



Digital Design

Chapter 3: Sequential Logic Design -- Controllers

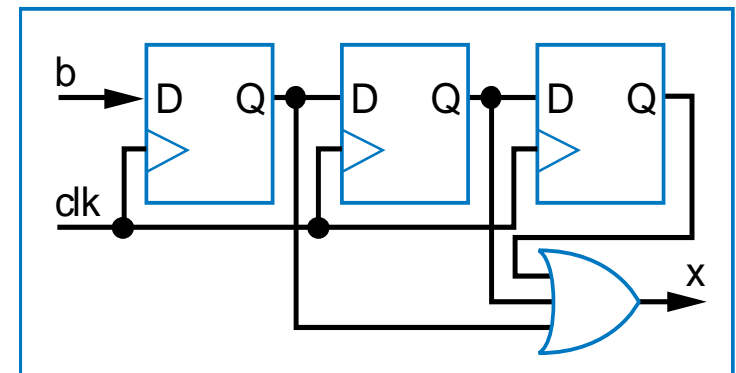
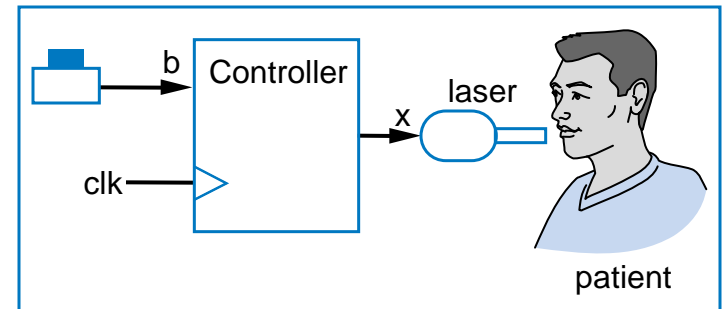
Slides to accompany the textbook *Digital Design*, First Edition,
by Frank Vahid, John Wiley and Sons Publishers, 2007.
<http://www.ddvahid.com>

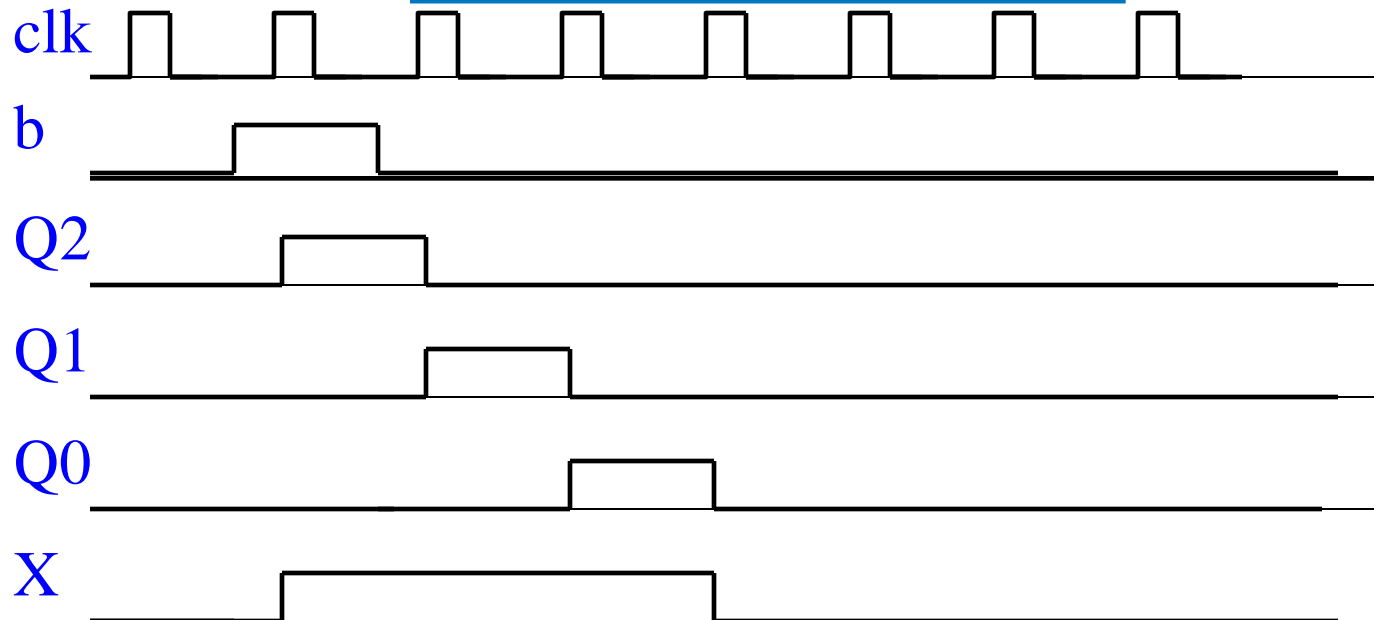
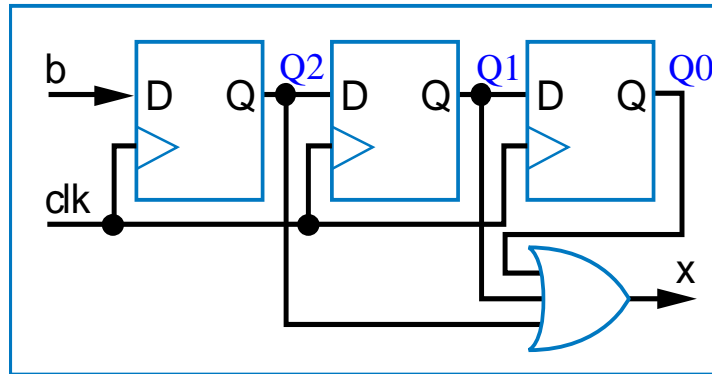
Copyright © 2007 Frank Vahid

Instructors of courses requiring Vahid's *Digital Design* textbook (published by John Wiley and Sons) have permission to modify and use these slides for customary course-related activities, subject to keeping this copyright notice in place and unmodified. These slides may be posted as unanimated pdf versions on publicly-accessible course websites.. PowerPoint source (or pdf with animations) may not be posted to publicly-accessible websites, but may be posted for students on internal protected sites or distributed directly to students by other electronic means. Instructors may make printouts of the slides available to students for a reasonable photocopying charge, without incurring royalties. Any other use requires explicit permission. Instructors may obtain PowerPoint source or obtain special use permissions from Wiley – see <http://www.ddvahid.com> for information.

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





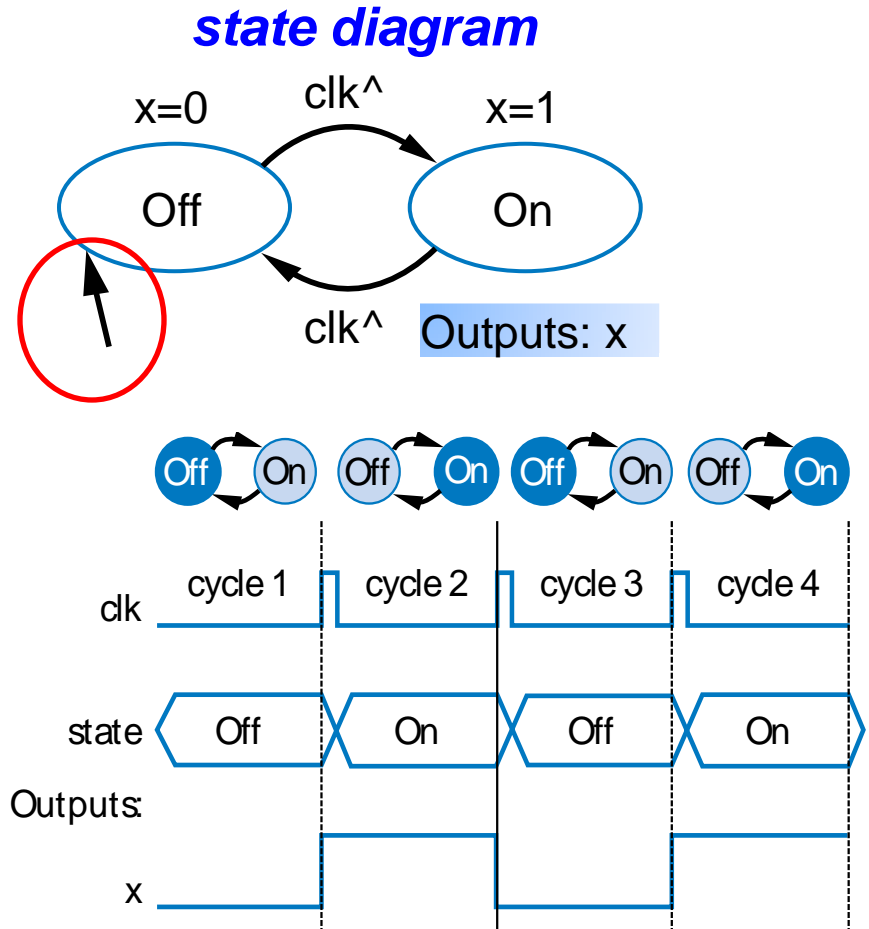
Need a Better Way to Design Sequential Circuits

- Trial and error is not a good design method
 - Will we be able to “guess” a circuit that works for other desired behavior?
 - How about counting up from 1 to 9? Pulsing an output for 1 cycle every 10 cycles? Detecting the sequence 1 3 5 in binary on a 3-bit input?
 - And, a circuit built by guessing may have undesired behavior
 - Laser timer: What if press button again while $x=1$? x then stays one another 3 cycles. Is that what we want?
- Combinational circuit design process had two important things
 1. A formal way to describe desired circuit behavior
 - Boolean equation, or truth table
 2. A well-defined process to convert that behavior to a circuit
- We need those things for sequence circuit design



Describing Behavior of Sequential Circuit: FSM

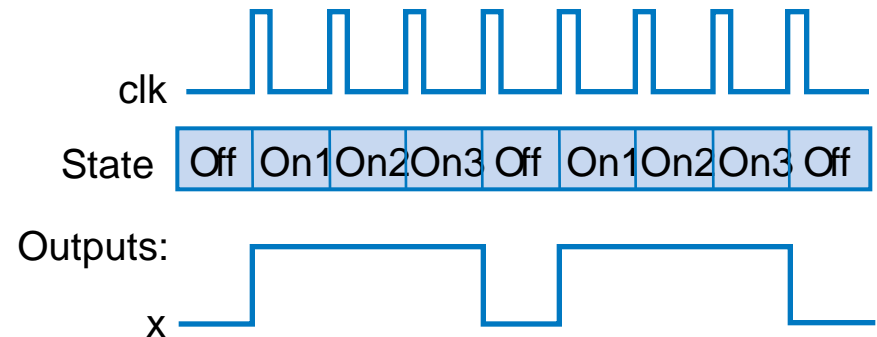
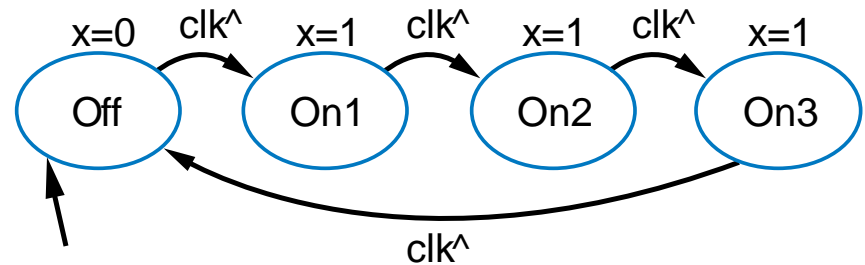
- Finite-State Machine (FSM)
 - A way to describe desired behavior of sequential circuit
 - Use state diagram to list states, and transitions among states
 - Example: Make x change toggle (0 to 1, or 1 to 0) every clock cycle
 - Two states: “Off” ($x=0$), and “On” ($x=1$)
 - Transition from Off to On, or On to Off, on rising clock edge
 - Arrow with no starting state points to initial state (when circuit first starts by asynchronous reset)



FSM Example: 0,1,1,1,repeat

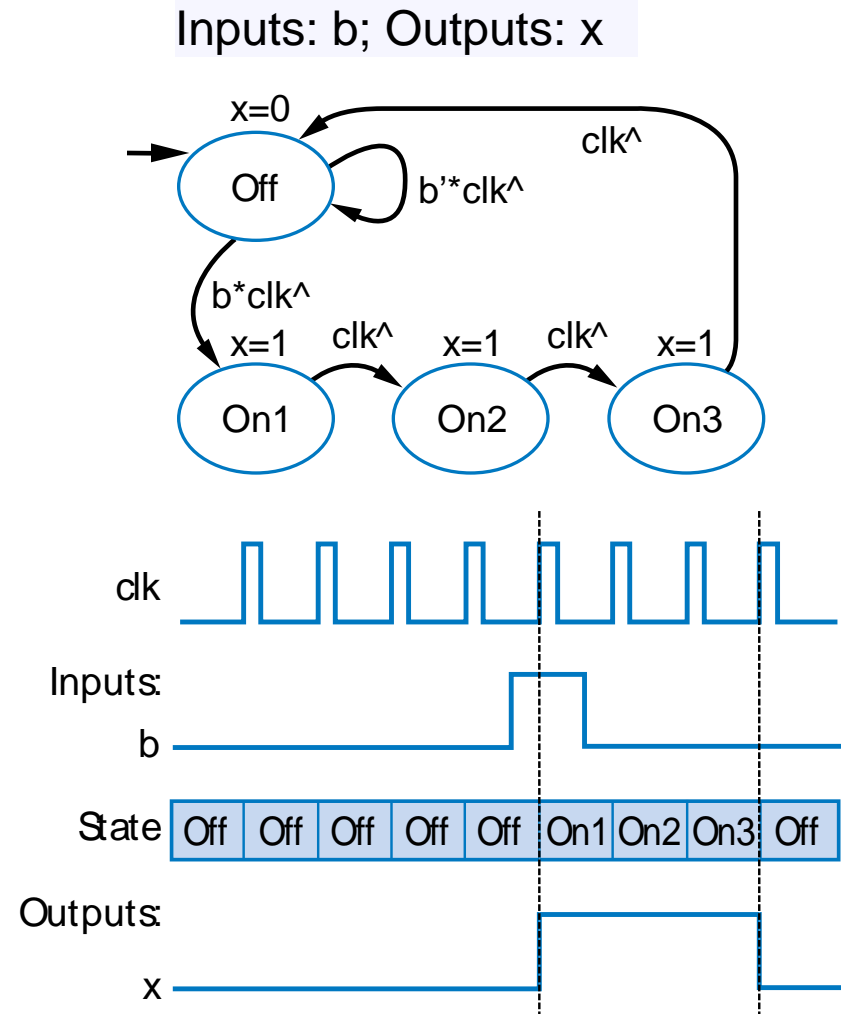
- Want 0, 1, 1, 1, 0, 1, 1, 1, ...
 - Each value for one clock cycle
- Can describe as FSM
 - Four states
 - Transition on rising clock edge to next state

Outputs: x



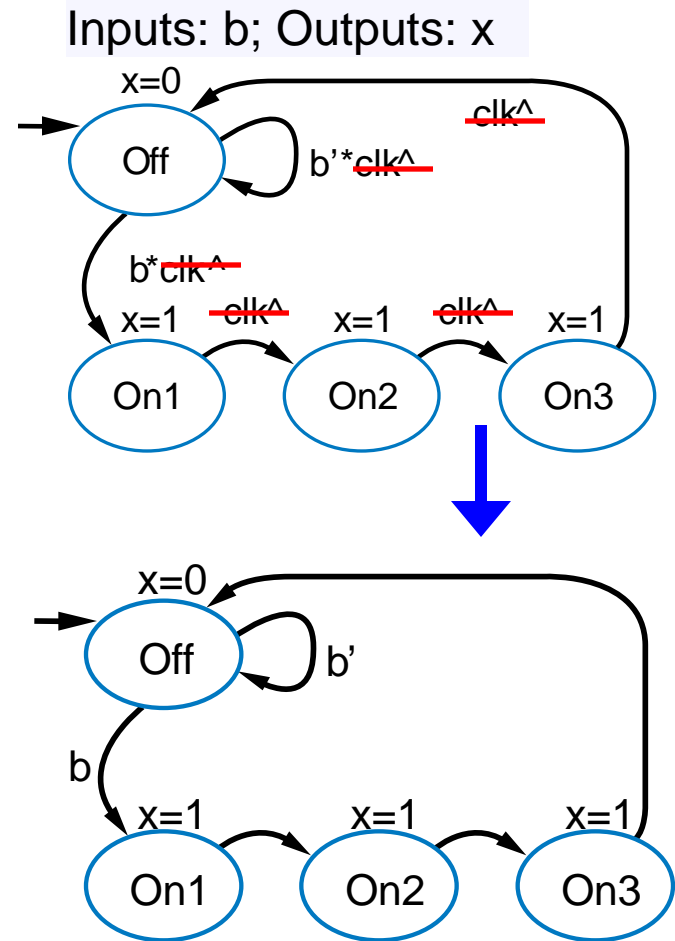
Extend FSM to Three-Cycles High Laser Timer

- 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



FSM Simplification: Rising Clock Edges Implicit

- Showing rising clock on every transition: cluttered
 - Make implicit -- assume every edge has rising clock, even if not shown
 - What if we wanted a transition *without* a rising edge
 - We don't consider such asynchronous FSMs -- less common, and advanced topic
 - Only consider **synchronous** FSMs -- rising edge on every transition



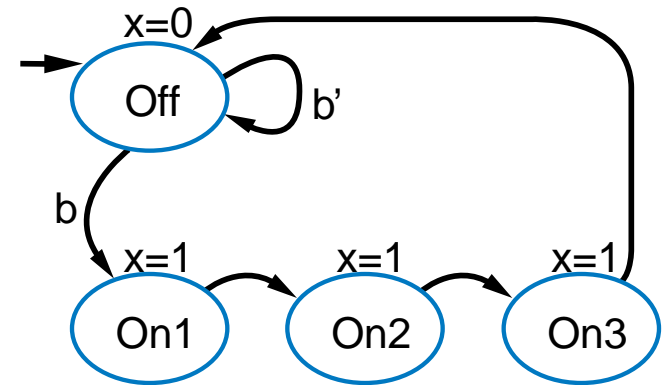
Note: Transition with no associated condition thus transitions to next state on next clock cycle



FSM Definition

- FSM consists of
 - Set of states (flip-flop outputs)
 - 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
 - Sets outputs while in states
 - Ex: $x=0$, $x=1$, $x=1$, and $x=1$

Inputs: b; Outputs: x

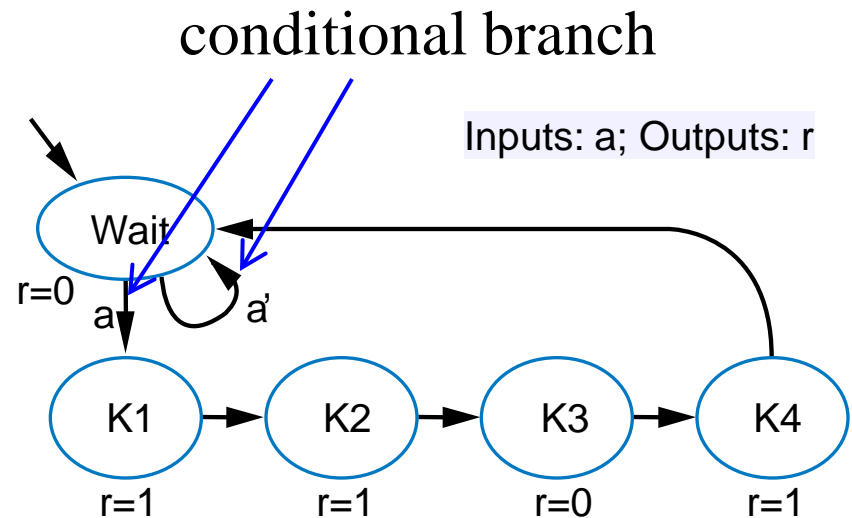


Can also use table (state table), or textual languages



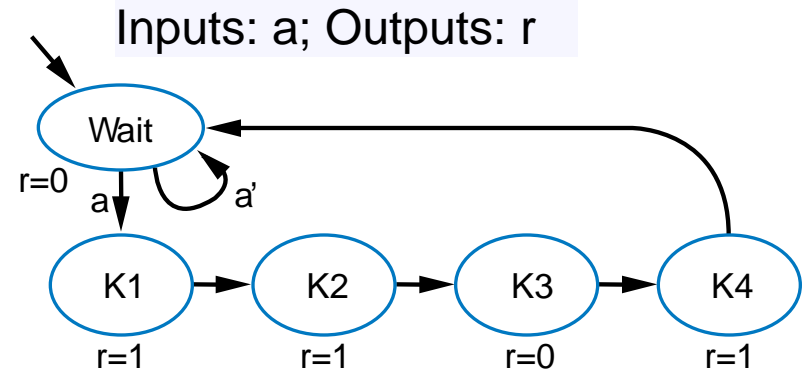
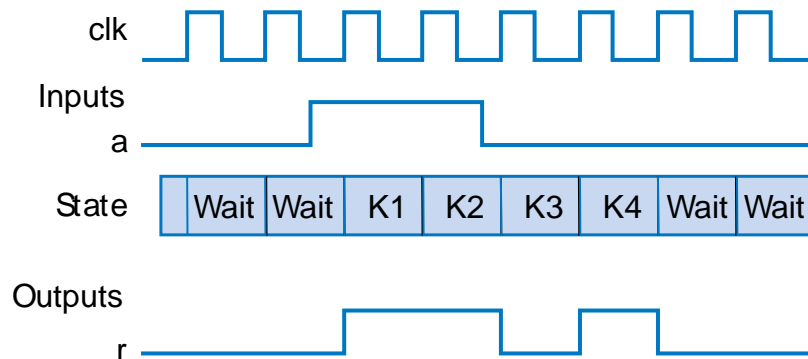
FSM Example: Secure Car Key

- 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)

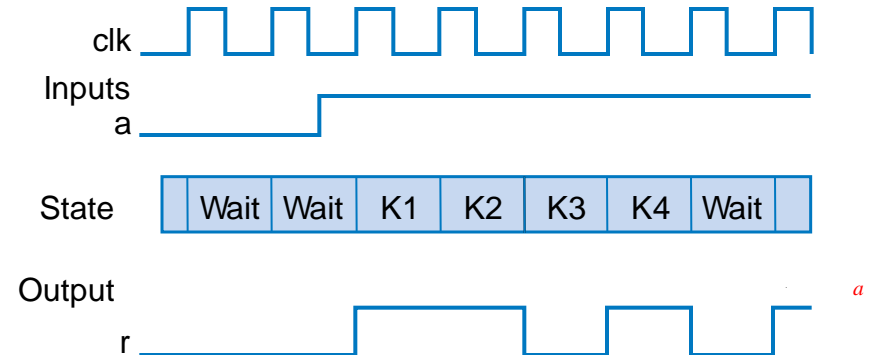


FSM Example: Secure Car Key (cont.)

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

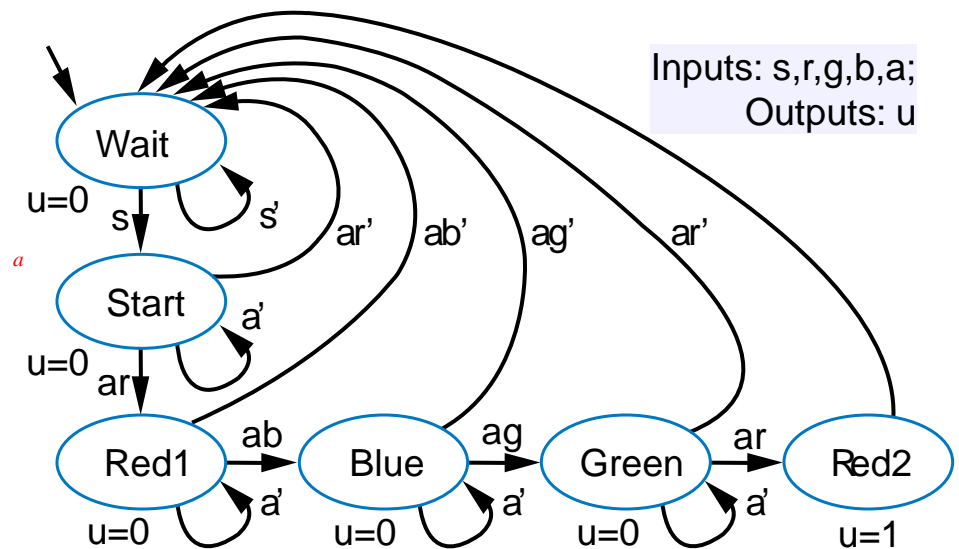
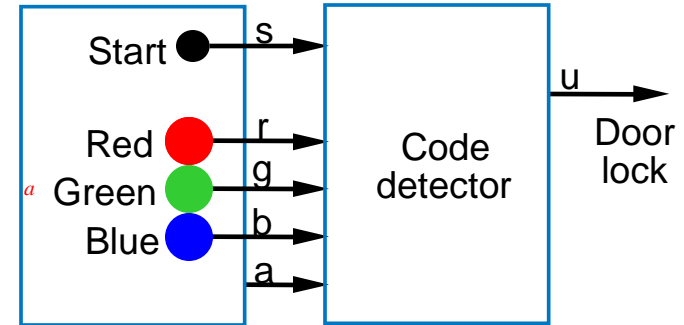


Q: Determine states and r value for given input waveform:



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



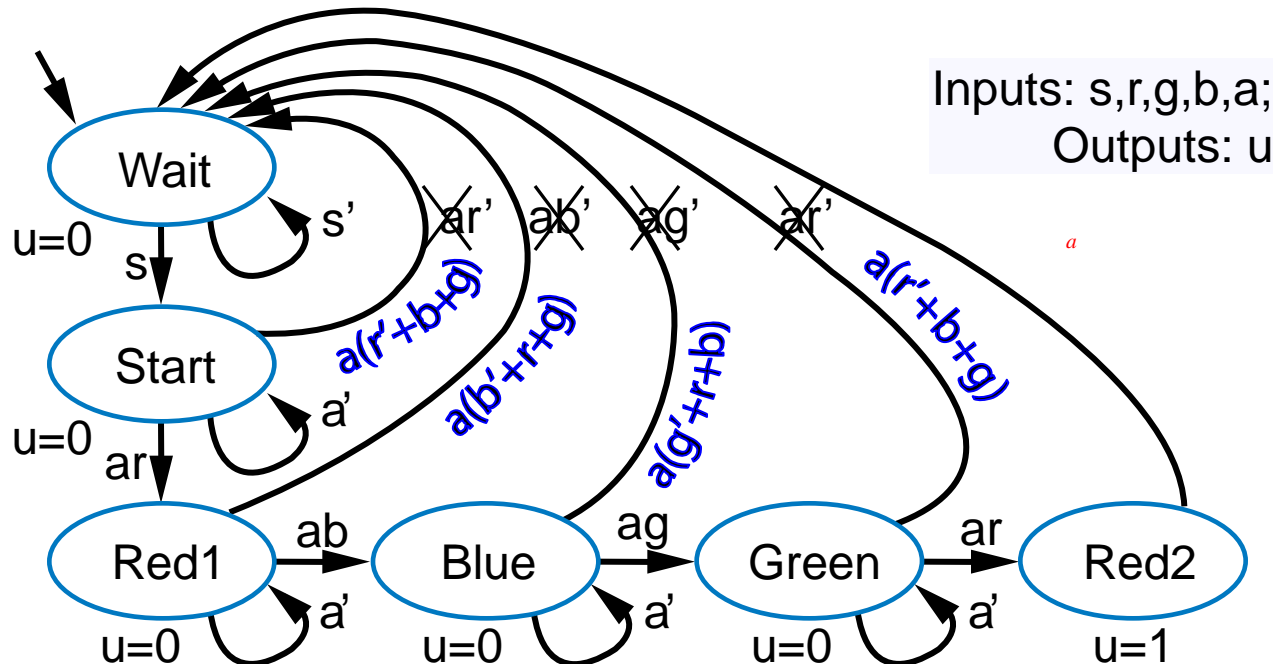
Q: Can you trick this FSM to open the door, without knowing the code?

A: Yes, hold all buttons simultaneously



Improve FSM for Code Detector

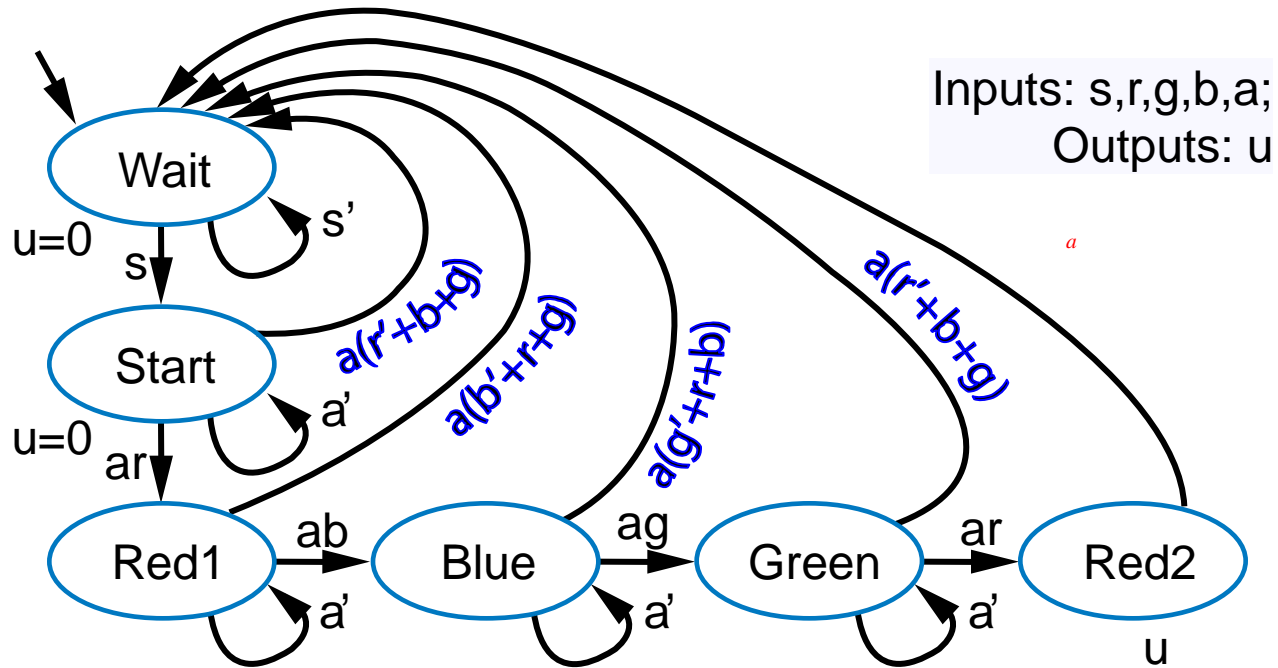
Inputs: b; Outputs: x



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



Alternate Notation for Outputs

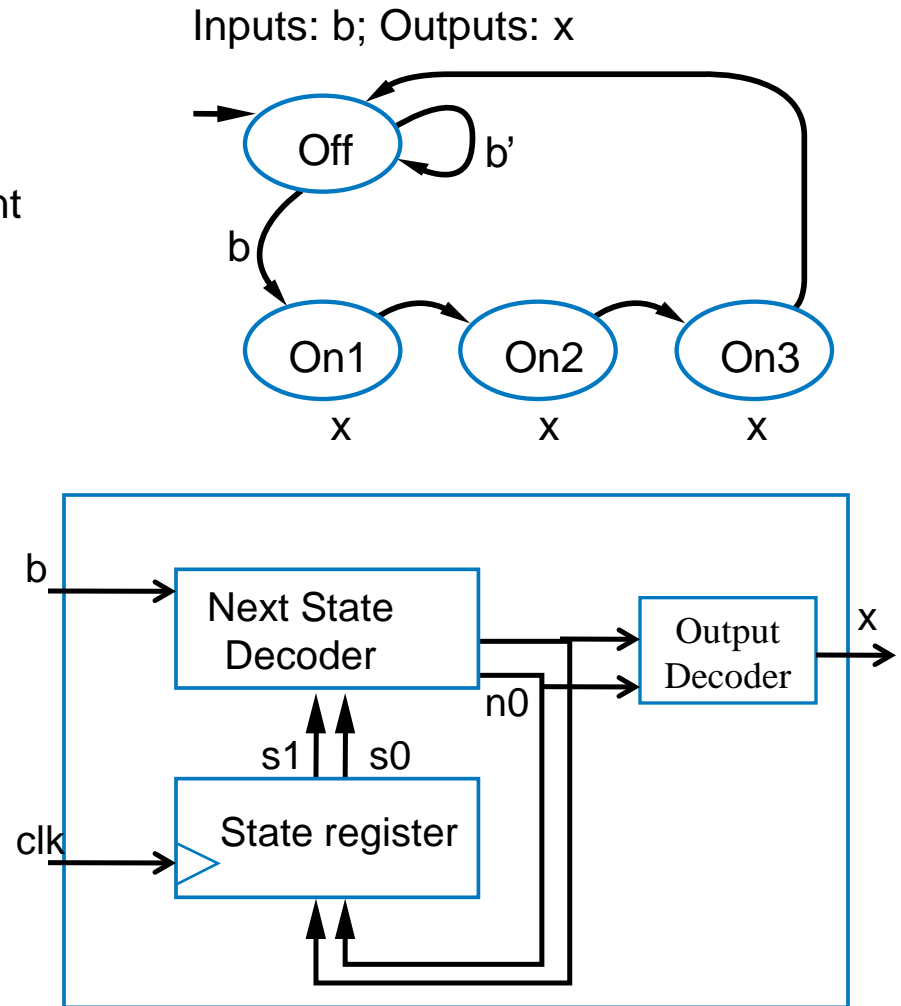
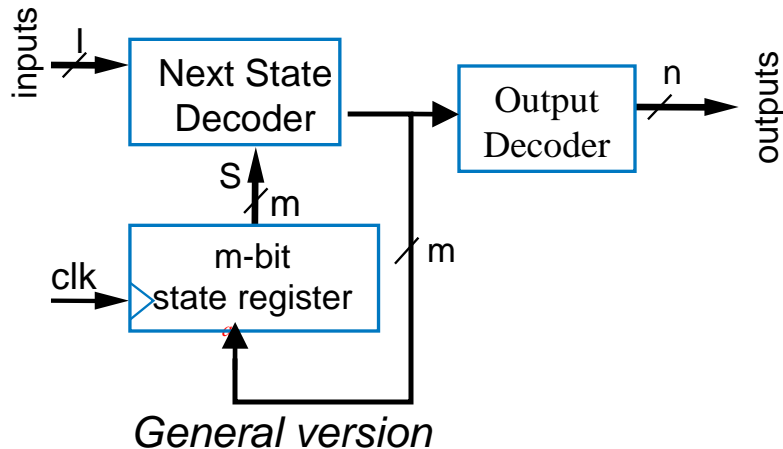


- Only draw outputs next to the state they are asserted

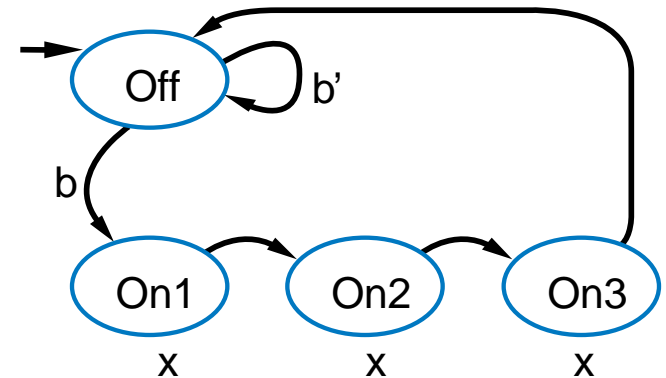


Standard Controller Architecture

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



Inputs: b; Outputs: x



Controller Design

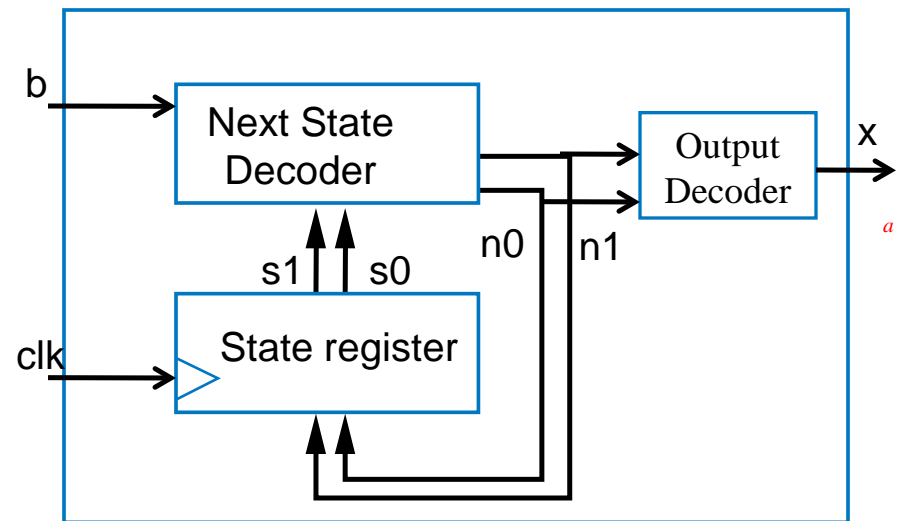
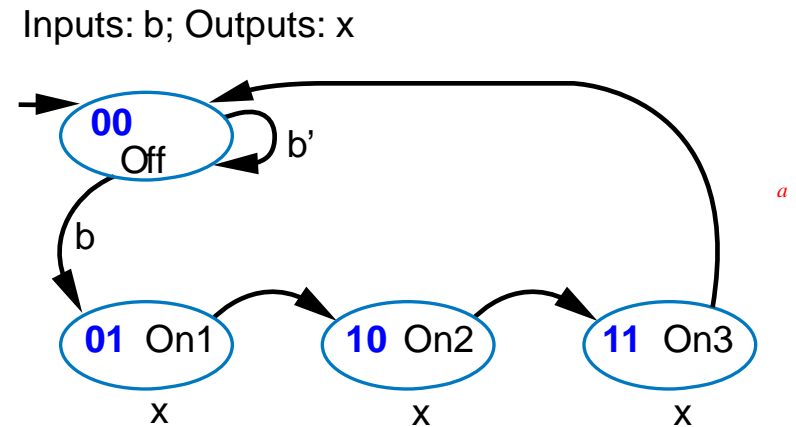
- Five step controller design process

	Step	Description
Step 1	<i>Capture the FSM</i>	Create an FSM that describes the desired behavior of the controller.
Step 2	<i>Create the architecture</i>	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	<i>Encode the states</i>	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	<i>Create the state table</i>	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	<i>Implement the combinational logic</i>	Implement the combinational logic using any method.

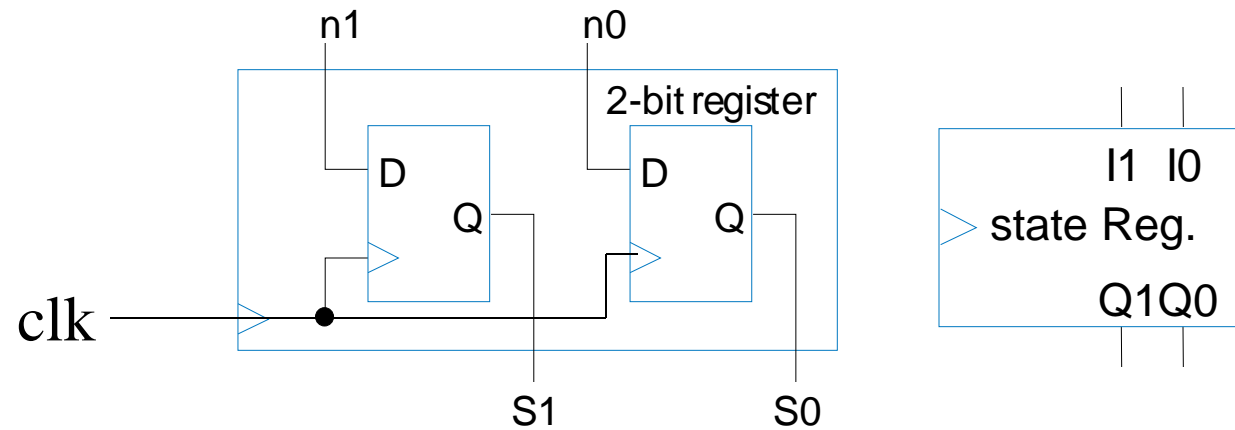


Controller Design: Laser Timer Example

- Step 1: Capture the FSM
 - Described using a state diagram
 - Use meaningful names for the states
- 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



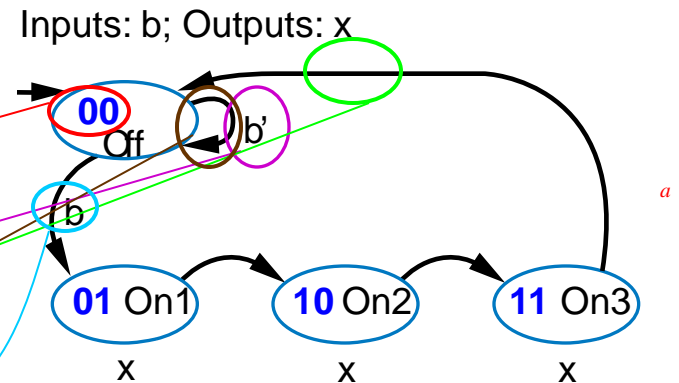
What is a State Register?



Controller Design: Laser Timer Example (cont)

- Step 4: Create state table

	Inputs			Outputs		
	s1	s0	b	x	n1	n0
<i>Off</i>	0	0	0	0	0	0
	0	0	1	0	0	1
<i>On1</i>	0	1	0	1	1	0
	0	1	1	1	1	0
<i>On2</i>	1	0	0	1	1	1
	1	0	1	1	1	1
<i>On3</i>	1	1	0	1	0	0
	1	1	1	1	0	0



Controller Design: Laser Timer Example (cont)

- Step 5: Implement combinational logic

present state before clock transistion

next state after clock transistion ^a

	Inputs			Outputs		
	s1	s0	b	x	n1	n0
<i>Off</i>	0	0	0	0	0	0
	0	0	1	0	0	1
<i>On1</i>	0	1	0	1	1	0
	0	1	1	1	1	0
<i>On2</i>	1	0	0	1	1	1
	1	0	1	1	1	1
<i>On3</i>	1	1	0	1	0	0
	1	1	1	1	0	0

$$x = s1 + s0 \text{ (note from the table that } x=1 \text{ if } s1 = 1 \text{ or } s0 = 1)$$

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

$$n1 = s1's0 + s1s0'$$

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

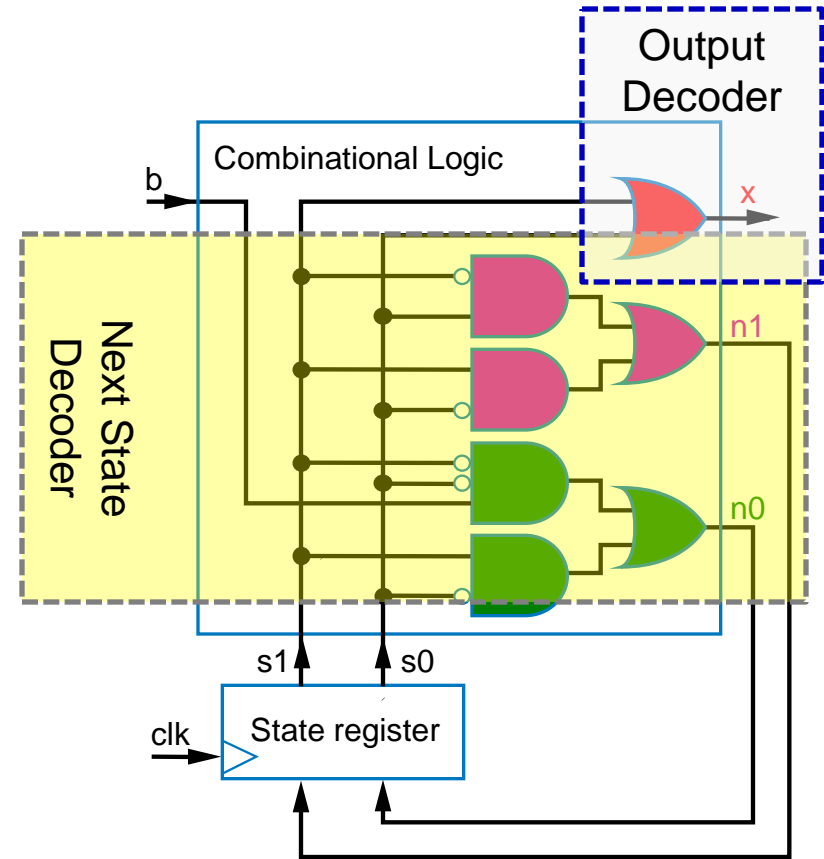
$$n0 = s1's0'b + s1s0'$$



Controller Design: Laser Timer Example (cont)

- Step 5: Implement combinational logic

	Inputs			Outputs		
	s1	s0	b	x	n1	n0
<i>Off</i>	0	0	0	0	0	0
	0	0	1	0	0	1
<i>On1</i>	0	1	0	1	1	0
	0	1	1	1	1	0
<i>On2</i>	1	0	0	1	1	1
	1	0	1	1	1	1
<i>On3</i>	1	1	0	1	0	0
	1	1	1	1	0	0



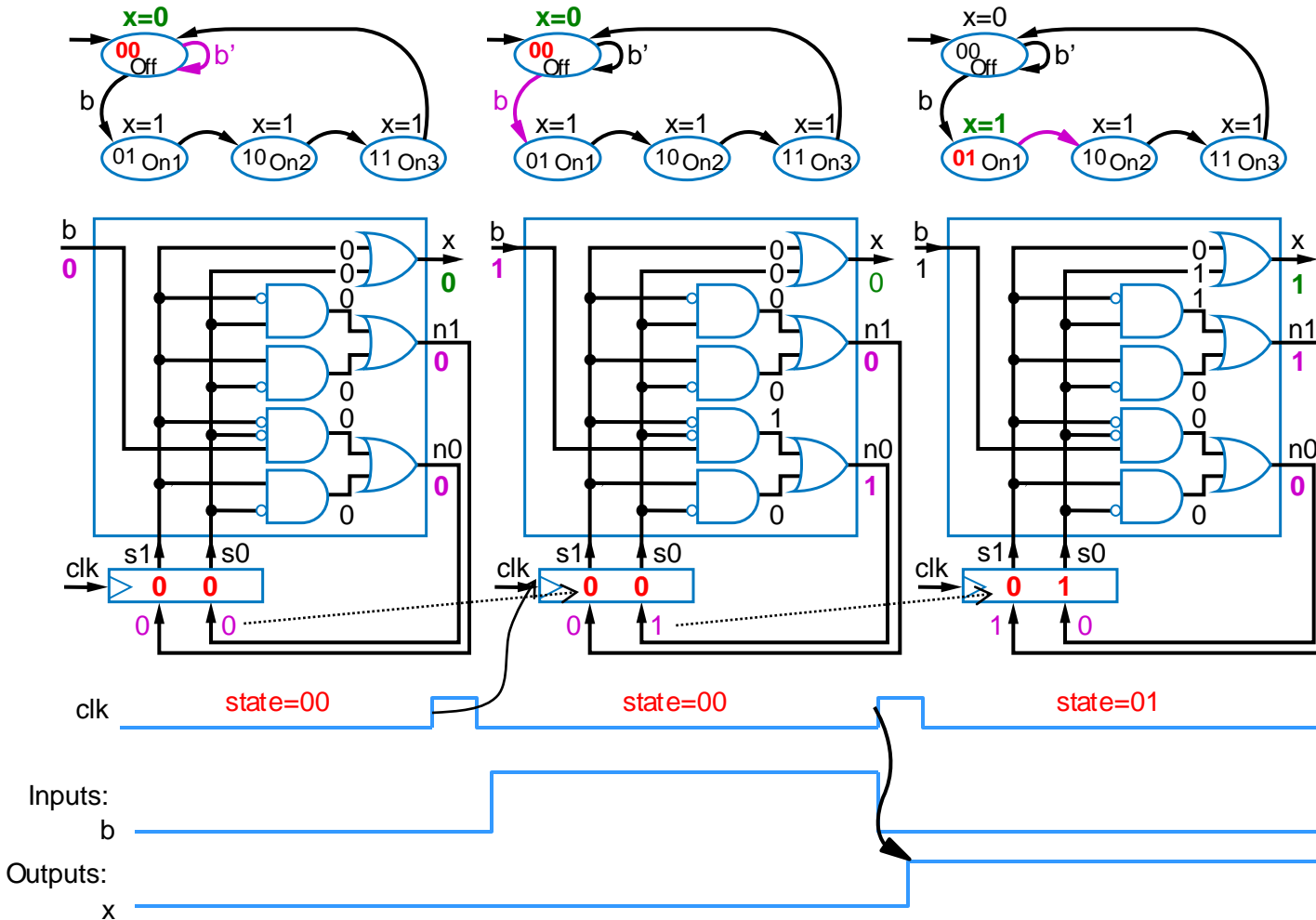
$$x = s1 + s0$$

$$n1 = s1's0 + s1s0'$$

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



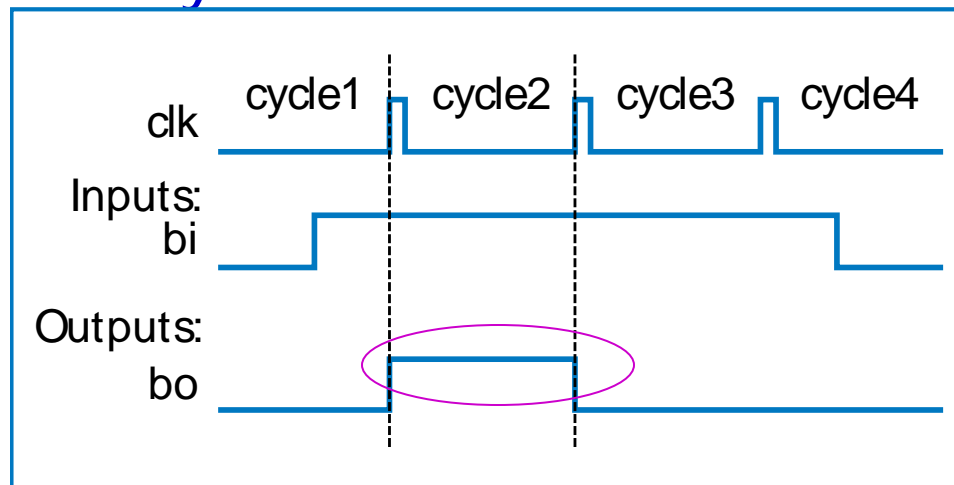
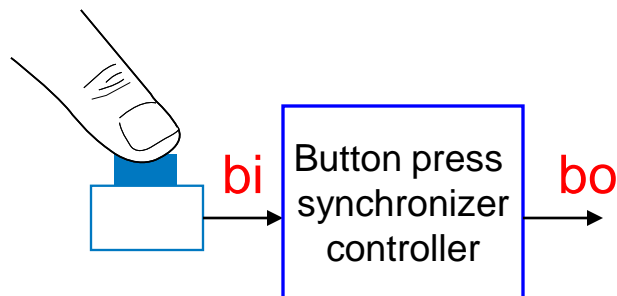
Understanding the Controller's Behavior



a



Controller Example: Button Press Synchronizer

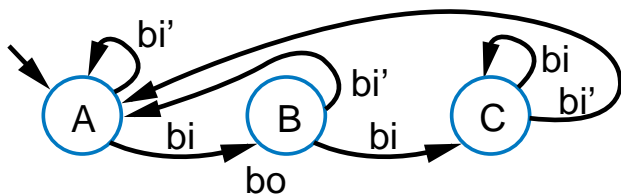


- Want simple sequential circuit that converts button press to single cycle duration, regardless of length of time that button actually pressed
 - We assumed such an ideal button press signal in earlier example, like the button in the laser timer controller



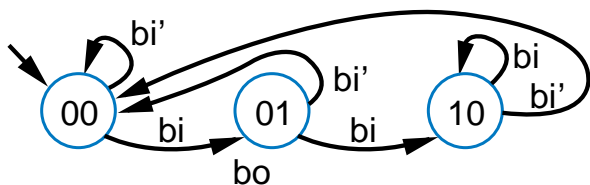
Controller Example: Button Press Synchronizer (cont)

FSM inputs: bi; FSM outputs: bo

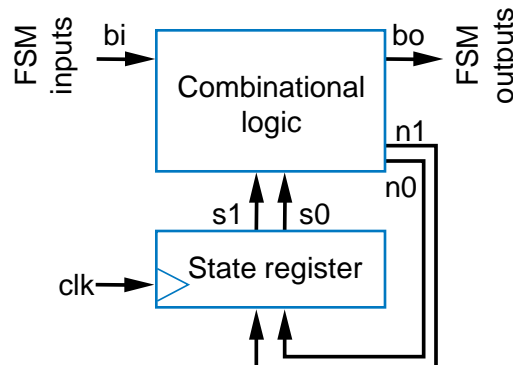


Step 1: FSM

FSM inputs: bi; FSM outputs: bo



Step 3: Encode states

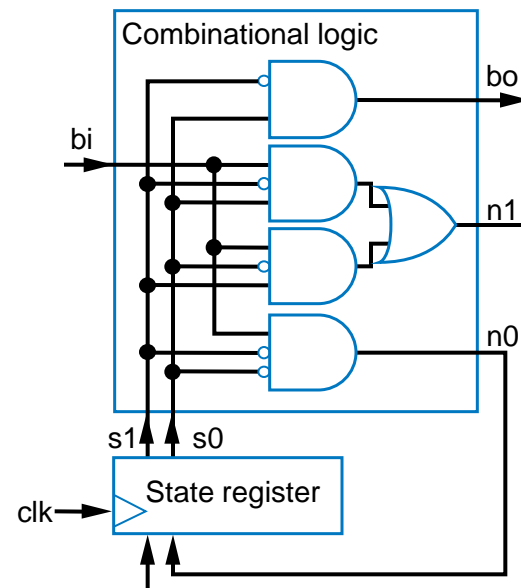


Step 2: Create architecture

$$n1 = s1's0bi + s1s0bi$$

$$n0 = s1's0'bi$$

$$bo = s1's0bi' + s1's0bi = s1s0$$



Step 5: Create
combinational circuit

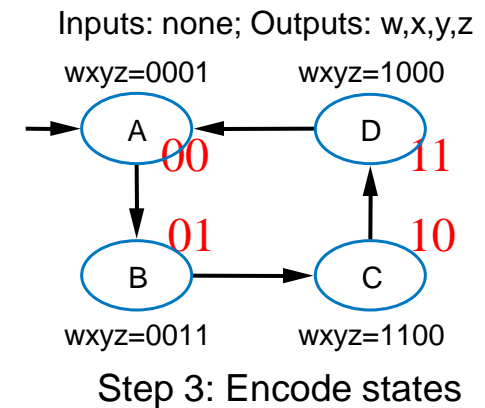
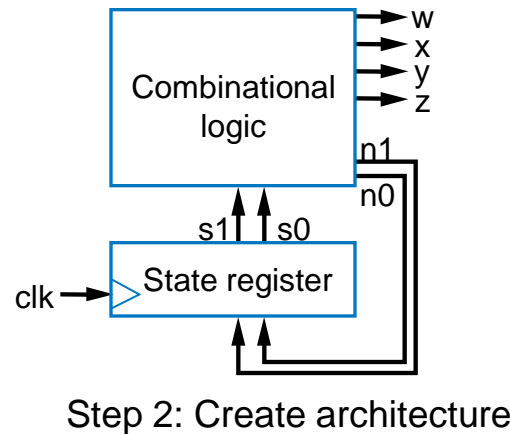
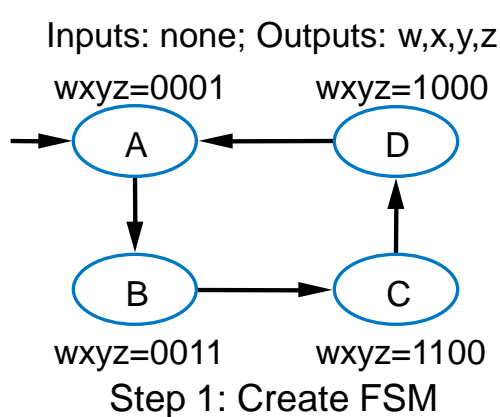
	Combinational logic Inputs			Outputs		
	s1	s0	bi	n1	n0	bo
A	0	0	0	0	0	0
	0	0	1	0	1	0
B	0	1	0	0	0	1
	0	1	1	1	0	1
C	1	0	0	0	0	0
	1	0	1	1	0	0
unused	1	1	0	0	0	0
	1	1	1	0	0	0

Step 4: State table



Controller Example: Sequence Generator

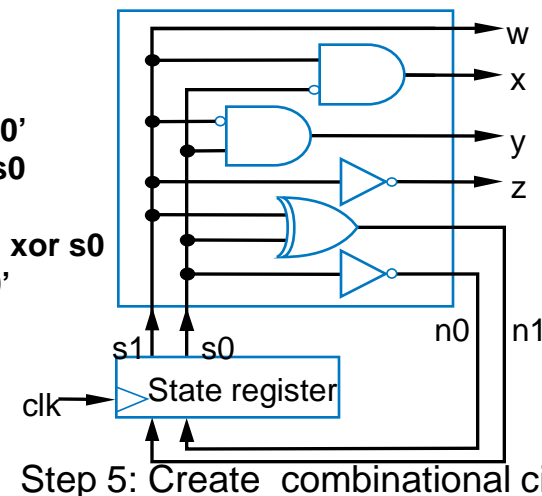
- Want generate sequence 0001, 0011, 1100, 1000, (repeat)
 - Each value for one clock cycle
 - Common, e.g., to create pattern in 4 lights, or control magnets of a “stepper motor”



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

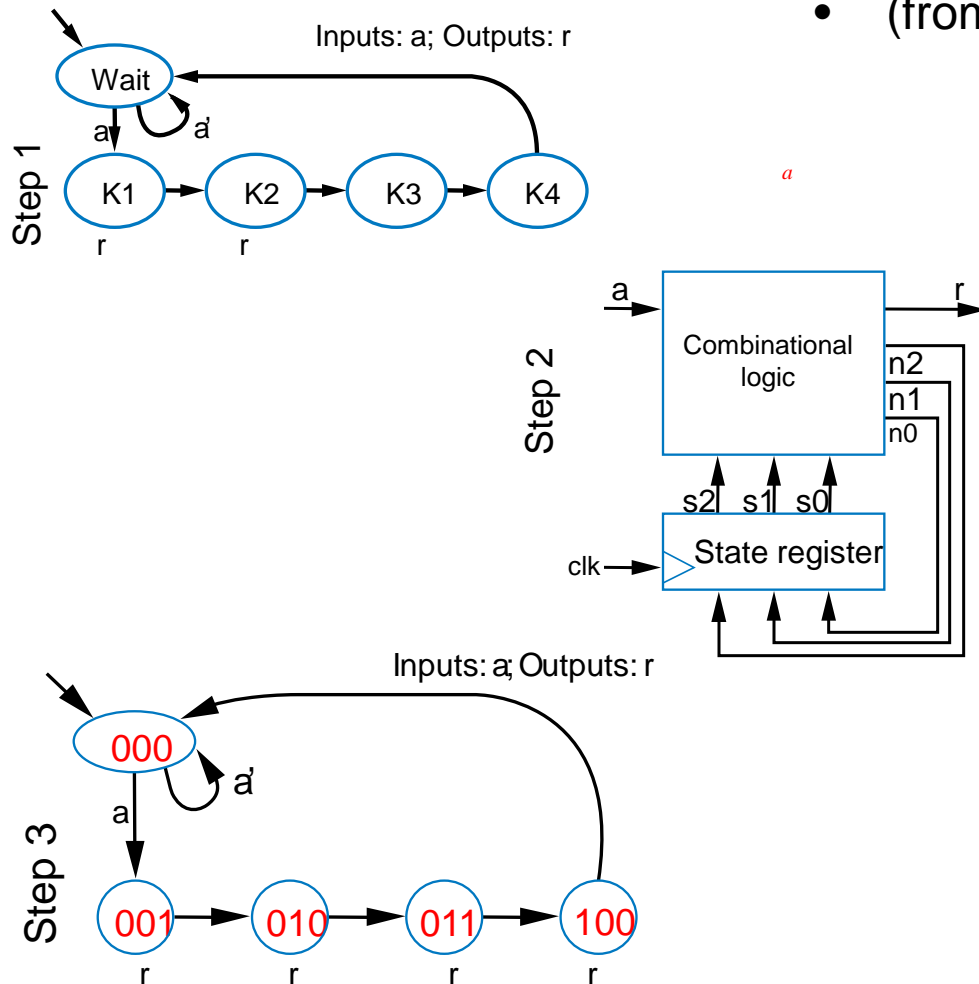
Step 4: Create state table

$$\begin{aligned}
 w &= s1 \\
 x &= s1s0' \\
 y &= s1's0 \\
 z &= s1' \\
 n1 &= s1 \text{ xor } s0 \\
 n0 &= s0'
 \end{aligned}$$



Controller Example: Secure Car Key

- (from earlier example)



	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

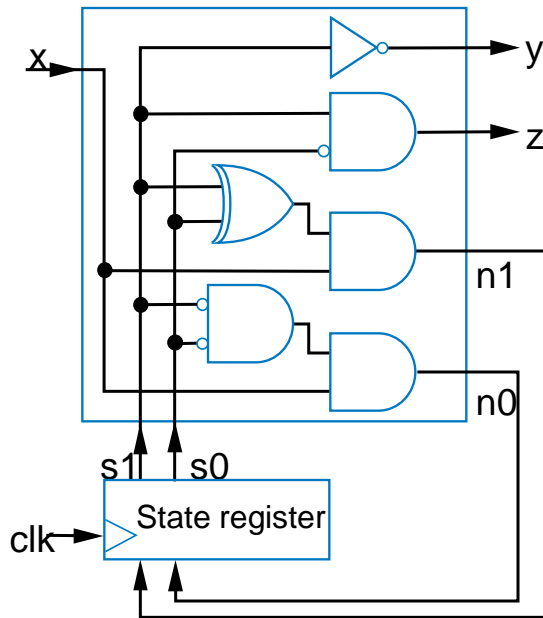
Step 4

We'll omit Step 5



Example: Seq. Circuit to FSM (Reverse Engineering)

What does this circuit do?

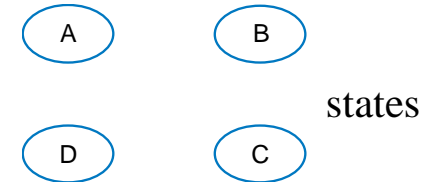


Work backwards

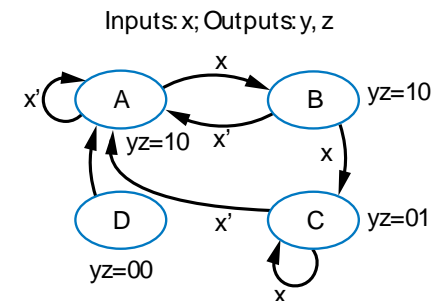
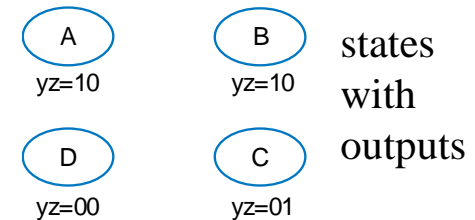
$$\begin{aligned} y &= s1' \\ z &= s1s0' \\ n1 &= (s1 \text{ xor } s0)x \\ n0 &= (s1' * s0')x \end{aligned}$$

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

Pick any state names you want



Outputs: y, z

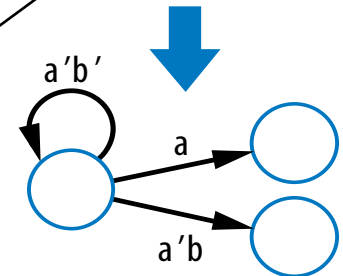
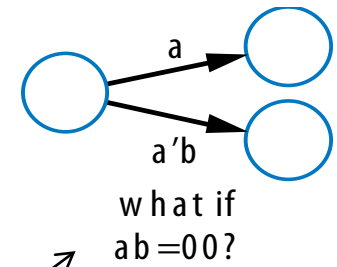
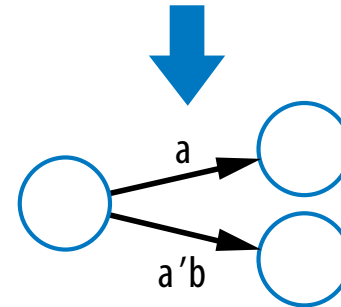
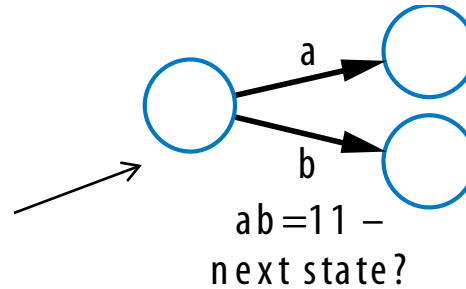


states with
outputs and
transitions



Common Pitfalls Regarding Transition Properties

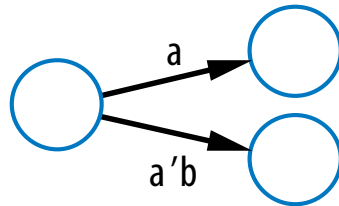
- *Only* one condition should be true
 - For all transitions leaving a state
 - Else, which one?
- *One* condition must be true
 - For all transitions leaving a state
 - Else, where go?



Verifying Correct Transition Properties

- Can verify using Boolean algebra

- Only one condition true: AND of each condition pair (for transitions leaving a state) should equal 0 \rightarrow proves pair can never simultaneously be true
- One condition true: OR of all conditions of transitions leaving a state) should equal 1 \rightarrow proves at least one condition must be true
- Example



Answer:

$$\begin{aligned} & a * a'b \\ &= (a * a') * b \\ &= 0 * b \\ &= 0 \\ &\text{OK!} \end{aligned}$$

$$\begin{aligned} & a + a'b \\ &= a*(1+b) + a'b \\ &= a + ab + a'b \\ &= a + (a+a')b \\ &= a + b \end{aligned}$$

Fails! Might not be 1 (i.e., $a=0$, $b=0$)

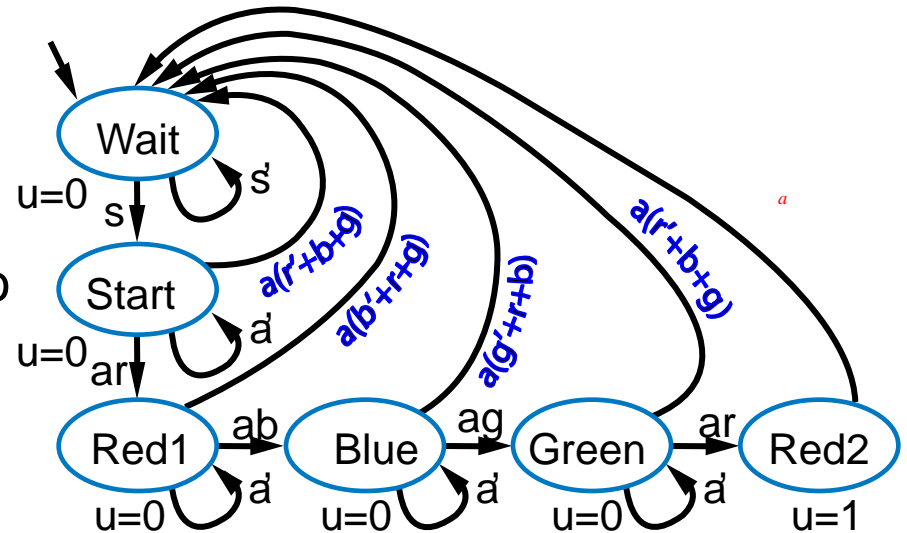
Q: For shown transitions, prove whether:

- * Only one condition true (AND of each pair is always 0)
- * One condition true (OR of all transitions is always 1)



Evidence that Pitfall is Common

- Recall code detector FSM
 - We “fixed” a problem with the transition conditions
 - Do the transitions obey the two required transition properties?
 - Consider transitions of state *Start*, and the “only one true” property



$$\begin{aligned} ar * a' &= (a * a')r \\ &= 0 \end{aligned}$$

$$\begin{aligned} a' * a(r'+b+g) &= 0 * r \\ &= 0 \end{aligned}$$

$$\begin{aligned} ar * a(r'+b+g) &= (a' * a) * (r' + b + g) = 0 * (r' + b + g) \\ &= (a * a) * r * (r' + b + g) = a * r * (r' + b + g) \\ &= arr' + arb + arg \\ &= 0 + arb + arg \\ &= arb + arg \\ &= ar(b+g) \end{aligned}$$

Fails! Means that two of Start's transitions could be true

Intuitively: press red and blue buttons at same time: conditions ar , and $a(r'+b+g)$ will both be true. Which one should be taken?

Q: How to solve?

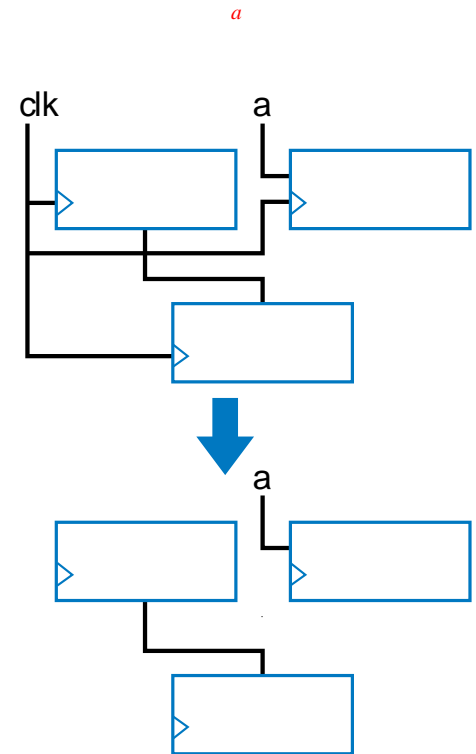
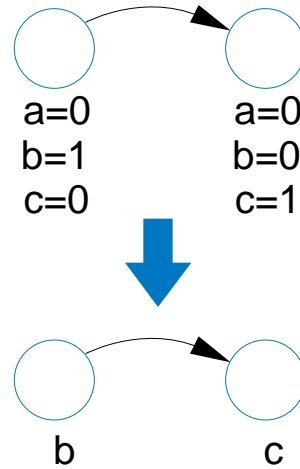
A: ar should be $arb'g'$ (likewise for ab , ag , ar)

Note: As evidence the pitfall is common, we admit the mistake was not intentional. A reviewer of the book caught it.



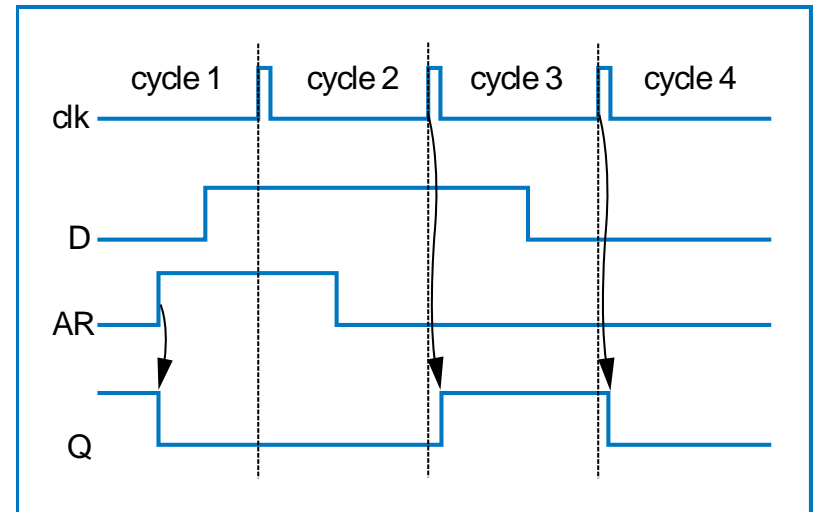
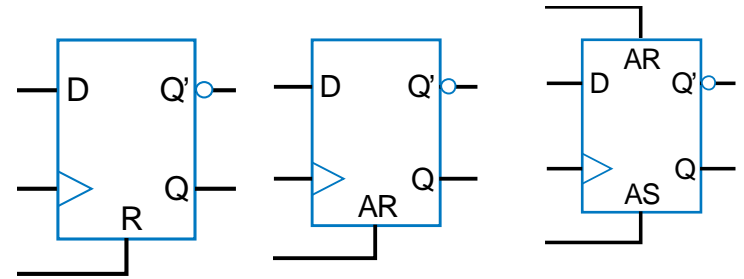
Simplifying Notations

- FSMs
 - Assume unassigned output implicitly assigned 0
- Sequential circuits
 - Assume unconnected clock inputs connected to same external clock



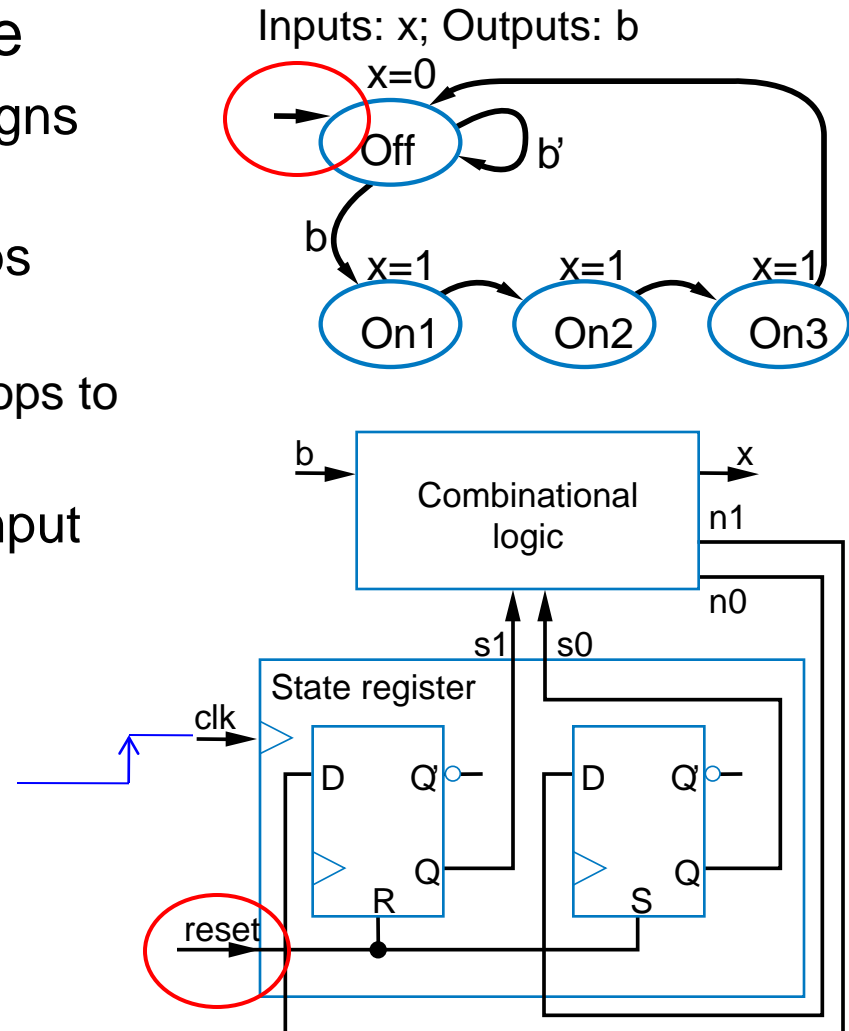
Flip-Flop Set and Reset Inputs

- Some flip-flops have additional inputs
 - Synchronous reset: clears Q to 0 on next clock edge
 - Synchronous set: sets Q to 1 on next clock edge
 - Asynchronous reset: clear Q to 0 immediately (not dependent on clock edge)
 - Example timing diagram shown
 - Asynchronous set: set Q to 1 immediately

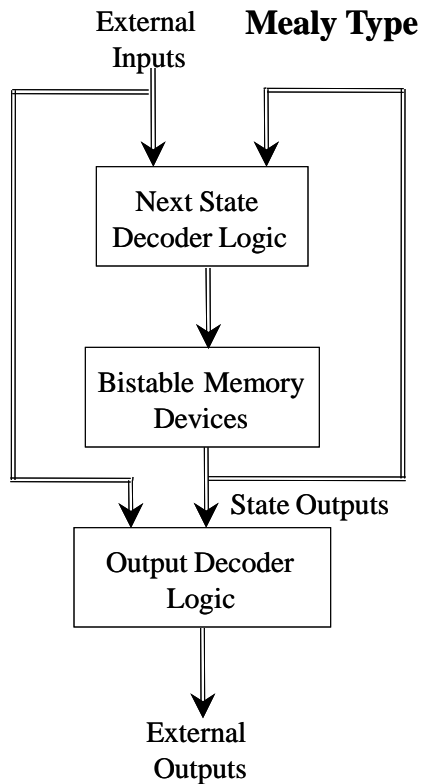
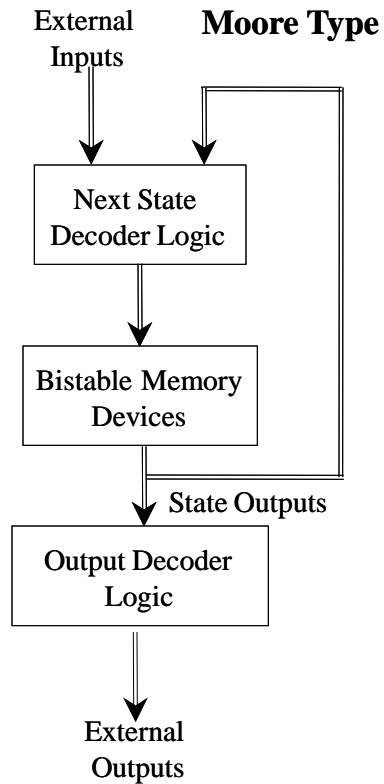


Initial State of a Controller

- All our FSMs had initial state
 - But our sequential circuit designs did not
 - Can accomplish using flip-flops with reset/set inputs
 - Shown circuit initializes flip-flops to 01
 - Designer must ensure reset input is 1 during power up of circuit
 - By electronic circuit design



Moore and Mealy FSM's



Chapter Summary

- Sequential circuits
 - Have state
- Created robust bit-storage device: D flip-flop
 - Put several together to build register, which we used to hold state
- Defined FSM formal model to describe sequential behavior
 - Using solid mathematical models -- Boolean equations for combinational circuit, and FSMs for sequential circuits -- is very important.
- Defined 5-step process to convert FSM to sequential circuit
 - Controller
- So now we know how to build the class of sequential circuits known as controllers

