

---

# Logic and Computer Design Fundamentals

## Chapter 4 – Sequential Circuits

### Part 2 – Sequential Circuit Design

Ming Cai

cm@zju.edu.cn

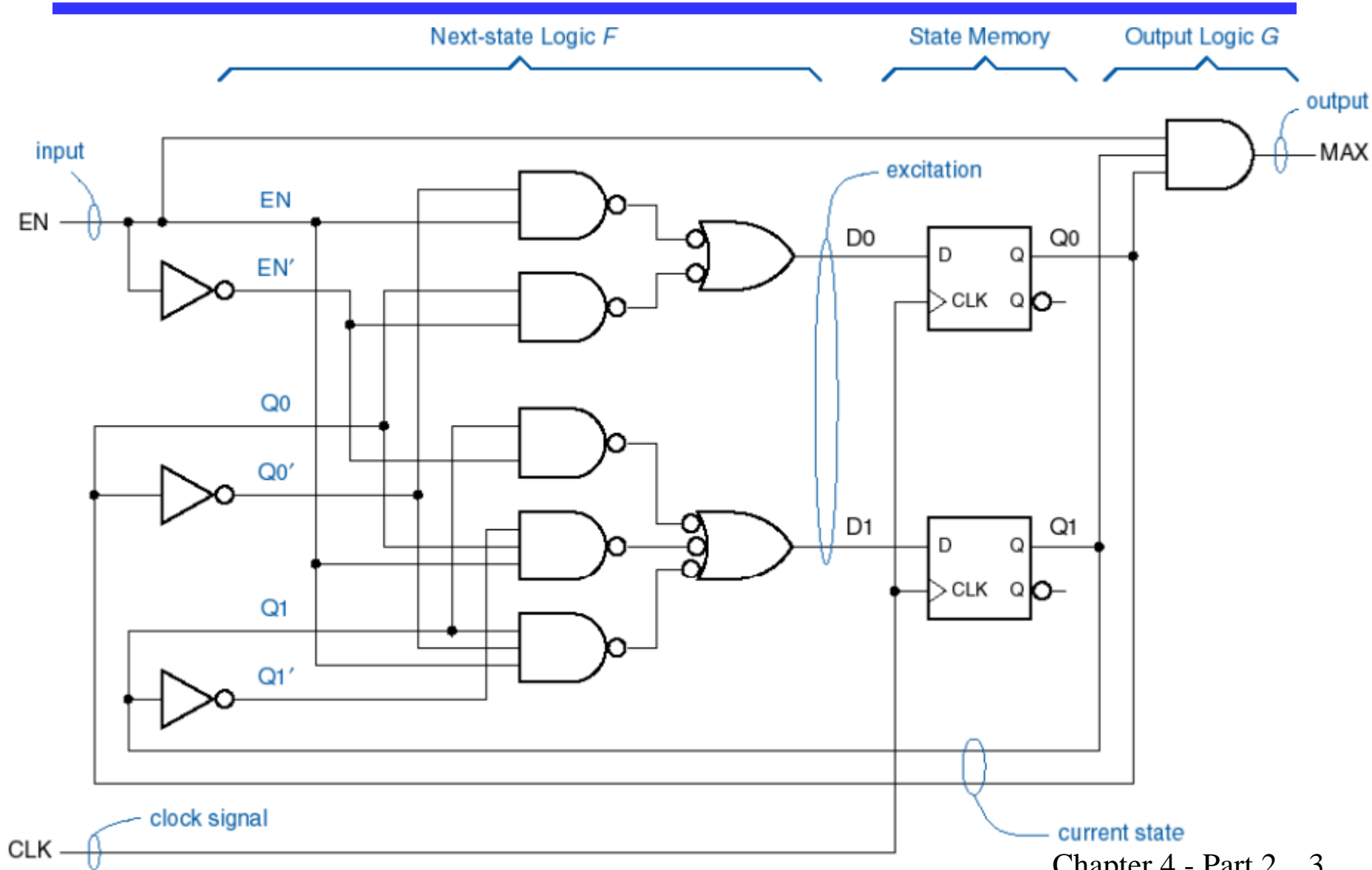
College of Computer Science and Technology,  
Zhejiang University

# Overview

---

- **Part 1 - Storage Elements and Sequential Circuit Analysis**
- **Part 2- Sequential Circuit Design**
  - **Specification**
  - **Formulation**
  - **State Assignment**
  - **Flip-Flop Input and Output Equation Determination**
  - **Verification**
- **Part 3 – State Machine Design**

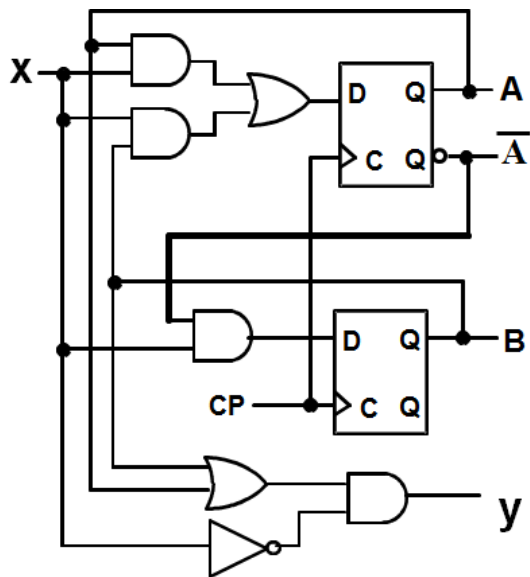
# Methods of Describing Sequential Circuit



# Methods of Describing Sequential Circuit

## Sequential Circuit Analysis

- Input Equation
  - Next State Equation
  - Output Equation
- Next State }  
 } → State Table → State Diagram  
 Output



Sequential Circuit

## Input Equation

- $D_A = A(t)x(t) + B(t)x(t)$

- $D_B = A(t)x(t)$

## Next State Equation

- $A(t+1) = D_A$

- $B(t+1) = D_B$

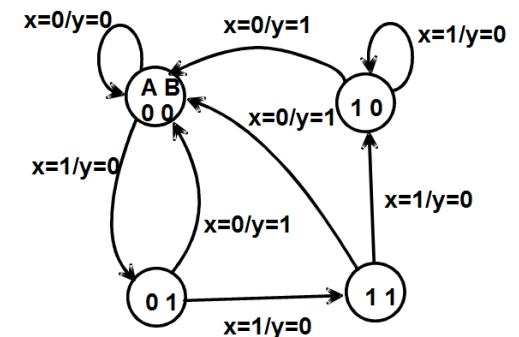
## Output Equation

- $y(t) = x(t)(B(t) + A(t))$

State Table

Present State	Input	Next State	Output
A(t) B(t)	x(t)	A(t+1) B(t+1)	y(t)
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

State Diagram

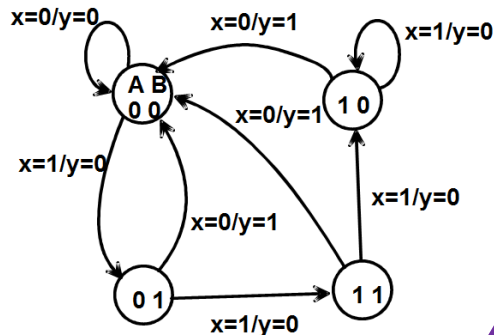


# Methods of Describing Sequential Circuit (continued)

## ■ Sequential Circuit Design

- State Diagram → State Table →  $\left\{ \begin{array}{l} \text{Output Equation} \\ \text{Excitation Equation} \\ \text{Input Equation} \end{array} \right.$

### State Diagram



### State Table

Present State		Input	Next State		Output
A(t)	B(t)	x(t)	A(t+1)	B(t+1)	y(t)
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

## ■ Output Equation

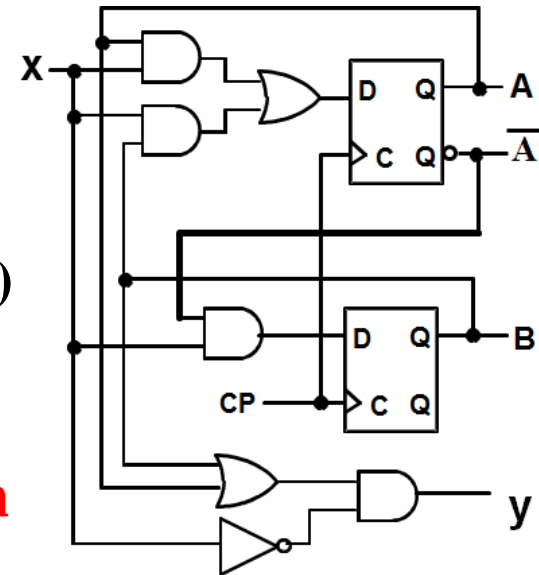
- $y(t) = x(t)(B(t) + A(t))$

## ■ Input Equation

- $D_A = A(t)x(t) + B(t)x(t)$
- $D_B = A(t)x(t)$

## ■ Excitation Equation

- $D_A = A(t+1)$
- $D_B = B(t+1)$



### Sequential Circuit

# The Design Procedure

---

- **Specification**
- **Formulation** - Obtain a state diagram or state table
- **State Minimization** - Minimize the number of states
- **State Assignment** - Assign binary codes to the states
- **Flip-Flop Input Equation Determination** - Select flip-flop types and derive flip-flop equations from next state entries in the table
- **Output Equation Determination** - Derive output equations from output entries in the table
- **Optimization** - Optimize the equations
- **Technology Mapping** - Find circuit from equations and map to flip-flops and gate technology
- **Verification** - Verify correctness of final design

# Specification

---

- **Component Forms of Specification**
  - **Written description**
  - **Mathematical description**
  - **Hardware description language\***
  - **Tabular description\***
  - **Equation description\***
  - **Diagram describing operation (not just structure)\***
- **Relation to Formulation**
  - **If a specification is rigorous at the binary level (marked with \* above), then all or part of formulation may be completed**

# Formulation: Finding a State Diagram

---

- A state is an abstraction of the history of the past applied inputs to the circuit (including power-up reset or system reset).
  - The interpretation of “past inputs” is tied to the synchronous operation of the circuit. E. g., an input value (other than an asynchronous reset) is measured only during the setup-hold time interval for an edge-triggered flip-flop.
- **Examples:**
  - State A represents the fact that a 1 input has occurred among the past inputs.
  - State B represents the fact that a 0 followed by a 1 have occurred as the most recent past two inputs.



# Formulation: Finding a State Diagram

---

- In specifying a circuit, we use states to remember meaningful properties of past input sequences that are essential to predicting future output values.
- A sequence recognizer is a sequential circuit that produces a distinct output value whenever a prescribed pattern of input symbols occur in sequence, i.e., recognizes an input sequence occurrence.
- We will develop a procedure specific to sequence recognizers to convert a problem statement into a state diagram.
- Next, the state diagram, will be converted to a state table from which the circuit will be designed.

# State Reduction (State Minimization)

---

- By reducing or minimizing the total number of states, the number of flip-flops required for a design is also reduced.
- For example if a finite state machine drops from 8 states to 4 states, only two flip-flops are required rather than three.
- The total number of states is reduced by eliminating the **equivalent states**.
- Two states are equivalent if they have the **same output** for all inputs, and if they transit to **equivalent states** on all inputs.

# State Assignment

---

- Each of the  $m$  states must be assigned a unique code.
- Minimum number of bits required is  $n$  such that
$$n \geq \lceil \log_2 m \rceil$$
where  $\lceil x \rceil$  is the smallest integer  $\geq x$
- There are useful state assignments that use more than the minimum number of bits
- There are  $2^n - m$  unused states

# Sequence Recognizer Procedure

---

- To develop a sequence recognizer state diagram:

- Proper sequence**
  - Begin in an **initial state** in which NONE of the initial portion of the sequence has occurred (typically “reset” state).
  - **Add a state** that recognizes that the first symbol has occurred.
  - **Add states** that recognize each successive symbol occurring.
- Improper sequence**
  - The **final state** represents the input sequence (possibly less the final input value) occurrence.
  - **Add state transition arcs** which specify what happens when a symbol *not* in the proper sequence has occurred.
  - **Add other arcs** on non-sequence inputs which transition to states that represent the **input subsequence** that has occurred.
- The last step is required because the circuit must recognize the input sequence *regardless of where it occurs within the overall sequence applied since “reset.”*

# Sequence Recognizer Example

---

- **Example: Recognize the sequence 1101**
  - Note that the sequence 111101 contains 1101 and "11" is a proper sub-sequence of the sequence.
- Thus, the sequential machine must remember that the first two one's have occurred as it receives another symbol.
- Also, the sequence 1101101 contains 1101 as both an initial subsequence and a final subsequence with some **overlap**, i. e., **1101**101 or 110**1101**.
- And, the 1 in the middle, 1101101, is in both subsequences.
- The sequence 1101 must be recognized each time it occurs in the input sequence.

# Example: Recognize 1101

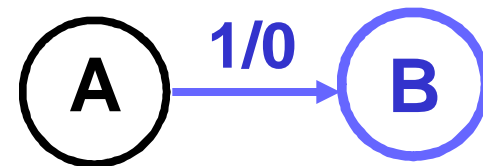
---

- **Define states for the sequence to be recognized:**

- assuming it starts with first symbol,
- continues through each symbol in the sequence to be recognized, and
- uses output 1 to mean the full sequence has occurred,
- with output 0 otherwise.

- **Starting in the initial state (Arbitrarily named "A"):**

- Add a state that recognizes the first "1."
- State "A" is the initial state, and state "B" is the state which represents the fact that the "first" one in the input subsequence has occurred. The output symbol "0" means that the full recognized sequence has not yet occurred.

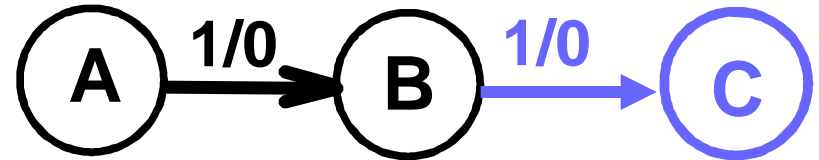


# Example: Recognize 1101 (continued)

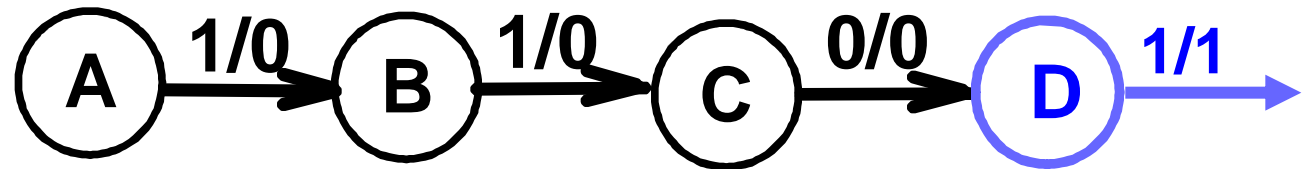
## —— Proper sequence

- After one more 1, we have:

- C is the state obtained when the input sequence has two "1"s.



- Finally, after 110 and a 1, we have:

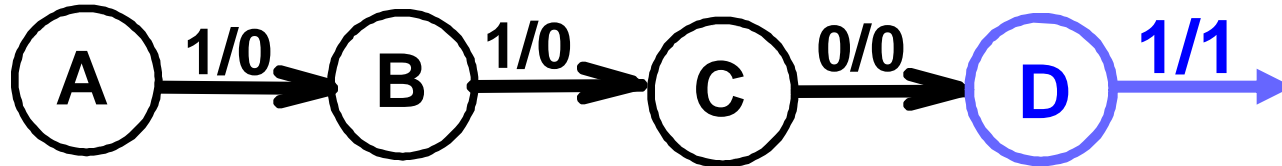


- Transition arcs are used to denote the output function (Mealy Model)
- Output 1 on the arc from D means the sequence has been recognized
- To what state should the arc from state D go? Remember: 1101101 ?
- Note that D is the last state but the output 1 occurs for the input applied in D. This is the case when a *Mealy model* is assumed.

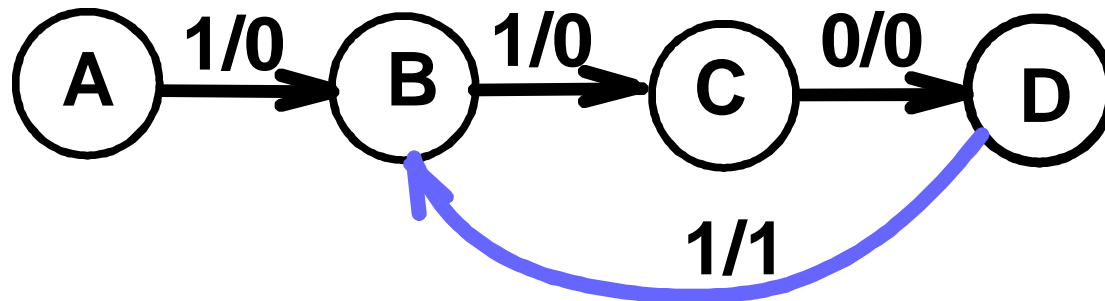
# Example: Recognize 1101 (continued)

## —— Proper sequence

---



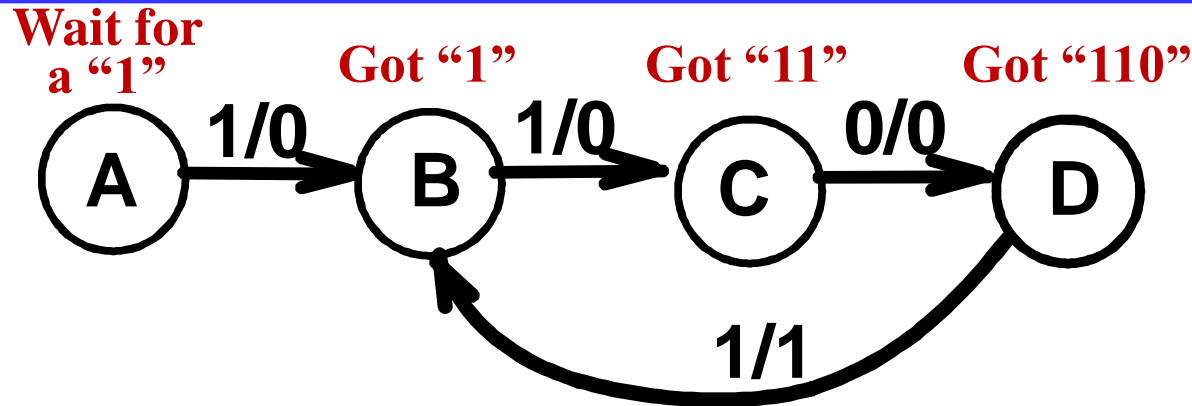
- Clearly the final 1 in the recognized sequence 1101 is a sub-sequence of 1101. It follows a 0 which is not a sub-sequence of 1101. Thus it should represent *the same state reached from the initial state after a first 1 is observed*. We obtain:





# Example: Recognize 1101 (continued)

## —— Proper sequence



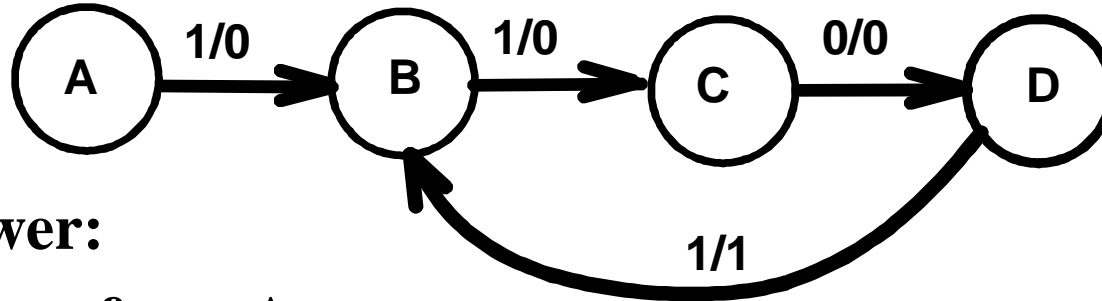
- The state have the following abstract meanings:
  - A: No proper sub-sequence of the sequence has occurred.
  - B: The sub-sequence 1 has occurred.
  - C: The sub-sequence 11 has occurred.
  - D: The sub-sequence 110 has occurred.
  - The 1/1 on the arc from D to B means that the last 1 has occurred and thus, the sequence is recognized.

# Example: Recognize 1101 (continued)

## —Improper sequence

---

- The other arcs are added to each state for inputs not yet listed. Which arcs are missing?



- Answer:

"0" arc from A

"0" arc from B

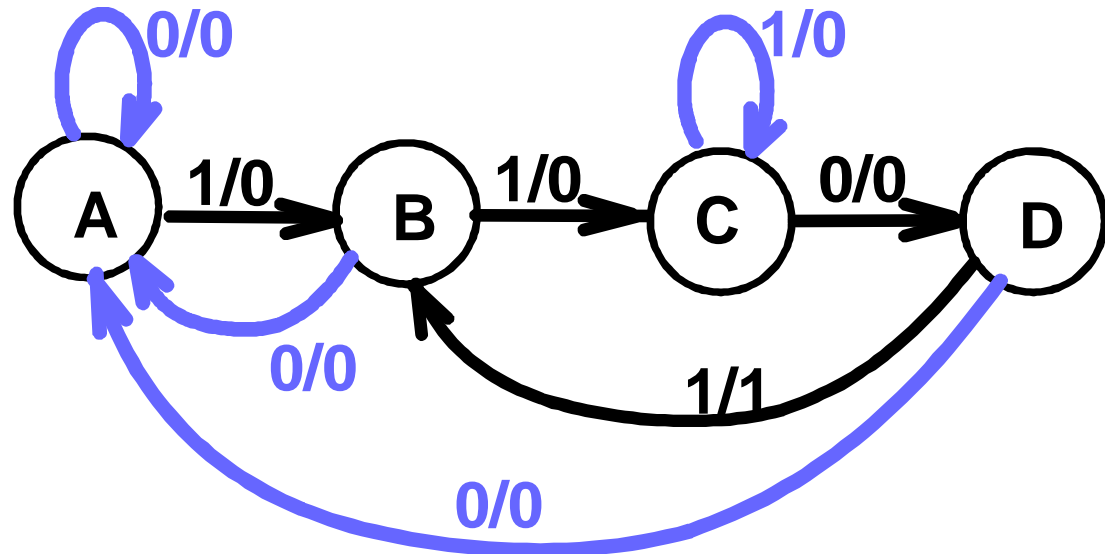
"1" arc from C

"0" arc from D.

# Example: Recognize 1101 (continued)

## ——Complete state diagram

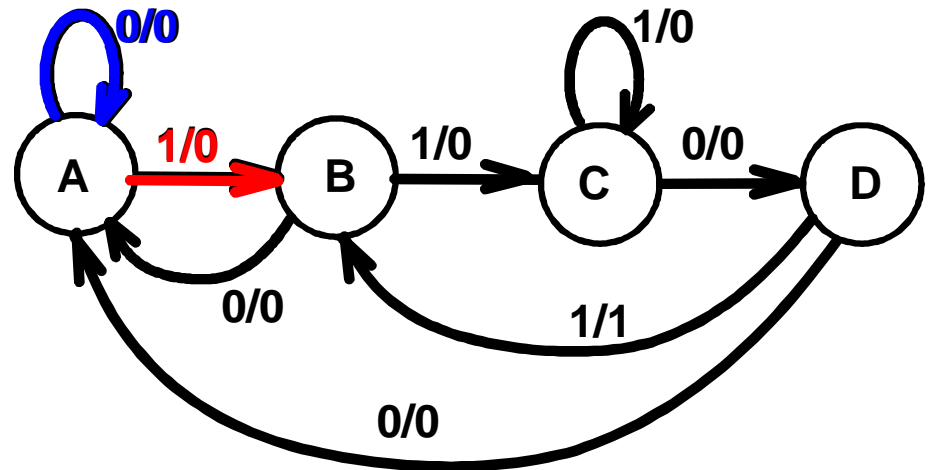
- State transition arcs must represent the fact that an input subsequence has occurred. Thus we get:



- Note that the 1 arc from state C to state C implies that State C means *two or more 1's have occurred*.

# Formulation: Find State Table

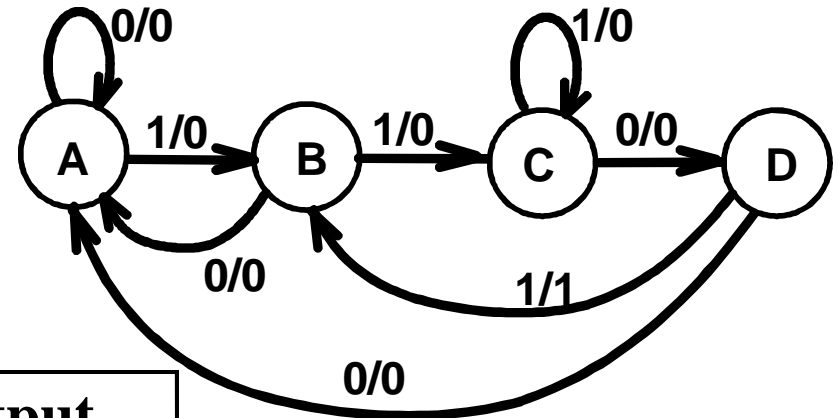
- From the State Diagram, we can fill in the State Table.
- There are 4 states, one input, and one output. We will choose the form with four rows, one for each current state.
- From State A, the 0 and 1 input transitions have been filled in along with the outputs.



Present State	Next State		Output	
	x=0	x=1	x=0	x=1
A	A	B	0	0
B				
C				
D				

# Formulation: Find State Table

- From the state diagram, we complete the state table.



Present State	Next State		Output	
	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

- What would the state diagram and state table look like for the **Moore model**?

# Example: Moore Model for Sequence 1101

---

- For the Moore Model, outputs are associated with states.
- We need to add a state "E" with output value 1 for the final 1 in the recognized input sequence.
  - This new state E, though similar to B, would generate an output of 1 and thus be different from B.
- The Moore model for a sequence recognizer usually *has more states* than the Mealy model.

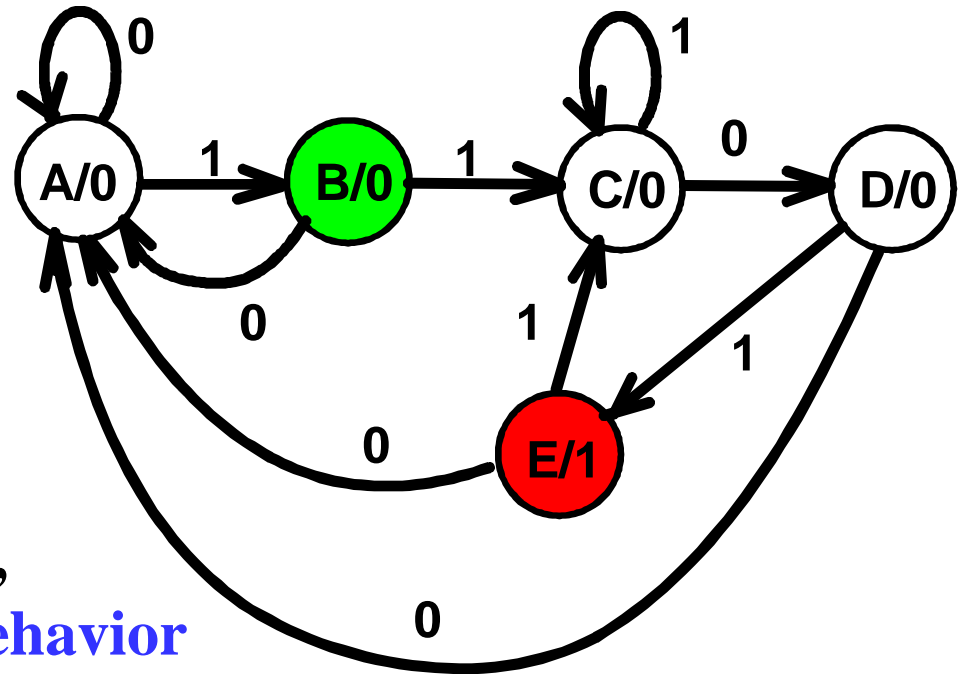
# Example: Moore Model (continued)

- We mark outputs on states for Moore model

- Arcs now show only state transitions

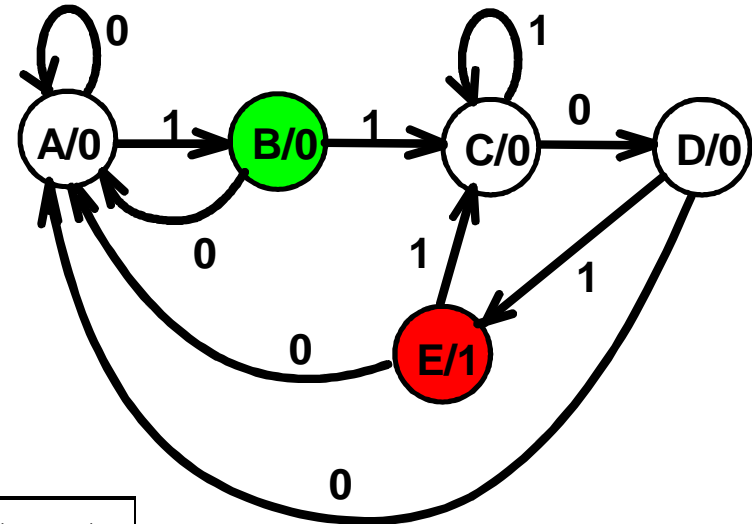
- Add a new state E to produce the output 1

- Note that the new state, **E produces the same behavior in the future as state B**. But it gives a **different output at the present time**. Thus these states do represent a *different abstraction* of the input history.



# Example: Moore Model (continued)

- The state table is shown below
- In the Moore model:  
“Moore is More.”



Present State	Next State		Output y
	x=0	x=1	
A	A	B	0
B	A	C	0
C	D	C	0
D	A	E	0
E	A	C	1

- Multiple-sequence Detector Example (Appendix A)



# Simplification of State Table

---

- Generally, original state table may contain some **redundant states**, which will increase the cost of the design
- Simplification of State Table: Get a minimized state table, which can satisfy the design requirement with least number of states

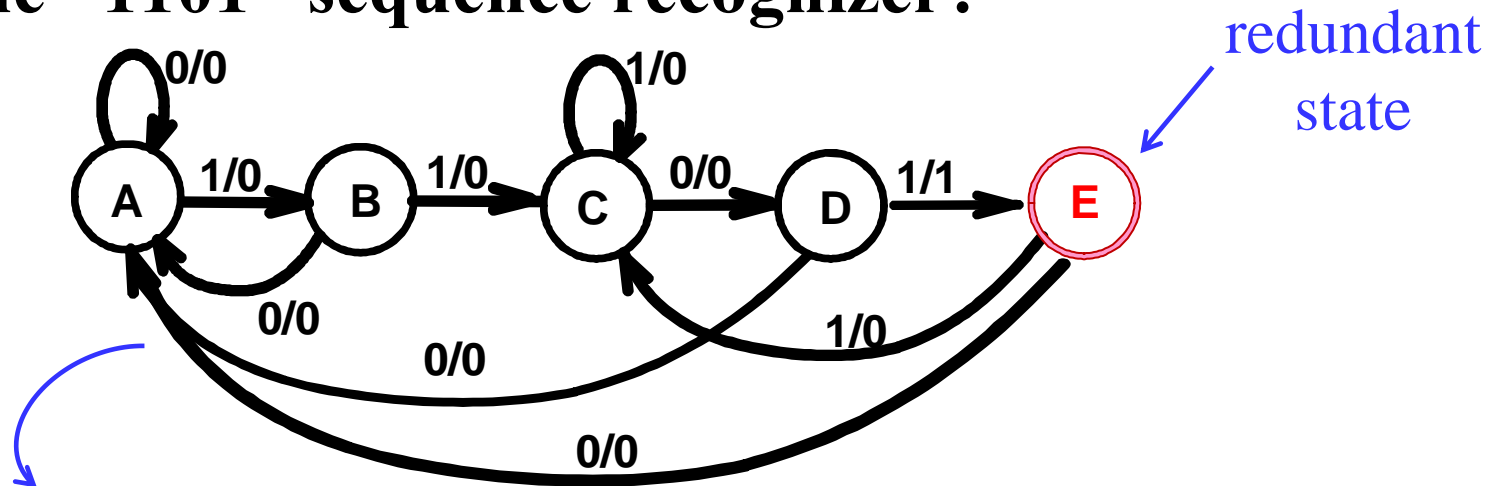
# Equivalent States

---

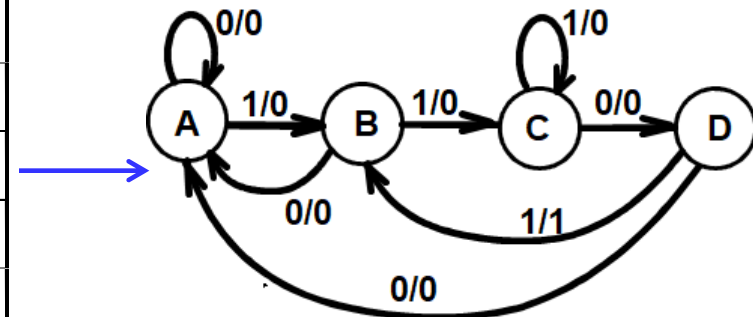
- **Complete State Table:** All next states and outputs in the state table have certain state and value.
- **Equivalent States:** Assume S1 and S2 are two states in the complete state table, and begin with S1 and S2, if we have exactly **same output sequence for any input sequence**, S1 and S2 are called equivalent states, denoted (S1, S2). **Equivalent states can be merged.** Here, “any input sequence” means that the length and structure of the sequence can be arbitrary.

# Simplification of State Table by Observation

- How to remove a redundant state introduced by the “1101” sequence recognizer?



Present State	Next State		Output	
	x=0	x=1	x=0	x=1
A	A	B	0	0
<b>B</b>	<b>A</b>	<b>C</b>	<b>0</b>	<b>0</b>
C	D	C	0	0
D	A	<b>E</b>	0	1
<b>E</b>	<b>A</b>	<b>C</b>	<b>0</b>	<b>0</b>



# Discrimination of Equivalent States

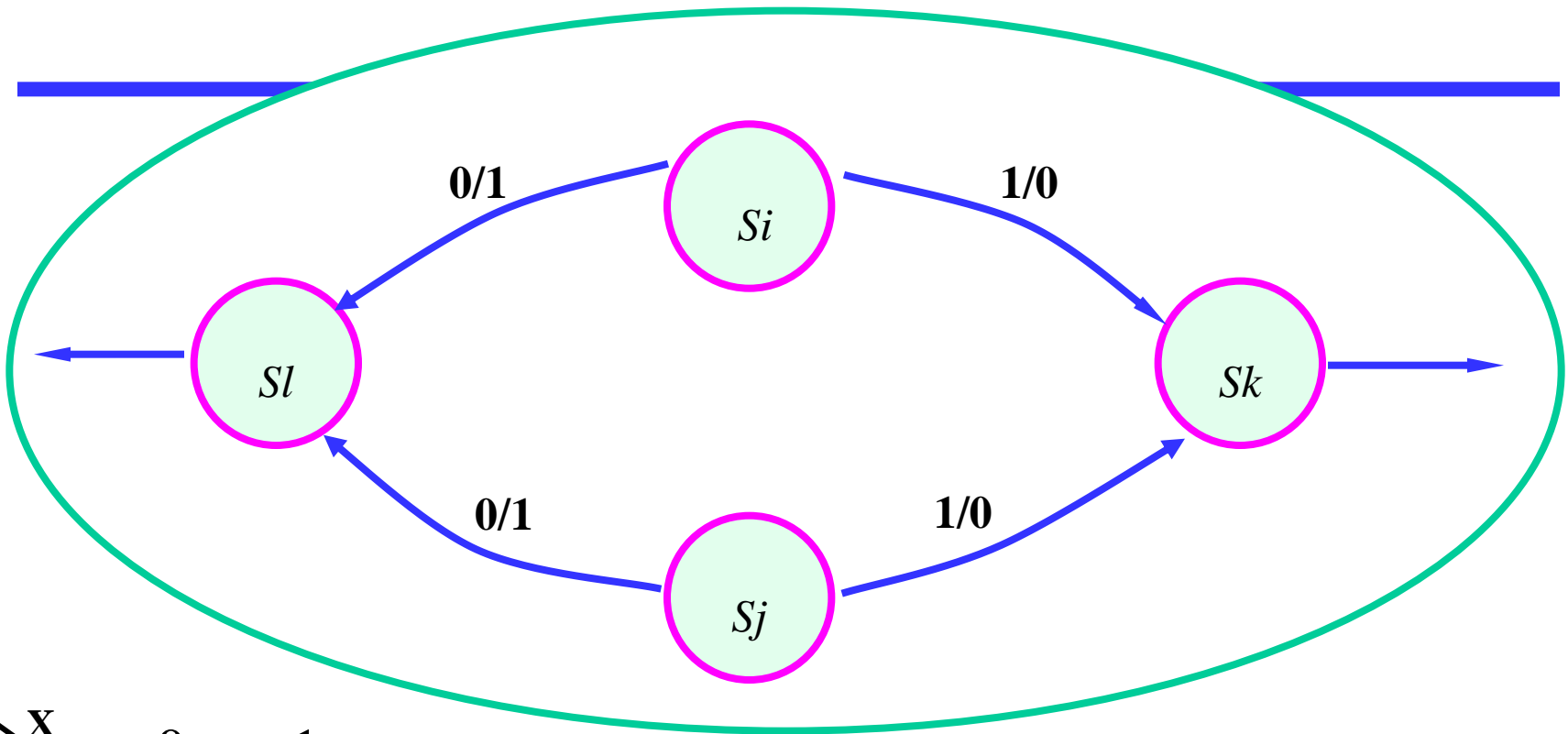
---

- Three cases of equivalent states

For the all inputs,

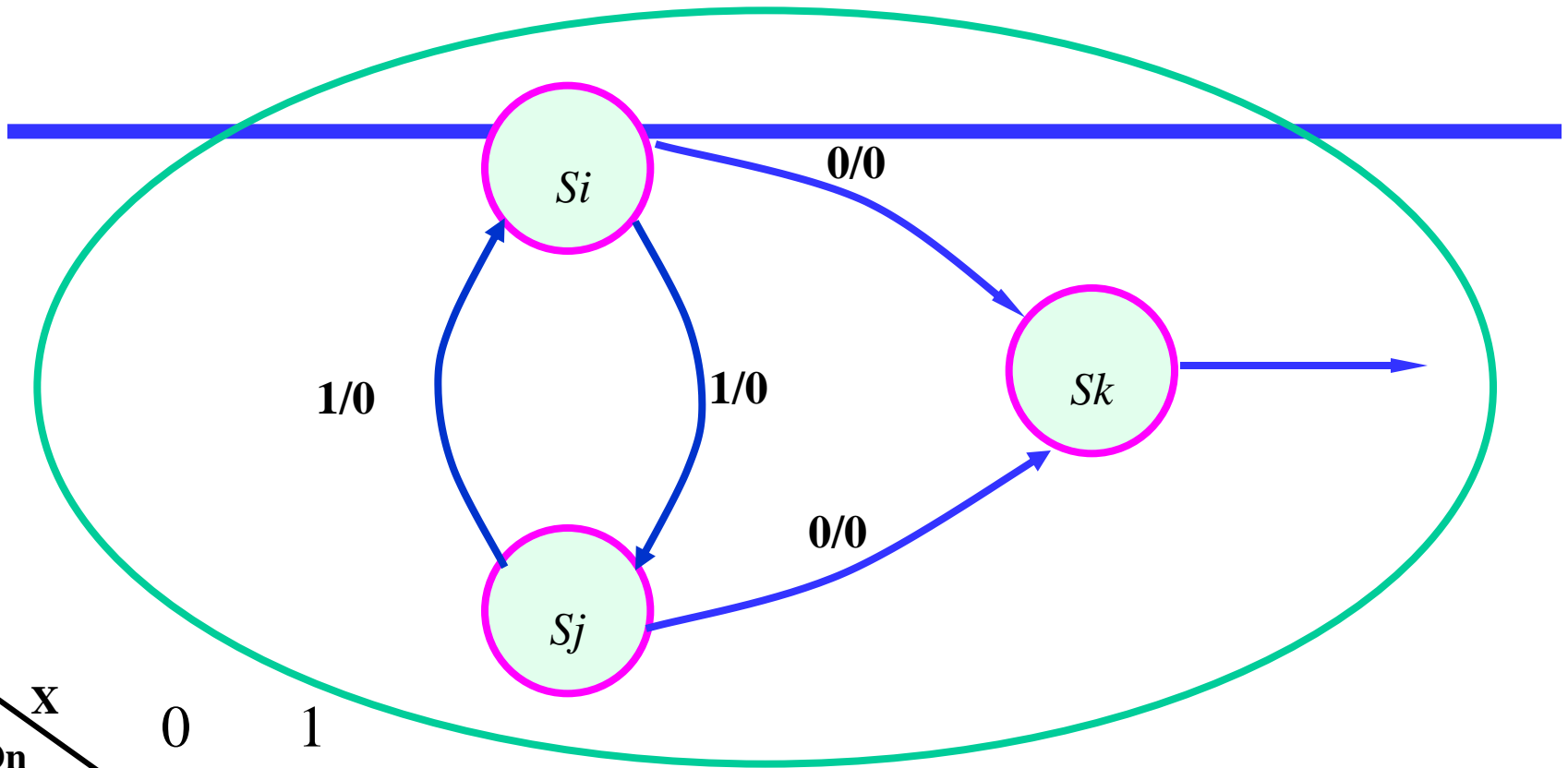
Firstly, **Outputs are the same**

And { **Next States are the same**  
**Or Next States are interleaved**  
**Or Next States are circulated**



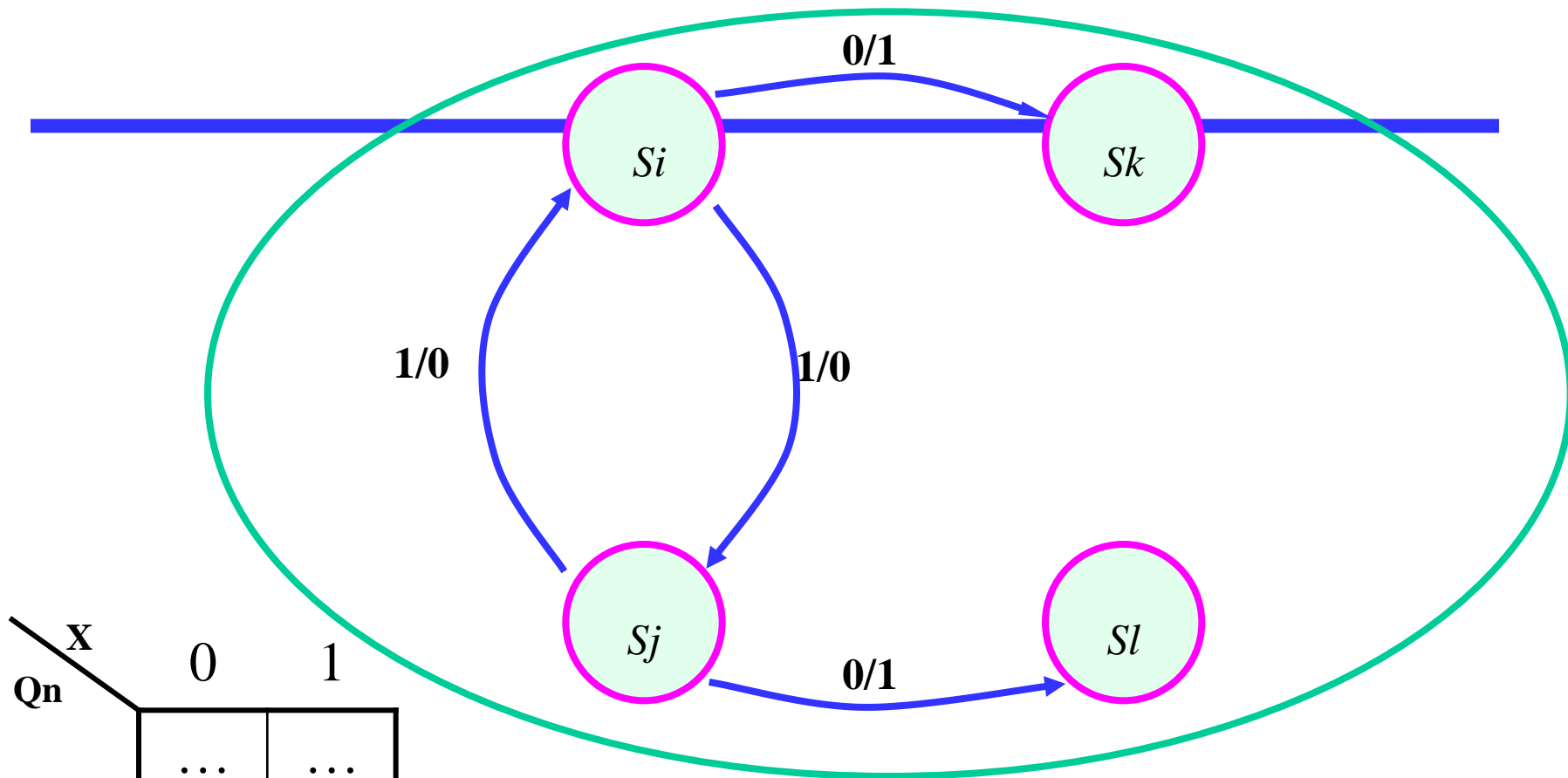
		$X$	
		0	1
$Q_n$	$S_i$	$S_l/1$	$S_k/0$
	$S_j$	$S_l/1$	$S_k/0$
		...	...

**Next States are the same**



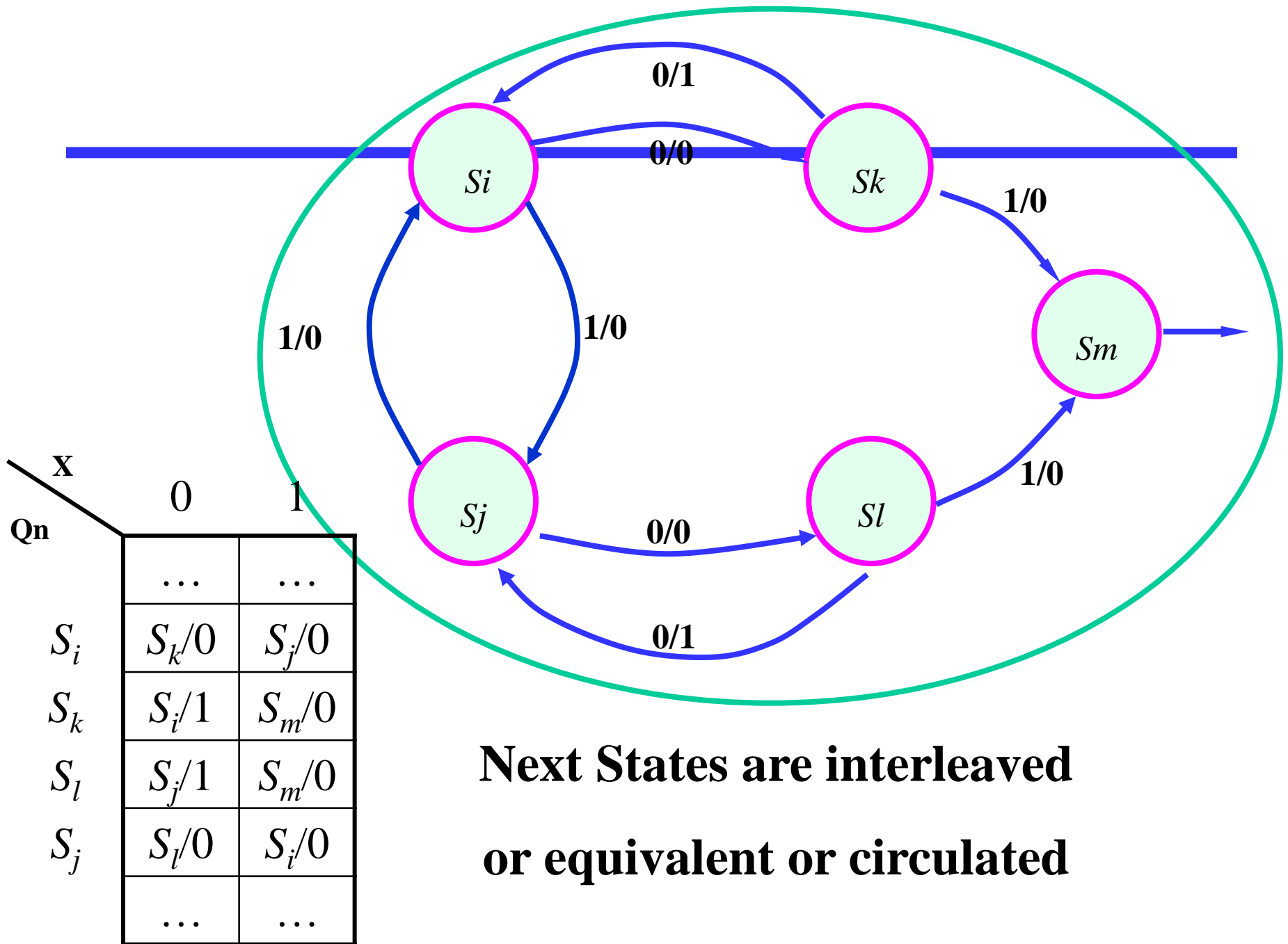
		<b>X</b>	
		0	1
<b>Qn</b>	$S_i$	...	...
	$S_j$	$S_k/0$	$S_i/0$
		...	...

**Next States are the same  
or interleaved**



Qn \ X	0	1
	...	...
$S_i$	$S_k/1$	$S_j/0$
$S_j$	$S_l/1$	$S_i/0$
	...	...

**Next States are interleaved  
or equivalent ( $S_k$ ,  $S_l$  are equivalent)**





# State Minimization and Reduction

- **Equivalence theorem:** For every state table there exists a **unique** equivalent table with **minimum** number of states.
- **How to minimize a state table?**
  - Manual ways: Observation, Implication Chart
  - Algorithms: **Hopcroft**, **Moore**, Brzozowski



John E. Hopcroft  
CORNELL  
UNIVERSITY  
1939-  
1986 Turing Prize



Edward F. Moore  
University of  
Wisconsin–Madison  
1925 - 2003

# Reduction by Observation

---

- **Example 1:**

<b>Present State</b>	<b>Next State/ Output</b>	
	<b><math>X=0</math></b>	<b><math>X=1</math></b>
<b>A</b>	<b>A/0</b>	<b>B/0</b>
<b>B</b>	<b>A/0</b>	<b>C/0</b>
<b>C</b>	<b>A/0</b>	<b>D/1</b>
<b>D</b>	<b>A/0</b>	<b>D/1</b>

# Reduction by Observation (Continued)

---

- **Outputs of A and B, C and D are the same**
- **The next states of C and D are the same for all the possible inputs combination, so C and D are equivalent**
- **Because B and C are not equivalent, A and B are not equivalent**
- **Max Equivalent Classes are {A}, {B}, {C,D}, labeled as A', B', C'**

# Reduction by Observation (Continued)

---

- Minimized State Table is

Present State	Next State/ Output	
	$X=0$	$X=1$
$A'$	$A'/0$	$B'/0$
$B'$	$A'/0$	$C'/0$
$C'$	$A'/0$	$C'/1$

- Reduction by **Implication Chart** (Appendix B)

# State Assignment Method

- Each of the  $m$  states must be assigned a unique code
- Minimum number of bits required is  $n$  such that
$$n \geq \lceil \log_2 m \rceil$$
where  $\lceil x \rceil$  is the smallest integer  $\geq x$
- Numbers of methods can be used to assign codes:

State	Counting Order Assignment	Gray Code Assignment	One-hot Assignment
A	00	00	0001
B	01	01	0010
C	10	11	0100
D	11	10	1000

# State Assignment – Example 1

---

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

- How many assignments of codes with a minimum number of bits?
  - Two –  $A = 0, B = 1$  or  $A = 1, B = 0$
- Does it make a difference?
  - Only in variable inversion, so small, if any.

# State Assignment – Example 2

---

Present State	Next State		Output	
	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

- How many assignments of codes with a minimum number of bits?
  - $4 \times 3 \times 2 \times 1 = 24$
- Does code assignment make a difference in cost?

# State Assignment – Example 2 (continued)

- **Counting Order Assignment:** A = 0 0, B = 0 1, C = 1 0, D = 1 1
- The resulting coded state table:

Present State	Next State		Output	
	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

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



# State Assignment – Example 2 (continued)

---

- **Gray Code Assignment:** A = 0 0, B = 0 1, C = 1 1, D = 1 0
- The resulting coded state table:

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

# Find Flip-Flop Input and Output Equations: Example 2 – Counting Order Assignment

- Assume D flip-flops
- Interchange the bottom two rows of the state table, to obtain K-maps for  $D_1$ ,  $D_2$ , and  $Z$ :

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

$D_1$

		X	
	0	0	
	0	1	
Y <sub>1</sub>	0	0	Y <sub>2</sub>
	1	1	

$D_2$

		X	
	0	1	
	0	0	
Y <sub>1</sub>	0	1	Y <sub>2</sub>
	1	0	

$Z$

		X	
	0	0	
	0	0	
Y <sub>1</sub>	0	1	Y <sub>2</sub>
	0	0	

# Optimization: Example 2: Counting Order Assignment

- Performing two-level optimization:

		D <sub>1</sub>		X
		Y <sub>1</sub>	Y <sub>2</sub>	
		0	0	
		0	1	
		0	0	
		1	1	

		D <sub>2</sub>		X
		Y <sub>1</sub>	Y <sub>2</sub>	
		0	1	
		0	0	
		0	1	
		1	0	

		Z		X
		Y <sub>1</sub>	Y <sub>2</sub>	
		0	0	
		0	0	
		0	1	
		0	0	

$$D_1 = Y_1 \bar{Y}_2 + X \bar{Y}_1 Y_2$$

$$D_2 = \bar{X} Y_1 \bar{Y}_2 + X \bar{Y}_1 \bar{Y}_2 + X Y_1 Y_2$$

$$Z = X Y_1 Y_2$$

Gate Input Cost = 22

# Find Flip-Flop Input and Output Equations: Example 2 – Gray Code Assignment

- Assume D flip-flops
- Obtain K-maps for  $D_1$ ,  $D_2$ , and  $Z$ :

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

$D_1$

		X	
	0	0	
	0	1	
Y <sub>1</sub>	1	1	Y <sub>2</sub>
	0	0	

$D_2$

		X	
	0	1	
	0	1	
Y <sub>1</sub>	0	1	Y <sub>2</sub>
	0	1	

$Z$

		X	
	0	0	
	0	0	
Y <sub>1</sub>	0	0	Y <sub>2</sub>
	0	1	

# Optimization: Example 2: Assignment 2

- Performing two-level optimization:

$D_1$		X	
	0	0	
	0	1	
	1	1	$Y_2$
$Y_1$	0	0	

$$D_1 = Y_1 Y_2 + X Y_2$$

$$D_2 = X$$

$$Z = X Y_1 \bar{Y}_2$$

$D_2$		X	
	0	1	
	0	1	
	0	1	$Y_2$
$Y_1$	0	1	
	0	1	

$Z$		X	
	0	0	
	0	0	
	0	0	$Y_2$
$Y_1$	0	1	

**Gate Input Cost = 9**

# State Assignment

---

- Generally different assignments will result in different output functions, input functions, and the complexity of circuit varies as well. The tasks of state assignment are:
  - Determine the length of code
  - Find the best state assignment or close to the best
- Finding a best assignment can be extremely hard when  $N$  is large enough. And the performance of the state assignment also depends on the type of triggers.
- In practice, state assignment is processed in an engineering way, where heuristic rules are applied (Appendix C).

# Map Technology

- **Library:**

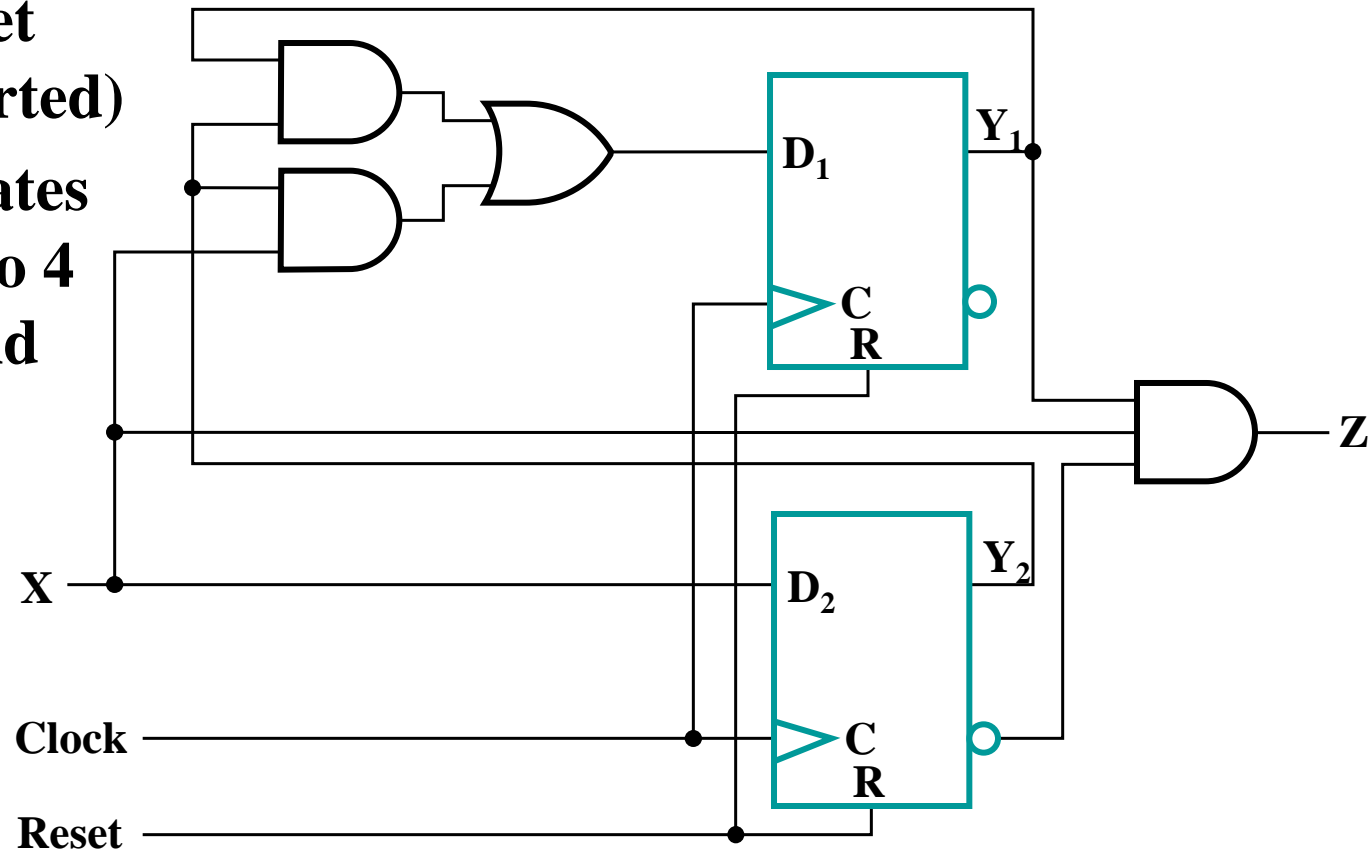
- **D Flip-flops with Reset (not inverted)**
- **NAND gates with up to 4 inputs and inverters**

- **Initial Circuit:**  
(in slide 52)

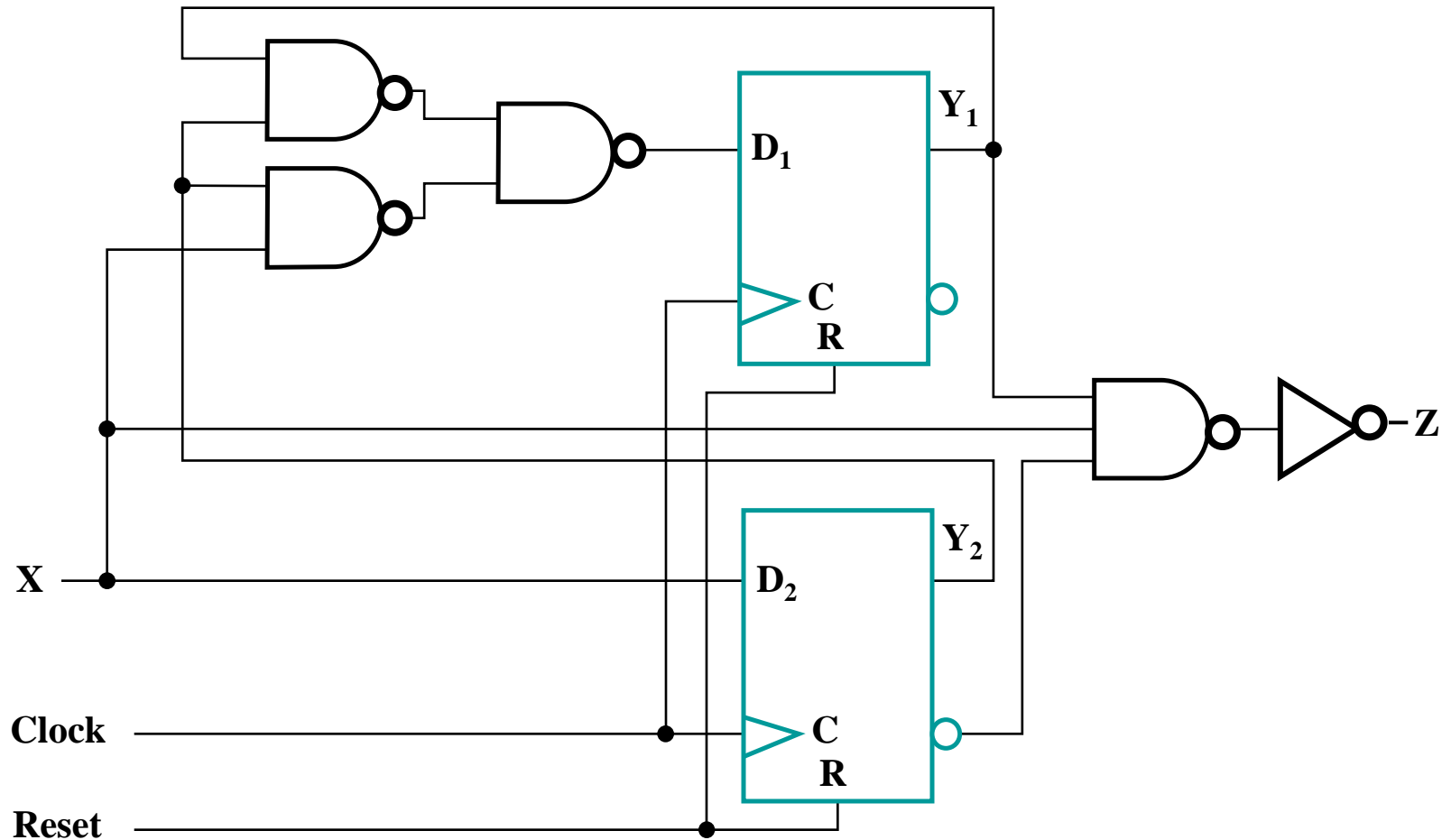
$$\mathbf{D}_1 = \mathbf{Y}_1 \mathbf{Y}_2 + \mathbf{X} \mathbf{Y}_2$$

$$\mathbf{D}_2 = \mathbf{X}$$

$$\mathbf{Z} = \mathbf{XY}_1\bar{\mathbf{Y}}_2$$



# Mapped Circuit - Final Result





# Sequential Design: Example 3

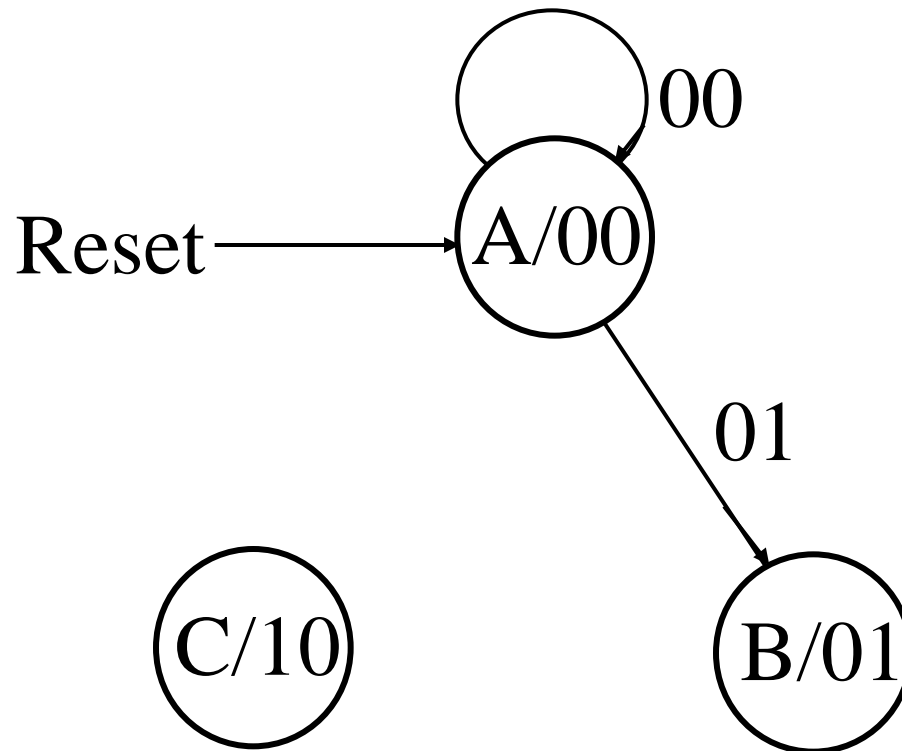
---

- **Design a sequential modulo 3 accumulator for 2-bit operands**
- **Definitions:**
  - **Modulo  $n$  adder** - an adder that gives the result of the addition as the remainder of the sum divided by  $n$ 
    - **Example:**  $2 + 2 \text{ modulo } 3 = \text{remainder of } 4/3 = 1$
  - **Accumulator** - a circuit that “accumulates” the sum of its input operands over time - it adds each input operand to the stored sum, which is initially 0.
- **Input:  $(X_1, X_0)$ , Stored sum:  $(Y_1, Y_0)$ , Output:  $(Z_1, Z_0)$**

# Example 3 (continued)

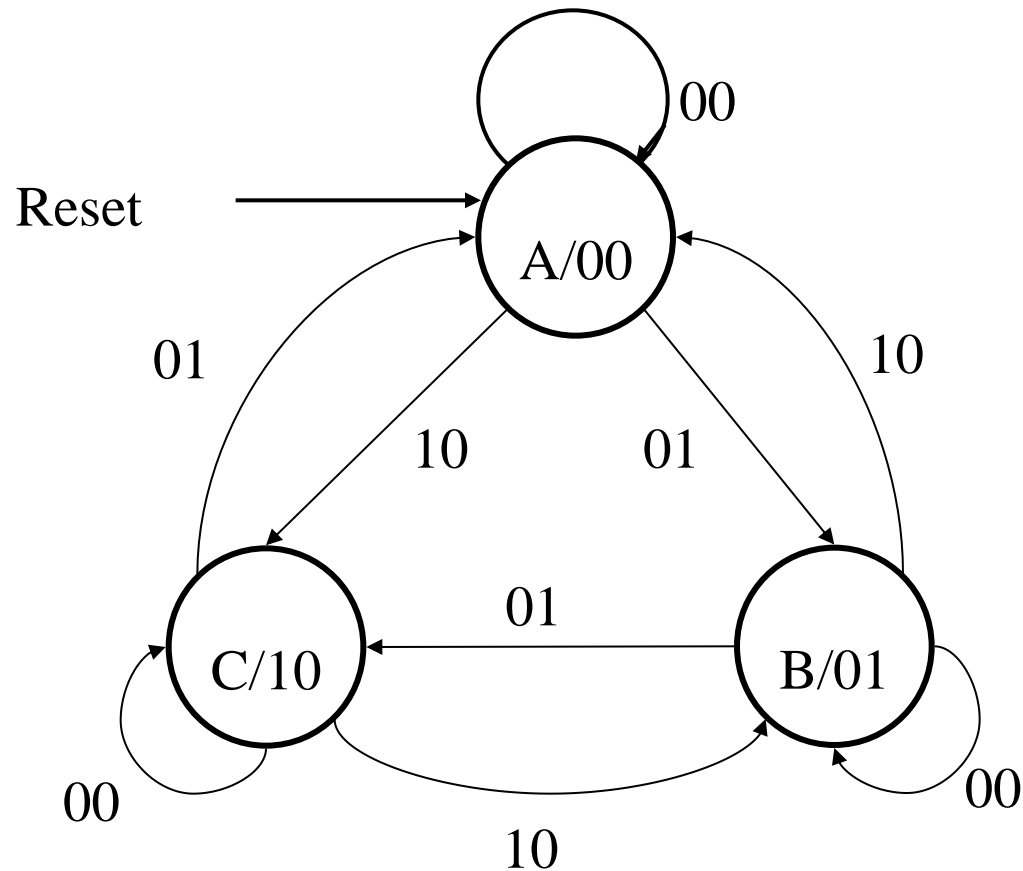
---

- Complete the state diagram:



# Example 3 (continued)

- Complete the state diagram:



# Example 3 (continued)

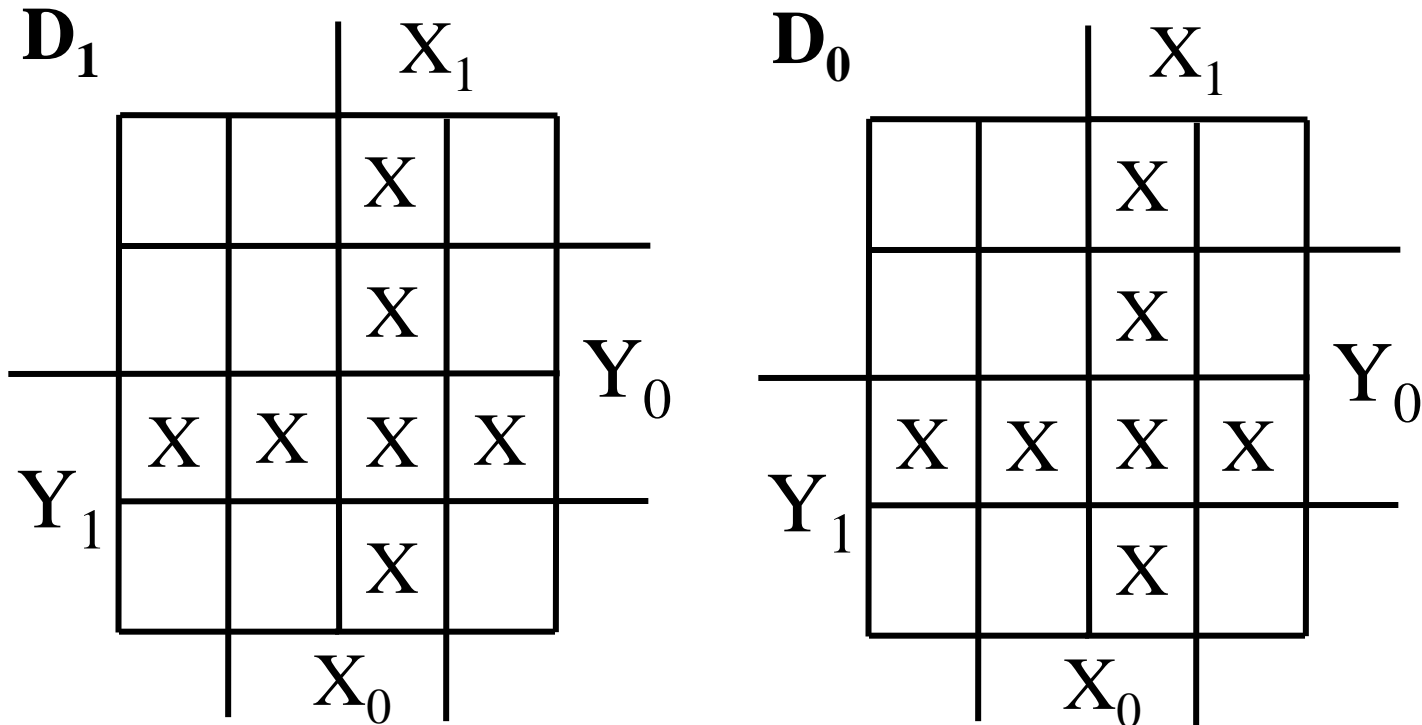
- Complete the state table

$X_1X_0$ $Y_1Y_0$	00	01	11	10	$Z_1Z_0$
	$Y_1(t+1),$ $Y_0(t+1)$	$Y_1(t+1),$ $Y_0(t+1)$	$Y_1(t+1),$ $Y_0(t+1)$	$Y_1(t+1),$ $Y_0(t+1)$	
A (00)	00	01	X	10	00
B (01)	01	10	X	00	01
- (11)	X	X	X	X	11
C (10)	10	00	X	01	10

- State Assignment:  $(Y_1, Y_0) = (Z_1, Z_0)$
- Codes are in gray code order to ease use of K-maps in the next step

# Example 3 (continued)

- Find optimized flip-flop input equations for D flip-flops



- $D_1 =$
- $D_0 =$

## Example 3 (continued)

- Find optimized flip-flop input equations for D flip-flops

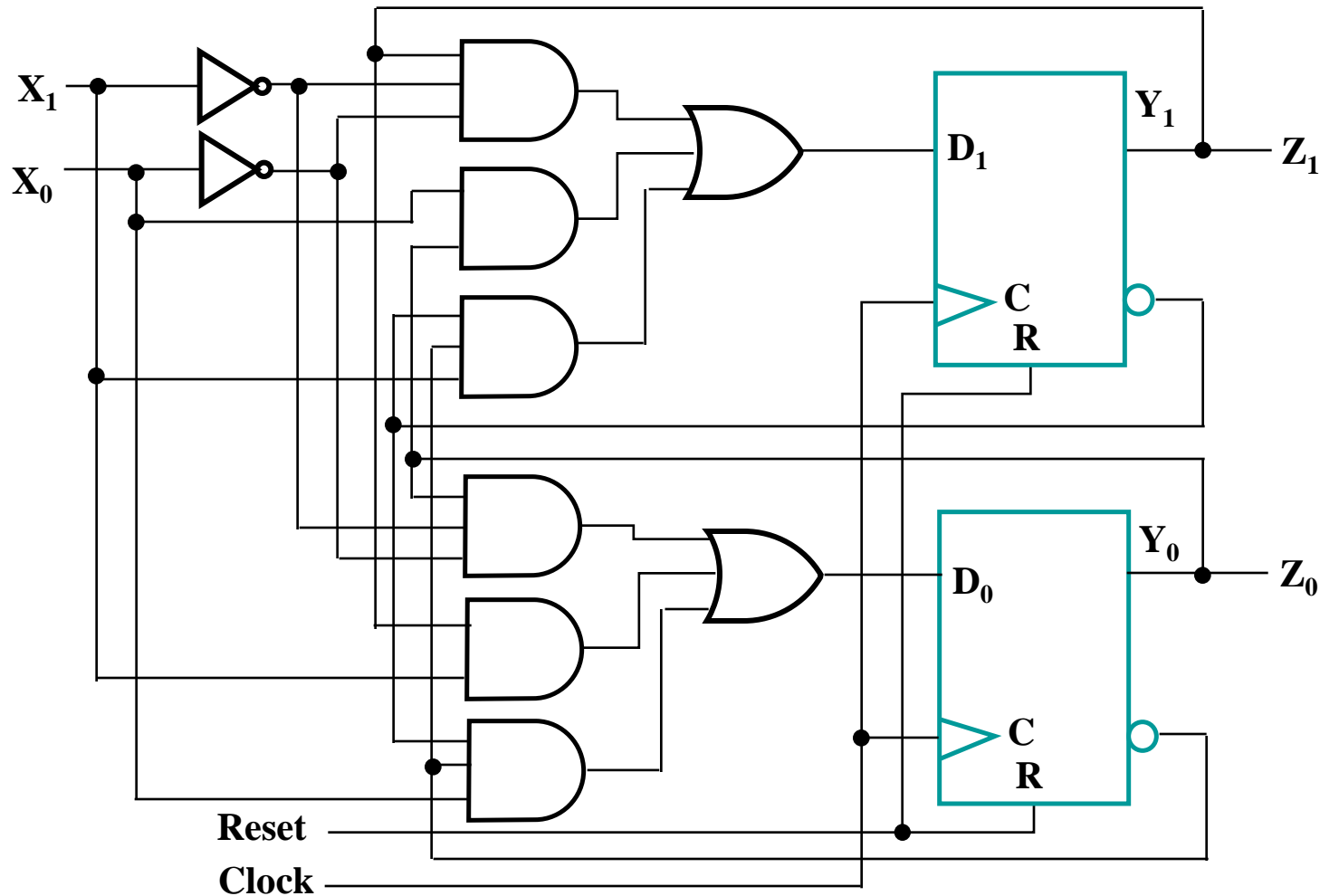
$D_1$			$X_1$	
	0	0	X	1
	0	1	X	0
$Y_1$	X	X	X	X
	1	0	X	0
			$X_0$	

$D_0$			$X_1$	
	0	1	X	0
	1	0	X	0
$Y_1$	X	X	X	X
	0	0	X	1
			$X_0$	

- $\mathbf{D}_1 = \mathbf{Y}_1 \overline{\mathbf{X}}_1 \overline{\mathbf{X}}_0 + \mathbf{Y}_0 \mathbf{X}_0 + \overline{\mathbf{Y}}_1 \overline{\mathbf{Y}}_0 \mathbf{X}_1$
- $\mathbf{D}_0 = \mathbf{Y}_0 \overline{\mathbf{X}}_1 \overline{\mathbf{X}}_0 + \mathbf{Y}_1 \mathbf{X}_1 + \overline{\mathbf{Y}}_1 \overline{\mathbf{Y}}_0 \mathbf{X}_0$

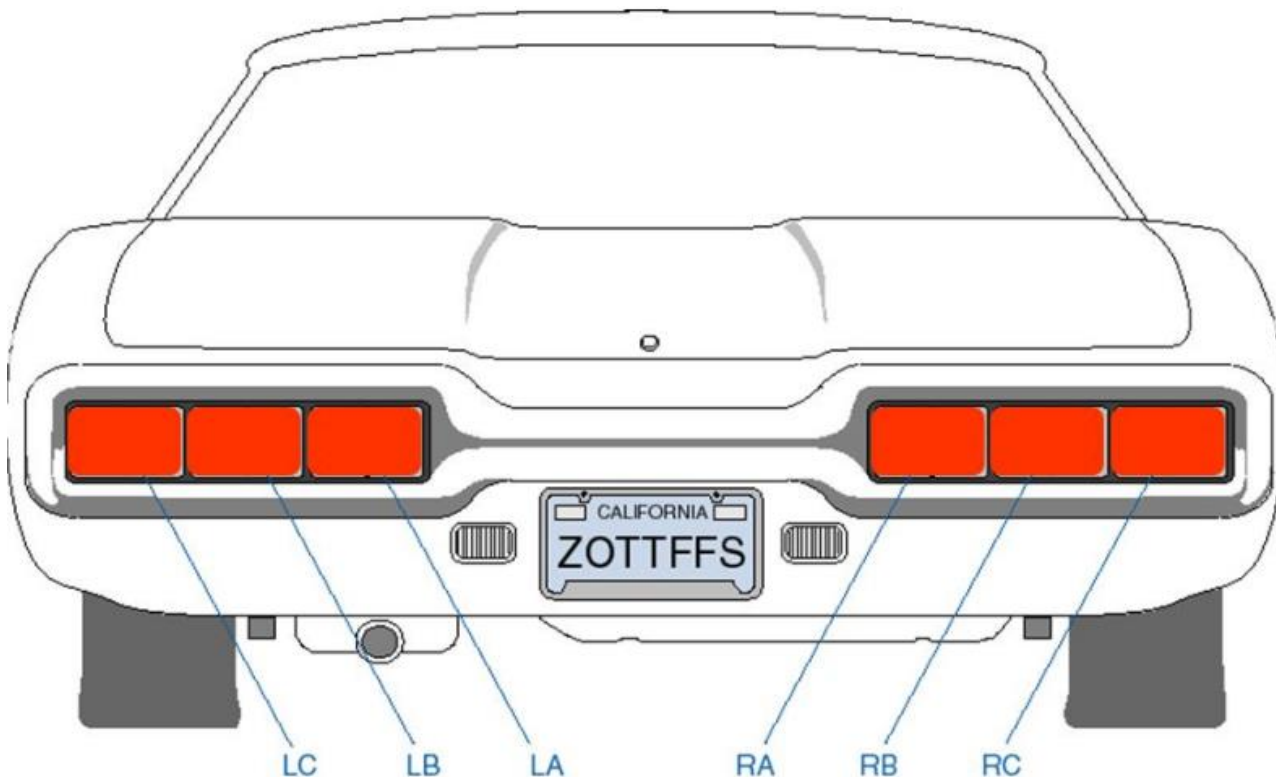
# Circuit - Final Result with AND, OR, NOT

---



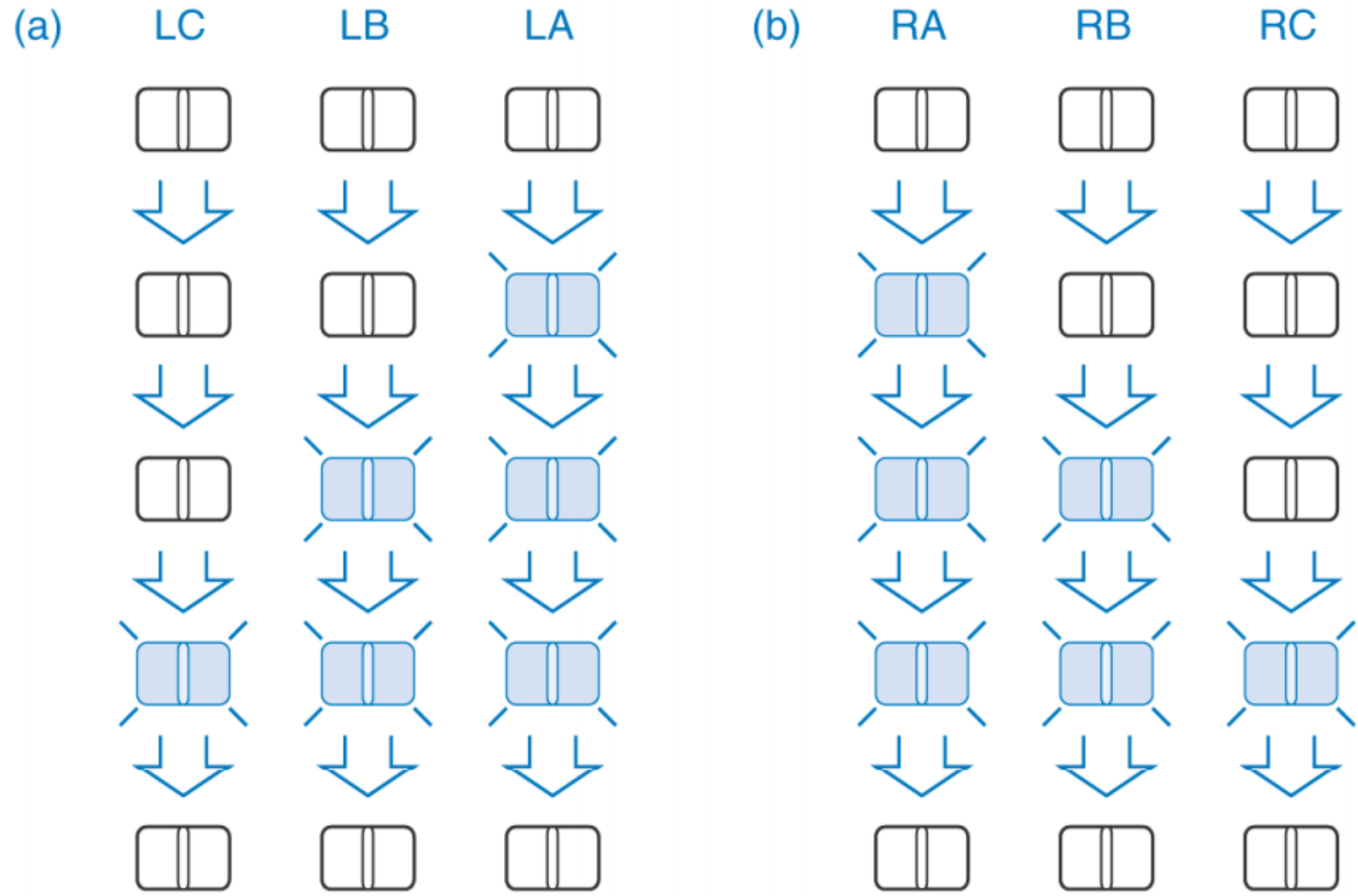
# Sequential Design: Example 4

- Design a **state-machine diagram** to control the tail lights of a 1965 Ford Thunderbird. The tail lights are composed of three light on each side.





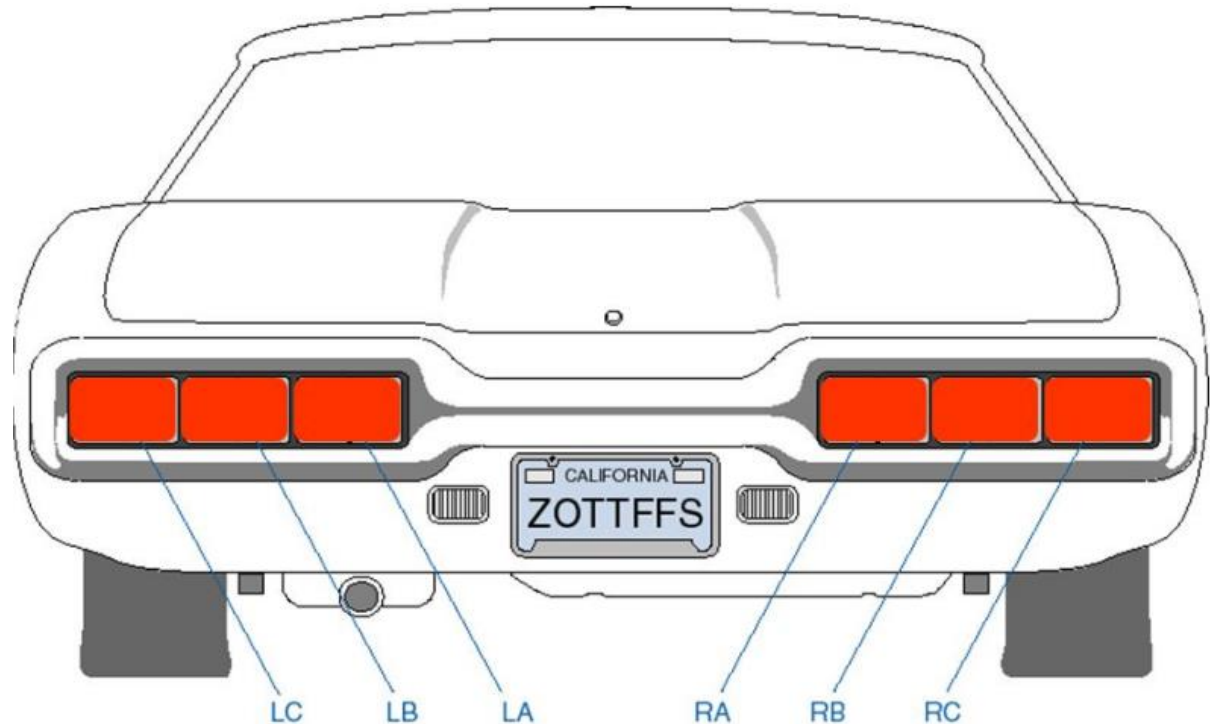
# Flashing sequence for T-bird tail-lights



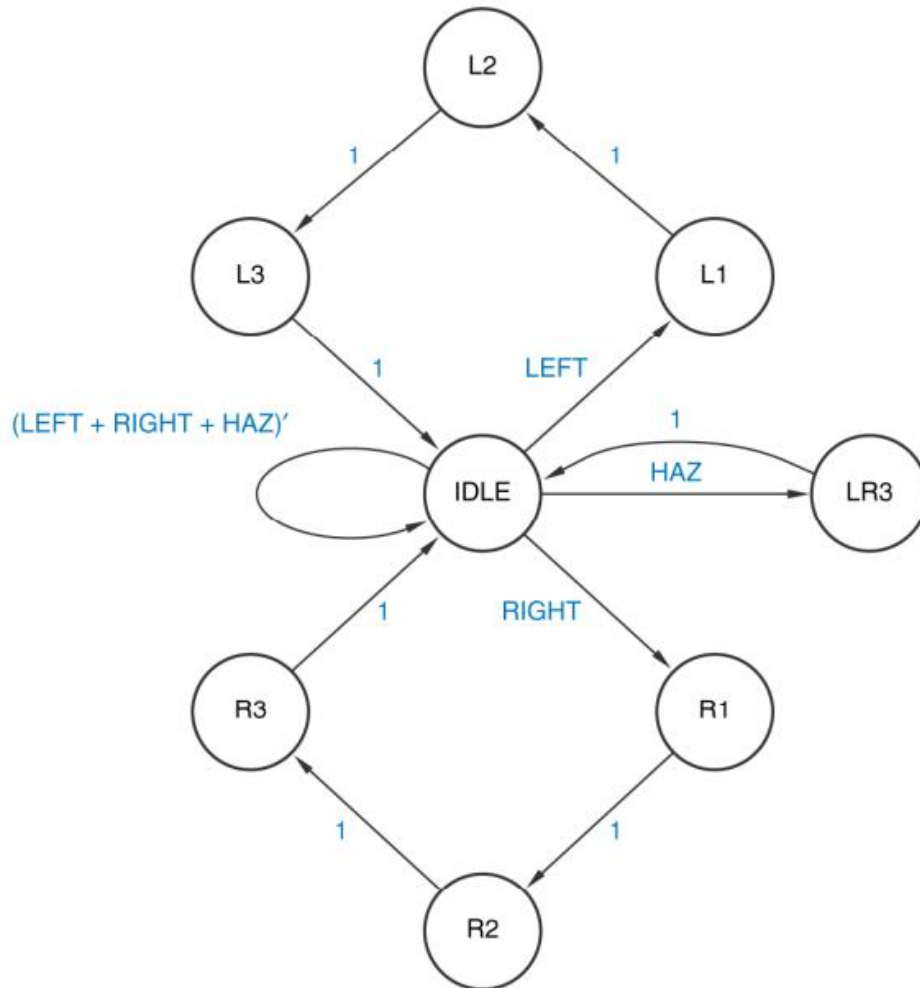
# Input and Output of T-bird Tail-lights

---

- The state machine has:
  - **Three inputs:** left, and right turns, and hazard.
  - **Six outputs:** LA, LB, LC, RA, RB, and RC.
  - Free running **clock** with frequency equal to the flashing rate.



# State-Machine Diagram for T-bird Tail-lights

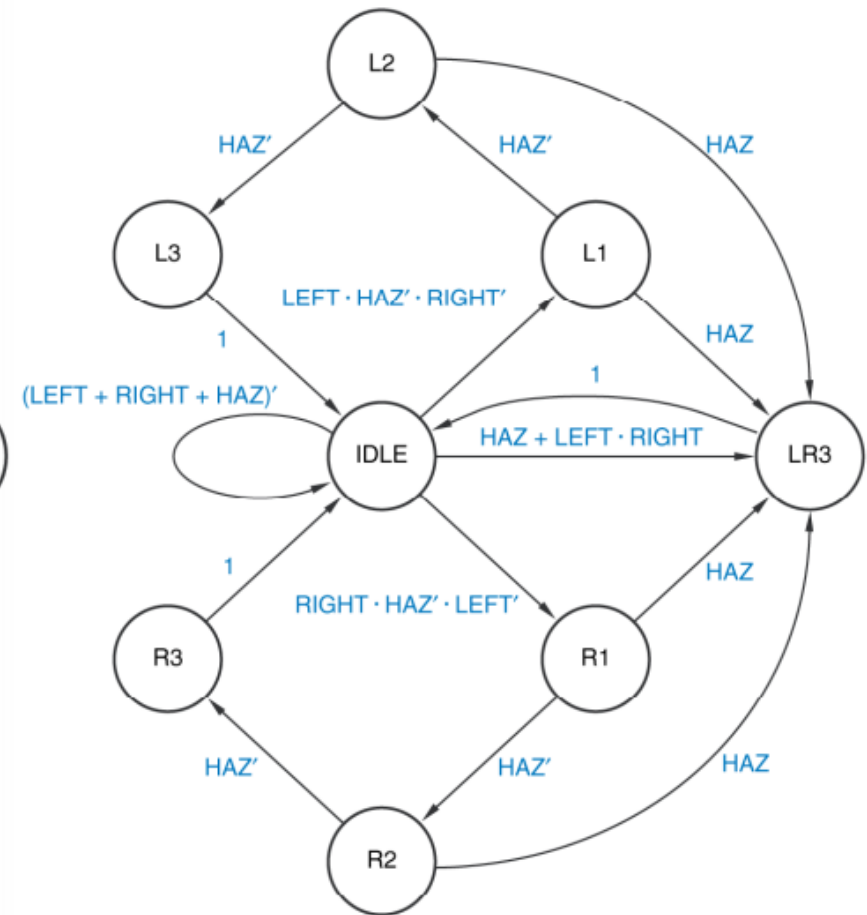
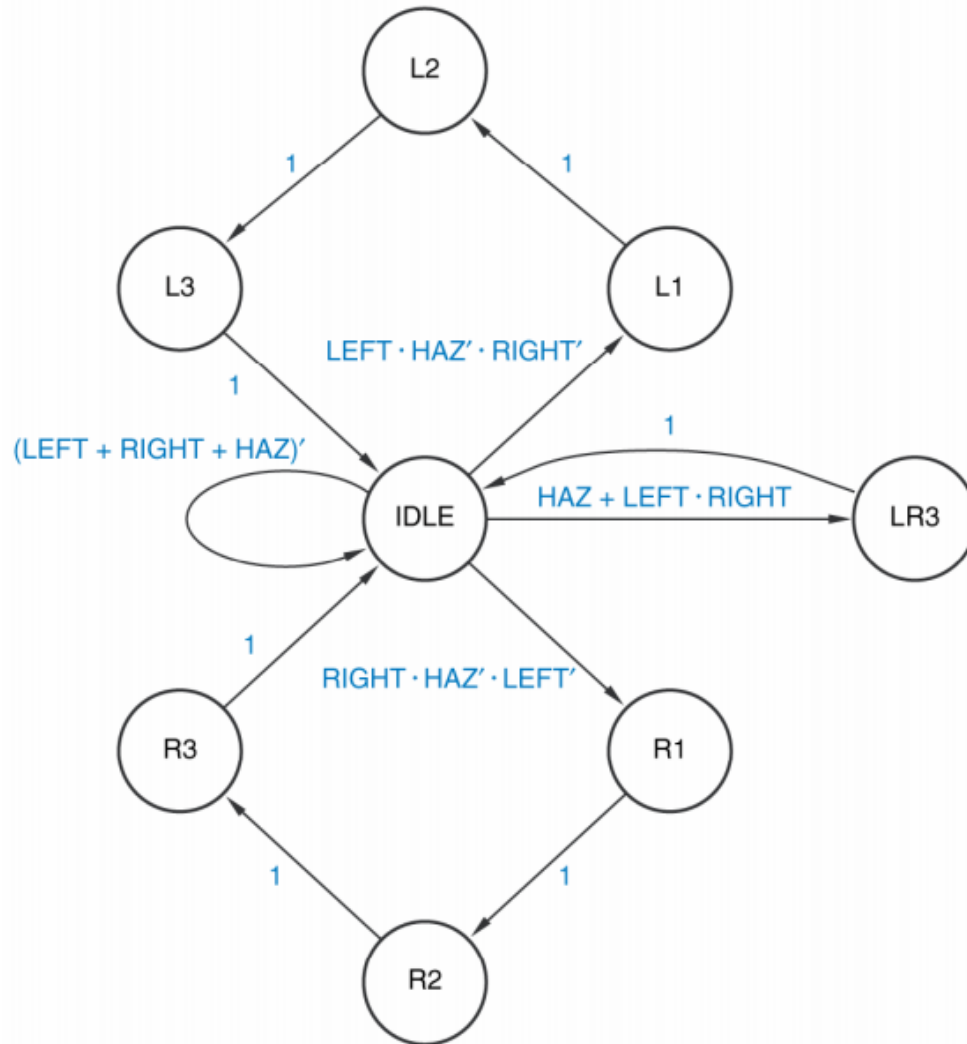


Output Table

State	LC	LB	LA	RA	RB	RC
IDLE	0	0	0	0	0	0
L1	0	0	1	0	0	0
L2	0	1	1	0	0	0
L3	1	1	1	0	0	0
R1	0	0	0	1	0	0
R2	0	0	0	1	1	0
R3	0	0	0	1	1	1
LR3	1	1	1	1	1	1

- $LC = L3 + LR3$
- $LB = L2 + L3 + LR3$
- $LA = L1 + L2 + L3 + LR3$
- $RA = R1 + R2 + R3 + LR3$
- $RB = R2 + R3 + LR3$
- $RC = R3 + LR3$

# Enhanced State-Machine Diagram for T-bird Tail-lights



# State Assignment for T-bird Tail-lights

- IDLE state of 000.
- Q1 and Q0 are used to count in gray code sequence:  
(IDLE→L1→L2→L3→IDLE)  
(IDLE→R1→R2→R3→IDLE)
- Q2 identifies LEFT or RIGHT turn.
- HAZ state of 100.

<i>State</i>	<i>Q2</i>	<i>Q1</i>	<i>Q0</i>
IDLE	0	0	0
L1	0	0	1
L2	0	1	1
L3	0	1	0
R1	1	0	1
R2	1	1	1
R3	1	1	0
LR3	1	0	0

# State Table for T-bird Tail-lights

<i>S</i>	<i>Q2</i>	<i>Q1</i>	<i>Q0</i>	<i>Transition Expression</i>	<i>S</i>	<i>Q2</i>	<i>Q1</i>	<i>Q0</i>
IDLE	0	0	0	$(\text{LEFT} + \text{RIGHT} + \text{HAZ})'$	IDLE	0	0	0
IDLE	0	0	0	$\text{LEFT} \cdot \text{HAZ}' \cdot \text{RIGHT}'$	L1	0	0	1
IDLE	0	0	0	$\text{HAZ} + \text{LEFT} \cdot \text{RIGHT}$	LR3	1	0	0
IDLE	0	0	0	$\text{RIGHT} \cdot \text{HAZ}' \cdot \text{LEFT}'$	R1	1	0	1
L1	0	0	1	$\text{HAZ}'$	L2	0	1	1
L1	0	0	1	$\text{HAZ}$	LR3	1	0	0
L2	0	1	1	$\text{HAZ}'$	L3	0	1	0
L2	0	1	1	$\text{HAZ}$	LR3	1	0	0
L3	0	1	0	1	IDLE	0	0	0
R1	1	0	1	$\text{HAZ}'$	R2	1	1	1
R1	1	0	1	$\text{HAZ}$	LR3	1	0	0
R2	1	1	1	$\text{HAZ}'$	R3	1	1	0
R2	1	1	1	$\text{HAZ}$	LR3	1	0	0
R3	1	1	0	1	IDLE	0	0	0
LR3	1	0	0	1	IDLE	0	0	0

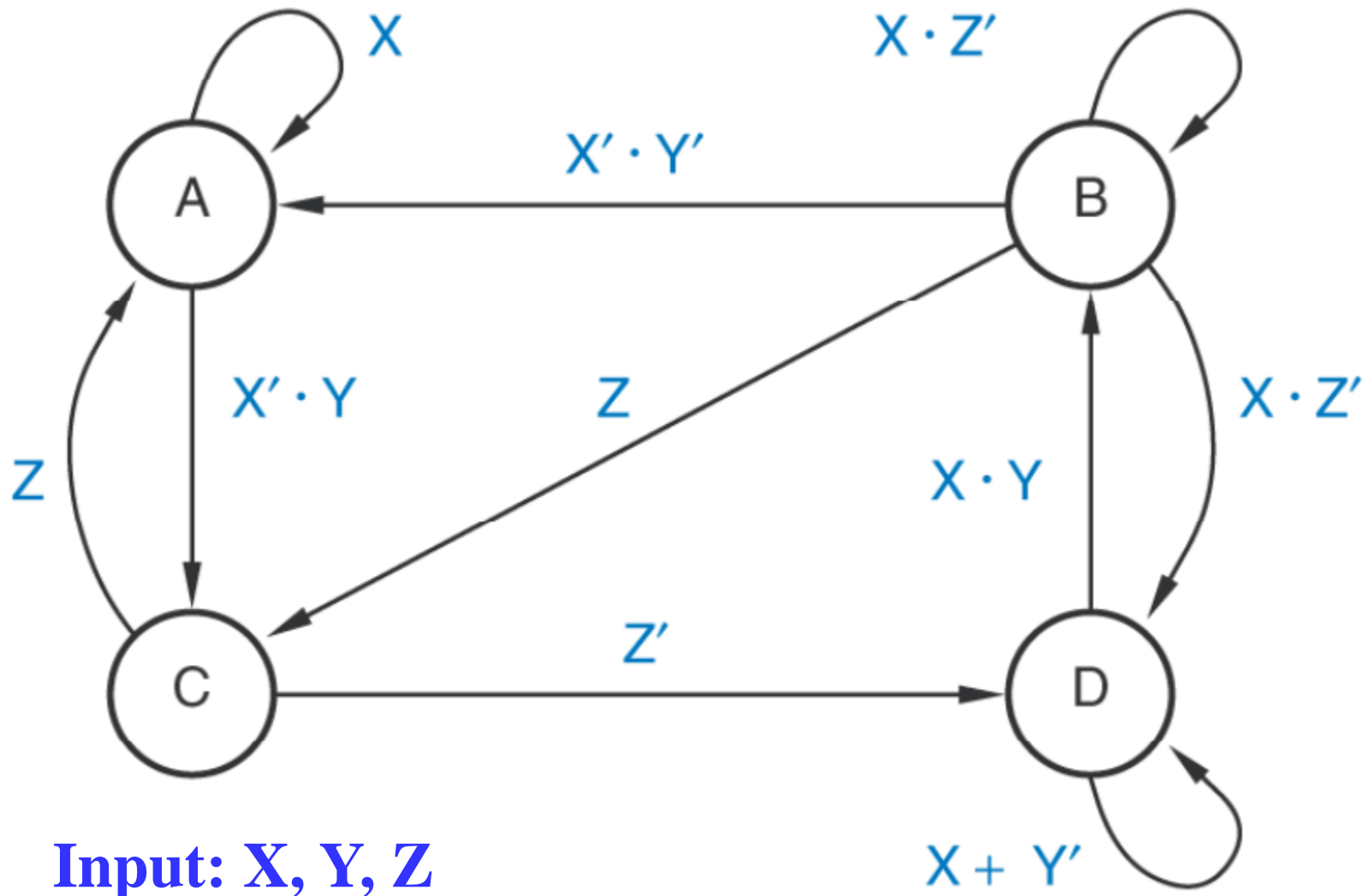
# Summary of Designing State-Machine Diagram

---

- **State-machine diagram design is simple but it is more prone to errors:**
  - **State table is an exhaustive listing of the next states for each state/input combination.**
  - **Each input combination should be covered exactly once by an expression of an outgoing arc.**
  - **Ambiguity in state diagram: **double-covered** or **uncovered**.**

# Example of an Ambiguous State-Machine Diagram

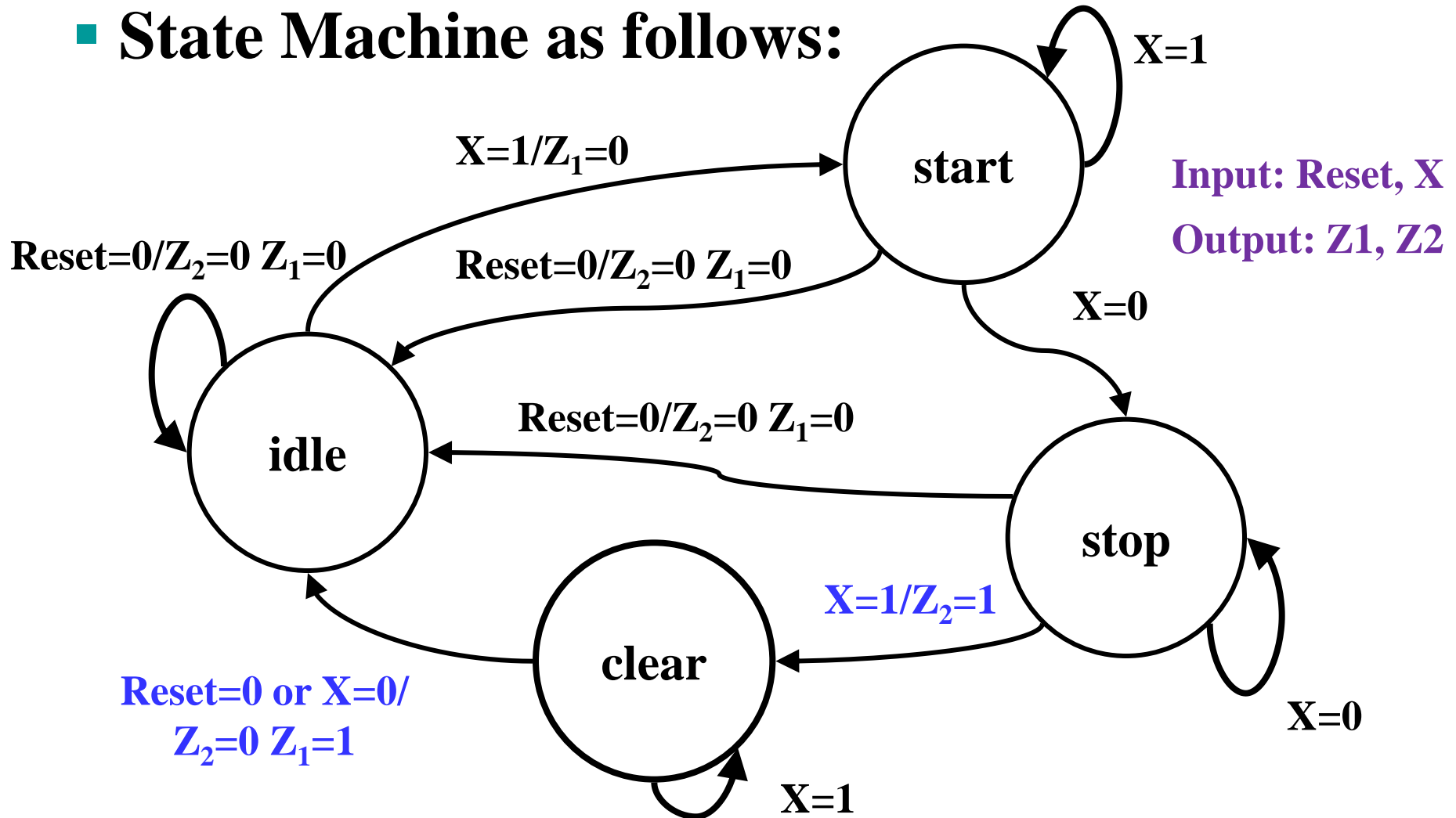
---





# How to Code a State Machine in Verilog

- State Machine as follows:



# How to Code a State Machine in Verilog

## ---- Use Gray Code as State Code

```
module fsm (Clock, Reset, X, Z2, Z1);  
  input Clock, Reset, X;  
  output Z2, Z1;  
  reg Z2, Z1;  
  reg [1:0] state ;  
  parameter Idle = 2'b00, Start = 2'b01,  
            Stop = 2'b10, Clear = 2'b11;
```

```
always @(posedge Clock)  
  if (!Reset)  
    begin  
      state <= Idle;  
      Z2<=0;  
      Z1<=0;  
    end  
  else  
    case (state)  
      Idle: if (X) begin  
              state <= Start;  
              Z1<=0;  
            end  
    end
```

```
    else begin  
      state <= Idle;  
      Z2<=0;  
      Z1<=0;  
    end  
  Start: if (!X) state <= Stop;  
        else state <= Start;  
  Stop: if (X) begin  
          state <= Clear;  
          Z2<= 1;  
        end  
        else state <= Stop;  
  Clear:  
    if (!X) begin  
      state <=Idle;  
      Z2<=0;  
      Z1<=1;  
    end  
    else state <= Clear;  
  endcase  
endmodule
```

# How to Code a State Machine in Verilog

## ---- Use One-Hot Code as State Code

```
module fsm (Clock, Reset, X, Z2, Z1);
  input Clock, Reset, X;
  output Z2, Z1;
  reg Z2, Z1;
  reg [3:0] state ;
  parameter Idle = 4'b1000, Start = 4'b0100,
             Stop = 4'b0010, Clear =4'b0001;

  always @(posedge Clock)
    if (!Reset)
      begin
        state <= Idle; Z2<=0; Z1<=0;
      end
    else
      case (state)
        Idle: if (X) begin
              state <= Start;
              Z1<=0;
            end
          else begin
            state <= Idle;
          end
      endcase
    end
```

```
        Z2<=0;
        Z1<=0;
      end
    Start: if (!X) state <= Stop;
           else state <= Start;
    Stop:  if (X) begin
           state <= Clear;
           Z2<= 1;
         end
           else state <= Stop;
    Clear: if (!X) begin
           state <=Idle;
           Z2<=0;
           Z1<=1;
         end
           else state <= Clear;
    default: state <= Idle;
  endcase
endmodule
```

# How to Code a State Machine in Verilog

## ----- Use Output Value as State Code

```
module fsm (Clock, Reset, X, Z2, Z1);
input Clock, Reset, X;
output Z2, Z1;
reg [3:0] state ;
    assign Z2= state[3]; //状态变量最高位输出Z2
    assign Z1= state[0]; //状态变量最低位输出Z1
parameter // Z2_S2_S1_Z1
    zero    = 4'b0000,
    Idle    = 4'b0001,
    Start   = 4'b0100,
    Stop    = 4'b0010,
    Clear   = 4'b1000;

always @(posedge Clock)
    if (!Reset)
        begin
            state <= Zero;
        end
    else
        case (state)
            Idle, Zero:
```

```
                if (X) begin
                    state <= Start;
                end
                else state <= Zero;
            Start: if (!X) state <= Stop;
                else state <= Start;
            Stop: if (X) begin
                    state <= Clear;
                end
                else state <= Stop;
            Clear: if (!X) begin
                    state <= Idle;
                end
                else state <= Clear;

            default: state <= Zero;
        endcase
endmodule
```

# How to Code a State Machine in Verilog

## ---- Complicate State Machine with multi-outputs

```
module fsm (Clock, Reset, X, Z2, Z1);  
input Clock, Reset, X;  
output Z2, Z1;  
reg Z2, Z1;  
reg [1:0] state, nextstate;  
parameter  
  Idle = 2'b00, Start = 2'b01,  
  Stop = 2'b10, Clear = 2'b11;  
  
always @(posedge Clock) //复位、状态存储  
  if (!Reset)  
    state <= Idle;  
  else state <= nextstate;  
  
always @(state or X) //产生下一状态  
  case (state)  
    Idle: if (X) nextstate <= Start;  
         else nextstate <= Idle;  
    end  
    Start: if (!X) nextstate <= Stop;  
          else nextstate <= Start;
```

```
    Stop: if (X) nextstate <= Clear;  
         else nextstate <= Stop;  
    Clear: if (!X) nextstate <= Idle;  
          else nextstate <= Clear;  
    default: nextstate <= 2'bxx;  
  endcase  
  
always @(state or Reset or X) //产生输出Z2  
  if (!Reset) Z2=0;  
  else  
    if (state == Stop && X)  
      Z2=1;  
    else Z2=0;  
  
always @(state or Reset or X) //产生输出Z1  
  if (!Reset) Z1=0;  
  else  
    if (state == Clear && !X)  
      Z1=1;  
    else Z1=0;  
  
endmodule
```

# Other Flip-Flop Types

---

- **J-K and T flip-flops**
  - Behavior
  - Implementation
- **Basic descriptors for understanding and using different flip-flop types**
  - Characteristic (truth) tables
  - Characteristic equations
  - Excitation tables
- **For actual use, see Reading Supplement - Design and Analysis Using J-K and T Flip-Flops**

# J-K Flip-flop

---

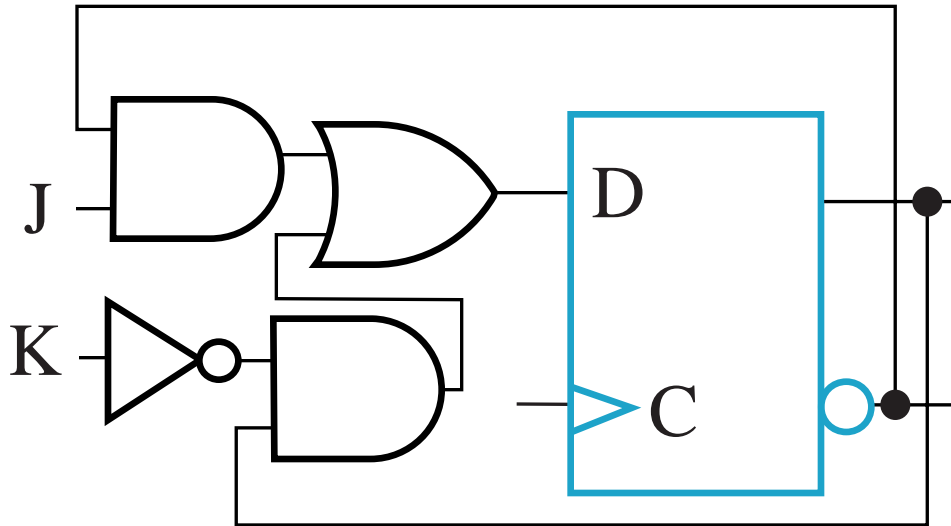
## ■ Behavior

- Same as S-R flip-flop with J analogous to S and K analogous to R
- Except that  $J = K = 1$  is allowed, and
- For  $J = K = 1$ , the flip-flop changes to the *opposite state*
- As a master-slave, has same “1s catching” behavior as S-R flip-flop
- If the master changes to the wrong state, that state will be passed to the slave
  - E.g., if master falsely set by  $J = 1, K = 1$  cannot reset it during the current clock cycle

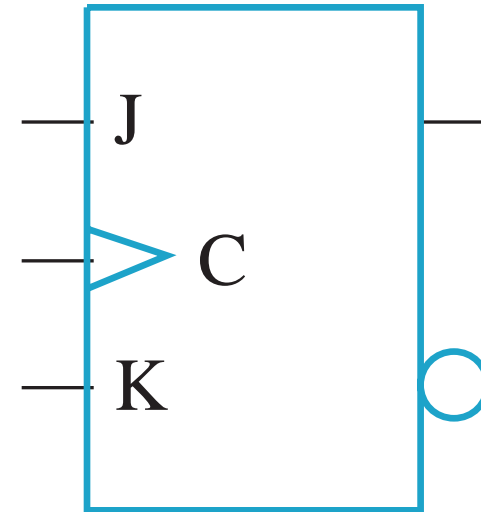
# J-K Flip-flop (continued)

## ■ Implementation

- To avoid 1s catching behavior, one solution used is to use an edge-triggered D as the core of the flip-flop



## ■ Symbol





# T (Toggle) Flip-flop

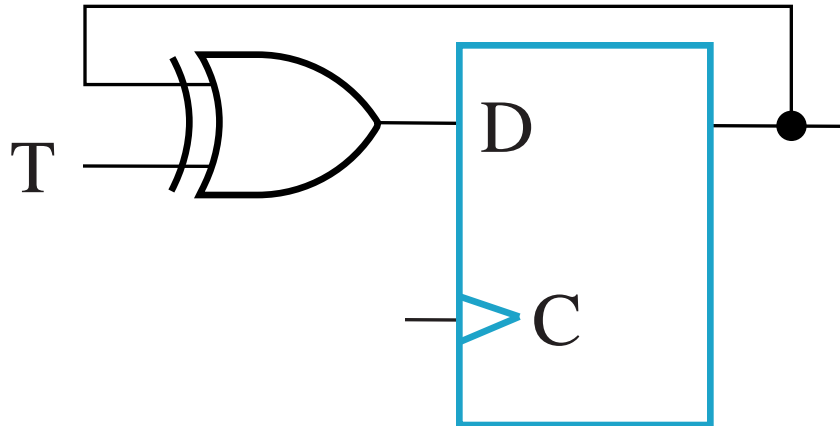
---

- **Behavior**
  - **Has a single input T**
    - For  $T = 0$ , no change to state
    - For  $T = 1$ , changes to opposite state
- **Same as a J-K flip-flop with  $J = K = T$**
- **As a master-slave, has same “1s catching” behavior as J-K flip-flop**
- **Cannot be initialized to a known state using the T input**
  - **Reset (asynchronous or synchronous) essential**

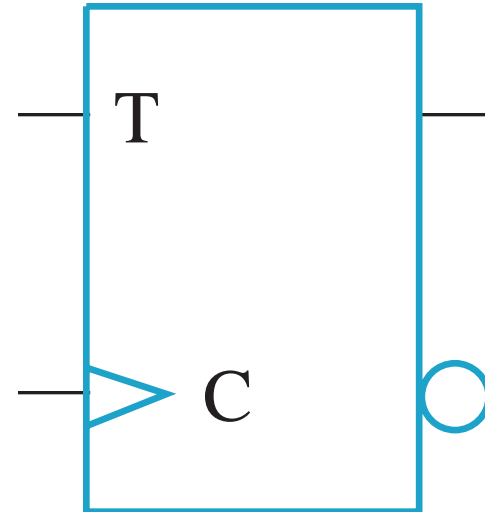
# T (Toggle) Flip-flop (continued)

## ■ Implementation

- To avoid 1s catching behavior, one solution used is to use an edge-triggered D as the core of the flip-flop



## ■ Symbol



# Basic Flip-Flop Descriptors

---

## ■ Used in analysis

- *Characteristic table* - defines the **next state** of the flip-flop in terms of flip-flop **inputs** and **current state**
- *Characteristic equation* - defines the **next state** of the flip-flop as a Boolean function of the flip-flop **inputs** and the **current state**

## ■ Used in design

- *Excitation table* - defines the flip-flop **input** variable values as function of the **current state** and **next state**

# D Flip-Flop Descriptors

- **Characteristic Table**

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

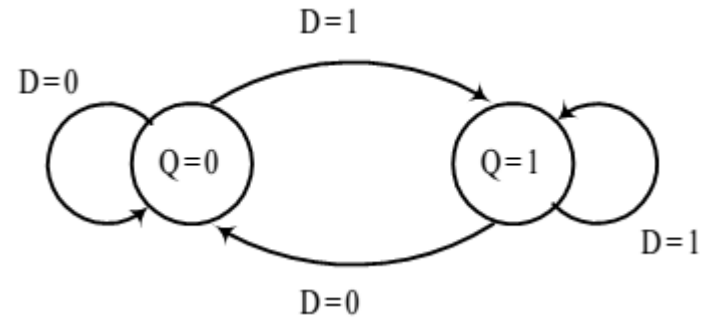
- **Characteristic Equation**

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

- **Excitation Table**

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

- **State Diagram**



# T Flip-Flop Descriptors

- **Characteristic Table**

T	Q(t+1)	Operation
0	$Q(t)$	No change
1	$\bar{Q}(t)$	Complement

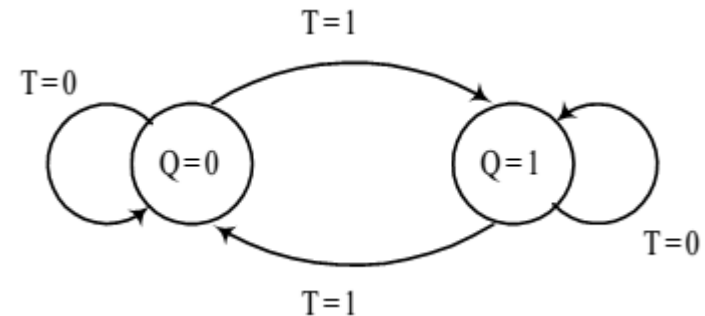
- **Characteristic Equation**

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

- **Excitation Table**

Q(t+1)	T	Operation
$Q(t)$	0	No change
$\bar{Q}(t)$	1	Complement

- **State Diagram**



# S-R Flip-Flop Descriptors

## ■ Characteristic Table

S	R	$Q(t+1)$	Operation
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	?	Undefined

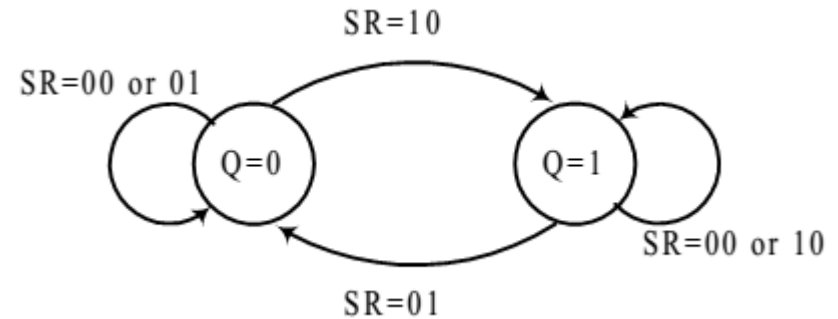
## ■ Characteristic Equation

$$Q(t+1) = S + \bar{R} Q, S \cdot R = 0$$

## ■ Excitation Table

$Q(t)$	$Q(t+1)$	S	R	Operation
0	0	0	X	No change
0	1	1	0	Set
1	0	0	1	Reset
1	1	X	0	No change

## ■ State Diagram



# J-K Flip-Flop Descriptors

## ■ Characteristic Table

J	K	Q(t+1)	Operation
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	$\bar{Q}(t)$	Complement

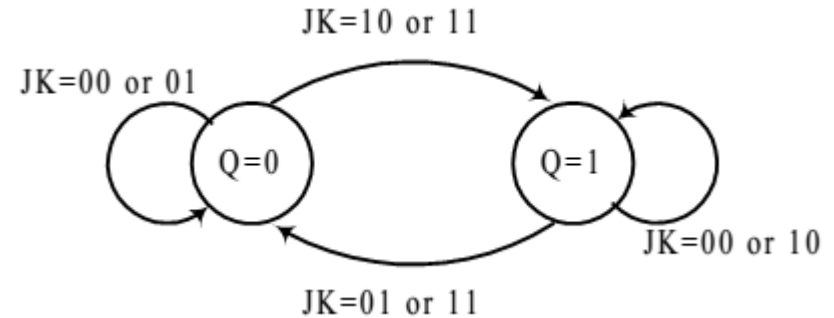
## ■ Characteristic Equation

$$Q(t+1) = J \bar{Q} + \bar{K} Q$$

## ■ Excitation Table

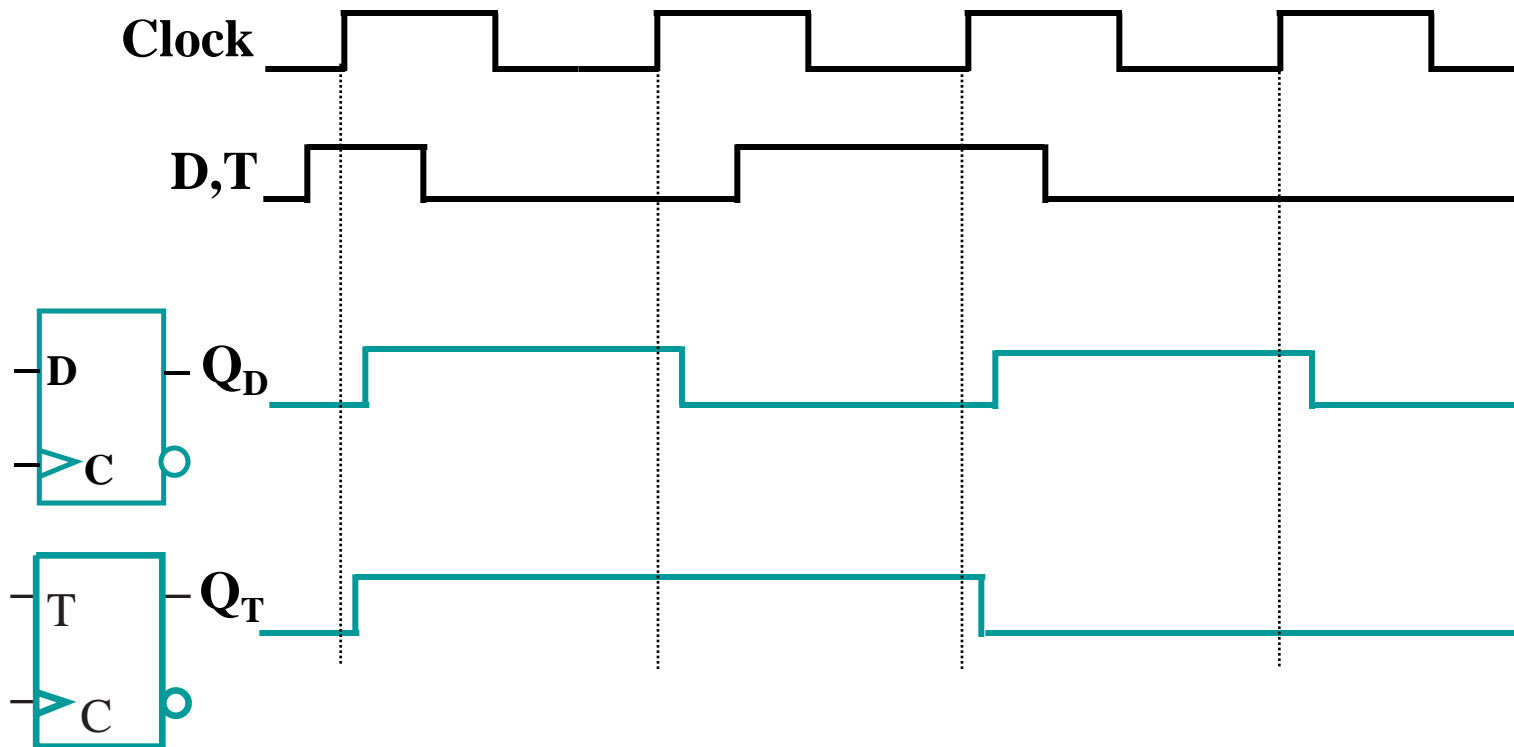
Q(t)	Q(t+1)	J	K	Operation
0	0	0	X	No change
0	1	1	X	Set
1	0	X	1	Reset
1	1	X	0	No Change

## ■ State Diagram



# Flip-flop Behavior Example

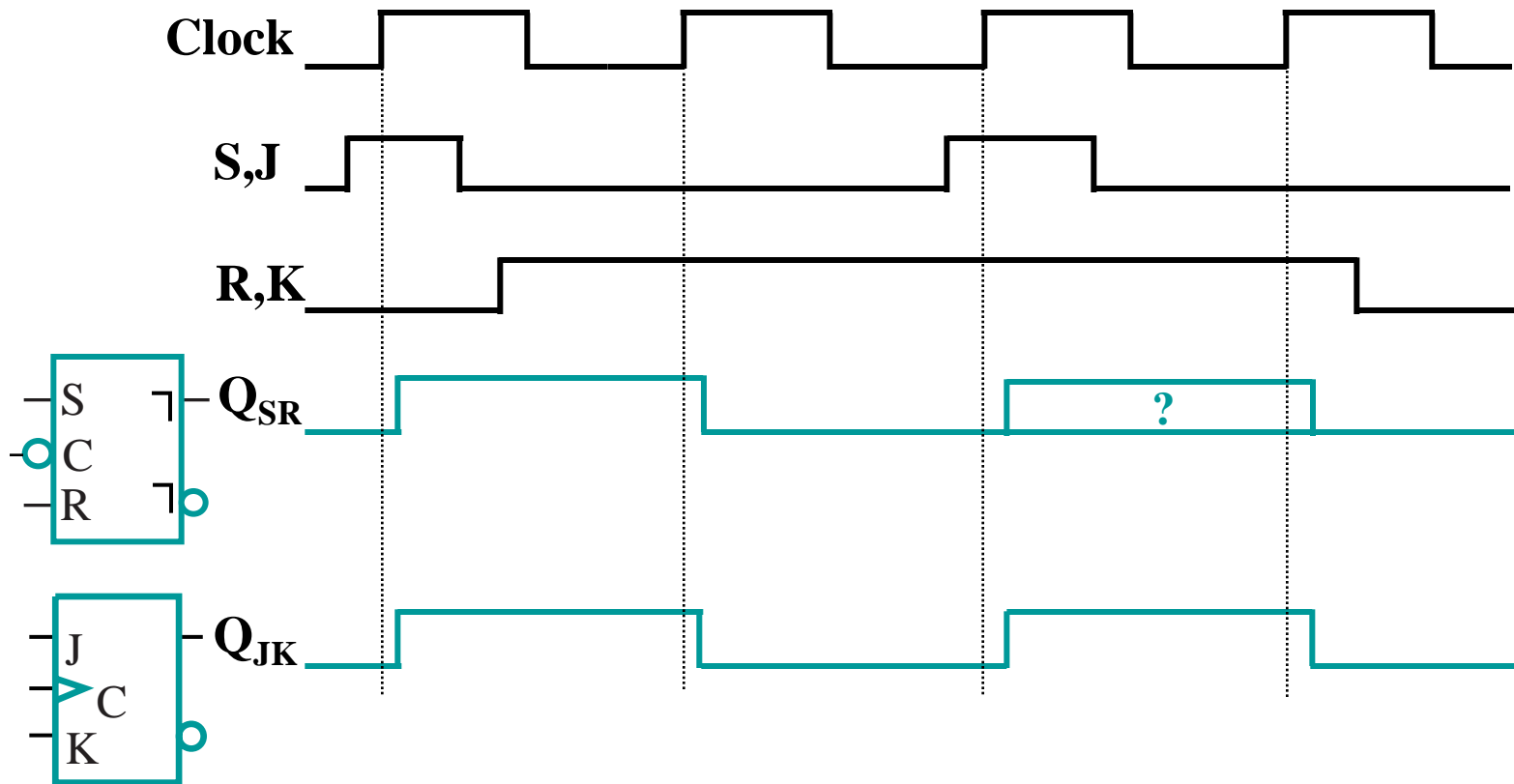
- Use the characteristic tables to find the output waveforms for the flip-flops shown:





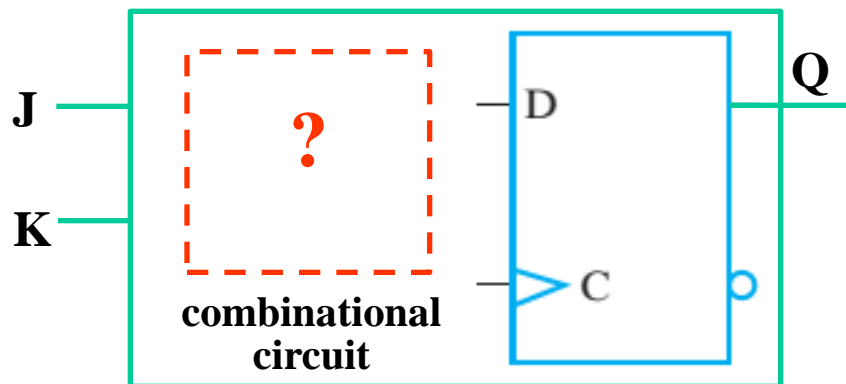
# Flip-Flop Behavior Example (continued)

- Use the characteristic tables to find the output waveforms for the flip-flops shown:



# Flip-Flop Conversion

- The various kinds of flip-flops in existence are S-R flip-flop, J-K flip-flop, D flip-flop, and T flip-flop. Each of them exhibits unique characteristics and result in different output for the same combination of input states.
- Whenever we want one flip-flop functionally equivalent to the behavior of the other, we need a **flip-flop conversion technique** to convert given Flip-Flop to desired Flip-Flop.



D Flip-Flop to J-K Flip-Flop

How to find the **input equation** for the given Flip-Flop (e.g., D Flip-Flop) ?

# Flip-Flop Conversion Example

## ■ D Flip-Flop to J-K Flip-Flop

### Characteristic Table of J-K Flip-Flop

J	K	Q(t)	Q(t+1)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

### Excitation Table of D Flip-Flop

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

### D Flip-Flop to J-K Flip-Flop Conversion Table

J	K	Q(t)	Q(t+1)	D
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	0	0
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	0	0

# Flip-Flop Conversion Example (continued)

## ■ D Flip-Flop to J-K Flip-Flop

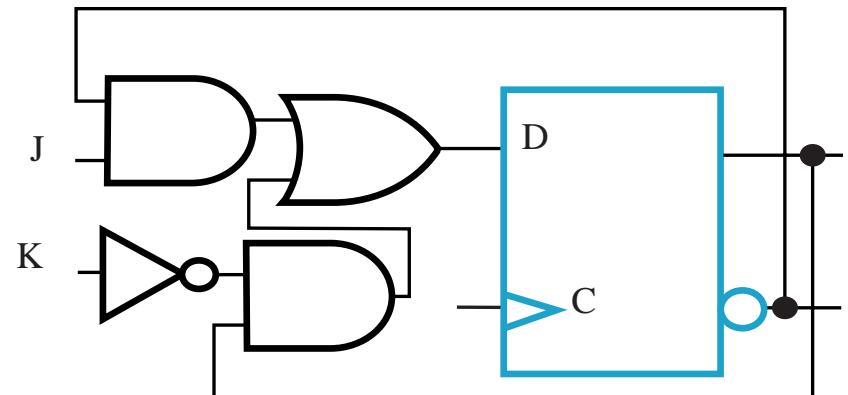
D Flip-Flop to J-K Flip-Flop  
Conversion Table

J	K	Q(t)	Q(t+1)	D
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	0	0
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	0	0



K/Q(t) \ J	00	01	11	10
0		1		
1	1	1		1

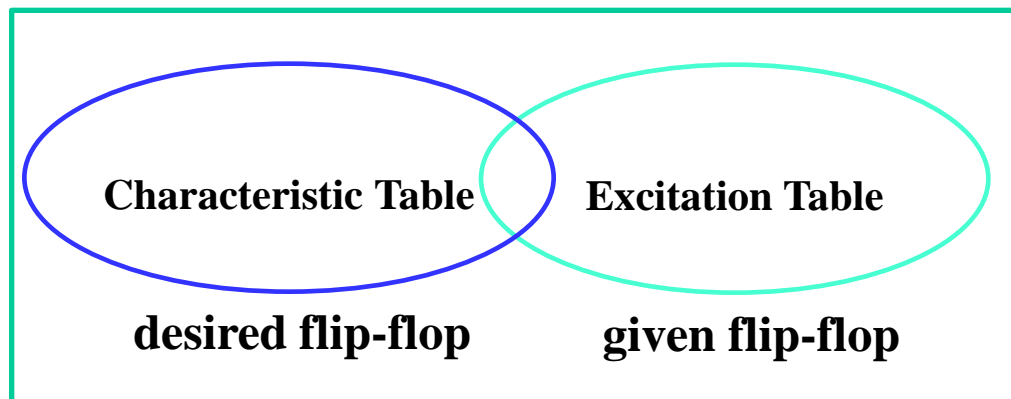
$$D = J\bar{Q} + \bar{K}Q$$



# Conversion of Flip-Flops

---

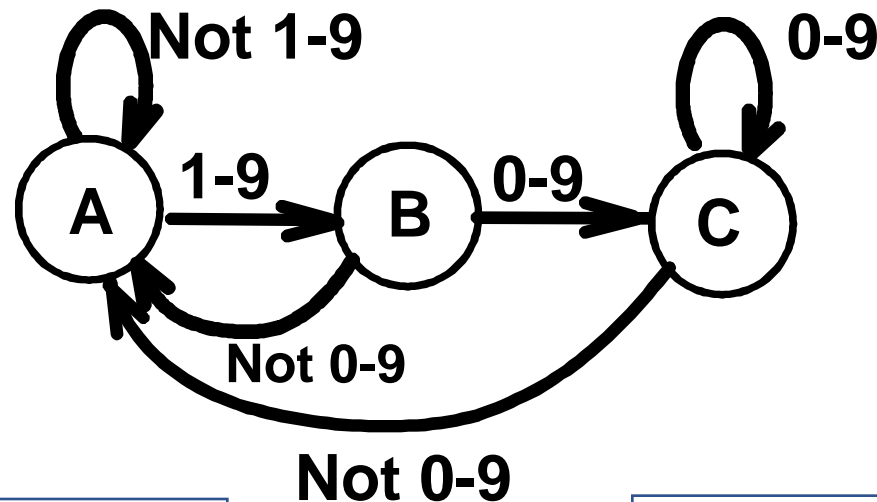
- Conversion of flip-flops involves following steps:
  - writing the **characteristic table** for the desired flip-flop
  - writing the **excitation table** for the given flip-flop
  - merging the above tables into **conversion table**
  - finding the **input equation** for the given flip-flop



Structure of a conversion table

# Limitations of Finite State Machines (1/2)

- FSM can use **limited states** to recognize **patterns** with **unlimited sequences** like a positive integer.
- positive integer:  $[1-9][0-9]^*$



**matches:** 1, 23, 788,  
470, 5074, 989500 ...

**but not:** 013, A432,  
23B, g432, @103...

# Limitations of Finite State Machines (2/2)

---

- Some examples that are not finite-state acceptable
  - $\{0^n 1^n \mid n \text{ a natural number}\} = \{\lambda, 01, 0011, 000111, \dots\}$
  - Language of balanced parentheses:  $\{\{\}, \{\{\}\}, \dots\}$
  - Palindromes:  $a, b, aa, bb, aaa, aba, \dots$
- Finite state machines can only accept **regular language**, while the above examples are all generated by **context-free grammar**.
- In regular expression, "**nesting**" is not allowed, as for context-free grammar, **recursive expressions** can be used in any arbitrary combinations.

# Assignments

---

## Reading:

- 4.5, 4.8, 4.12

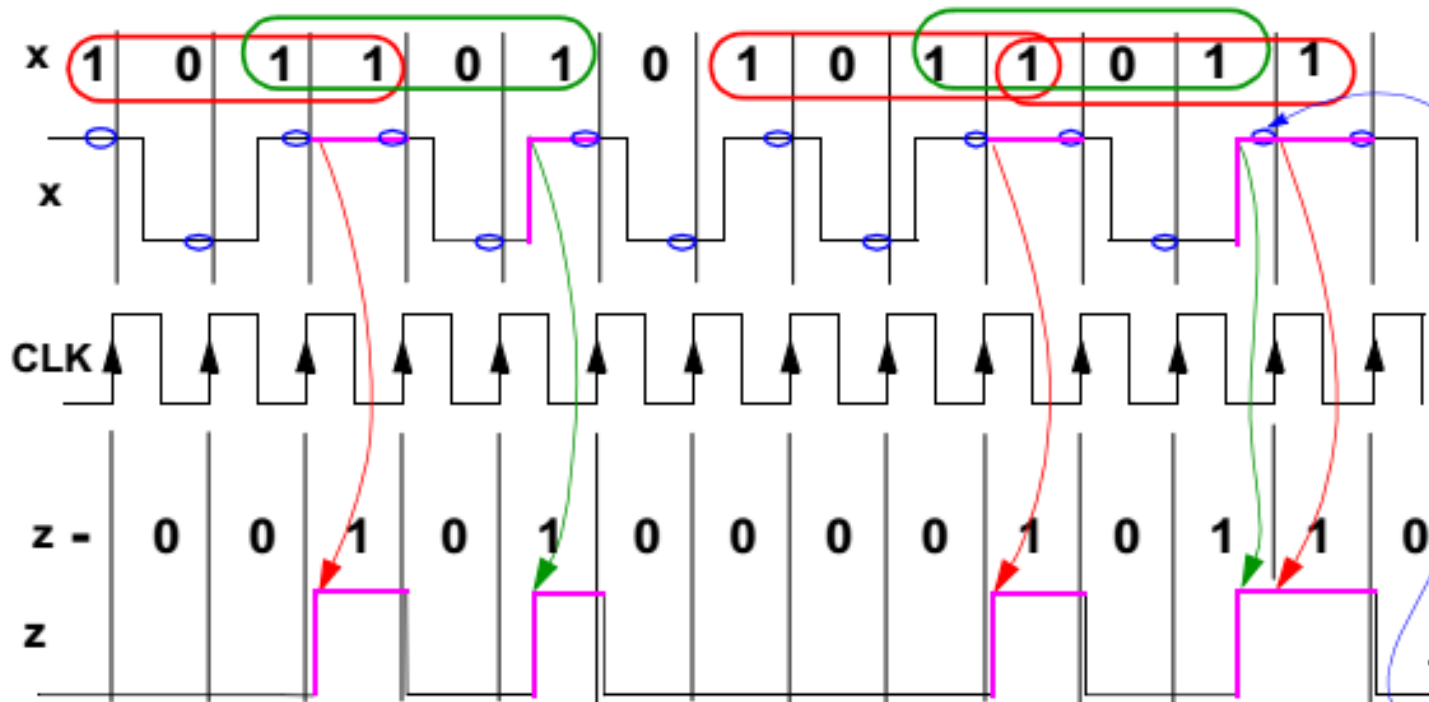
## Problem assignment:

- 4-13; 4-21; 4-22; 4-25; 4-29



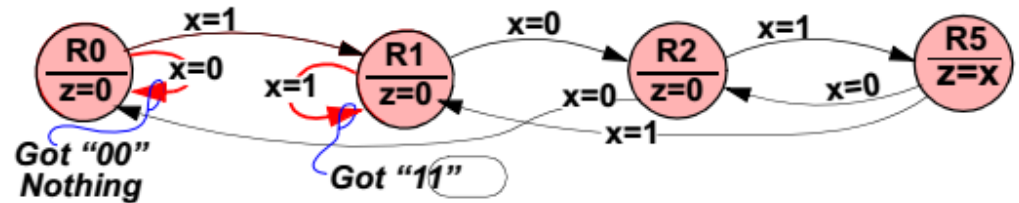
# Appendix A: Multiple-sequence Detector Example

- Example: Design a **Mealy** circuit to recognize **two** sequences “**1011**” or “**1101**”, with overlap.

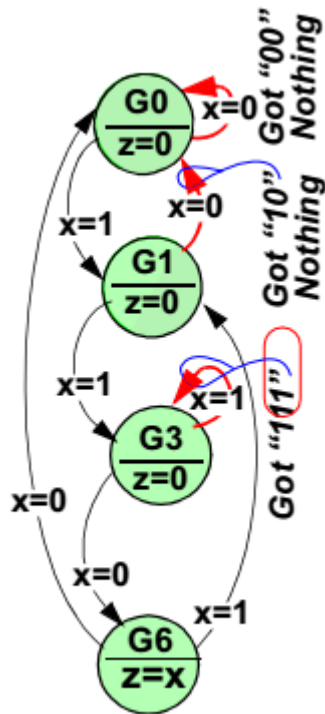


# Multiple-sequence Detector Example

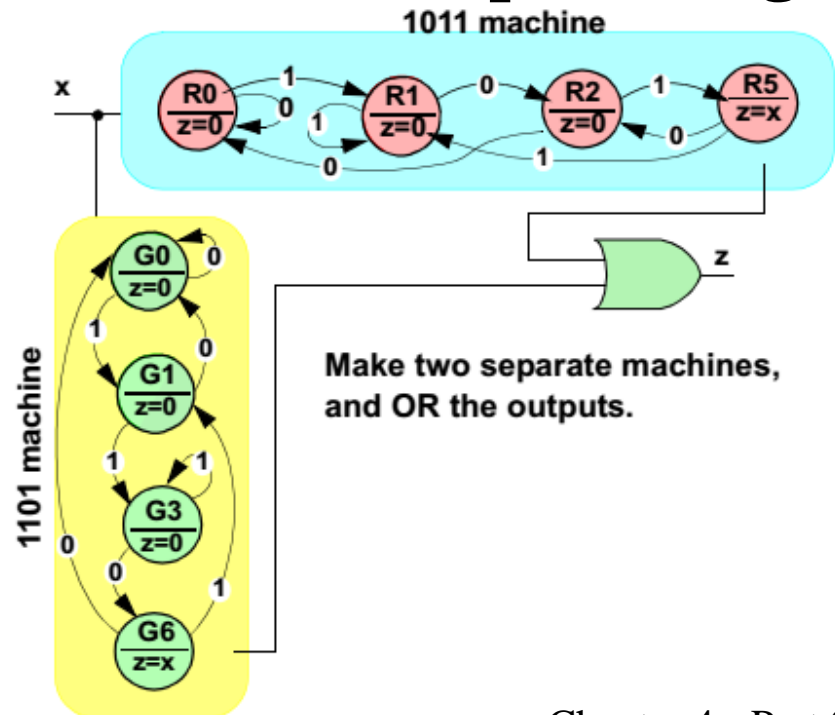
- Step 1: Do the **1011** sequence alone.



- Step 2: Do the **1101** sequence alone.

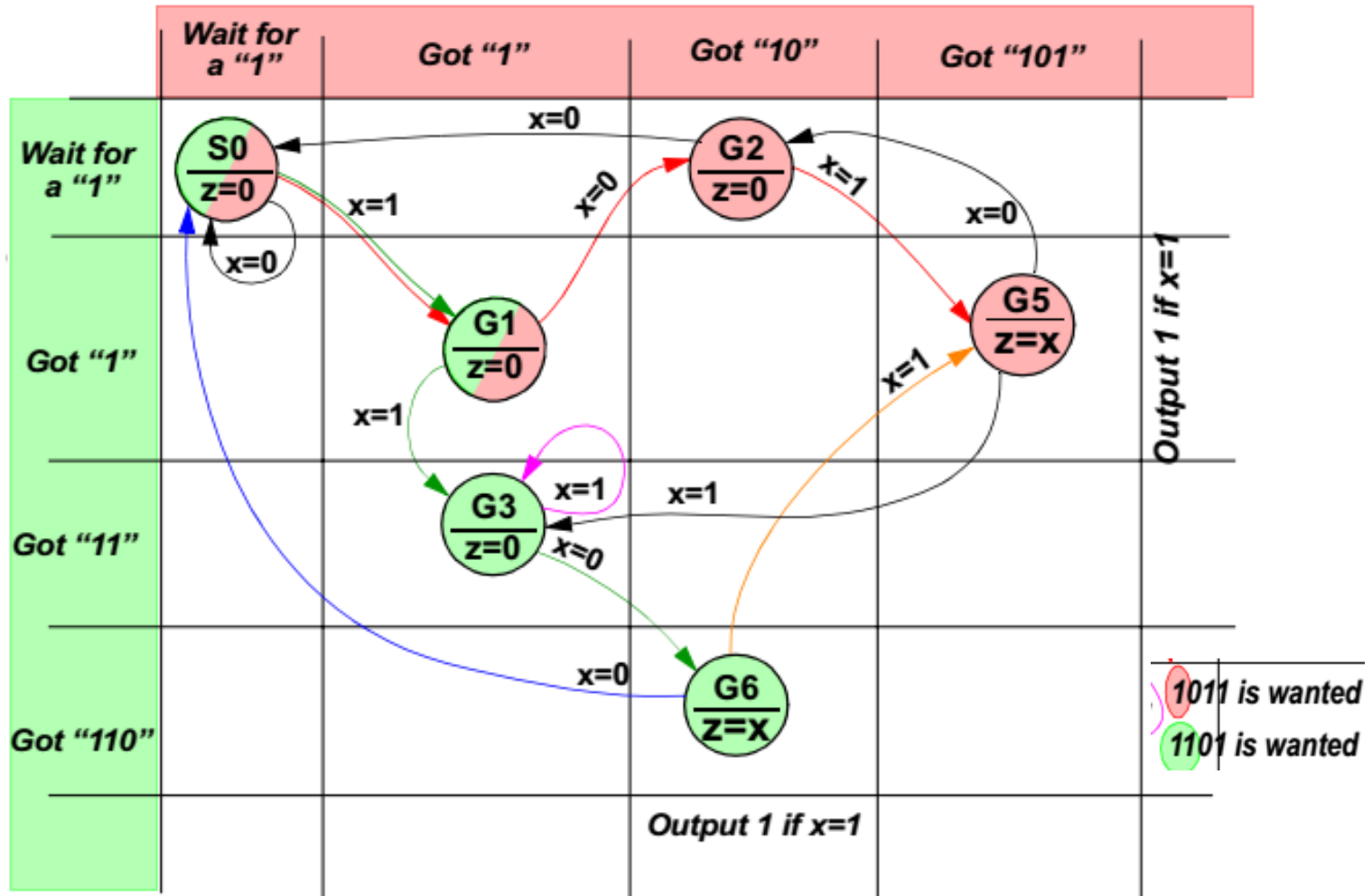


- Step 3: Combine two separate machines into a product graph.



# Multiple-sequence Detector Example

- Merge two state graphs into one.



# Appendix B: Reduction by Implication Chart

---

## ■ Example 2:

Present State	Next State/ Output	
	$X=0$	$X=1$
A	C/0	B/1
B	F/0	A/1
C	D/0	G/0
D	D/1	E/0
E	C/0	E/1
F	D/0	G/0
G	C/1	D/0

# Reduction by Implication Chart (Cont.)

## ■ Compare orderly to find equivalent states

- If states equivalent, mark “√”
- If states not equivalent, mark “×”
- If need **further comparison**, label pair of next states

		X	
		0	1
Qn	A	C/0	B/1
	B	F/0	A/1
	C	D/0	G/0
	D	D/1	E/0
	E	C/0	E/1
	F	D/0	G/0
	G	C/1	D/0

	A	B	C	D	E	F
B	CF					
C	×	×				
D	×	×	×			
E	BE	FC AE	×	×		
F	×	×	√	×	×	
G	×	×	×	CD DE	×	×


# Reduction by Implication Chart (Continued)

- Correlatively compare to find equivalent states

CD, DE are not equivalent, so DG are not equivalent

AB:  $AB \rightarrow CF$

AE:  $AE \rightarrow BE \rightarrow CF$



B	CF					
C	X	X				
D	X	X	X			
E	BE	AE CF	X	X		
F	X	X	✓	X	X	
G	X	X	X	<del>CD</del> <del>DE</del>	X	X
	A	B	C	D	E	F

All the states in the cycle are equivalent

# Reduction by Implication Chart (Continued)

---

- **Determine Max Equivalent Classes**
  - **Four equivalent pairs (A,B), (A,E), (B,E), (C,F)**
  - **Max Equivalent Classes (A,B,E), (C,F)**
  - **Four states (A,B,E), (C,F), (D), (G)**
  - **Label the four states as a, b, c, d**

# Reduction by Implication Chart (Continued)

---

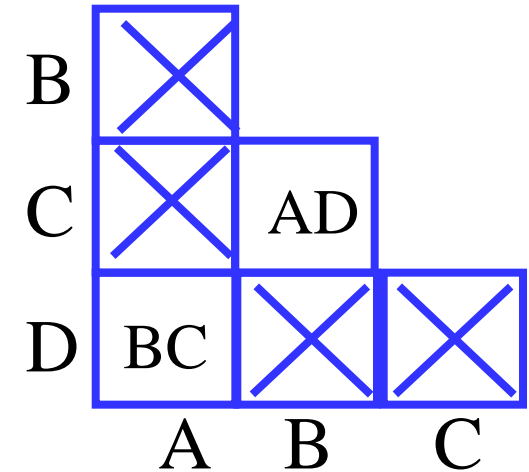
- **Minimized State Table:**

<b>Present State</b>	<b>Next State/ Output</b>	
	<b><math>x=0</math></b>	<b><math>x=1</math></b>
<b>a</b>	<b>b/0</b>	<b>a/1</b>
<b>b</b>	<b>c/0</b>	<b>d/0</b>
<b>c</b>	<b>c/1</b>	<b>a/0</b>
<b>d</b>	<b>b/1</b>	<b>c/0</b>



# Reduction by Implication Chart (Continued)

- AD equivalent
- BC equivalent



Q \ X	0	1
A	A/0	B/0
B	A/1	C/0
C	D/1	C/0
D	A/0	C/0

Q \ X	0	1
A	A/0	B/0
B	A/1	B/0

# Appendix C: Heuristic Methods for State Assignment

Present State	Next State		Output	
	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

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

Counting Order Assignment

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

Gray Code Assignment

- The state assignment guidelines attempt to maximize the adjacent groupings of 1's in the next-state and output functions.

# Heuristic Methods for State Assignment (Continued)

- **Highest priority:** States with the **same next state for a given input transition** should be given adjacent assignments in the state table (**fan-in oriented**)

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

Gray Code Assignment

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

Gray Code Assignment

- **Medium priority:** **Next states of the same state with adjacent inputs** should be given adjacent assignments (**fan-out oriented**)

# Heuristic Methods for State Assignment (Continued)

---

- **Lowest priority: States with the same output** for a given input should be given adjacent assignments (**output oriented**)
- **Assigning “0”**: **starting state** or **the most frequently used state** in the state table should be assigned to “0”
- The first rule (**fan-in oriented**) is the most important, should be considered first
- According to the former three rules, state pairs with more present times should be assigned adjacent codes with high priority

# Example: State Assignment

---

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

- Rule 1: A-B, A-C
- Rule 2: C-D, A-C, B-D, A-B
- Rule 3: A-B, A-C, B-C
- Rule 4:  $A = 0$

# Example: State Assignment (Continued)

---

- **n=2, since 2 triggers is enough to represent 4 states**
- **Determine the assignment**
  - **Rule 1: A-B, A-C**
  - **Rule 2: C-D, A-C, B-D, A-B**
  - **Rule 3: A-B, A-C, B-C**
  - **Rule 4: A = 0**

# Example: State Assignment (Continued)

$Y_2 \backslash Y_1$	0	1
0	A	B
1	C	D

	$Y_2$	$Y_1$
A	0	0
B	0	1
C	1	0
D	1	1

- Rule 1: **A-B**, **A-C**
- Rule 2: **C-D**, **A-C**, **B-D**, **A-B**
- Rule 3: **A-B**, **A-C**, **B-C**
- Rule 4: **A = 0**

# Example: State Assignment (Continued)

## ■ Final State Table

Present State	Next State/ Output	
	$X=0$	$X=1$
<b>A</b>	<b>C/0</b>	<b>D/0</b>
<b>B</b>	<b>C/0</b>	<b>A/0</b>
<b>C</b>	<b>B/0</b>	<b>D/0</b>
<b>D</b>	<b>A/1</b>	<b>B/1</b>

Present State $y_2 \ y_1$	$y_2^{(t+1)}y_1^{(t+1)} /$ Output	
	$X=0$	$X=1$
<b>0 0</b>	<b>10/0</b>	<b>11/0</b>
<b>0 1</b>	<b>10/0</b>	<b>00/0</b>
<b>1 1</b>	<b>00/1</b>	<b>01/1</b>
<b>1 0</b>	<b>01/0</b>	<b>11/0</b>

- Sometimes, the assignment that fits the requirement is not unique. And any of them is suitable.