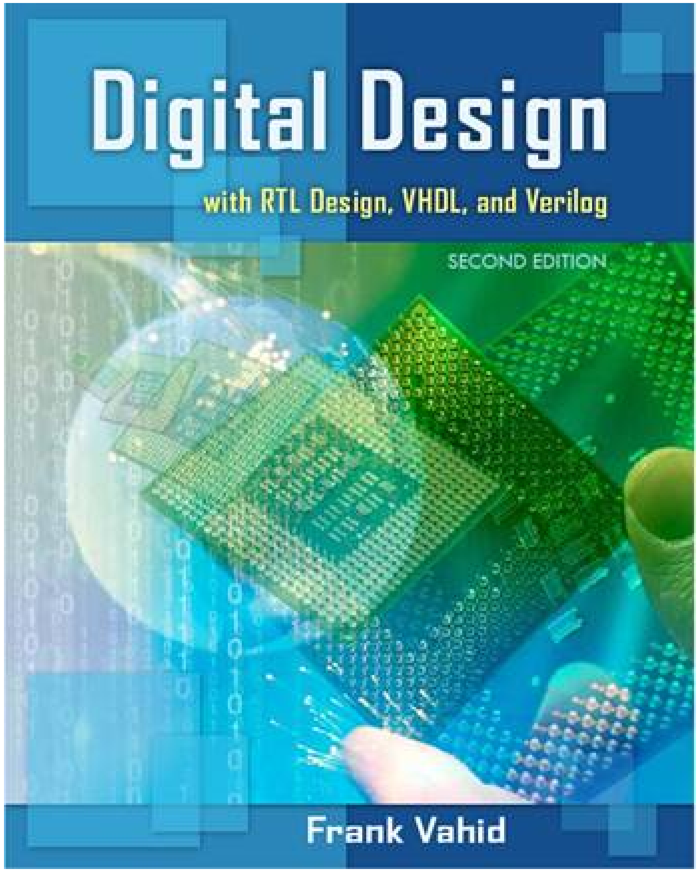


Digital Design

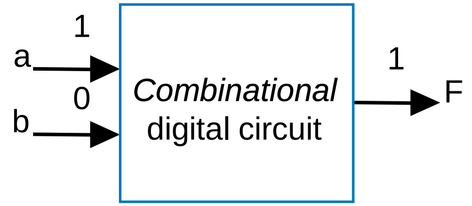
Chapter 3:

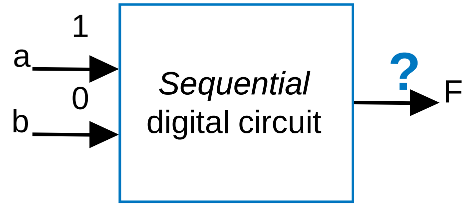
Sequential Logic Design -- Controllers

Slides to accompany the textbook *Digital Design, with RTL Design, VHDL, and Verilog*, 2nd Edition,
by Frank Vahid, John Wiley and Sons Publishers, 2010.
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
- This chapter will:
  - Design a new building block, a **flip-flop**, to store one bit
  - Combine flip-flops to build multi-bit storage – **register**
  - Describe sequential behavior with **finite state machines**
  - Convert a finite state machine to a **controller** – sequential circuit with a register and combinational logic





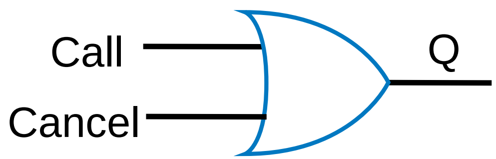
Must know sequence of past inputs to know output

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

Storing One Bit – Flip-Flops

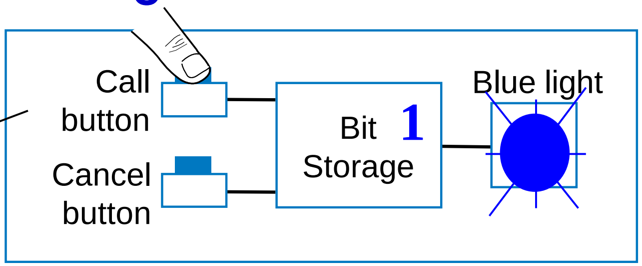
Example Requiring Bit Storage

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

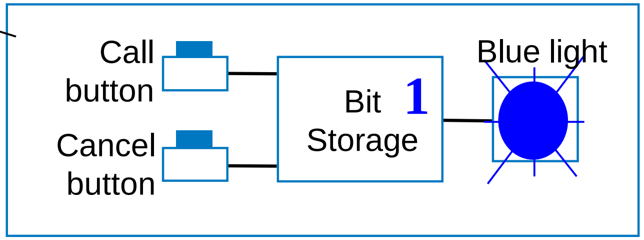


Doesn’t work. Q=1 when Call=1, but doesn’t stay 1 when Call returns to 0

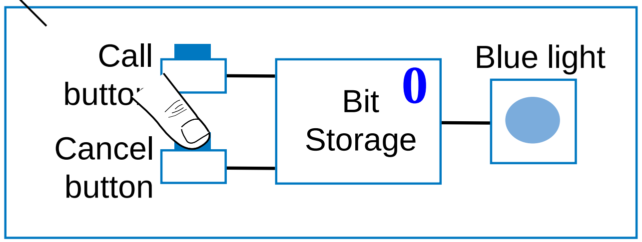
Need some form of “feedback” in the circuit



1. Call button pressed – light turns on



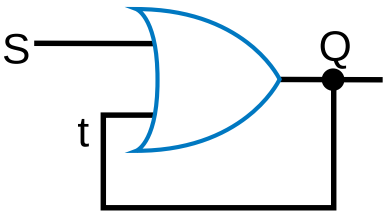
*2. Call button released – light **stays on***

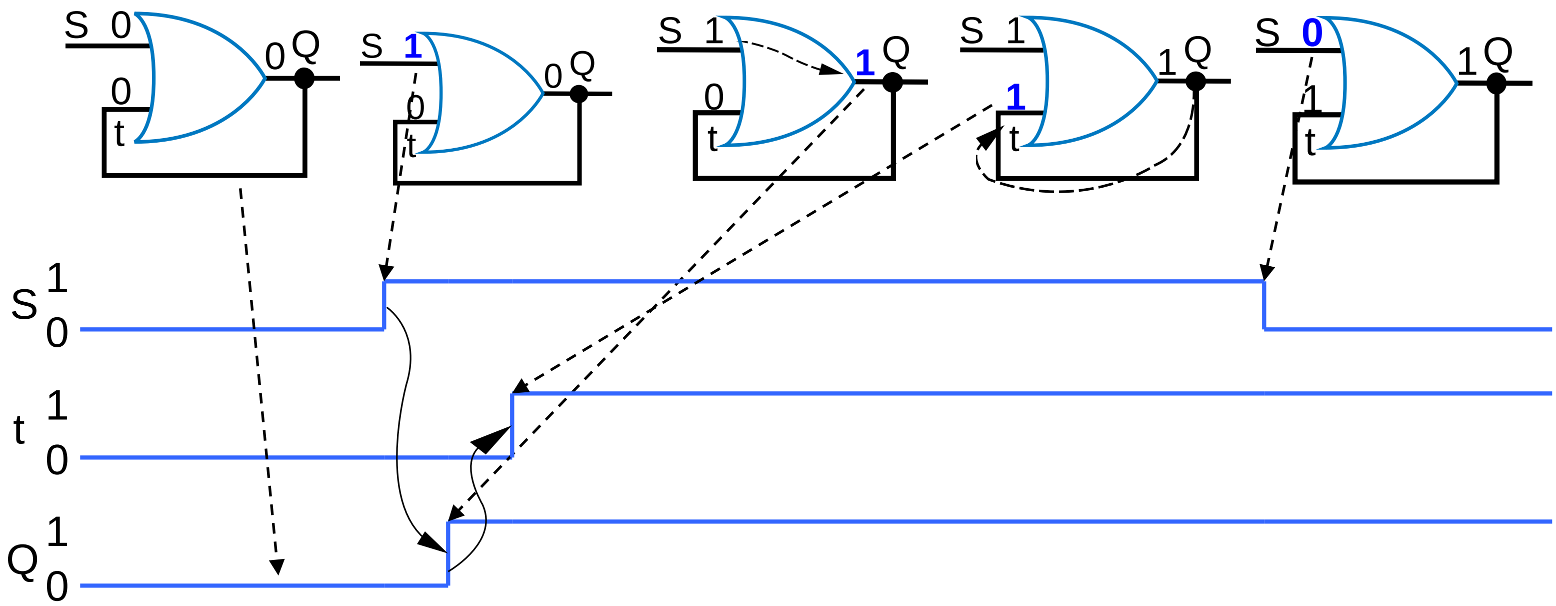


3. Cancel button pressed – light turns off

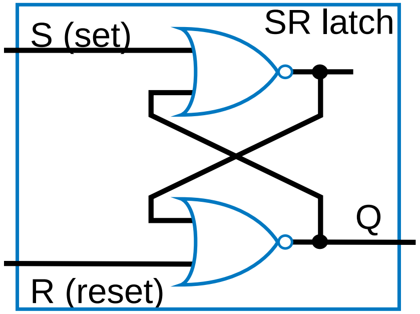
First attempt at Bit Storage

- 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

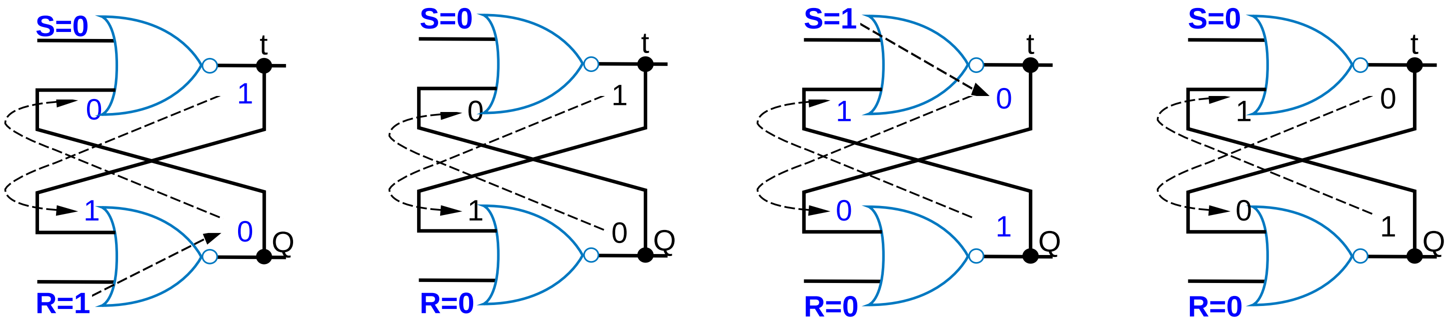




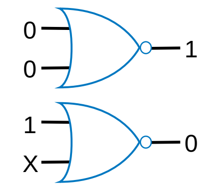
Bit Storage Using an SR Latch

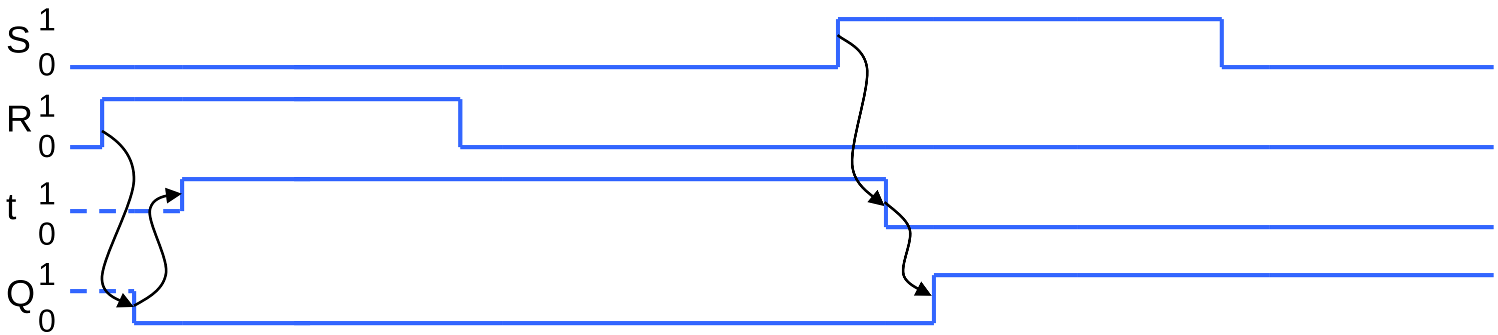


- Does the circuit to the right, with cross-coupled NOR gates, do what we want?
  - Yes! How did someone come up with that circuit? Maybe just trial and error, a bit of insight...



Recall NOR…

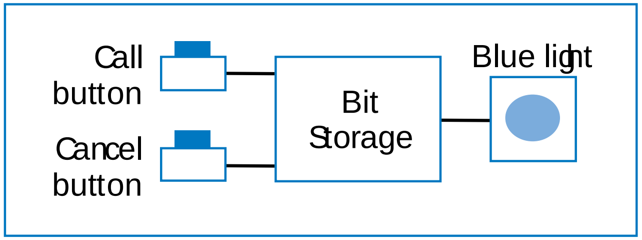


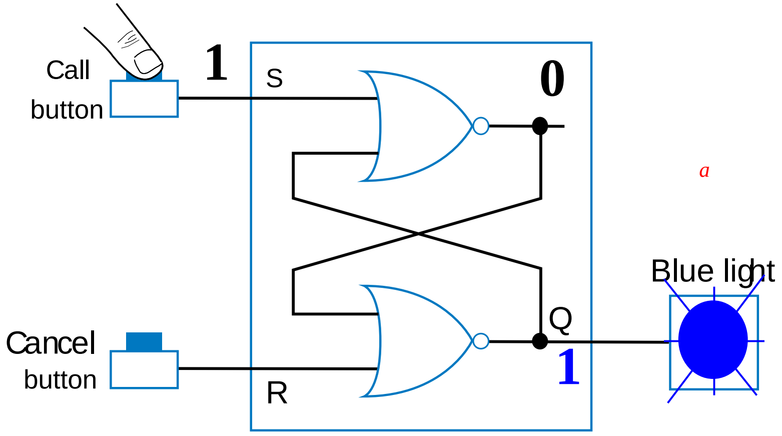


Example Using SR Latch for Bit Storage

- 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

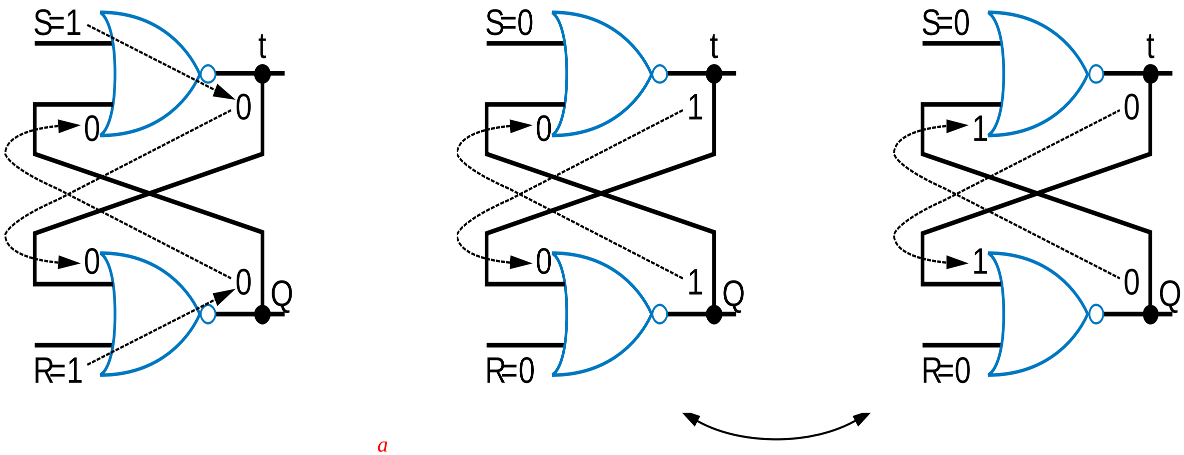
- But, there's a problem...

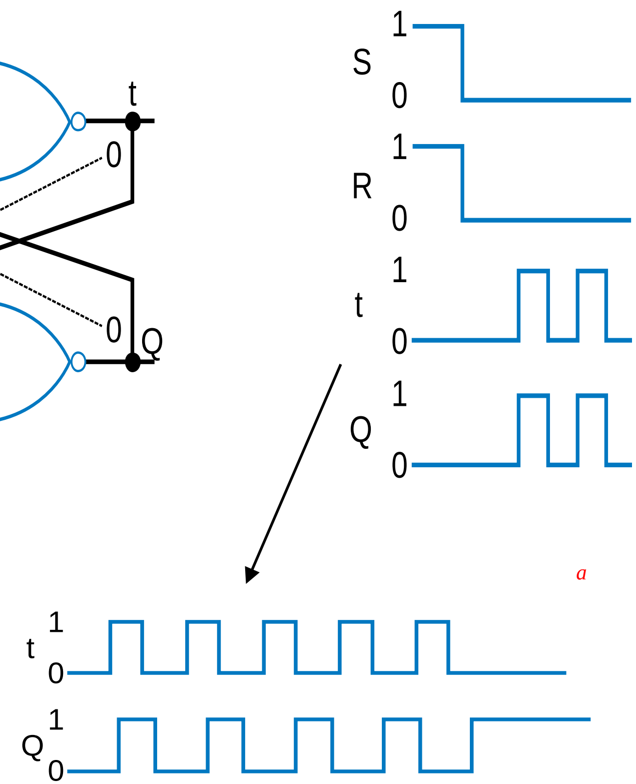




Problem with SR Latch

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

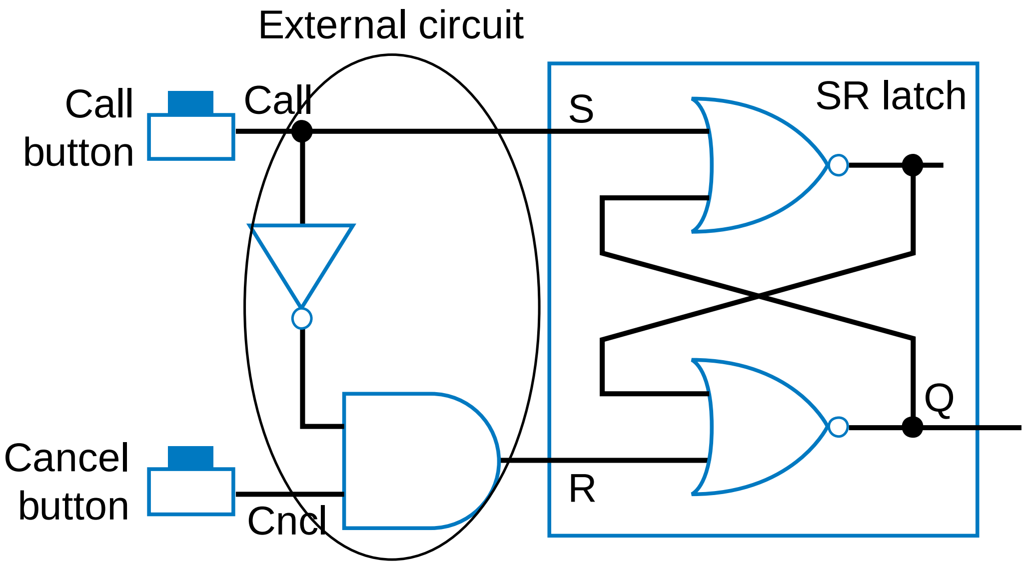


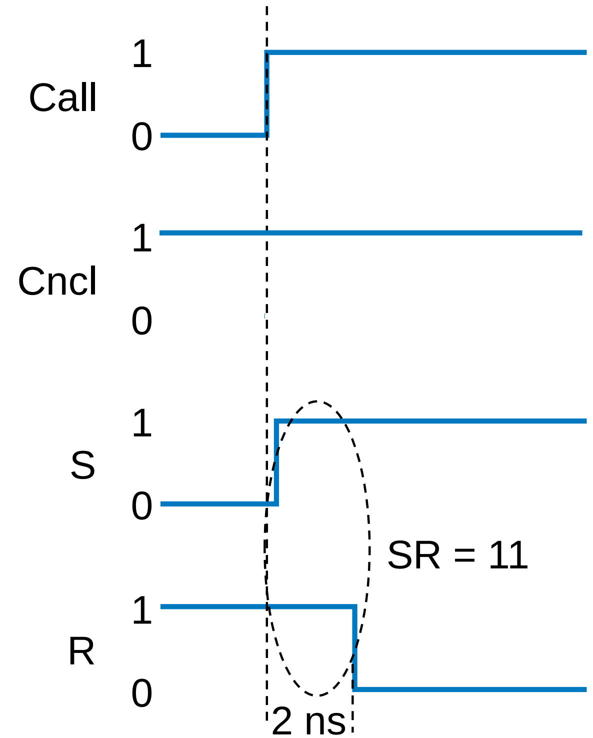


Q may oscillate. Then, because one path will be slightly longer than the other, Q will eventually settle to 1 or 0 – but we don't know which. Known as a *race condition.*

Problem with SR Latch

- Designer might try to avoid problem using external circuit
  - Circuit should prevent SR from ever being 11
  - But 11 can occur due to different path delays

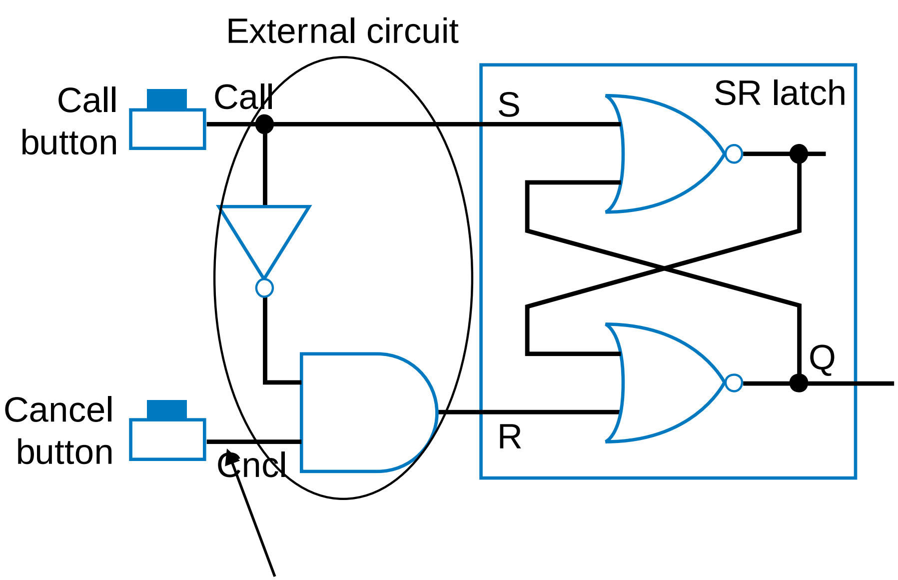




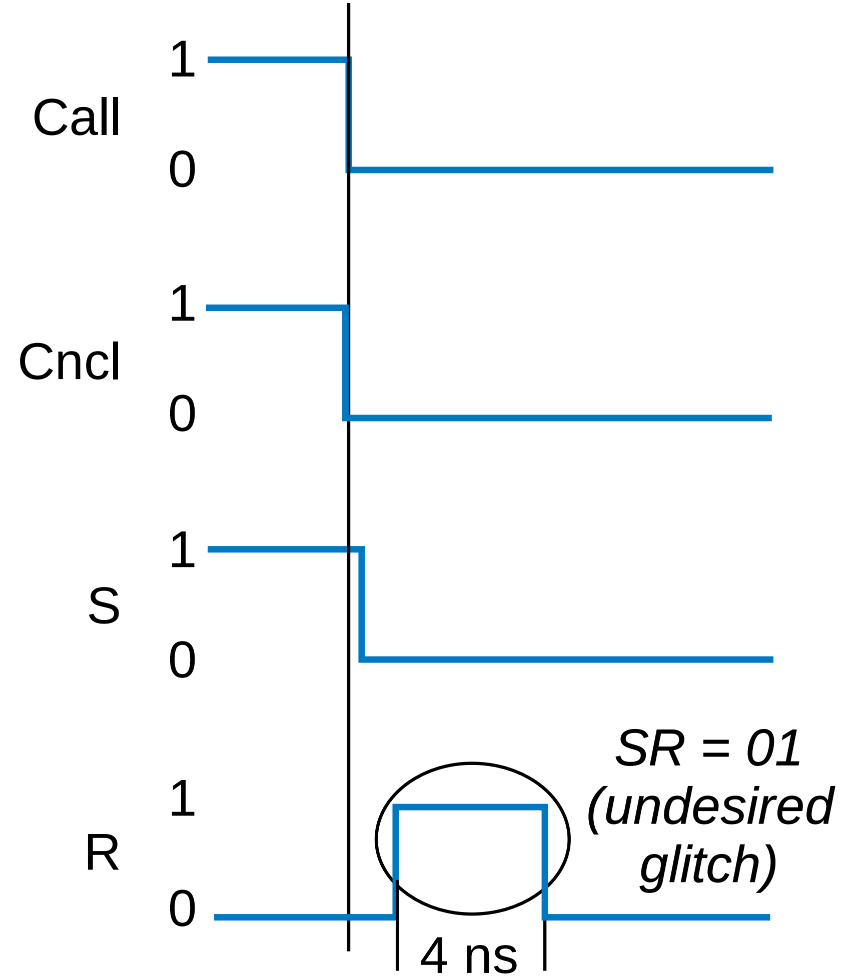
Assume 1 ns delay per gate. The longer path from Call to R than from Call to S causes SR=11 for short time – could be long enough to cause oscillation

Problem with SR Latch

- Glitch can also cause undesired set or reset

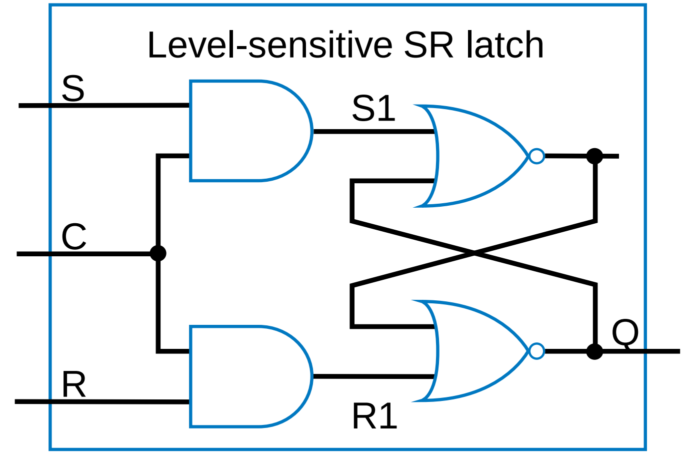


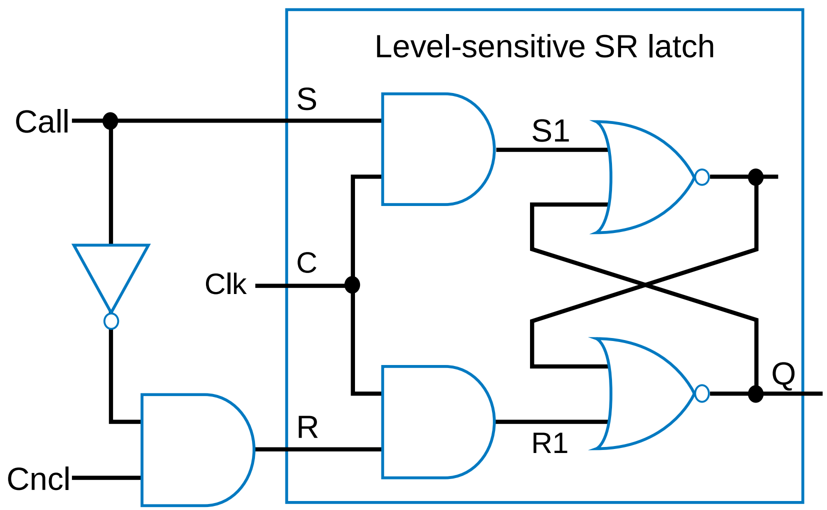
Suppose this wire has 4 ns delay

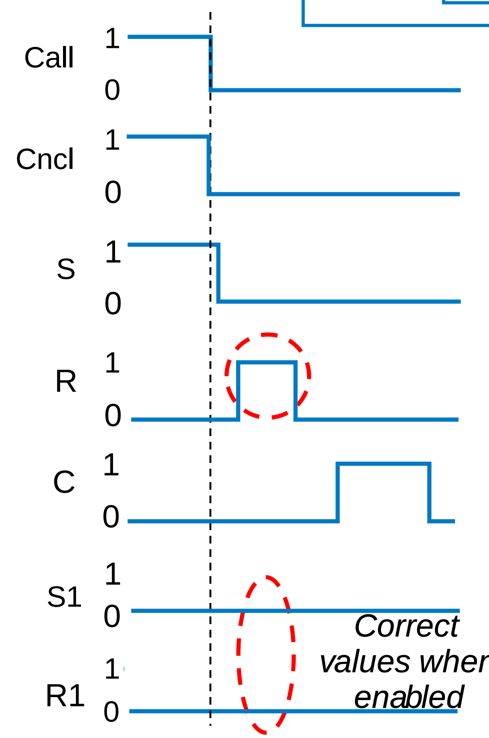


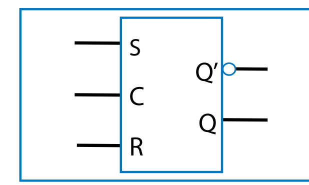
Solution: Level-Sensitive SR Latch

- Add enable input "C"
- Only let S and R change when C=0
  - Ensure circuit in front of SR never sets SR=11, except briefly due to path delays
  - Set C=1 after 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.







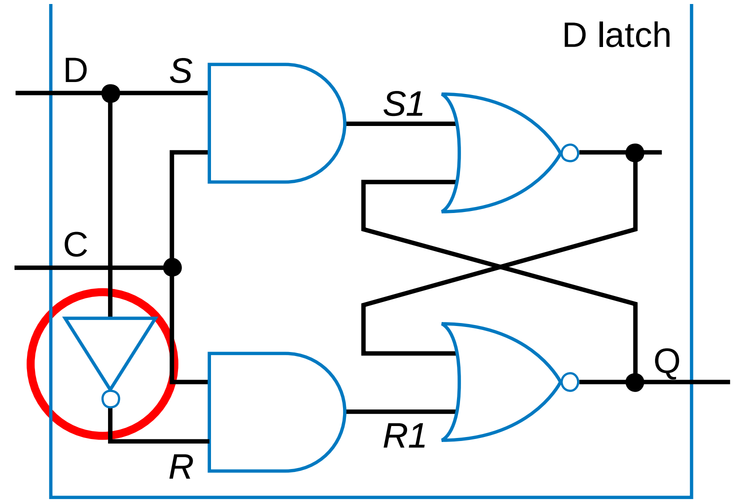


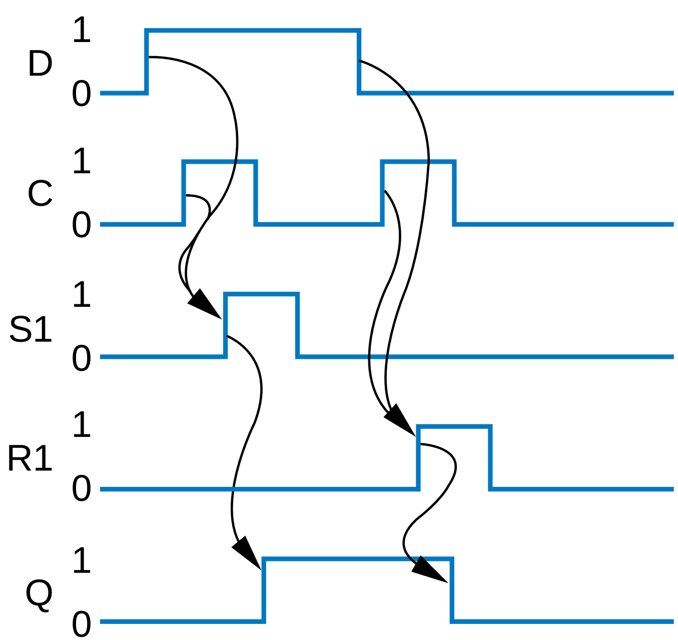
Level-sensitive SR latch symbol

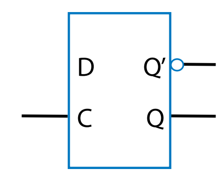
Glitch on R (or S) doesn't affect R1 (or (S1)

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



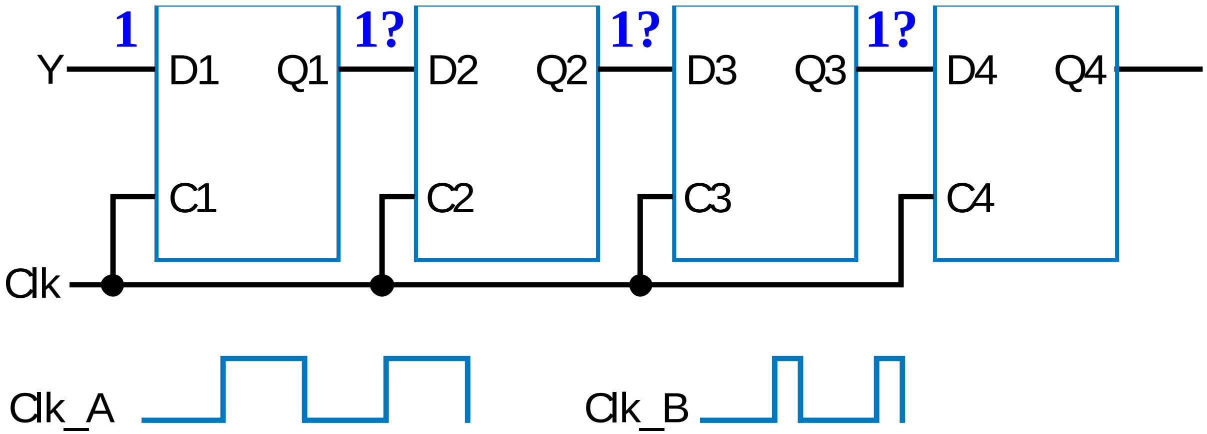




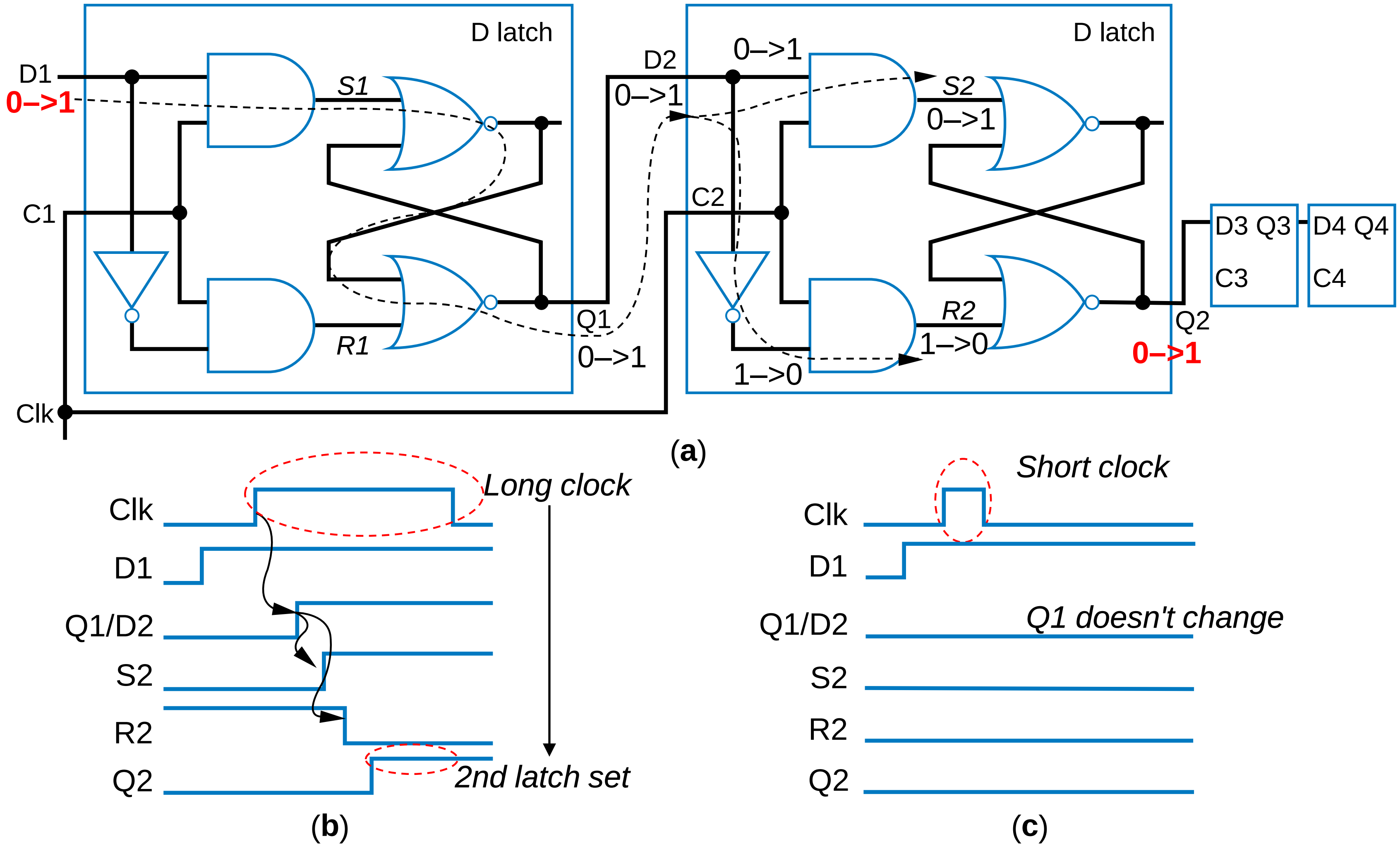
D latch symbol

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 how long C=1
    - Clk_A – signal may travel through multiple latches
    - Clk_B – signal may travel through fewer latches



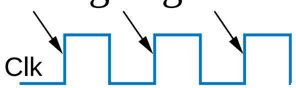
Problem with Level-Sensitive D Latch



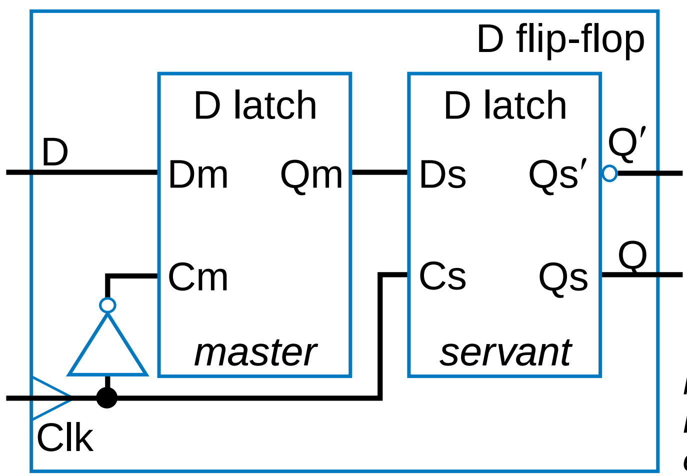
D Flip-Flop

Can we design bit storage that only stores a value on the *rising* edge of a clock signal?

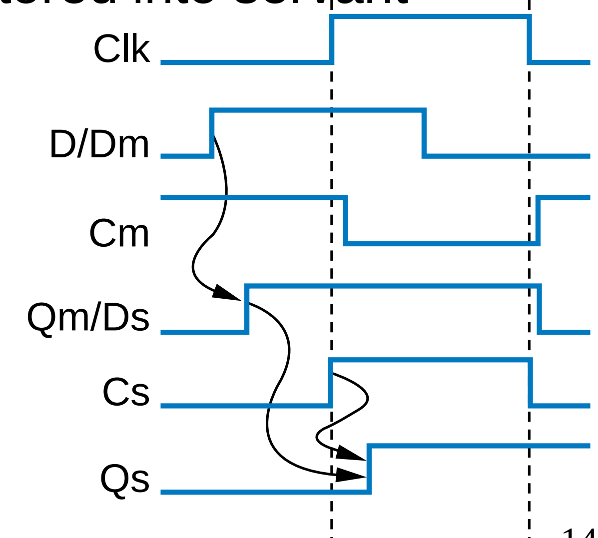
rising edges



- ***Flip-flop***: Bit storage that stores on clock ***edge***
- One design – master-servant
  - Clk = 0 – master enabled, loads D, appears at Qm. Servant disabled.
  - Clk = 1 – Master disabled, Qm stays same. Servant latch enabled, loads Qm, appears at Qs.
  - Thus, value at *D (and hence at Qm) when Clk changes from 0 to 1* gets stored into servant

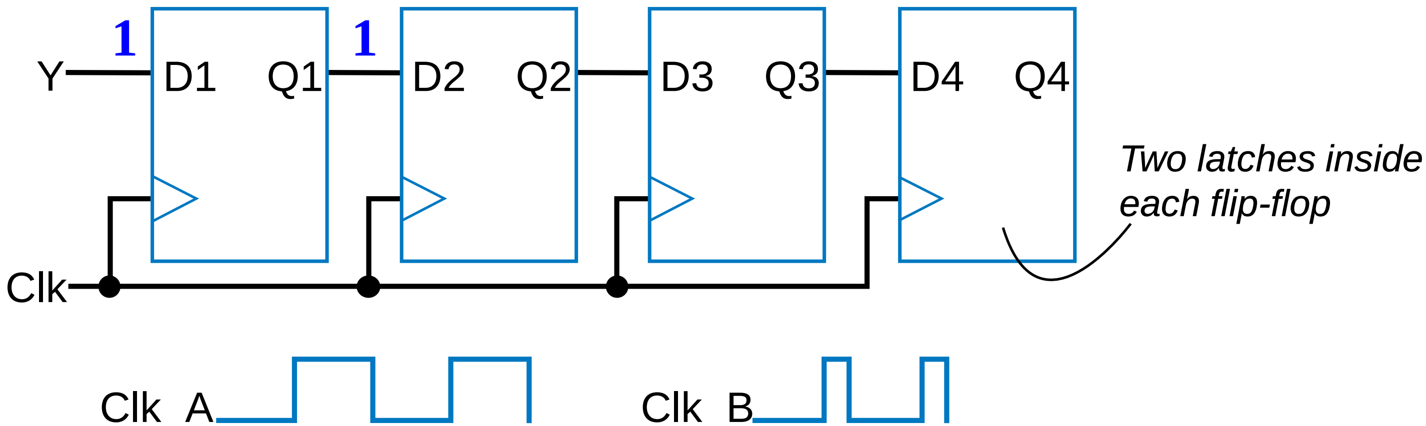


Note: Hundreds of different flip-flop designs exist

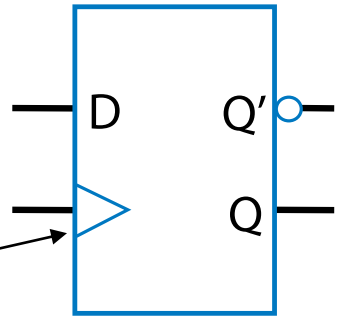


D Flip-Flop

- 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.

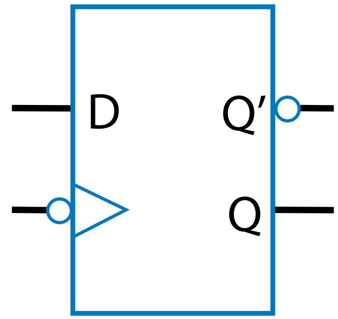


D Flip-Flop



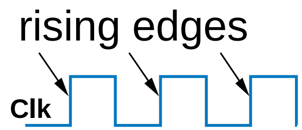
The triangle means edge-triggered clock input

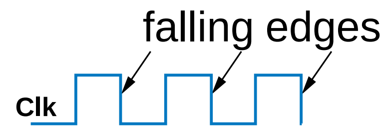
Symbol for rising-edge triggered D flip-flop



Symbol for falling-edge triggered D flip-flop

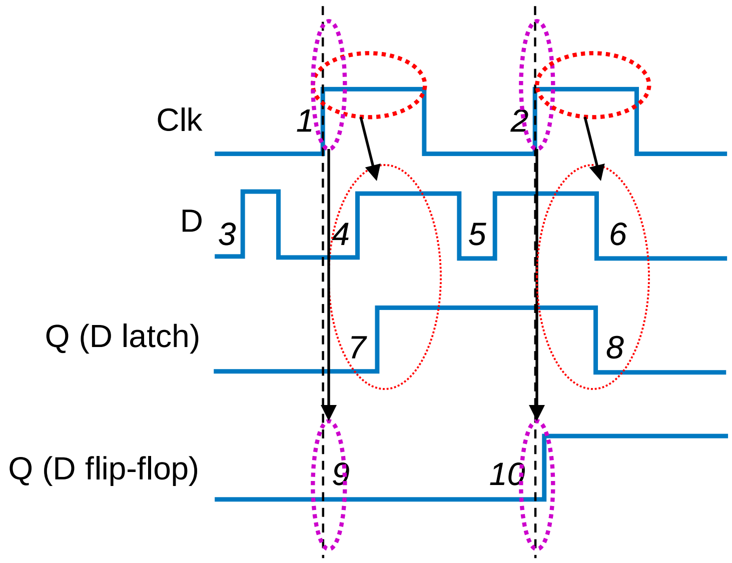
Internal design: Just invert servant clock rather than master





D 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
- Comparing behavior of latch and flip-flop:

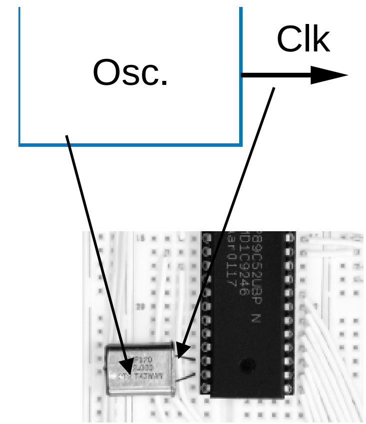


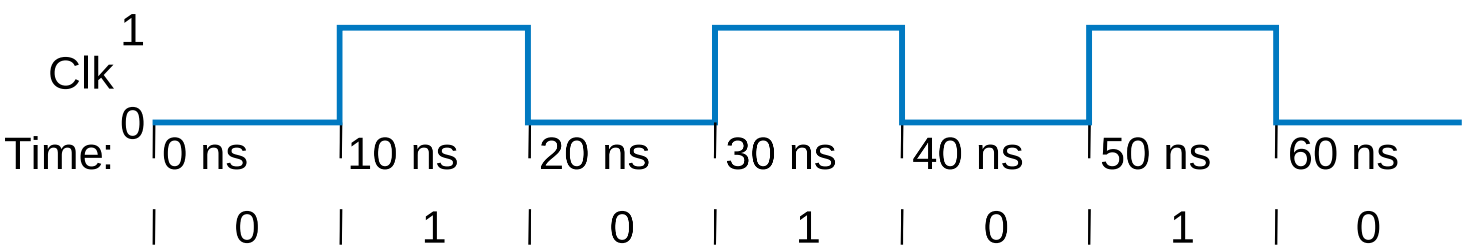
Latch follows D while Clk is 1

Flip-flop only loads D during Clk rising edge

Clock Signal

- Flip-flop Clk inputs typically connect to one clock signal
  - Coming from an oscillator component
  - Generates periodic pulsing signal
    - Below: "Period" = 20 ns, "Frequency" = 1/20 ns = 50 MHz
    - "Cycle" is duration of 1 period (20 ns); below shows 3.5 cycles



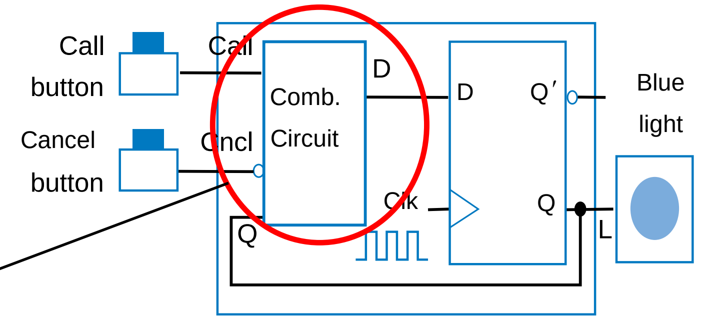


Freq.	Period
100 GHz	0.01 ns
10 GHz	0.1 ns
1 GHz	**1 ns**
100 MHz	10 ns
10 MHz	100 ns

Period/Freq shortcut: Remember 1 ns → 1 GHz

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



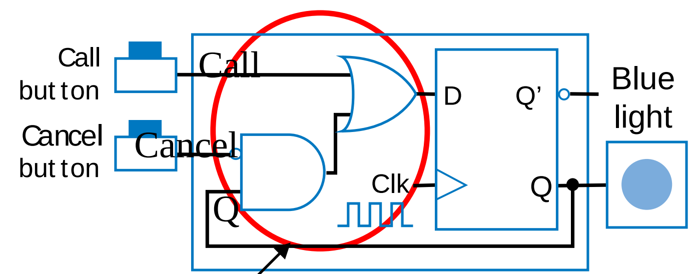
Call	Cancel	Q	D
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

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

Cancel -- make D=0

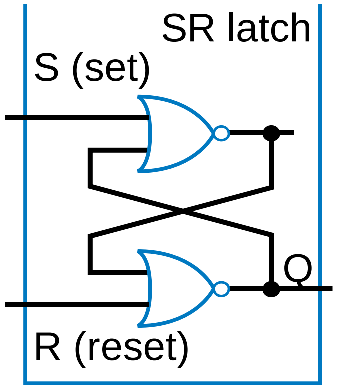
Call -- make D=1

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

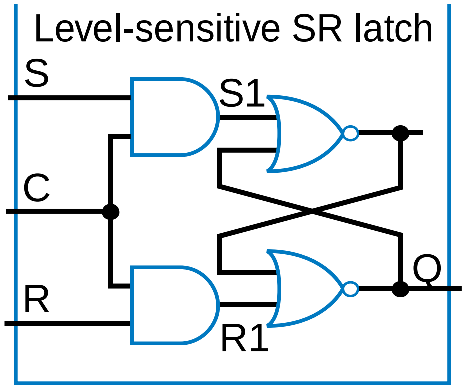


Circuit derived from truth table, using Chapter 2 combinational logic design process

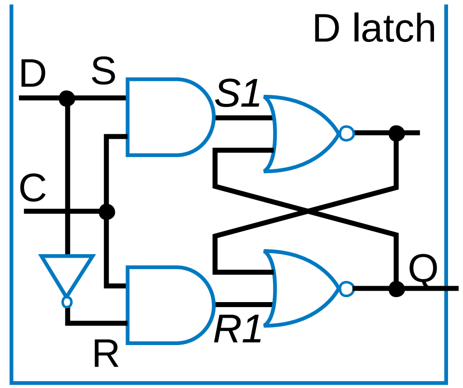
Bit Storage Summary



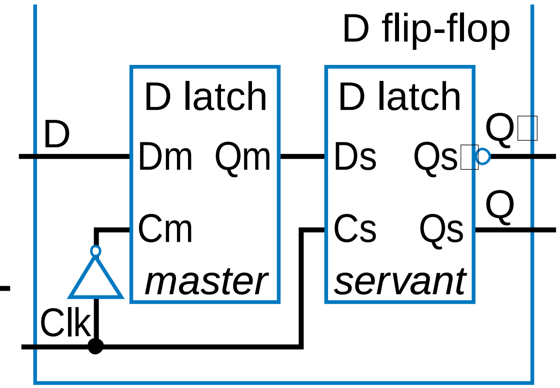
Feature: S=1 sets Q to 1, R=1 resets Q to 0. *Problem:* SR=11 yields undefined Q, other glitches may set/reset inadvertently.



Feature: S and R only have effect when C=1. An external circuit can prevent SR=11 when C=1. *Problem:* avoiding SR=11 can be a burden.



Feature: SR can’t be 11. *Problem:* C=1 for too long will propagate new values through too many latches; for too short may not result in the bit being stored.

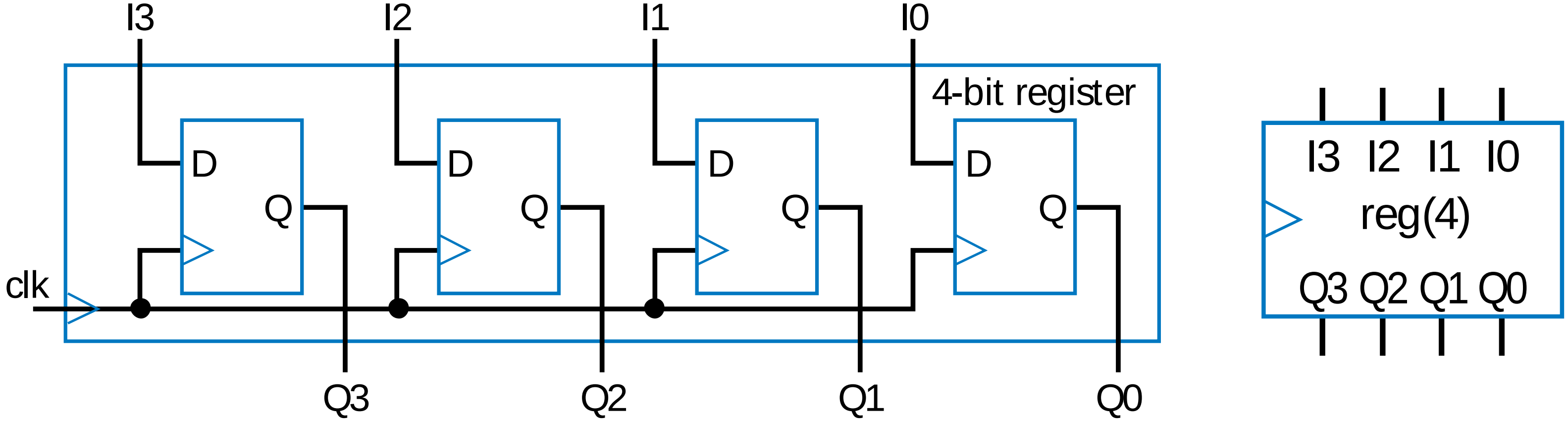


Feature: Only loads D value present at rising clock edge, so values can't propagate to other flip-flops during same clock cycle. *Tradeoff:* uses more gates internally, and requires more external gates than SR—but transistors today are more plentiful and cheaper.

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

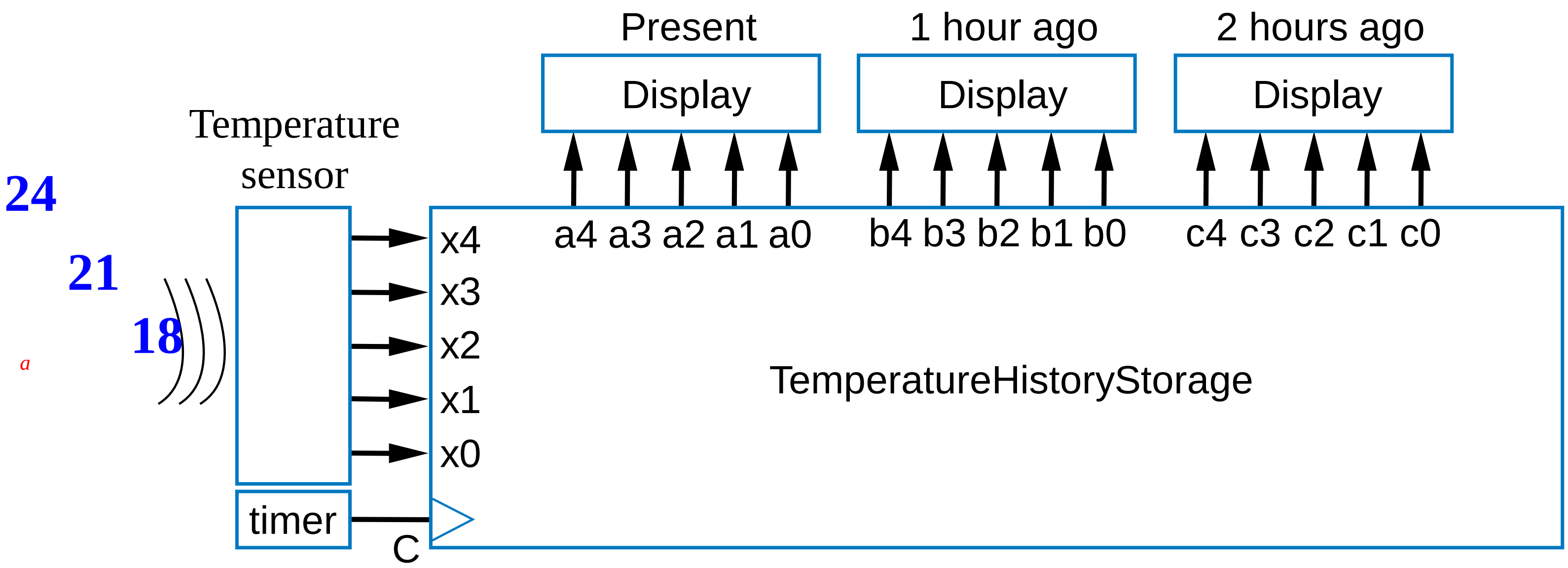
Basic Register

- 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



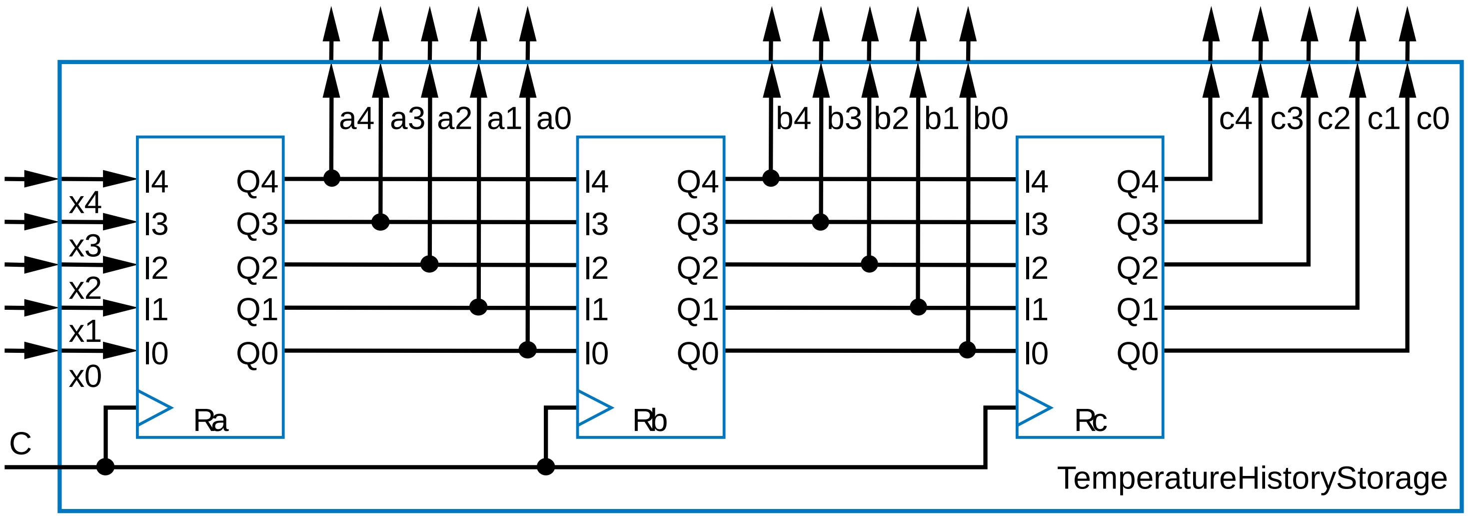
Example Using Registers: Temperature Display

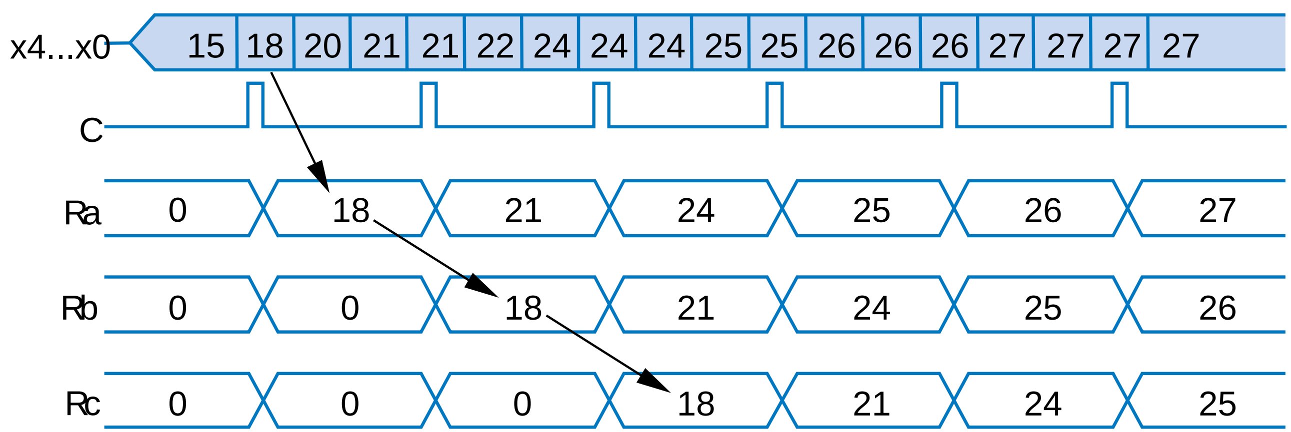
- Use three 5-bit registers

24

21

18

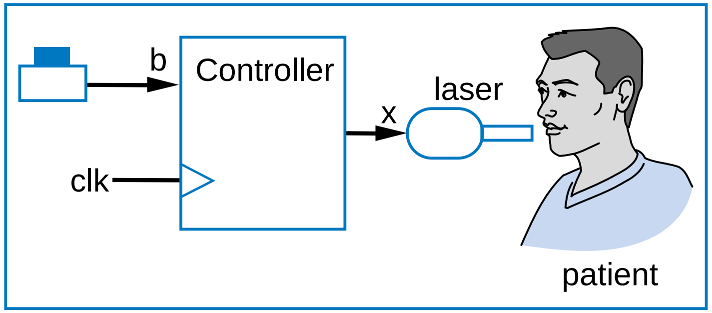


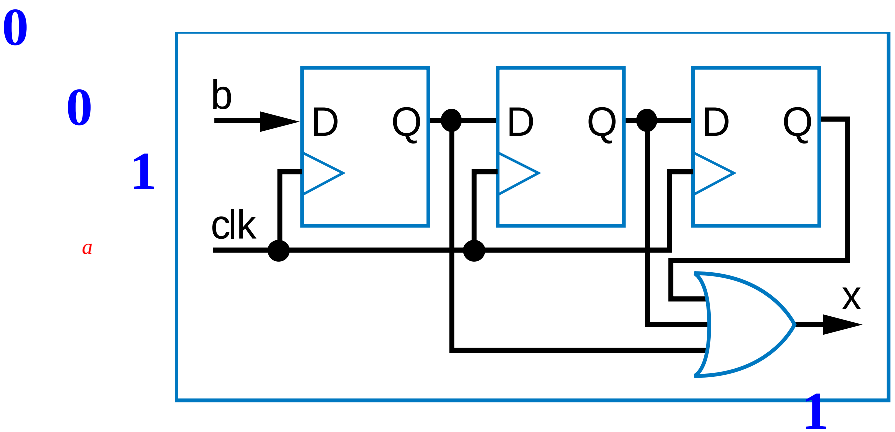


Note that registers only loaded on rising clock edges

Finite-State Machines (FSMs) and Controllers

- Want sequential circuit with particular behavior over time
- Example: Laser timer
  - Pushing button causes x=1 for exactly 3 clock cycles
    - Precisely-timed laser pulse
  - 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





Bad job – what if button pressed a second time during those 3 cycles?

Need a Better Way to Design Sequential Circuits

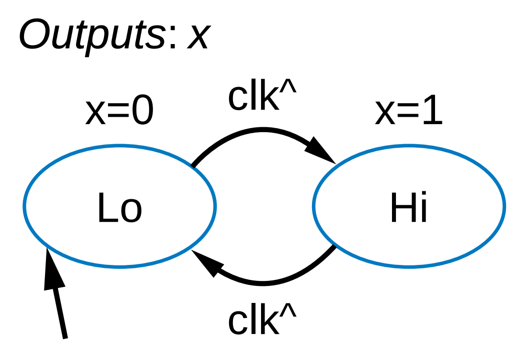
- Also bad because of ad hoc design process
  - How create other sequential circuits?
- Need
  - A way to ***capture*** desired sequential behavior
  - A way to ***convert*** such behavior to a sequential circuit

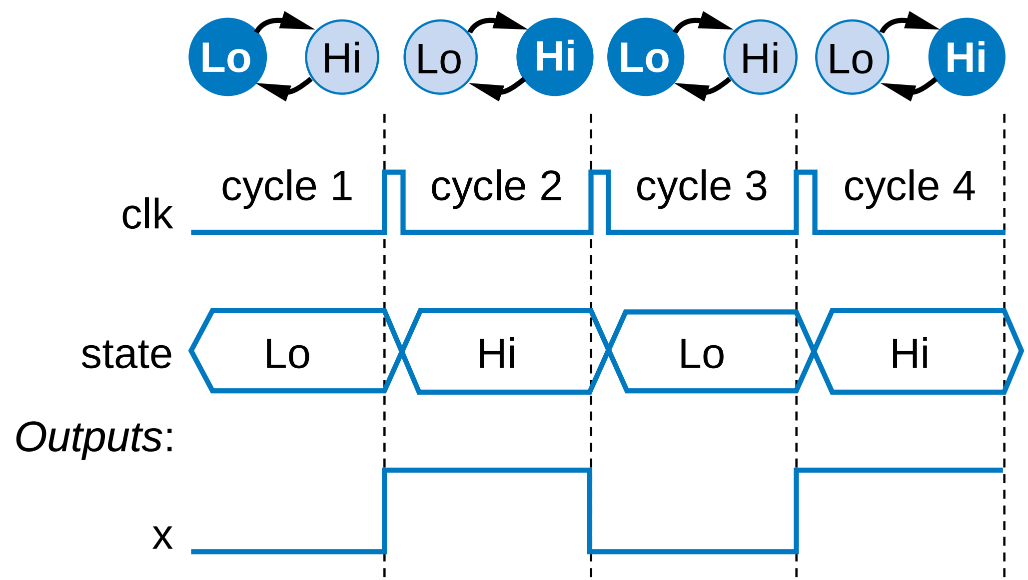
Like we had for designing combinational circuits

Step		Description
Step 1: Capture behavior	**Capture** the function	Create a truth table or equations, ***whichever is most natural for the given problem***, to describe the desired behavior of each output of the combinational logic.
Step 2: Convert to circuit	2A: **Create** equations	This substep is only necessary if you captured the function using a truth table of equations. Create an equation for each output by ORing all the minterms for that output. Simplify the equations if desired.
	2B: **Implement** as a gate-based circuit	For each output, create a circuit corresponding to the output's equation. (Sharing gates among multiple outputs is OK optionally.)

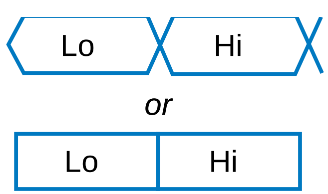
Capturing Sequential Circuit Behavior as FSM

- Finite-State Machine (FSM)
  - Describes desired behavior of sequential circuit
    - Akin to Boolean equations for combinational behavior
- List states, and transitions among states
  - Example: Toggle x every clock cycle
  - Two states: “Lo” (x=0), and “Hi” (x=1)
  - Transition from Lo to Hi, or Hi to Lo, on rising clock edge (clk^)
  - Arrow points to initial state (when circuit first starts)



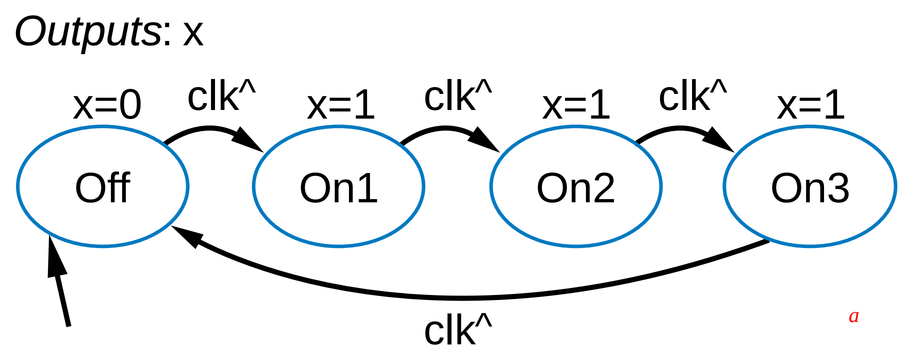


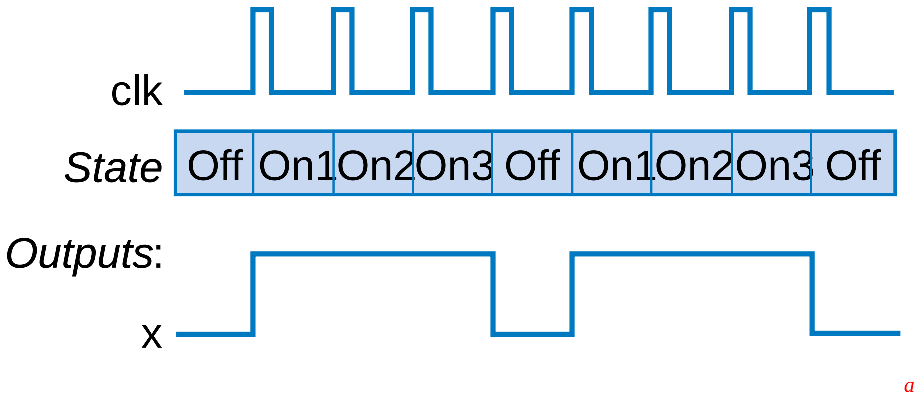
Depicting multi-bit or other info in a timing diagram



FSM Example: Three Cycles High System

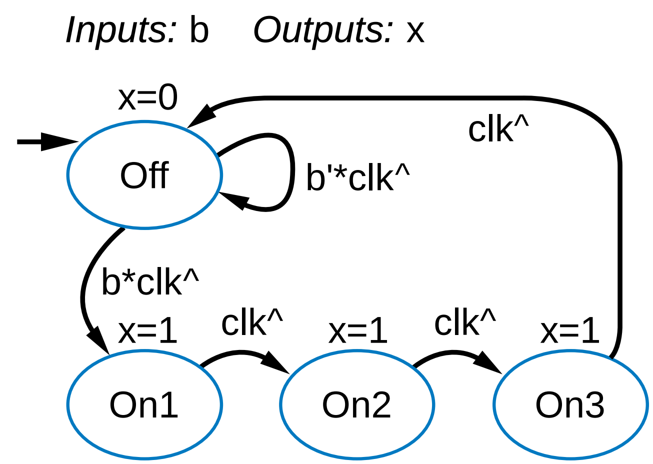
- Want 0, 1, 1, 1, 0, 1, 1, 1, ...
  - For one clock cycle each
- Capture as FSM
  - Four states: 0, first 1, second 1, third 1
  - Transition on rising clock edge to next state

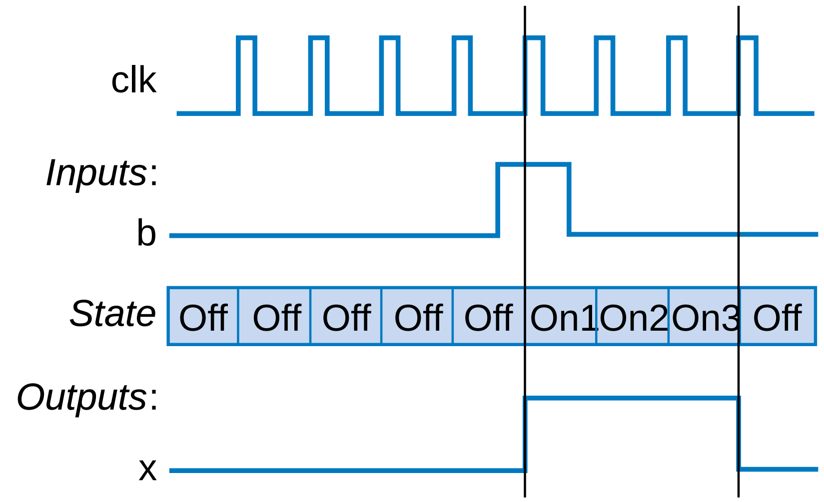




Three-Cycles High System with Button Input

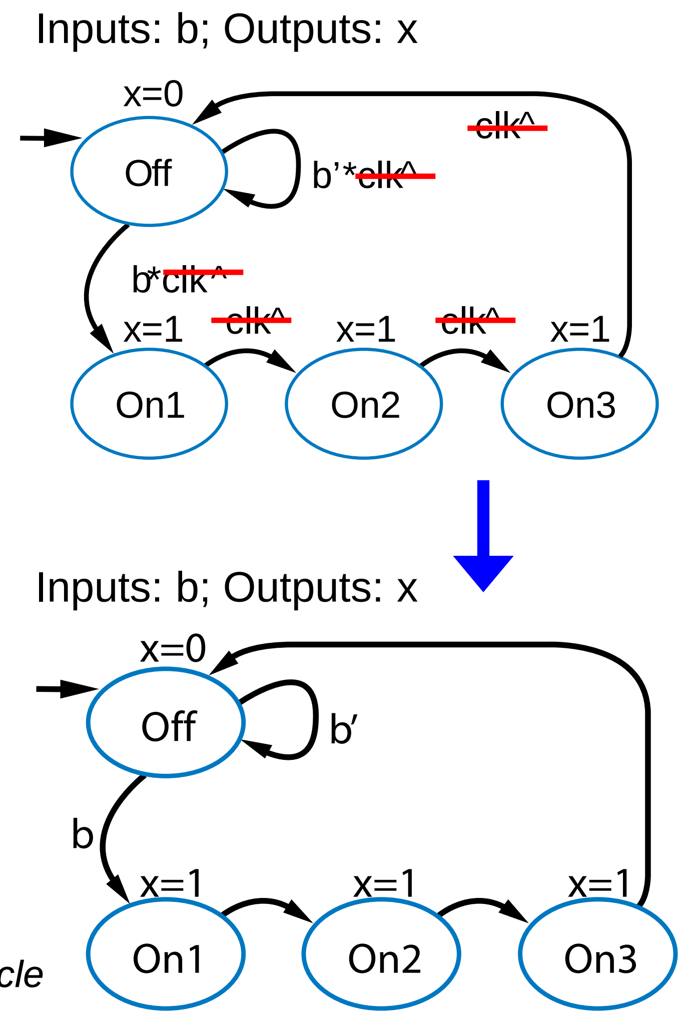
- Four states
- Wait in "Off" while b is 0 (b'*clk^)
- When b is 1 (b*clk^), transition to On1
  - Sets x=1
  - Next two clock edges, transition to On2, then On3
- So x=1 for three cycles after button pressed





FSM Simplification: Rising Clock Edges Implicit

- Every edge ANDed with rising 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

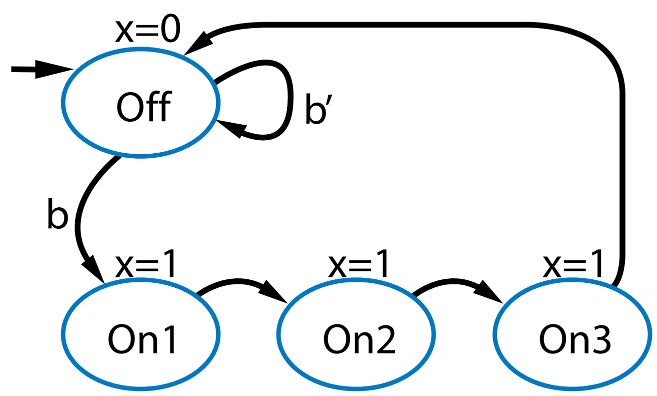


Note: Transition with no associated condition thus transistions to next state on next clock cycle

FSM Definition

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

Inputs: b; Outputs: x

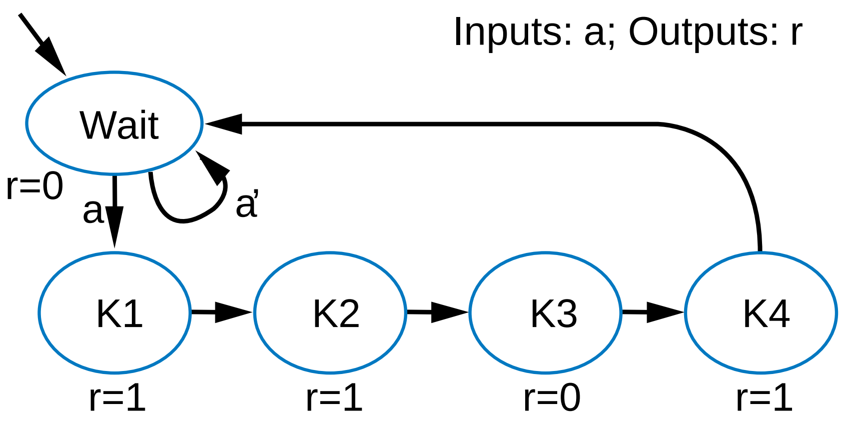


We often draw FSM graphically, known as ***state diagram***

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

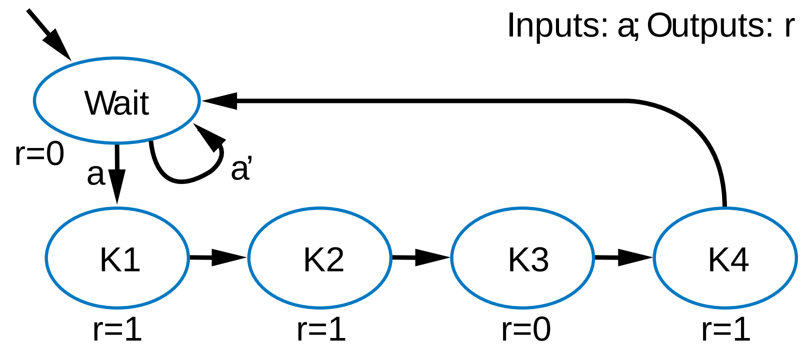
FSM Example: Secure Car Key

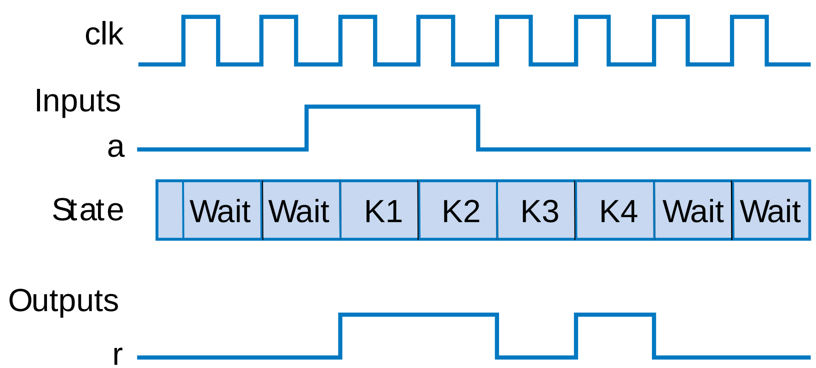
- Many new car keys include tiny computer chip
  - When key turned, car's computer (under engine hood) requests identifier from key
  - Key transmits identifier
    - Else, computer doesn't start car
- FSM
  - Wait until computer requests ID (a=1)
  - Transmit ID (in this case, 1 1 0 1)



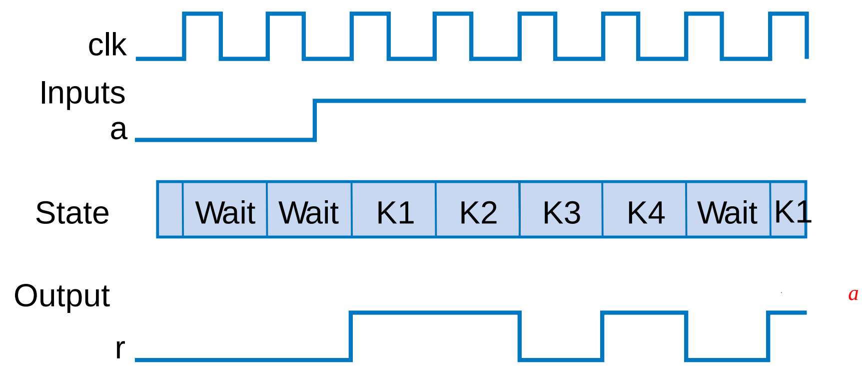
FSM Example: Secure Car Key (cont.)

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



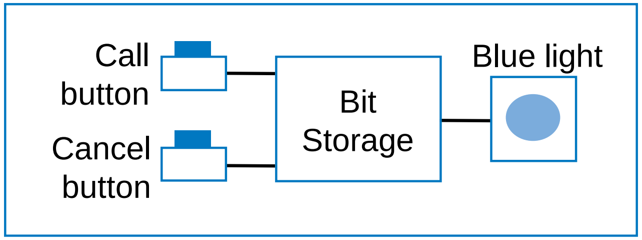


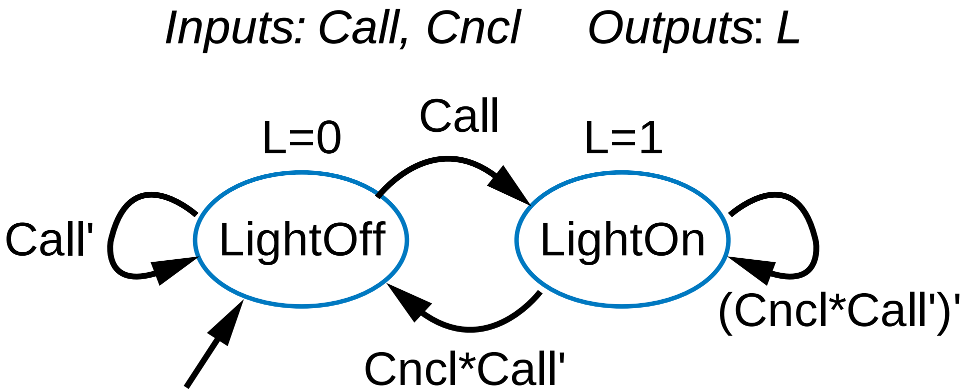
Q: Determine states and r value for given input waveform:



Ex: Earlier Flight-Attendant Call Button

- Previously built using SR latch, then D flip-flop
- Capture desired bit storage behavior using FSM instead
  - Clear and precise description of desired behavior
  - We'll later convert to a circuit



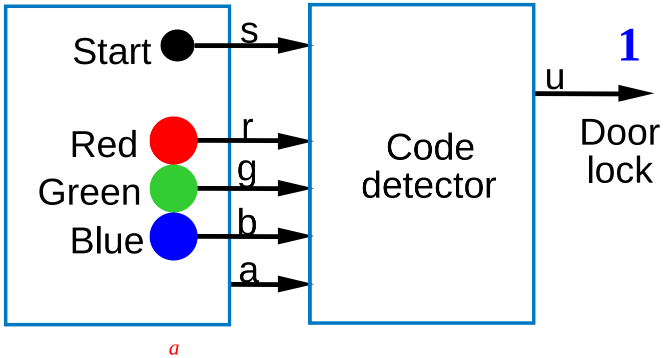


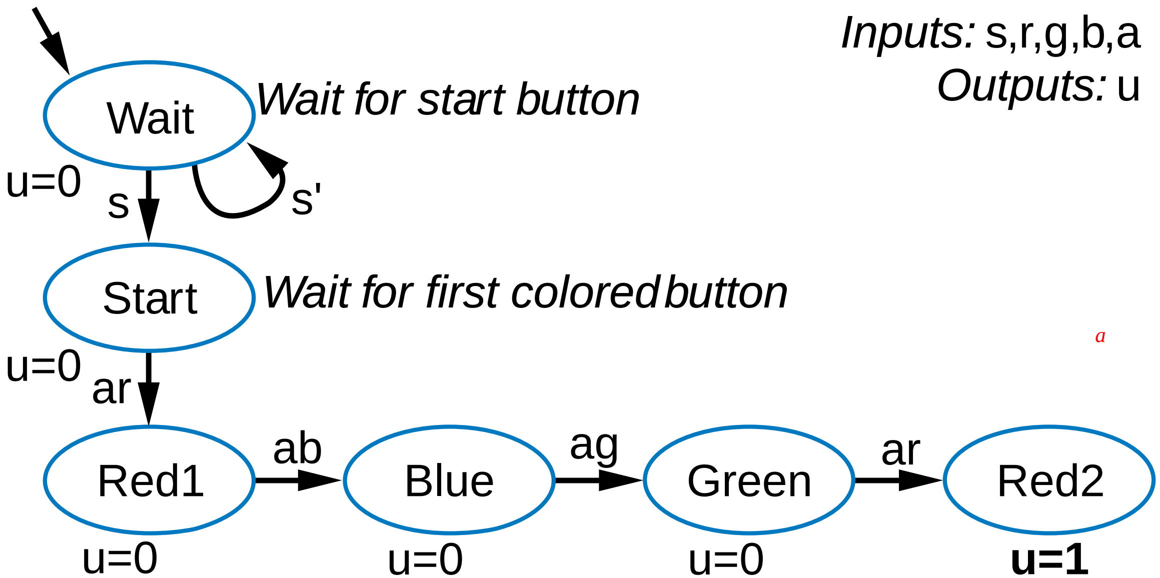
How To Capture Desired Behavior as FSM

- *List states*
  - Give meaningful names, show initial state
  - Optionally add some transitions if they help
- *Create transitions*
  - For each state, define all possible transitions leaving that state.
- *Refine the FSM*
  - Execute the FSM mentally and make any needed improvements.

FSM Capture Example: Code Detector

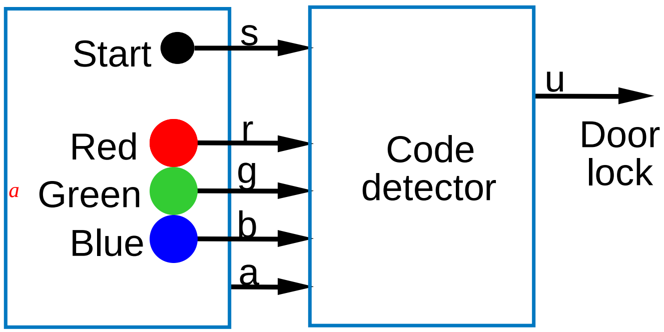
- Unlock door (u=1) only when buttons pressed in sequence:
  - start, then red, blue, green, red
- Input from each button: *s, r, g, b*
  - Also, output *a* indicates that some colored button pressed
- Capture as FSM
  - ***List states***
    - *Some transitions included*



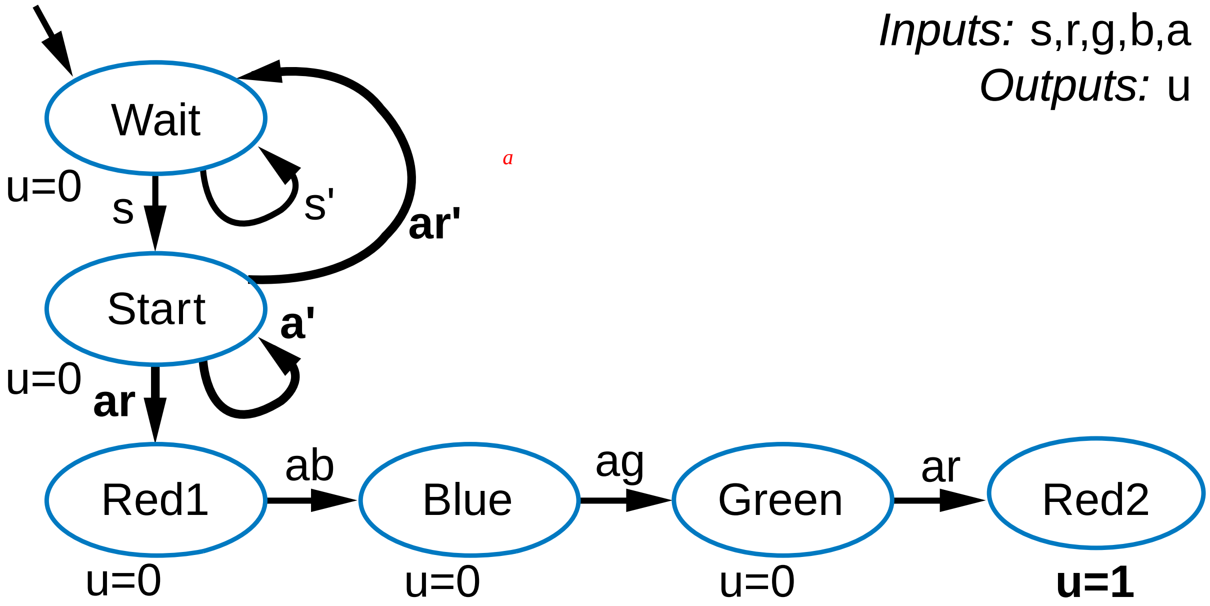


FSM Capture Example: Code Detector

- Capture as FSM
  - *List states*
  - ***Create transitions***

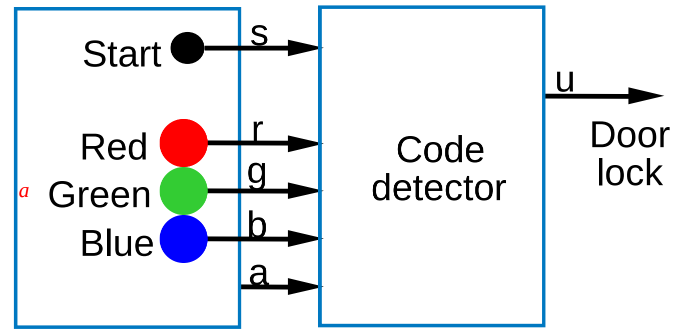


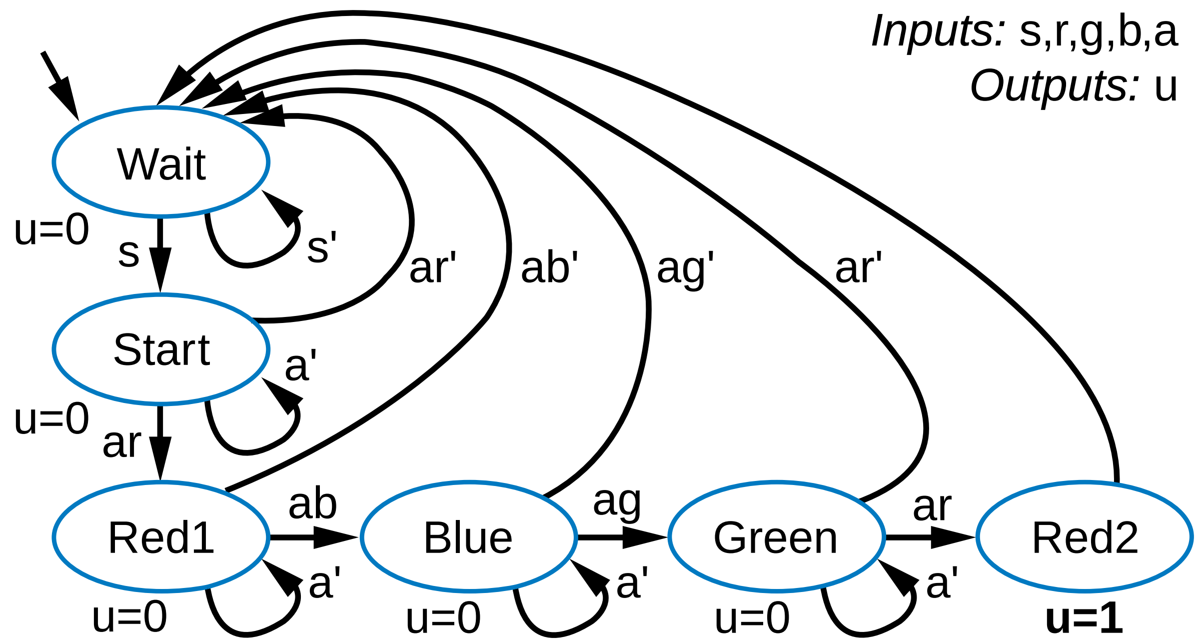
Inputs: s,r,g,b,a
Outputs: u



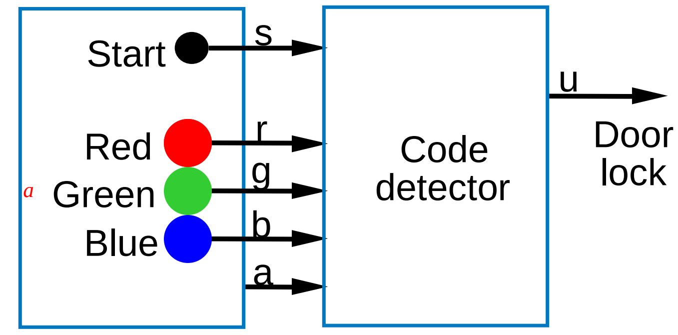
FSM Capture Example: Code Detector

- Capture as FSM
  - *List states*
  - *Create transitions*
    - Repeat for remaining states
  - Refine FSM
    - Mentally execute
    - Works for normal sequence
    - Check unusual cases
    - All colored buttons pressed
      - Door opens!
      - Change conditions: other buttons NOT pressed also



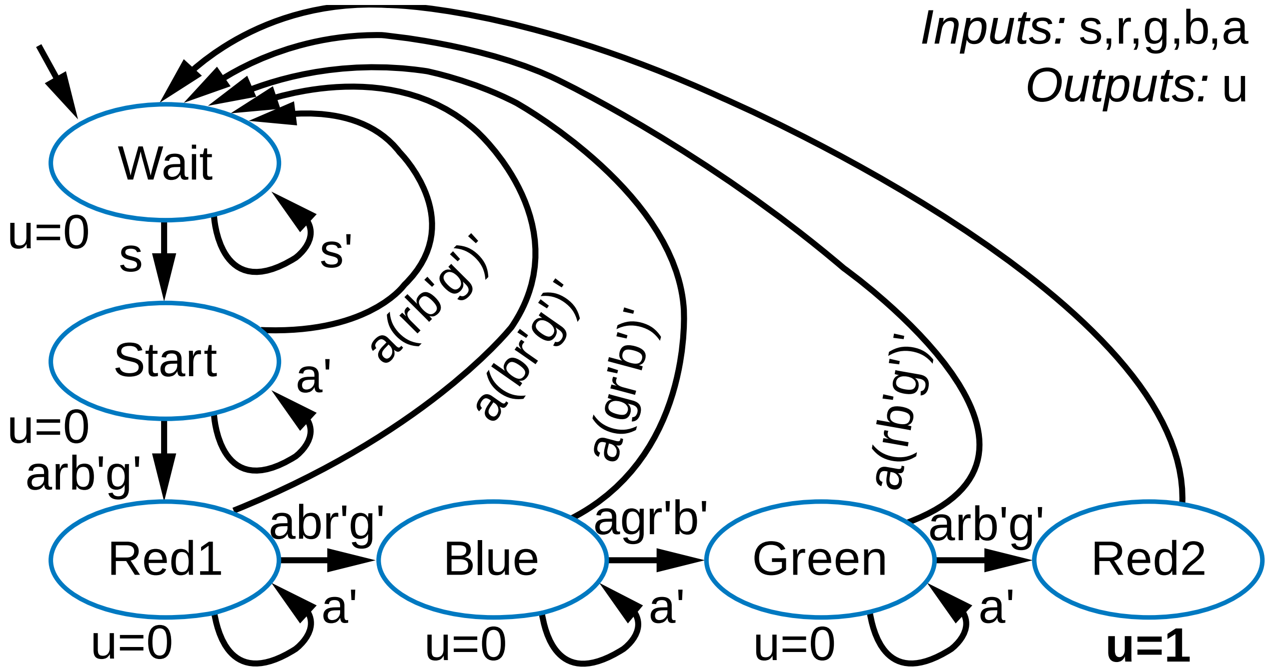


FSM Capture Example: Code Detector



Inputs: s,r,g,b,a

Outputs: u

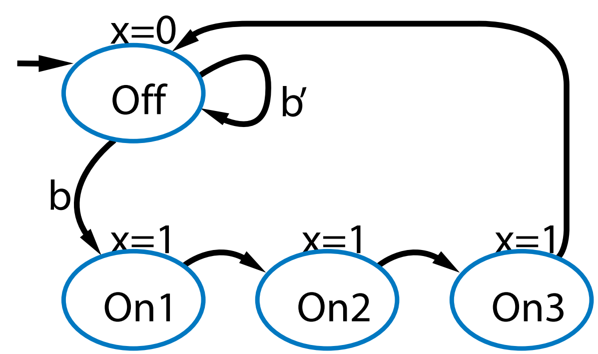


Controller Design

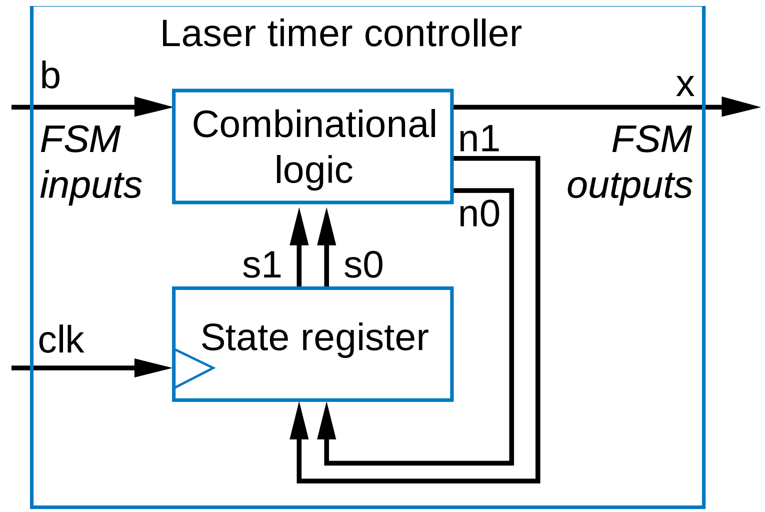
- Converting FSM to sequential circuit
  - Circuit called *controller*
  - Standard controller architecture
    - State register stores encoding of current state
      - e.g., Off:00, On1:01, On2:10, On3:11
    - Combinational logic computes outputs and next state from inputs and current state
    - Rising clock edge takes controller to next state

Laser timer FSM

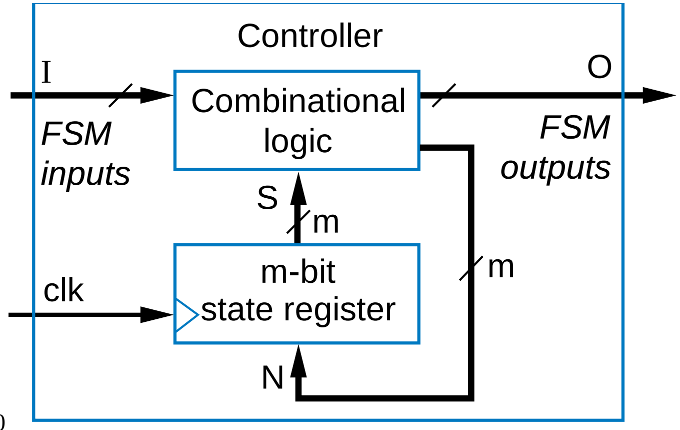
Inputs: b; Outputs: x



Controller for laser timer FSM



General form

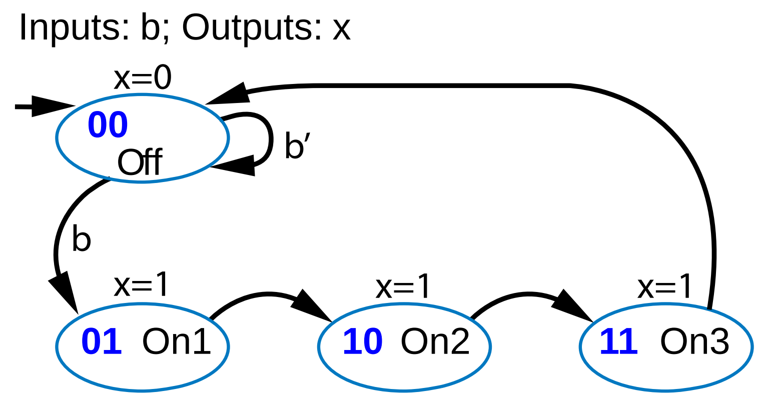


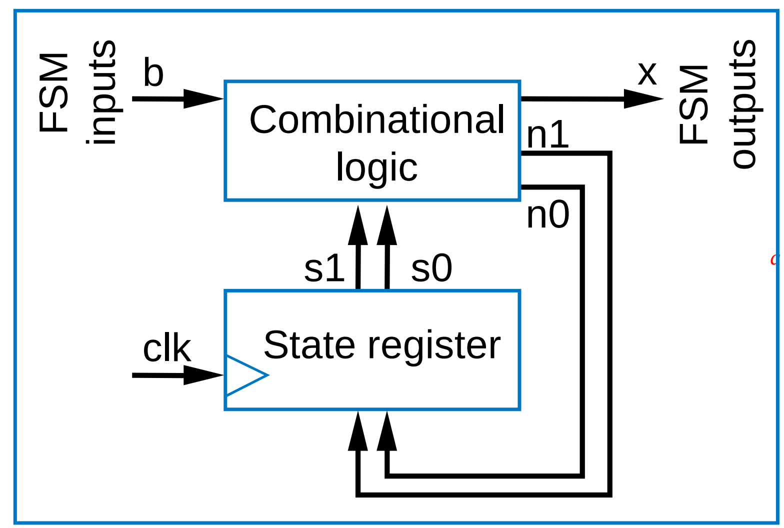
Controller Design Process

Step		Description
Step 1: Capture behavior	**Capture** the FSM	Create an FSM that describes the desired behavior of the controller.
Step 2: Convert to circuit	2A: **Set up** architecture	Use state register of appropriate width and combinational logic. The logic's inputs are the state register bits and the FSM inputs; outputs are next state bits and the FSM outputs.
	2B: **Encode** the states	Assign unique binary number (encoding) to each state. Usually use fewest bits, assign encoding to each state by counting up in binary.
	2C: **Fill in** the truth table	Translate FSM to truth table for combinational logic such that the logic will generate the outputs and next state signals for the given FSM. Ordering the inputs with state bits first makes the correspondence between the table and the FSM clear.
	2D: **Implement** combinational logic	Implement the combinational logic using any method.

Controller Design: Laser Timer Example

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

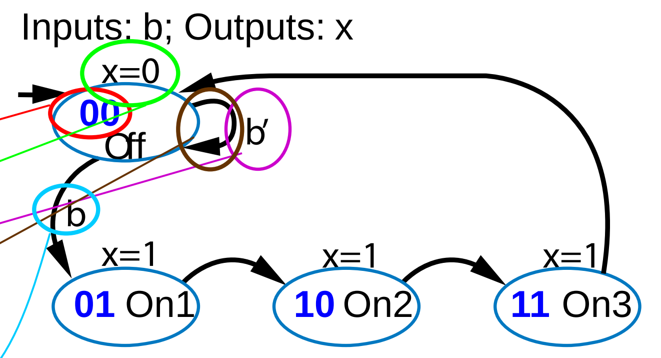


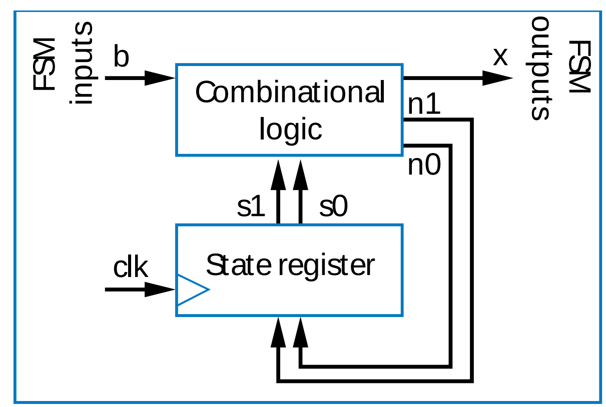


Controller Design: Laser Timer Example (cont)

- Step 2C: Fill in truth table

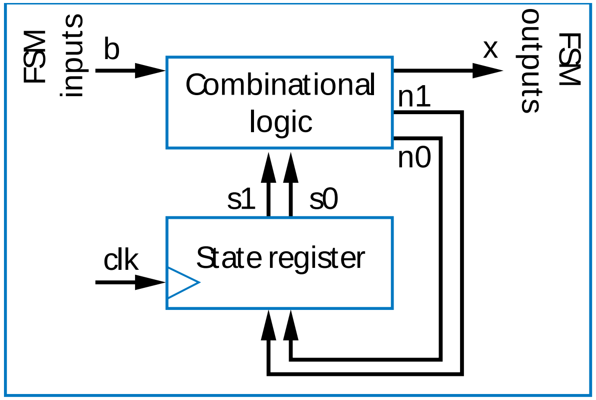
	Inputs			Outputs		
	s1	s0	b	x	n1	n0
Off	0	0	0	0	0	0
	0	0	1	0	0	1
On1	0	1	0	1	1	0
	0	1	1	1	1	0
On2	1	0	0	1	1	1
	1	0	1	1	1	1
On3	1	1	0	1	0	0
	1	1	1	1	0	0





Controller Design: Laser Timer Example (cont)

- Step 2D: 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 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

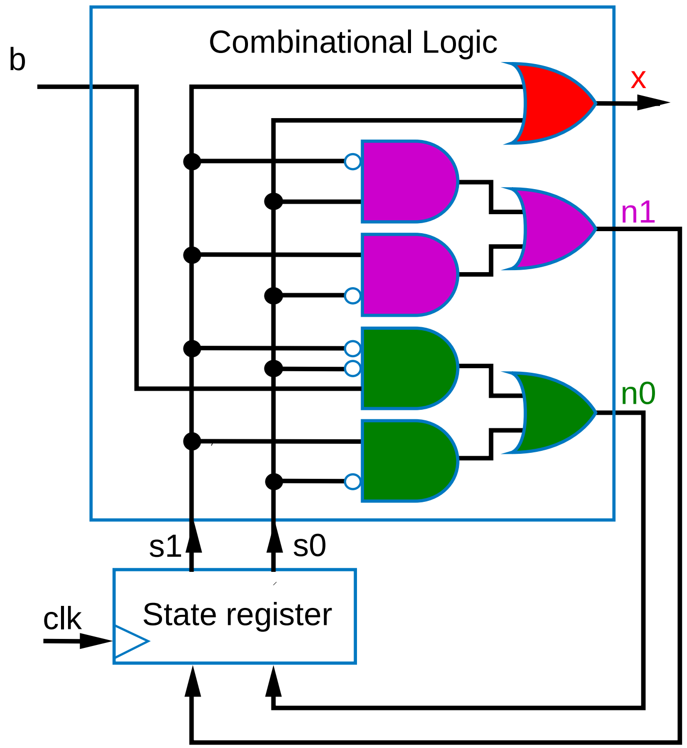
n0 = s1’s0’b + s1s0’

a

Controller Design: Laser Timer Example (cont)

- Step 2D: 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

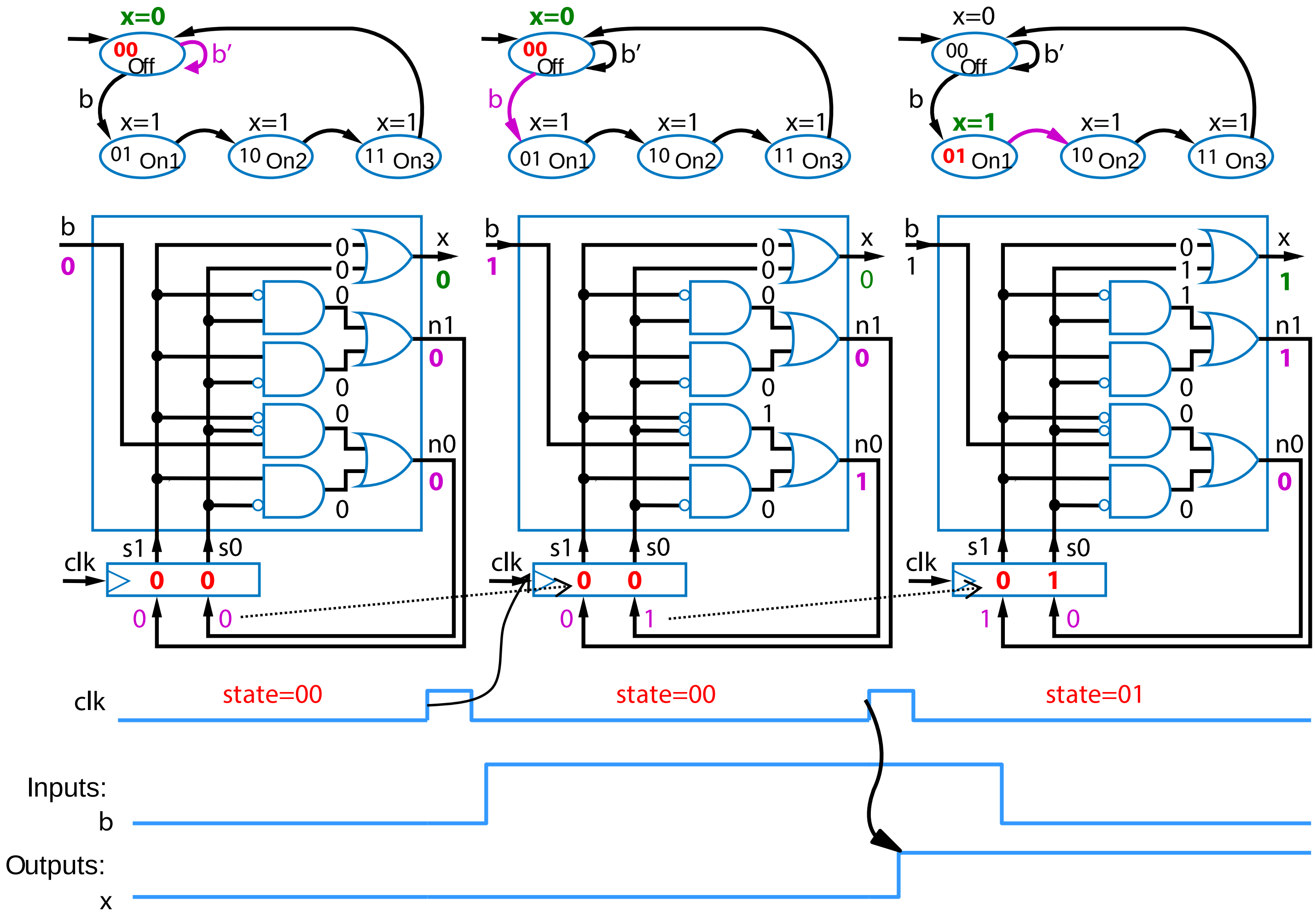


x = s1 + s0

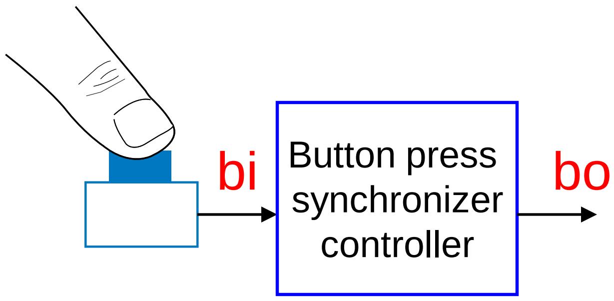
n1 = s1’s0 + s1s0’

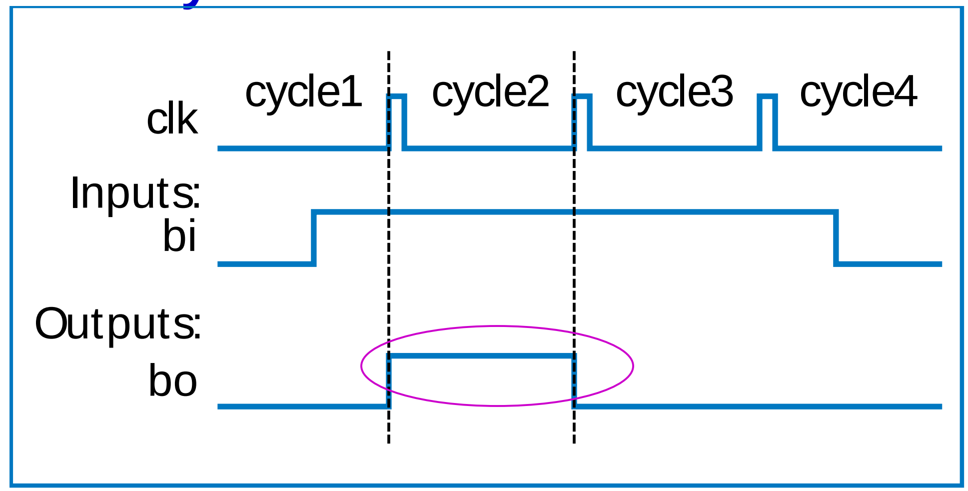
n0 = s1’s0’b + s1s0’

Understanding the Controller's Behavior



Controller Example:
Button Press Synchronizer

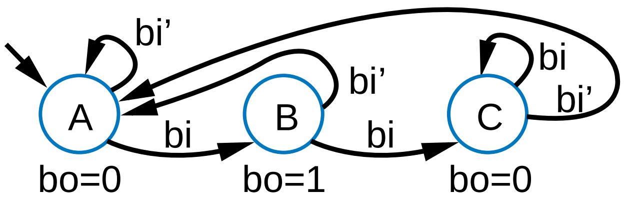




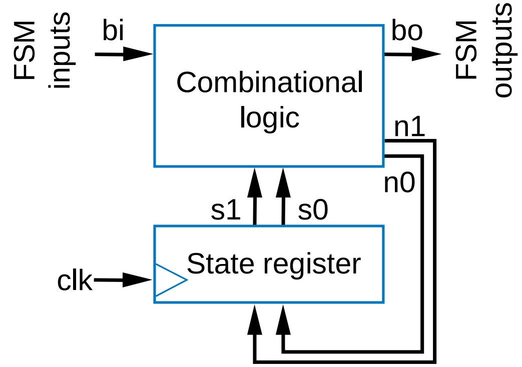
- Want simple sequential circuit that converts button press to single cycle duration, regardless of length of time that button was actually pressed
  - We assumed such an ideal button press signal in earlier example, like the button in the laser timer controller

Controller Example:
Button Press Synchronizer (cont)

FSM inputs: bi; FSM outputs: bo

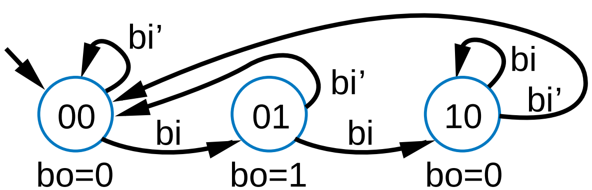


Step 1: Capture FSM



Step 2A: Set up architecture

FSM inputs: bi; FSM outputs: bo



Step 2B: 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

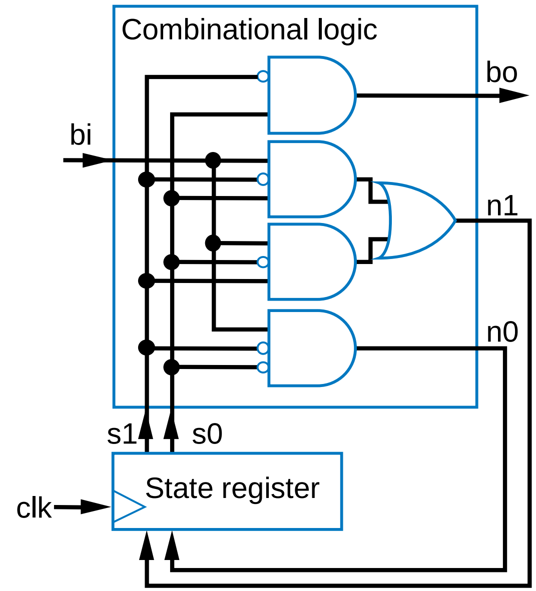
Step 2C: Fill in truth table

n1 = s1’s0bi + s1s0bi

n0 = s1’s0’bi

bo = s1’s0bi’ + s1’s0bi = s1s0

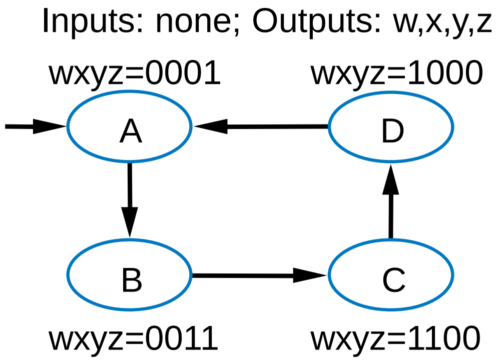
a



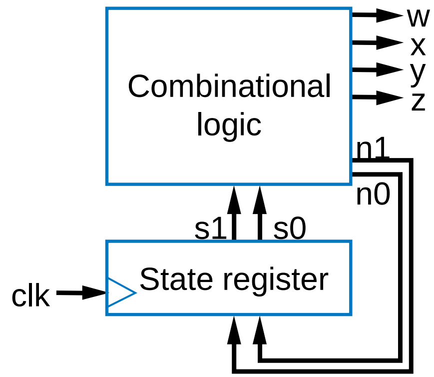
Step 2D: Implement combinational logic

Controller Example: Sequence Generator

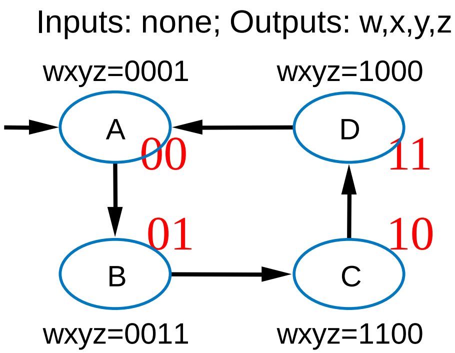
- Want generate sequence 0001, 0011, 1100, 1000, (repeat)
  - Each value for one clock cycle
  - Common, e.g., to create pattern in 4 lights, or control magnets of a “stepper motor”



Step 1: Create FSM



Step 2A: Set up architecture

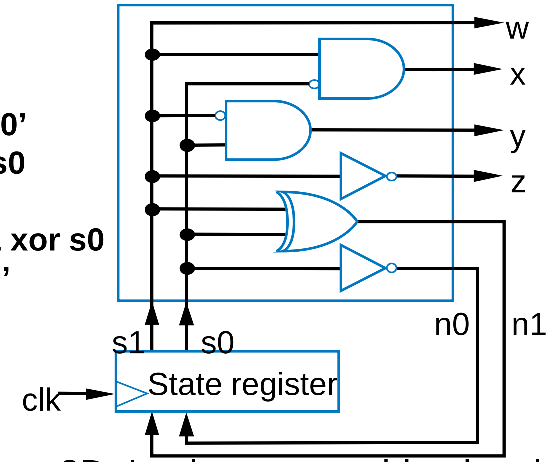


Step 2B: Encode states

	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 2C: Fill in truth table

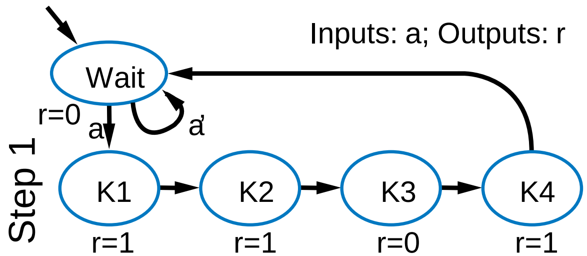
w = s1
x = s1s0’
y = s1’s0
z = s1’
n1 = s1 xor s0
n0 = s0’

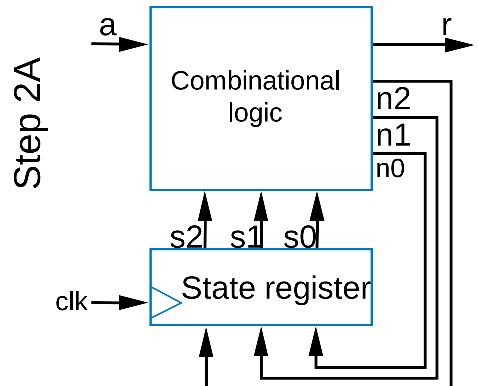


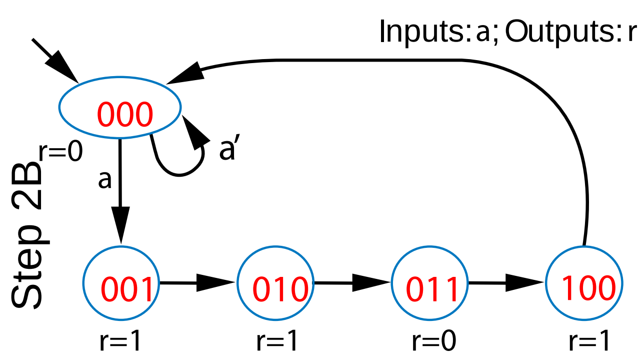
Step 2D: Implement combinational logic

Controller Example: Secure Car Key

- (from earlier example)







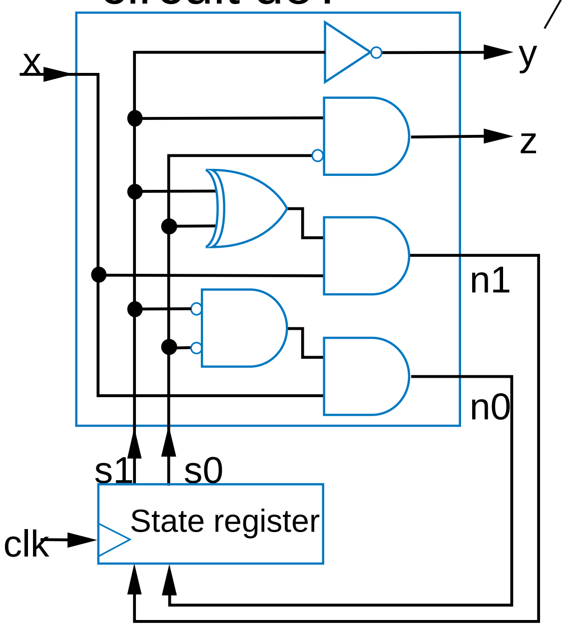
	Inputs				Outputs			
	s2	s1	s0	a	r	n2	n1	n0
Wait	0	0	0	0	0	0	0	0
	0	0	0	1	0	0	0	1
K1	0	0	1	0	1	0	1	0
	0	0	1	1	1	0	1	0
K2	0	1	0	0	1	0	1	1
	0	1	0	1	1	0	1	1
K3	0	1	1	0	0	1	0	0
	0	1	1	1	0	1	0	0
K4	1	0	0	0	1	0	0	0
	1	0	0	1	1	0	0	0
Unused	1	0	1	0	0	0	0	0
	1	0	1	1	0	0	0	0
	1	1	0	0	0	0	0	0
	1	1	0	1	0	0	0	0
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

Step 2C

We'll omit Step 2D

Converting a Circuit to FSM (Reverse Engineering)

What does this circuit do?



Work backwards

2D: Circuit to eqns

y=s1'

z = s1s0'

n1=(s1 xor s0)x

n0=(s1'*s0')x

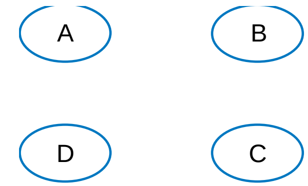
2C: Truth table

	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

2B: (Un)encode states
Pick any state names you want

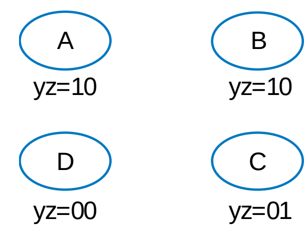
2A: Set up arch – already done

Step 1: FSM (get from table)



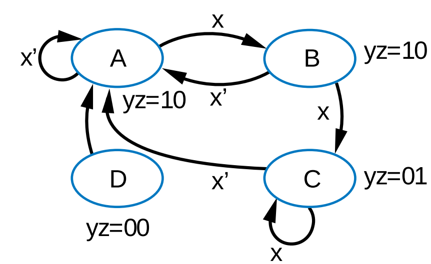
states

Outputs: y, z



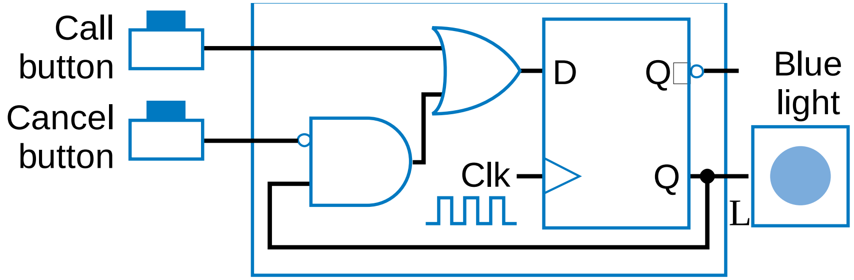
states with outputs

Inputs: x; Outputs: y, z



states with outputs and transitions

Reverse Engin. the D-flip-flop Flight Atten. Call Button



2C: Truth table

	Inputs			Outputs	
	Q	Call	Cncl	D	L
Light Off	0	0	0	0	0
	0	0	1	0	0
	0	1	0	1	0
	0	1	1	1	0
Light On	1	0	0	1	1
	1	0	1	0	1
	1	1	0	1	1
	1	1	1	1	1

2B: (Un)encode states

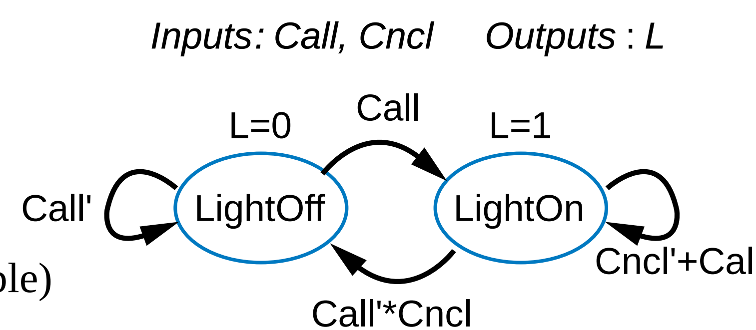
2D: Circuit to eqns

L = Q

D = Cncl'Q + Call (next state)

2A: Set up arch (nothing to do)

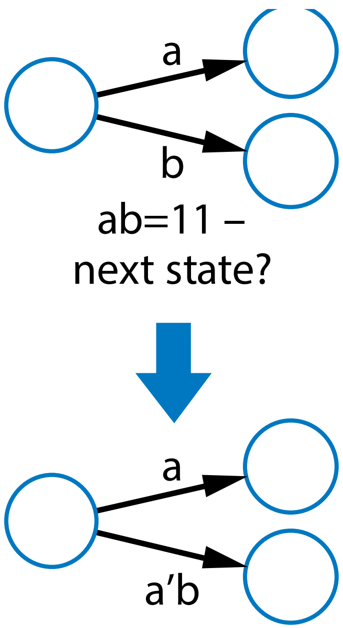
Don't let the way the circuit is drawn confuse you; the combinational logic is everything outside the register



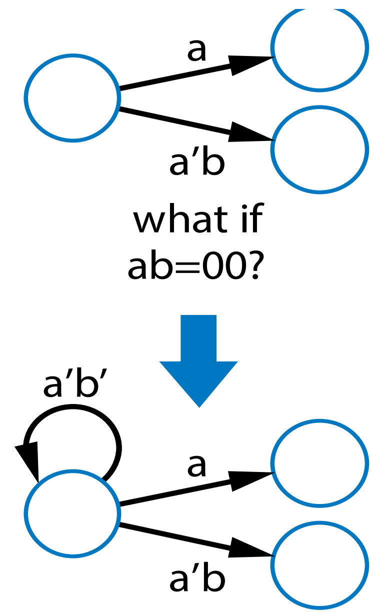
Step 1: FSM (get from table)

Common Mistakes when Capturing FSMs

- Non-exclusive transitions

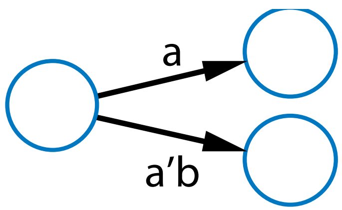


- Incomplete transitions



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



Answer:

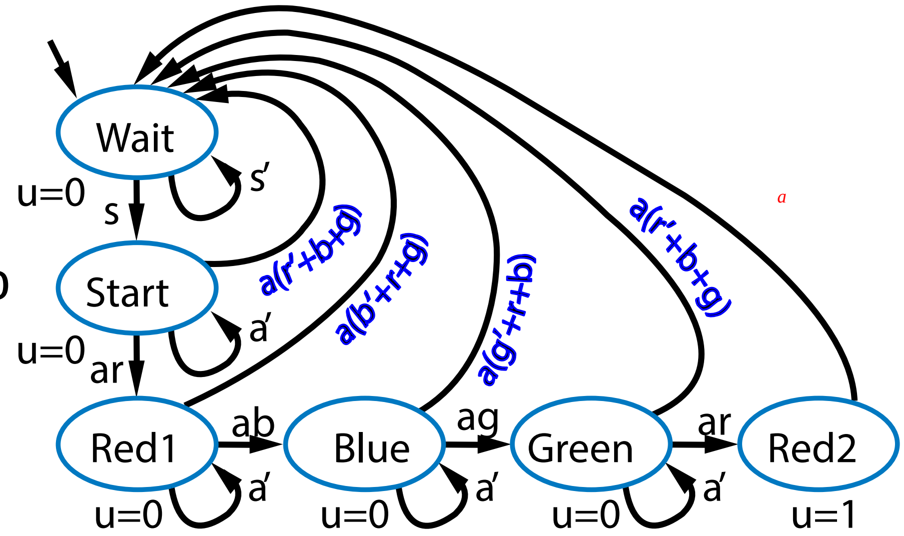
a * a'b
= (a * a') * b
= 0 * b
= 0
OK!

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)

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)

Verifying transition properties

- 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?

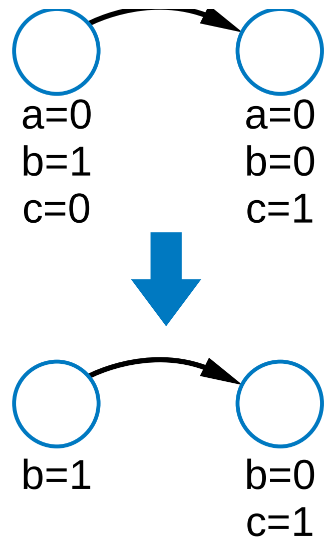
Q: How to solve?

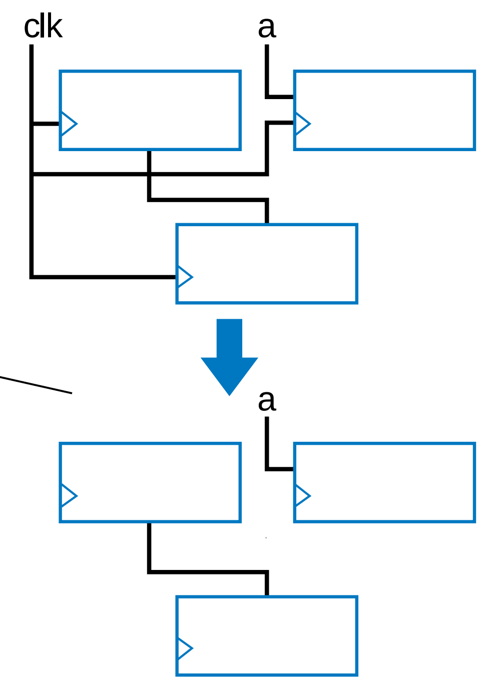
A: ar should be arb'g' (likewise for ab, ag, ar)

Note: As evidence the pitfall is common, we admit the mistake was not initially intentional. A reviewer of an earlier edition 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



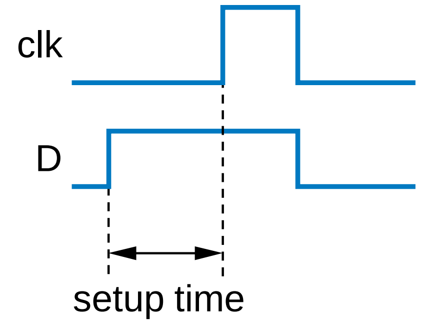


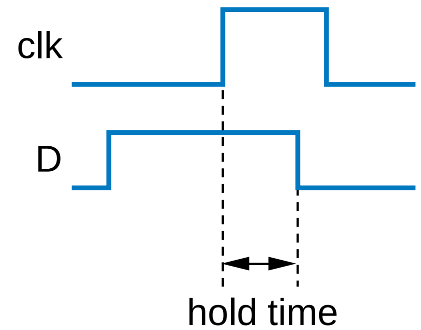
Mathematical Formalisms

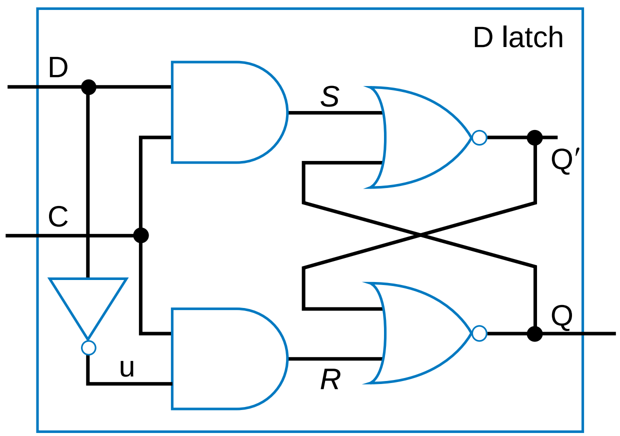
- Two formalisms to capture behavior thus far
  - Boolean equations for combinational circuit design
  - FSMs for sequential circuit design
- Not *necessary*
  - But tremendously *beneficial*
    - Structured methodology
    - Correct circuits
    - Automated design, automated verification, many more advantages

More on Flip-Flops and Controllers

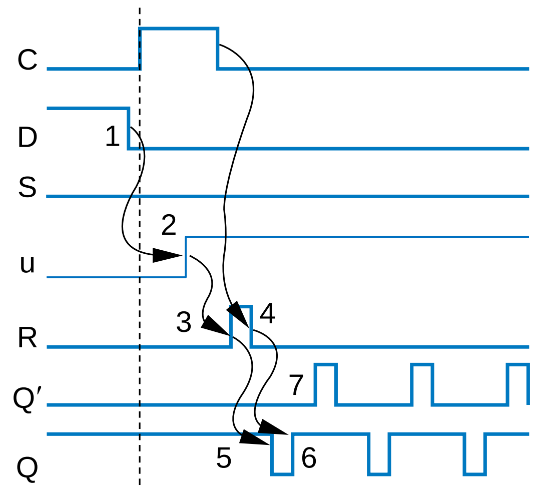
- Non-ideal flip-flop behavior
  - Can't change flip-flop input too close to clock edge
  - Setup time: time D must be stable *before* edge
    - Else, stable value not present at internal latch
  - Hold time: time D must be held stable *after* edge
    - Else, new value doesn't have time to loop around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







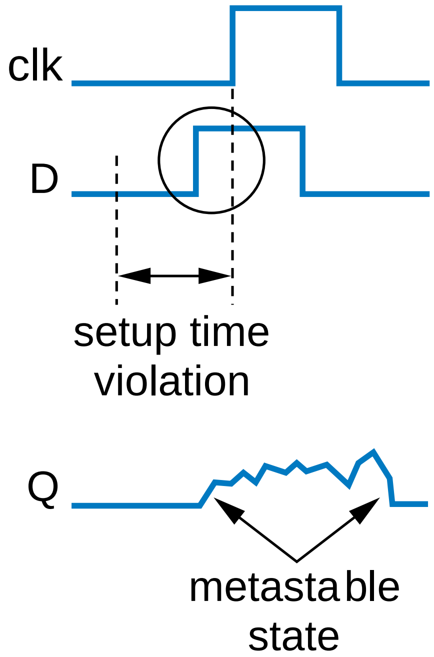
Setup time violation

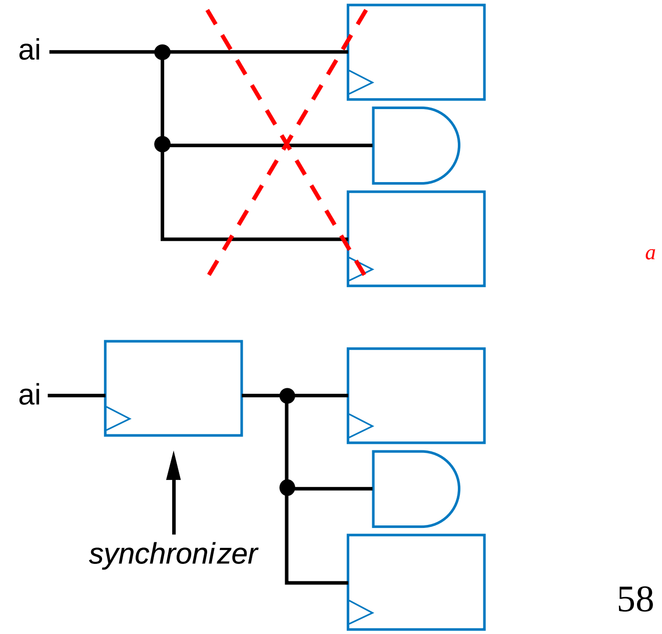


Leads to oscillation!

Metastability

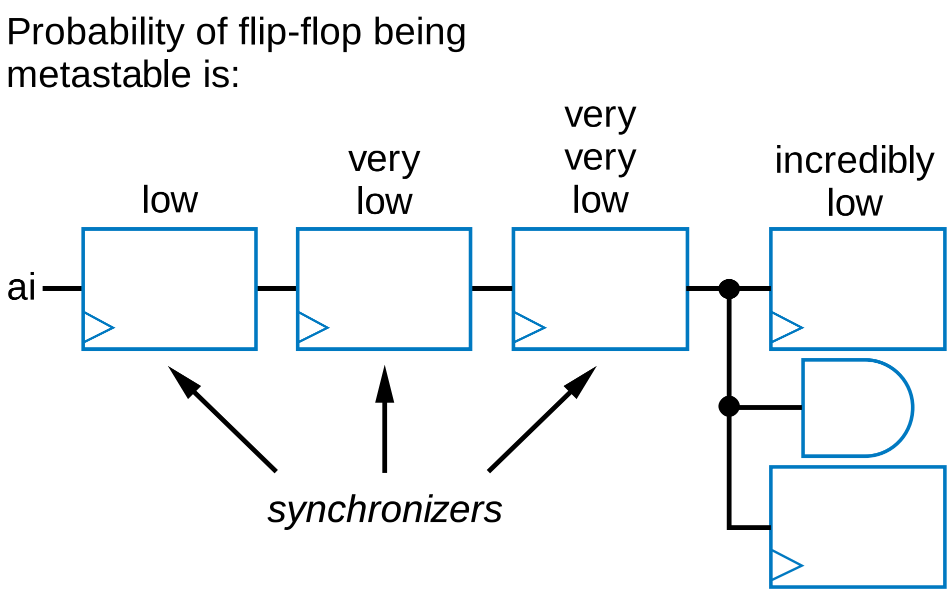
- Violating setup/hold time can lead to bad situation
  - Metastable state: Any flip-flop state other than stable 1 or 0
    - Eventually settles to either, but we don't know which
  - For internal circuits, we can make sure to observe setup time
  - But what if input is from external (asynchronous)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





Metastability

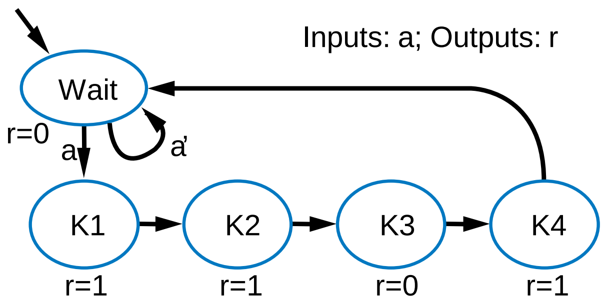
- Synchronizer flip-flop doesn’t completely prevent metastability
  - But reduces probability of metastability in dozens/hundreds of internal flip-flops storing important values
  - Adding more synchronizer flip-flops further reduces probability
    - First ff likely stable before next clock; second ff very unlikely to have setup time violated
  - Drawback: Change on input is delayed to internal flip-flops
    - By three clock cycles in below circuit

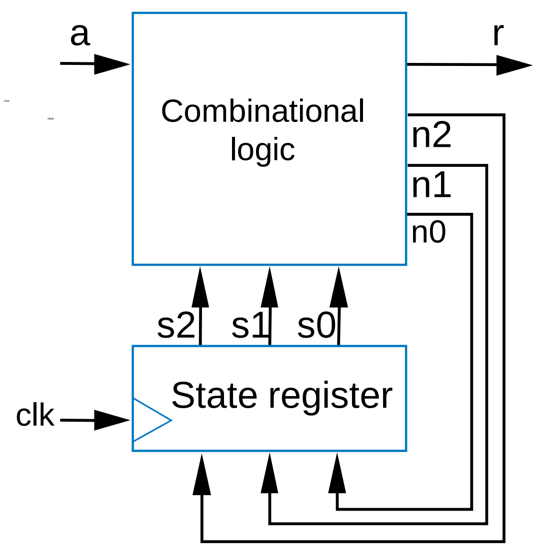


a

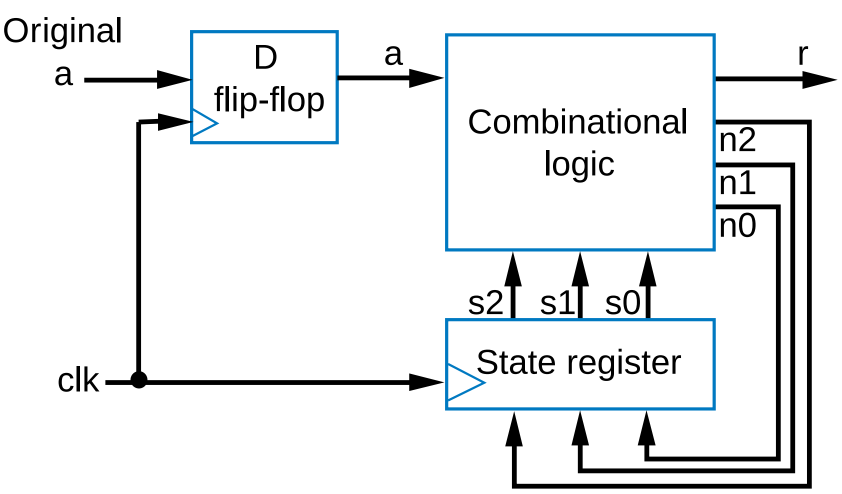
Example of Reducing Metastability Probability

- Recall earlier secure car key controller



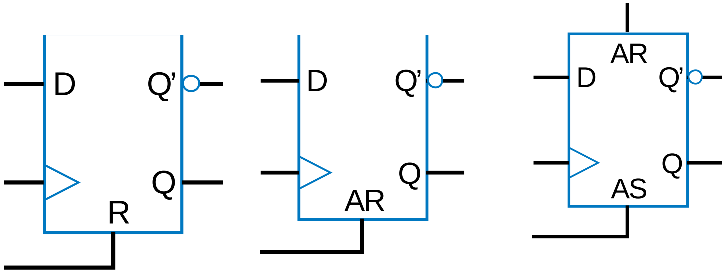


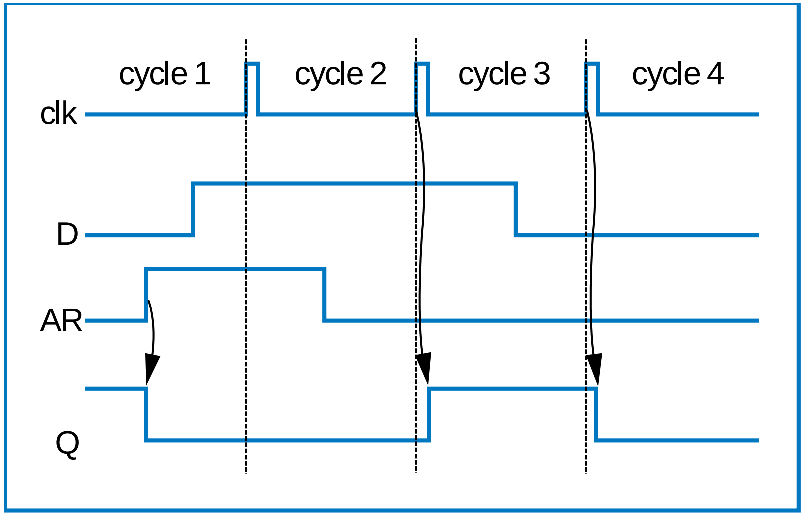
Adding synchronizer flip-flop reduces metastability probability in state register, at expense of 1 cycle delay



Flip-Flop Set and Reset Inputs

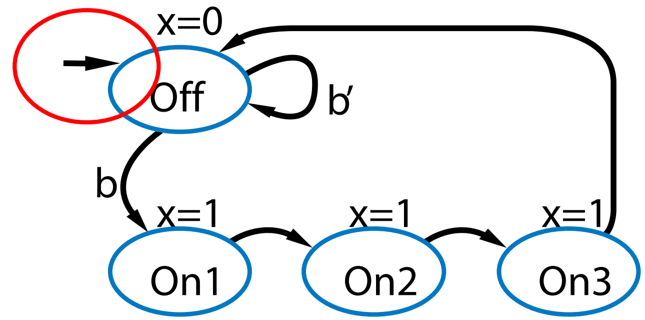
- Some flip-flops have additional reset/set inputs
  - Synchronous
    - Synch. reset: Clears Q to 0 on next clock edge
    - Synch. set: Sets Q to 1 on next clock edge
    - Have priority over D input
  - Asynchronous
    - Asynch. reset: Clear Q to 0, independently of clock
      - Example timing diagram shown
    - Asynch. set: set Q to 1, indep. of clock

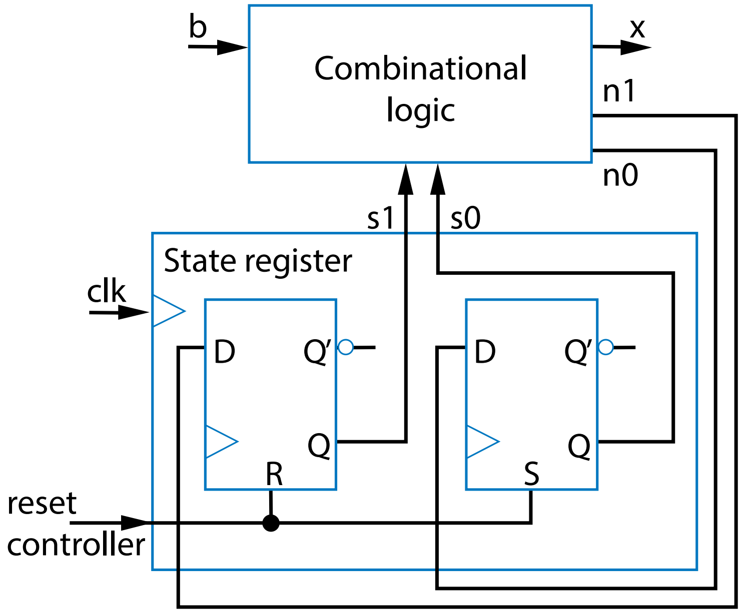




Initial State of a Controller

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

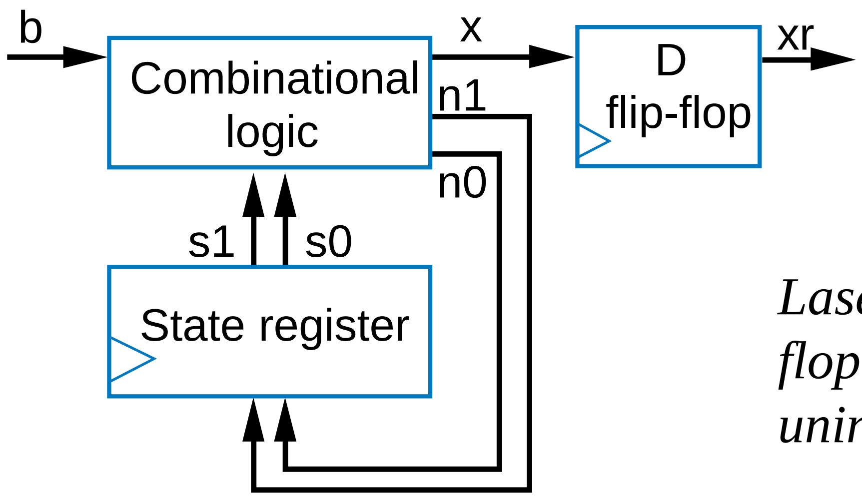




Controller with reset to initial state 01 (assuming state Off was encoded as 01).

Glitching

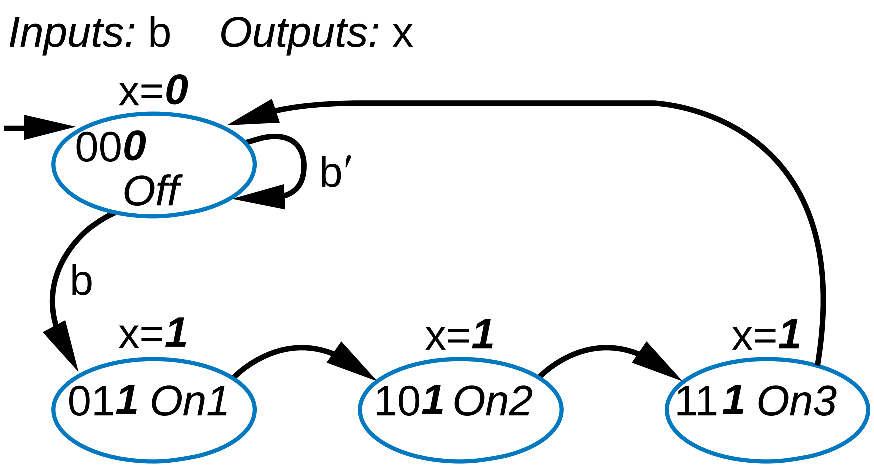
- Glitch: Temporary values on outputs that appear soon after input changes, before stable new output values
- Designer must determine whether glitching outputs may pose a problem
  - If so, may consider adding flip-flops to outputs
    - Delays output by one clock cycle, but may be OK
    - Called *registered*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

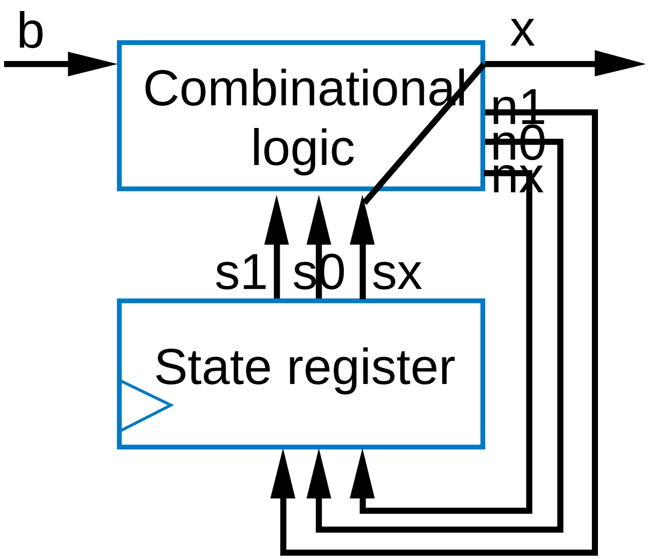


Laser timer controller with flip-flop to prevent glitches on x from unintentionally turning on laser

Glitching

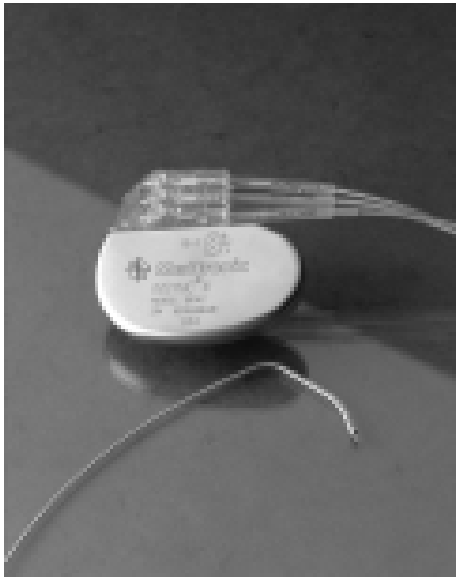
- Alternative registered output approach, avoid 1 cycle delay:
  - Add extra state register bit for each output
  - Connect output directly to its bit
  - No logic between state register flip-flop and output, hence no glitches

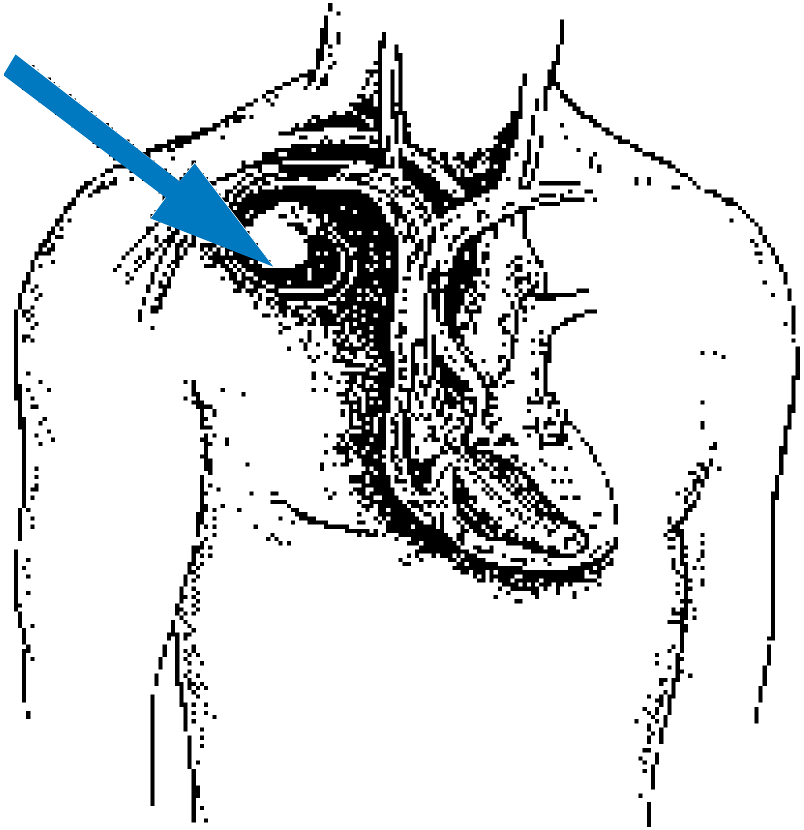




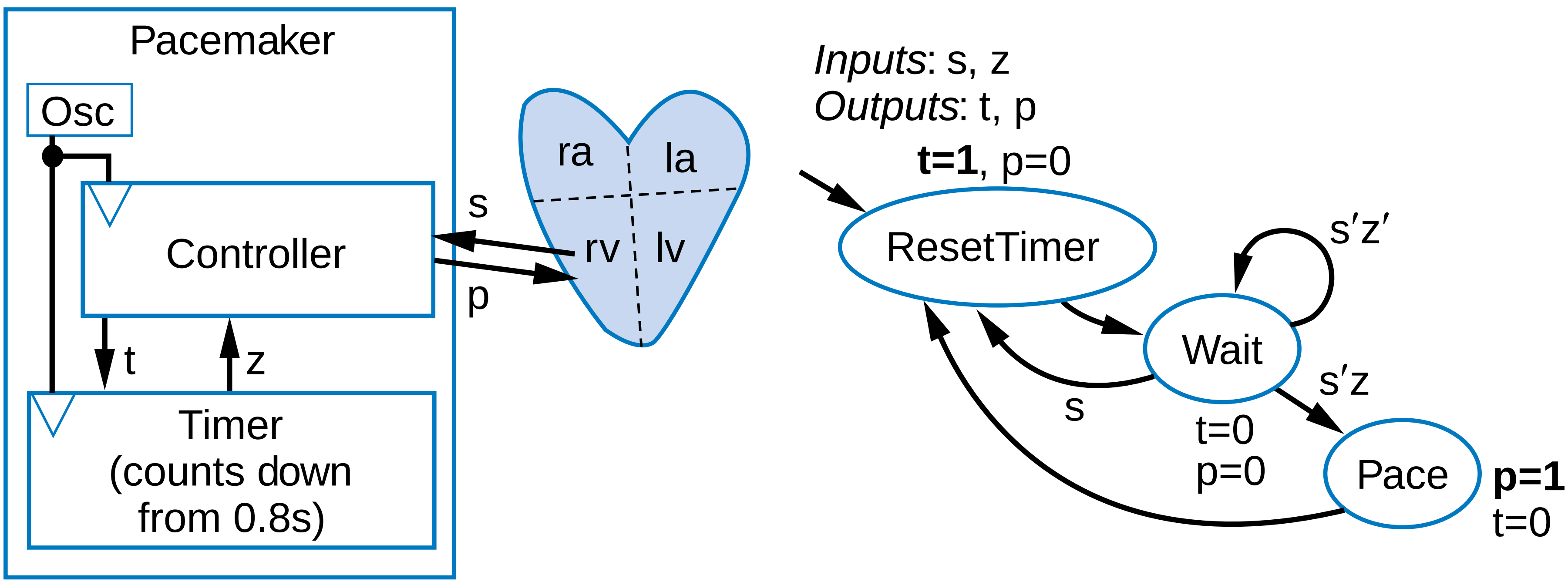
But, uses more flip-flops, plus more logic to compute next state

Product Profile: Pacemaker



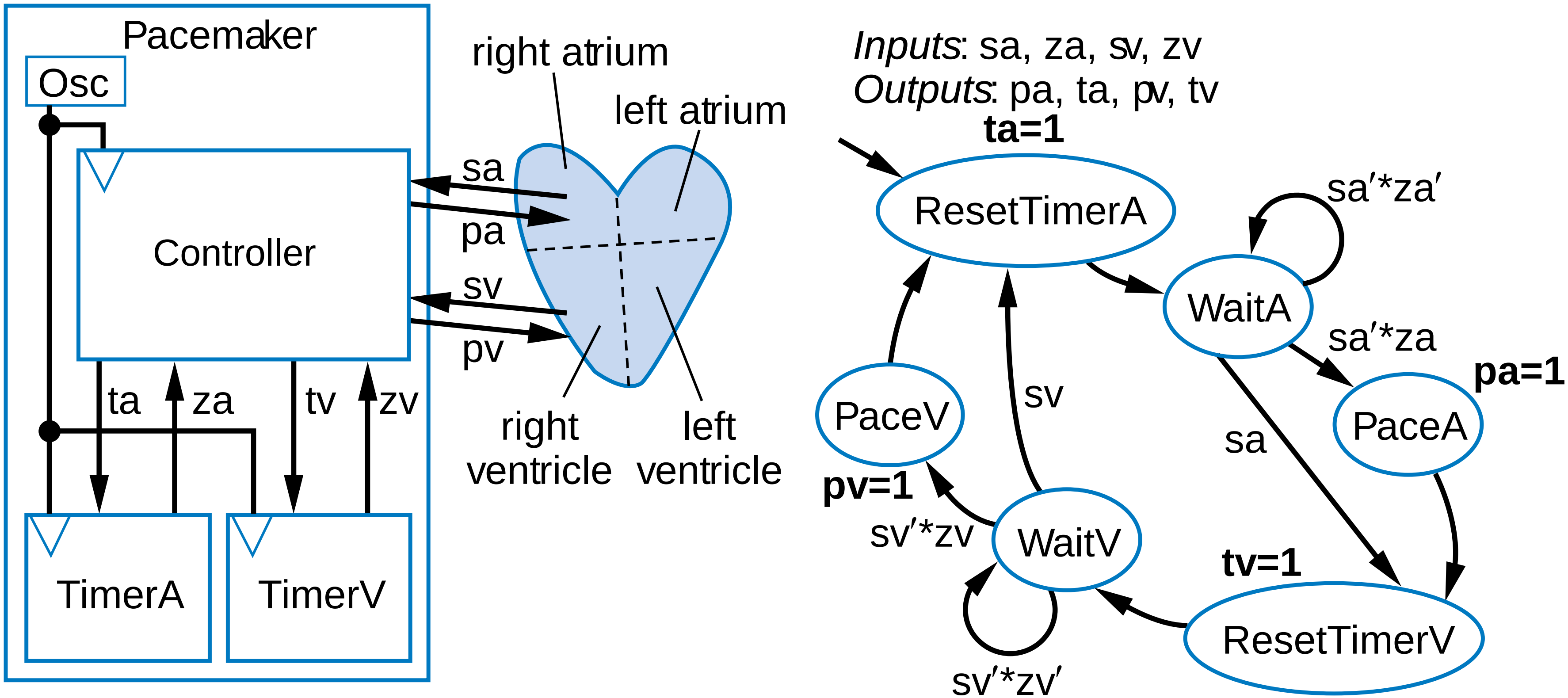


Product Profile: Pacemaker



Basic pacemaker

Product Profile: Pacemaker



Atrioventricular pacemaker

Chapter Summary

- Sequential circuits
  - Have state
- Created robust bit-storage device: D flip-flop
  - Put several together to build register, which we used to store state
- Defined FSM model to capture sequential behavior
  - Using mathematical models – Boolean equations for combinational circuit, and FSMs for sequential circuits – is important
- Defined Capture/Convert process for sequential circuit design
  - Converted FSM to standard controller architecture
- So now we know how to build the class of sequential circuits known as controllers