



NCDC

NUST CHIP DESIGN CENTRE

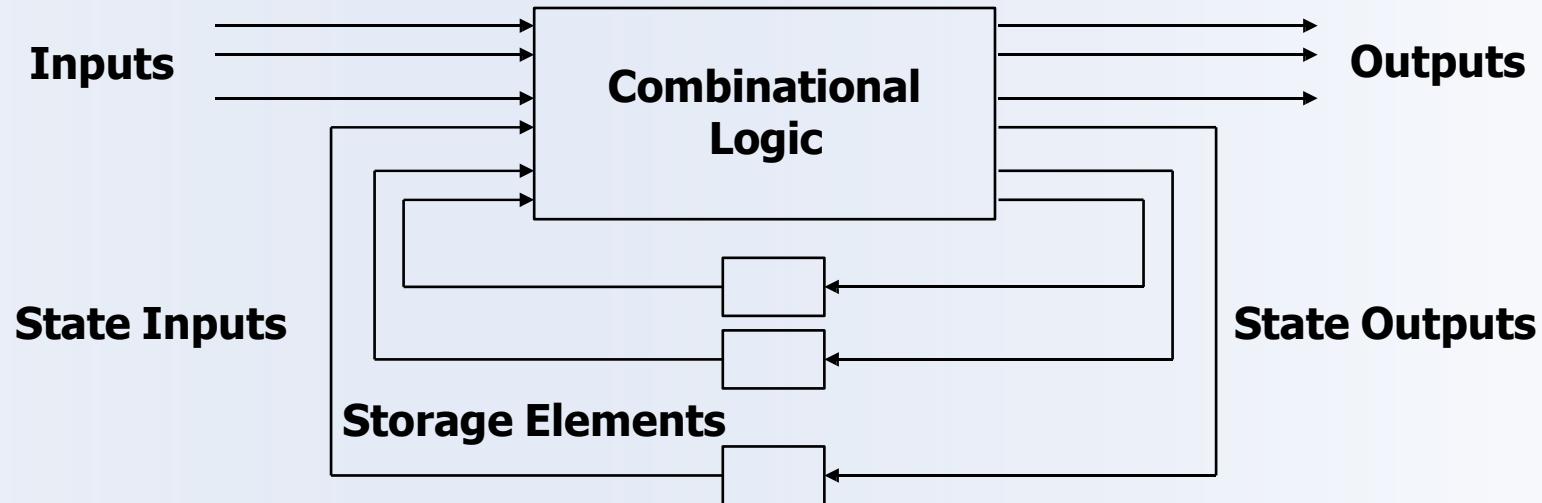
Digital Logic Design

FSM

Abstraction of state elements



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

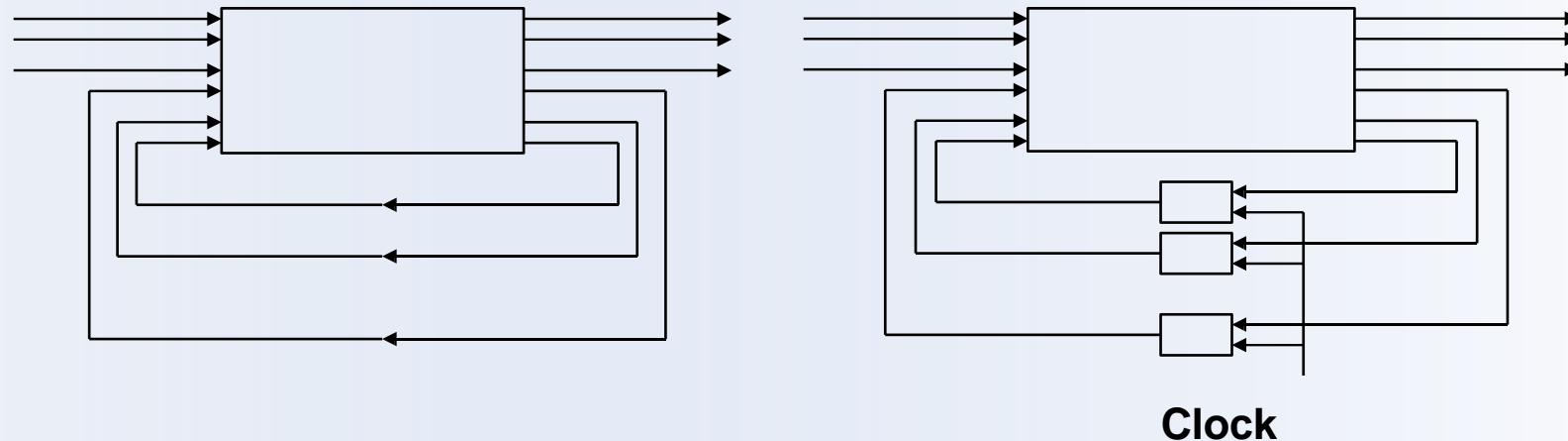


Now we are gonna break down a sequential logic system into two parts: combinational logic part and memory part (states). Multiple storage elements will be used to abstract the system states. The state, together with inputs, will determine the system operation: e.g. what is the next state, output?

Forms of sequential logic

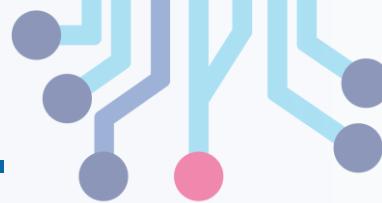


- Asynchronous sequential logic – state changes occur whenever state inputs change (elements may be simple wires or delay elements)
- Synchronous sequential logic – state changes occur in lock step across all storage elements (using a periodic waveform - the clock)



Asynchronous sequential logic circuits are operating without a clock, as shown on the left. The majority of sequential logic is synchronous logic circuits operating with clock signals, as illustrated on the right. With the clock signal, it is more convenient to control state transitions.

Concept of the State Machine



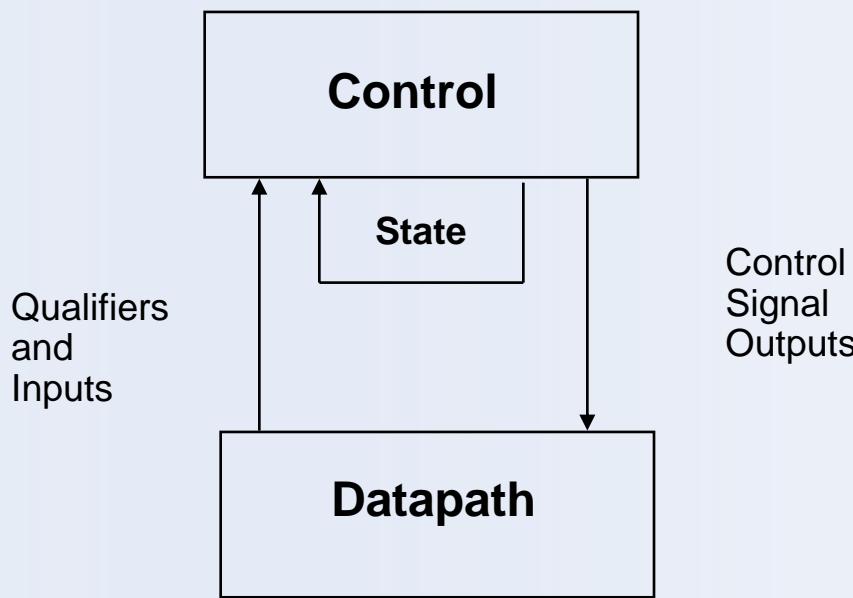
Computer Hardware = Datapath + Control

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

Qualifiers

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

Control

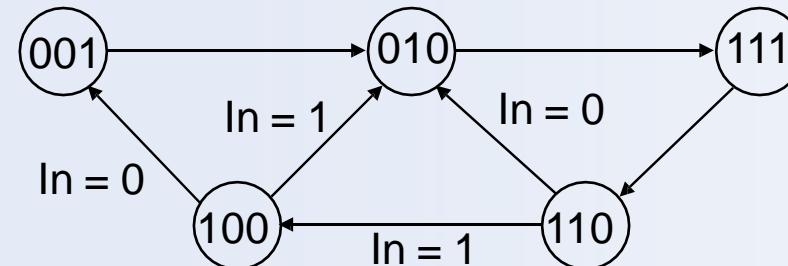


Finite state machine representations



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

- Sequential logic
 - sequences through a series of states
 - based on sequence of values on input signals
 - clock period defines elements of sequence



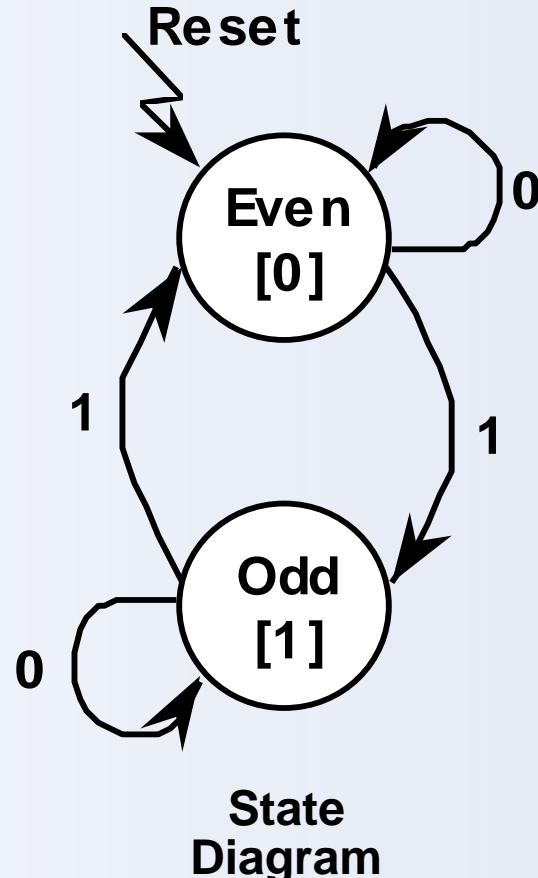
Now we use 5 states to describe the system behavior. The number in the circle represents the state. The state transition is determined by the current state and the input. Sometimes, the state transition takes place without an input, e.g. just by a clock tick.

Concept of the State Machine



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

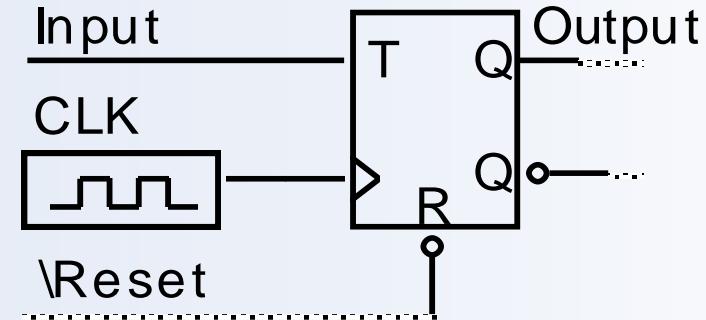
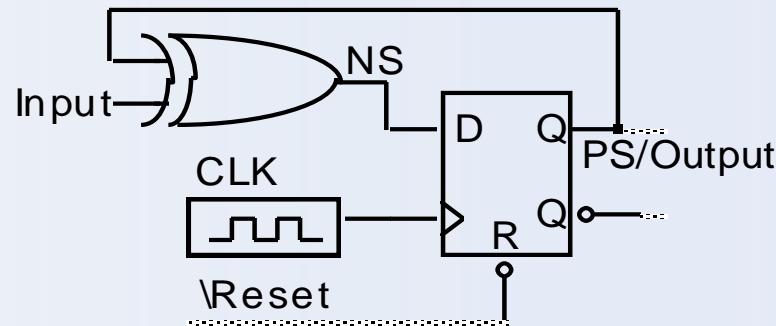
Concept of the State Machine



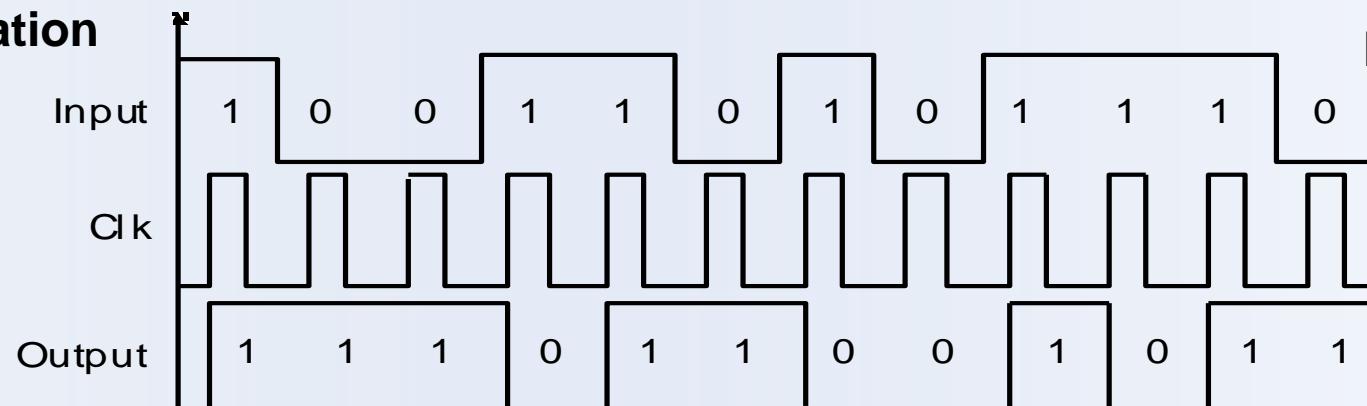
Example: Odd Parity Checker

Next State/Output Functions

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



D FF Implementation



T FF Implementation

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

Concept of State Machine



Timing: 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

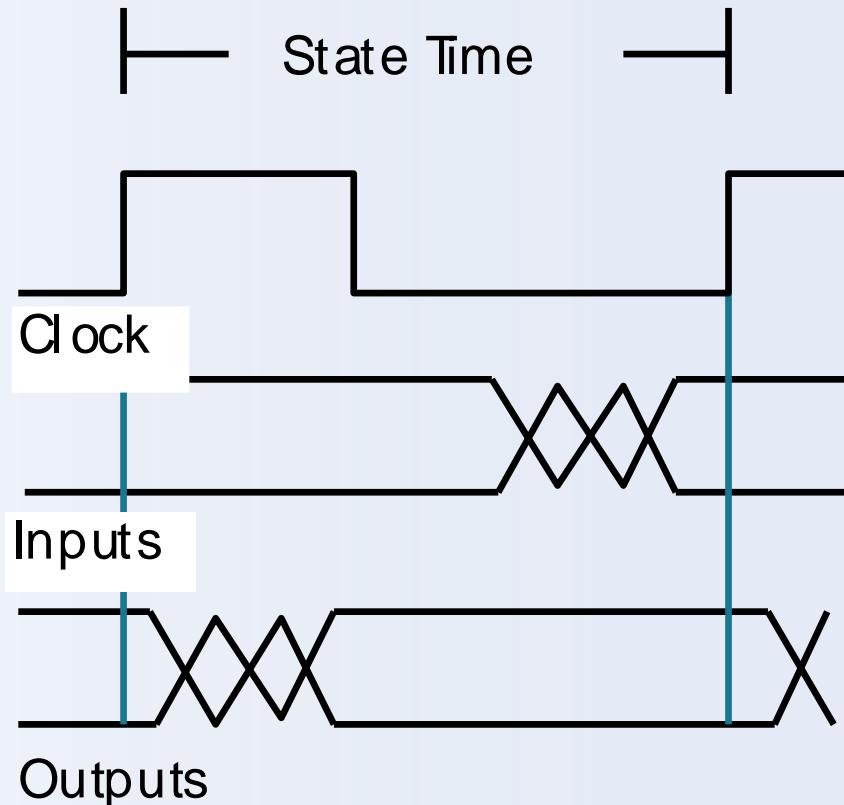
Synchronous signals take effect at the next clocking event

E.g. tri-state enable: effective immediately sync
counter clear: effective at next clock event

Concept of State Machine



Example: Positive Edge Triggered Synchronous System



On rising edge, inputs sampled
outputs, next state computed

After propagation delay, outputs and
next state are stable

Immediate Outputs:
affect datapath immediately
could cause inputs from datapath to change

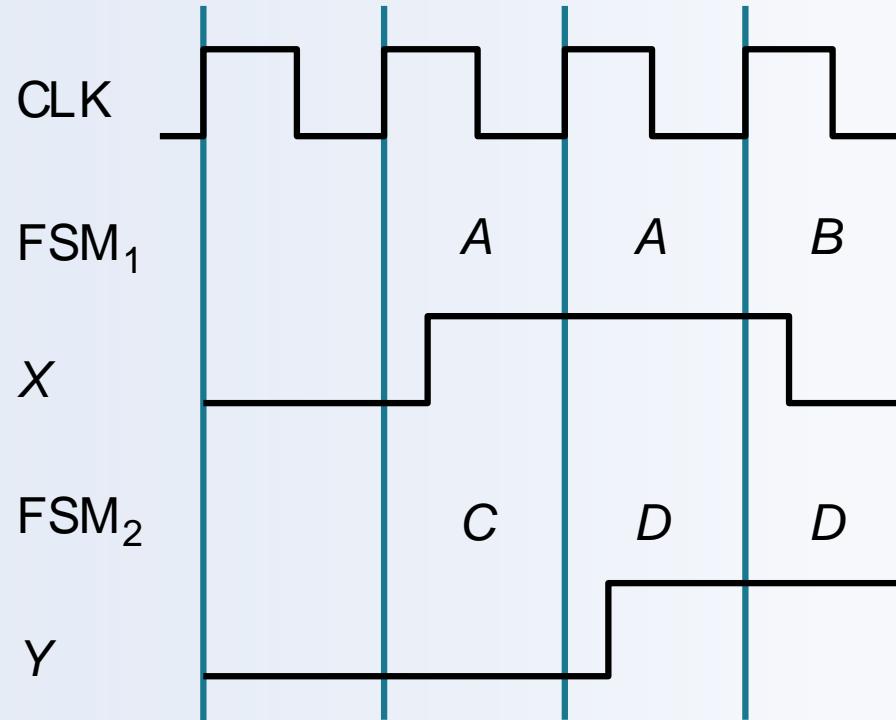
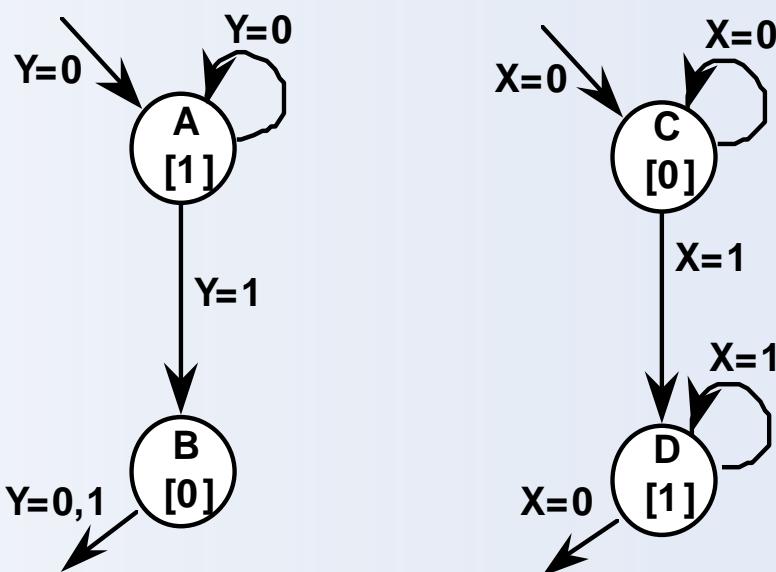
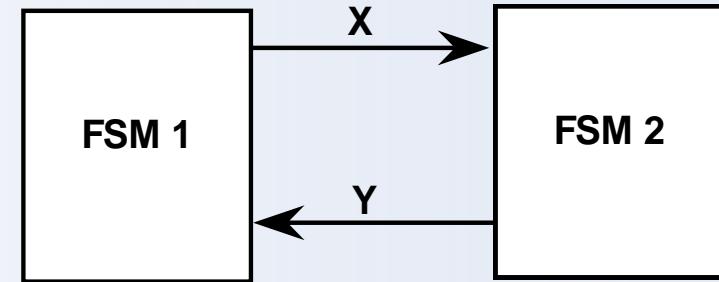
Delayed Outputs:
take effect on next clock edge
propagation delays must exceed hold times

Concept of the State Machine



Communicating State Machines

One machine's output is another machine's input



Machines advance in lock step

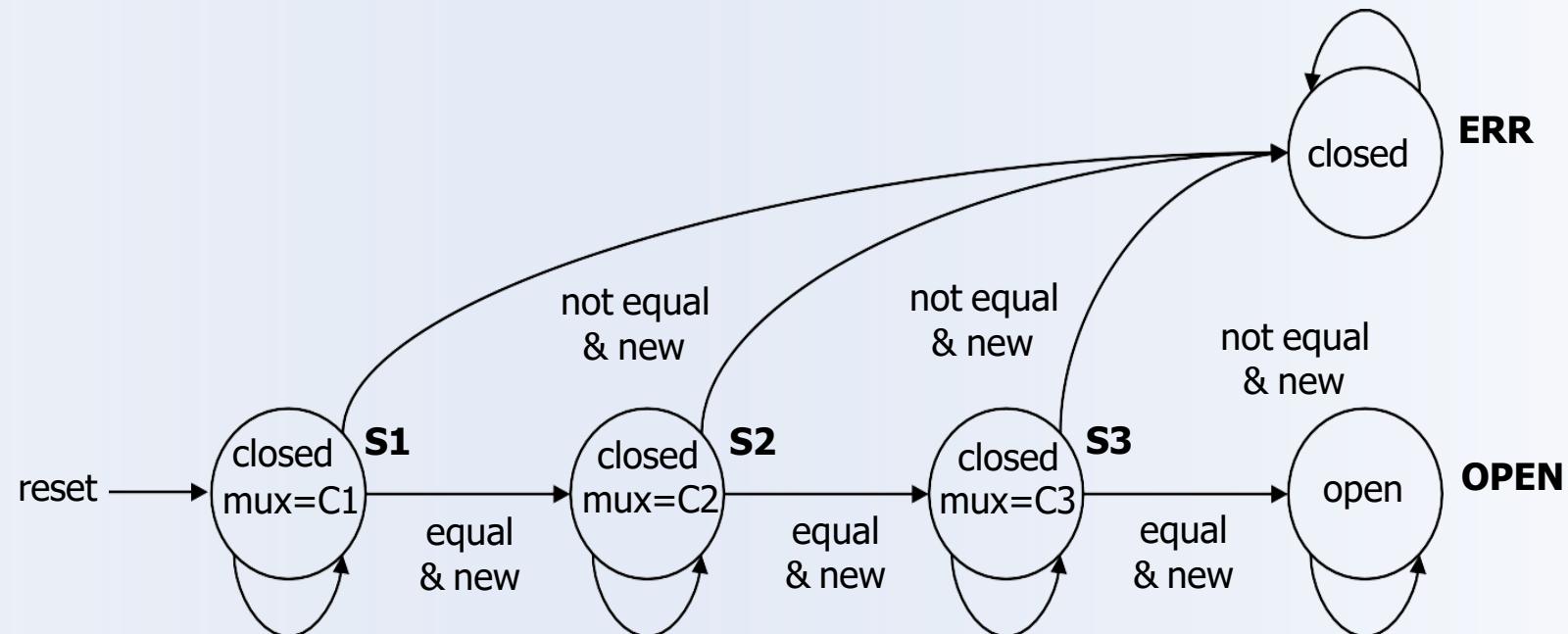
Initial inputs/outputs: X = 0, Y = 0

Example finite state machine diagram



■ Combination lock from introduction to course

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

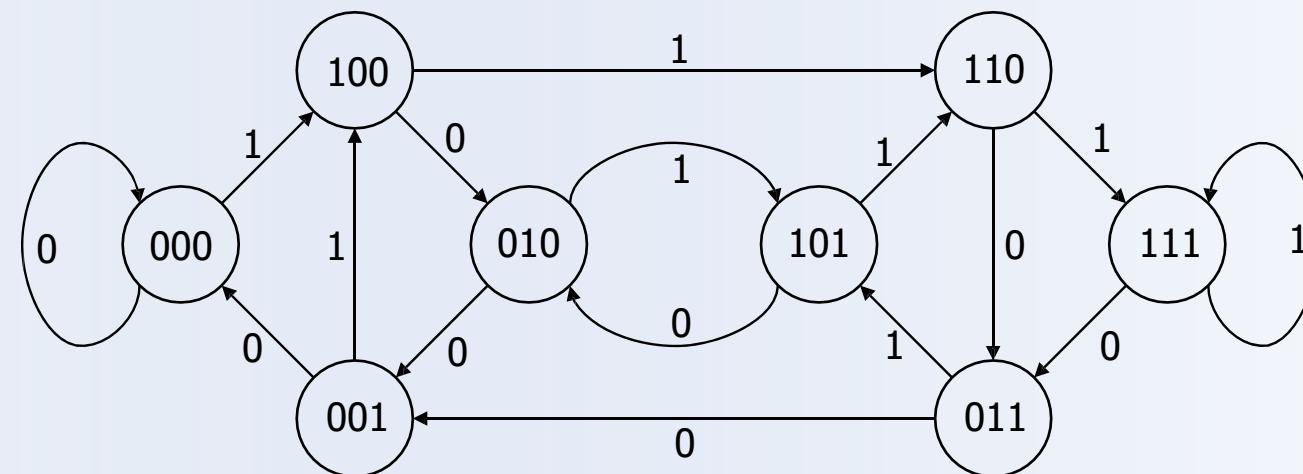
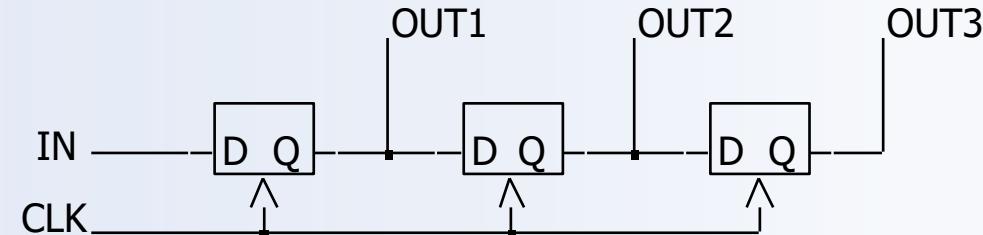


Can any sequential system be represented with a state diagram?



■ Shift register

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

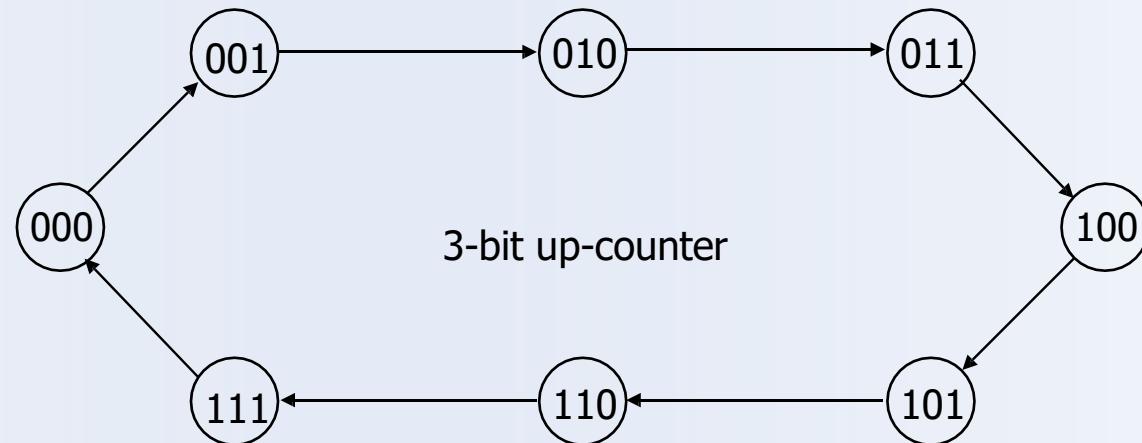


This state diagram shows all the possible states and transitions of a 3 bit shift register. In this case, the new state values are equal to output values. For example, when the system moves from 100 to 010, the output is 010. First of all, 3 bit will represent 8 states. And in each state, two kinds of input values are expected.

Counters are simple finite state machines



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

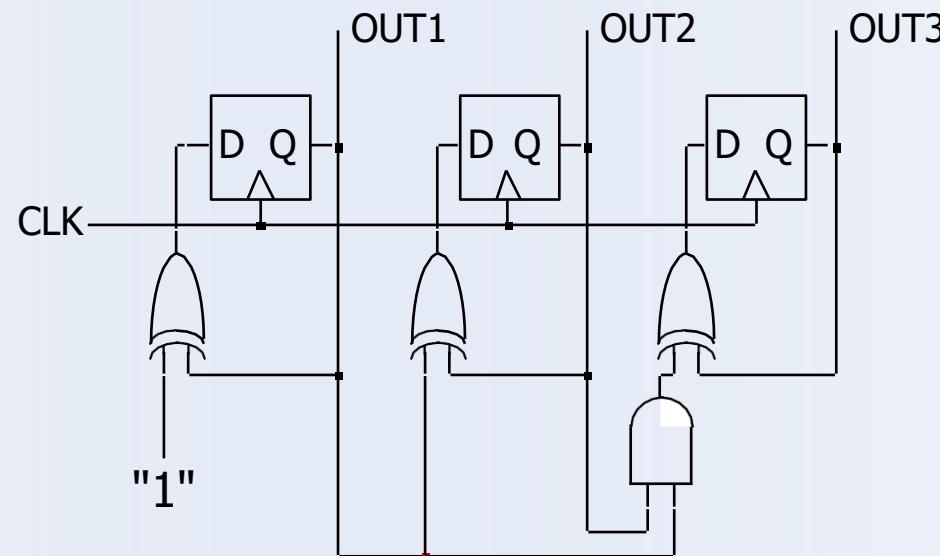


In the case of a 3bit up counter, every clock tick will make a transition without any inputs. In this case the numbers follow binary coding; hence it is called a binary counter.

How do we turn a state diagram into logic?



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



This one is a 3bit binary (up) counter. Recall that when all the lower bits are true, the higher bit should be toggled at the next clock tick.

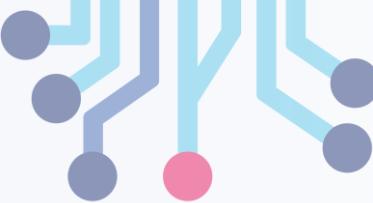
9-Step Design Approach



- Step 1: State/output table or diagram
 - Step 2: Minimize # of states if possible
 - Step 3: State variable assignment
 - Step 4: Transition/output table
 - Step 5: Choose a f/f type
 - Step 6: Excitation table
 - Step 7: Excitation equations
 - Step 8: Output equations
 - Step 9: Draw a logic diagram
- 


- State/output
diagram
- State/Output
table
- Excitation/Output
equations

Problem Statement

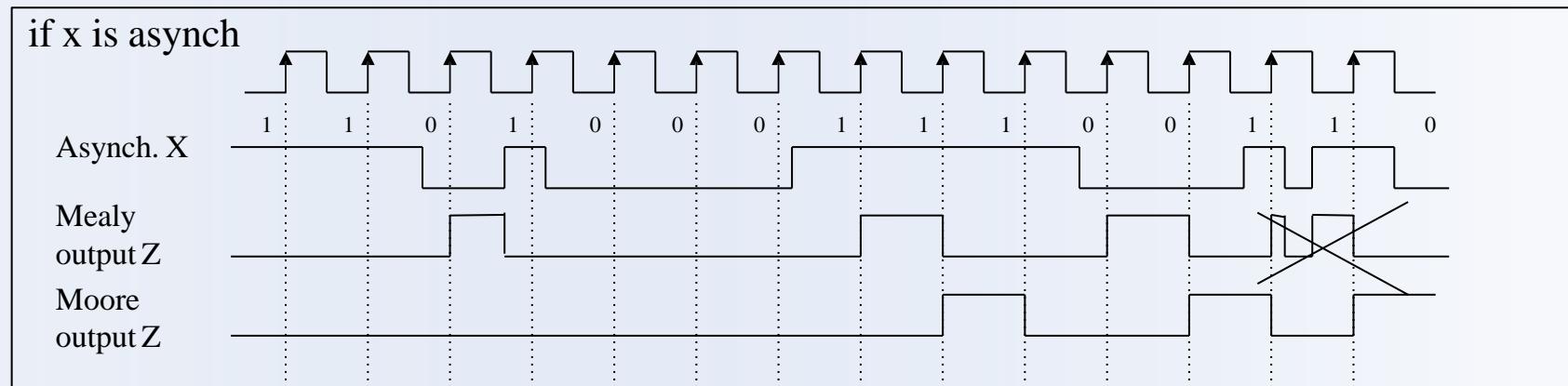
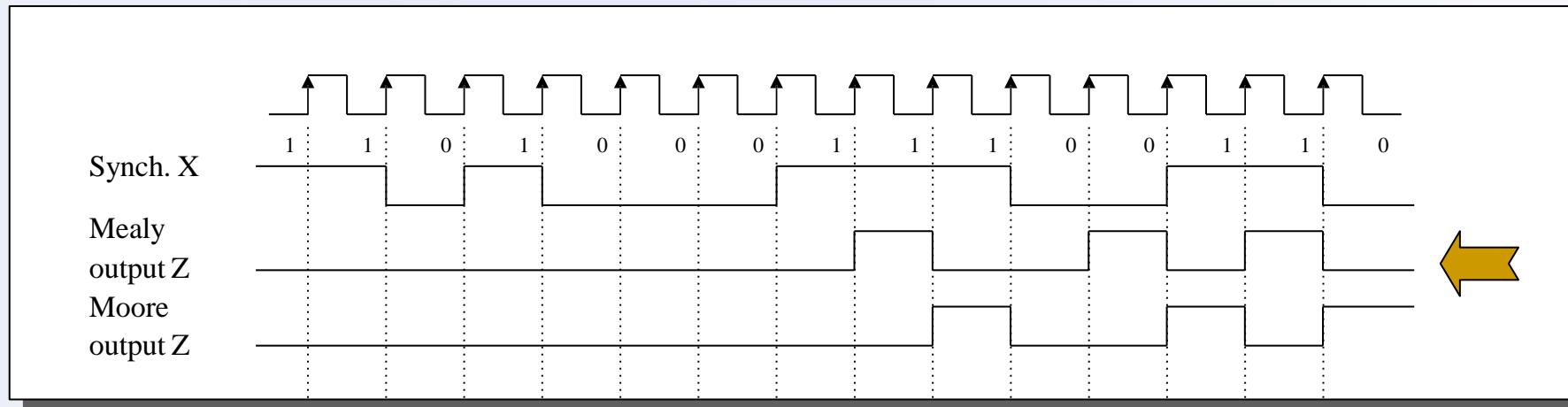


- Design a synchronous state-machine with one input X and an output Z. Whenever the input sequence consists of two consecutive 0's followed by two consecutive 1's or vice versa, the output will be 1. Otherwise, the output will be 0.

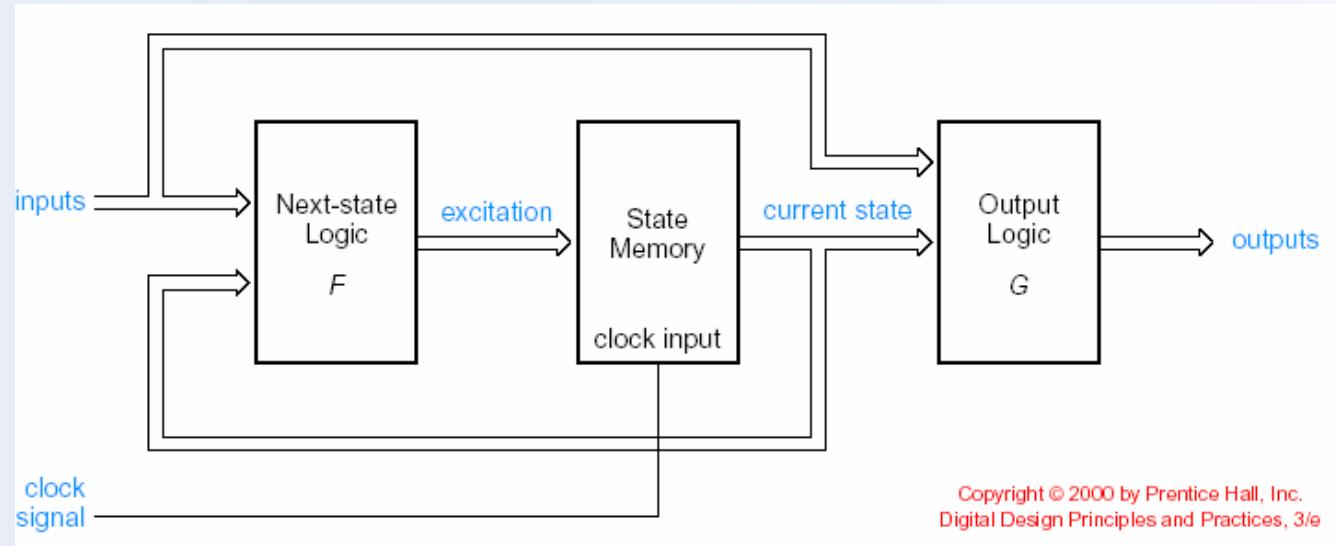
Problem Interpretation



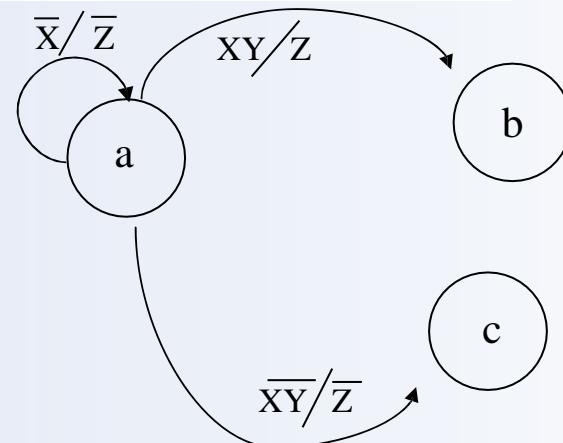
- Problem statement is sometimes ambiguous
 - assume that input X is synchronous



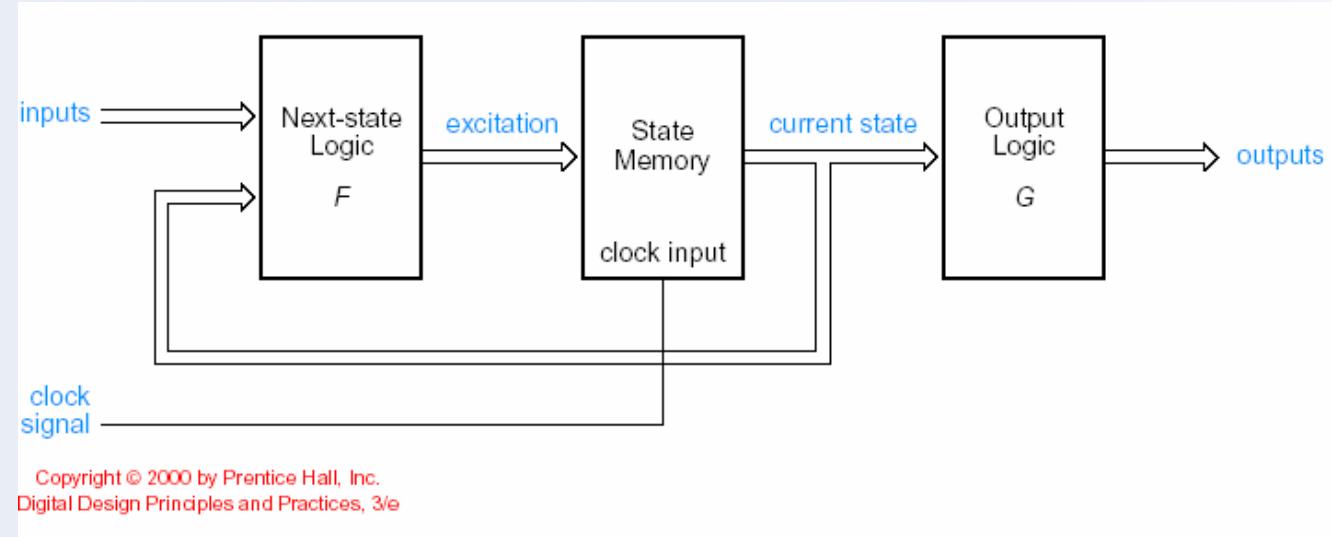
Mealy Machine



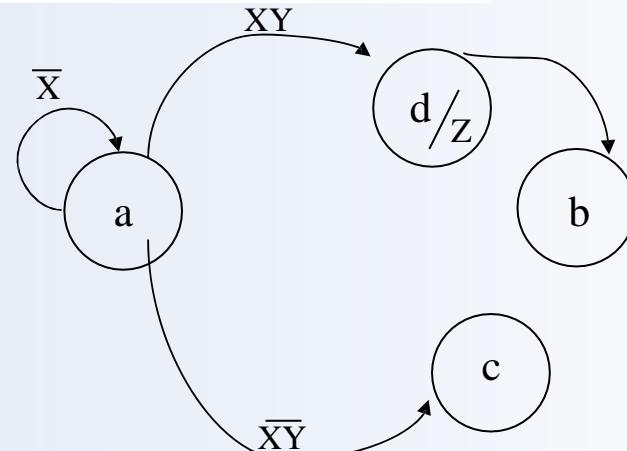
- Output are a function of P.S. and inputs
- Output associated with transition
- Tend to give the fewest states necessary
- Can cause problems when the inputs are asynchronous



Moore Machine



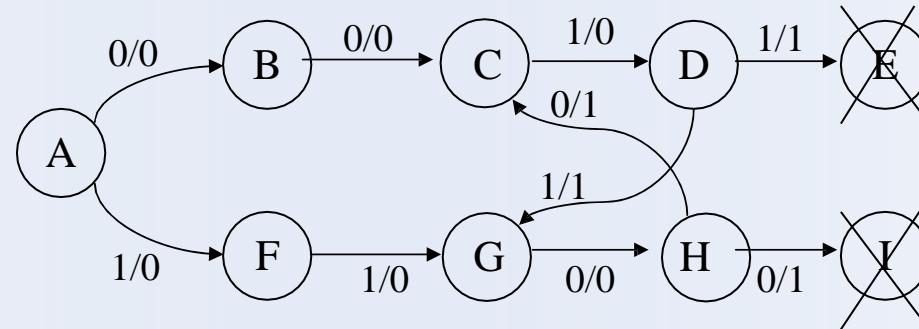
- Output are a function of P.S.
- Output associated with state
- Tend to have more states
- Output change only with a state change
 - do not rely on asynchronous inputs



Step 1: State/Output Diagram



- Draw bubbles for all correct sequences

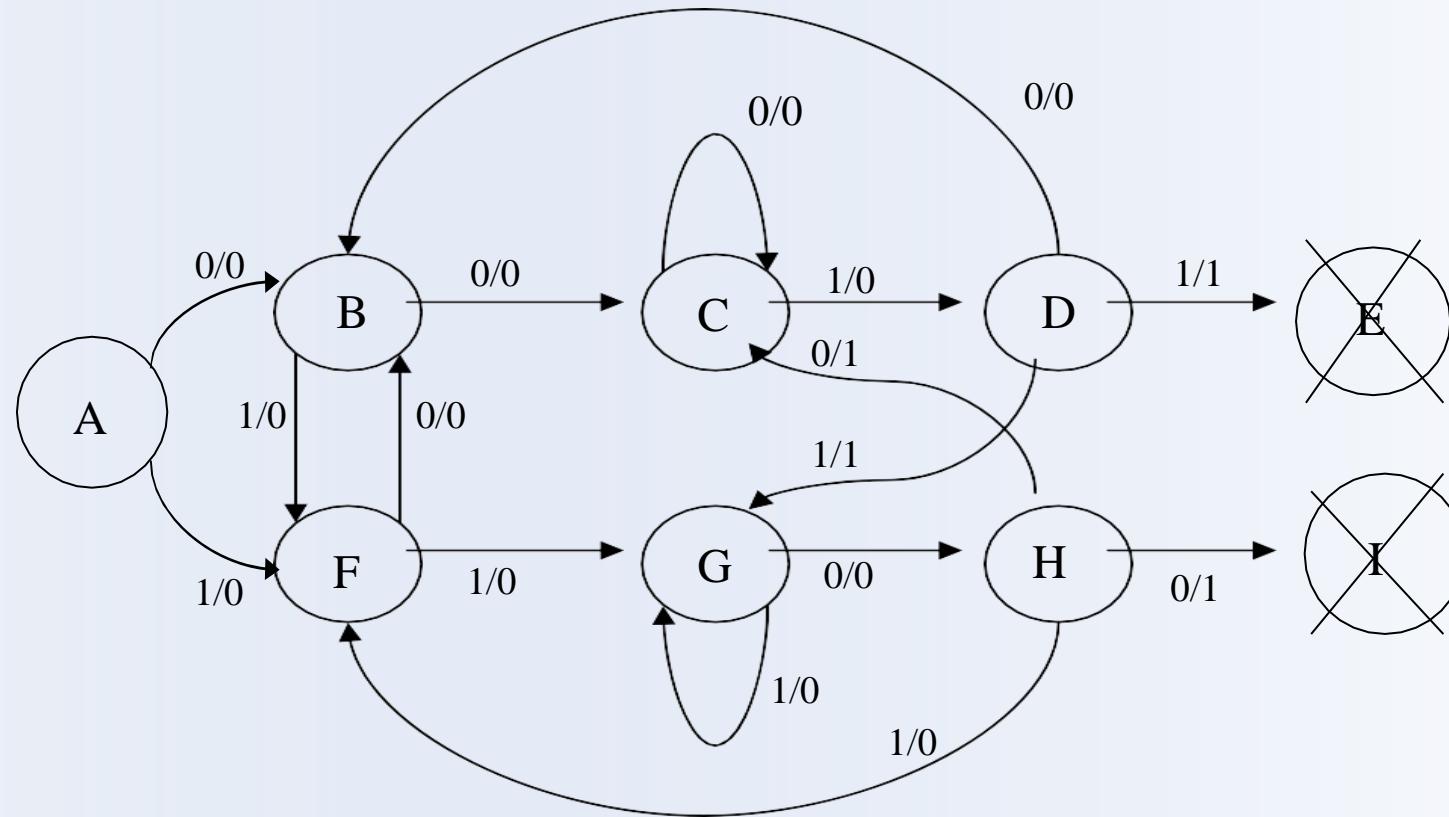


- **Meanings:** A state is a meaningful abstraction of previous history
(How many differentiated states? Infinite? What you need to remember?
What you can forget?)
 - (A – got nothing),
 - (B – got 0), (C – got 00), (D – got 001), (E – got 0011)
 - (F – got 1), (G – got 11), (H – got 110), (I – got 1100)
- In (E-got 0011), the future will not depend on 00 before 11 E=G
- Similarly, I=C

Step 1: State/Output Diagram



- Adding other input sequences

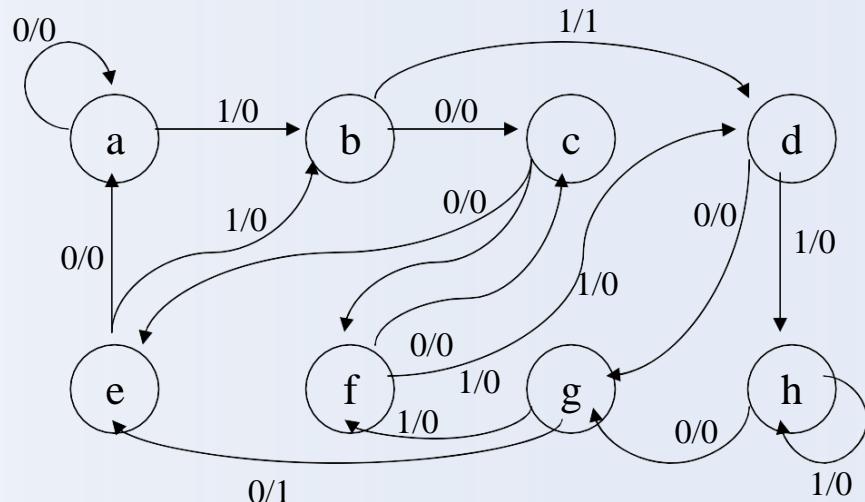


Step 1: State/Output Diagram



■ Another approach

- A state is understood as remembering something (previous history)
- All we have to remember is the last three inputs
- Eight states will be needed
 - (a=000, b=001, c=010, d=011, e=100, f=101, g=110, h=111)



P.S. X	N.S. Z
a	0
a	1
b	0
b	1
c	0
c	1
d	0
d	1
e	0
e	1
f	0
f	1
g	0
g	1
h	0
h	1

Step 2: State Minimization



- Identify equivalent states
 - Same output and next state (sometimes – use circular reasoning)
 - Formal minimization is beyond of scope

P.S. X	N.S. Z
a 0	a. 0
a 1	b. 0
b 0	c. 0
b. 1	d. 1
c. 0	e. 0
c 1	f. 0
d 0	g. 0
	h. 0
e 0	a. 0
	b. 0
f 0	c. 0
f. 1	d. 0
g. 0	e. 1
g 1	f. 0
h 0	g. 0
	h. 0

equivalent

equivalent

$a = e \text{ (got } x00) \leftrightarrow C \text{ (got } 00)$ $b \text{ (got } 001) \leftrightarrow D \text{ (got } 001)$
 $c \text{ (got } 010) \leftrightarrow B \text{ (got } 0)$
 $d = h \text{ (got } x11) \leftrightarrow G \text{ (got } 11)$ $f \text{ (got } 101) \leftrightarrow F \text{ (got } 1)$
 $g \text{ (got } 110) \leftrightarrow H \text{ (got } 110)$

Step 3: State Assignment



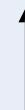
- Major effect on circuit cost
- Practical guidelines
 - 00...00 for initial state
 - Minimize # of state variables that change on transition
 - Maximize # of state variables that do not change in group of related states
 - Exploit symmetries – related states or group one bit difference
 - Use unused states well: Minimal risk or Minimal cost
 - Decompose – into individual bits or fields such that each bit has well defined meaning w.r.t inputs and outputs
 - Consider using more than the minimum # of state variables
 - One-hot assignment □ small excitation equations, good for 1-out-of-s coded output

State Assignment Examples



Assignment

State Name	Simplest $Q_C Q_B Q_A$	Decomposed $Q_C Q_B Q_A$	Arbitrary $Q_C Q_B Q_A$	One-hot Q_6-Q_1
a (got 000)	000	000	000	000001
b (got 001)	001	001	001	000010
c (got 010)	010	010	011	000100
d (got 011)	011	011	010	001000
f (got 101)	100	101	110	010000
g (got 110)	101	110	111	100000



We will use this assignment although it is not particularly good

Step 4-6: Transition/Excitation/Output Table



P.S.	$Q_C Q_B Q_A$	X	N.S.	$Q_C Q_B Q_A$	Z	$D_C D_B$	D_A
a	0 0 0	0	a	0 0 0	0		
a	0 0 0	1	b	0 0 1	0		
b	0 0 1	0	c	0 1 1	0		
b	0 0 1	1	d	0 1 0	1		
c	0 1 1	0	e=a	0 0 0	0		
c	0 1 1	1	f	1 1 0	0		
d	0 1 0	0	g	1 1 1	0		
d	0 1 0	1	h=d	0 1 0	0		
e	0		a		0		
e	1		b		0		
f	1 1 0	0	c	0 1 1	0		
f	1 1 0	1	d	0 1 0	0		
g	1 1 1	0	e=a	0 0 0	1		
g	1 1 1	1	f	1 1 0	0		
h	0		g		0		
h	1		h		0		

Step 7-8: Excitation/Output Eqs.



		Q _A X				
		00	01	11	10	
Q _C Q _B		00	0	0	0	0
01	1	0	1	0	0	
11	0	0	1	0	0	
10	-	-	-	-	-	

Minimum Cost

		Q _A X				
		00	01	11	10	
Q _C Q _B		00	0	0	1	1
01	1	1	1	1	0	
11	1	1	1	1	0	
10	-	-	-	-	-	

		Q _A X				
		00	01	11	10	
Q _C Q _B		00	0	1	0	1
01	1	0	0	0	0	
11	1	0	0	0	0	
10	-	-	-	-	-	

$$D_C = \overline{Q_C} Q_B \overline{Q_A} \overline{X} + Q_B Q_A X$$

$$D_B = Q_B \overline{Q_A} + Q_B X + \overline{Q_B} Q_A$$

$$D_A = Q_B \overline{Q_A} \overline{X} + \overline{Q_B} \overline{Q_A} X + \overline{Q_B} Q_A \overline{X}$$

		Q _A X				
		00	01	11	10	
Q _C Q _B		00	0	0	1	0
01	0	0	0	0	0	
11	0	0	0	0	1	
10	-	-	-	-	-	

$$Z = \overline{Q_B} Q_A X + Q_C Q_A \overline{X}$$

Step 9: Logic Diagram



- Will skip
- How much logic?
 - 14 NANDs
 - 3 F/Fs
 - 1 Invertor
- Minimum Cost Design: Unused states assigned for “minimum cost”
 - Risk?

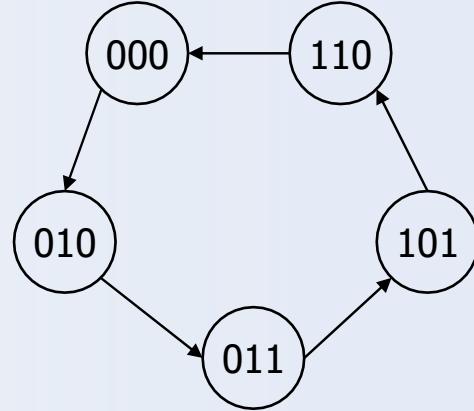
P.S.	X	N.S.
100	0	000
100	1	001
101	0	011
101	1	010

- Little risk – go into used states
- If it is not acceptable, may have to change don’t care to specific states

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



Note: The don't care conditions that arise from the unused state codes

In this case, only five states are used out of 8 possible binary values; so three don't care cases appear. Note that the next state literals are denoted with + symbol.

Present State			Next State		
Qc	Qb	Qa	Qc	Qb	Qa
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	-	-	-

More complex counter example (cont'd)



- Step 3: K-maps for next state functions

Dc	Qa	Qb	Qc
Qa	X	1	X
Qb			

Db	Qa	Qb	Qc
Qa	X	0	X
Qb			

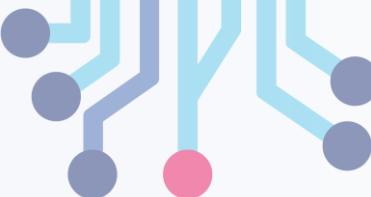
Da	Qa	Qb	Qc
Qa	X	1	X
Qb			

$$Dc = Qa$$

$$Db = Qb' + Qa'Qc' \quad Da = QbQc'$$

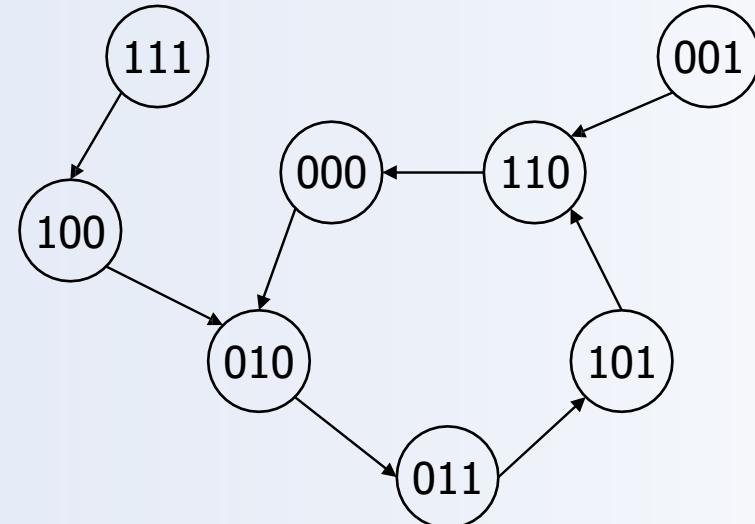
We see K-maps for three counter variables here.

Self-starting counters (cont'd)



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

Present State			Next State		
Qc	Qb	Qa	Qc	Qb	Qa
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



Dc	Qc		
	0	0	0
Qa	0	1	1
Qb	1	1	1

Db	Qc		
	1	1	0
Qa	1	0	1
Qb	0	0	1

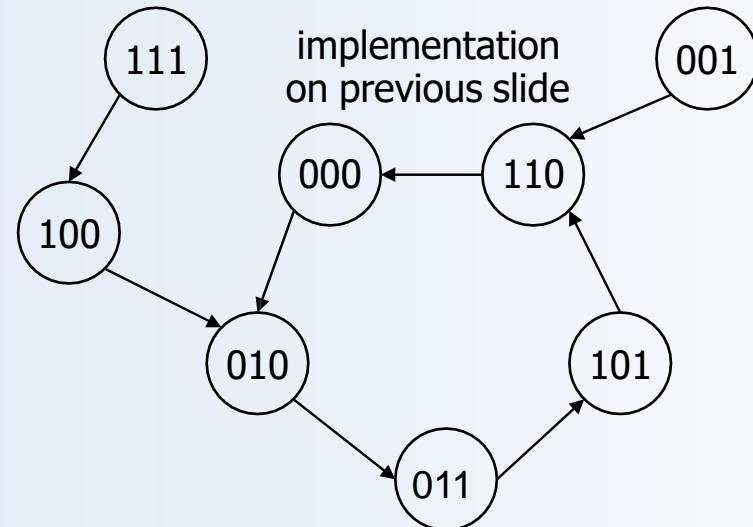
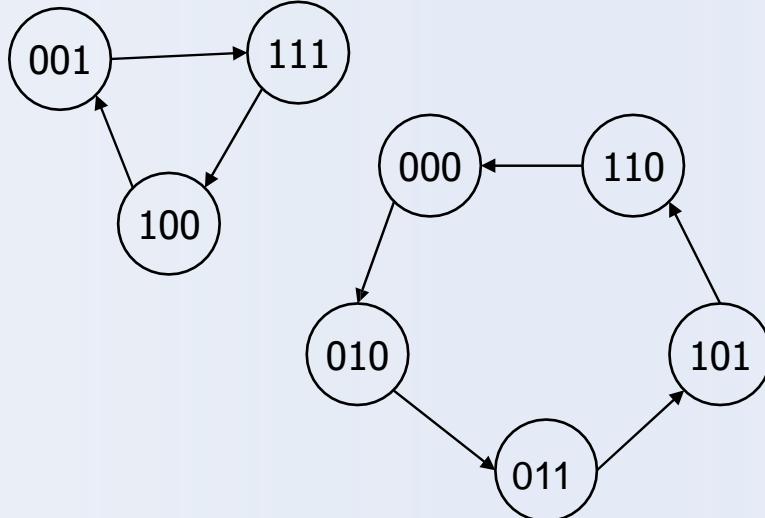
Da	Qc		
	0	1	0
Qa	0	1	0
Qb	0	1	0

Counters have two categories: self-starting and non self-starting. In self-starting, even though the system starts in one of all possible states, which may not be legal, the system will eventually go to one of valid states. And then, the system will remain in the set of legitimate states. Note that there are no don't care terms.

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

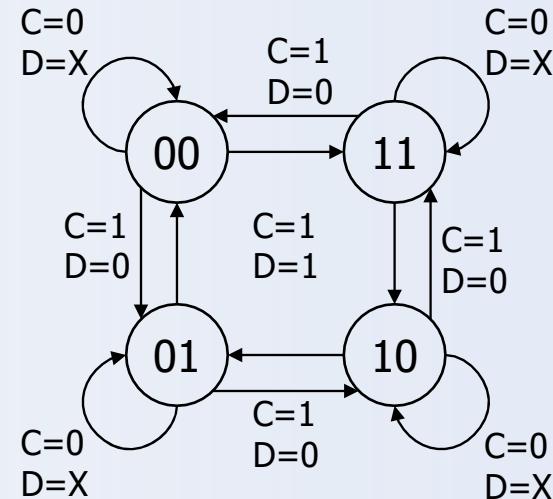


The left two counters are non-self-starting since if the system is in the state not shown in the state diagram, it will not work. Meanwhile the right one is self-starting.

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



Q1	Q0	C	D	NQ1	NQ0
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	1	1
0	1	0	0	0	1
0	1	0	1	0	1
0	1	1	0	1	0
0	1	1	1	0	0
1	0	0	0	1	0
1	0	0	1	1	0
1	0	1	0	1	1
1	0	1	1	0	1
1	1	0	0	1	1
1	1	0	1	1	1
1	1	1	0	0	0
1	1	1	1	1	0

Activity (cont'd)

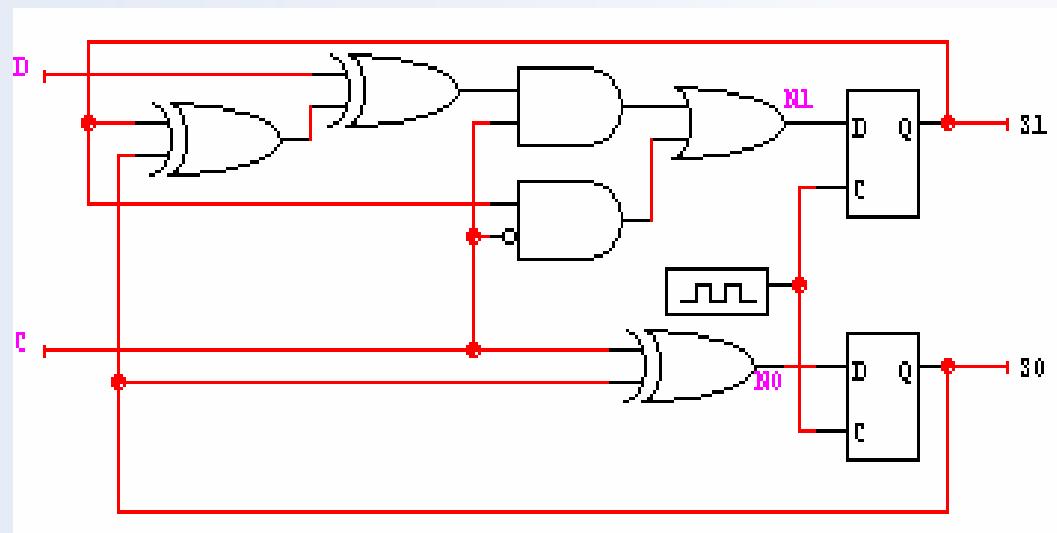


Q1	Q0	C	D	D1	D0	
0	0	0	0	0	0	
0	0	0	1	0	0	
0	0	1	0	0	1	
0	0	1	1	1	1	
0	1	0	0	0	1	
0	1	0	1	0	1	
0	1	1	0	1	0	
0	1	1	1	0	0	
1	0	0	0	1	0	
1	0	0	1	1	0	
1	0	1	0	1	1	
1	0	1	1	0	1	
1	1	0	0	1	1	
1	1	0	1	1	1	
1	1	1	0	0	0	
1	1	1	1	1	0	

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

$$\begin{aligned}
 D1 &= C'S1 \\
 &\quad + CDS0'S1' + CDS0S1 \\
 &\quad + CD'S0S1' + CD'S0'S1 \\
 &= C'S1 \\
 &\quad + C(D'(S1 \oplus S0) + D(S1 = S0))
 \end{aligned}$$

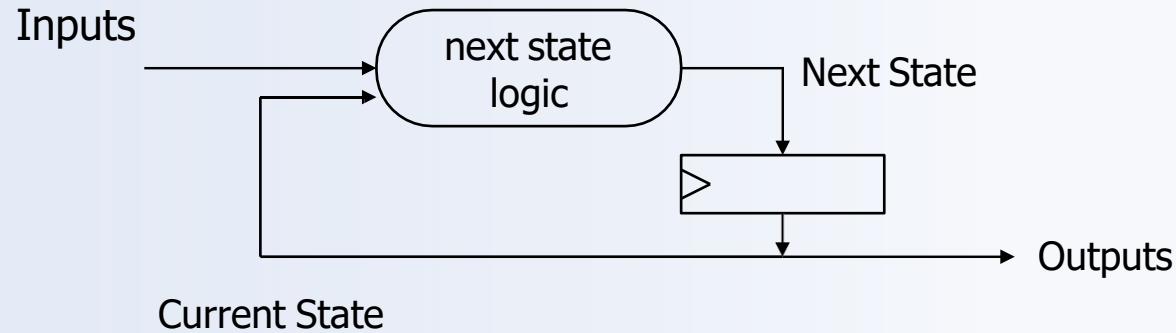
$$D0 = CS0' + C'S0$$



Counter/shift-register model



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

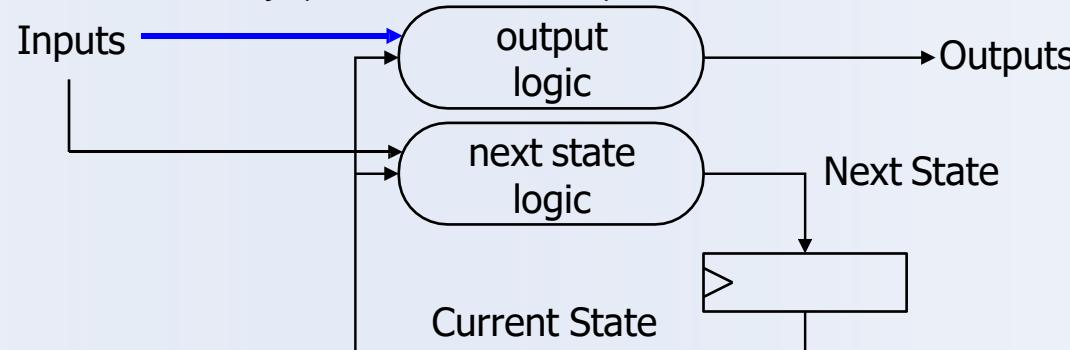


Here is the big picture of counter or shift register-based sequential logic systems. The current state and the input will decide the next state by forming a combinational logic in the oval. In the case of counters or registers, the values in the storage elements form the output. What if the state is not exactly the output?

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)

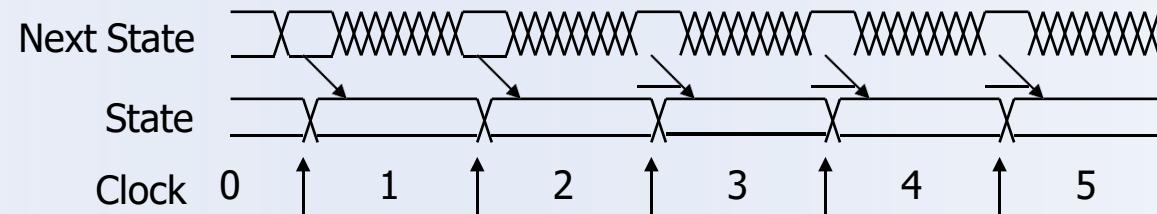
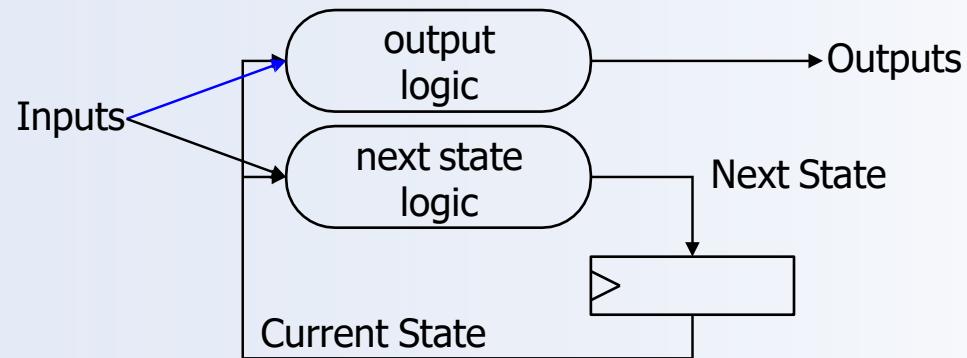


If output is different from the state, there should be one more combinational logic, the upper oval. There is another important classification: depending on the combinational logic for outputs. If outputs are functions of only current state, that model is called a Moore machine. On the other hand, if outputs are also dependent on external inputs, this is called a Mealy machine (drawn by a blue arrow).

State machine model (cont'd)



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



Again, the state transition time is the reference time, which is typically positive- (or negative-) edge of the clock signal depending on FF types. The clock period should be long enough to allow full propagation of input and the current state signals through combinational logic parts.

Comparison of Mealy and Moore machines



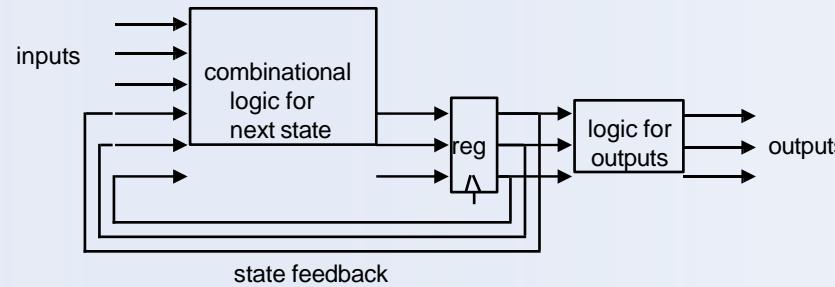
- Mealy machines tend to have less states
 - different outputs on arcs (n^2) rather than states (n)
- Moore machines are safer to use
 - outputs change at clock edge (always one cycle later)
 - in Mealy machines, input change can cause output change as soon as logic is done
 - a big problem when two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

As outputs of a Mealy machine are functions of the external inputs and the present state, the number of states may be less. Information for the next state transition is split between inputs from outside and the state. In Moore machines, outputs are dependent only on the present state, the output will change synchronously if combinational logic has no problem. In Mealy machines, external inputs can change the output anytime with combinational logic delay somewhat independently of the clock. If two machines perform the same function, Mealy machines react faster since inputs are already changing the combinational logic for output.

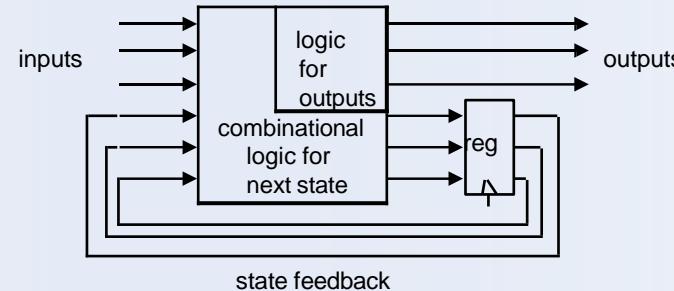
Comparison of Mealy and Moore machines (cont'd)



- Moore



- Mealy

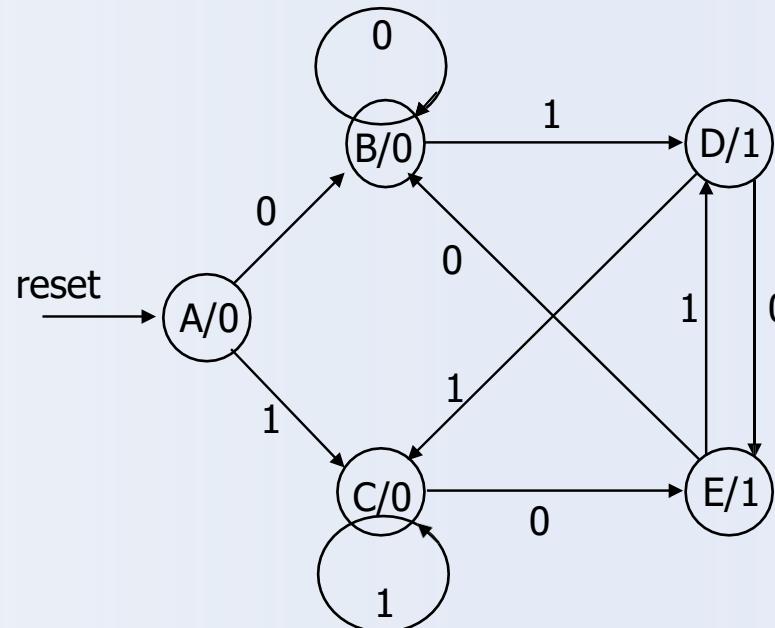


This slide illustrates the three types of sequential systems. Synchronous Mealy machines solve the potential glitches and asynchronous change of outputs of Mealy machines by inserting clock-triggered memory elements.

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



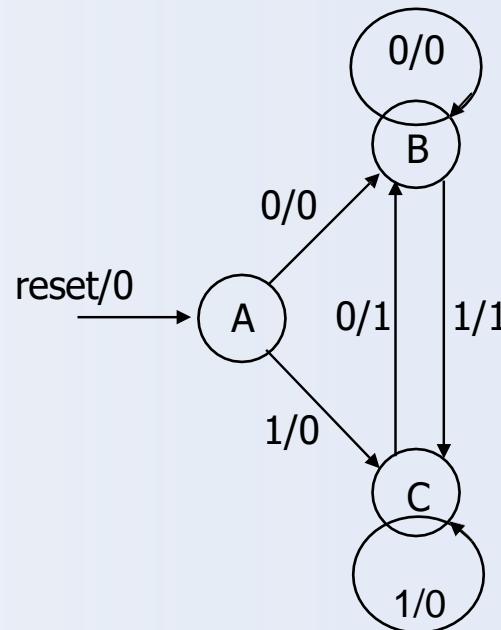
reset	input	current state	next state	output
1	-	-	A	
0	0	A	B	0
0	1	A	C	0
0	0	B	B	0
0	1	B	D	0
0	0	C	E	0
0	1	C	C	0
0	0	D	E	1
0	1	D	C	1
0	0	E	B	1
0	1	E	D	1

Let's see how a Moore machine can be described. Here, X/Y is the tuple of state X and the output Y. The label in each incoming arc is the input. The output is associated with the current state. Actually, the output signal will be asserted until the system goes to the next state. This Moore machine detects whether the recent input string is 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



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

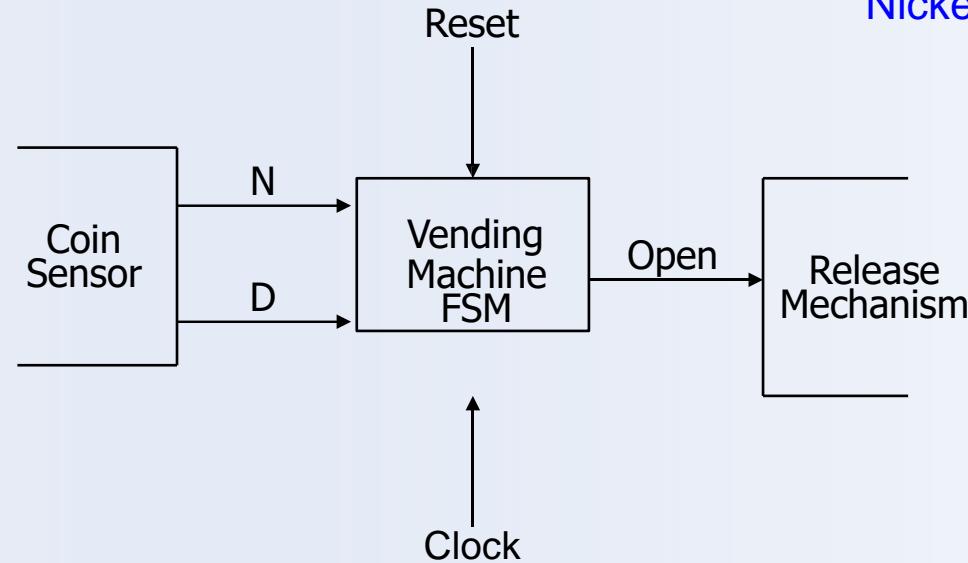
In a Mealy machine, both the input and present state determine the next state. X/Y notation in each arrow means input X will generate output Y. Compare the number of states; the Mealy model has only 3 states. The problem of the Mealy machine is that we cannot be sure of the exact timing of output change, not to mention glitch.

Example: vending machine



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

Dime: 10 cent coin
Nickel: 5 cent coin

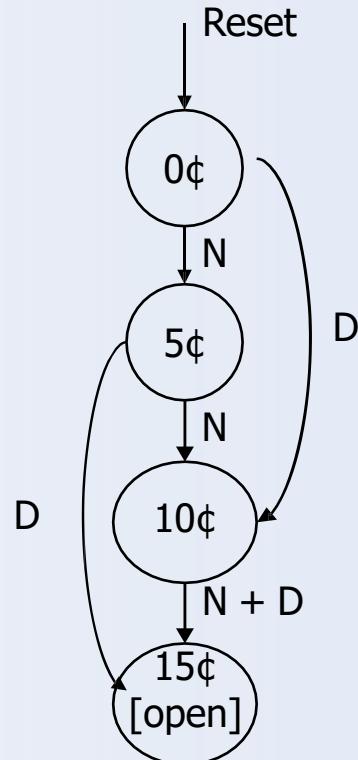


Now we will see three or four implementations of the same vending machine that sells an item which costs 15 cents. We don't need to figure out the exact mechanism of identifying dimes and nickels. Just assume that the corresponding wire will be asserted: N for nickel and D for dime. Also, for simplicity, we do not care about change.

Example: vending machine (cont'd)



State diagram



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

Here is the simple state transition table for the vending machine. This is kind of a Moore machine since the output becomes 1 after the system moves to state 15¢. First of all, a dime and a nickel cannot be inserted at the same time, which implies don't care terms.

Example: vending machine (cont'd)



- Uniquely encode states

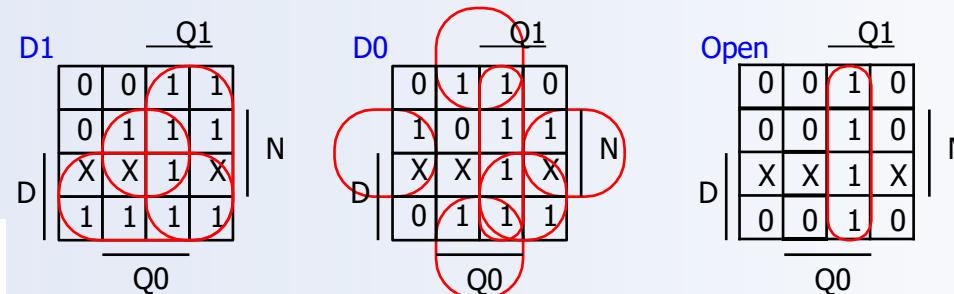
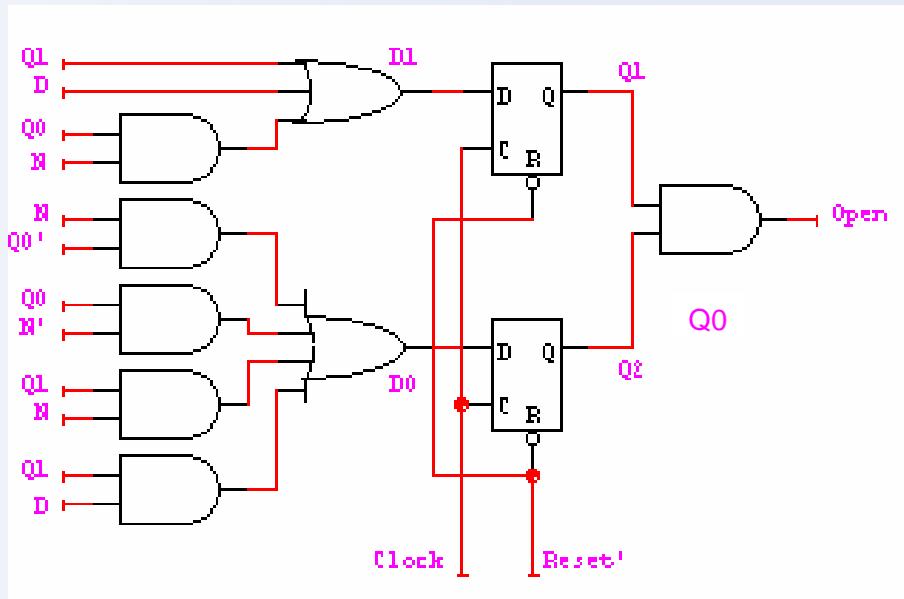
present state		inputs		next state		output
Q1	Q0	D	N	NQ1	NQ0	open
0	0	0	0	0	0	0
		0	1	0	1	0
		1	0	1	0	0
		1	1	—	—	—
0	1	0	0	0	1	0
		0	1	1	0	0
		1	0	1	1	0
		1	1	—	—	—
1	0	0	0	1	0	0
		0	1	1	1	0
		1	0	1	1	0
		1	1	—	—	—
1	1	—	—	1	1	1

So there are 4 states of the system (0,5,10,15¢), which requires minimum two FFs. The number of bits to represent states can be determined in many ways; we will look at two cases here. Two external inputs and one external output are already explained. This is a simple Moore machine since the output is dependent only on state.

Example: Moore implementation



- Mapping to logic



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

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

$$OPEN = Q1 Q0$$

There are total 4 input variables for each output. OPEN seems to be the simplest logic. In this case, the external output is a function of only state variables. For simplicity, we skip the feedback parts of Q1 and Q0 wires.

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	0	1	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

$$D_0 = Q_0 D' N'$$

$$D_1 = Q_0 N + Q_1 D' N'$$

$$D_2 = Q_0 D + Q_1 N + Q_2 D' N'$$

$$D_3 = Q_1 D + Q_2 D + Q_2 N + Q_3$$

$$OPEN = Q_3$$

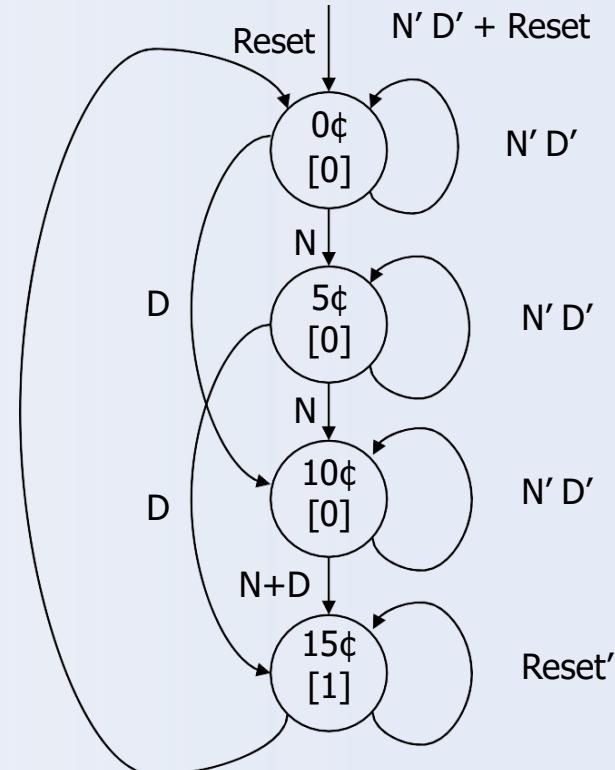
One-hot encoding means only one bit is ON for each state; that is, only one state variable is set, or "hot," for each state. So the number of bits to represent states is the same as the number of states. The benefit is that the next state generation function may be simple since the number of product terms for each output is typically small. In this case, we use 4 bits or FFs.

Equivalent Mealy and Moore state diagrams



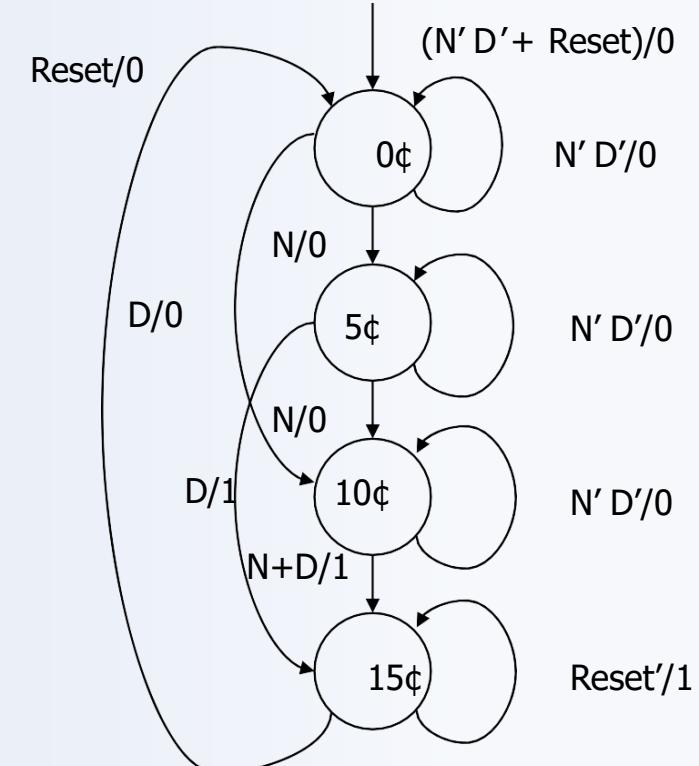
- Moore machine

- outputs associated with state



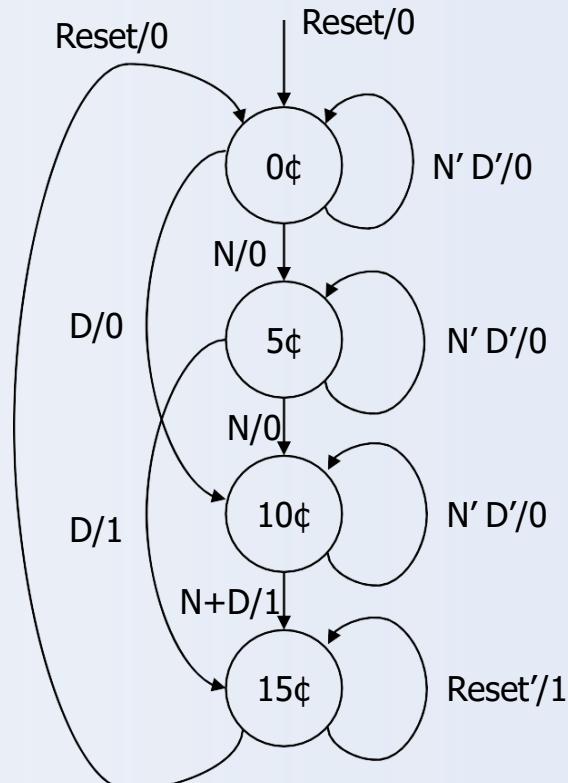
- Mealy machine

- outputs associated with transitions



This slide shows the complete state transition diagram of the vending machine in two versions. In the Moore model, the number in [] is the output. Whereas, in the Mealy model, the output is associated with each arc.

Example: Mealy implementation



present state Q1 Q0	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	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

Open	Q1		N
	Q0	Q1	
0	0	1 0	
0	0	1 1	
X	X	1 X	
0	1	1 1	

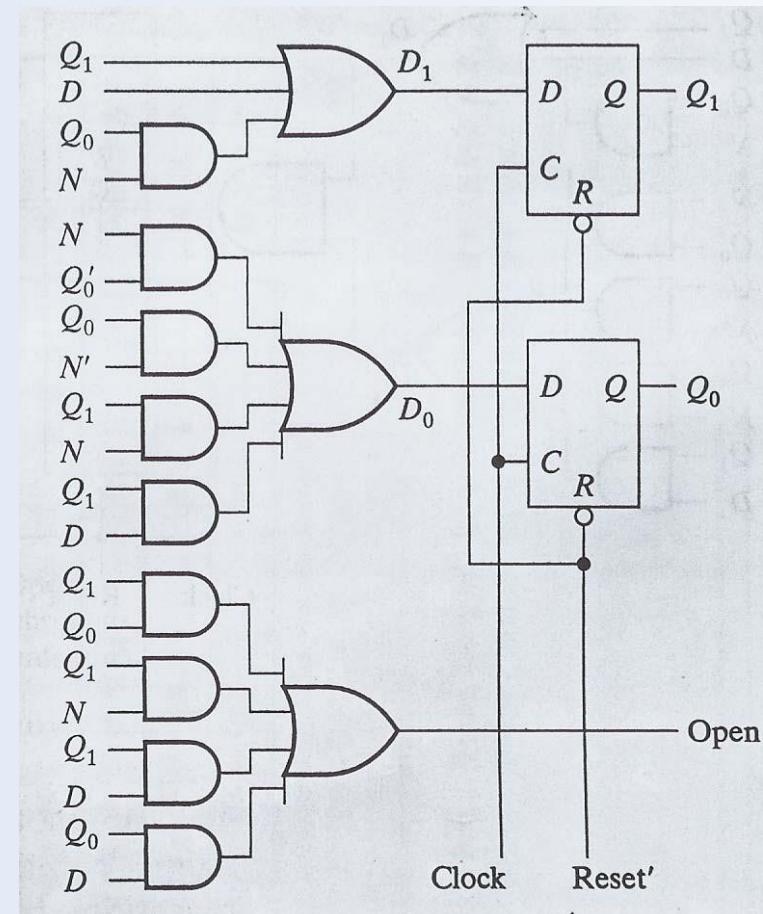
$$\begin{aligned}
 D_0 &= Q_0'N + Q_0N' + Q_1N + Q_1D \\
 D_1 &= Q_1 + D + Q_0N \\
 \text{OPEN} &= Q_1Q_0 + Q_1N + Q_1D + Q_0D
 \end{aligned}$$

In the case of a Mealy machine, the output, OPEN, is a function of state and the inputs.

Example: Mealy implementation



$$\begin{aligned}D_0 &= Q_0'N + Q_0N' + Q_1N + Q_1D \\D_1 &= Q_1 + D + Q_0N \\OPEN &= Q_1Q_0 + Q_1N + Q_1D + Q_0D\end{aligned}$$



Here is the overall implementation of the vending machine based on the Mealy model.

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:

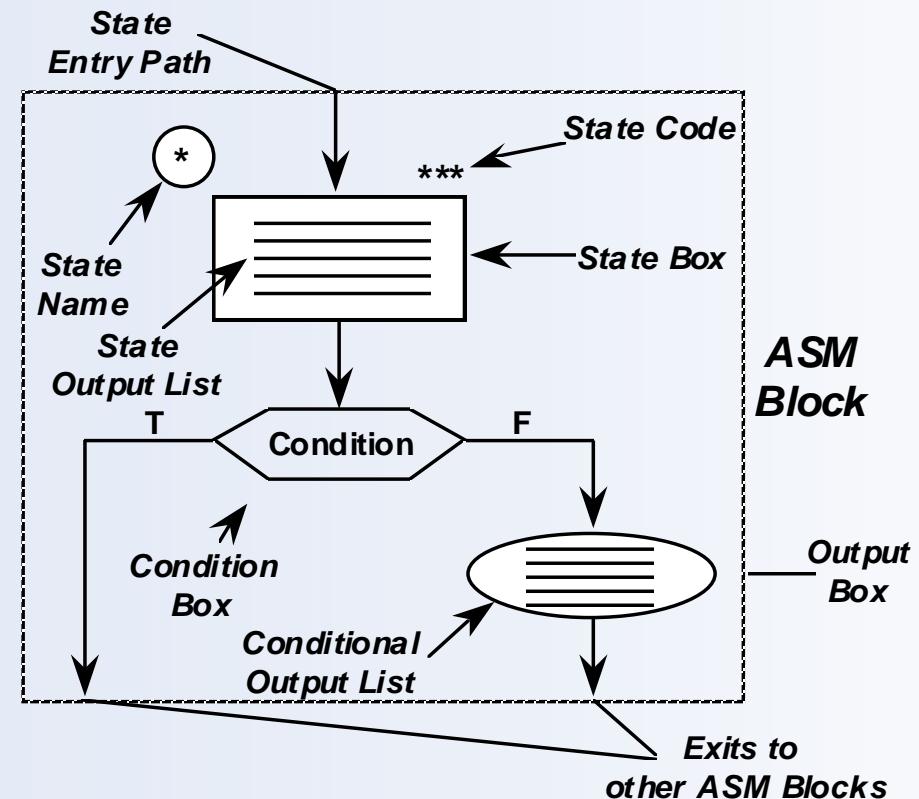
- State Box
- Decision Box
- Output Box

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



Alternative State Machine Representations



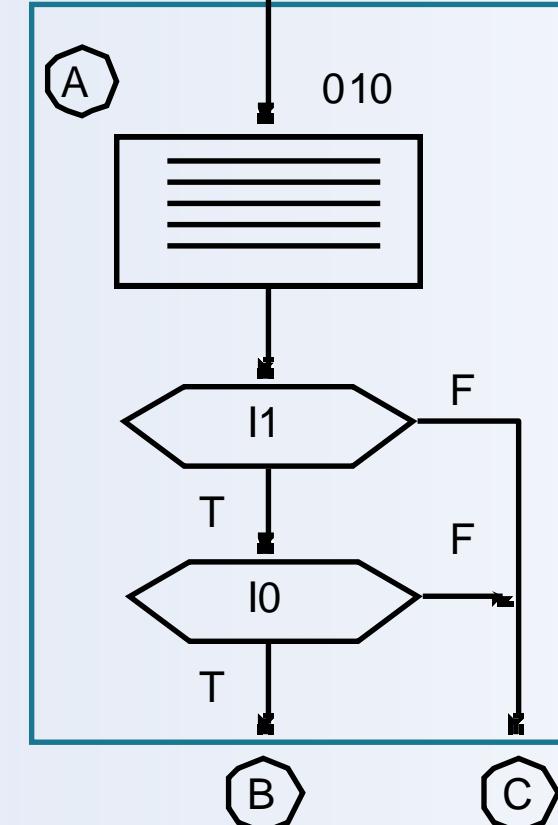
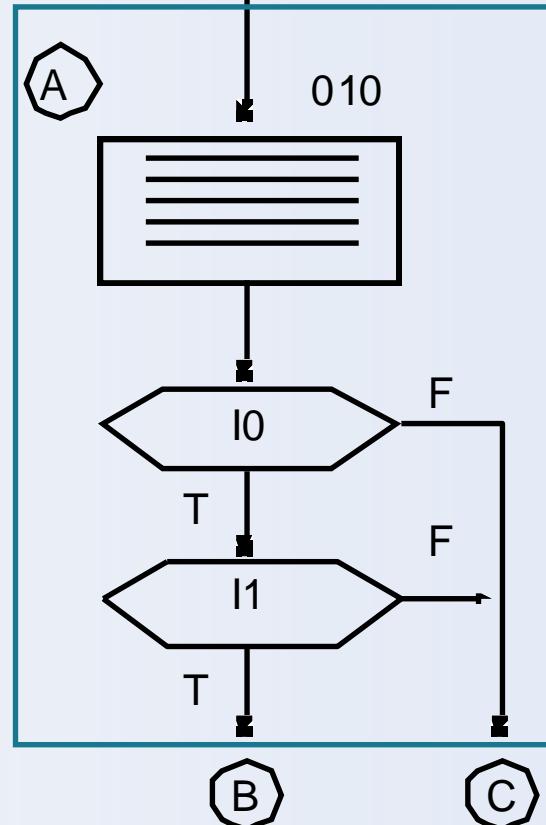
ASM Notation

Condition Boxes:

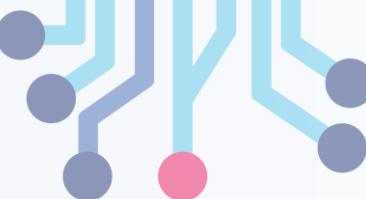
Ordering has no effect on final outcome

Equivalent ASM charts:

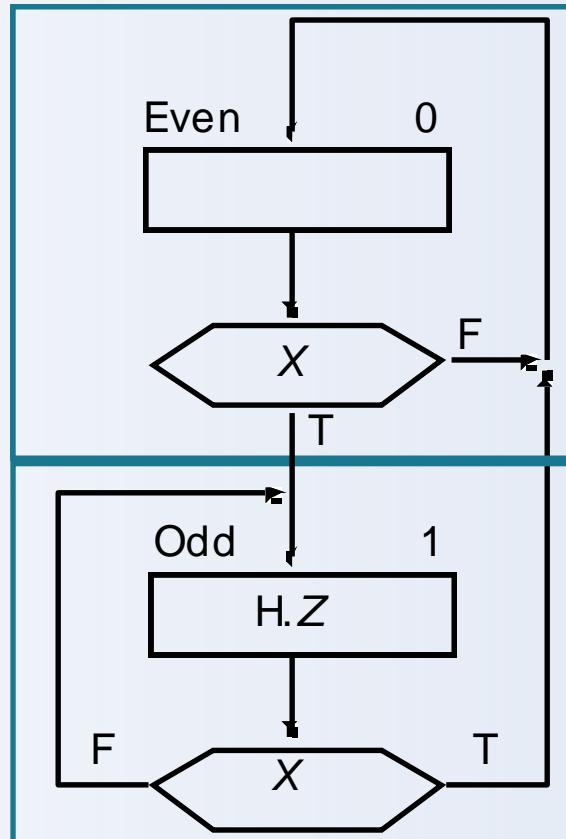
A exits to B on ($I0 \cdot I1$) else exit to C



Alternative State Machine Representations



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:

Input	Present State	Next State	Output
F	Even	Even	—
T	Even	Odd	—
F	Odd	Odd	A
T	Odd	Even	A

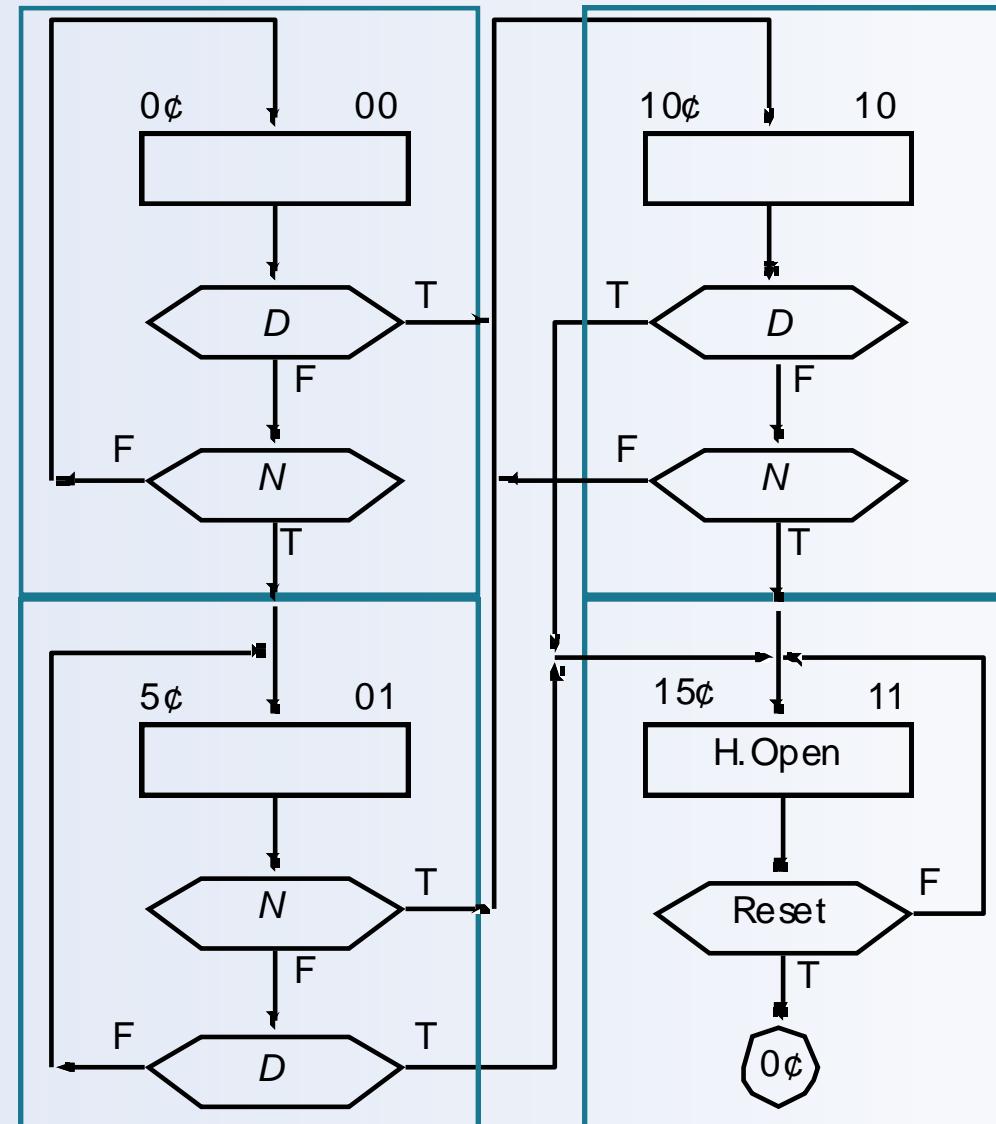
Encoded State Table:

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

Alternative State Machine Representations



ASM Chart for Vending Machine



Finite State Machine Word Problems



Mapping English Language Description to Formal Specifications

Four Case Studies:

- Finite String Pattern Recognizer
- Complex Counter with Decision Making
- Traffic Light Controller
- Digital Combination Lock

We will use state diagrams and ASM Charts

Finite State Machine Word Problems



Finite String Pattern Recognizer

A finite string recognizer has one input (X) and one output (Z).
The output is asserted whenever the input sequence ...010...
has been observed, as long as the sequence 100 has never been
seen.

Step 1. Understanding the problem statement

Sample input/output behavior:

X: 00101010010...

Z: 00010101000...

X: 11011010010...

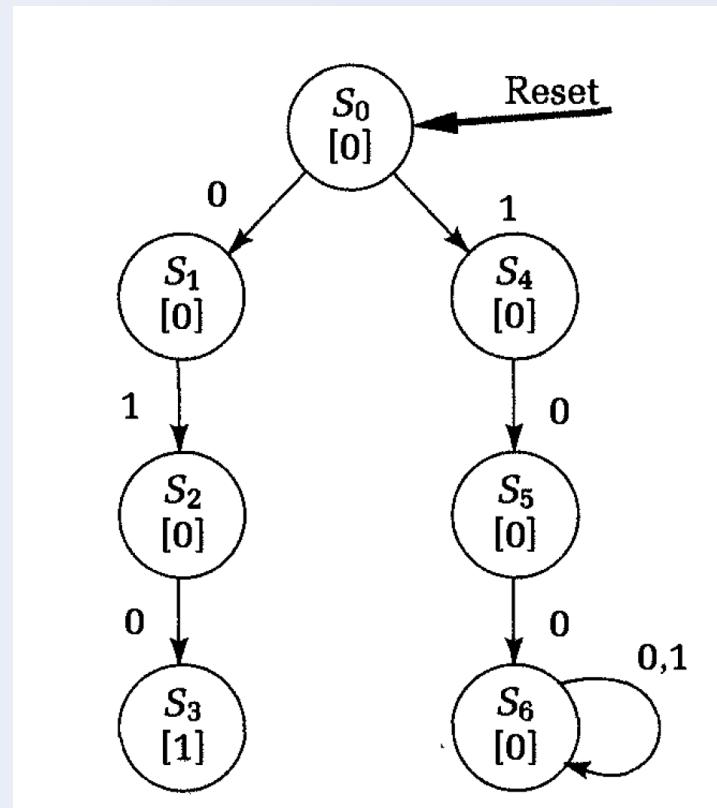
Z: 00000001000...

Finite State Machine Word Problems



Finite String Recognizer

Step 2. Draw State Diagrams/ASM Charts for the strings that must be recognized. I.e., 010 and 100.



Outputs 1

**Moore State Diagram
Reset signal places
FSM in S_0**

Loops in State



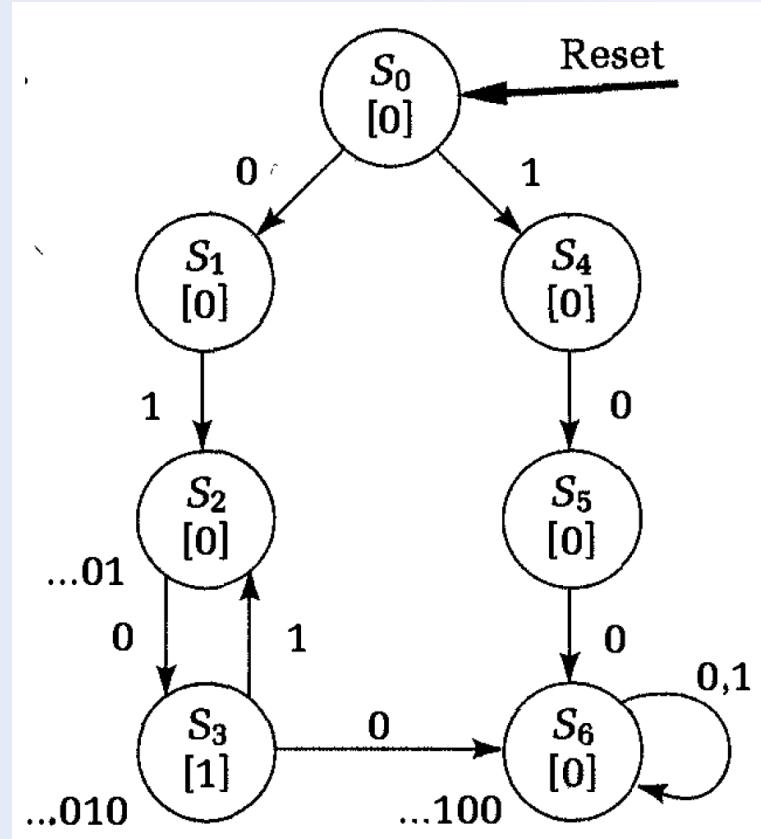
Finite State Machine Word Problems

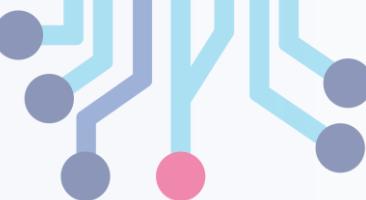
Finite String Recognizer

Exit conditions from state S3: have recognized ...010

if next input is 0 then have ...0100!

if next input is 1 then have ...0101 = ...01 (state S2)



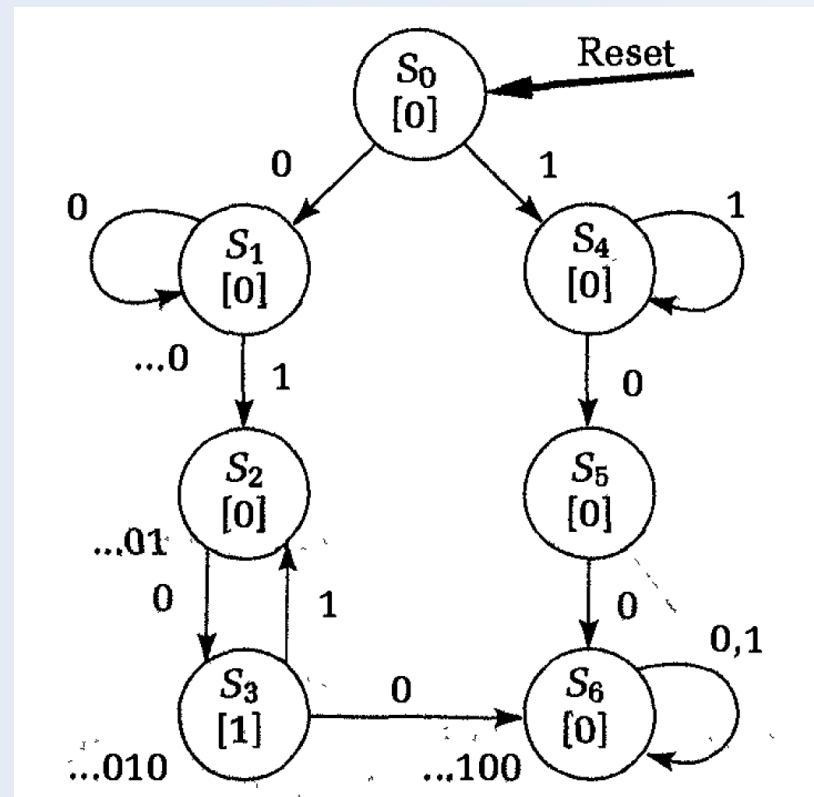


Finite State Machine Word Problems

Finite String Recognizer

Exit conditions from S1: recognizes strings of form ...0 (no 1 seen)
loop back to S1 if input is 0

Exit conditions from S4: recognizes strings of form ...1 (no 0 seen)
loop back to S4 if input is 1



Finite State Machine Word Problems

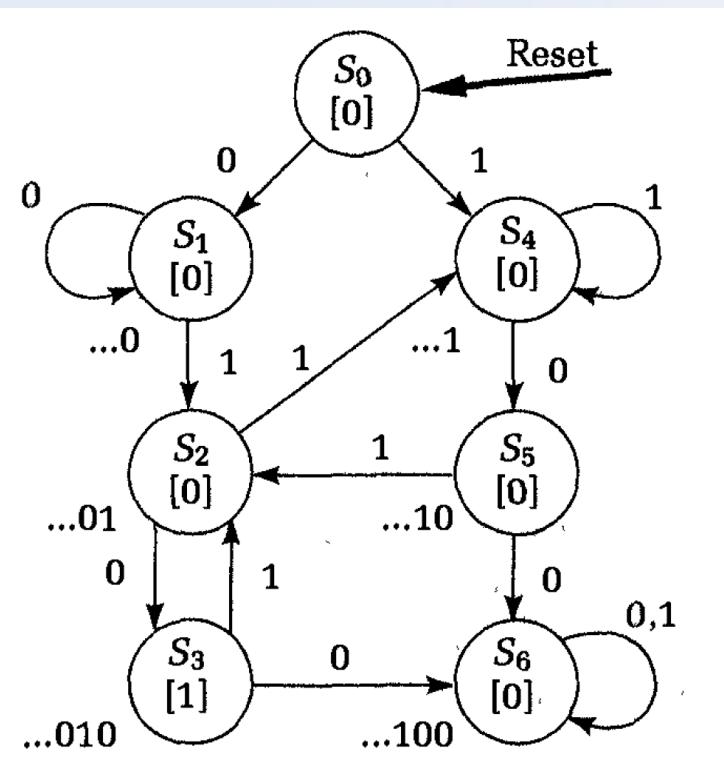


Finite String Recognizer

S2, S5 with incomplete transitions

S2 = ...01; If next input is 1, then string could be prefix of (01)1(00)
S4 handles just this case!

S5 = ...10; If next input is 1, then string could be prefix of (10)1(0)
S2 handles just this case!



Final State Diagram

Finite State Machine Word Problems

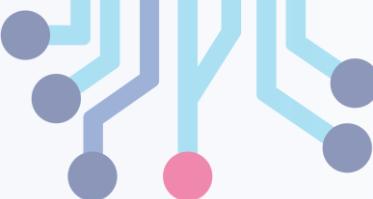


Finite String Recognizer

Review of Process:

- Write down sample inputs and outputs to understand specification
- Write down sequences of states and transitions for the sequences to be recognized
- Add missing transitions; reuse states as much as possible
- Verify I/O behavior of your state diagram to insure it functions like the specification

Activity-1



Complex Counter

A sync. 3 bit counter has a mode control M. When M = 0, the counter counts up in the binary sequence. When M = 1, the counter advances through the Gray code sequence.

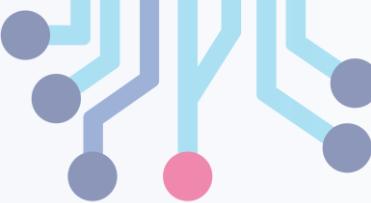
Binary: 000, 001, 010, 011, 100, 101, 110, 111

Gray: 000, 001, 011, 010, 110, 111, 101, 100

Valid I/O behavior:

Mode Input M	Current State	Next State (Z2 Z1 Z0)
0	000	001
0	001	010
1	010	110
1	110	111
1	111	101
0	101	110
0	110	111

Activity-2



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.

Activity-3



Digital Combination Lock "3 bit serial lock controls entry to locked room. Inputs are RESET, ENTER, 2 position switch for bit of key data. Locks generates an UNLOCK signal when key matches internal combination. ERROR light illuminated if key does not match combination. Sequence is: (1) Press RESET, (2) enter key bit, (3) Press ENTER, (4) repeat (2) & (3) two more times."

Problem specification is incomplete:

- **how do you set the internal combination?**
- **exactly when is the ERROR light asserted?**

Make reasonable assumptions:

- hardwired into next state logic vs. stored in internal register
- assert as soon as error is detected vs. wait until full combination has been entered

Our design: registered combination plus error after full combination

Thank You



NCDC NUST
CHIP
DESIGN
CENTRE