

# **FINITE STATE MACHINES**

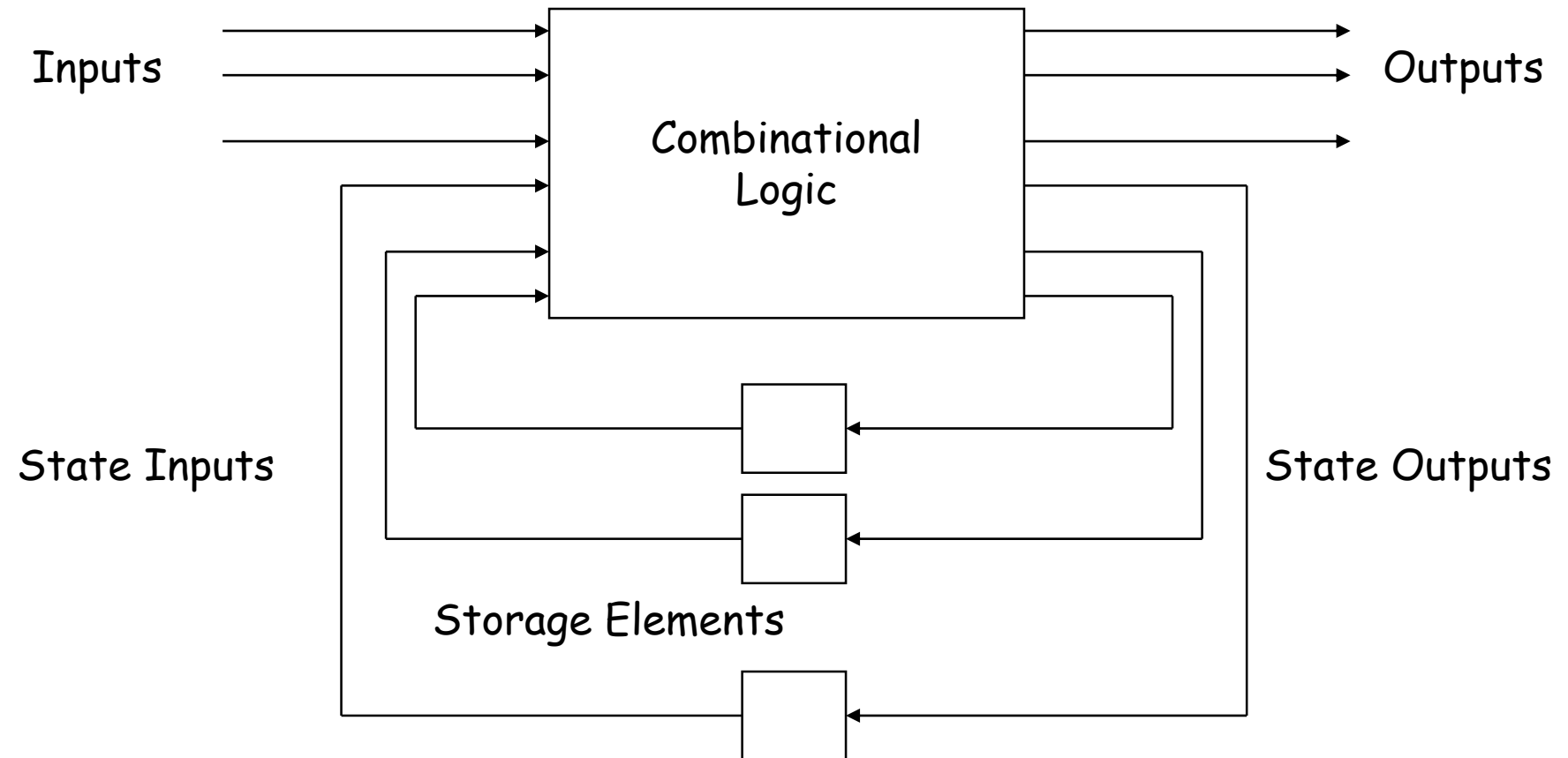
# Sequential Logic Implementation

- Models for representing sequential circuits
  - Finite-state machines (Moore and Mealy)
  - Representation of memory (states)
  - Changes in state (transitions)
- Design procedure
  - State diagrams
  - State transition table
  - Next state functions

# Abstraction of State Elements

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

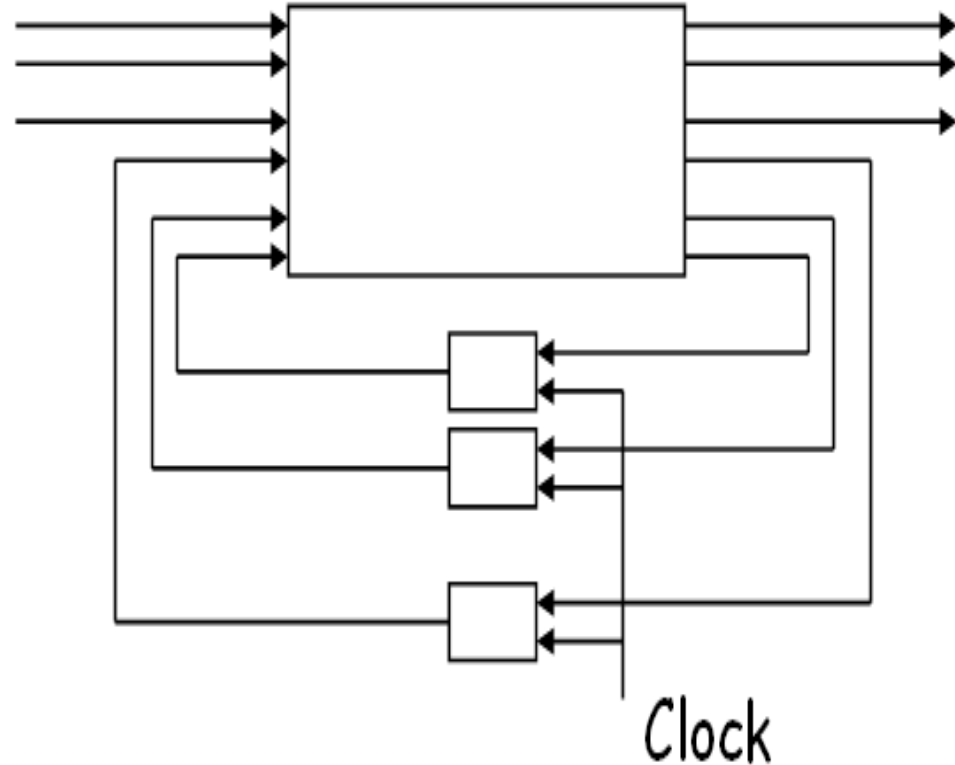
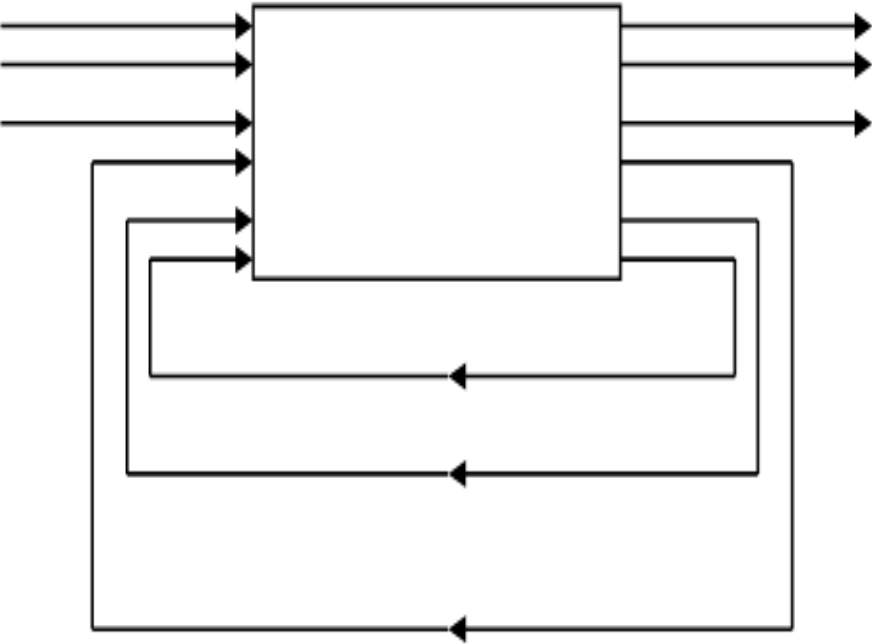
# Abstraction of State Elements



# 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 locked steps across all storage elements (using a periodic waveform - the clock)

# Forms of Sequential Logic



# Finite State Machine

- A generic model for sequential circuits used in sequential circuit design

# Finite State Machine - Definition

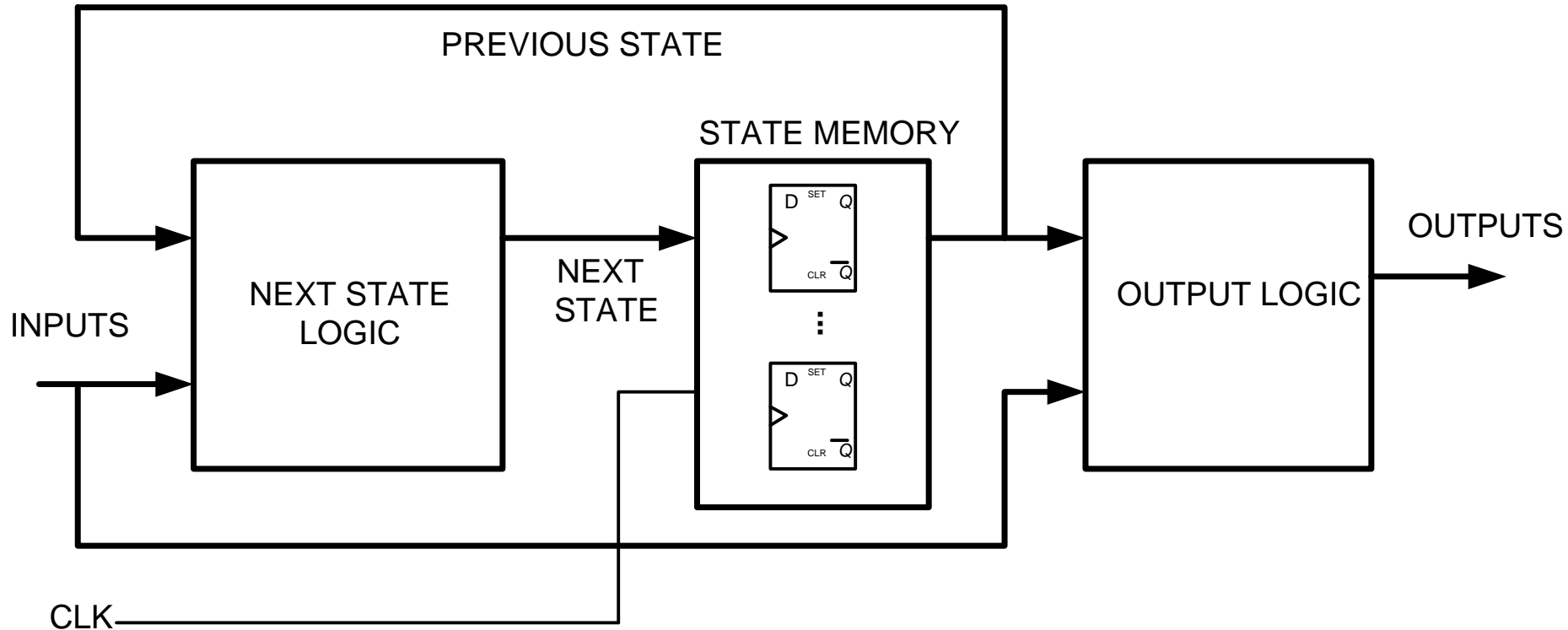
- FSA – Finite State Automation
- Models of a behavior of a system or a complex object, with a limited number of defined conditions or modes.
- Four Elements:
  - States – define behavior and may produce actions.
  - State Transitions – movement from one state to another.
  - Rules/Conditions – must be met to allow a state transition.
  - Input Events – Externally or internally generated, may possibly trigger rules and lead to state transitions.



# A Finite State Machine

- A sequential logic unit which
  - Takes an input and a current state
  - Produces an output and a new state
- It is called a Finite State Machine because it can have, at most, a finite number of states.
- It is composed of a combinational logic unit and flip-flops placed in such a way as to maintain state information.

# Finite state machine block diagram



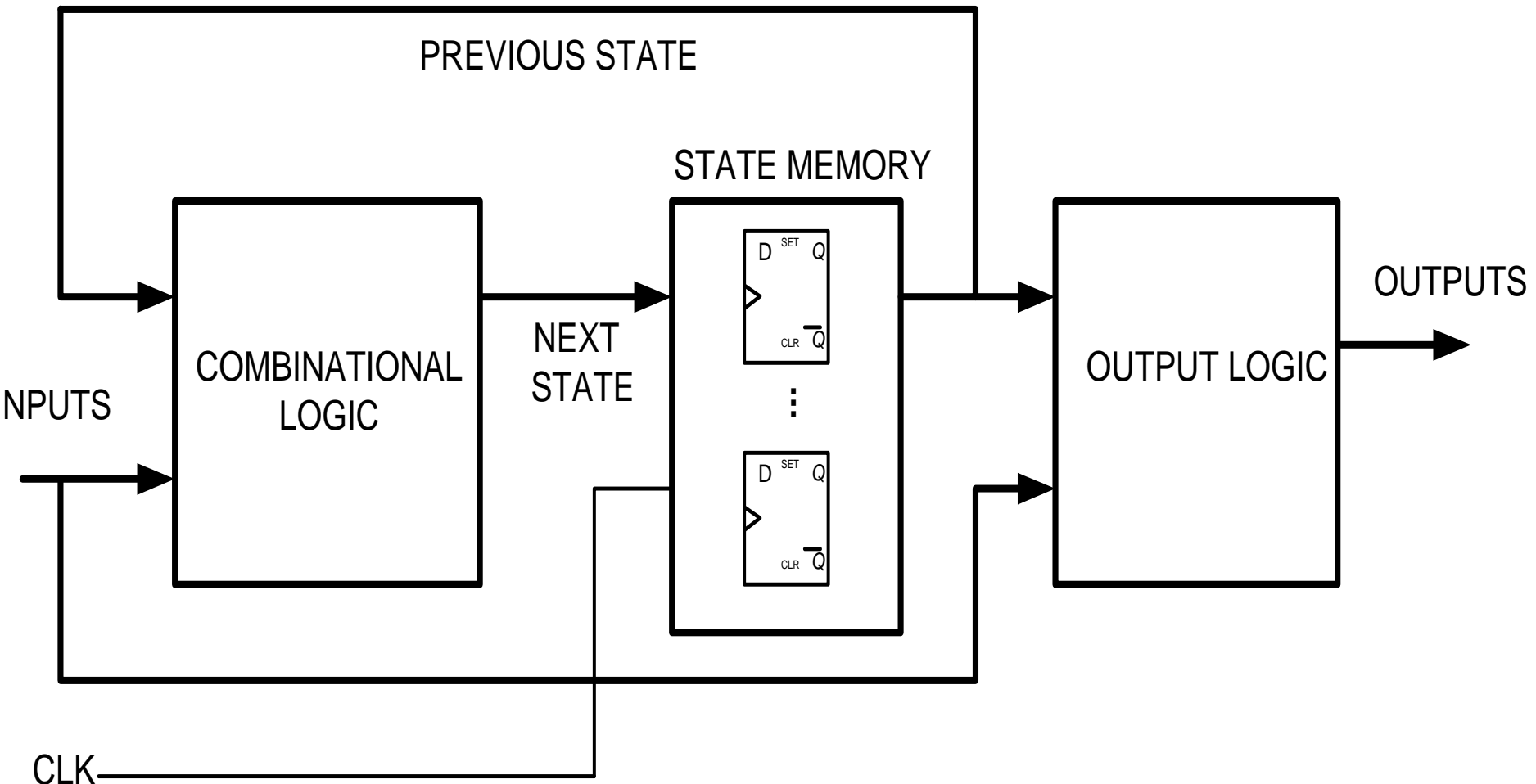
# Finite state machine block diagram

- State memory: Set of  $n$  flip-flops that hold the state of the machine (up to  $2^n$  distinct states)
- Next state logic: Combinational circuit that determines the next state as a function of the current state and the input
- Output logic: Combinational circuit that determines the output as a function of the current state and the input

# Finite State Machine types:

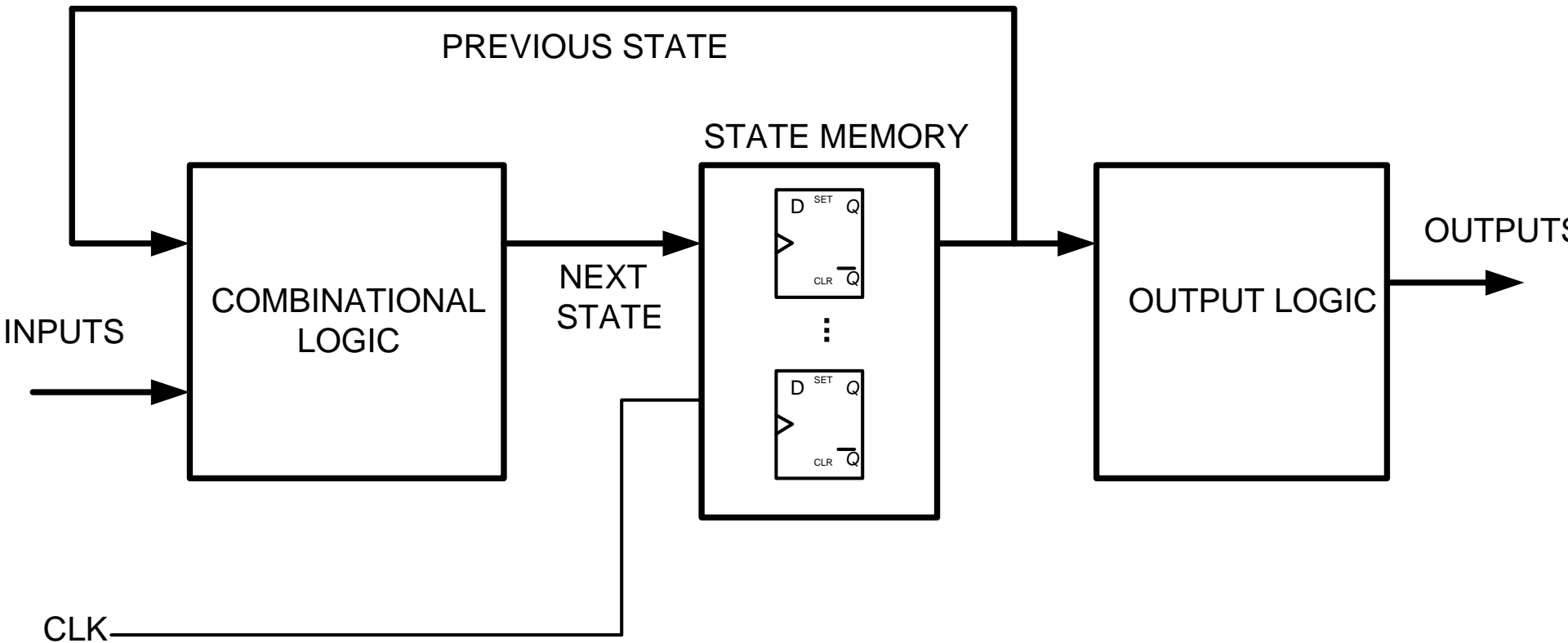
## Mealy machine

- The output depends on the current state and input



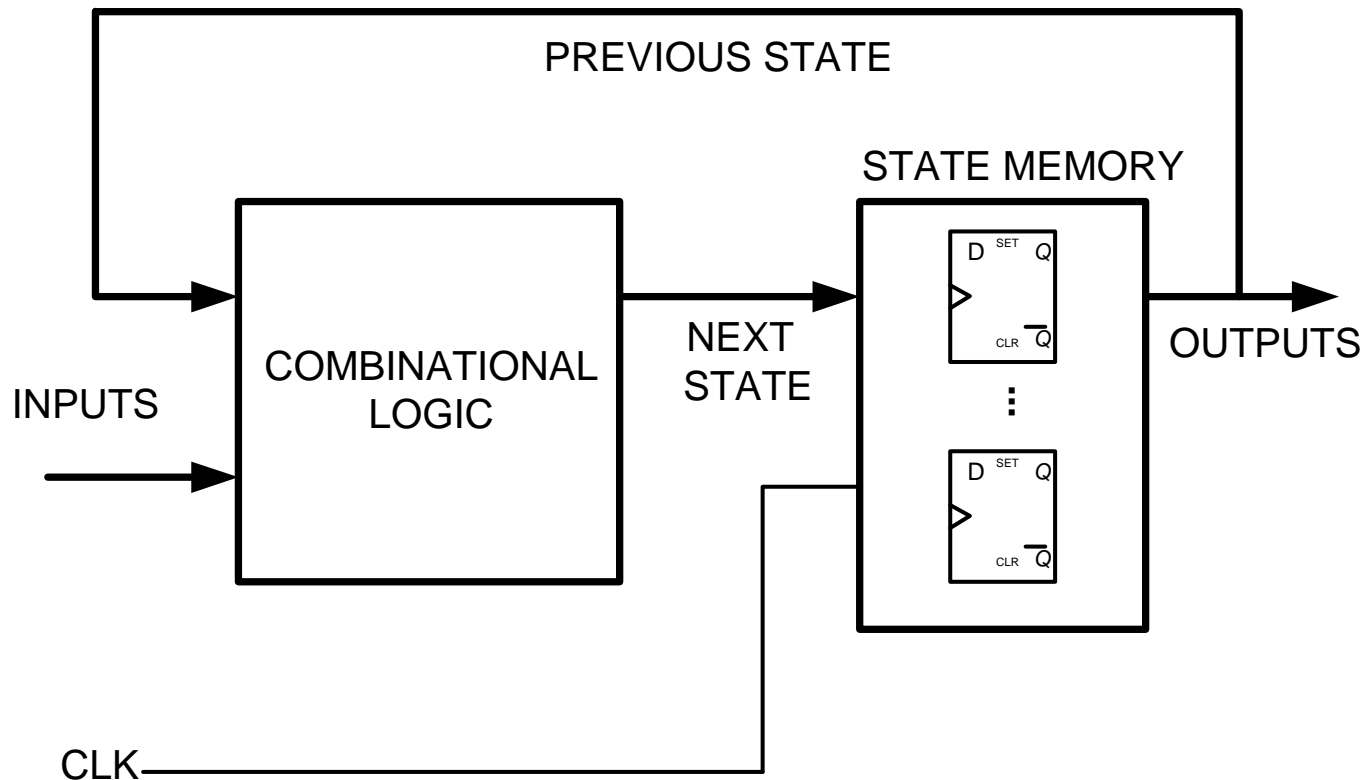
# Moore Machine

Moore machine: The output depends only on the current state



# State

- State = output state machine: A Moore type FSM where the current state **is** the output



# DFSM and nonDFSM

- **FSM is typically used as a type of control system where knowledge is represented in the states, and actions are constrained by rules.**
- **An adopted AI technique initially used for language representation.**
- **Two main types of FSM.**
- **Deterministic FSM, meaning that given an input and the current state, the state transition can be predicted.**
- **Non-deterministic finite state machine. This is where given the current state; the state transition is not predictable. It may be the case that multiple inputs are received at various times, means the transition from the current state to another state cannot be known until the inputs are received (event driven).**
- **Non deterministic FSM example : Use a random number generator to select a triggered rule. Adds unpredictability. ( Example : MoveUnit action may be both EvadeEnemy state and AttackEnemy state ). [ State may involve one or more actions ].**

# Fuzzy State Machines

- In AI and computer games *“a player feels like they are playing against a realistic simulation of intelligence, and not against a reproduction of a sequence of actions.”*
- The "sequence" which is one of the key benefits of FSM, should not be blindingly obvious in computer games. There are a number of extensions to FSM and workarounds for "mixing up" the sequence to make it harder to predict actions. One of these non-deterministic approaches involves the application of another proven artificial intelligence technique; Fuzzy Logic, called Fuzzy State Machines (FuSM).
- A fuzzy value can be applied to various state transitions. When a conflict set is encountered the higher the fuzzy value for a transition, the higher the likelihood of the state transition.
- An implementation of FuSM may involve the assignment of fuzzy values to various inputs to represent the degree an input is defined. The fuzzy system would use these weighted input values in the evaluation of rules, triggering only state transitions whose assessed value is above a specified threshold.



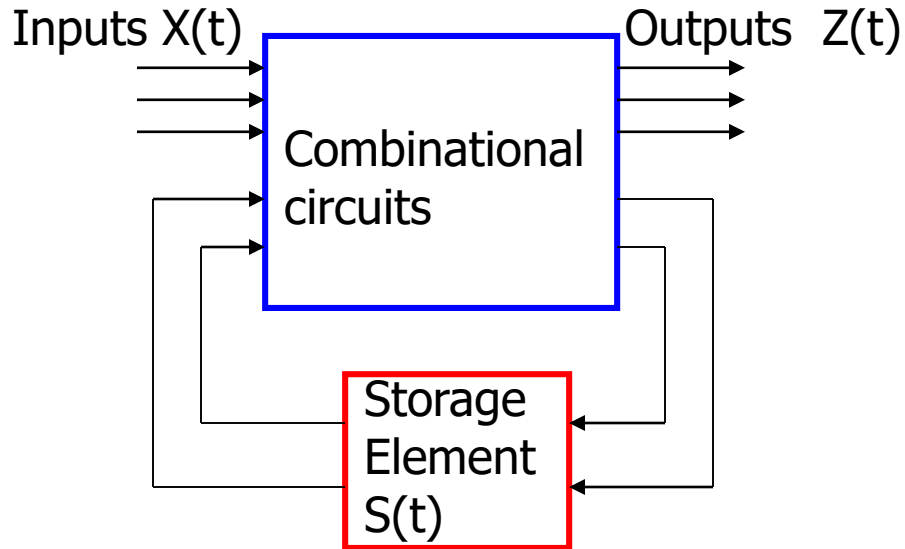
- A Finite State Machine (FSM) is simply a state register that holds the current state and some combinational logic which calculates the next state and outputs based on the current state and the inputs
- FSM types
  - Moore machine : the outputs are functions of the present state only
  - Mealy machine : the outputs are functions of both the present state and the inputs

# Moore and Mealy Machines

- Both these machine types follow the basic characteristics of state machines, but differ in the way that outputs are generated.
- **Moore Machine:**
  - Outputs are independent of the inputs, ie outputs are effectively produced from within the state of the state machine.
- **Mealy Machine:**
  - Outputs can be determined by the present state alone, or by the present state and the present inputs, ie outputs are produced as the machine makes a transition from one state to another.

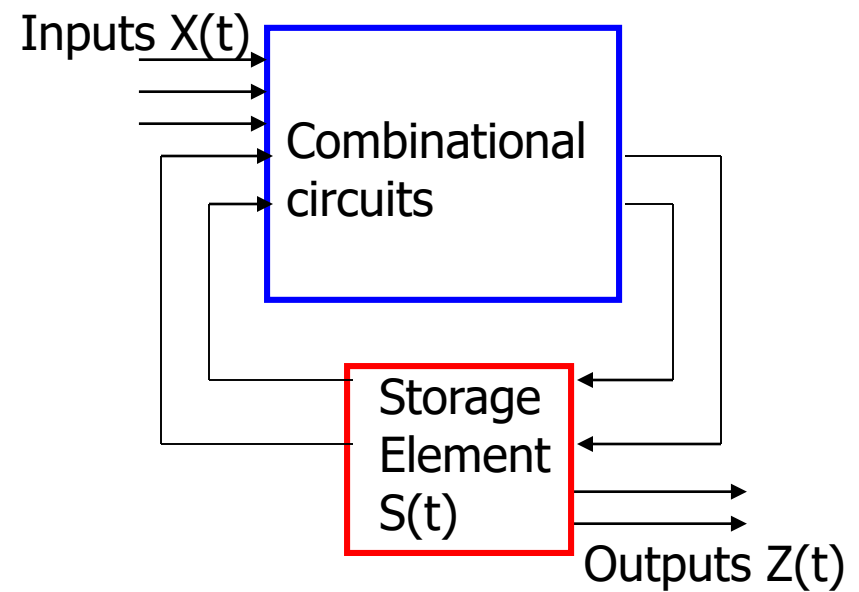
# Mealy and Moore Machines<sup>19</sup>

## MEALY MACHINE



$$Z(t) = \lambda\{S(t), X(t)\}$$

## MOORE MACHINE



$$Z(t) = \lambda\{S(t)\}$$

# Moore vs. Mealy FSM (1)

- Moore and Mealy FSMs Can Be Functionally Equivalent
- Mealy FSM Has Richer Description and Usually Requires Smaller Number of States
  - Smaller circuit area

# Moore vs. Mealy FSM (2)

- Mealy FSM Computes Outputs as soon as Inputs Change
  - Mealy FSM responds one clock cycle sooner than equivalent Moore FSM
- Moore FSM Has No Combinational Path Between Inputs and Outputs
  - Moore FSM is less likely to have a shorter critical path

# FSM Advantages

- Simple
- Predictable ( deterministic FSM ) - given a set of inputs and a known current state, the state transition can be predicted, allowing for easy testing.
- Due to their simplicity, FSMs are quick to design, quick to implement and quick in execution.
- FSM is an old knowledge representation and system modeling technique, and its been around for a long time, as such it is well proven even as an artificial intelligence technique, with lots of examples to learn from.
- Easy to transfer from a meaningful abstract representation to a coded implementation.

# FSM Disadvantages

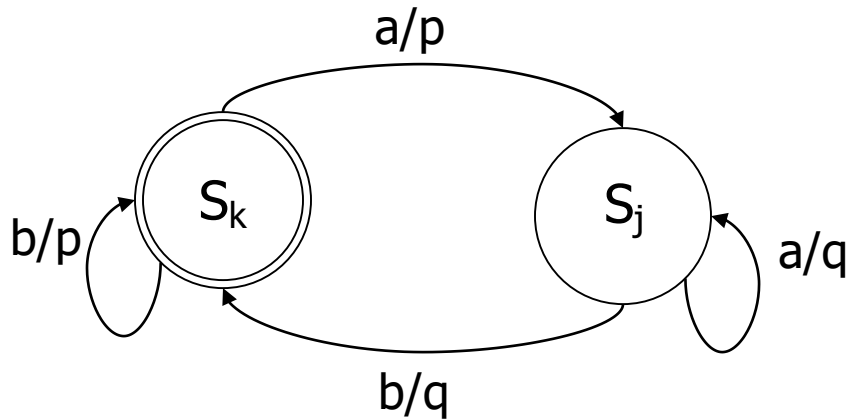
- The predictable nature of deterministic FSMs can be unwanted in some domains such as computer games (non-DFSM tries to solve this).
- The conditions for state transitions are ridged, meaning they are fixed.
- Not suited to all problem domains, should only be used when a systems behavior can be decomposed into separate states with well defined conditions for state transitions. This means that all states, transitions and conditions need to be known up front and be well defined !!

# State and State Diagram<sup>24</sup>

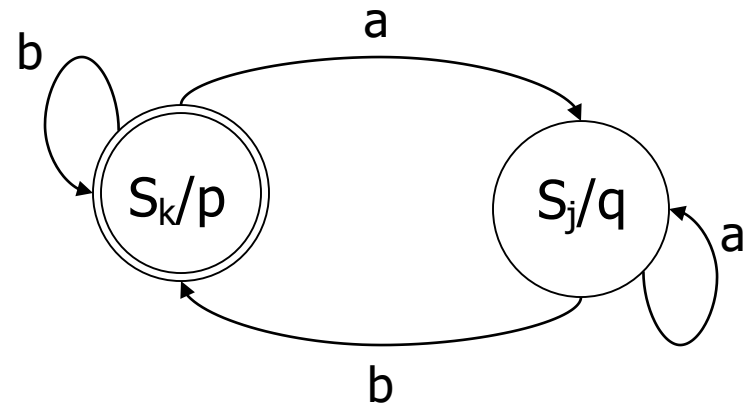
- A **state** represents the machine snapshot at a given clock period
- A clock is typically used to **synchronize** the state transition
- A graph consists of a set of
  - Circles:
    - Each represents a **state**
    - Use double circle to represent the **initial state**
  - Directed arc: each represents a state transition
  - Inputs/outputs
- Mealy machine:
  - Label **input/output** along each arc
- Moore machine:
  - Label **input** along each arc
  - Label **output** inside the circle (i.e. state)



# State Diagrams



A Mealy machine example



A Moore machine example

## Example:

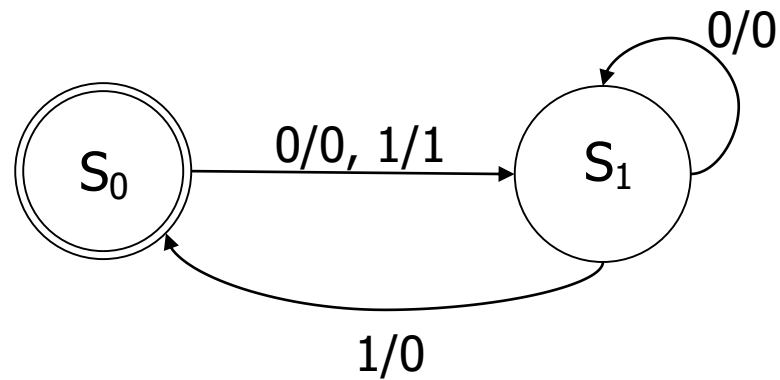
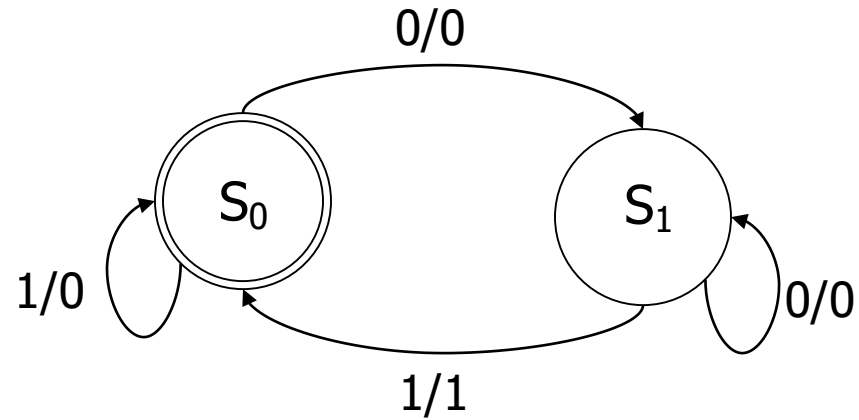
State:  $S(t) \in \{S_k, S_j\}$

Inputs:  $X(t) \in \{a, b\}$

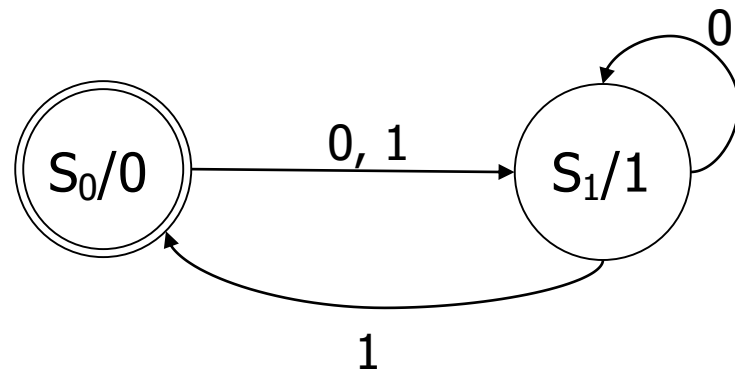
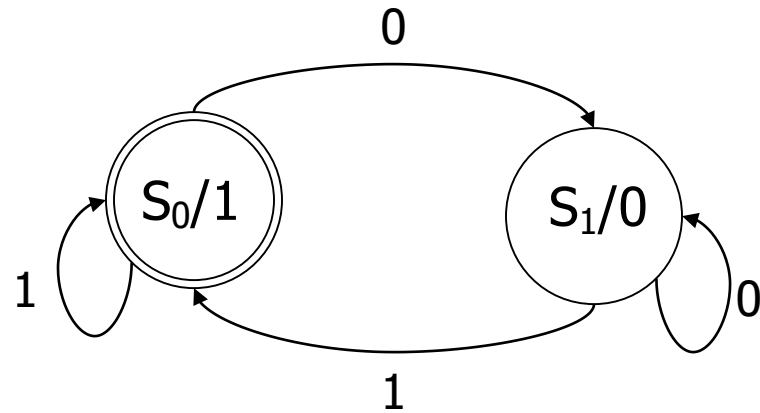
Outputs:  $Z(t) \in \{p, q\}$

Initial state:  $S(0) = S_k$

# State Diagram Examples (Mealy)



# State Diagram Examples (Moore)



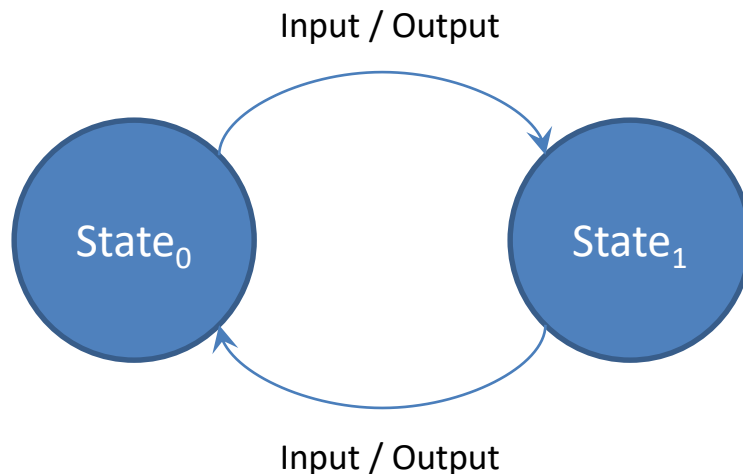
# Representing a Finite State Machine

- It can be represented using a **state transition table** which shows the *current state*, *input*, *any outputs*, and the *next state*.

Input Current State	Input <sub>0</sub>	Input <sub>1</sub>	....	Input <sub>n</sub>
	Next State / Output			
State <sub>0</sub>	....	....	....	....
State <sub>1</sub>	....	....	....	....
....	....	....	....	....
State <sub>n</sub>	....	....	....	....

## Representing a Finite State Machine

- It can also be represented using a **state diagram** which has the same information as the state transition diagram.



# Concept of the State Machine

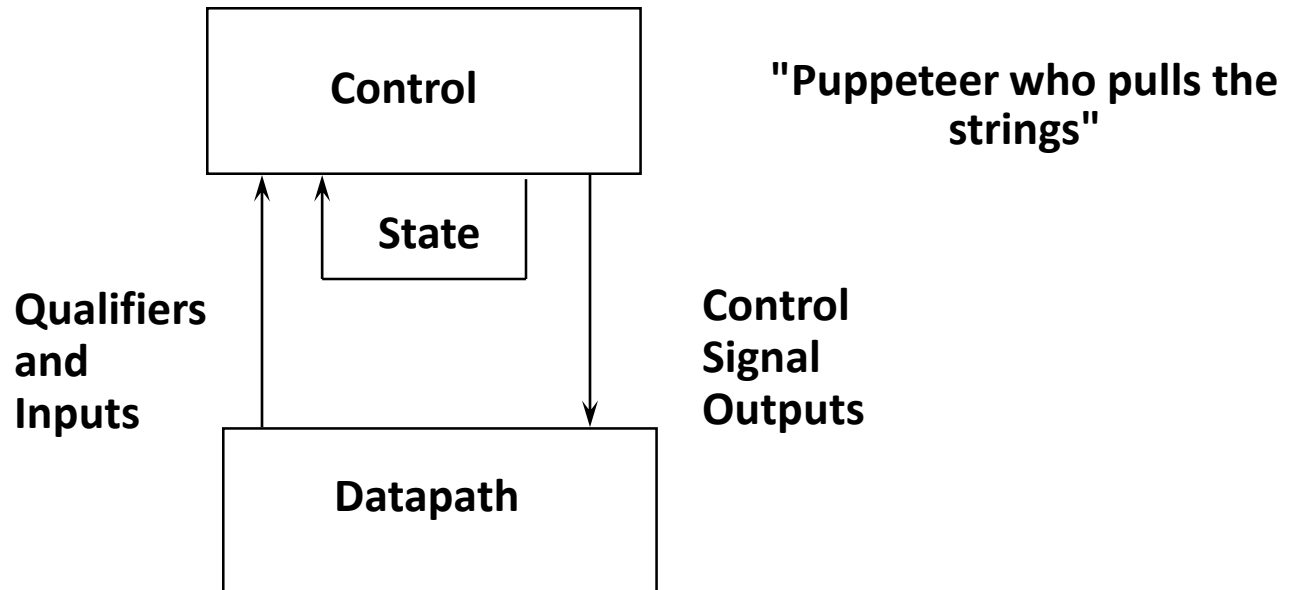
**Computer Hardware = Datapath + Control**

**Registers  
Combinational Functional  
Units (e.g., ALU)  
Busses**

*Qualifiers*

**FSM generating sequences  
of control signals  
Instructs datapath what to  
do next**

*Control*



# State Machines: Definition of Terms

- State Diagram

- Illustrates the form and function of a state machine. Usually drawn as a bubble-and-arrow diagram.

- State

- A uniquely identifiable set of values measured at various points in a digital system.

- Next State

- The state to which the state machine makes the next transition, determined by the inputs present when the device is clocked.

- Branch

- A change from present state to next state.

- Mealy Machine

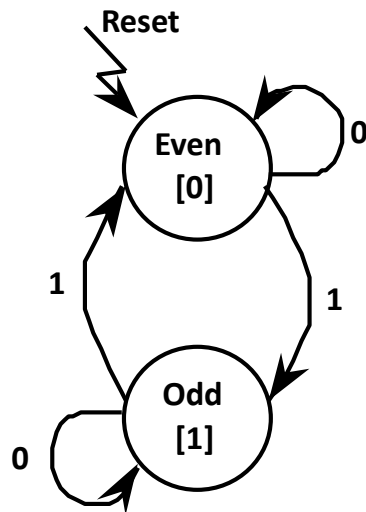
- A state machine that determines its outputs from the present state and from the inputs.

- Moore Machine

- A state machine that determines its outputs from the present state only.

# 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

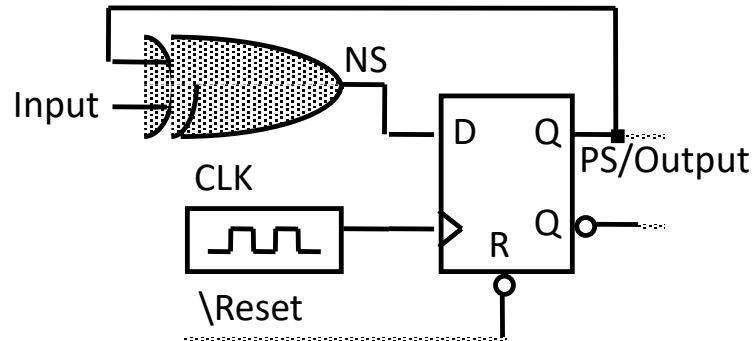
Encoded State Transition Table



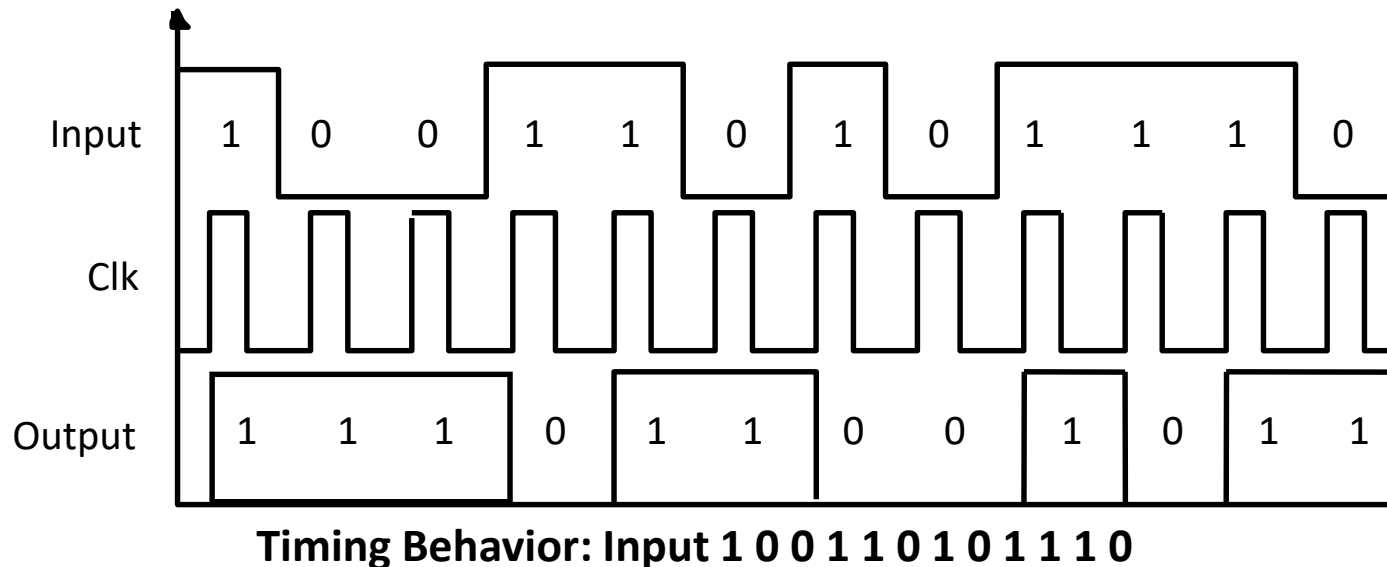
# Odd Parity Checker Design

Next State/Output Functions

$$NS = PS \text{ xor } PI; \quad OUT = PS$$



D FF Implementation



# Timing of State Machines

When are inputs sampled, next state computed, outputs asserted?

***State Time:*** Time between clocking events

- Clocking event causes state/outputs to transition, based on inputs
- For set-up/hold time considerations:

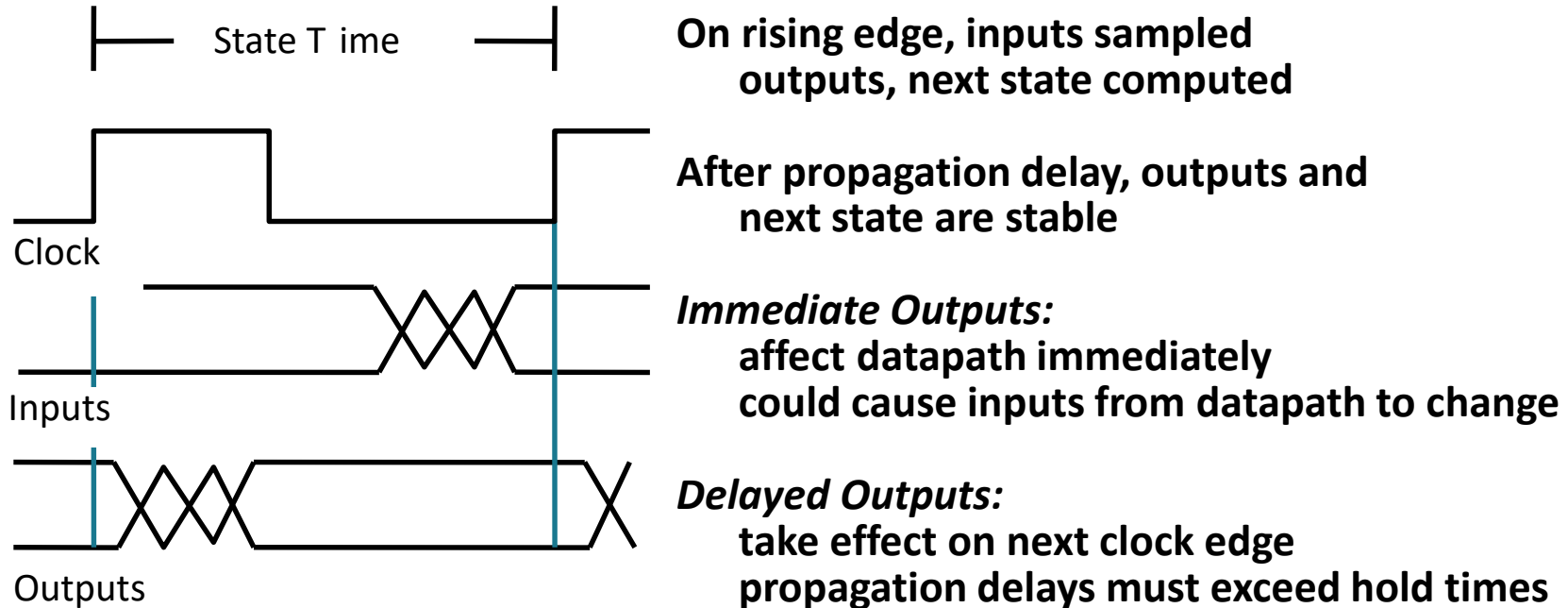
Inputs should be stable before clocking event

- After propagation delay, Next State entered, Outputs are stable

**NOTE:** Asynchronous signals take effect immediately  
Synchronous signals take effect at the next clocking event

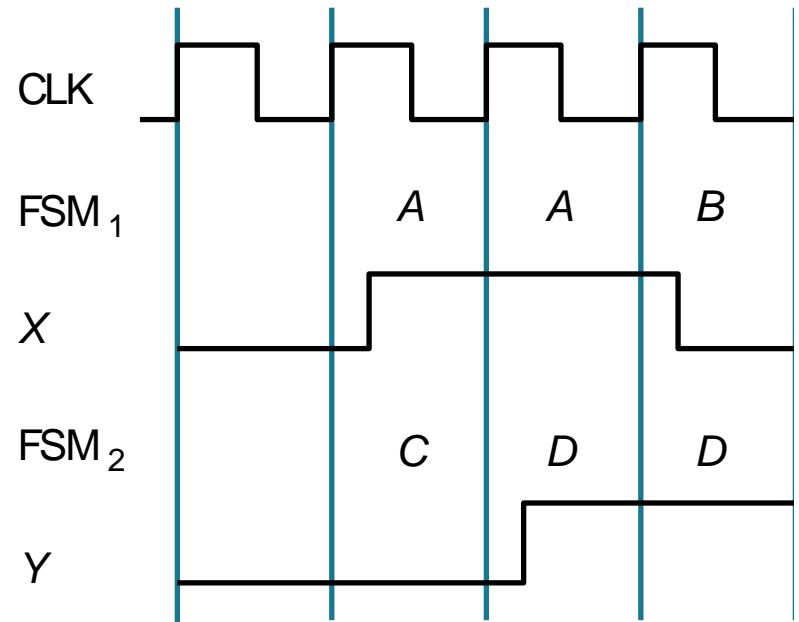
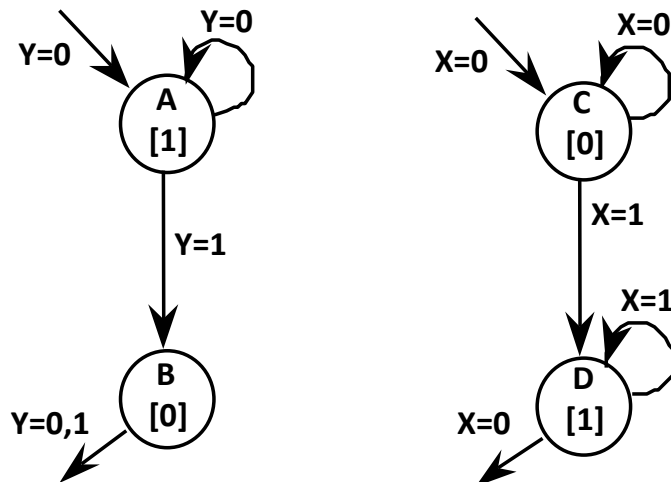
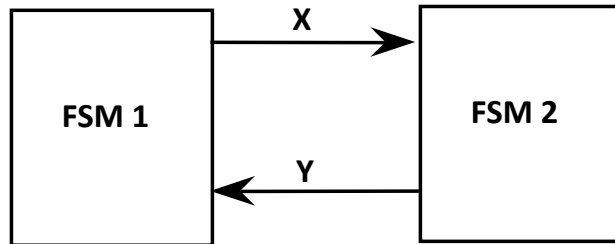
# Timing of State Machine

## *Example: Positive Edge Triggered Synchronous System*



# Communicating State Machines

One machine's output is another machine's input

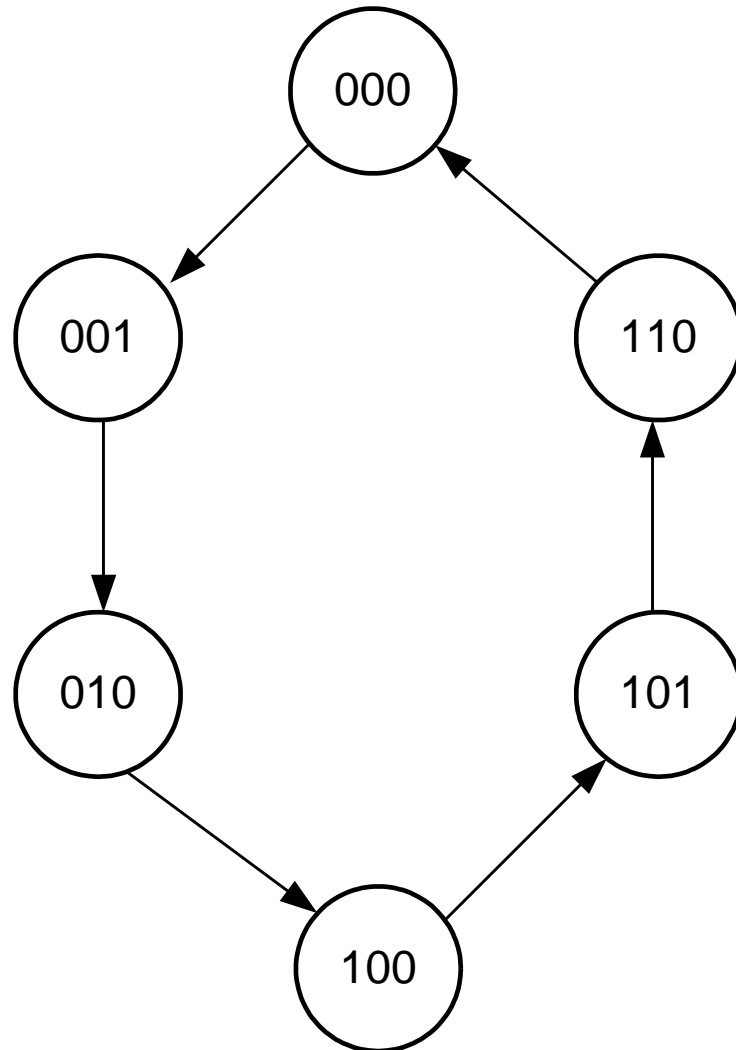


**Machines advance in lock step**  
**Initial inputs/outputs:  $X = 0$ ,  $Y = 0$**

# Analysis of FSMs with D flip-flops

- Determine the next state and output functions
- Use the functions to create a state/output table that specifies every possible next state and output for any combination of current state and input

# Example: counter

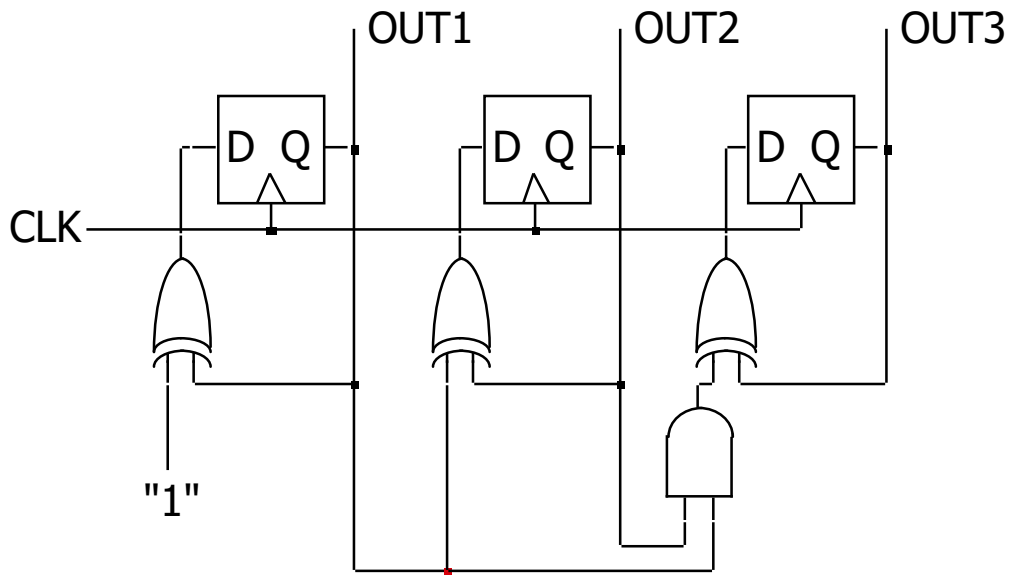


# Self-correcting state machines

- The previous example did not include two possible states “011” and “111”. If the counter unexpectedly falls into one of those states there are two possibilities:
  - The counter will recover by entering a valid state after a finite number of cycles (self-correcting)
  - The counter will stay in a non-valid state until the f/fs are reset (not self-correcting)
- Finite state machines should be designed to be self correcting by assigning non-valid states to a valid next state (no don't cares in the excitation table)

# Turning a State Diagram into Logic

- Counter
  - Three 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



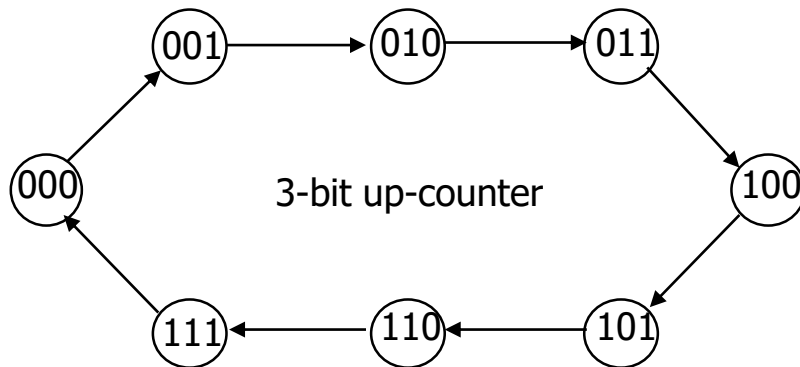


# FSM Design Procedure

- 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



current 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

notation to show  
function represent  
input to D-FF

$$N1 := C1'$$

$$N2 := C1C2' + C1'C2$$

$$:= C1 \text{ xor } C2$$

$$N3 := C1C2C3' + C1'C3 + C2'C3$$

$$:= C1C2C3' + (C1' + C2')C3$$

$$:= (C1C2) \text{ xor } C3$$

N3

			C3
	0	0	1 1
C1	0	1	0 1
		C2	

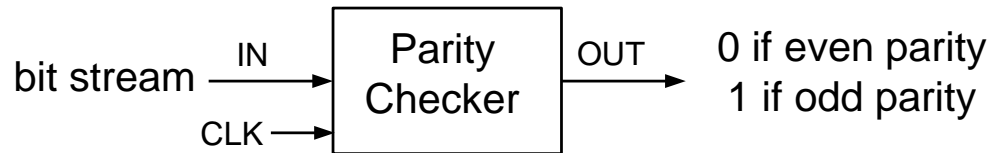
N2

			C3
	0	1 1	0
C1	1	0 0	1
		C2	

N1

			C3
	1	1	1 1
C1	0	0	0 0
		C2	

# Parity Checker FSM

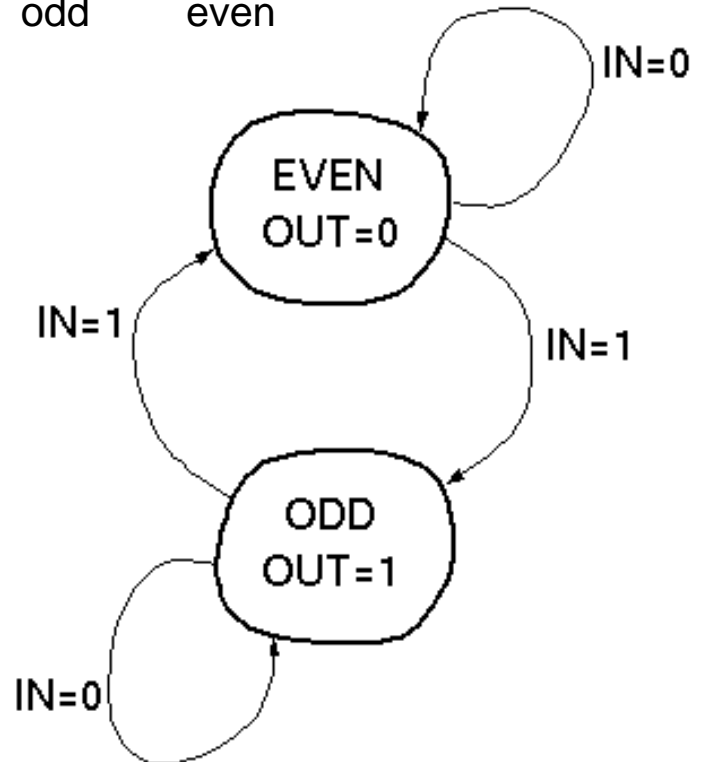


example:    0            0            1            1            1            0            1  
              even        even        odd        even        odd        odd        even

-----> time

## ⌘ "State Transition Diagram"

- ⊞ circuit is in one of two states.
- ⊞ transition on each cycle with each new input, over exactly one arc (edge).
- ⊞ Output depends on which state the circuit is in.



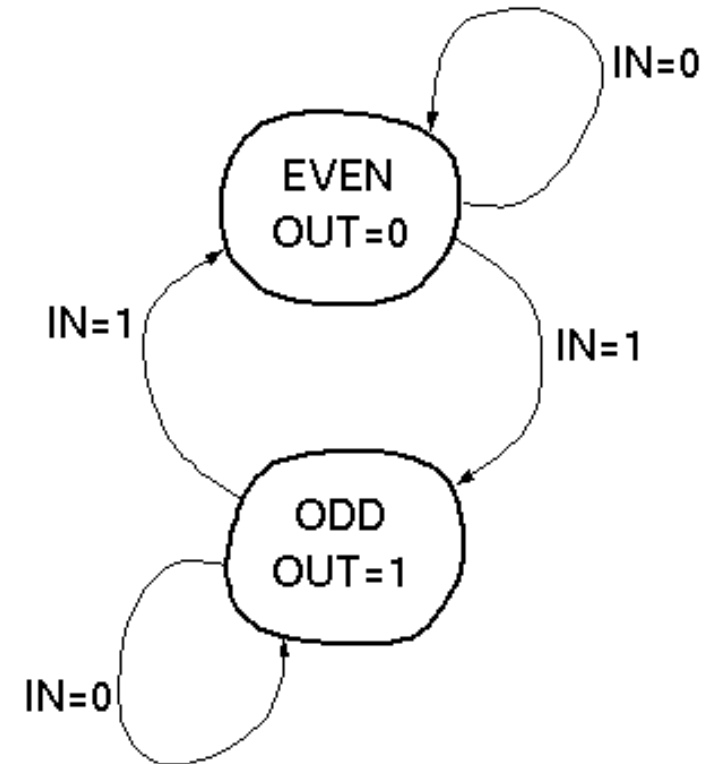
# Formal Design Process

## ⌘ State Transition Table:

present state	OUT	IN	next state
EVEN	0	0	EVEN
EVEN	0	1	ODD
ODD	1	0	ODD
ODD	1	1	EVEN

## ⌘ Invent a code to represent states:

Let 0 = EVEN state, 1 = ODD state



present state (ps)	OUT	IN	next state (ns)
0	0	0	0
0	0	1	1
1	1	0	1
1	1	1	0

Derive logic equations from table (how?):

$$OUT = PS$$

$$NS = PS \text{ xor } IN$$

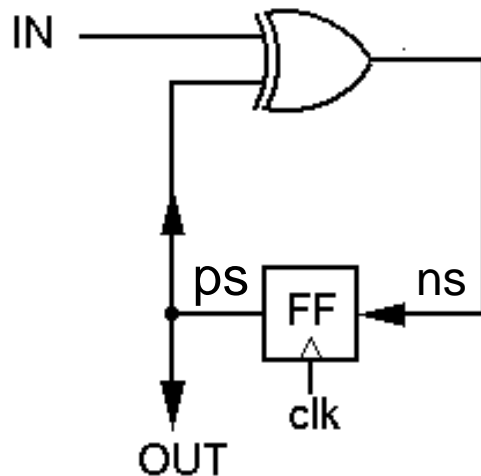
# Formal Design Process

Logic equations from table:

$$\text{OUT} = \text{PS}$$

$$\text{NS} = \text{PS} \text{ xor } \text{IN}$$

⌘ Circuit Diagram:



- ☒ XOR gate for ns calculation
- ☒ DFF to hold present state
- ☒ no logic needed for output

⌘ Review of Design Steps:

1. Circuit functional specification
2. State Transition Diagram
3. Symbolic State Transition Table
4. Encoded State Transition Table
5. Derive Logic Equations
6. Circuit Diagram

FFs for state

CLK for NS and OUT

# Another example: Door combination lock

⌘ Door combination lock:

- ☐ punch in 3 values in sequence and the door opens; if there is an error the lock must be reset; once the door opens the lock must be reset

- ☐ inputs: sequence of input values, reset

- ☐ outputs: door open/close

- ☐ memory: must remember combination  
or always have it available as an input

# Sequential example: abstract control

## ⌘ Finite-state diagram

- ▣ States: 5 states

  - ▣ represent point in execution of machine

  - ▣ each state has outputs

- ▣ Transitions: 6 from state to state, 5 self transitions, 1 global

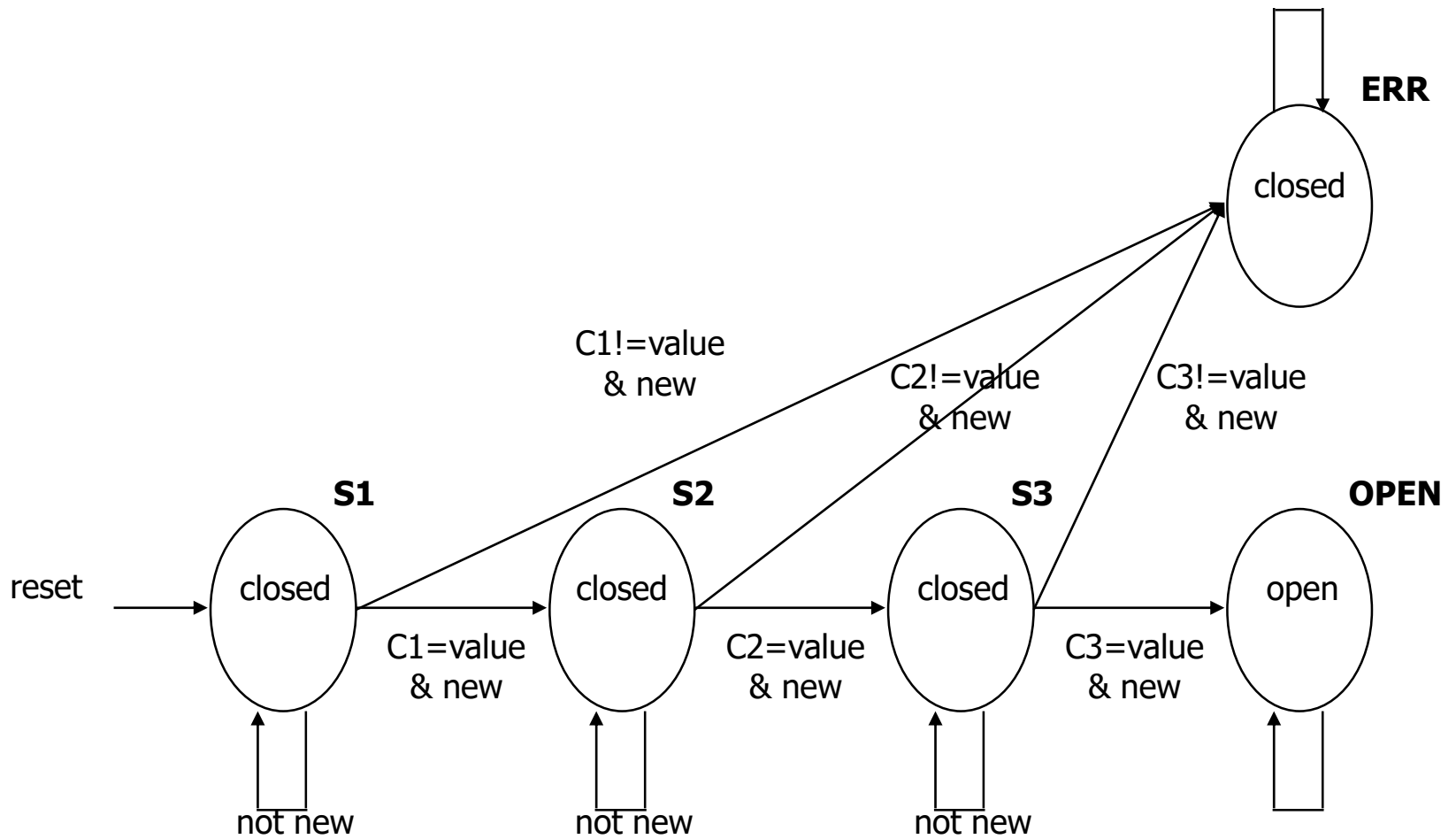
  - ▣ changes of state occur when clock says it's ok

  - ▣ based on value of inputs

- ▣ Inputs: reset, new, results of comparisons

- ▣ Output: open/closed

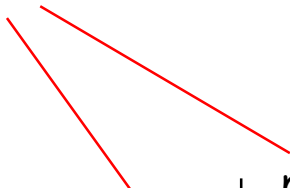




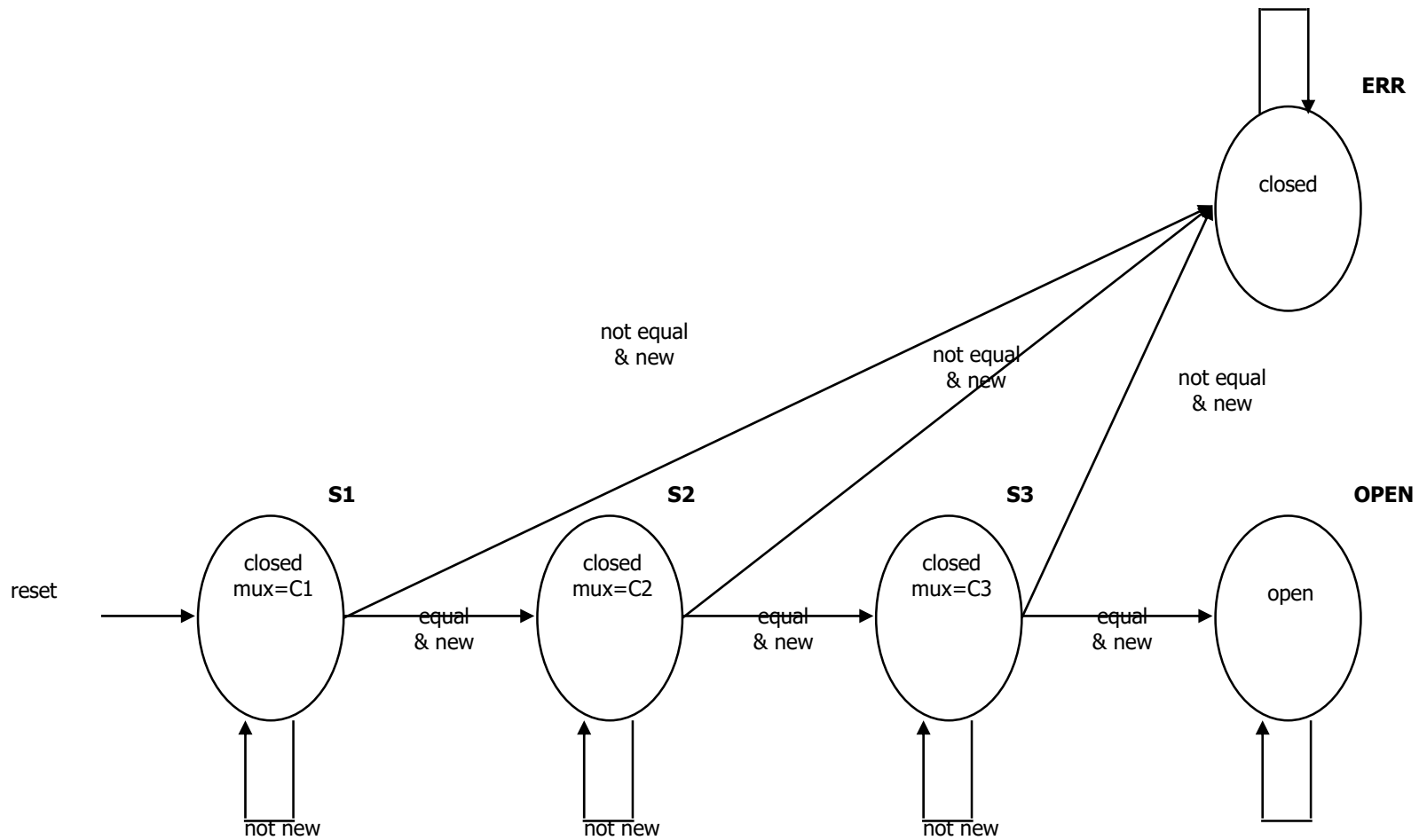
# Sequential example: finite-state machine

⌘ Finite-state machine

Symbolic states



reset	new	equal	state	state	next mux	open/closed
1	-	-	-	S1	C1	closed
0	0	-	S1	S1	C1	closed
0	1	0	S1	ERR	-	closed
0	1	1	S1	S2	C2	closed
0	0	-	S2	S2	C2	closed
0	1	0	S2	ERR	-	closed
0	1	1	S2	S3	C3	closed
0	0	-	S3	S3	C3	closed
0	1	0	S3	ERR	-	closed
0	1	1	S3	OPEN	-	closed
0	-	-	OPEN	OPEN	-	open
0	-	-	ERR	ERR	-	closed



# Sequential example: encoding

## ⌘ Encode state table

⊡ state can be: S1, S2, S3, OPEN, or ERR

⊡ needs at least 3 bits to encode: 000, 001, 010, 011, 100

⊡ and as many as 5: 00001, 00010, 00100, 01000, 10000

⊡ choose 4 bits: 0001, 0010, 0100, 1000, 0000

binary

One-hot

hybrid

## ⌘ Encode outputs

⊡ output mux can be: C1, C2, or C3

⊡ needs 2 to 3 bits to encode

⊡ choose 3 bits: 001, 010, 100

⊡ output open/closed can be: open or closed

⊡ needs 1 or 2 bits to encode

⊡ choose 1 bits: 1, 0

# Sequential example: encoding

## ⌘ Encode state table

⊡ state can be: S1, S2, S3, OPEN, or ERR

⊡ choose 4 bits: 0001, 0010, 0100, 1000, 0000

⊡ output mux can be: C1, C2, or C3

⊡ choose 3 bits: 001, 010, 100

⊡ output open/closed can be: open or closed

⊡ choose 1 bits: 1, 0

# Sequential example: encoding

reset	new	equal	state	next state	mux	open/closed
1	-	-	-	0001	001	0
0	0	-	0001	0001	001	0
0	1	0	0001	0000	-	0
0	1	1	0001	0010	010	0
0	0	-	0010	0010	010	0
0	1	0	0010	0000	-	0
0	1	1	0010	0100	100	0
0	0	-	0100	0100	100	0
0	1	0	0100	0000	-	0
0	1	1	0100	1000	-	1
0	-	-	1000	1000	-	1
0	-	-	0000	0000	-	0

good choice of encoding!

mux is identical to  
last 3 bits of next state

open/closed is  
identical to first bit  
of state

# State Minimization

- Fewer states may mean fewer state variables
- High-level synthesis may generate many redundant states
- Two states are equivalent if they are impossible to distinguish from the outputs of the FSM, i. e., for any input sequence the outputs are the same
- Two conditions for two states to be equivalent:
  - 1) Output must be the same in both states
  - 2) Must transition to equivalent states for all input combinations

# Example :

## A Modulo-4 Synchronous Counter

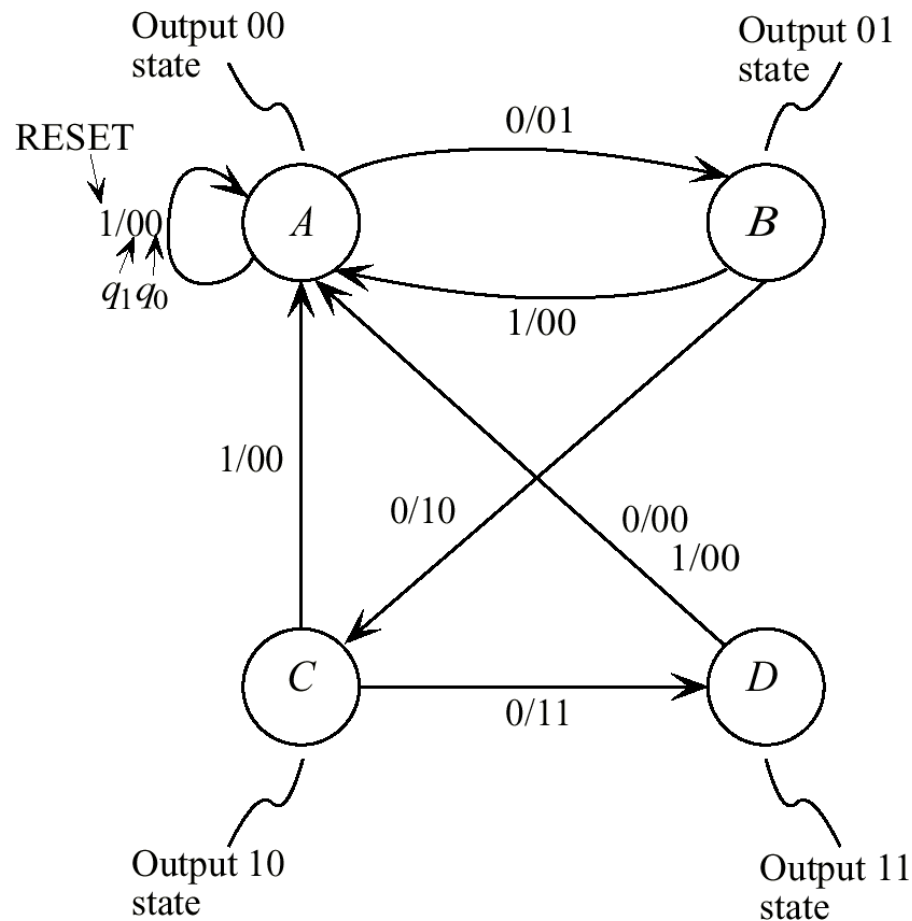
- Counts from 0 to 3 and then repeats.
- It has a clock input (CLK) and a RESET input.
- Outputs appear as a sequence of values ( $q_1$  and  $q_0$ ) at time steps corresponding to the clock.
- As the outputs are generated, a new state ( $s_1s_0$ ) is generated which takes on values of 00, 01, 10, and 11 and are fed back to the input.



# The Mod-4 Counter

- Requires four states, encoded in binary, with at least two bits for them to be encoded uniquely.
  - $A = 00$
  - $B = 01$
  - $C = 10$
  - $D = 11$
- The input requires only a single bit.
  - $0 \mid 1$

# State Transition Diagram for the Mod-4 Counter



# State Table for the Mod-4 Counter

<div>Input</div> <div>Present state</div>	<i>RESET</i>	
	0	1
<i>A</i>	<i>B</i> /01	<i>A</i> /00
<i>B</i>	<i>C</i> /10	<i>A</i> /00
<i>C</i>	<i>D</i> /11	<i>A</i> /00
<i>D</i>	<i>A</i> /00	<i>A</i> /00

Next state

Output

# State Assignment for the Mod-4 Counter

Present state ( $S_t$ ) \ Input	<i>RESET</i>	
	0	1
<i>A</i> :00	01/01	00/00
<i>B</i> :01	10/10	00/00
<i>C</i> :10	11/11	00/00
<i>D</i> :11	00/00	00/00

# Truth Table for the Mod-4 Counter

<i>RESET</i> $r(t)$	$s_1(t)$	$s_0(t)$	$s_1s_0(t+1)$	$q_1q_0(t+1)$
0	0	0	01	01
0	0	1	10	10
0	1	0	11	11
0	1	1	00	00
1	0	0	00	00
1	0	1	00	00
1	1	0	00	00
1	1	1	00	00

$$s_0(t+1) = r(t)'s_1(t)'s_0(t)' + r(t)'s_1(t)s_0(t)'$$

$$s_1(t+1) = r(t)'s_1(t)'s_0(t) + r(t)'s_1(t)s_0(t)'$$

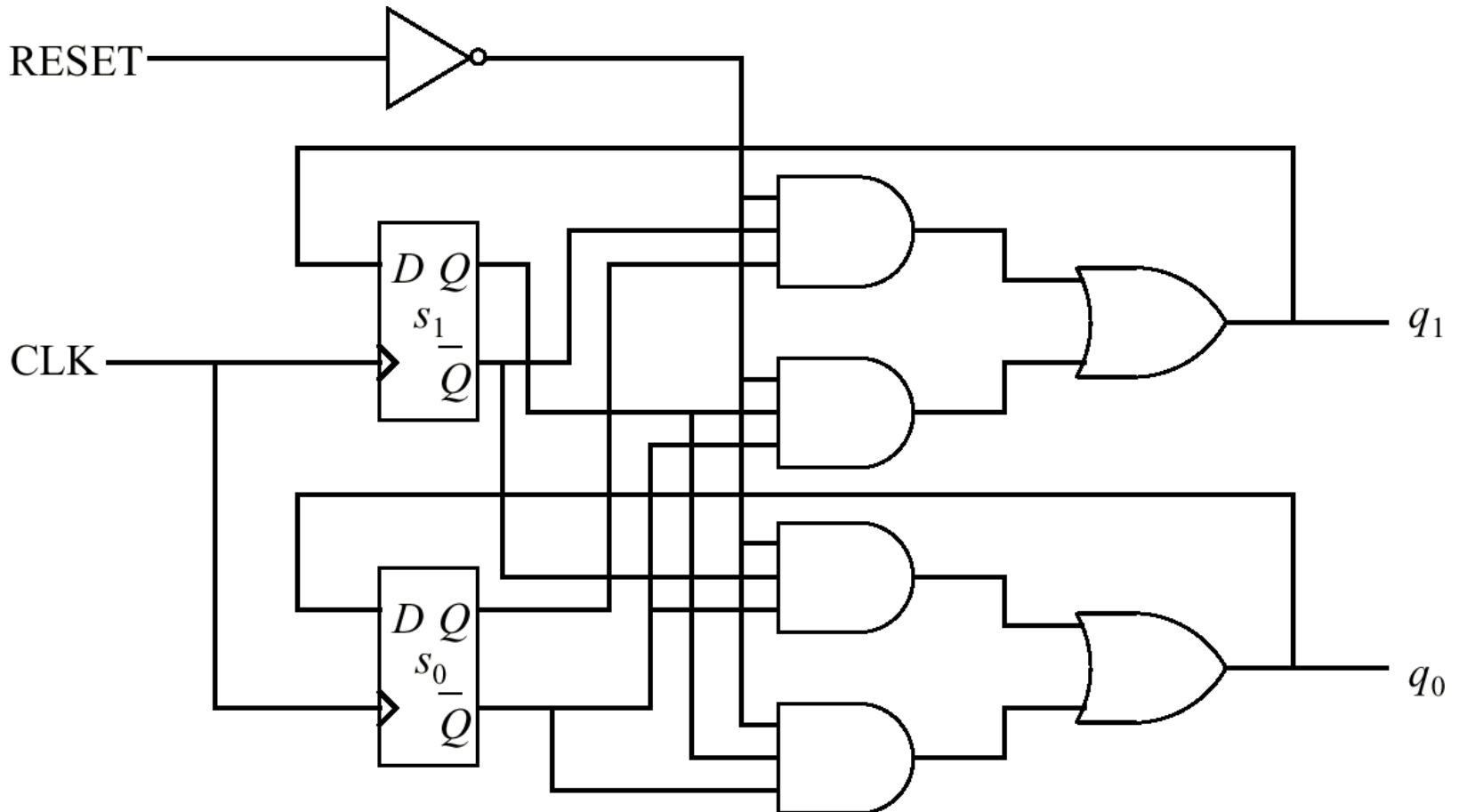
$$q_0(t+1) = r(t)'s_1(t)'s_0(t)' + r(t)'s_1(t)s_0(t)'$$

$$q_1(t+1) = r(t)'s_1(t)'s_0(t) + r(t)'s_1(t)s_0(t)'$$

Notice that  $s_0(t+1) = q_0(t+1)$

and  $s_1(t+1) = q_1(t+1)$

# Logic Design for Mod-4 Counter



# Example : A Sequence Detector

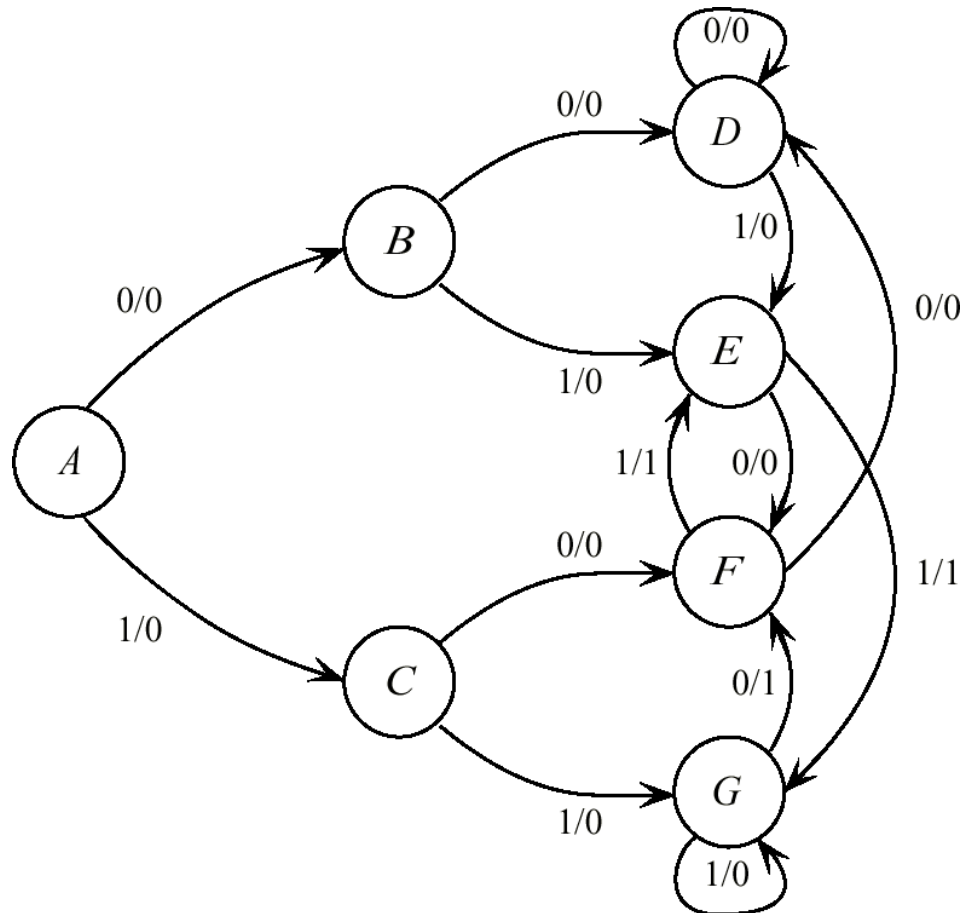
- Design a sequence detector using D flip-flops and 8-to-1 multiplexers.
- The sequence detector outputs a 1 when exactly two of the last three inputs are 1.
  - *An input of 011011100 produces the output of 001111010*
- There is a one-bit serial input line and we will assume that initially no inputs have been seen.
- **Note:** the sequence detector cannot output a 1 until at least three inputs have been read.

# The Sequence Detector

- There are a total of eight sequences that the machine can observe:
  - 000, 001, 010, 011, 100, 101, 110, 111
- We will assume that state A is the initial state where no inputs have been fed into the machine.
- In states B and C, only one input has been fed into the machine and therefore we cannot output a 1.



# State Transition Diagram for the Sequence Detector



# State Table and State Assignment for the Sequence Detector

Input Present state	$X$	
	0	1
$A$	$B/0$	$C/0$
$B$	$D/0$	$E/0$
$C$	$F/0$	$G/0$
$D$	$D/0$	$E/0$
$E$	$F/0$	$G/1$
$F$	$D/0$	$E/1$
$G$	$F/1$	$G/0$

Input Present state	$X$	
	0	1
$s_2s_1s_0$	$s_2s_1s_0Z$	$s_2s_1s_0Z$
$A: 000$	$001/0$	$010/0$
$B: 001$	$011/0$	$100/0$
$C: 010$	$101/0$	$110/0$
$D: 011$	$011/0$	$100/0$
$E: 100$	$101/0$	$110/1$
$F: 101$	$011/0$	$100/1$
$G: 110$	$101/1$	$110/0$

# Truth Table for the Sequence Detector

Input and  
state at  
time  $t$

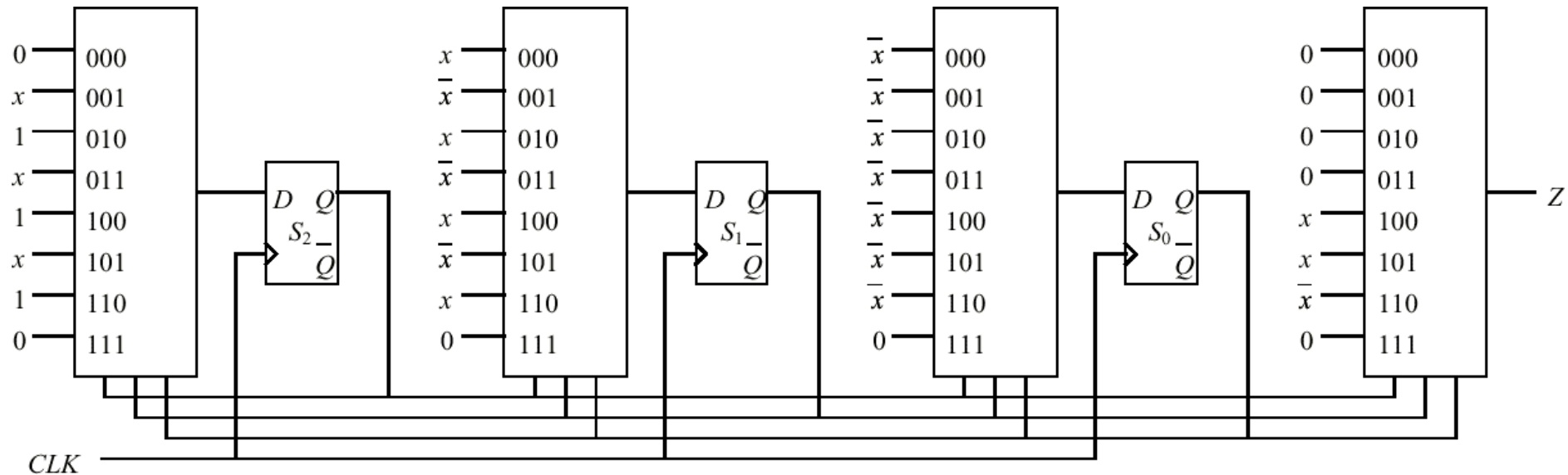
Next state  
and output at  
time  $t+1$

$s_2$	$s_1$	$s_0$	$x$	$s_2$	$s_1$	$s_0$	$z$
0	0	0	0	0	0	1	0
0	0	0	1	0	1	0	0
0	0	1	0	0	1	1	0
0	0	1	1	1	0	0	0
0	1	0	0	1	0	1	0
0	1	0	1	1	1	0	0
0	1	1	0	0	1	1	0
0	1	1	1	1	0	0	0
1	0	0	0	1	0	1	0
1	0	0	1	1	1	0	1
1	0	1	0	0	1	1	0
1	0	1	1	1	0	0	1
1	1	0	0	1	0	1	1
1	1	0	1	1	1	0	0
1	1	1	0	d	d	d	d
1	1	1	1	d	d	d	d

# Creating the Circuit for the Sequence Detector

- There needs to be one flip-flop for each state variable, so a total of three are needed.
- Also, there are three next state functions and one output function, so four 8-to-1 multiplexers are needed.

# Logic Diagram for the Sequence Detector



## Design Example: Sequence Recognizer

- A sequential circuit that recognizes the occurrence of a particular bit sequence
- Input:  $X(t) \in \{0, 1\}$
- Output:  $Z(t) \in \{0, 1\}$

$$Z(t) = \begin{cases} 1, & \text{if } X(t-3, t) = 1101 \\ 0, & \text{Otherwise} \end{cases}$$

# Sequence Recognizer

71

$$Z(t) = \begin{cases} 1, & \text{if } X(t-3, t) = 1101 \\ 0, & \text{Otherwise} \end{cases}$$

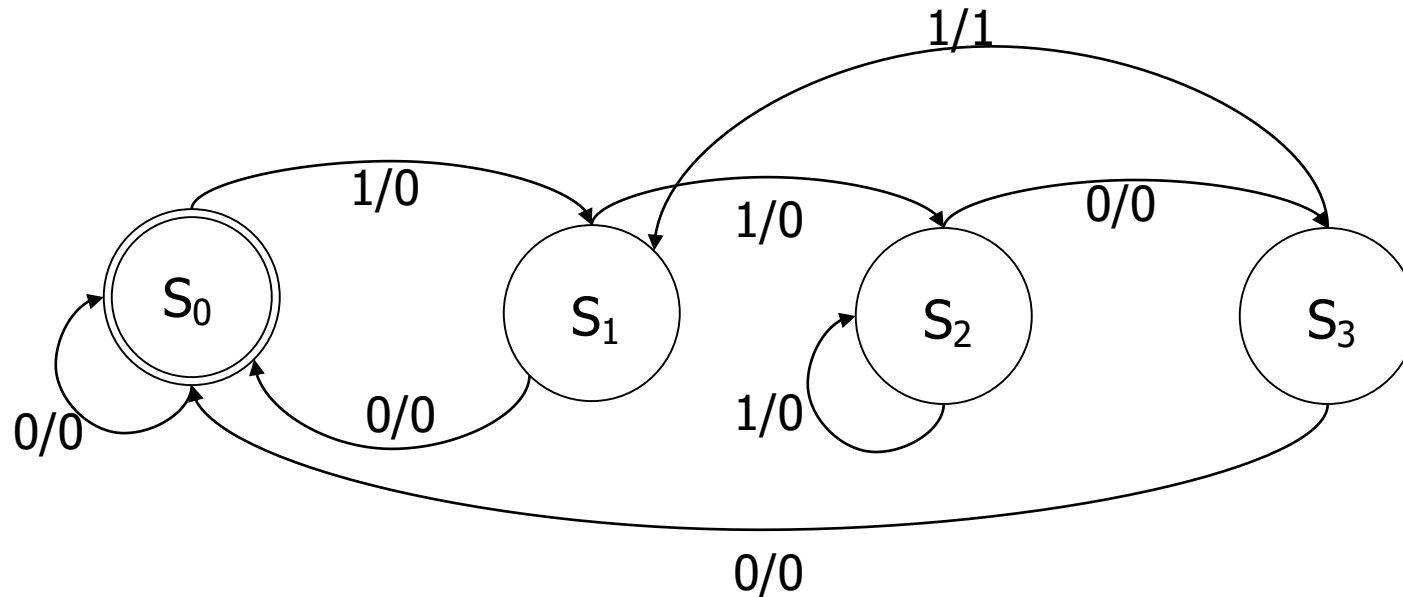
Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
X(t)	1	0	0	1	0	1	1	0	1	0	1	1	0	1	1	0	1
Z(t)	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	1

# Sequence Recognizer

72

$$Z(t) = \begin{cases} 1, & \text{if } X(t-3, t) = 1101 \\ 0, & \text{Otherwise} \end{cases}$$

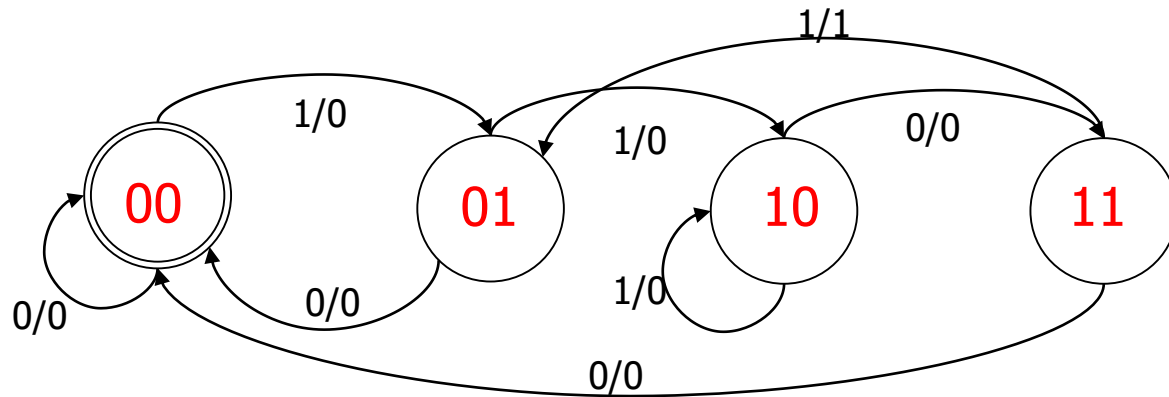
Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
X(t)	1	0	0	1	0	1	1	0	1	0	1	1	0	1	1	0	1
Z(t)	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	1



A Mealy Machine



# State Table



Present State		Input X	Next State		Output Z
P1	P0		N1	N0	
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	1	1	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	1	0	1	1

# Logic Circuits Design

Present State		X	Next State		Z
P1	P0		N1	N0	
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	1	1	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	1	0	1	1

X	P1P0			
	00	01	11	10
0	0	0	0	1
1	0	1	0	1

$$N1 = X\overline{P_1}\overline{P_0} + P_1\overline{P_0}$$

X	P1P0			
	00	01	11	10
0	0	0	0	1
1	1	0	1	0

$$N0 = X\overline{P_1}\overline{P_0} + X\overline{P_1}P_0 + \overline{X}P_1\overline{P_0}$$

$$= X(\overline{P_1}\overline{P_0} + \overline{P_1}P_0) + \overline{X}P_1\overline{P_0}$$

$$= X(\overline{P_1}(\overline{P_0} + P_0)) + \overline{X}P_1\overline{P_0}$$

$$= X\overline{P_1} + \overline{X}P_1\overline{P_0}$$

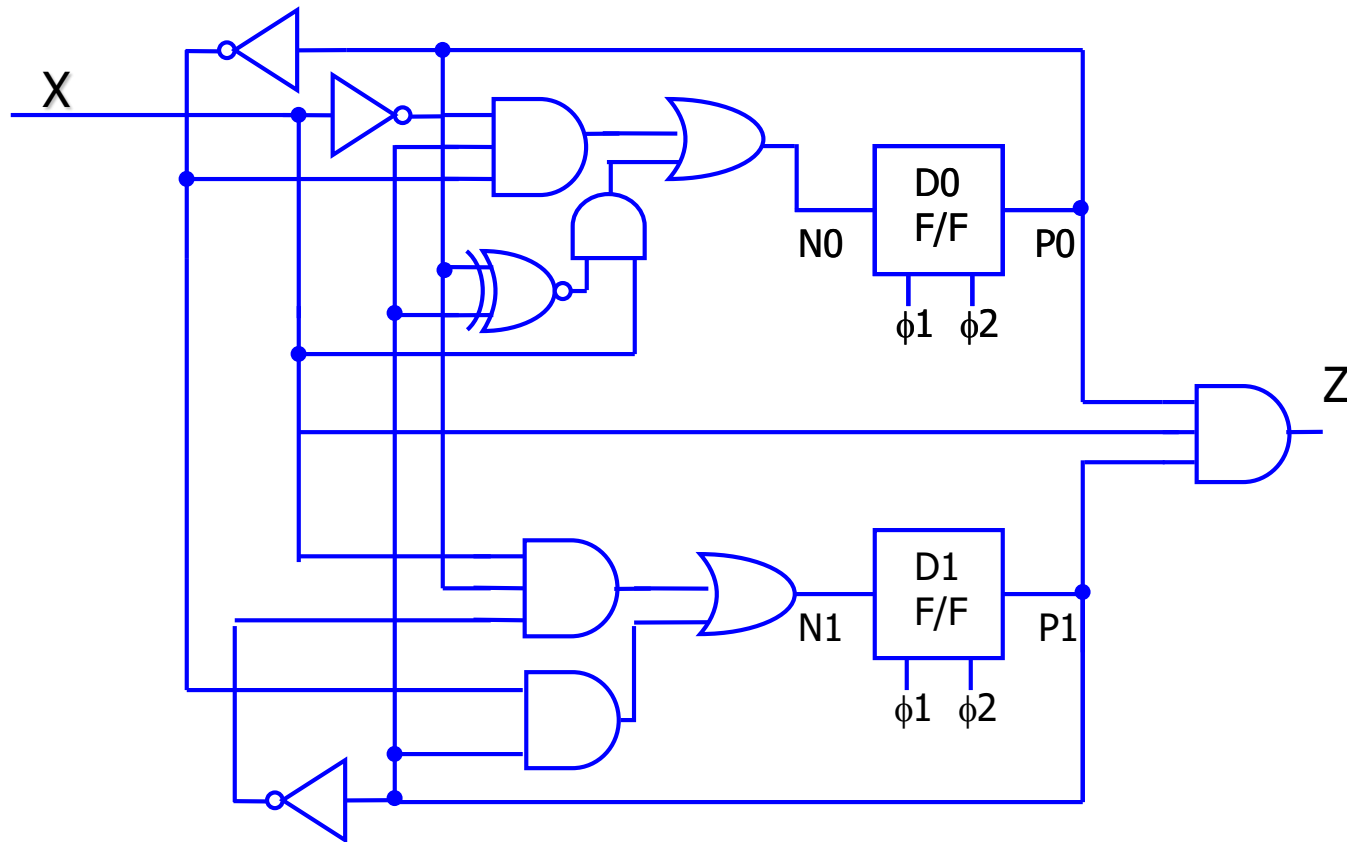
$$Z = X\overline{P_1}P_0$$

# Logic Circuits Design

$$N0 = X(\overline{P1 \oplus P0}) + \overline{X}P_1\overline{P_0}$$

$$N1 = X\overline{P_1}P_0 + P_1\overline{P_0}$$

$$Z = XP_1P_0$$



# Example

76

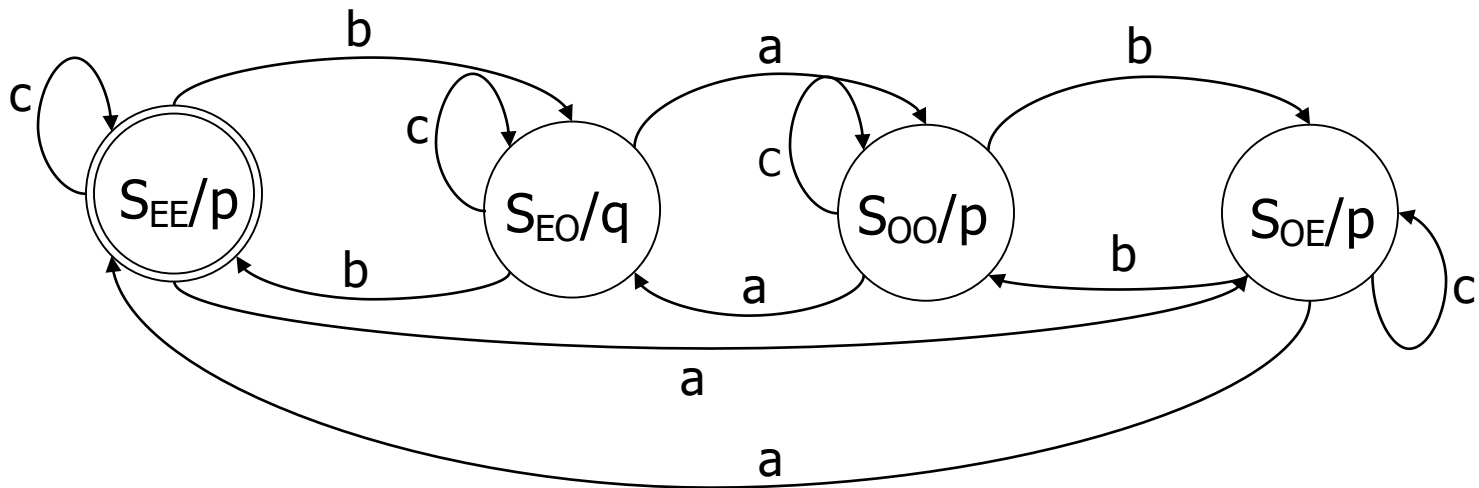
- Input:  $X(t) \in \{a, b, c\}$
- Output:  $Z(t) \in \{q, p\}$

$$Z(t) = \begin{cases} q, & \text{when input sequence has} \\ & \text{even \# of a's and odd \# of b's} \\ p, & \text{Otherwise} \end{cases}$$

# State Diagram

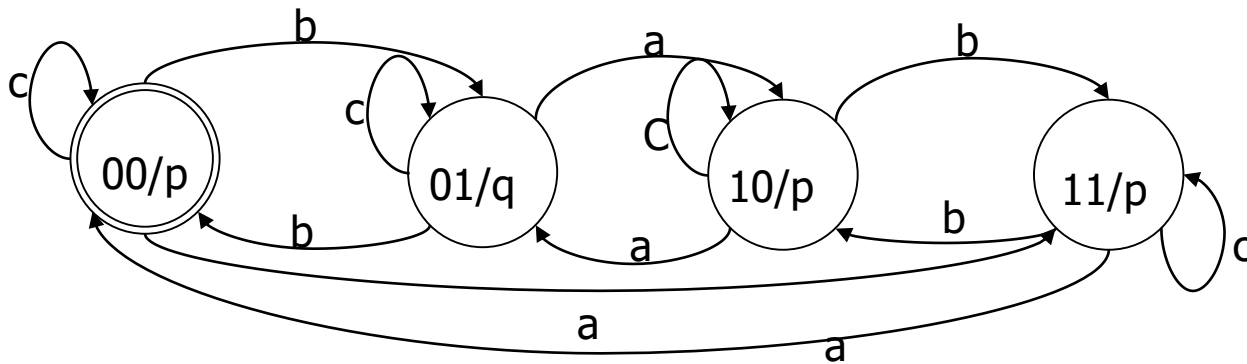
77

$$Z(t) = \begin{cases} q, & \text{when input sequence has} \\ & \text{even \# of a's and odd \# of b's} \\ p, & \text{Otherwise} \end{cases}$$



A Moore Machine

# State Table



$S_{EE} = 00$

$S_{EO} = 01$

$S_{OO} = 10$

$S_{OE} = 11$

$a = 00$

$b = 01$

$c = 10$

$p = 0$

$q = 1$

Present State		Input (a, b, c)		Next State		Output
P1	P0	X1	X0	N1	N0	Z
0	0	0	0	1	1	0
0	0	0	1	0	1	0
0	0	1	0	0	0	0
0	1	0	0	1	0	1
0	1	0	1	0	0	1
0	1	1	0	0	1	1
1	0	0	0	0	1	0
1	0	0	1	1	1	0
1	0	1	0	1	0	0
1	1	0	0	0	0	0
1	1	0	1	1	0	0
1	1	1	0	1	1	0

# Logic Circuit Design

Present State		Input		Next State		Output
P1	P0	X1	X0	N1	N0	Z
0	0	0	0	1	1	0
0	0	0	1	0	1	0
0	0	1	0	0	0	0
0	1	0	0	1	0	1
0	1	0	1	0	0	1
0	1	1	0	0	1	1
1	0	0	0	0	1	0
1	0	0	1	1	1	0
1	0	1	0	1	0	0
1	1	0	0	0	0	0
1	1	0	1	1	0	0
1	1	1	0	1	1	0

**N1**

P1P0 \ X1X0	00	01	11	10
00	1	0	X	0
01	1	0	X	0
11	0	1	X	1
10	0	1	X	1

$$\begin{aligned}
 N1 &= \overline{P1} \overline{X1} \overline{X0} + P1 X0 + P1 X1 \\
 &= \overline{P1} (\overline{X1} + \overline{X0}) + P1 (X0 + X1) \\
 &= \overline{P1} \oplus (\overline{X1} + \overline{X0})
 \end{aligned}$$

**N0**

P1P0 \ X1X0	00	01	11	10
00	1	1	X	0
01	0	0	X	1
11	0	0	X	1
10	1	1	X	0

$$\begin{aligned}
 N0 &= \overline{P0} \overline{X1} + P0 X1 \\
 &= \overline{P0} \oplus X1
 \end{aligned}$$

**Z**

P1P0 \ X1X0	00	01	11	10
00	0	0	X	0
01	1	1	X	1
11	0	0	X	0
10	0	0	X	0

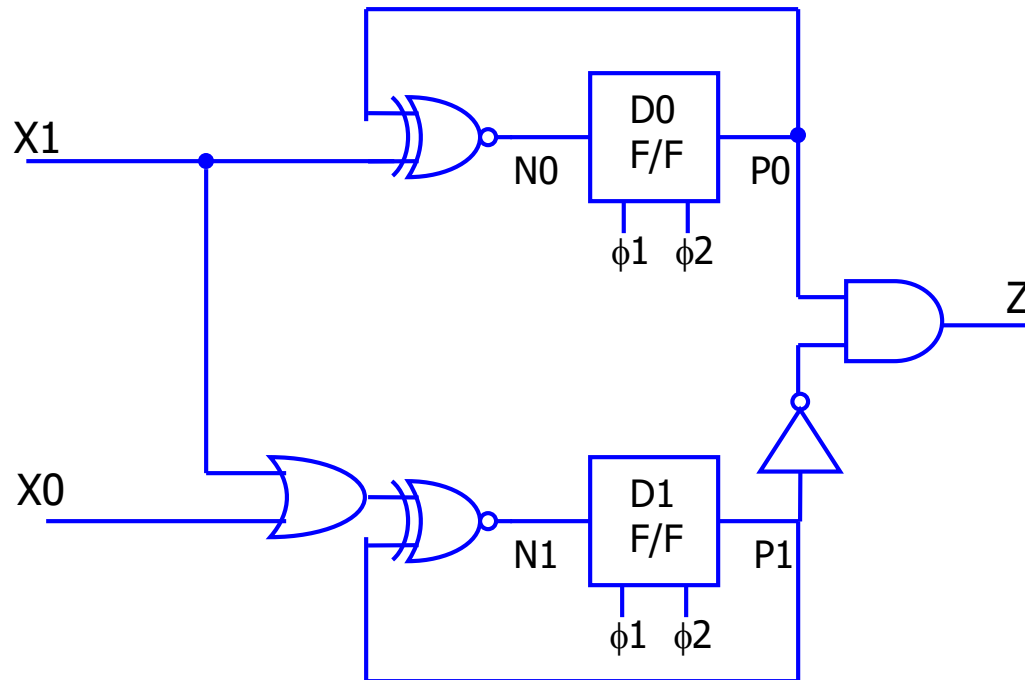
$$Z = \overline{P1} P0$$

# Logic Circuit Design

$$N1 = \overline{P1 \oplus (X1 + X0)}$$

$$N0 = \overline{P0 \oplus X1}$$

$$Z = \overline{P1}P0$$



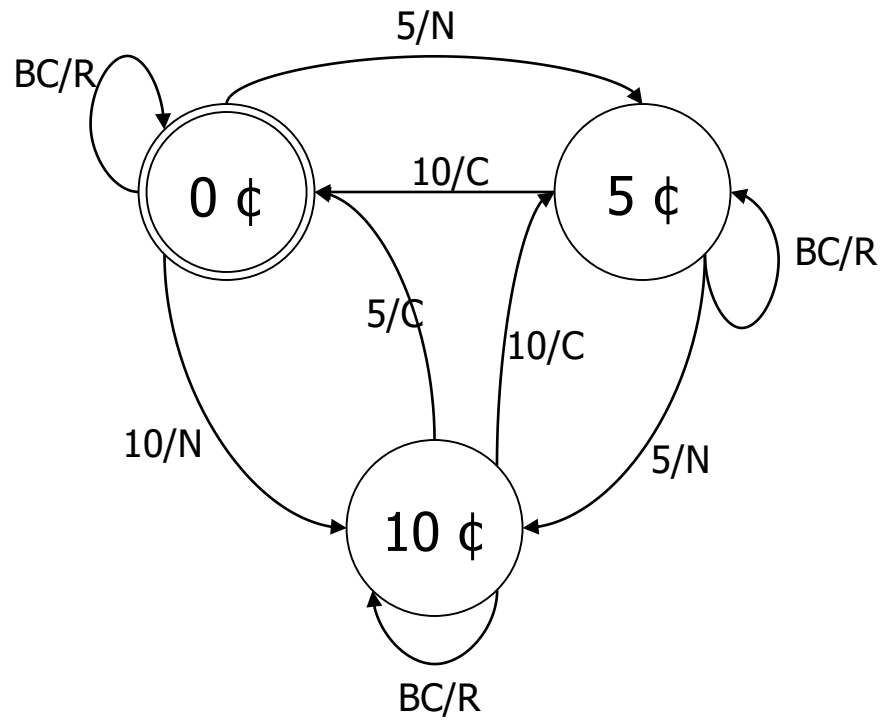


# Vending Machine State Machine<sup>81</sup>

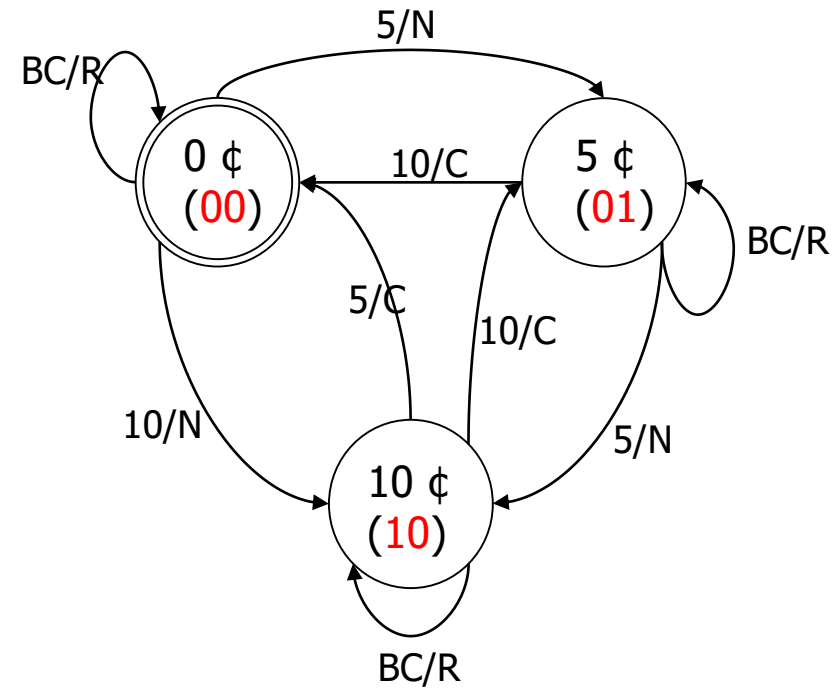
- Dispense a Coke when depositing 15 ¢
- Inputs
  - 5 = a nickel
  - 10 = a dime
  - BC = bad coin (including quarters in this example)
- Outputs
  - R = reject
  - C = coke
  - N = no coke

# State Diagram

82



# State Table



5: 00      N: 00  
 10: 01     C: 01  
 BC: 10     R: 10

Present State (0¢, 5¢, 10¢)		Input (5¢, 10¢, BC)		Next State (0¢, 5¢, 10¢)		Output (C, N, R)	
P1	P0	X1	X0	N1	N0	C1	C0
0	0	0	0	0	1	0	0
0	0	0	1	1	0	0	0
0	0	1	0	0	0	1	0
0	1	0	0	1	0	0	0
0	1	0	1	0	0	0	1
0	1	1	0	0	1	1	0
1	0	0	0	0	0	0	1
1	0	0	1	0	1	0	1
1	0	1	0	1	0	1	0
X	X	1	1	X	X	X	X
1	1	X	X	X	X	X	X

# Logic Circuits Design

Present State		Input		Next State		Output	
P1	P0	X1	X0	N1	N0	C1	C0
0	0	0	0	0	1	0	0
0	0	0	1	1	0	0	0
0	0	1	0	0	0	1	0
0	1	0	0	1	0	0	0
0	1	0	1	0	0	0	1
0	1	1	0	0	1	1	0
1	0	0	0	0	0	0	1
1	0	0	1	0	1	0	1
1	0	1	0	1	0	1	0
X	X	1	1	X	X	X	X
1	1	X	X	X	X	X	X

P1P0	X1X0			
	00	01	11	10
00	0	1	X	0
01	1	0	X	0
11	X	X	X	X
10	0	0	X	1

$$N1 = \overline{P0} \overline{X1} \overline{X0} + \overline{P1} \overline{P0} X0 + P1 X1$$

P1P0	X1X0			
	00	01	11	10
00	1	0	X	0
01	0	0	X	1
11	X	X	X	X
10	0	1	X	0

$$N0 = \overline{P1} \overline{P0} \overline{X1} \overline{X0} + P1 X0 + P0 X1$$

# Logic Circuits Design

Present State		Input		Next State		Output	
P1	P0	X1	X0	N1	N0	C1	C0
0	0	0	0	0	1	0	0
0	0	0	1	1	0	0	0
0	0	1	0	0	0	1	0
0	1	0	0	1	0	0	0
0	1	0	1	0	0	0	1
0	1	1	0	0	1	1	0
1	0	0	0	0	0	0	1
1	0	0	1	0	1	0	1
1	0	1	0	1	0	1	0
X	X	1	1	X	X	X	X
1	1	X	X	X	X	X	X

**C1**

		X1X0			
		00	01	11	10
P1P0	00	0	0	X	1
	01	0	0	X	1
	11	X	X	X	X
	10	0	0	X	1

$$C1 = X1$$

**C0**

		X1X0			
		00	01	11	10
P1P0	00	0	0	X	0
	01	0	1	X	0
	11	X	X	X	X
	10	1	1	X	0

$$C0 = P1X1 + P0X0$$

# Logic Circuits of the Vending Machine

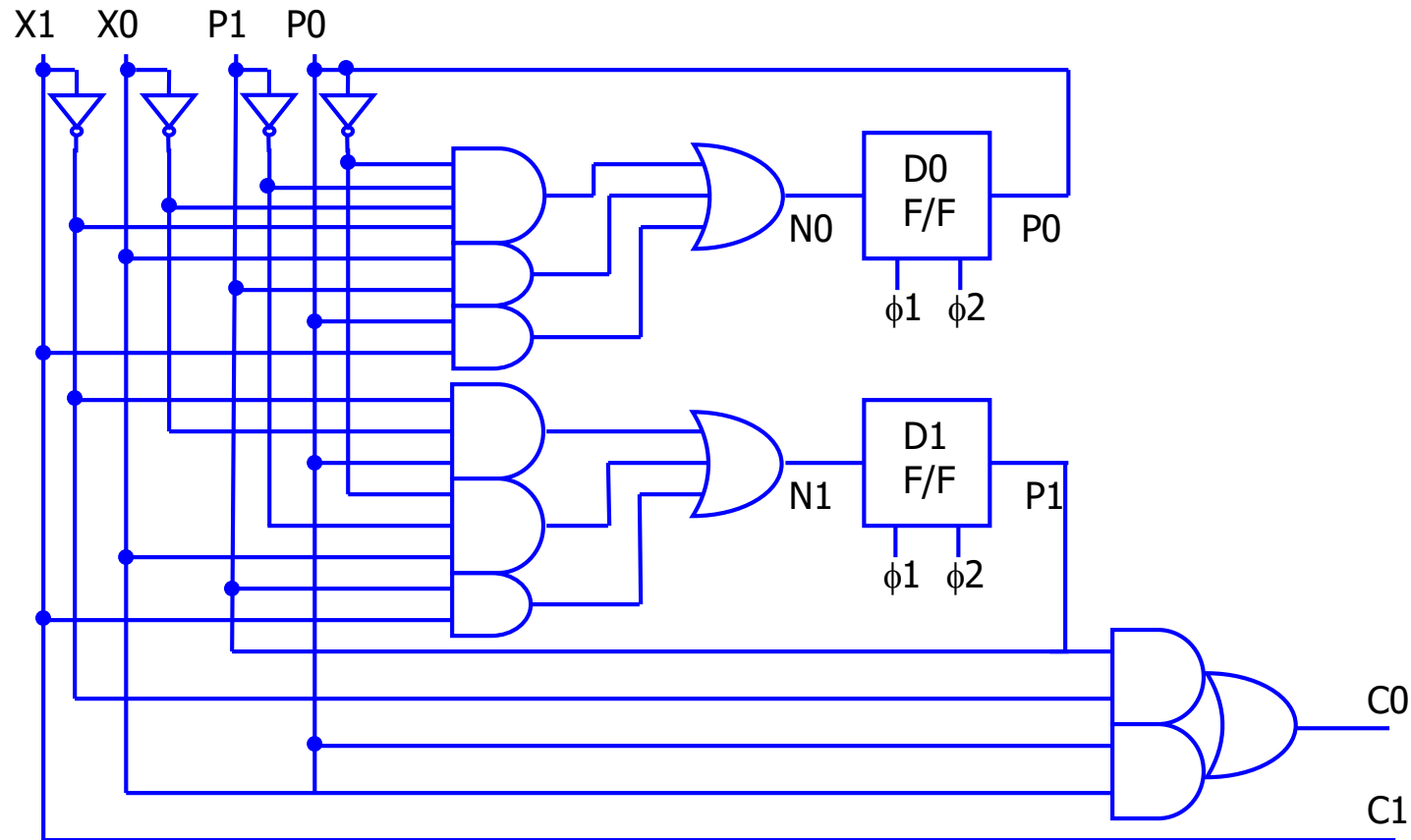
86

$$N1 = P0\overline{X1}\overline{X0} + \overline{P1}\overline{P0}X0 + P1X1$$

$$N0 = \overline{P1}\overline{P0}X1\overline{X0} + P1X0 + P0X1$$

$$C1 = X1$$

$$C0 = P1\overline{X1} + P0X0$$



# Example: Vending Machine FSM

General Machine Concept:

deliver package of gum after 15 cents deposited

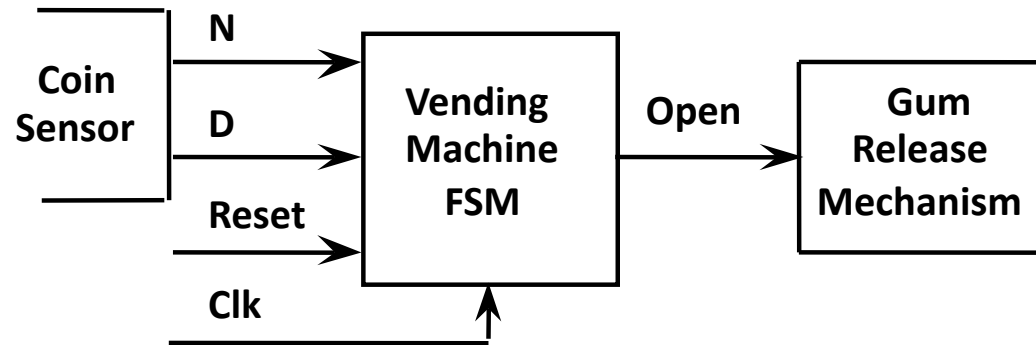
single coin slot for dimes, nickels

no change

Step 1. *Understand the problem:*

Draw a picture!

*Block Diagram*



# Vending Machine Example

**Step 2. Map into more suitable abstract representation**

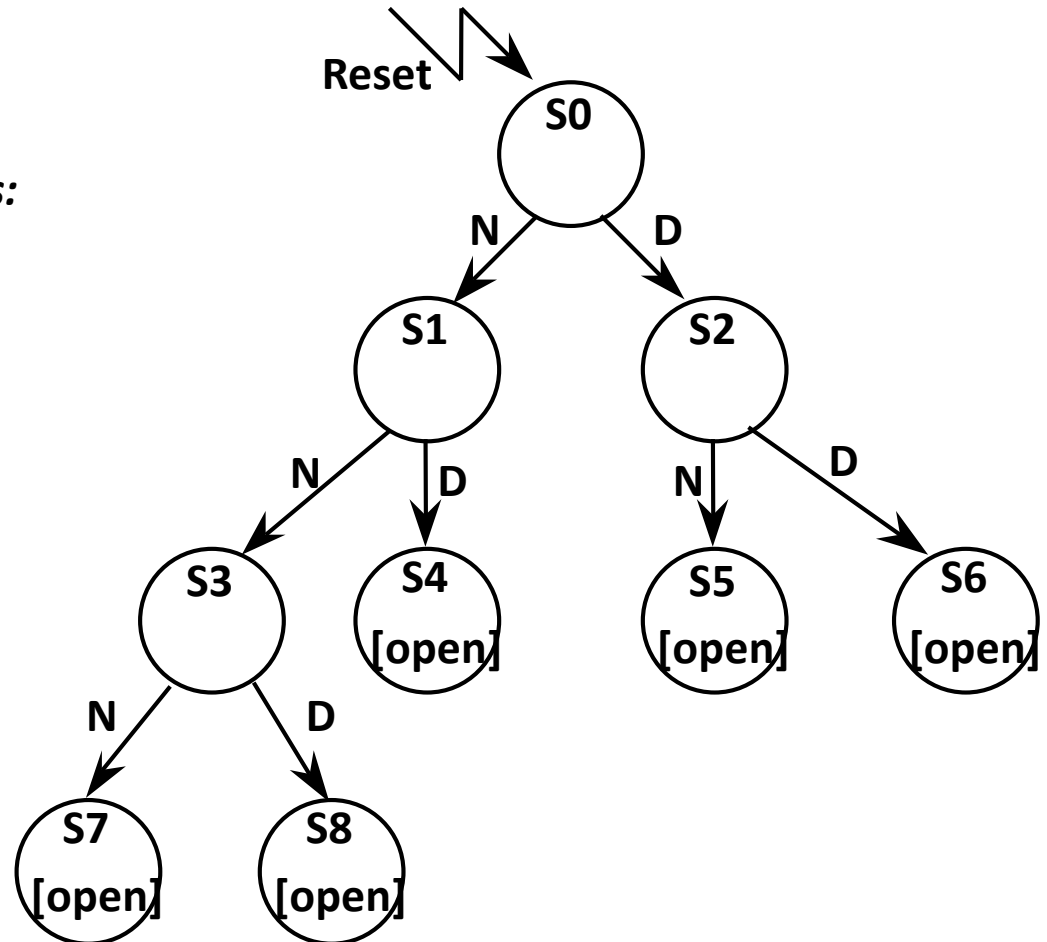
***Tabulate typical input sequences:***

three nickels  
nickel, dime  
dime, nickel  
two dimes  
two nickels, dime

***Draw state diagram:***

Inputs: N, D, reset

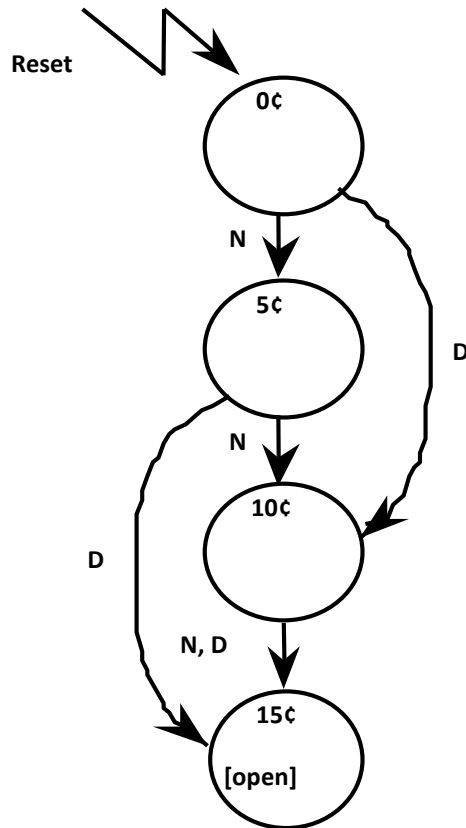
Output: open





# Vending Machine Example

## Step 3: State Minimization



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	X	X
5¢	0	0	5¢	0
	0	1	10¢	0
	1	0	15¢	0
	1	1	X	X
10¢	0	0	10¢	0
	0	1	15¢	0
	1	0	15¢	0
	1	1	X	X
15¢	X	X	15¢	1

Symbolic State Table

# Vending Machine Example

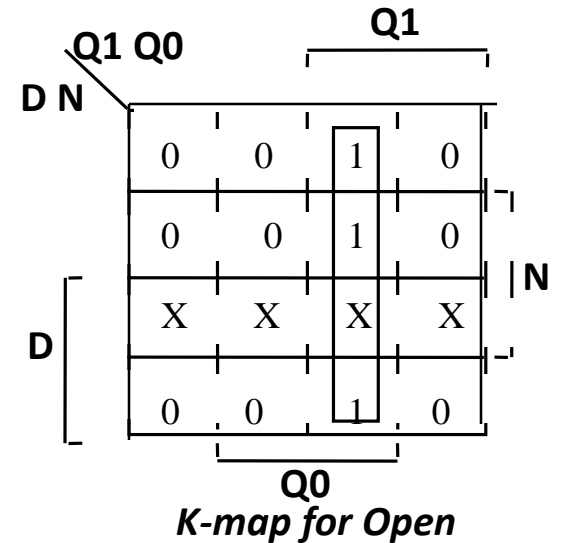
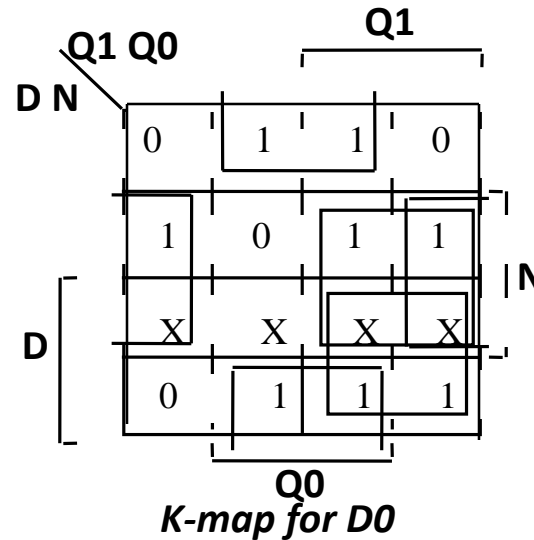
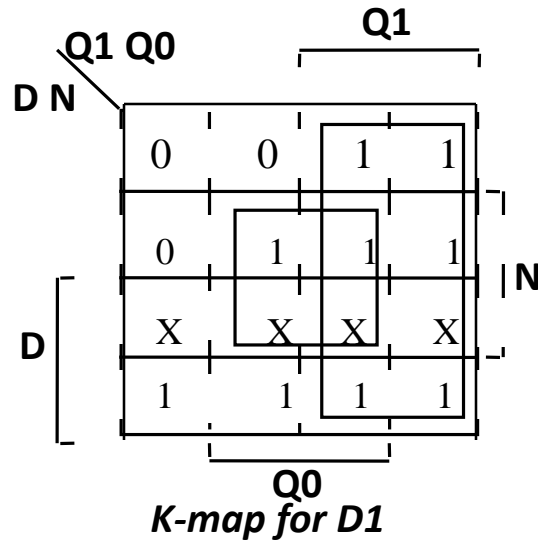
## Step 4: State Encoding

Present State		Inputs		Next State		Output
$Q_1$	$Q_0$	D	N	$D_1$	$D_0$	Open
0	0	0	0	0	0	0
		0	1	0	1	0
		1	0	1	0	0
		1	1	X	X	X
0	1	0	0	0	1	0
		0	1	1	0	0
		1	0	1	1	0
		1	1	X	X	X
1	0	0	0	1	0	0
		0	1	1	1	0
		1	0	1	1	0
		1	1	X	X	X
1	1	0	0	1	1	1
		0	1	1	1	1
		1	0	1	1	1
		1	1	X	X	X

# Vending Machine Example

Step 5. Choose FFs for implementation

D FF easiest to use



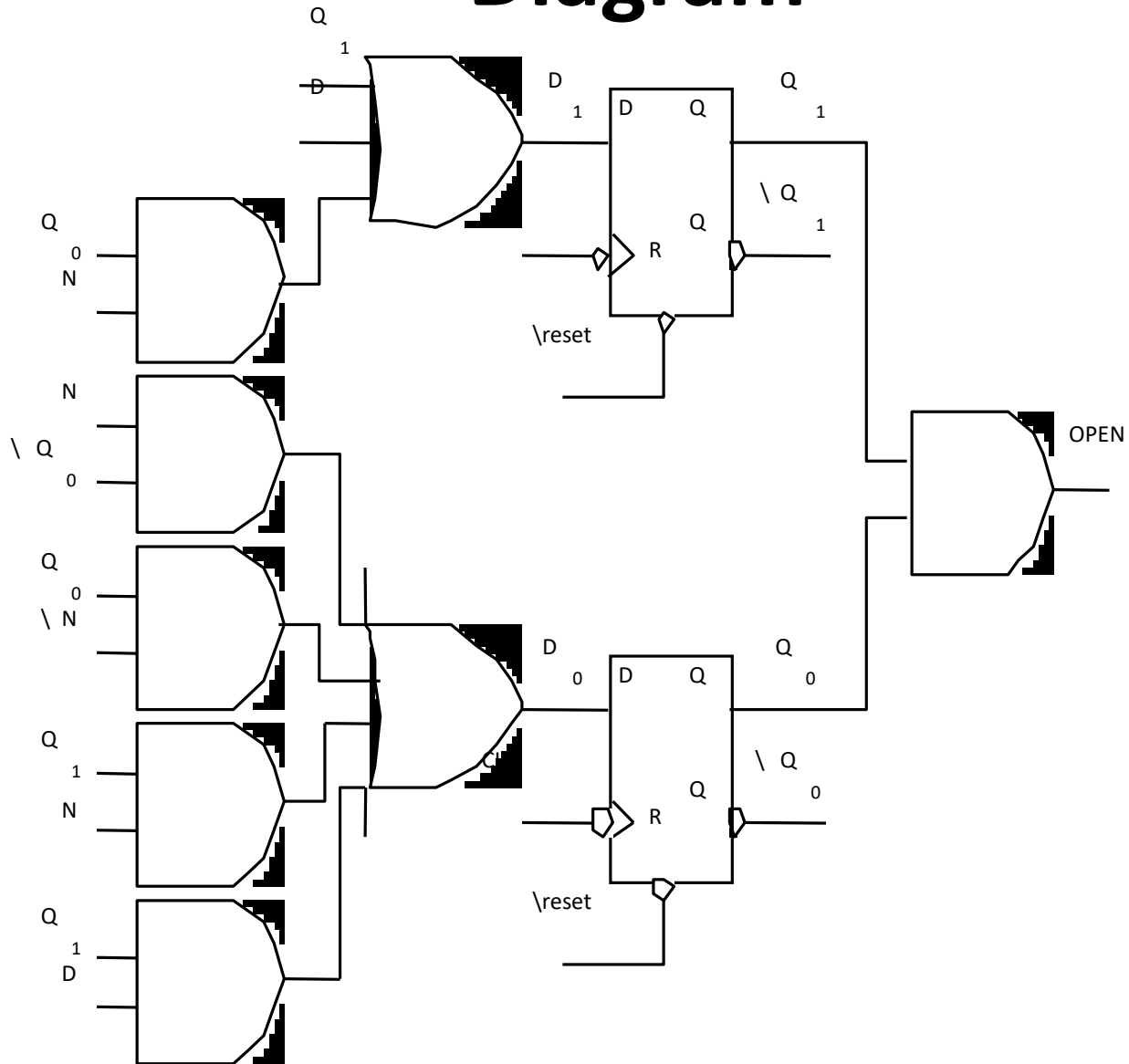
$$D1 = Q1 + D + Q0 N$$

$$D0 = \overline{N} \overline{Q0} + Q0 \overline{N} + Q1 \overline{N} + Q1 D$$

$$OPEN = Q1 Q0$$

8 Gates

# Diagram



# Alternative State Machine Representations

## *Why State Diagrams Are Not Enough*

Not flexible enough for describing very complex finite state machines

Not suitable for gradual refinement of finite state machine

Do not obviously describe an *algorithm*: that is, well specified sequence of actions based on input data

algorithm = sequencing + data manipulation

separation of control and data

## *Gradual shift towards program-like representations:*

- Algorithmic State Machine (ASM) Notation
- Hardware Description Languages (e.g., VHDL)

# Alternative State Machine Representations

## *Algorithmic State Machine (ASM) Notation*

Three Primitive Elements:

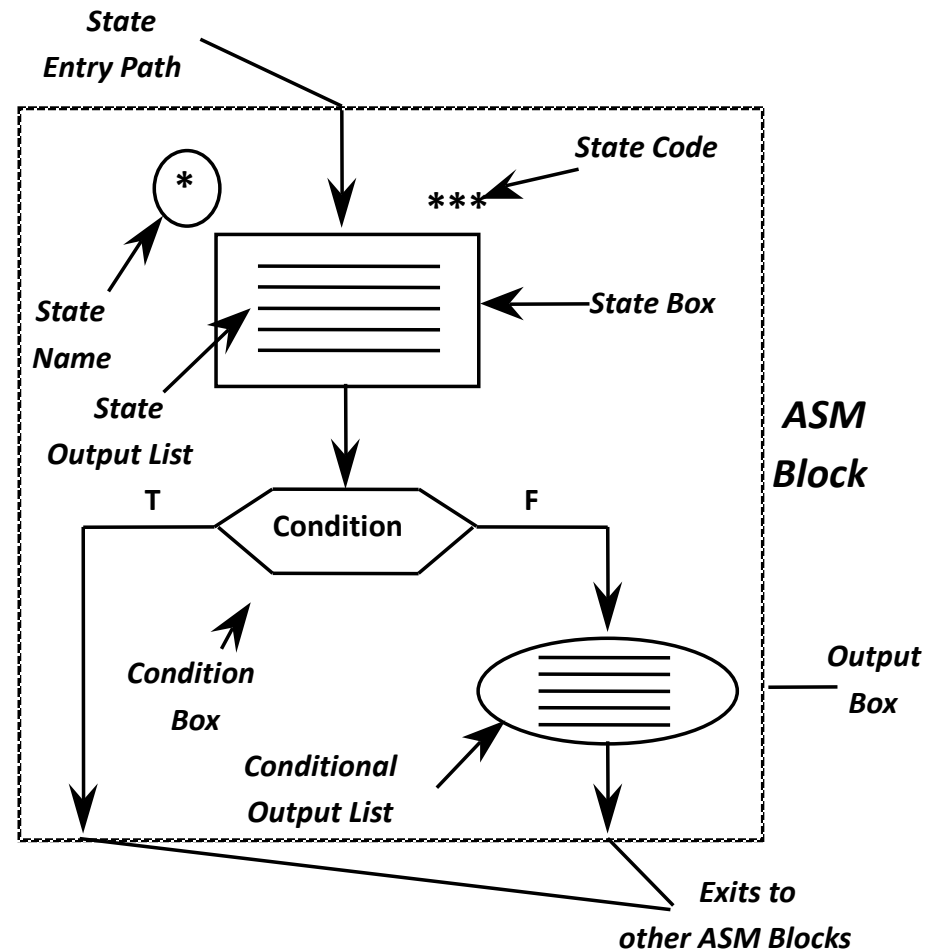
- State Box
- Decision Box
- Output Box

*State Machine in one state block per state time*

*Single Entry Point*

*Unambiguous Exit Path for each combination of inputs*

*Outputs asserted high (.H) or low (.L); Immediate (I) or delayed til next clock*



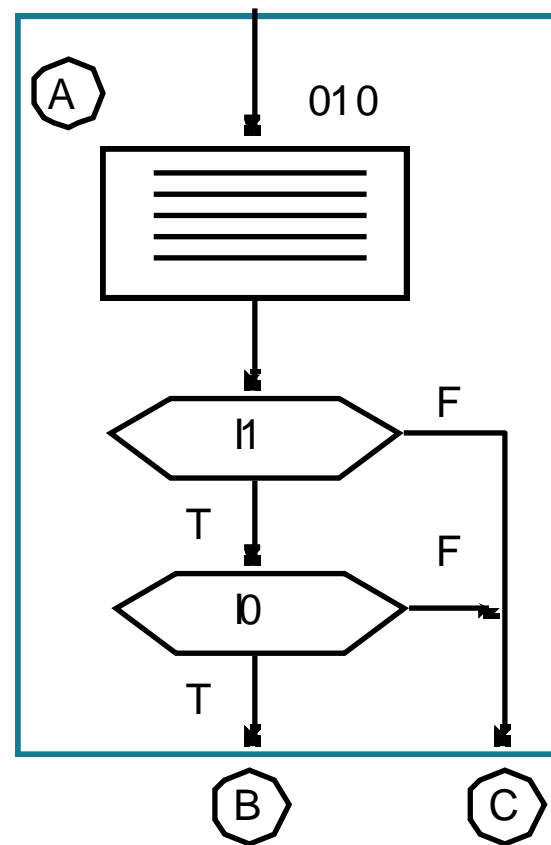
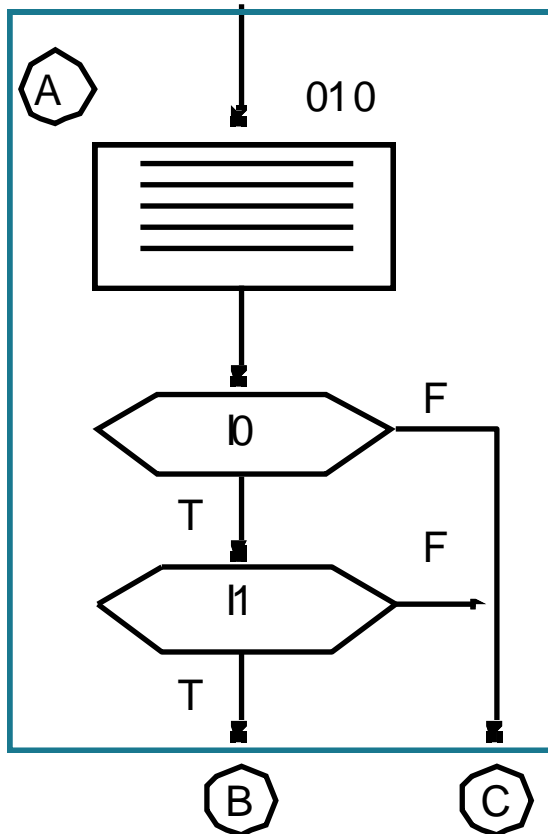
# ASM Notation

Condition Boxes:

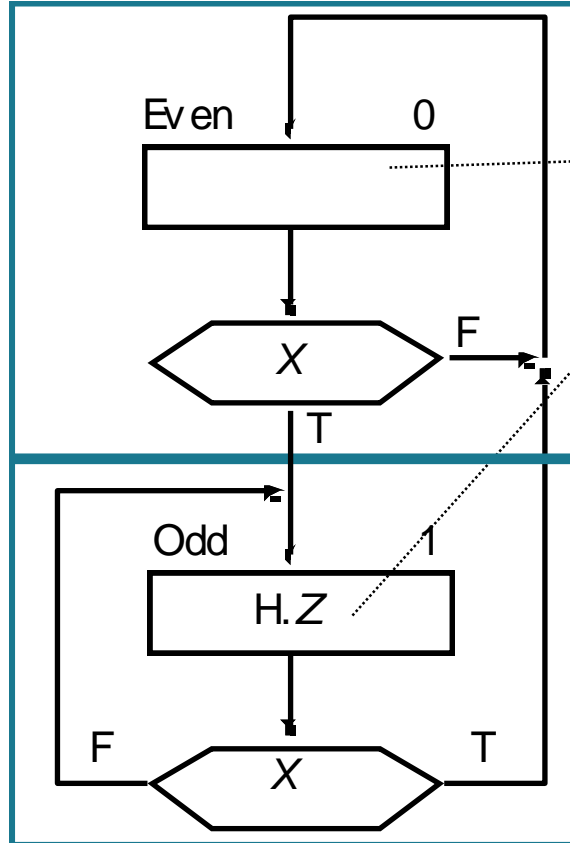
Ordering has no effect on final outcome

Equivalent ASM charts:

*A exits to B on  $(I0 \bullet I1)$  else exit to C*



# ASM Example: Parity Checker



Trace paths to derive  
state transition tables

Input X, Output Z

Nothing in output list implies Z not asserted

Z asserted in State Odd

Symbolic State Table:

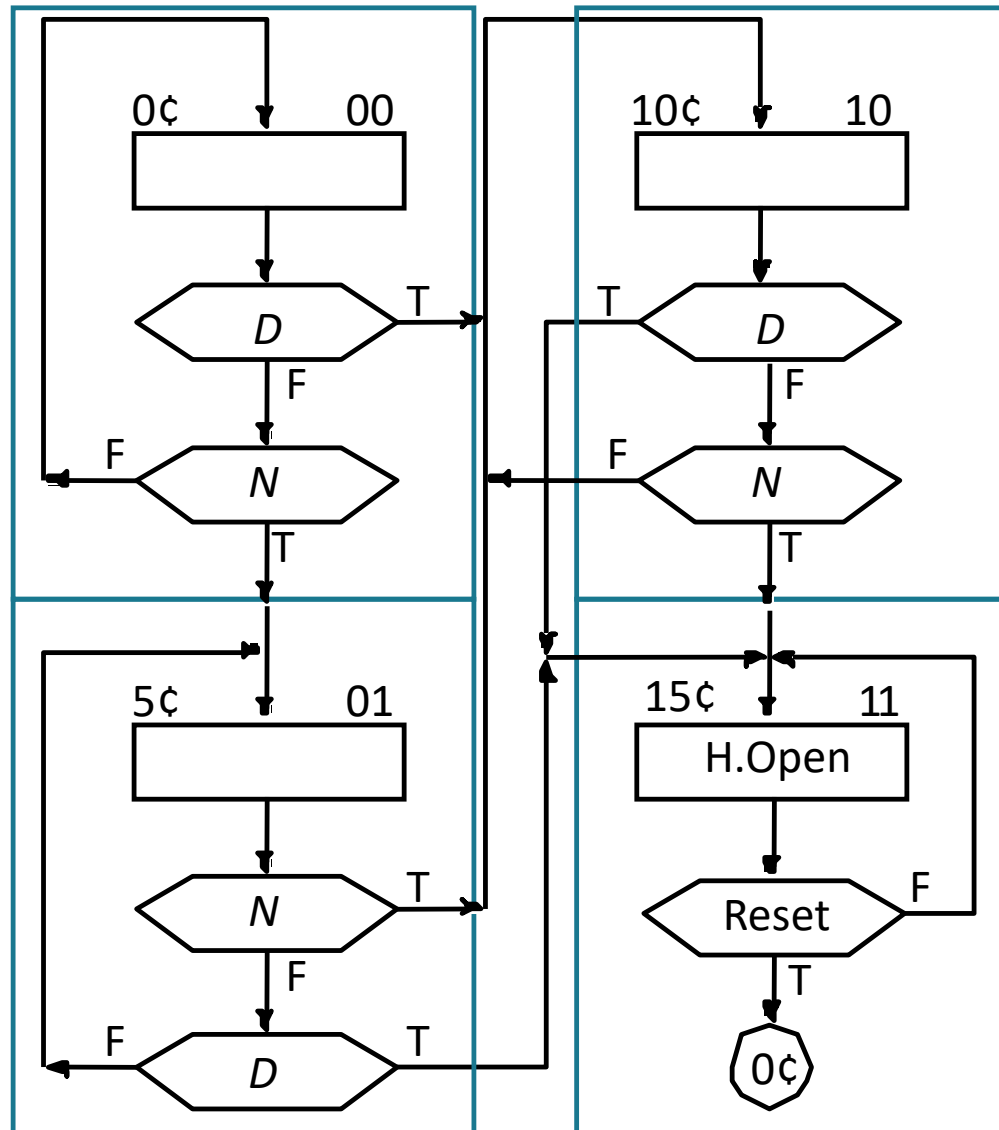
<i>Input</i>	<i>Present State</i>	<i>Next State</i>	<i>Output</i>
F	Even	Even	—
T	Even	Odd	—
F	Odd	Odd	A
T	Odd	Even	A

Encoded State Table:

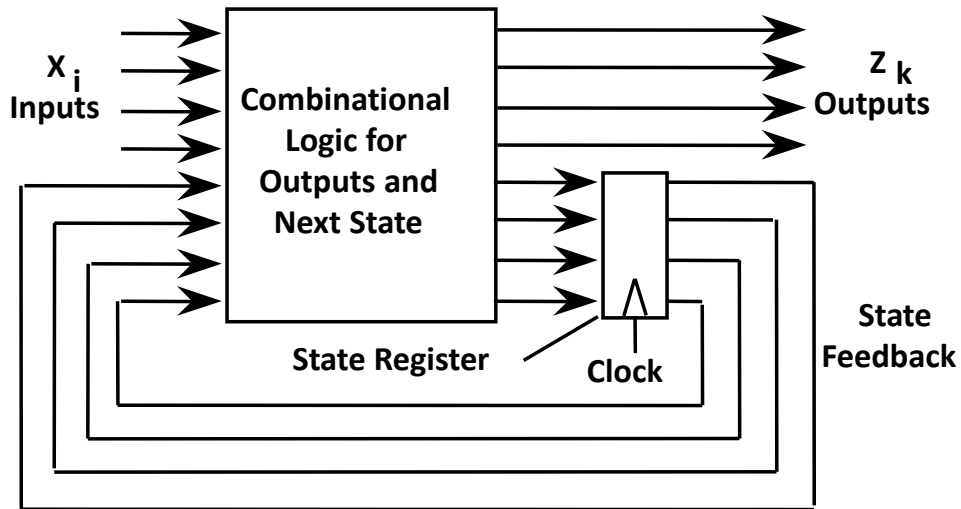
<i>Input</i>	<i>Present State</i>	<i>Next State</i>	<i>Output</i>
0	0	0	0
1	0	1	0
0	1	1	1
1	1	0	1



# ASM Chart: Vending Machine



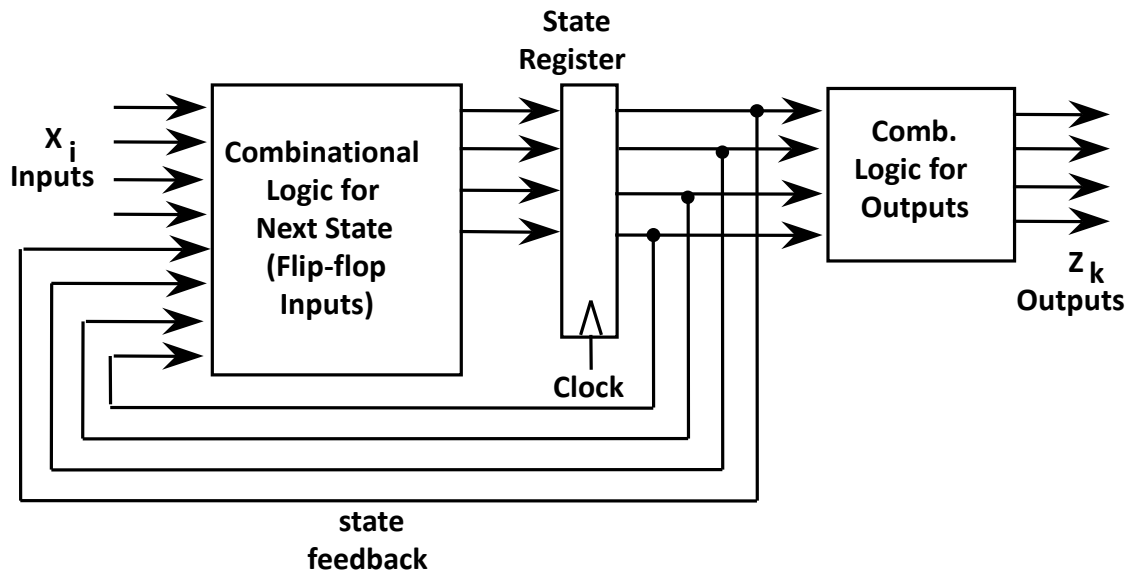
# Moore and Mealy Machine Design Procedure



## Moore Machine

*Outputs are function solely of the current state*

*Outputs change synchronously with state changes*



## Mealy Machine

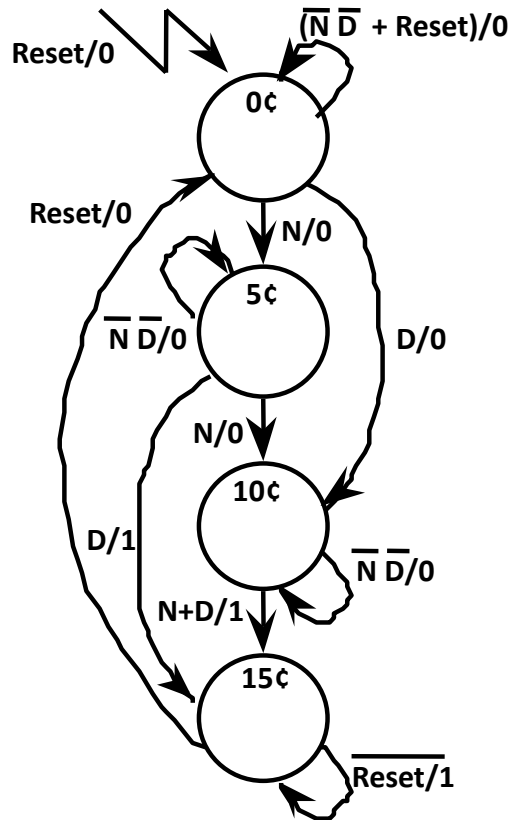
*Outputs depend on state AND inputs*

*Input change causes an immediate output change*

*Asynchronous signals*

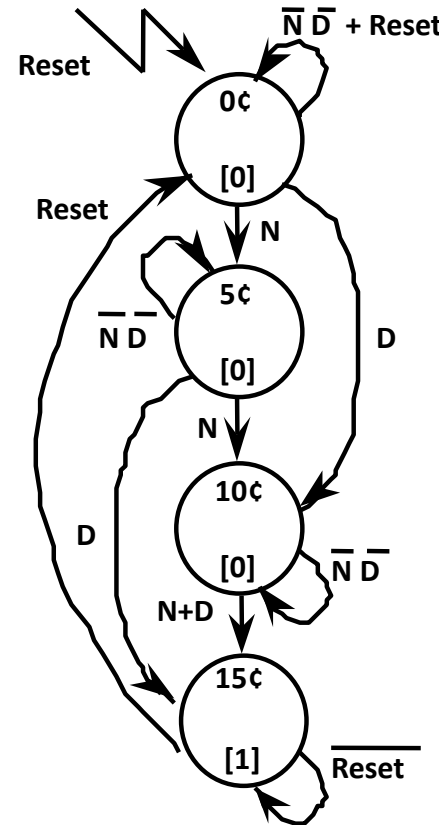
# Equivalence of Moore and Mealy Machines

**Moore Machine**



**Outputs are associated with State**

**Mealy Machine**



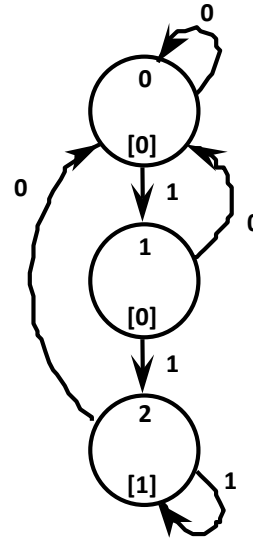
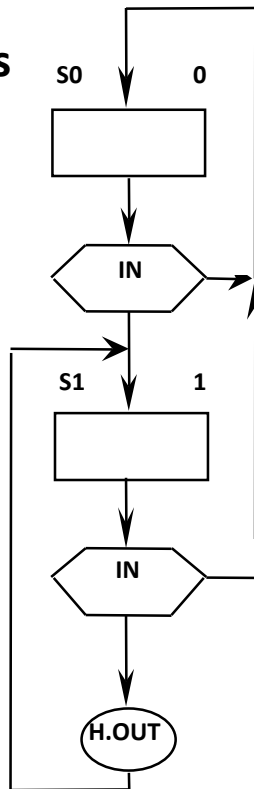
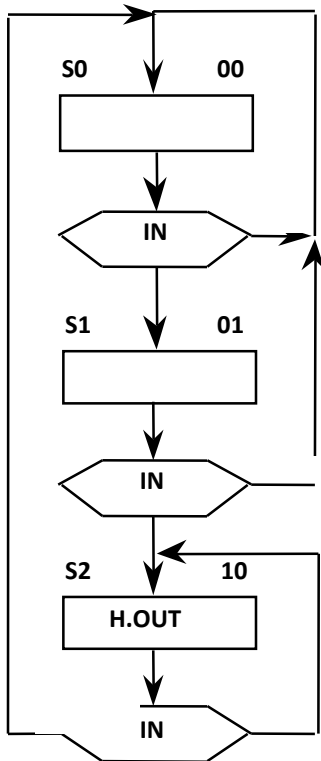
**Outputs are associated with Transitions**

# States vs Transitions

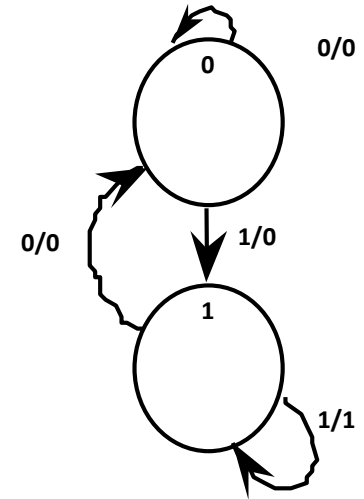
Mealy Machine typically has fewer states than Moore Machine  
for same output sequence

Same I/O behavior

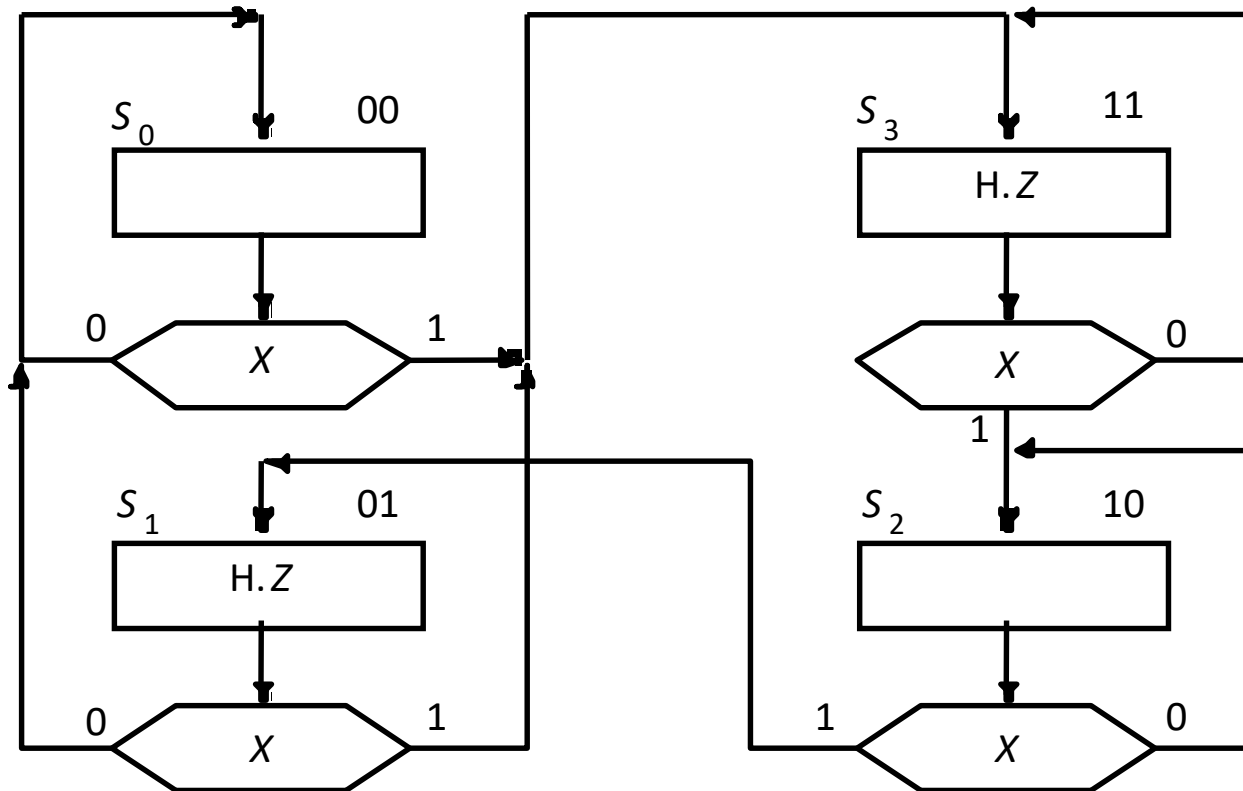
Different # of states



Equivalent  
ASM Charts



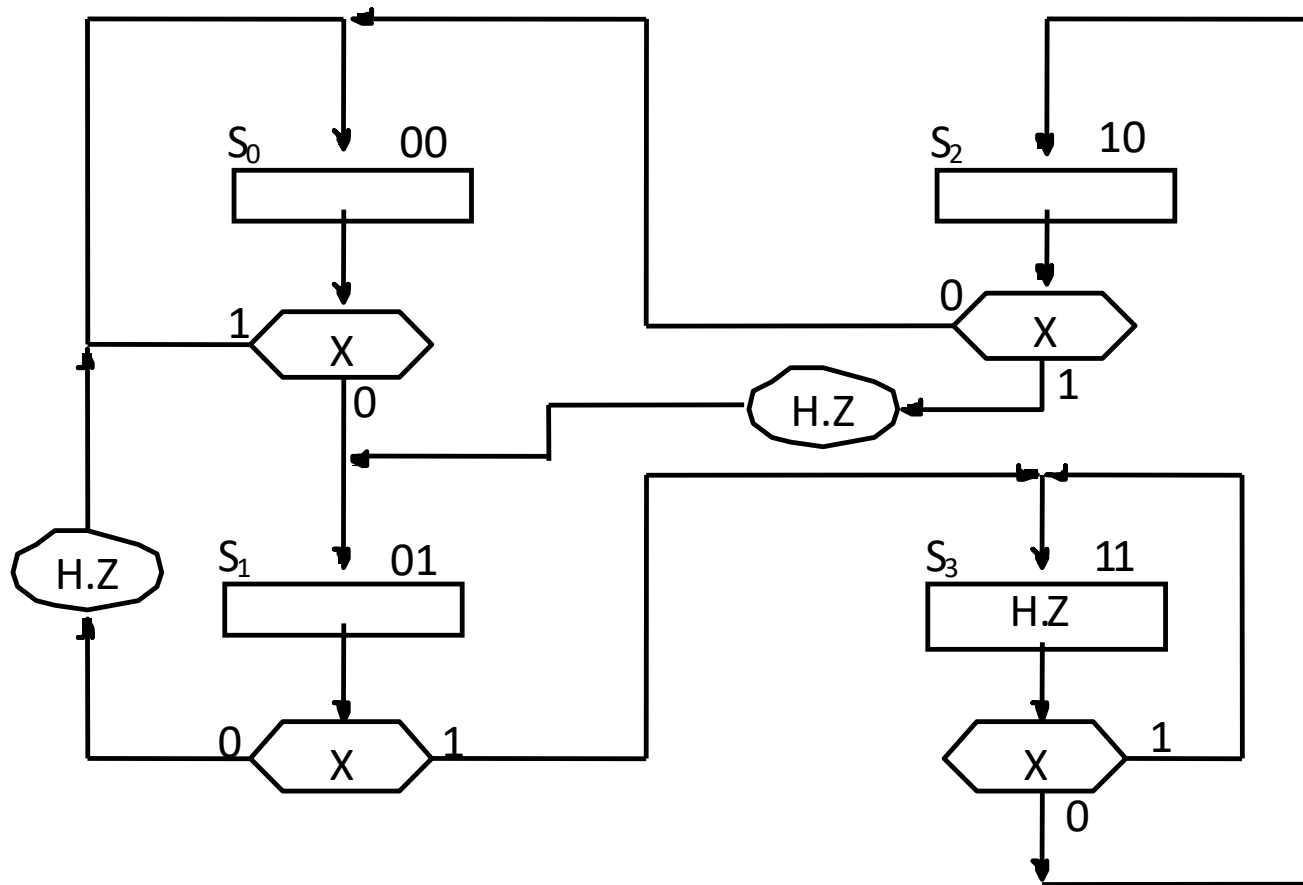
# Complete ASM Chart of Moore Machine



***Note: All Outputs Associated With State Boxes  
No Separate Output Boxes — Intrinsic in Moore Machines***

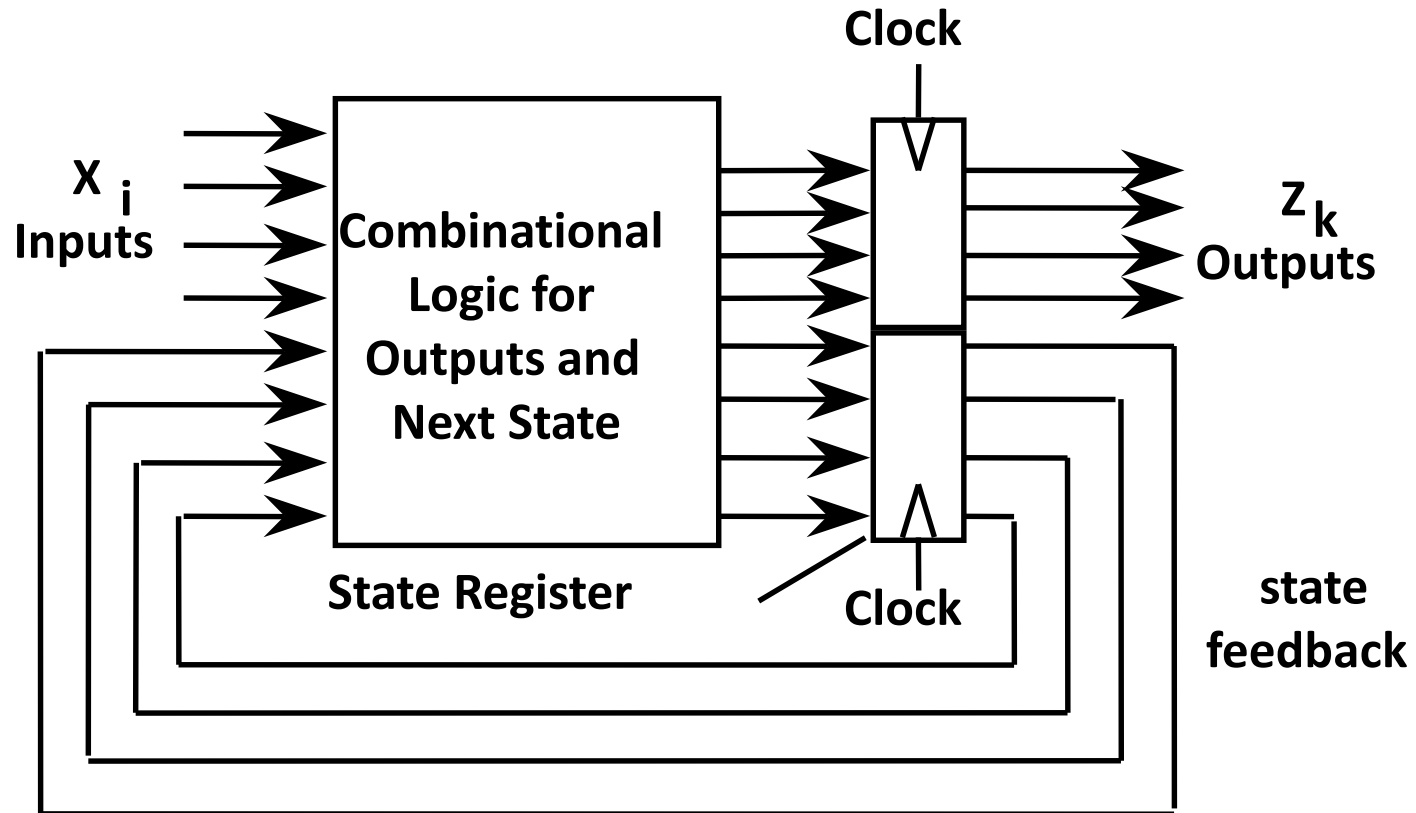
# ASM Chart of Mealy Machine

$S_0 = 00$ ,  $S_1 = 01$ ,  $S_2 = 10$ ,  $S_3 = 11$



**NOTE: Some Outputs in Output Boxes as well as State Boxes**  
This is intrinsic in Mealy Machine implementation

# Synchronous Mealy Machines



**latched state AND outputs**

**avoids glitchy outputs!**

# Traffic Light Controller

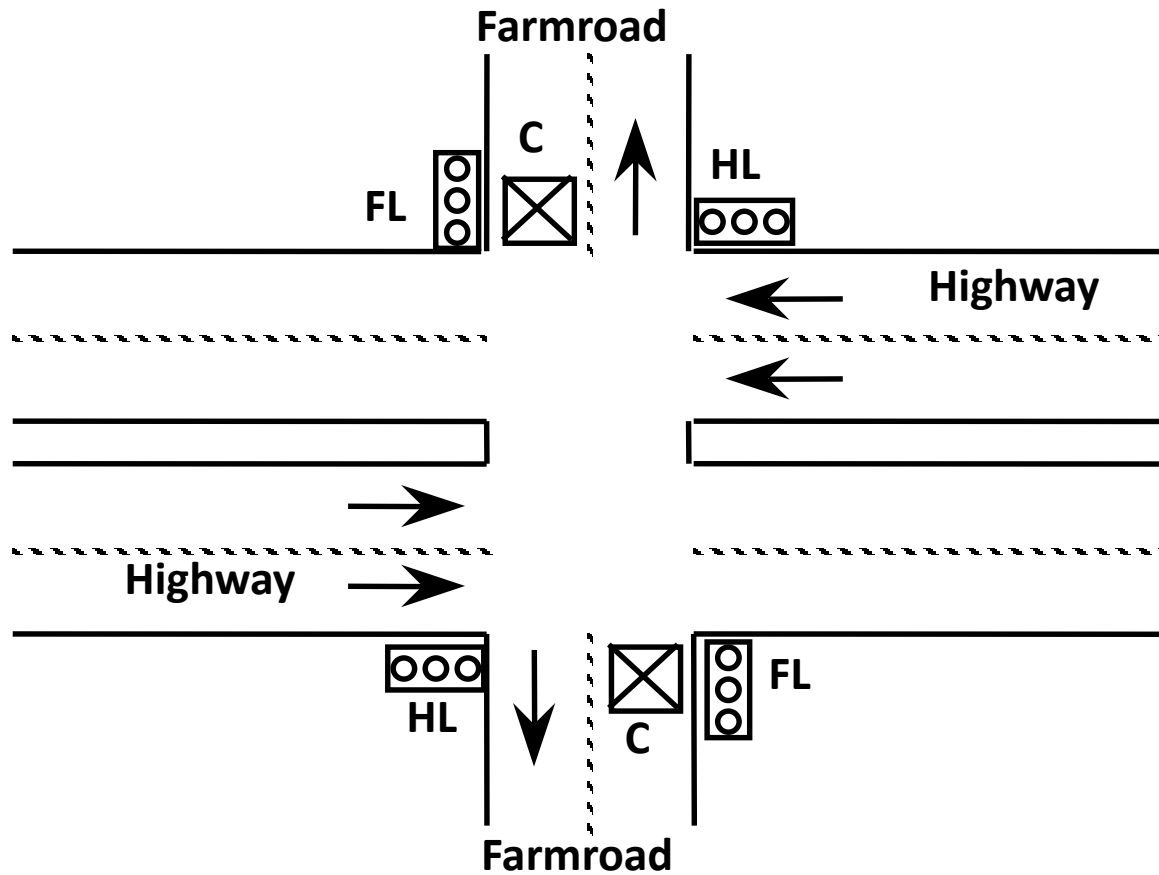
A busy highway is intersected by a little used farmroad. Detectors C sense the presence of cars waiting on the farmroad. With no car on farmroad, light remain green in highway direction. If vehicle on farmroad, highway lights go from Green to Yellow to Red, allowing the farmroad lights to become green. 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.



# Traffic Light Controller

Picture of Highway/Farmroad Intersection:



# 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)

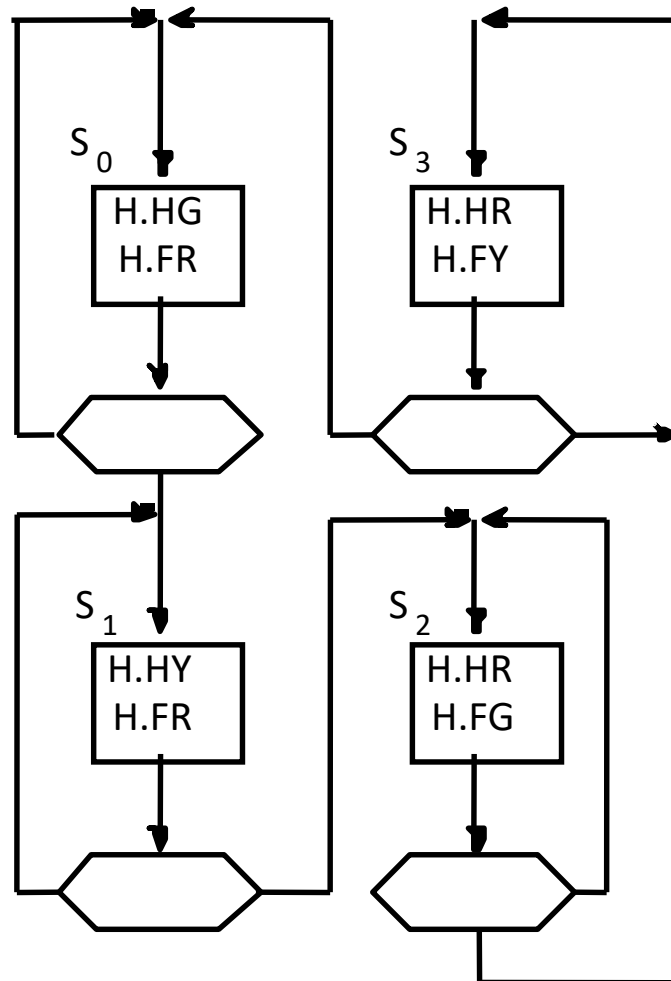
Farmroad green (highway red)

Farmroad yellow (highway red)

# Traffic Light Controller

Refinement of ASM Chart:

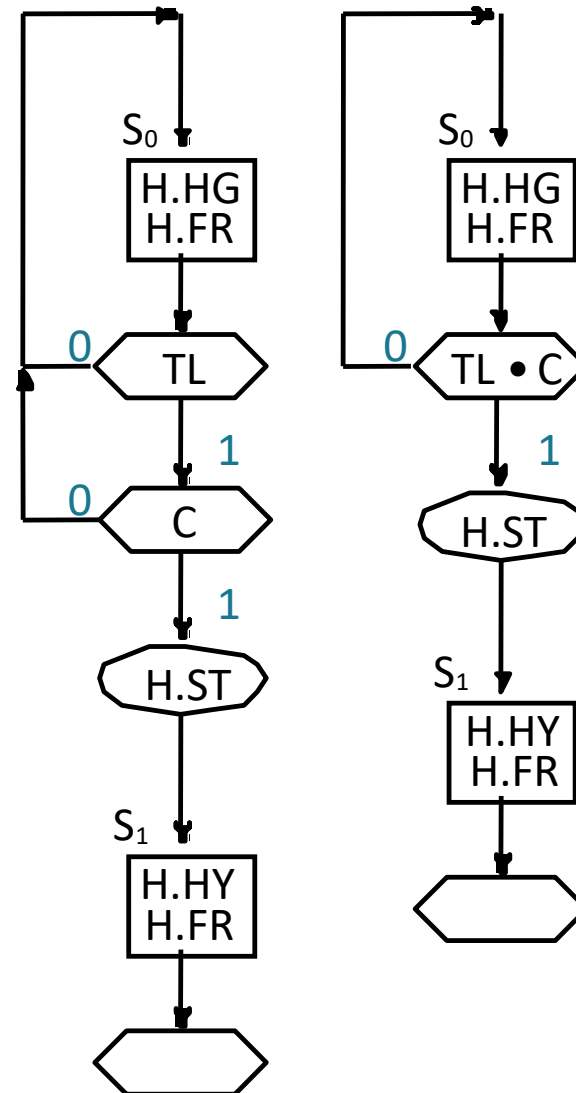
Start with basic sequencing and outputs:



# Traffic Light Controller

Determine Exit Conditions for S0:

Car waiting and Long Time Interval Expired- C • TL



*Equivalent ASM Chart Fragments*

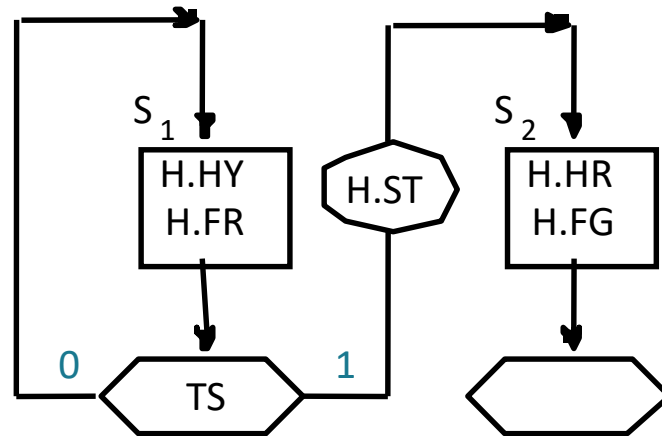
# Traffic Light Controller

**S1 to S2 Transition:**

**Set ST on exit from S0**

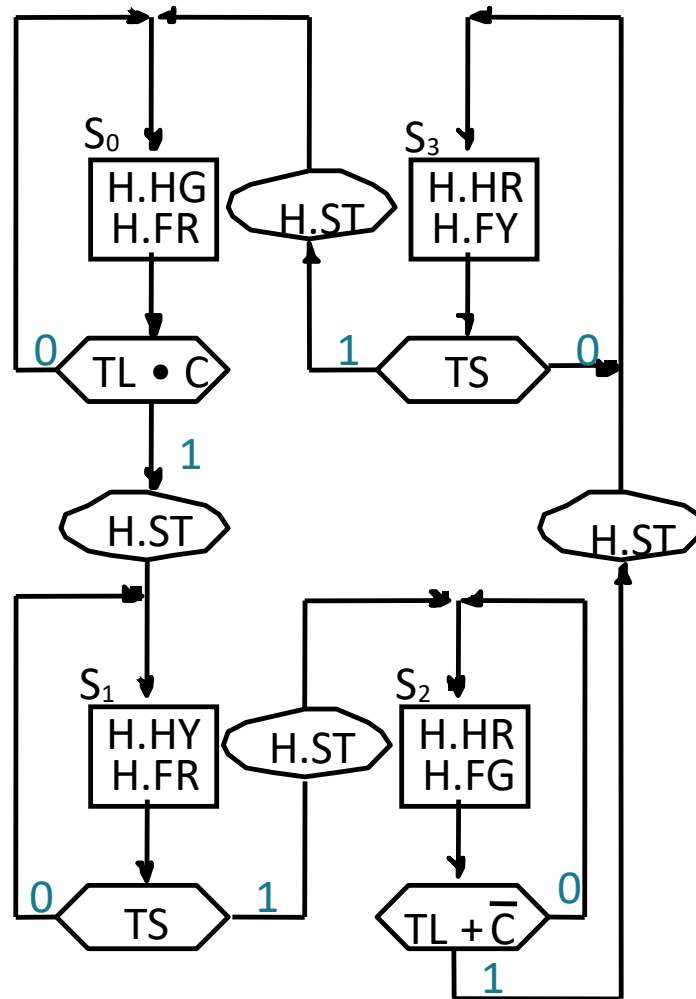
**Stay in S1 until TS asserted**

**Similar situation for S3 to S4 transition**



# Traffic Light Controller

**S2 Exit Condition: no car waiting OR long time interval expired**



**Complete ASM Chart for Traffic Light Controller**