

# **CH-7: Finite State Machines**

*Contemporary Logic Design*

YONSEI UNIVERSITY

Fall 2016

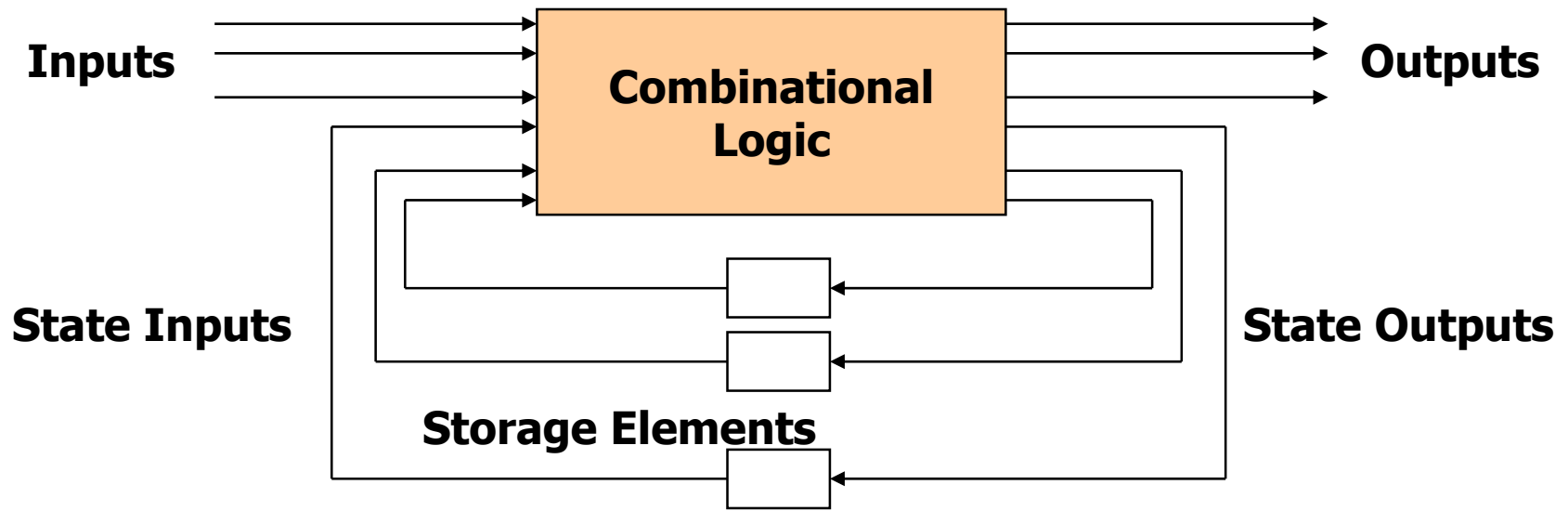
# Finite State Machines

---

- *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

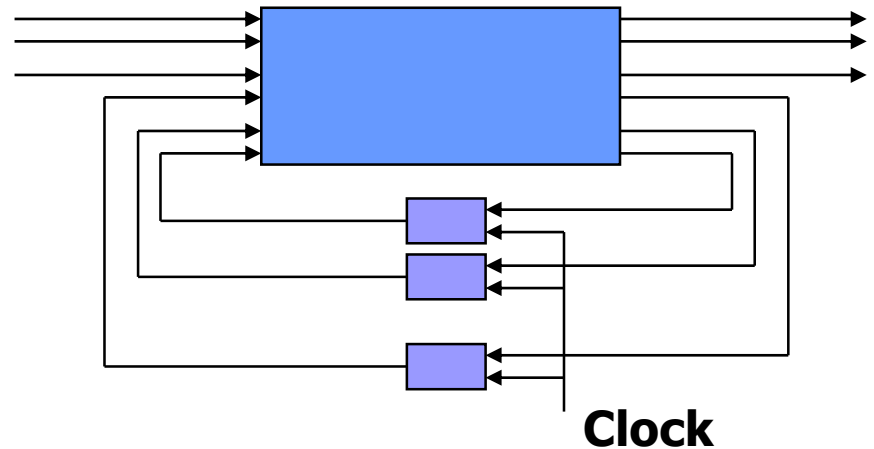
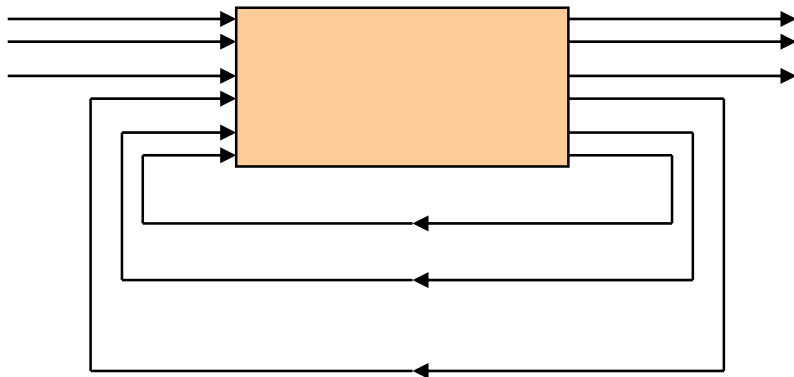
# Abstraction of State Elements

- Divide seq circuit into *combinational logic and state parts*
- Implementation of storage elements leads to various forms of sequential logic



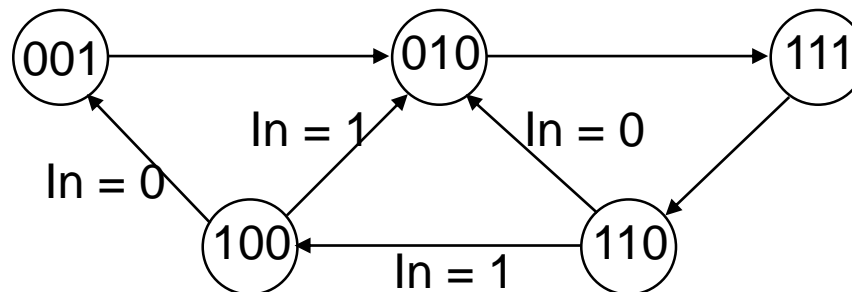
# Forms of Sequential Logic

- 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)



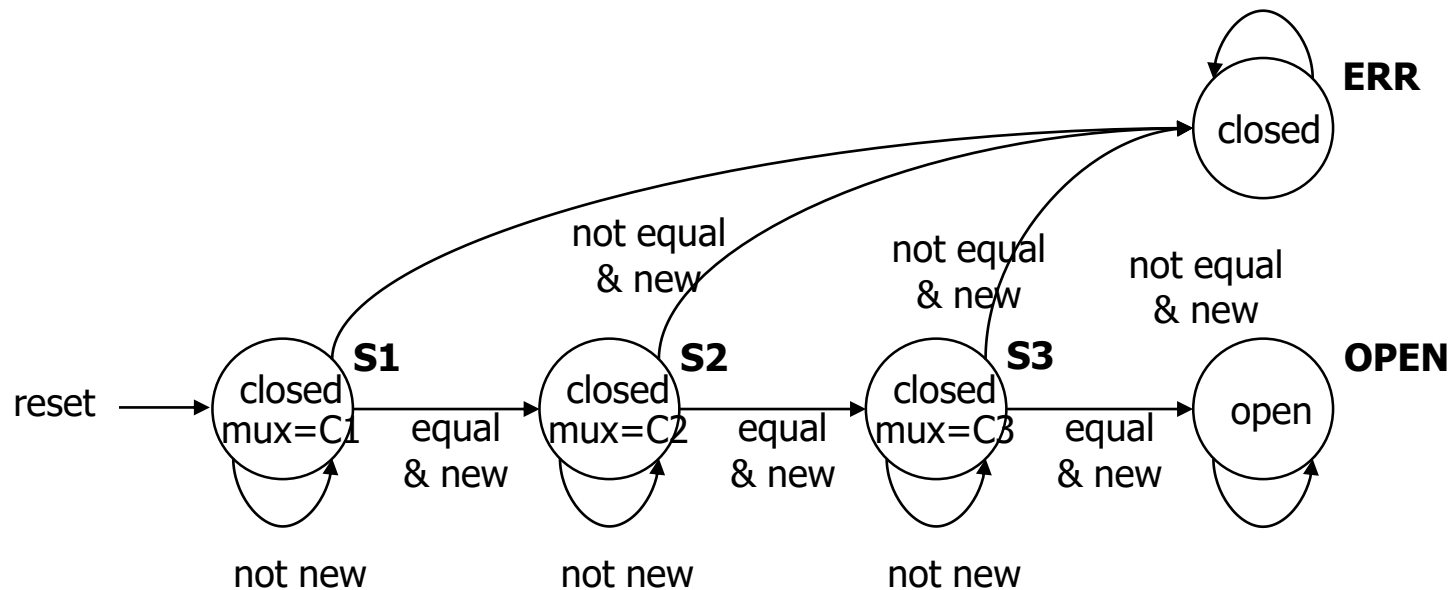
# Finite State Machine Representations

- *States*: determined by possible values in sequential storage elements
- *Transitions*: change of state
- *Clock*: control when state can change by controlling storage elements
- Sequential logic
  - ◆ sequencing through a series of states
  - ◆ based on the values on input signals
  - ◆ clock period specifies when to change states



# Example Finite State Machine Diagram

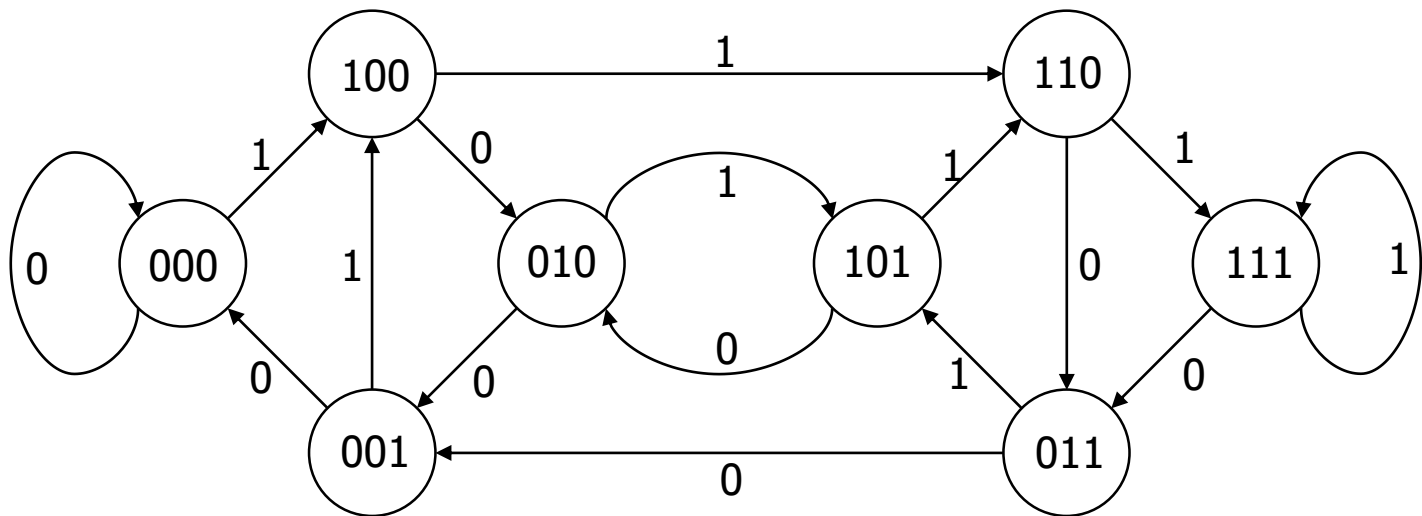
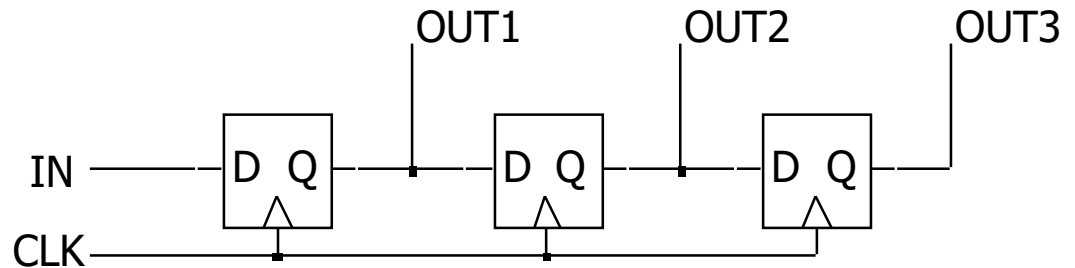
- Combination lock from introduction in chap. 1
  - ◆ 5 states: S1, S2, S3, OPEN, ERR
  - ◆ 5 self-transitions
  - ◆ 6 other transitions between states
  - ◆ 1 reset transition (from all states) to state S1



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

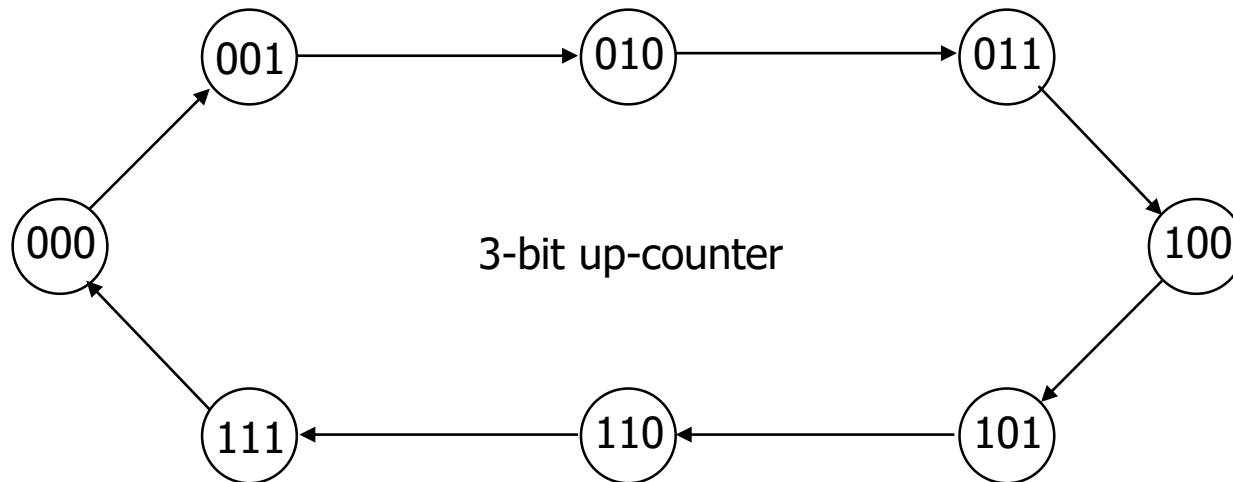
- Shift register: can draw SD for all possible inputs at each state

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



# Counters: Simple Finite State Machines

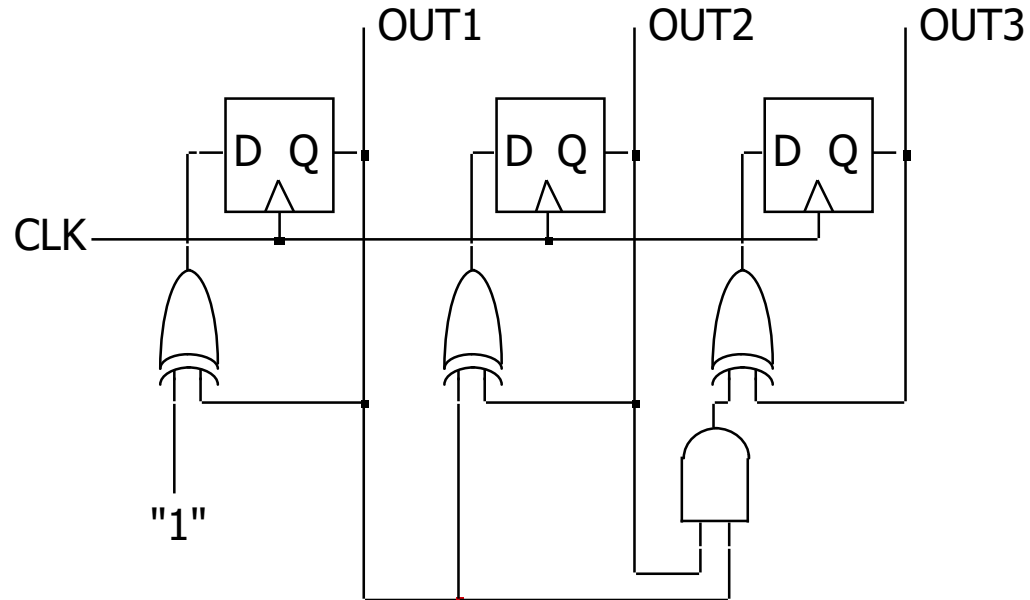
- Counters
  - ◆ proceed through well-defined sequence of states in response to enable
- Many types of counters: binary, BCD, Gray-code
  - ◆ 3-bit up-counter: 000, 001, 010, 011, 100, 101, 110, 111, 000, ...
  - ◆ 3-bit down-counter: 111, 110, 101, 100, 011, 010, 001, 000, 111, ...





# State Diagram into Logic

- Counter
  - ◆ 3 *flip-flops* to hold state
  - ◆ *Combinational logic* to compute next states
  - ◆ *clock signal* controls when flip-flop memory can change
    - wait long enough for combinational logic to compute new value



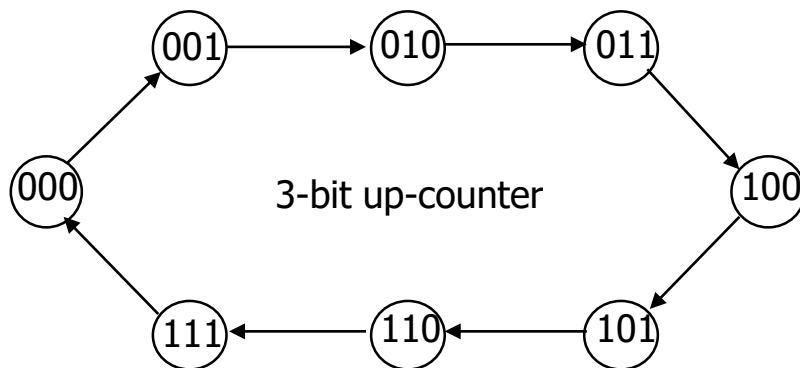
# FSM Design Procedure

---

- 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 on each sequence
  
- Implementation
  - ◆ flip-flop for each state bit
  - ◆ combinational logic based on encoding

# 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 will be based on encoding

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

| 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  |

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

N3

|    |   |    |    |
|----|---|----|----|
|    |   |    | C3 |
|    | 0 | 0  | 1  |
|    |   |    | 1  |
| C1 | 0 | 1  | 0  |
|    |   |    | 1  |
|    |   | C2 |    |

N2

|    |   |    |    |
|----|---|----|----|
|    |   |    | C3 |
|    | 0 | 1  | 1  |
|    |   |    | 0  |
| C1 | 1 | 0  | 0  |
|    |   |    | 1  |
|    |   | C2 |    |

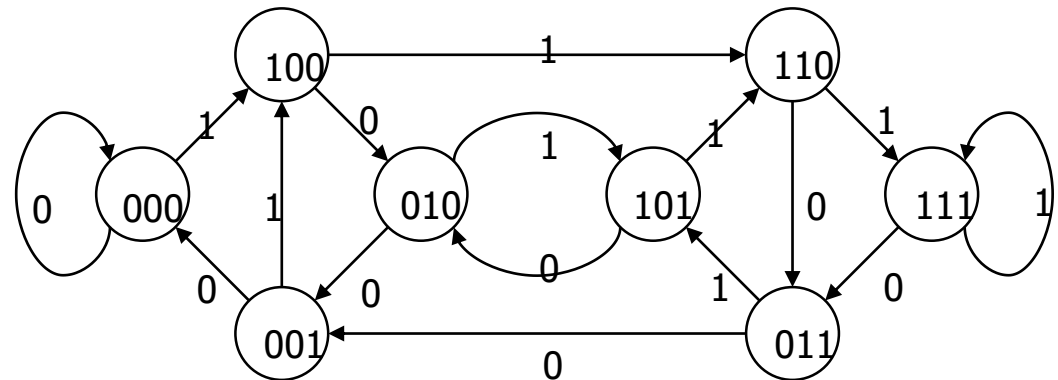
N1

|    |   |    |    |
|----|---|----|----|
|    |   |    | C3 |
|    | 1 | 1  | 1  |
|    |   |    | 1  |
| C1 | 0 | 0  | 0  |
|    |   |    | 0  |
|    |   | C2 |    |

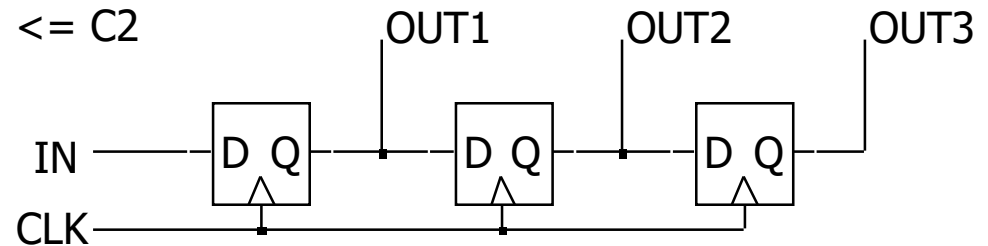
# Back to Shift Register Case

- 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  | 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  | 1  | 1  | 1  | 0  |
| 1  | 1  | 1  | 0  | 1  | 1  | 1  |
| 1  | 1  | 1  | 1  | 1  | 1  | 1  |

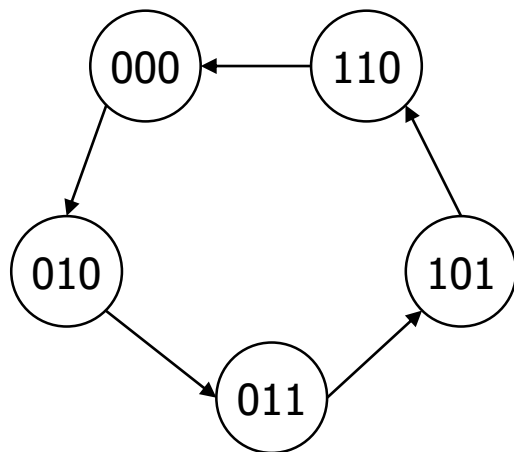


$N1 \leq In$   
 $N2 \leq C1$   
 $N3 \leq C2$



# 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

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

|    |   | C |   |   |   |
|----|---|---|---|---|---|
| C+ |   | 0 | 0 | 0 | X |
| A  | B | X | 1 | X | 1 |

|    |   | C |   |   |   |
|----|---|---|---|---|---|
| B+ |   | 1 | 1 | 0 | X |
| A  | B | X | 0 | X | 1 |

|    |   | C |   |   |   |
|----|---|---|---|---|---|
| A+ |   | 0 | 1 | 0 | X |
| A  | B | X | 1 | X | 0 |

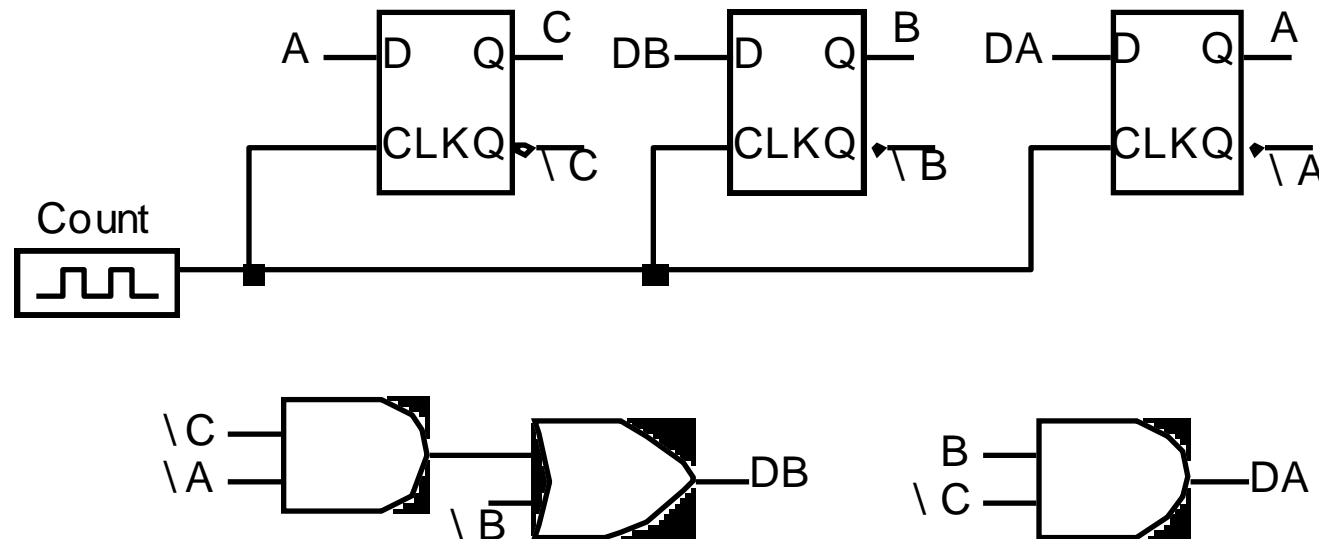
$$C+ \leq A$$

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

$$A+ \leq BC'$$

# More Complex Counter Example

- Implementation of 3-bit counter

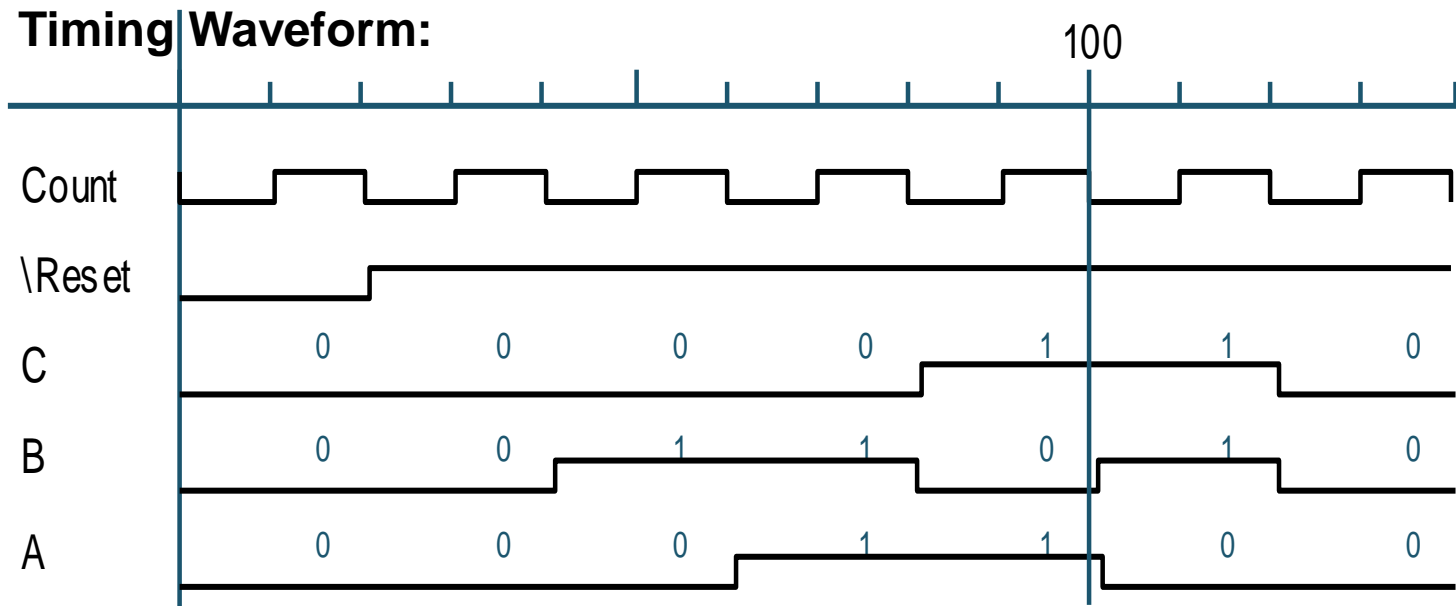


**Resulting Logic Level Implementation:**  
**3 Gates, 8 Input Literals + Flipflop connections**



# More Complex Counter Example

- Timing waveform of 3-bit counter



# Self-starting Counters

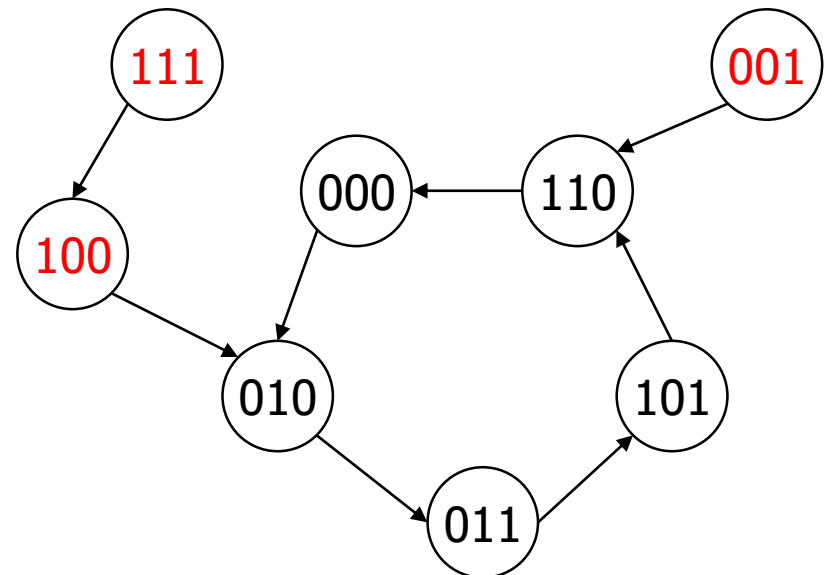
- Re-deriving state transition table from don't care assignment

|    |   |   |   |
|----|---|---|---|
| C+ |   | C |   |
|    | 0 | 0 | 0 |
| A  | 1 | 1 | 1 |
|    | B |   |   |

|    |   |   |   |
|----|---|---|---|
| B+ |   | C |   |
|    | 1 | 1 | 0 |
| A  | 1 | 0 | 0 |
|    | B |   |   |

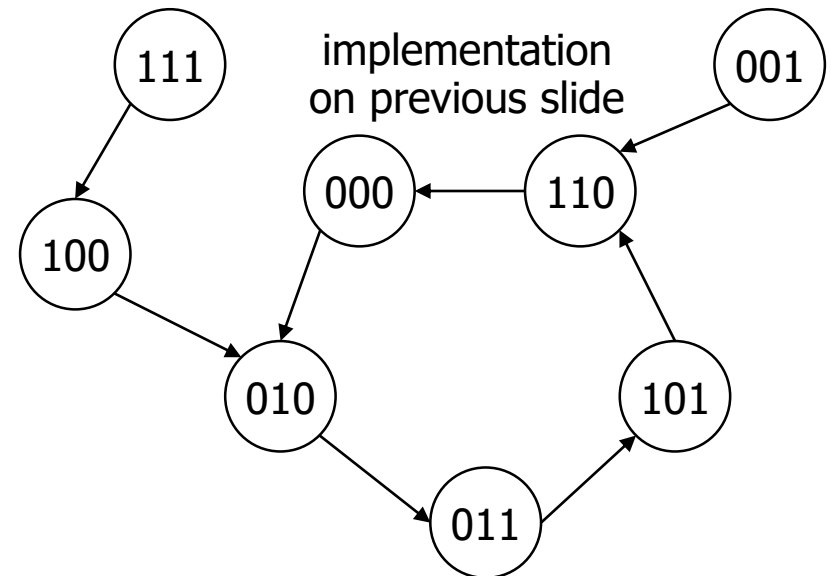
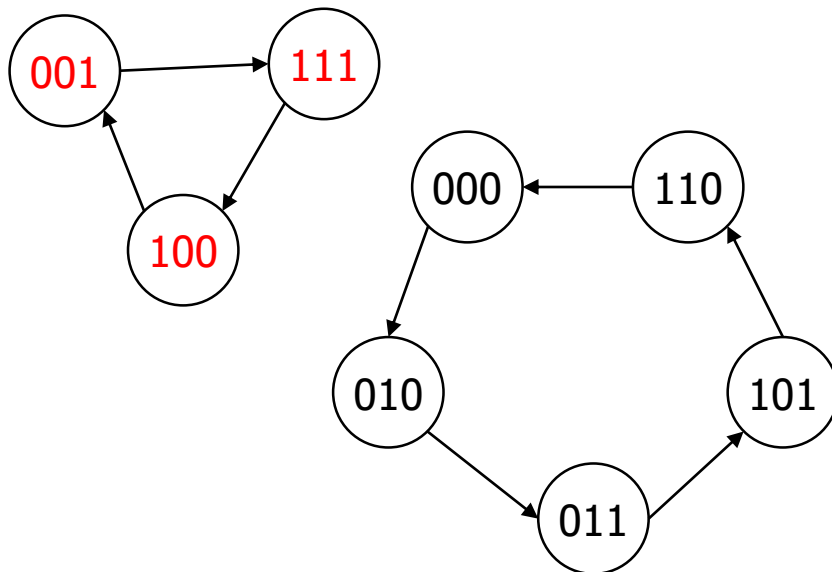
|    |   |   |   |
|----|---|---|---|
| A+ |   | C |   |
|    | 0 | 1 | 0 |
| A  | 0 | 1 | 0 |
|    | B |   |   |

| 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 transit to a valid state



# Activity

---

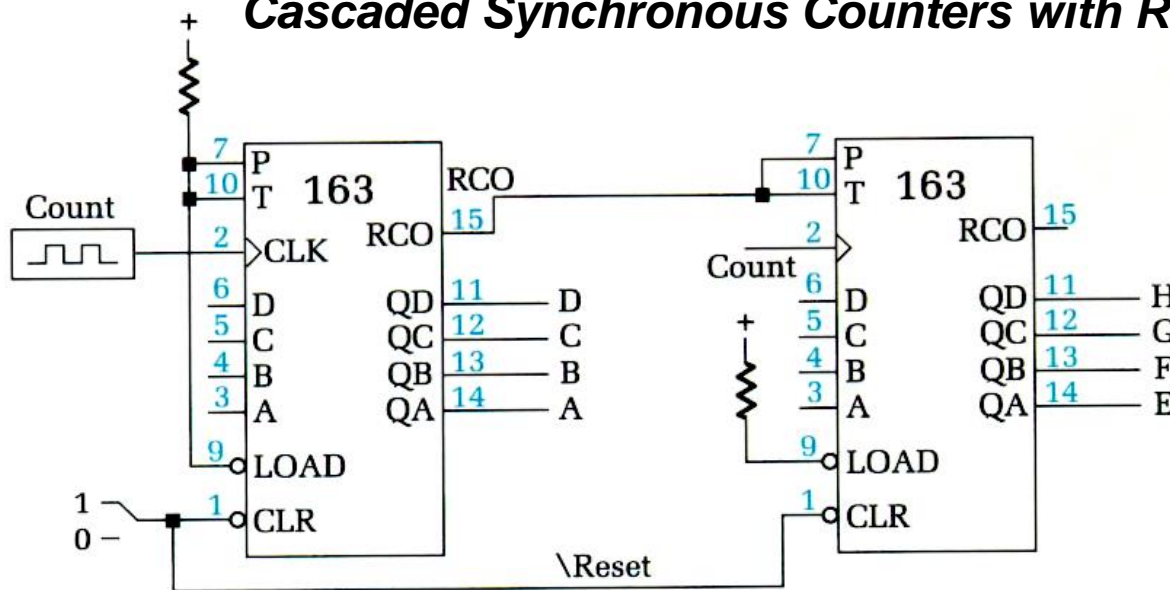
- 2-bit up-down counter (2 inputs)
  - ◆ direction:  $D = 0$  for up,  $D = 1$  for down
  - ◆ count:  $C = 0$  for hold,  $C = 1$  for count

# Activity

---

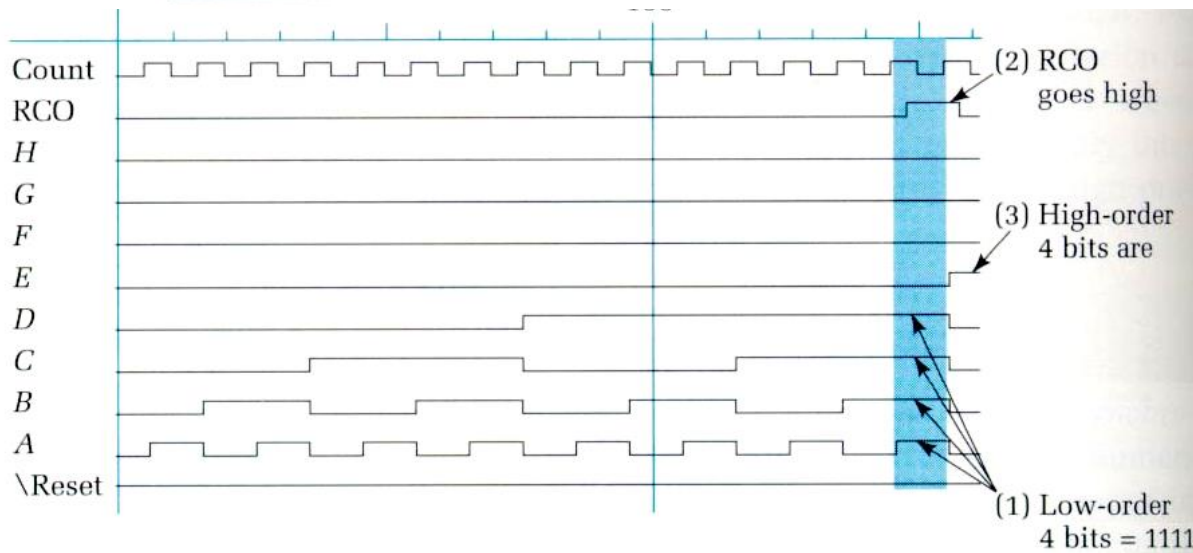
# Catalog Counter

## Cascaded Synchronous Counters with Ripple Carry Outputs



**First stage RCO enables second stage for counting**

**RCO asserted soon after stage enters state 1111**

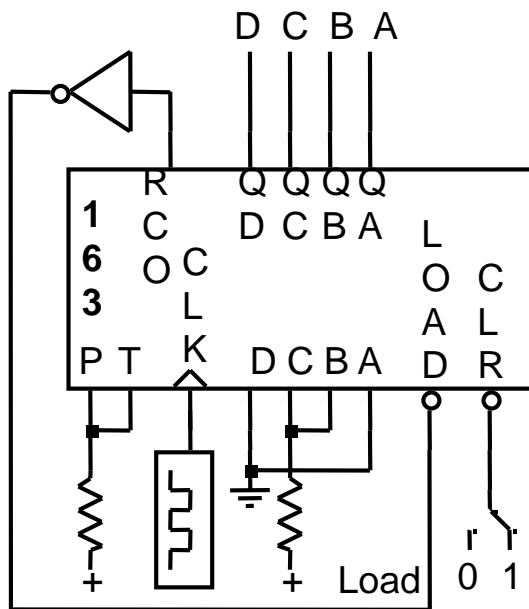


# Starting Offset Counter

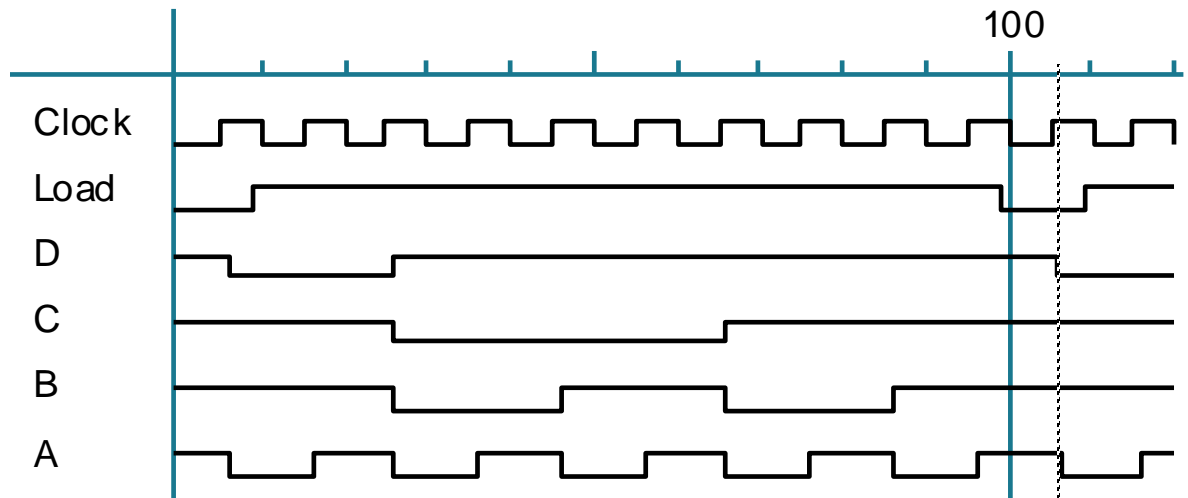
## *The Power of Synchronous Clear and Load*

**Starting Offset Counters:**

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



**0110**  
*is the state  
to be loaded*



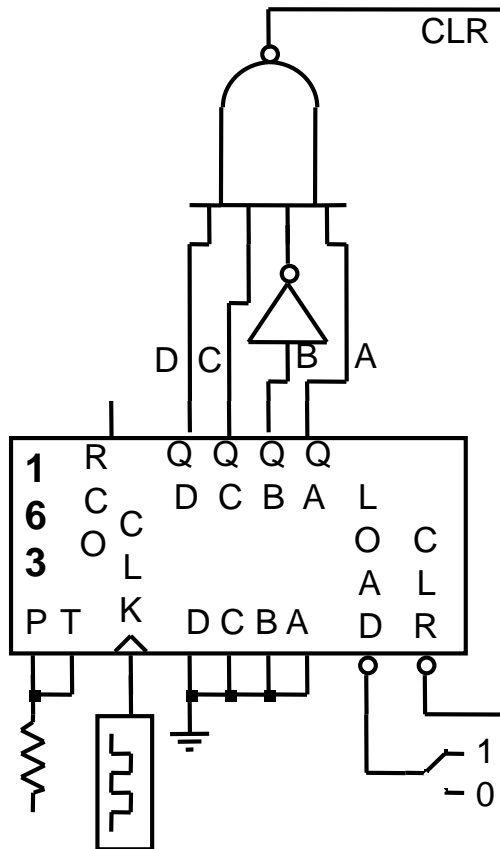
**Use RCO signal to trigger Load of a new state**

**Since Load signal is synchronous, state changes only on the next rising clock edge**

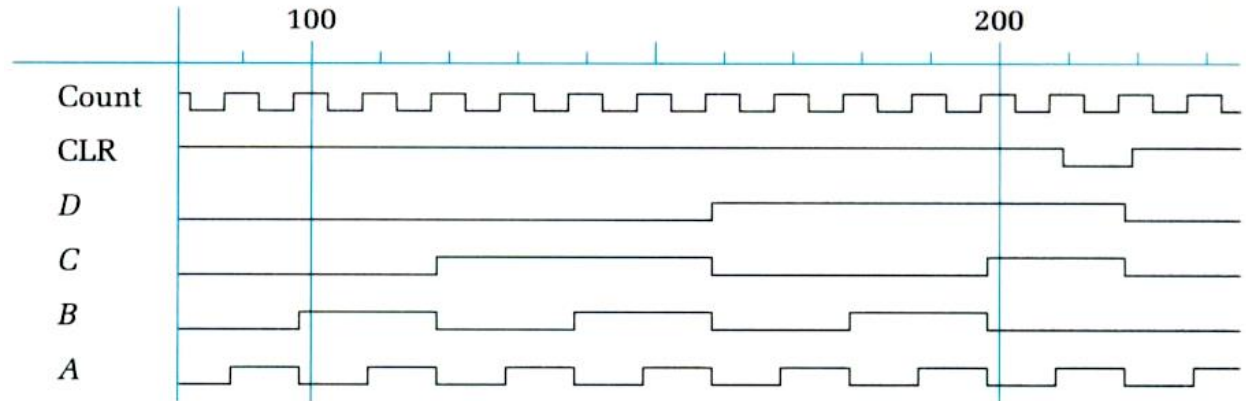
# Cutoff Limit Counter

**Ending Offset Counter:**

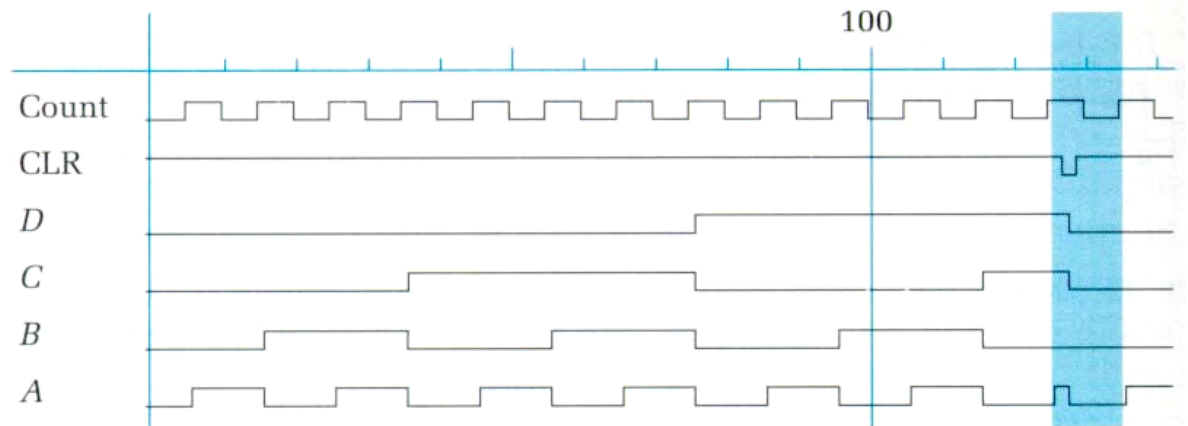
e.g., 0000, 0001, 0010, ..., 1100, 1101, 0000



**Decode state to  
determine when to  
reset to 0000**



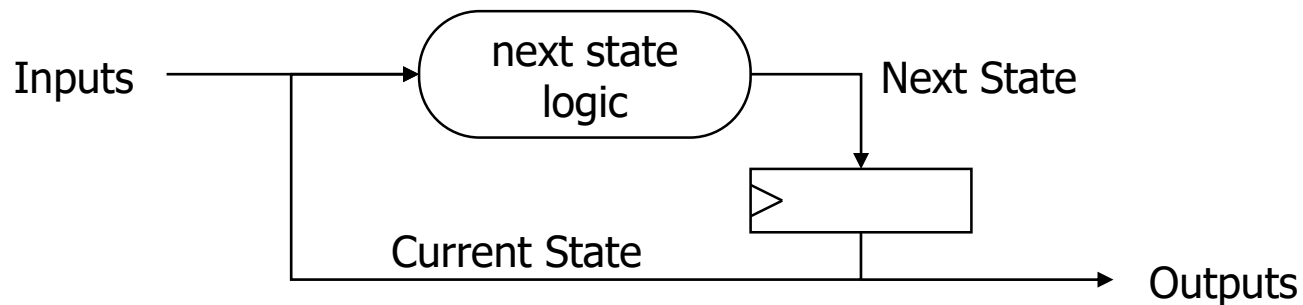
**Clear signal takes effect on the rising count edge**





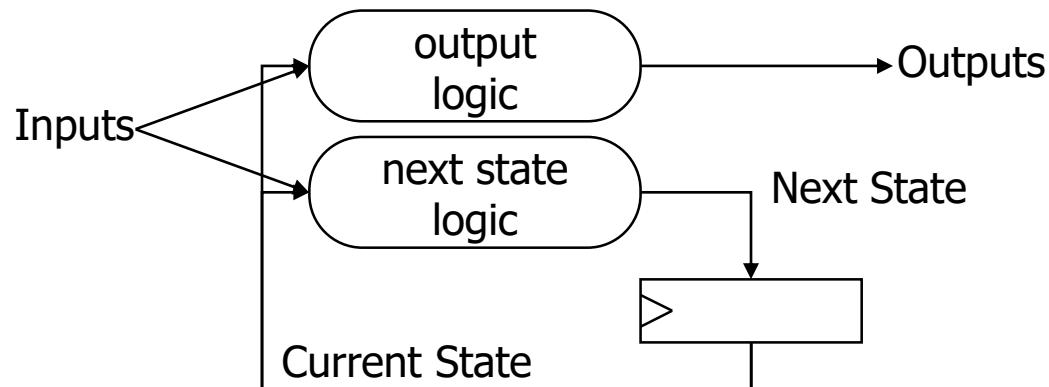
# Counter/shift-Register Model

- 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



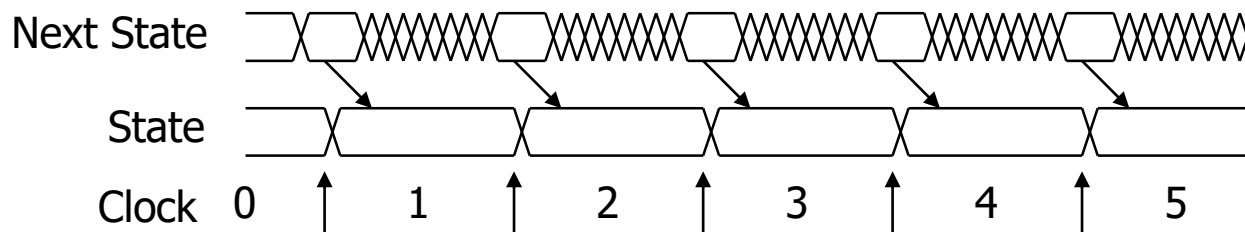
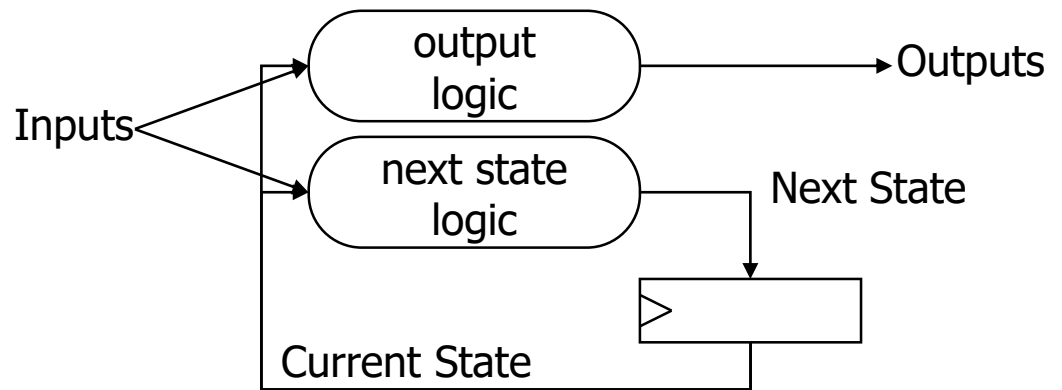
# General State Machine Model

- 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

- 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:  $F_s(S_i, I_j)$
- Output function:  $F_o(S_i)$  or  $F_o(S_i, I_j)$



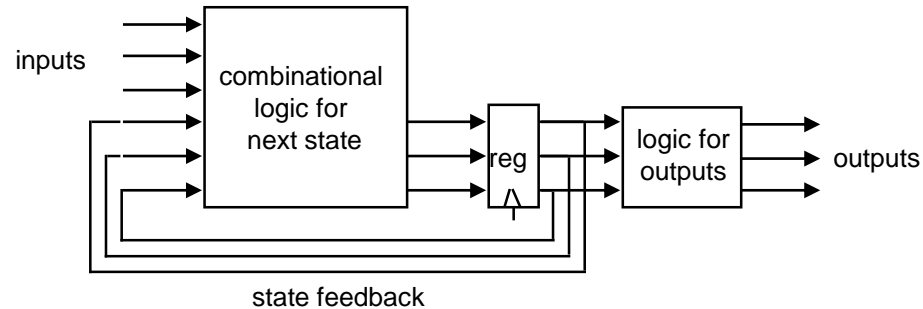
# Mealy vs. Moore Machines

---

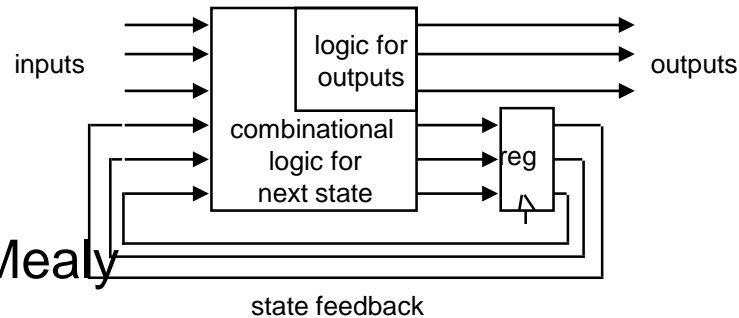
- Mealy machines tend to have less states
  - ◆ different outputs on arcs ( $n^2$ ) rather than states ( $n$ )
- Moore machines are safer to use
  - ◆ outputs change at clock edge (always one cycle later)
  - ◆ in Mealy machines, input change can cause output change as soon as logic is done – a big problem 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

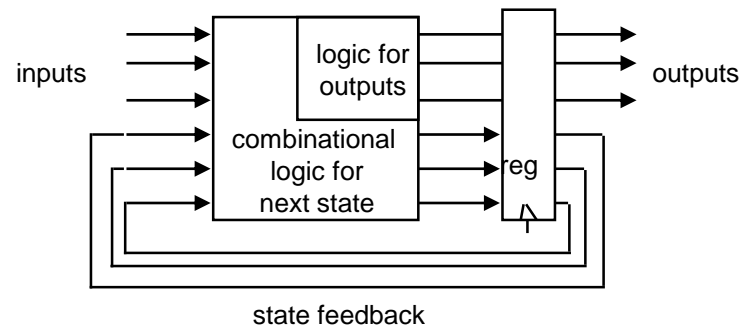
## ■ Moore



## ■ Mealy

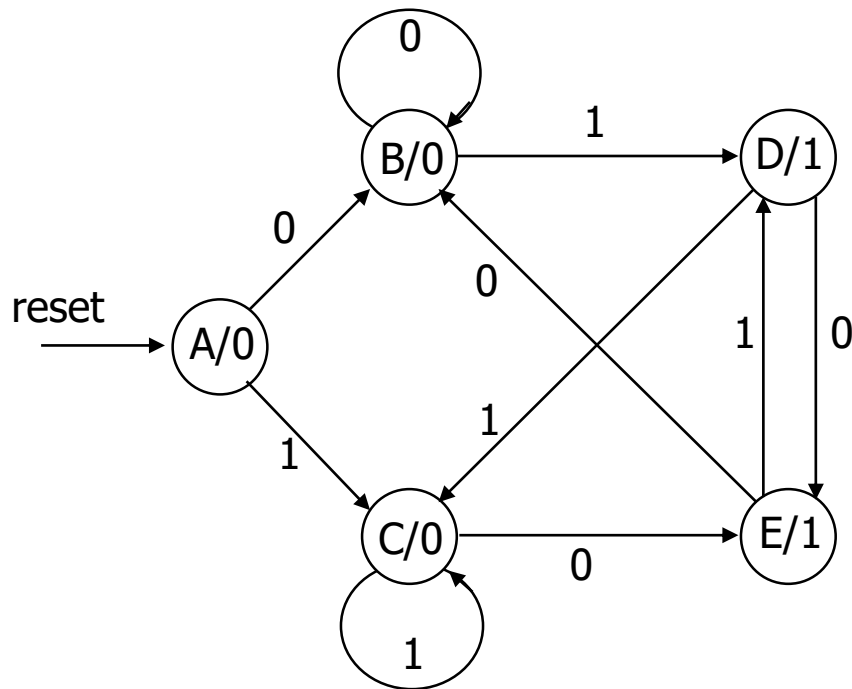


## ■ Synchronous Mealy



# Specifying Outputs for a Moore Machine

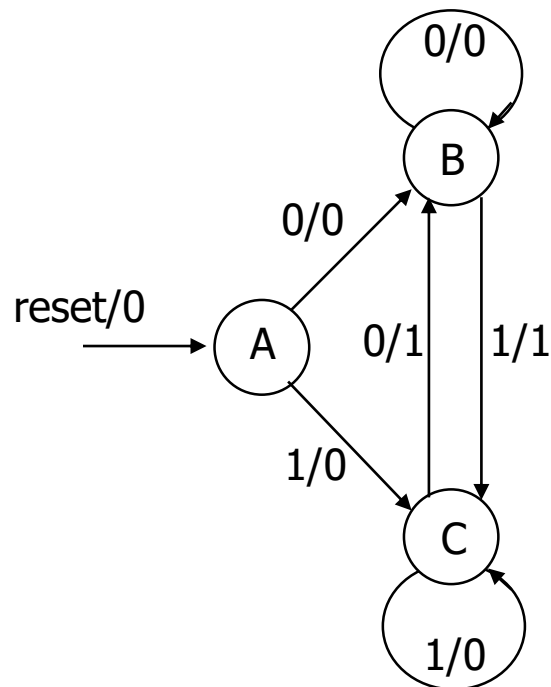
- Output is only function of state
  - ◆ specify the output in state circle in state diagram
  - ◆ example: sequence detector for 01 or 10



| 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

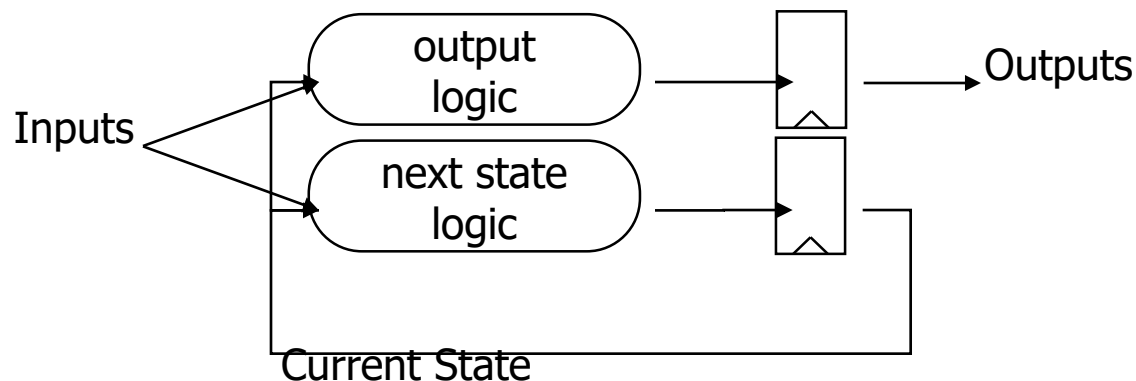


| reset | input | current state | next state | output |
|-------|-------|---------------|------------|--------|
| 1     | —     | —             | A          | 0      |
| 0     | 0     | A             | B          | 0      |
| 0     | 1     | A             | C          | 0      |
| 0     | 0     | B 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

Mealy is simpler with fewer states

# Registered Mealy Machine (really Moore)

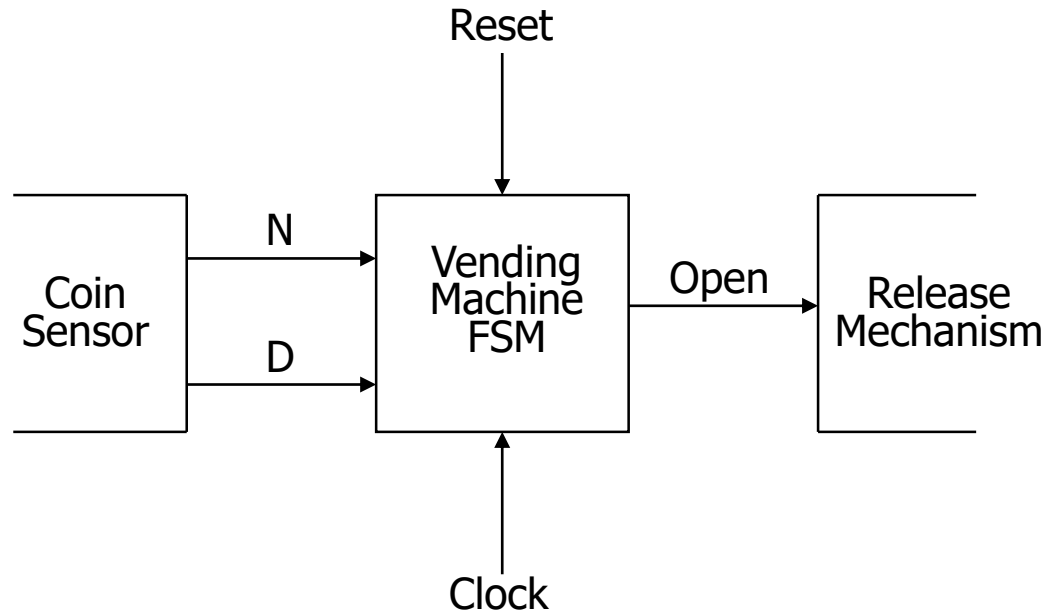
- Synchronous (or registered) Mealy machine
  - ◆ registered state AND outputs
  - ◆ avoid 'glitchy' outputs
  - ◆ easy to implement in PLDs
- Moore machine with no output decoding
  - ◆ outputs computed on transition to next state rather than after entering





# Example: Vending Machine

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



# Example: Vending Machine

- Suitable abstract representation

- ◆ tabulate typical input sequences:

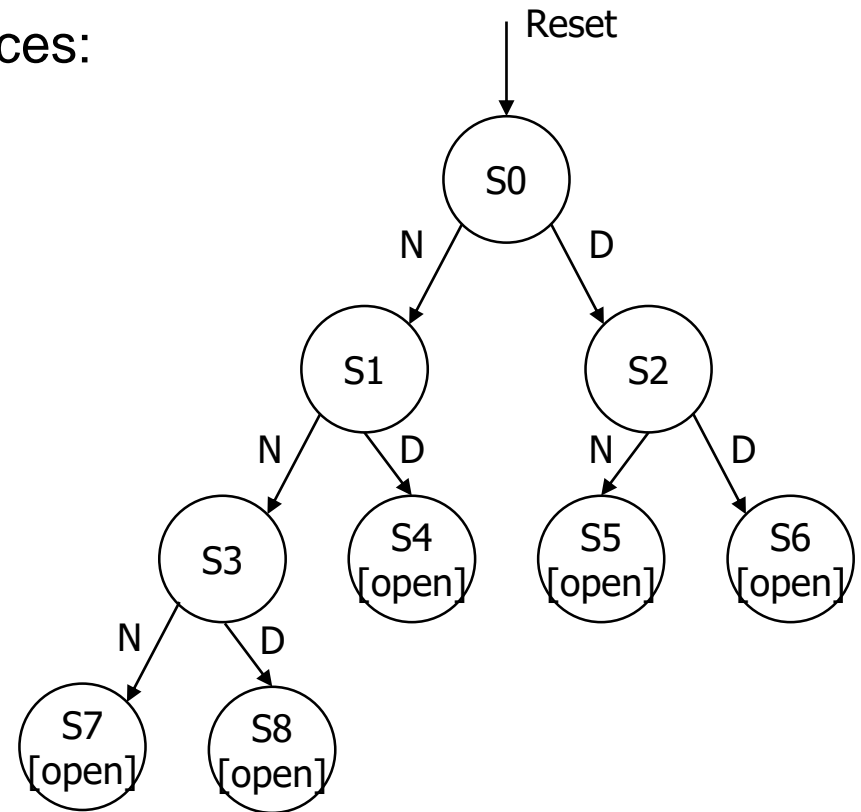
- 3 nickels
    - nickel, dime
    - dime, nickel
    - two dimes

- ◆ draw state diagram:

- inputs: N, D, reset
    - output: open signal

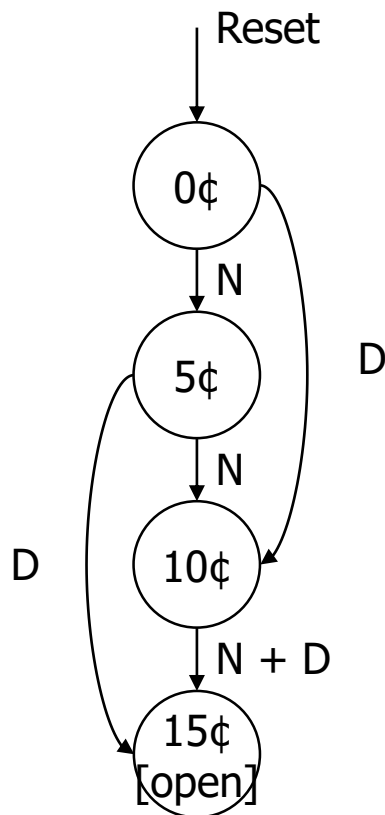
- ◆ assumptions:

- assume N or D is asserted for one cycle
    - each state has a self loop for  $N = D = 0$  (no coin)



# Example: Vending Machine

- 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

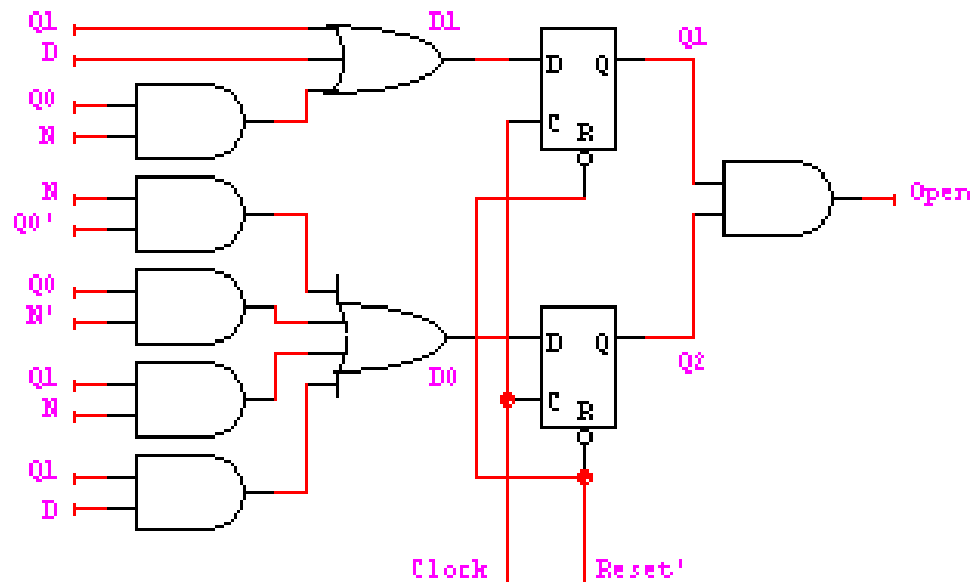
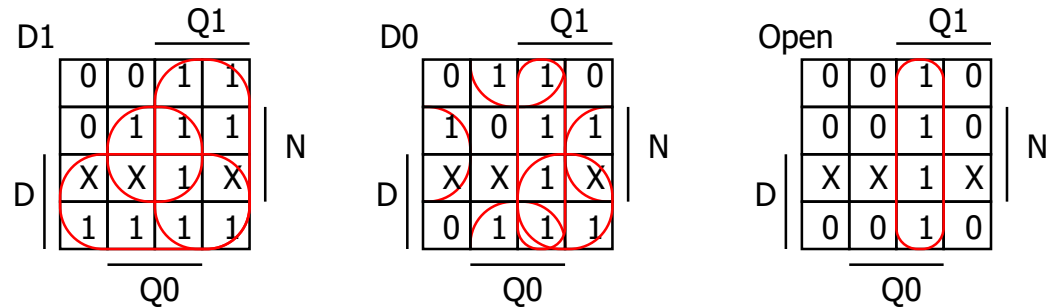
# Example: Vending Machine

- Uniquely encode states

| 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  | 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



# Example: Vending Machine

- One-hot encoding: only one bit is assigned as true for each state (4 bits)

| 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    |

$$D0 = Q0 D' N'$$

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

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

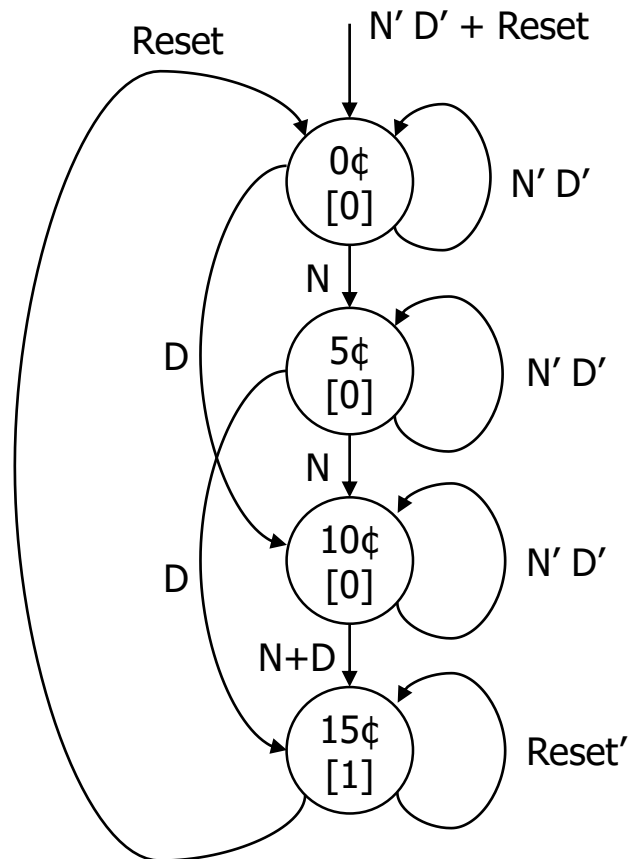
$$D3 = Q1 D + Q2 D + Q2 N + Q3$$

$$OPEN = Q3$$

# Equivalent Mealy & Moore State Diagrams

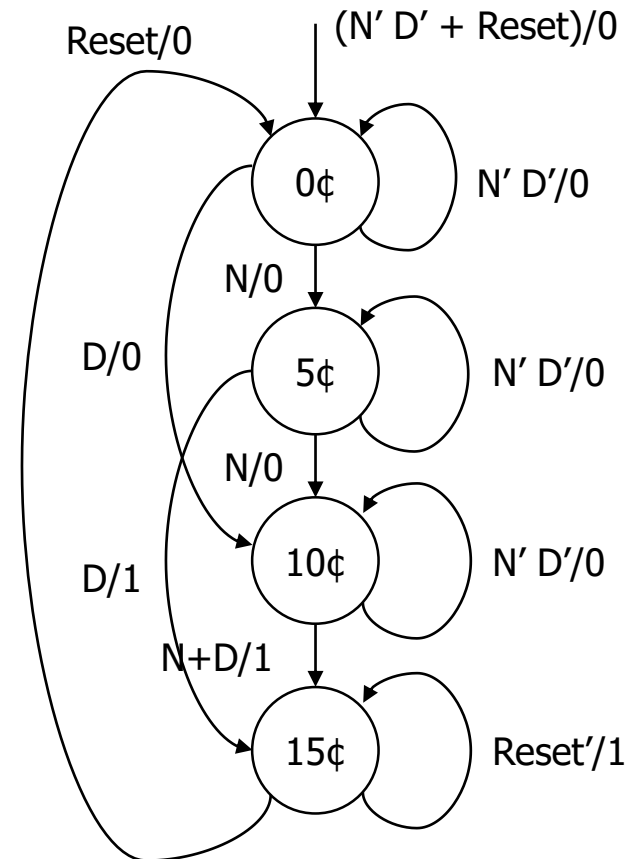
## ■ Moore machine

- ◆ outputs associated with state

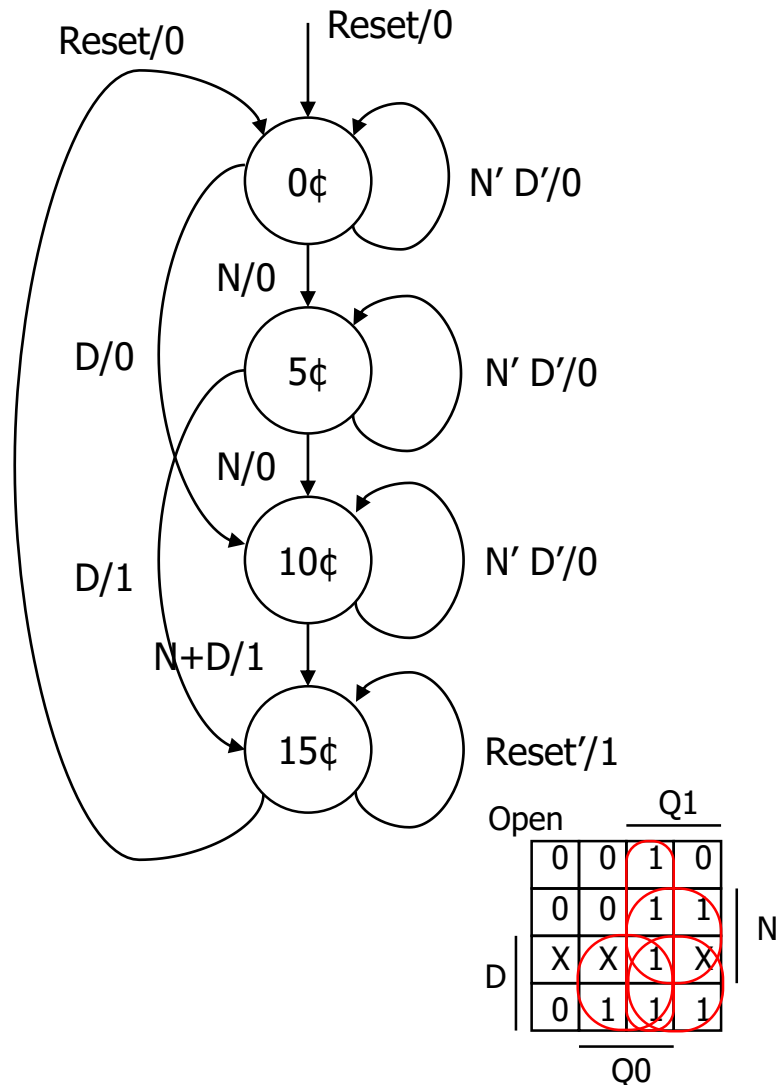


## ■ Mealy machine

- ◆ outputs associated with transitions



# Example: Mealy Implementation



| 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      |

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

$$D1 = Q1 + D + Q0N$$

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



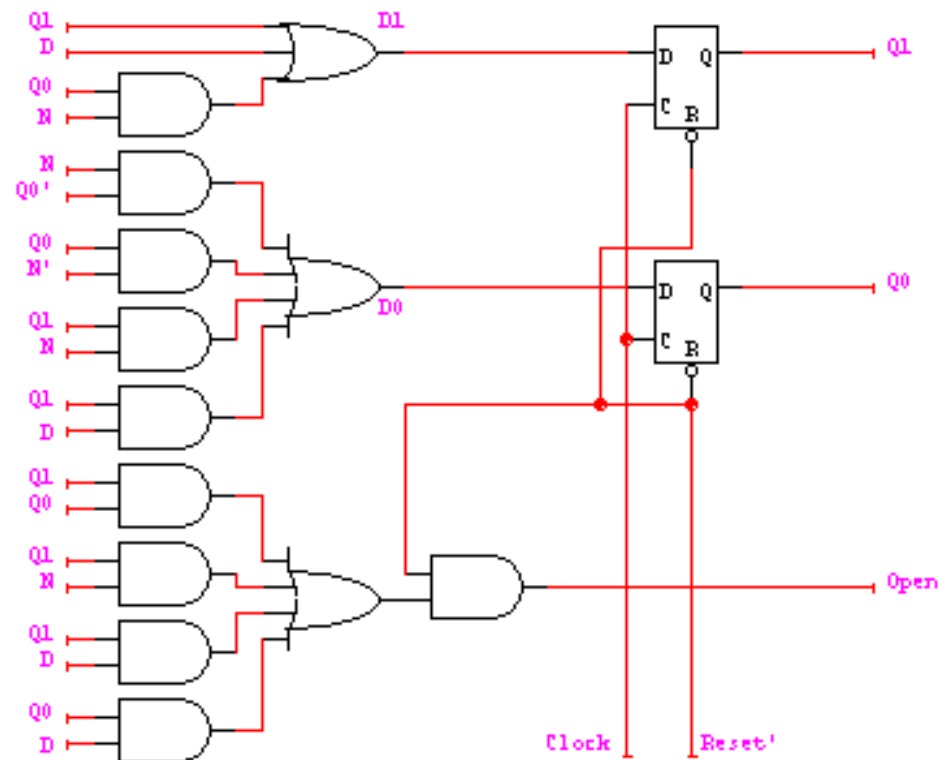
# Example: Mealy Implementation

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

$$D1 = Q1 + D + Q0N$$

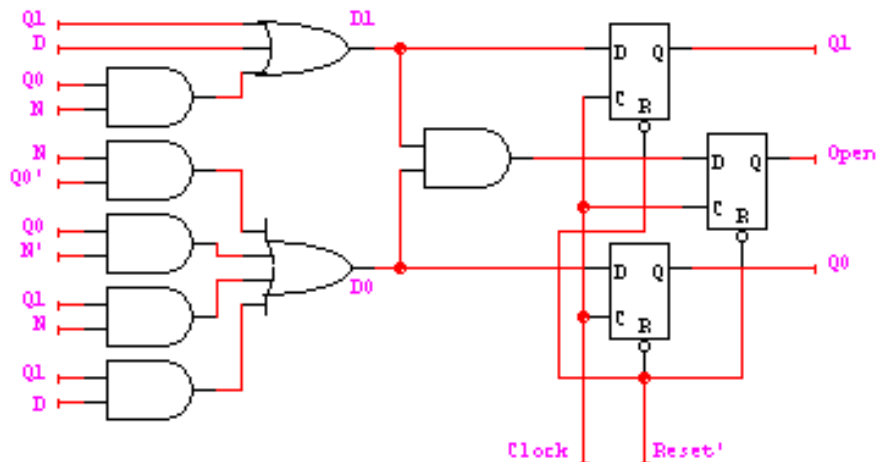
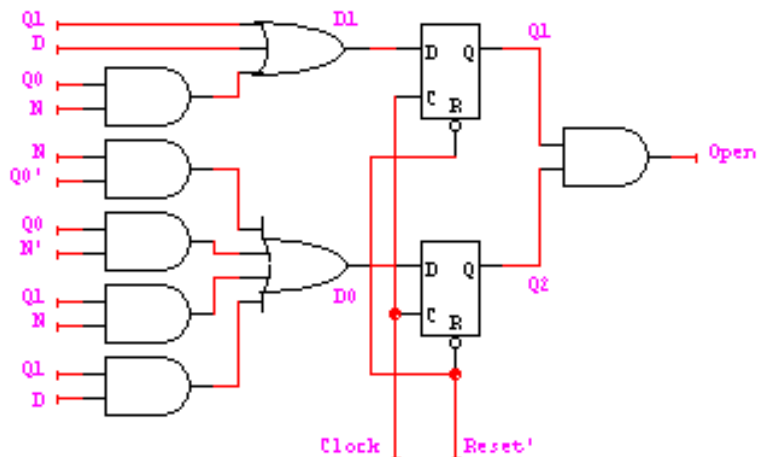
$$OPEN = Q1Q0 + Q1N + Q1D + Q0D$$

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



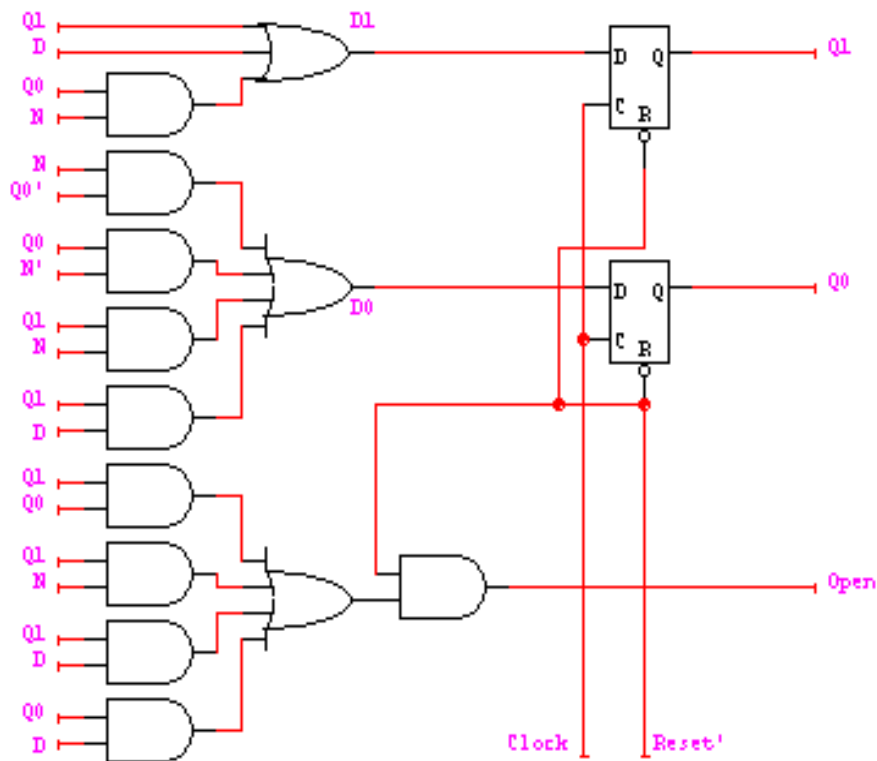
# Vending Machine: Moore to synch. Mealy

- 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 from state output side to state input side to improve delay
- $OPEN.d = (Q1 + D + Q0N)(Q0'N + Q0N' + Q1N + Q1D)$   
 $= Q1Q0N' + Q1N + Q1D + Q0'ND + Q0N'D$
- 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

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

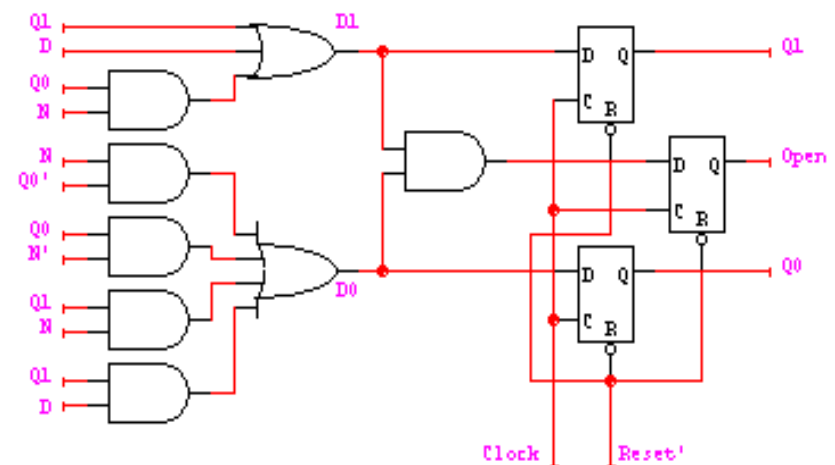


Open.d

|   | Q1 |   |
|---|----|---|
|   | 0  | 1 |
| D | 0  | 0 |
|   | 0  | 1 |
|   | 1  | 0 |
|   | 1  | 1 |
|   | Q0 |   |

Open.d

|   | Q1 |   |
|---|----|---|
|   | 0  | 1 |
| D | 0  | 0 |
|   | 0  | 1 |
|   | X  | X |
|   | 0  | 1 |
|   | Q0 |   |



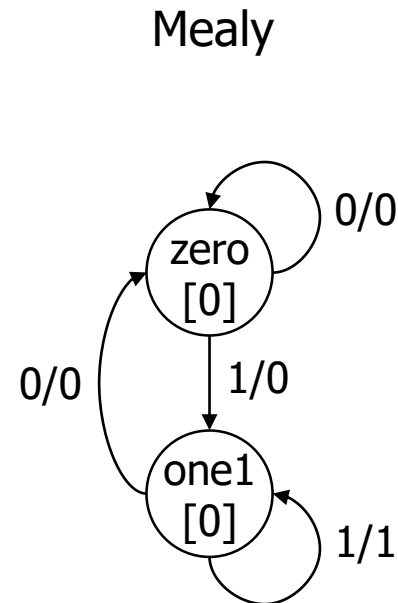
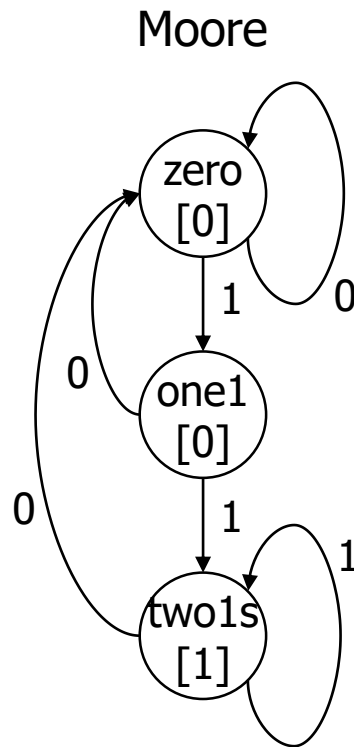
# HDL and Sequential Logic

---

- Flip-flops
  - ◆ representation of clocks - timing of state changes
  - ◆ asynchronous vs. synchronous
- FSMs
  - ◆ structural view (FFs separate from combinational logic)
  - ◆ behavioral view (synthesis of sequencers – not in this course)
- Data-paths = data computation (e.g., ALUs, comparators) + registers
  - ◆ use of arithmetic/logical operators
  - ◆ control of storage elements

# Example: Reduce-1-String-by-1

- Remove one 1 from every string of 1s on the input

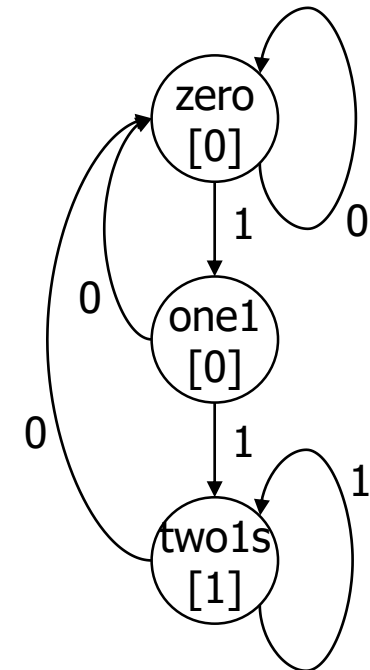


# Verilog FSM - Reduce 1s Example

## ■ Moore machine

```
module reduce (clk, reset, in, out);  
  input clk, reset, in;  
  output out;  
  
  parameter zero  = 2'b00;  
  parameter one1  = 2'b01;  
  parameter two1s = 2'b10;  
  
  reg out;  
  reg [2:1] state;      // state variables  
  reg [2:1] next_state;  
  
  always @(posedge clk)  
    if (reset) state = zero;  
    else      state = next_state;
```

state assignment  
(easy to change,  
if in one place)

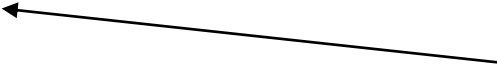


# Moore Verilog FSM


```
always @(in or state)
```

```
    case (state)
      zero:
        // last input was a zero
        begin
          if (in) next_state = one1;
          else    next_state = zero;
        end
      one1:
        // we've seen one 1
        begin
          if (in) next_state = twos;
          else    next_state = zero;
        end
      twos:
        // we've seen at least 2 ones
        begin
          if (in) next_state = twos;
          else    next_state = zero;
        end
    endcase
```

crucial to include  
all signals that are  
input to state determination



note that output  
depends only on state



```
always @(state)
  case (state)
    zero: out = 0;
    one1: out = 0;
    twos: out = 1;
  endcase
```

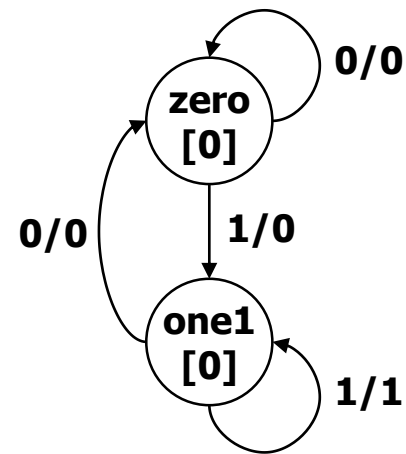
```
endmodule
```

# Mealy Verilog FSM

```
module reduce (clk, reset, in, out);
  input clk, reset, in;
  output out;
  reg out;
  reg state; // state variables
  reg next_state;

  always @(posedge clk)
    if (reset) state = zero;
    else      state = next_state;

  always @(in or state)
    case (state)
      zero:           // last input was a zero
      begin
        out = 0;
        if (in) next_state = one;
        else   next_state = zero;
      end
      one:            // we've seen one 1
      if (in) begin
        next_state = one; out = 1;
      end else begin
        next_state = zero; out = 0;
      end
    endcase
endmodule
```





# Synchronous Mealy Machine

---

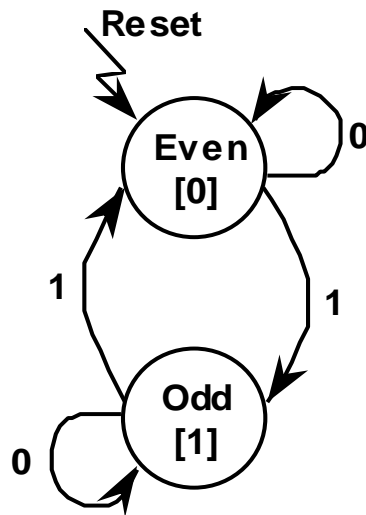
```
module reduce (clk, reset, in, out);
    input clk, reset, in;
    output out;
    reg out;
    reg state; // state variables

    always @(posedge clk)
        if (reset) state = zero;
        else
            case (state)
                zero:      // last input was a zero
                begin
                    out = 0;
                    if (in) state = one;
                    else    state = zero;
                end
                one:       // we've seen one 1
                if (in) begin
                    state = one; out = 1;
                end else begin
                    state = zero; out = 0;
                end
            endcase
endmodule
```

# Finite State Machine Word Problems (1)

## ■ Example: Odd Parity Checker

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



State  
Diagram

| 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

# Finite State Machine Word Problems (1)

## ■ Example: Odd Parity Checker

**NS** is input of D FF, **OUT** is output of D FF

**NS = PS xor PI; OUT = PS**

|    |   | PS |   |
|----|---|----|---|
|    |   | 0  | 1 |
| PI | 0 | 0  | 1 |
|    | 1 | 1  | 0 |

NS

$$NS = PIPS' + PI'PS$$

|    |   | PS |   |
|----|---|----|---|
|    |   | 0  | 1 |
| PI | 0 | 0  | 1 |
|    | 1 | 0  | 1 |

OUT = PS

# Finite State Machine Word Problems

---

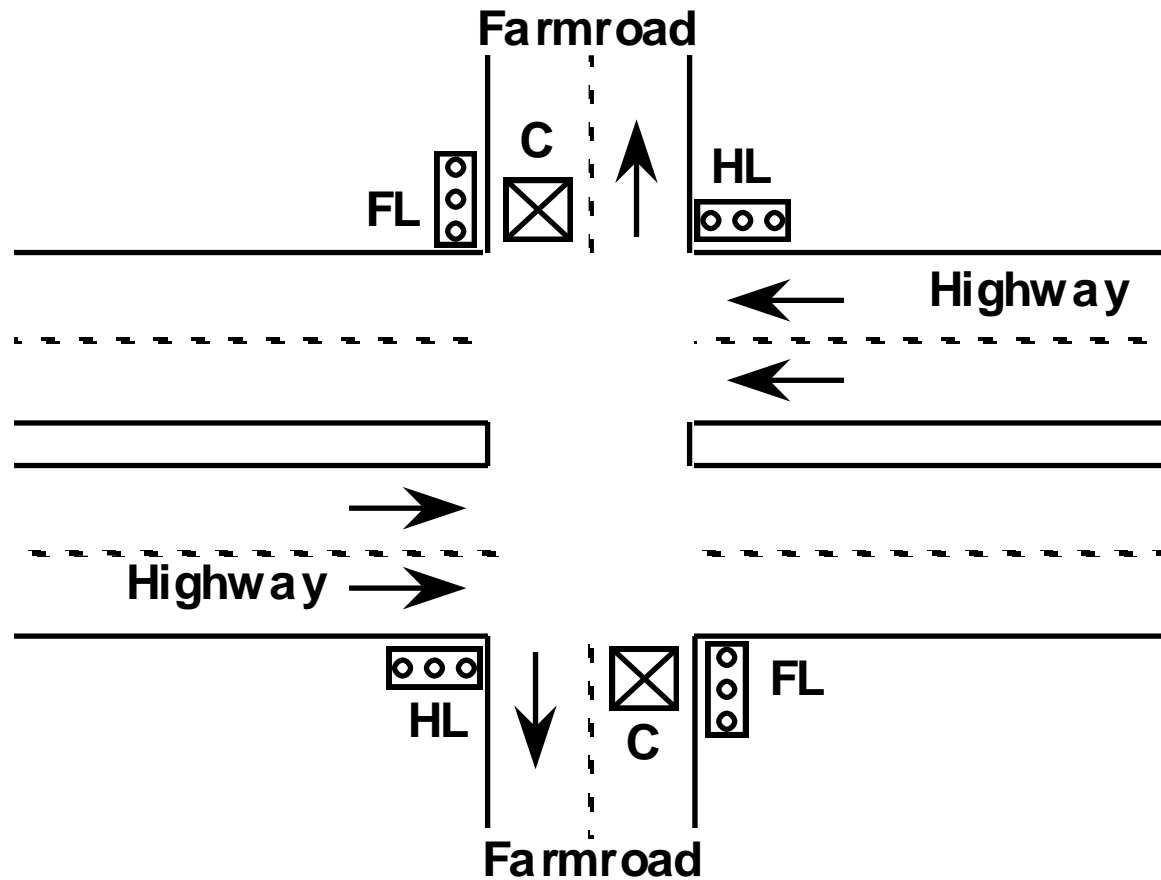
## ■ Traffic Light Controller

A busy highway is intersected by a little used farmroad. Detectors C sense the presence of cars waiting on the farmroad. With no car on farmroad, light remain green in highway direction. *If vehicle on farmroad, highway lights go from Green to Yellow to Red*, allowing the farmroad lights to become green. *These stay green only as long as a farmroad car is detected* but never longer than a set interval. When these are met, farm lights transition from Green to Yellow to Red, allowing highway to return to green. *Even if farmroad vehicles are waiting, highway gets at least a set interval as green.*

Assume you have an interval timer that generates a short time pulse (TS) and a long time pulse (TL) in response to a set (ST) signal. TS is to be used for timing yellow lights and TL for green lights.

# Finite State Machine Word Problems

Picture of Highway/Farmroad Intersection:



# Finite State Machine Word Problems

## ■ Traffic Light Controller

### • Tabulation of Inputs and Outputs:

#### *Input Signal*

**reset**

**C**

**TS**

**TL**

#### *Description*

place FSM in initial state

detect vehicle on farmroad

short time interval expired

long time interval expired

#### *Output Signal*

**HG, HY, HR**

**FG, FY, FR**

**ST**

#### *Description*

assert green/yellow/red highway lights

assert green/yellow/red farmroad lights

start timing a short or long interval

### • Tabulation of Unique States: Some light configuration imply others

#### *State*

**S0**

**S1**

**S2**

**S3**

#### *Description*

Highway green (farmroad red)

Highway yellow (farmroad red)

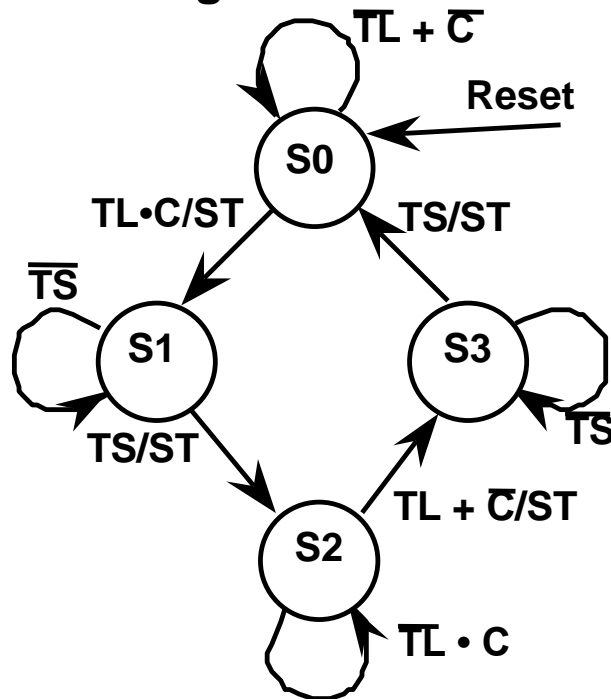
Farmroad green (highway red)

Farmroad yellow (highway red)

# Finite State Machine Word Problems

## ■ Traffic Light Controller

Compare with state diagram:



**S0: HG**

**S1: HY**

**S2: FG**

**S3: FY**

# 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
- Hardware description languages