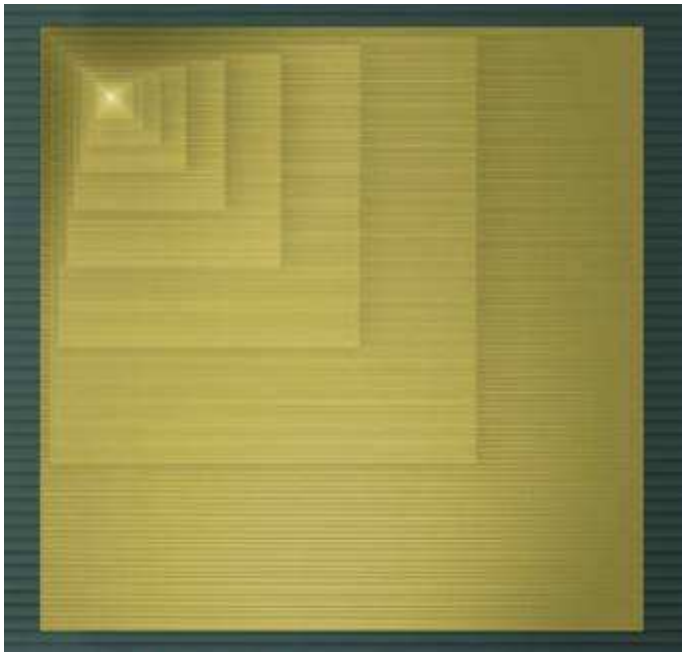


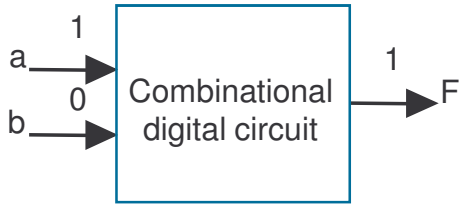
Digital Design

Sequential Logic Design -- Controllers

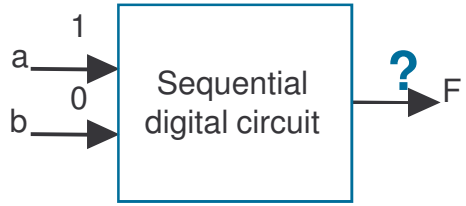
Slides to accompany the textbook *Digital Design*, First Edition,
by Frank Vahid, John Wiley and Sons Publishers, 2007.
http://www.ddvahid.com

Introduction

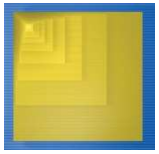
- Sequential circuit
  - Output depends not just on present inputs (as in combinational circuit), but on past sequence of inputs
    - Stores bits, also known as having “state”
  - Simple example: a circuit that counts up in binary



- In this chapter, we will:
  - Design a new building block, a **flip-flop**, that stores one bit
  - Combine that block to build multi-bit storage – a **register**
  - Describe the sequential behavior using a **finite state machine**
  - Convert a finite state machine to a **controller** – a sequential circuit having a register and combinational logic



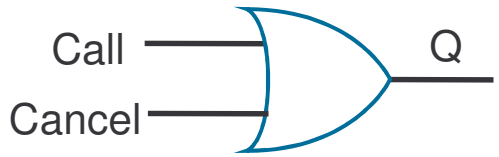
Must know sequence of past inputs to know output



Note: Slides with animation are denoted with a small red "a" near the animated items

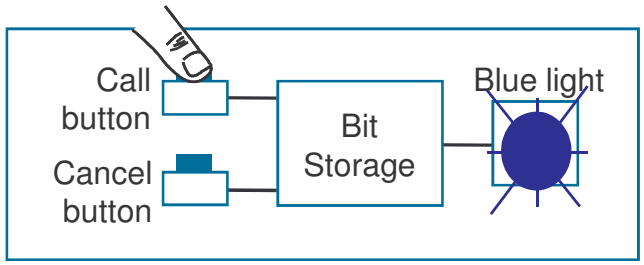
Example Needing Bit Storage

- Flight attendant call button
  - Press call: light turns on
    - ***Stays on*** after button released
  - Press cancel: light turns off
  - Logic gate circuit to implement this?

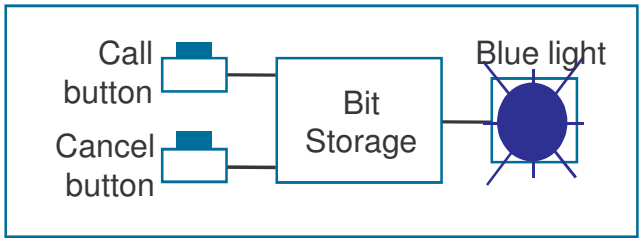


Doesn't work. Q=1 when Call=1, but doesn't stay 1 when Call returns to 0

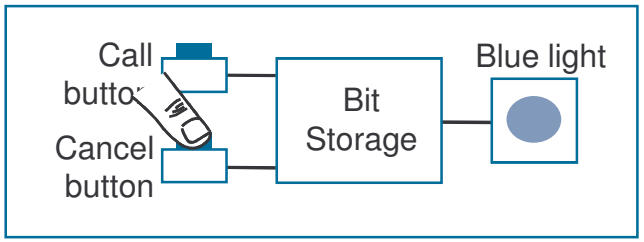
Need some form of "feedback" in the circuit



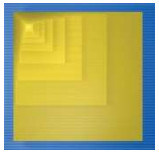
1. Call button pressed – light turns on



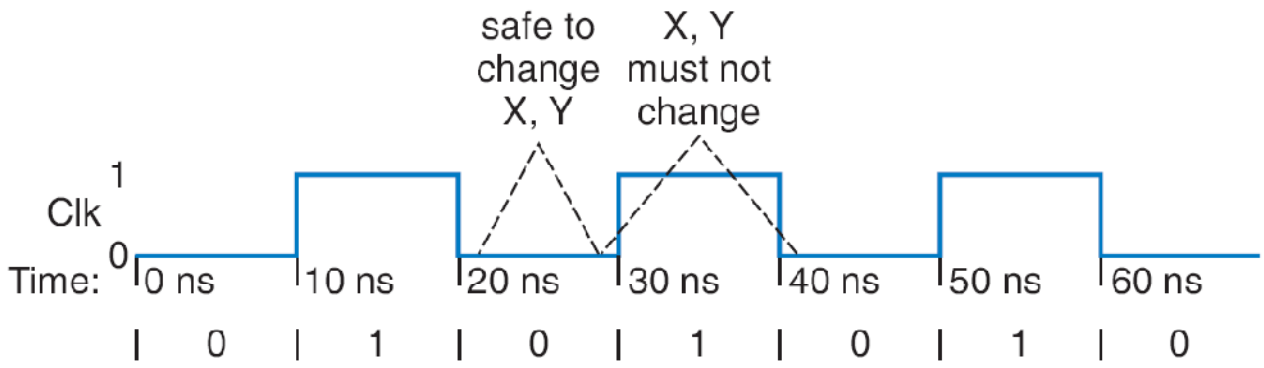
*2. Call button released – light **stays on***



3. Cancel button pressed – light turns off

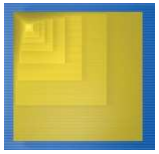


Clocks



- ***Clock period***: time interval between pulses
  - Above signal: period = 20 ns
- ***Clock cycle***: one such time interval
  - Above signal shows 3.5 clock cycles
- ***Clock frequency***: 1/period
  - Above signal: frequency = 1 / 20 ns = 50 MHz
    - 1 Hz = 1/s

Freq	Period
100 GHz	0.01 ns
10 GHz	0.1 ns
1 GHz	1 ns
100 MHz	10 ns
10 MHz	100 ns



Finite-State Machines (FSMs) and Controllers

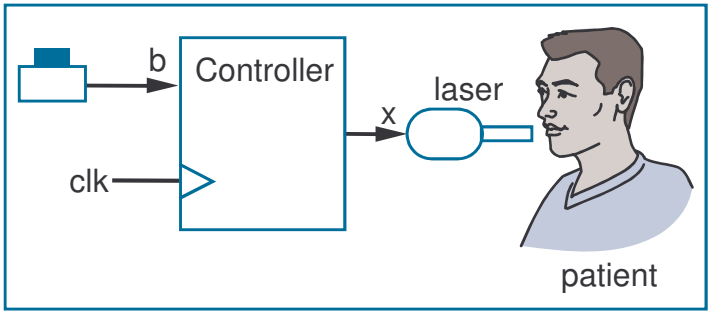
- Want sequential circuit with particular behavior over time

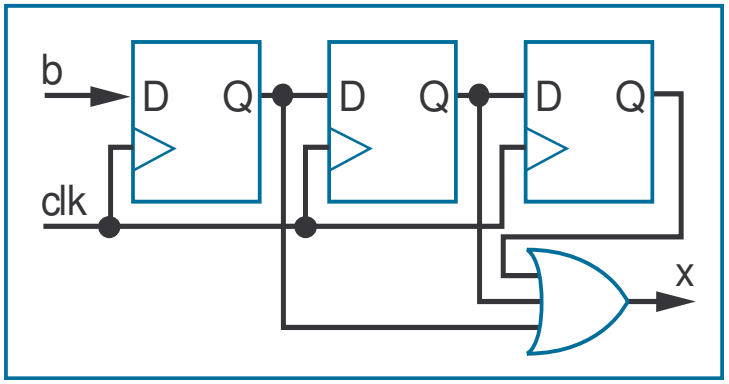
- Example: Laser timer
  - Push button: x=1 for 3 clock cycles

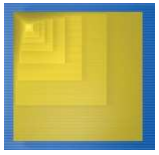
  - How? Let's try three flip-flops
    - b=1 gets stored in first D flip-flop

    - 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

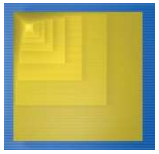






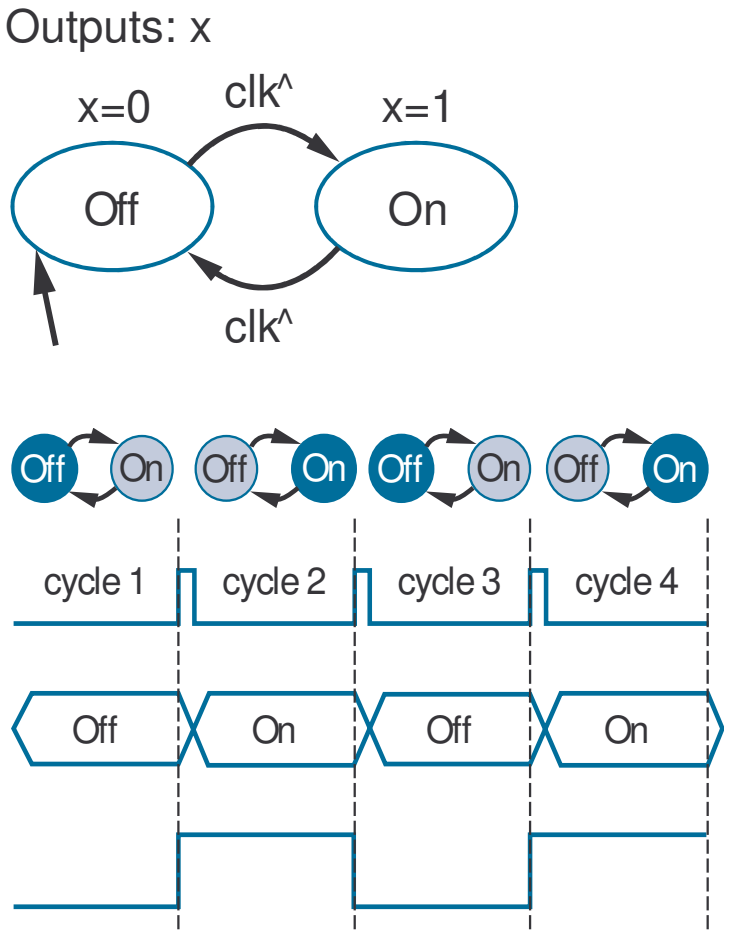
Need a Better Way to Design Sequential Circuits

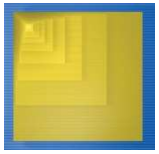
- Trial and error is not a good design method
  - Will we be able to "guess" a circuit that works for other desired behavior?
    - How about counting up from 1 to 9? Pulsing an output for 1 cycle every 10 cycles? Detecting the sequence 1 3 5 in binary on a 3-bit input?
  - And, a circuit built by guessing 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

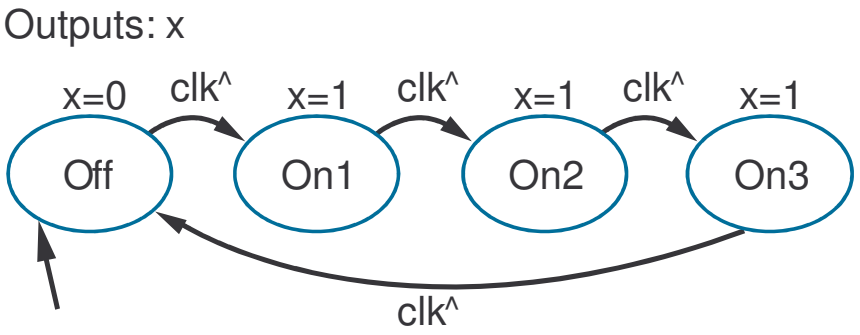
- Finite-State Machine (FSM)
  - A way to describe desired behavior of sequential circuit
    - Akin to Boolean equations for combinational behavior
  - List states, and transitions among states
    - *Example:* Make x change toggle (0 to 1, or 1 to 0) every clock cycle
    - Two states: “Off” (x=0), 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)

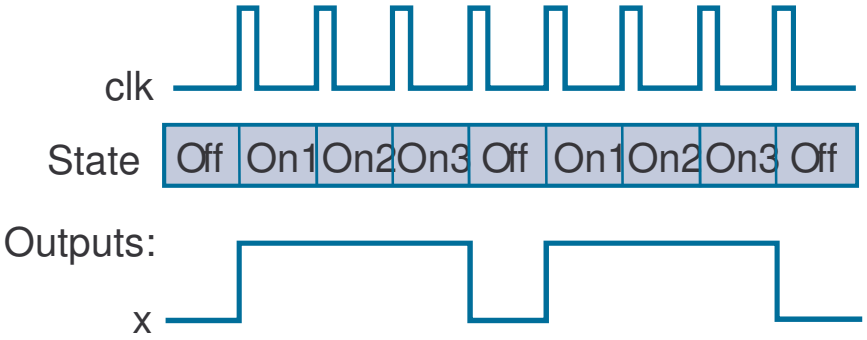


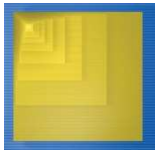


FSM Example: 0,1,1,1,repeat

- 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

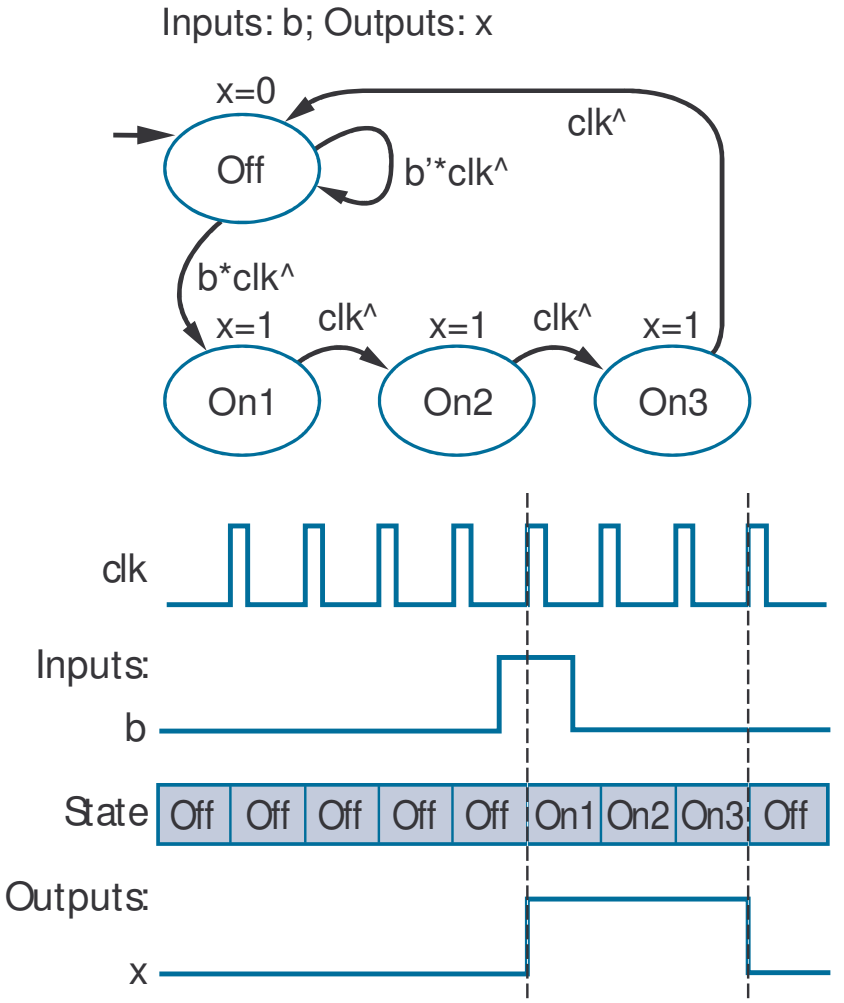


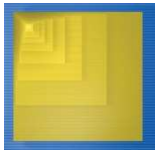




Extend FSM to Three-Cycles High Laser Timer

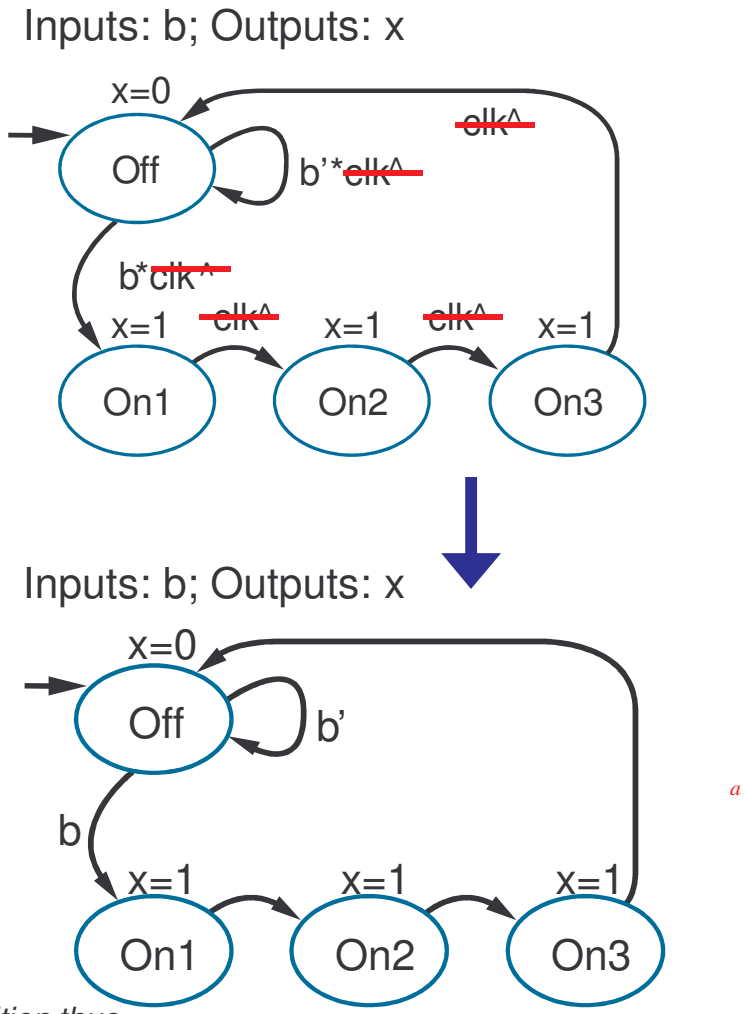
- Four states
- Wait in “Off” state while b is 0 (b’)
- When b is 1 (and rising clock edge), transition to On1
  - Sets x=1
  - On next two clock edges, transition to On2, then On3, which also set x=1
- So x=1 for three cycles after button pressed



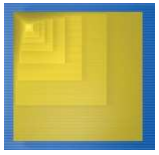


FSM Simplification: Rising Clock Edges Implicit

- Showing rising clock on every transition: *cluttered and unnecessary*
  - Make implicit -- assume every edge has rising clock, even if not shown.
    - Eg., it is understood that a transition out of a state is on a clock edge.
  - 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

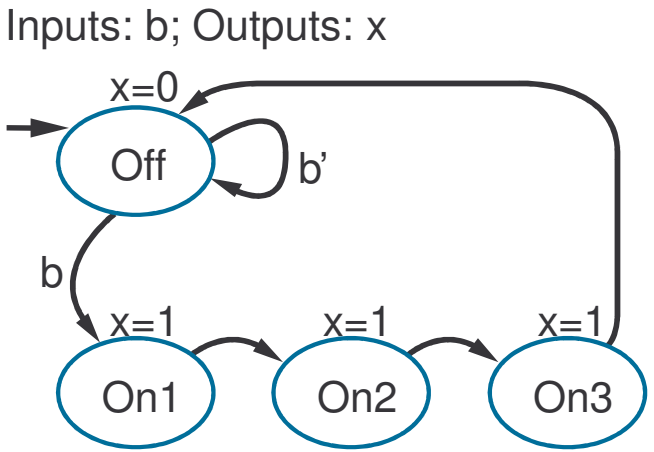


Note: Transition with no associated condition thus transistions to next state on next clock cycle



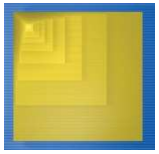
FSM Definition

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



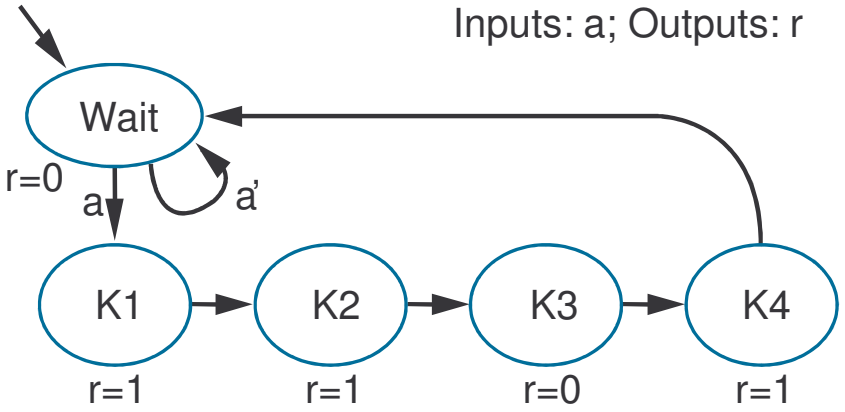
We often draw FSM graphically, known as ***state diagram***

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

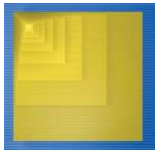


FSM Example: Secure Car Key

- 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)

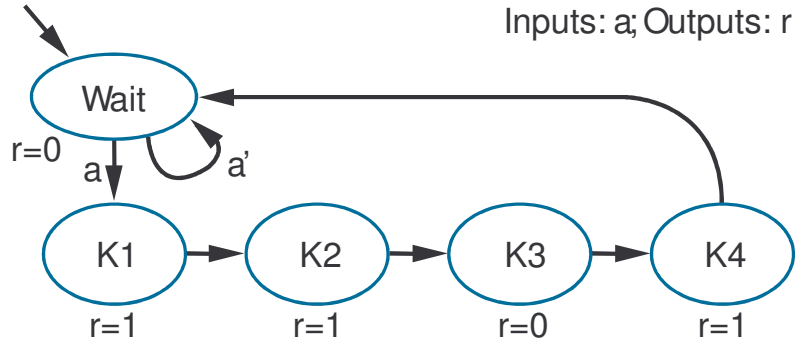


a.k.a., A Sequencer

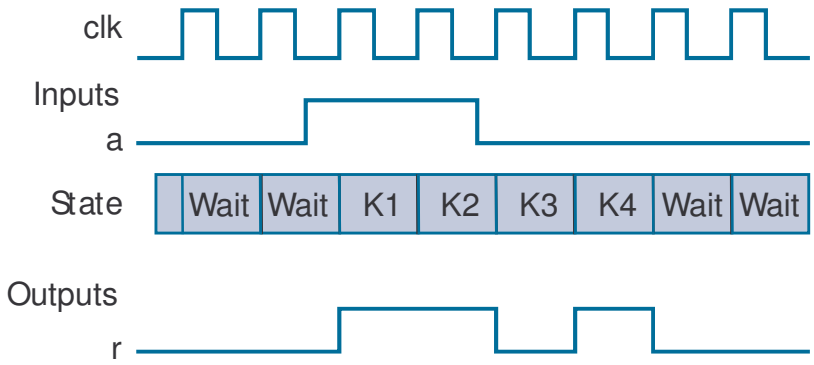


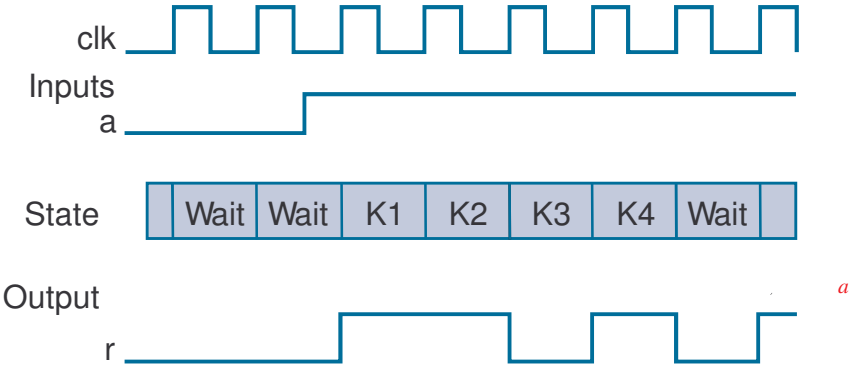
FSM Example: Secure Car Key (cont.)

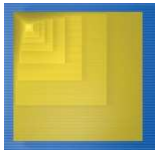
- Nice feature of FSM
  - Can evaluate output behavior for different input sequence
  - Timing diagrams show states and output values for different input waveforms



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

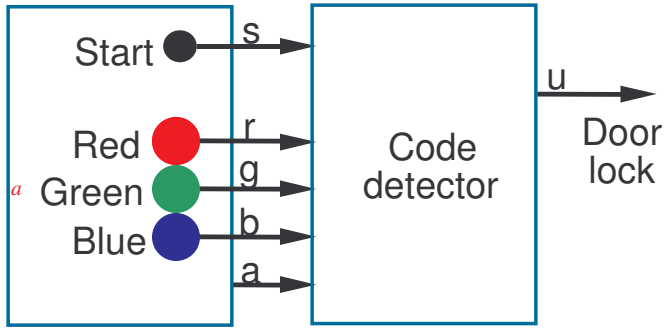


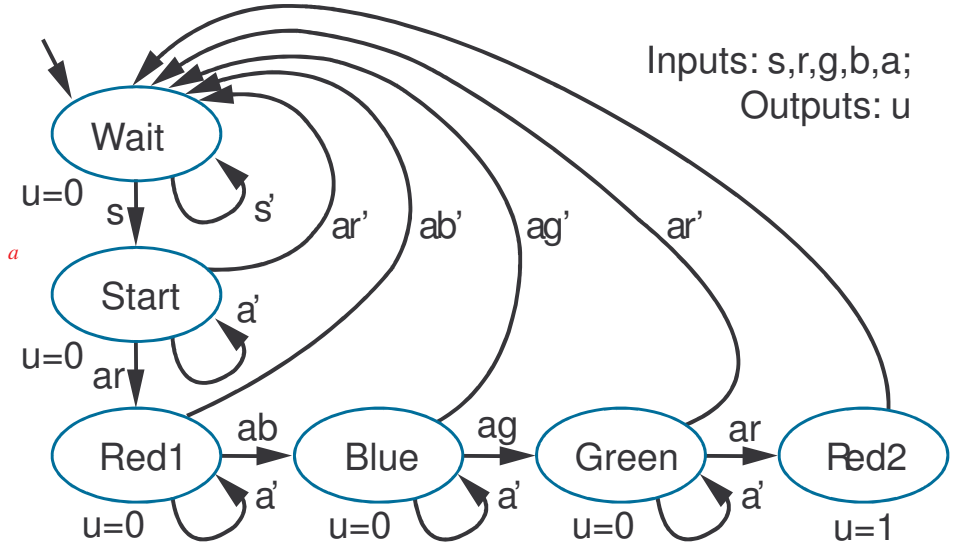




FSM Example: Code Detector

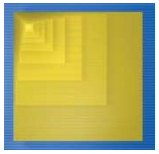
- Unlock door (u=1) only when buttons pressed in sequence:
  - start, then red, blue, green, red
- Input from each button: *s, r, g, b*
  - Also, output *a* indicates that some colored button pressed
- FSM
  - Wait for start (s=1) in "Wait"
  - Once started ("Start")
    - If see red, go to "Red1"
    - 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



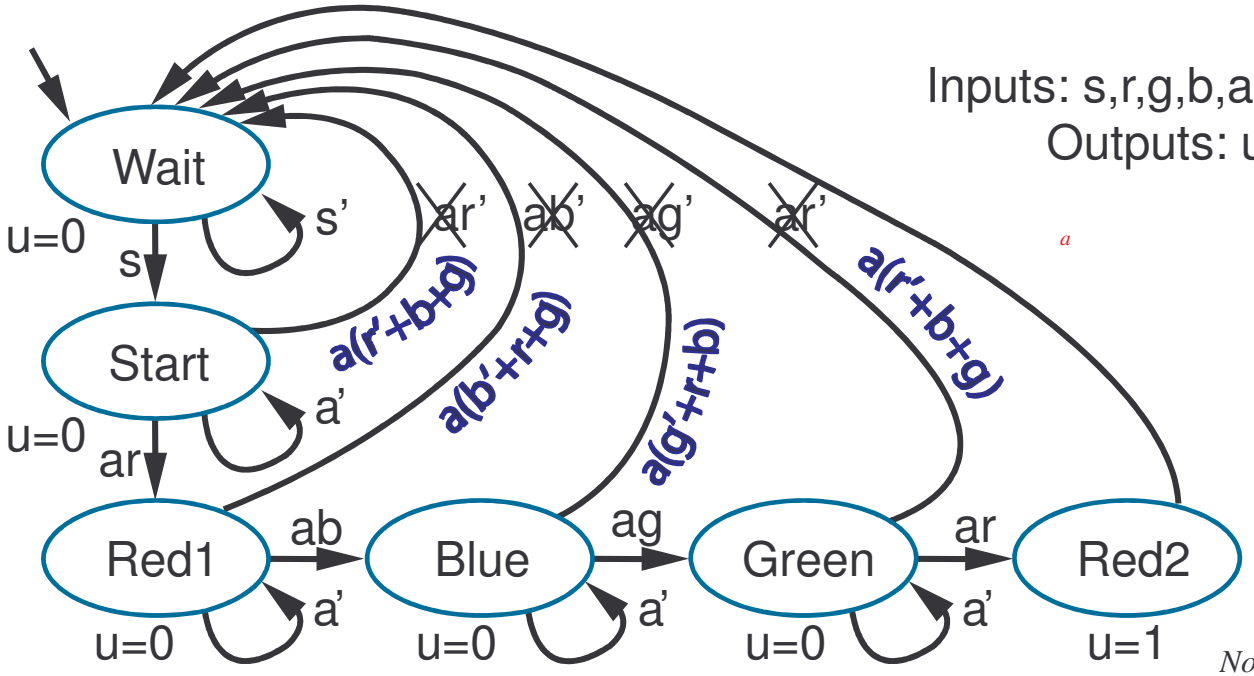


Q: Can you trick this FSM to open the door, without knowing the code?

- A: Yes, hold all buttons simultaneously

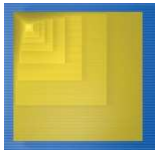


Improve FSM for Code Detector



Note: small problem still remains; we'll discuss later

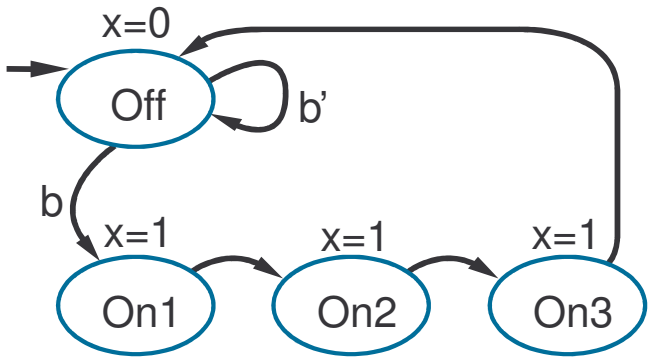
- New transition conditions detect if wrong button pressed, returns to "Wait"
- FSM provides formal, concrete means to accurately define desired behavior

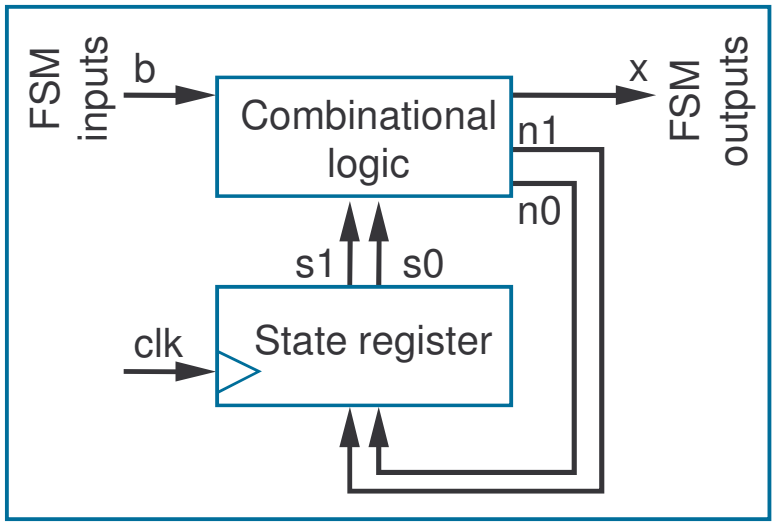


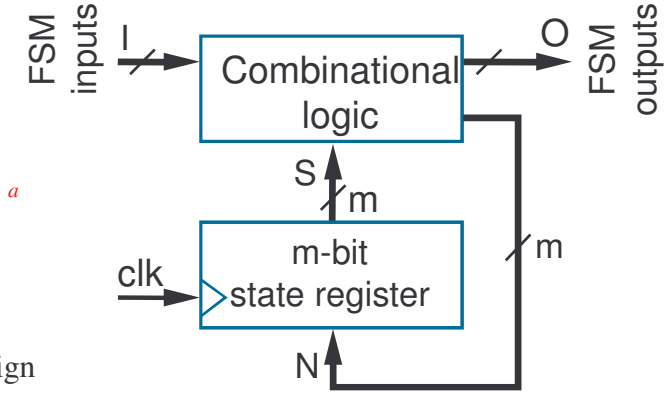
Standard Controller Architecture

- 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***

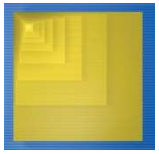
Inputs: b; Outputs: x







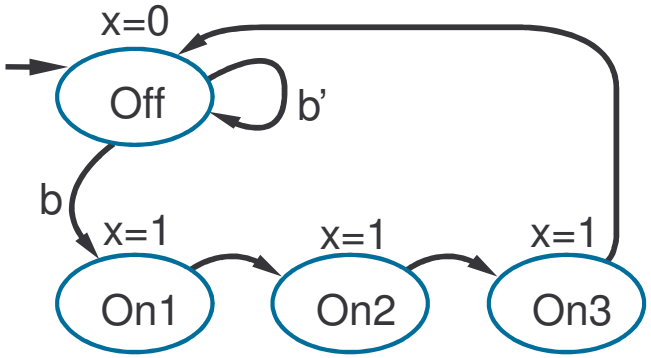
General version



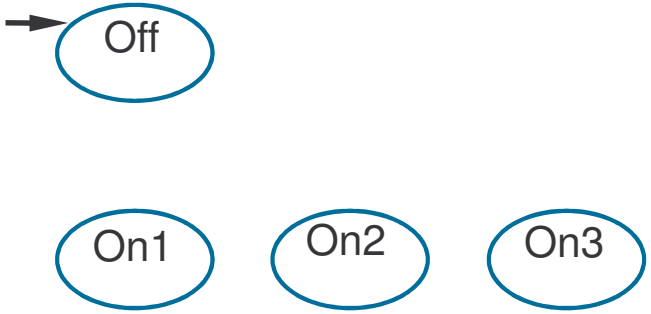
State Diagram Example

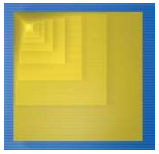
Modify the laser pulse generator state diagram (below left), so that the new FSM will only produce one pulse (3 clk cycles wide) for each actuation of ‘b’ ... or stated another way, one pulse out for one pulse in.

Inputs: b; Outputs: x



Inputs: b; Outputs: x

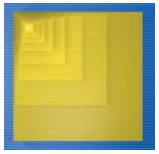




Controller Design

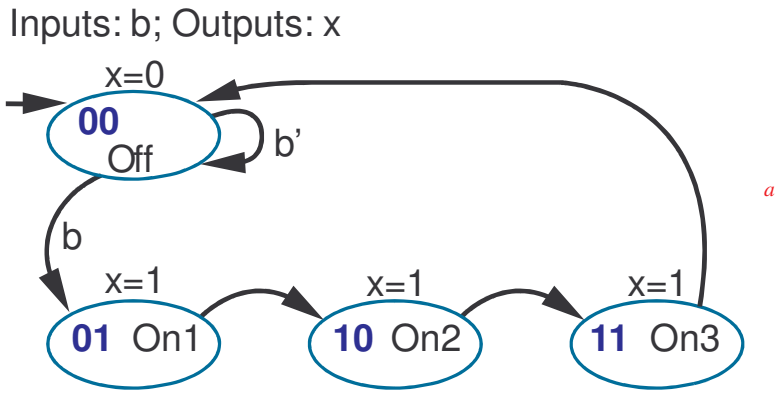
- Five step controller design process

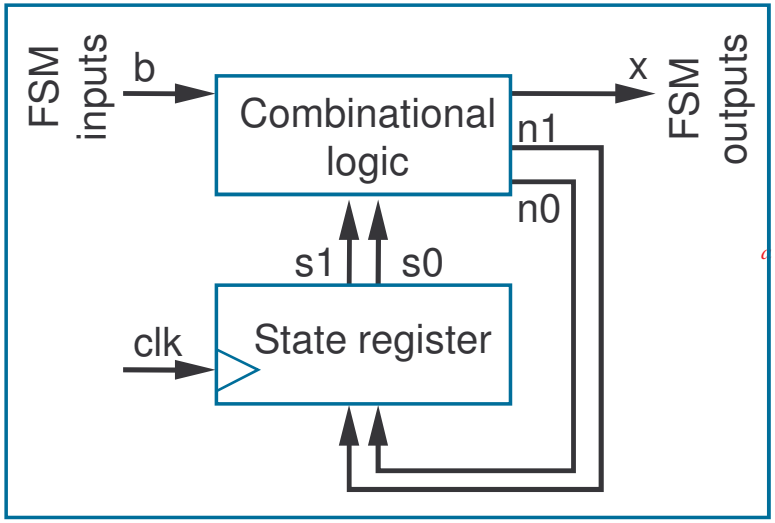
	Step	Description
Step 1	*Capture the FSM*	Create an FSM that describes the desired behavior of the controller.
Step 2	*Create the architecture*	Create the standard architecture by using a state register of appropriate width, and combinational logic with inputs being the state register bits and the FSM 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.

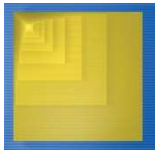


Controller Design: Laser Timer Example

- 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 (in blue)



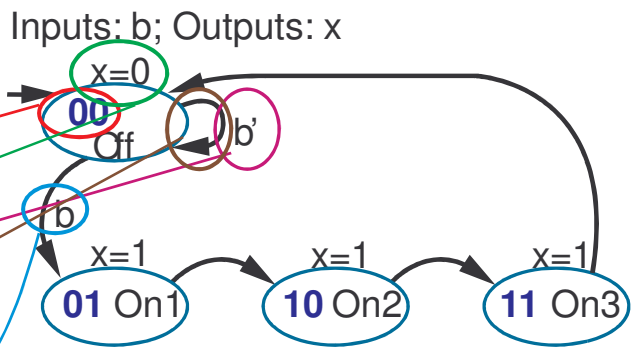


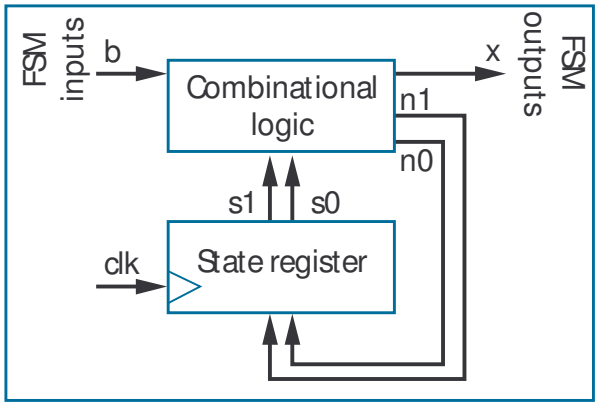


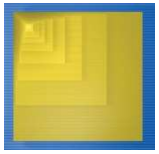
Controller Design: Laser Timer Example (cont)

- 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

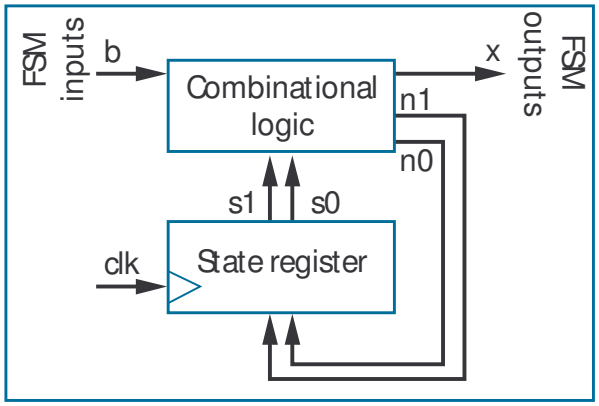






Controller Design: Laser Timer Example (cont)

- 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

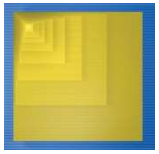
x = s1 + s0 (note from the table that x=1 if s1 = 1 or s0 = 1)

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

n1 = s1's0 + s1s0'

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

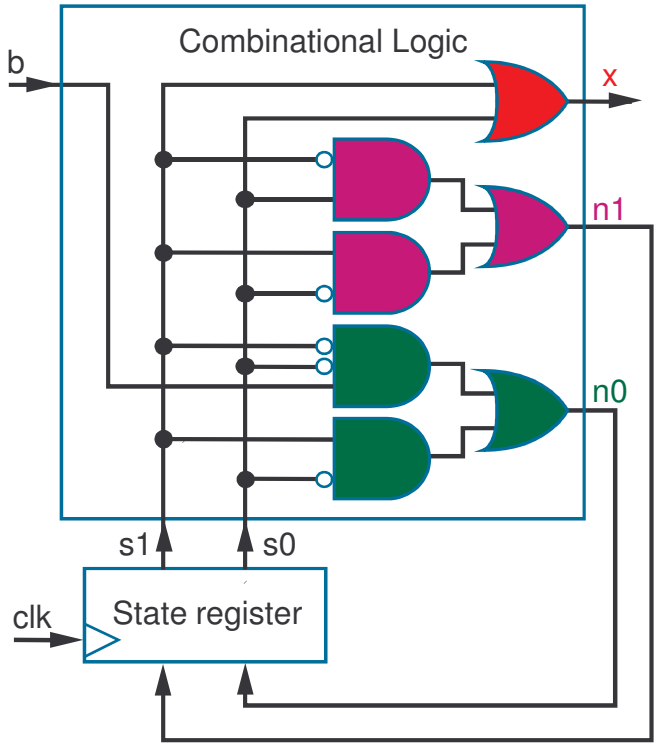
n0 = s1's0'b + s1s0'



Controller Design: Laser Timer Example (cont)

- Step 5: Implement combinational logic (cont)

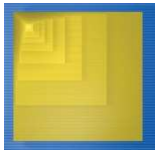
	Inputs			Outputs		
	s1	s0	b	x	n1	n0
Off	0	0	0	0	0	0
	0	0	1	0	0	1
On1	0	1	0	1	1	0
	0	1	1	1	1	0
On2	1	0	0	1	1	1
	1	0	1	1	1	1
On3	1	1	0	1	0	0
	1	1	1	1	0	0



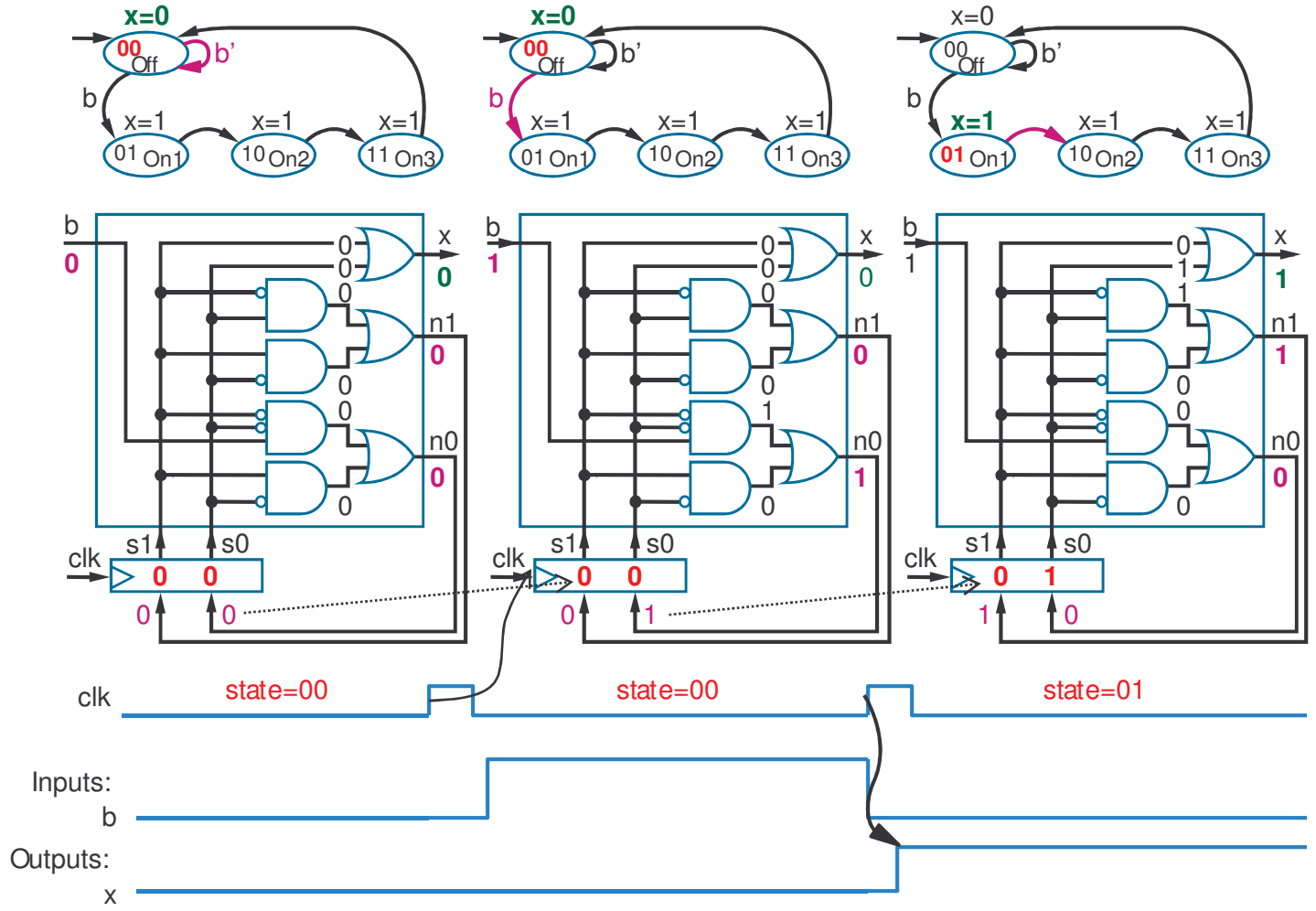
x = s1 + s0

n1 = s1's0 + s1s0'

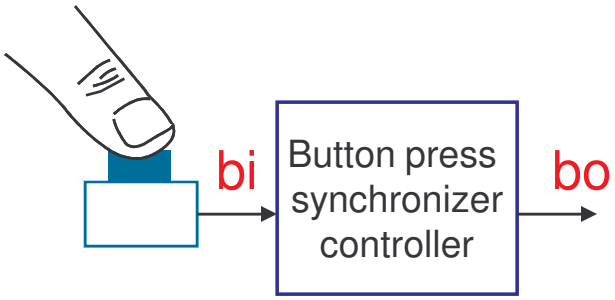
n0 = s1's0'b + s1s0'

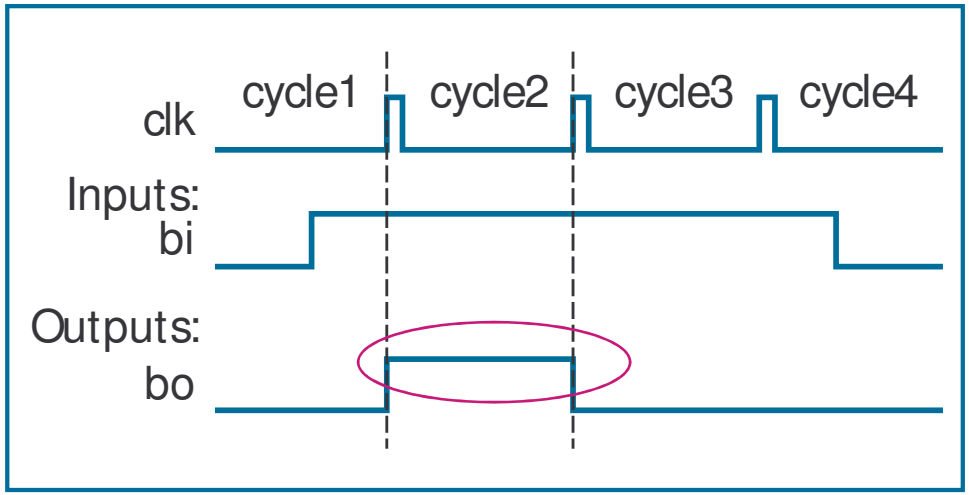


Understanding the Controller's Behavior

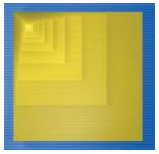


Controller Example:
Button Press Synchronizer



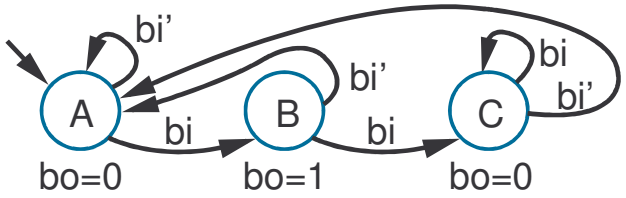


- Want simple sequential circuit that converts button press to single cycle duration, regardless of length of time that button actually pressed
  - We assumed such an ideal button press signal in earlier example, like the button in the laser timer controller

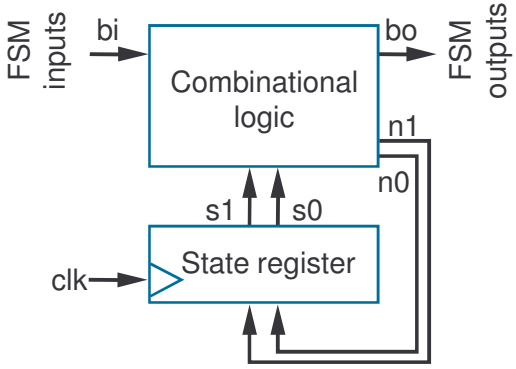


Controller Example:
Button Press Synchronizer (cont)

FSM inputs: bi; FSM outputs: bo

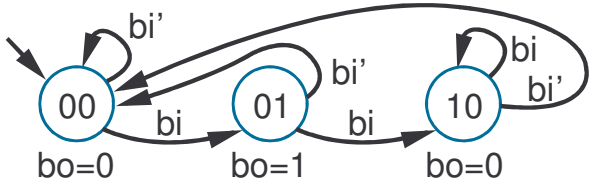


Step 1: FSM



Step 2: Create architecture

FSM inputs: bi; FSM outputs: bo

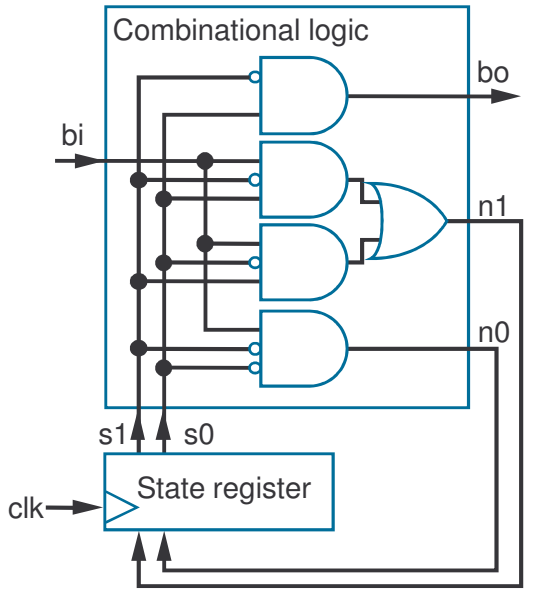


Step 3: Encode states

	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

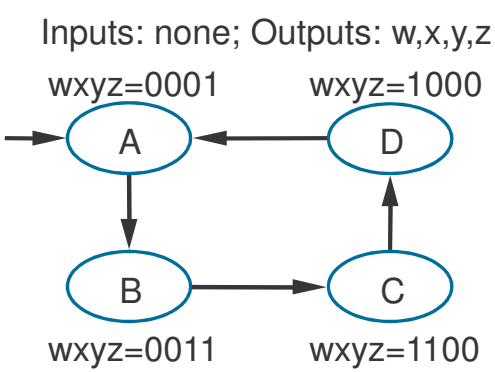
n1 = s1's0bi + s1s0bi
n0 = s1's0'bi
bo = s1's0bi' + s1's0bi = s1s0



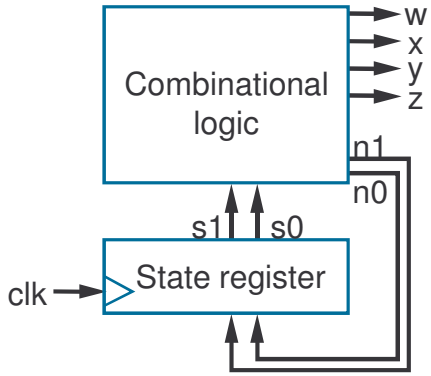
Step 5: Create combinational circuit

Controller Example: Sequence Generator

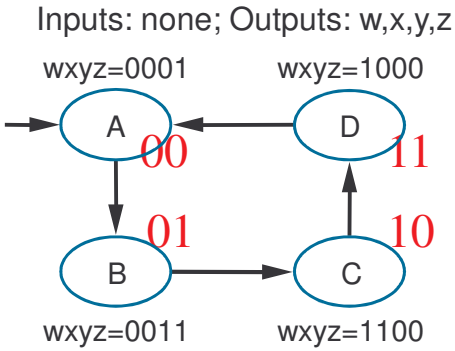
- Want generate sequence 0001, 0011, 1100, 1000, (repeat)
  - Each value for one clock cycle
  - Common, e.g., to create pattern in 4 lights, or control magnets of a "stepper motor"



Step 1: Create FSM



Step 2: Create architecture

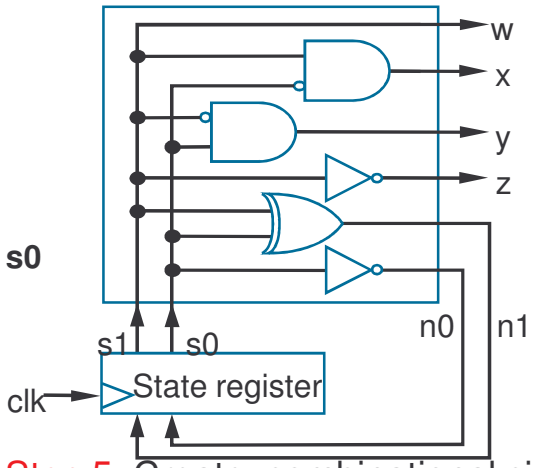


Step 3: Encode states

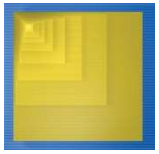
	Inputs		Outputs					
	s1	s0	w	x	y	z	n1	n0
A	0	0	0	0	0	1	0	1
B	0	1	0	0	1	1	1	0
C	1	0	1	1	0	0	1	1
D	1	1	1	0	0	0	0	0

Step 4: Create state table

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

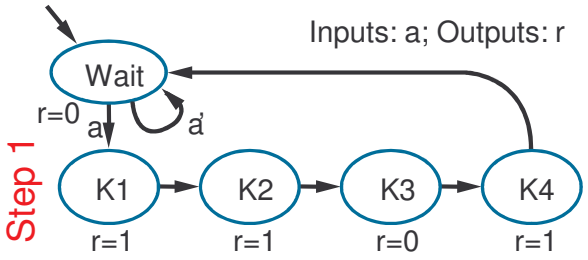


Step 5: Create combinational circuit

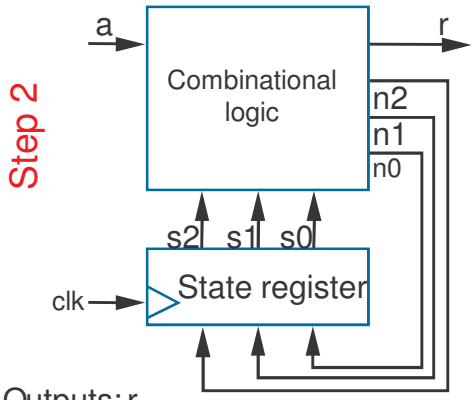


Controller Example: Secure Car Key

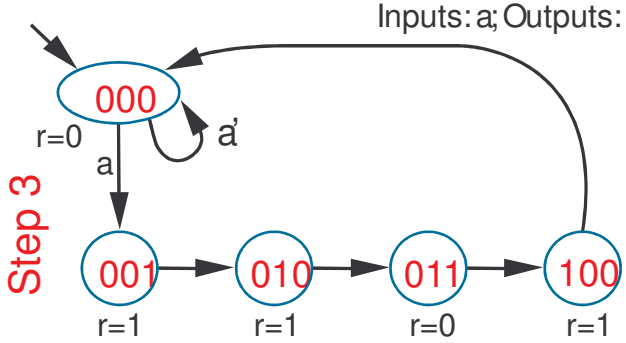
- (from earlier example)



Step 1



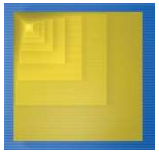
Step 2



Step 3

	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

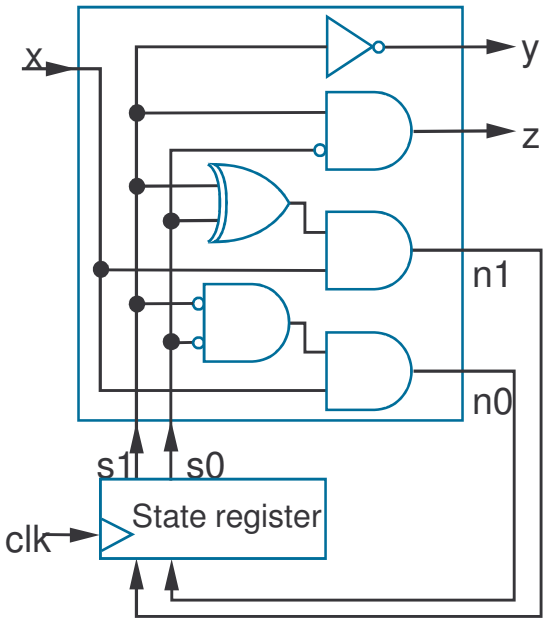
Step 4



We'll omit Step 5 … circuits from equations is old hat by now.

Example: Seq. Circuit to FSM (Reverse Engineering) … a.k.a. Circuit Analysis

What does this circuit do?

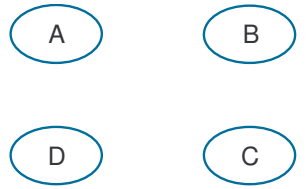


Work backwards

y=s1'
z = s1s0'
n1=(s1 xor s0)x
n0=(s1'*s0')x

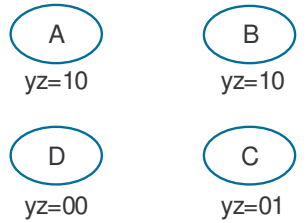
	Inputs			Outputs			
	s1	s0	x	n1	n0	y	z
A	0	0	0	0	0	1	0
	0	0	1	0	1	1	0
B	0	1	0	0	0	1	0
	0	1	1	1	0	1	0
C	1	0	0	0	0	0	1
	1	0	1	1	0	0	1
D	1	1	0	0	0	0	0
	1	1	1	0	0	0	0

Pick any state names you want



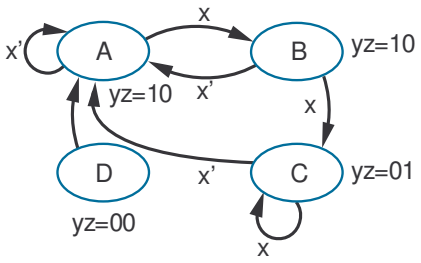
states

Outputs: y, z

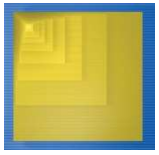


states with outputs

Inputs: x; Outputs: y, z

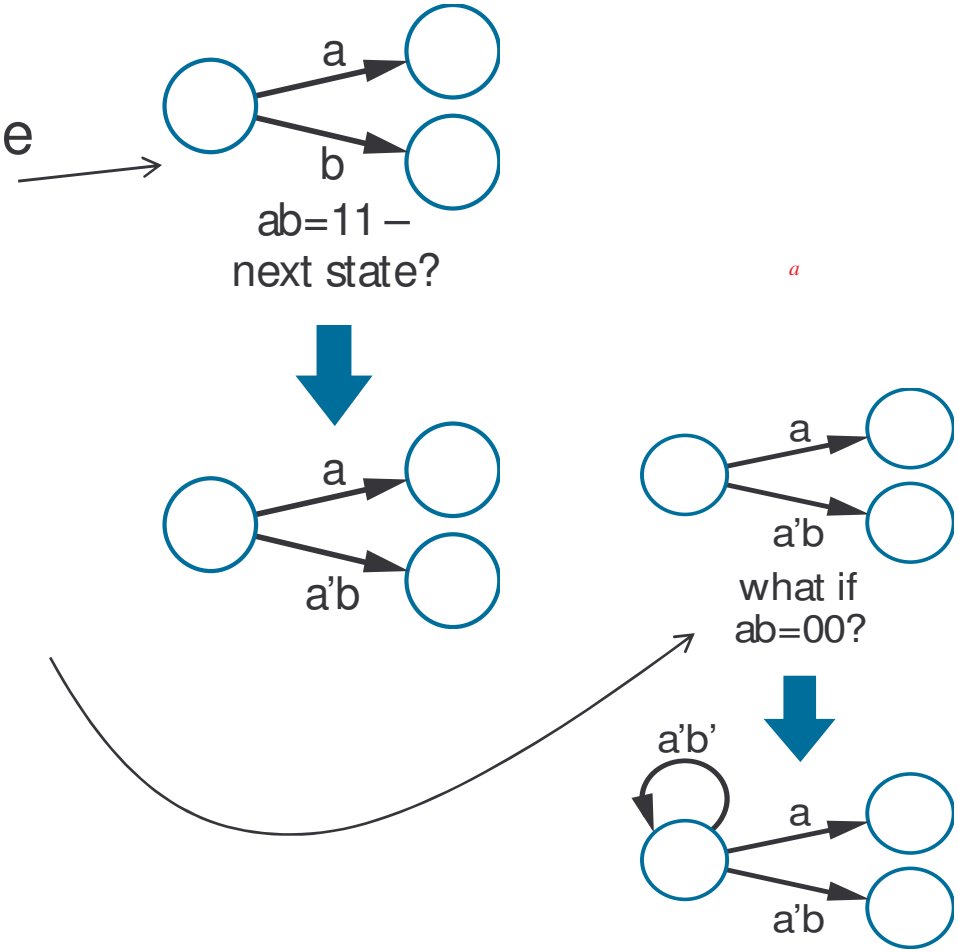


states with outputs and transitions



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?



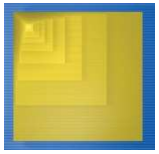
Defining the FSM using VHDL

- Using the model put forth by the author … there will be two process statements
  - Combinational logic (next state logic and output logic combined)
  - State register

- Modeling the state register
  - Assume cs (current state) and ns (next state) are bit vectors

```
process (clk)
begin
  if (clk'event and clk='1') then
    cs <= ns;
  end if;
end process;
```

  - This register is built with rising edge triggered D-FF's.



FSM w/ VHDL (cont.)

- For a falling-edge triggered register …

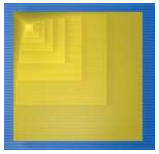
```
process (clk)
begin
   if (clk'event and
   clk='0') then
           cs <= ns;
   end if;
end process;
```

- For a rising-edge triggered register, with async. reset …

```
process (clk, reset)
begin
if (reset = '1') then
   cs <= "000";
elsif (clk'event and clk='1') then
      cs <= ns;
   end if;
end process;
```

- For a falling-edge triggered register, with sync. active-low reset …

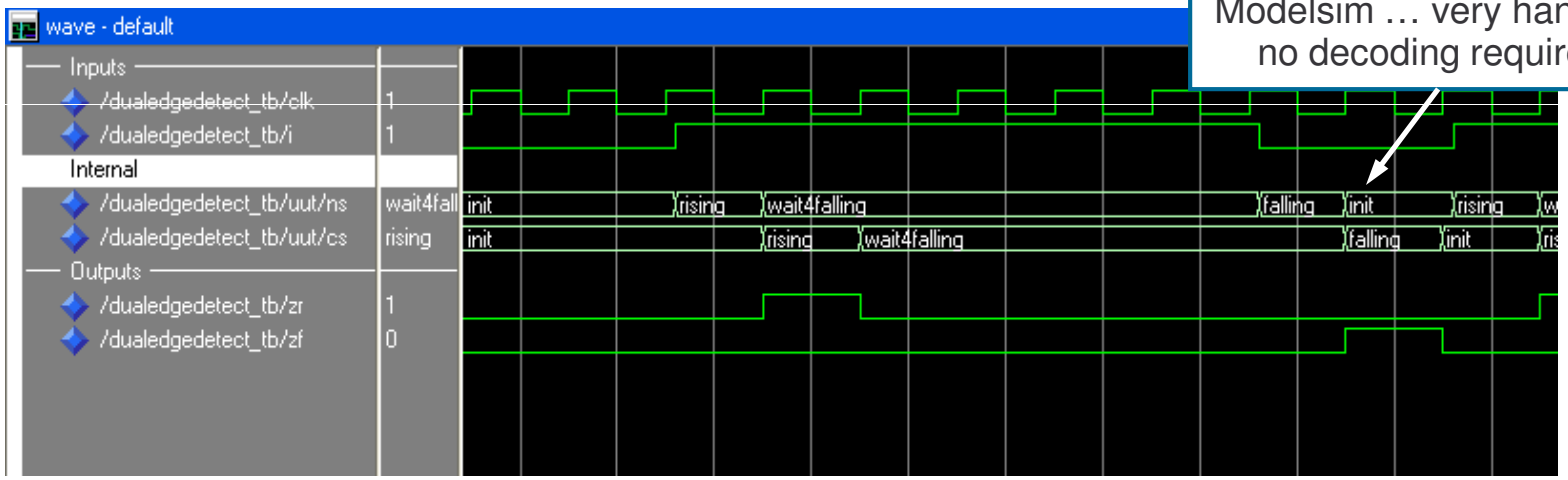
```
process (clk, reset)
begin
   if (clk'event and clk='0') then
      if (reset = '0') then
         cs <= "000";
      else
         cs <= ns;
      end if;
   end if;
end process;
```



FSM w/ VHDL (cont.)

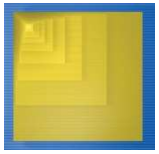
- If you define enumerate a new type, life will be easier
  - Example:

```
architecture Behavioral of dualEdgeDetect is
   type stateType is (init, rising, wait4falling, falling);
   signal ns, cs: stateType;
begin
```



Note the state names in Modelsim … very handy … no decoding required.

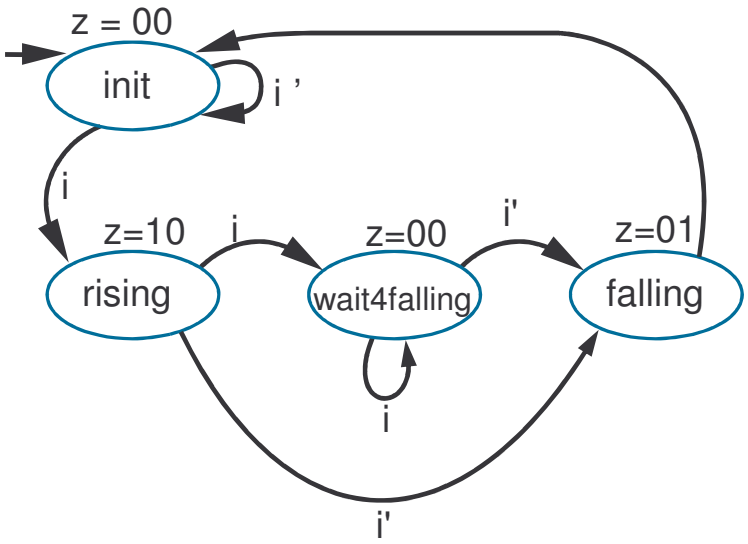
- Another benefit for using enumerated types for naming states is when designing the combinatorial logic block



FSM w/ VHDL (cont.)

- Setup combinatorial logic block in a process statement and a case statement:

Inputs: i; Outputs: z = zr,zf




```
process(i, cs)
begin
    case cs is
       when init =>
          if i='0' then    ns <= init;
          else             ns <= rising;
          end if;
          zr <= '0';       zf <= '0';
       when rising =>
          if i='0' then    ns <=falling;
          else             ns <= wait4falling;
          end if;
          zr <= '1'; zf <= '0';
       when wait4falling =>
          if i='0' then    ns <= falling;
          else             ns <= wait4falling;
          end if;
          zr <= '0';       zf <= '0';
       when falling =>
          ns <= init;
          zr <= '0';       zf <= '1';
       when others =>
          ns <= init;
          zr <= '0';       zf <= '0';
     end case;
end process;
```