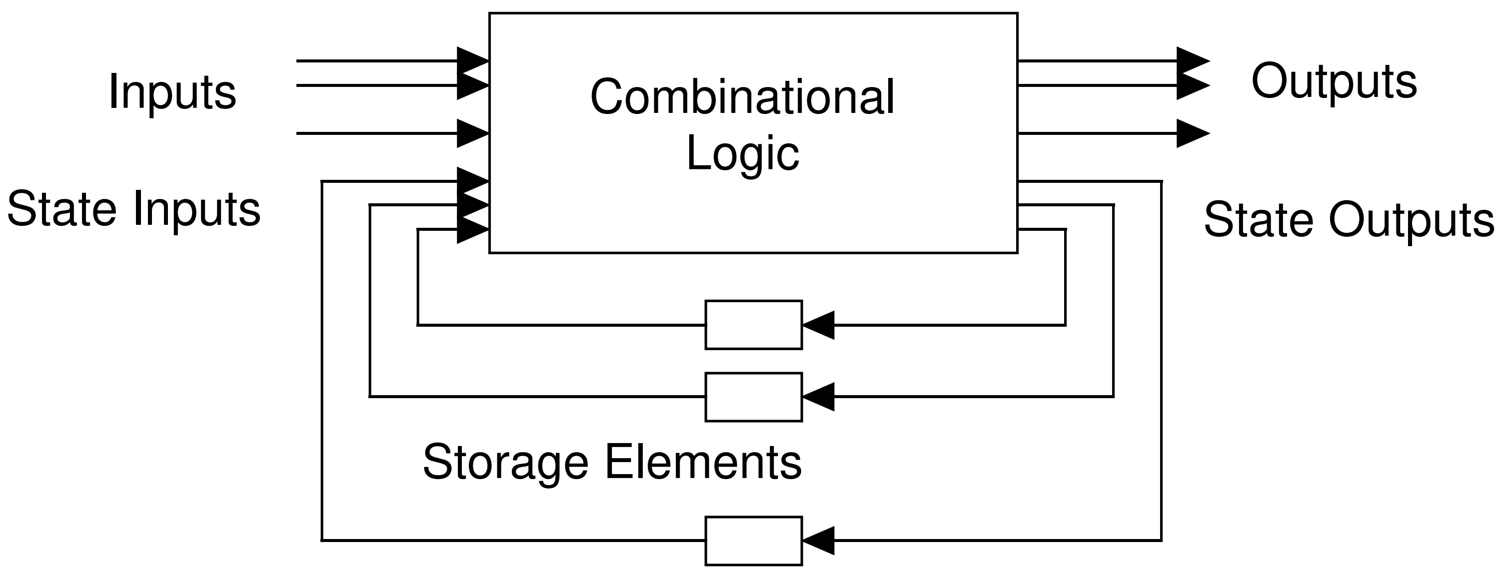
Sequential logic implementation

- **Sequential circuits**
  - primitive sequential elements
  - combinational logic
- **Models for representing sequential circuits**
  - finite-state machines (Moore and Mealy)
  - representation of memory (states)
  - changes in state (transitions)
- **Basic sequential circuits**
  - shift registers
  - counters
- **Design procedure**
  - state diagrams
  - state transition table
  - next state functions

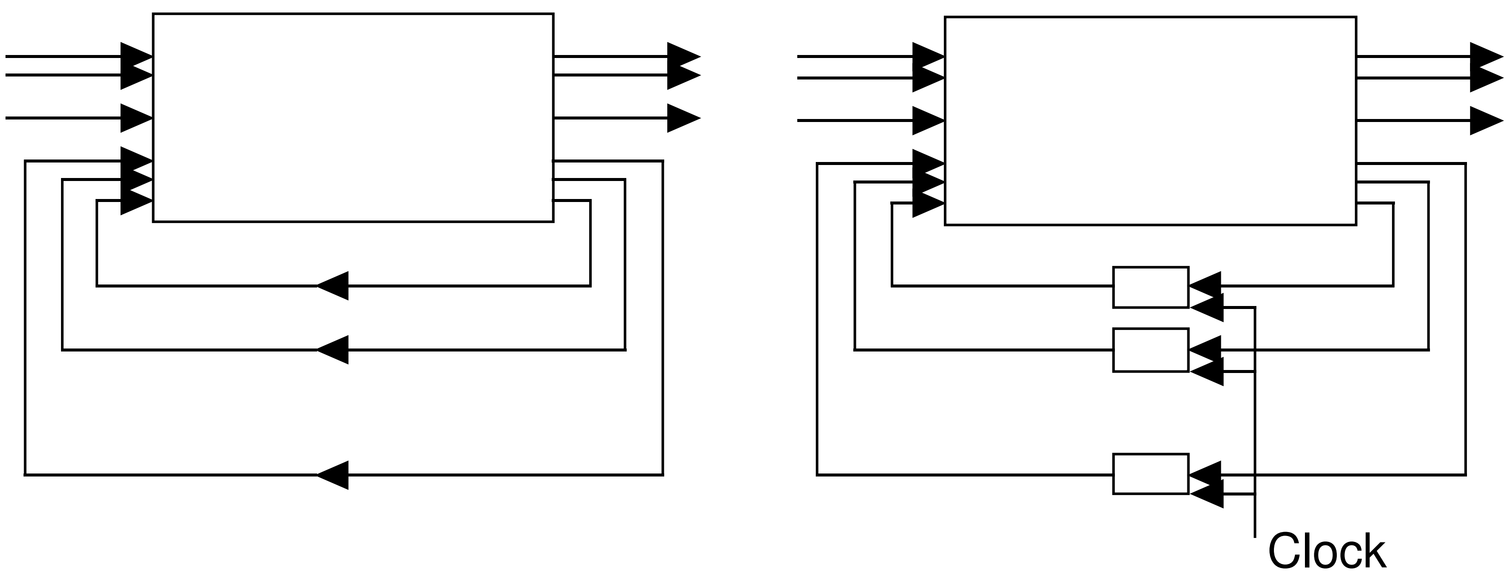
Abstraction of state (or memory) elements

- **Separate state elements from logic**
  - so that the two can be treated independently
- **Localize the feedback loops**
  - make it easy to break cycles
- **Implementation of storage elements for state**
  - leads to different families of sequential logic



Families of sequential logic

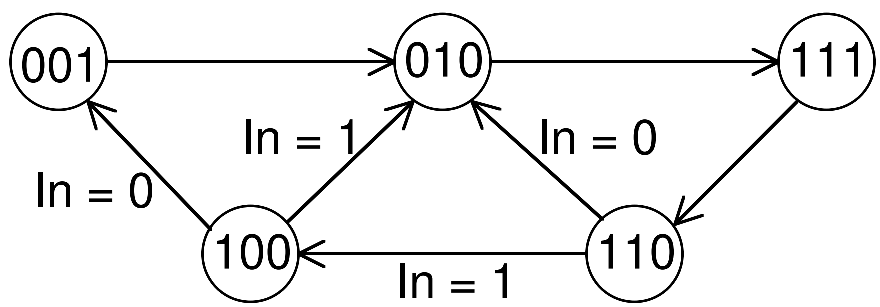
- **Asynchronous sequential logic**
  - state changes are not explicitly controlled
  - state outputs affect state inputs immediately
  - storage elements may be simple wires or delay elements
- **Synchronous sequential logic**
  - state changes occur in lock step across all storage elements
  - using a clock signal – a periodic waveform



Finite state machine representations

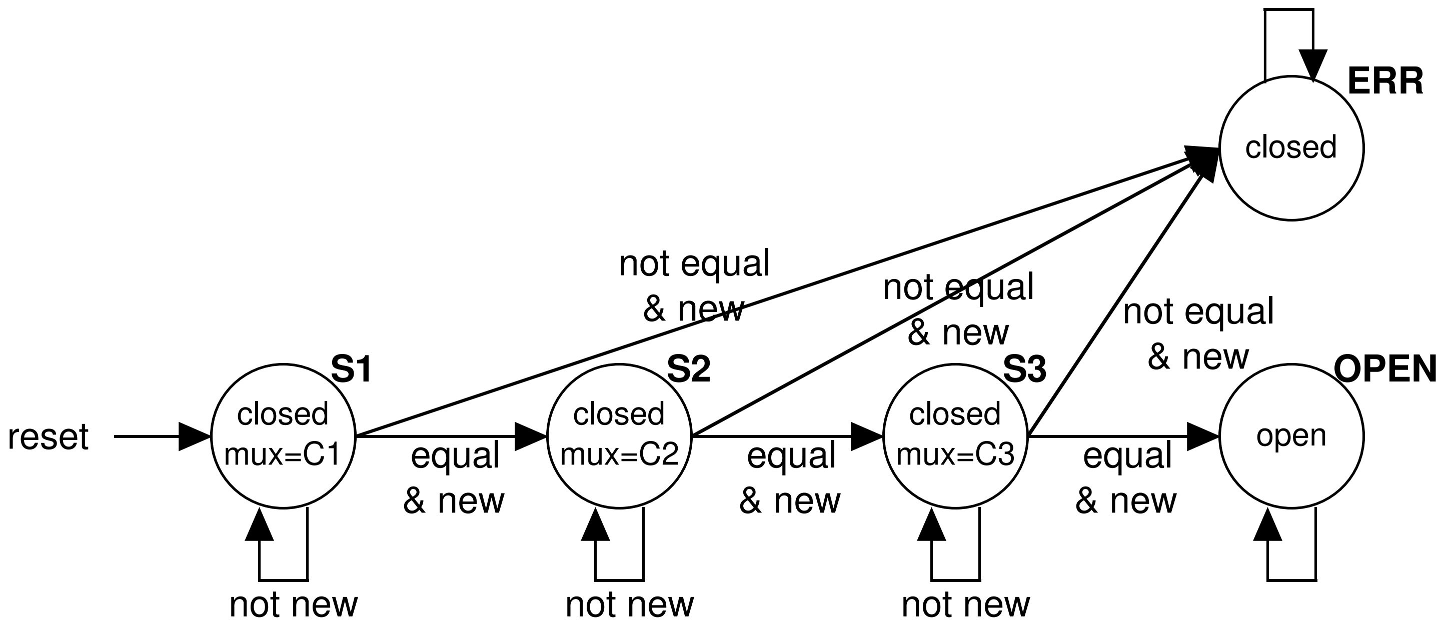
- **States: determined by possible values in sequential storage elements**
- **Transitions: change of state**
- **Clock: controls when state can change by controlling storage elements**

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



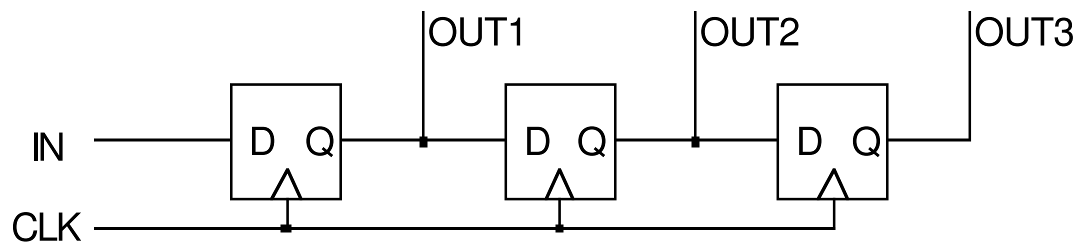
Example finite state machine diagram

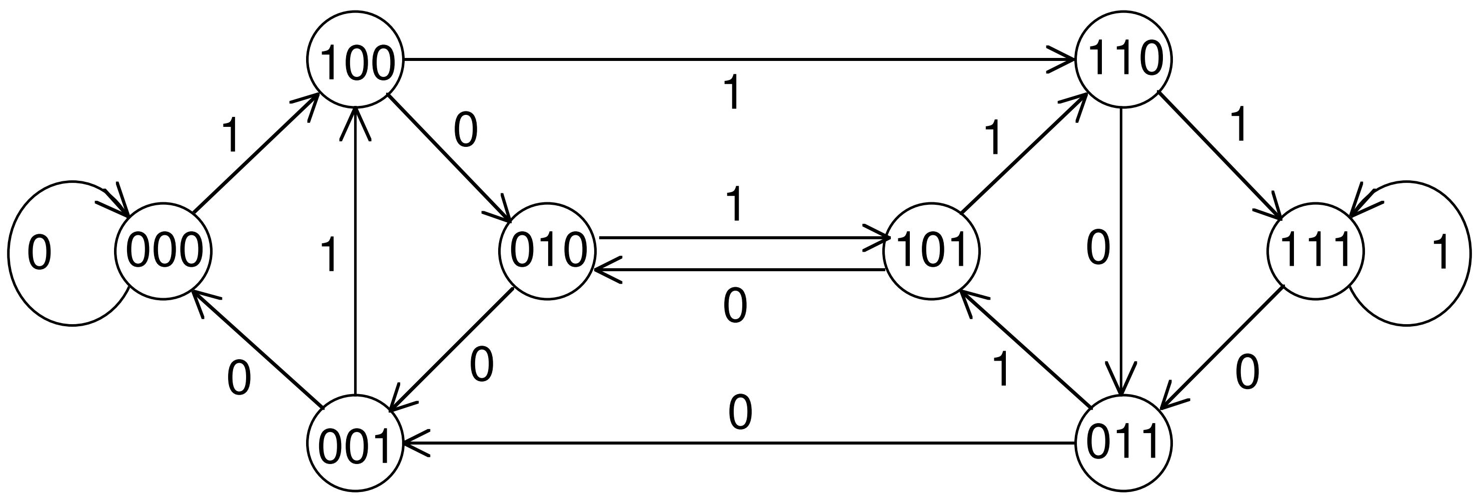
- **Combination lock from introduction to course**



Can any sequential system be represented with a state diagram?

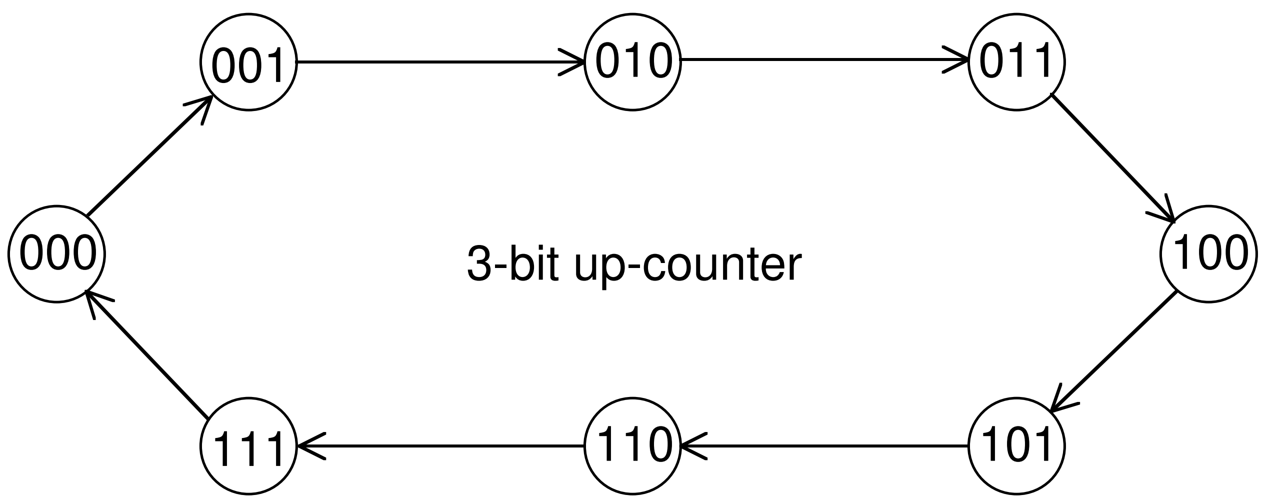
- **Shift register**
  - input value shown on transition arcs
  - output values shown within state bubble (or node)





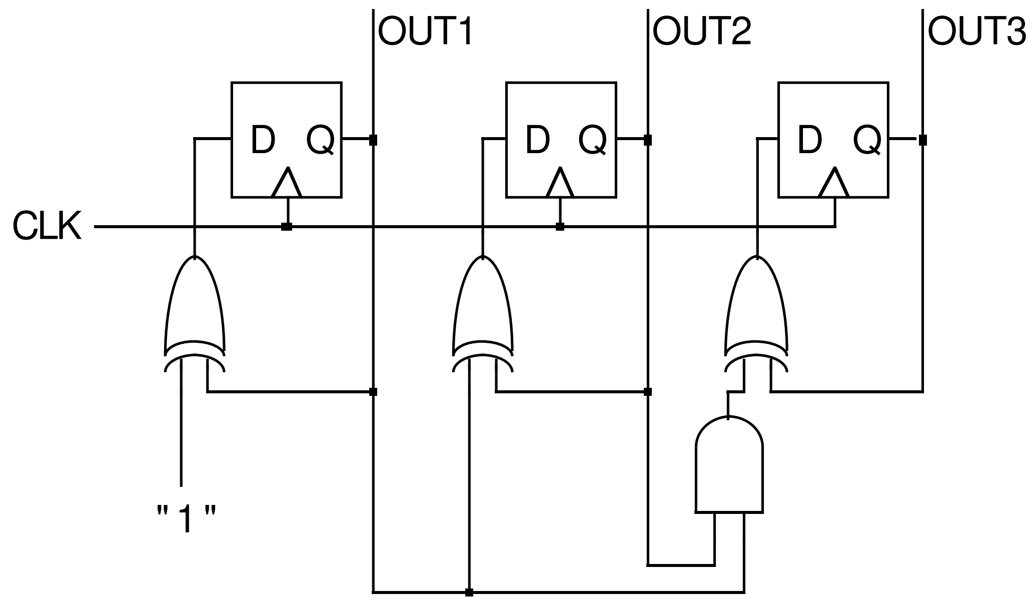
Counters are simple finite state machines

- **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, ...



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



FSM design procedure

- **Start with counters**
  - simple because output is just state
  - simple because no choice of next state based on input

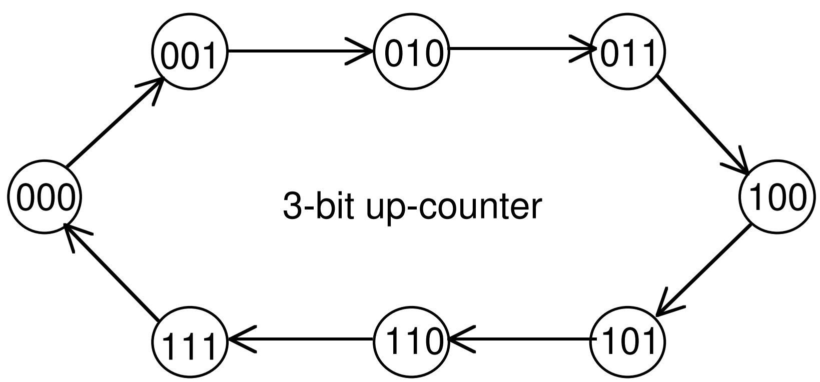
- **State diagram to state transition table**
  - tabular form of state diagram
  - like a truth-table

- **State encoding**
  - decide on representation of states
  - for counters its 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

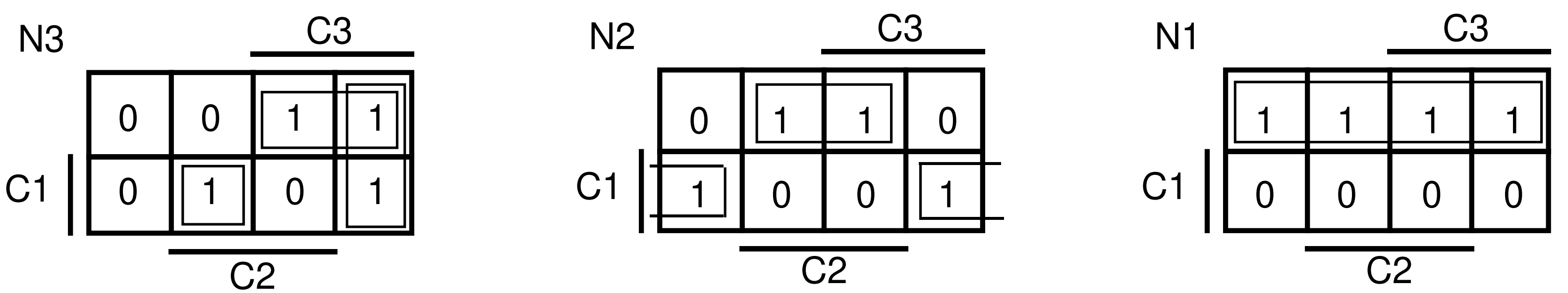
Implementation

- **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

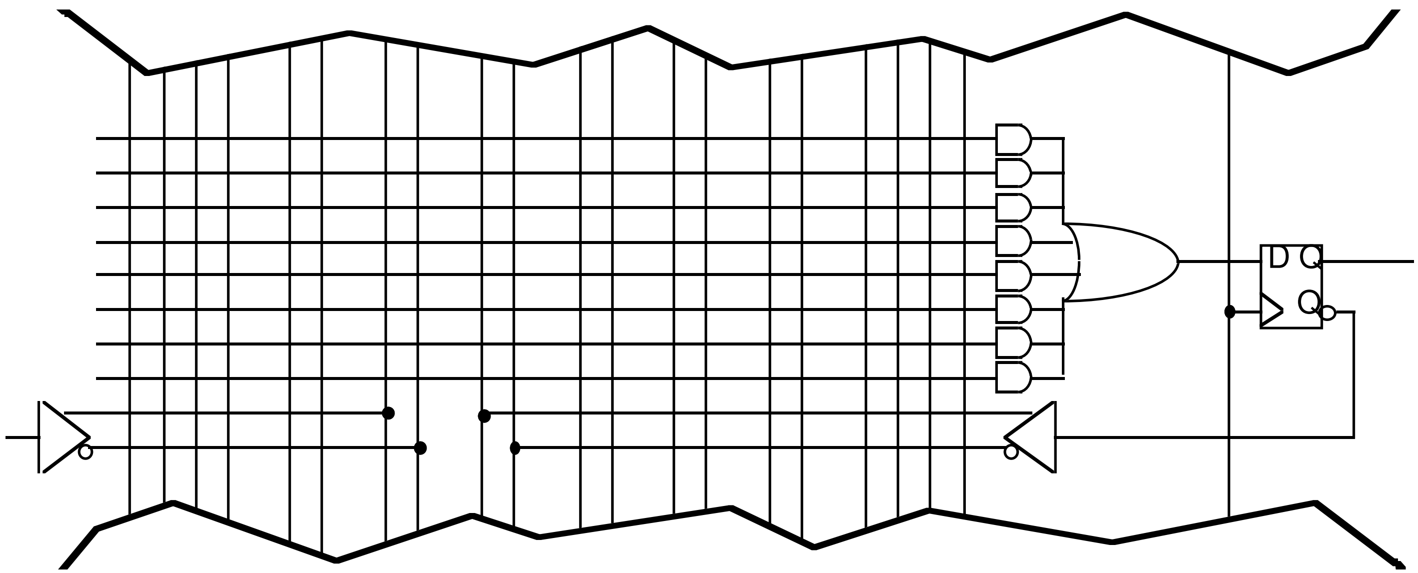
notation to show function represent input to D-FF

N1 := C1'
N2 := C1C2' + C1'C2
:= C1 xor C2
N3 := C1C2C3' + C1'C3 + C2'C3
:= C1C2C3' + (C1' + C2')C3
:= (C1C2) xor C3



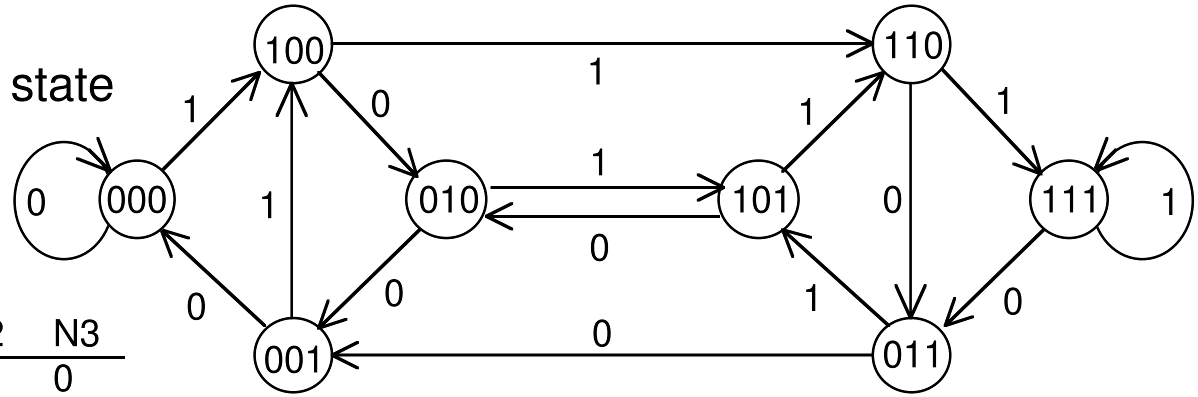
Implementation (cont'd)

- **Programmable logic building block for sequential logic**
  - macro-cell = DFF + two-level logic capability like PAL (e.g., 8 product terms)



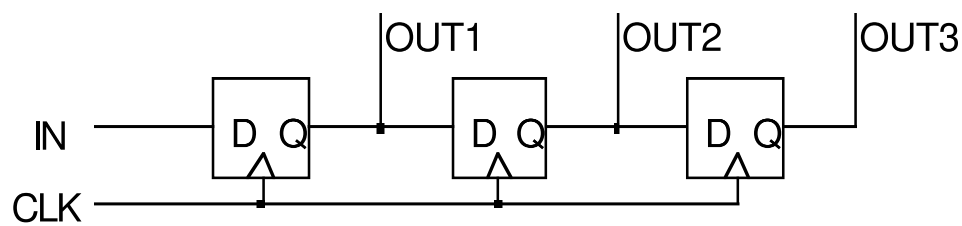
Another example

- **Shift register**
  - input determines next state



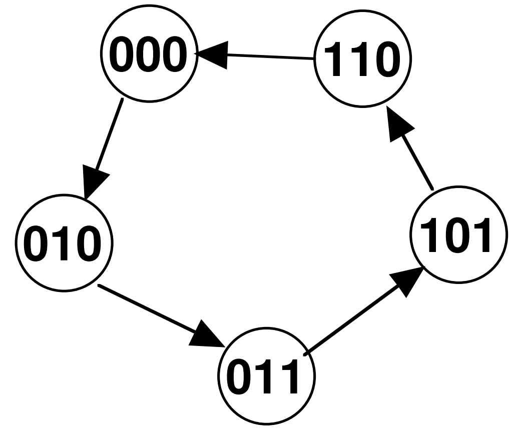
In	C1	C2	C3	N1	N2	N3
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	0	0	1
0	0	1	1	0	0	1
0	1	0	0	0	1	0
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	0	1	1
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	1	0	1
1	0	1	1	1	0	1
1	1	0	0	1	1	0
1	1	0	1	1	1	0
1	1	1	0	1	1	1
1	1	1	1	1	1	1

N1 := In
N2 := C1
N3 := C2



More complex counter example

- **Complex counter**
  - repeats 5 states in sequence
  - not a binary number representation



- **Step 1: derive the state transition diagram**
  - count sequence: 000, 010, 011, 101, 110

- **Step 2: derive the state transition table from the state transition diagram**

Note: don't care conditions arise from unused states

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

More complex counter example (cont'd)

- **Step 3: K-maps for next state functions**

C+

			C	
	0	0	0	X
A	X	1	X	1
		B		

B+

			C	
	1	1	0	X
A	X	0	X	1
		B		

A+

			C	
	0	1	0	X
A	X	1	X	0
		B		

C+ := A

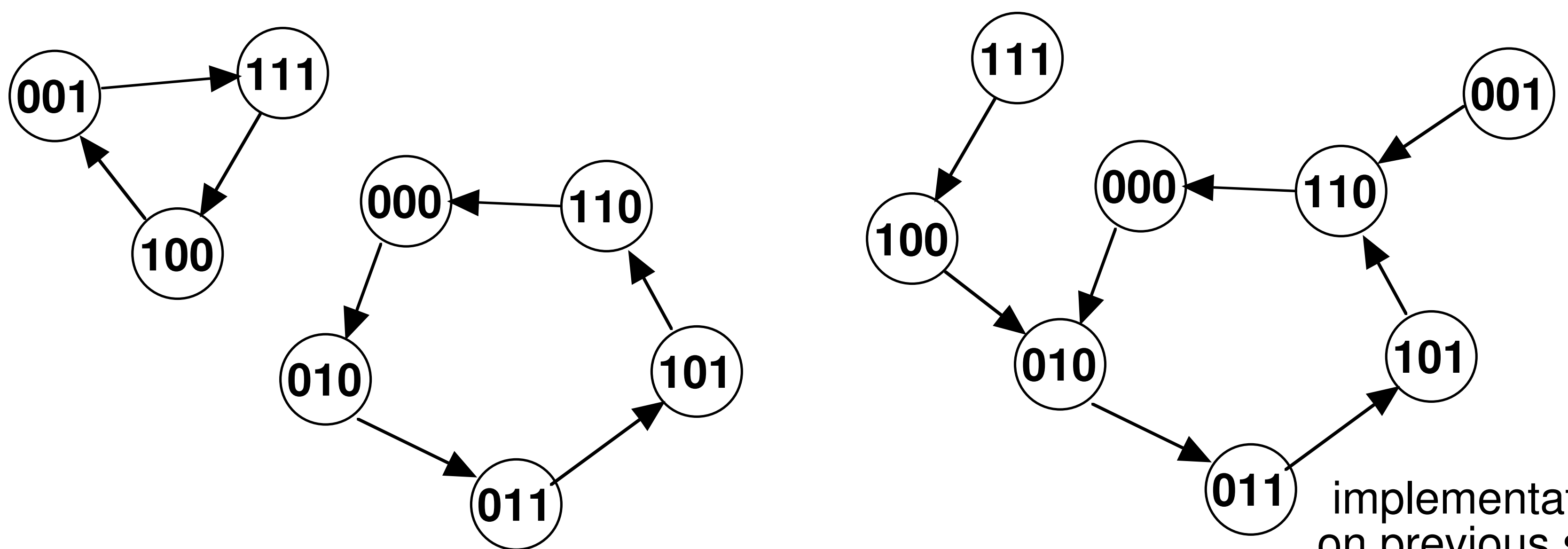
B+ := B' + A'C'

A+ := BC'

Self-starting counters

- **Start-up states**
  - at power-up, counter may be in an unused or invalid state
  - designer must guarantee that it (eventually) enters a valid state
  - especially a problem for counters that validly use a subset of states

- **Self-starting solution**
  - design counter so that all invalid states eventually transition to a valid state
  - may limit exploitation of don't cares



implementation on previous slide

Self-starting counters (cont'd)

- **Deriving state transition table from don't care assignment**

C+

	0	0	0	0
A	1	1	1	1

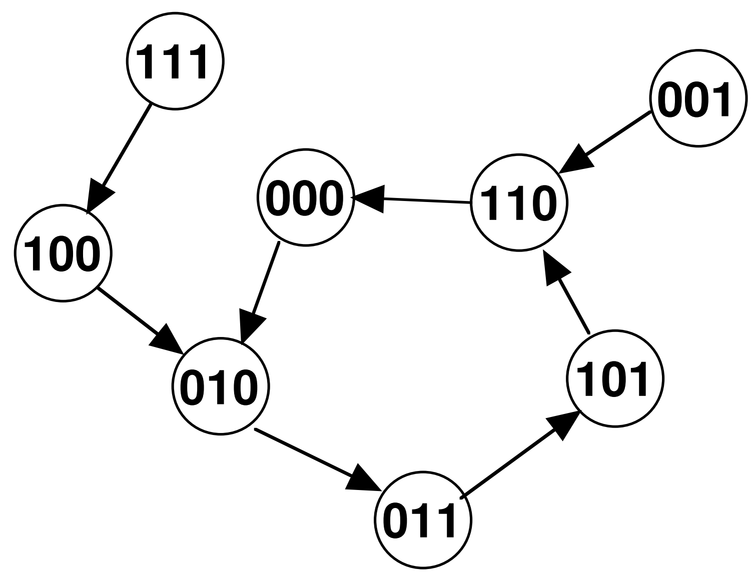
B+

	1	1	0	1
A	1	0	0	1

A+

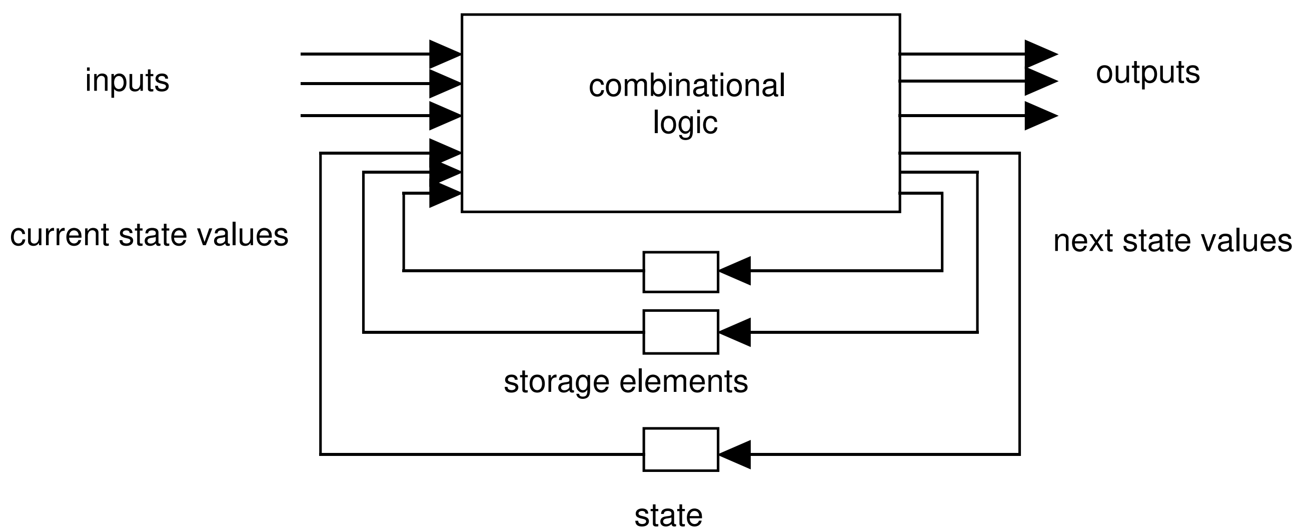
	0	1	0	0
A	0	1	0	0

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

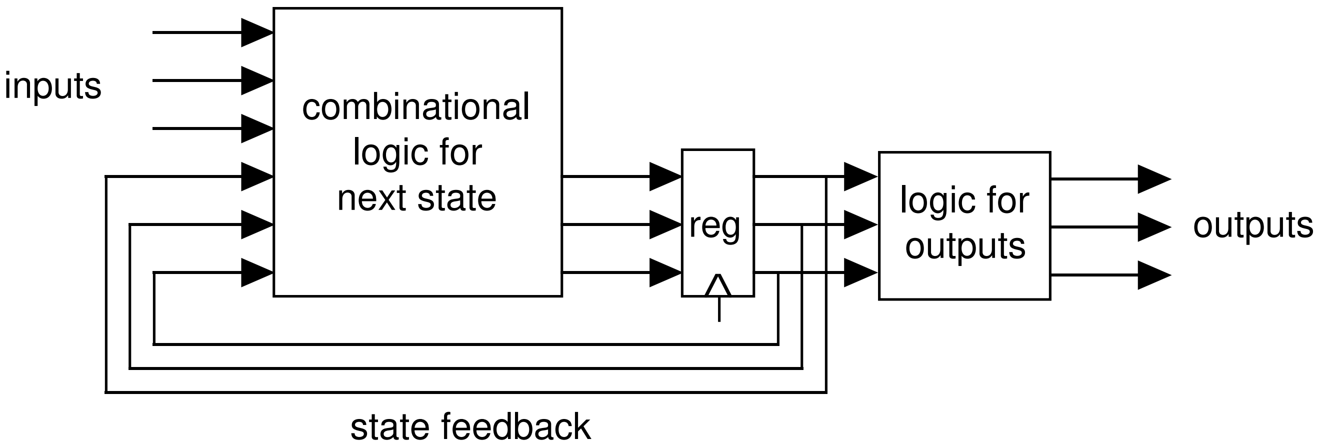


State machine model

- **Circuit state represented by values of state variables (state vector)**
- **Combinational logic computes:**
  - next state (values for next state variables)
    - as function of current state and inputs
  - outputs
    - as function of current state and inputs (Mealy machine)
    - as function of current state only (Moore machine)



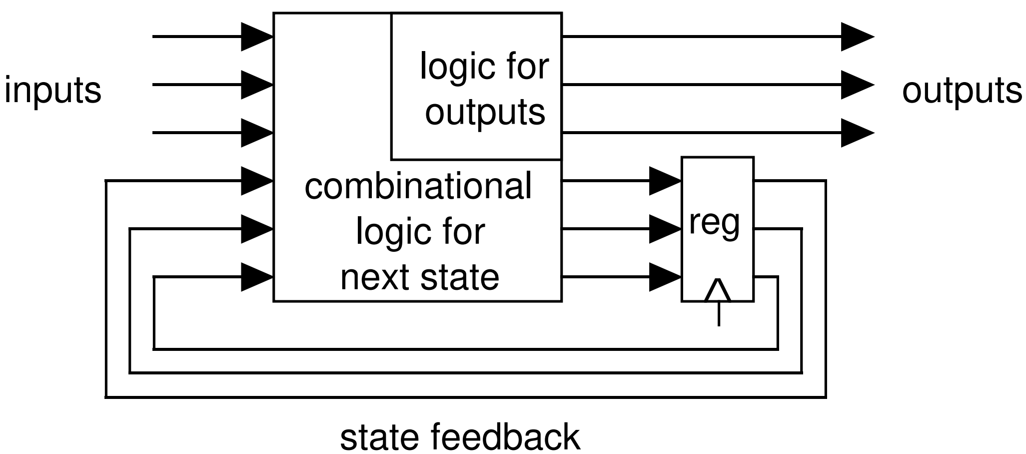
Moore and Mealy machines



Moore machine

outputs are solely a function of the current state

outputs change synchronously with state changes after some propagation delay



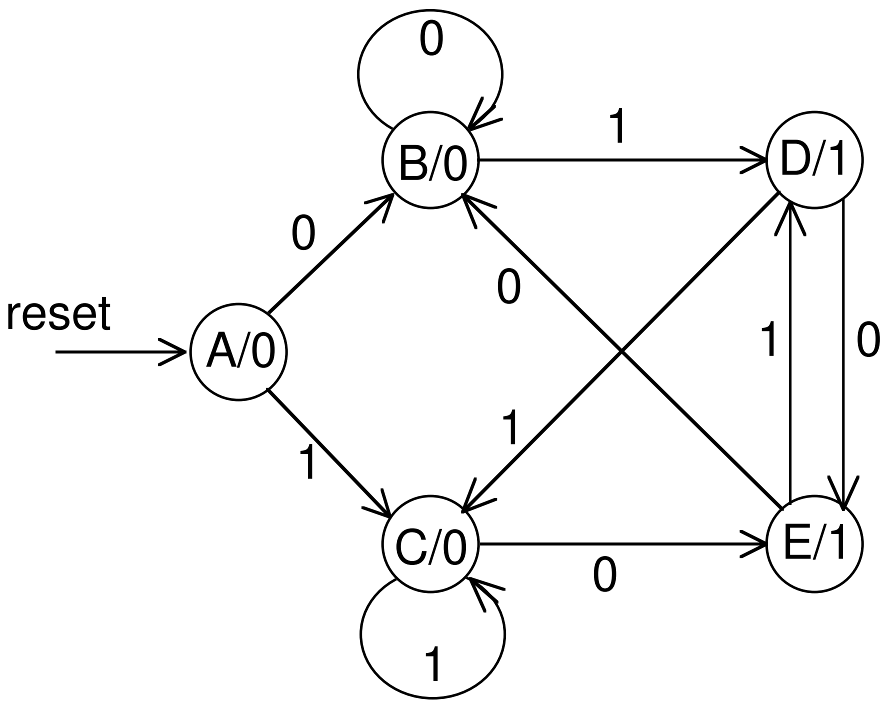
Mealy machine

outputs depend on state AND inputs

input changes cause immediate output changes after propagating through logic – asynchronous signals

Specifying outputs for a Moore machine

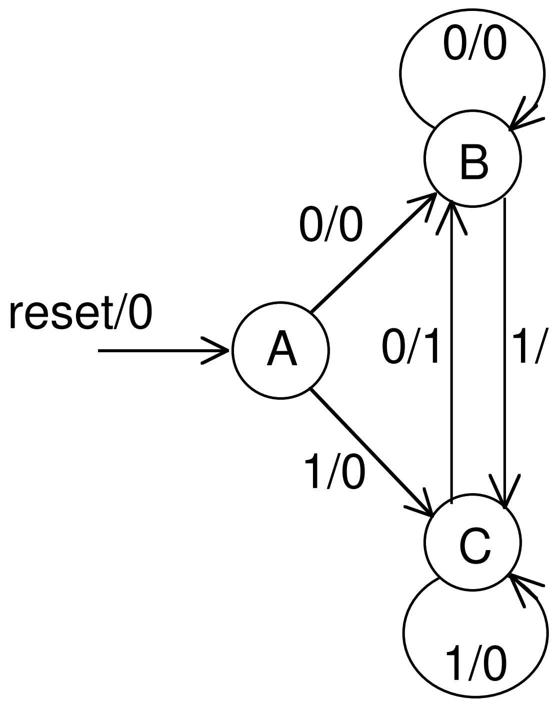
- **Output is only function of state**
  - specify in state bubble in state diagram
  - example: sequence detector for 01 or 10



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

Specifying outputs for a Mealy machine

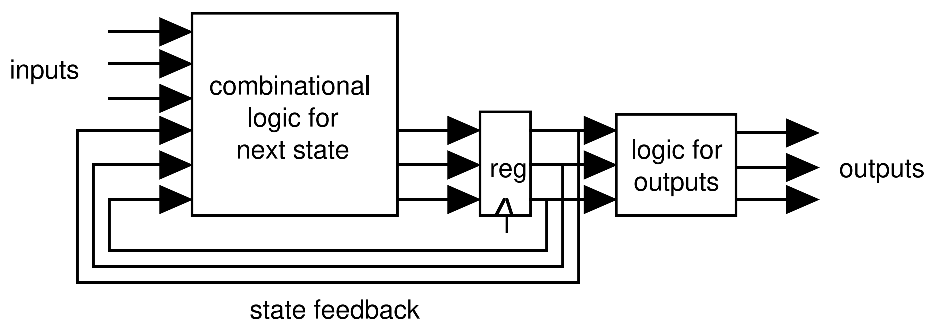
- **Output is function of state and inputs**
  - specify output on transition arc between states
  - example: sequence detector for 01 or 10

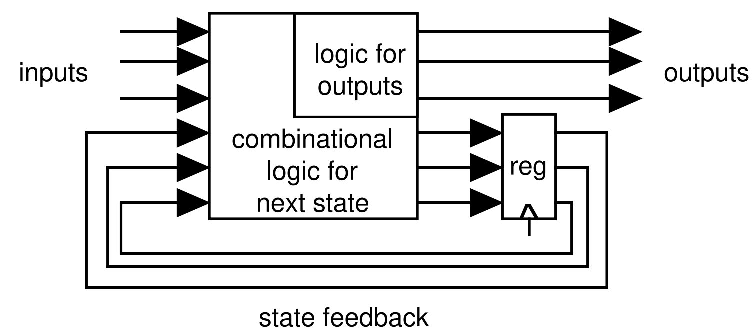


reset	input	current state	next state	output
1	–	–	A	0
0	0	A	B	0
0	1	A	C	0
0	0	B	B	0
0	1	B	C	1
0	0	C	B	1
0	1	C	C	0

Comparison of Mealy and Moore machines

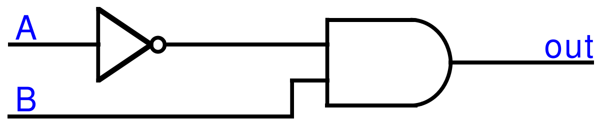
- **Mealy machines tend to have less states**
  - different outputs on arcs (n^2) rather than states (n)
- **Moore machines are safer to use**
  - outputs change at clock edge (always one cycle later)
  - in Mealy machines, input change can cause output change as soon as logic is done – a big problem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

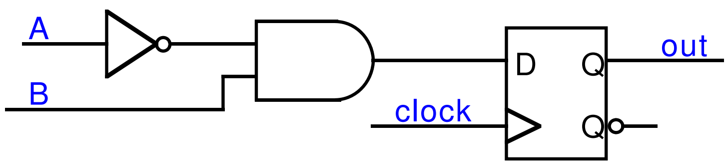


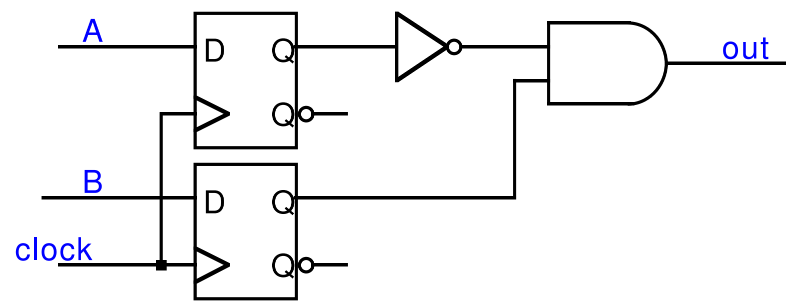


Mealy and Moore examples

- **Recognize A,B = 0,1**
  - Mealy or Moore?

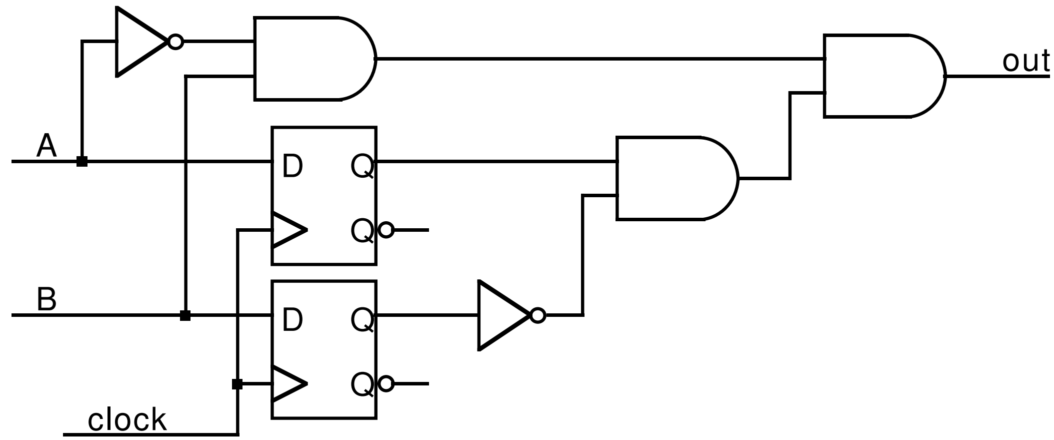


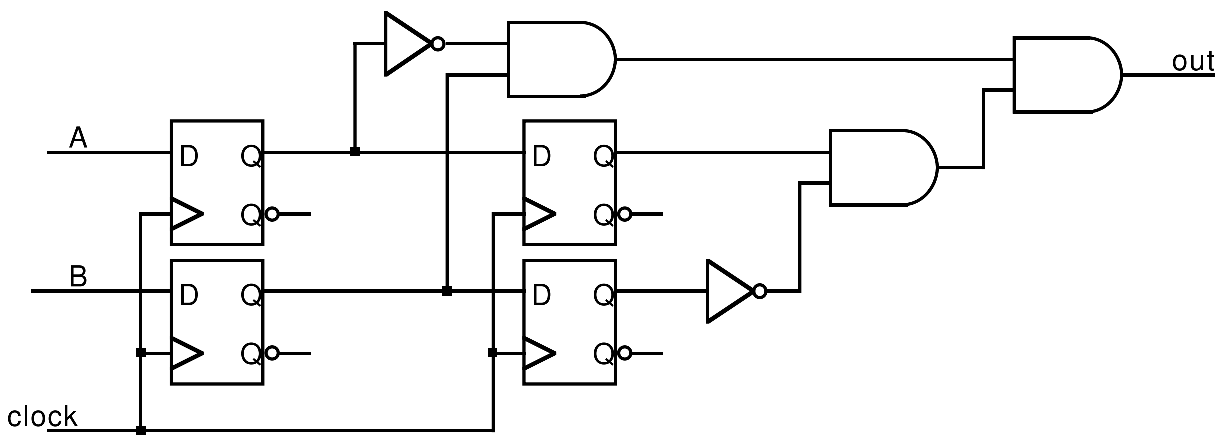




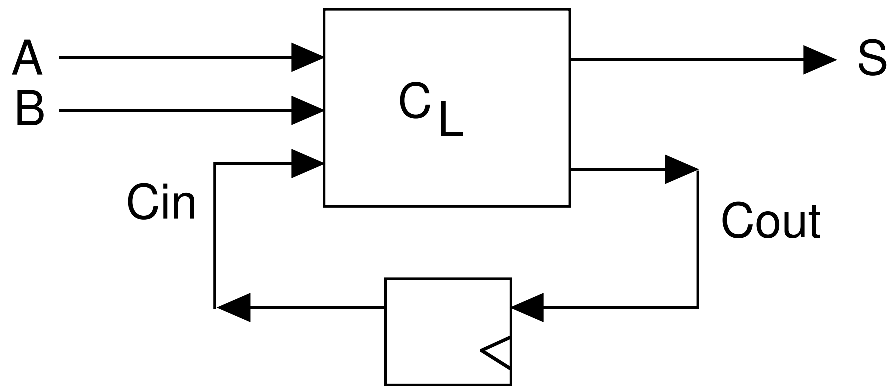
Mealy and Moore examples (cont'd)

- **Recognize A,B = 1,0 then 0,1**
  - Mealy or Moore?

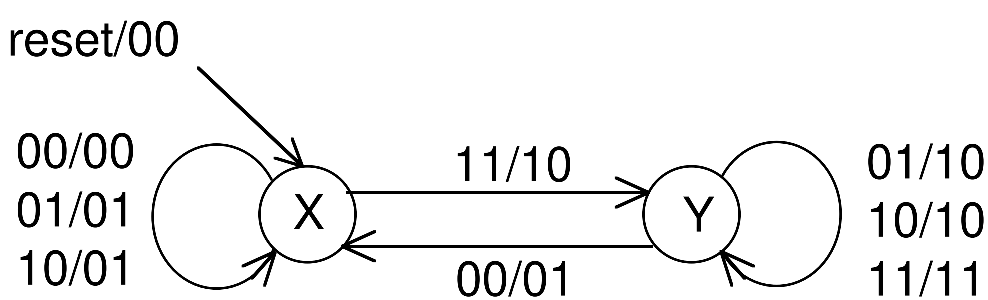




Example: serial adder



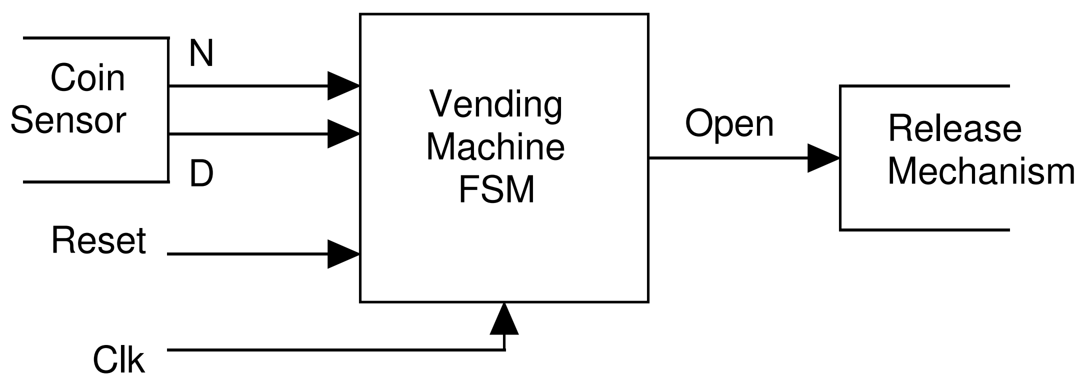
		P.S.	N.S.	
A	B	Cin	Cout	S
0	0	X:0	X:0	0
0	0	Y:1	X:0	1
0	1	X:0	X:0	1
0	1	Y:1	Y:1	0
1	0	X:0	X:0	1
1	0	Y:1	Y:1	0
1	1	X:0	Y:1	0
1	1	Y:1	Y:1	1



X: carry is 0
Y: carry is 1

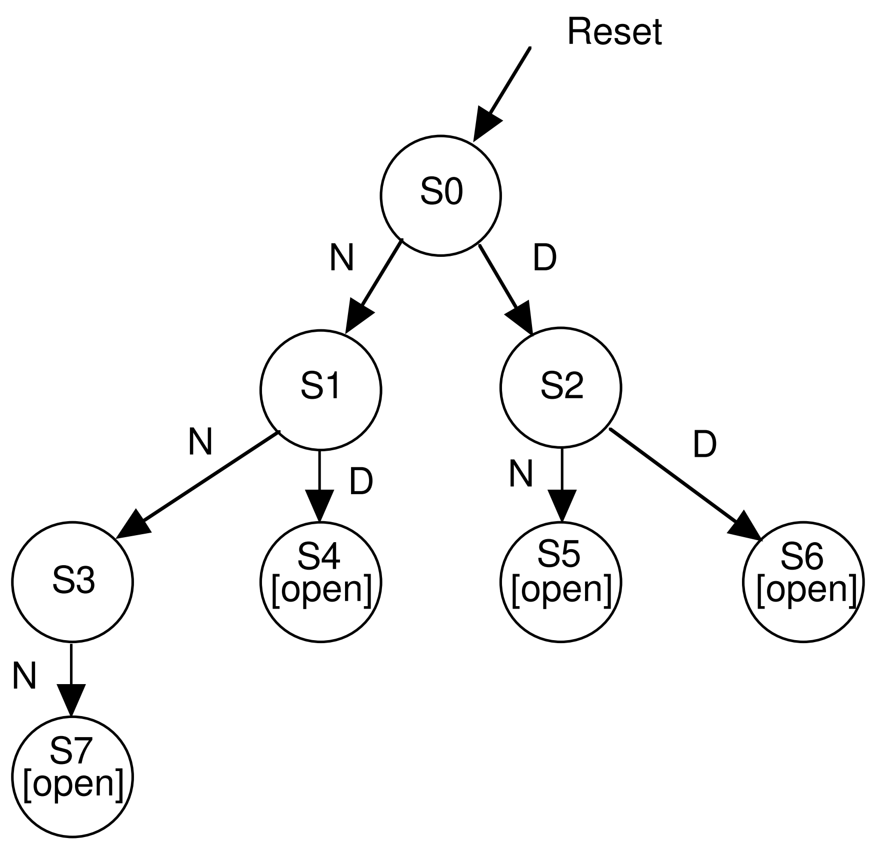
Example: vending machine

- Release item after 15 cents are deposited (its an old example)
- Single coin slot for dimes and nickels (coin sensor figures out which)
- Machine does not give change



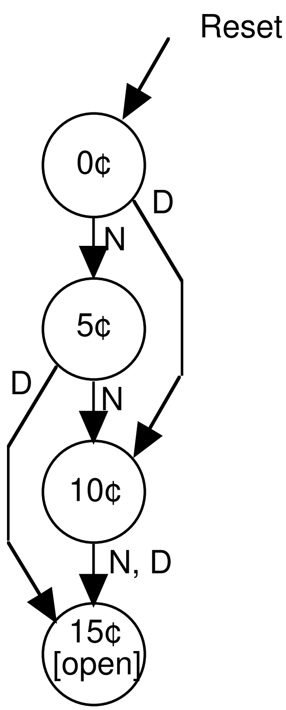
Example: vending machine (state diagram)

- **Suitable abstract representation**
  - tabulate typical input sequences
    - 3 nickels
    - nickel, dime
    - dime, nickel
    - two dimes
- **Draw state diagram**
  - inputs: N, D, reset
  - output: open



Example: vending machine (symbolic state transition table)

- **Minimize number of states**
  - reuse states whenever possible
  - many have the same meaning
- **Derive symbolic state transition table**



present state	inputs D	inputs 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

Example: vending machine (state encoding)

- **Uniquely encode states**
  – 4 states require a minimum of 2 bits (log number of states)
  – 4 staets require a maximum of 4 (number of states)

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

Example: vending machine (two-level logic implementation)

K-map for D1

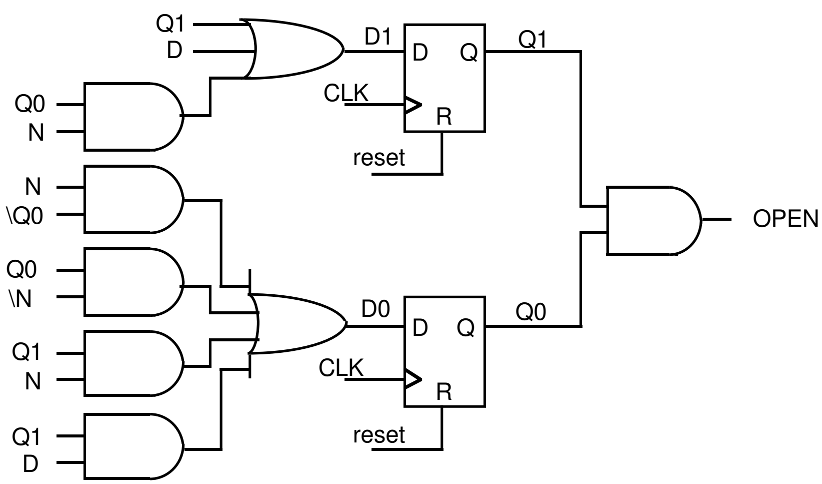
DN \ Q1Q0	00	01	11	10
00	0	0	1	1
01	0	1	1	1
11	X	X	X	X
10	1	1	1	1

K-map for D0

DN \ Q1Q0	00	01	11	10
00	0	1	1	0
01	1	0	1	1
11	X	X	X	X
10	0	1	1	1

K-map for Open

DN \ Q1Q0	00	01	11	10
00	0	0	1	0
01	0	0	1	0
11	X	X	X	X
10	0	0	1	0



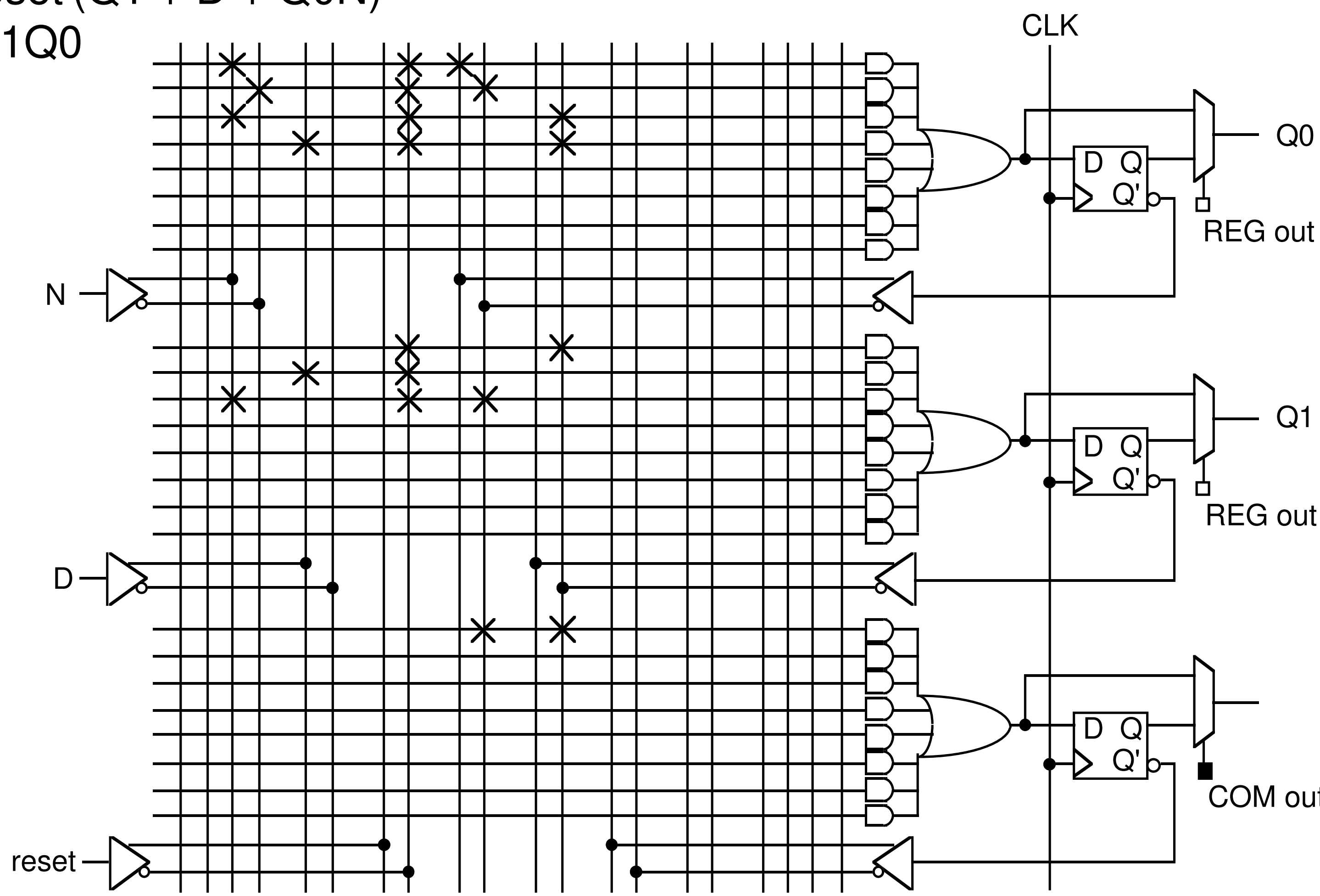
D1 = Q1 + D + Q0N
D0 = Q0'N + Q0N' + Q1N + Q1D
OPEN = Q1Q0

if no FFs do not have a reset pin then

D1 = reset'(Q1 + D + Q0N)
D0 = reset'(Q0'N + Q0N' + Q1N + Q1D)

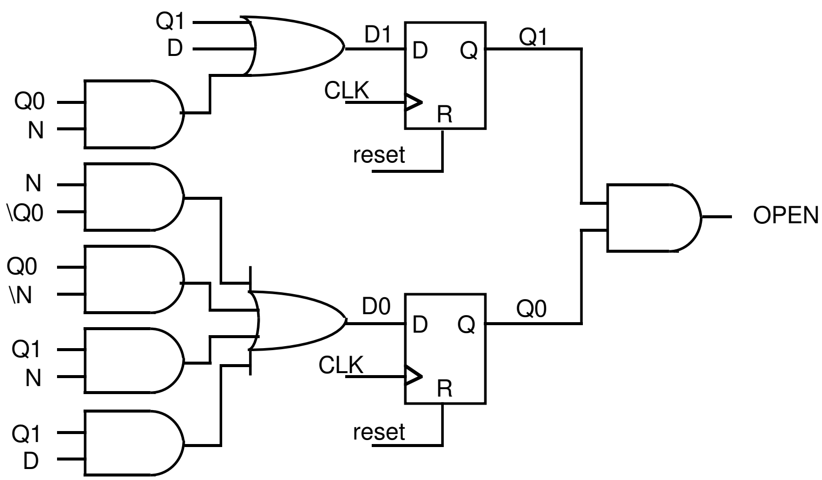
Example: vending machine (PLD mapping)

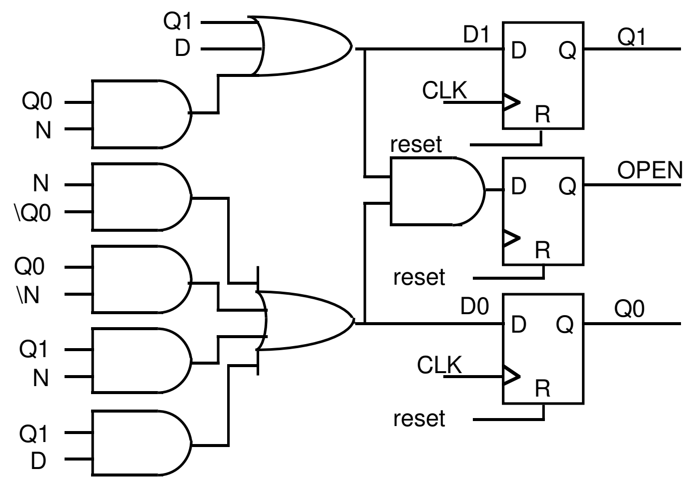
D0 = reset'(Q0'N + Q0N' + Q1N + Q1D)
D1 = reset'(Q1 + D + Q0N)
OPEN = Q1Q0



Example: vending machine (retiming)

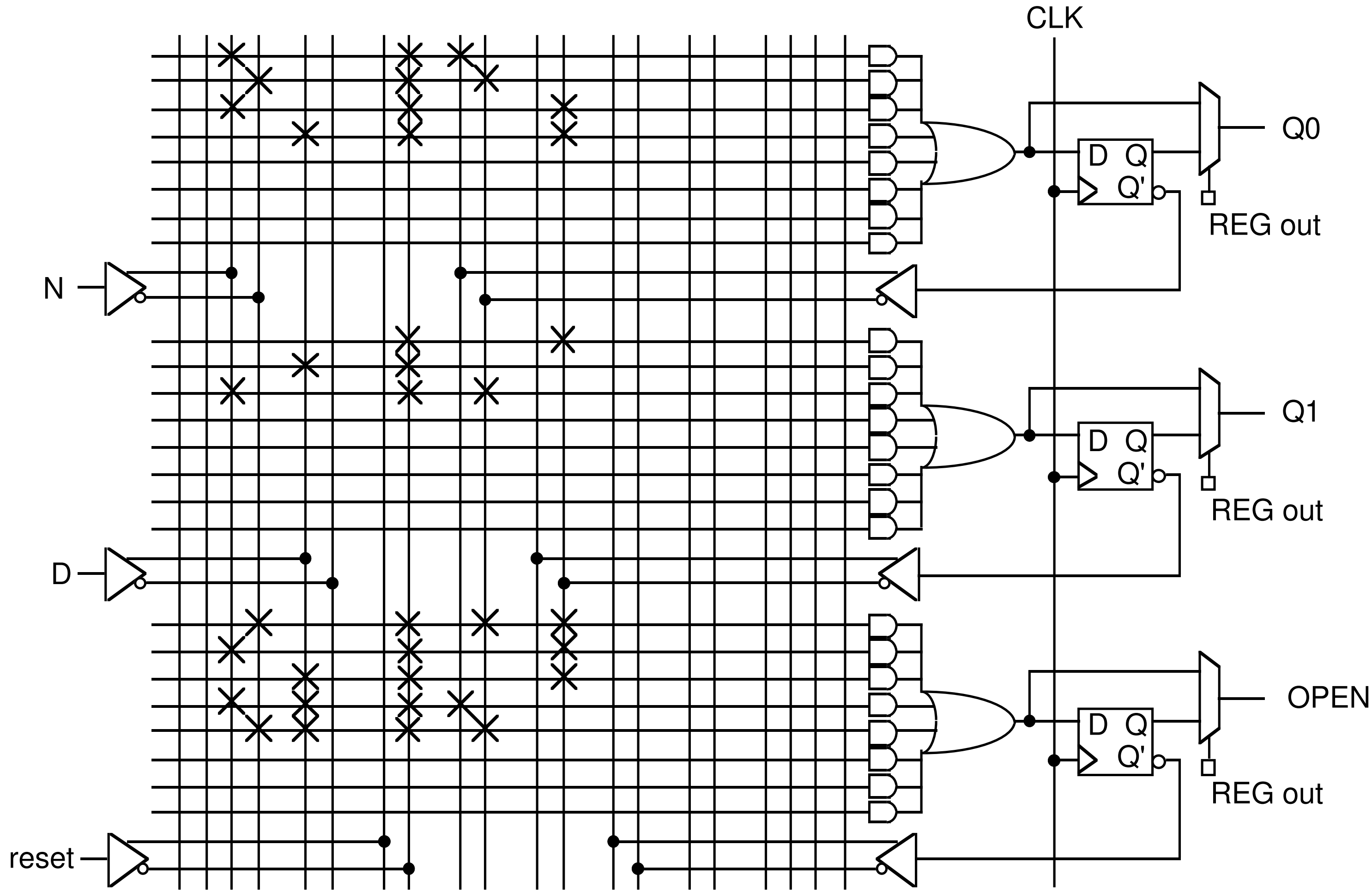
- **OPEN = Q1Q0**
  - combinational delay after Q1 and Q0 change
- **This can be corrected by retiming**
  - move flip-flops and logic through each other to improve delay
- **OPEN = reset'(Q1 + D + Q0N)(Q0'N + Q0N' + Q1N + Q1D)**
  = reset'(Q1Q0N' + Q1N + Q1D + Q0'ND + Q0N'D)



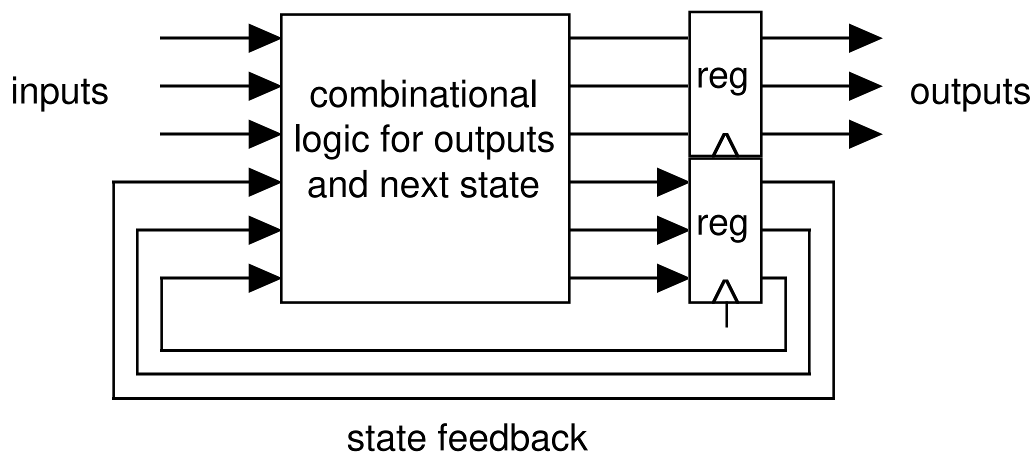


Example: vending machine (alternate PLD mapping)

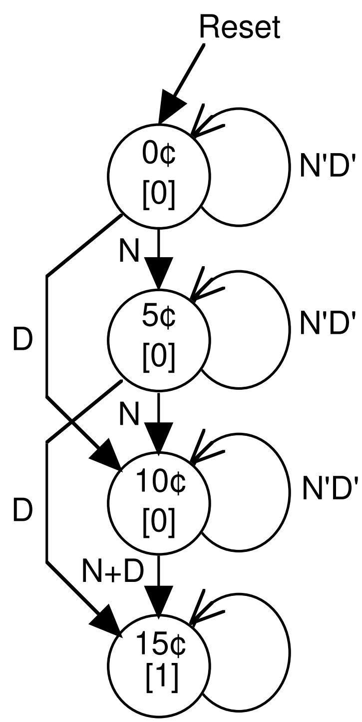
OPEN = reset'(Q1Q0N' + Q1N + Q1D + Q0'ND + Q0N'D)



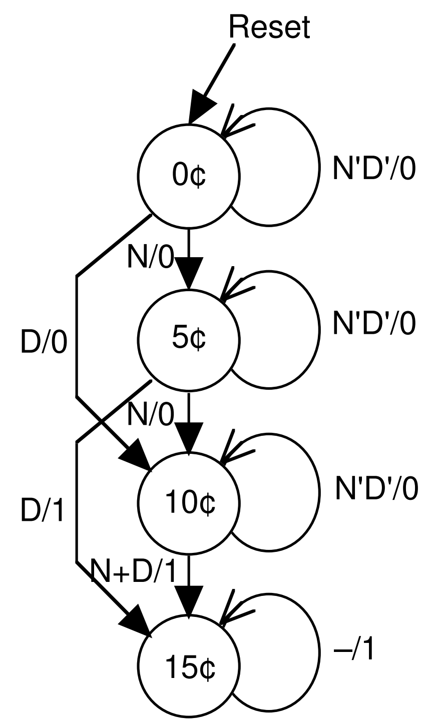
- **Synchronous (or registered) Mealy machine**
  - Moore machine with no output decoding
  - best of both worlds
    - fewer states advantage of Mealy machine
    - no race conditions between communication machines advantage of Moore machine
  - added benefit – no glitches on outputs
    - due to combinational logic decoding states for outputs



Equivalent Mealy and Moore state diagrams for vending machine



Moore machine



Mealy machine

Note: N and D can not be asserted simultaneously

General FSM design procedure

(1) Determine inputs and outputs

(2) Determine possible states of machine

– state minimization

(3) Encode states and outputs into a binary code

– state assignment or state encoding
– output encoding
– possibly input encoding (if under our control)

(4) Realize logic to implement functions for states and outputs

– combinational logic implementation and optimization
– choices made in steps 2 and 3 can have large effect on resulting logic

Section review

- **Models for representing sequential circuits**
  - abstraction of sequential elements
  - finite state machines and their state diagrams
  - inputs/outputs
  - Mealy, Moore, and synchronous Mealy machines
- **Finite state machine design procedure**
  - deriving state diagram
  - deriving state transition table
  - determining next state and output functions
  - implementing combinational logic
- **Implementation of sequential logic**
  - support in programmable logic devices
  - specification language support