
Digital Logic Design

ECEN 3233

Module 8 – Finite State Machines

Weihua Sheng
School of Electrical and Computer Engineering
Oklahoma State University

Spring 2007

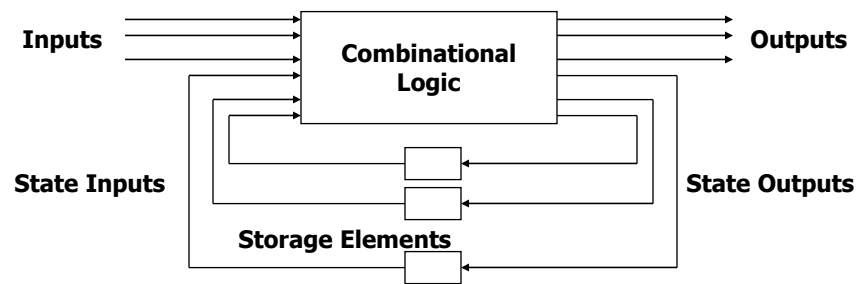
Finite State Machines

- Introduction to finite state machines
 - Simple finite state machines (counters)
 - Moore and Mealy machines
 - Analysis of finite state machines

- Finite state machine design procedure
 - state diagrams
 - state transition table
 - state encoding
 - combinational logic for next state

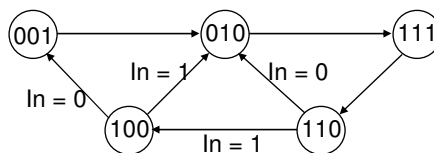
Sequential logic circuit

- combinational logic, state and feedback loop



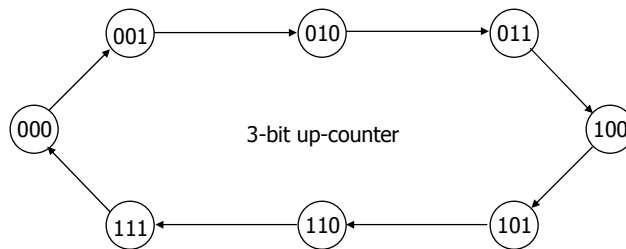
Finite state machine representations

- States: determined by possible values in sequential storage elements
- Transitions: change of state
- Clock: controls when state can change by controlling storage elements



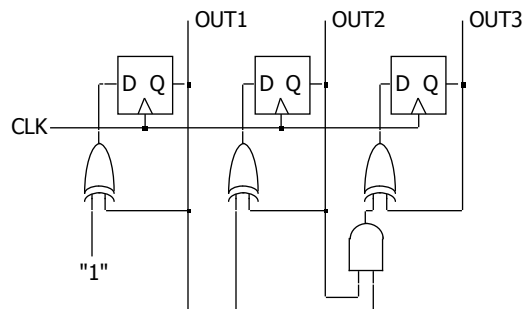
Simple finite state machine: counters

- Counters
 - proceed through well-defined sequence of states
- 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, ...



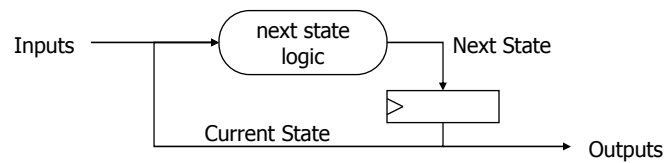
Implementation of counters

- 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



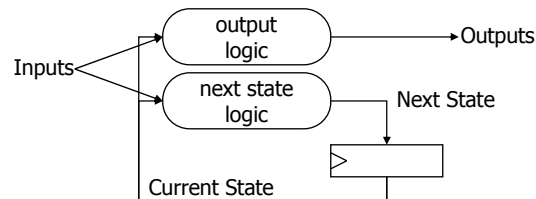
Counter/shift-register model

- 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

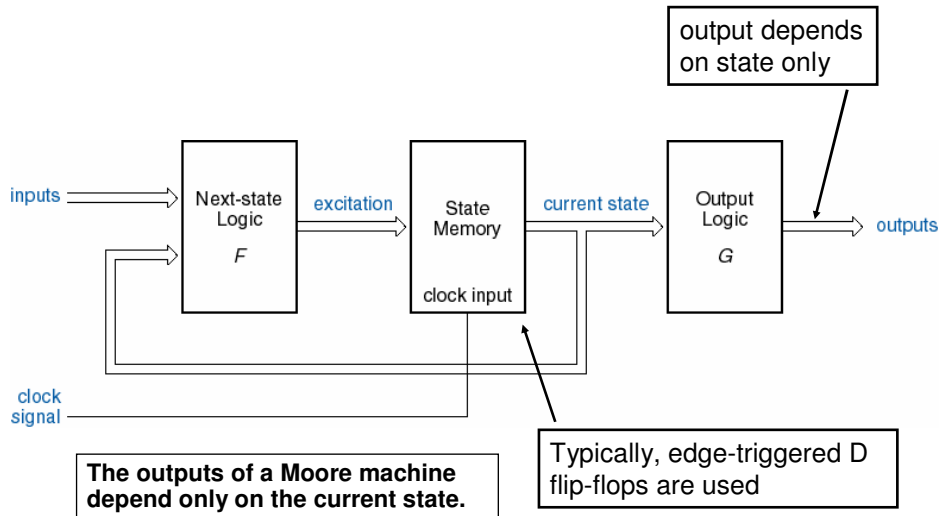


General state machine model

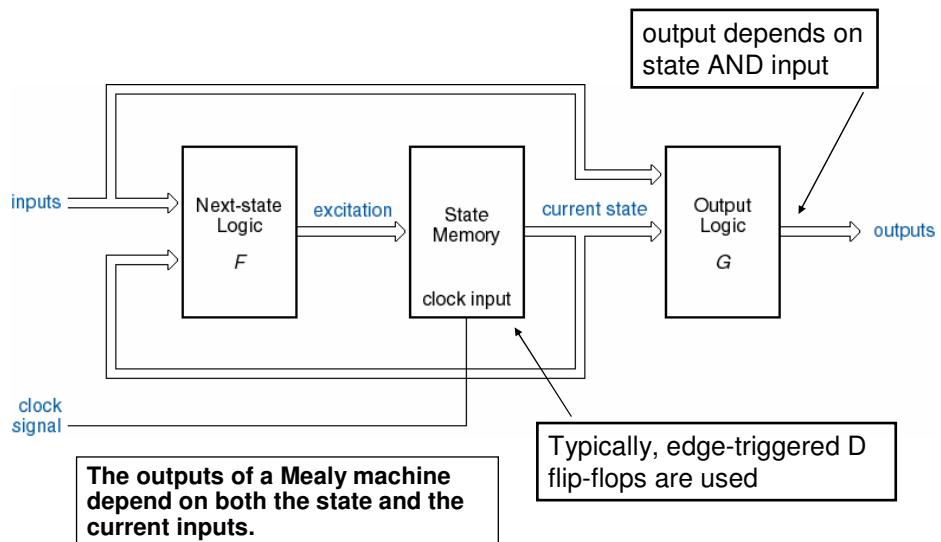
- Values stored in registers represent the state of the circuit
- Combinational logic computes:
 - next state
 - function of current state and inputs
 - outputs
 - function of current state and inputs (Mealy machine)
 - function of current state only (Moore machine)



State-Machine Structure (Moore)

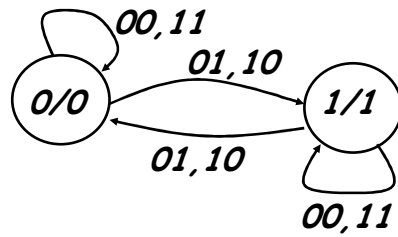


State-Machine Structure (Mealy)



Moore model state diagram

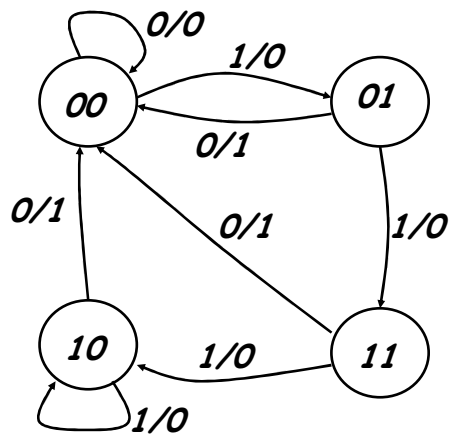
State Diagram



Reads as:
When at state $s1$ with output 01 and apply input I , we proceed to state $s2$ with Output 02 .

Mealy model state diagram

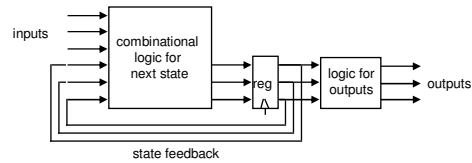
State Diagram



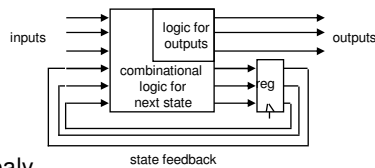
Reads as:
When at state $s1$ and apply input I , we get output O and proceed to state $s2$.

Comparison of Moore and Mealy machines

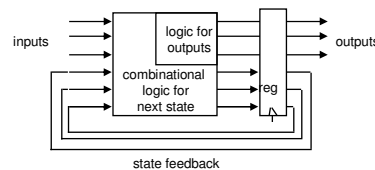
- Moore



- Mealy



- Synchronous Mealy

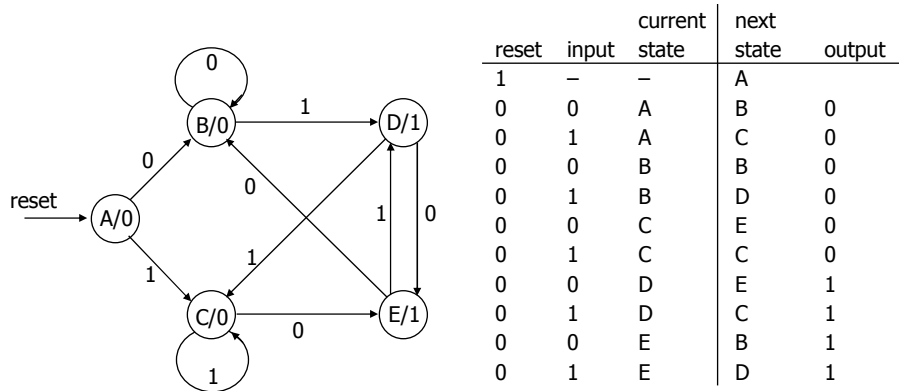


Comparison of Mealy and Moore machines

- Mealy machines tend to have less states
- 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

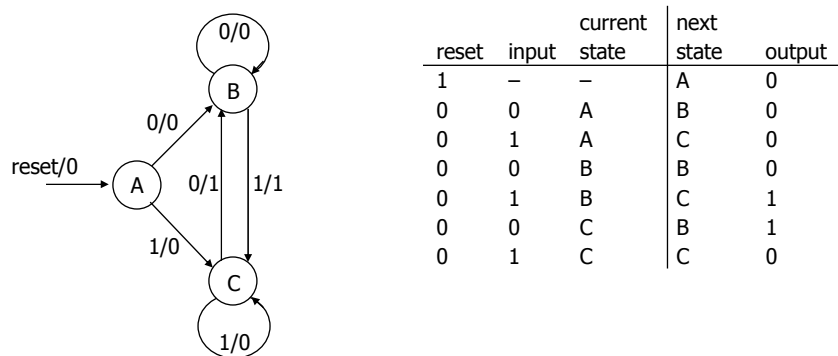
Specifying outputs for a Moore machine

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



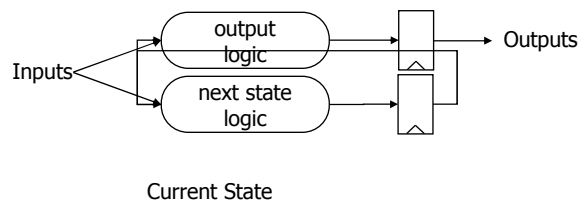
Specifying outputs for a Mealy machine

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



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

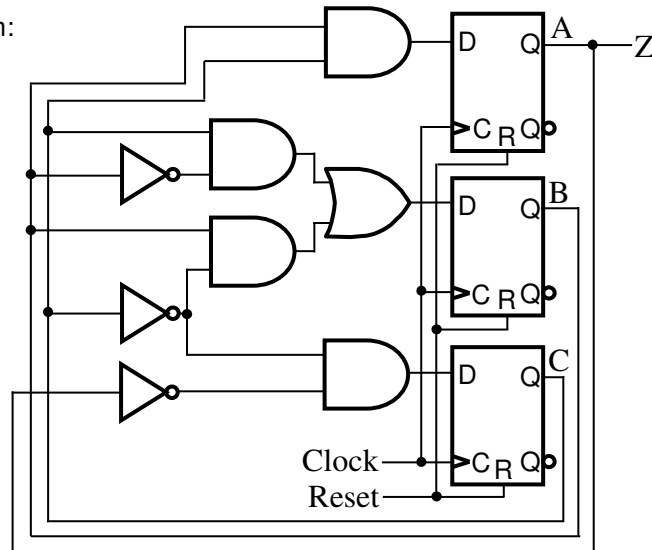


State-machine analysis steps

- Assumption: Starting point is a logic diagram.
1. Determine next-state function and output function.
 - 2a. Construct state table
 - For each state/input combination, determine the excitation value.
 - Using the characteristic equation, determine the corresponding next-state values (trivial with D flip flops).
 - 2b. Construct output table
 - For each state/input combination, determine the output value. (Can be combined with state table.)
 3. (Optional) Draw state diagram

Example FSM Analysis 1

- Logic Diagram:



Excitation Equations

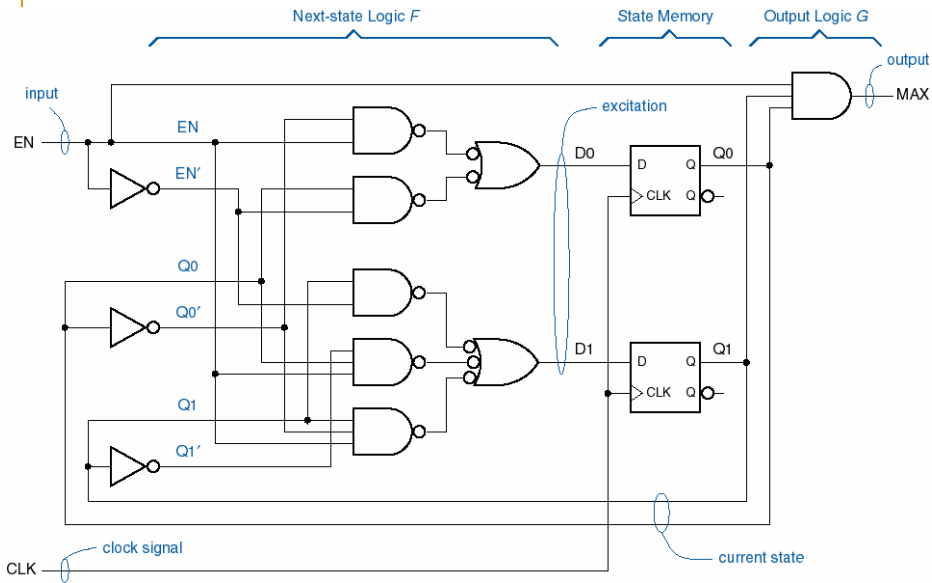
- Variables
 - Inputs: None
 - Outputs: Z
 - State Variables: A, B, C
- Initialization: Reset to (0,0,0)
- Excitation Equations
 - $A^+ = DA =$
 - $B^+ = DB =$
 - $C^+ = Dc =$
- Output
 - $Z =$

State Table

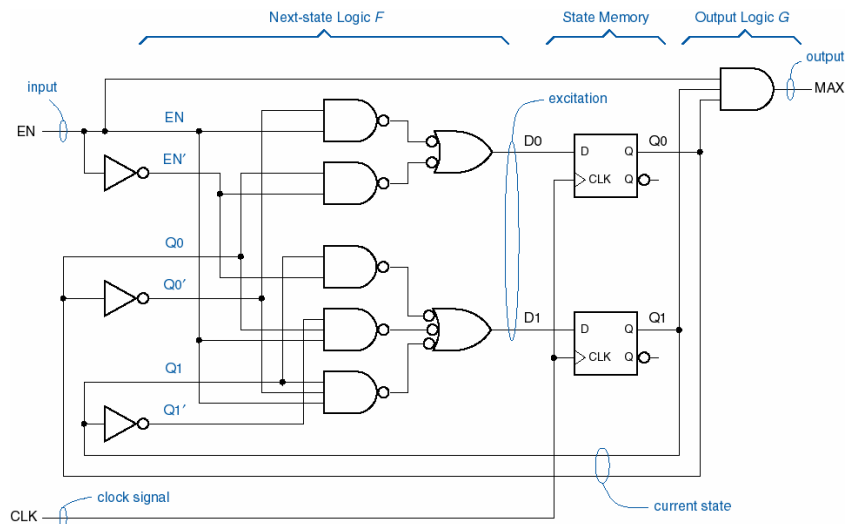
Current State ABC	Next State A+ B+ C+	Z
0 0 0		
0 0 1		
0 1 0		
0 1 1		
1 0 0		
1 0 1		
1 1 0		
1 1 1		

State Diagram

Example FSM analysis 2



Excitation Equations



$$D0 = Q0 \cdot EN' + Q0' \cdot EN$$

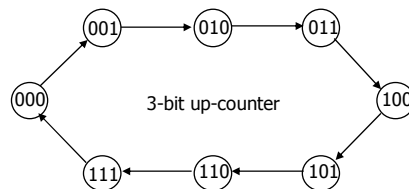
$$D1 = Q1 \cdot EN' + Q1' \cdot Q0 \cdot EN + Q1 \cdot Q0' \cdot EN$$

FSM design procedure: counters

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

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

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

$$Q^* = D$$

So, we fill the K-Maps with the values we need D to be in order to make Q^* take on the correct new value. For a counter, the next state ($N3, N2, N1$) is just a function of the current state ($C3, C2, C1$), so the K-Maps are all functions of only the current state;

e.g., the N3 column is used to fill the N3 (D3) K-Map.

N3

C3	C2	C1
0	0	1
0	1	1
1	0	0
1	1	0

C1

N2

C3	C2	C1
0	1	1
0	0	0
1	0	0
1	1	0

C1

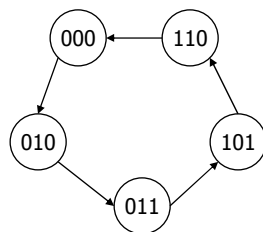
N1

C3	C2	C1
1	1	1
1	1	1
0	0	0
0	0	0

C1

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

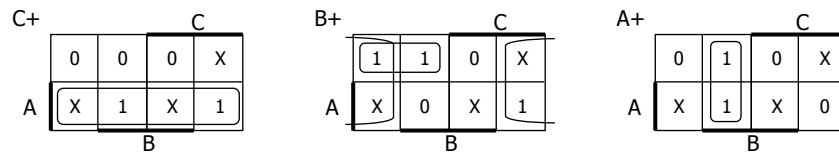


Present State			Next State		
C	B	A	C+	B+	A+
0	0	0	0	1	0
0	0	1	-	-	-
0	1	0	0	1	1
0	1	1	1	0	1
1	0	0	-	-	-
1	0	1	1	1	0
1	1	0	0	0	0
1	1	1	-	-	-

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

More complex counter example (cont'd)

- Step 3: K-maps for next state functions



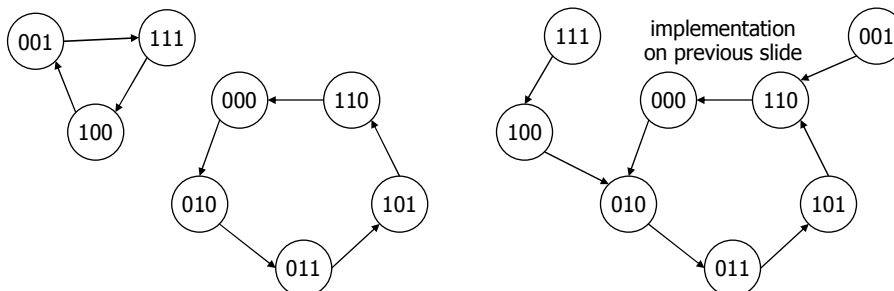
$$C+ = A$$

$$B+ = B' + A'C'$$

$$A+ = BC'$$

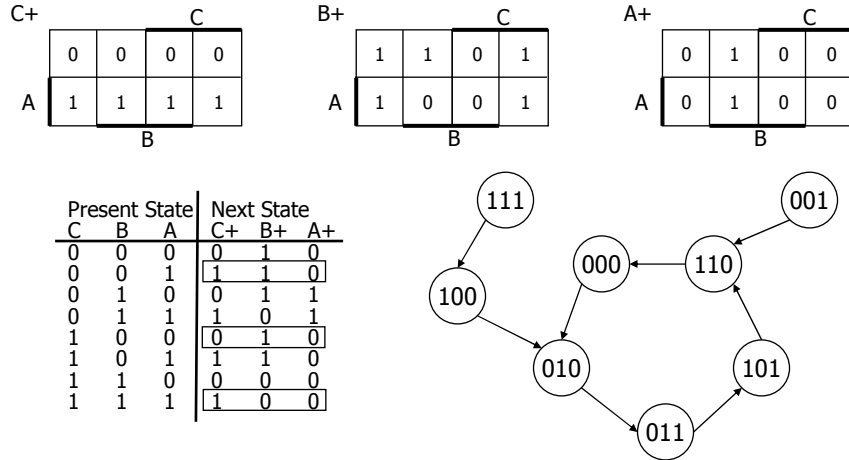
Self-starting counters

- 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



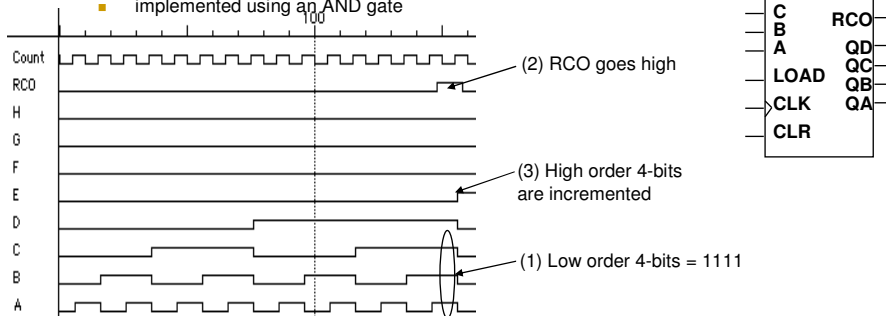
Self-starting counters (cont'd)

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



Four-bit binary synchronous up-counter

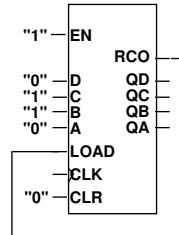
- Standard component with many applications
 - positive edge-triggered FFs w/ synchronous load and clear inputs
 - parallel load data from D, C, B, A
 - enable inputs: must be asserted to enable counting
 - RCO: ripple-carry out used for cascading counters
 - high when counter is in its highest state 1111
 - implemented using an AND gate



Offset counters

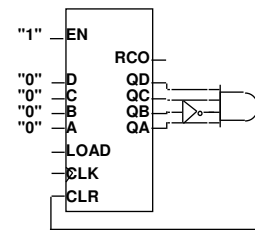
- Starting offset counters – use of synchronous load

- e.g., 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1111, 0110, . . .



- Ending offset counter – comparator for ending value

- e.g., 0000, 0001, 0010, . . . , 1100, 1101, 0000



- Combinations of the above (start and stop value)

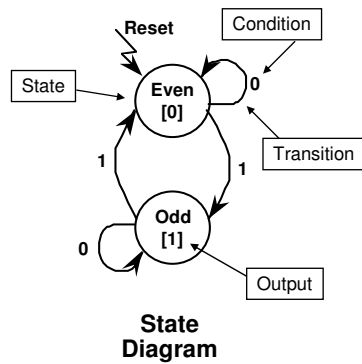
General FSM design procedure

- Understand the problem
 - Make sure you understand the conditions under which the FSM transitions between states and the various outputs are asserted
- Obtain an abstract representation of the FSM
 - Put into state diagram form
- Perform state minimization
 - Remove certain paths and states if they are redundant
 - Not needed in counter design
- Perform state assignment
 - A good choice of how to encode the states can lead to simpler implementation
- Implement the finite state machine
 - As done in the counter design

Odd Parity Checker

Example: Odd Parity Checker

Assert output whenever input bit stream has odd # of 1's



Present State	Input	Next State	Output
Even	0	Even	0
Even	1	Odd	0
Odd	0	Odd	1
Odd	1	Even	1

Symbolic State Transition Table

Present State	Input	Next State	Output
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

Encoded State Transition Table

Odd Parity Checker (continued)

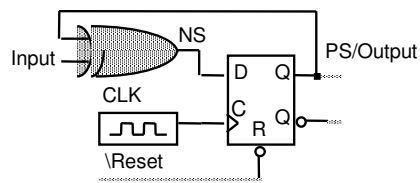
Example: Odd Parity Checker

D flip-flop: $Q \leftarrow D$ after clock is applied

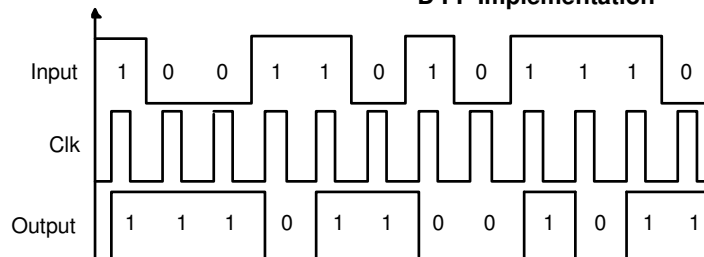
(Characteristic Equation)

Next State/Output Functions:

$$NS = PS \text{ xor } INPUT; \text{ OUT} = PS$$



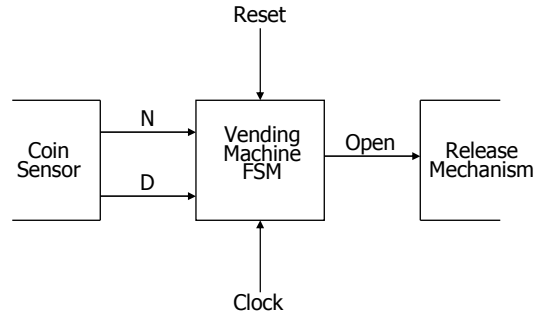
D FF Implementation



Timing Behavior: Input 1 0 0 1 1 0 1 0 1 1 1 0

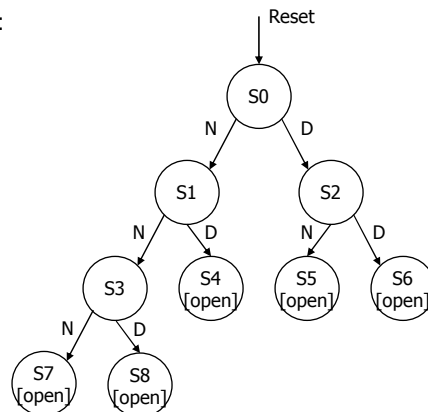
Example: vending machine

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



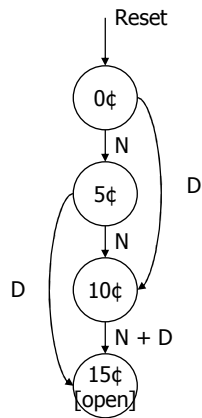
Example: vending machine (cont'd)

- Suitable abstract representation
 - tabulate typical input sequences:
 - 3 nickels
 - nickel, dime
 - dime, nickel
 - two dimes
 - 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 O1 O0	inputs		next state D1 D0	output open
	D	N		
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	0
	1	1	- -	-
1 0	0	0	1 0	0
	0	1	1 1	0
	1	0	1 1	0
	1	1	- -	-
1 1	-	-	1 1	1

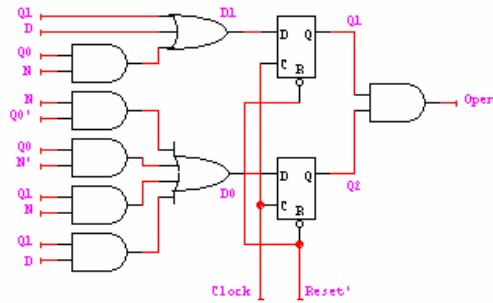
Example: Moore implementation

- Mapping to logic

D1	Q1		
	0	1	N
	0	1	1
	1	1	1
D	X	X	X
	1	1	1
	Q0		

D0	Q1		
	0	1	0
	1	0	1
	X	X	X
D	X	X	X
	0	1	1
	Q0		

Open	Q1		
	0	1	0
	0	0	1
	X	X	X
D	X	X	X
	0	0	1
	Q0		



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

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

$$OPEN = Q1 Q0$$

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

$$D0 = Q0 D' N'$$

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

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

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

$$OPEN = Q3$$

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