Chapter 7.
finite state machines (FSMs)

In chapter 6, we looked at counters, whose values are useful for representing states. Normally the number of states is finite. And a circuit or a system is modeled as a machine that makes transitions among states. The state is the main theme of this chapter. Some counters are moving among states without external inputs but FSMs usually have external inputs. So FSM is a kind of a superset.

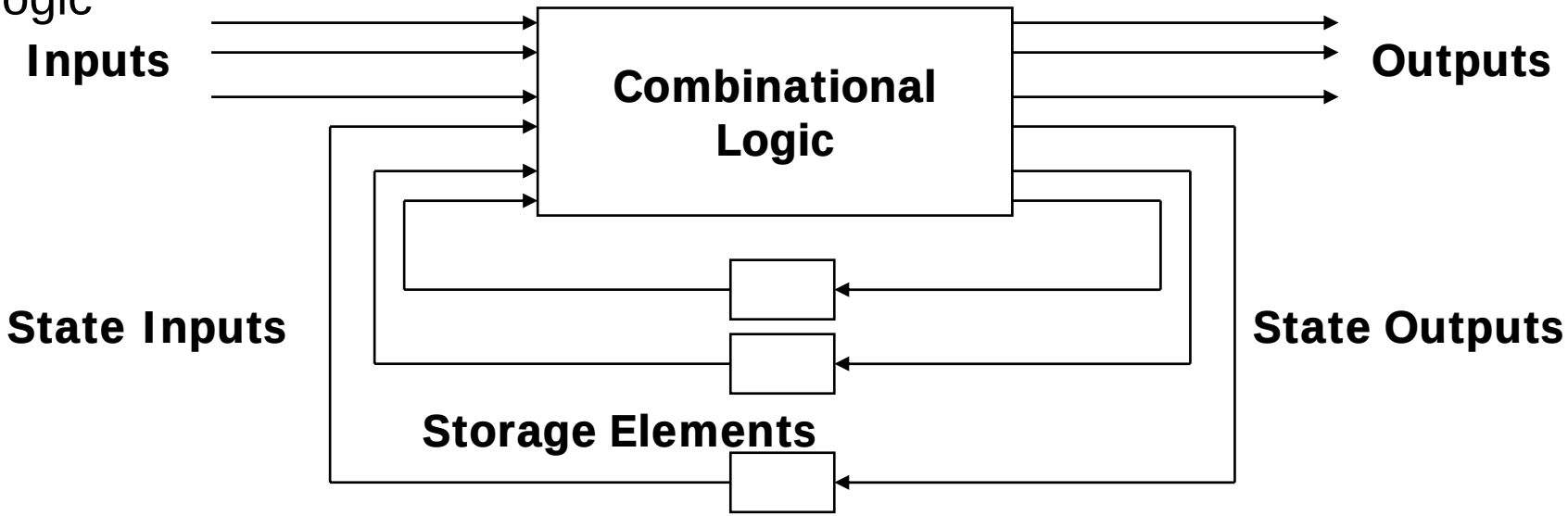
Finite State Machines

- Sequential circuits
  - primitive sequential elements
  - combinational logic
- Models for representing sequential circuits
  - finite-state machines
- Design procedure
  - State/output diagrams
  - State/output tables
  - next state/output equations
- Basic sequential circuits revisited
  - shift registers
  - counters
- Hardware description languages

These are the topics that will be discussed in this chapter. Of course the next state depends on the current state and the input values. The FSMs fall into two categories: Moore and Mealy machines. Also we will look at how inputs are handled in sequential systems. Still registers and counters are key parts of the sequential circuits.

Abstraction of state elements

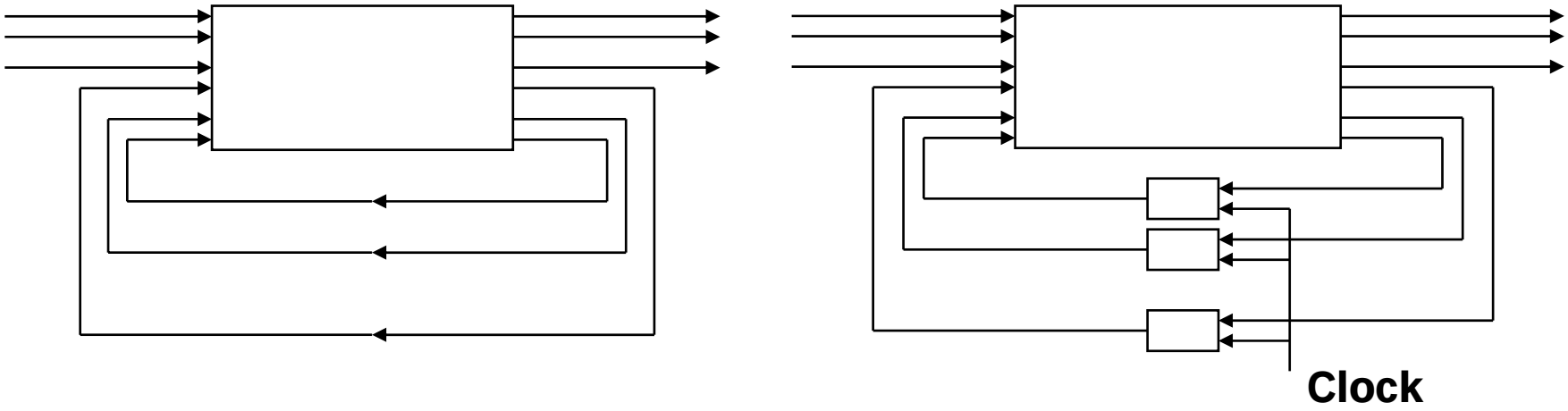
- Divide circuit into combinational logic and state
- Localize the feedback loops and make it easy to break cycles
- Implementation of storage elements leads to various forms of sequential logic



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)

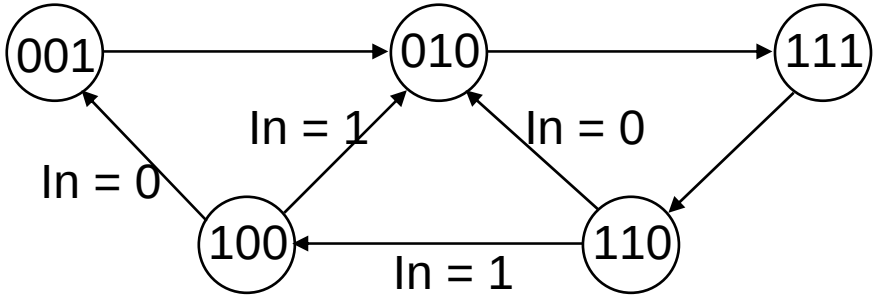


Asynchronous sequential logic circuits are operating without a clock, as shown on the left. The majority of sequential logic is synchronous logic circuits operating with clock signals, as illustrated on the right. With the clock signal, it is more convenient to control state transitions.

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

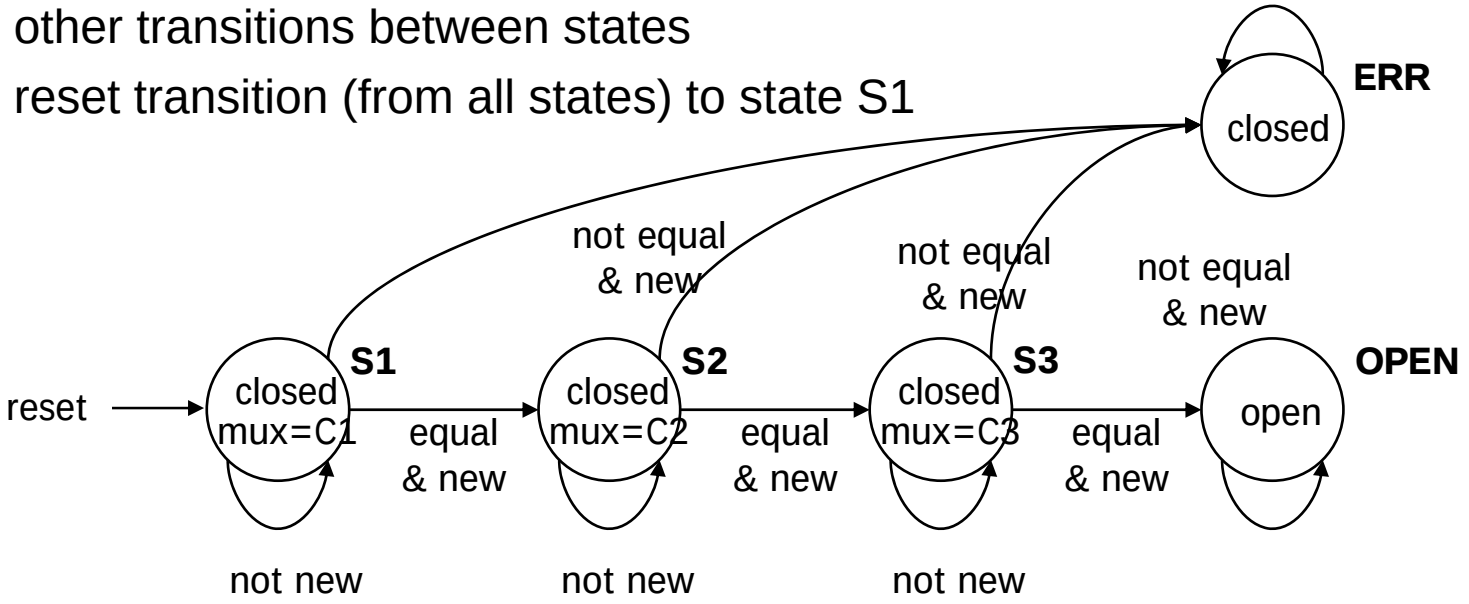
- Sequential logic
  - sequences through a series of states
  - based on sequence of values on input signals
  - clock period defines elements of sequence



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.

Example finite state machine diagram

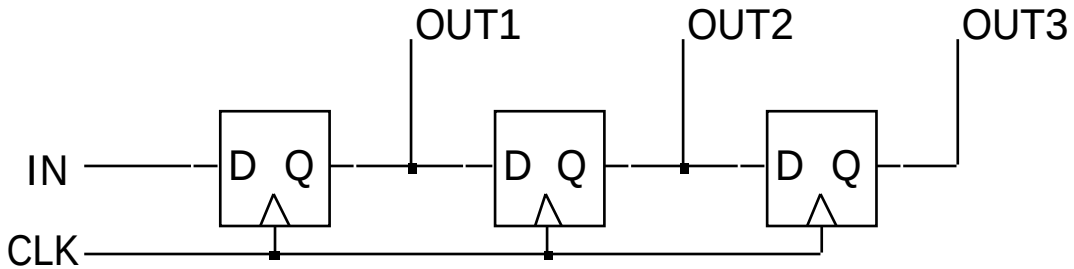
- Combination lock from introduction to course
  - 5 states
  - 5 self-transitions
  - 6 other transitions between states
  - 1 reset transition (from all states) to state S1

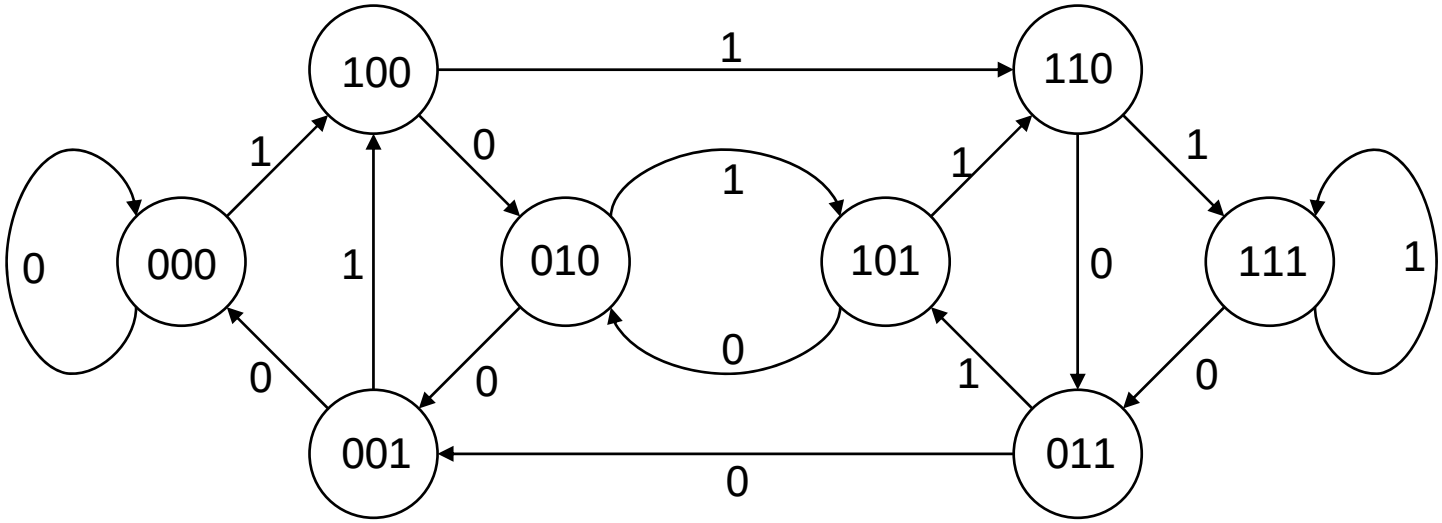


Let's revisit the door combination lock system briefly. In the example in chapter 1, there were 5 states. There are two kinds of transitions: self-transition, and ordinary transition.

Can any sequential system be represented with a state diagram?

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

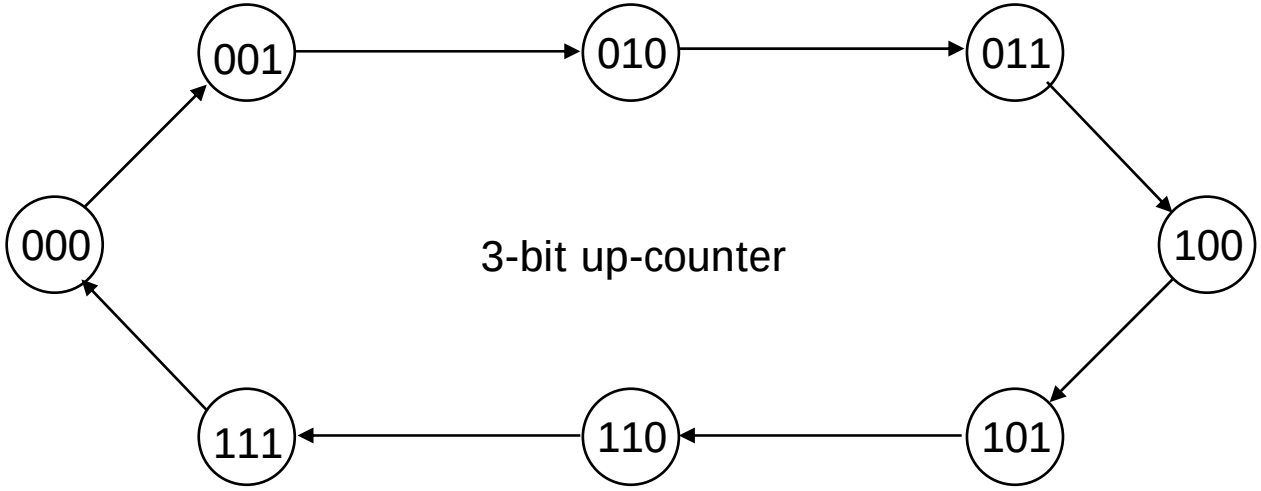




This state diagram shows all the possible states and transitions of a 3 bit shift register. In this case, the new state values are equal to output values. For example, when the system moves from 100 to 010, the output is 010. First of all, 3 bit will represent 8 states. And in each state, two kinds of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

Counters are simple finite state machines

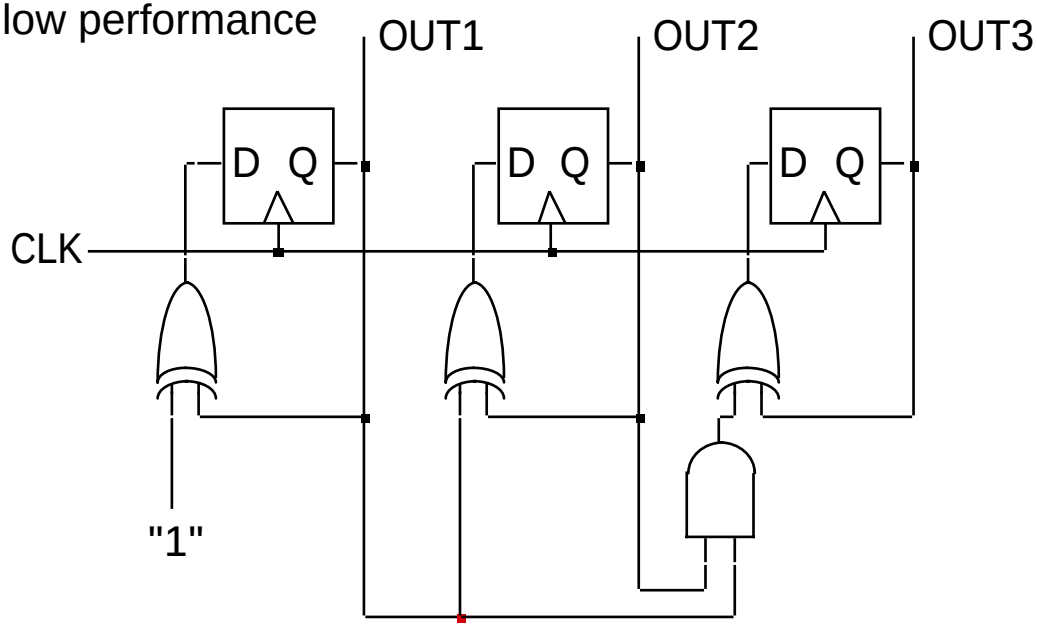
- Counters
  - proceed through well-defined sequence of states in response to enable
- Many types of counters: binary, BCD, Gray-code
  - 3-bit up-counter: 000, 001, 010, 011, 100, 101, 110, 111, 000, ...
  - 3-bit down-counter: 111, 110, 101, 100, 011, 010, 001, 000, 111, ...



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?

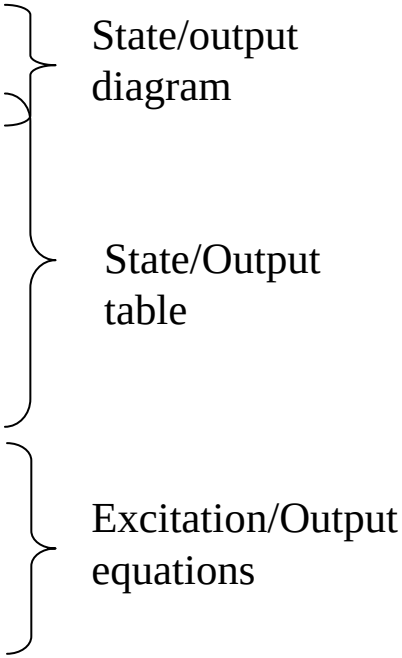
- Counter
  - 3 flip-flops to hold state
  - logic to compute next state
  - clock signal controls when flip-flop memory can change
    - wait long enough for combinational logic to compute new value
    - don't wait too long as that is low performance



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

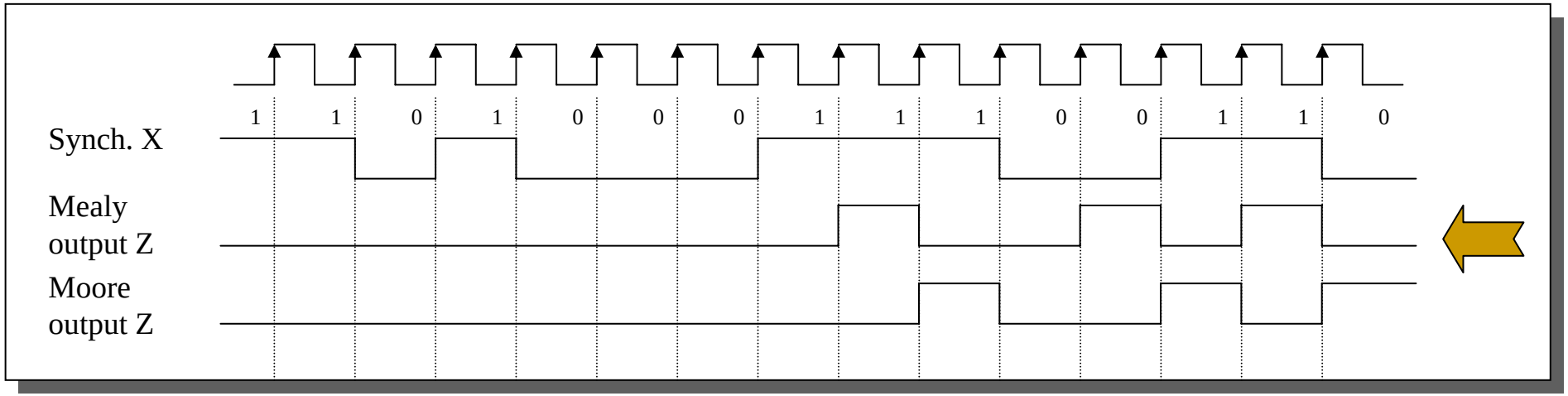


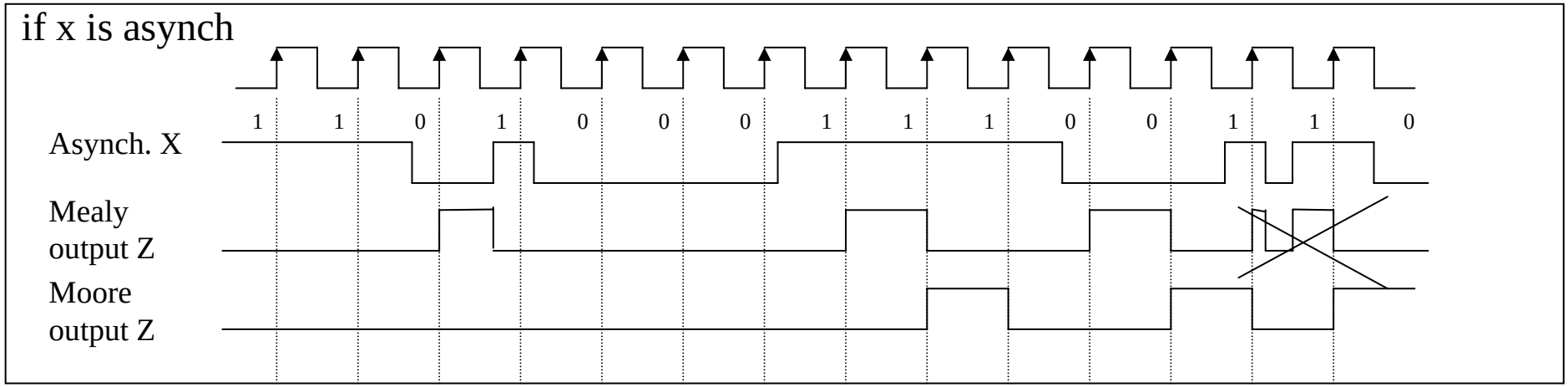
Problem Statement

- 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.

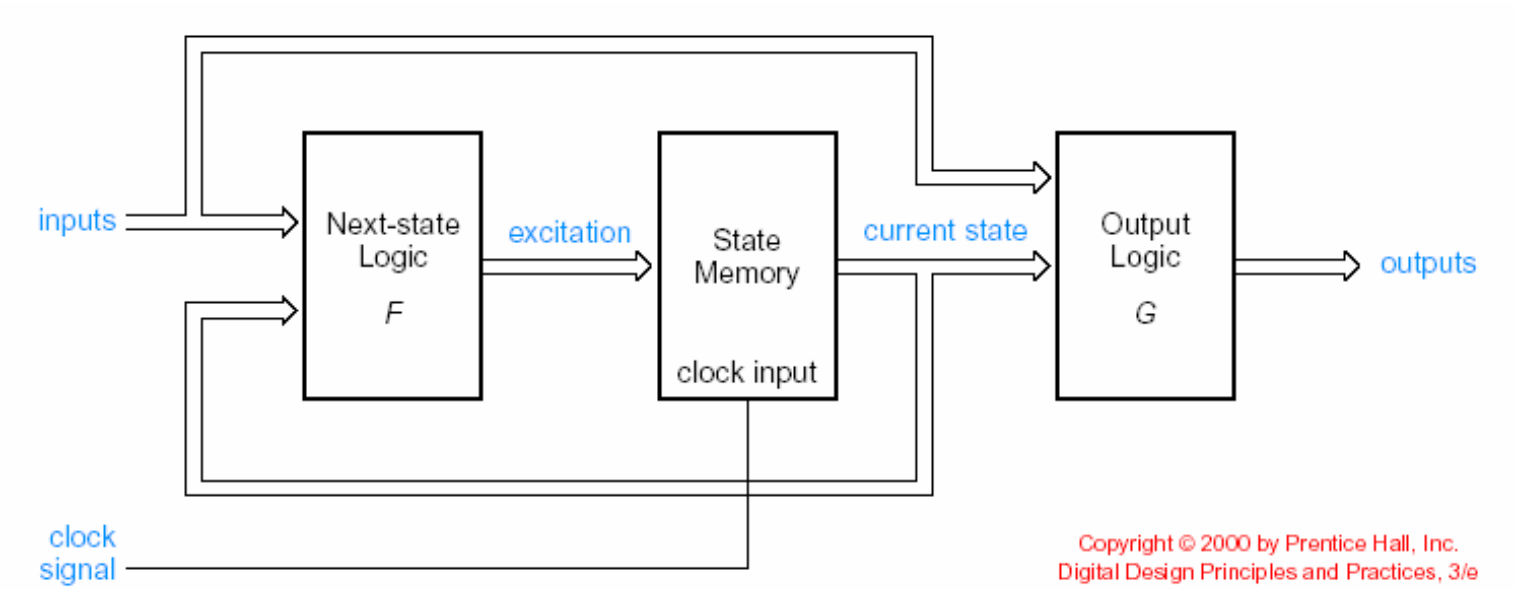
Problem Interpretation

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

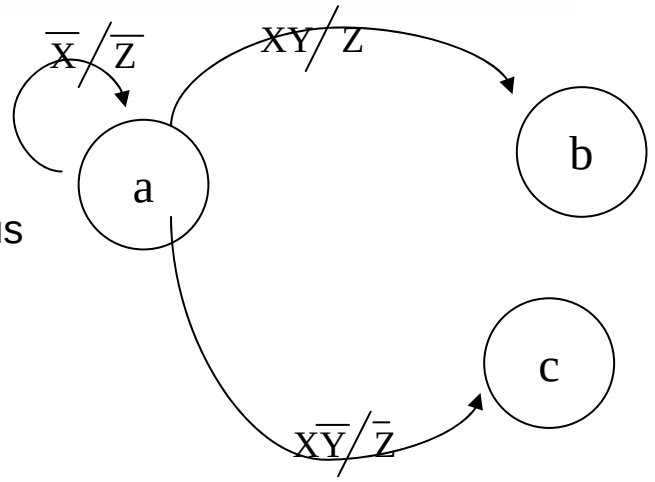




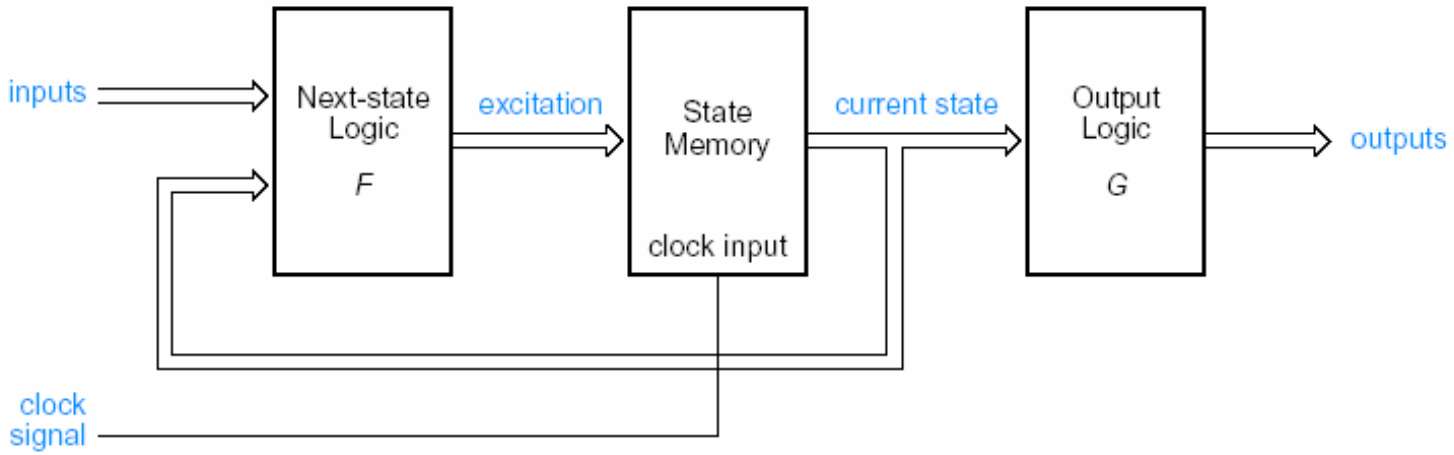
Mealy Machine



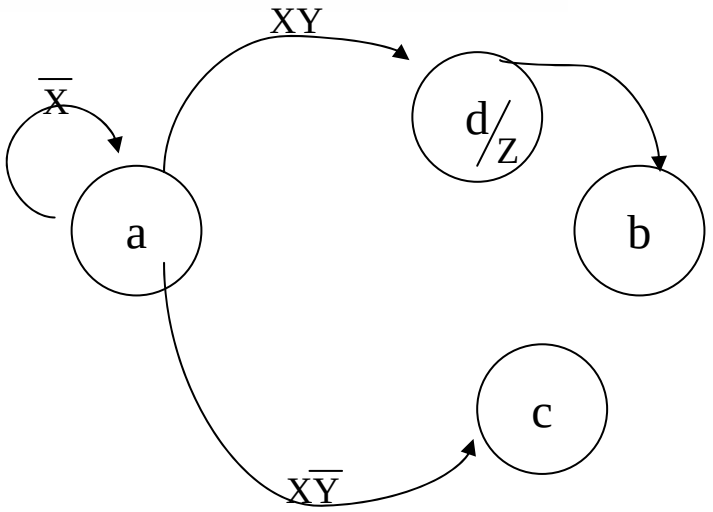
- Output are a function of P.S. and inputs
- Output associated with transition
- Tend to give the fewest states necessary
- Can cause problems when the inputs are asynchronous



Moore Machine

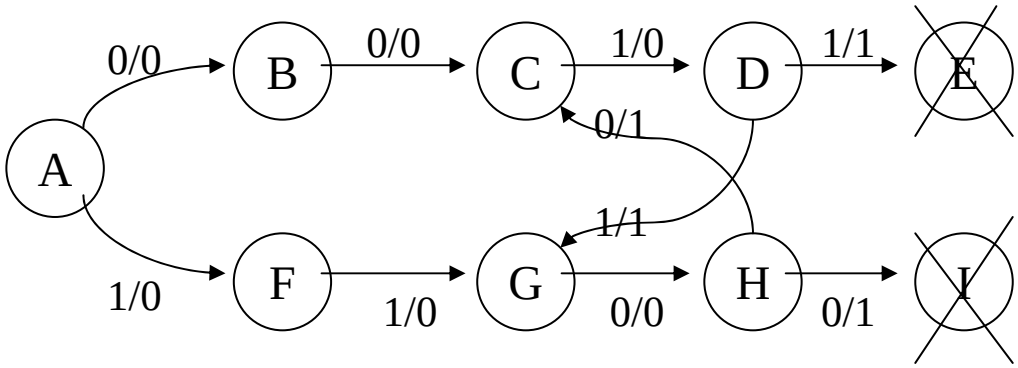


- Output are a function of P.S.
- Output associated with state
- Tend to have more states
- Output change only with a state change
  - do not rely on asynchronous inputs



Step 1: State/Output Diagram

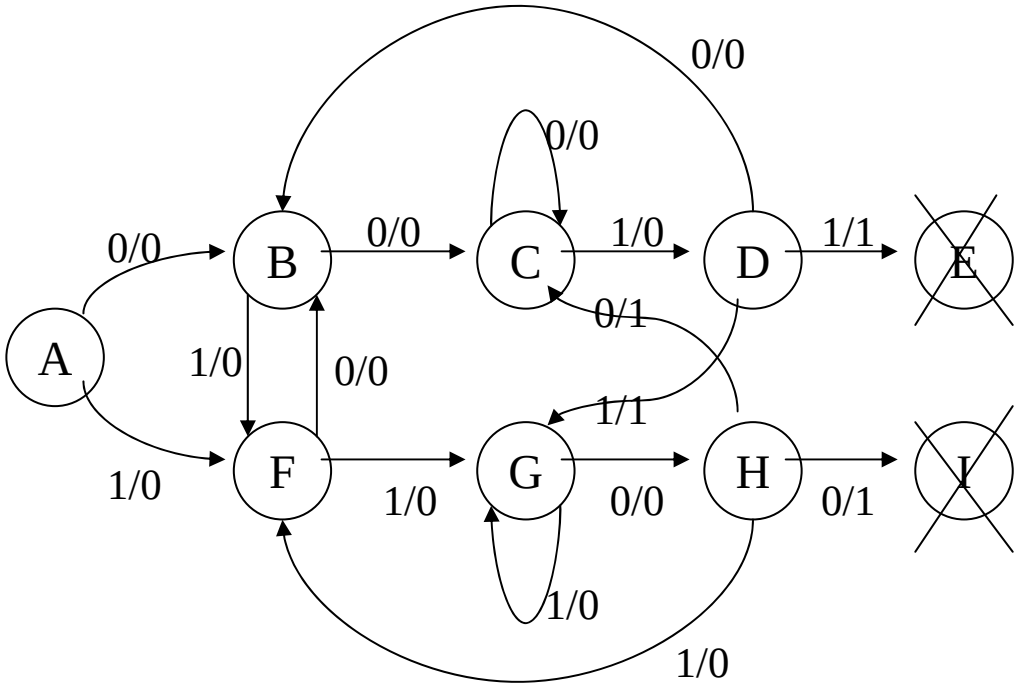
- Draw bubbles for all correct sequences



- Meanings: <u>A state is a meaningful abstraction of previous history</u> (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

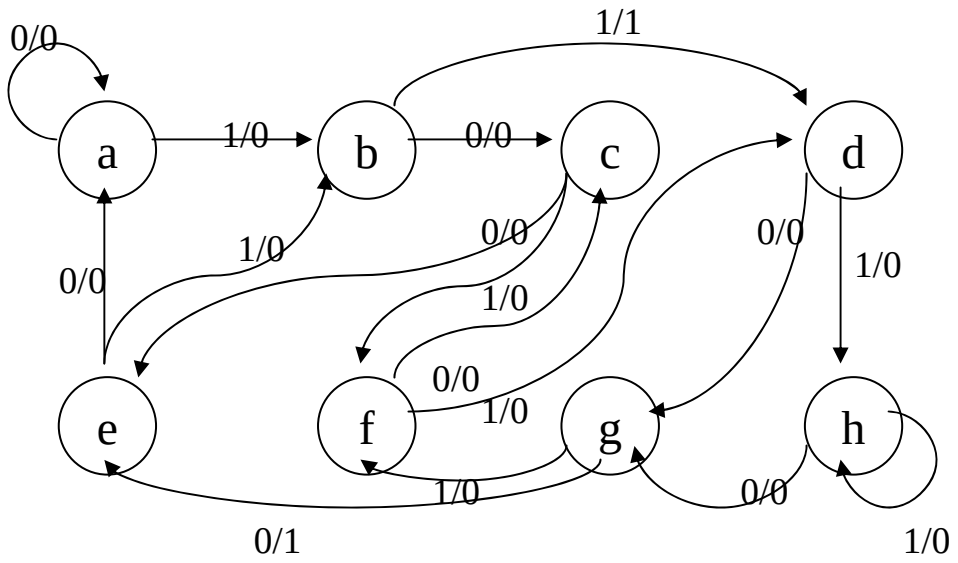
Step 1: State/Output Diagram

- Adding other input sequences



Step 1: State/Output Diagram

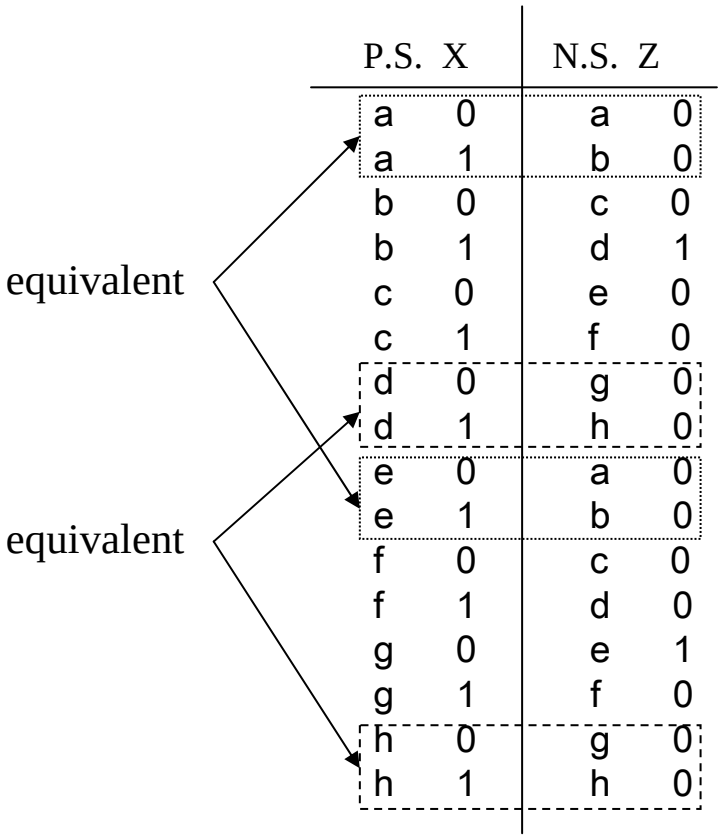
- Another approach
  - A state is understood as remembering something (previous history)
  - All we have to remember is the last three inputs
  - Eight states will be needed
    - (a=000, b=001, c=010, d=011, e=100, f=101, g=110, h=111)



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

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
d	1	h	0
e	0	a	0
e	1	b	0
f	0	c	0
f	1	d	0
g	0	e	1
g	1	f	0
h	0	g	0
h	1	h	0

a = e (got x00) ↔ C (got 00)

b (got 001) ↔ D (got 001)

c (got 010) ↔ B (got 0)

d = h (got x11) ↔ G (got 11)

f (got 101) ↔ F (got 1)

g (got 110) ↔ H (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

State Name	Assignment			
	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_CQ_B \ Q_AX	00	01	11	10
00	0	0	0	0
01	1	0	1	0
11	0	0	1	0
10	-	-	-	-

Minimum Cost

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

Q_CQ_B \ Q_AX	00	01	11	10
00	0	0	1	1
01	1	1	1	0
11	1	1	1	0
10	-	-	-	-

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

Q_CQ_B \ Q_AX	00	01	11	10
00	0	1	0	1
01	1	0	0	0
11	1	0	0	0
10	-	-	-	-

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

Q_CQ_B \ Q_AX	00	01	11	10
00	0	0	1	0
01	0	0	0	0
11	0	0	0	1
10	-	-	-	-

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

Step 9: Logic Diagram

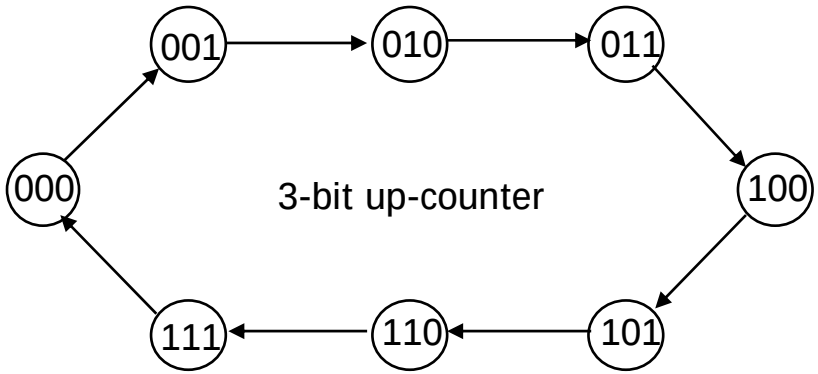
- Will skip
- How much logic?
  - 14 NANDs
  - 3 F/Fs
  - 1 Invertor
- Minimum Cost Design: Unused states assigned for “minimum cost”
  - Risk?

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

3-bit Binary Counter

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



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

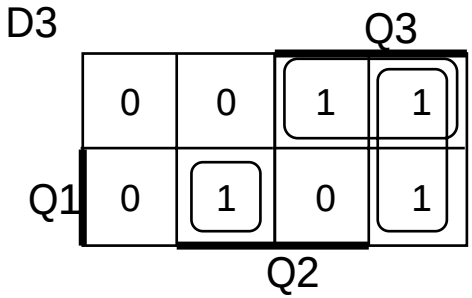
For the 3-bit up counter, here is the state transition table. It's like there are three inputs and three outputs. In this case the literals for states are the inputs for the state transition. If there are other outside inputs, those should be also written in the table.

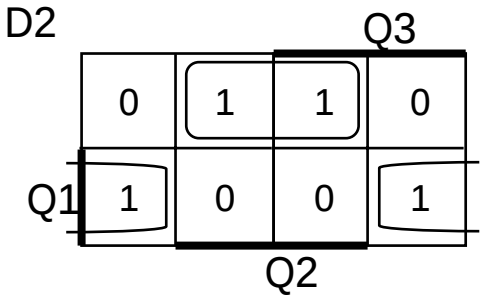
Implementation

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

Q3	Q2	Q1	D3	D2	D1
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

D1 <= Q1'
D2 <= Q1Q2' + Q1'Q2
<= Q1 xor Q2
D3 <= Q1Q2Q3' + Q1'Q3 + Q2'Q3
<= (Q1Q2)Q3' + (Q1' + Q2')Q3
<= (Q1Q2)Q3' + (Q1Q2)'Q3
<= (Q1Q2) xor Q3





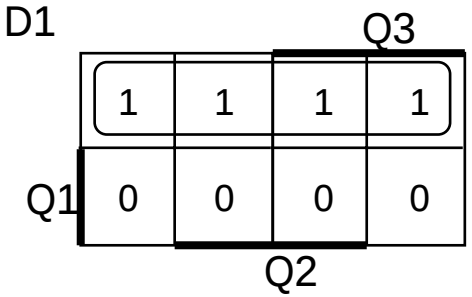
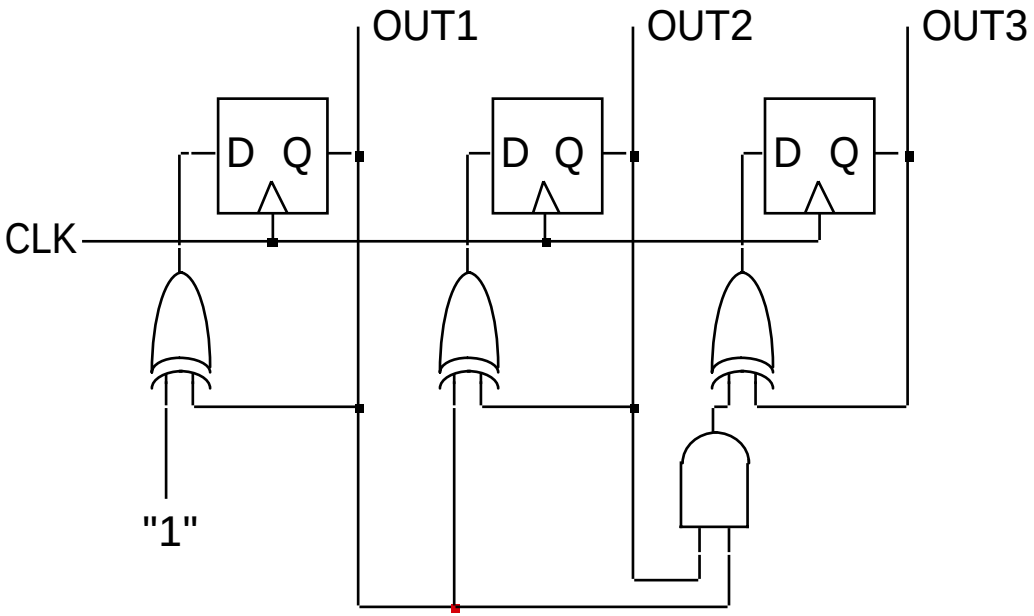


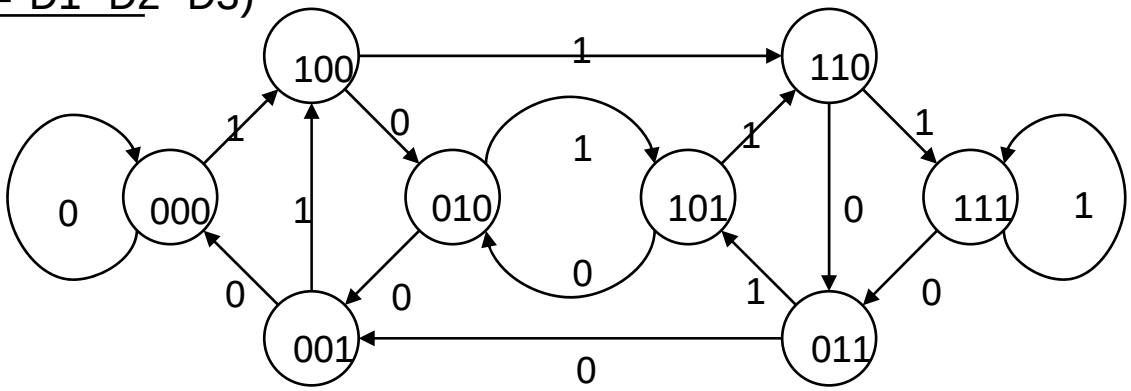
Diagram for the 3-bit Binary Counter



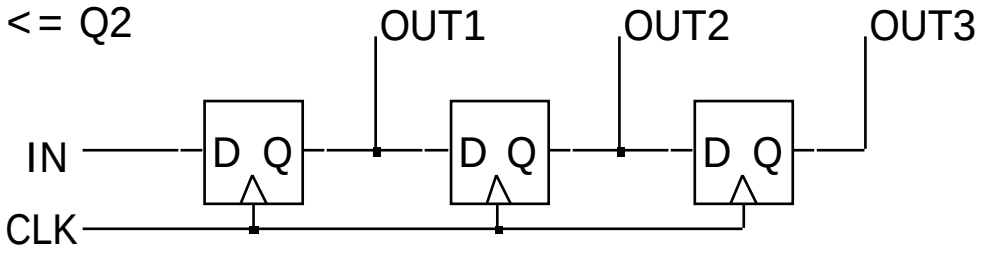
Back to the shift register

- Input determines next state

In	Q1	Q2	Q3	NQ1	NQ2	NQ3 (= D1	D2	D3)
0	0	0	0	0	0	0		
0	0	0	1	0	0	0		
0	0	1	0	0	0	1		
0	0	1	1	0	0	1		
0	1	0	0	0	1	0		
0	1	0	1	0	1	0		
0	1	1	0	0	1	1		
0	1	1	1	0	1	1		
1	0	0	0	1	0	0		
1	0	0	1	1	0	0		
1	0	1	0	1	0	1		
1	0	1	1	1	0	1		
1	1	0	0	1	1	0		
1	1	0	1	1	1	0		
1	1	1	0	1	1	1		
1	1	1	1	1	1	1		



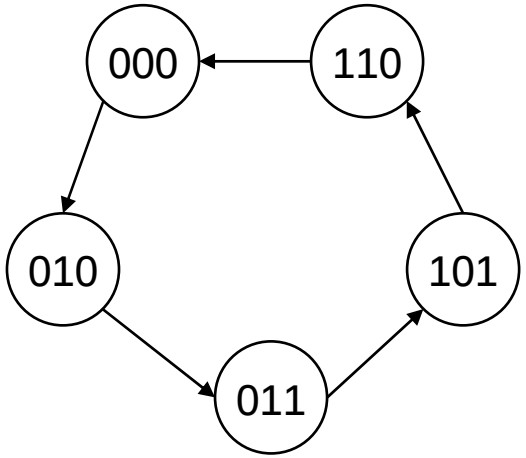
D1 <= In
D2 <= Q1
D3 <= Q2



Here is the state transition table for a 3bit shift register. In this case, there is one external input in addition to the current 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



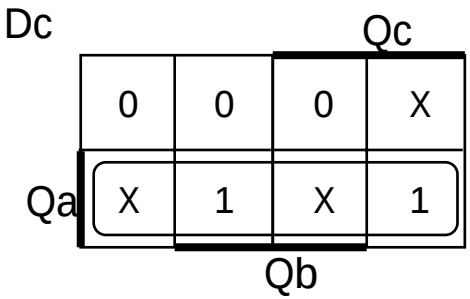
Present State			Next State		
Qc	Qb	Qa	Qc	Qb	Qa
0	0	0	0	1	0
0	0	1	–	–	–
0	1	0	0	1	1
0	1	1	1	0	1
1	0	0	–	–	–
1	0	1	1	1	0
1	1	0	0	0	0
1	1	1	–	–	–

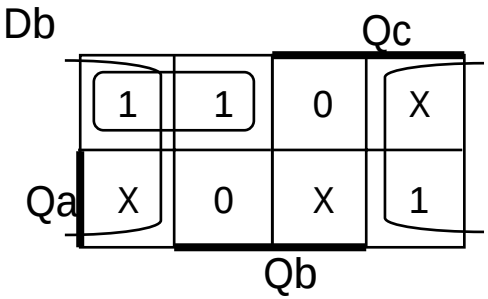
note the don't care conditions that arise from the unused state codes

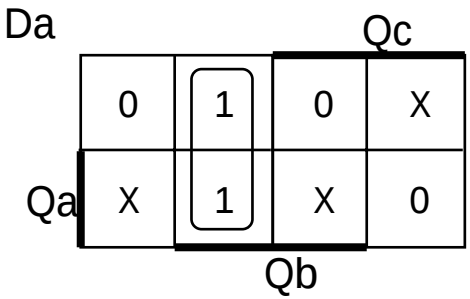
In this case, only five states are used out of 8 possible binary values; so three don't care cases appear. Note that the next state literals are denoted with + symbol.

More complex counter example (cont'd)

- Step 3: K-maps for next state functions







Dc = Qa

Db = Qb' + Qa'Qc'

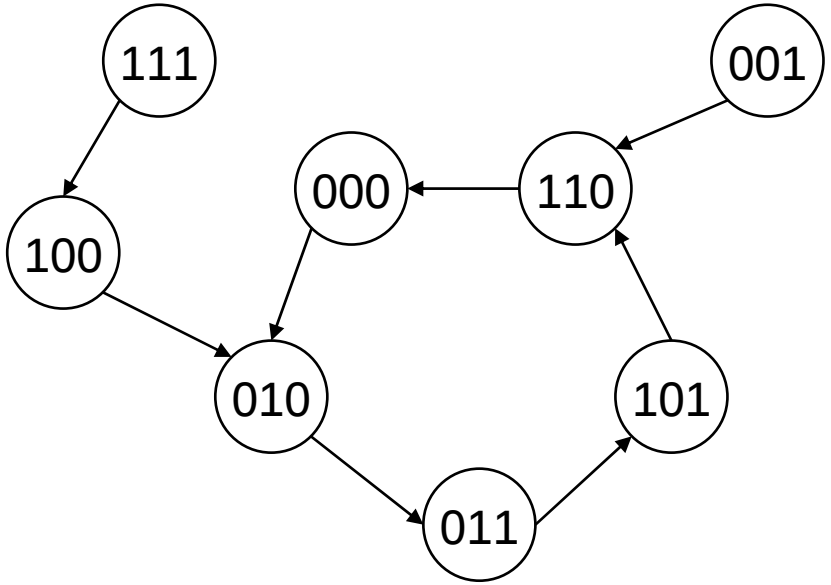
Da = QbQc'

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	0	1	0	0



Dc

Qa \ Qb	00	01	11	10
0	0	0	0	0
1	1	1	1	1

Db

Qa \ Qb	00	01	11	10
0	1	1	0	1
1	1	0	0	1

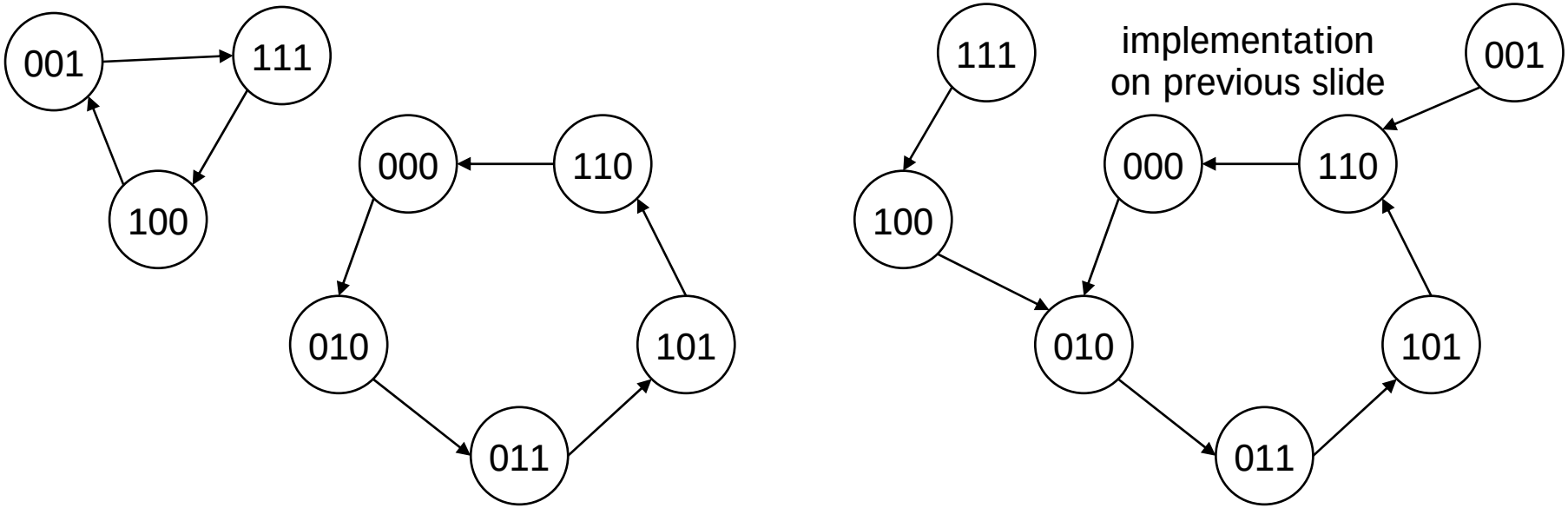
Da

Qa \ Qb	00	01	11	10
0	0	1	0	0
1	0	1	0	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

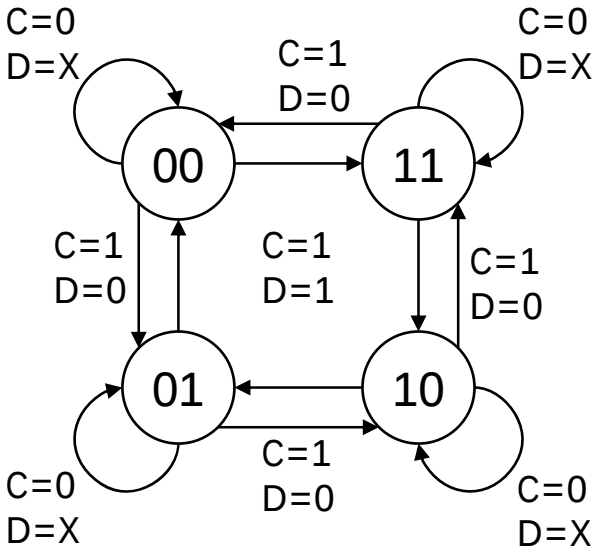
- Start-up states
  - at power-up, counter may be in an unused or invalid state
  - designer must guarantee that it (eventually) enters a valid state
- Self-starting solution
  - design counter so that invalid states eventually transition to a valid state
  - may limit exploitation of don't cares



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

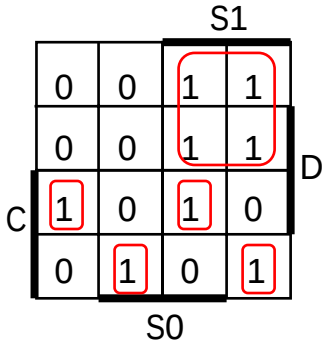
- 2-bit up-down counter (2 inputs)
  - direction: D = 0 for up, D = 1 for down
  - count: C = 0 for hold, C = 1 for count



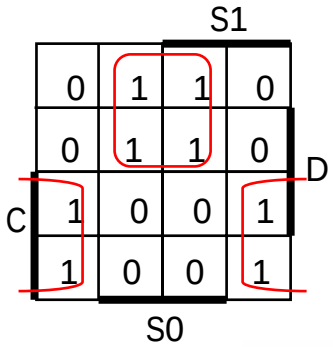
Q1	Q0	C	D	NQ1	NQ0
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	1	1
0	1	0	0	0	1
0	1	0	1	0	1
0	1	1	0	1	0
0	1	1	1	0	0
1	0	0	0	1	0
1	0	0	1	1	0
1	0	1	0	1	1
1	0	1	1	0	1
1	1	0	0	1	1
1	1	0	1	1	1
1	1	1	0	0	0
1	1	1	1	1	0

Activity (cont'd)

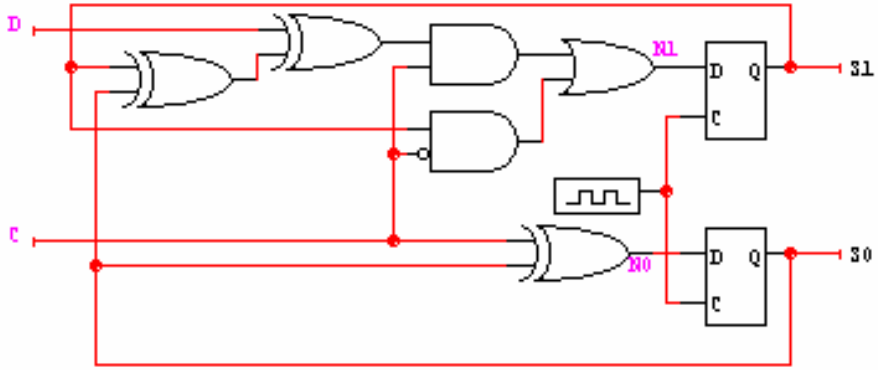
Q1	Q0	C	D	D1	D0
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	1	1
0	1	0	0	0	1
0	1	0	1	0	1
0	1	1	0	1	0
0	1	1	1	0	0
1	0	0	0	1	0
1	0	0	1	1	0
1	0	1	0	1	1
1	0	1	1	0	1
1	1	0	0	1	1
1	1	0	1	1	1
1	1	1	0	0	0
1	1	1	1	1	0



$$D1 = C'S1 + CDS0'S1' + CDS0S1 + CD'S0S1' + CD'S0'S1 = C'S1 + C(D'(S1 \oplus S0) + D(S1 \equiv S0))$$

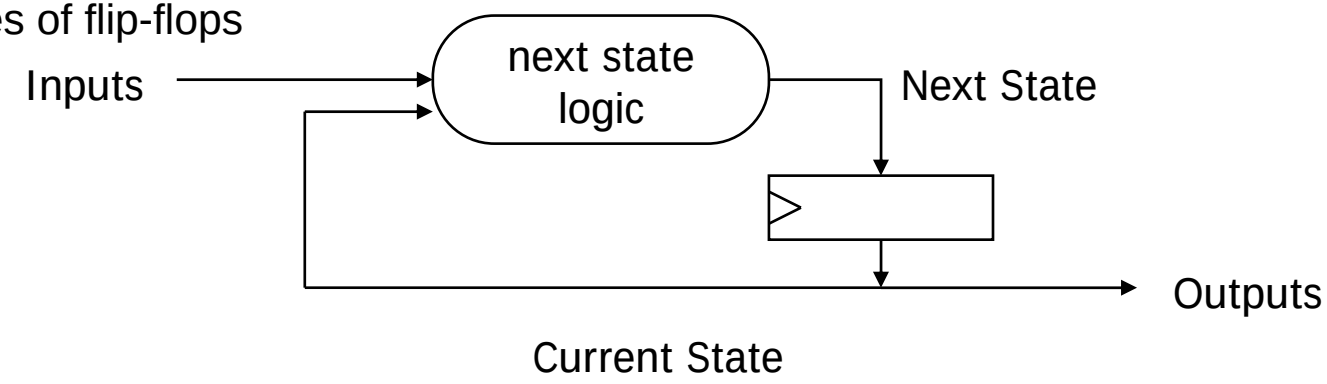


$$D0 = CS0' + C'S0$$



Counter/shift-register model

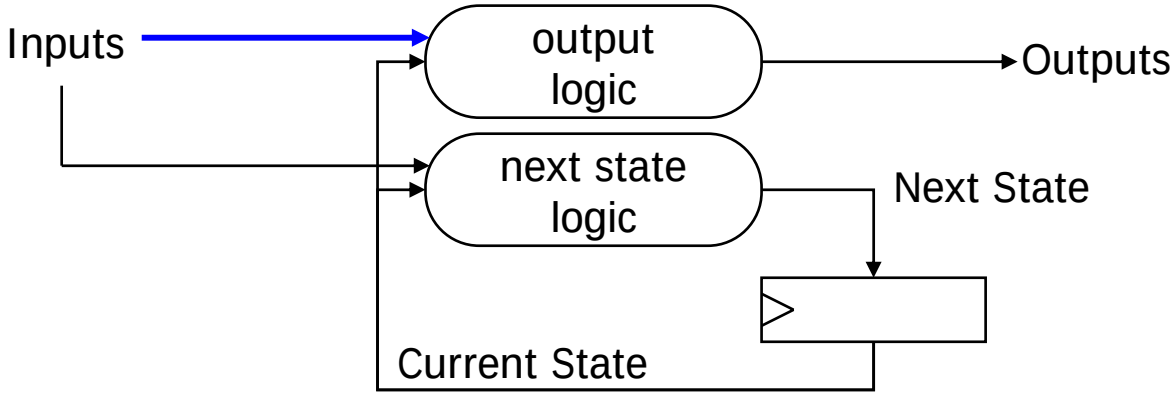
- Values stored in registers represent the state of the circuit
- Combinational logic computes:
  - next state
    - function of current state and inputs
  - outputs
    - values of flip-flops



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

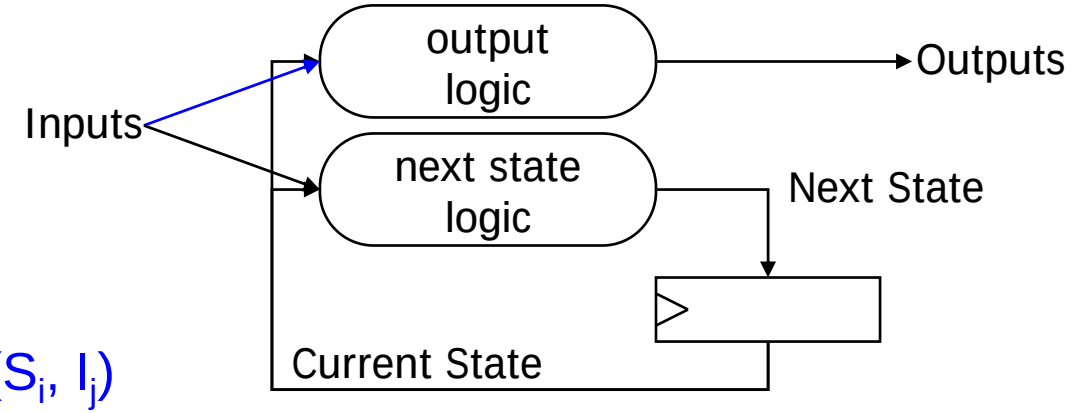
- Values stored in registers represent the state of the circuit
- Combinational logic computes:
  - next state
    - function of current state and inputs
  - outputs
    - function of current state and inputs (Mealy machine)
    - function of current state only (Moore machine)

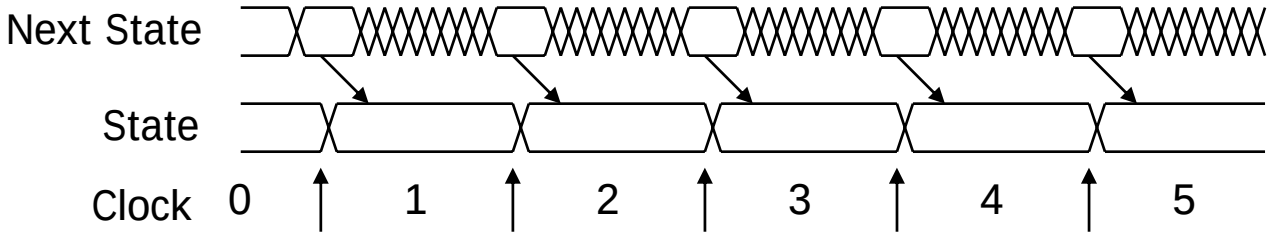


If output is different from the state, there should be one more combinational logic, the upper oval. There is another important classification: depending on the combinational logic for outputs. If outputs are functions of only current state, that model is called a Moore machine. On the other hand, if 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, ..., S_k
- Inputs: I_1, I_2, ..., I_m
- Outputs: O_1, O_2, ..., 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.

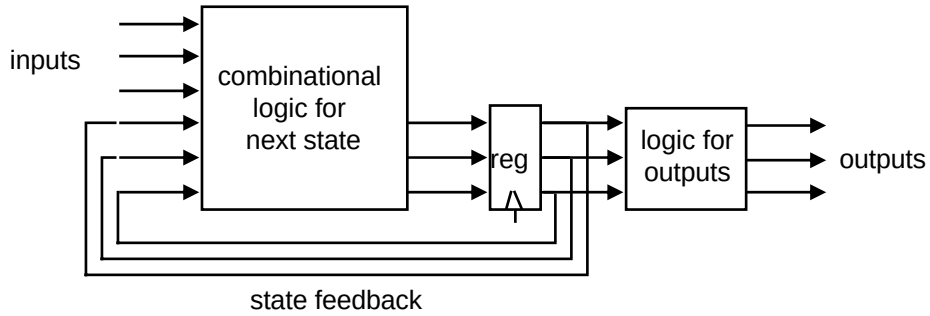
Comparison of Mealy and Moore machines

- 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

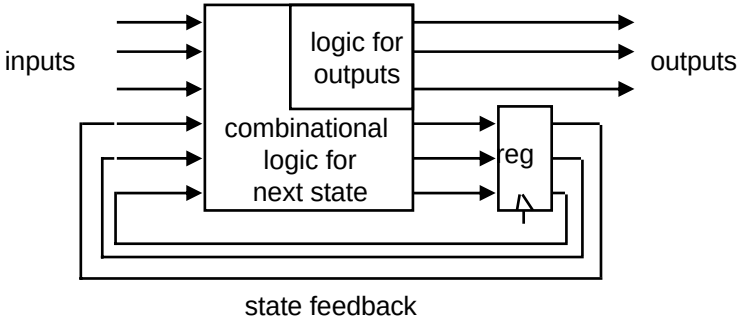
As outputs of a Mealy machine are functions of the external inputs and the present state, the number of states may be less. Information for the next state transition is split between inputs from outside and the state. In Moore machines, outputs are dependent only on the present state, the output will 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



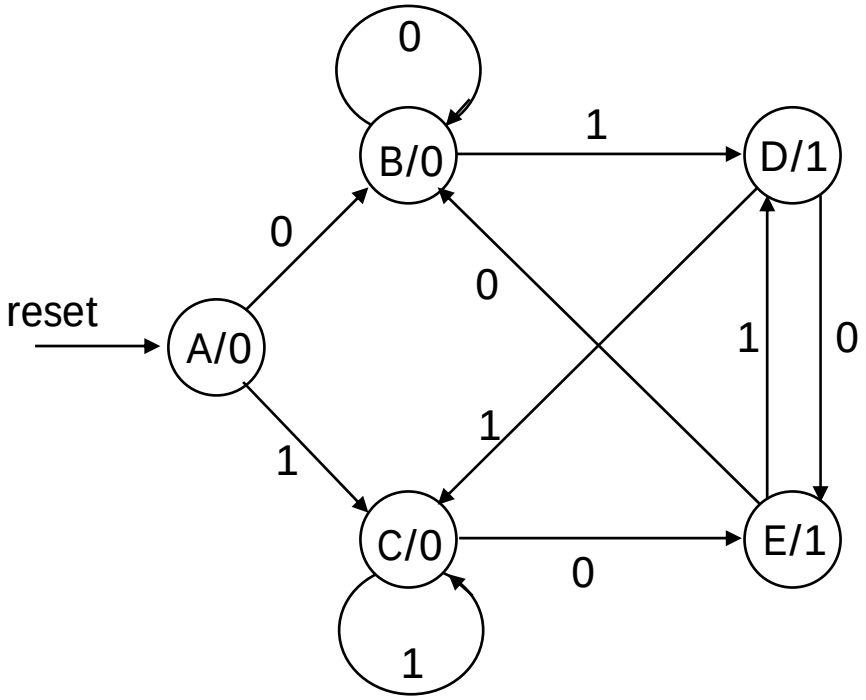
- Mealy



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

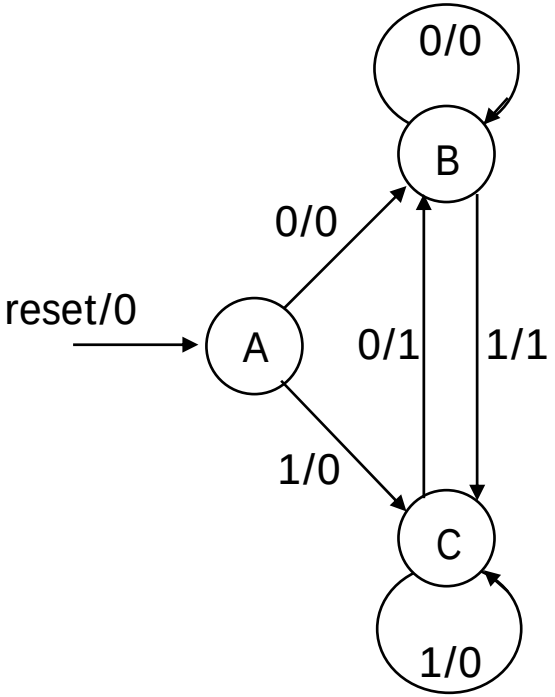


reset	input	current state	next state	output
1	–	–	A	
0	0	A	B	0
0	1	A	C	0
0	0	B	B	0
0	1	B	D	0
0	0	C	E	0
0	1	C	C	0
0	0	D	E	1
0	1	D	C	1
0	0	E	B	1
0	1	E	D	1

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



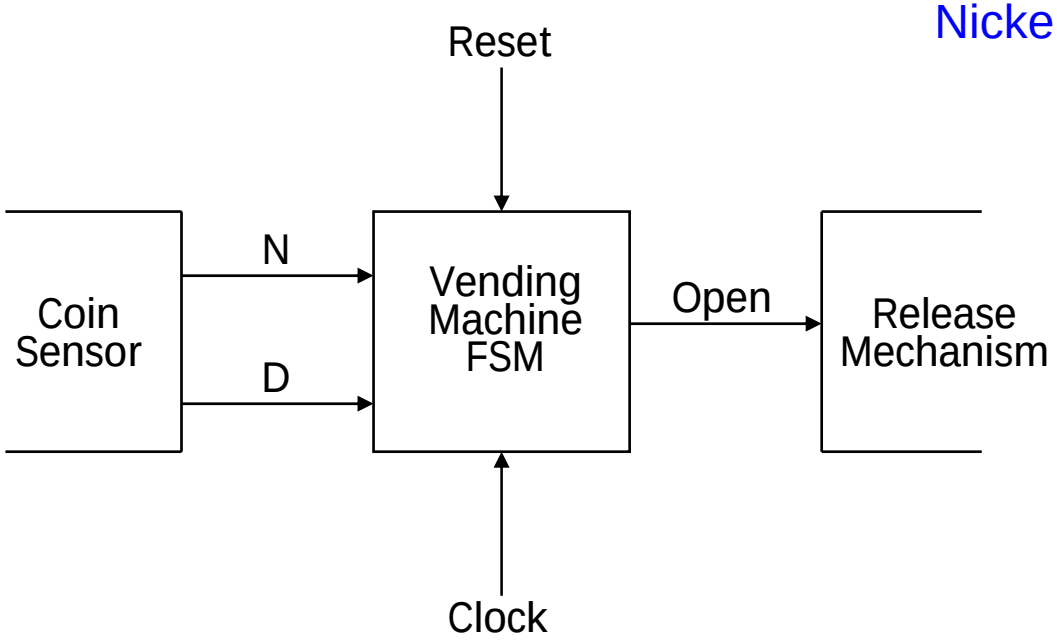
reset	input	current state	next state	output
1	–	–	A	0
0	0	A	B	0
0	1	A	C	0
0	0	B	B	0
0	1	B	C	1
0	0	C	B	1
0	1	C	C	0

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

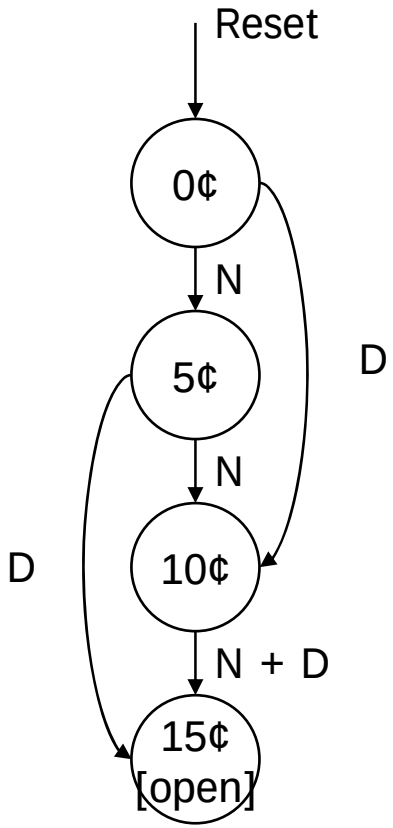
Dime: 10 cent coin
Nickel: 5 cent coin



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 D	N	next state	output open
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)

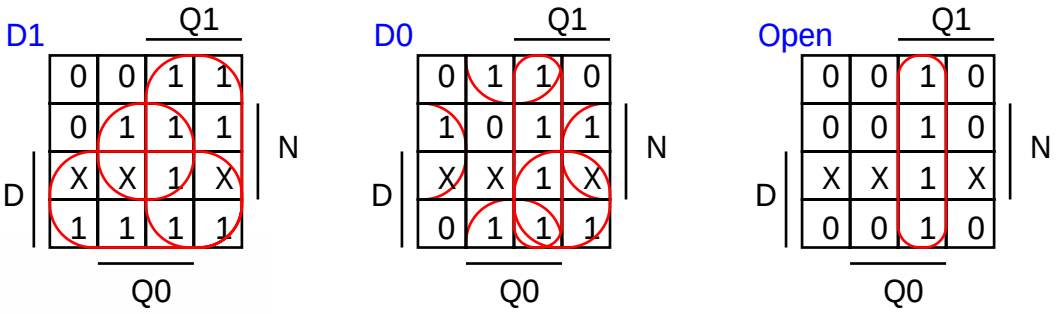
- Uniquely encode states

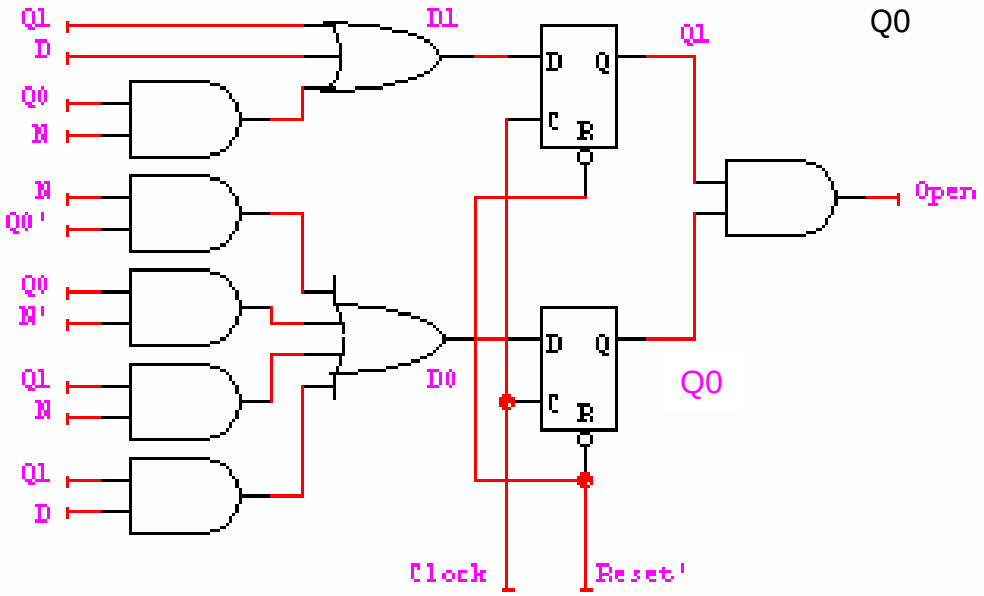
present state Q1	Q0	inputs D	N	next state NQ1	NQ0	output open
0	0	0	0	0	0	0
		0	1	0	1	0
		1	0	1	0	0
		1	1	–	–	–
0	1	0	0	0	1	0
		0	1	1	0	0
		1	0	1	1	0
		1	1	–	–	–
1	0	0	0	1	0	0
		0	1	1	1	0
		1	0	1	1	0
		1	1	–	–	–
1	1	–	–	1	1	1

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

D0 = Q0 D' N'

D1 = Q0 N + Q1 D' N'

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

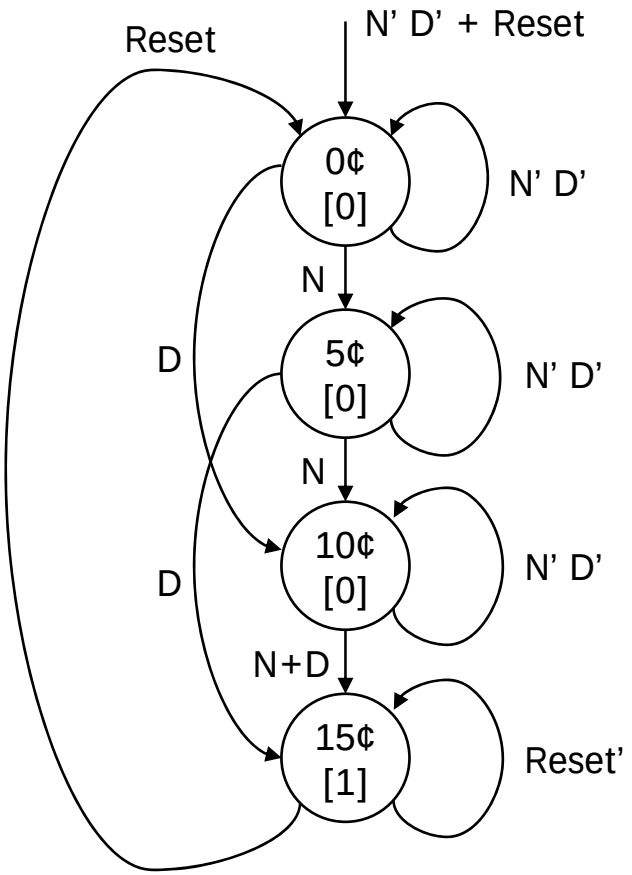
D3 = Q1 D + Q2 D + Q2 N + Q3

OPEN = Q3

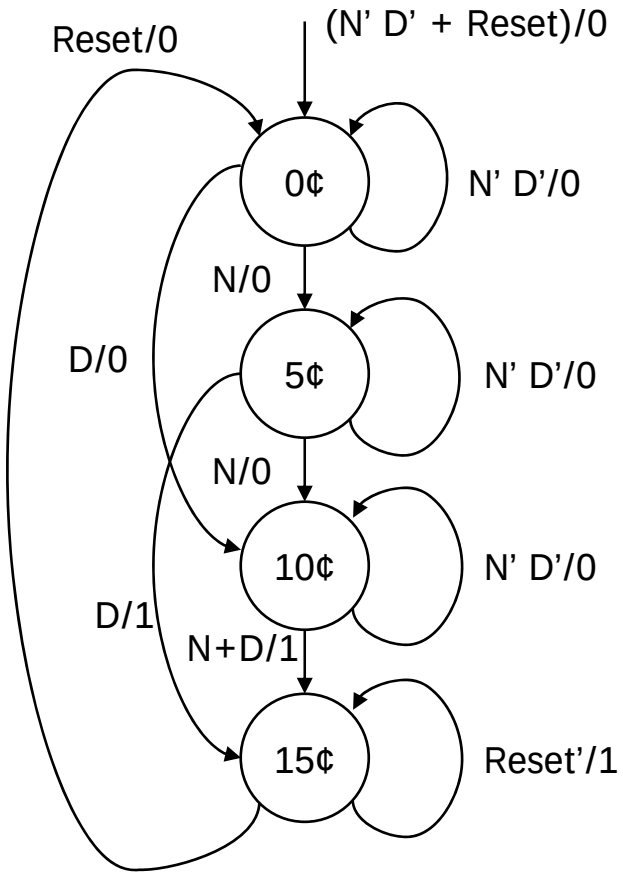
One-hot encoding means only one bit is ON for each state; that is, only one state variable is set, or "hot," for each state. So the number of bits to represent states is the same as the number of states. The benefit is that the next state generation function may be simple since the number of product 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

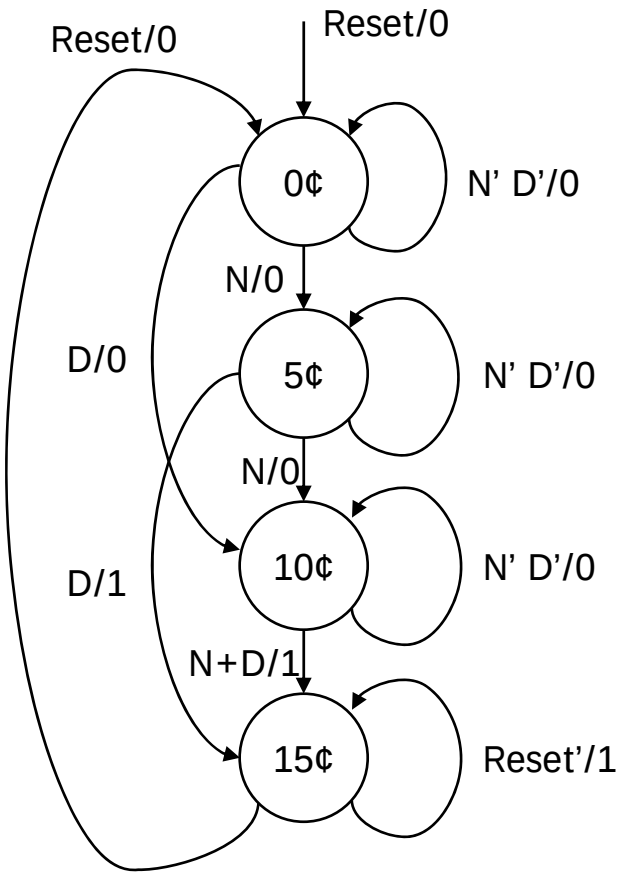


- Mealy machine
  - outputs associated with transitions

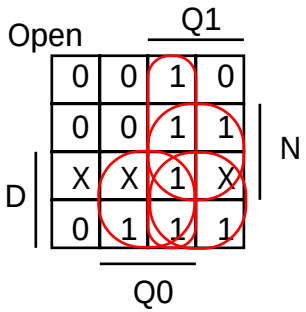


This slide shows the complete state transition diagram of the vending machine in two versions. In the Moore model, the number in [ ] is the output. Whereas, in the Mealy model, the output is associated with each arc.

Example: Mealy implementation



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



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

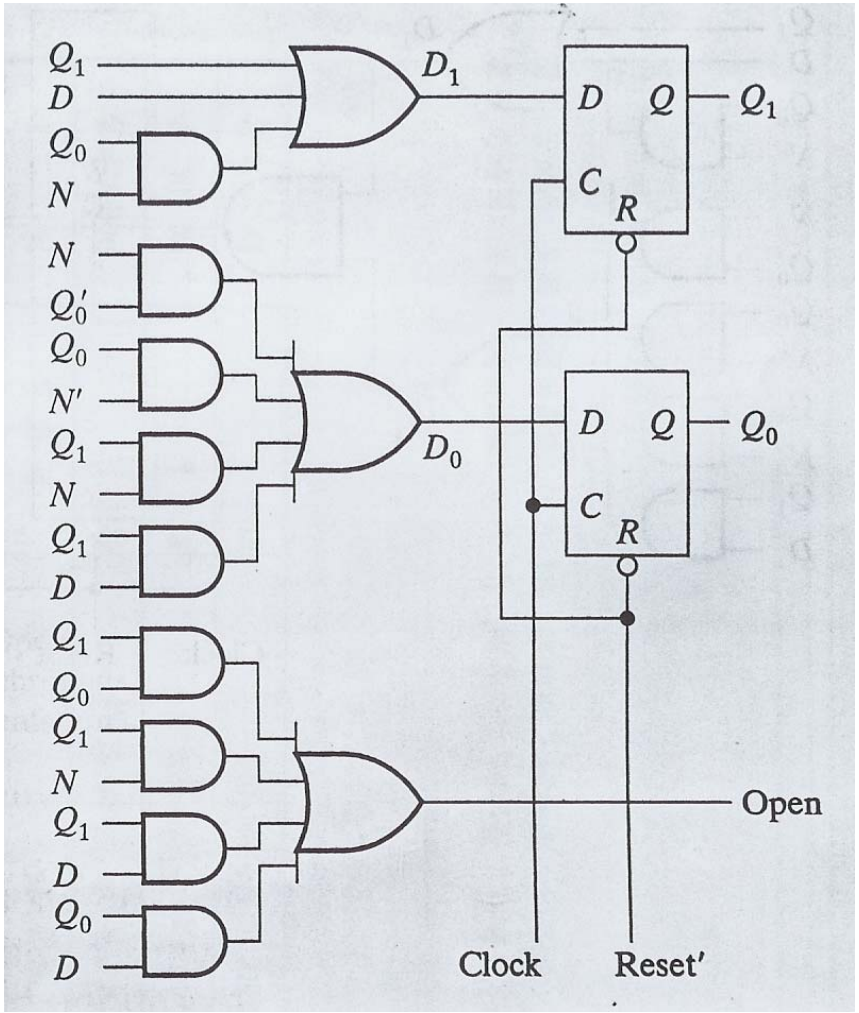
D1 = Q1 + D + Q0N

OPEN = Q1Q0 + Q1N + Q1D + Q0D

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

Example: Mealy implementation

D0 = Q0'N + Q0N' + Q1N + Q1D
D1 = Q1 + D + Q0N
OPEN = Q1Q0 + Q1N + Q1D + Q0D

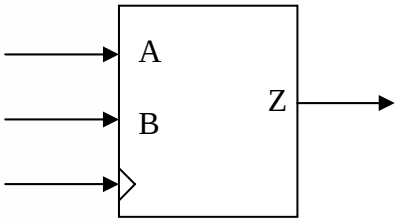


Here is the overall implementation of the vending machine based on the Mealy model.

Hardware Description Languages
VHDL for FSM

	A B				
S	***00***	***01***	***11***	***10***	***Z***
INIT	A0	A0	A1	A1	0
A0	OK0	OK0	A1	A1	0
A1	A0	A0	OK1	OK1	0
OK0	OK0	OK0	OK1	A1	1
OK1	A0	OK0	OK1	OK1	1
			S*		

```
library IEEE;
use IEEE.std_logic_1164.all;

entity smexamp is
  port ( CLOCK, A, B: in STD_LOGIC;
         Z: out STD_LOGIC );
end;

architecture smexamp_arch of smexamp is
type Sreg_type is (INIT, A0, A1, OK0, OK1);
signal Sreg: Sreg_type;
begin

  process (CLOCK) -- state-machine states and transitions
  begin
    if CLOCK'event and CLOCK = '1' then
      case Sreg is
        when INIT => if    A='0' then Sreg <= A0;
                     elsif A='1' then Sreg <= A1;   end if;
        when A0 =>   if    A='0' then Sreg <= OK0;
                     elsif A='1' then Sreg <= A1;  end if;
        when A1 =>   if    A='0' then Sreg <= A0;
                     elsif A='1' then Sreg <= OK1;  end if;
        when OK0 =>  if    A='0' then Sreg <= OK0;
                     elsif A='1' and B='0' then Sreg <= A1;
                    elsif A='1' and B='1' then Sreg <= OK1;  end if;
        when OK1 =>  if    A='0' and B='0' then Sreg <= A0;
                     elsif A='0' and B='1' then Sreg <= OK0;
                     elsif A='1' then Sreg <= OK1;            end if;
        when others => Sreg <= INIT;
      end case;
    end if;
  end process;

  with Sreg select  -- output values based on state
    Z <= '0' when INIT | A0 | A1,
         '1' when OK0 | OK1,
         '0' when others;

end smexamp_arch;
```



Define a new type of signal for use in symbolic state table

Selected assignment statement

- for multiple cases

Assigning specific codes to the states?

- Not necessary since synthesis tools will do for you
- Sometimes necessary
  - Alternative 1: Use constant definition
  - Alternative 2: use Synopsys “attribute” enum_encoding

```
library IEEE;
use IEEE.std_logic_1164.all;
...
architecture smexampc_arch of smexamp is
subtype Sreg_type is STD_LOGIC_VECTOR (1 to 4);
constant INIT: Sreg_type := "0000";
constant A0  : Sreg_type := "0001";
constant A1  : Sreg_type := "0010";
constant OK0 : Sreg_type := "0100";
constant OK1 : Sreg_type := "1000";
signal Sreg: Sreg_type;
...
```

```
library IEEE;
use IEEE.std_logic_1164.all;
library SYNOPSYS;
use SYNOPSYS.attributes.all;
...
architecture smexampe_arch of smexamp is
type Sreg_type is (INIT, A0, A1, OK0, OK1);
attribute enum_encoding of Sreg_type: type is
          "0000 0001 0010 0100 1000";
signal Sreg: Sreg_type;
...
```

Finite state machines summary

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

We start with simple FSMs like counters and shift registers, where states are outputs directly. We should differentiate Moore and Mealy models. With either model, we should be able to design a FSM.