

# CEDT – Digital Logic 2023

Day 6

FSM



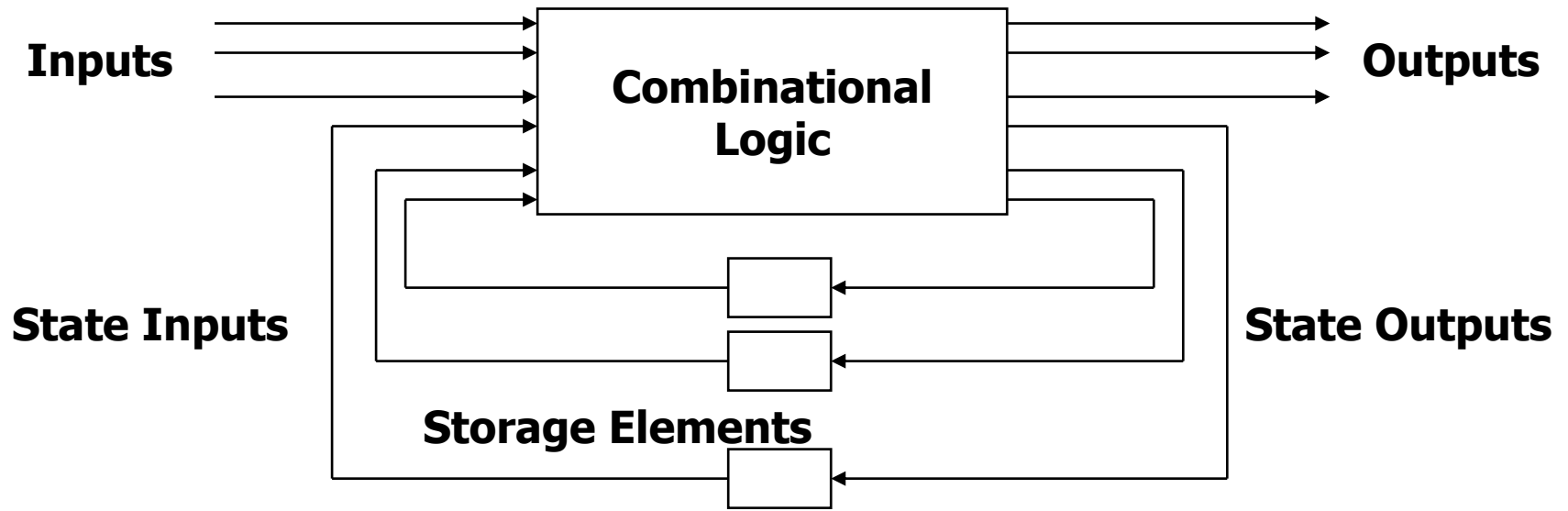
# Finite State Machines

combinational  
clock  
memory

- Sequential circuits
  - primitive sequential elements
  - combinational logic
- Models for representing sequential circuits
  - finite-state machines (Moore and Mealy)
- Basic sequential circuits revisited
  - shift registers
  - counters
- Design procedure
  - state diagrams
  - state transition table
  - next state functions
- Hardware description languages

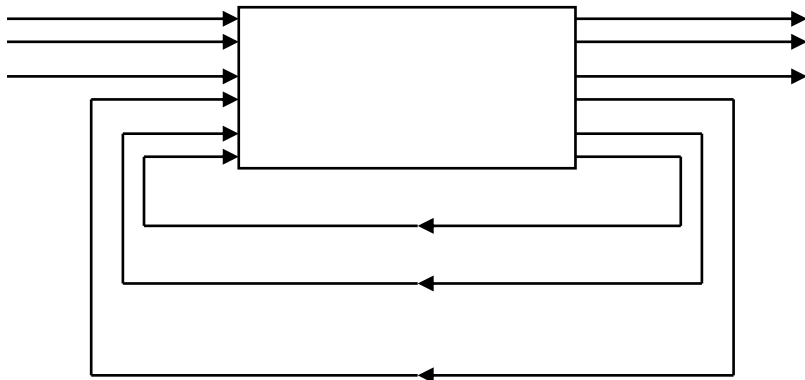
# Abstraction of state elements

- Divide circuit into combinational logic and state
- Localize the feedback loops and make it easy to break cycles
- Implementation of storage elements leads to various forms of sequential logic

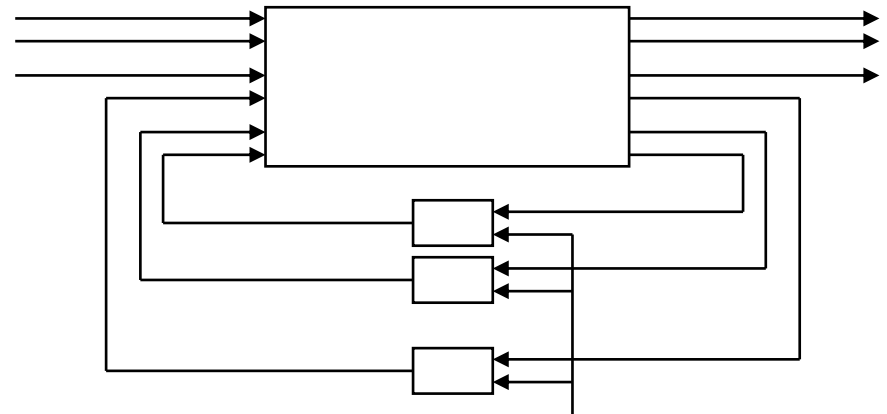


# Forms of sequential logic

- **Asynchronous sequential logic** – state changes occur whenever state inputs change (elements may be simple wires or delay elements)
- **Synchronous sequential logic** – state changes occur in lock step across all storage elements (using a periodic waveform - the clock)



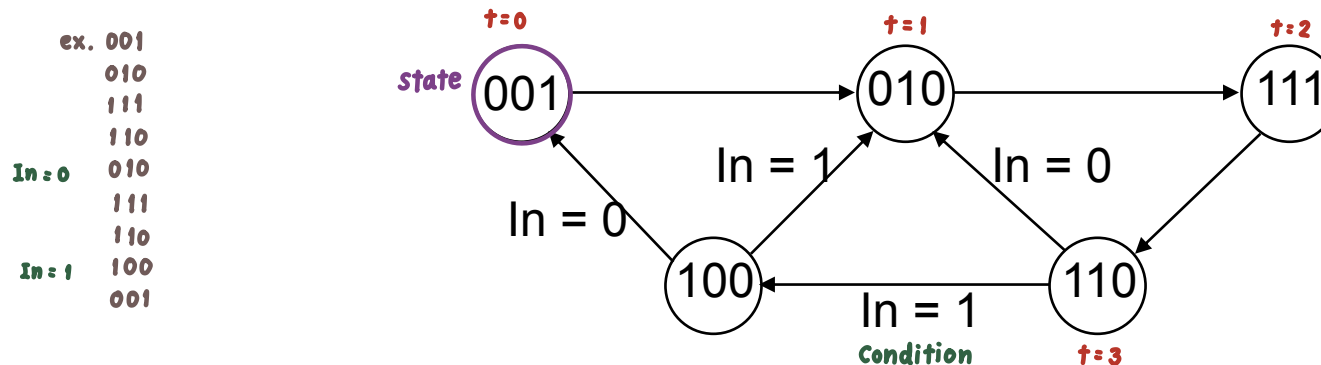
*Asynchronous sequential logic*



*Synchronous sequential logic* **Clock**

# Finite state machine representations ASM chart

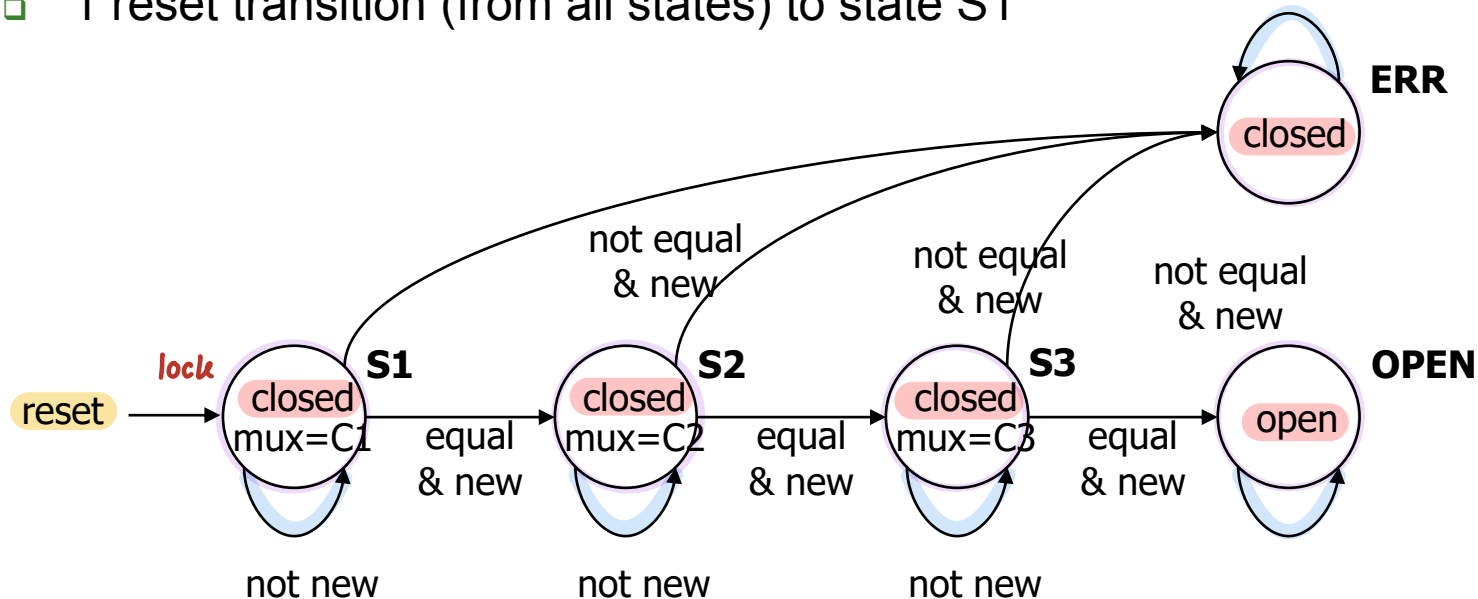
- **States**: determined by possible values in sequential storage elements
- **Transitions**: change of state
- **Clock**: controls when state can change by controlling storage elements
- Sequential logic
  - sequences through a series of states
  - based on sequence of values on input signals
  - clock period defines elements of sequence



# Example finite state machine diagram

- Combination lock from introduction to course

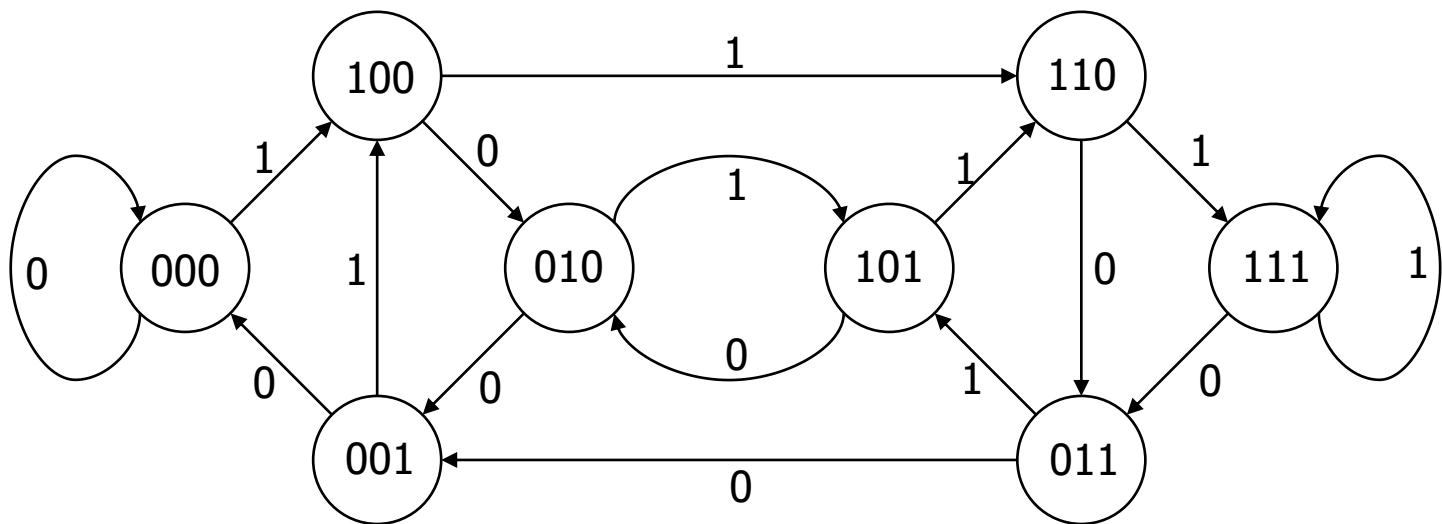
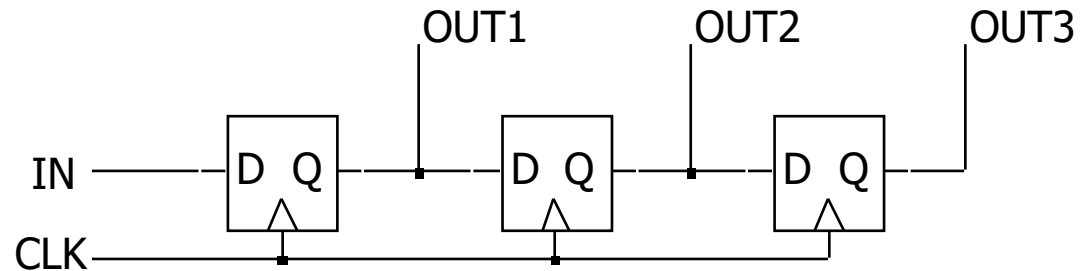
- 5 states
- 5 self-transitions
- 6 other transitions between states
- 1 reset transition (from all states) to state S1



# Can any sequential system be represented with a state diagram?

## ■ Shift register

- input value shown on transition arcs
- output values shown within state node



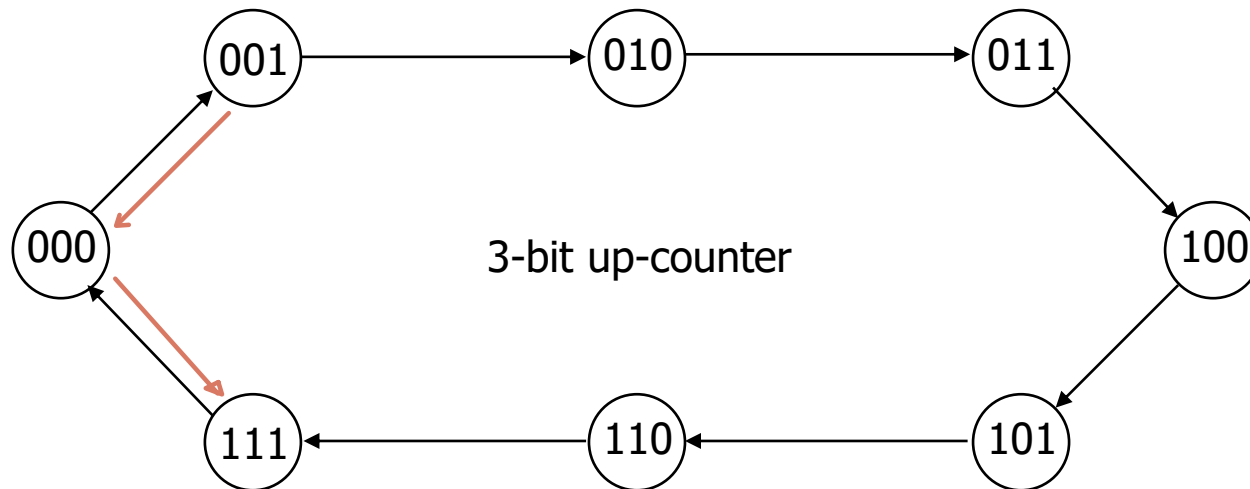
# Counters are simple finite state machines

## ■ Counters

- proceed through well-defined sequence of states in response to enable

## ■ Many types of counters: binary, BCD, Gray-code

- 3-bit up-counter: 000, 001, 010, 011, 100, 101, 110, 111, 000, ...
- 3-bit down-counter: 111, 110, 101, 100, 011, 010, 001, 000, 111, ...

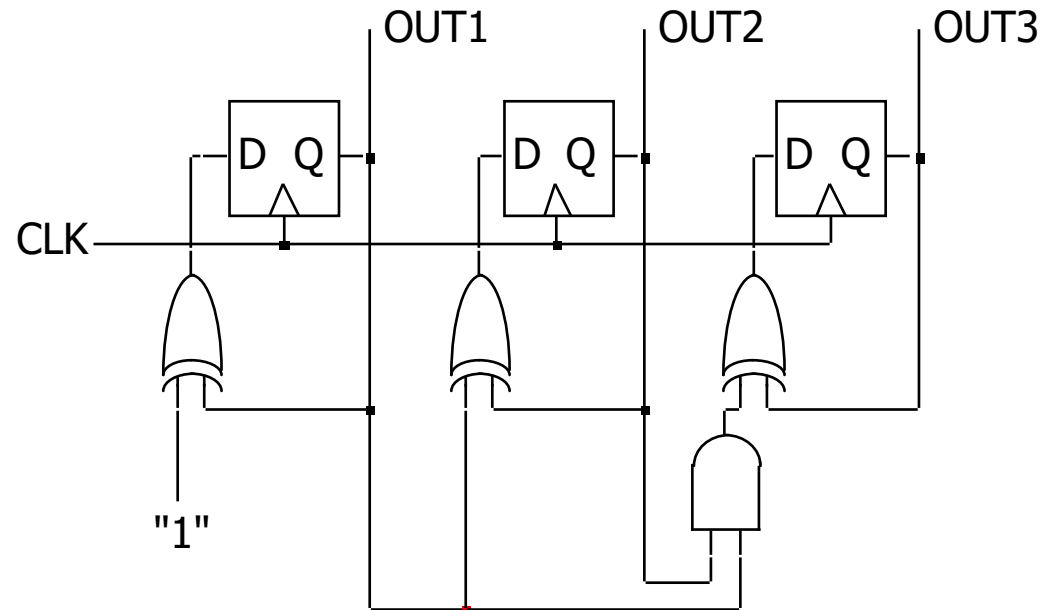




# How do we turn a state diagram into logic?

## ■ Counter

- 3 flip-flops to hold state
- logic to compute next state
- clock signal controls when flip-flop memory can change
  - wait long enough for combinational logic to compute new value
  - don't wait too long as that is low performance

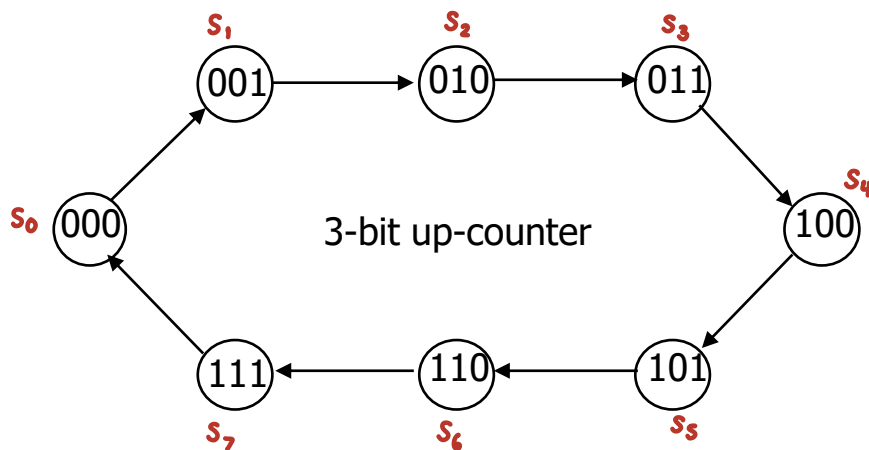


# FSM design procedure

- Start with counters
  - simple because output is just state
  - simple because no choice of next state based on input
  
- State diagram to state transition table
  - tabular form of state diagram
  - like a truth-table
- State encoding
  - decide on representation of states
  - for counters it is simple: just its value
- Implementation
  - flip-flop for each state bit
  - combinational logic based on encoding

# FSM design procedure: state diagram to encoded state transition table

- Tabular form of state diagram
- Like a truth-table (specify output for all input combinations)
- Encoding of states: easy for counters – just use value



present state		next state	
0	000	001	1
1	001	010	2
2	010	011	3
3	011	100	4
4	100	101	5
5	101	110	6
6	110	111	7
7	111	000	0

# Implementation

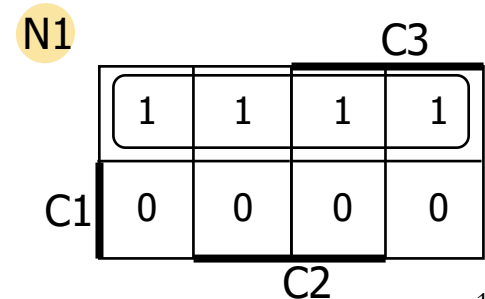
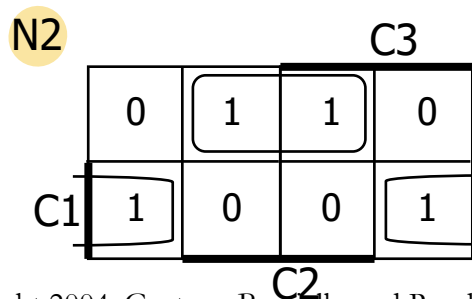
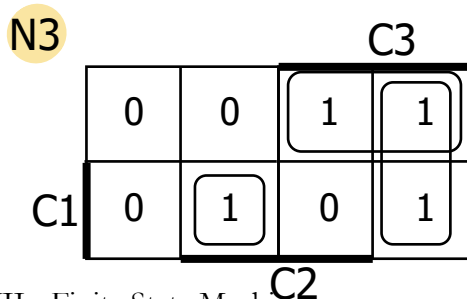
- D flip-flop for each state bit
- Combinational logic based on encoding

Transition Table

present stage			next stage		
C3	C2	C1	N3	N2	N1
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0

Verilog notation to show function represents an input to D-FF

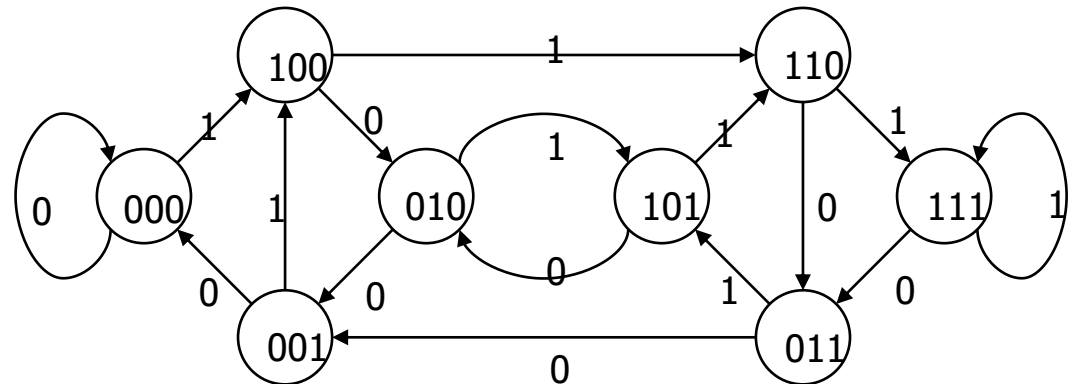
$N1 \leq C1'$   
 $N2 \leq C1C2' + C1'C2$   
 $\leq C1 \text{ xor } C2$   
 $N3 \leq C1C2C3' + C1'C3 + C2'C3$   
 $\leq (C1C2)C3' + (C1' + C2')C3$   
 $\leq (C1C2)C3' + (C1C2)'C3$   
 $\leq (C1C2) \text{ xor } C3$



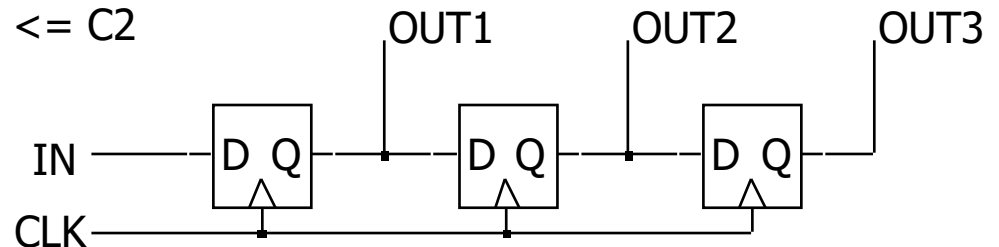
# Back to the shift register

- Input determines next state

In	C1	C2	C3	N1	N2	N3
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	0	0	1
0	0	1	1	0	0	1
0	1	0	0	0	1	0
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	0	1	1
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	1	0	1
1	0	1	1	1	0	1
1	1	0	0	1	1	0
1	1	0	1	1	1	0
1	1	1	0	1	1	1
1	1	1	1	1	1	1

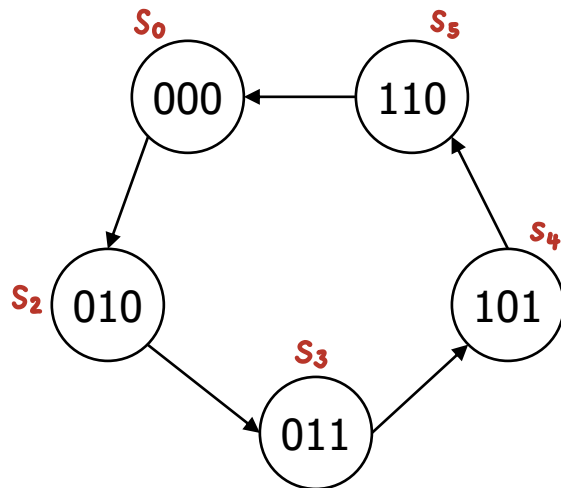


N1 <= In  
N2 <= C1  
N3 <= C2



# More complex counter example

- Complex counter
  - repeats 5 states in sequence
  - not a binary number representation
- Step 1: derive the state transition diagram
  - count sequence: 000, 010, 011, 101, 110
- Step 2: derive the state transition table from the state transition diagram



Present State			Next State		
C	B	A	C+	B+	A+
0	0	0	0	1	0
0	0	1	—	—	—
0	1	0	0	1	1
0	1	1	1	0	1
1	0	0	—	—	—
1	0	1	1	1	0
1	1	0	0	0	0
1	1	1	—	—	—

note the don't care conditions that arise from the unused state codes

# More complex counter example (cont'd) *lab7*

## ■ Step 3: K-maps for next state functions

C+		C	
		0	X
A		X	1
	B		

B+		C	
		0	X
A		X	1
	B		

A+		C	
		0	X
A		X	0
	B		

$$C+ \leq A$$

$$B+ \leq B' + A'C'$$

$$A+ \leq BC'$$

# Self-starting counters (cont'd)

- Re-deriving state transition table from don't care assignment

C+

			C
	0	0	0
	0	0	0
A	1	1	1
			B

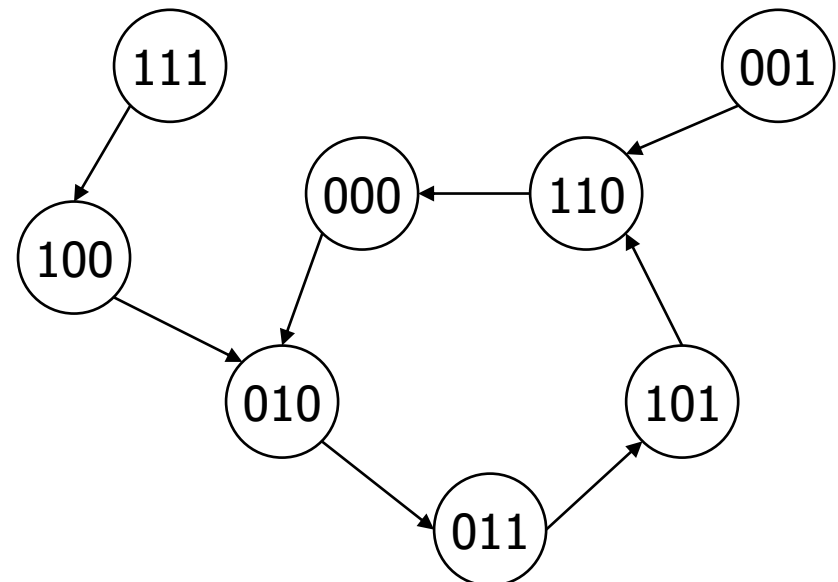
B+

			C
	1	1	0
	1	1	1
A	1	0	0
			B

A+

			C
	0	1	0
	0	1	0
A	0	1	0
			B

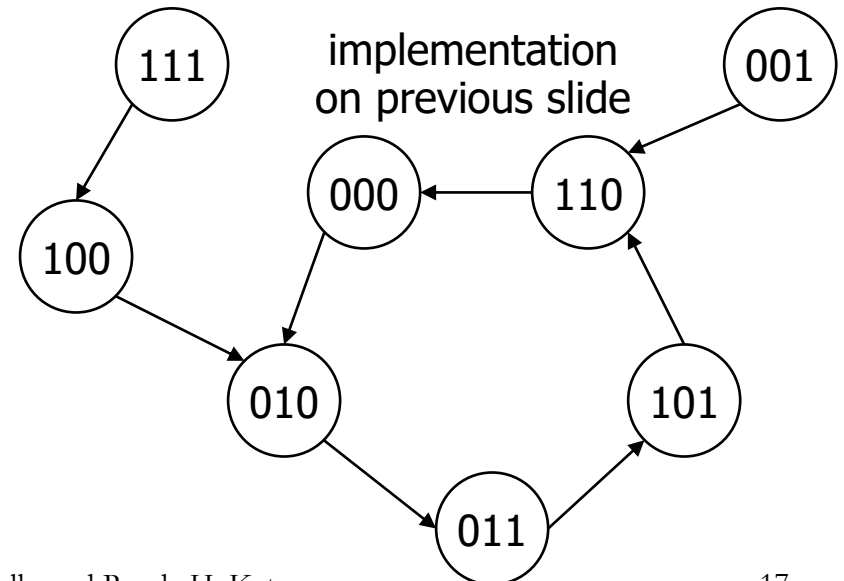
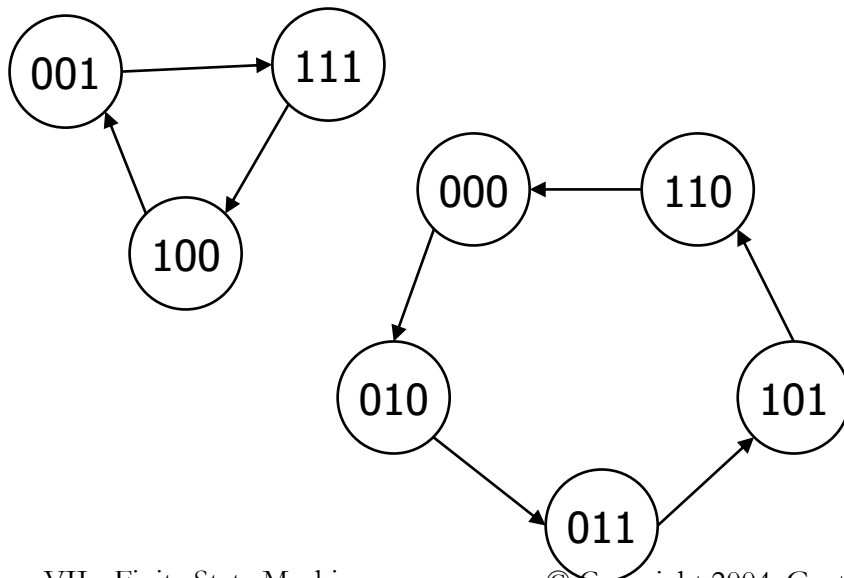
Present State			Next State		
C	B	A	C+	B+	A+
0	0	0	0	1	0
0	0	1	1	1	0
0	1	0	0	1	1
0	1	1	1	0	1
1	0	0	0	1	0
1	0	1	1	1	0
1	1	0	0	0	0
1	1	1	1	0	0





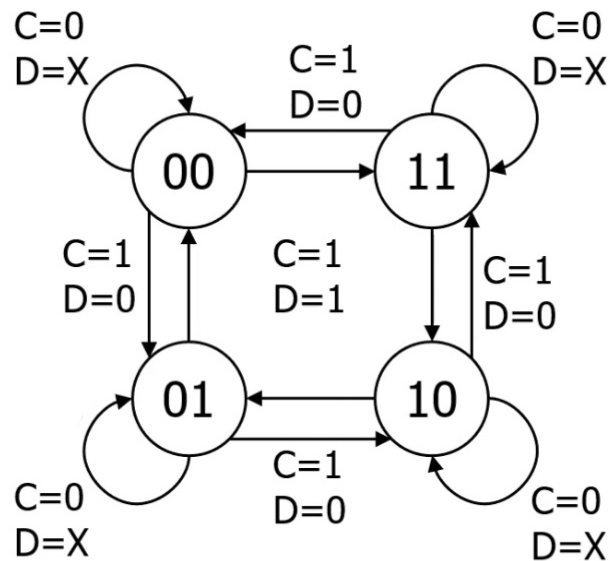
# Self-starting counters

- Start-up states
  - ❑ at power-up, counter may be in an unused or invalid state
  - ❑ designer must guarantee that it (eventually) enters a valid state
- Self-starting solution
  - ❑ design counter so that invalid states eventually transition to a valid state
  - ❑ may limit exploitation of don't cares



# Activity

- 2-bit up-down counter (2 inputs)
  - direction:  $D = 0$  for up,  $D = 1$  for down
  - count:  $C = 0$  for hold,  $C = 1$  for count



S1	S0	C	D	N1	N0
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	1	1
0	1	0	0	0	1
0	1	0	1	0	1
0	1	1	0	1	0
0	1	1	1	0	0
1	0	0	0	1	0
1	0	0	1	1	0
1	0	1	0	1	1
1	0	1	1	0	1
1	1	0	0	1	1
1	1	0	1	1	1
1	1	1	0	0	0
1	1	1	1	1	0

# Activity (cont'd)

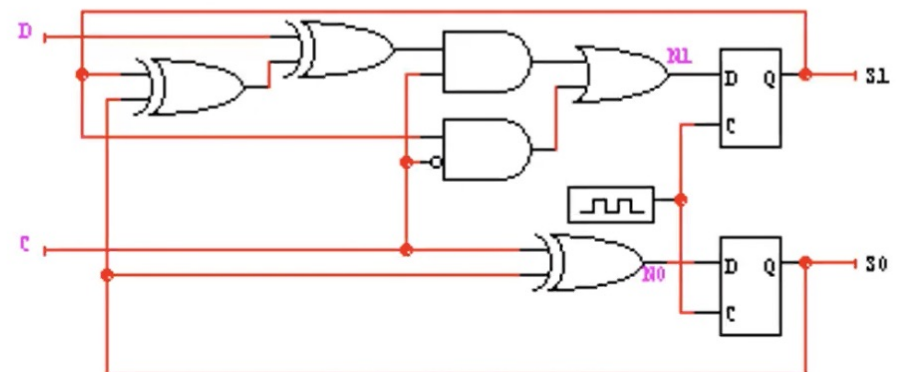
S1	S0	C	D	N1	N0
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	1	1
0	1	0	0	0	1
0	1	0	1	0	1
0	1	1	0	1	0
0	1	1	1	0	0
1	0	0	0	1	0
1	0	0	1	1	0
1	0	1	0	1	1
1	0	1	1	0	1
1	1	0	0	1	1
1	1	0	1	1	1
1	1	1	0	0	0
1	1	1	1	1	0

		S1			
		0	0	1	1
		0	0	1	1
C		1	0	1	0
		0	1	0	1
		S0			

		S1			
		0	1	1	0
		0	1	1	0
C		1	0	0	1
		1	0	0	1
		S0			

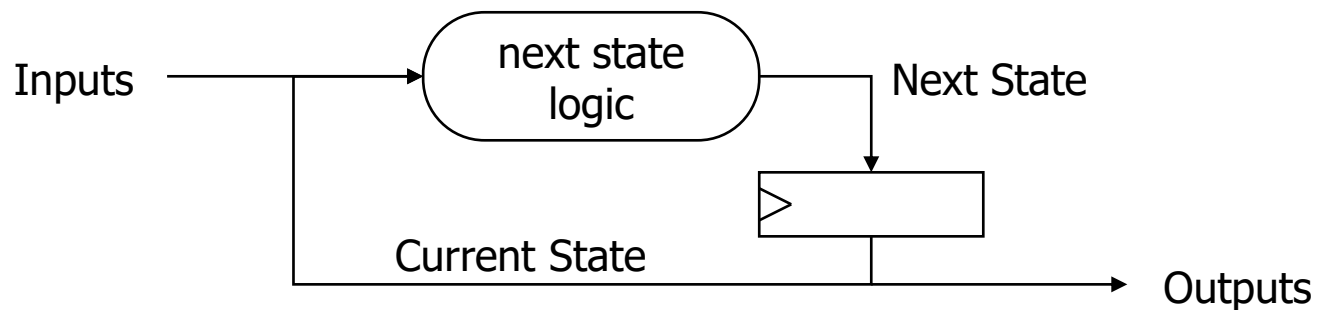
$$\begin{aligned}
 N1 &= C'S1 \\
 &+ CDS0'S1' + CDS0S1 \\
 &+ CD'S0S1' + CD'S0'S1 \\
 &= C'S1 \\
 &+ C(D'(S1 \oplus S0) + D(S1 \equiv S0))
 \end{aligned}$$

$$N0 = CS0' + C'S0$$



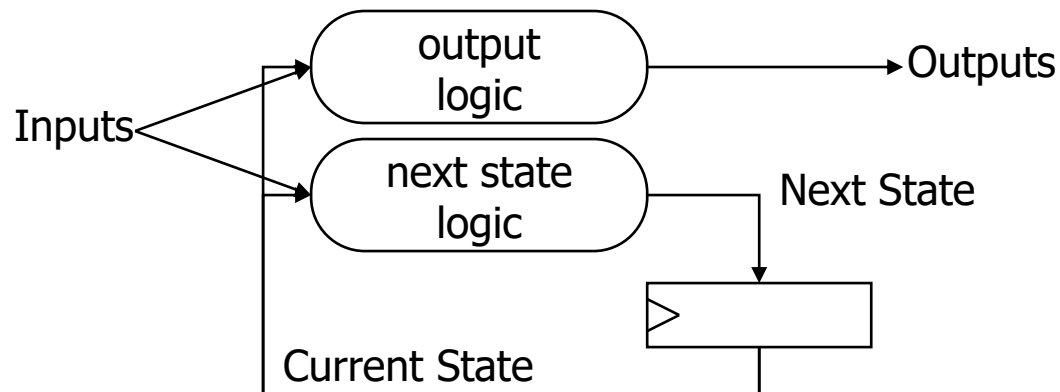
# Counter/shift-register model

- Values stored in registers represent the state of the circuit
- Combinational logic computes:
  - next state
    - function of current state and inputs
  - outputs
    - values of flip-flops



# General state machine model

- Values stored in registers represent the state of the circuit
- Combinational logic computes:
  - next state
    - function of current state and inputs
  - outputs
    - function of current state and inputs (Mealy machine)
    - function of current state only (Moore machine)

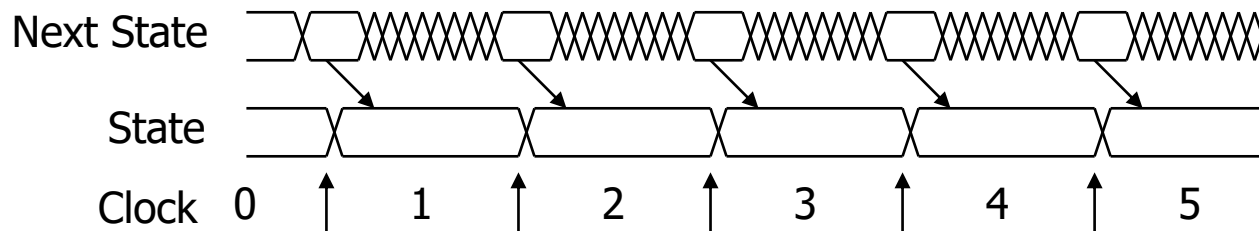
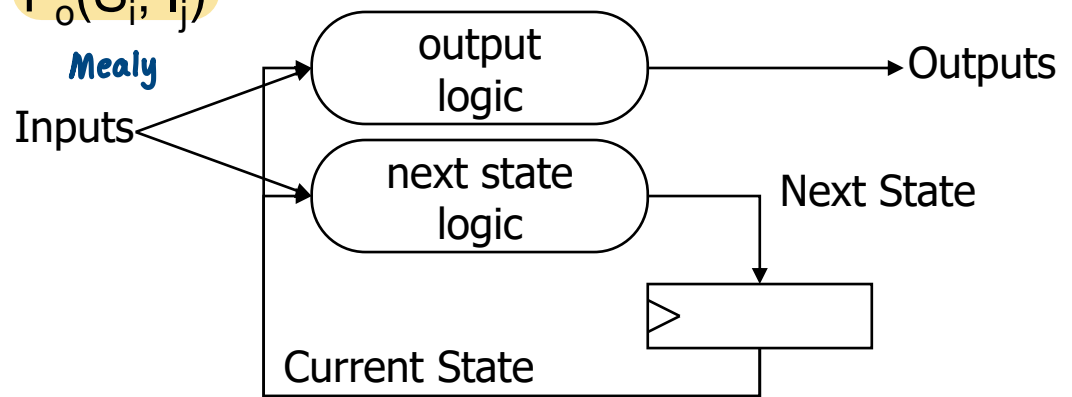


# State machine model (cont'd)

- States:  $S_1, S_2, \dots, S_k$
- Inputs:  $I_1, I_2, \dots, I_m$
- Outputs:  $O_1, O_2, \dots, O_n$
- Transition function:  $F_s(S_i, I_j)$  *Mealy*
- Output function:  $F_o(S_i)$  or  $F_o(S_i, I_j)$

*Moore*

*Mealy*



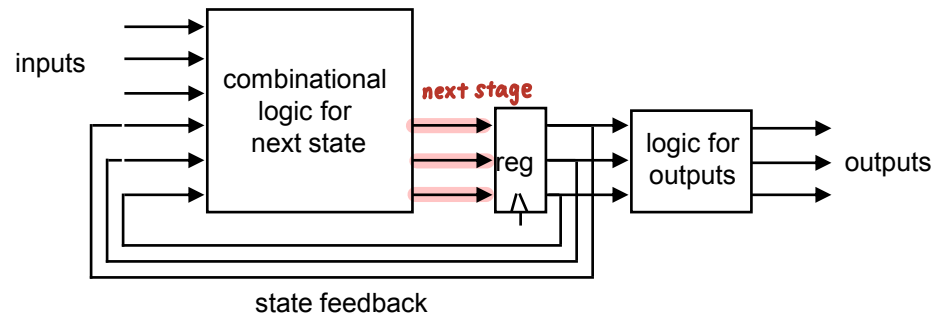
# Comparison of Mealy and Moore machines

- Mealy machines tend to have less states
  - ❑ different outputs on arcs ( $n^2$ ) rather than states ( $n$ )
- Moore machines are safer to use
  - ❑ outputs change at clock edge (always one cycle later)
  - ❑ in Mealy machines, input change can cause output change as soon as logic is done – a big problem when two machines are interconnected – asynchronous feedback may occur if one isn't careful
- Mealy machines react faster to inputs
  - ❑ react in same cycle – don't need to wait for clock
  - ❑ in Moore machines, more logic may be necessary to decode state into outputs – more gate delays after clock edge

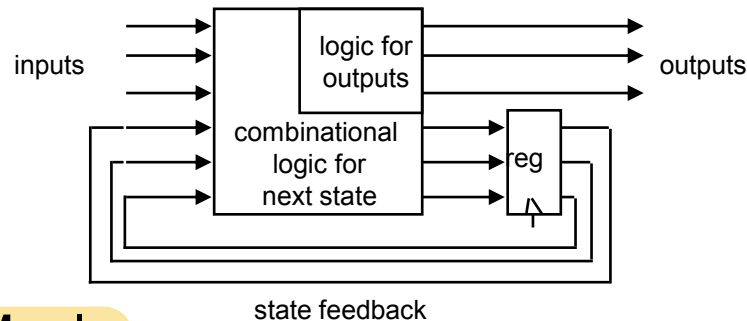
# Comparison of Mealy and Moore machines (cont'd)

## ■ Moore

\* ใช้งานมากกว่า



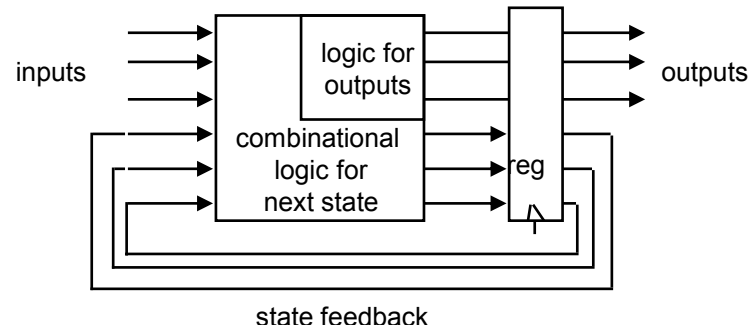
## ■ Mealy



## ■ Synchronous Mealy

output → D flip-flops

Delay 1 step, Design ยากกว่า

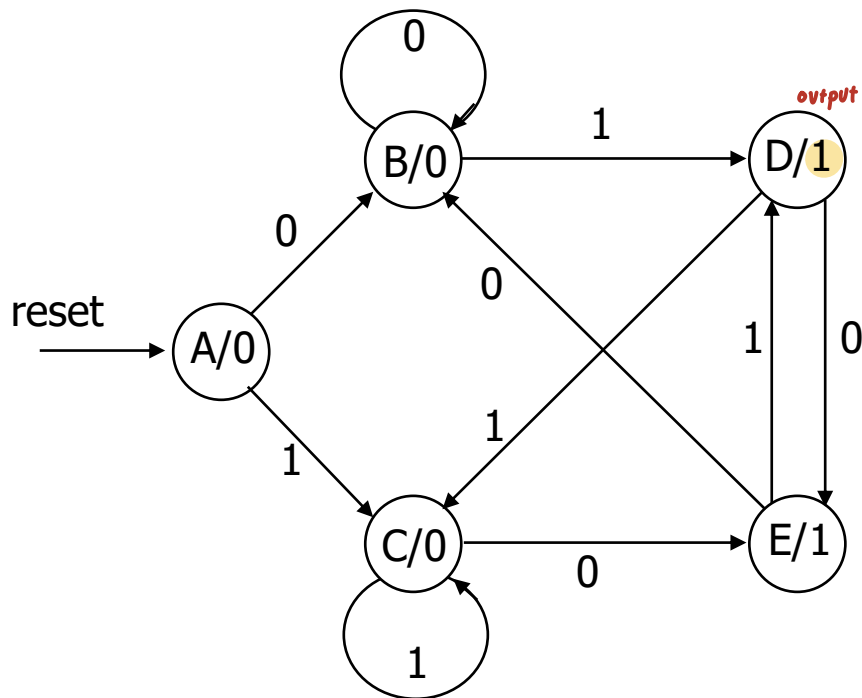




# Specifying outputs for a Moore machine

0 1 0 0  
A B D E  
0 1 1

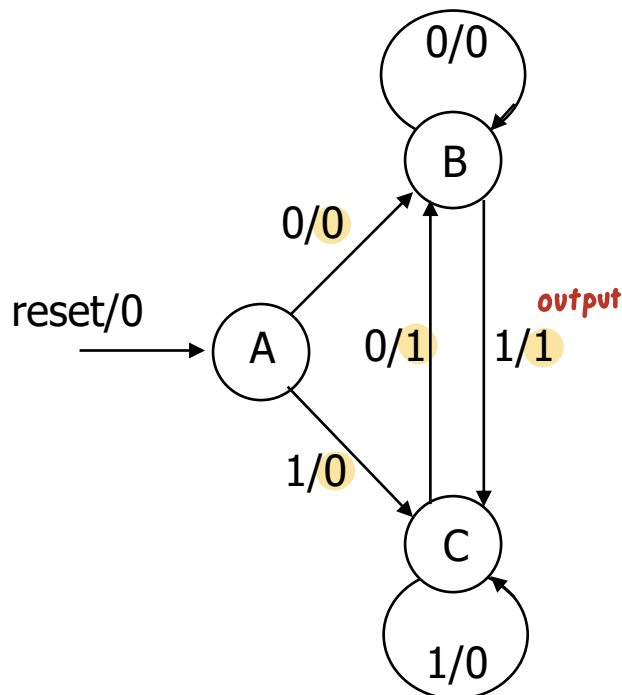
- Output is only function of state
  - specify in state bubble in state diagram
  - example: sequence detector for 01 or 10



reset	input	current state	next state	output
1	—	—	A	
0	0	A	B	0
0	1	A	C	0
0	0	B	B	0
0	1	B	D	0
0	0	C	E	0
0	1	C	C	0
0	0	D	E	1
0	1	D	C	1
0	0	E	B	1
0	1	E	D	1

# Specifying outputs for a Mealy machine

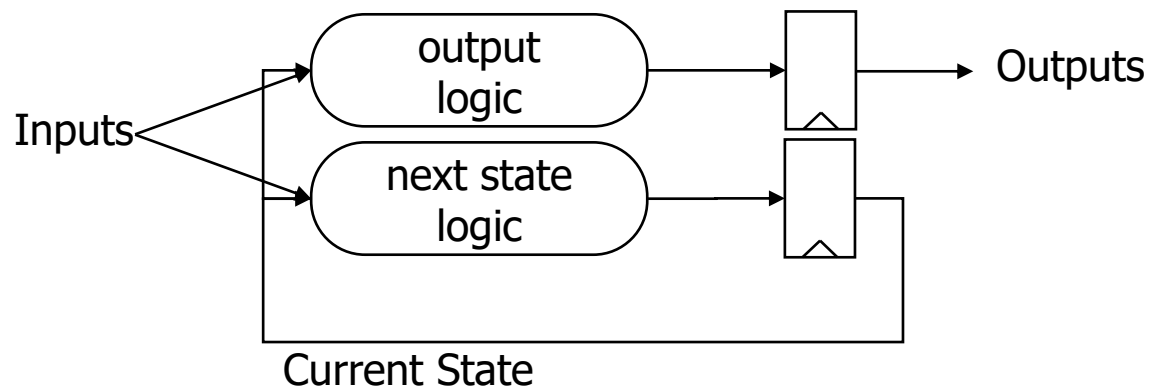
- Output is function of state and inputs
  - specify output on transition arc between states
  - example: sequence detector for 01 or 10



reset	input	current state	next state	output
1	—	—	A	0
0	0	A	B	0
0	1	A	C	0
0	0	B	B	0
0	1	B	C	1
0	0	C	B	1
0	1	C	C	0

# Registered Mealy machine (really Moore)

- Synchronous (or registered) Mealy machine
  - registered state AND outputs
  - avoids 'glitchy' outputs
  - easy to implement in PLDs
- Moore machine with no output decoding
  - outputs computed on transition to next state rather than after entering
  - view outputs as expanded state vector

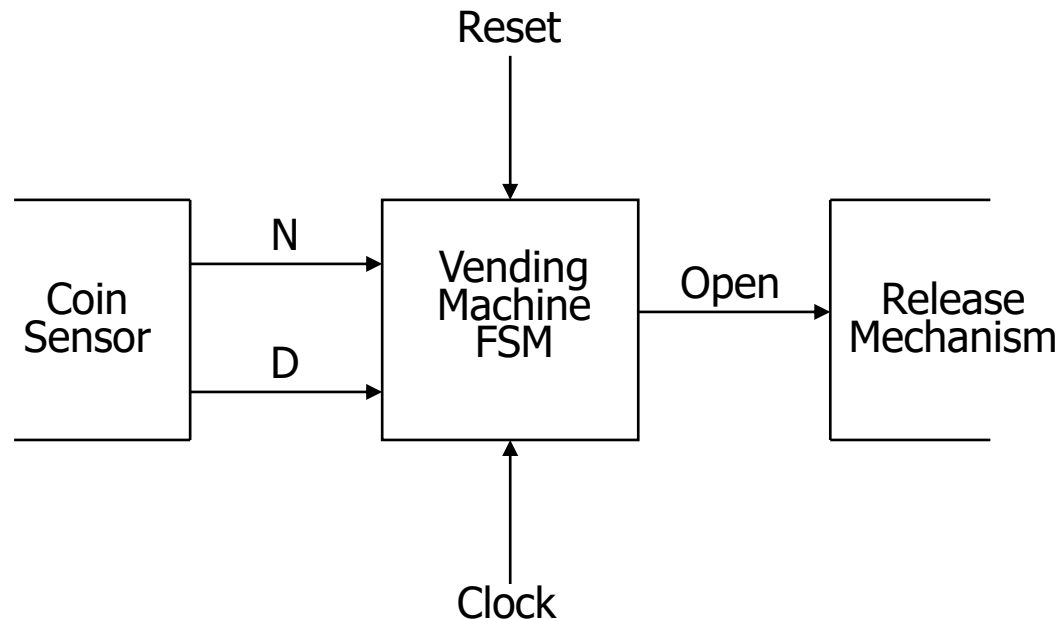


# Example: vending machine

- Release item after 15 cents are deposited
- Single coin slot for dimes, nickels
- No change

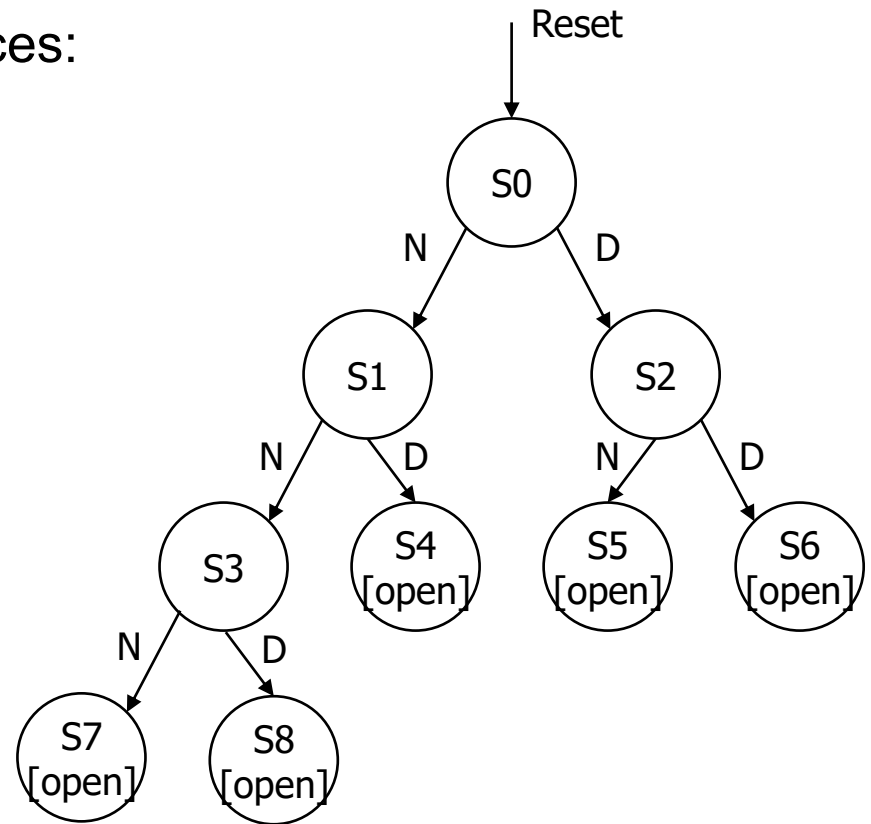
10 ¢

5 ¢



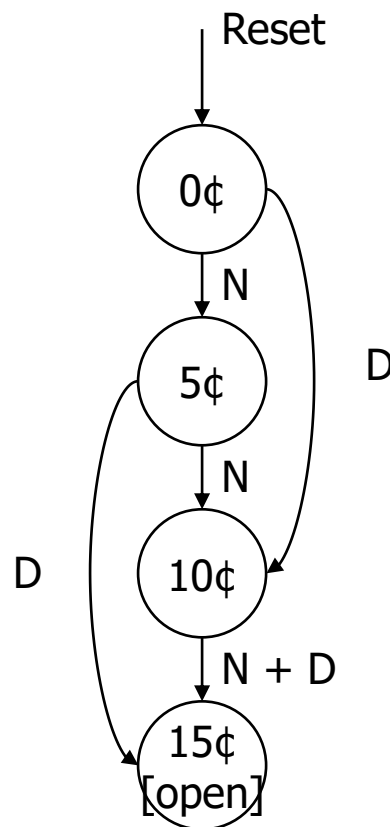
# Example: vending machine (cont'd)

- Suitable abstract representation
  - tabulate typical input sequences:
    - 3 nickels  $5+5+5$
    - nickel, dime  $5+10$
    - dime, nickel  $10+5$
    - two dimes  $10+10$
  - draw state diagram:
    - inputs: N, D, reset
    - output: open chute
  - assumptions:
    - assume N and D asserted for one cycle
    - each state has a self loop for  $N = D = 0$  (no coin)



# Example: vending machine (cont'd)

- Minimize number of states - reuse states whenever possible



present state	inputs		next state	output open
	D	N		
0¢	0	0	0¢	0
	0	1	5¢	0
	1	0	10¢	0
	1	1	—	—
5¢	0	0	5¢	0
	0	1	10¢	0
	1	0	15¢	0
	1	1	—	—
10¢	0	0	10¢	0
	0	1	15¢	0
	1	0	15¢	0
	1	1	—	—
15¢	—	—	15¢	1

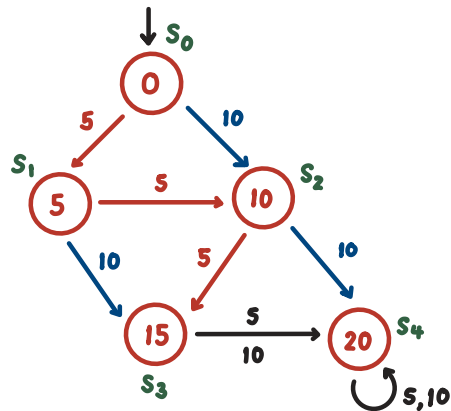
symbolic state table

# Example: vending machine (cont'd)

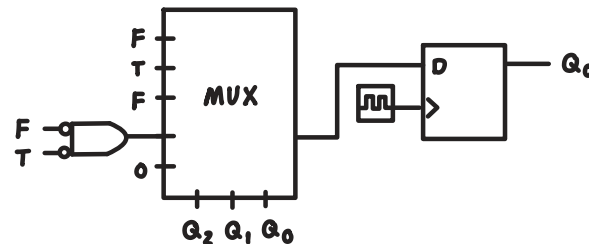
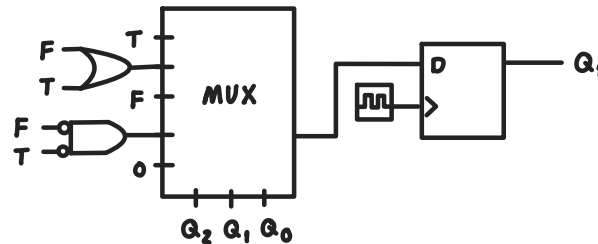
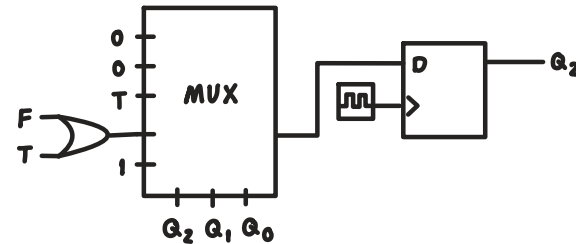
## ■ Uniquely encode states

	present state		inputs		next state		output
	Q1	Q0	D	N	D1	D0	open
00 $\Leftrightarrow$ 0¢	0	0	0	0	0	0	0
			0	1	0	1	0
			1	0	1	0	0
			1	1	—	—	—
01 $\Leftrightarrow$ 5¢	0	1	0	0	0	1	0
			0	1	1	0	0
			1	0	1	1	0
			1	1	—	—	—
10 $\Leftrightarrow$ 10¢	1	0	0	0	1	0	0
			0	1	1	1	0
			1	0	1	1	0
			1	1	—	—	—
11 $\Leftrightarrow$ 15¢	1	1	—	—	1	1	1

# Example: vending machine



	$Q_2 Q_1 Q_0$	$Q_2^+ Q_1^+ Q_0^+$	
$S_0$	0 0 0	0 0 1	$F=1$
		0 1 0	$T=1$
$S_1$	0 0 1	0 1 0	$F=1$
		0 1 1	$T=1$
$S_2$	0 1 0	0 1 1	$F=1$
		1 0 0	$T=1$
$S_3$	0 1 1	1 0 0	$F=1, T=1$
		0 1 1	$F=0, T=0$
$S_4$	1 0 0	1 0 0	None

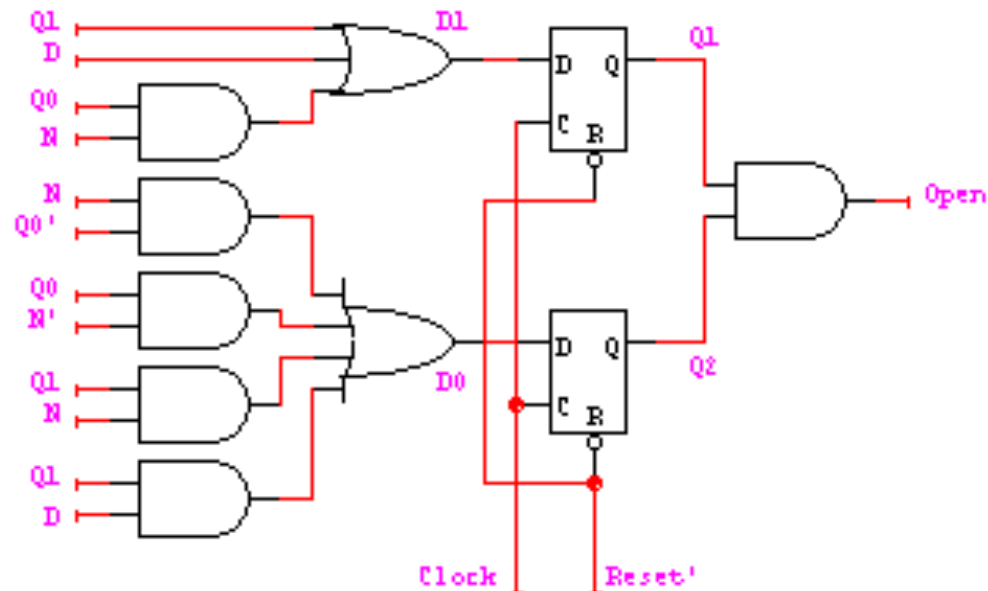
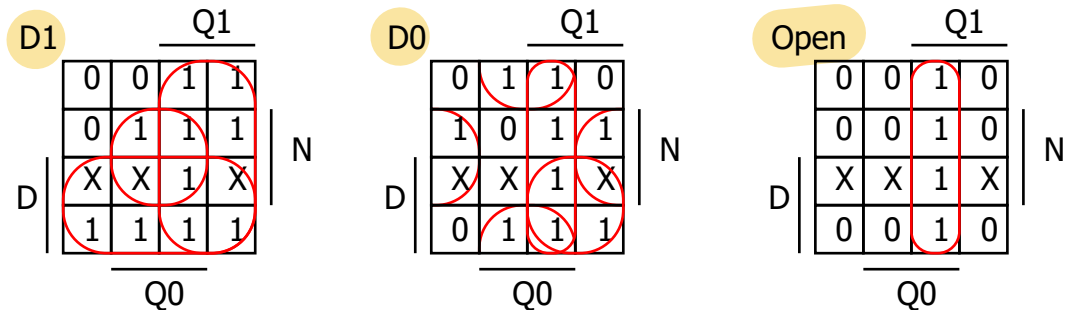




# Example: Moore implementation

## D-flip flop

### ■ Mapping to logic



$$D1 = Q1 + D + Q0 N$$

$$D0 = Q0' N + Q0 N' + Q1 N + Q1 D$$

$$OPEN = Q1 Q0$$

# Example: vending machine (cont'd)

## ■ One-hot encoding

present state				inputs		next state				output
Q3	Q2	Q1	Q0	D	N	D3	D2	D1	D0	open
0	0	0	1	0	0	0	0	0	1	0
				0	1	0	0	1	0	0
				1	0	0	1	0	0	0
				1	1	-	-	-	-	-
0	0	1	0	0	0	0	0	1	0	0
				0	1	0	1	0	0	0
				1	0	1	0	0	0	0
				1	1	-	-	-	-	-
0	1	0	0	0	0	0	1	0	0	0
				0	1	1	0	0	0	0
				1	0	1	0	0	0	0
				1	1	-	-	-	-	-
1	0	0	0	-	-	1	0	0	0	1

$$D0 = Q0 D' N'$$

$$D1 = Q0 N + Q1 D' N'$$

$$D2 = Q0 D + Q1 N + Q2 D' N'$$

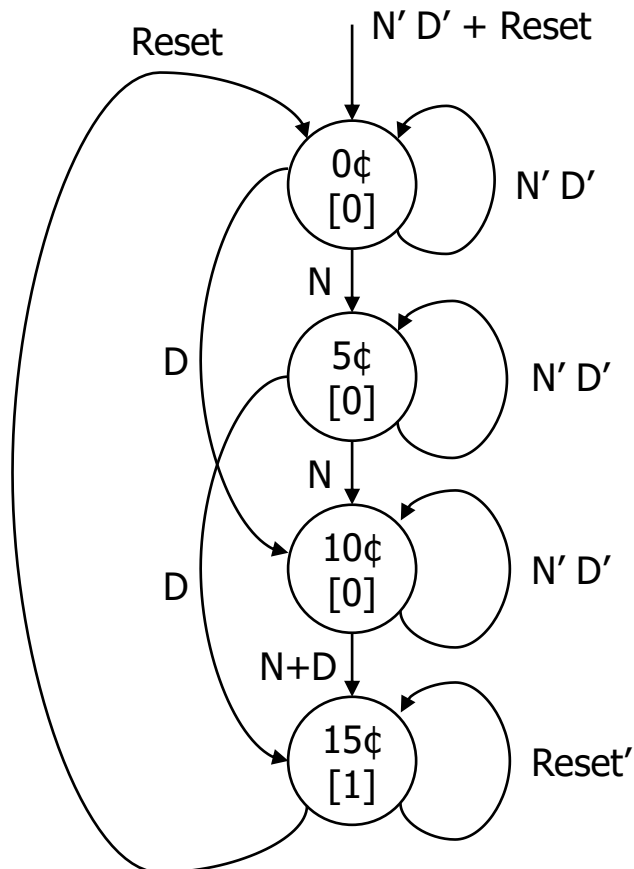
$$D3 = Q1 D + Q2 D + Q2 N + Q3$$

$$OPEN = Q3$$

# Equivalent Mealy and Moore state diagrams

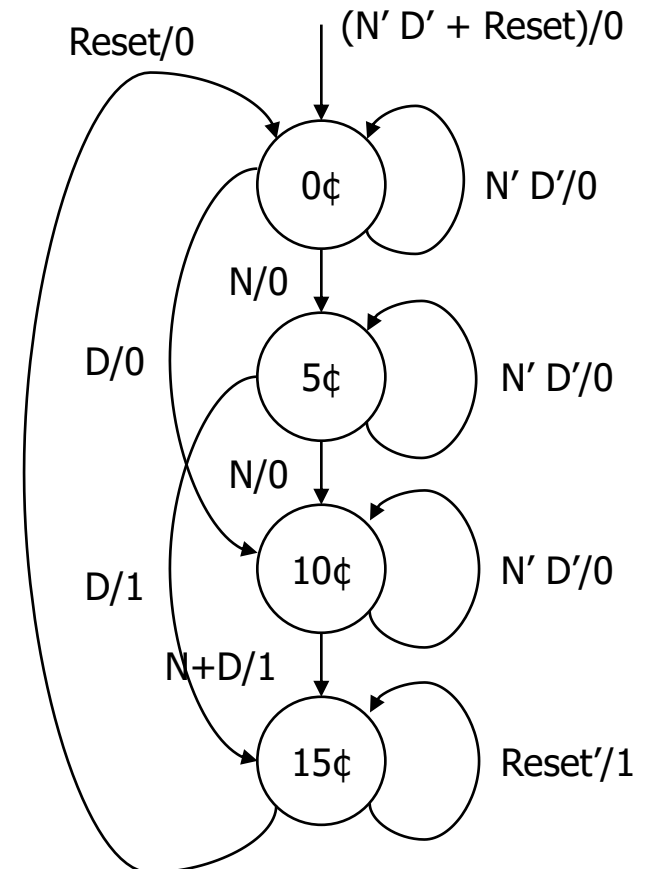
## ■ Moore machine

- outputs associated with state

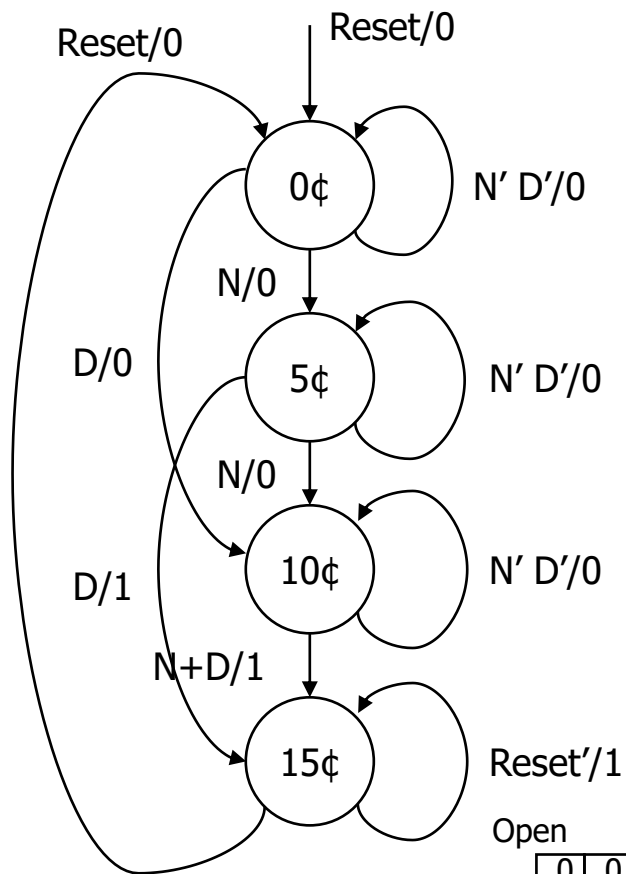


## ■ Mealy machine

- outputs associated with transitions



# Example: Mealy implementation



		Q1			
D	Open	0	0	1	0
	N	0	0	1	1
D		X	X	1	X
		0	1	1	1

present state		inputs		next state		output
Q1	Q0	D	N	D1	D0	open
0	0	0	0	0	0	0
		0	1	0	1	0
		1	0	1	0	0
		1	1	—	—	—
0	1	0	0	0	1	0
		0	1	1	0	0
		1	0	1	1	1
		1	1	—	—	—
1	0	0	0	1	0	0
		0	1	1	1	1
		1	0	1	1	1
		1	1	—	—	—
1	1	—	—	1	1	1

$$D0 = Q0'N + Q0N' + Q1N + Q1D$$

$$D1 = Q1 + D + Q0N$$

$$OPEN = Q1Q0 + Q1N + Q1D + Q0D$$

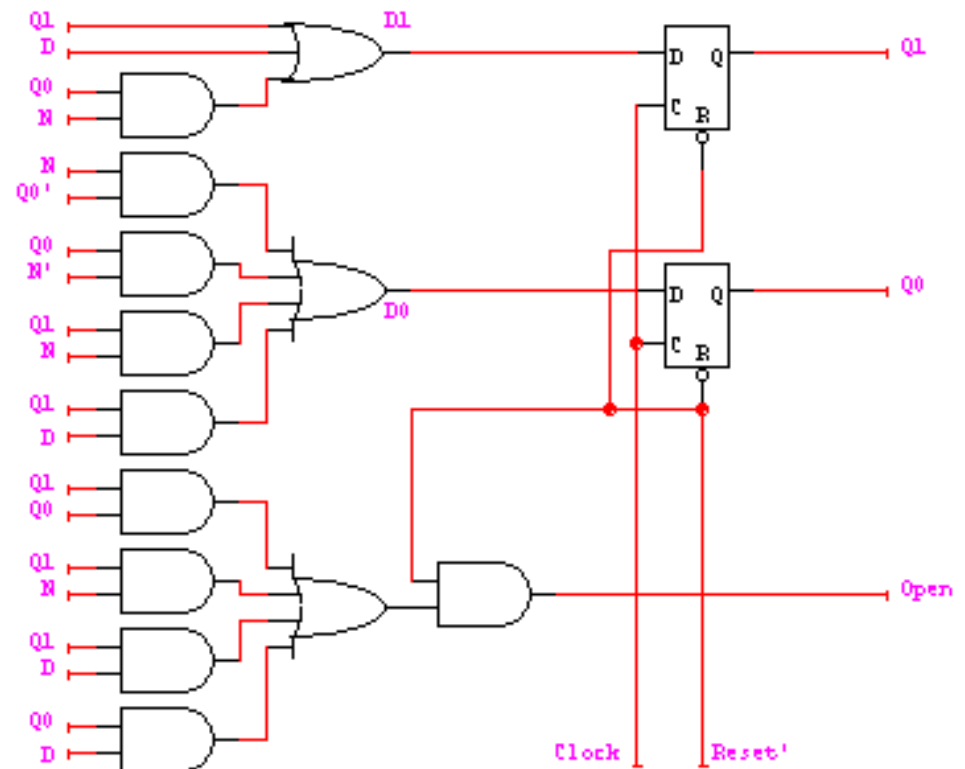
# Example: Mealy implementation

$$D0 = Q0'N + Q0N' + Q1N + Q1D$$

$$D1 = Q1 + D + Q0N$$

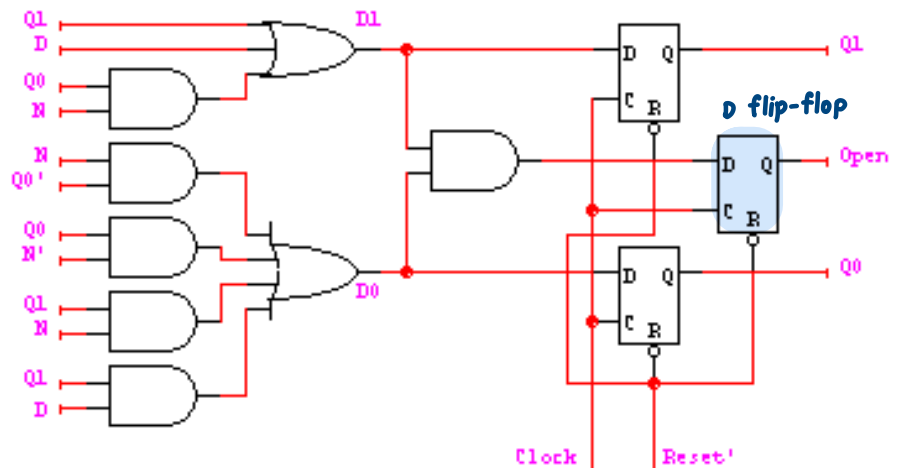
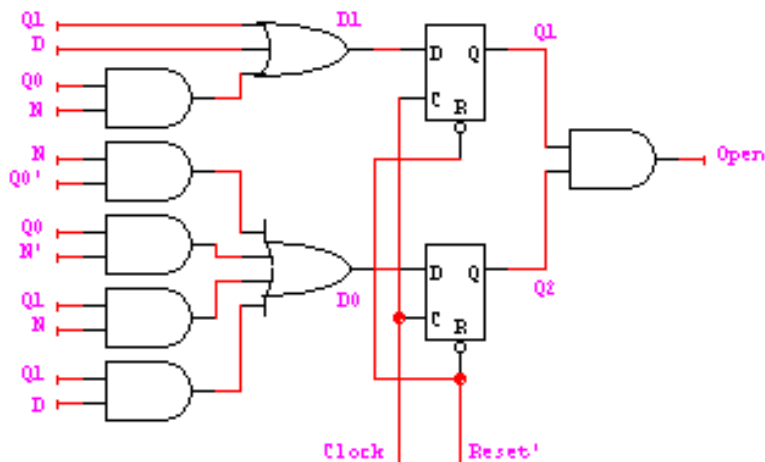
$$OPEN = Q1Q0 + Q1N + Q1D + Q0D$$

make sure OPEN is 0 when reset  
– by adding AND gate



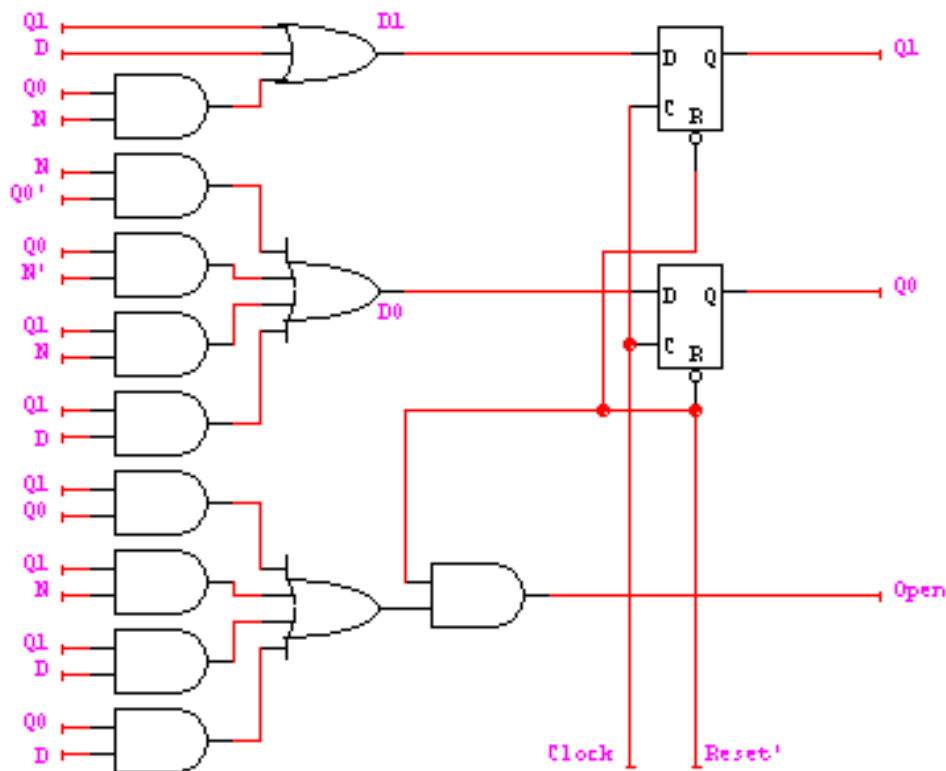
# Vending machine: Moore to synch. Mealy

- OPEN = Q1Q0 creates a combinational delay after Q1 and Q0 change in Moore implementation
- This can be corrected by retiming, i.e., move flip-flops and logic through each other to improve delay
- $$\begin{aligned}\text{OPEN}.d &= (Q1 + D + Q0'N)(Q0'N + Q0N' + Q1N + Q1D) \\ &= Q1Q0'N' + Q1N + Q1D + Q0'ND + Q0N'D\end{aligned}$$
- Implementation now looks like a **synchronous Mealy machine**
  - it is common for programmable devices to have FF at end of logic



# Vending machine: Mealy to synch. Mealy

- $OPEN.d = Q1Q0 + Q1N + Q1D + Q0D$
- $OPEN.d = (Q1 + D + Q0N)(Q0'N + Q0N' + Q1N + Q1D)$   
 $= Q1Q0N' + Q1N + Q1D + Q0'ND + Q0N'D$



Open.d

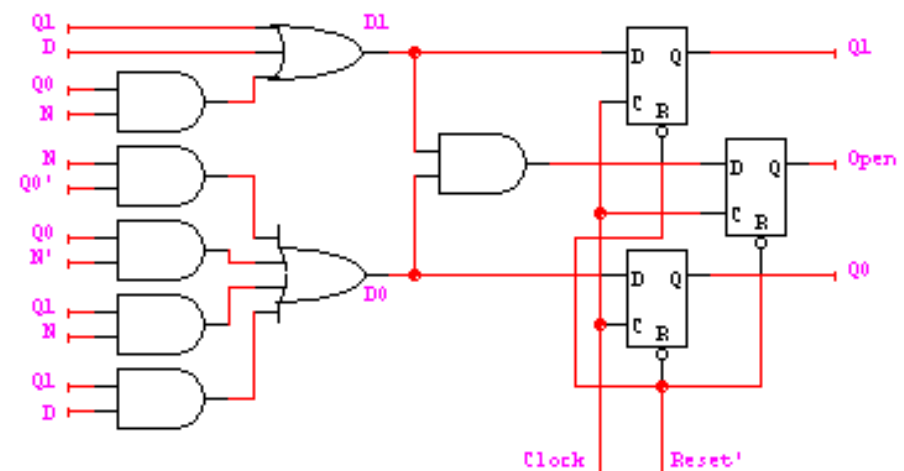
	Q1	
	0	1
D	0	0
	0	1
	1	0
	0	1

Q0

Open.d

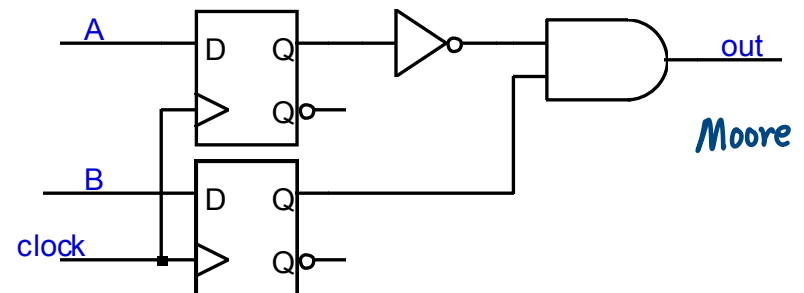
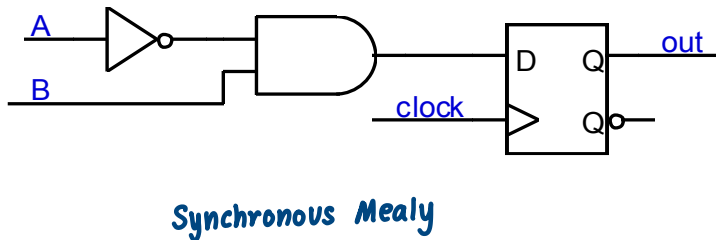
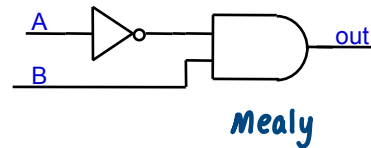
	Q1	
	0	1
D	0	0
	0	1
	X	X
	0	1

Q0



# Mealy and Moore examples

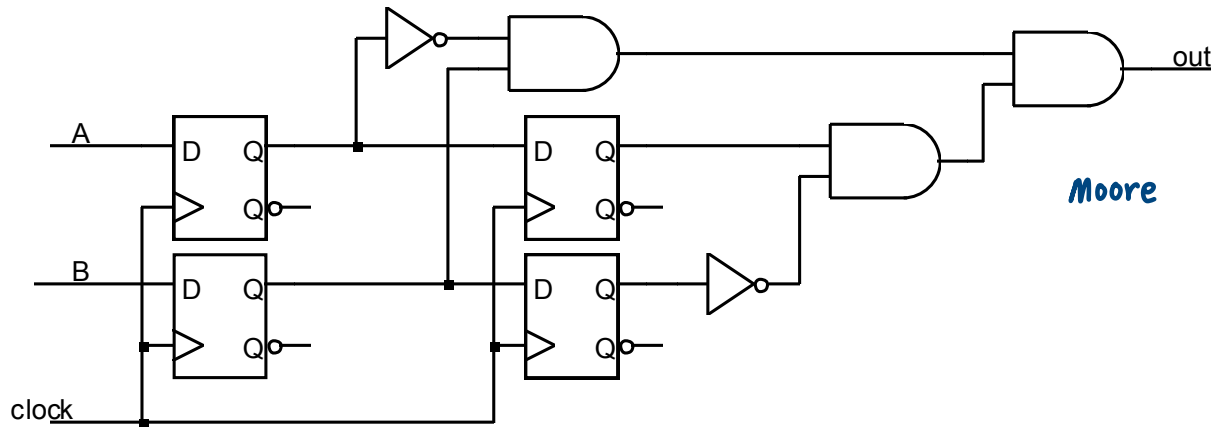
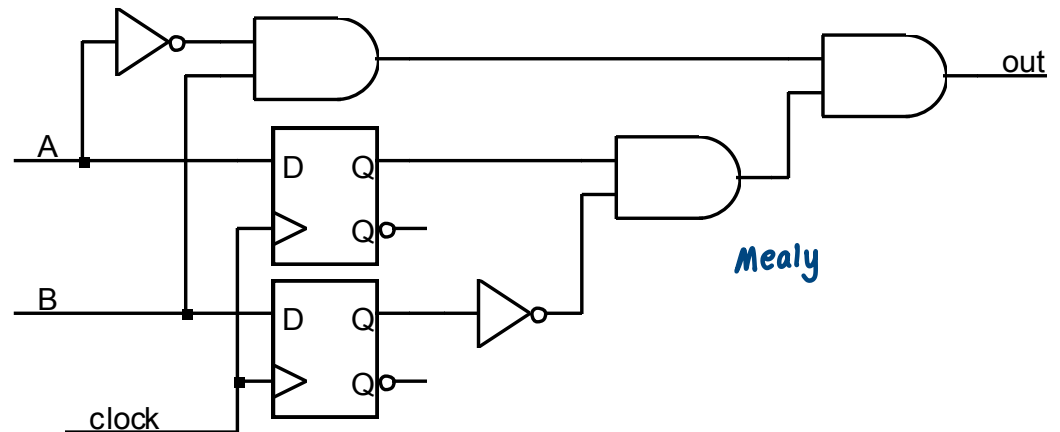
- Recognize  $A, B = 0, 1$ 
  - Mealy or Moore?





# Mealy and Moore examples (cont'd)

- Recognize A,B = 1,0 then 0,1
  - Mealy or Moore?



# Finite state machines summary

- Models for representing sequential circuits
  - abstraction of sequential elements
  - finite state machines and their state diagrams
  - inputs/outputs
  - Mealy, Moore, and synchronous Mealy machines
- Finite state machine design procedure
  - deriving state diagram
  - deriving state transition table
  - determining next state and output functions
  - implementing combinational logic