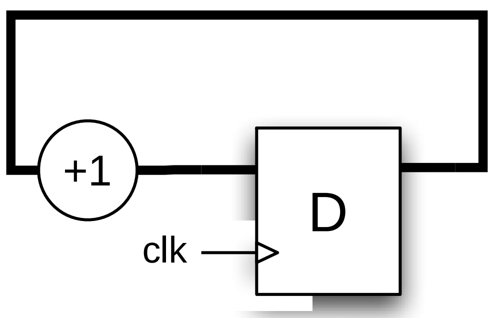
CX1005
Digital Logic

Finite State Machines

Sequences and States

- We have previously seen how we can design a binary counter
- Binary counters output a sequence of numbers, each being one more than the previous value
- Can I provide an abstract description of the behavior of the counter?

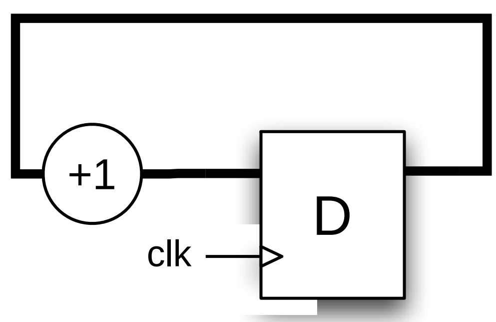



```
module simplecnt (input clk, rst,
                  output reg [3:0] q);

always@(posedge clk)
begin
    if(!rst)
        q <= 4'b0000;
    else
        q <= q + 1'b1;
end

endmodule
```

Sequences and States

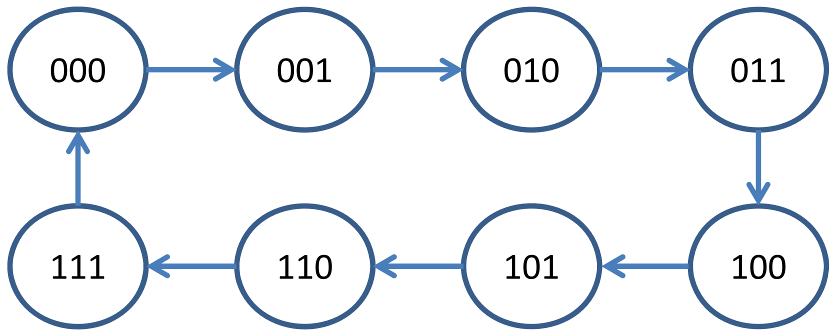
- For the counter, the sequence is easy to reason about
- At each point in time, the system is in a *state*, that determines what the output should be
- At each transition point, the circuit moves from its current state to the next state, corresponding to the one with the next largest output
- We can think of circuits as being *state machines*

0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1



Sequences and States

- We can use a figure to show these transitions:



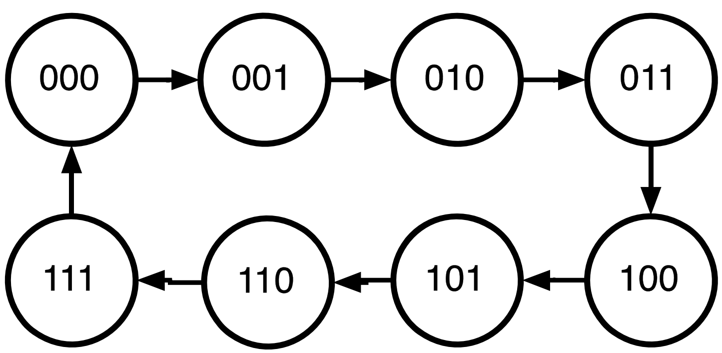
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1

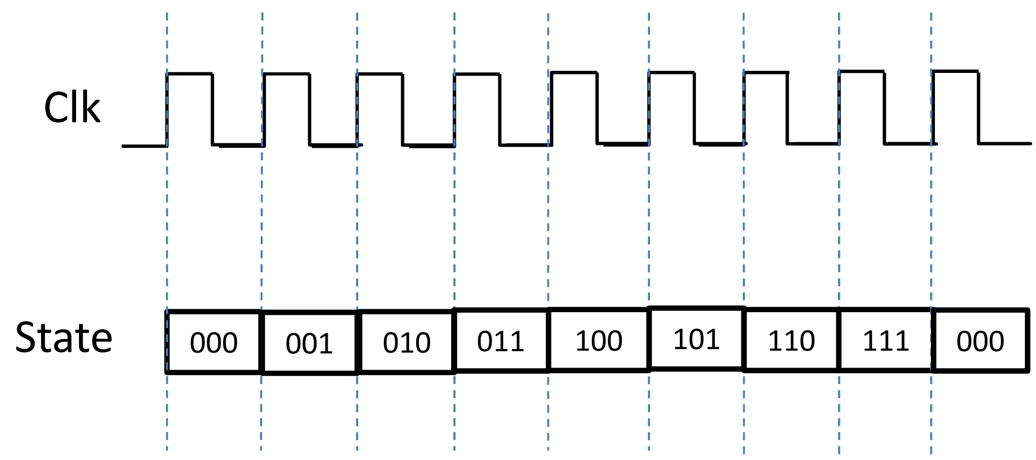
This is a called *state transition diagram*

- Each node is a possible *state* of the system
- It shows the movement from one state to the next
- In this case, the states are simply labeled with their output values

Sequences and States

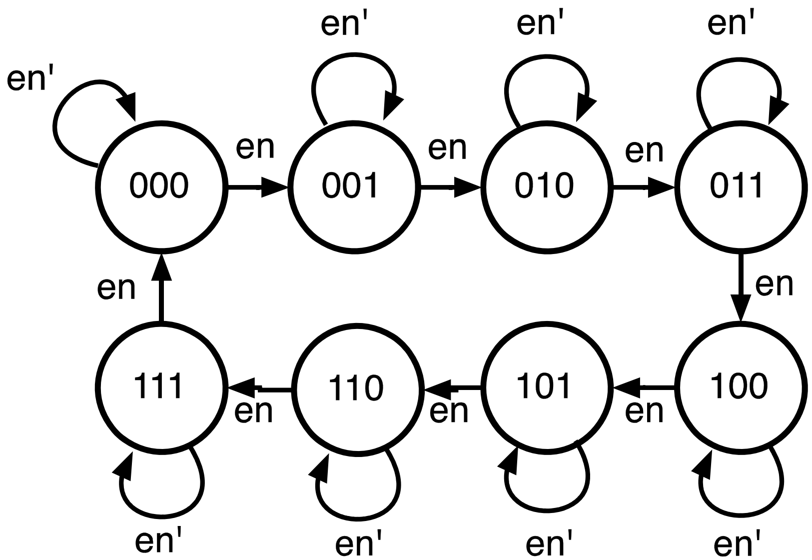
- In the previous diagram, the system always moved from one state to the next

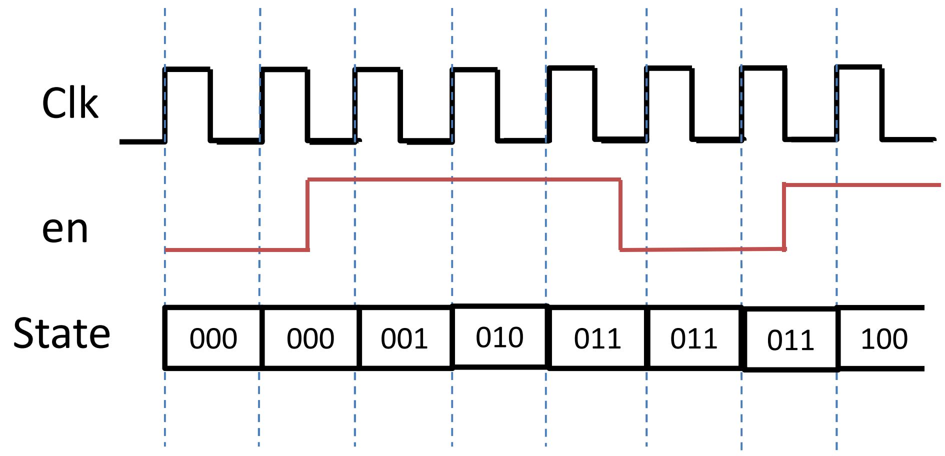




Sequences and States

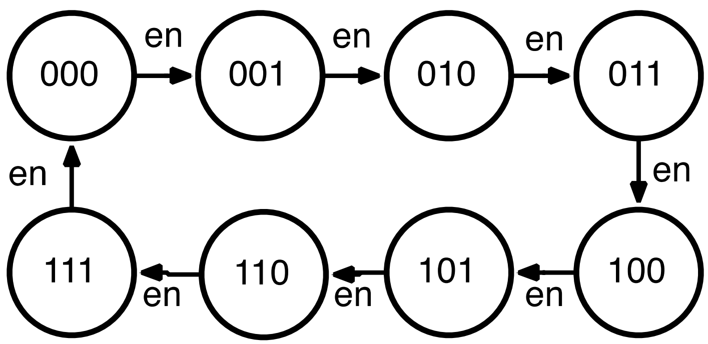
- In the previous diagram, the system always moved from one state to the next
- What about an enabled counter that only counts when an *en* input is high?
- We can indicate conditions for transition on the transition arrows:





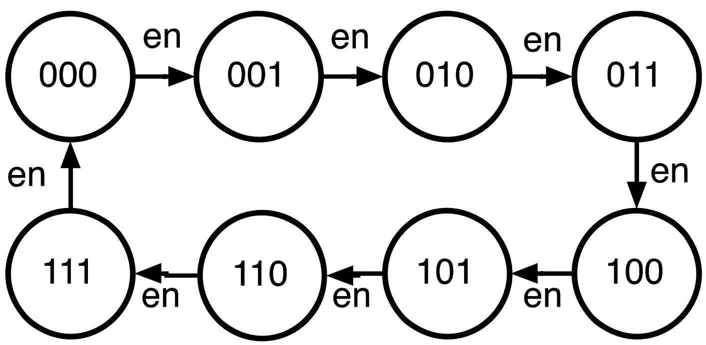
Sequences and States

- Here, we follow the standard count sequence when *en* is high
- If *en* is low, we stay in the current state
- We can simplify the diagram by assuming that conditions not shown result in self-transitions
- The state only changes when *en* is 1, otherwise, stay



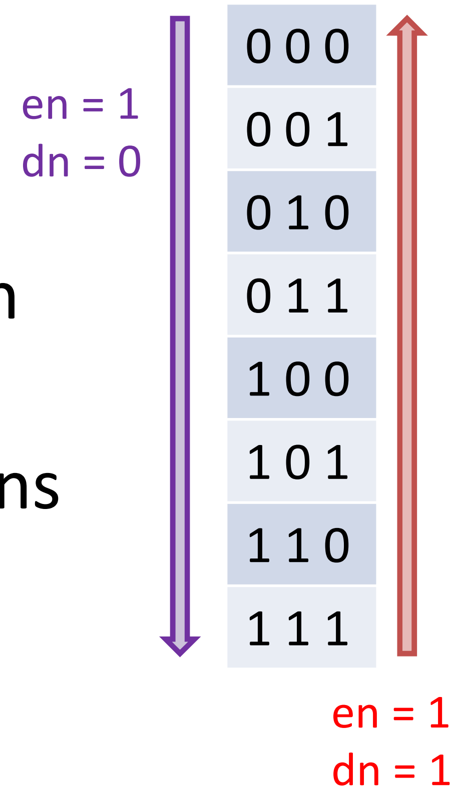
Sequences and States

- So the state transition diagram consists of:
  - Circle nodes that represent the states
  - Arrows between states to represent possible transitions
  - Conditions place on the arrows to indicate when the transition occurs

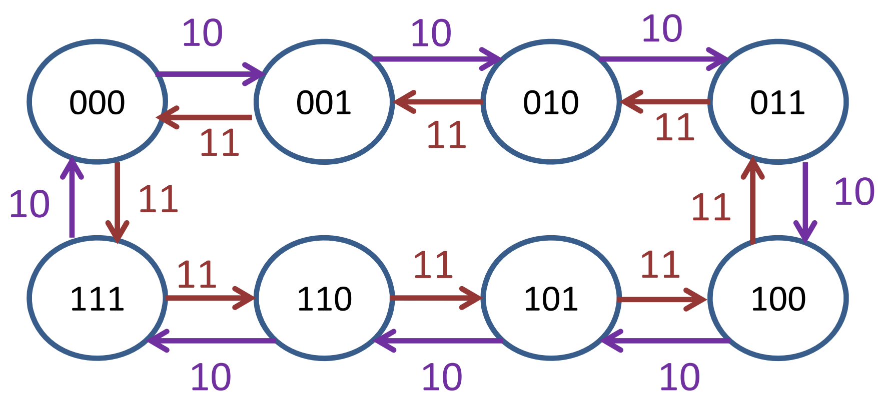


Sequences and States

- What about an up/down counter?
  - Counts up whenever *dn* input is 0
  - Counts down whenever *dn* input is 1
- Labeling all inputs by name on arrows can be cumbersome
- Hence, use a legend: in this case, 10 means *en*=1, *dn*=0
- Other input combinations result in no transition



Inputs: en, dn



Sequences and States

- The preceding dealt with counters, with a relatively fixed sequence
- We used the output as the state label
- The sequence was quite restricted
- How can we generalize this for more complex systems?
- *Finite State Machines (FSMs)* are a key modelling method for digital systems

Finite State Machines

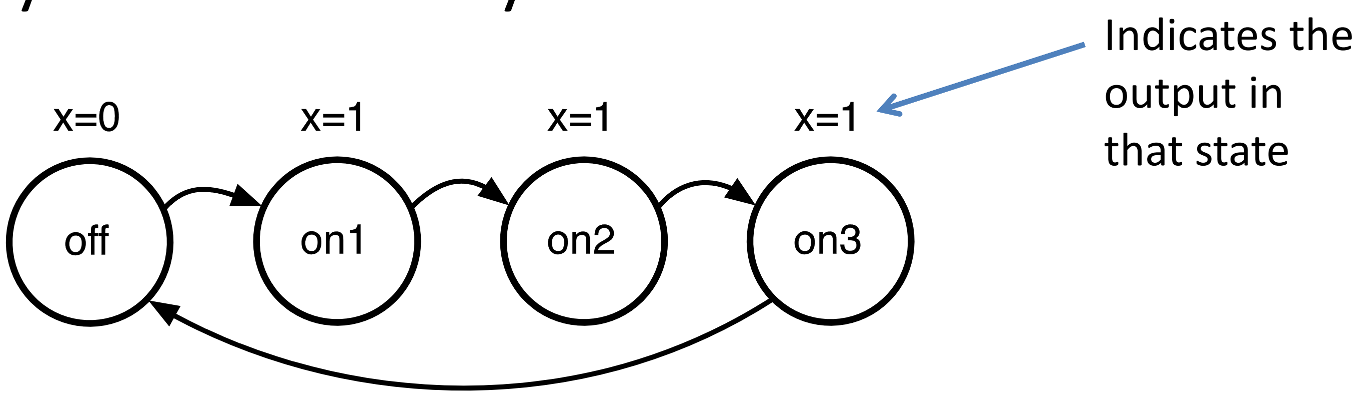
- A *finite state machine (FSM)* describes a system using a finite number of states and the associated transitions
- In synchronous design, an FSM is in one state for the duration of each clock cycle
- At every rising clock edge, the FSM may transition to another state
- Whether it transitions or not depends on the input values at that rising edge
- Once it makes the transition, it remains in that state for the next cycle

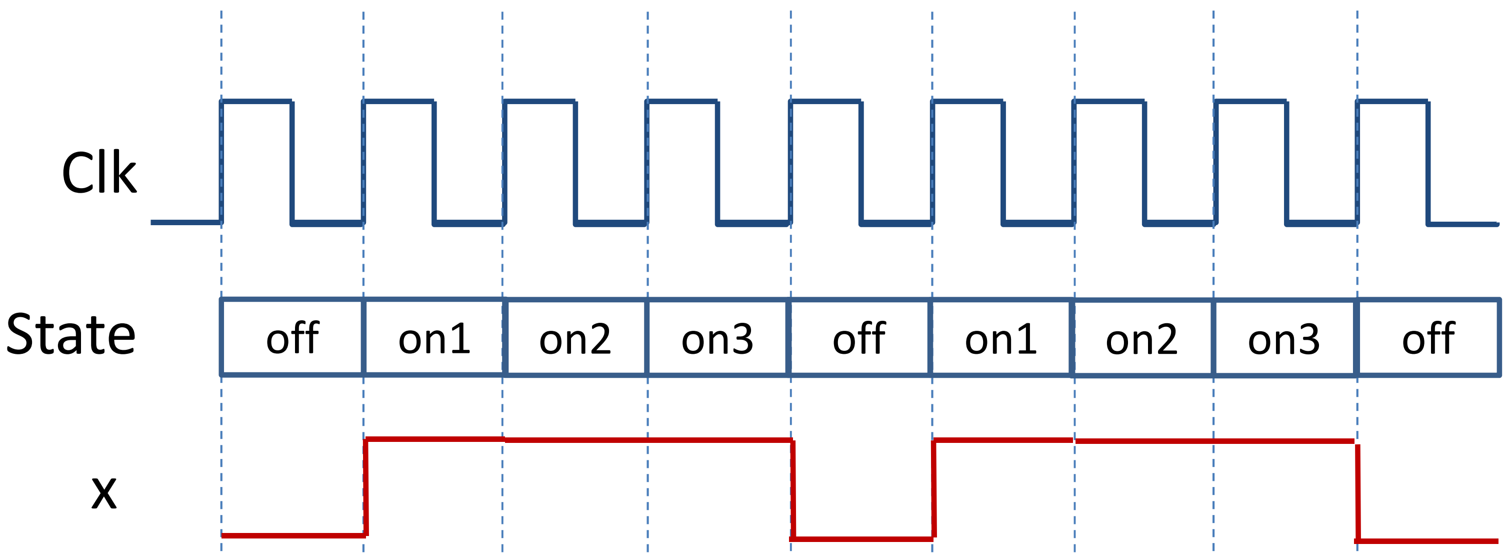
Finite State Machines

- We can label the states in a finite state machine with any meaningful name
  - In the counter sequences we saw, we just used the counter value, but often we want more meaningful information
- They are called “finite” because FSMs have a finite number of states (unlike some similar representations in other domains)
- “Machine” refers to a general object that can execute an abstract language

A Simple Example

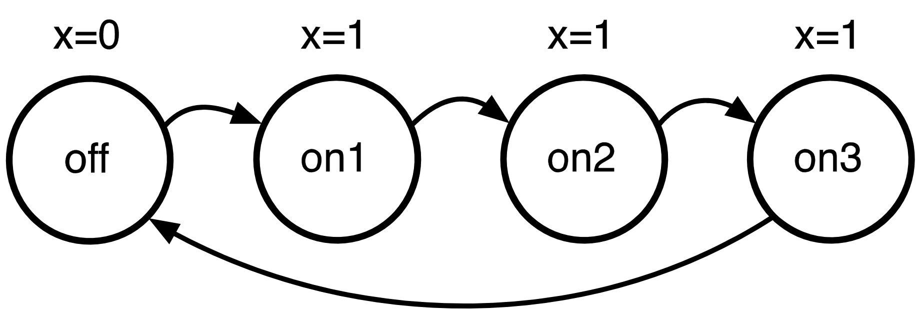
- An always-on state machine that produces three high output cycles followed by one low:





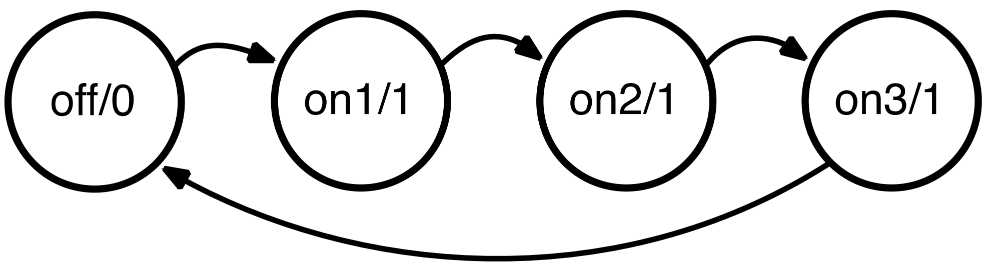
A Simple Example

- That FSM will always output high for three cycles, then low for one cycle
- The names of the states can be anything: using meaningful names helps us understand the behavior
- In this case, the output depends on what state the FSM is in: in *off*, it is low, in *on1*, *on2*, and *on3*, it is high



A Simple Example

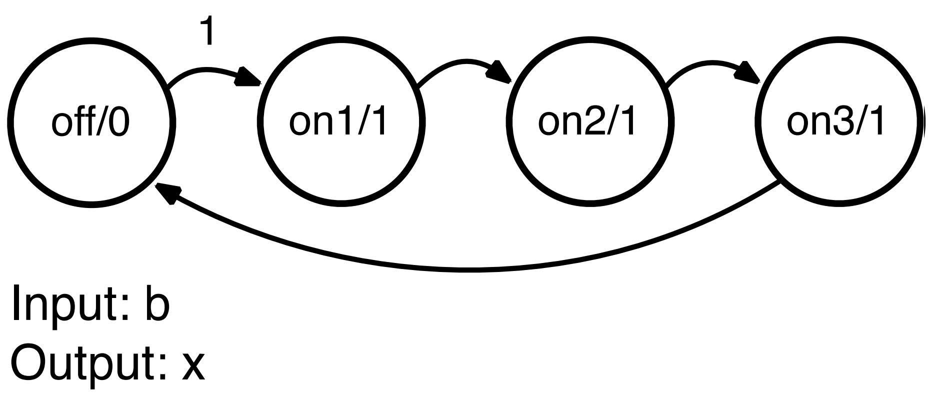
- We can indicate the outputs above, below, or beside the corresponding states
- It is also possible to put the outputs inside the state bubbles (can be on a separate line to state name)
- Include a legend so it is clear what the output signal is called



Output: x

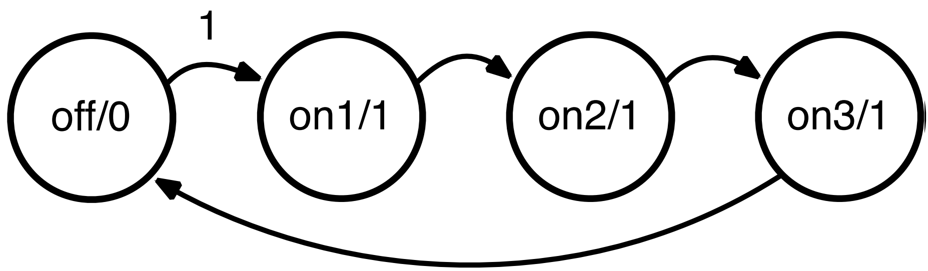
Adding Inputs

- What if we want the previous FSM to only output the three cycle high pulse when an input signal is high
- We showed that we can indicate input transition conditions on the arrows:

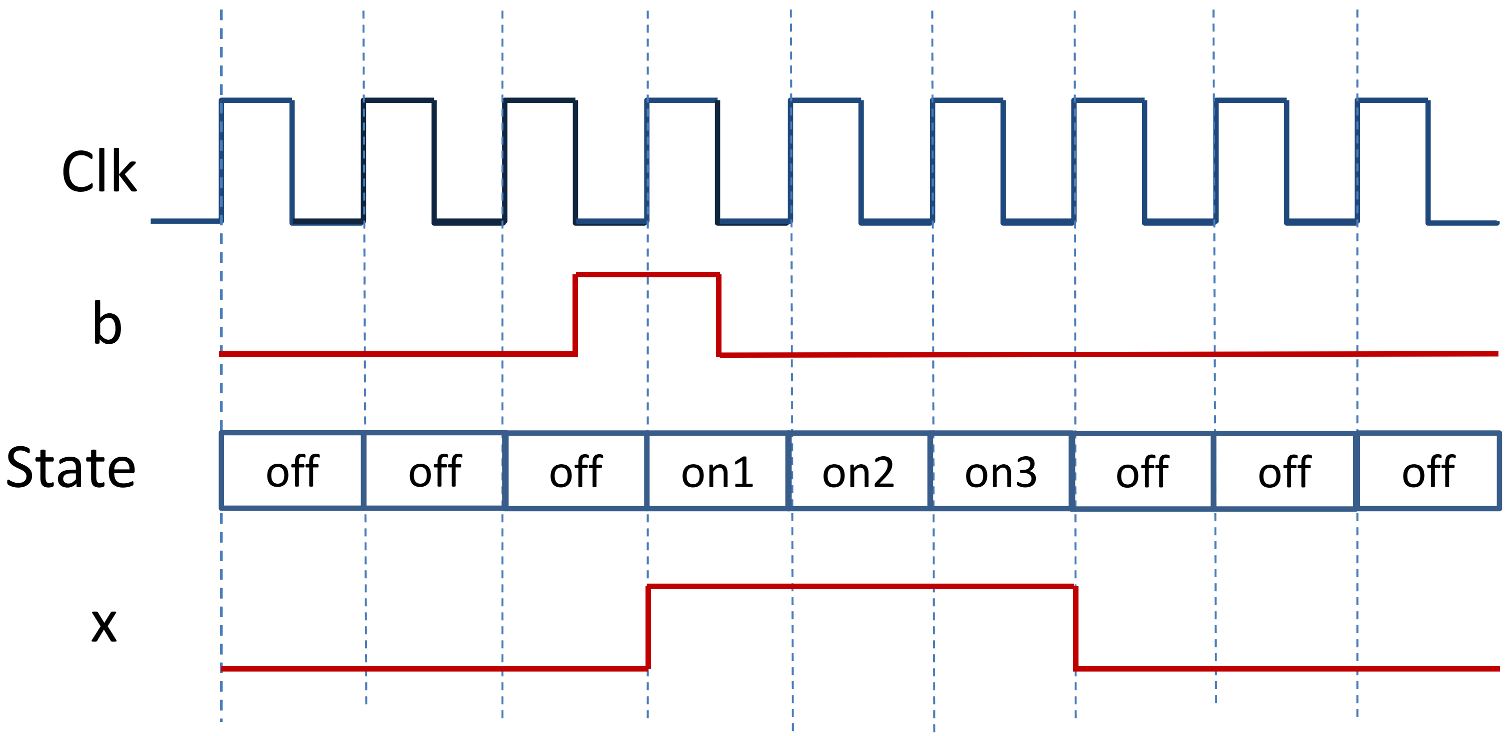


- Now, the FSM remains in state *off* until the *b* input goes high
- At that point it outputs a 1 as it passes through *on1*, *on2*, and *on3*, before returning to *off*

Adding Inputs

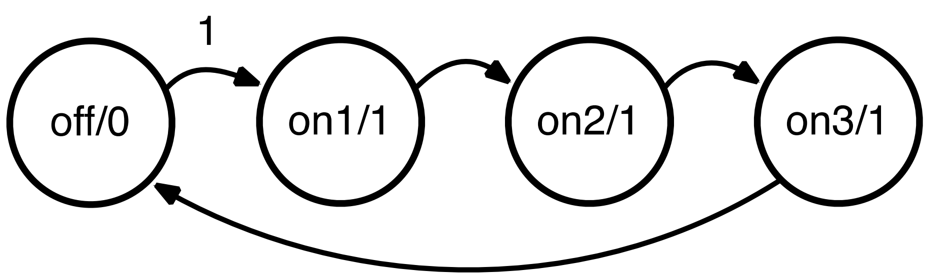


Input: b
Output: x



State Tables

- While diagrams can be useful, sometimes it helps to write the FSM information in a state transition table



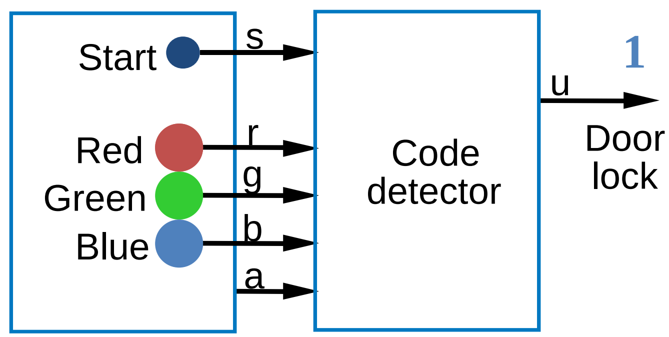
Input: b
Output: x

	Next State		
Current State	b=0	b=1	Output
off	off	on1	0
on1	on2	on2	1
on2	on3	on3	1
on3	off	off	1

Current State	b	Next State	Output x
off	0	off	0
off	1	on1	0
on1	0	on2	1
on1	1	on2	1
on2	0	on3	1
on2	1	on3	1
on3	0	off	1
on3	1	off	1

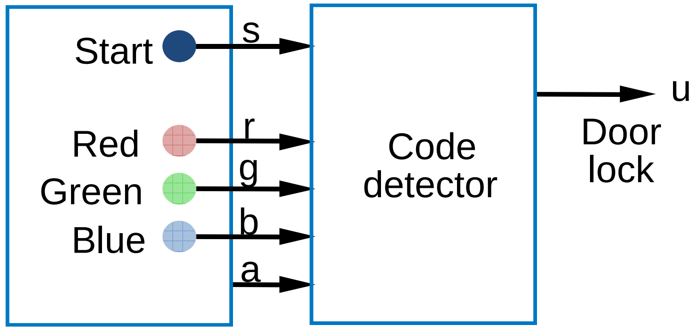
FSM Example

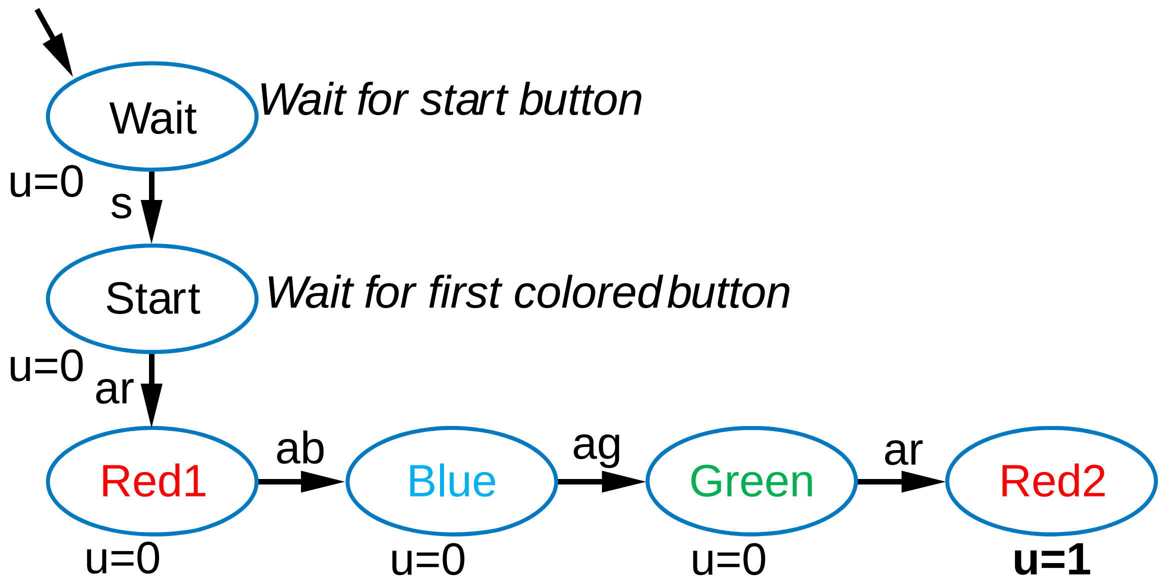
- Design a door lock that only unlocks (outputs u=1) when input buttons are pressed in a fixed sequence:
  - start, red, blue, green, red
- Inputs are s, r, g, b
- Input *a* indicates that a button has been pressed



FSM Example

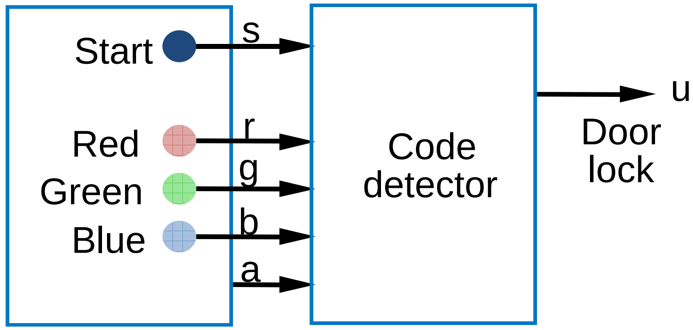
- We can capture the FSM behavior in a state diagram

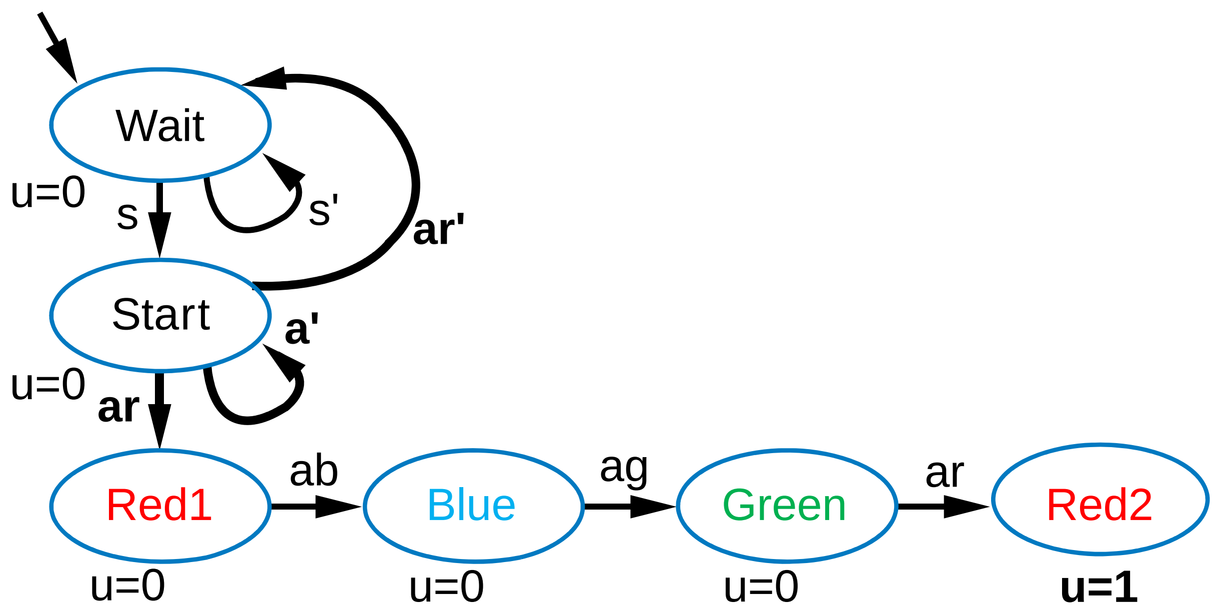




FSM Example

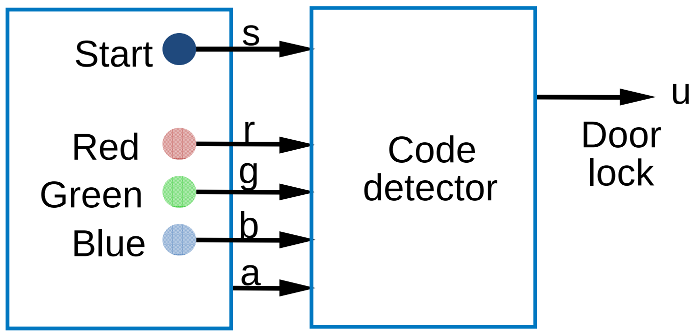
- We then need to add other transitions:

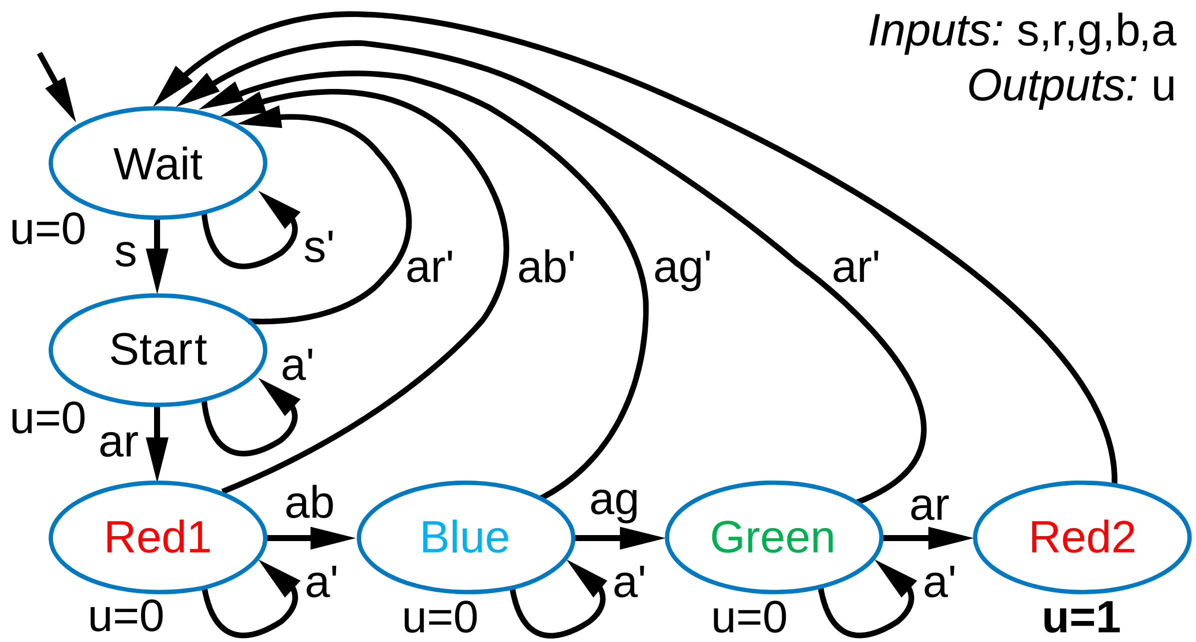




FSM Example

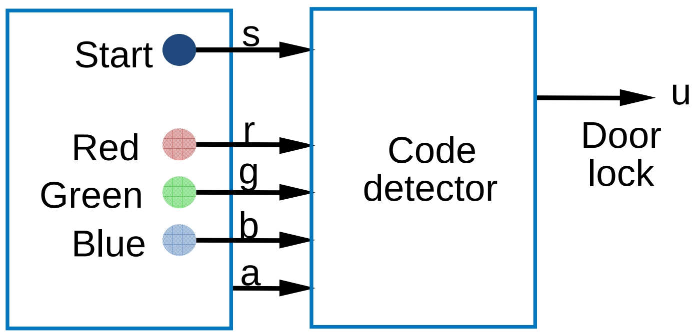
- We do the same for all states:

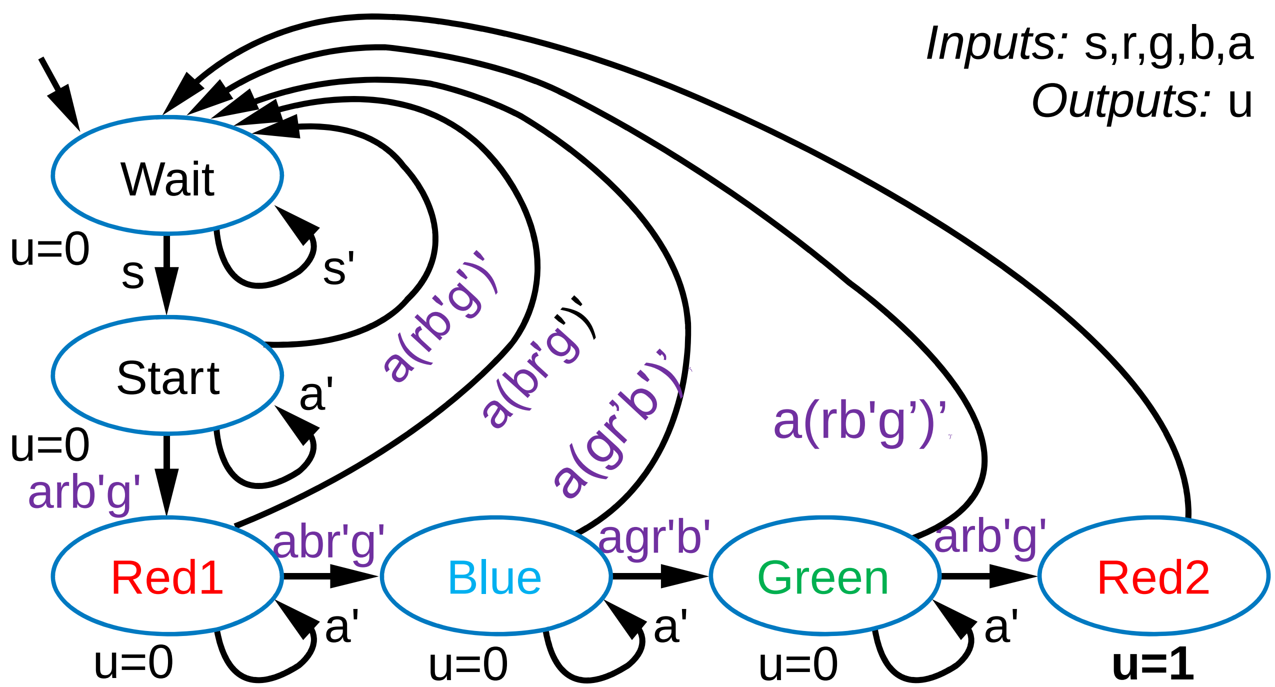




FSM Example

- However, we can see that if you press all buttons at the same time, the FSM will reach the end state, so we need to add further conditions:



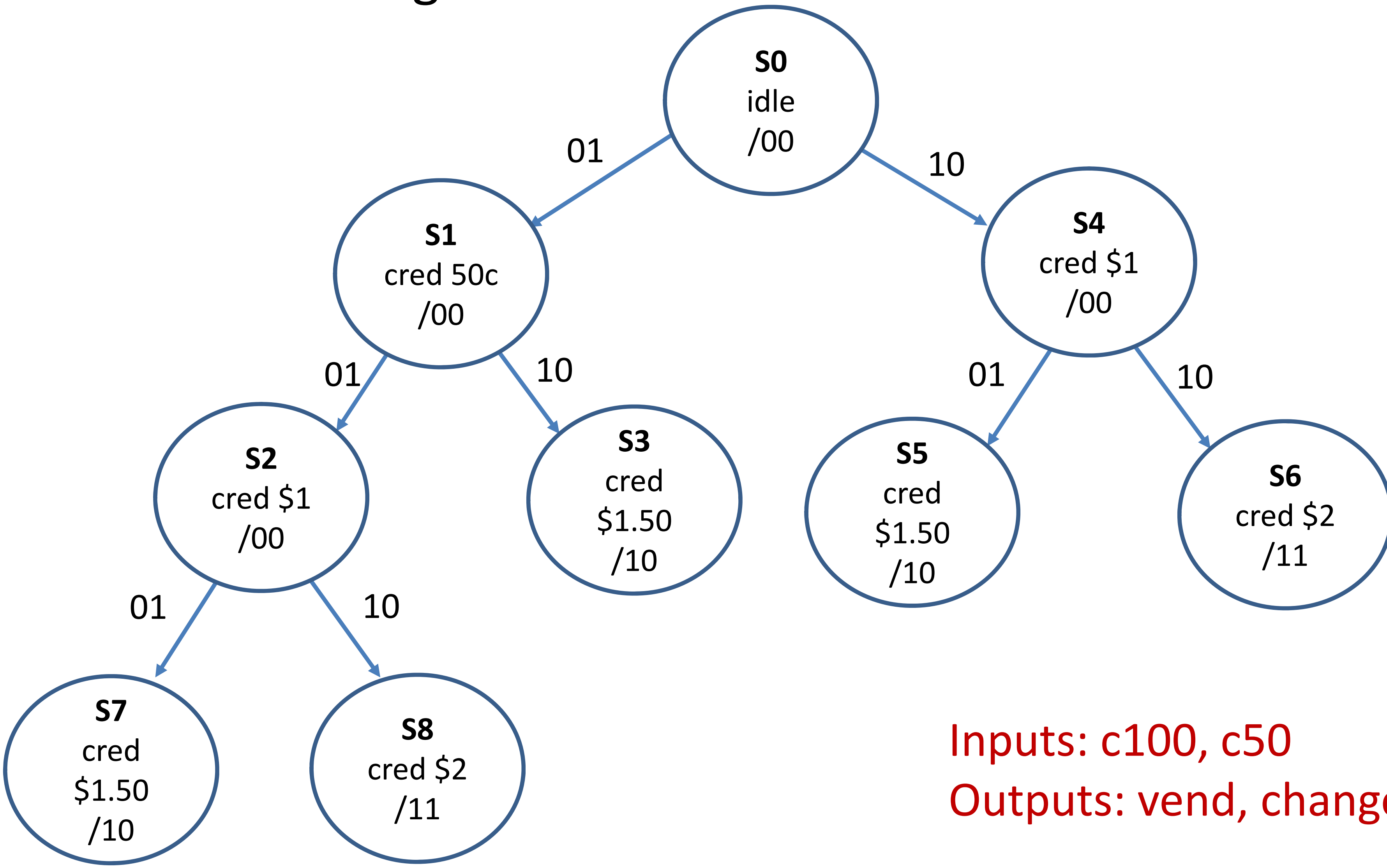


Another Example

- We want to design an FSM for a vending machine, that accepts only $1 and 50c coins, dispensing a drink that costs $1.50, and change if necessary
- This can be a circuit with two inputs, c100, for a $1 coin inserted and c50 for a 50c coin
- We assume only 1 input is ever high, and it tells us what coin was inserted
- The FSM has two outputs, vend, to release a drink, and change, to give 50c of change

Another Example

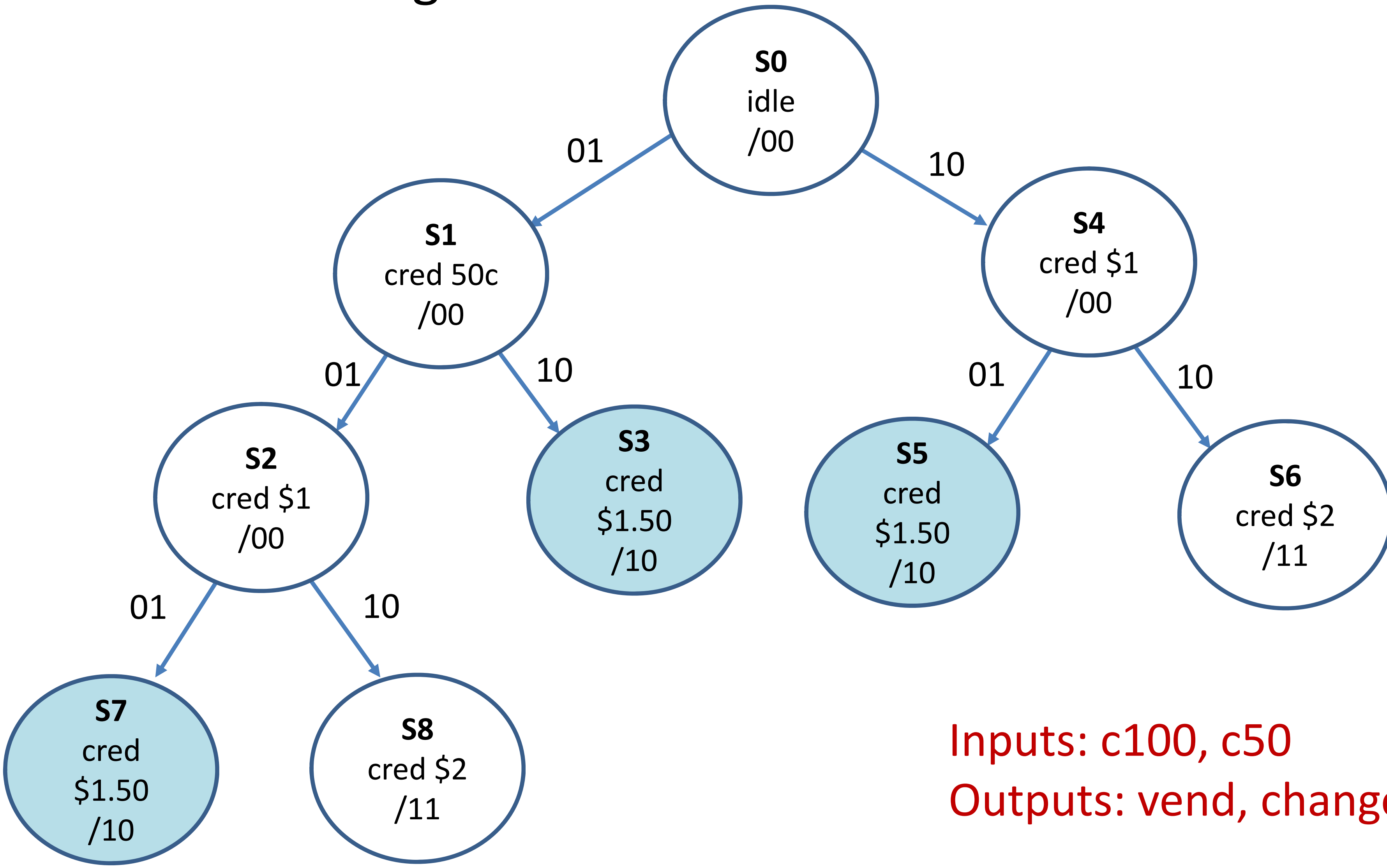
- The state diagram would look like this:



Inputs: c100, c50
Outputs: vend, change

Another Example

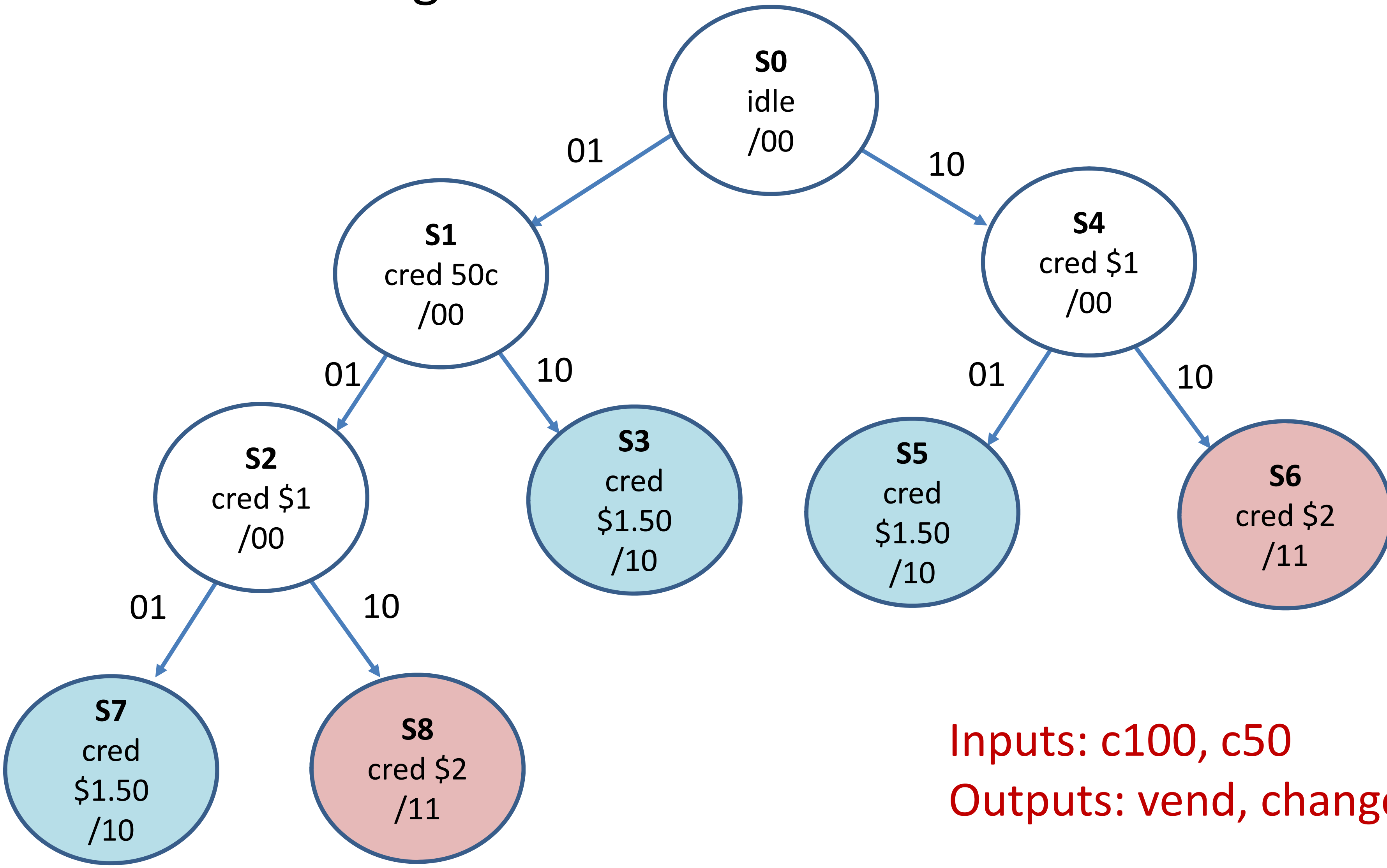
- The state diagram would look like this:



Inputs: c100, c50
Outputs: vend, change

Another Example

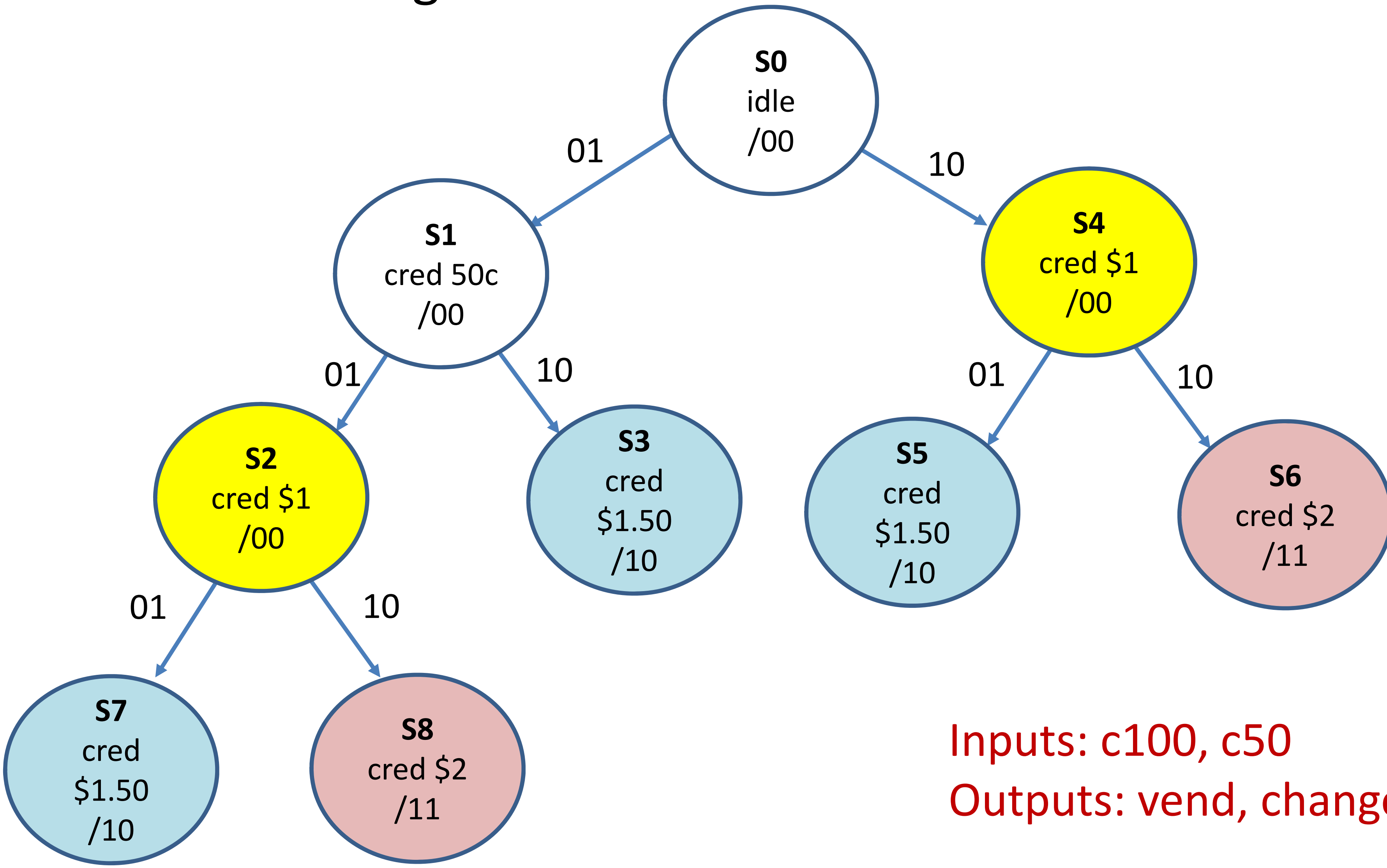
- The state diagram would look like this:



Inputs: c100, c50
Outputs: vend, change

Another Example

- The state diagram would look like this:



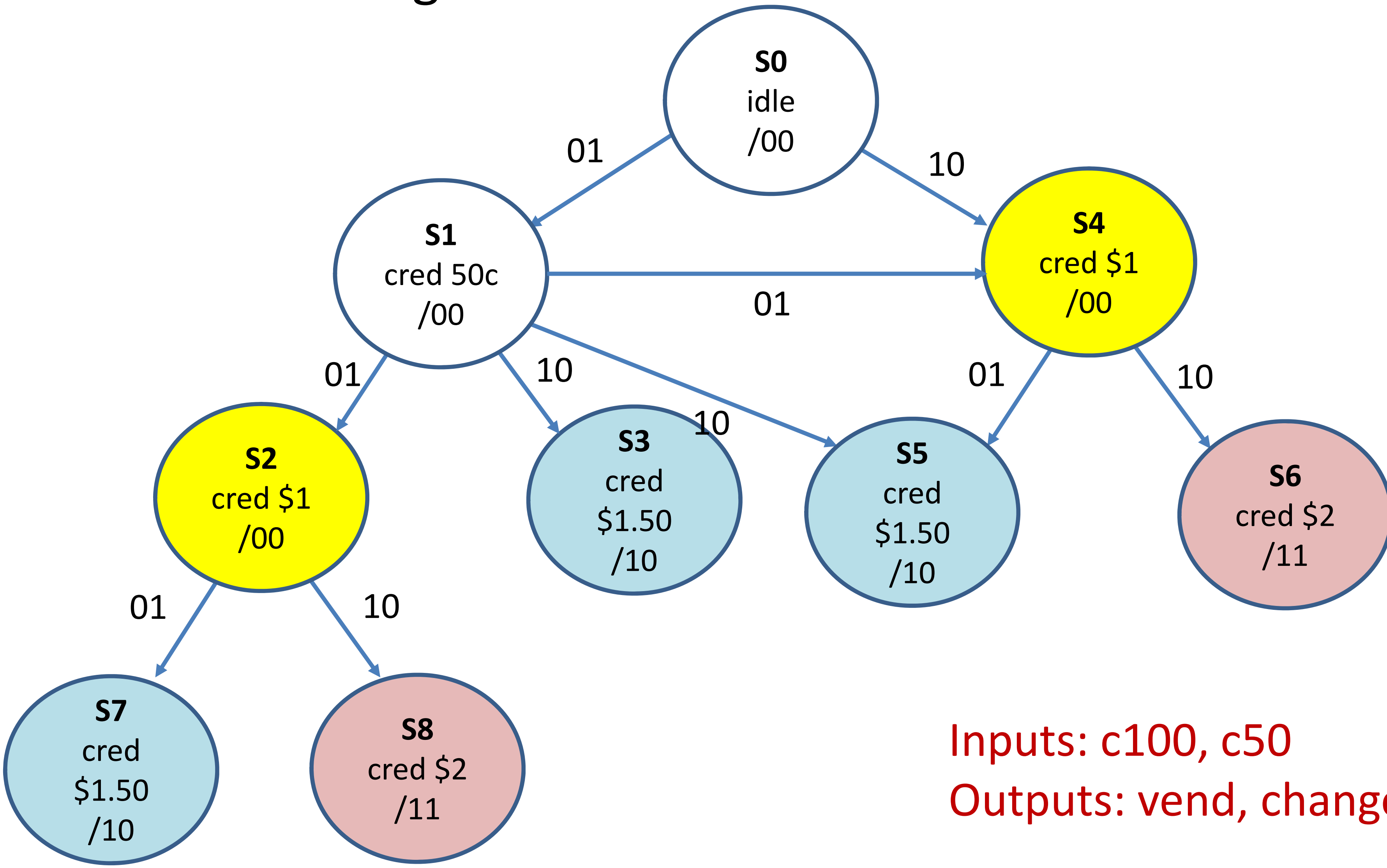
Inputs: c100, c50
Outputs: vend, change

Another Example

- Another strength of using FSMs, is that we can see where simplifications can be made in the behavior representation
- In the previous diagram, in states S2 and S4, the vending machine has $1 credit and will react identically to future inputs, hence we can merge them
- Similarly, states S3, S5, and S7 are the same. So are states S6 and S8

Another Example

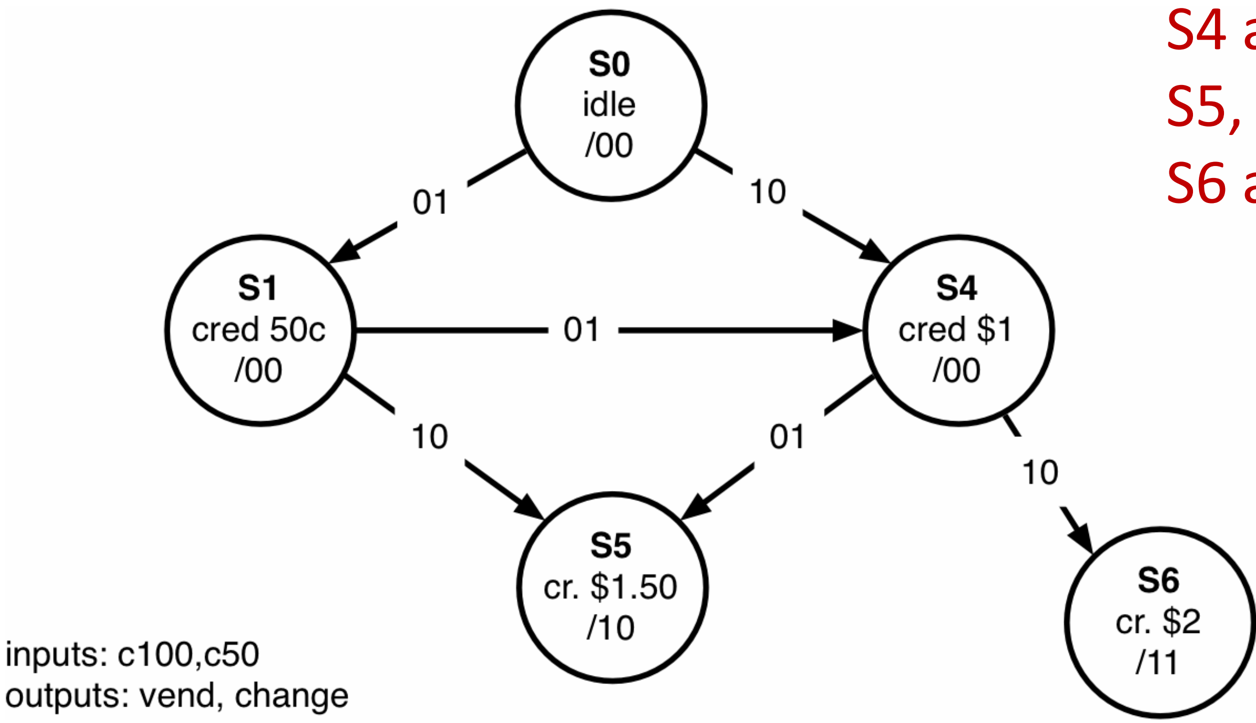
- The state diagram would look like this:



Inputs: c100, c50
Outputs: vend, change

Another Example

- Merging states is done simply by redrawing the arrows to the merged states:



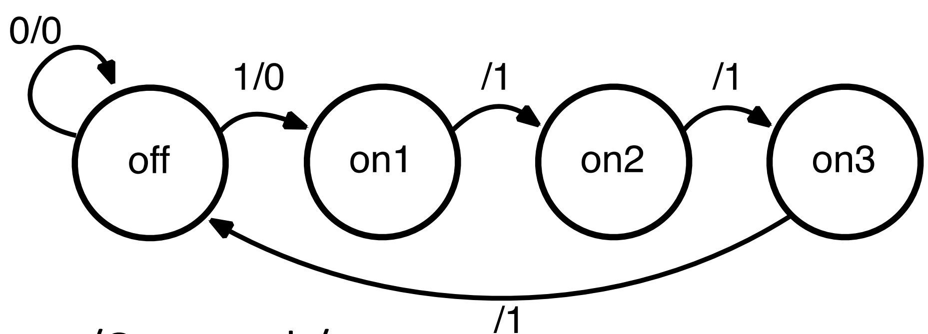
Merge:
S4 and S2;
S5, S3 and S7;
S6 and S8

Moore and Mealy Machines

- In all previous examples, the outputs depend only on the state, i.e. in state x, the output is always f
- These are called Moore Machines
- Another type of state machine is called a Mealy Machine
- In Mealy Machines, the output depends on the state and the current value of the inputs
  - Outputs can change mid-state, if the inputs change
- Mealy machines are a little harder to design and analyze, but can be more compact

Mealy Machines

- For Mealy machines, we can't draw the outputs inside the state circles, so we add them to the arrows instead, after a slash
- The output written on an arrow tells us what the output should be during the emitting state, when the input values match those on the same arrow
- A previous example with outputs on arrows:



Inputs/Outputs: b/x