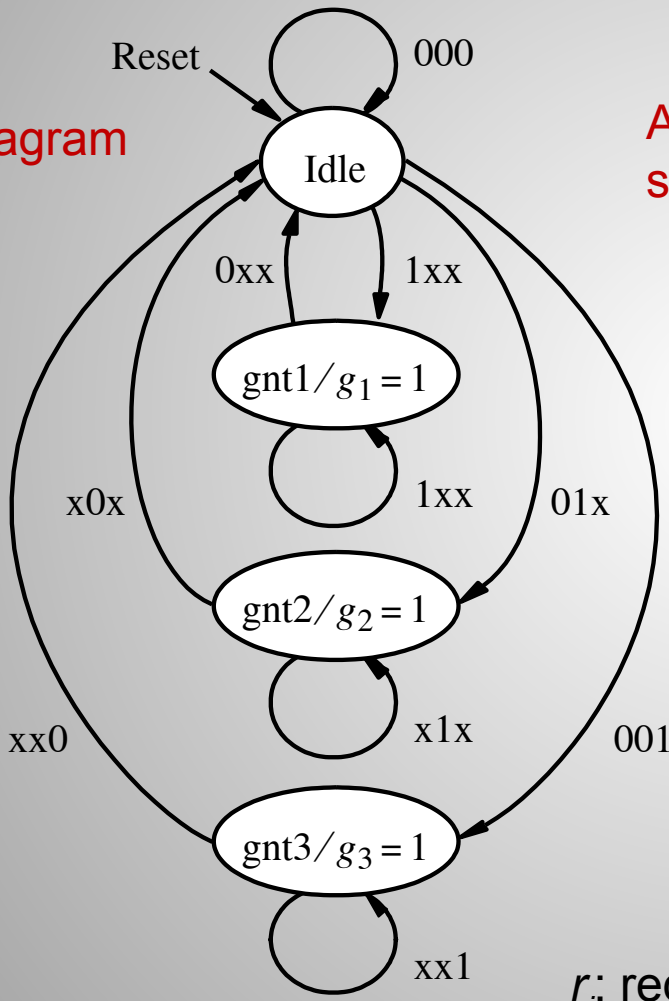
(b) Circuit that generates *N*

Example: Arbiter FSM

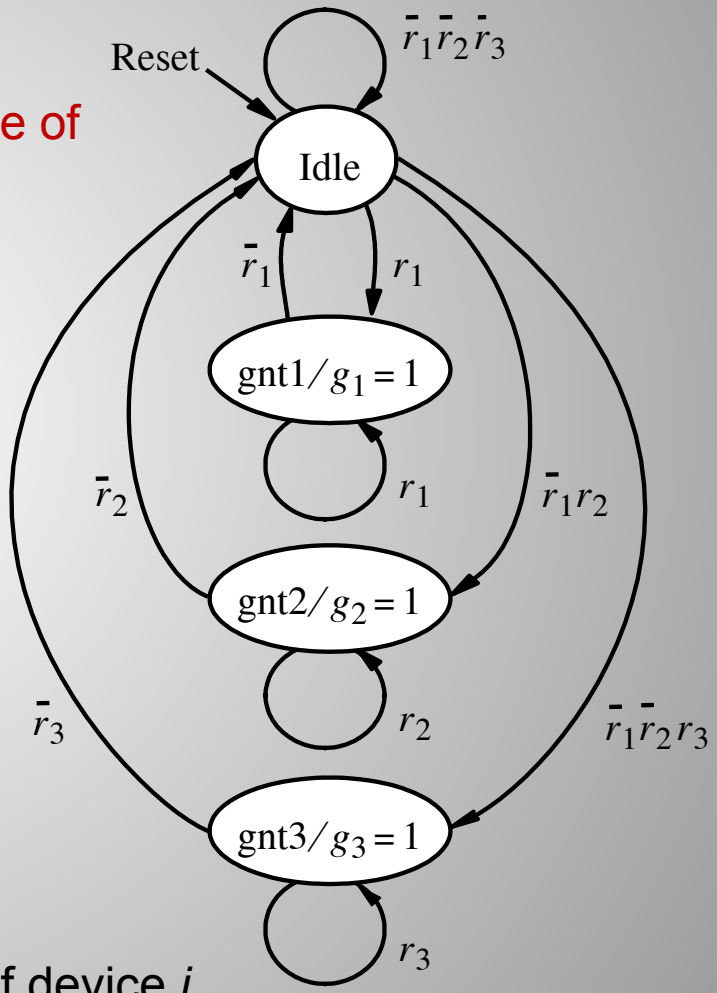
- Control access to a shared resources by different devices in a system
  - Each device provides one input (request) to the arbiter
  - The arbiter provides one output (grant) for each device
  - A device asserts its request signal to request for the resource
  - When the resource is not used, the arbiter selects the highest priority requesting device and asserts its grant signal
  - When a device finished using the resource, it deasserts its request signal
- E.g. Devices 1 to 3 where device 1 has highest priority, followed by device 2

Arbiter FSM

State diagram



Alternative style of state diagram



r_i: request signal of device i
g_i: grant signal for device i

Verilog Code for Arbiter FSM

```
module arbiter (r, Reset, Clock, g);
      input [1:3] r;         // request signals
      input Reset, Clock;
      output wire [1:3] g;      // grant signals
      reg [2:1] state, next;
      parameter Idle = 2'b00, gnt1 = 2'b01, gnt2 = 2'b10, gnt3 = 2'b11;

      // Sequential block
      always @(posedge Clock)
            if (Reset == 1)         state <= Idle;
            else     state <= next;

      //  Define output
      assign g[1] = (state == gnt1);
      assign g[2] = (state == gnt2);
      assign g[3] = (state == gnt3);
```

```
    // Next state combinational circuit
    always @(r, state)
        case (state)
            Idle: casex (r)
                    3'b000: next = Idle;
                    3'b1xx: next = gnt1;
                    3'b01x: next = gnt2;
                    3'b001: next = gnt3;
                    default: next = Idle;
                endcase
            gnt1: if (r[1])  next = gnt1;
                else  next = Idle;
            gnt2: if (r[2])  next = gnt2;
                else  next = Idle;
            gnt3: if (r[3])  next = gnt3;
                else  next = Idle;
            default: next = Idle;
        endcase
endmodule
```

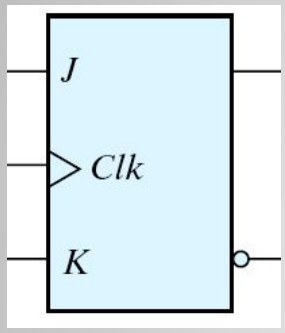
Verilog Code for FSM

- There are more than one way to describe a FSM
- In the sample code above
  - 1st always block introduces flip-flops into the circuit
  - 2nd always block describes the combinational circuit for computing the next state
  - The outputs are defined with conditional assignment statements
- State transition should use non-blocking assignment (<=) as FFs of synchronous sequential circuit are updated concurrently by a common clock

附錄

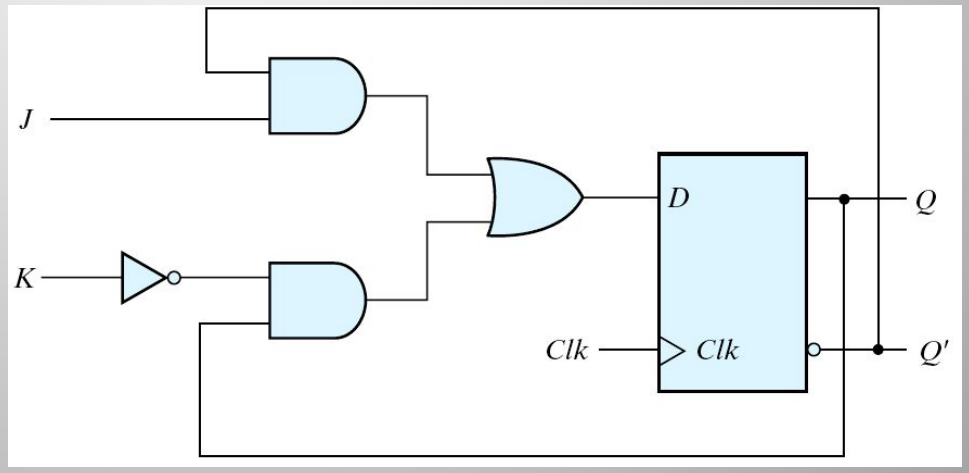
JK Flip-Flop

. E.g. Positive-edge triggered *JK* flip-flop

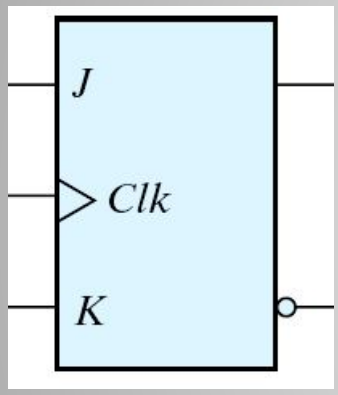


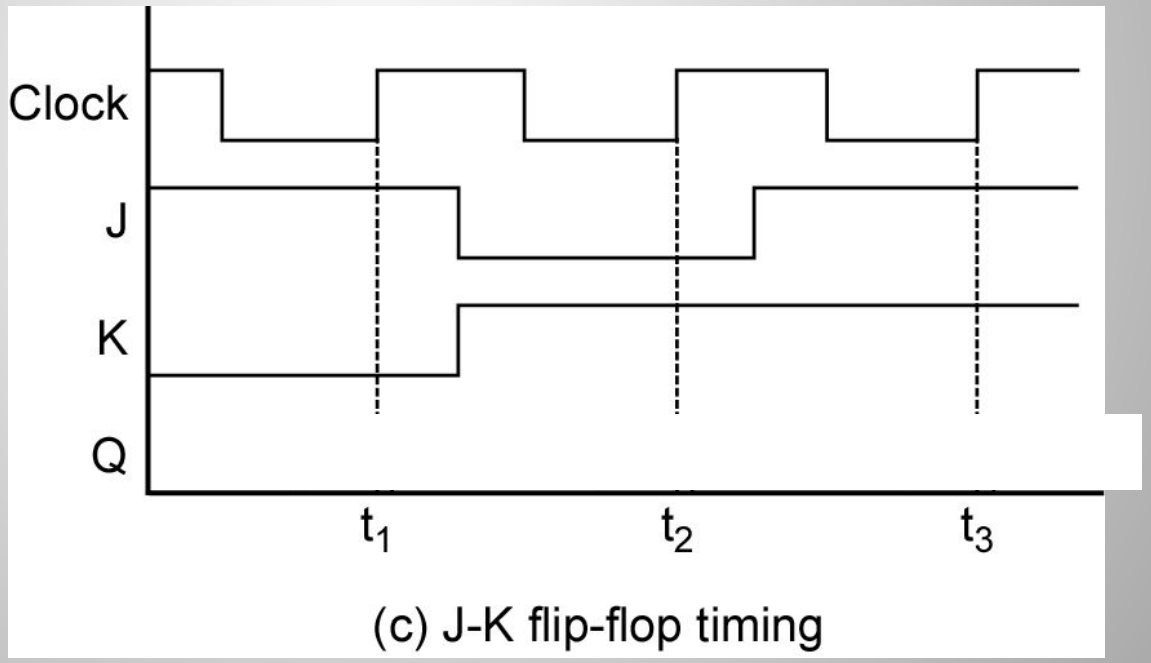
J	K	Q(t + 1)
0	0	$Q(t)$
0	1	0
1	0	1
1	1	$Q'(t)$

From *D* flip-flop:



Exercise

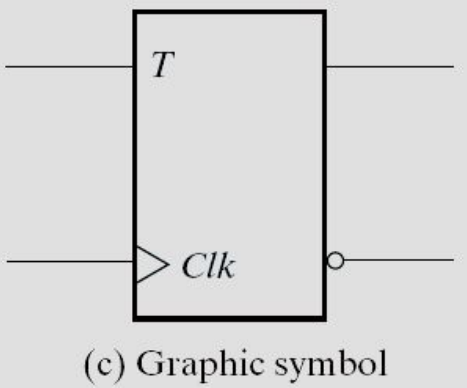




(c) J-K flip-flop timing

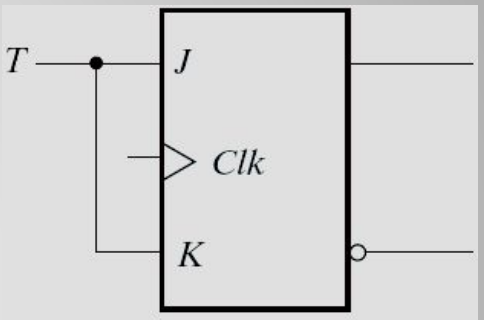
T Flip-Flop

- E.g. Positive-edge triggered *T* flip-flop



(c) Graphic symbol

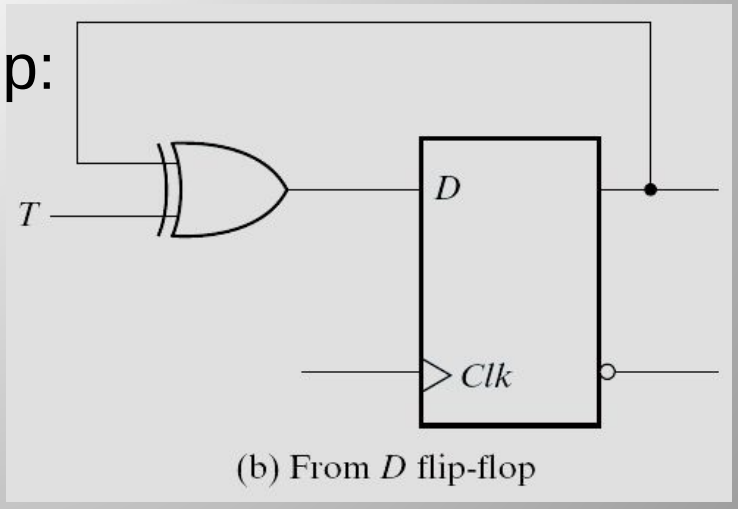
From *JK* flip-flop:



(a) From *JK* flip-flop

T	*Q*(*t* + 1)
0	*Q*(*t*)
1	*Q*′(*t*)

From *D* flip-flop:



(b) From *D* flip-flop

Characteristic Table (Next State Table)

- Show the next state as a function of the inputs and present state

Table 5.1
Flip-Flop Characteristic Tables

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

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

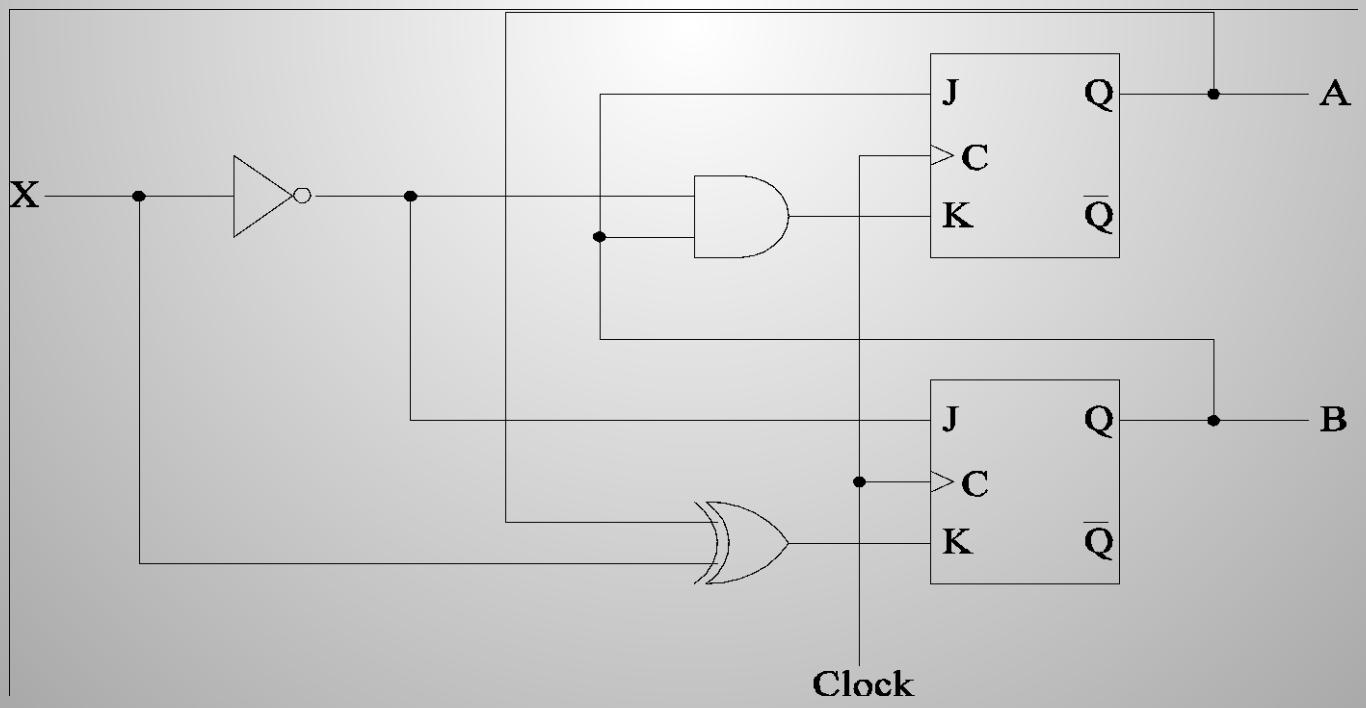
Characteristic Equation

- Characteristic equations
  - *D* flip-flop
    - $Q(t+1) = D$
  - *JK* flip-flop
    - $Q(t+1) = JQ'+K'Q$
  - *T* flop-flop
    - $Q(t+1) = T \oplus Q$

Analysis Example with *JK* flip-flops

. Obtain the flip-flop input equations in terms of the present state and input variables

. Use the corresponding flip-flop characteristic table or equation to determine the next state



1. FF input equations:

$$J_A = B,\ K_A = BX',\ J_B = X',\ K_B = A \oplus X$$

2. Compute the next state

Table 5.4
State Table for Sequential Circuit with JK Flip-Flops

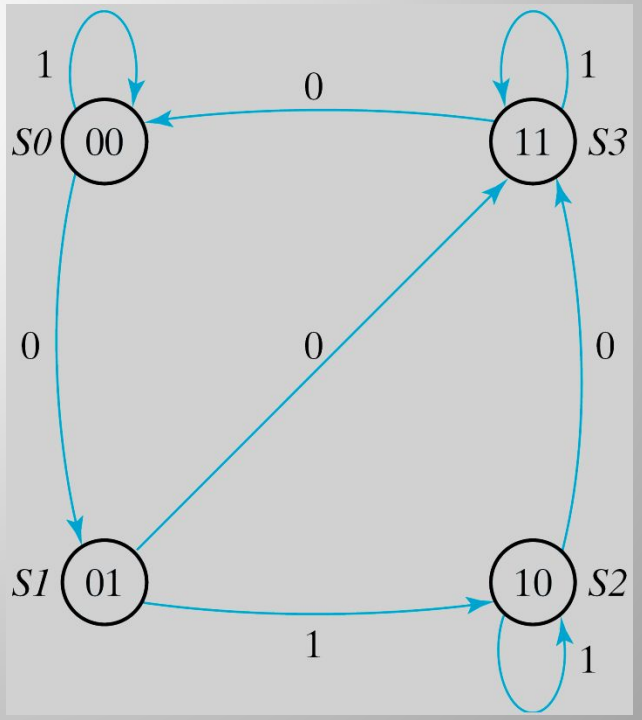
Present State		Input	Next State		Flip-Flop Inputs			
A	B	x	A	B	J_A	K_A	J_B	K_B
0	0	0	0	1	0	0	1	0
0	0	1	0	0	0	0	0	1
0	1	0	1	1	1	1	1	0
0	1	1	1	0	1	0	0	1
1	0	0	1	1	0	0	1	1
1	0	1	1	0	0	0	0	0
1	1	0	0	0	1	1	1	1
1	1	1	1	1	1	0	0	0

. State equations for A and B:

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

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

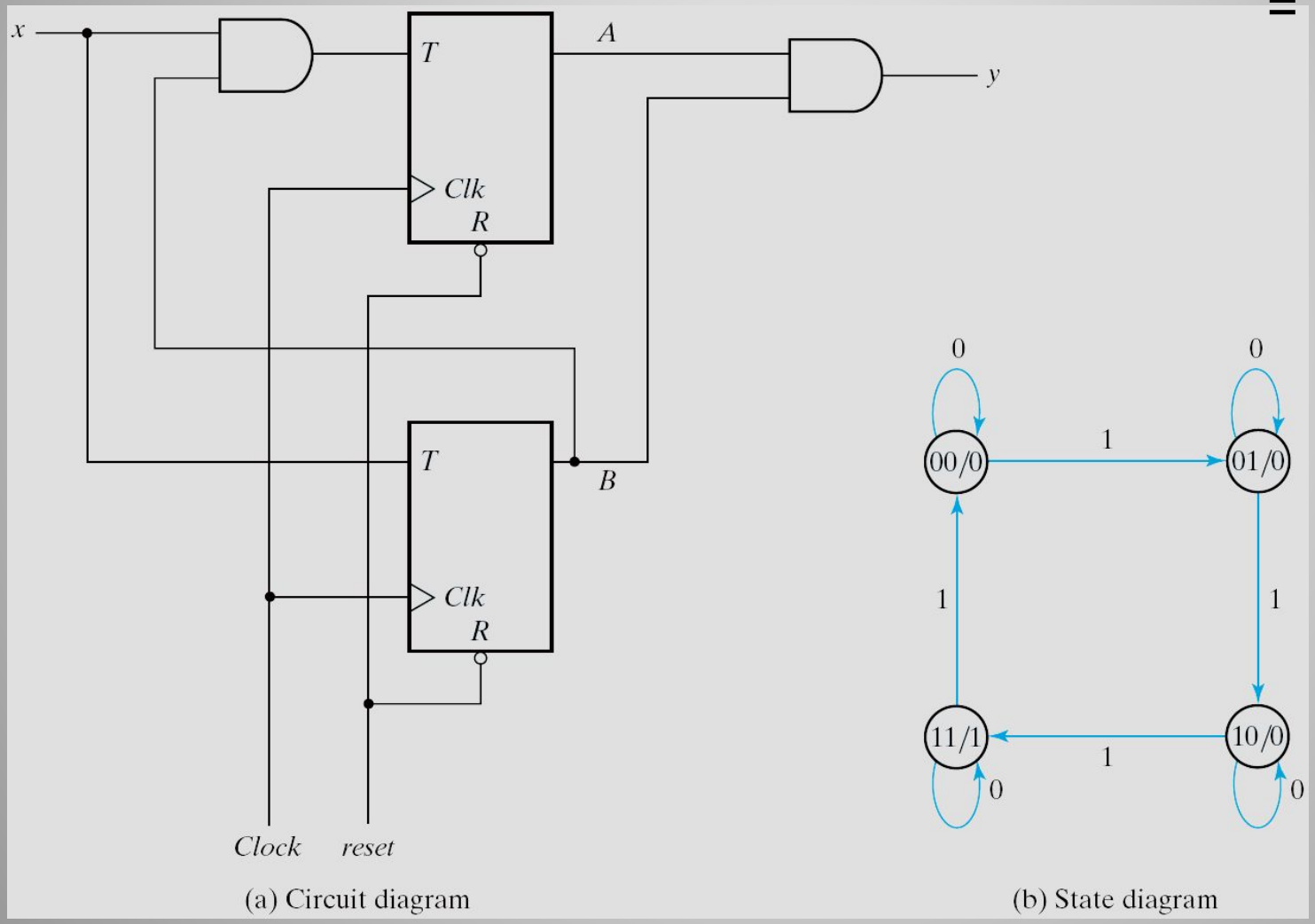
. State transition diagram



Analysis Example with *T* flip-flops

. $Q(t+1) = T \oplus Q$

$= TQ' + T'Q$



(a) Circuit diagram

(b) State diagram

- The input and output functions
  - $T_A = Bx$
  - $T_B = x$
  - $y = AB$
- The state equations
  - $A(t+1) = (Bx)'A + (Bx)A'$
    $= AB' + Ax' + A'Bx$
  - $B(t+1) = x \oplus B$

Table 5.5
State Table for Sequential Circuit with T Flip-Flops

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

Design Using JK Flip-Flops or T Flip-Flops

- Need to know the FF input values to bring about the required state change
- Excitation tables for JK flip-flop and T flip-flop:

$Q(t)$	$Q(t+1)$	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

Design Example with JK flip-flops

Complete the following design with the given state diagram and the given states to binary codes mapping using *JK* flip-flops.

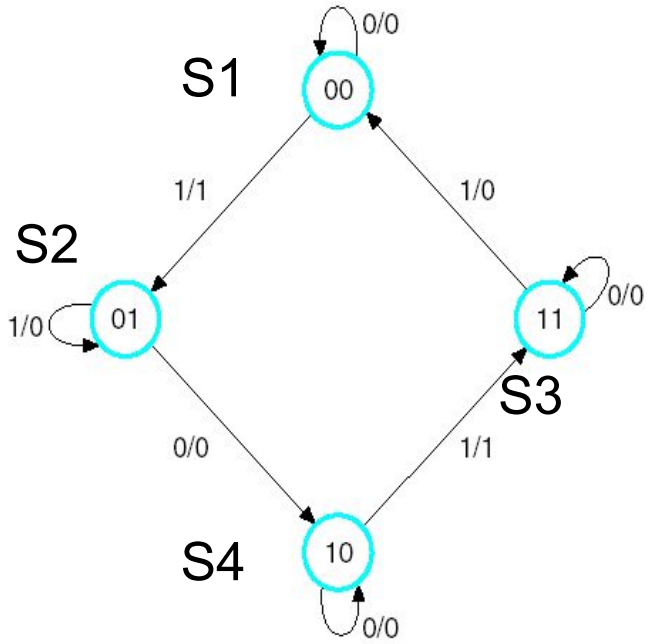


Fig. 4-23 State Diagram for Design Example

TABLE 4-8
State Table for Design Example

Present State		Input	Next State		Output
A	B	X	A	B	Y
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	1	0	0
0	1	1	0	1	0
1	0	0	1	0	0
1	0	1	1	1	1
1	1	0	1	1	0
1	1	1	0	0	0

Table 4-8 State Table for Design Example

Deriving the flip-flop input equations and output function:

Output $Y(A,B,X) = \Sigma m(1,5)$

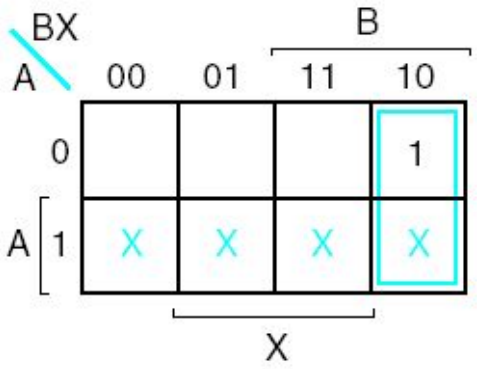
JK flip-flops are used, we derive the required J and K input values:

TABLE 4-11
State Table with *JK* Flip-Flop Inputs

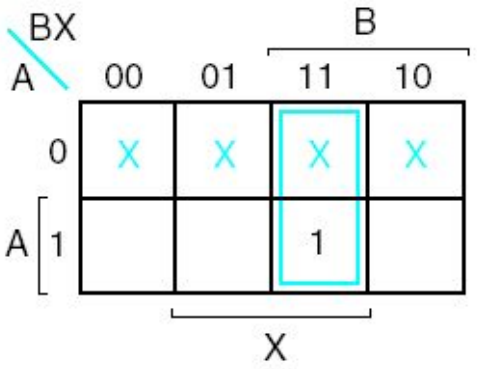
Present State		Input	Next State		Flip-Flop Inputs			
A	B	X	A	B	J_A	K_A	J_B	K_B
0	0	0	0	0	0	X	0	X
0	0	1	0	1	0	X	1	X
0	1	0	1	0	1	X	X	1
0	1	1	0	1	0	X	X	0
1	0	0	1	0	X	0	0	X
1	0	1	1	1	X	0	1	X
1	1	0	1	1	X	0	X	0
1	1	1	0	0	X	1	X	1

Table 4-11 State Table with *JK* Flip-Flop Inputs

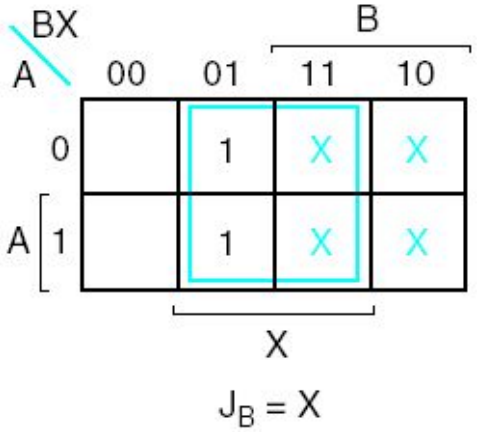
J_A:



K_A:



J_B:



K_B:

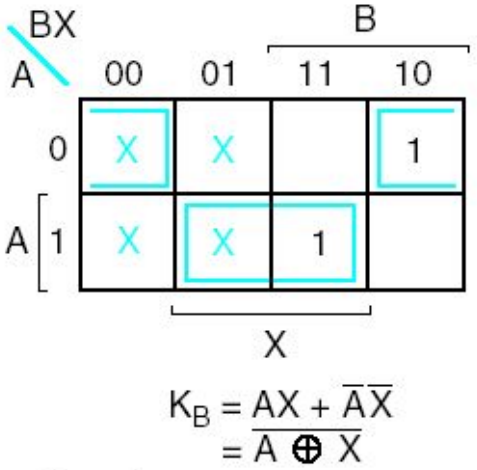


Fig. 4-27 Maps for *J* and *K* Input Equations

Hence, $J_A = BX'$ $K_A = BX$

$J_B = X$ $K_B = (A \oplus X)'$

$Y = B'X$

Final logic diagram:

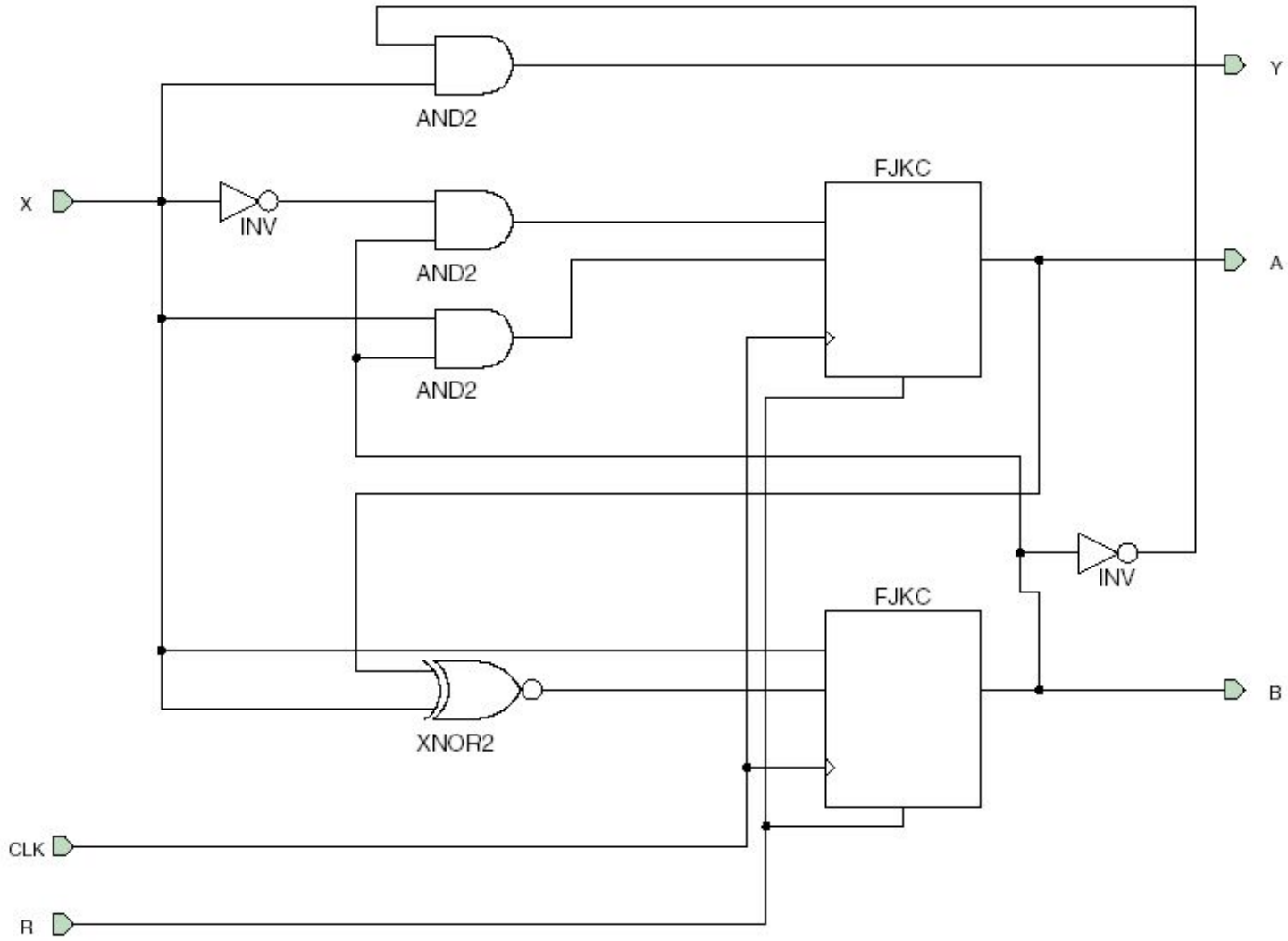
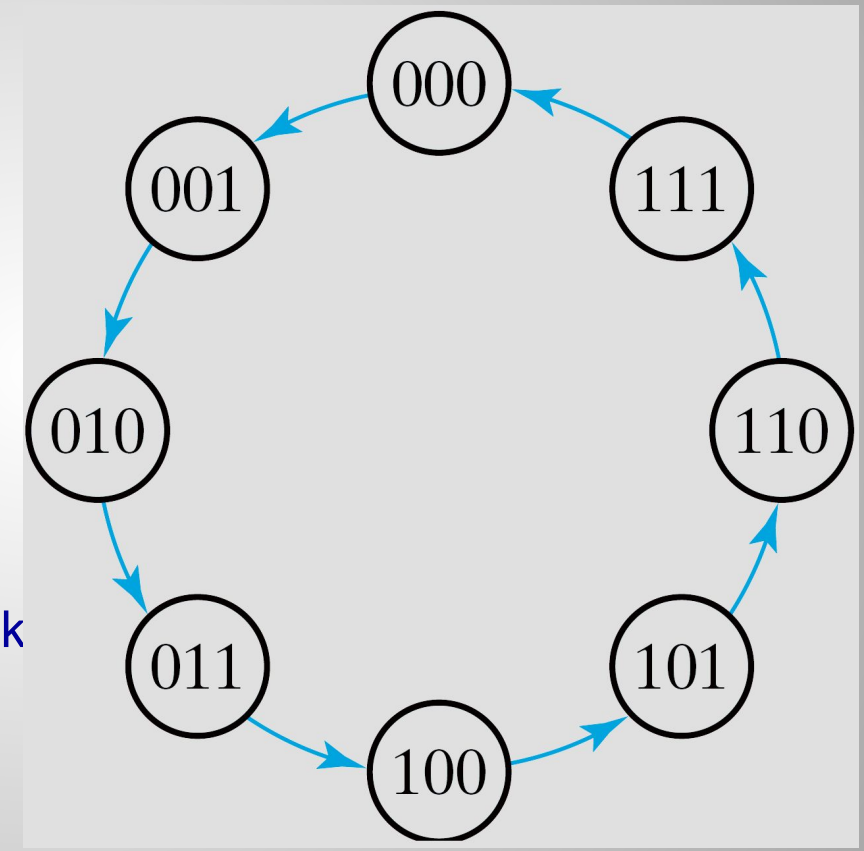


Fig. 4-28 Logic Diagram for Sequential Circuit with *JK* Flip-Flops

Design Example with *T* flip-flops

- A *n*-bit binary counter
  - the state diagram

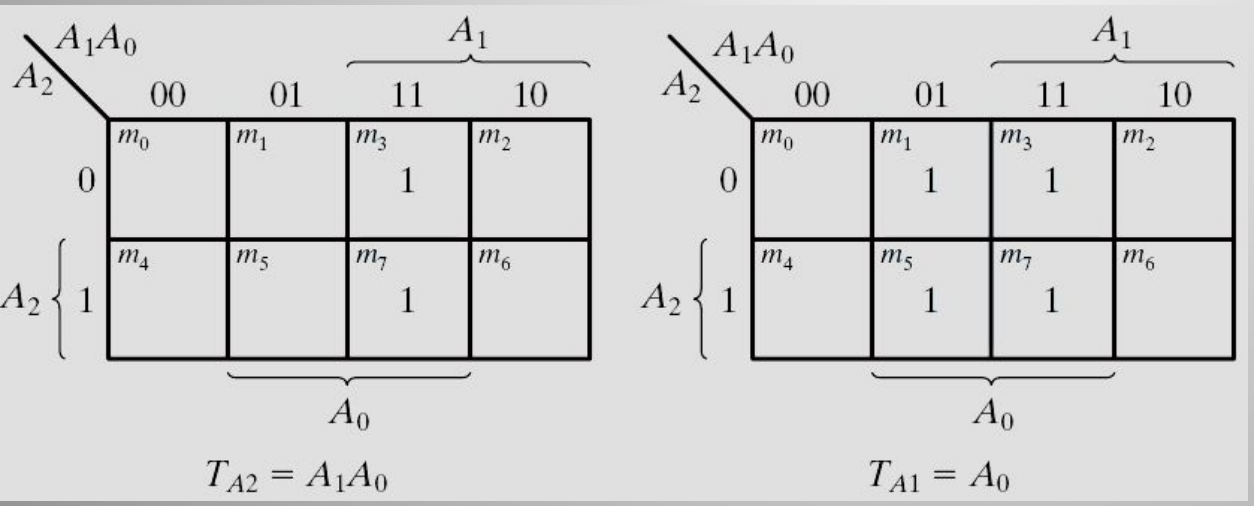


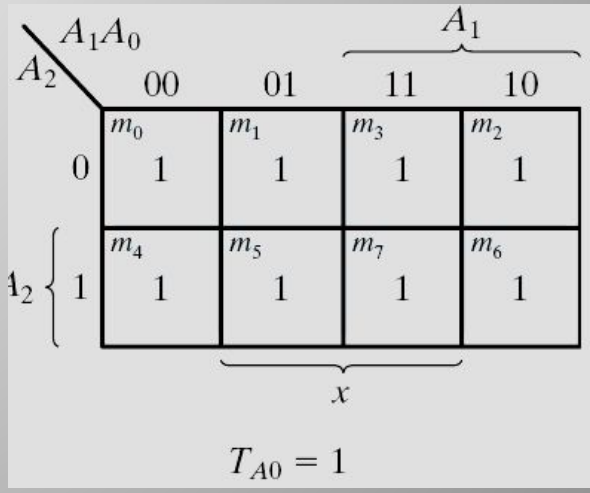
  - no input (except for the clock

. The state table and required flip-flop inputs:

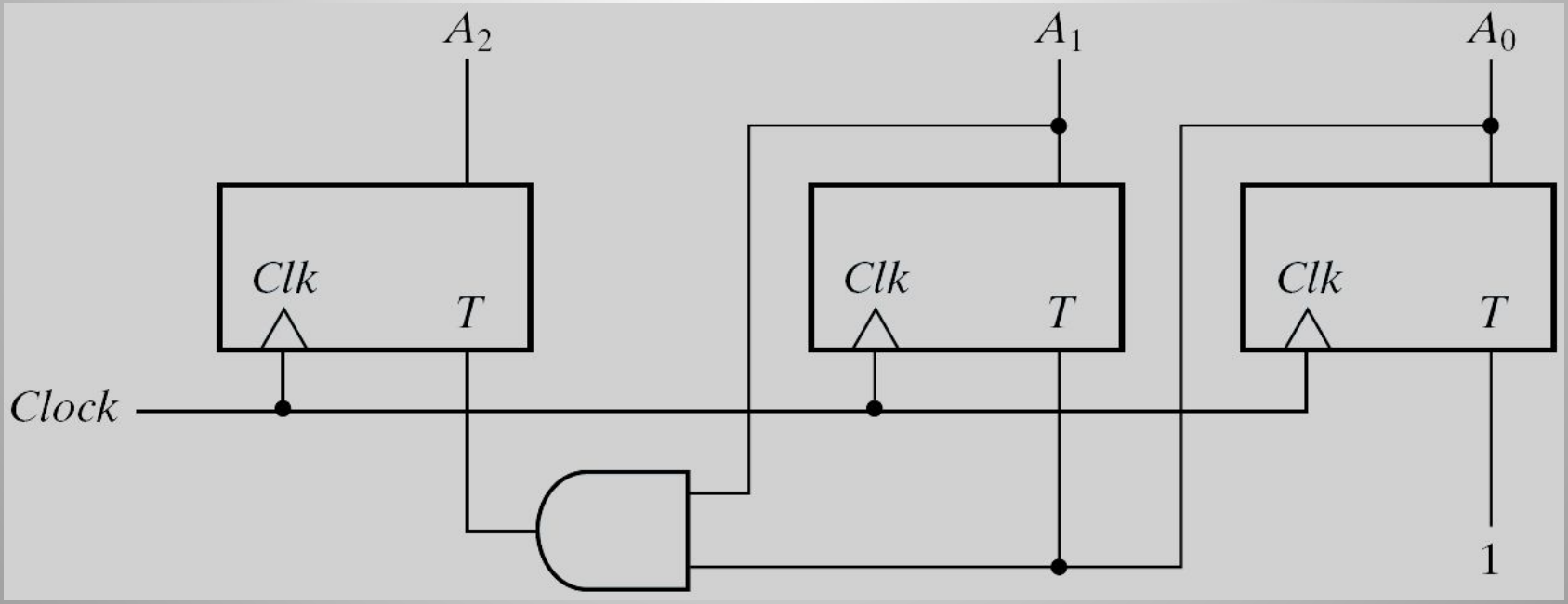
Table 5.14
State Table for Three-Bit Counter

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



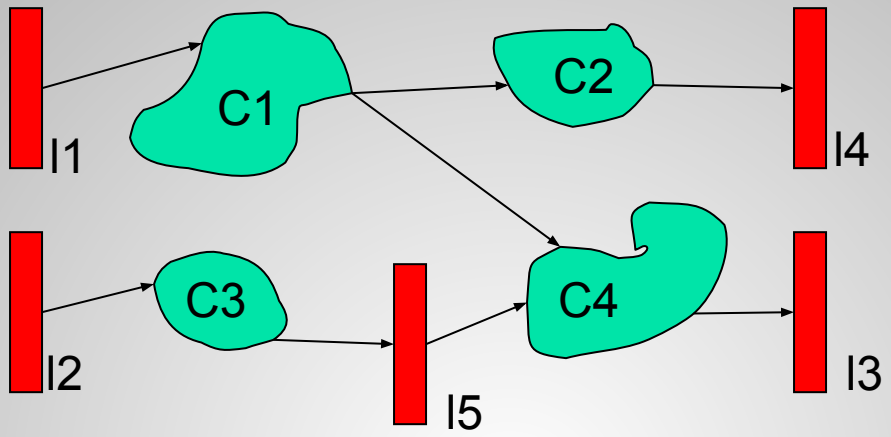


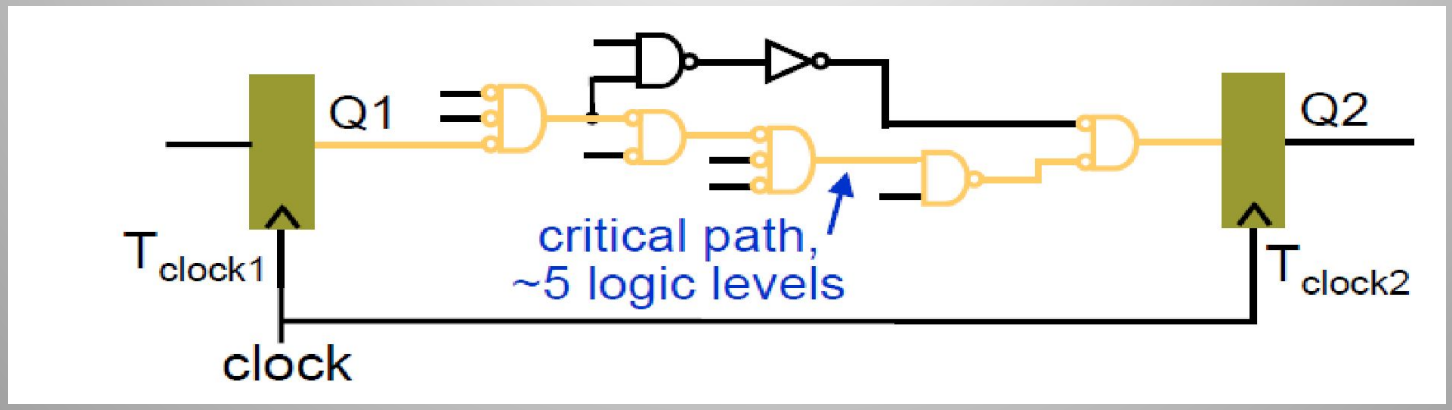
- Logic simplification using the *K* map
  - $T_{A2} = A_1A_0$
  - $T_{A1} = A_0$
  - $T_{A0} = 1$
- The logic diagram



附錄

Sequential networks





Test bench

. https://verilogguide.readthedocs.io/en/latest/verilog/testbench.html

```
`timescale 1ns / 1psmodule
Lab1_Team5_Comparator_4bits_tb;
reg [3:0] A,B;
wire A_lt_B, A_gt_B, A_eq_B;

Comparator_4bits c(A, B, A_lt_B, A_gt_B, A_eq_B);

initial begin
  #10 A=4'b1010;    B=4'b1010;    // edit your own testbench
 $display (“time = %t   A = %b , B = %b , A_lt_B=%b,A_gt_B=%b,A_eq_B=%b",$time, A,B,A_lt_B, A_gt_B, A_eq_B);


#10 A= 4’b0000;   B= 4’b1111;
$display (“time = %t   A = %b , B = %b , A_lt_B=%b,A_gt_B=%b,A_eq_B=%b",$time, A,B,A_lt_B, A_gt_B, A_eq_B);


 $finish;
End

initial begin    $monitor(“time = %t, A = %b , B = %b , A_lt_B=%b,A_gt_B=%b,A_eq_B=%b", $time, A,B,A_lt_B, A_gt_B,
A_eq_B);
End

endmodule
```