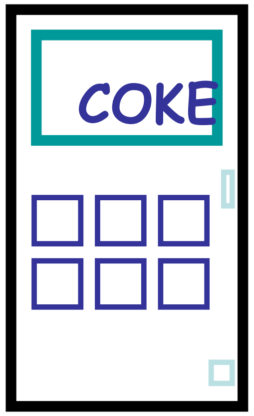
DEPARTMENT OF
INFORMATION SYSTEMS
AND COMPUTER SCIENCE

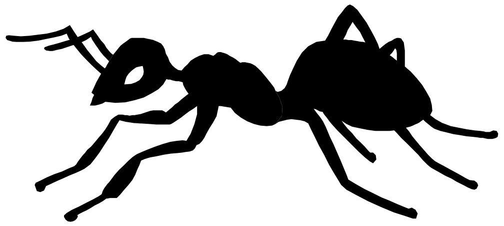
Finite State Machines

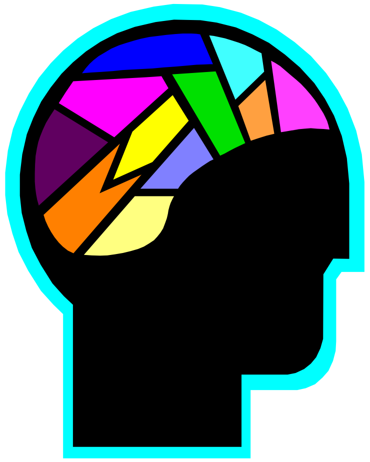
Theory, Design, Implementation, and Optimization

State Machines

- Everything in the world is a state machine:
  - has state
  - has observable outputs
  - has inputs that affect state and outputs
- Computers are *programmable* state machines.
  - They can perform a desired computational task.







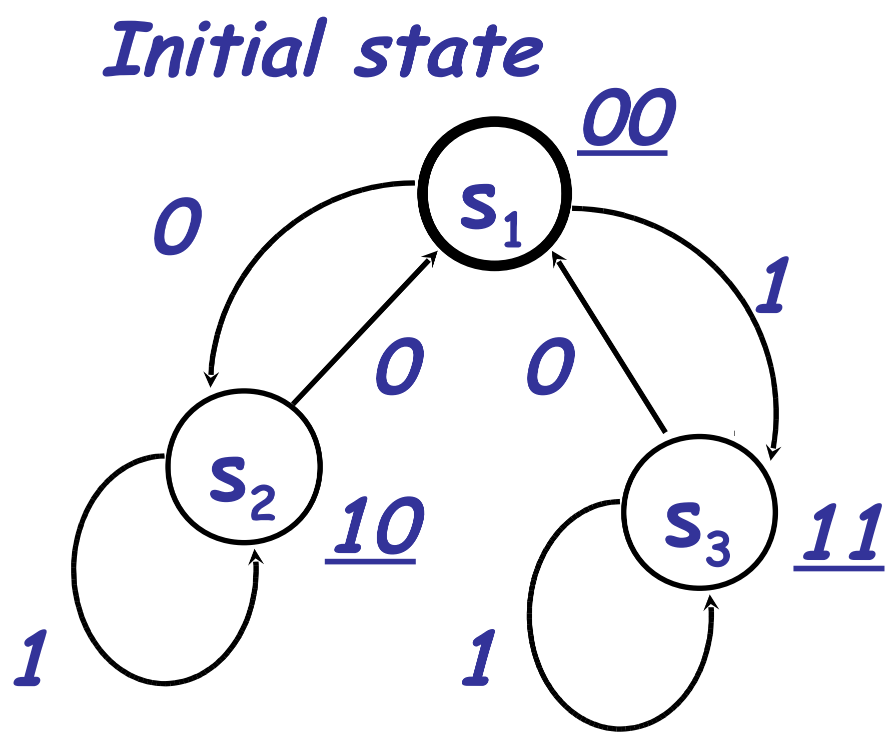
A Little Theory

- A Finite State Machine (FSM) has:
  - K states, $S = \{s_1, s_2, \ldots, s_K\}$, initial state s_{init}
  - N inputs, $I = \{i_1, i_2, \ldots, i_N\}$
  - M outputs, $O = \{o_1, o_2, \ldots, o_M\}$
  - Transition function $T(S, I)$ mapping each current state and input to a next state.
  - Output function $O(S)$ mapping each current state to an output.
- Given a sequence of inputs, the FSM produces a sequence of outputs which is dependent on s_{init}, $T(S, I)$ and $O(S)$.

FSM Representations

State Transition Graph

- states -> circles
- transitions -> arcs
- outputs -> underlined



	t	*t+1*	*t+2*
Inputs:	*0*	*1*	*0*
Outputs:	*00*	____	____

State Transition Table

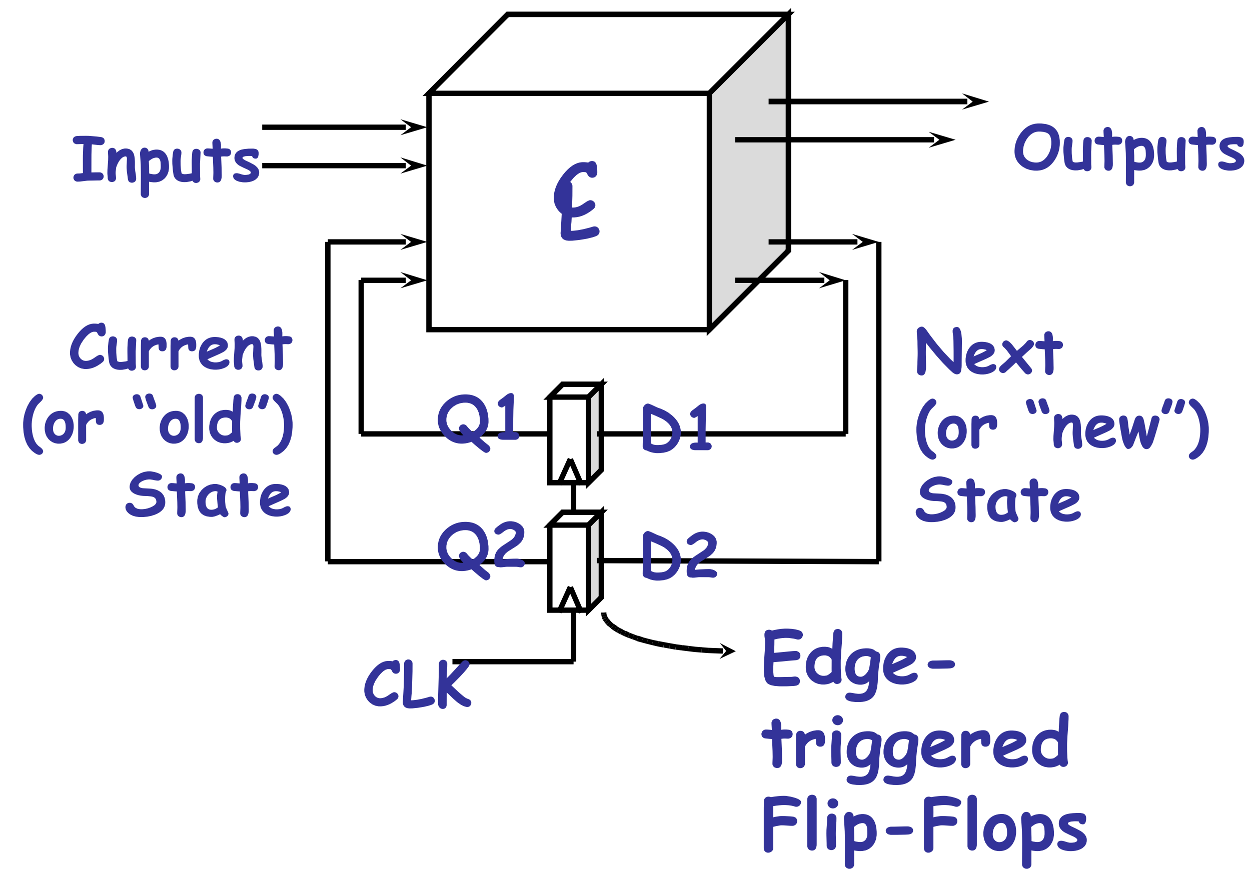
s_{init}

cur state S	cur input I	next state T(S, I)
s_1	0	s_2
s_1	1	s_3
s_2	0	s_1
s_2	1	s_2
s_3	0	s_1
s_3	1	s_3

cur state S	cur output O(S)
s_1	00
s_2	10
s_3	11

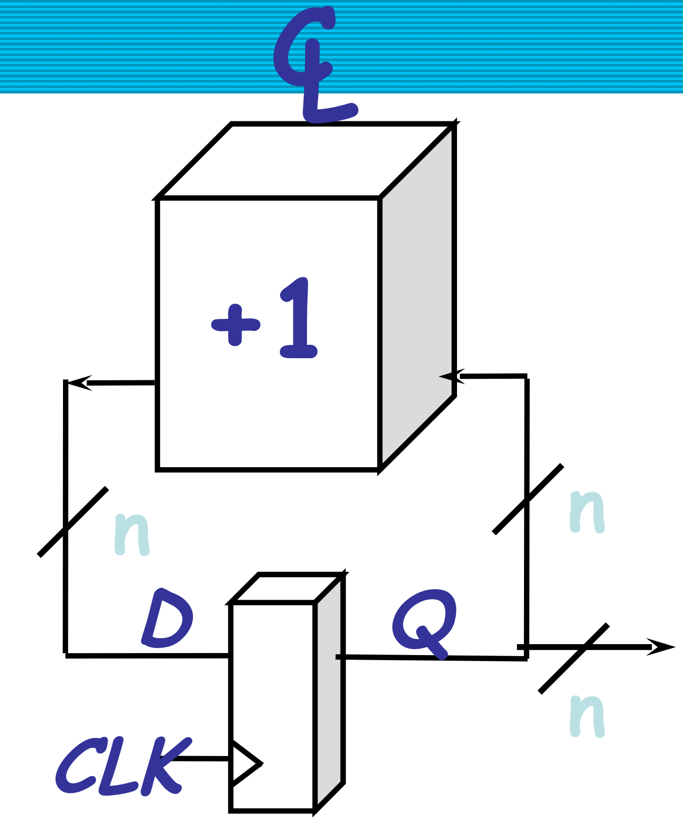
Implementation: Synchronous FSM

- State is stored in FF's.
- Current State and Current Inputs produce Next State and Current Output.
- Machine changes state on clock tick:
  - At clk edge, D gets copied to Q so next state becomes current state.
  - Current outputs and next state are recomputed based on current inputs.
  - No combinational cycles.
- CL can be a ROM:
  - Transition and Output Tables can be placed in a truth table.
  - Size depends on # of states.

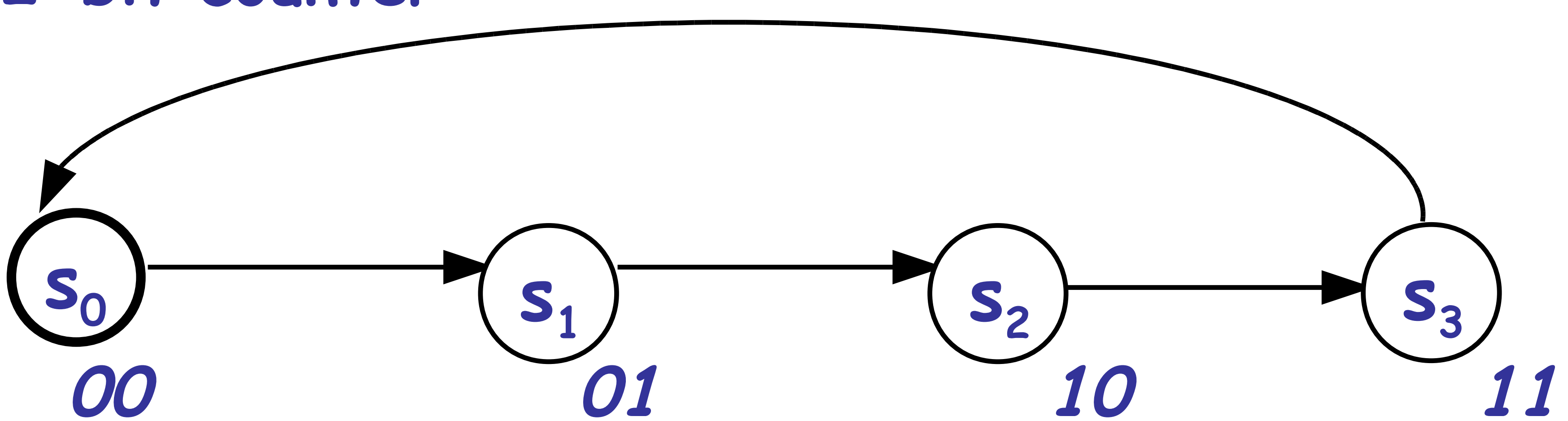


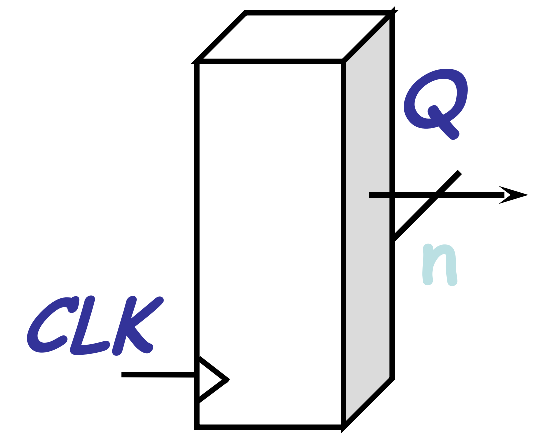
Simple Example: n-bit Counter

- States: 2^n states (one for each no.)
  - Encoded as n-bit binary no., stored in reg's Q.
- Inputs: none, Outputs: n bits
- T(S) = Q+1
  - Implemented as a +1 combinational circuit.
- O(S) = Q
  - Output simply taken from register output Q.



A 2-bit counter

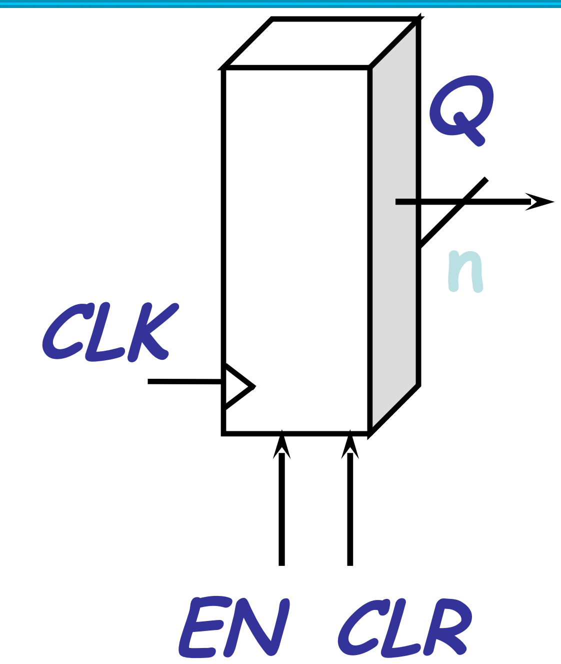


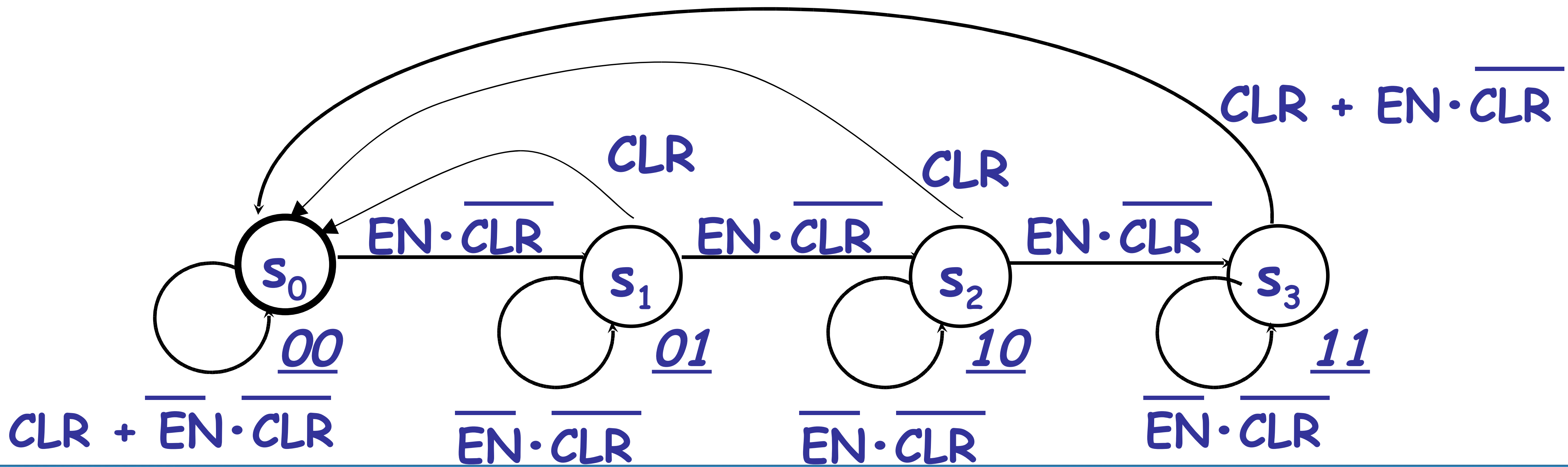


DISCS

n-bit Counter w/ EN and CLR

- States and O(S) same as before.
- Inputs: 2 bits: EN and CLR
- T(S,I) = 0 if CLR
  Q+1 if EN • !(CLR)
  Q, otherwise (!(EN + CLR))
- Mark arcs with condition for arc.
  - *For each state, all possible input combinations must be covered!*

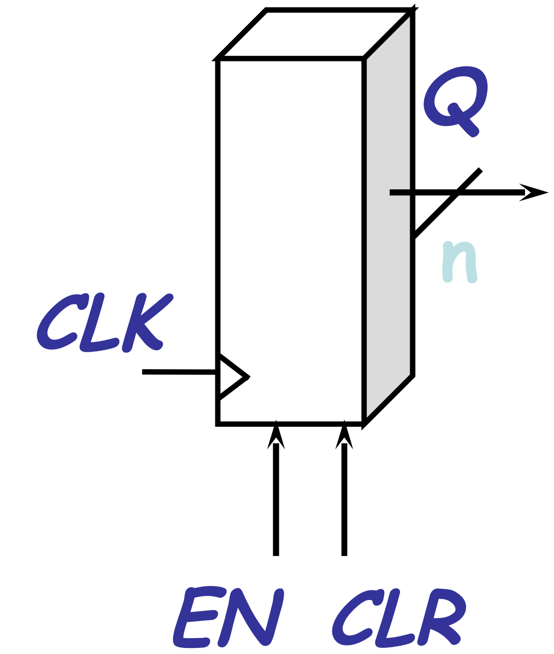




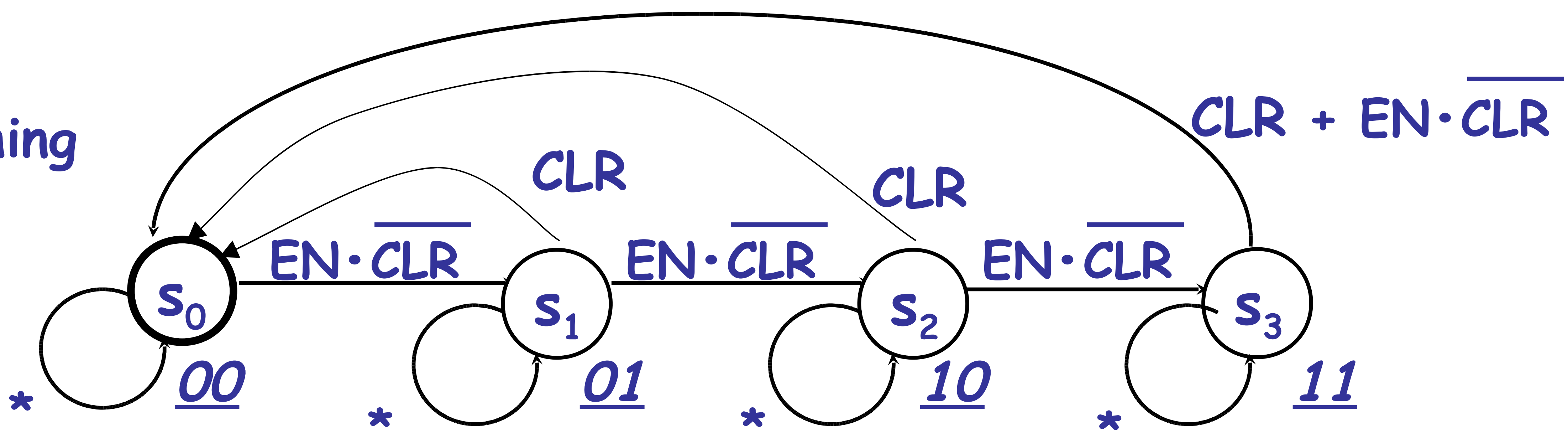
n-bit Counter w/ EN and CLR

IMPORTANT: You also need to make sure that for each state, there is no input that satisfies more than one transition!

What if you removed !CLR from the EN(!CLR) transition in s1?

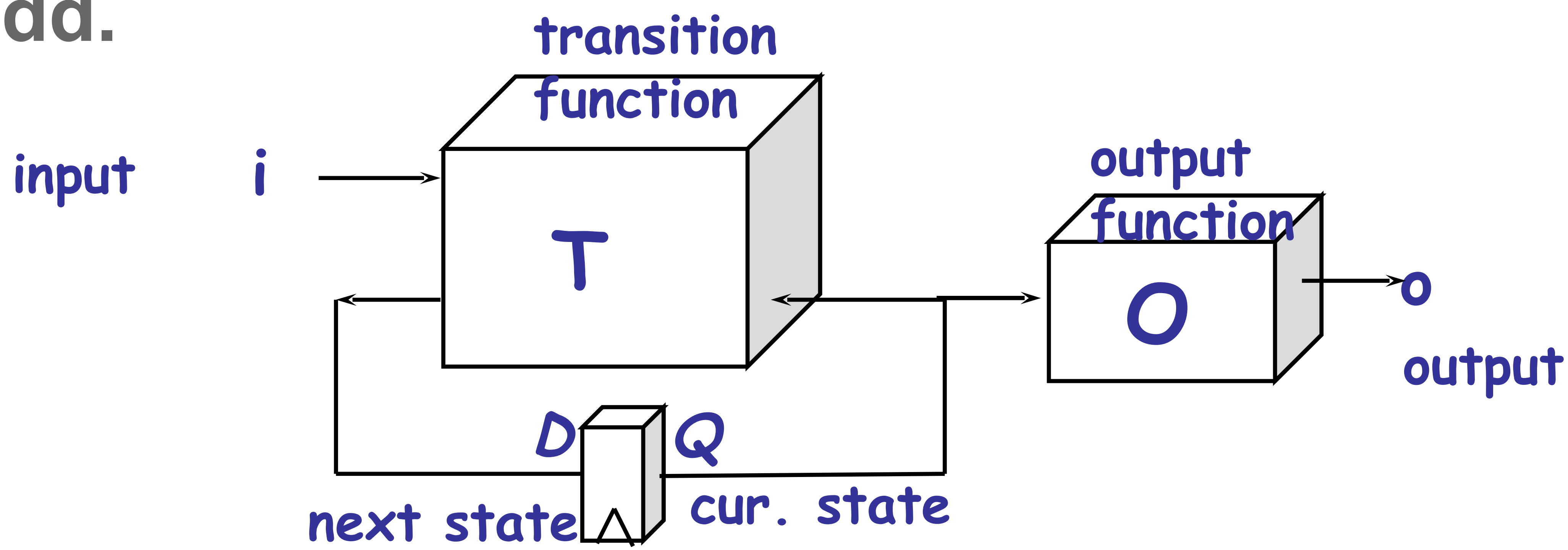


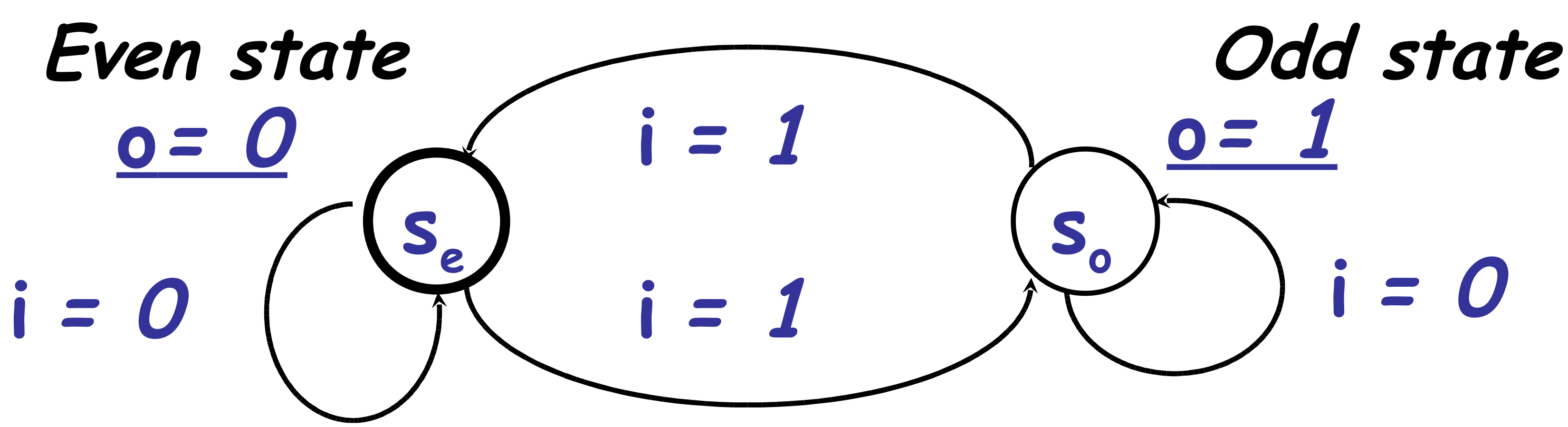
* means "everything else"



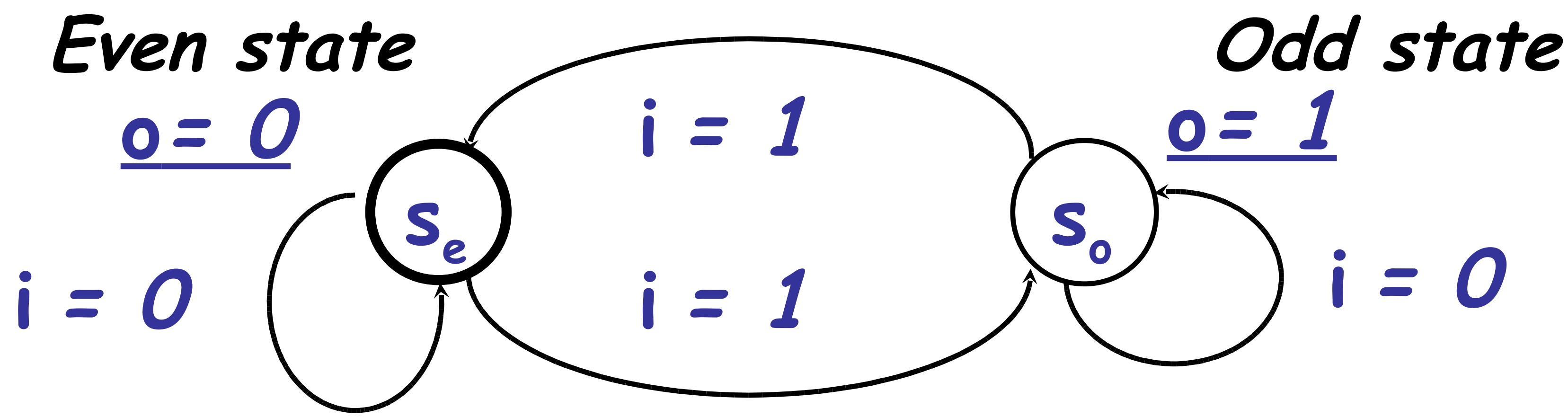
Another Example: Parity Machine

- Design an FSM that outputs a 1 if and only if the number of 1's in the input sequence is odd.





State Encoding



- **State Encoding: Choose a <u>unique</u> binary code for each s_i so the combinational logic (implementation) can be specified.**
  - **Can choose $s_e = 0$ and $s_o = 1$**
  - **Or, can also choose $s_e = 1$ and $s_o = 0$**
  - **Note: encoding can be arbitrary.**

DISCS

Encoding may affect implementation!

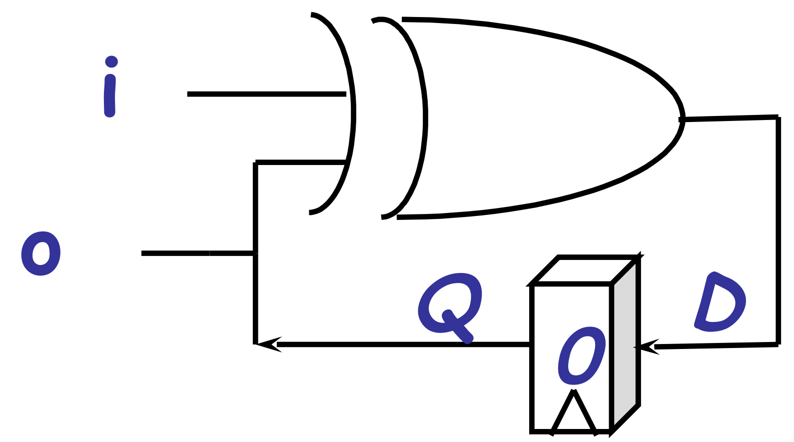
Choose $s_e = 0$ and $s_o = 1$

T

Q	i	D
0	0	0
0	1	1
1	0	1
1	1	0

O

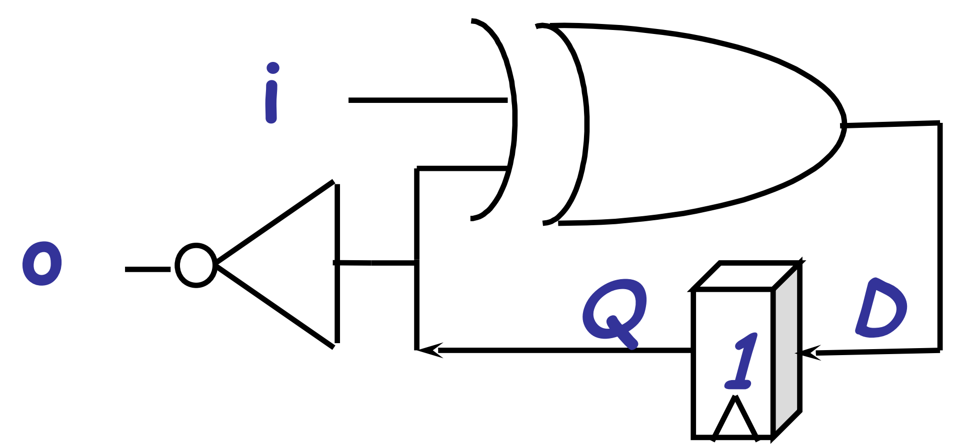
Q	o_1
0	0
1	1



Choose $s_e = 1$ and $s_o = 0$

Q	i	D
1	0	1
1	1	0
0	0	0
0	1	1

Q	o_1
1	0
0	1



Observations

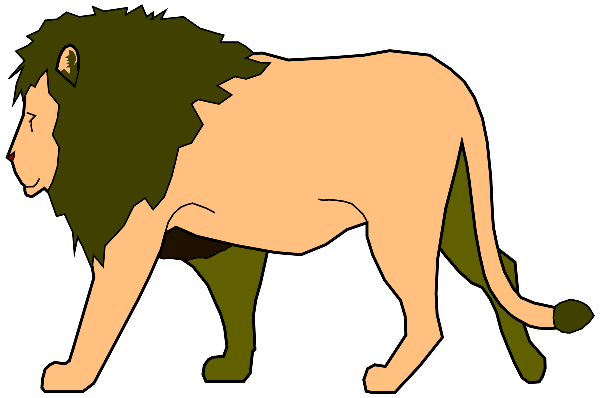
- Number of bits required to encode **K** states is $\lceil \log_2 K \rceil$ (amount of information).
- Encoding states results in combinational logic specifications for **T(Q, I)** and **O(Q)**.
- Choice of encoding affects complexity of logic implementation.
  - How does one find the optimum state encoding?
    - This is a hard problem, but it's not for CS152.
    - Sometimes, encoding is "natural", e.g., counters, numbers, etc. If so, it's easier to use it to ease thinking. Otherwise, arbitrary encoding will do.

FSM Design

- Understand the Problem
- Identify States, Inputs, Outputs
- Draw State Transition Diagram
- Reduce Redundant States
- Choose State Encoding
- Implement Logic

Tips on Identifying States

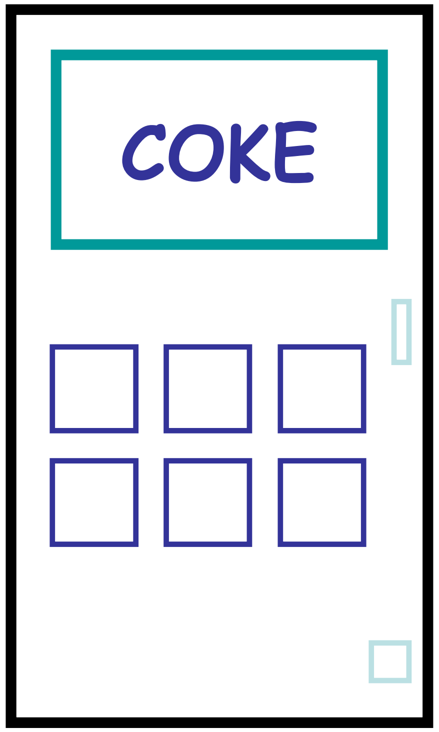
- Ask “What does the FSM need to remember?” Sometimes, answer is obvious.
  - Count in counters, odd/even in parity, empty/full bathroom, amount of money received/contained, etc.
- In general, look for “modes” where we get *different* ***behavior*** for the *same input*.
  - Give a hungry lion a big zebra, he’ll eat it. Give it another zebra right after, and he won’t. Ergo: Lion has (at least) 2 states -- hungry & full.



- *Usually*, different outputs means different states.
  - Light bulb has 2 states: OFF and ON
  - True for *Moore* machines.
  - Not true for *Mealy* machines, where outputs depend on inputs directly.

Design Example: The CS152 Coke Vending Machine

- Coke costs $.15 (roughly P7.50)
  - No, we are NOT going to argue about cost.
- Only nickels and dimes accepted.
  - Yes, you have nickels and dimes with you.
- FSM inputs:
  - N: Nickel
  - D: Dime
  - nothing (~N•~D)
- FSM outputs:
  - C: Drop a coke
  - R5: Return $.05

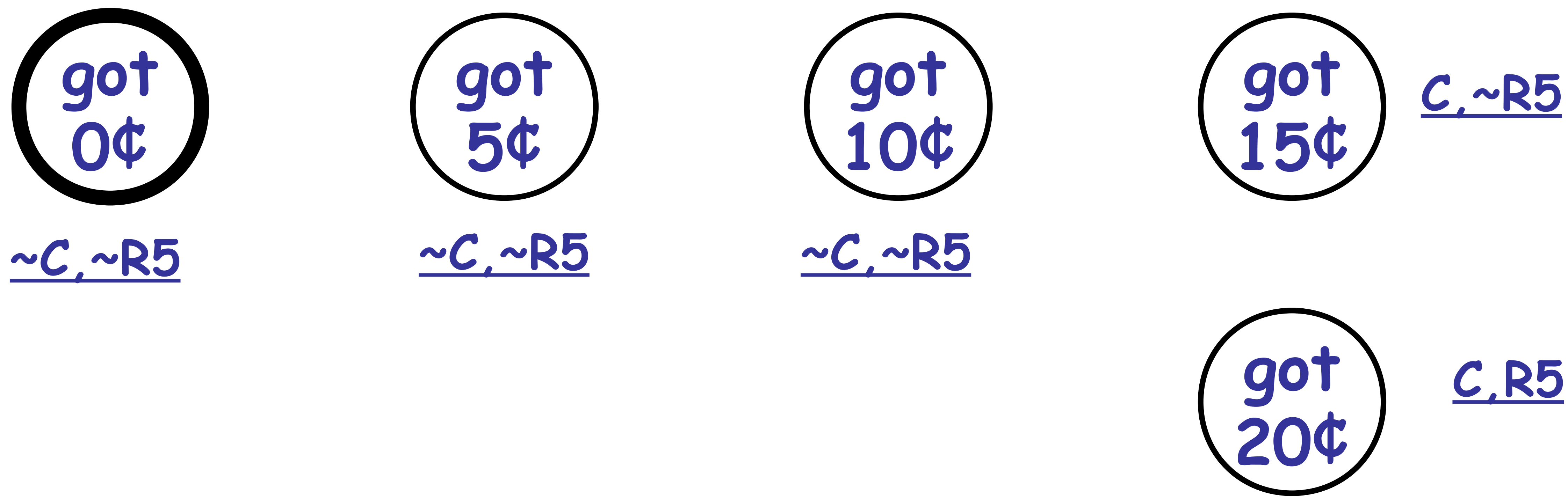


First Step: Identifying States

- "What does the FSM need to remember?"
- How many different modes are there?
- What is the output at each state?

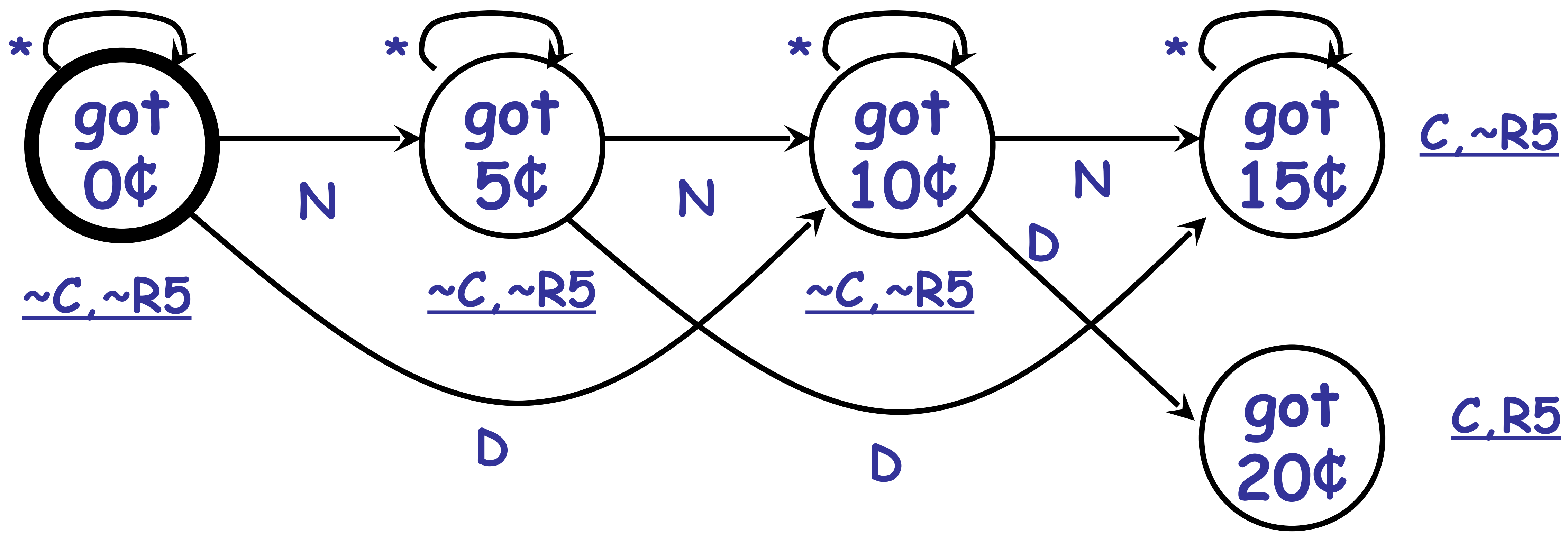
Next Step: Draw Diagrams

- Initially, there is no money in the vending machine.
- The machine only needs to remember how much was put in before and when it has enough to release a can of Coke.
  - You do not care how much money previous buyers have put in the vending machine so far, you just need to know if you have already put in enough to get a can of Coke, and if you need change.



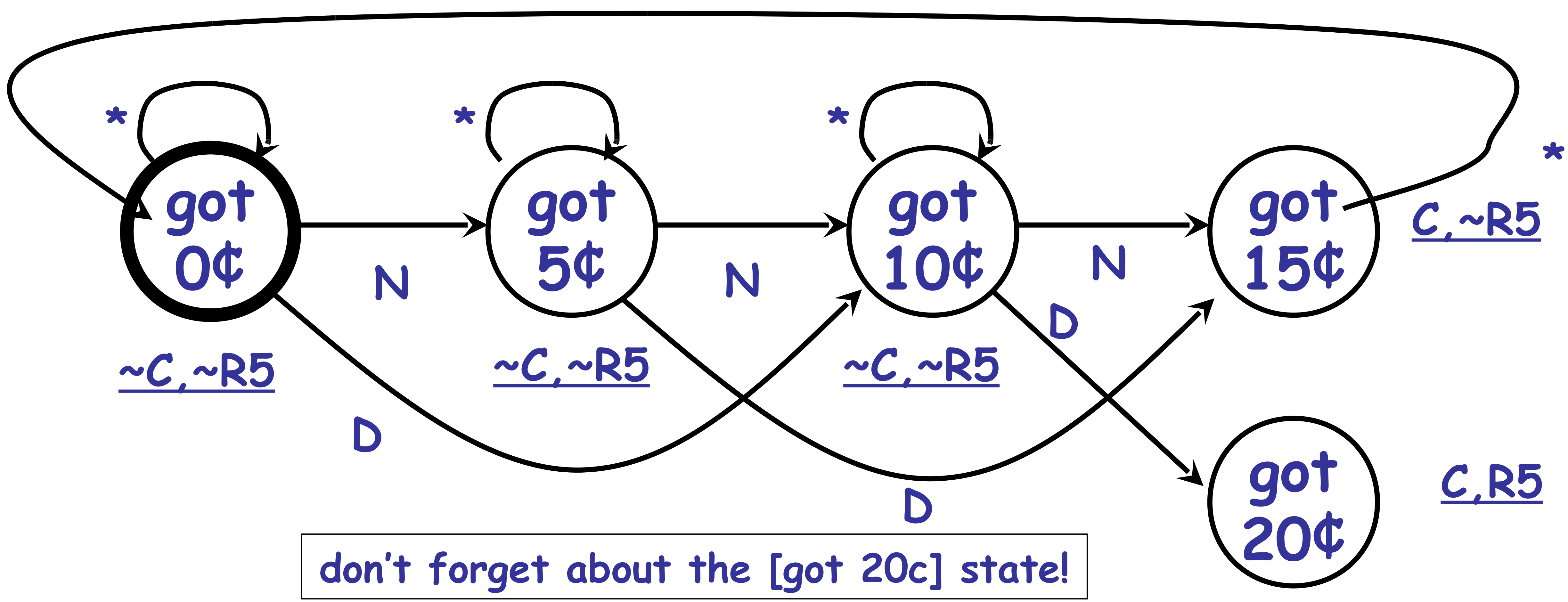
Next Step: Draw Diagrams

- Be sure to cover all input possibilities, but don't allow any input combination to point at more than one transition.
  - Remember to use * to indicate "everything else".
    - Assumed to stay in same state if not drawn.
  - We assume N and D never on at same time – better to make an error state for ND, and convert N to N(!D) and D to (!N)D?
- Here, if no input, *stay* in the same state.



Next Step: Draw Diagrams

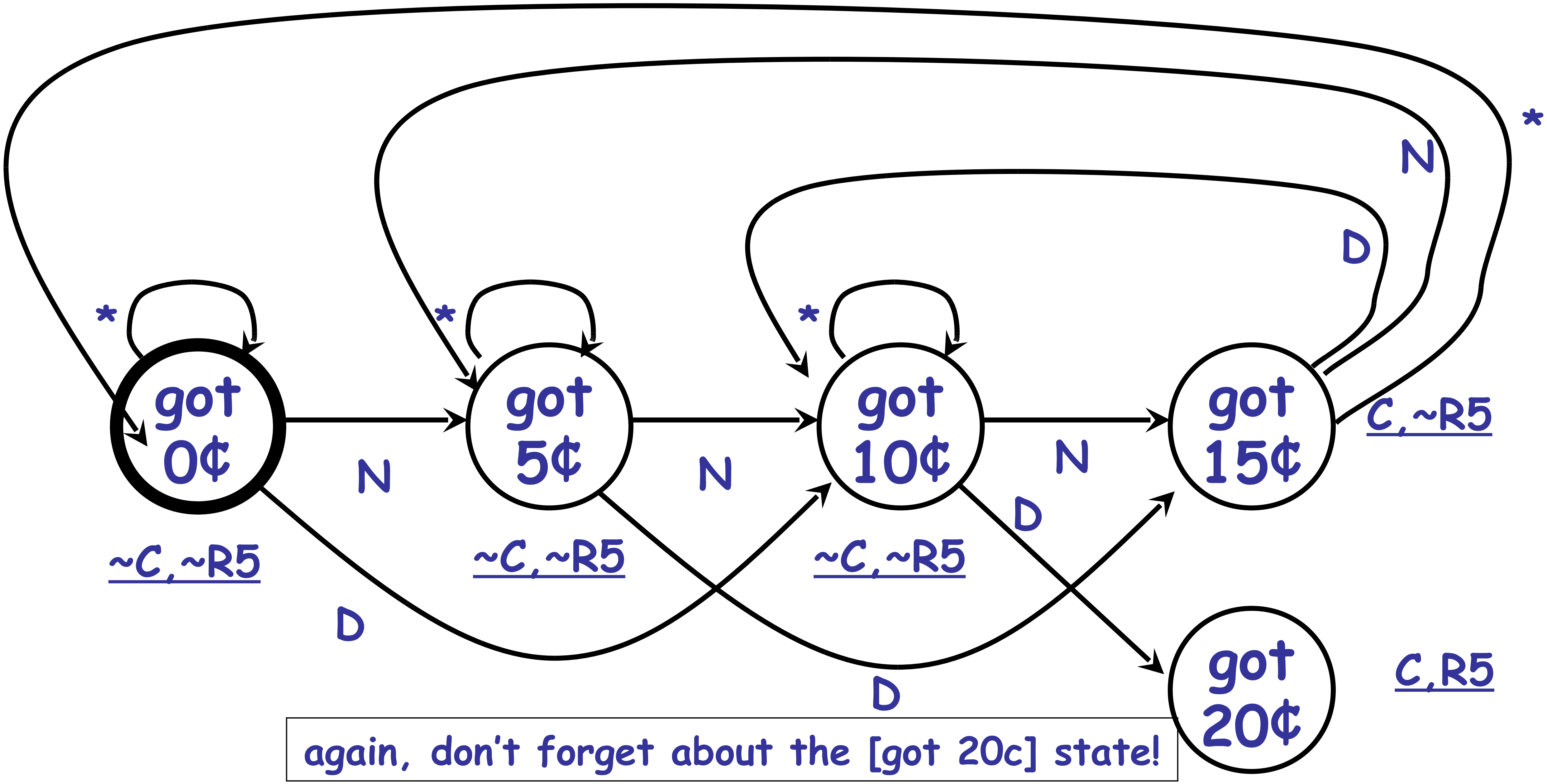
- What happens after Coke is given?
  - Need to go back to receiving state?
  - The current diagram ignores inputs during the cycle that "outputs" or gives Coke! (How do we fix this?)



DISCS

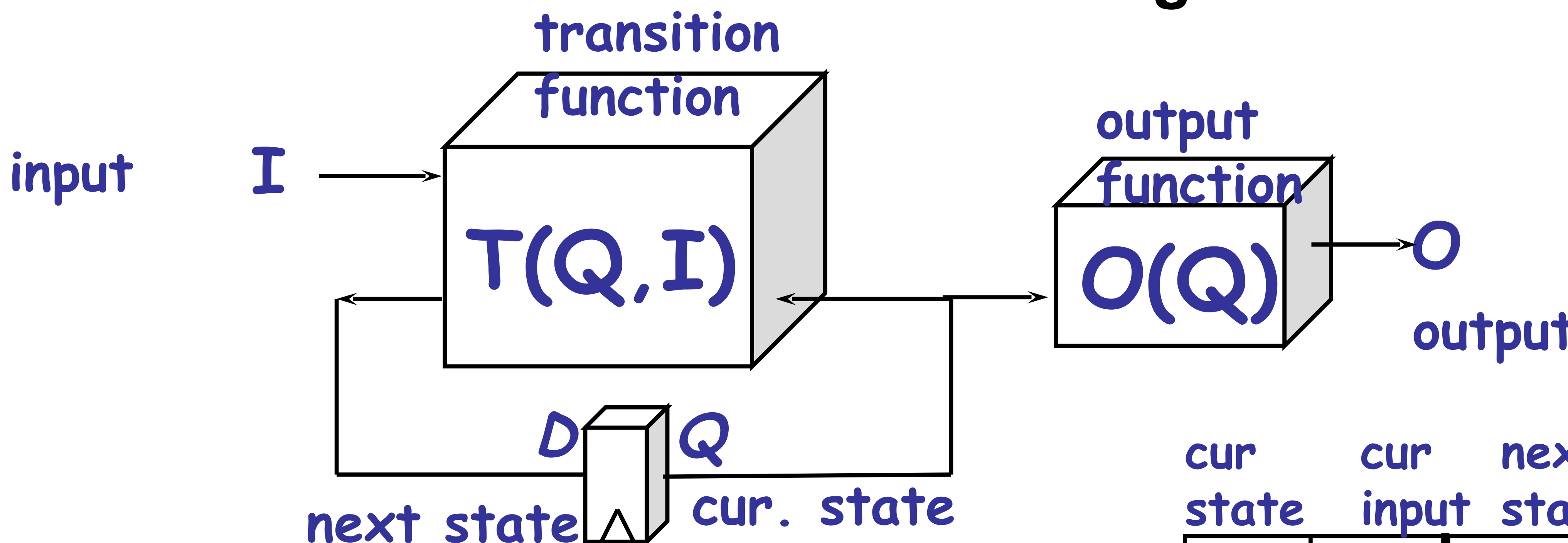
Next Step: Draw Diagrams

- Fix: go to the appropriate state directly!

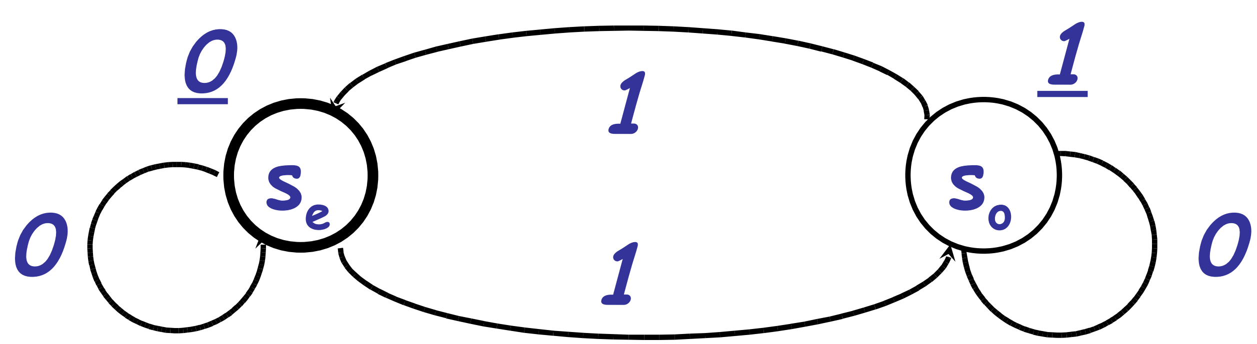


Moore Machines

- Output O is a function of the current state Q *only*.
  - This is what we have been using so far.

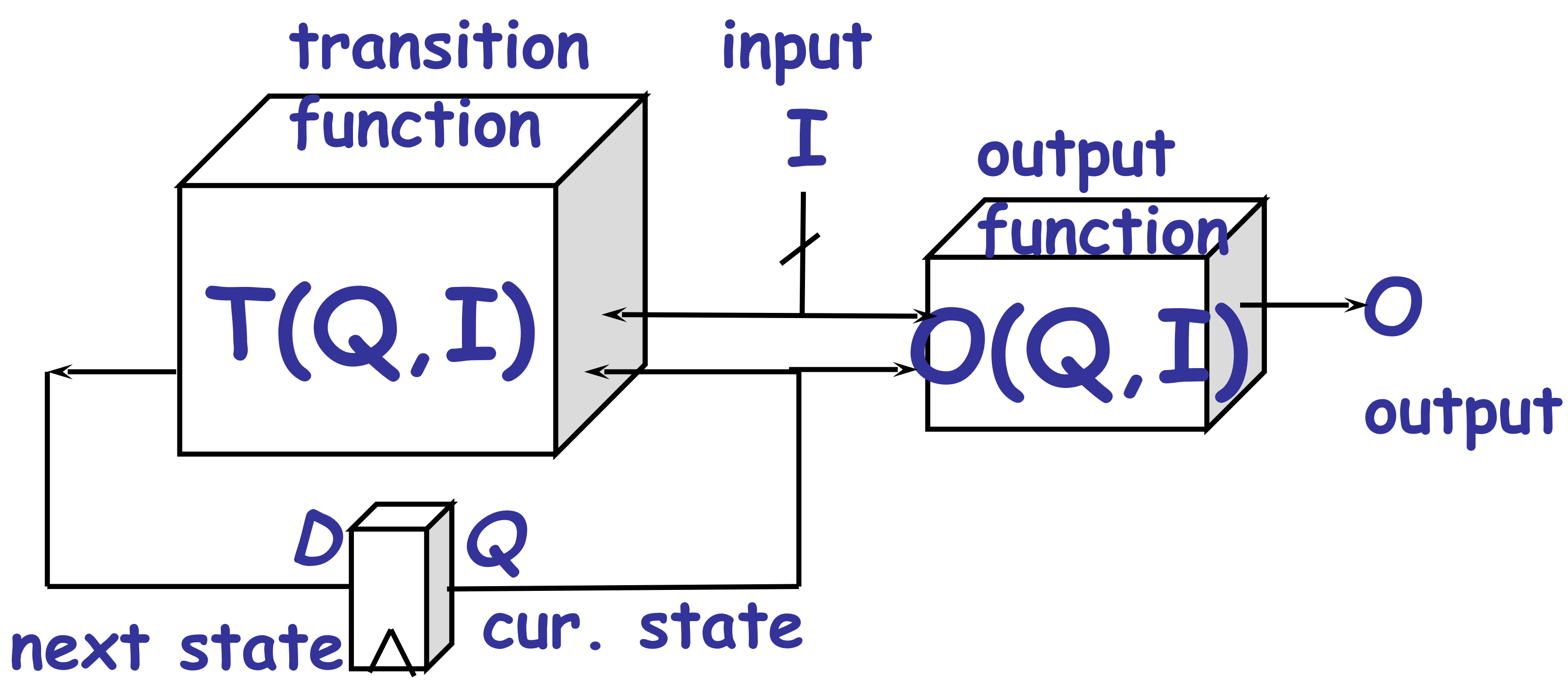


cur state	cur input	next state	cur output
se	0	se	0
se	1	so	0
so	0	so	1
so	1	se	1



Mealy Machines

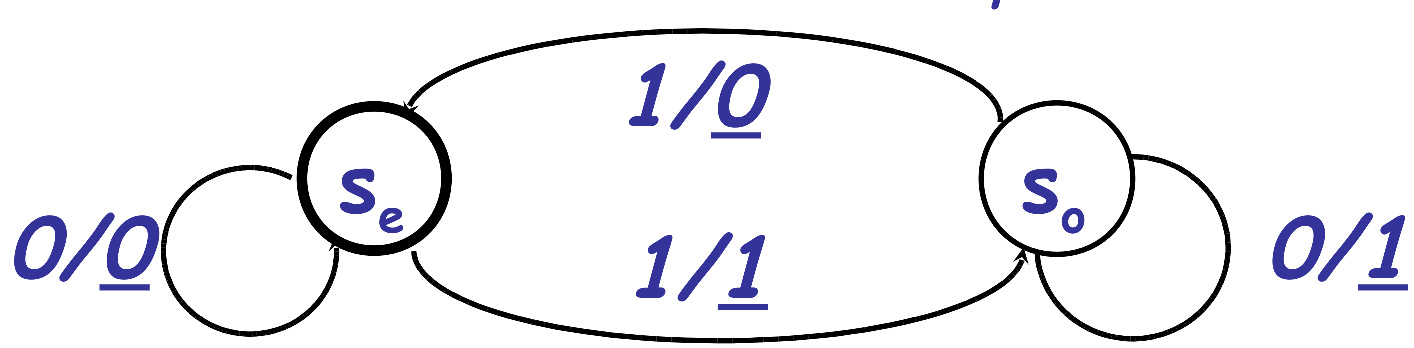
- Current output O is a function of both the current state Q and input I.



- Possibility of different outputs for current state.
- Output can change *sooner*.

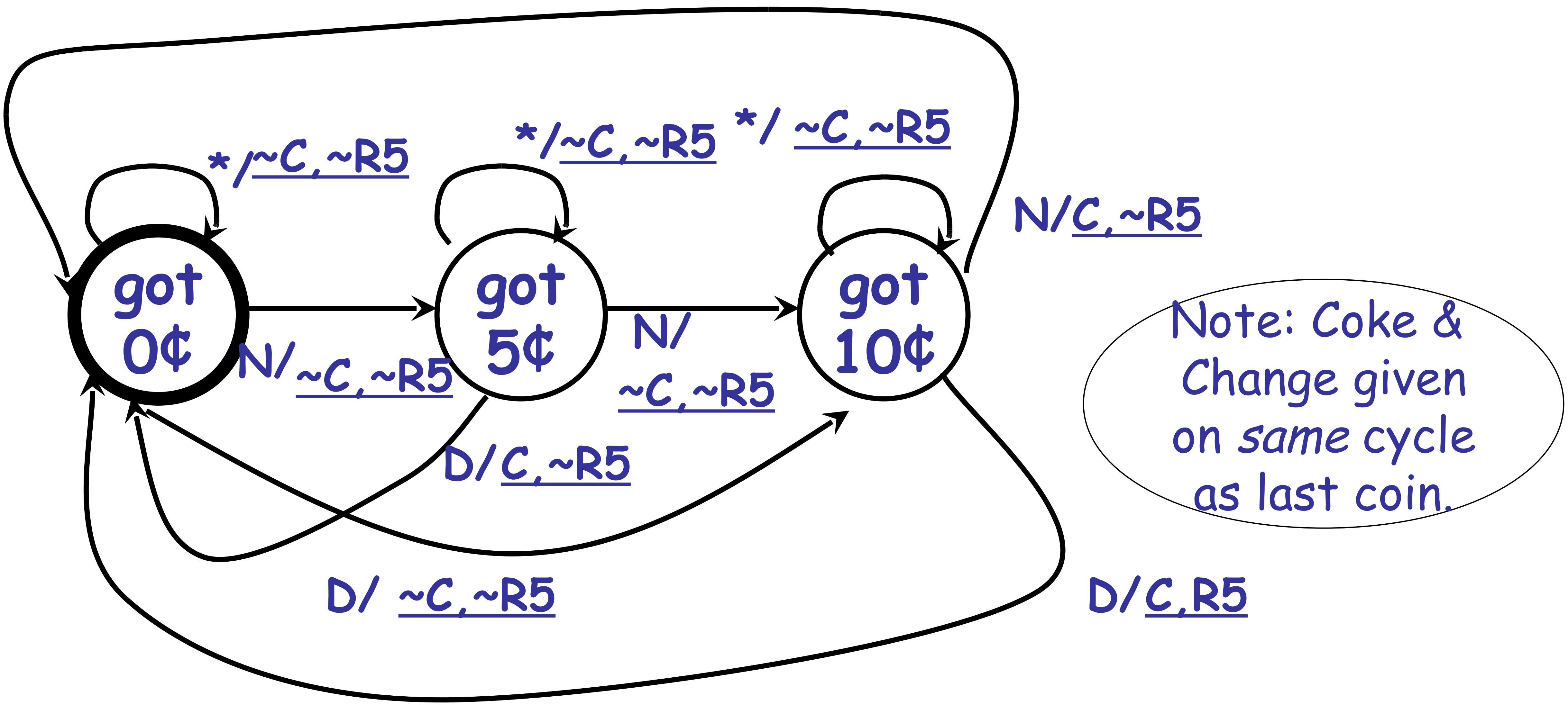
cur state	cur input	next state	cur output
se	0	se	0
se	1	so	1
so	0	so	1
so	1	se	0

Note: This FSM is NOT equivalent to the Moore FSM in the previous slide!



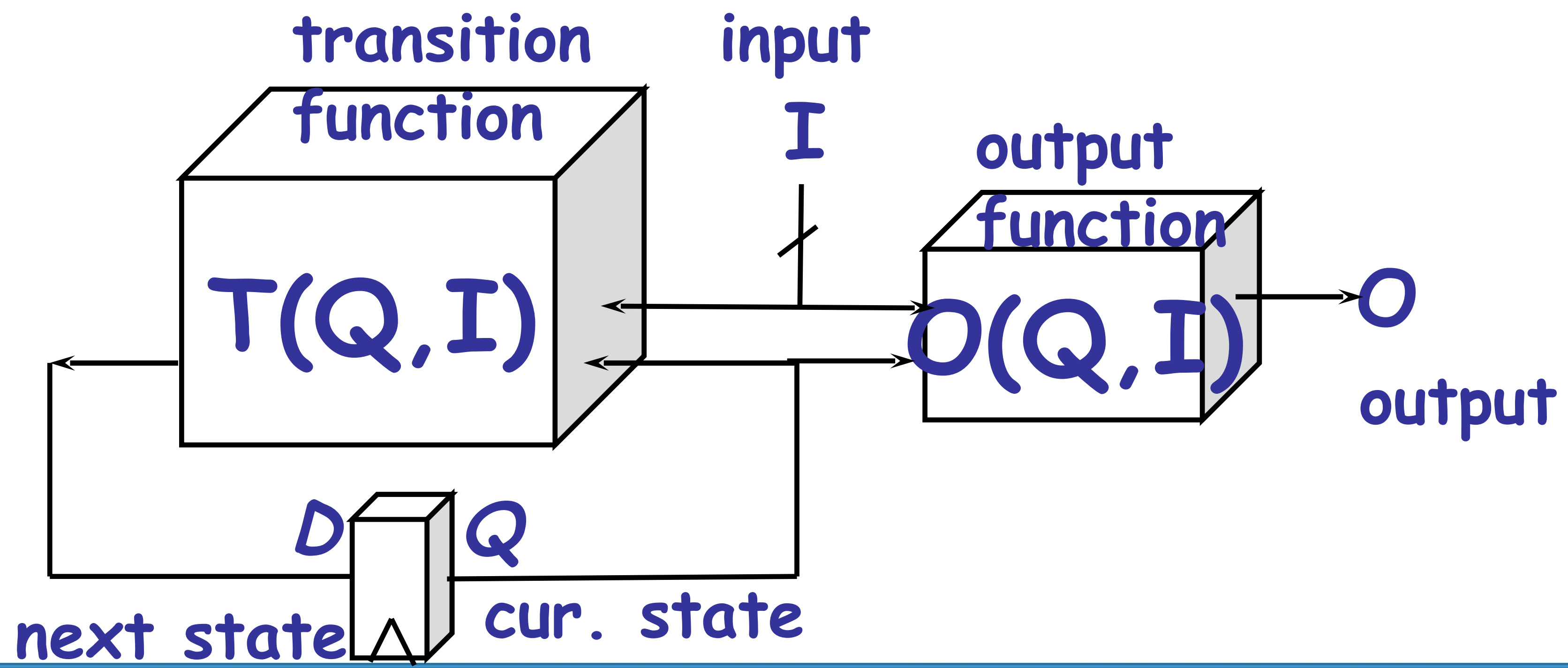
Mealy Coke Machine

- Uses fewer states and also makes it easier to fix give-Coke-then-reset problem.



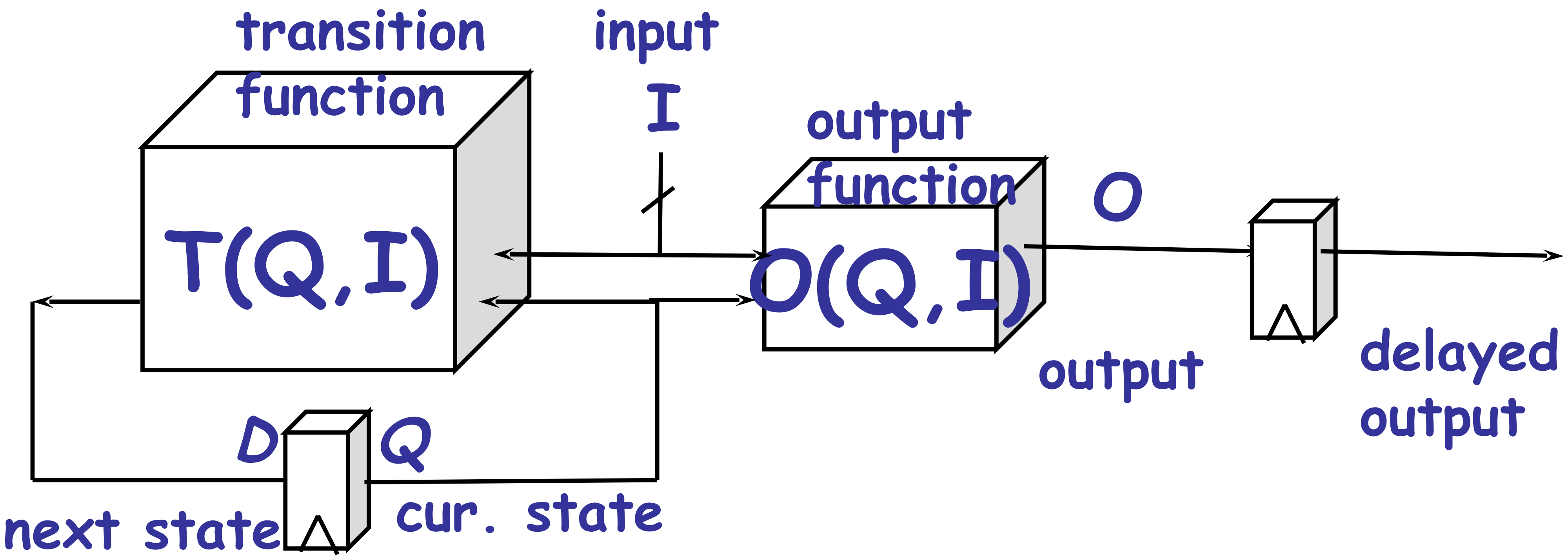
Mealy Machines

- In general, Mealy is more expressive.
  - More functionality for less states.
- But dangerous!
  - Combinational path from I to O!
  - Output vulnerable to glitches caused by input changes.
    - Sensitive to hazards, bad for things like traffic lights!
  - Can form cycles if FSMs are chained together!
    - What if output of one is input of another?



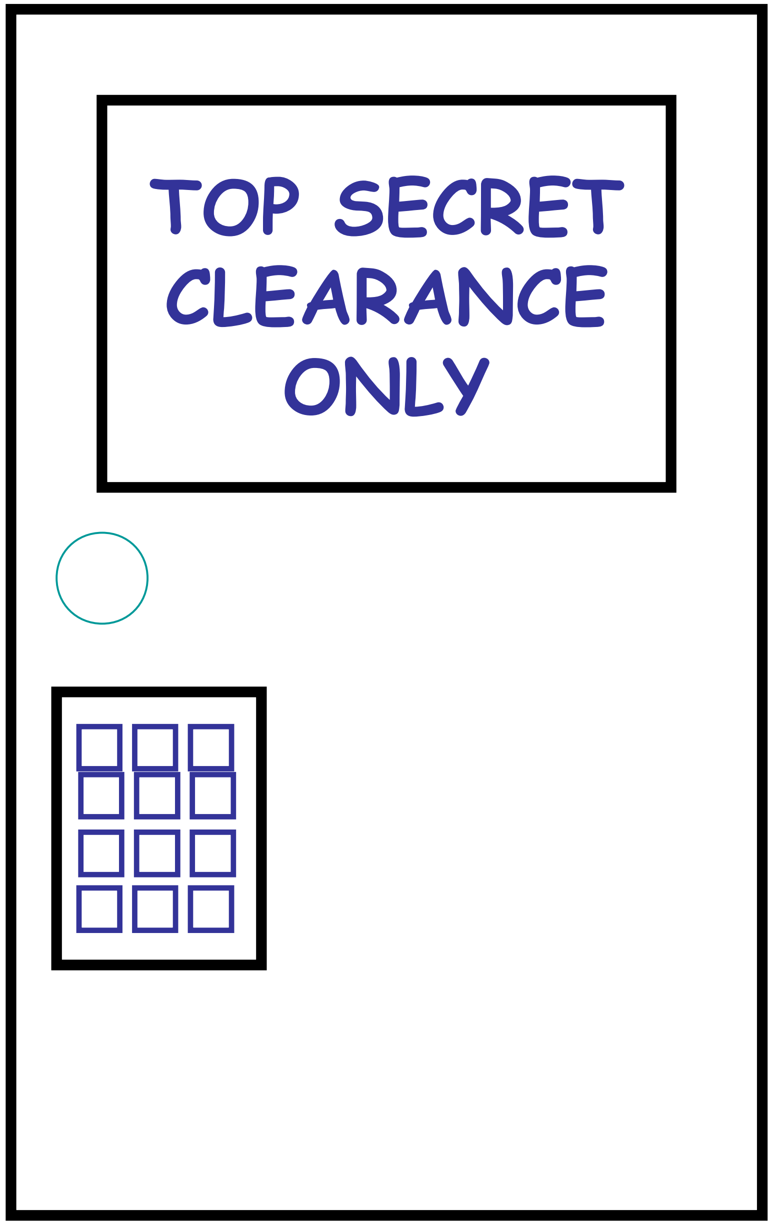
Synchronous Mealy Machines

- Putting a reg on output helps, but delays output.
- This causes the same issue found in Moore machines.
  - Have to wait for the cycle after input is given to see a change in the output.



Another Design Example: Password Lock

- Inputs:
  - digits 0 to 9
  - T (timer stopped): asserted 10 secs. after timer starts
- Outputs:
  - U: unlock door, start timer
- Behavior:
  - If the *last 5 digits* seen equals “12123”, unlock and start timer.
  - When timer stops, lock again.
    - Better get out quickly!



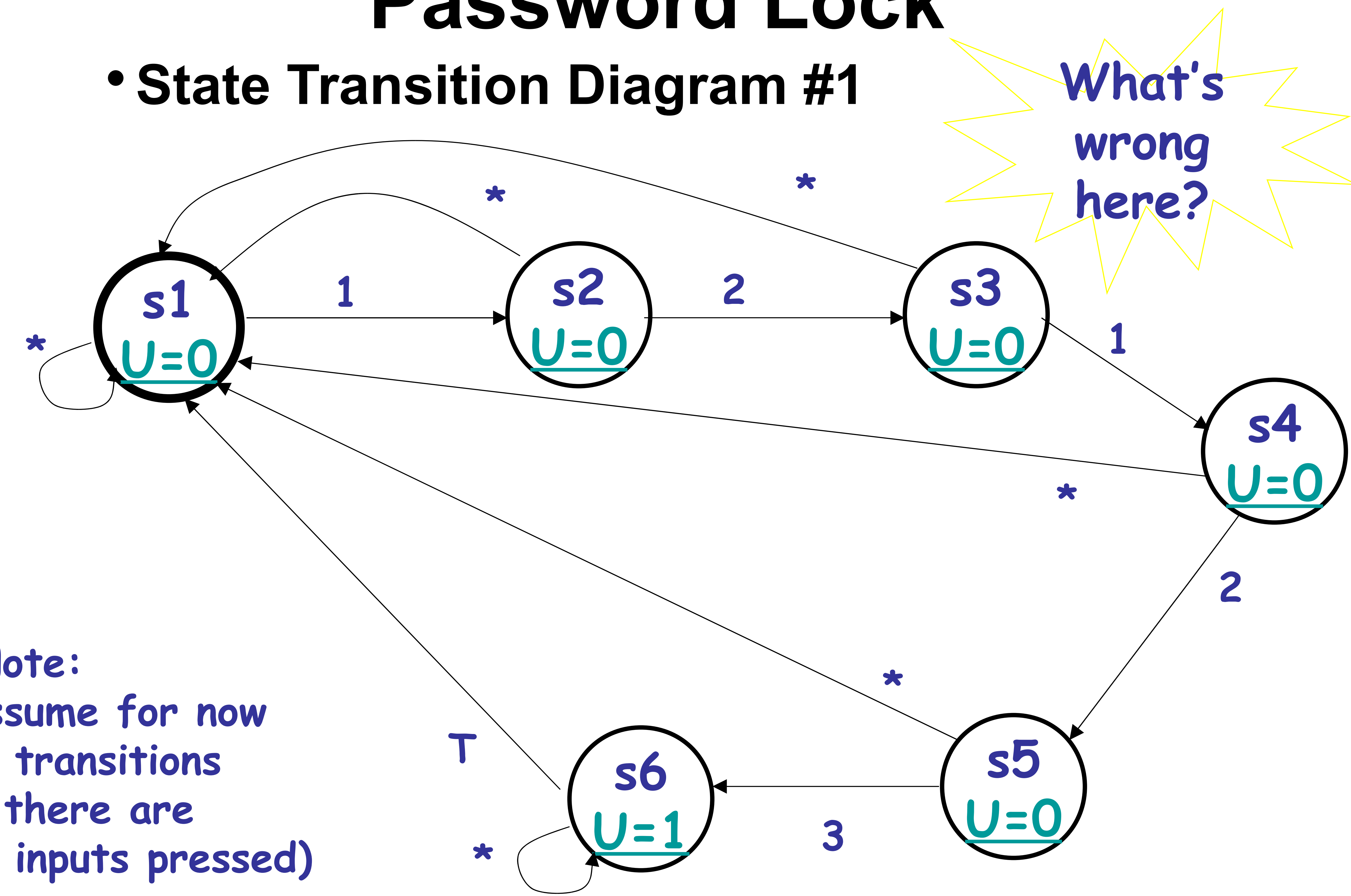
Password Lock

- State Transition Diagram (let's try it on our own first!)

(Note:
Assume for now
no transitions
if there are
no inputs pressed)

Password Lock

- State Transition Diagram #1

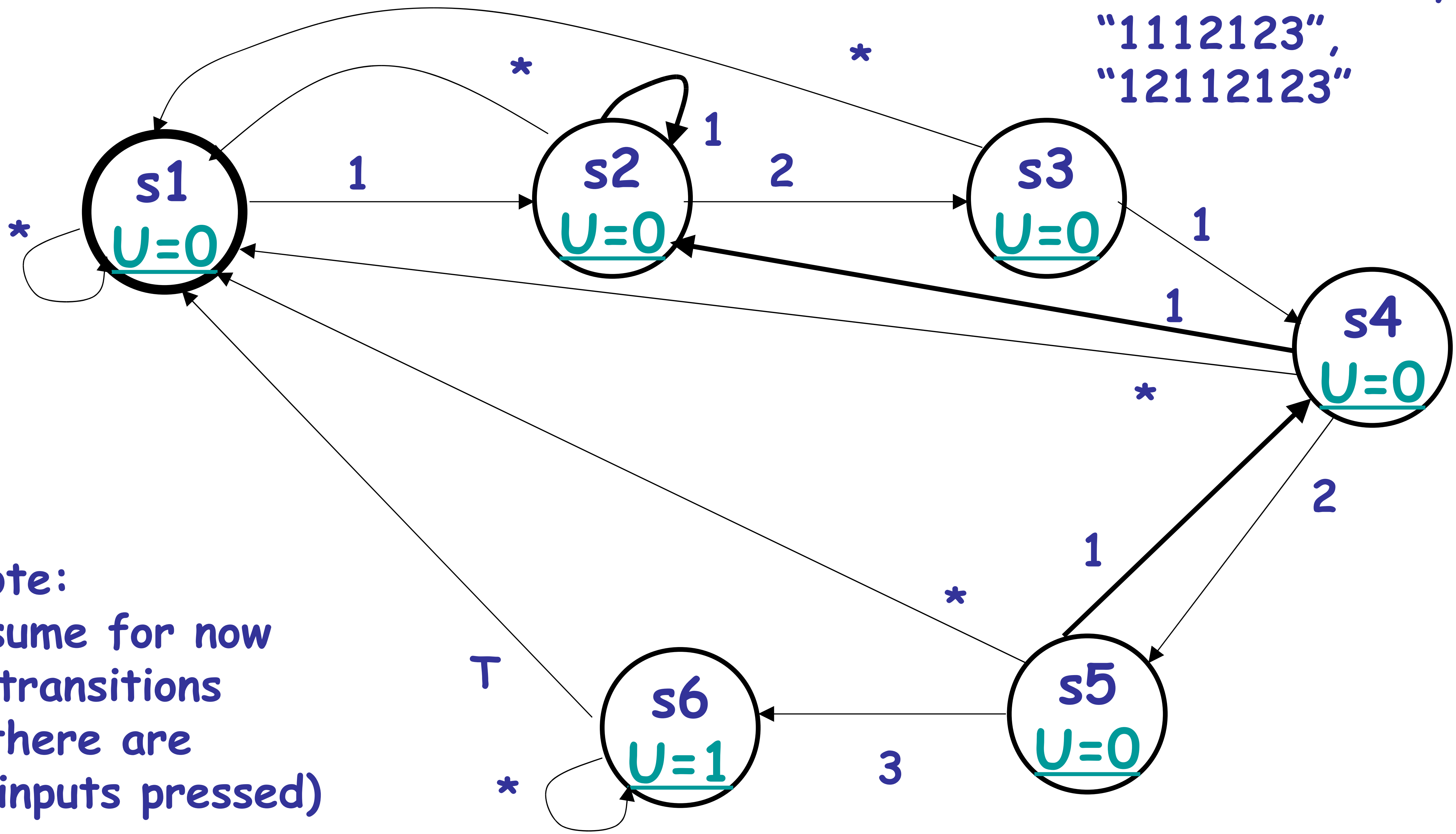


(Note:
Assume for now
no transitions
if there are
no inputs pressed)

Password Lock

- **Fixed State Transition Diagram**

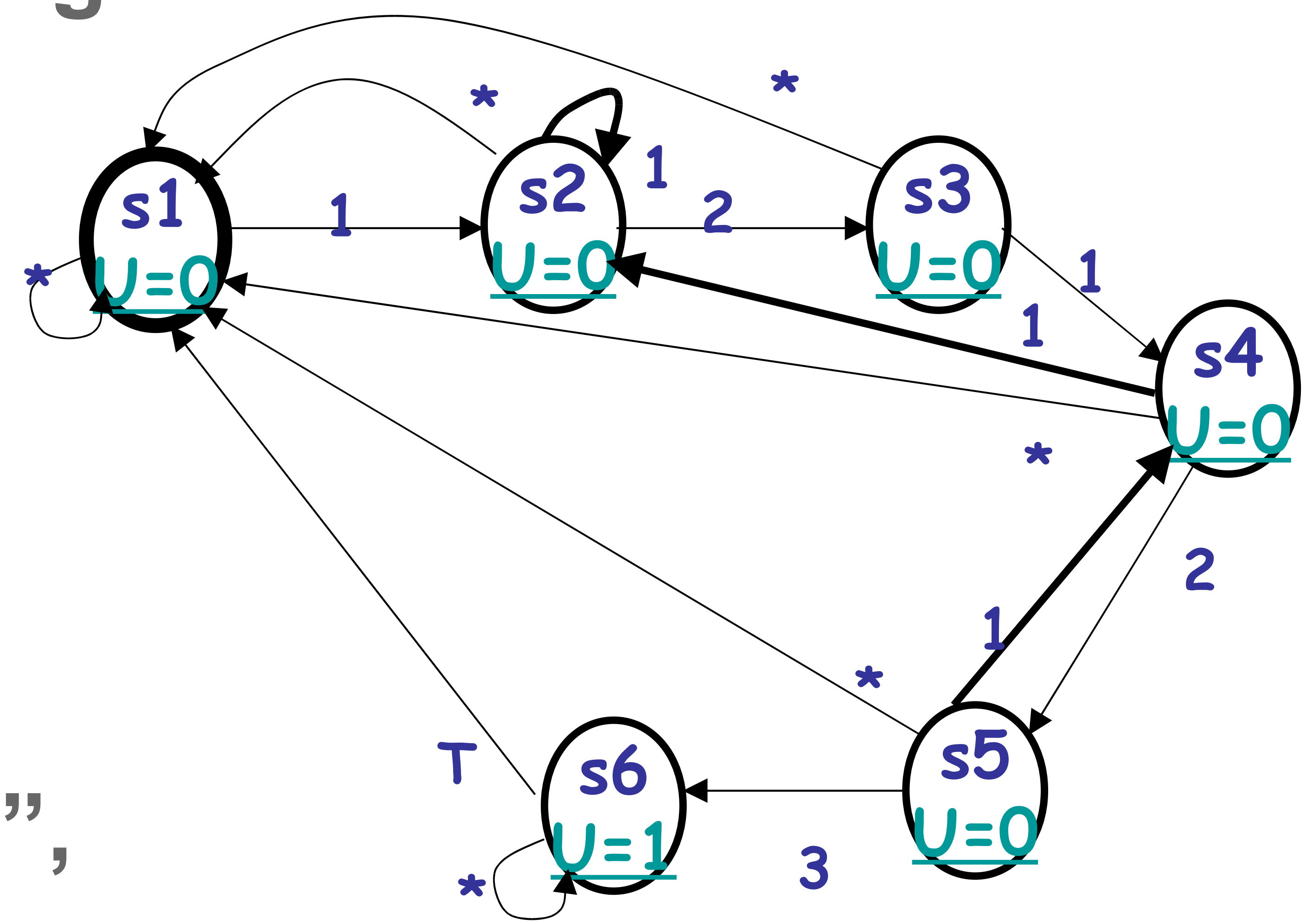
This now works for: "1212123", "1112123", "12112123"



(Note:
Assume for now no transitions if there are no inputs pressed)

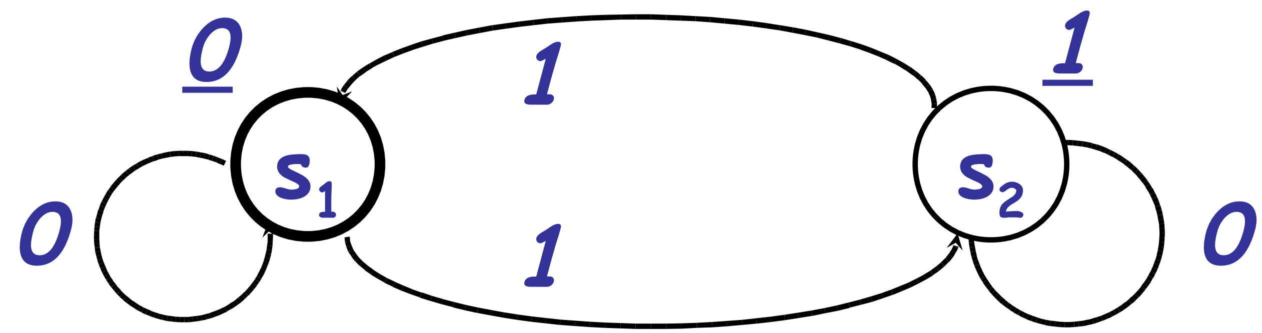
Password Lock

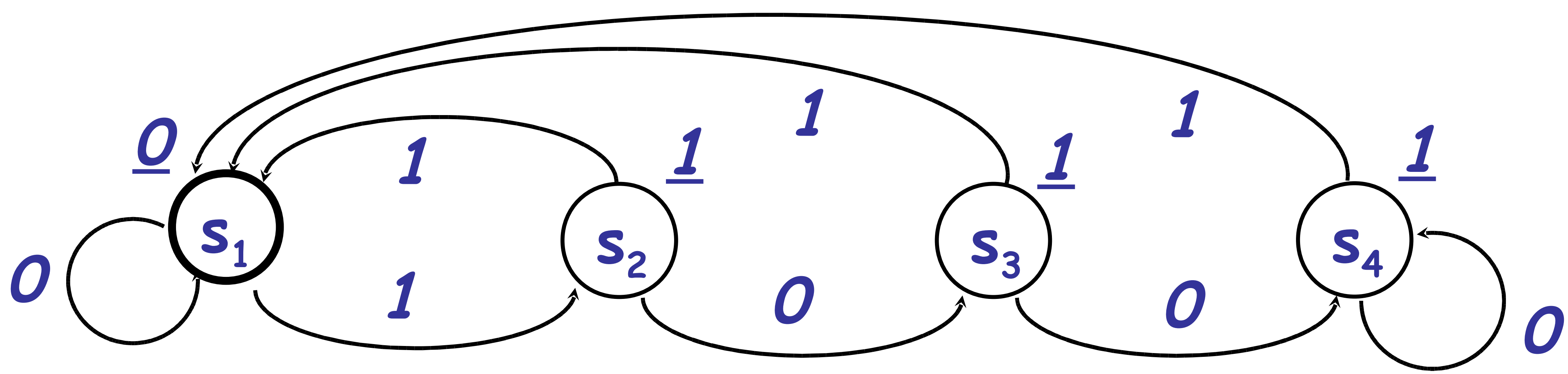
- Fixed State Transition Diagram
- s1 = seen nothing
- s2 = seen “1”
- s3 = seen “12”
- s4 = seen “121”
- s5 = seen “1212”
- s6 = seen “12123”, unlock until T ________



FSM Equivalence

- Are these two FSMs equivalent?





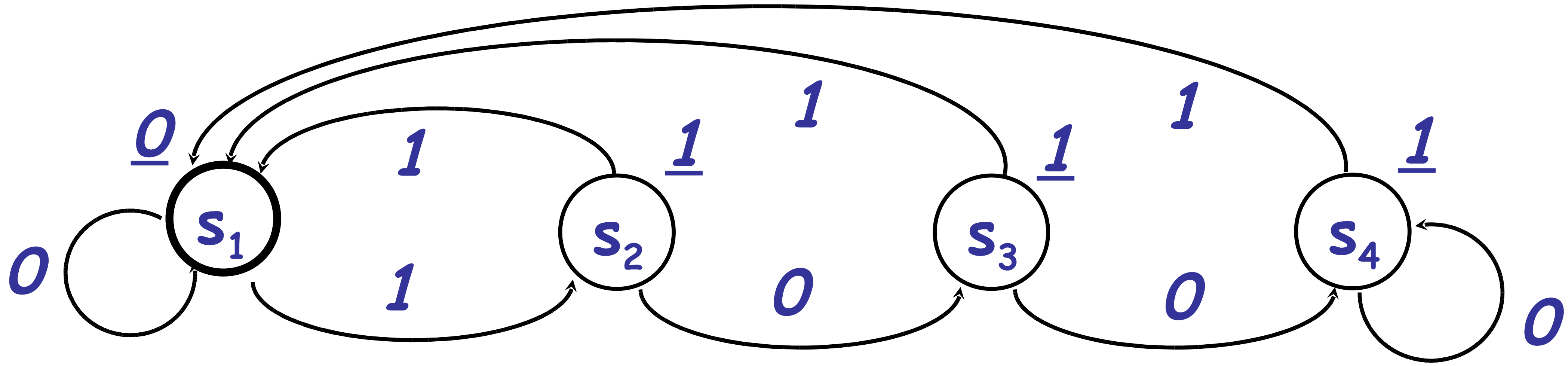
Equivalence and Minimization

- Two FSMs are equivalent if and only if every input sequence yields identical output sequences.

- **Goal**: Given an FSM, find the simplest equivalent FSM with a minimum number of states.
  - Two states s_1 and s_2 in an FSM are *equivalent* if and only if *each input sequence beginning from s_1 yields an output sequence identical to that obtained by starting from s_2.*

DISCS

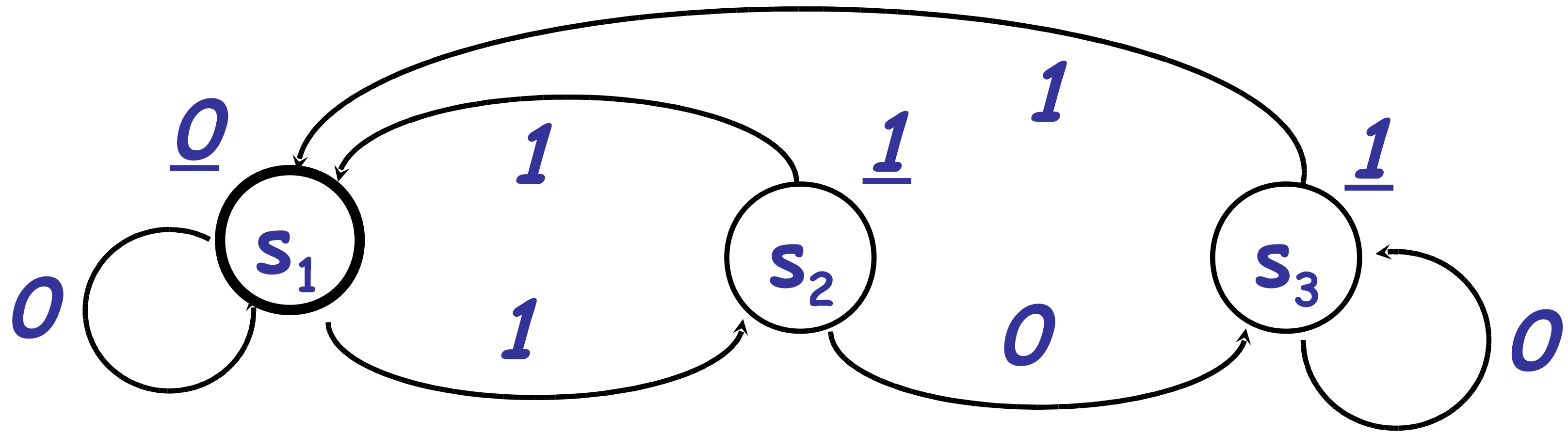
Example Minimization



Current State	Next State I=0	Next State I=1	Output
s1	s1	s2	0
s2	s3	s1	1
s3	s4	s1	1
s4	s4	s1	1

for same input,
same next state,
same output
-> equivalent!

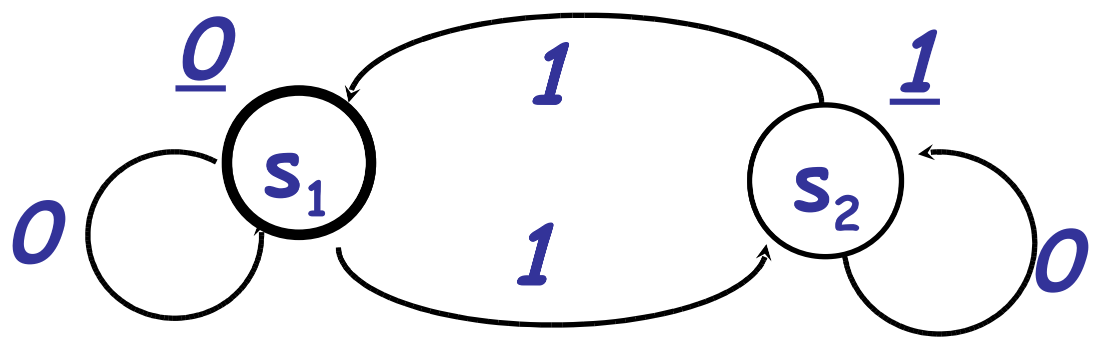
Example Minimization



Current State	Next State I=0	Next State I=1	Output
s1	s1	s2	0
s2	s3	s1	1
s3	s3	s1	1

for same input,
same next state,
same output
-> equivalent!

Example Minimization



Current State	Next State I=0	Next State I=1	Output
s1	s1	s2	0
s2	s2	s1	1

for same input,
diff. next state,
diff. output
-> NOT equivalent!

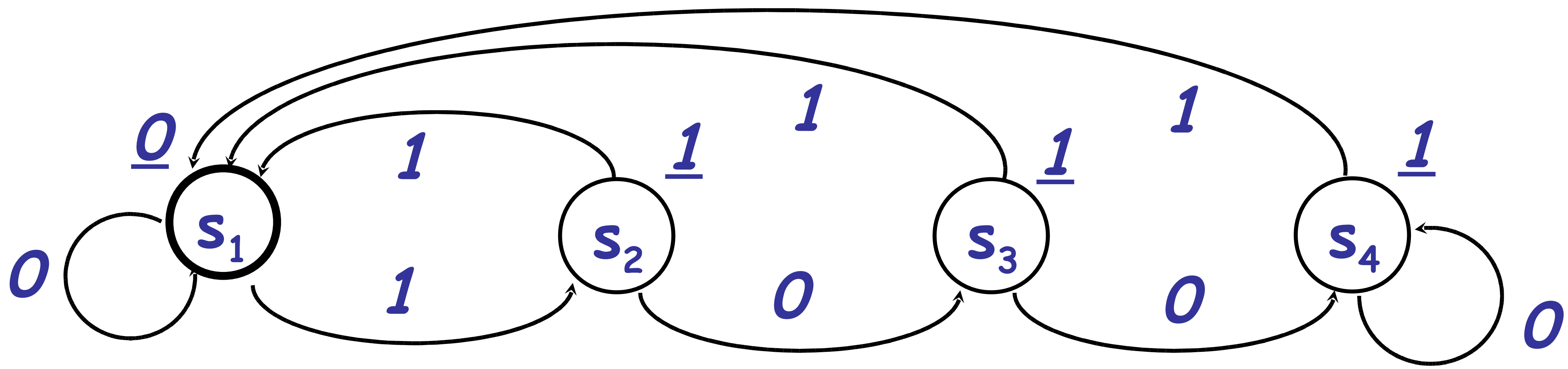
FSM Minimization

- Look at each pair of states (s_1, s_2) in the FSM.
- If s_1 produces different outputs from s_2 , mark them non-equivalent.

- For each state pair (s_1, s_2) not yet marked, and for each input i, find state pair $(T(s_1, i), T(s_2, i))$.
- If $(T(s_1, i), T(s_2, i))$ are marked non-equivalent for any i, mark (s_1, s_2) non-equivalent.
- Iterate until no more marking is possible.

- Unmarked state pairs are equivalent, simplify FSM accordingly.

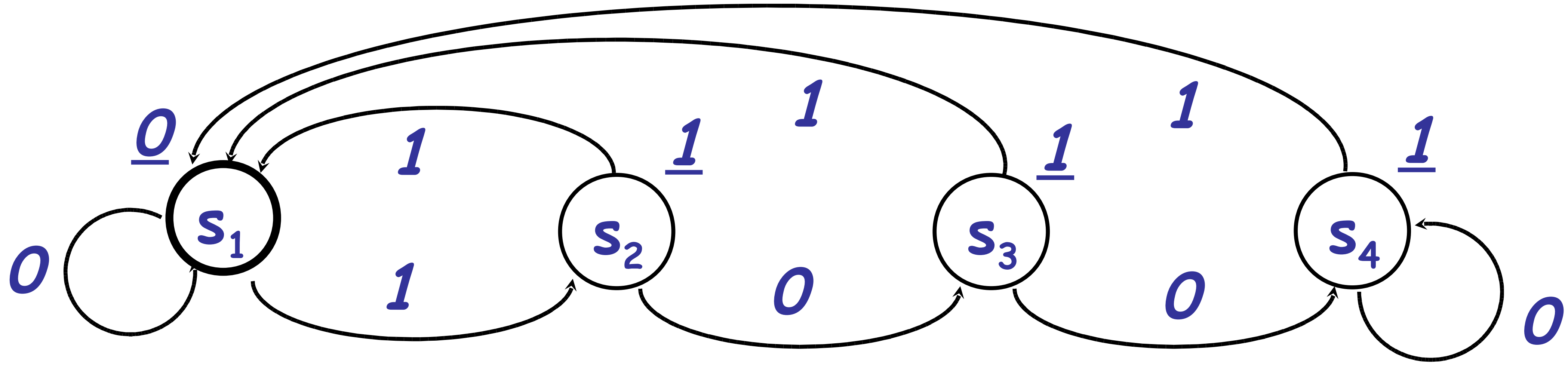
Minimization



s2			
s3			
s4			
	s1	s2	s3

For each box (state pair):
- Mark X if different output.
- If same output, write next state pairs depending input.
- If state pairs are not equiv., mark X.
- Do from top to bottom, left to right.

Minimization

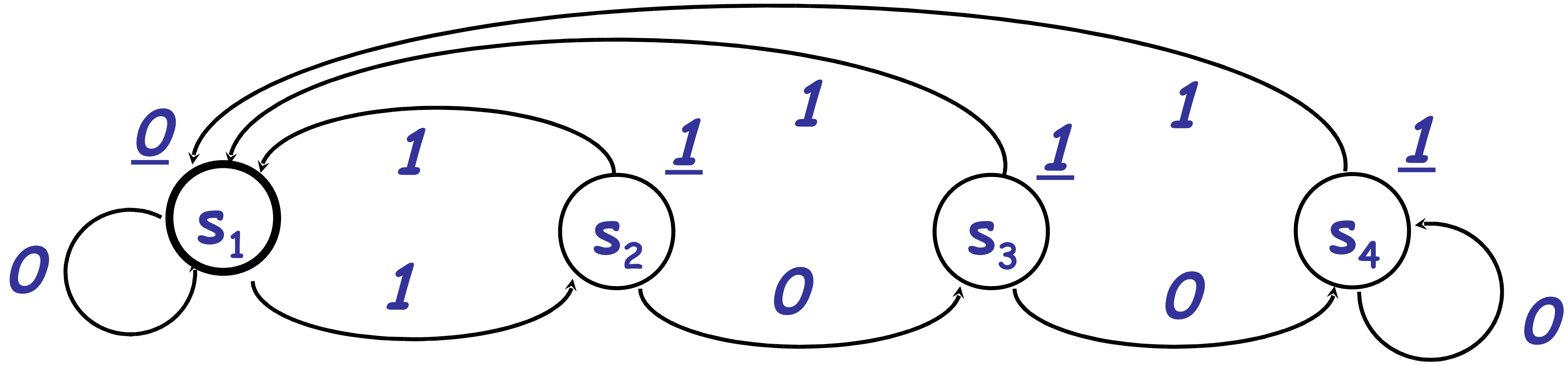


	s1	s2	s3
s2	X		
s3	X		
s4	X		

For each box (state pair):

- Mark X if different output.
- If same output, write next state pairs depending input.
- If state pairs are not equiv., mark X.
- Do from top to bottom, left to right.

Minimization



next state on 0
from s3

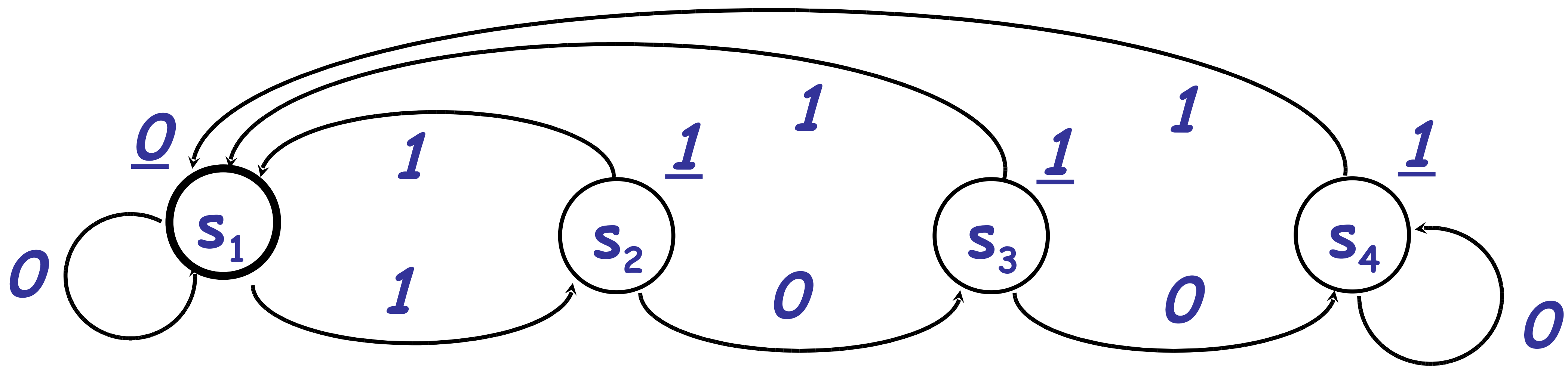
next state on 1
from s3

s2	X		
s3	X	s3:(s4,s1) s2:(s3,s1)	
s4	X	s4:(s4,s1) s2:(s3,s1)	s4:(s4,s1) s3:(s4,s1)
	s1	s2	s3

For each box (state pair):

- Mark X if different output.
- If same output, write next state pairs depending input.
- If state pairs are not equiv., mark X.
- Do from top to bottom, left to right.

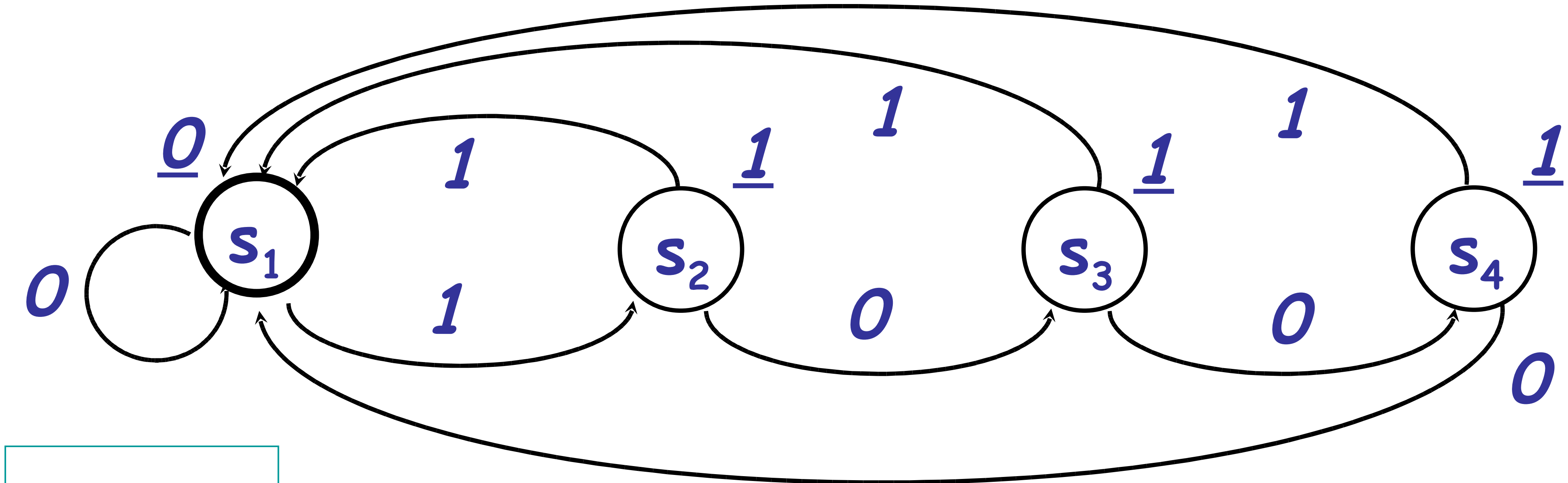
Minimization



	s1	s2	s3
s2	X		
s3	X	s3:(s4,s1) s2:(s3,s1)	
s4	X	s4:(s4,s1) s2:(s3,s1)	s4:(s4,s1) s3:(s4,s1)

For each box (state pair):
- Mark X if different output.
- If same output, write next state pairs depending input.
- If state pairs are not equiv., mark X.
- Do from top to bottom, left to right.

Another Example



	s1	s2	s3
s2			
s3			
s4			

For each box (state pair):

- Mark X if different output.
- If same output, write next state pairs depending input.
- If state pairs are not equiv., mark X.
- Do from top to bottom, left to right.