Digital Logic Design

Chapter 5 – Sequential Logic (II)

Sequential Circuit Design

浙江大学计算机学院　王总辉
zhwang@zju. edu. cn
http://course.zju.edu.cn

2021年10月

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**

Sequential Circuit Design

Event Description →State diagram

→State table/compressed state

→ state allocation/Assignment

→ state equation

→ Output Equation

→choose Flip-Flop

→The Excitation/input equation

→ Optimization

→Logic circuit/module mapping

The Design Procedure

- **Specification**
- **Formulation - Obtain a state diagram or state table**
- **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 occurence.
- 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.

Sequence Recognizer Procedure

- **To develop a sequence recognizer state diagram:**

properly 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.
- ☐ The final state represents the input sequence (possibly less the final input value) occurrence.

non-sequence

- ☐ 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.

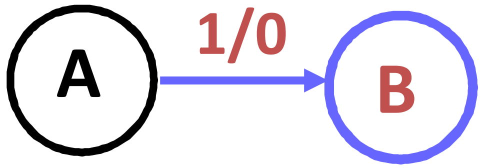
- **The last step is required because the circuit must recognize the input sequence *regardless of where it occurs within the overall sequence applied since "reset."*.**

Sequence Recognizer Example

- **Example: Recognize the sequence 1101**
  - Note that the sequence 1111101 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., 1101101 or 1101101.**
- **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."

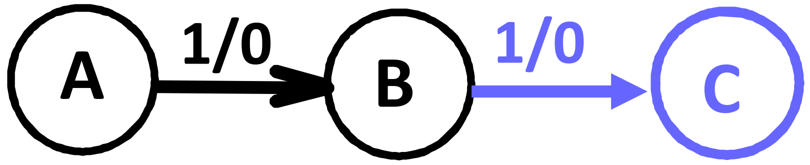


  - State "A" is the initial state, and state "B" is the state which represents the fact that the "first" one in the input subsequence has occurred. The output symbol "0" means that the full recognized sequence has not yet occurred.

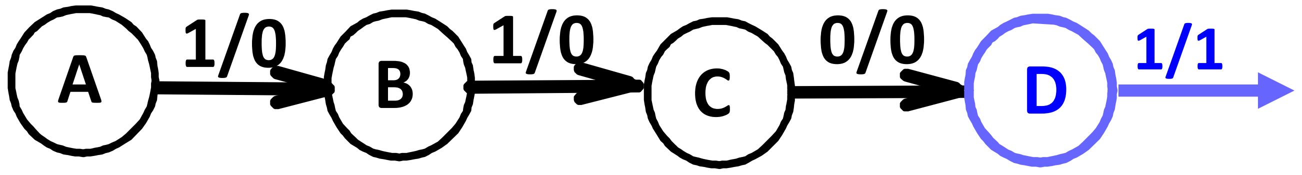
Example: Recognize 1101:

Properly sequence

- **After one more 1, we have:**
  - C is the state obtained when the input sequence has two "1"s.

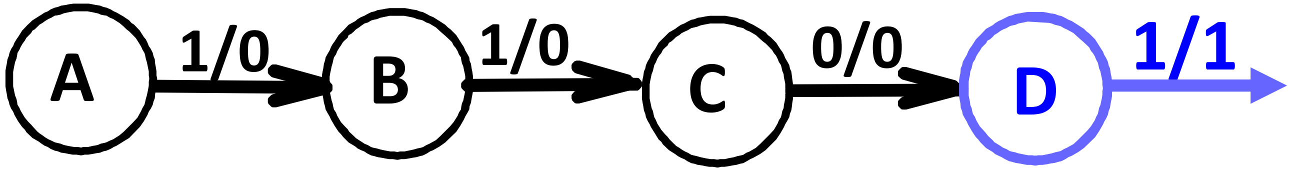


- **Finally, after 110 and a 1, we have:**

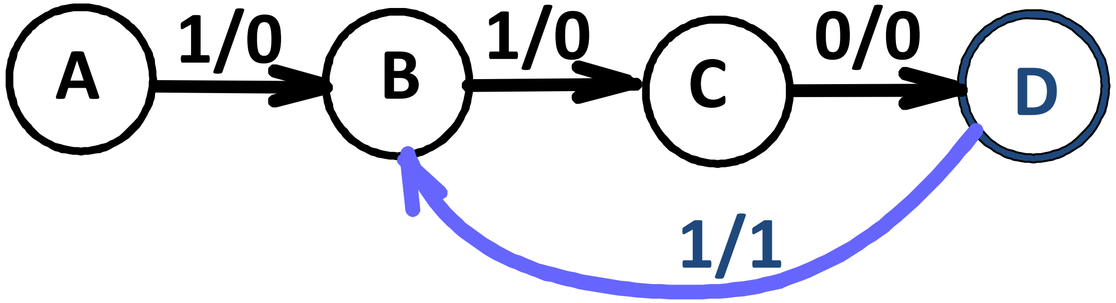


  - Transition arcs are used to denote the output function (Mealy Model)
  - Output 1 on the arc from D means the sequence has been recognized
  - To what state should the arc from state D go? Remember: 110<u>1101</u> ?
  - Note that D is the last state but the output 1 occurs for the input applied in D. This is the case when a *Mealy model* is assumed.

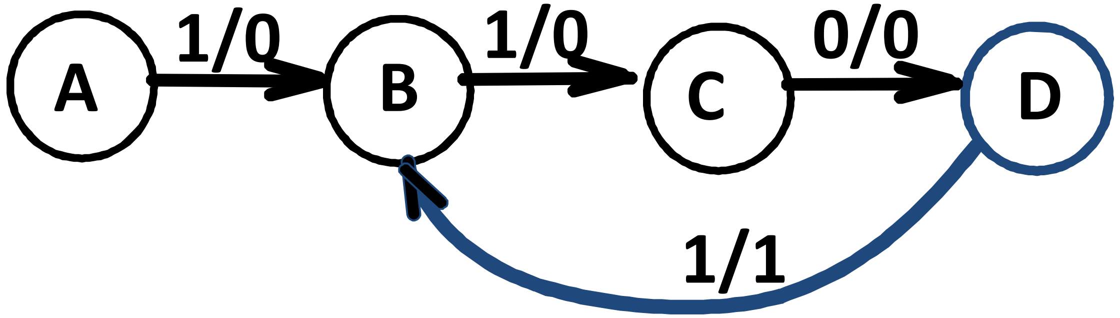
Example: Recognize 1101 (continued)



- 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)

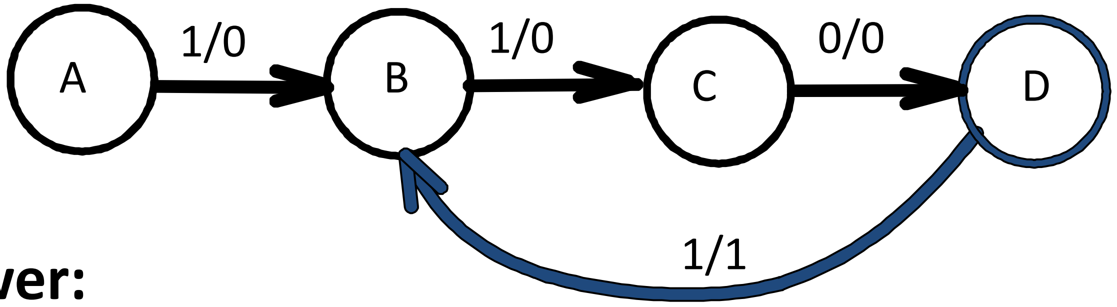

$$\begin{tikzcd} A \arrow[r, "1/0"] & B \arrow[r, "1/0"] & C \arrow[r, "0/0"] & D \arrow[ll, bend left, "1/1"] \end{tikzcd}$$

- **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 :

non-sequence

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



- **Answer:**

"0" arc from A

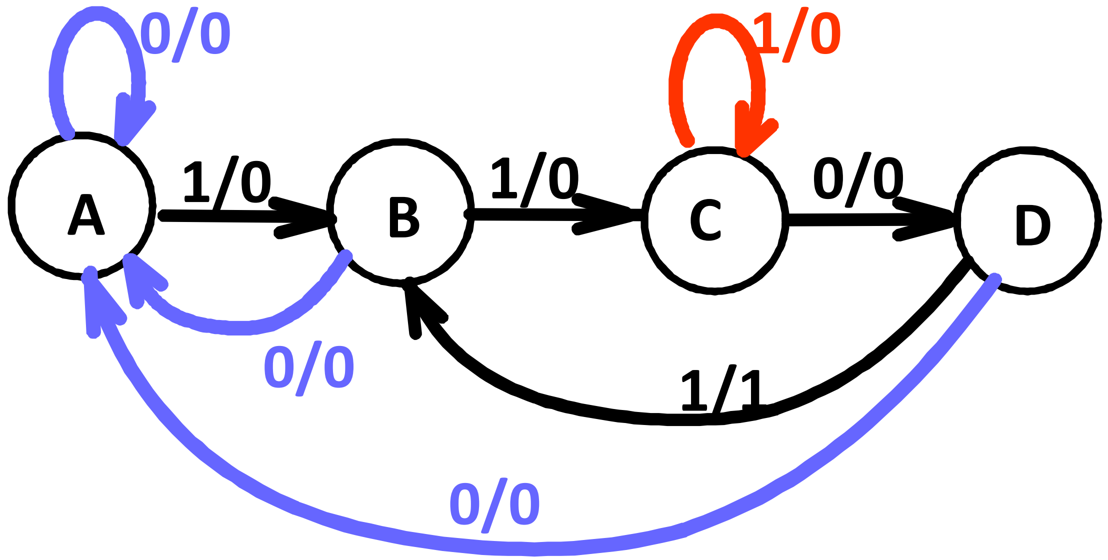
"0" arc from B

"1" arc from C

"0" arc from D.

Example: Recognize 1101 (continued)

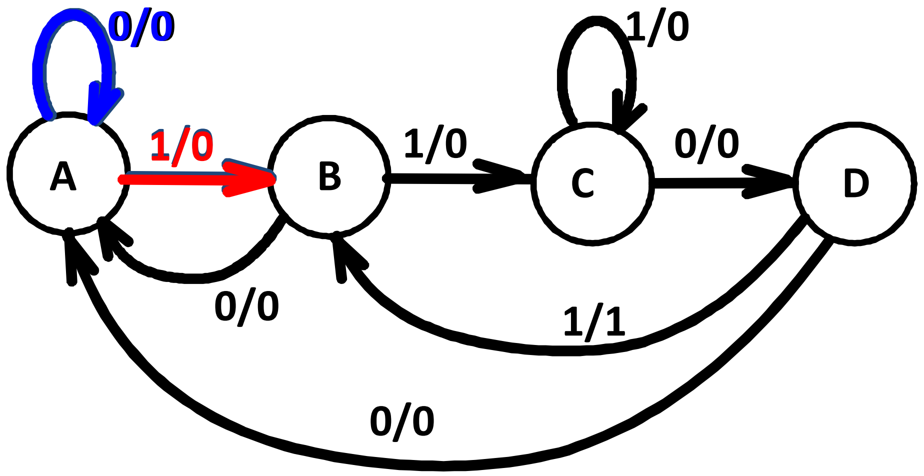
- State transition arcs must represent the fact that an input subsequence has occurred. Thus we get:


$$\begin{tikzcd} A \arrow[loop above, "0/0"] \arrow[r, "1/0"] & B \arrow[l, bend left, "0/0"] \arrow[r, "1/0"] & C \arrow[loop above, "1/0"] \arrow[r, "0/0"] & D \arrow[ll, bend left, "1/1"] \arrow[lll, bend left=50, "0/0"] \end{tikzcd}$$

- 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

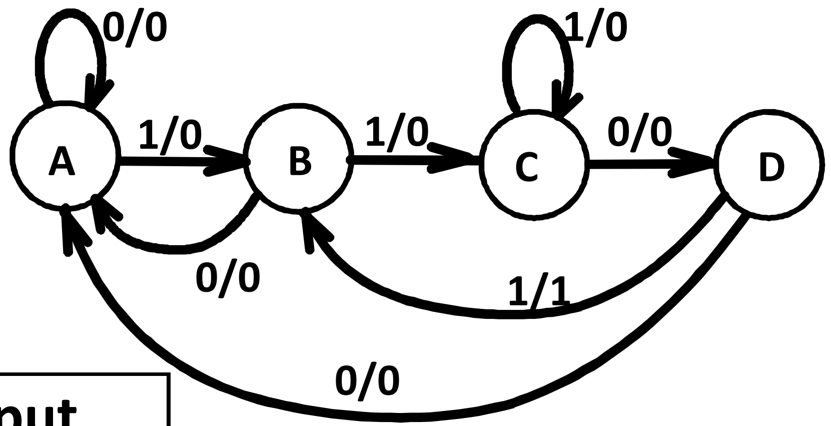
- From the State Diagram, we can fill in the State Table.
- There are 4 states, one input, and one output. We will choose the form with four rows, one for each current state.
- From State A, the 0 and 1 input transitions have been filled in along with the outputs.



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

Formulation: Find State Table

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


$$\begin{tikzcd}
A \arrow[loop above, "0/0"] \arrow[r, "1/0"] & B \arrow[l, bend left, "0/0"] \arrow[r, "1/0"] & C \arrow[loop above, "1/0"] \arrow[r, "0/0"] & D \arrow[ll, bend left, "1/1"] \arrow[lll, bend left=60, "0/0"]
\end{tikzcd}$$

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

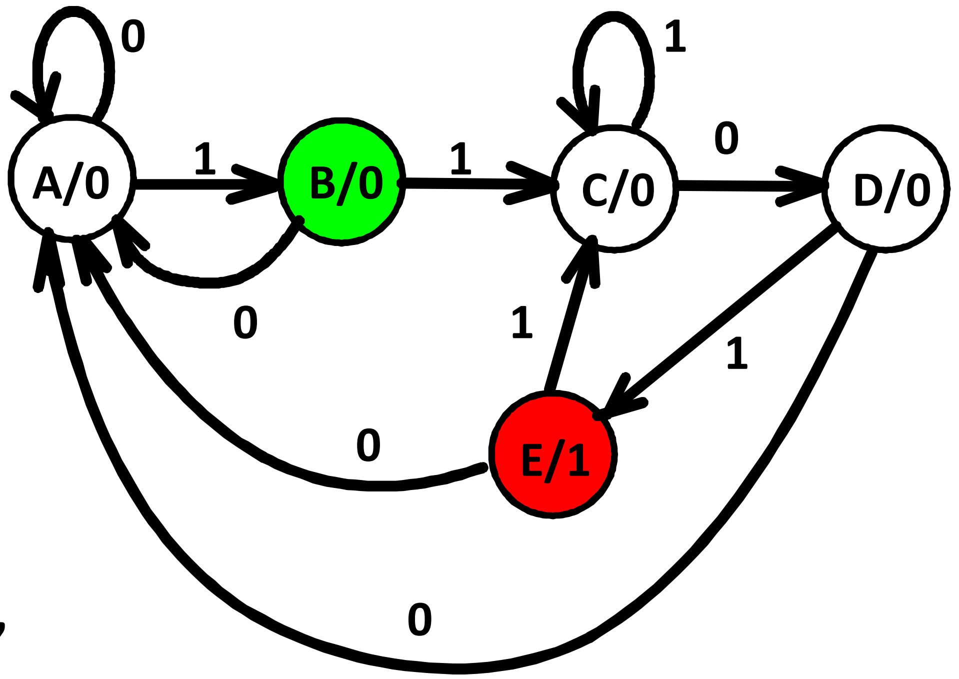
- What would the state diagram and state table look like for the Moore model?

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)

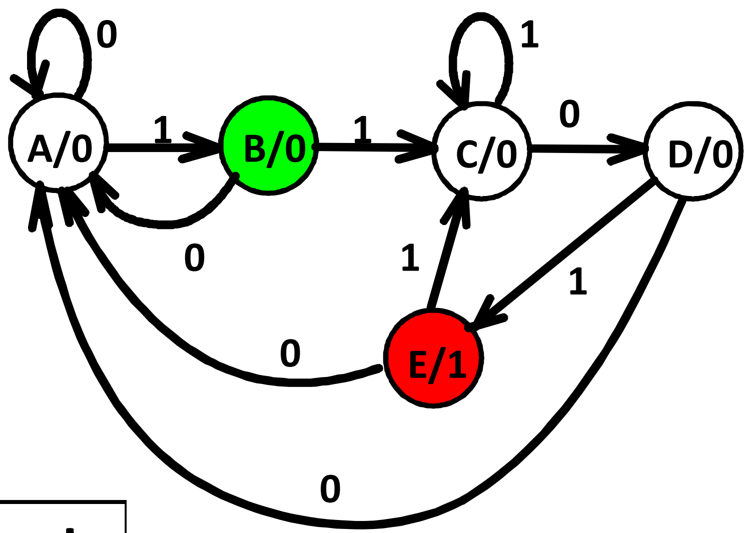
- We mark outputs on states for Moore model
- Arcs now show only state transitions
- Add a new state E to produce the output 1
- Note that the new state, E produces the same behavior



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
- Memory aid re more state in the Moore model: "Moore is More."



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

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**

State Assignment – Example 1

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

- **How may 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 x=0	Next State x=1	Output x=0	Output x=1
A	A	B	0	0
B	A	C	0	0
C	D	C	0	0
D	A	B	0	1

- **How may 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: state equation

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

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

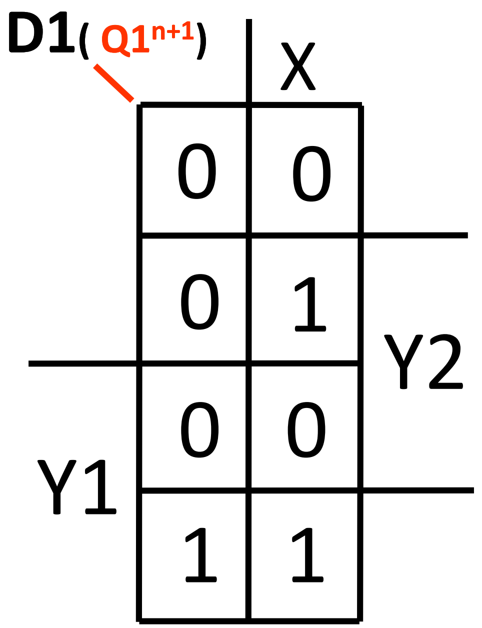
- **Gray Code Assignment: A = 0 0, B = 0 1, C = 1 1, D = 1 0**
- **The resulting coded state table:**

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

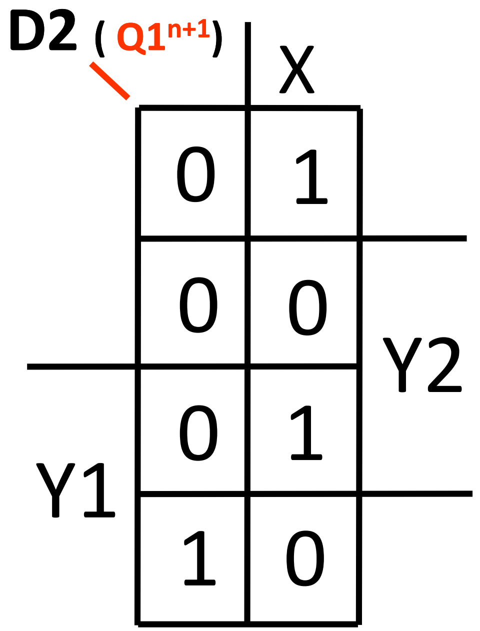
Find Flip-Flop Input and Output Equations: Example 2 – Counting Order Assignment

1. State equation
2. Choose Flip-Flop
3. Input equation

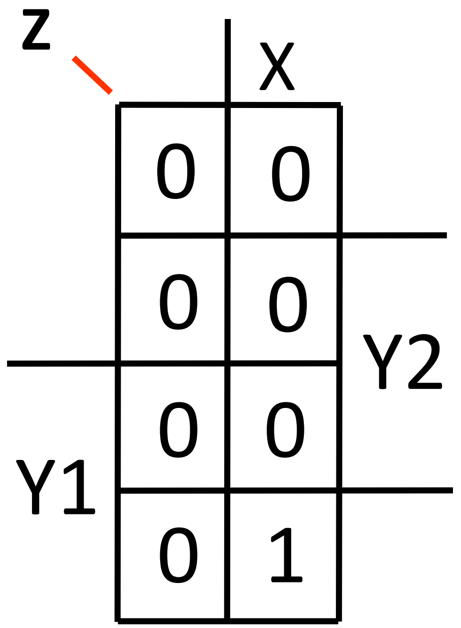
- **Assume D flip-flops $Q^{(n+1)} = D$**
- **Interchange the bottom two rows of the state table, to obtain K-maps for D1($Q1^{n+1}$), D2($Q2^{n+1}$), and Z:**



0	0
0	1
0	0
1	1



0	1
0	0
0	1
1	0

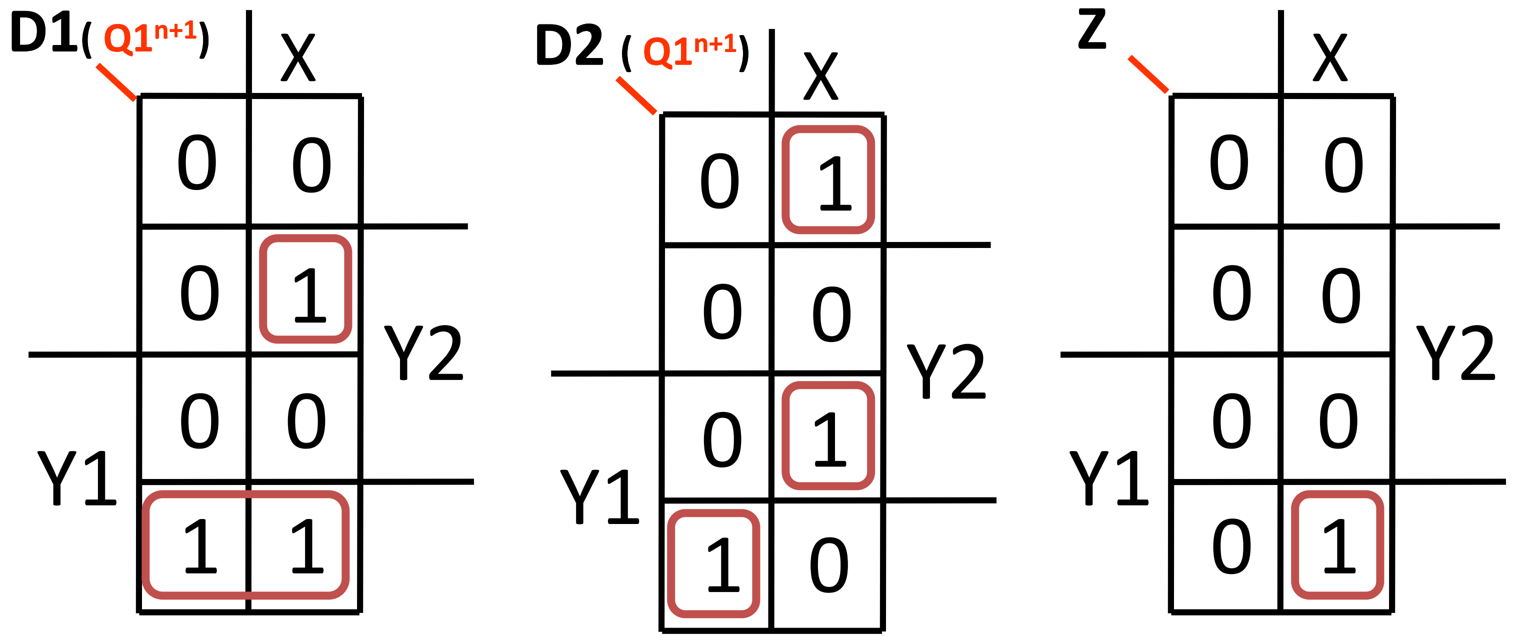


0	0
0	0
0	0
0	1

Optimization: Example 2: Counting Order Assignment

- **Performing two-level optimization:**

$Q^{(n+1)} = D$



$D_1 = Y_1\overline{Y_2} + X\overline{Y_1}Y_2$

$D_2 = \overline{X}Y_1\overline{Y_2} + X\overline{Y_1}\,\overline{Y_2} + XY_1Y_2$

$Z = XY_1\overline{Y_2}$

Gate Input Cost = 22

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

- **Assume D flip-flops**
- **Obtain K-maps for $D_1(Q_1^{n+1})$, $D_2(Q_2^{n+1})$, and Z:**

D1($Q1^{n+1}$)

		X	
	0	0	
	0	1	Y2
Y1	1	1	Y2
Y1	0	0	

D2 ($Q1^{n+1}$)

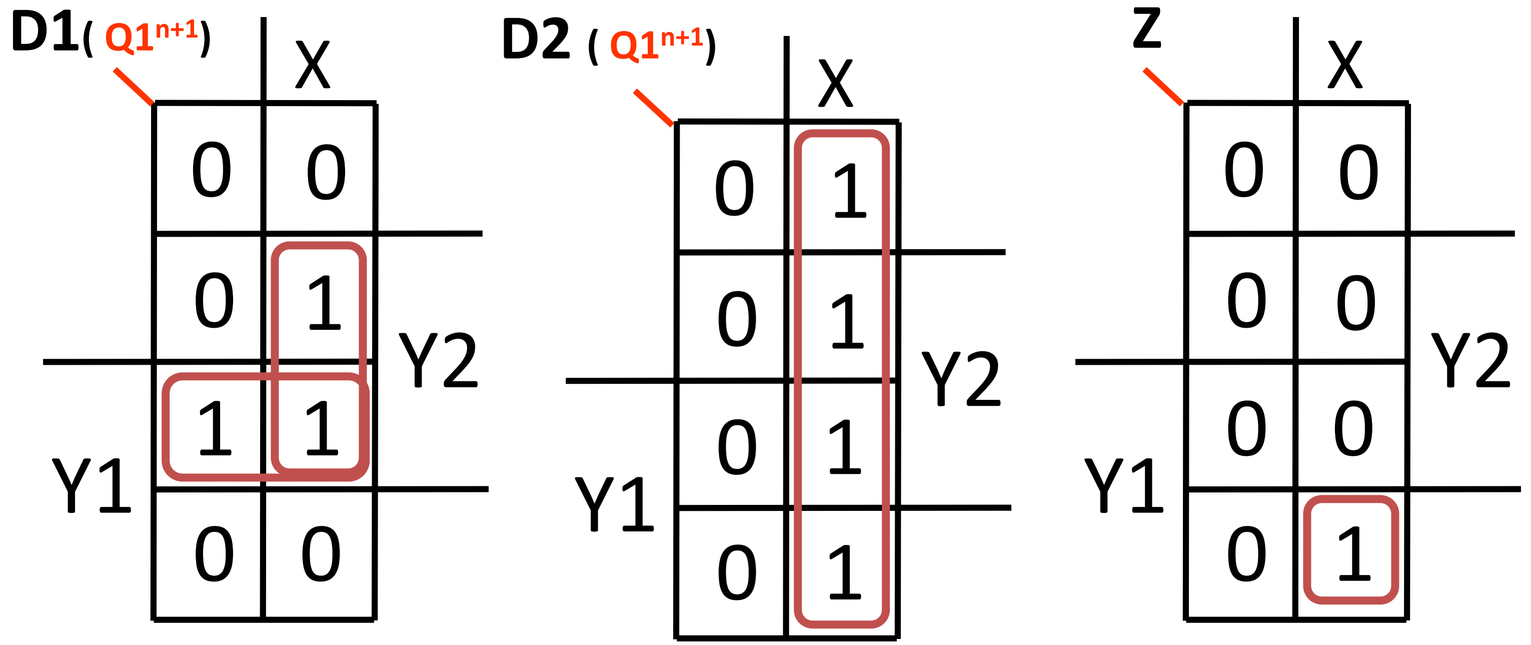
		X	
	0	1	
	0	1	Y2
Y1	0	1	Y2
Y1	0	1	

Z

		X	
	0	0	
	0	0	Y2
Y1	0	0	Y2
Y1	0	1	

Optimization: Example 2: Assignment 2

- **Performing two-level optimization:**



$D_1 = Y_1Y_2 + XY_2$
$D_2 = X$
$Z = XY_1\overline{Y_2}$

Gate Input Cost = 9
Select this state assignment to complete design in slide

One Flip-flop per State (One-Hot) Assignment

- Example codes for four states: (Y_3, Y_2, Y_1, Y_0) = 0001, 0010, 0100, and 1000.
- In equations, need to include only the variable that is 1 for the state, e. g., state with code 0001, is represented in equations by Y_0 instead of $\overline{Y_3}\,\overline{Y_2}\,\overline{Y_1}\,Y_0$ because all codes with 0 or two or more 1s have don’t care next state values.
- Provides simplified analysis and design
- Combinational logic may be simpler, but flip-flop cost higher – may or may not be lower cost

State Assignment – Example 2 (One-Hot)

- **One-Hot Assignment : A = 0001, B = 0010, C = 0100, D = 1000**
  The resulting coded state table:

Present State	Next State x = 0	Next State x = 1	Output x = 0	Output x = 1
0001	0001	0010	0	0
0010	0001	0100	0	0
0100	1000	0100	0	0
1000	0001	0010	0	1

Optimization: Example 2: One Hot Assignment

- **Equations read from 1 next state variable entries in table:**

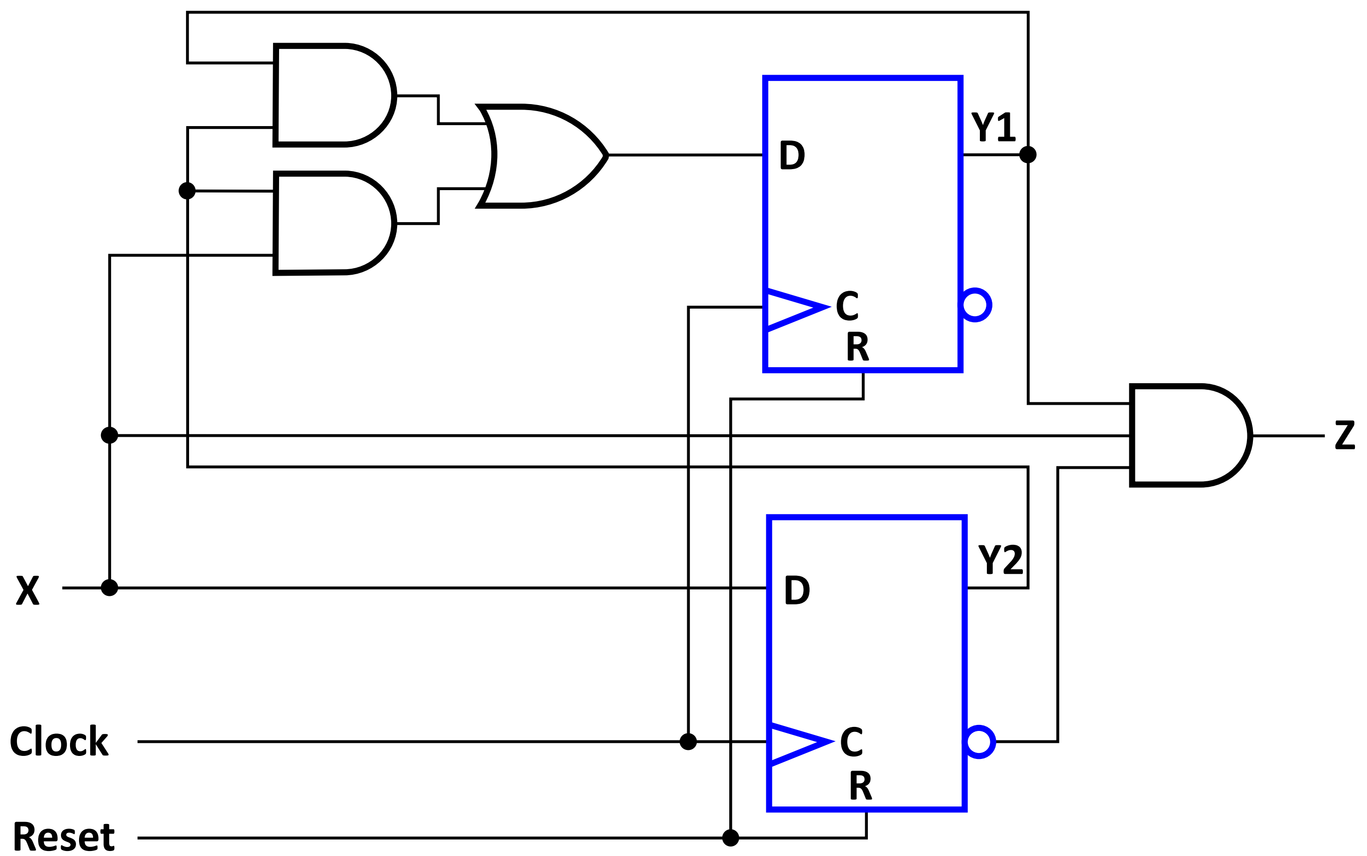
  $D_0 = \overline{X}(Y_0 + Y_1 + Y_3)$ or $\overline{X}\,\overline{Y_2}$
  $D_1 = X(Y_0 + Y_3)$
  $D_2 = X(Y_1 + Y_2)$ or $X(\overline{Y_0} + \overline{Y_3})$
  $D_3 = \overline{X}\,Y_2$
  $Z = XY_3$ **Gate Input Cost = 19**

- **Combinational cost intermediate plus cost of two more flip-flops needed.**

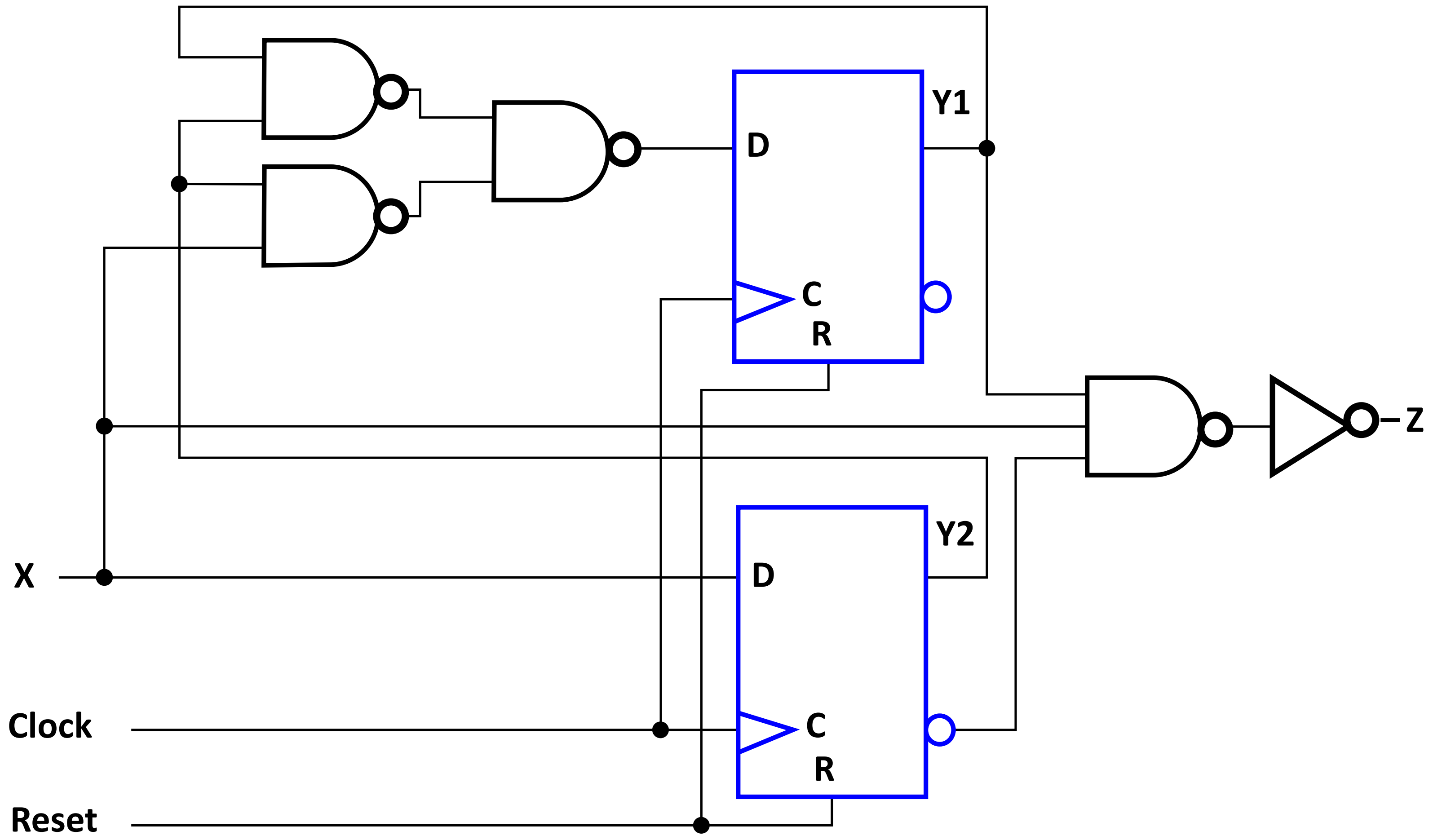
Map Technology

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

- **Initial Circuit:**



Mapped Circuit - Final Result

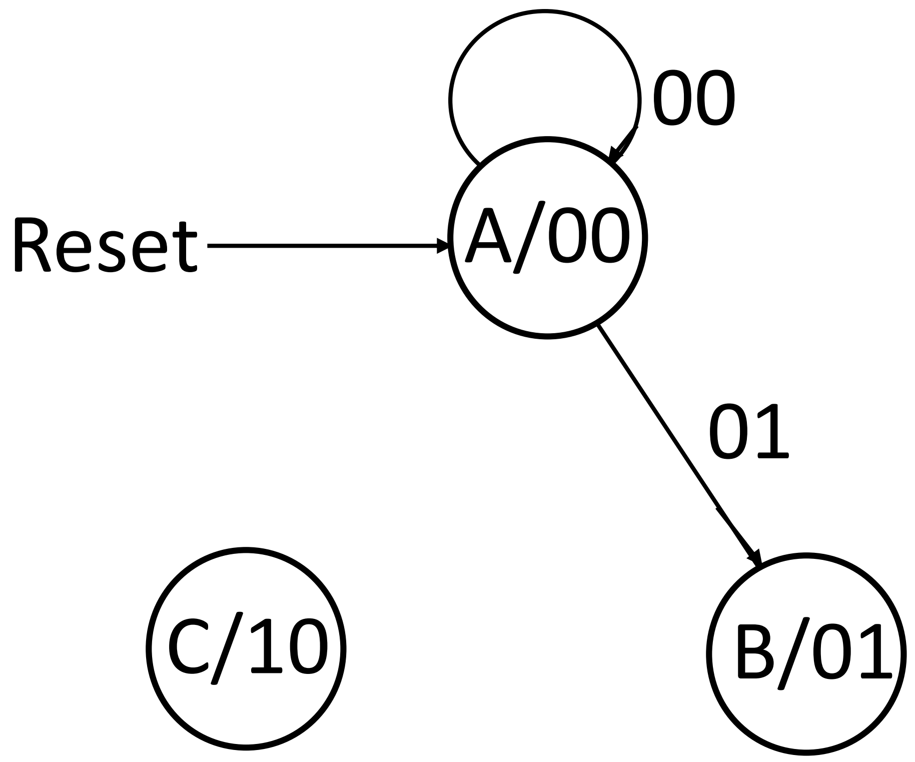


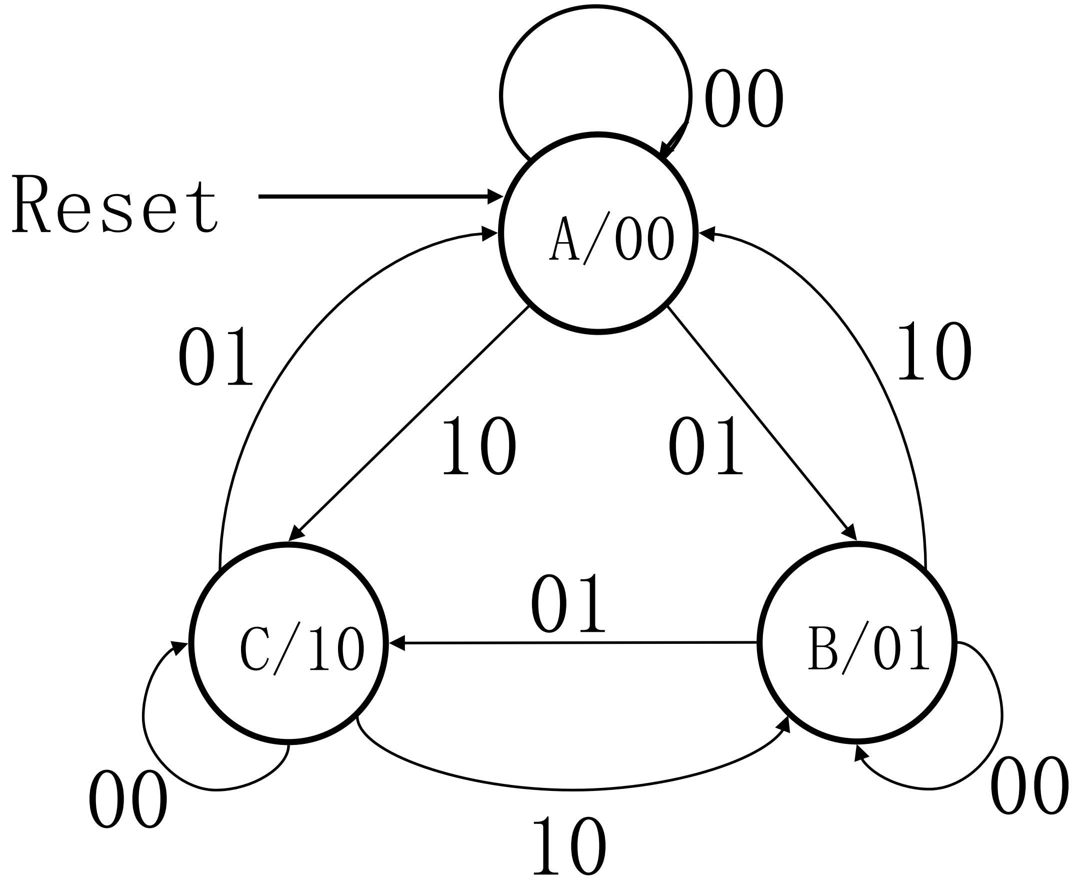
Sequential Design: Example 3

- **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 modulo 3 = 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.
- **Stored sum: (Y_1,Y_0), Input: (X_1,X_0), Output: (Z_1,Z_0)**

Example 3 (continued)

- **Complete the state diagram:**




00
Reset
A/00
01
10
10
01
C/10
01
B/01
00
00
10

Example 3 (continued)

- Complete the state table

Y1Y0 \ X1X0	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		X		00
B (01)			X		01
- (11)	X	X	X	X	11
C (10)			X		10

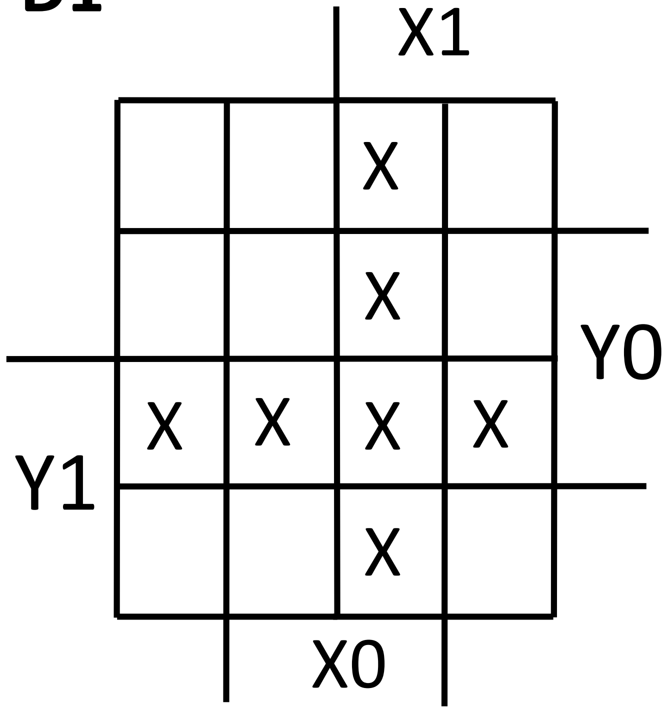
- State Assignment: $(Y_1,Y_0) = (Z_1,Z_0)$
- Codes are in gray code order to ease use of K-maps in the next step

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

Example 3 (continued)

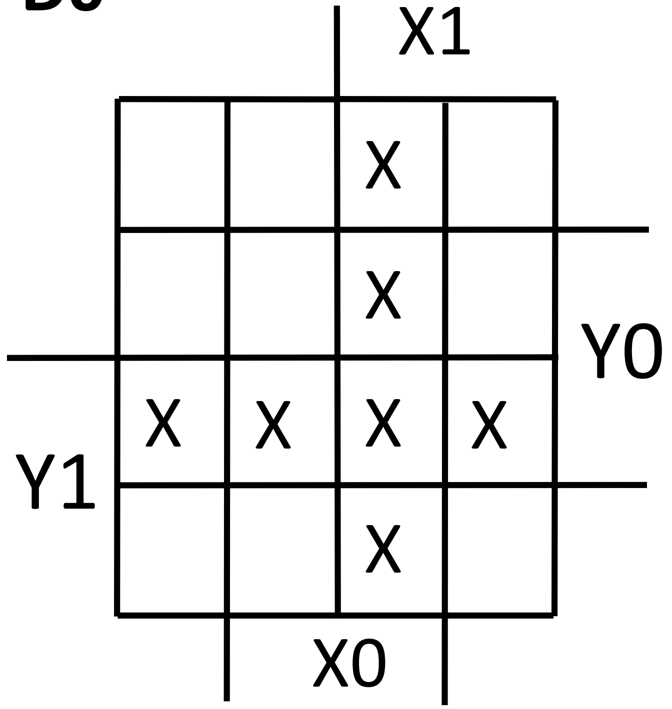
- **Find optimized flip-flop input equations for D flip-flops**

D1



		X	
		X	
X	X	X	X
		X	

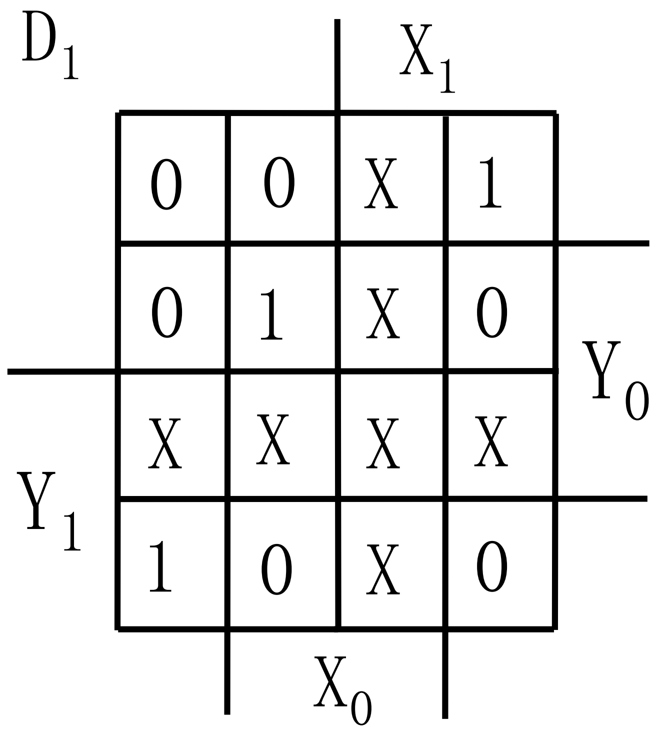
D0

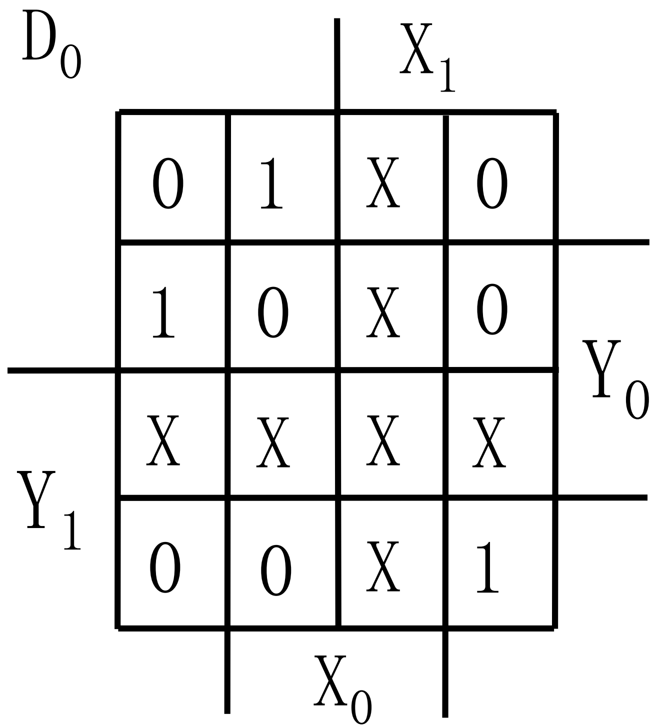


		X	
		X	
X	X	X	X
		X	

- **D_1 =**
- **D_0 =**

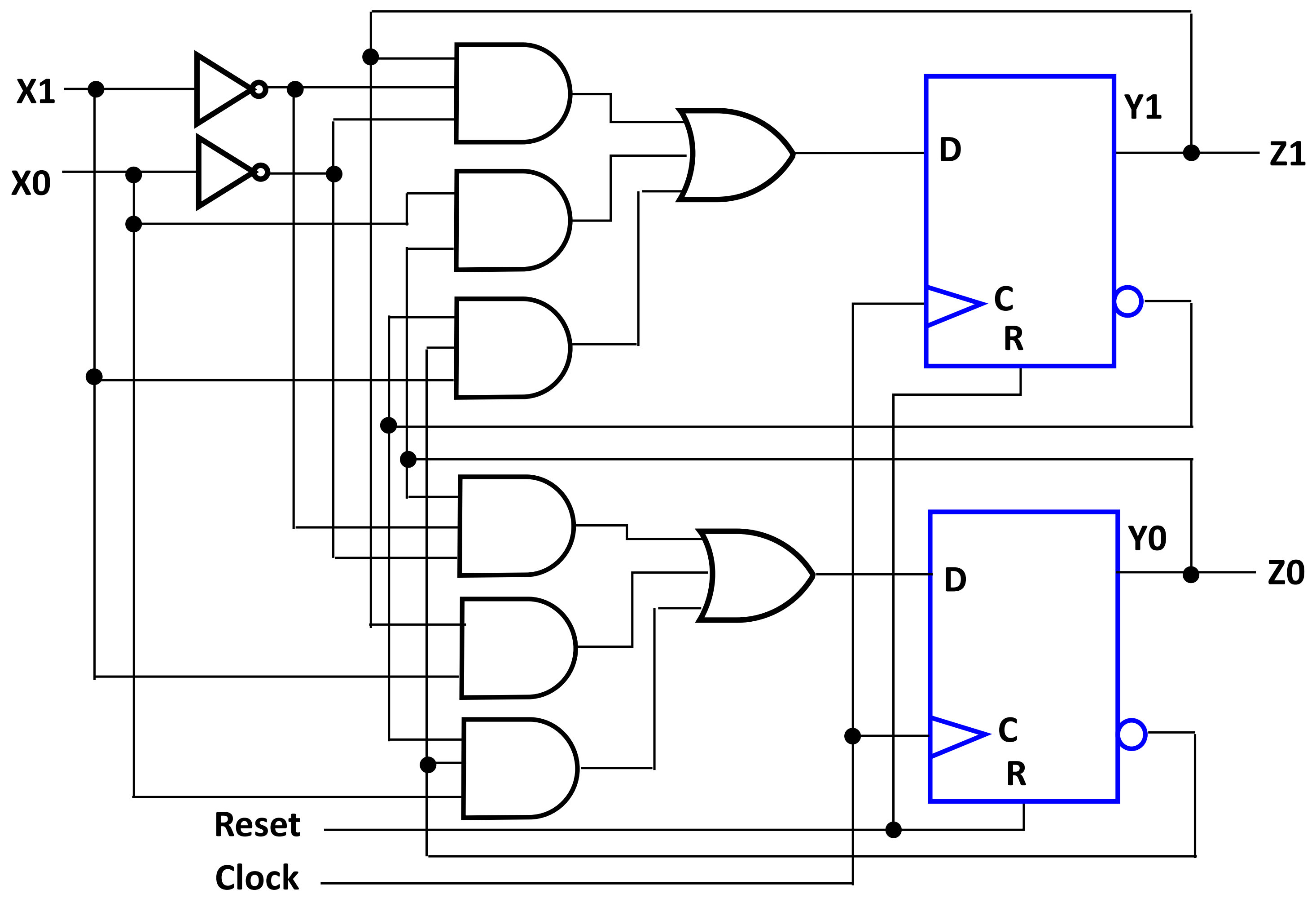
- **Find optimized flip-flop input equations for D flip-flops**





- $\mathbf{D_1 = Y_1\overline{X_1}\,\overline{X_0} + Y_0X_0 + \overline{Y_1}\,\overline{Y_0}X_1}$
- $\mathbf{D_0 = Y_0\overline{X_1}\,\overline{X_0} + Y_1X_1 + \overline{Y_1}\,\overline{Y_0}X_0}$

Circuit - Final Result with AND, OR, NOT



Other Flip-Flop Types

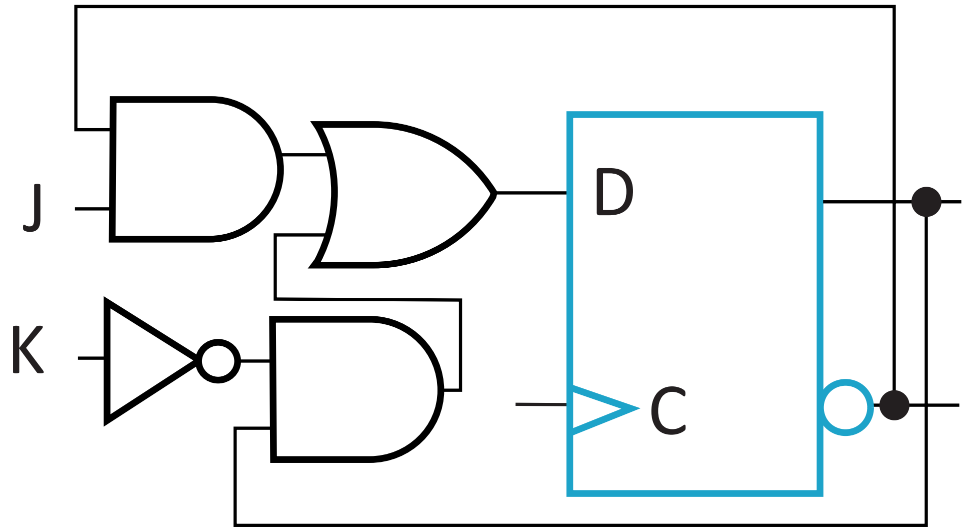
- **J-K and T flip-flops**
  - Behavior
  - Implementation
- **Basic descriptors for understanding and using different flip-flop types**
  - Characteristic tables
  - Characteristic equations
  - Excitation tables
- **For actual use, see Reading Supplement - Design and Analysis Using J-K and T Flip-Flops**

J-K Flip-flop

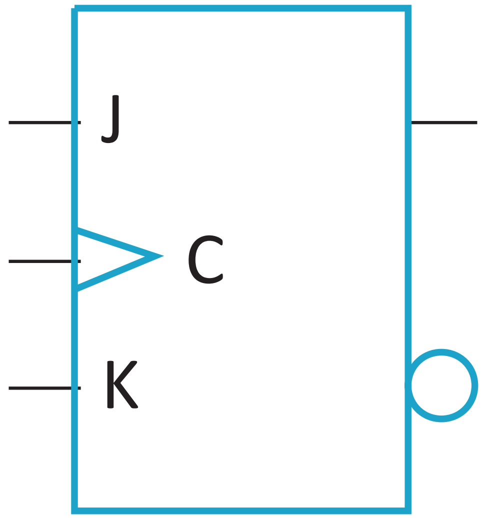
- **Behavior**
  - Same as S-R flip-flop with J analogous to S and K analogous to R
  - Except that J = K = 1 is allowed, and
  - For J = K = 1, the flip-flop changes to the *opposite state*
  - As a master-slave, has same “1s catching” behavior as S-R flip-flop
  - If the master changes to the wrong 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**

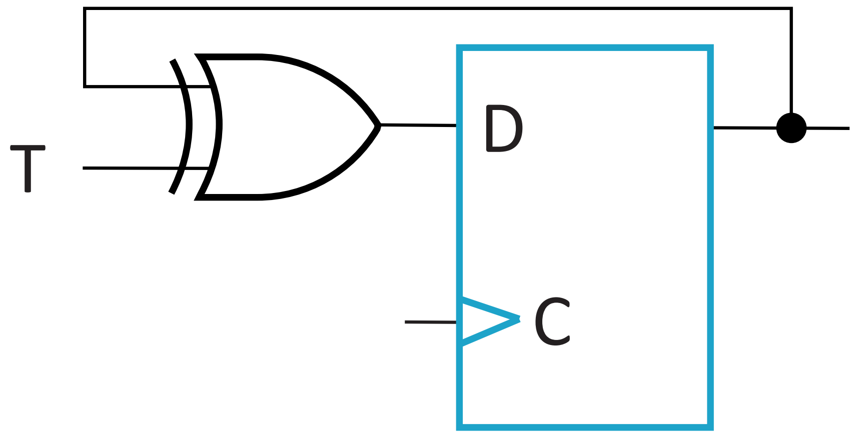


T Flip-flop

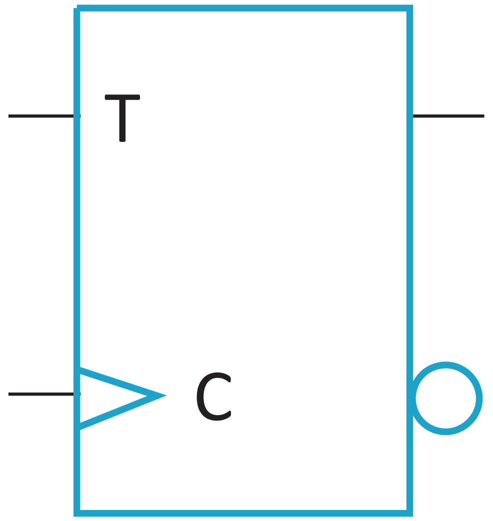
- **Behavior**
  - Has a single input T
    - For T = 0, no change to state
    - For T = 1, changes to opposite state
- **Same as a J-K flip-flop with J = K = T**
- **As a master-slave, has same "1s catching" behavior as J-K flip-flop**
- **Cannot be initialized to a known state using the T input**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

T 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

T Flip-Flop Descriptors

- **Characteristic Table**

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

- **Characteristic Equation**

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

- **Excitation Table**

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

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**

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

- **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

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	$\overline{Q}(t)$	Complement

- **Characteristic Equation**

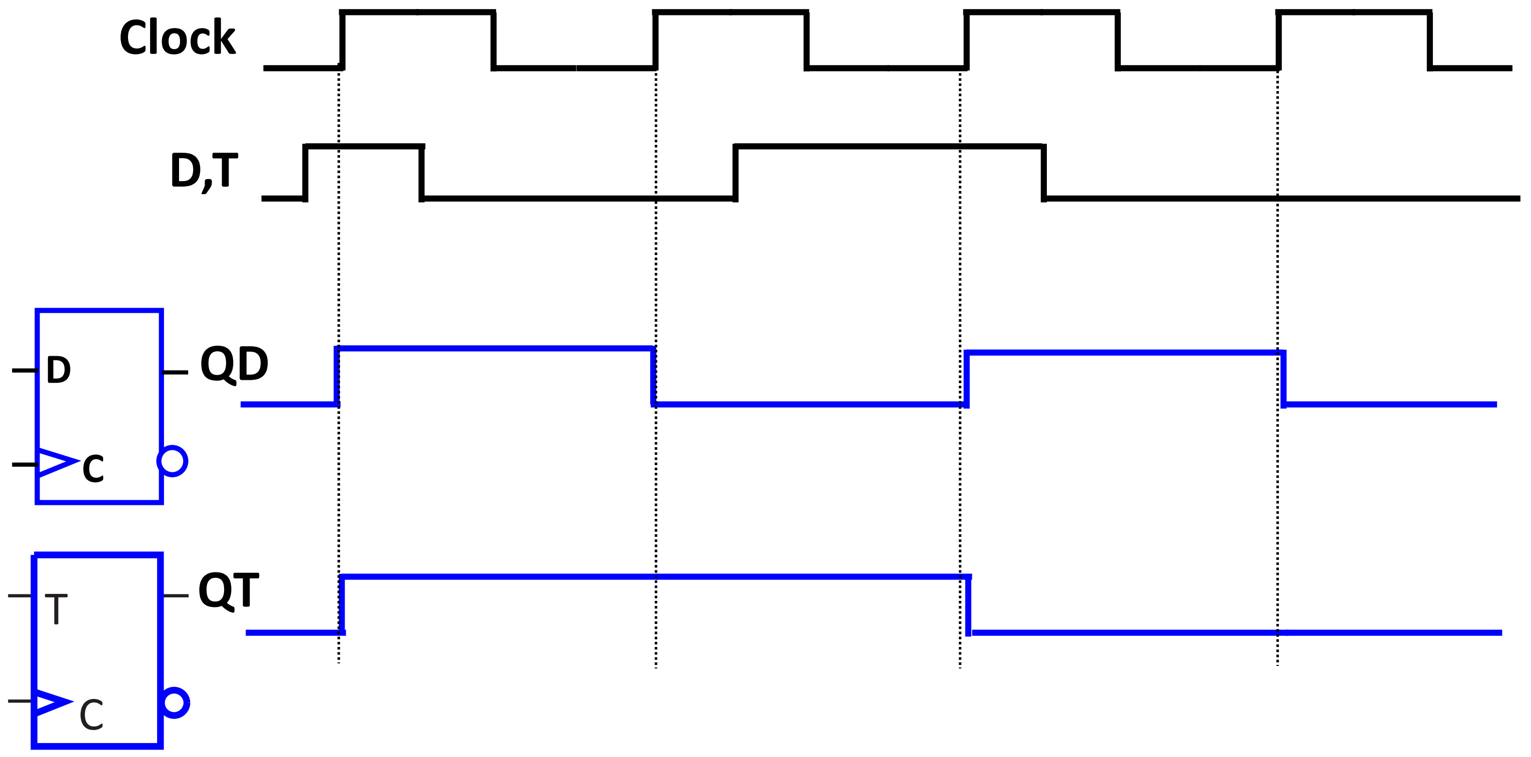
$$Q(t+1) = J\,\overline{Q} + \overline{K}\,Q$$

- **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

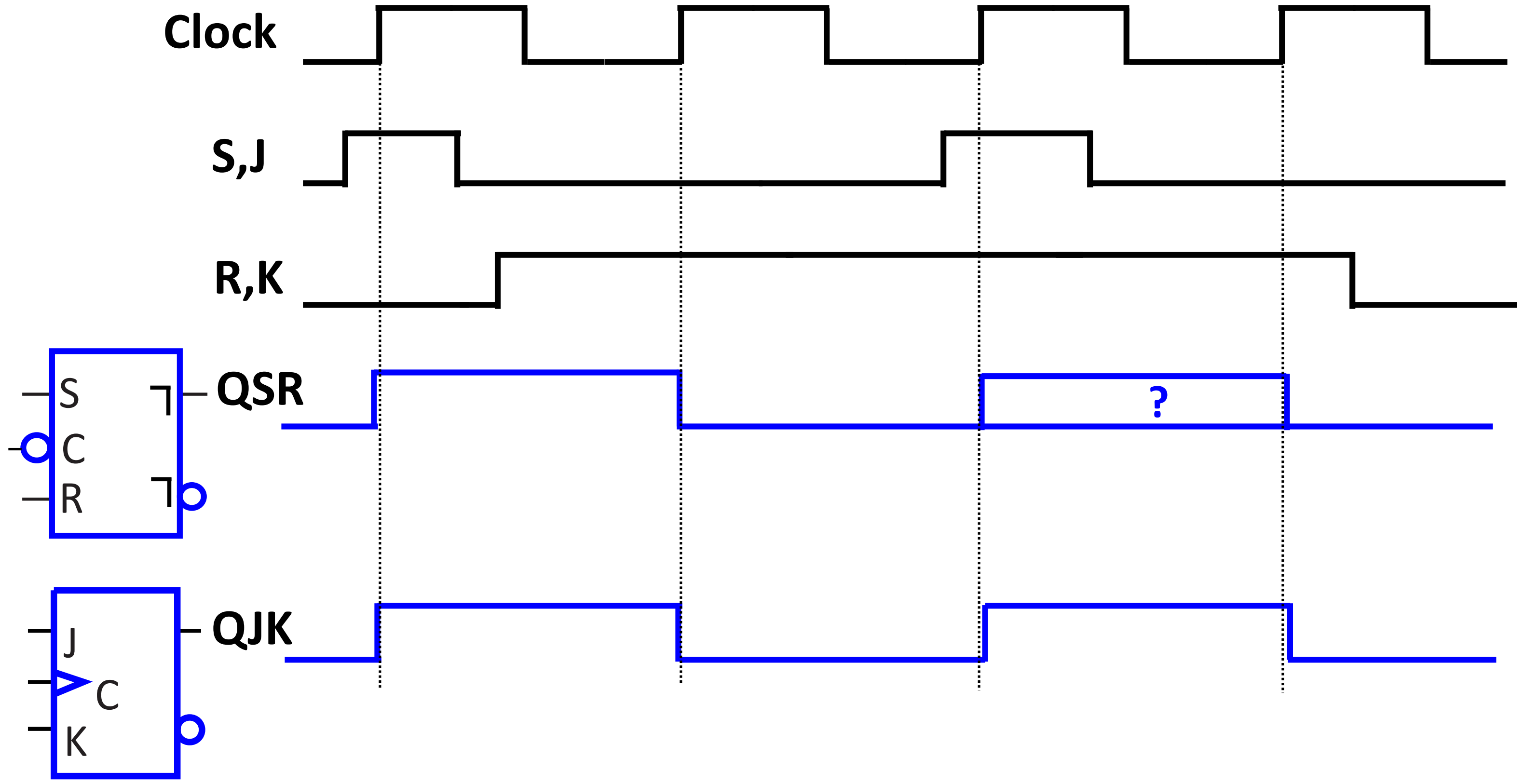
Flip-flop Behavior Example

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



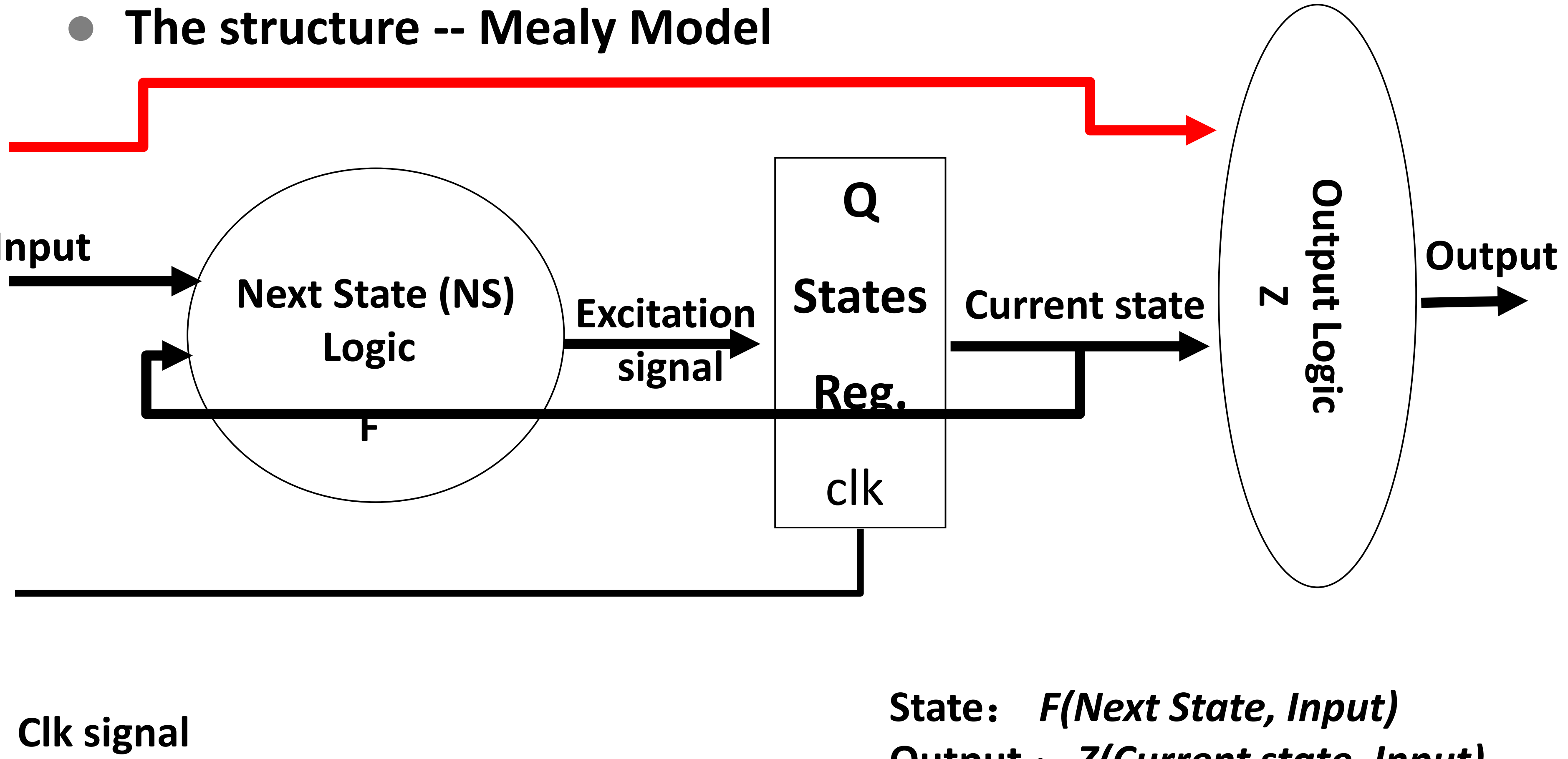
Flip-Flop Behavior Example (continued)

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



State machine description -1

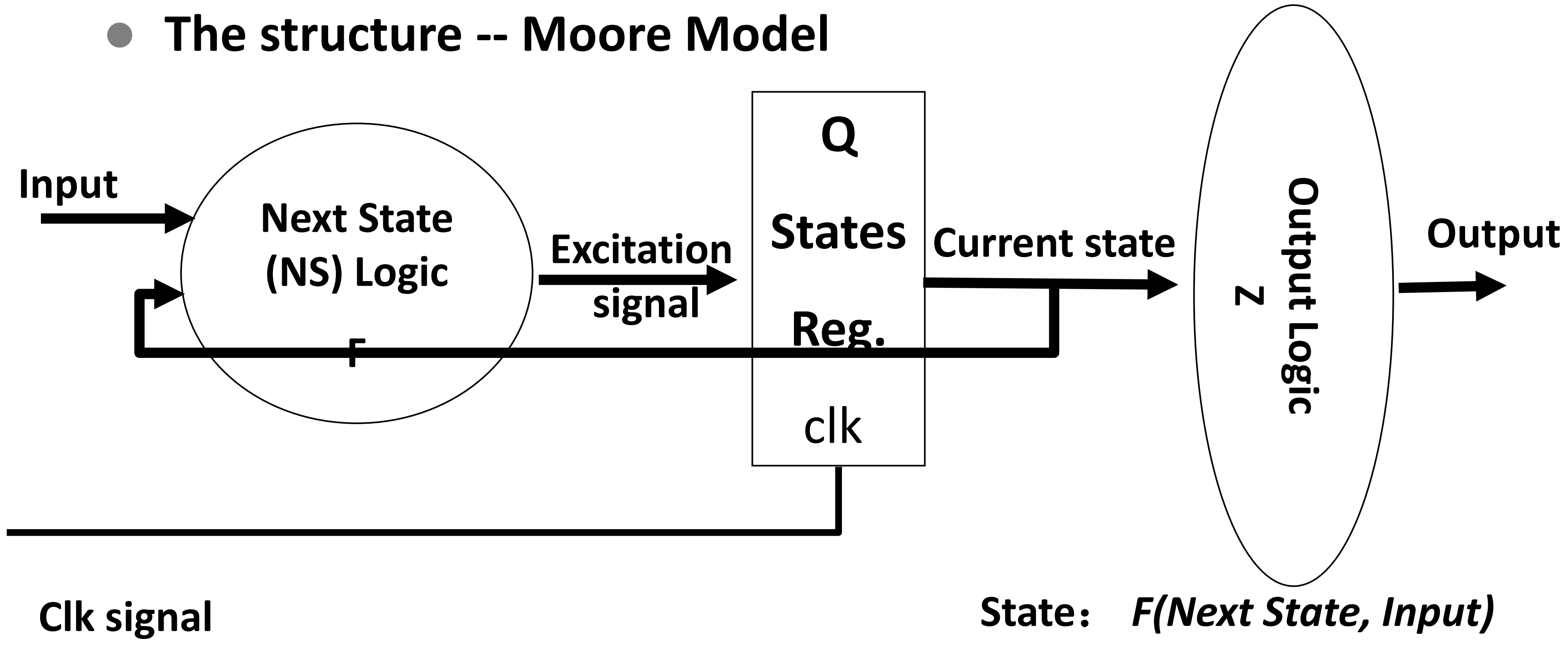
- **The structure -- Mealy Model**



State： ***F(Next State, Input)***
Output： ***Z(Current state, Input)***

State machine description -2

- **The structure -- Moore Model**



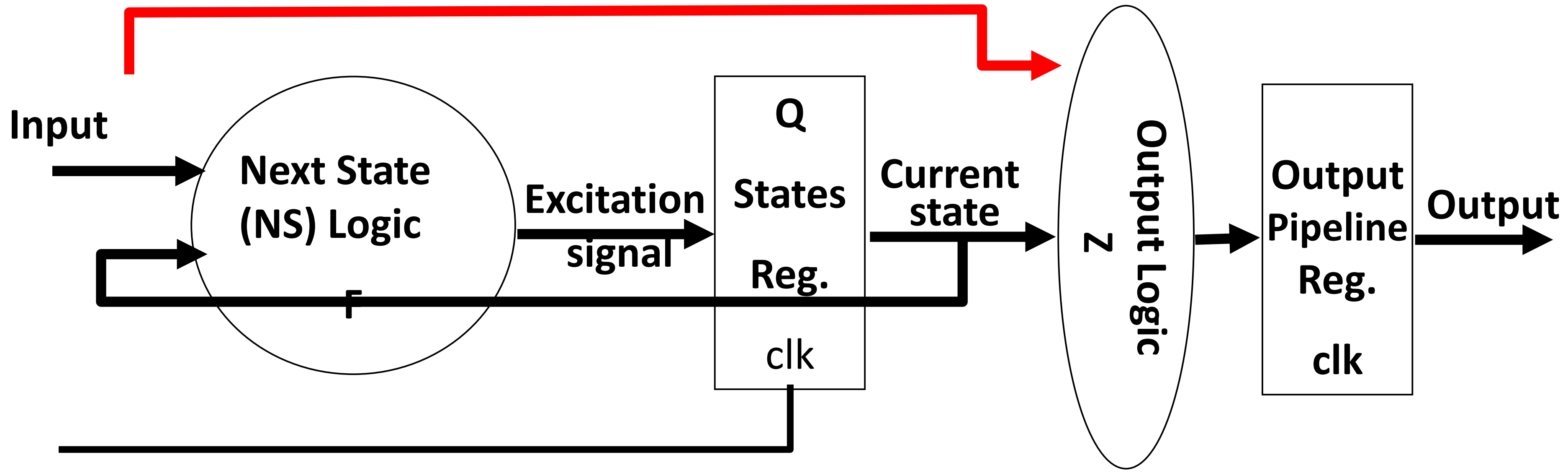
State： ***F(Next State, Input)***
Output： ***Z(Current state)***

In the high-speed circuit design, the states are used directly as the output

- the pipeline
- Output signal is fully synchronized with the clock

State machine description -3

- **The Pineline structure -- Mealy Model**

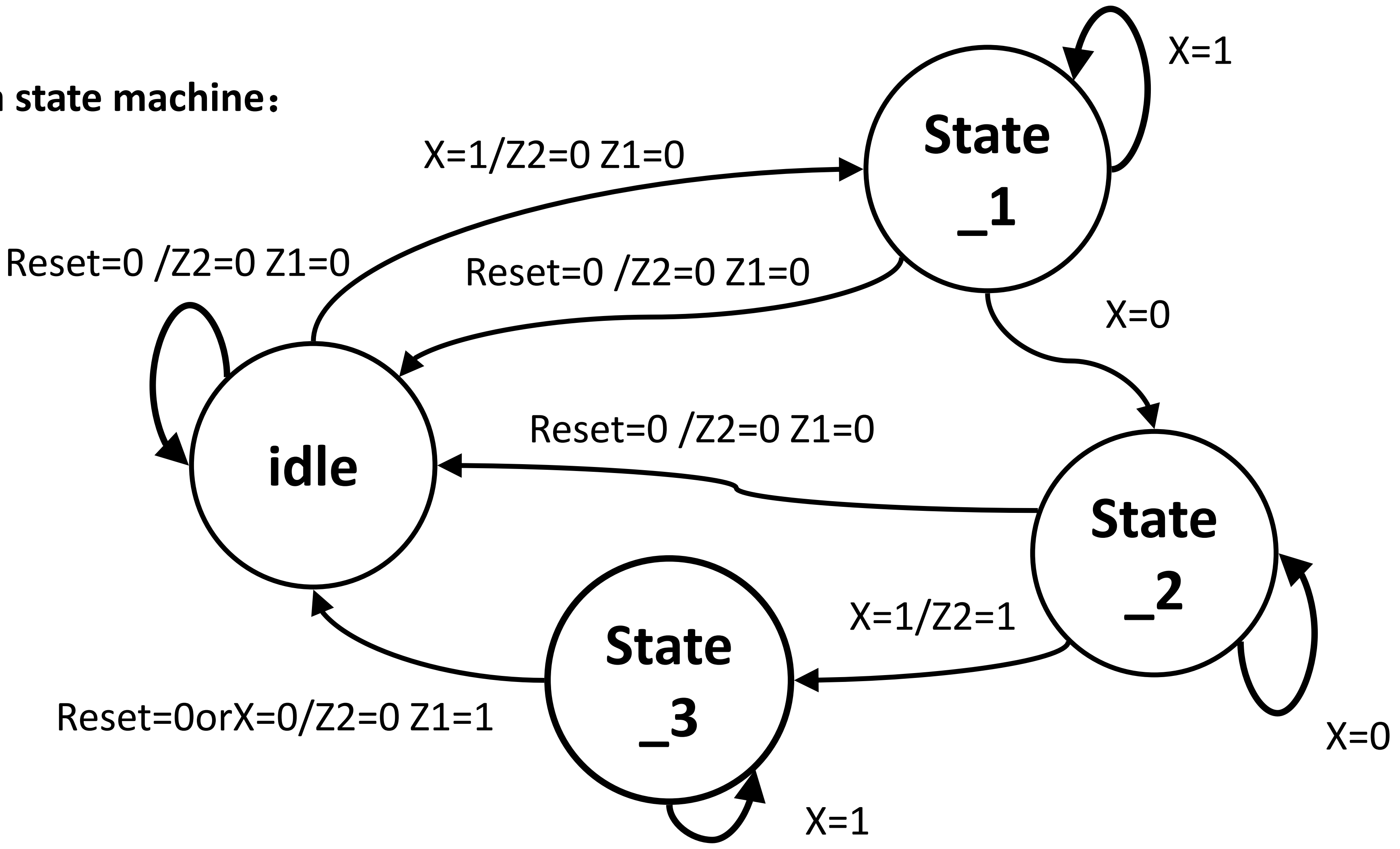


Clk signal

In the design of high-speed circuit, it is often necessary so that the state machine output almost completely synchronized with the clock. There is a way to be used directly as the output of the state variables.

Verilog HDL synthesis state machine

Assume a state machine：



Verilog HDL synthesis state machine 1: Gray 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,  State_1 = 2'b01,
      State_2 = 2'b10,   State_3 = 2'b11;
      always @(posedge Clock)
             if (!Reset)
                begin
                state <= Idle;  Z2<=0; Z1<=0;
                end
             else
                case (state)
                Idle:  begin
                       if (X)  begin
                         state <= State_1;
                         Z1<=0;
                          end
                       else    state <= Idle;
                      end
                State_1: begin
                        if (!X)  state <= State_2;
                       else    state <= State_1;
                       end
                State_2:  begin
                        if (X) begin
                               state <= State_3;
                               Z2<= 1;
                              end
                        else   state <= State_2;
                        end
                State_3: begin
                       if (!X)  begin
                                state <=Idle;
                                Z2<=0;  Z1<=1;
                                end
                       else   state <= State_3;
                       end
                endcase
endmodule
```

Verilog HDL synthesis state machine 2: one-hot 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,  State_1 = 4'b0100,
               State_2 =4'b0010, State_3 =4'b0001;
      always @(posedge clock)
        if (!Reset)
               begin
               state <= Idle;  Z2<=0; Z1<=0;
               end
        else
               case (state)
               Idle:  begin
               if (X)  begin
                       state <= State_1;
                        Z1<=0;
                     end
               else state <= Idle;
               end
          State_1: begin
                 if (!X)  state <= State_2;
                  else  state <= State_1;
                end
          State_2:  begin
          if (X) begin
                 state <= State_3;
                 Z2<= 1;
                end
          else  state <= State_2;
          end
          State_3: begin
          if (!X)  begin
                   state <=Idle;
                   Z2<=0;  Z1<=1;
                  end
          else   state <= State_3;
          end
          default:  state <=Idle;
          endcase
endmodule
```

Verilog HDL synthesis state machine 3:State Code as Output

```
module  fsm (Clock, Reset, X,  Z2, Z1);
      input Clock, Reset, X;
      output Z2, Z1;
      reg [3:0] state ;
      assign Z2= state[3];
      // 把状态变量的最高位用作输出Z2
      assign Z1= state[0];
      // 把状态变量的最低位用作输出Z1
      parameter //   Z2_S2_S1_Z1
      zero     =  4'b0000,
      Idle     =  4'b0001,
      State_1    =  4'b0100,
      State_2     =  4'b0010,
      State_3    =  4'b1000;
      always @(posedge Clock)
       if (!Reset)
            begin
            state <= Zero;
            end
       else
             case (state)
             Idle,Zero:  begin
                    if (X)  begin
                         state <= State_1;
                        end
                    else state <= Idle;
                    end
             State_1: begin
                    if (!X)  state <= State_2;
                     else  state <= State_1;
                  end
             State_2:  begin  if (X) begin
                              state <= State_3;
                             end
                        else  state <= State_2;
                  end
             State_3: begin  if (!X)  begin
                               state <=Idle;
                              end
                         else   state <= State_3;
                   end
             default:  state <=Zero;
             endcase

endmodule
```

Verilog HDL synthesis state machine 4: Multi-output complex state machine

```
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,  State_1  = 2'b01,
         State_2 =  2'b10, State_3 =  2'b11;
        always @(posedge Clock)
        if (!Reset)
             state <= Idle;
        else
         case (state)
          Idle:  begin  if (X)  begin
                          state <= State_1;
                         end
                   else state <= Idle;
               end
          State_1: begin
                      if (!X)  state <= State_2;
                      else  state <= State_1;
               end
          State_2:  begin
                      if (X)  state <= State_3;
                       else  state <= State_2;
               end
          State_3: begin
                      if (!X) state <=Idle;
                      else   state <= State_3;
               end
          default:  state <=2'bxx;
          endcase
      always @(state or Reset) //产生输出Z2模块
           if (!Reset) Z2=0;
           else
           if (state == State_3) Z2=1;
           else Z2= 0;
      always @(state or Reset) //产生输出Z1模块
         if (!Reset) Z1=0;
         else
           if (state == Idle) Z1=1;
            else Z1=0;
endmodule
```

Thank You !