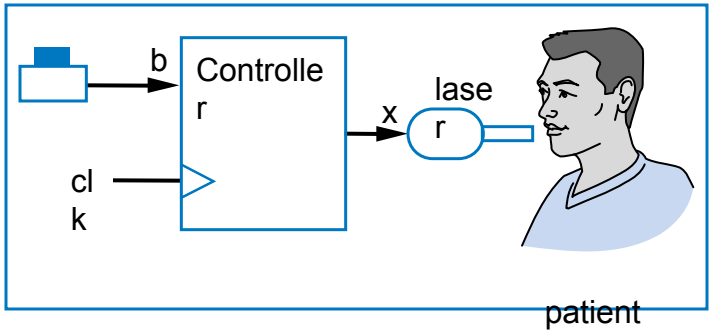
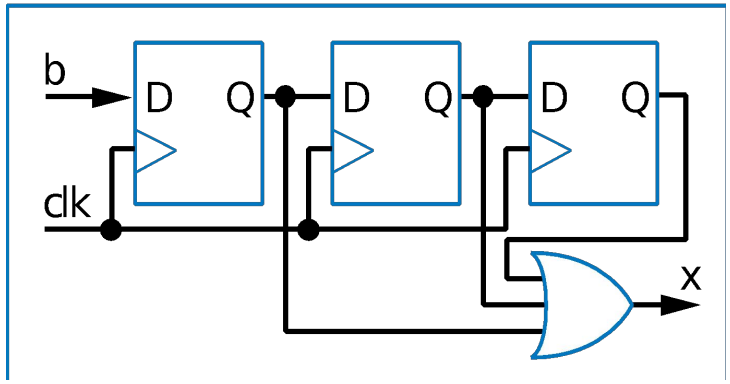
# Finite-State Machines (FSMs) and Controllers

# Finite-State Machines (FSMs) and Controllers

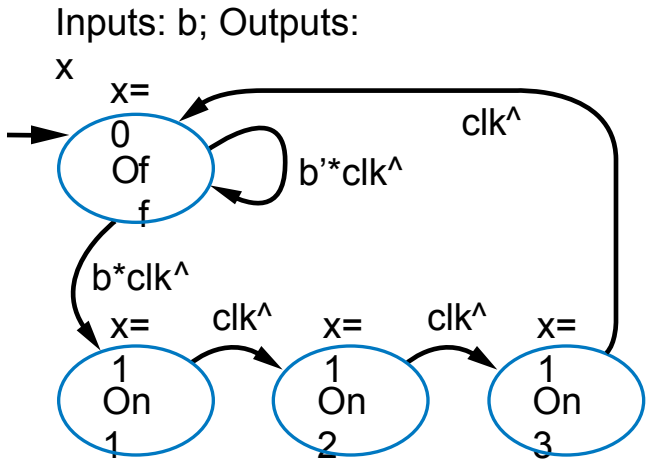
- Want sequential circuit with particular behavior over time
- Example: Laser timer
  - Push button: x=1 for 3 clock cycles
  - How? Let's try three flip-flops
    - b=1 gets stored in first D flip-flop
    - Then 2nd flip-flop on next cycle, then 3rd flip-flop on next
    - OR the three flip-flop outputs, so x should be 1 for three cycles

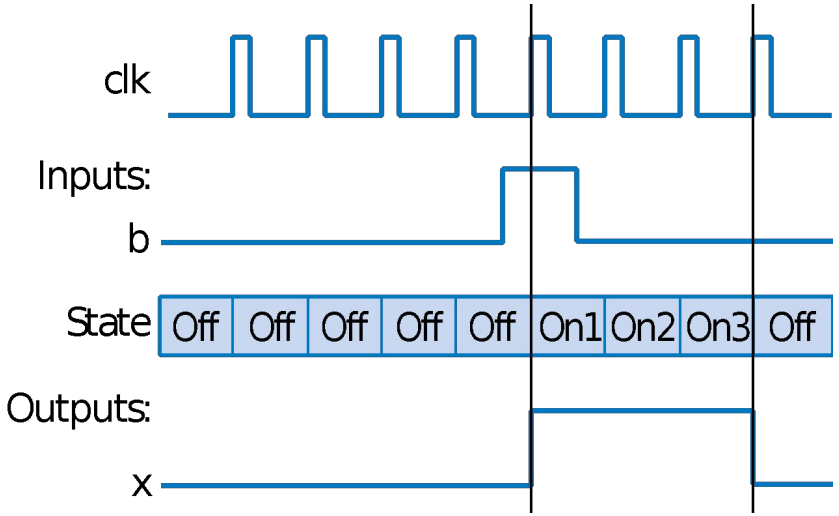


# Extend FSM to Three-Cycles High Laser Timer

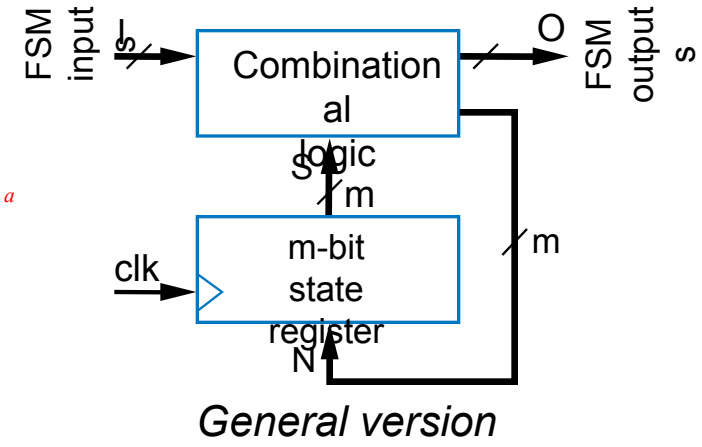
- Four states
- Wait in “Off” state while b is 0 (b’)
- When b is 1 (and rising clock edge), transition to On1
  - Sets x=1
  - On next two clock edges, transition to On2, then On3, which also set x=1
- So x=1 for three cycles after button pressed



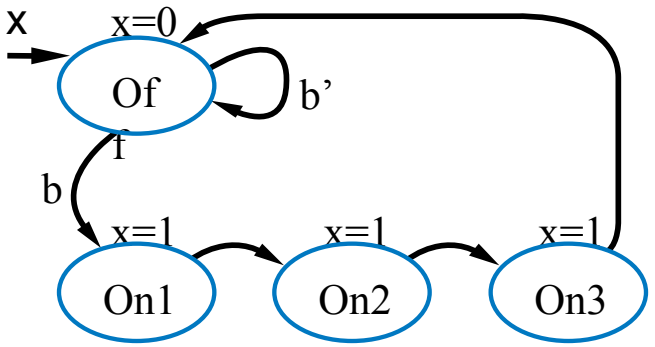
# Standard Controller Architecture

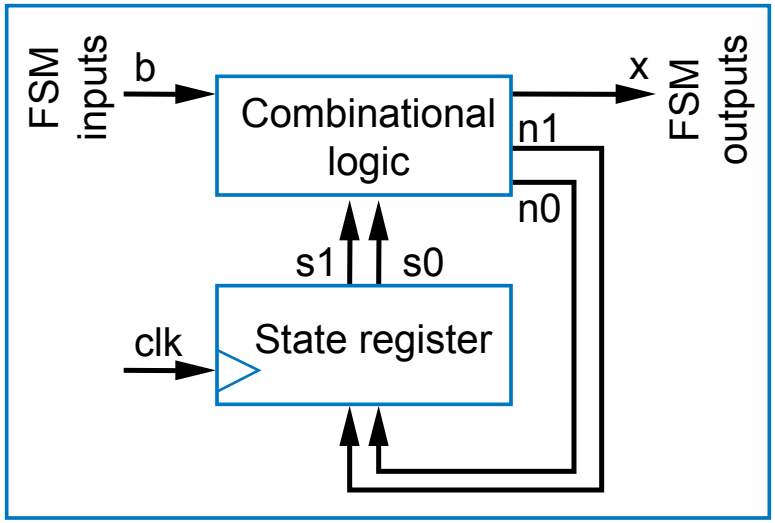
- How implement FSM as sequential circuit?
  - Use standard architecture
    - State register -- to store the present state
    - Combinational logic -- to compute outputs, and next state
    - For laser timer FSM
      - 2-bit state register, can represent four states
      - Input b, output x
  - Known as ***controller***




*General version*

Inputs: b; Outputs:



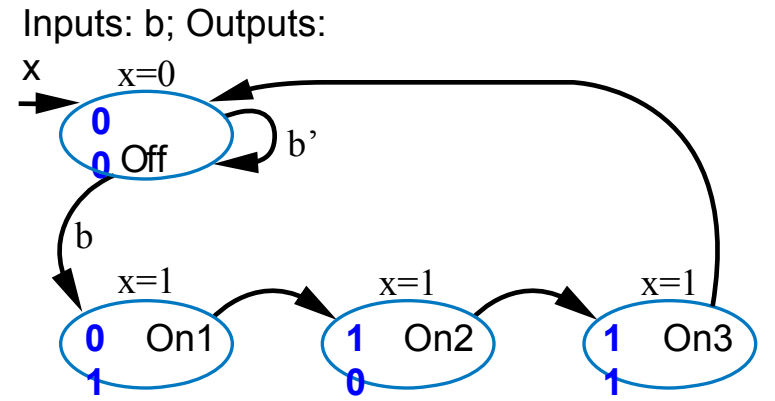
# Controller Design

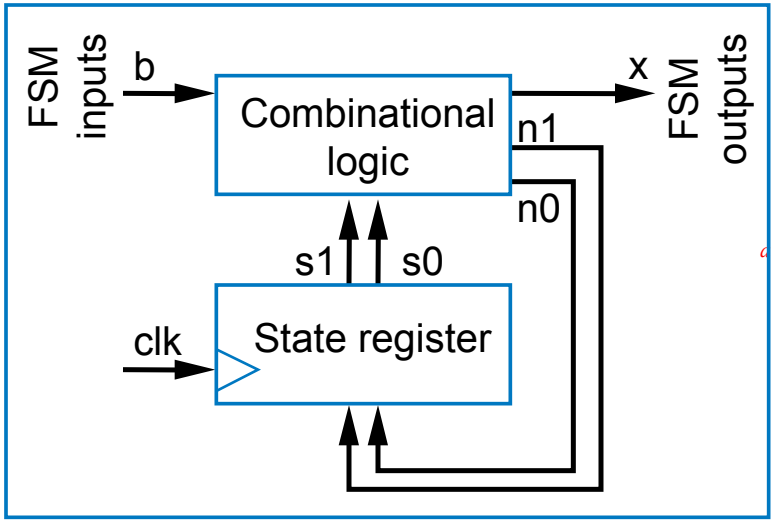
- Five step controller design process

| | 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. |

# Controller Design: Laser Timer Example

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

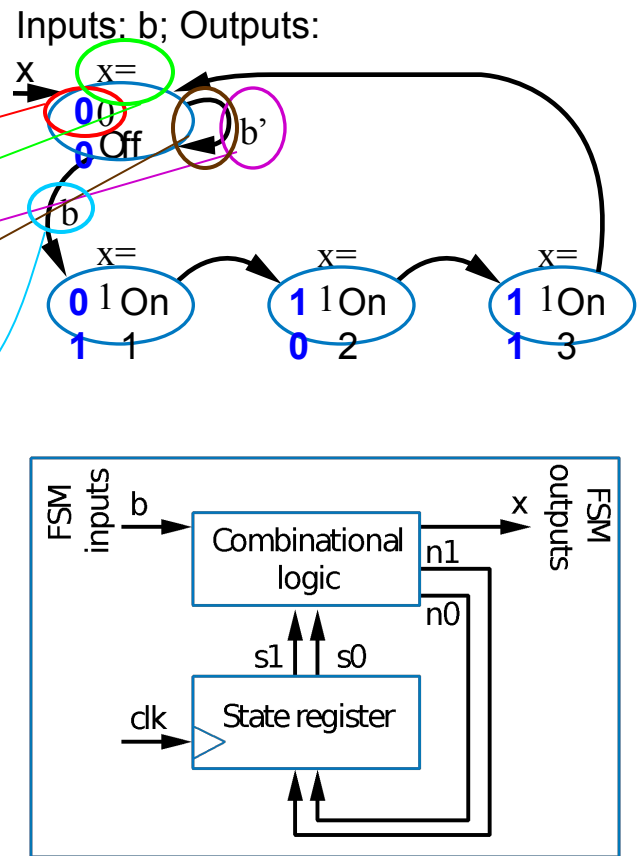


# Controller Design: Laser Timer Example (cont)

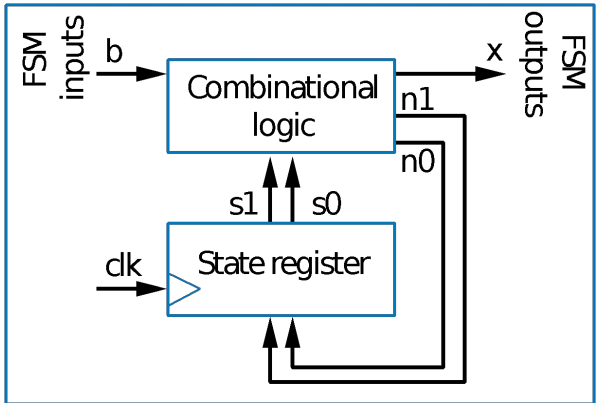
- Step 4: Create state table

| | 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 |



# Controller Design: Laser Timer Example (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 (note from the table that x=1 if s1 = 1 or s0 = 1)

n1 = s1's0b' + s1's0b + s1s0'b' + s1s0'b

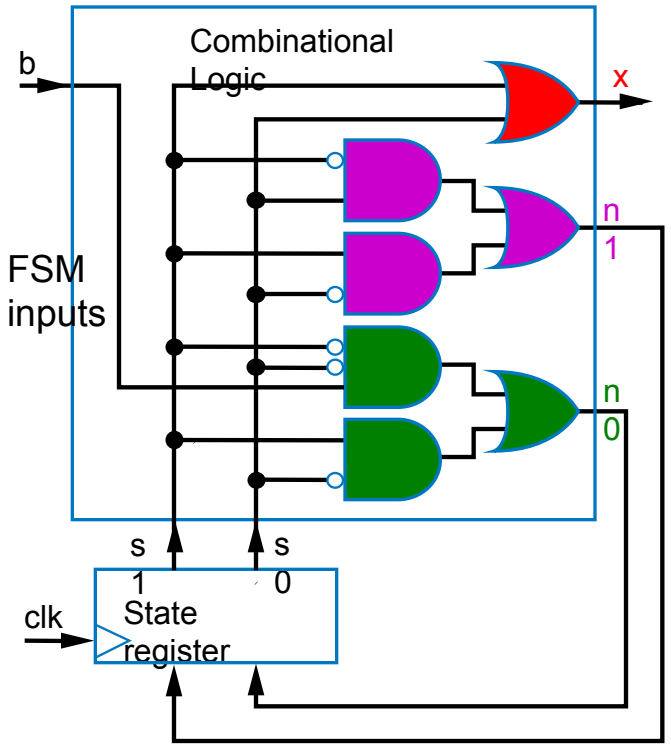
n1 = s1's0 + s1s0'

n0 = s1's0'b + s1s0'b' + s1s0'b

n0 = s1's0'b + s1s0'

# Controller Design: Laser Timer Example (cont)

- Step 5: Implement combinational logic (cont)



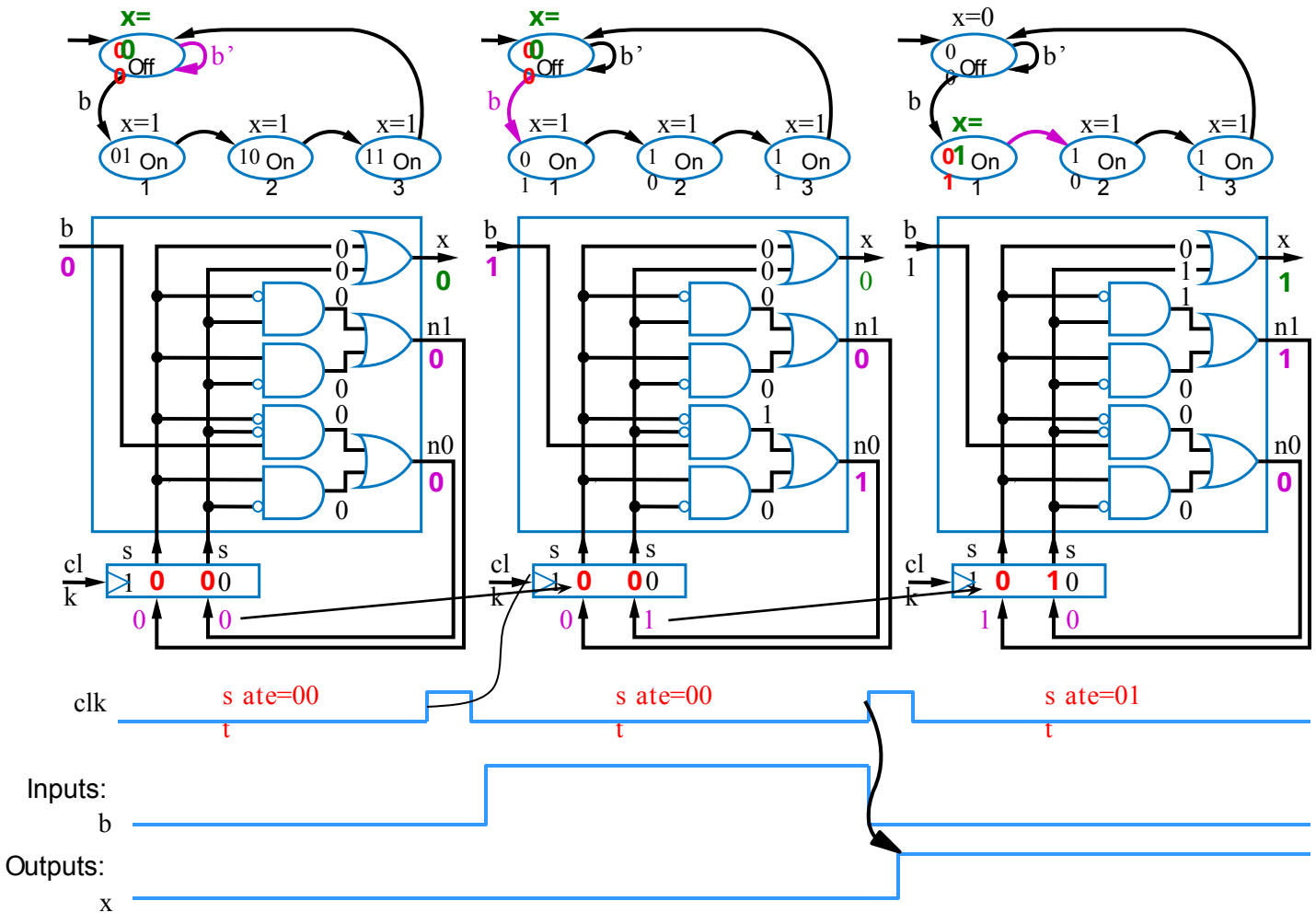

| | Inputs | | | Outputs | | |
|---|---|---|---|---|---|---|
| | s1 | s0 | b | x | n1 | n0 |
| *Off* | 0 | 0 | 0 | 0 | 0 | 0 |
| | 0 | 0 | 1 | 0 | 0 | 1 |
| *On1* | 0 | 1 | 0 | 1 | 1 | 0 |
| | 0 | 1 | 1 | 1 | 1 | 0 |
| *On2* | 1 | 0 | 0 | 1 | 1 | 1 |
| | 1 | 0 | 1 | 1 | 1 | 1 |
| *On3* | 1 | 1 | 0 |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

x = s1 + s0

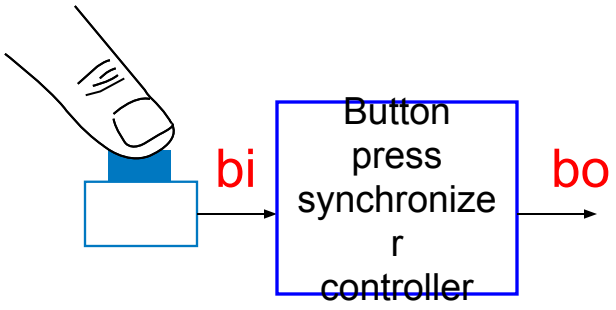
n1 = s1's0 + s1s0'

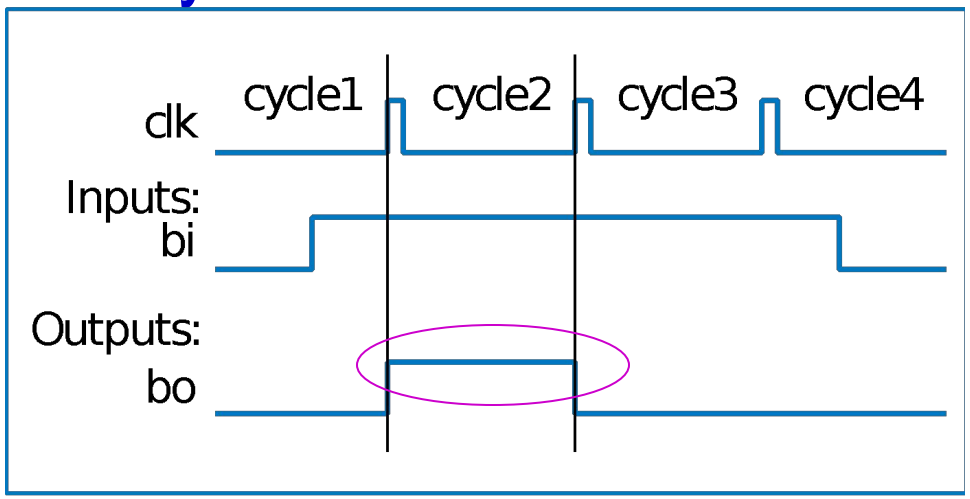
n0 = s1's0'b + s1s0'

# Understanding the Controller's Behavior



# Controller Example:
## Button Press Synchronizer

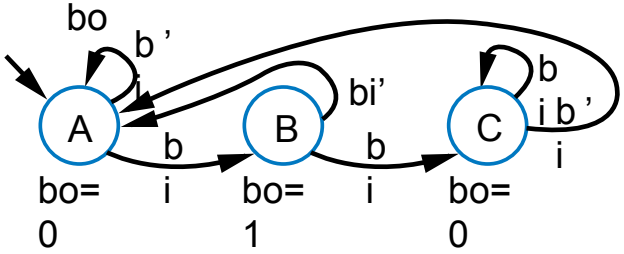


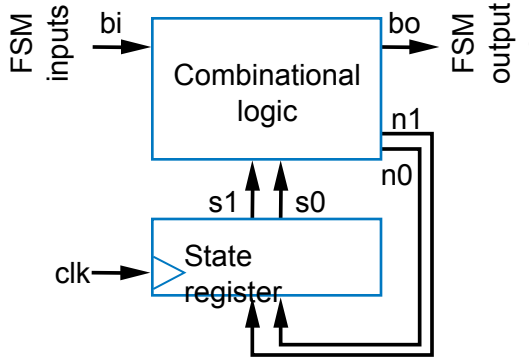

- Want simple sequential circuit that converts button press to single cycle duration, regardless of length of time that button actually pressed
  - We assumed such an ideal button press signal in earlier example, like the button in the laser timer controller

# Controller Example:
## Button Press Synchronizer (cont)

FSM inputs: bi; FSM outputs: bo

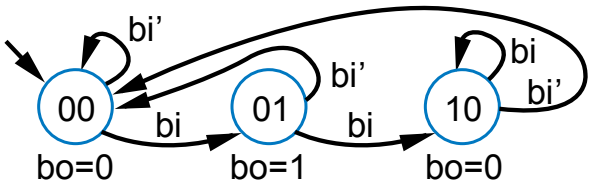



Step 1: FSM




Step 2: Create architecture

FSM inputs: bi; FSM outputs: bo

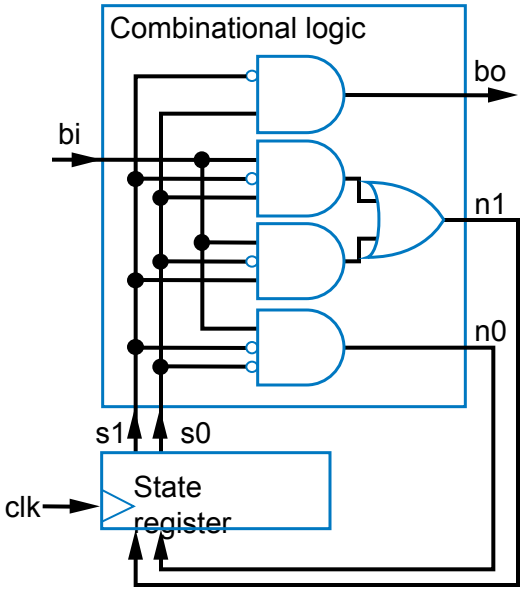



Step 3: Encode states

| | 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

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



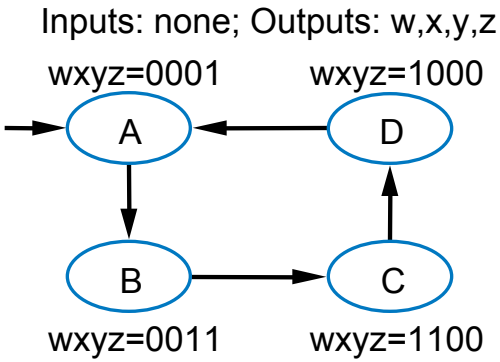

Step 5: Create combinational circuit



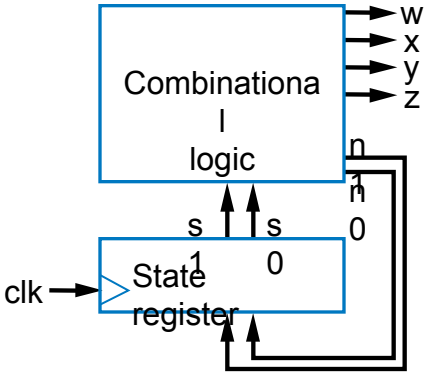
Step 5: Create combinational circuit

# Controller Example: Sequence Generator

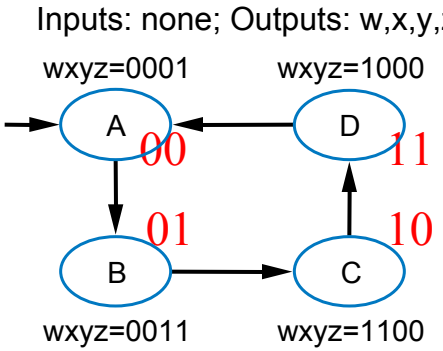
- Want generate sequence 0001, 0011, 1100, 1000, (repeat)
  - Each value for one clock cycle
  - Common, e.g., to create pattern in 4 lights, or control magnets of a “stepper motor”



Step 1: Create FSM



Step 2: Create architecture

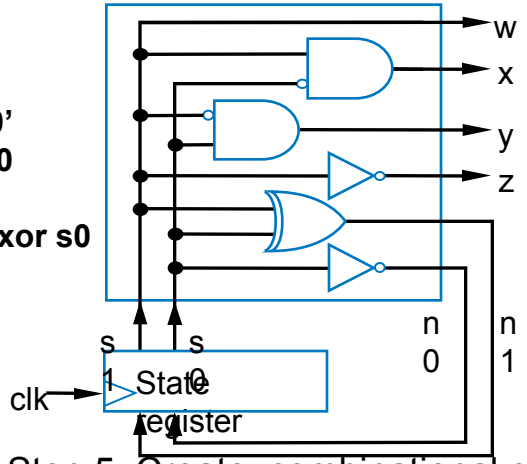


Step 3: Encode states

| 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

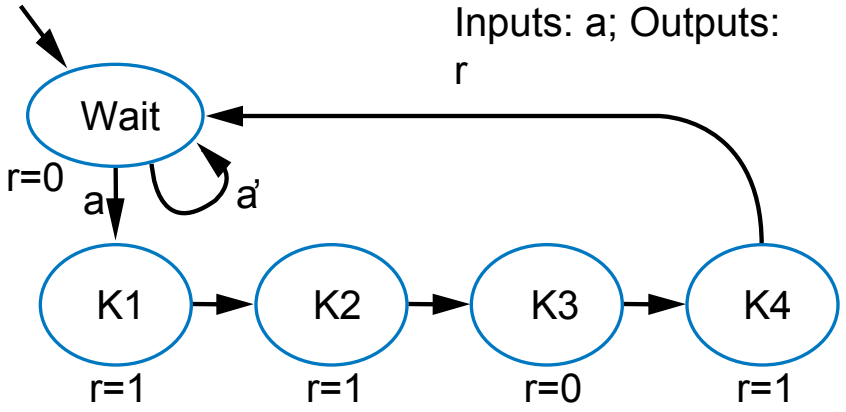
**w = s1**
**x = s1s0’**
**y = s1’s0**
**z = s1’**
**n1 = s1 xor s0**
**n0 = s0’**



Step 5: Create combinational circuit

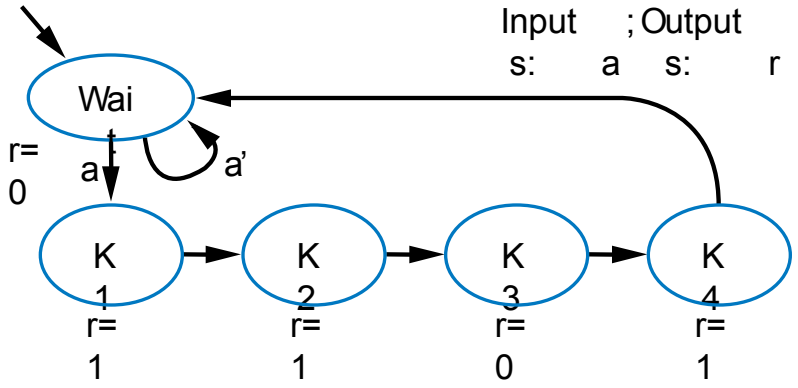
# FSM Example: Secure Car Key

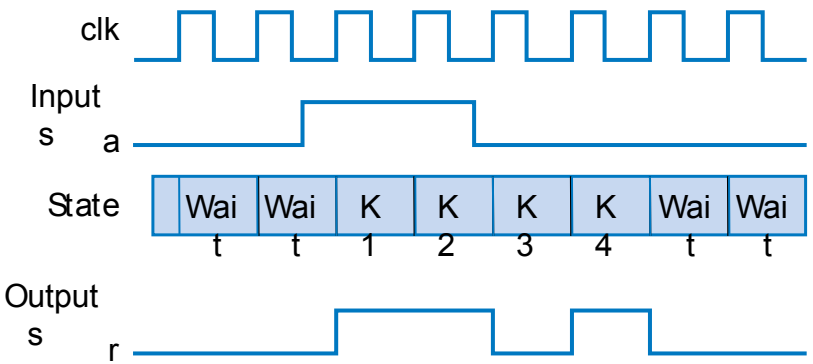
- Many new car keys include tiny computer chip
  - When car starts, car's computer (under engine hood) requests identifier from key
  - Key transmits identifier
    - If not, computer shuts off car
- FSM
  - Wait until computer requests ID (a=1)
  - Transmit ID (in this case, 1101)

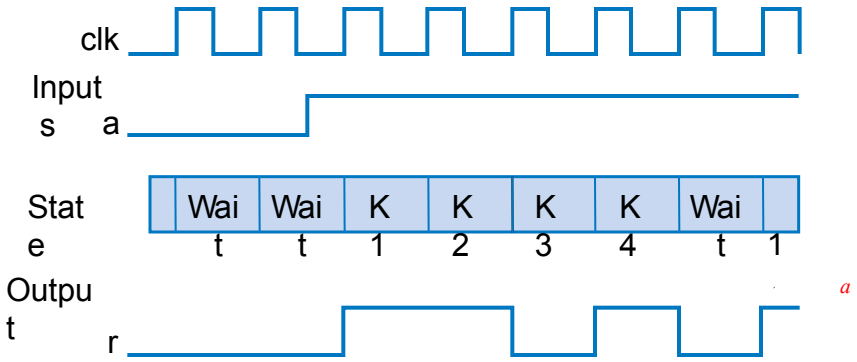


# FSM Example: Secure Car Key (cont.)

- Nice feature of FSM
  - Can evaluate output behavior for different input sequence
  - Timing diagrams show states and output values for different input waveforms

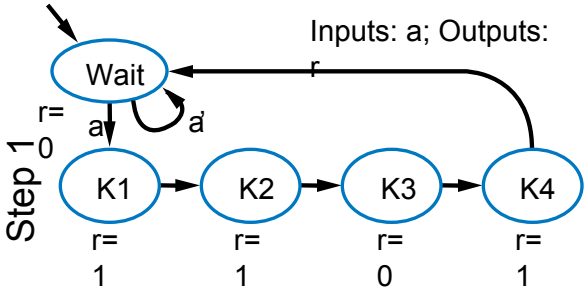


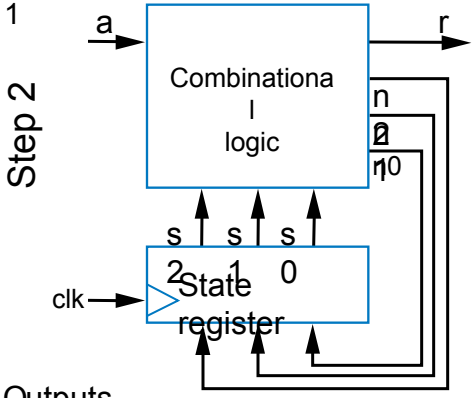



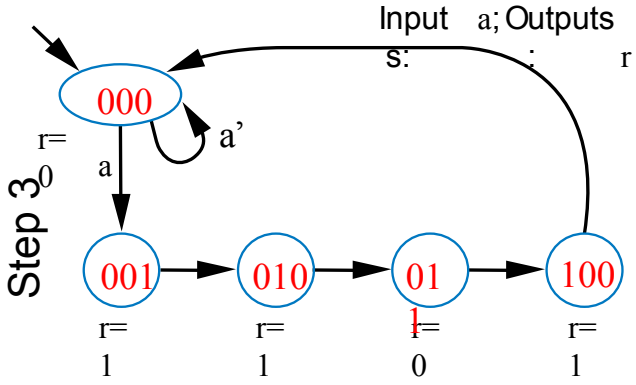



# Controller Example: Secure Car Key

- (from earlier example)



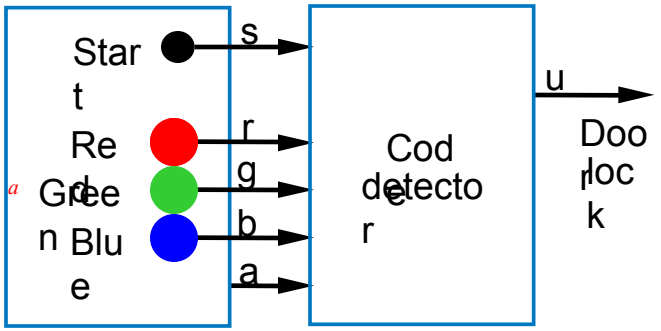

*Step 5-homework*

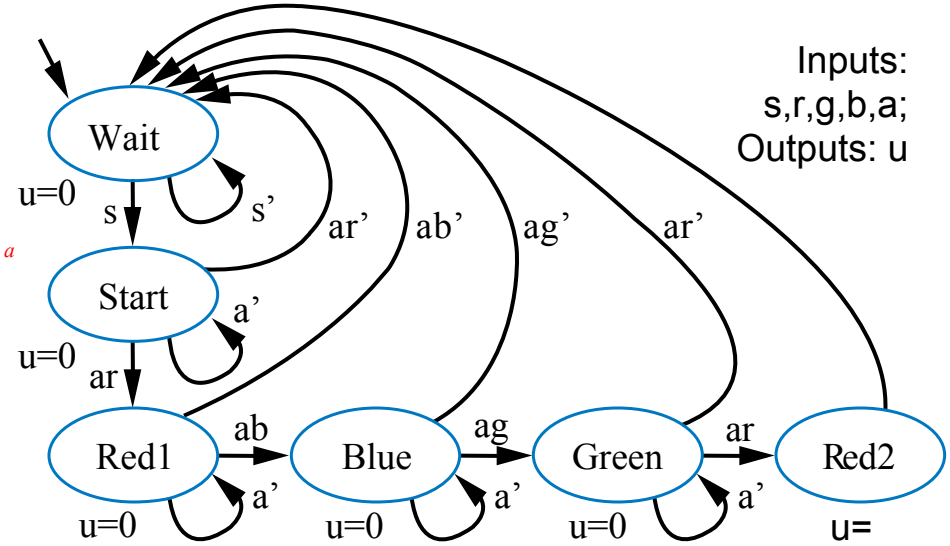
| | Inputs | | | | Outputs | | | |
|---|---|---|---|---|---|---|---|---|
| | s2 | s1 | s0 | a | r | n2 | n1 | n0 |
| *Wait* | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| *K1* | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 |
| *K2* | 0 | 1 | 0 | 0 | 1 | 0 | 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

# FSM Example: Code Detector

- Unlock door (u=1) only when buttons pressed in sequence:
  - start, then red, blue, green, red
- Input from each button: *s, r, g, b*
  - Also, output *a* indicates that some colored button pressed
- FSM
  - Wait for start (s=1) in "Wait"
  - Once started ("Start")
    - 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"
      - In that state, open the door (u=1)
    - Wrong button at any step, return to "Wait", without opening door

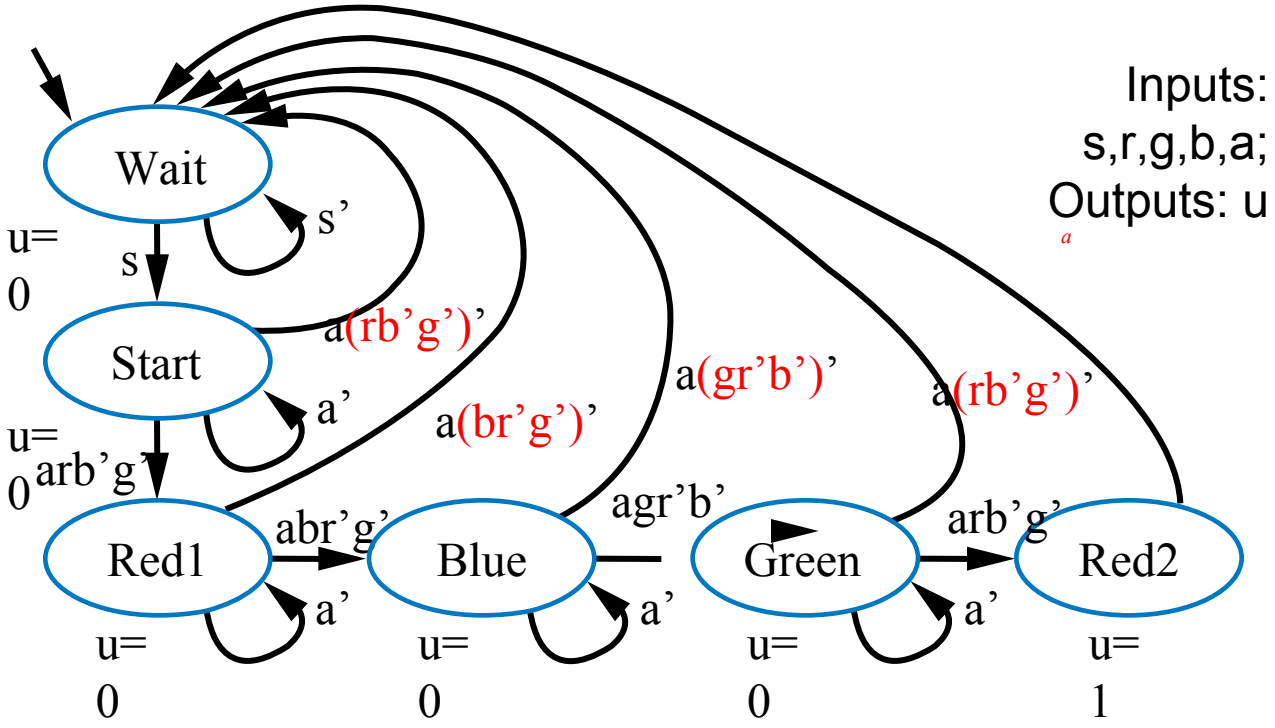



Q: Can you trick this FSM to open the door, without knowing the code?

A: Yes, hold all buttons simultaneously

# Improve FSM for Code Detector




- New transition conditions detect if wrong button pressed, returns to "Wait"
- FSM provides formal, concrete means to accurately define desired behavior