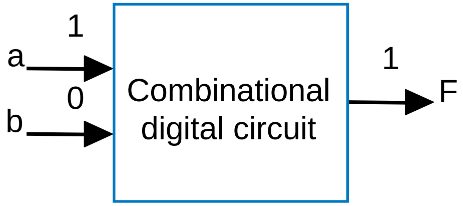
Digital Design

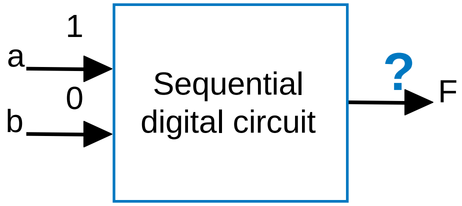
Chapter 3:

Sequential Logic Design -- Controllers

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

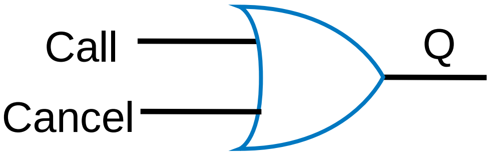




Must know sequence of past inputs to know output

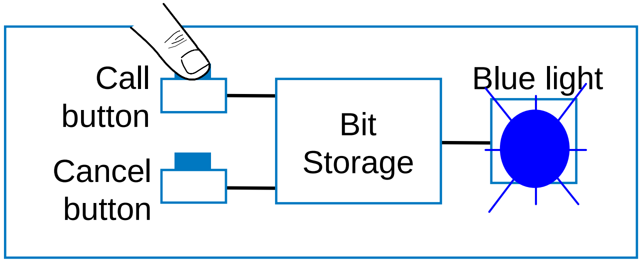
Example Needing Bit Storage

- 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?

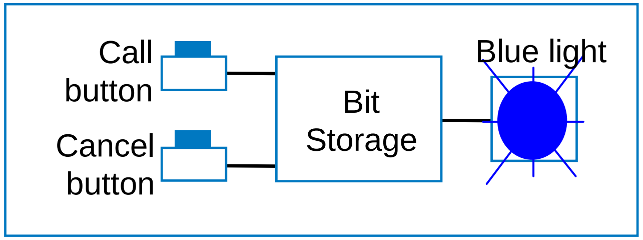


Doesn’t work. Q=1 when Call=1, but doesn’t stay 1 when Call returns to 0

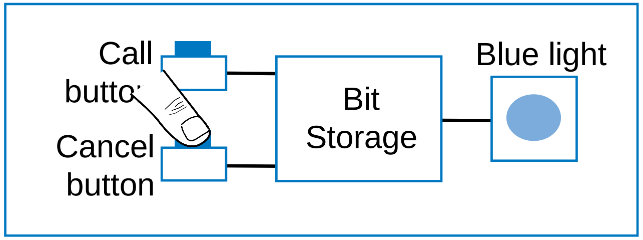
Need some form of “feedback” in the circuit



1. Call button pressed – light turns on



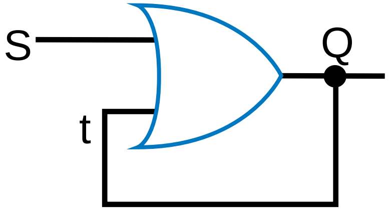
*2. Call button released – light **stays on***

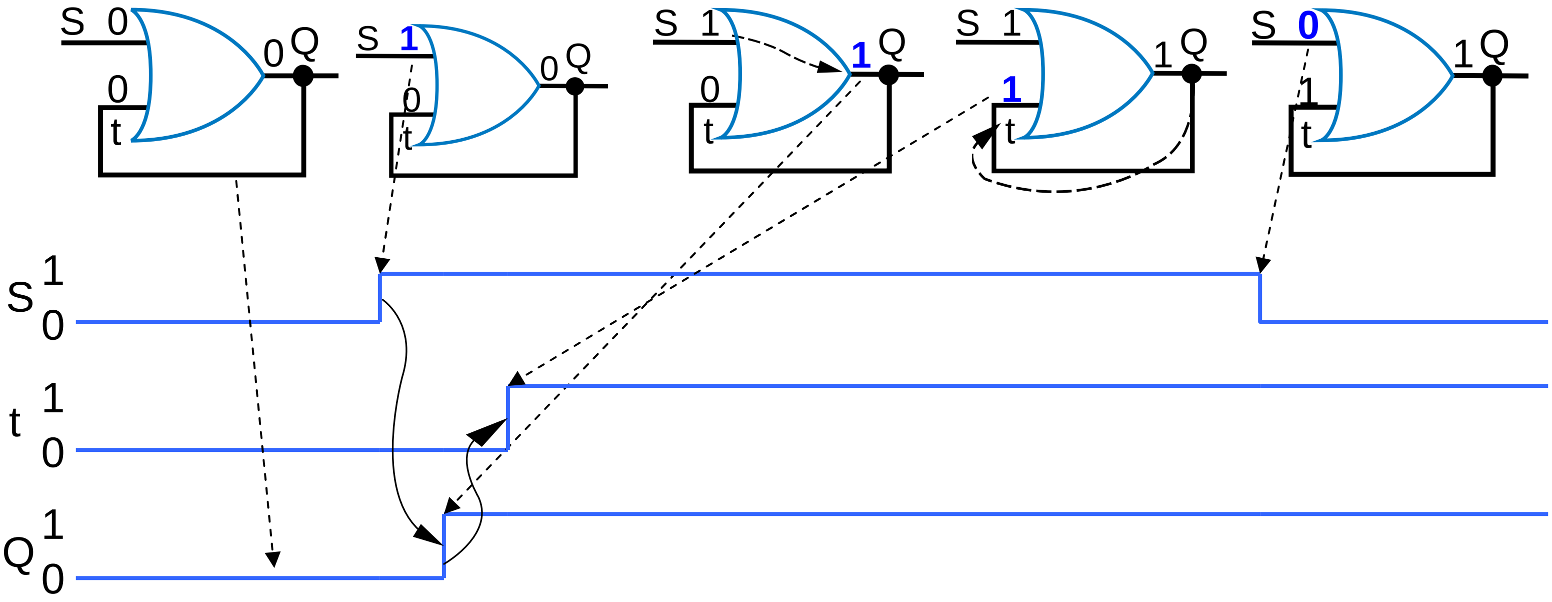


3. Cancel button pressed – light turns off

First attempt at Bit Storage

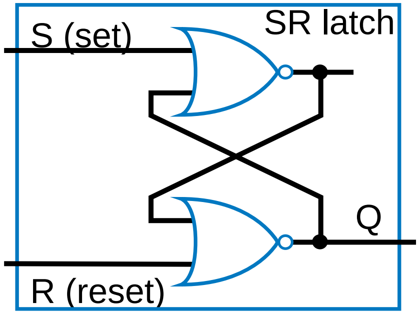
- We need some sort of feedback
  - Does circuit on the right do what we want?
    - No: Once Q becomes 1 (when S=1), Q stays 1 forever – no value of S can bring Q back to 0

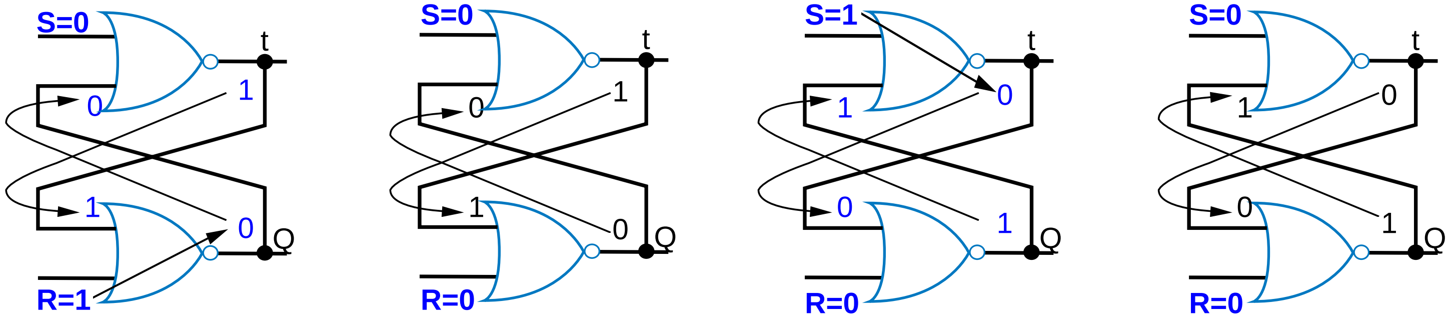


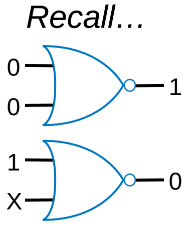


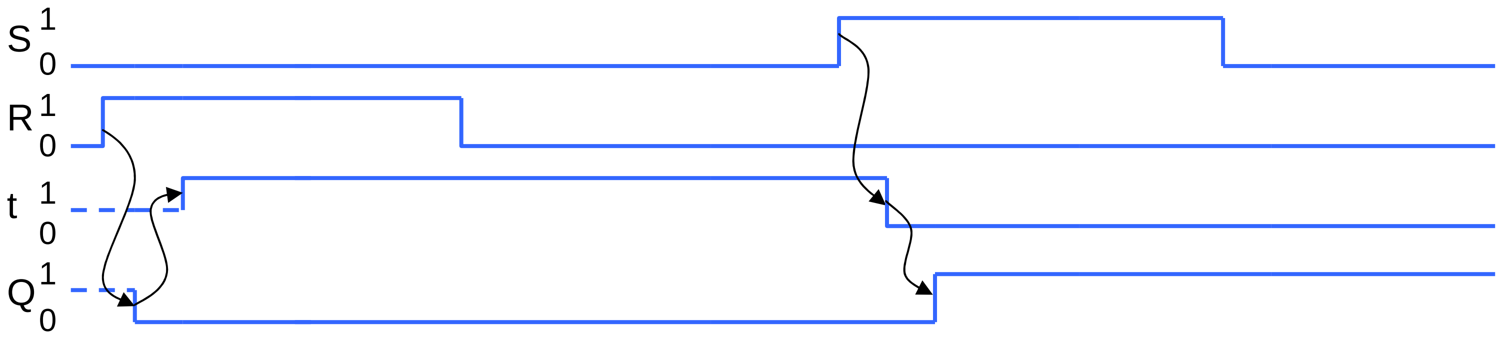
Bit Storage Using an SR Latch

- Does the circuit to the right, with cross-coupled NOR gates, do what we want?
  - Yes! How did someone come up with that circuit? Maybe just trial and error, a bit of insight...





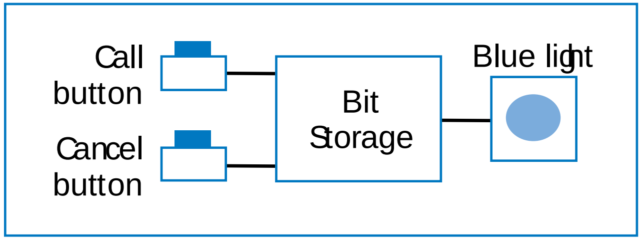


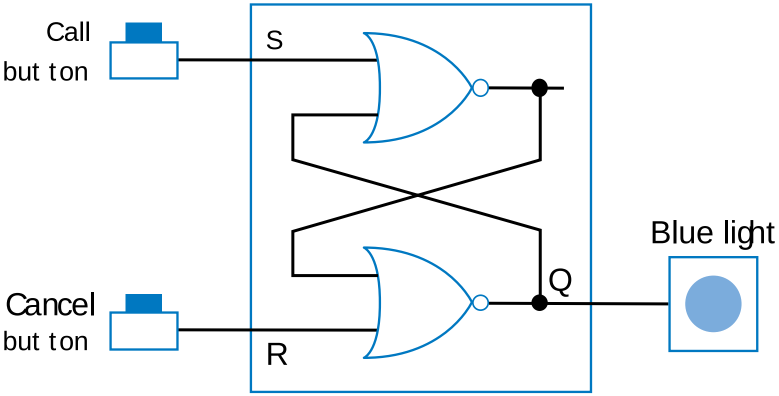


Example Using SR Latch for Bit Storage

- SR latch can serve as bit storage in previous example of flight-attendant call button
  - Call=1 : sets Q to 1
    - Q stays 1 even after Call=0
  - Cancel=1 : resets Q to 0

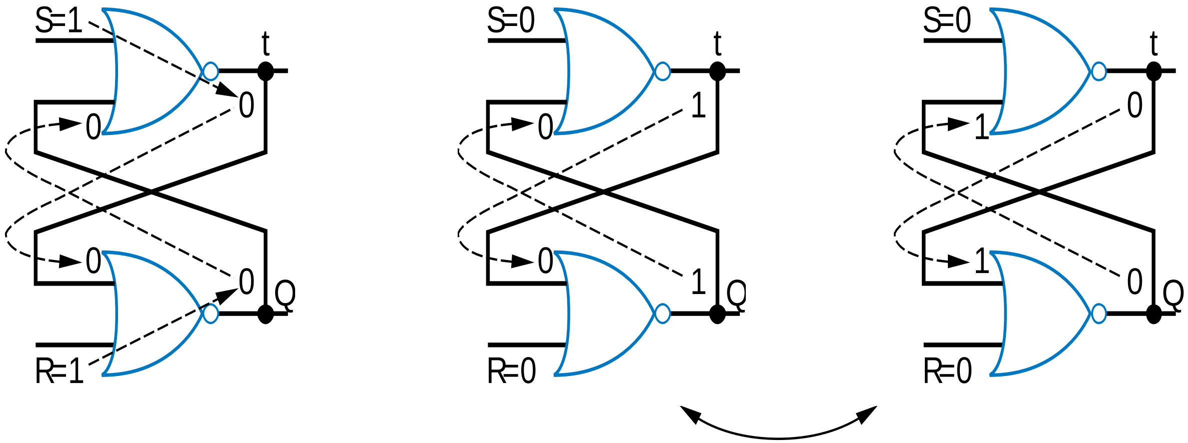
- But, there’s a problem...

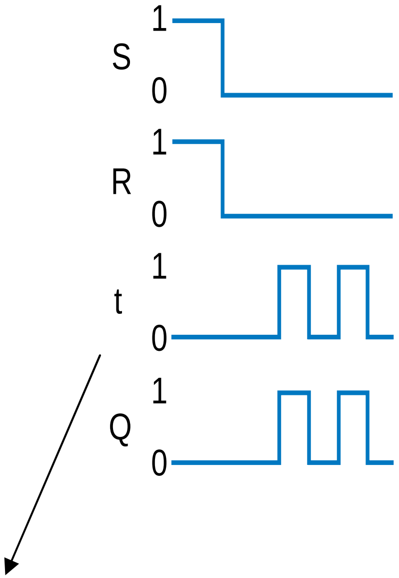




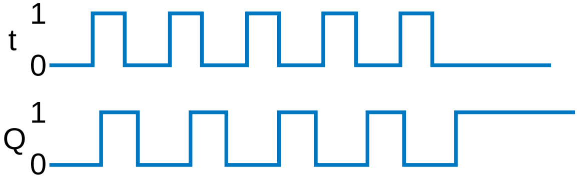
Problem with SR Latch

- Problem
  - If S=1 and R=1 simultaneously, we don’t know what value Q will take



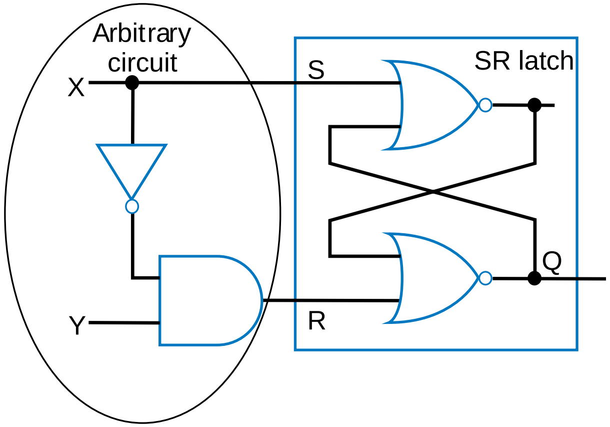


Q may oscillate. Then, because one path will be slightly longer than the other, Q will eventually settle to 1 or 0 – but we don’t know which.

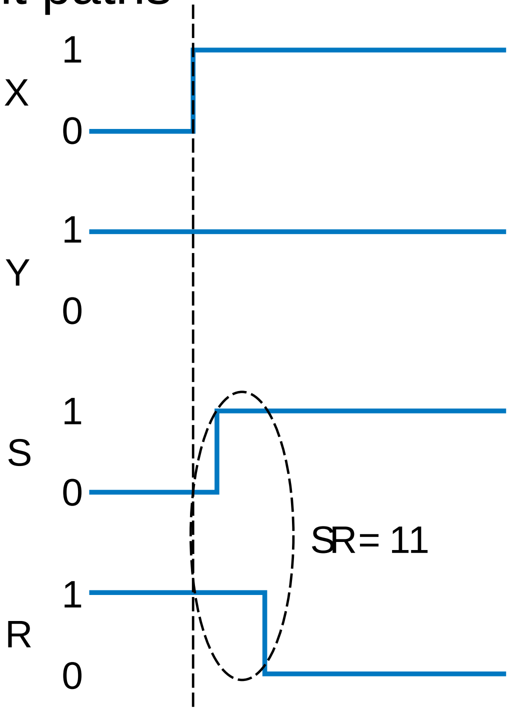


Problem with SR Latch

- Problem not just one of a user pressing two buttons at same time
- Can also occur even if SR inputs come from a circuit that supposedly never sets S=1 and R=1 at same time
  - But does, due to different delays of different paths

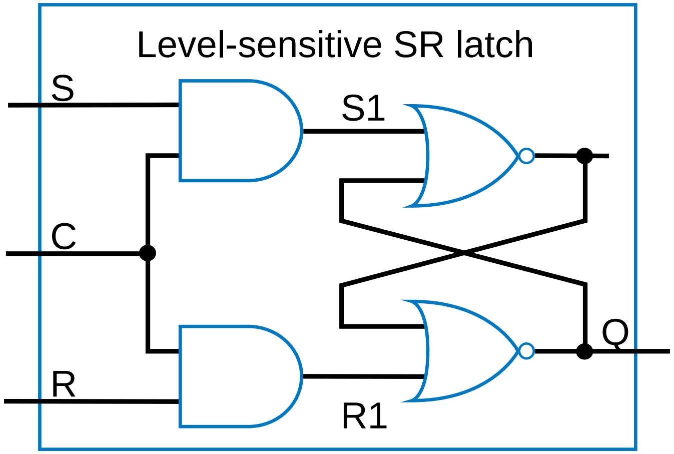


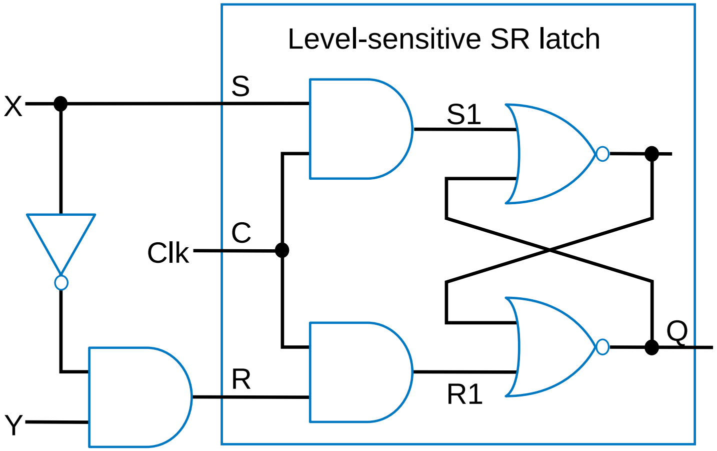
The longer path from X to R than to S causes SR=11 for short time – could be long enough to cause oscillation



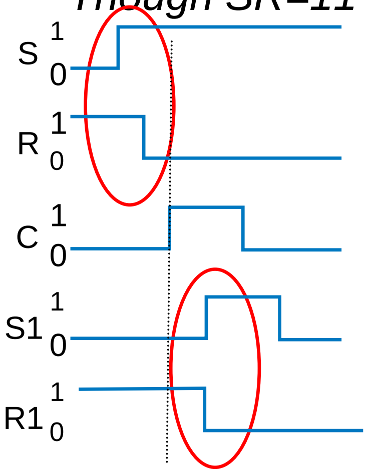
Solution: Level-Sensitive SR Latch

- Add enable input “C” as shown
  - Only let S and R change when C=0
    - Enure circuit in front of SR never sets SR=11, except briefly due to path delays
  - Change C to 1 only after sufficient time for S and R to be stable
  - When C becomes 1, the stable S and R value passes through the two AND gates to the SR latch’s S1 R1 inputs.

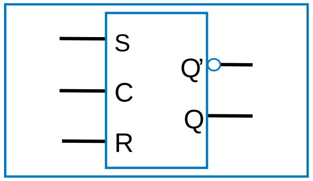




Though SR=11 briefly...

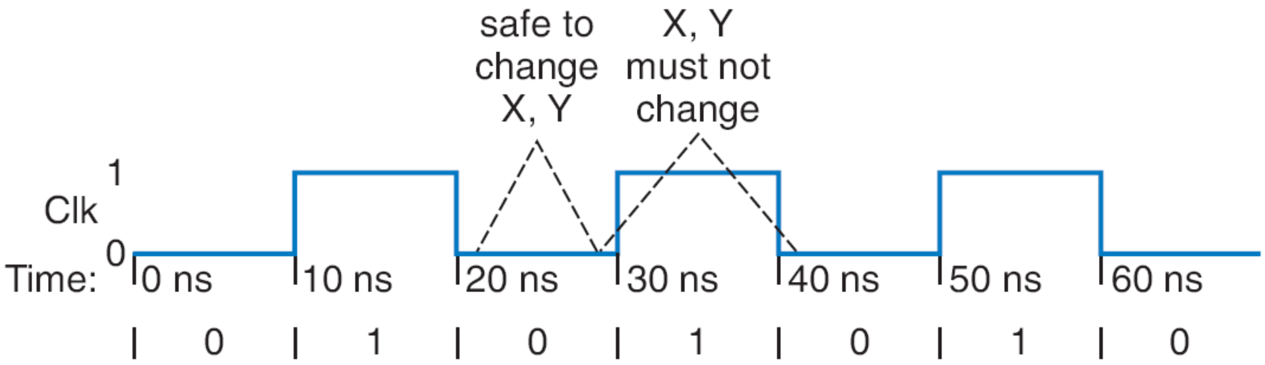


...S1R1 never = 11

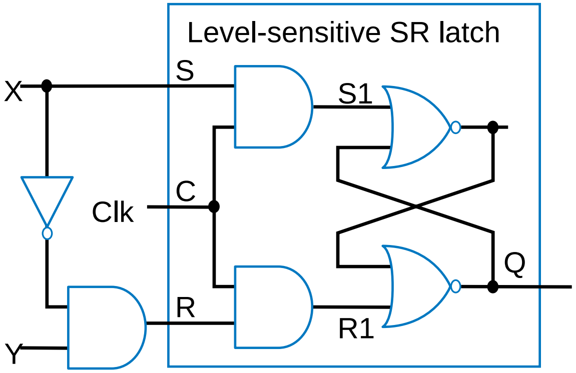


Level-sensitive SR latch symbol

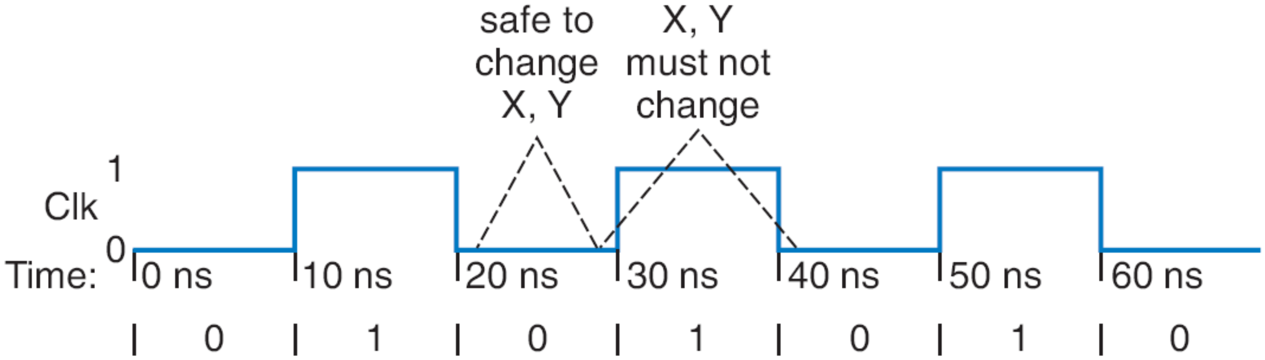
Clock Signals for a Latch



- How do we know when it’s safe to set C=1?
  - Most common solution –make C pulse up/down
    - C=0: Safe to change X, Y
    - C=1: Must *not* change X, Y
    - We’ll see how to ensure that later
  - ***Clock*** signal -- Pulsing signal used to enable latches
    - Because it ticks like a clock
  - Sequential circuit whose storage components all use clock signals: ***synchronous*** circuit
    - Most common type
    - Asynchronous circuits – important topic, but left for advanced course



Clocks

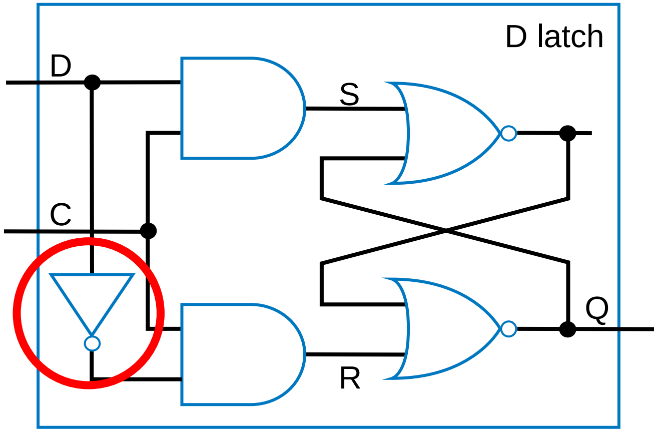


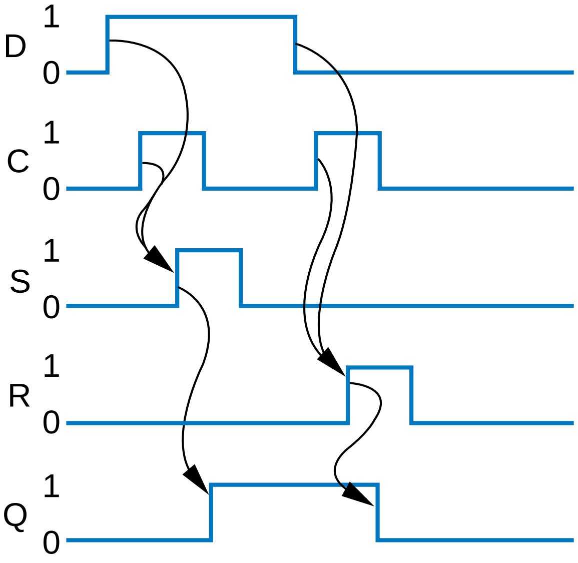
- ***Clock period***: time interval between pulses
  - Above signal: period = 20 ns
- ***Clock cycle***: one such time interval
  - Above signal shows 3.5 clock cycles
- ***Clock frequency***: 1/period
  - Above signal: frequency = 1 / 20 ns = 50 MHz
    - 1 Hz = 1/s

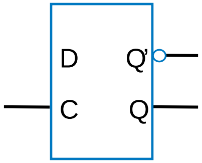
Freq	Period
100 GHz	0.01 ns
10 GHz	0.1 ns
1 GHz	1 ns
100 MHz	10 ns
10 MHz	100 ns

Level-Sensitive D Latch

- SR latch requires careful design to ensure SR=11 never occurs
- D latch relieves designer of that burden
  - Inserted inverter ensures R always opposite of S



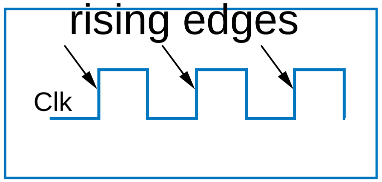


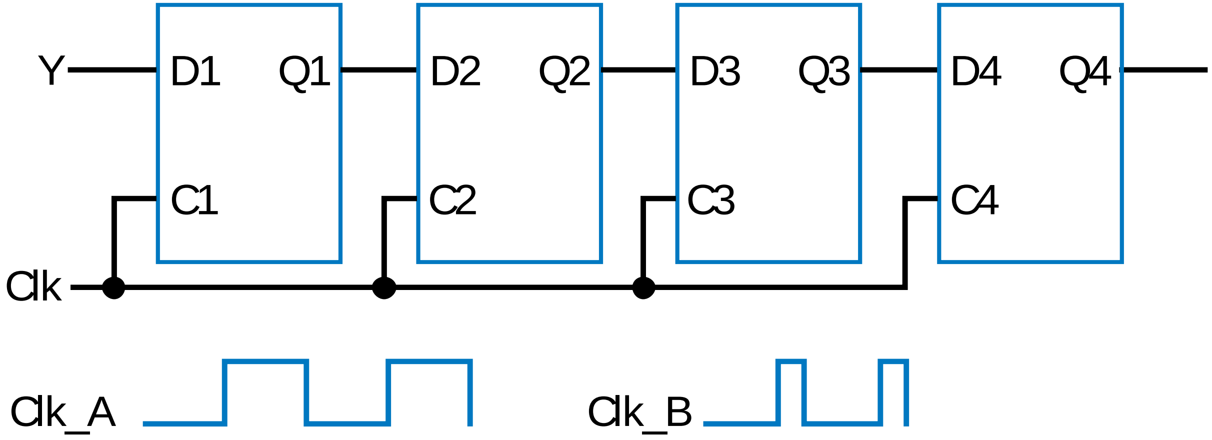


D latch symbol

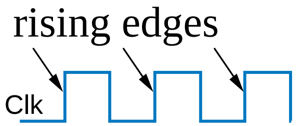
Problem with Level-Sensitive D Latch

- D latch still has problem (as does SR latch)
  - When C=1, through how many latches will a signal travel?
  - Depends on for how long C=1
    - Clk_A -- signal may travel through multiple latches
    - Clk_B -- signal may travel through fewer latches
  - Hard to pick C that is just the right length
    - Can we design bit storage that only stores a value on the rising edge of a clock signal?


rising edges

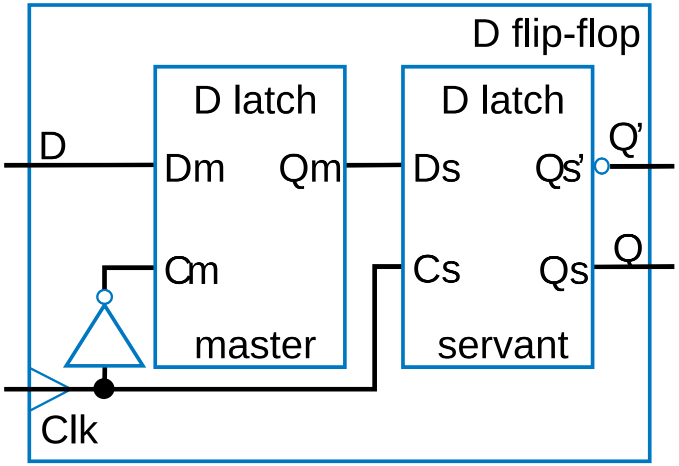


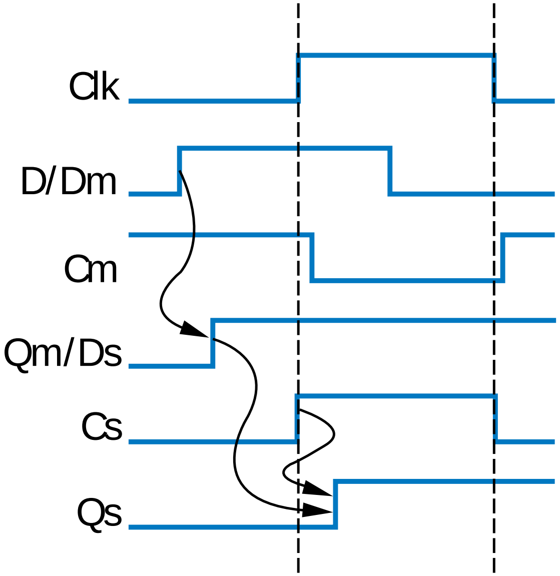
D Flip-Flop



- ***Flip-flop***: Bit storage that stores on clock edge, not level
- One design -- master-servant
  - Two latches, output of first goes to input of second, master latch has inverted clock signal
  - So master loaded when C=0, then servant when C=1
  - When C changes from 0 to 1, master disabled, servant loaded with value that was at D just before C changed -- i.e., value at D during rising edge of C

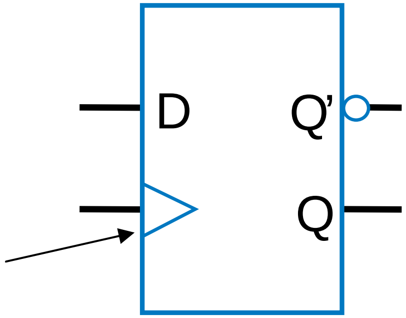
Note: Hundreds of different flip-flop designs exist



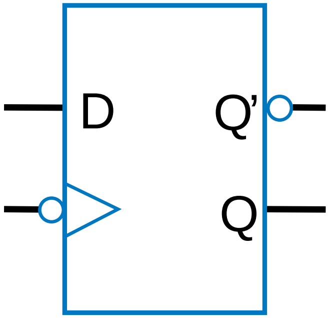


D Flip-Flop

The triangle means clock input, edge triggered

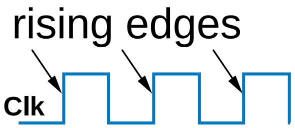


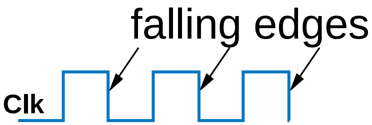
Symbol for rising-edge triggered D flip-flop



Symbol for falling-edge triggered D flip-flop

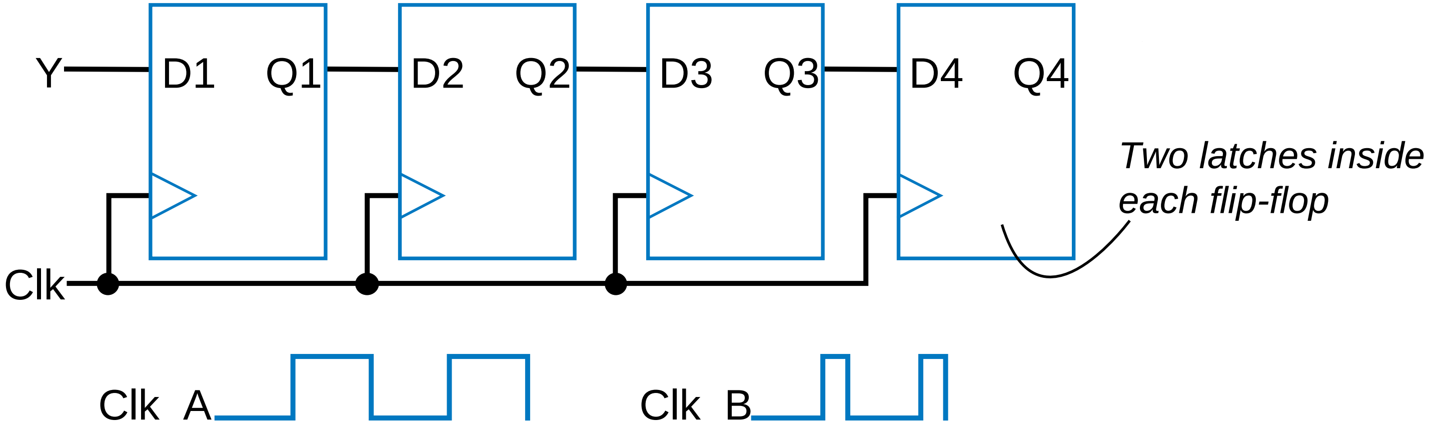
Internal design: Just invert servant clock rather than master





D Flip-Flop

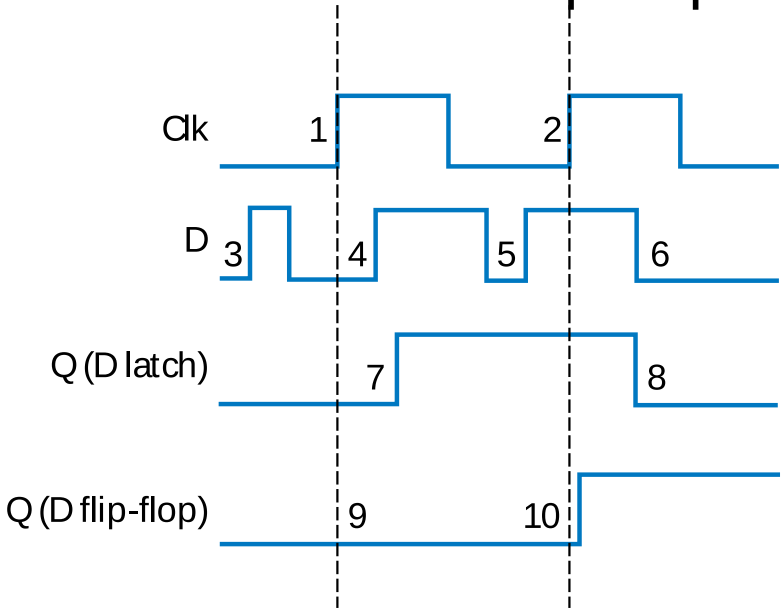
- Solves problem of not knowing through how many latches a signal travels when C=1
  - In figure below, signal travels through exactly one flip-flop, for Clk_A or Clk_B
  - Why? Because on rising edge of Clk, all four flip-flops are loaded simultaneously -- then all four no longer pay attention to their input, until the next rising edge. Doesn't matter how long Clk is 1.



Two latches inside each flip-flop

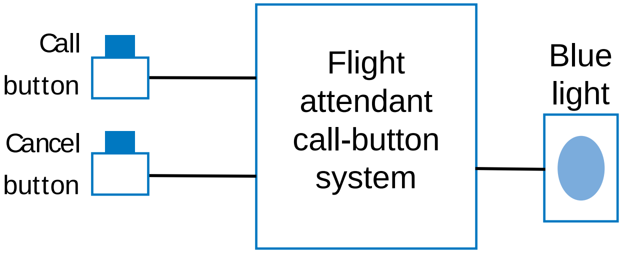
D Latch vs. D Flip-Flop

- Latch is level-sensitive: Stores D when C=1
- Flip-flop is edge triggered: Stores D when C changes from 0 to 1
  - Saying “level-sensitive latch,” or “edge-triggered flip-flop,” is redundant
  - Two types of flip-flops -- rising or falling edge triggered.
- Comparing behavior of latch and flip-flop:



Flight-Attendant Call Button Using D Flip-Flop

- D flip-flop will store bit
- Inputs are Call, Cancel, and present value of D flip-flop, Q
- Truth table shown below



Call	Cancel	Q	D
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

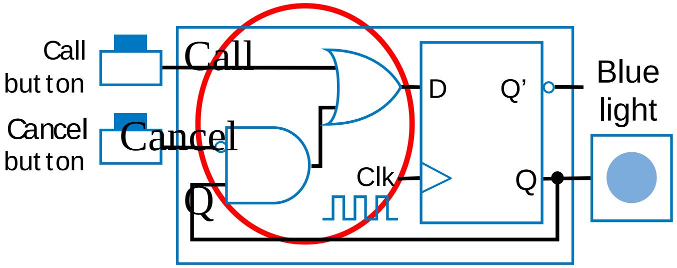
Preserve value: if Q=0, make D=0; if Q=1, make D=1

Cancel -- make D=0

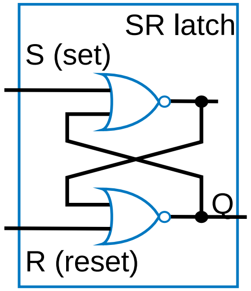
Call -- make D=1

Let's give priority to Call -- make D=1

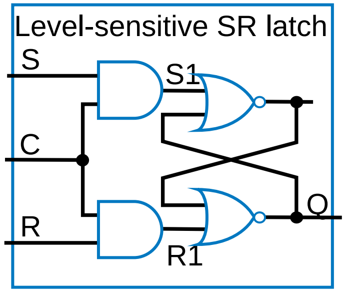
Circuit derived from truth table, using Chapter 2 combinational logic design process



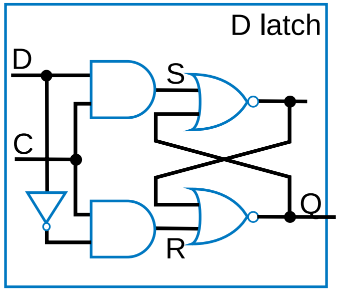
Bit Storage Summary



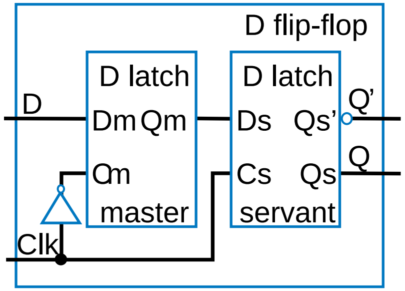
Feature: S=1 sets Q to 1, R=1 resets Q to 0. Problem: SR=11 yield undefined Q.



Feature: S and R only have effect when C=1. We can design outside circuit so SR=11 never happens when C=1. Problem: avoiding SR=11 can be a burden.



Feature: SR can't be 11 if D is stable before and while C=1, and will be 11 for only a brief glitch even if D changes while C=1. Problem: C=1 too long propagates new values through too many latches: too short may not enable a store.

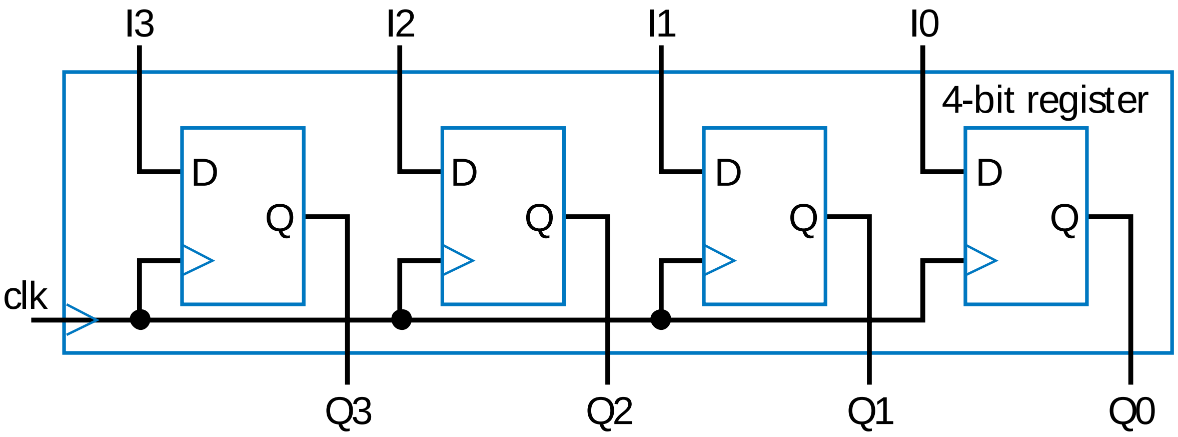


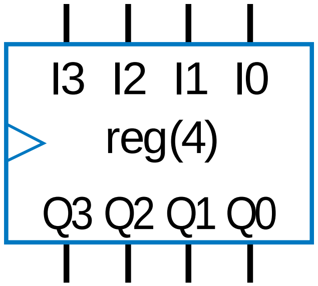
Feature: Only loads D value present at rising clock edge, so values can't propagate to other flip-flops during same clock cycle. Tradeoff: uses more gates internally than D latch, and requires more external gates than SR – but gate count is less of an issue today.

- We considered increasingly better bit storage until we arrived at the robust D flip-flop bit storage

Basic Register

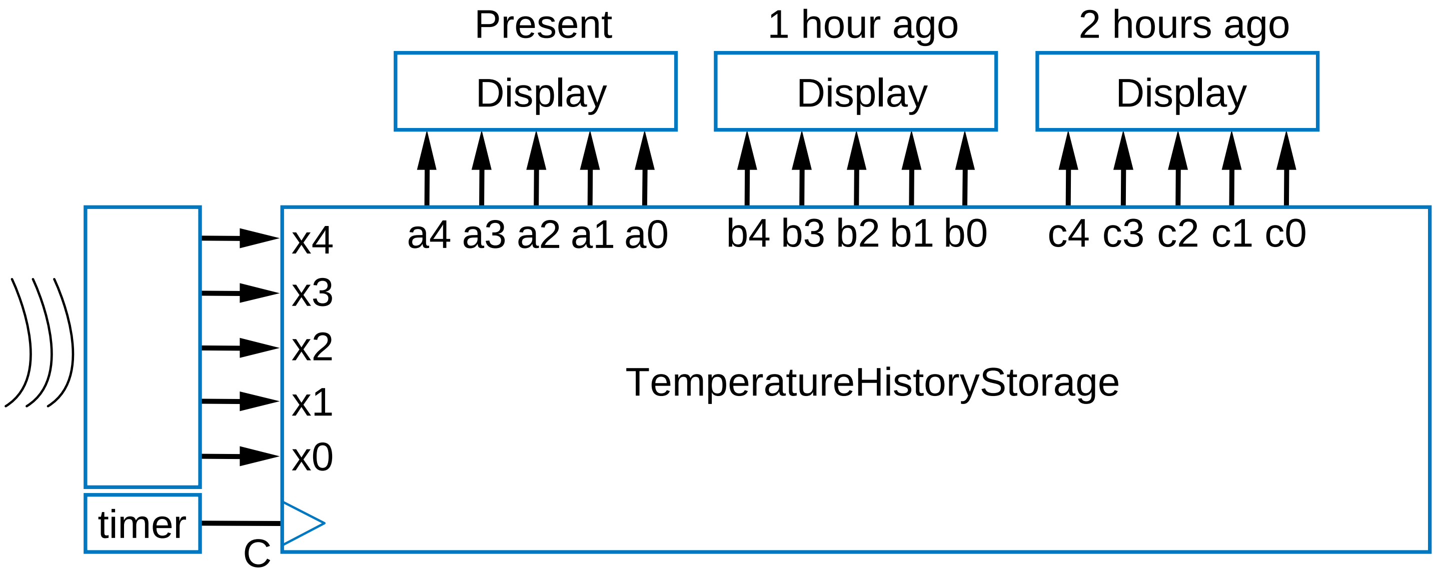
- Typically, we store multi-bit items
  - e.g., storing a 4-bit binary number
- ***Register***: multiple flip-flops sharing clock signal
  - From this point, we'll use registers for bit storage
    - No need to think of latches or flip-flops
    - But now you know what's inside a register





Example Using Registers: Temperature Display

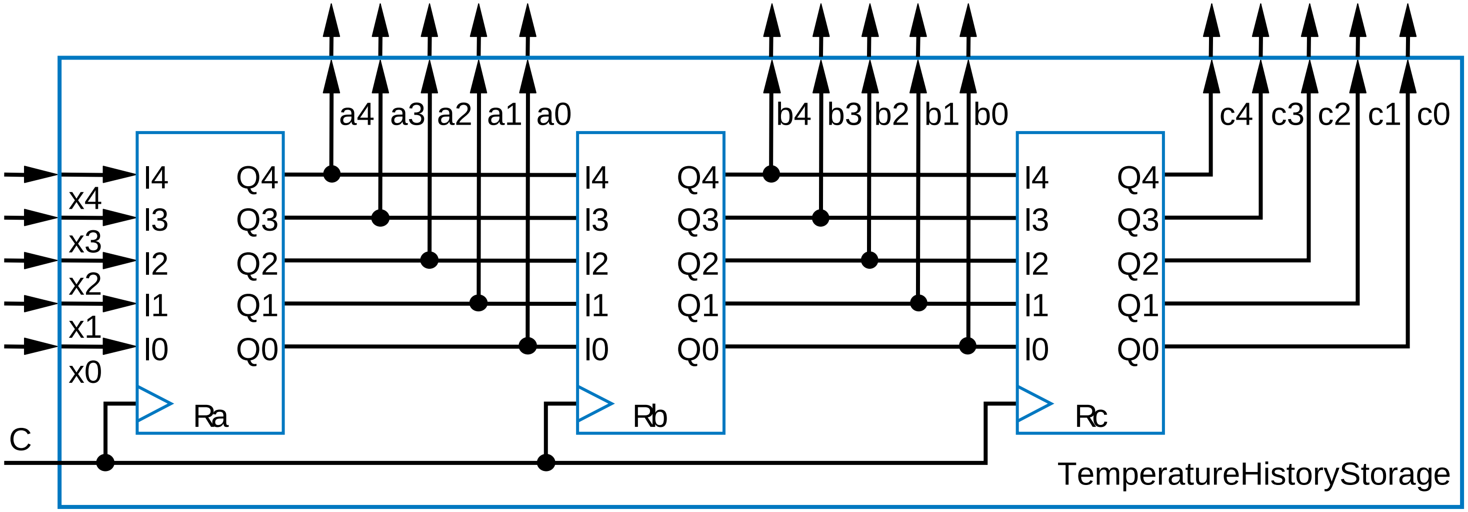
- Temperature history display
  - Sensor outputs temperature as 5-bit binary number
  - Timer pulses C every hour
  - Record temperature on each pulse, display last three recorded values

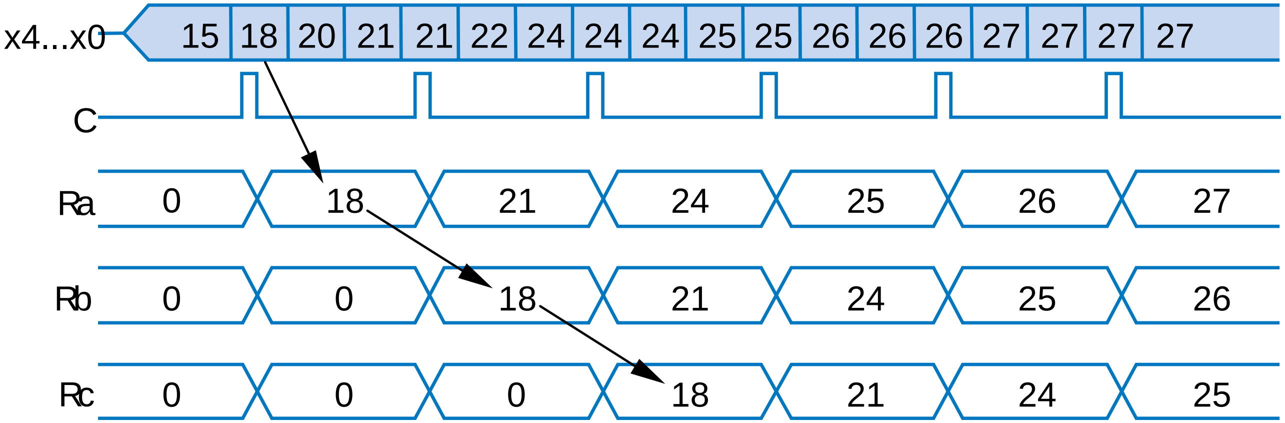


(In practice, we would actually avoid connecting the timer output
C to a clock input, instead only connecting an oscillator output to a clock input.)

Example Using Registers: Temperature Display

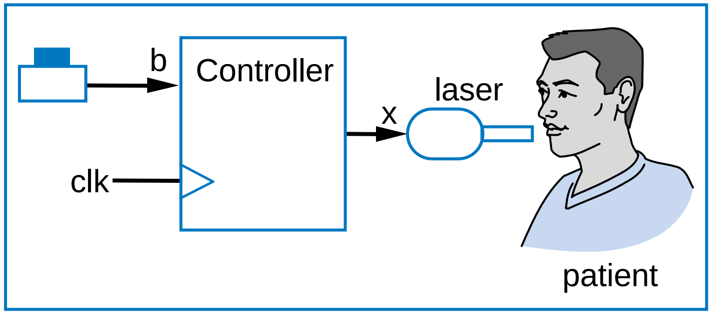
- Use three 5-bit registers

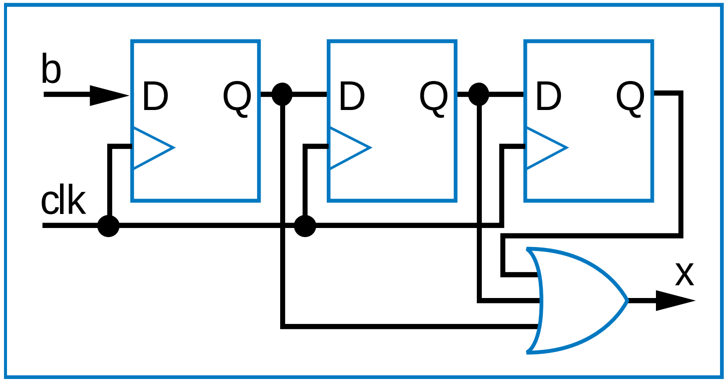




Finite-State Machines (FSMs) and Controllers

- 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



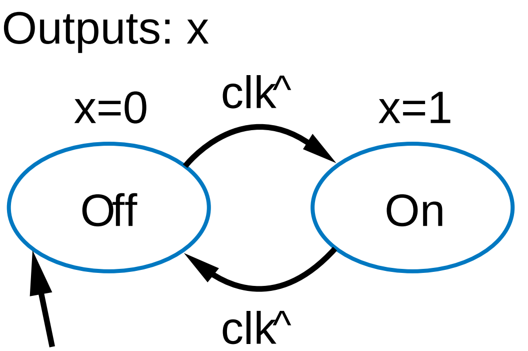


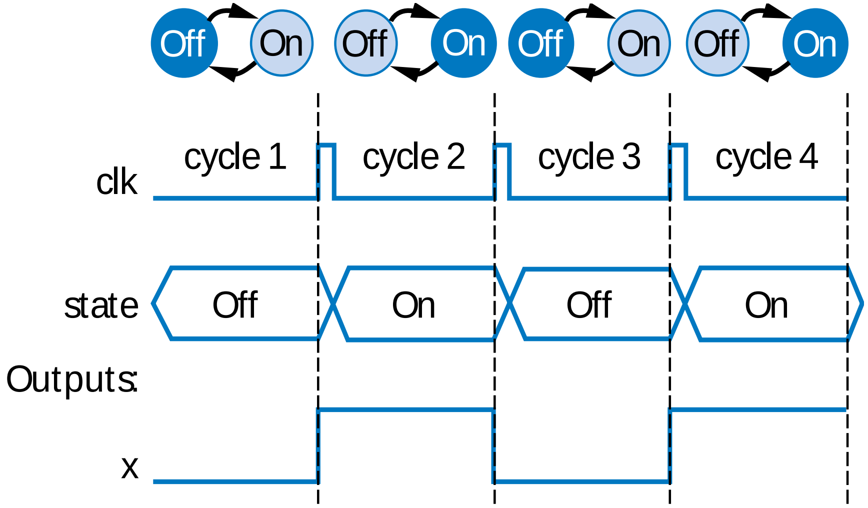
Need a Better Way to Design Sequential Circuits

- 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

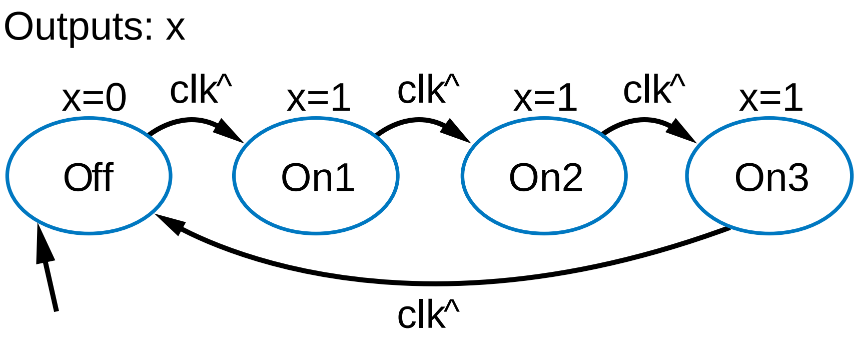
- Finite-State Machine (FSM)
  - A way to describe desired behavior of sequential circuit
    - Akin to Boolean equations for combinational behavior
  - List states, and transitions among states
    - Example: Make x change toggle (0 to 1, or 1 to 0) every clock cycle
    - Two states: “Off” (x=0), and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



Outputs: x

x=0 Off

clk^

x=1 On1

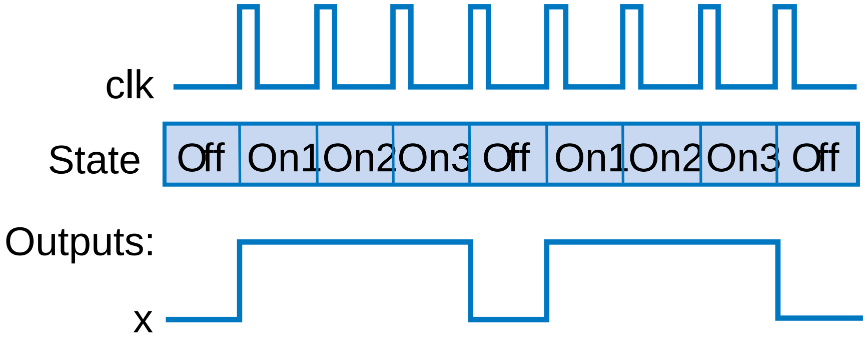
clk^

x=1 On2

clk^

x=1 On3

clk^



clk

State

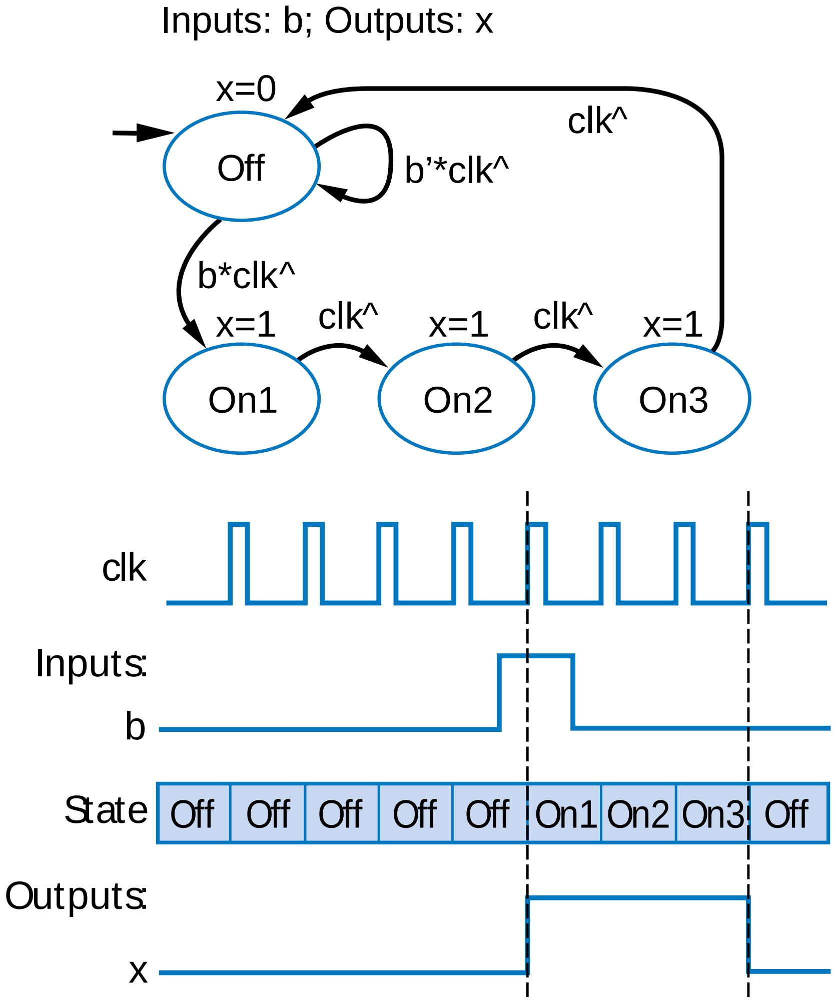
Off	On1	On2	On3	Off	On1	On2	On3	Off

Outputs:

x

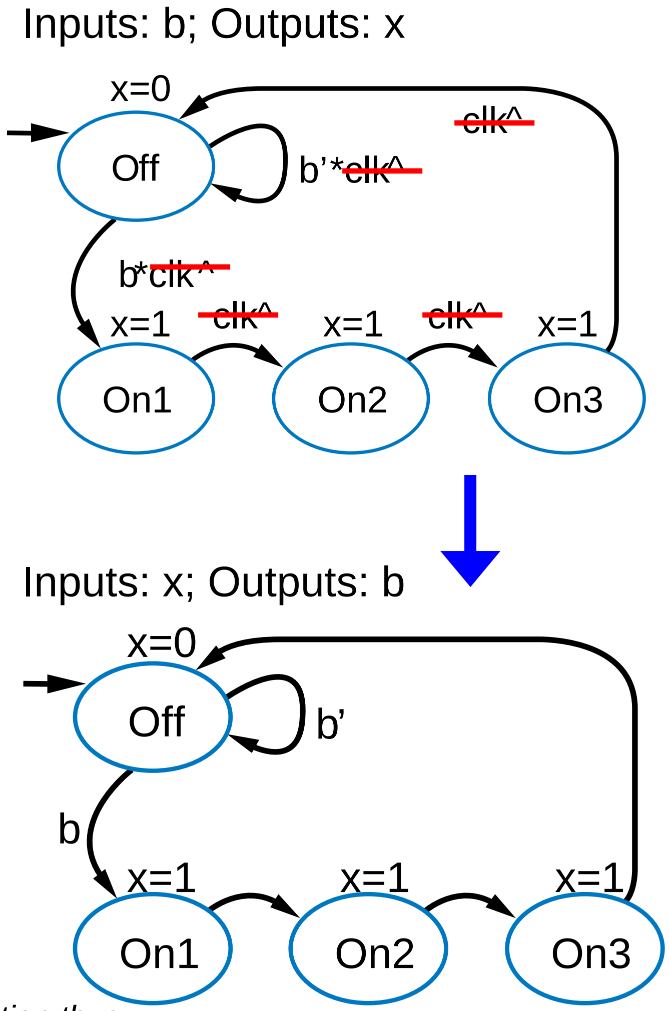
Extend FSM to Three-Cycles High Laser Timer

- 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

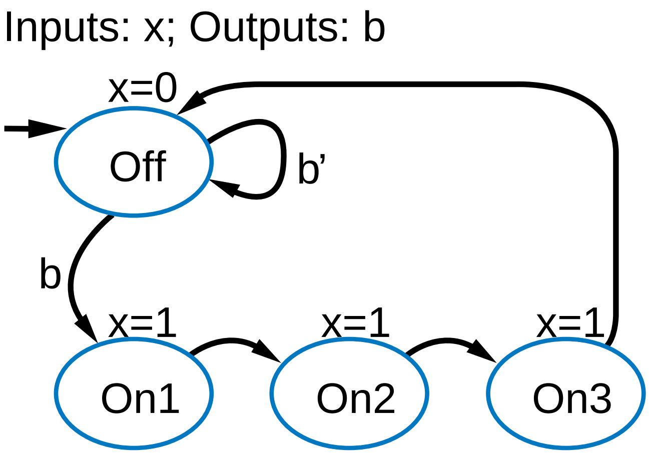
- Showing rising clock on every transition: cluttered
  - Make implicit -- assume every edge has rising clock, even if not shown
  - What if we wanted a transition *without* a rising edge
    - We don’t consider such asynchronous FSMs -- less common, and advanced topic
    - Only consider ***synchronous***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



Note: Transition with no associated condition thus 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
    - Ex: Has 5 transitions
  - Set of actions
    - Sets outputs while in states
    - Ex: x=0, x=1, x=1, and x=1

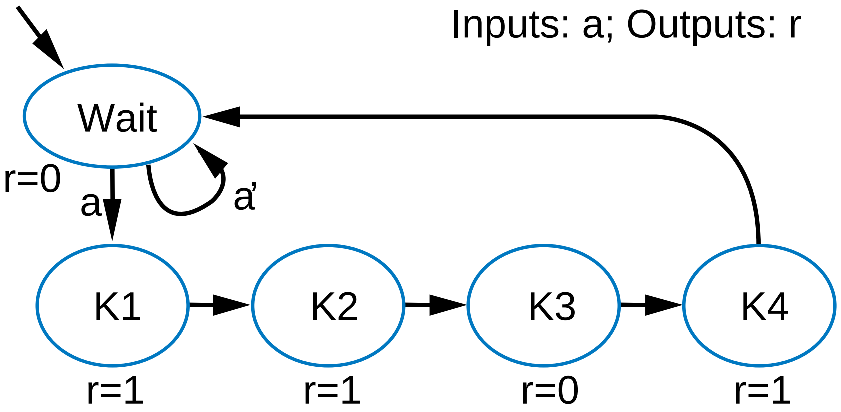


We often draw FSM graphically, known as ***state diagram***

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

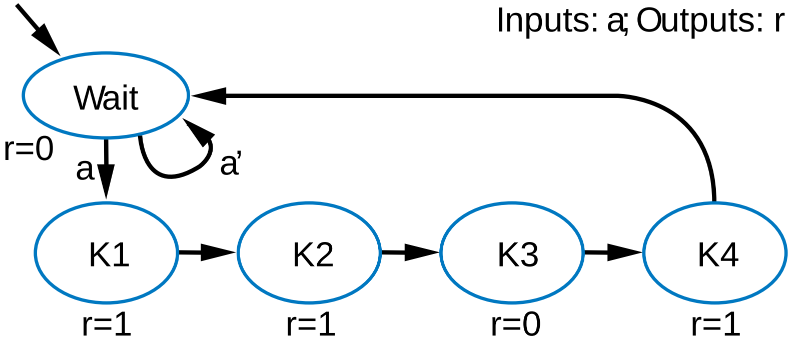
FSM Example: Secure Car Key

- 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)

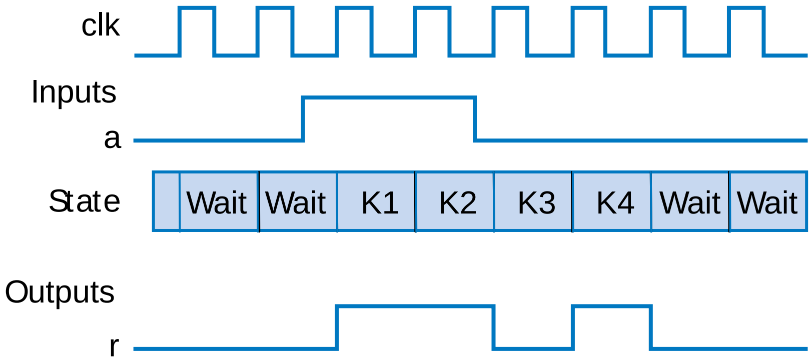


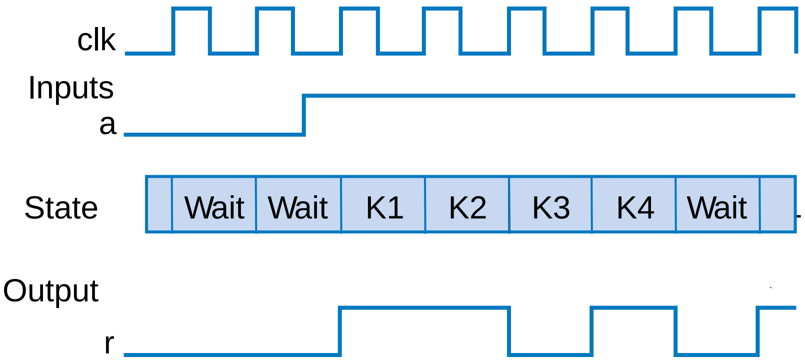
FSM Example: Secure Car Key (cont.)

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



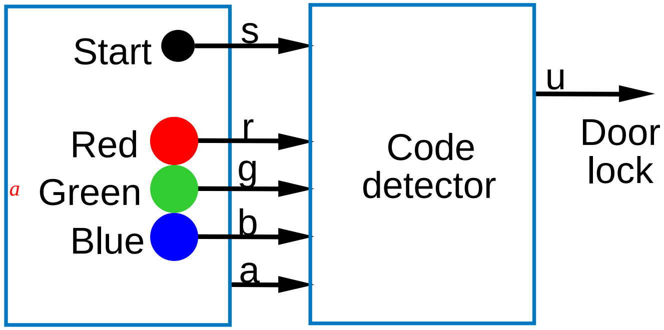
Q: Determine states and r value for given input waveform:

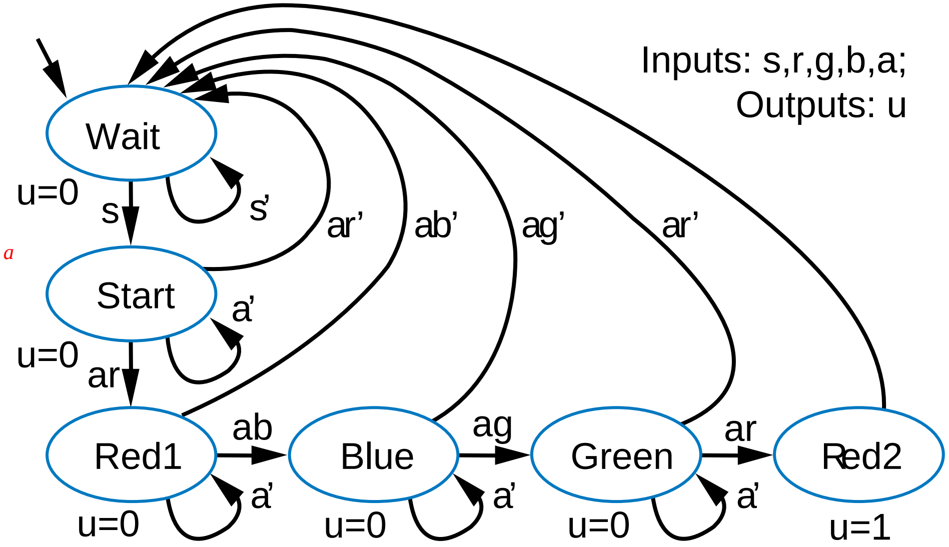




FSM Example: Code Detector

- 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

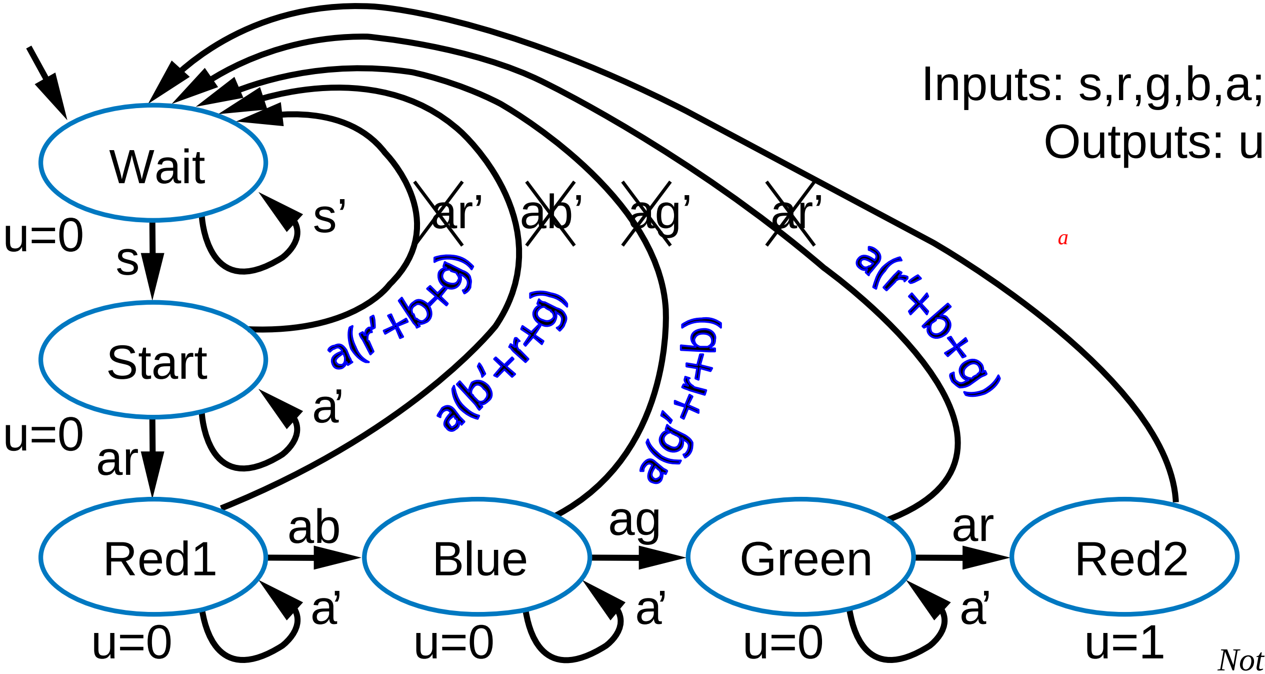




Q: Can you trick this FSM to open the door, without knowing the code?

A: Yes, hold all buttons simultaneously

Improve FSM for Code Detector



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

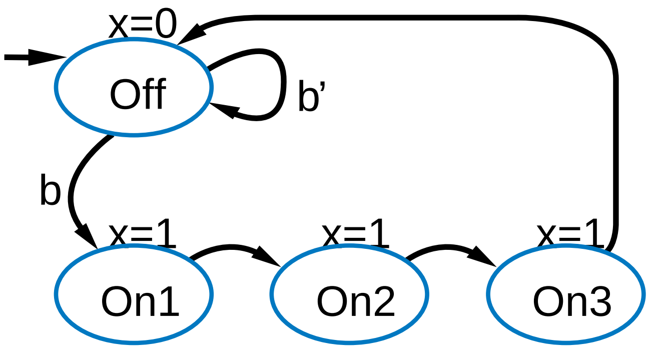
Note: small problem still remains; we'll discuss later

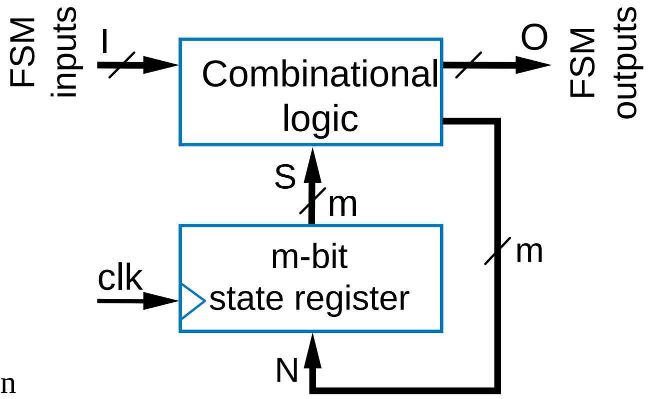
- New transition conditions detect if wrong button pressed, returns to "Wait"
- FSM provides formal, concrete means to accurately define desired behavior

Standard Controller Architecture

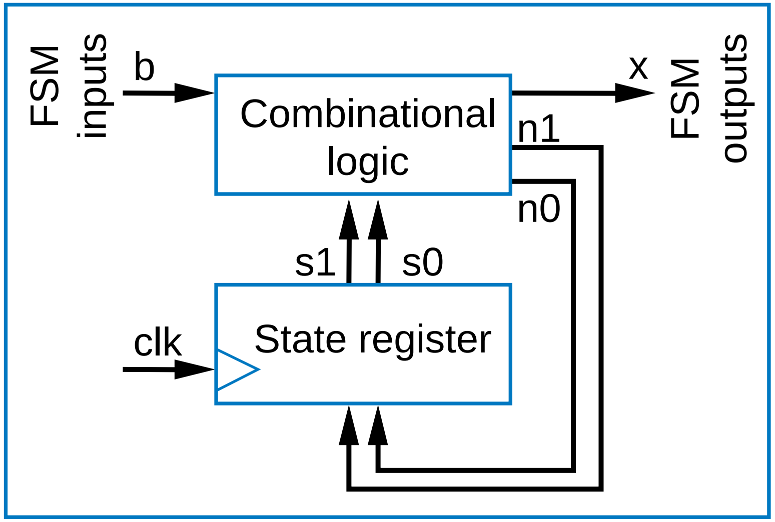
- How implement FSM as sequential circuit?
  - Use standard architecture
    - State register -- to store the present state
    - Combinational logic -- to compute outputs, and next state
    - For laser timer FSM
      - 2-bit state register, can represent four states
      - Input b, output x
  - Known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

Inputs: x; Outputs: b



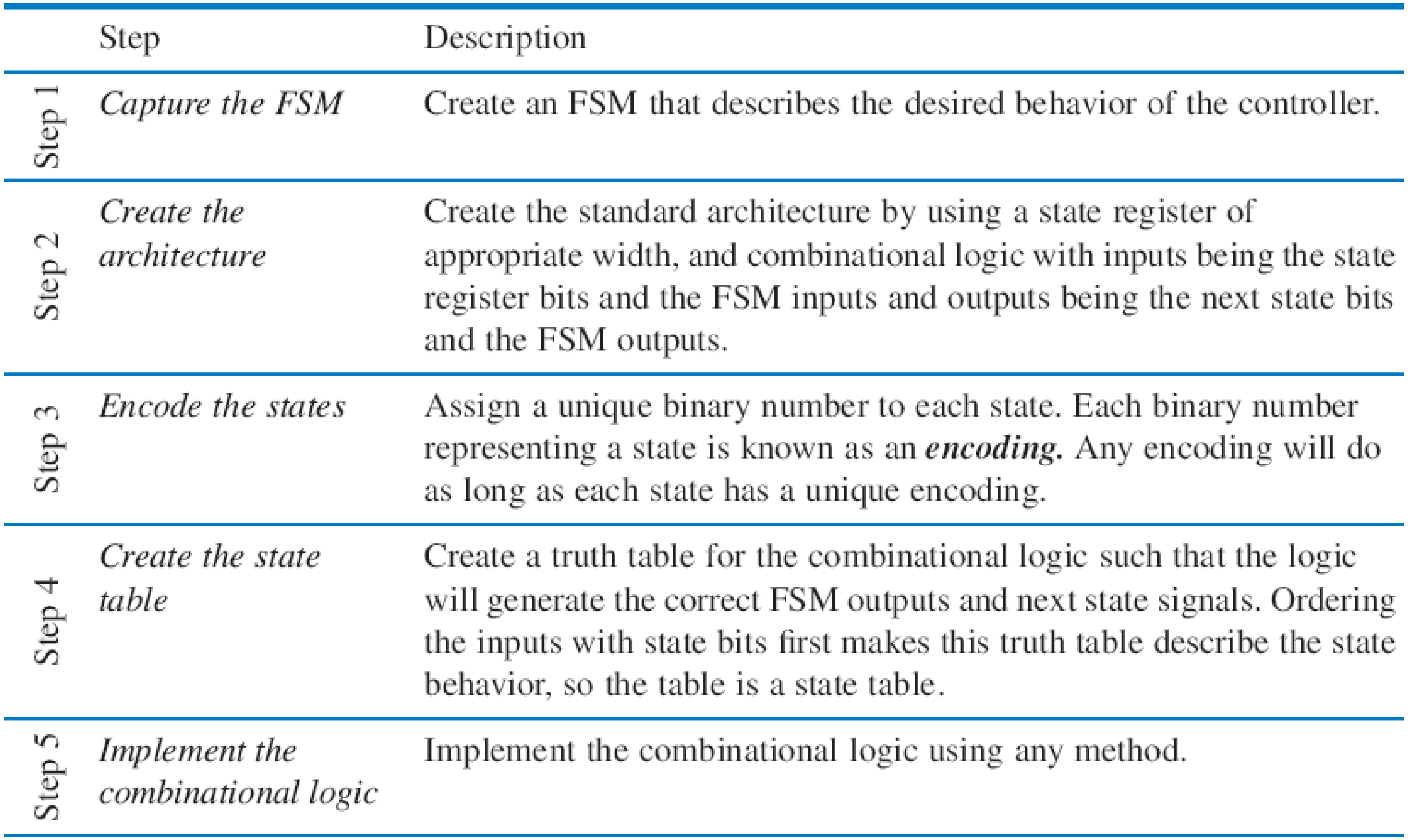


General version



Controller Design

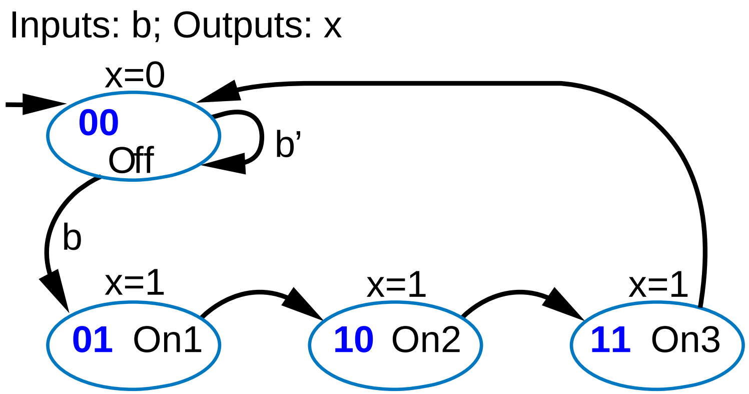
- Five step controller design process

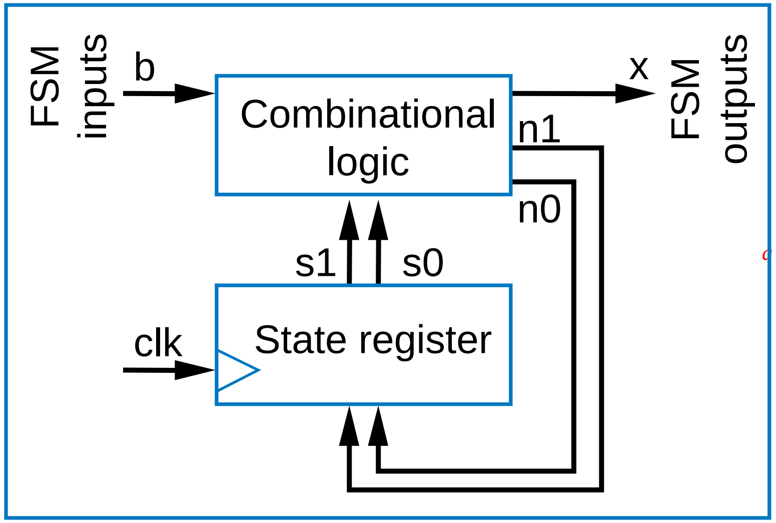


	Step	Description
Step 1	*Capture the FSM*	Create an FSM that describes the desired behavior of the controller.
Step 2	*Create the architecture*	Create the standard architecture by using a state register of appropriate width, and combinational logic with inputs being the state register bits and the FSM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

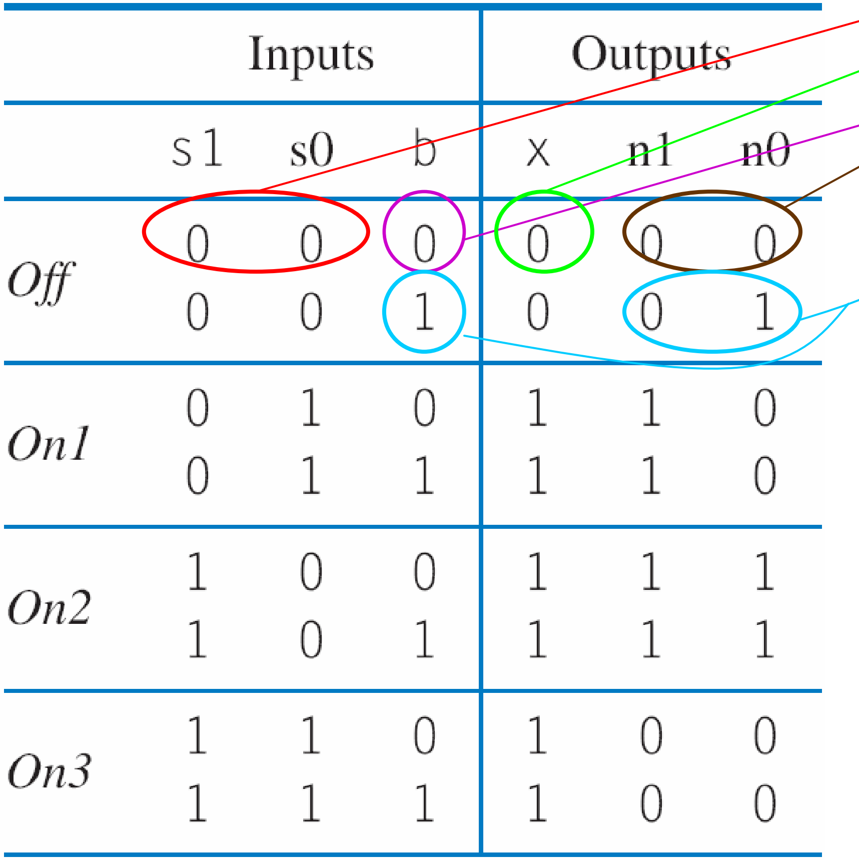
- Step 1: Capture the FSM
  - Already done
- Step 2: Create architecture
  - 2-bit state register (for 4 states)
  - Input b, output x
  - Next state signals n1, n0
- Step 3: Encode the states
  - Any encoding with each state unique will work



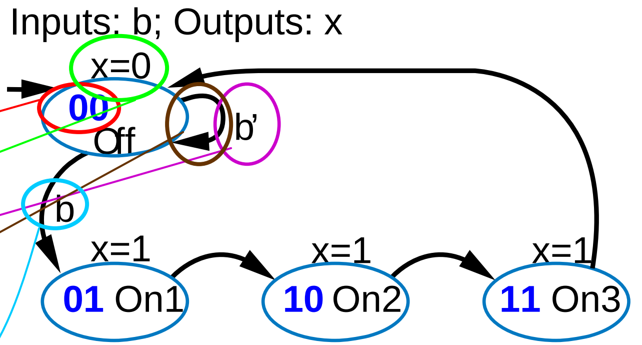


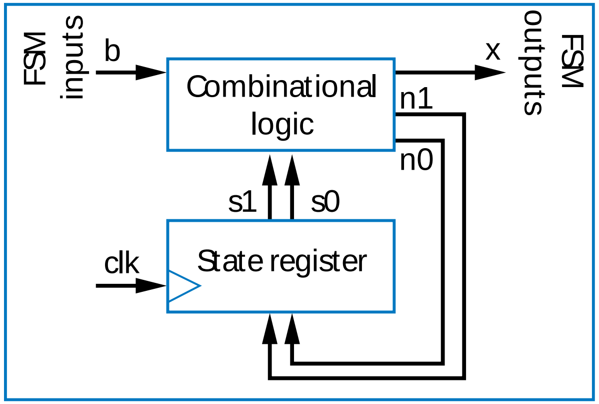
Controller Design: Laser Timer Example (cont)

- Step 4: Create state table



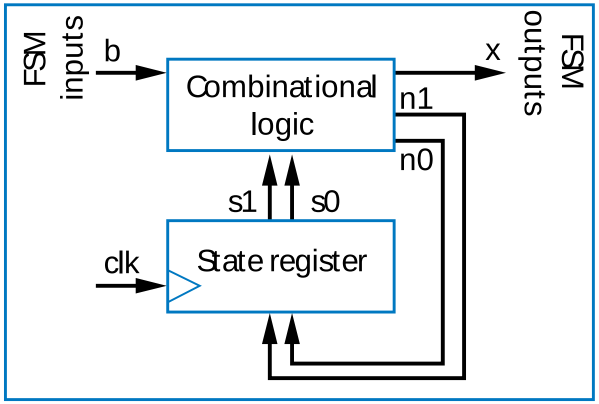
	Inputs			Outputs		
	s1	s0	b	x	n1	n0
Off	0	0	0	0	0	0
	0	0	1	0	0	1
On1	0	1	0	1	1	0
	0	1	1	1	1	0
On2	1	0	0	1	1	1
	1	0	1	1	1	1
On3	1	1	0	1	0	0
	1	1	1	1	0	0

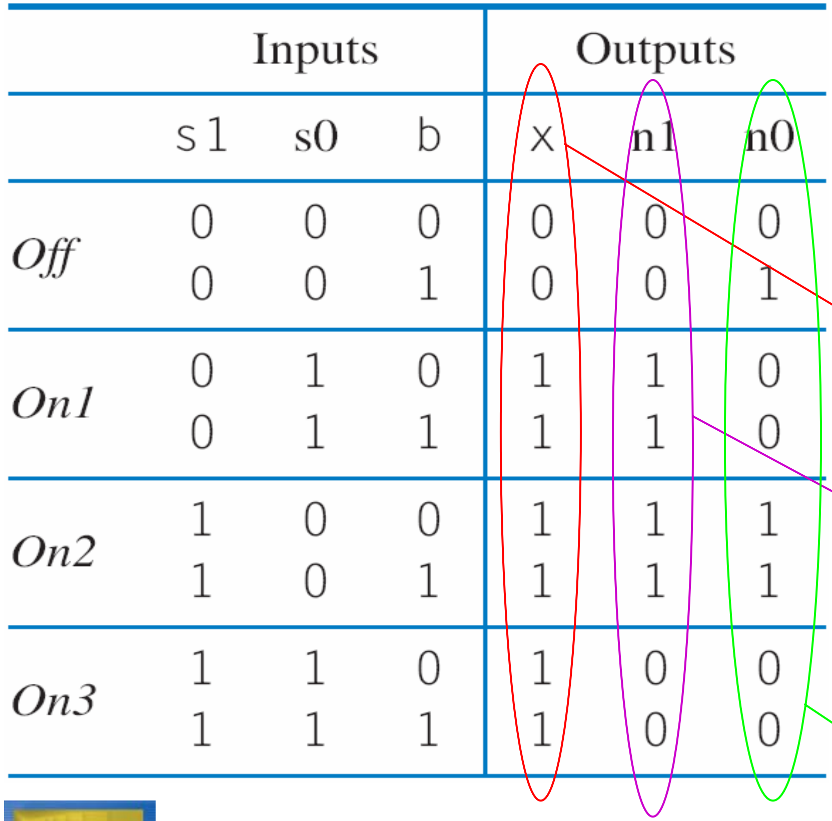




Controller Design: Laser Timer Example (cont)

- 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

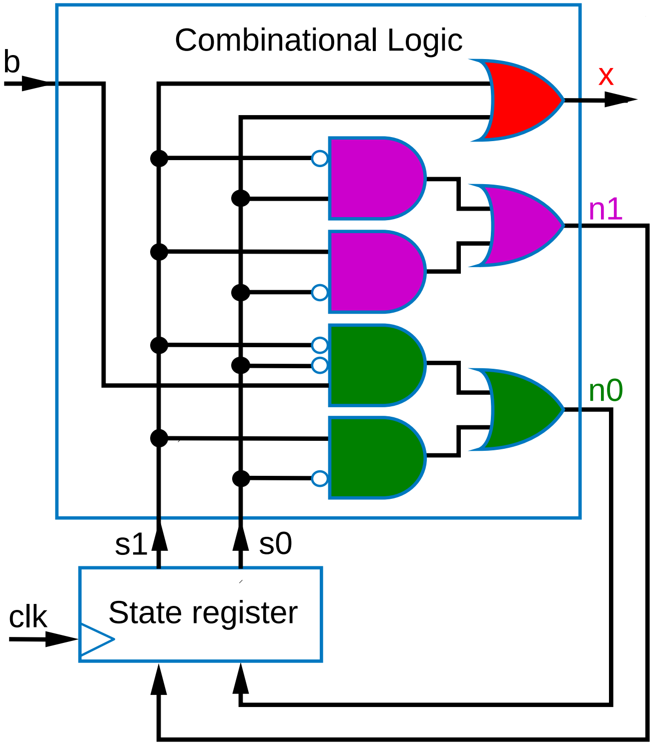
n1 = s1’s0 + s1s0’

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)



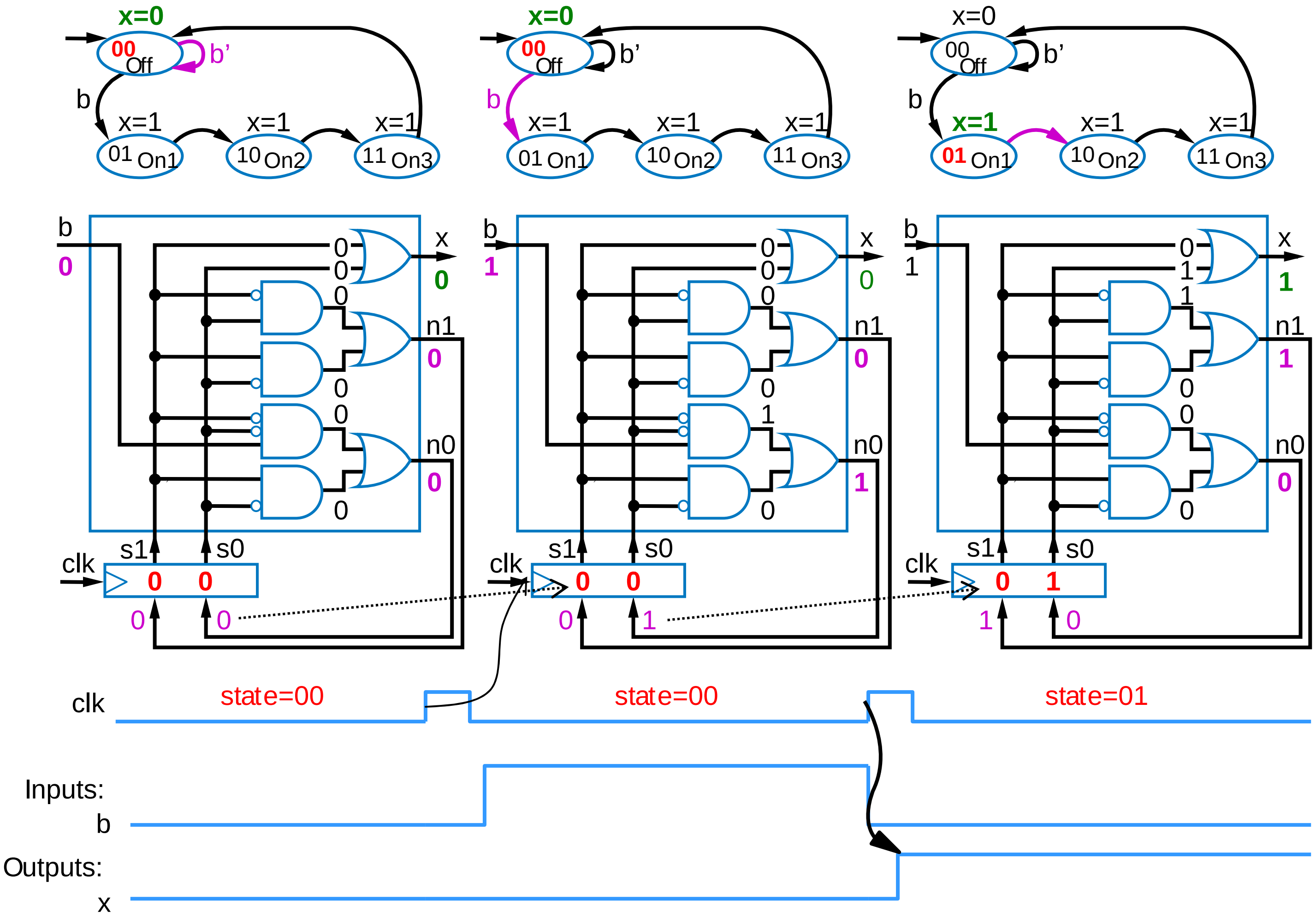
	Inputs			Outputs		
	s1	s0	b	x	n1	n0
Off	0	0	0	0	0	0
	0	0	1	0	0	1
On1	0	1	0	1	1	0
	0	1	1	1	1	0
On2	1	0	0	1	1	1
	1	0	1	1	1	1
On3	1	1	0	1	0	0
	1	1	1	1	0	0

x = s1 + s0

n1 = s1’s0 + s1s0’

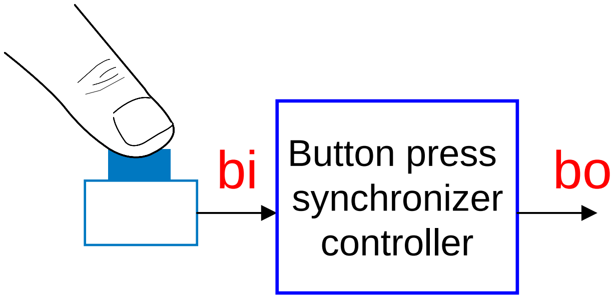
n0 = s1’s0’b + s1s0’

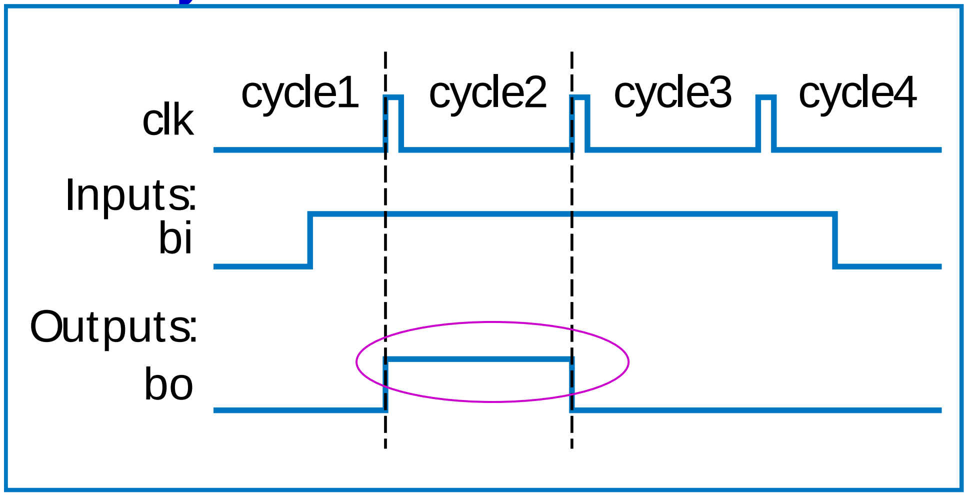
Understanding the Controller's Behavior



a

Controller Example:
Button Press Synchronizer

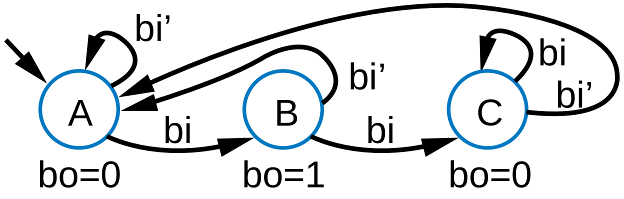




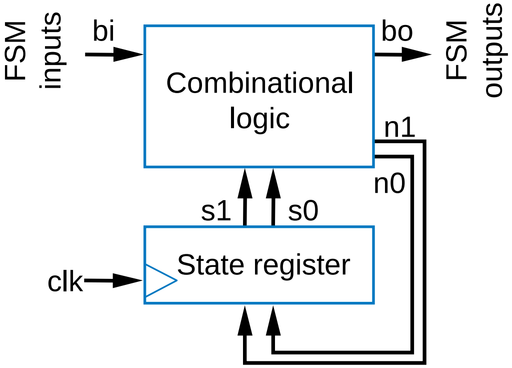
- Want simple sequential circuit that converts button press to single cycle duration, regardless of length of time that button actually pressed
  - We assumed such an ideal button press signal in earlier example, like the button in the laser timer controller

Controller Example: Button Press Synchronizer (cont)

FSM inputs: bi; FSM outputs: bo

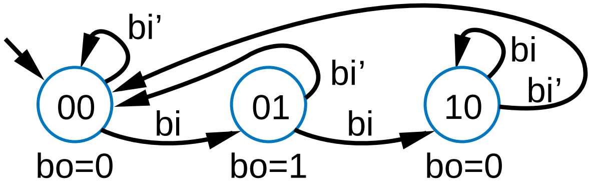


Step 1: FSM



Step 2: Create architecture

FSM inputs: bi; FSM outputs: bo

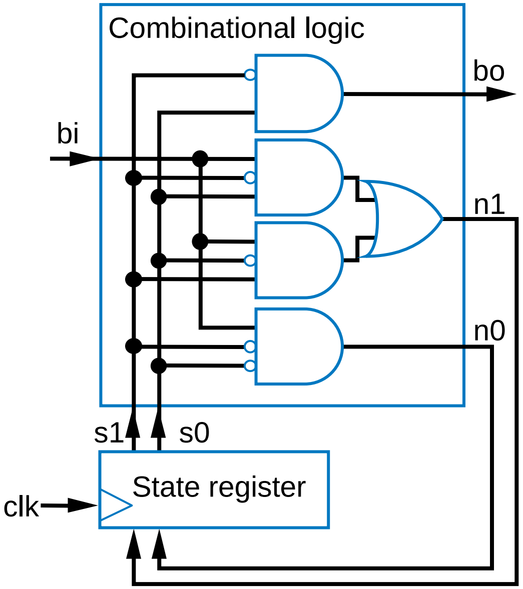


Step 3: Encode states

	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

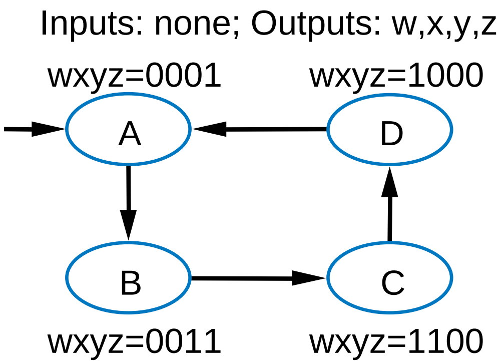


Step 5: Create combinational circuit

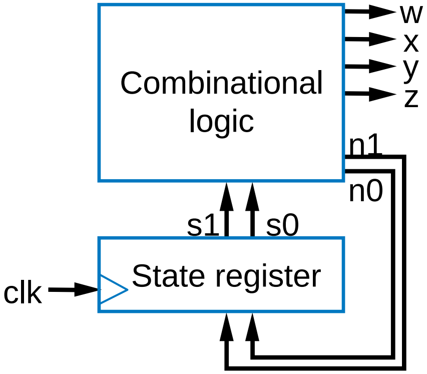
Step 5: Create combinational circuit

Controller Example: Sequence Generator

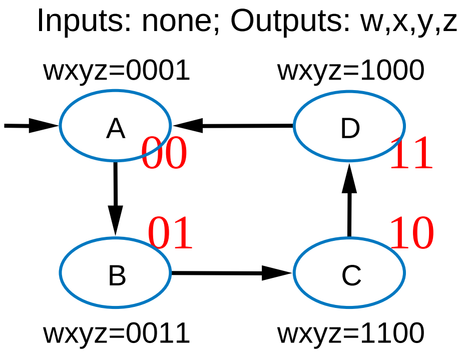
- Want generate sequence 0001, 0011, 1100, 1000, (repeat)
  - Each value for one clock cycle
  - Common, e.g., to create pattern in 4 lights, or control magnets of a “stepper motor”



Step 1: Create FSM



Step 2: Create architecture

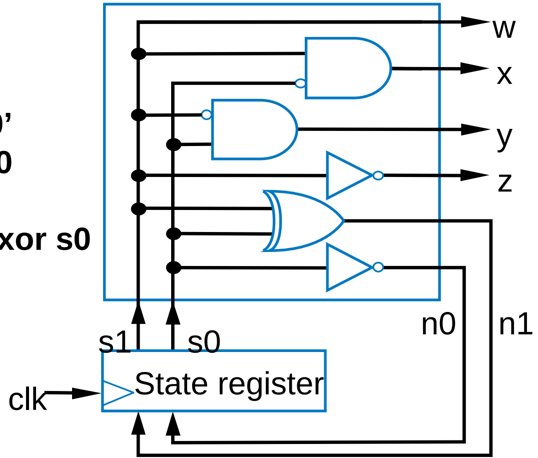


Step 3: Encode states

	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’

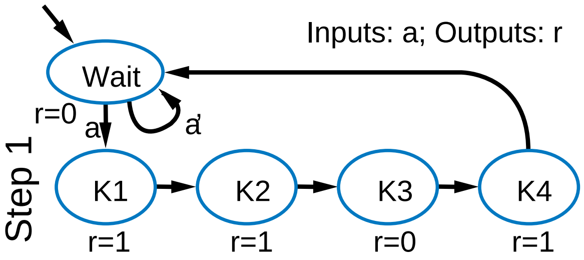


Step 5: Create combinational circuit

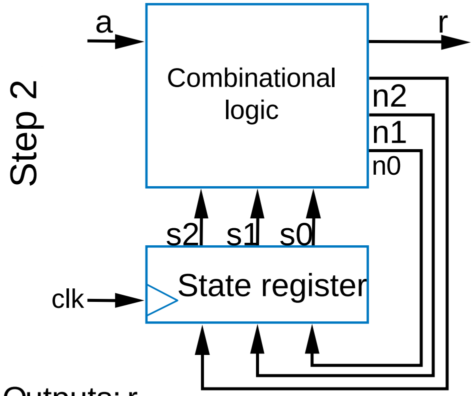
a

Controller Example: Secure Car Key

- (from earlier example)

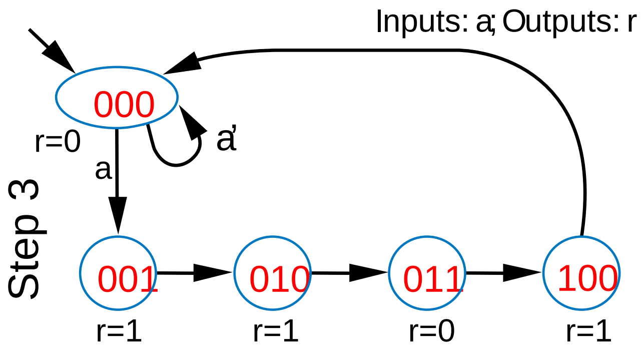


a



	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

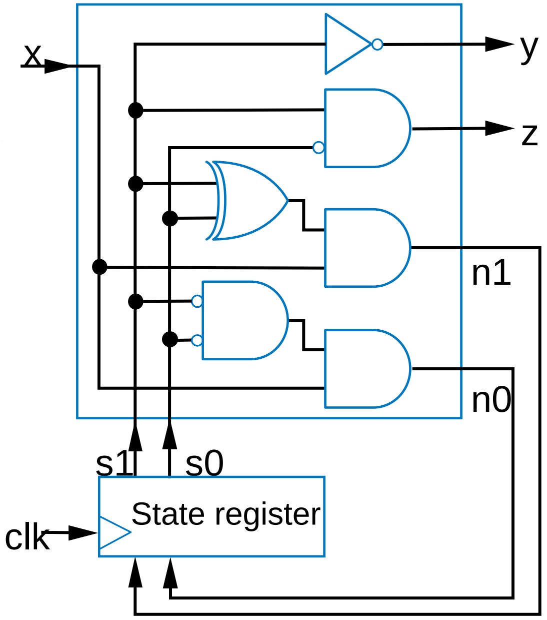
Step 4



We'll omit Step 5

Example: Seq. Circuit to FSM (Reverse Engineering)

What does this circuit do?



Work backwards

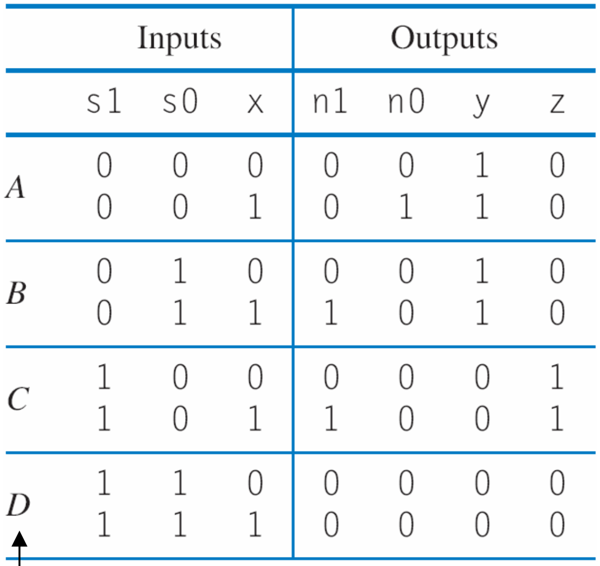
$y=s1'$

$z = s1s0'$

$n1=(s1 \text{ xor } s0)x$

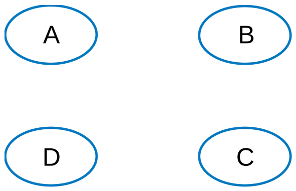
$n0=(s1'*s0')x$

a



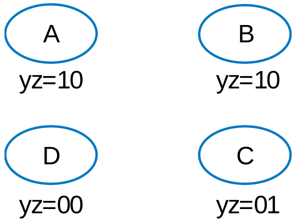
	Inputs			Outputs			
	s1	s0	x	n1	n0	y	z
A	0	0	0	0	0	1	0
	0	0	1	0	1	1	0
B	0	1	0	0	0	1	0
	0	1	1	1	0	1	0
C	1	0	0	0	0	0	1
	1	0	1	1	0	0	1
D	1	1	0	0	0	0	0
	1	1	1	0	0	0	0

Pick any state names you want



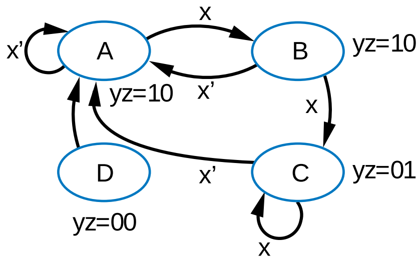
states

Outputs: y, z



states with outputs

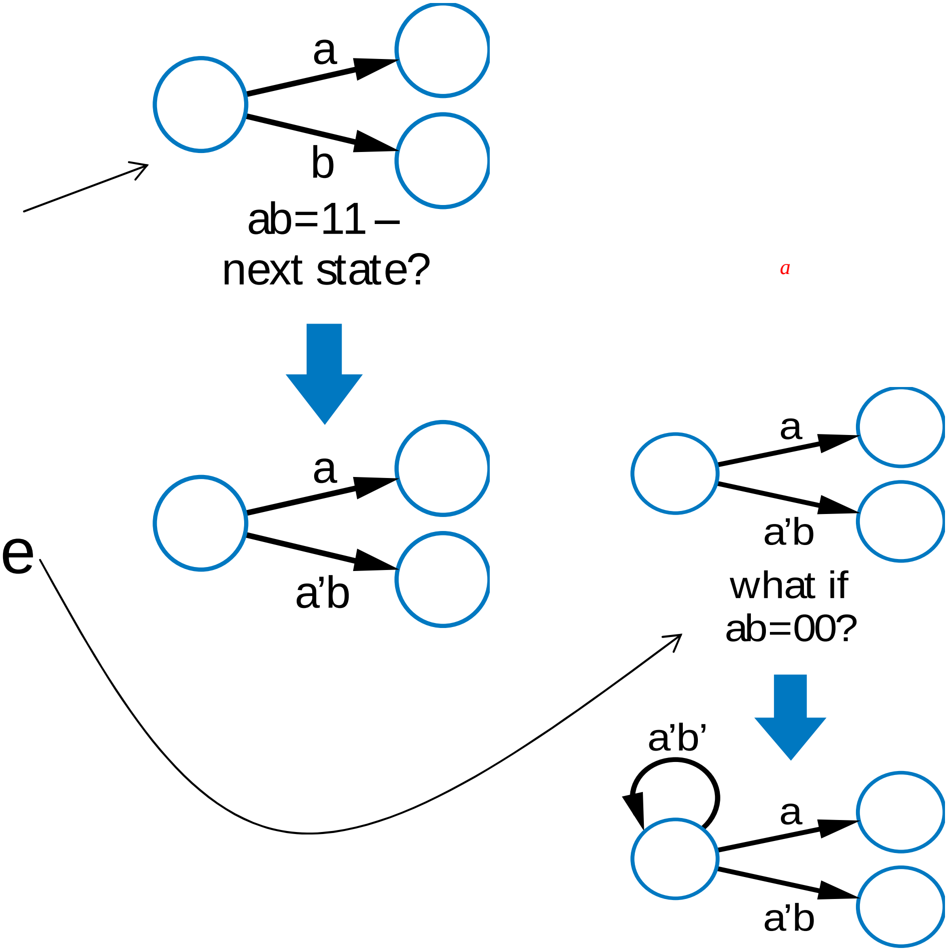
Inputs: x; Outputs: y, z



states with outputs and transitions

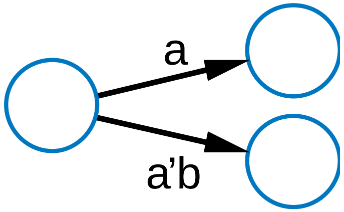
Common Pitfalls Regarding Transition Properties

- *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?



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 → proves pair can never simultaneously be true
  - One condition true: OR of all conditions of transitions leaving a state) should equal 1 → proves at least one condition must be true
  - Example



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)

Answer:

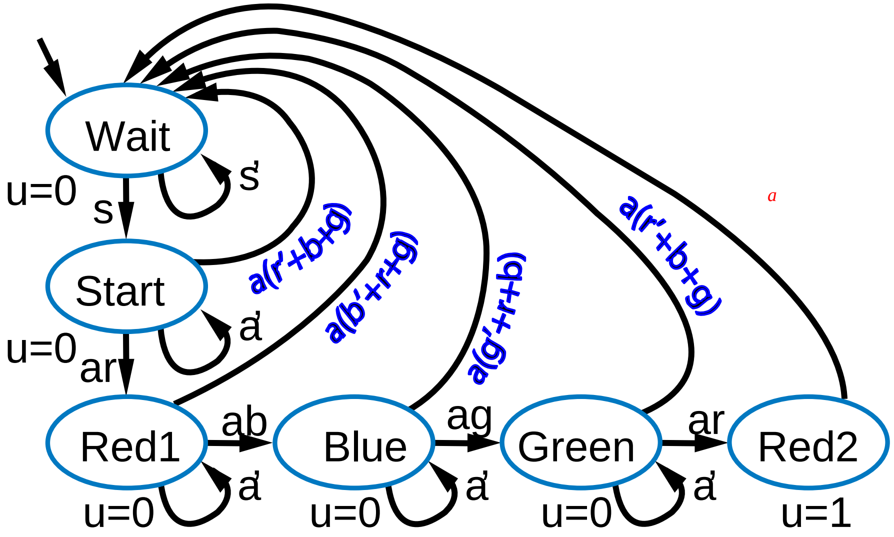
a * a’b
= (a * a’) * b
= 0 * b
= 0
OK!

a

a + a’b
= a*(1+b) + a’b
= a + ab + a’b
= a + (a+a’)b
= a + b
Fails! Might not be 1 (i.e., a=0, b=0)

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



$ar * a'$
$= (a*a')r$
$= 0$

$a' * a(r'+b+g)$
$= 0*r$
$= 0$

$ar * a(r'+b+g)$
$= (a'*a)*(r'+b+g) = 0*(r'+b+g)$
$= (a*a)*r*(r'+b+g) = a*r*(r'+b+g)$
$= arr'+arb+arg$
$= 0 + arb+arg$
$= arb + arg$
$= ar(b+g)$

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?

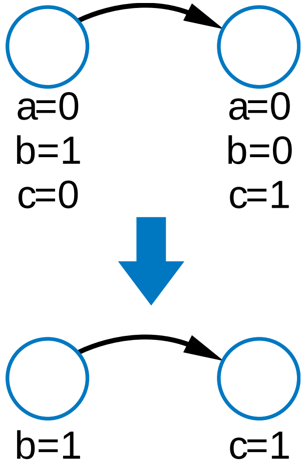
Q: How to solve?

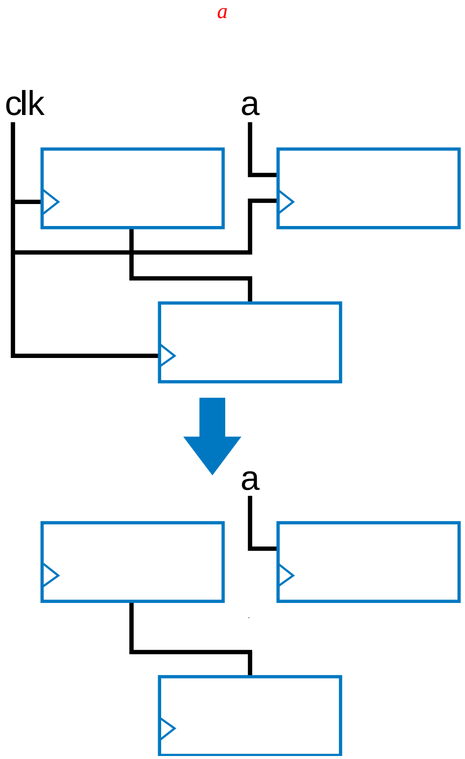
A: ar should be arb’g’ (likewise for ab, ag, ar)

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



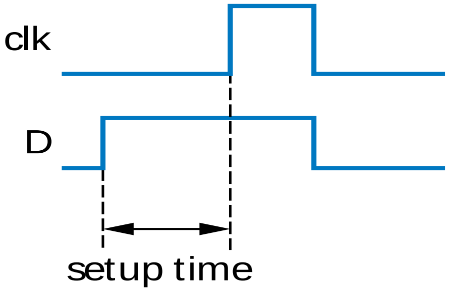


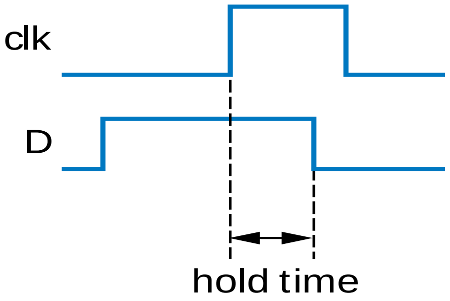
More on Flip-Flops and Controllers

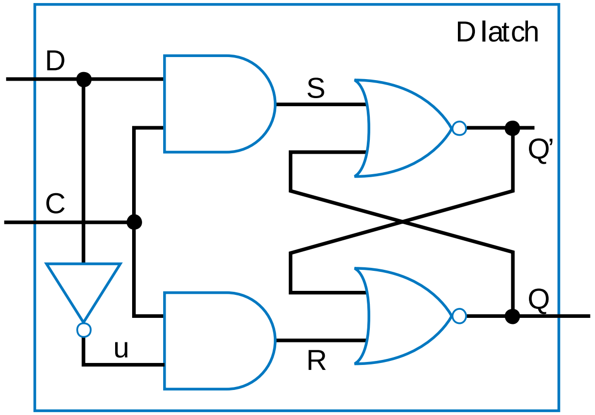
- Other flip-flop types
  - SR flip-flop: like SR latch, but edge triggered
  - JK flip-flop: like SR (S→J, R→K)
    - But when JK=11, toggles
    - 1→0, 0→1
  - T flip-flop: JK with inputs tied together
    - Toggles on every rising clock edge
  - Previously utilized to minimize logic outside flip-flop
    - Today, minimizing logic to such extent is not as important
    - D flip-flops are thus by far the most common

Non-Ideal Flip-Flop Behavior

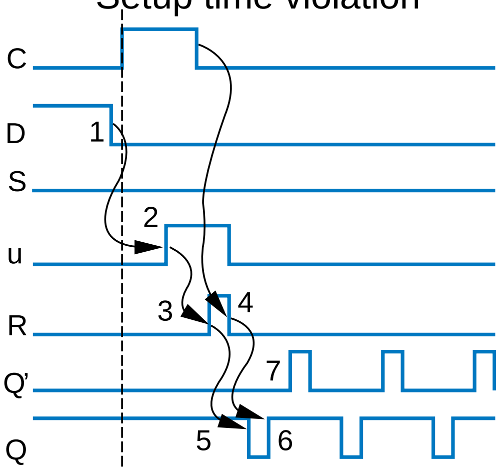
- Can’t change flip-flop input too close to clock edge
  - Setup time: time that D must be stable *before* edge
    - Else, stable value not present at internal latch
  - Hold time: time that D must be held stable *after* edge
    - Else, new value doesn’t have time to loop around and stabilize in internal latch







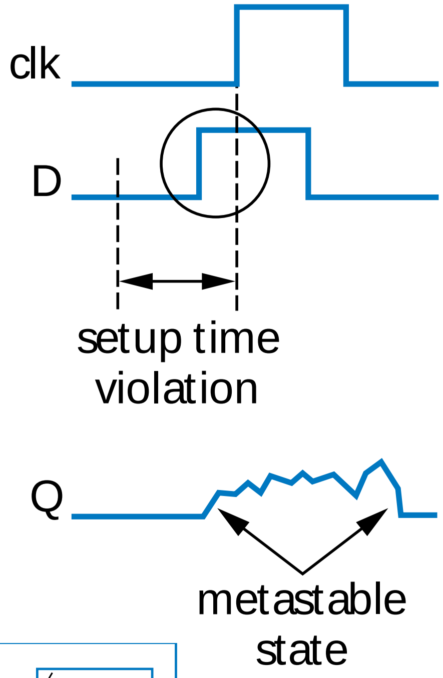
Setup time violation

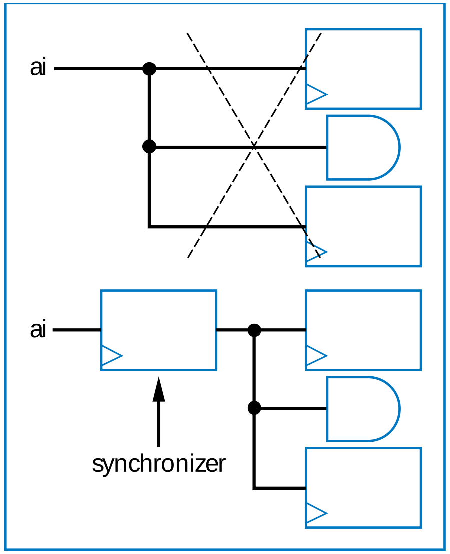


Leads to oscillation!

Metastability

- Violating setup/hold time can lead to bad situation known as ***metastable*** state
  - Metastable state: Any flip-flop state other than stable 1 or 0
    - Eventually settles to one or other, but we don't know which
  - For internal circuits, we can make sure observe setup time
  - But what if input comes from external (asynchronous) source, e.g., button press?
- Partial solution
  - Insert synchronizer flip-flop for asynchronous input
    - Special flip-flop with very small setup/hold time
  - Doesn't completely prevent metastability



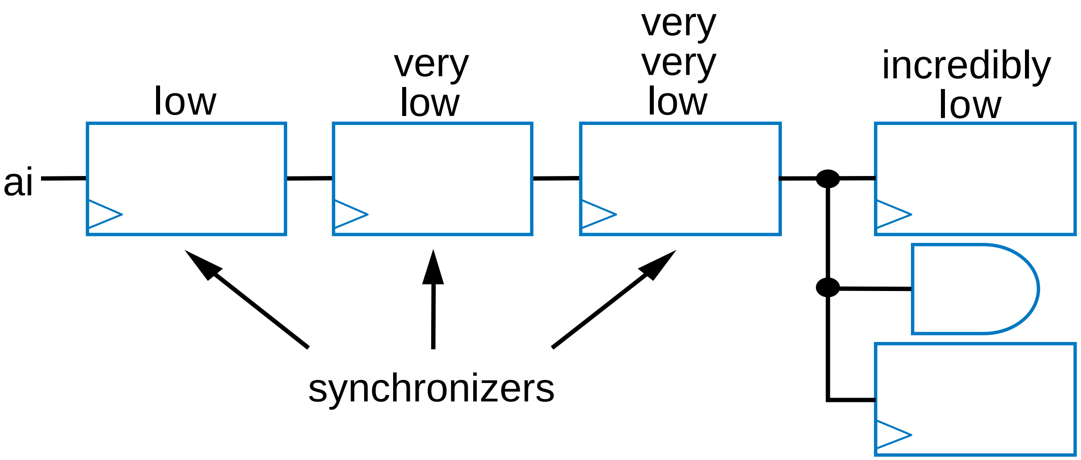


a

Metastability

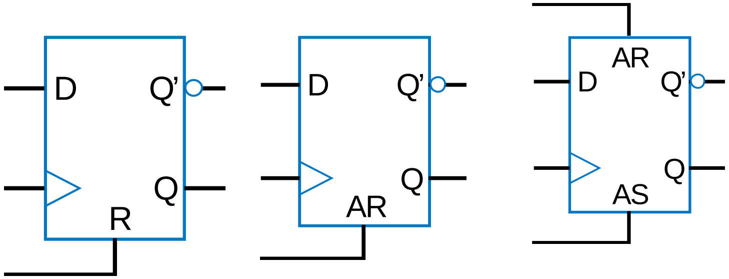
- One flip-flop doesn't completely solve problem
- How about adding more synchronizer flip-flops?
  - Helps, but just decreases probability of metastability
- So how solve completely?
  - Can't! May be unsettling to new designers. But we just can't guarantee a design that won't ever be metastable. We can just minimize the mean time between failure (MTBF) -- a number often given along with a circuit

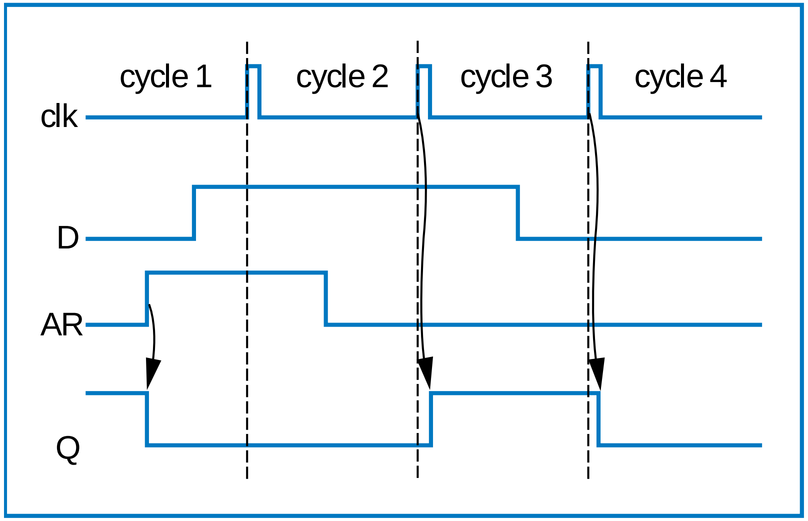
Probability of flip-flop being metastable is…



Flip-Flop Set and Reset Inputs

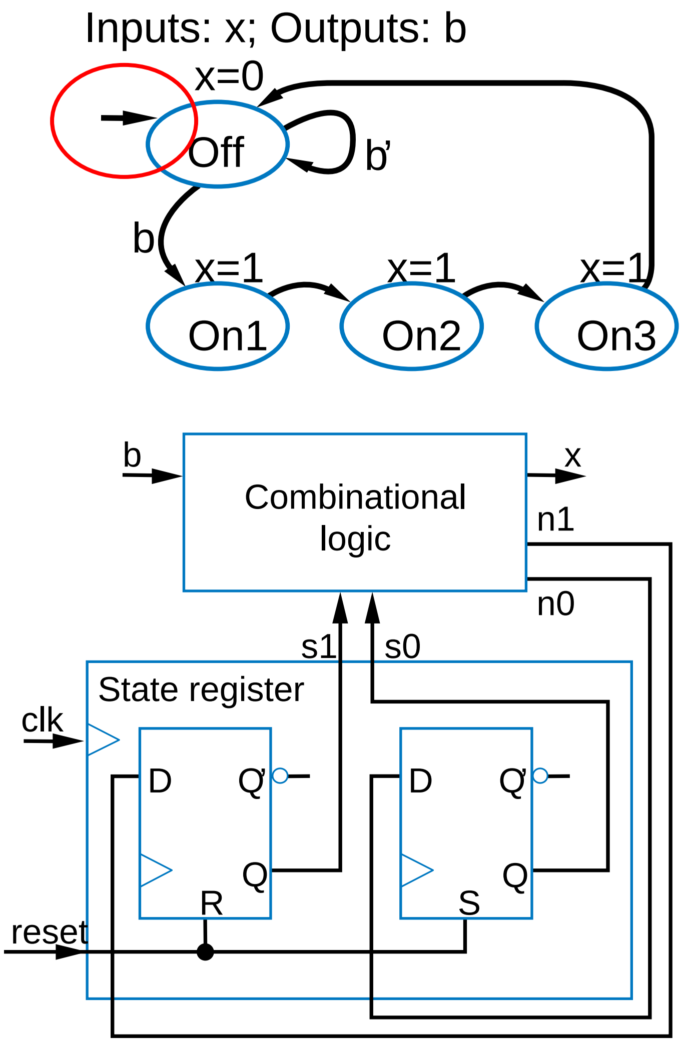
- Some flip-flops have additional inputs
  - Synchronous reset: clears Q to 0 on next clock edge
  - Synchronous set: sets Q to 1 on next clock edge
  - Asynchronous reset: clear Q to 0 immediately (not dependent on clock edge)
    - Example timing diagram shown
  - Asynchronous set: set Q to 1 immediately





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

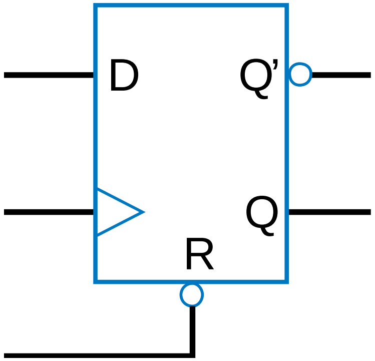


Glitching

- Glitch: Temporary values on outputs that appear soon after input changes, before stable new output values
- Designer must determine whether glitching outputs may pose a problem
  - If so, may consider adding flip-flops to outputs
    - Delays output by one clock cycle, but may be OK

Active Low Inputs

- We've assumed input action occur when input is 1
  - Some inputs are instead active when input is 0 -- "active low"
  - Shown with inversion bubble
  - So to reset the shown flip-flop, set R=0. Else, keep R=1.



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