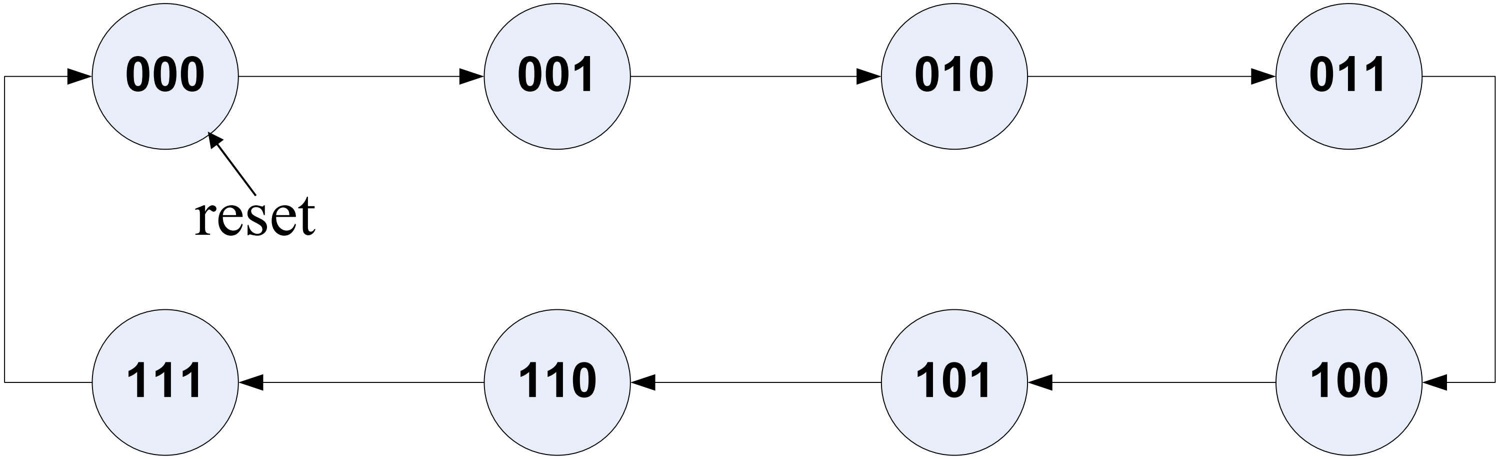
Topic 10

Finite State Machine

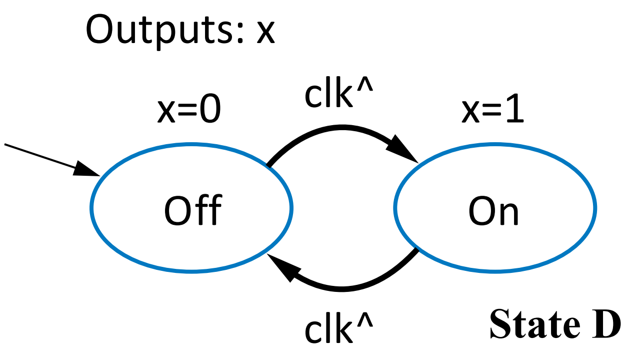
Recall Synchronous Binary Counter

- A circuit that changes its value (state) at every clock edge
- The counting sequence describes the behavior of the circuit

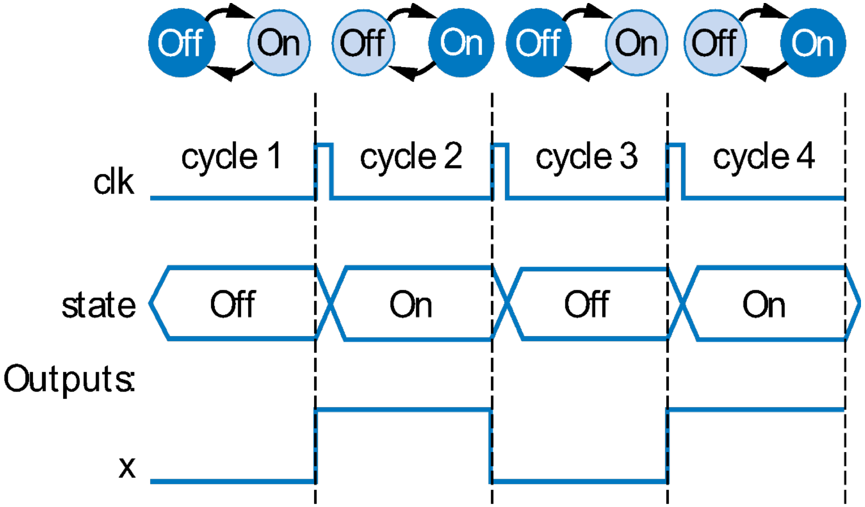


Describing Behavior of Sequential Circuit

- Finite-State Machine (FSM)
  - A way to describe **desired behavior** of a sequential circuit
  - Consists of a set of states, transitions between states, and maybe inputs and outputs
    - ***present state***: currently happening
    - ***next state***: next to happen
  - Example: Toggle output x every clock cycle
    - Two states: “Off” and “On”
    - Corresponding outputs: x=0 or 1
    - No input
    - Transition from Off to On, or On to Off, on rising clock edge
    - Arrow with no starting state points to initial state



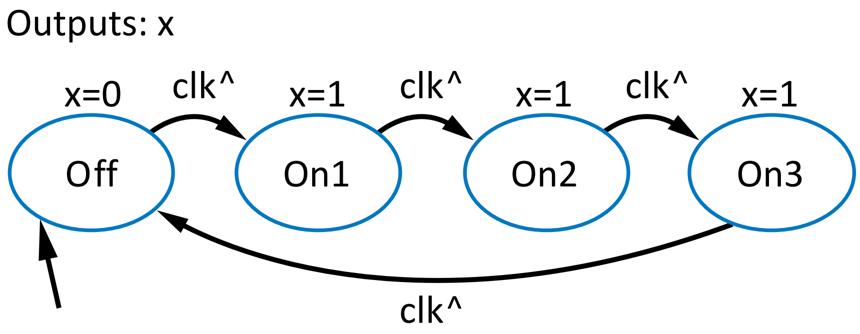
State Diagram

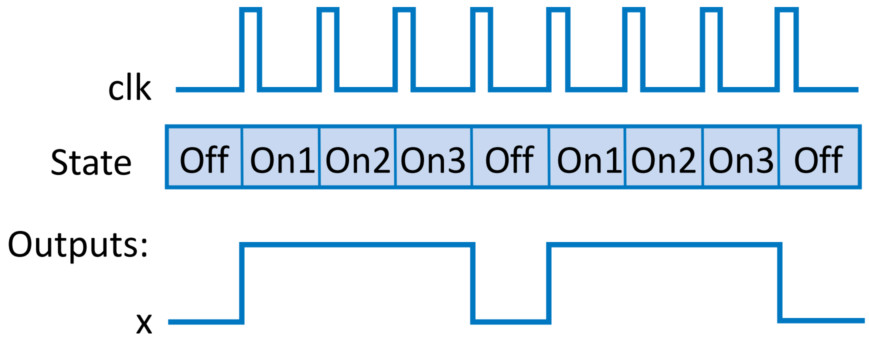


Example: Output Special Pattern

- Want a circuit to output

  0, 1, 1, 1, 0, 1, 1, 1, ...
  - One bit at a time
  - Each bit for one clock cycle
- Can be described as FSM
  - Four states
    - Each state corresponds to an output, 0, 1, 1, 1
    - Then repeat
  - Transition on rising clock edge to next state

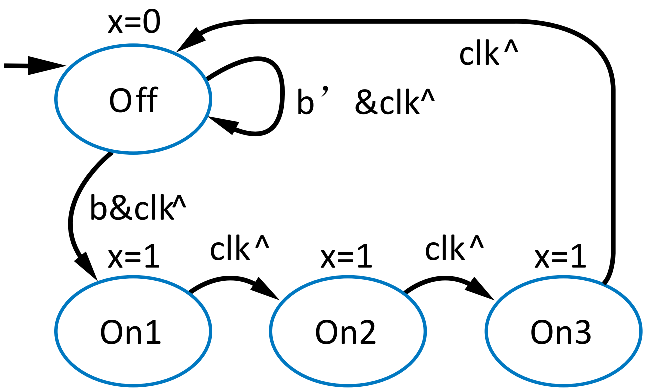


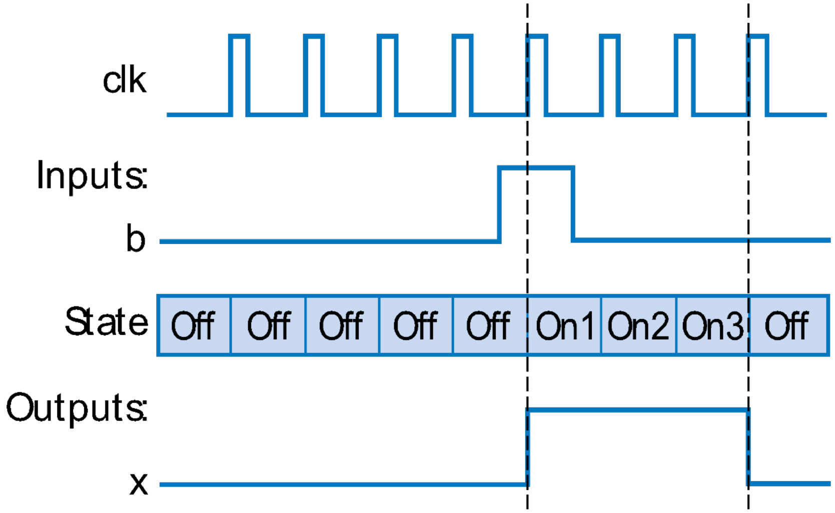


Example: FSM with Input

- b is a push button, output x to stay on for exactly 3 clock cycles no matter how long b is pushed
- Wait in "Off" state while b is 0 (b')
- When b is 1 (and rising clock edge), transition to On1
  - Sets x=1
  - On next two clock edges, transition to On2, then On3, which also set x=1
- So x=1 for three cycles after button pressed
- Potential issue: if button b stays on, what will happen?

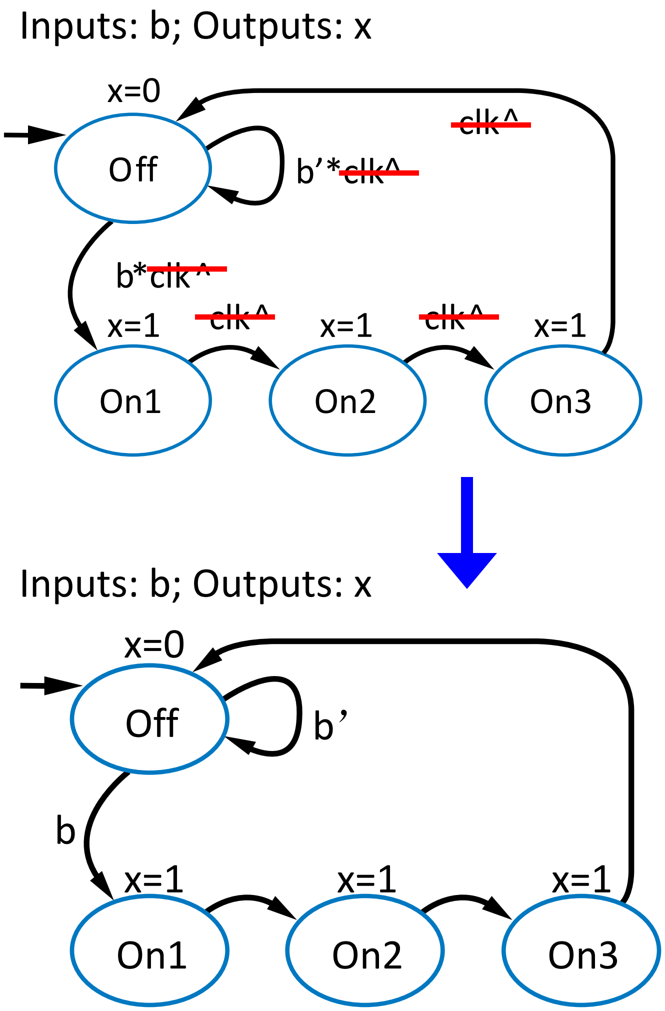
Inputs: b; Outputs: x





State Diagram Simplification: Clock Implicit

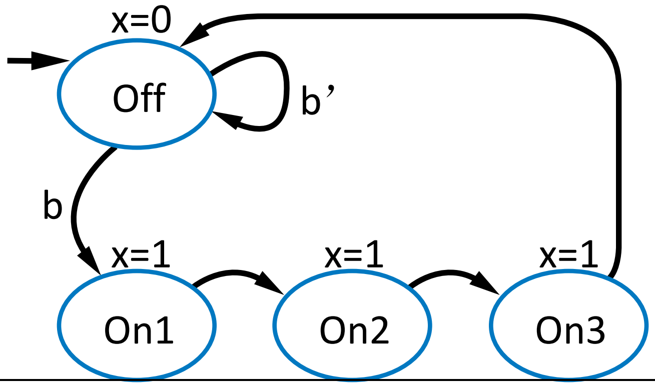
- ***Synchronous*** FSMs – FSM behaviors synchronized to active edge of clock
  - Asynchronous FSMs -- less common
- Make implicit – all state transitions are triggered by rising edge of clock
- Make implicit – Unlabeled path means transition is triggered only by clock, inputs don't matter



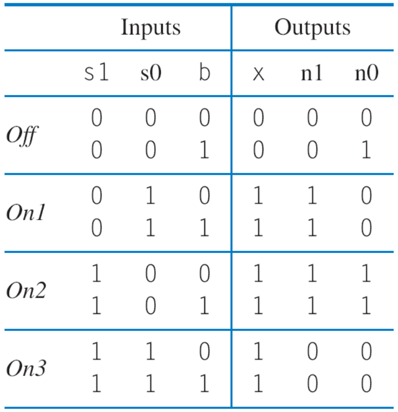
FSM Definition

- FSM consists of
  - Set of states
    - Ex: {Off, On1, On2, On3}
  - Set of inputs, set of outputs
    - Ex: Inputs: {b}, Outputs: {x}
  - Initial state
    - Ex: "Off"
  - Set of transitions
    - Describes next states
    - Ex: Has 5 transitions
  - Set of actions (outputs)
    - Sets outputs while in states
    - Ex: x=0, x=1, x=1, and x=1

Inputs: b; Outputs: x



FSM can be represented graphically, known as ***state diagram***

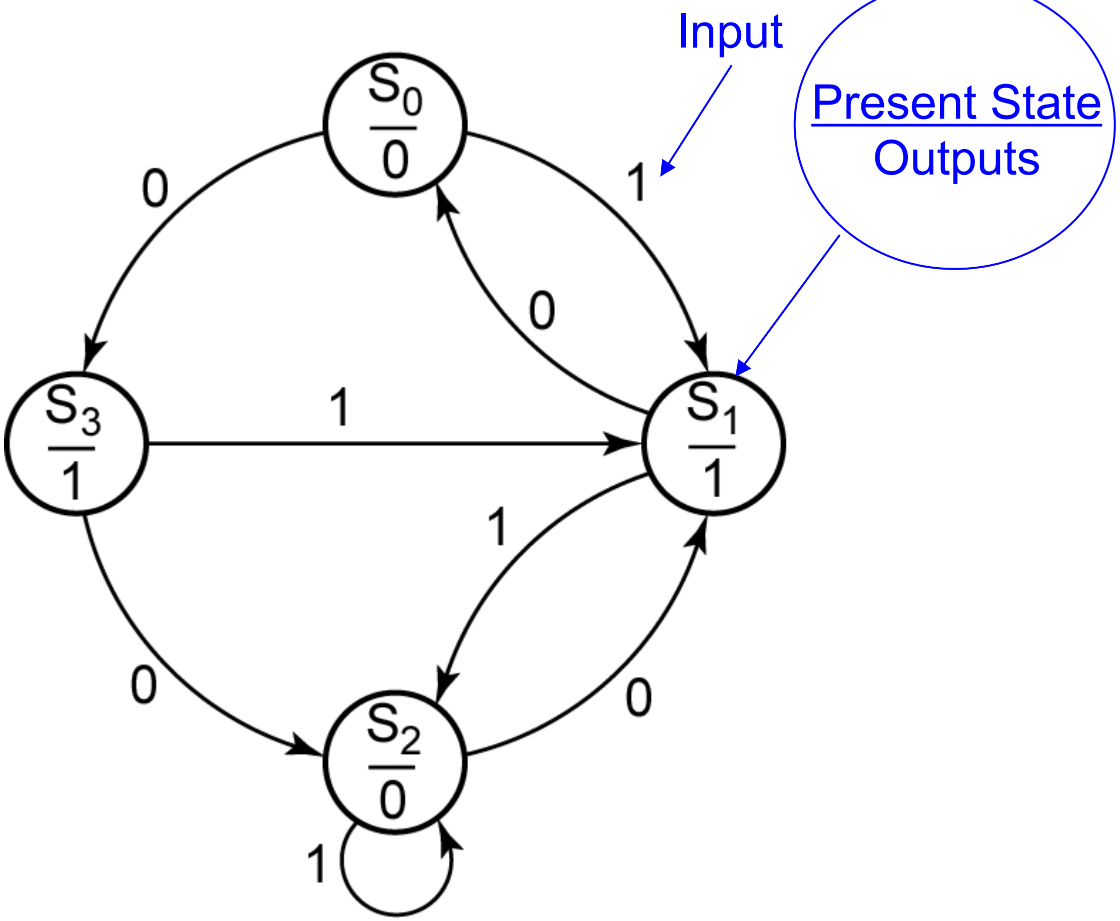


	Inputs			Outputs		
	s1	s0	b	x	n1	n0
Off	0	0	0	0	0	0
	0	0	1	0	0	1
On1	0	1	0	1	1	0
	0	1	1	1	1	0
On2	1	0	0	1	1	1
	1	0	1	1	1	1
On3	1	1	0	1	0	0
	1	1	1	1	0	0

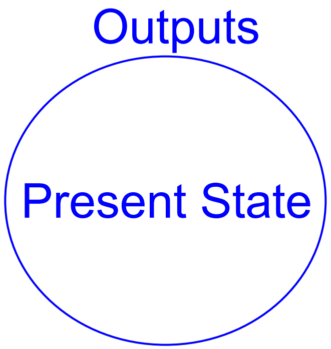
FSM can also be represented in tabular form, known as ***state table***

State Diagram and State Table

State Diagram



Or

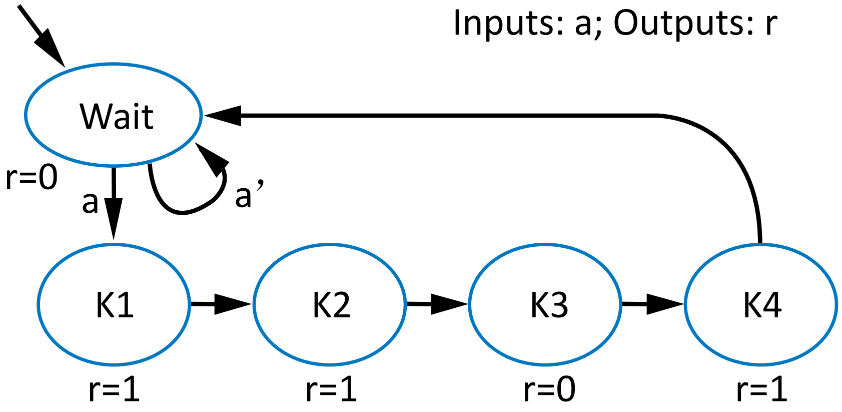


State Table

In	P.S.	N.S.	Out
0	S0	S3	0
1	S0	S1	0
0	S1	S0	1
1	S1	S2	1
0	S2	S1	0
1	S2	S2	0
0	S3	S2	1
1	S3	S1	1

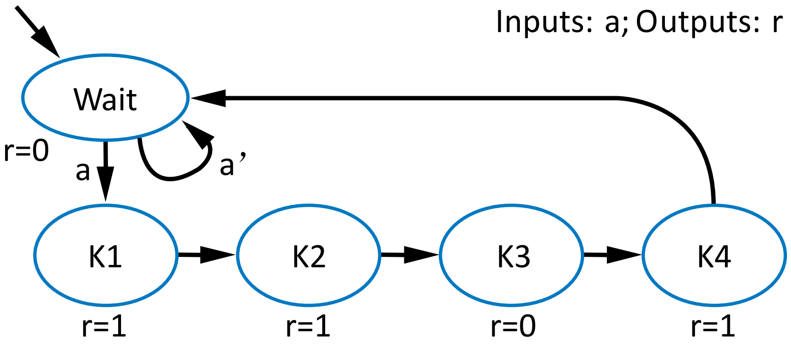
Example: Secure Car Key

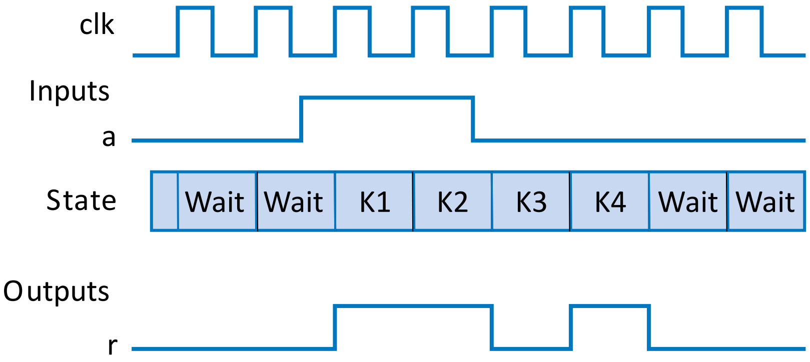
- All new car keys contain a tiny computer chip
  - When car starts, car's computer (under engine hood) requests identifier from key
  - Key transmits identifier
    - If not, computer shuts off car
- FSM
  - Wait until computer requests ID (a=1)
  - Transmit ID (in this case, 1101)

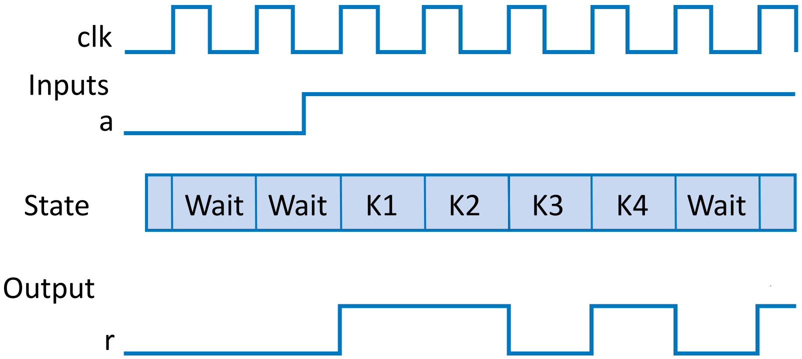


FSM Example: Secure Car Key (cont.)

- Timing diagrams show states and output values for different input waveforms





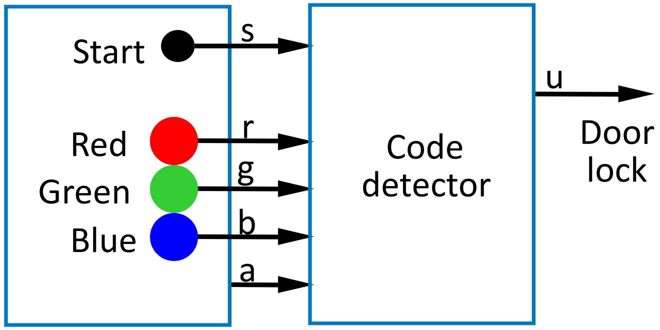


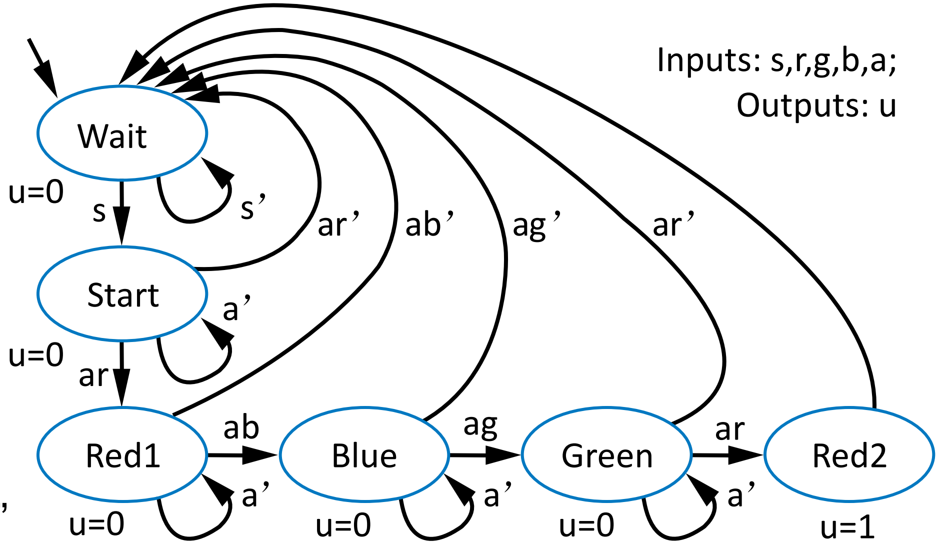
Example: Digital Lock

- Unlock door (u=1) only when buttons pressed in sequence:

  - start, then red, blue, green, red

- Input signals: *s, r, g, b*
- Input signal *a* indicates that some color button pressed, *lasts for only one clock cycle with each button press*
- FSM
  - Wait for start (s=1) in "Wait"
  - Once started, go to "Start", then
    - If see red, go to "Red1"
    - Then, if see blue, go to "Blue"
    - Then, if see green, go to "Green"
    - Then, if see red, go to "Red2", and u=1
    - Wrong button at any step, return to "Wait", without opening door



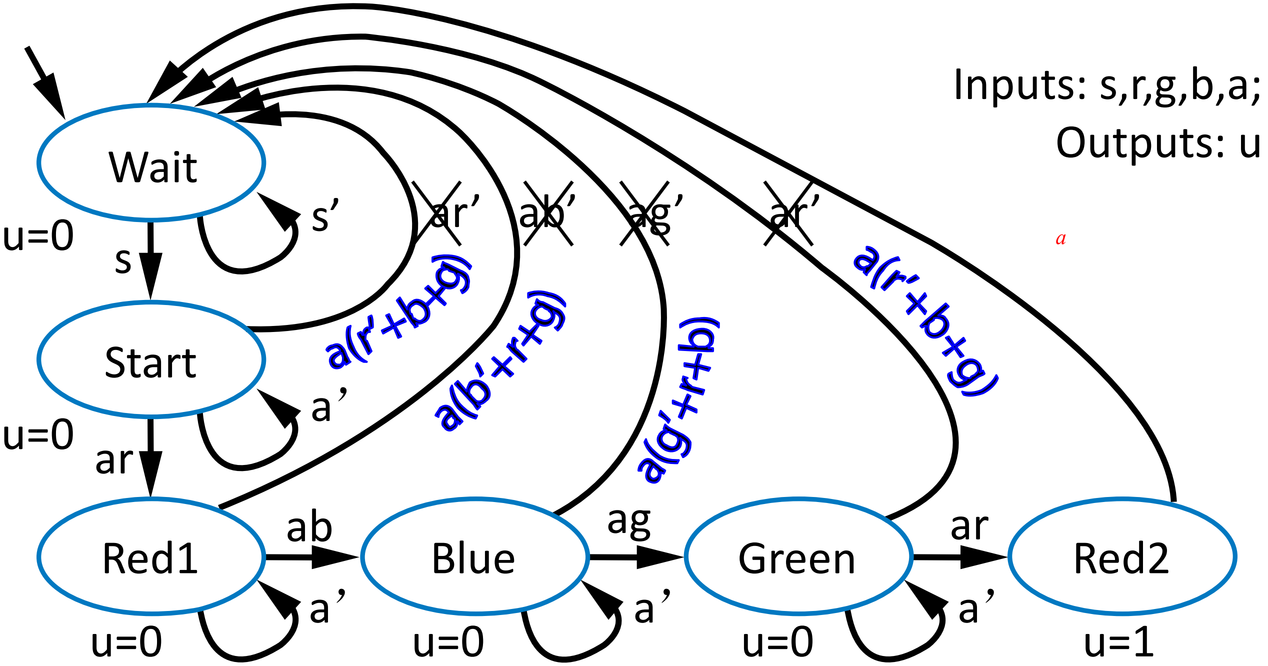


Inputs: s,r,g,b,a;
Outputs: u

Q: Can you trick this FSM to open the door, without knowing the code?

A: Yes, hold all buttons simultaneously

Improve FSM for Code Detector

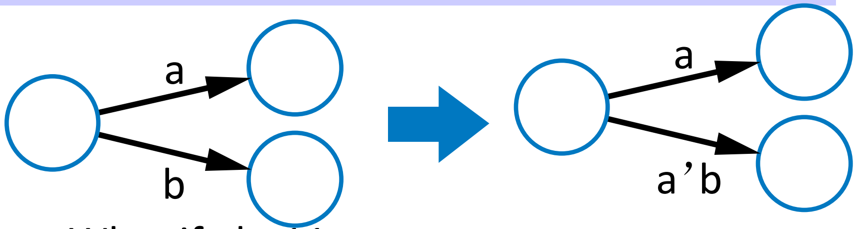


Inputs: s,r,g,b,a;
Outputs: u

- New transition conditions detect if wrong button pressed, returns to "Wait"

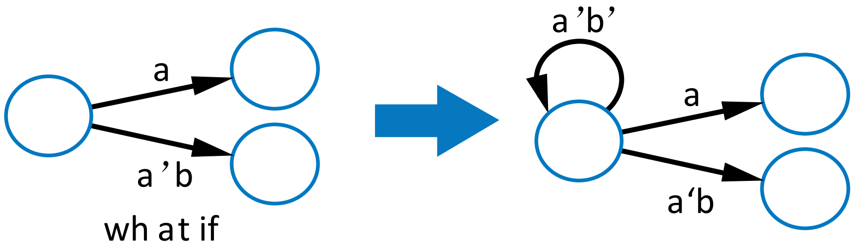
Common State Transition Property

- *Only* one condition should be true, among all transitions leaving a state



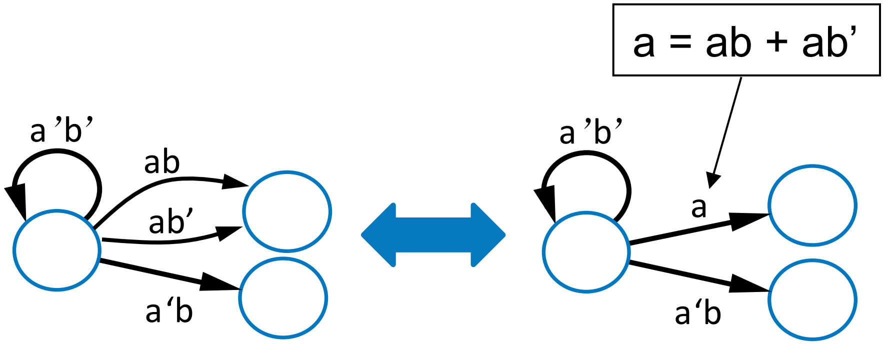
What if ab=11 next state?

- *One* condition must be true
  - For any input combination



wh at if ab=00?

- All conditions must be considered when leaving a state



Pitfall is Common

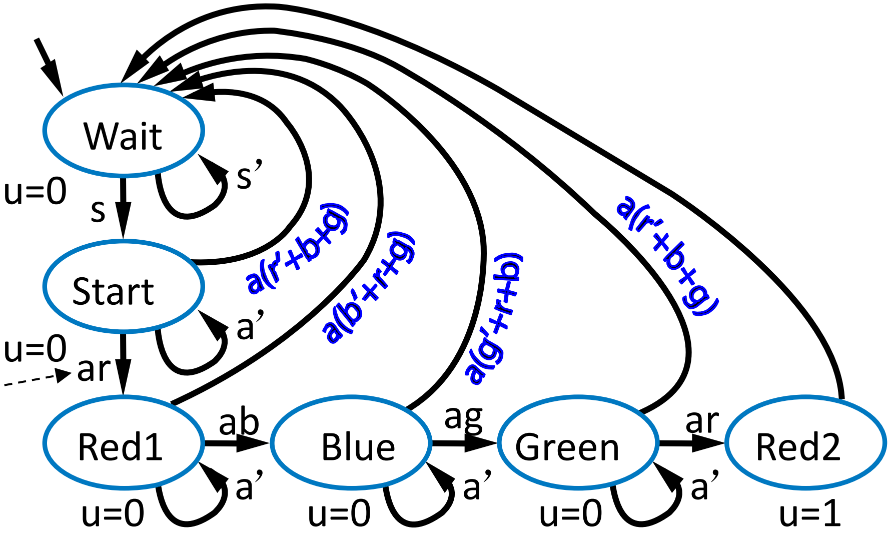
- Recall code detector FSM
  - Do the transitions obey the two required transition properties?

    NO!

  - How would it go wrong?

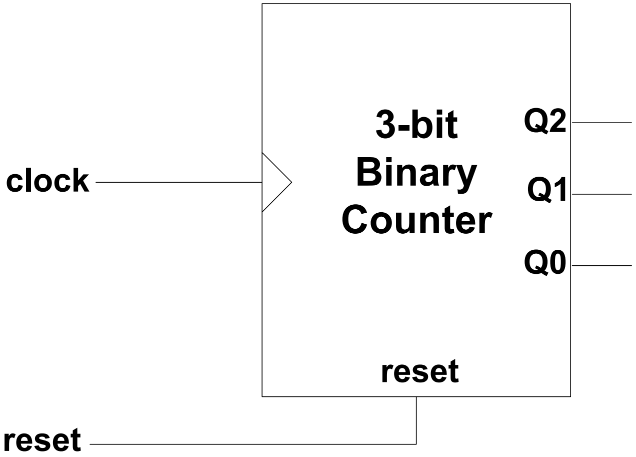
    E.g. arbg = 1111
    How to solve?

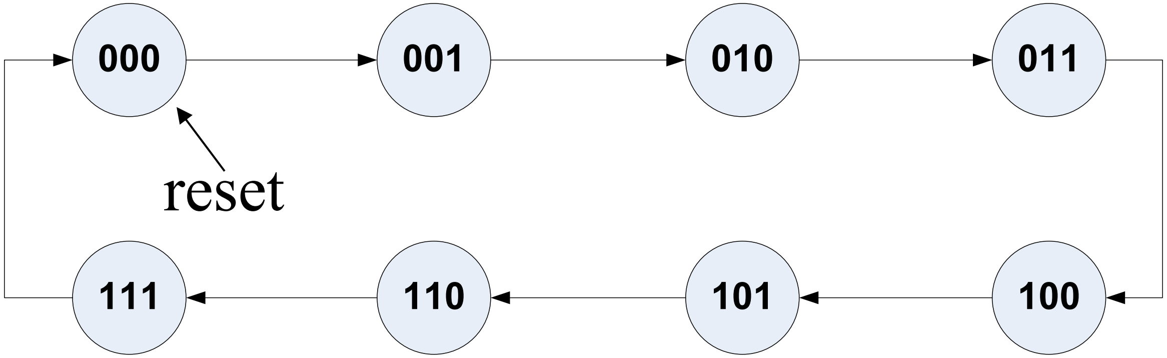
Answer: **ar** should be **arb'g'**
(likewise for ab, ag, ar)



Implementation of FSM
Example: Synchronous Binary Counter

- FSM that counts binary numbers – counter
- 3-bit counter has 3 outputs: Q2, Q1, Q0
- Counts 2^3=8 numbers (8 states)
- Counts a number at every clock edge (state transition at every clock edge)





State diagram of 3-bit synchronous up counter

Synchronous Binary Counter Design

- An FSM (without external inputs)
  - State Table

Present State			Next State		
Q2	Q1	Q0	$Q2^{+}$	$Q1^{+}$	$Q0^{+}$
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

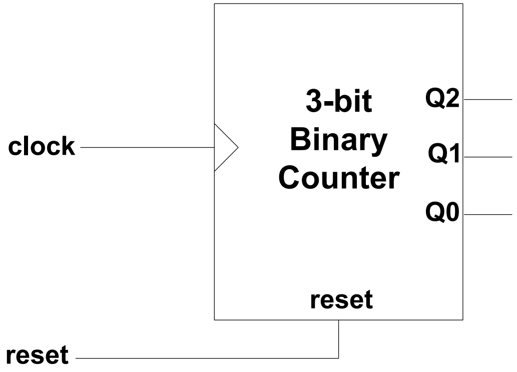
Counter Implemented with D Flip-Flop

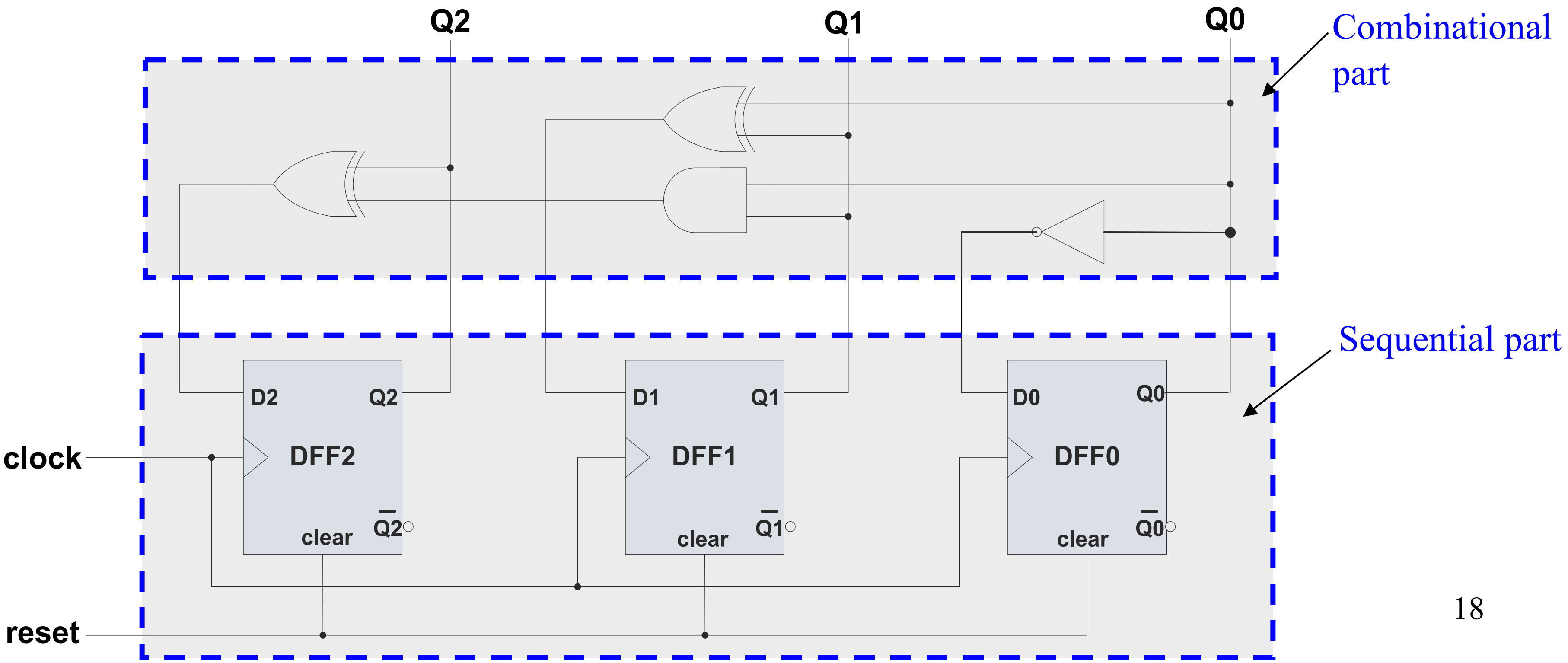
- Use D flip flops to hold values：**Q^+ = D upon active edge**
- Next state equations

Present State			Next State			D flip flop input		
Q2	Q1	Q0	$Q2^+$	$Q1^+$	$Q0^+$	D2	D1	D0
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	0
0	1	0	0	1	1	0	1	1
0	1	1	1	0	0	1	0	0
1	0	0	1	0	1	1	0	1
1	0	1	1	1	0	1	1	0
1	1	0	1	1	1	1	1	1
1	1	1	0	0	0	0	0	0

Synchronous Binary Counter with D Flip-Flop

- 3-bit binary counter by D FF



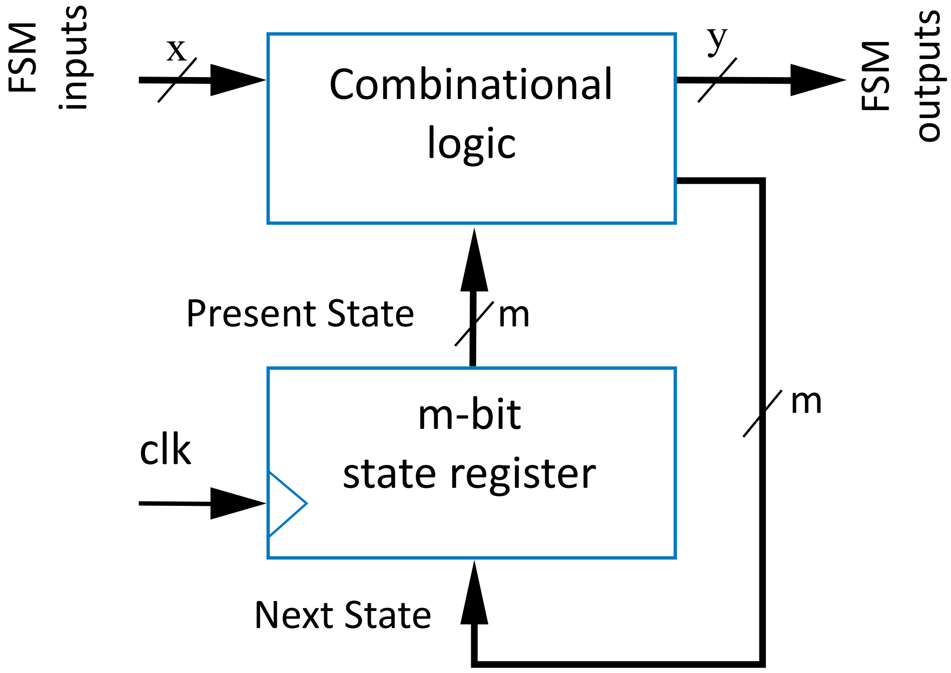


Combinational part

Sequential part

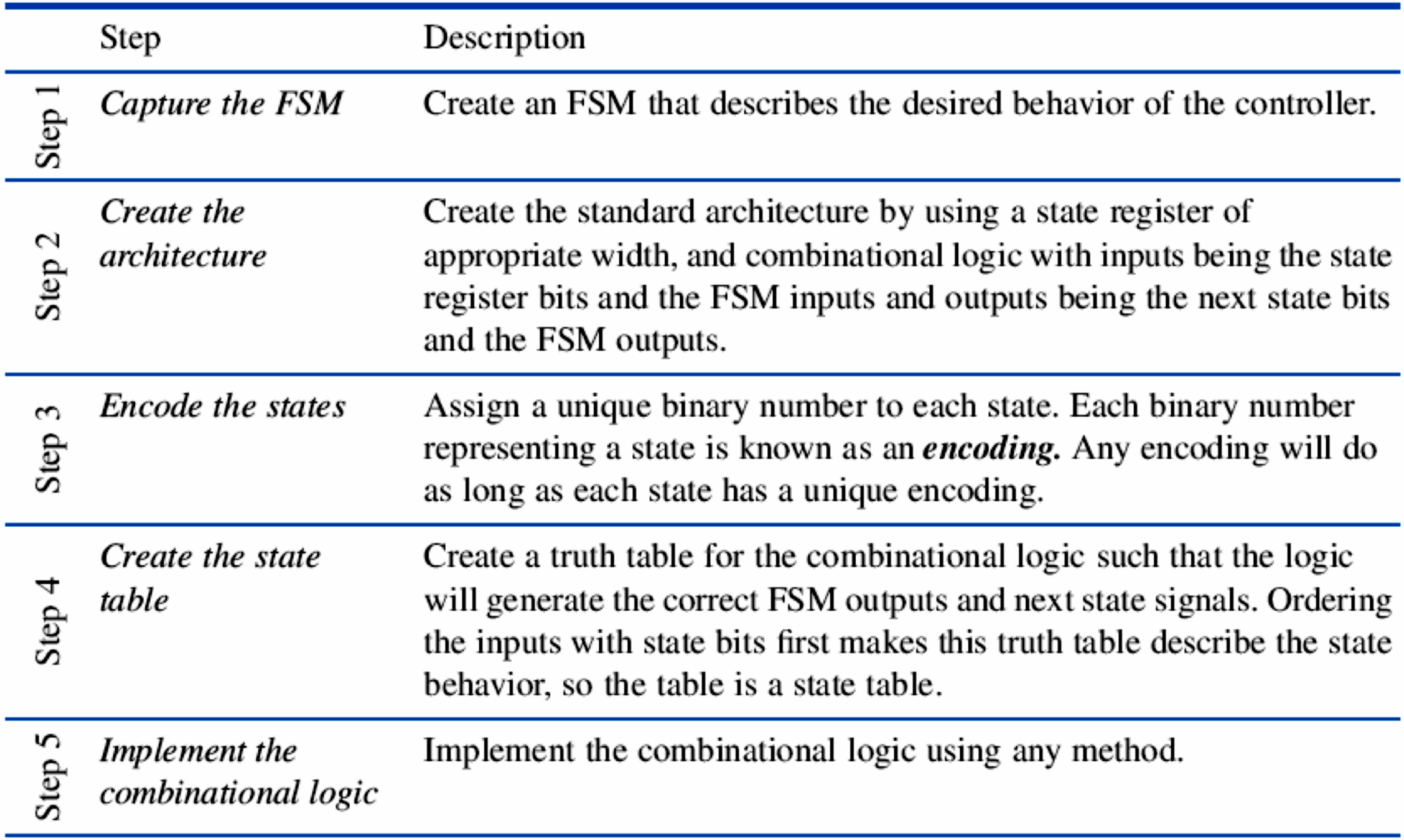
Standard FSM Architecture

- How to design sequential circuit?
  - Design as FSM
  - Use standard architecture
    - **State register** -- to store the present state
    - **Combinational logic** -- to compute outputs, and next state
  - Known as ***controller***



FSM (Controller) Design

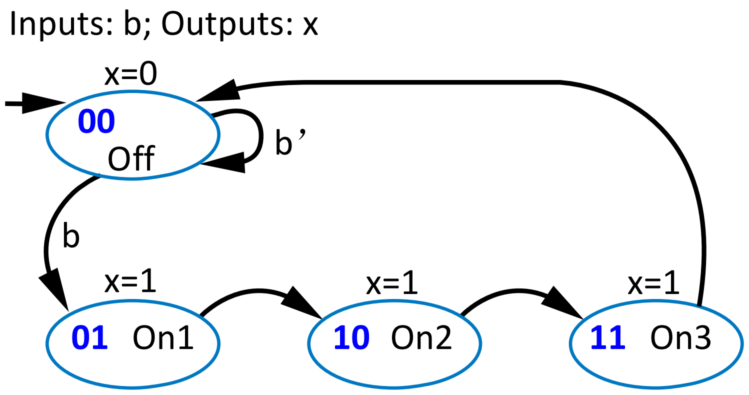
- Five step FSM design process

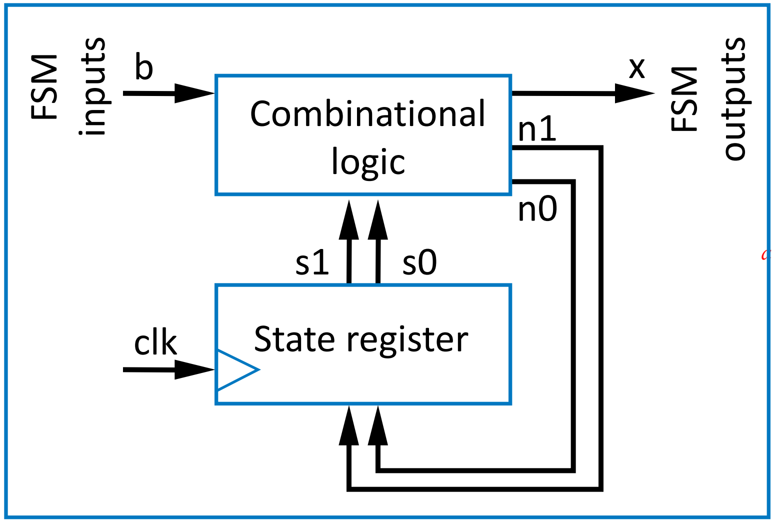


	Step	Description
Step 1	*Capture the FSM*	Create an FSM that describes the desired behavior of the controller.
Step 2	*Create the architecture*	Create the standard architecture by using a state register of appropriate width, and combinational logic with inputs being the state register bits and the FSM inputs and outputs being the next state bits and the FSM outputs.
Step 3	*Encode the states*	Assign a unique binary number to each state. Each binary number representing a state is known as an ***encoding.*** Any encoding will do as long as each state has a unique encoding.
Step 4	*Create the state table*	Create a truth table for the combinational logic such that the logic will generate the correct FSM outputs and next state signals. Ordering the inputs with state bits first makes this truth table describe the state behavior, so the table is a state table.
Step 5	*Implement the combinational logic*	Implement the combinational logic using any method.

FSM Design Example: Push Button

- Step 1: Capture the FSM
  - Already done
- Step 2: Create architecture
  - 2-bit state register (for 4 states)
  - Input b, output x
  - Present state signals (s1, s0)
  - Next state signals (n1, n0)
- Step 3: Encode the states
  - Any encoding with unique representation for each state will work





FSM Design Example: Push Button (cont.)

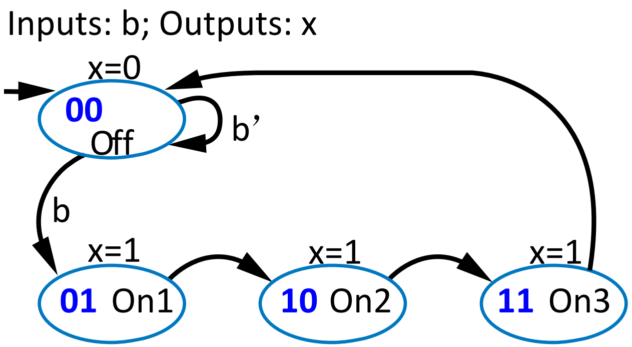
- Step 4: Create state table

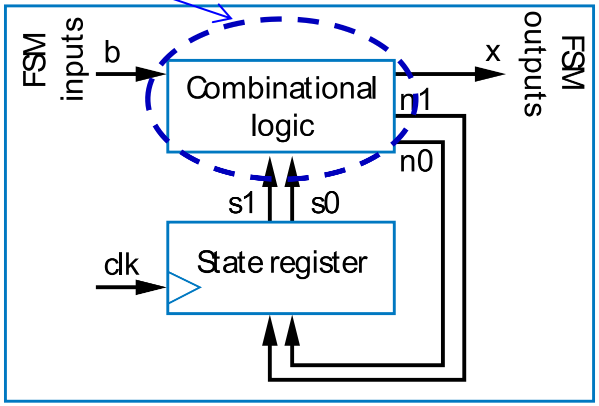
	Inputs			Outputs		
	Present state		b	x	**Next state**	
Off	0	0	0	0	0	0
	0	0	1	0	0	1
On1	0	1	0	1	1	0
	0	1	1	1	1	0
On2	1	0	0	1	1	1
	1	0	1	1	1	1
On3	1	1	0	1	0	0
	1	1	1	1	0	0

$\underbrace{\qquad\qquad\qquad}_{\text{Combinational Logic Inputs}}$ $\underbrace{\qquad\qquad\qquad}_{\text{Combinational Logic Outputs}}$

Combinational Logic Inputs

Combinational Logic Outputs



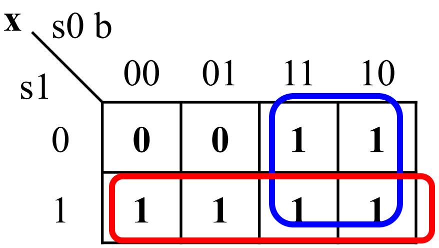


FSM Design Example: Push Button (cont.)

- Step 5: Implement combinational logic

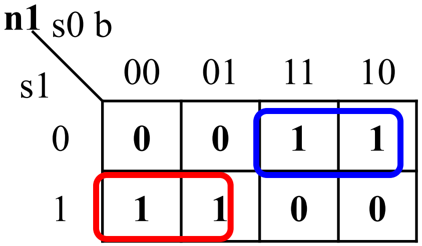
	Inputs			Outputs		
	s1	s0	b	x	n1	n0
Off	0	0	0	0	0	0
	0	0	1	0	0	1
On1	0	1	0	1	1	0
	0	1	1	1	1	0
On2	1	0	0	1	1	1
	1	0	1	1	1	1
On3	1	1	0	1	0	0
	1	1	1	1	0	0

x = s1 + s0



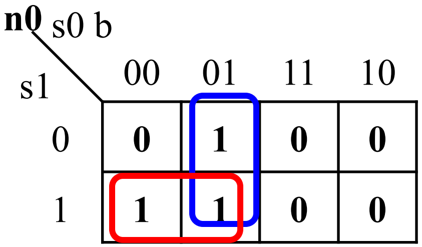
x s1 \ s0 b	00	01	11	10
0	**0**	**0**	**1**	**1**
1	**1**	**1**	**1**	**1**

n1 = s1’s0 + s1s0’



n1 s1 \ s0 b	00	01	11	10
0	**0**	**0**	**1**	**1**
1	**1**	**1**	**0**	**0**

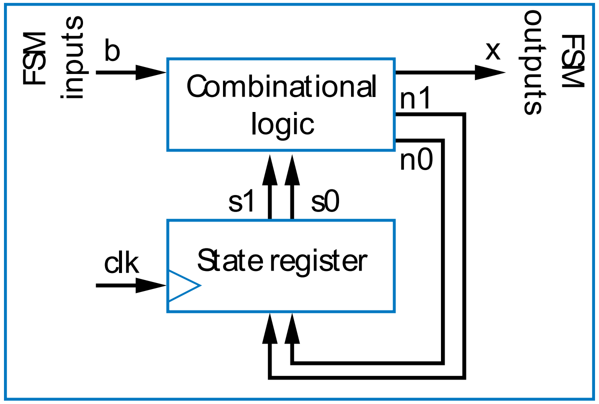
n0 = s0’b + s1s0’

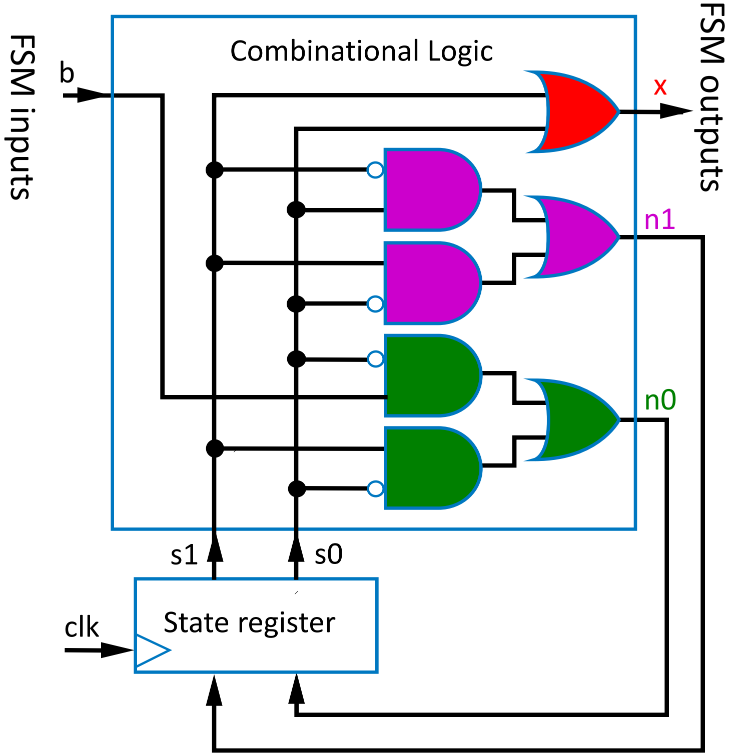


n0 s1 \ s0 b	00	01	11	10
0	**0**	**1**	**0**	**0**
1	**1**	**1**	**0**	**0**

FSM Design Example: Push Button (cont.)

- Step 5: Implement combinational logic (cont)



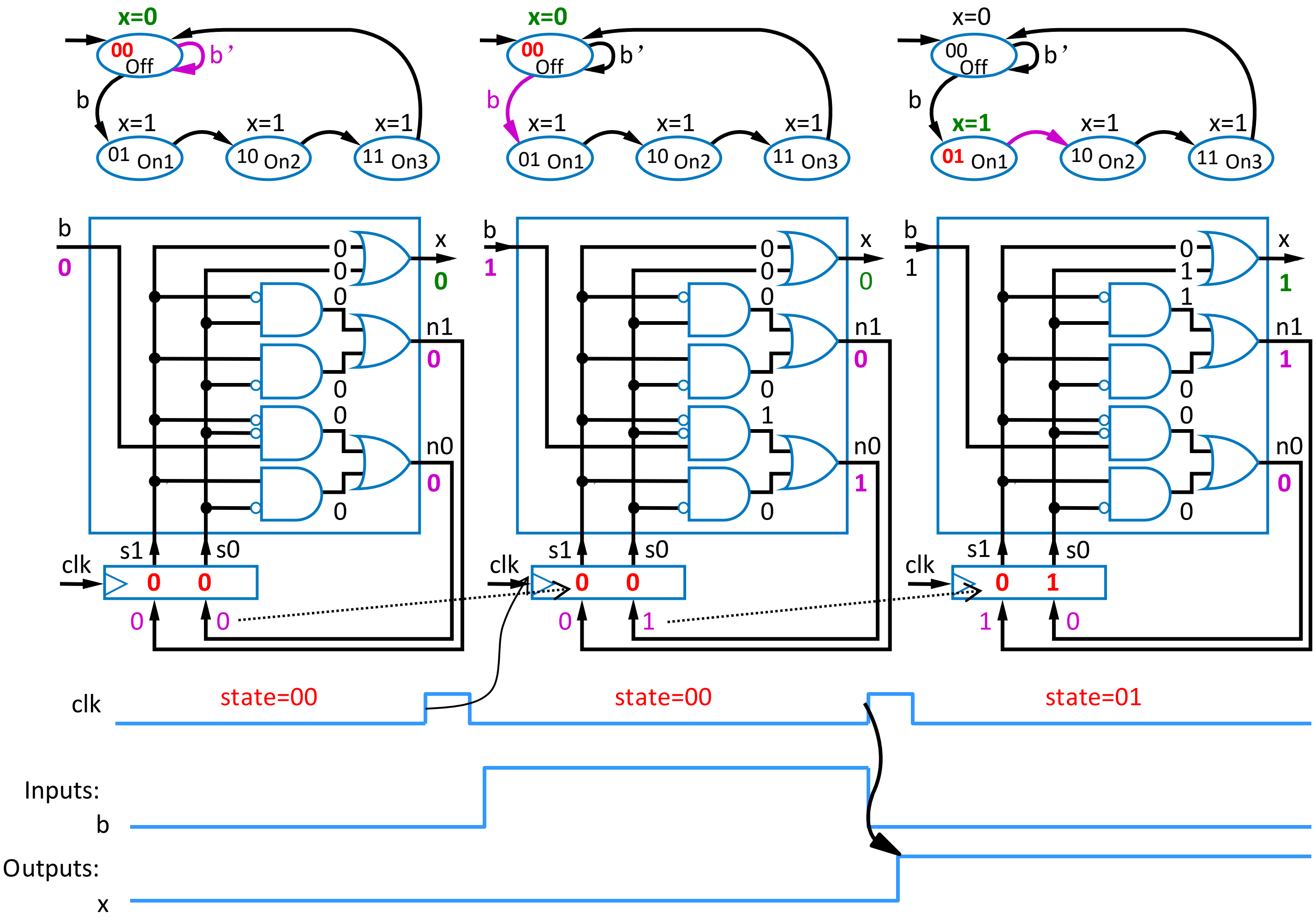


x = s1 + s0

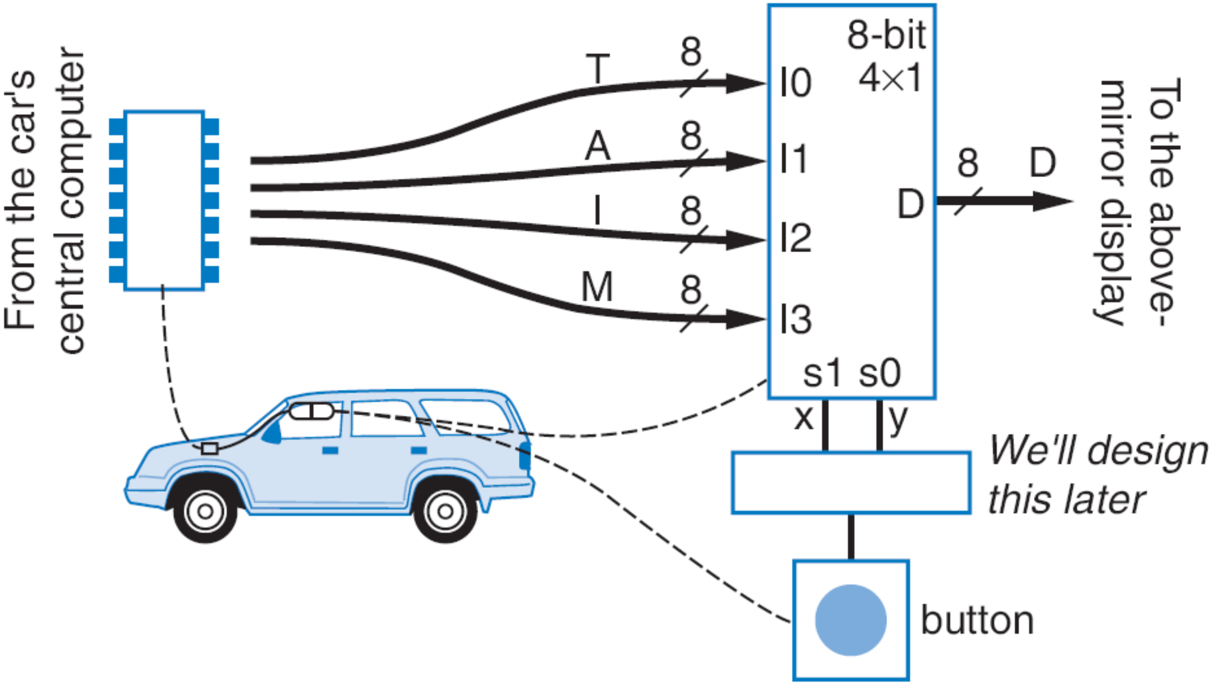
n1 = s1’s0 + s1s0’

n0 = s0’b + s1s0’

Understanding the Controller’s Behavior

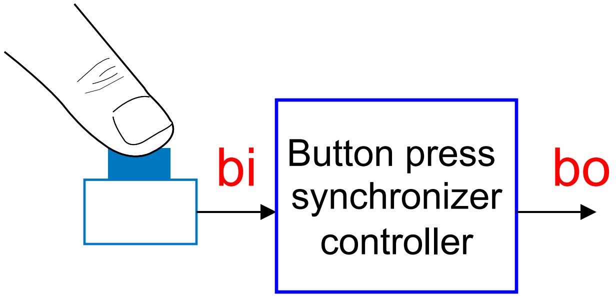


Recall: N-bit Mux Example



- Four possible display items
  - Temperature (T), Average miles-per-gallon (A), Instantaneous mpg (I), and Miles remaining (M) -- each is 8-bits wide
  - Choose which to display using two inputs x and y
  - Use 8-bit 4x1 mux

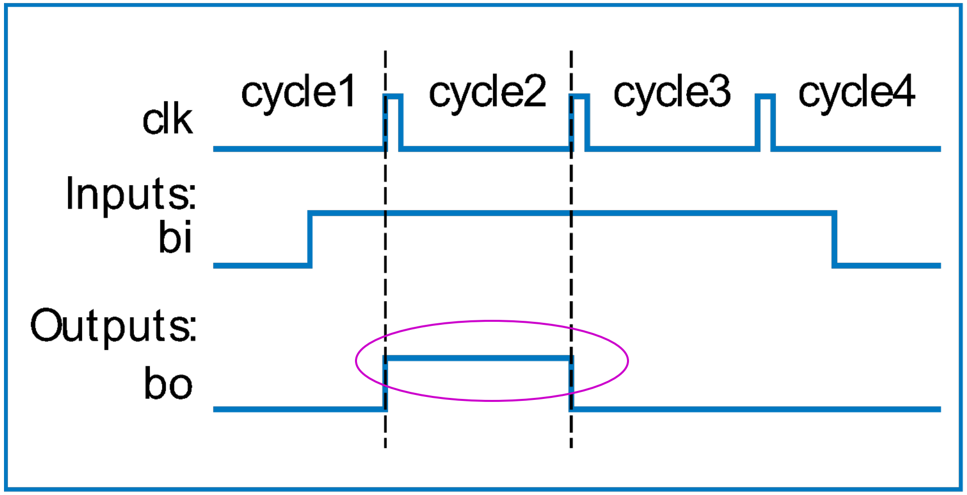
FSM Design Example:
Button Press Synchronizer



bi

Button press synchronizer controller

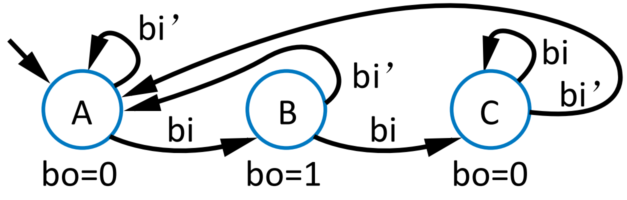
bo



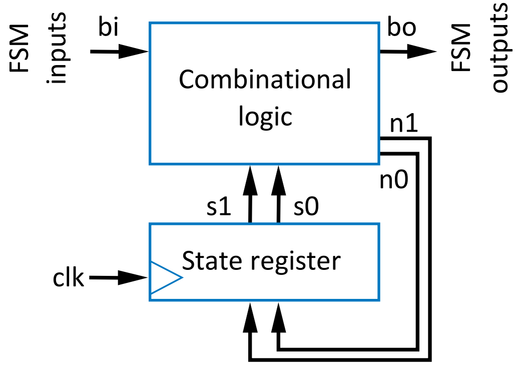
- Want simple sequential circuit that converts button press to single clock cycle duration, regardless of length of time that button actually pressed
- Producing a signal like ‘a’ in the secure door example

FSM Design Example:
Button Press Synchronizer (cont.)

FSM inputs: bi; FSM outputs: bo

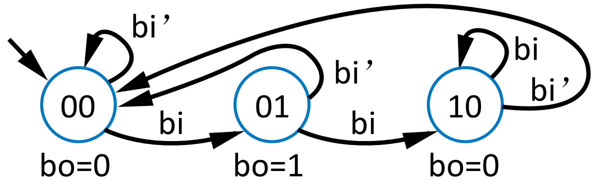


Step 1: FSM



Step 2: Create architecture

FSM inputs: bi; FSM outputs: bo



Step 3: Encode states

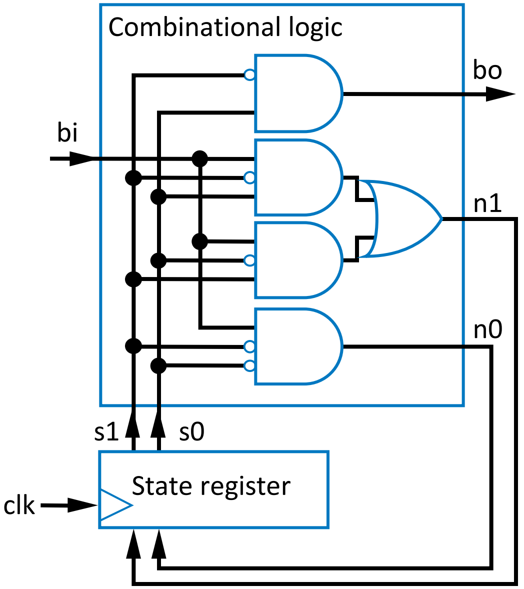
Combinational logic

	Inputs			Outputs		
	s1	s0	bi	n1	n0	bo
A	0	0	0	0	0	0
	0	0	1	0	1	0
B	0	1	0	0	0	1
	0	1	1	1	0	1
C	1	0	0	0	0	0
	1	0	1	1	0	0
unused	1	1	0	0	0	0
	1	1	1	0	0	0

Step 4: State table

may be 'x'

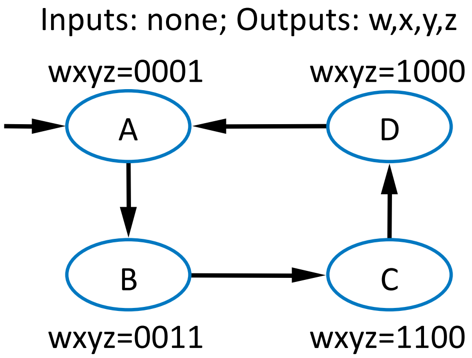
n1 = s1's0bi + s1s0bi
n0 = s1's0'bi
bo = s1's0bi' + s1's0bi = s1s0



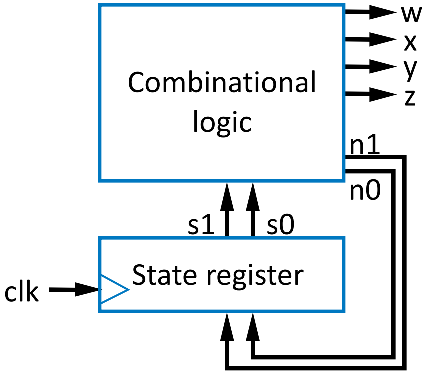
Step 5: Create combinational circuit

FSM Example: Sequence Generator

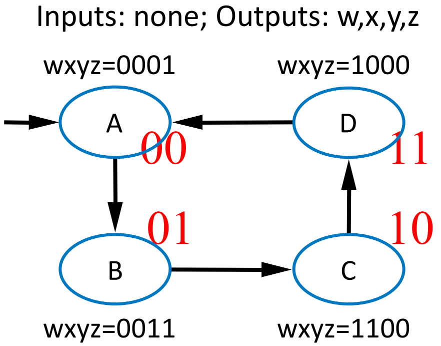
- Want generate sequence 0001, 0011, 1100, 1000, (repeat)
    - Each value for one clock cycle



Step 1: Create FSM



Step 2: Create architecture

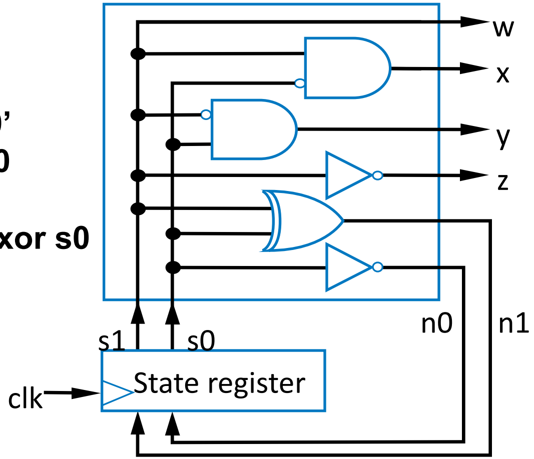


Step 3: Encode states

	Inputs		Outputs					
	s1	s0	w	x	y	z	n1	n0
A	0	0	0	0	0	1	0	1
B	0	1	0	0	1	1	1	0
C	1	0	1	1	0	0	1	1
D	1	1	1	0	0	0	0	0

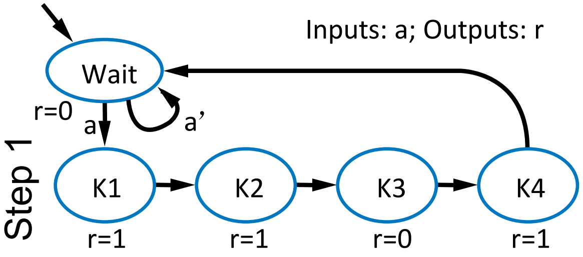
Step 4: Create state table

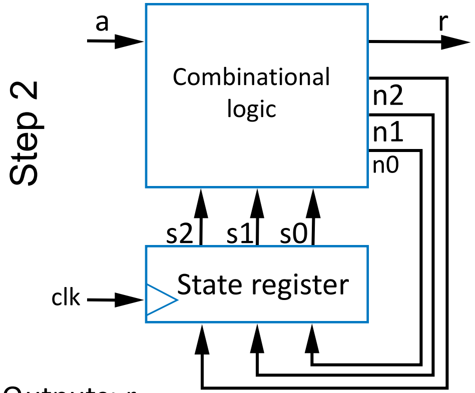
w = s1
x = s1s0’
y = s1’s0
z = s1’
n1 = s1 xor s0
n0 = s0’

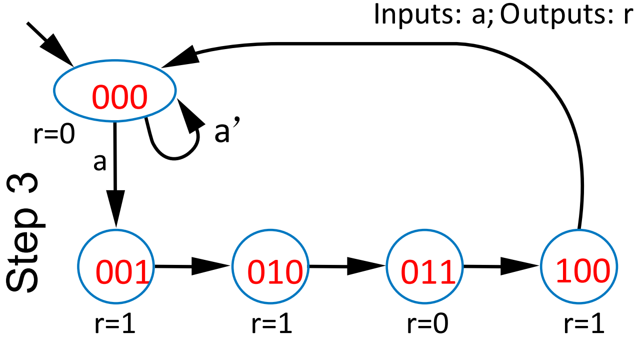


Step 5: Create combinational circuit

FSM Example: Secure Car Key







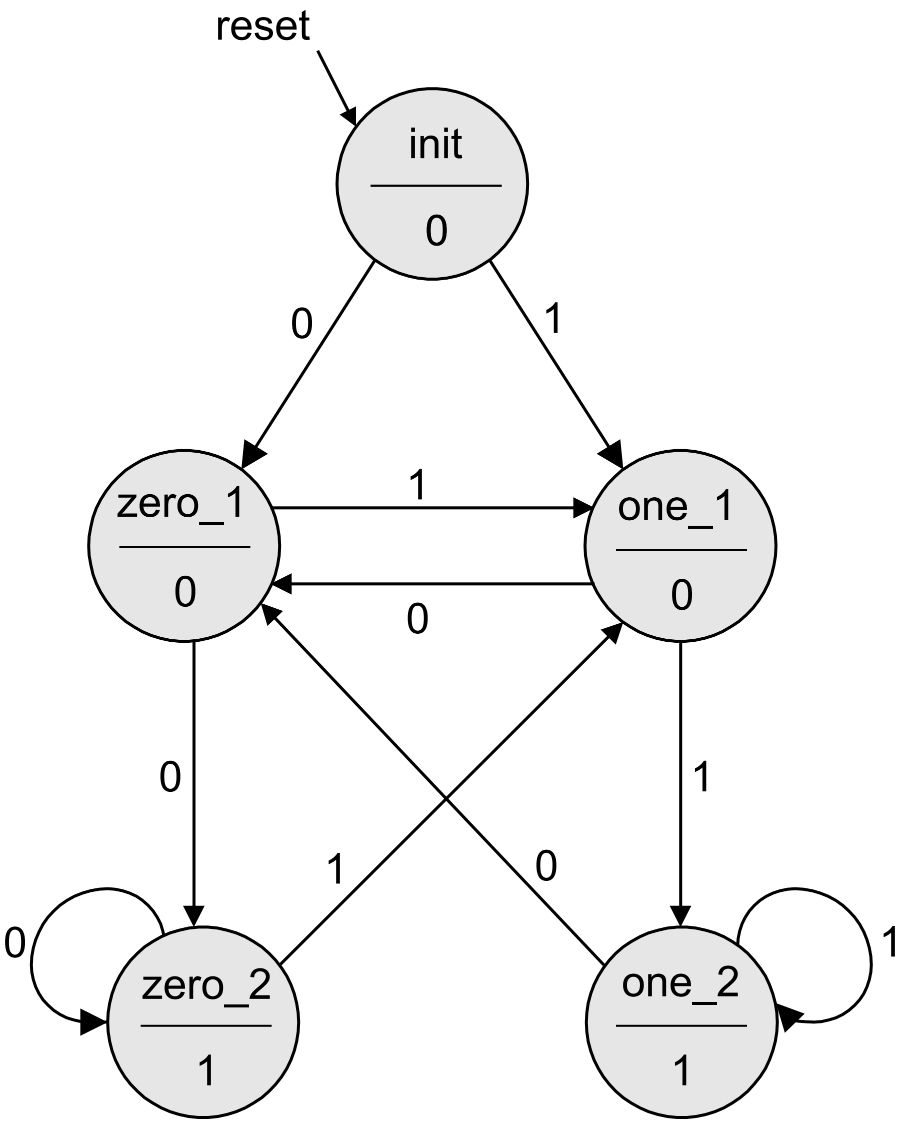
We omit Step 5

	Inputs				Outputs			
	s2	s1	s0	a	r	n2	n1	n0
Wait	0	0	0	0	0	0	0	0
	0	0	0	1	0	0	0	1
K1	0	0	1	0	1	0	1	0
	0	0	1	1	1	0	1	0
K2	0	1	0	0	1	0	1	1
	0	1	0	1	1	0	1	1
K3	0	1	1	0	0	1	0	0
	0	1	1	1	0	1	0	0
K4	1	0	0	0	1	0	0	0
	1	0	0	1	1	0	0	0
Unused	1	0	1	0	0	0	0	0
	1	0	1	1	0	0	0	0
	1	1	0	0	0	0	0	0
	1	1	0	1	0	0	0	0
	1	1	1	0	0	0	0	0
	1	1	1	1	0	0	0	0

Step 4

may be 'x'

Verilog Modeling of FSM



```
Module example_FSM (clock, reset, in_bit, out_bit);
  input      clock, reset, in_bit;
  output     out_bit;

  reg [2:0]  curr_state, next_state;

  parameter  init   = 3'b000;
  parameter  zero_1 = 3'b001;
  parameter  one_1  = 3'b010;
  parameter  zero_2 = 3'b011;
  parameter  one_2  = 3'b100;

  always @ (posedge clock or posedge reset)
    if (reset == 1) curr_state <= init;
    else            curr_state <= next_state;

  always @ (curr_state or in_bit)
    case (curr_state)
      init: if (in_bit == 0) next_state <= zero_1; else
            if (in_bit == 1) next_state <= one_1;  else
                             next_state <= init;
```

State register

Combinational logic for next state

```
      zero_1: if (in_bit == 0) next_state <= zero_2; else
              if (in_bit == 1) next_state <= one_1;  else
                               next_state <= init;
      zero_2: if (in_bit == 0) next_state <= zero_2; else
              if (in_bit == 1) next_state <= one_1;  else
                               next_state <= init;
      one_1:  if (in_bit == 0) next_state <= zero_1; else
              if (in_bit == 1) next_state <= one_2;  else
                               next_state <= init;
      one_2:  if (in_bit == 0) next_state <= zero_1; else
              if (in_bit == 1) next_state <= one_2;  else
                               next_state <= init;
      default:                 next_state <= init;
    endcase

  assign out_bit = ((curr_state==zero_2)||(curr_state==one_2))
                   ? 1 : 0;

endmodule
```

Combinational logic for FSM outputs