

ECE-332:437
DIGITAL SYSTEMS DESIGN (DSD)

Fall 2016 – Lecture 5 –
Sequential Circuits

Nagi Naganathan
September 22, 2016

Topics to cover today – September 22, 2016

- Lecture 4
 - Recap
- Lecture 5
 - Sequential Circuits
 - Finite State Machines (FSM)

Announcements - September 22, 2016

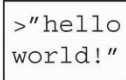


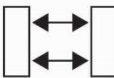
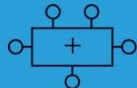

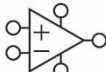


- Swapnil will cover the lecture next week (9/29/16)
- Guest Lecture on SystemVerilog/Verification – Not yet confirmed – Possibly Oct 30

Part 1

Sequential Circuits

Chapter 3 :: Topics

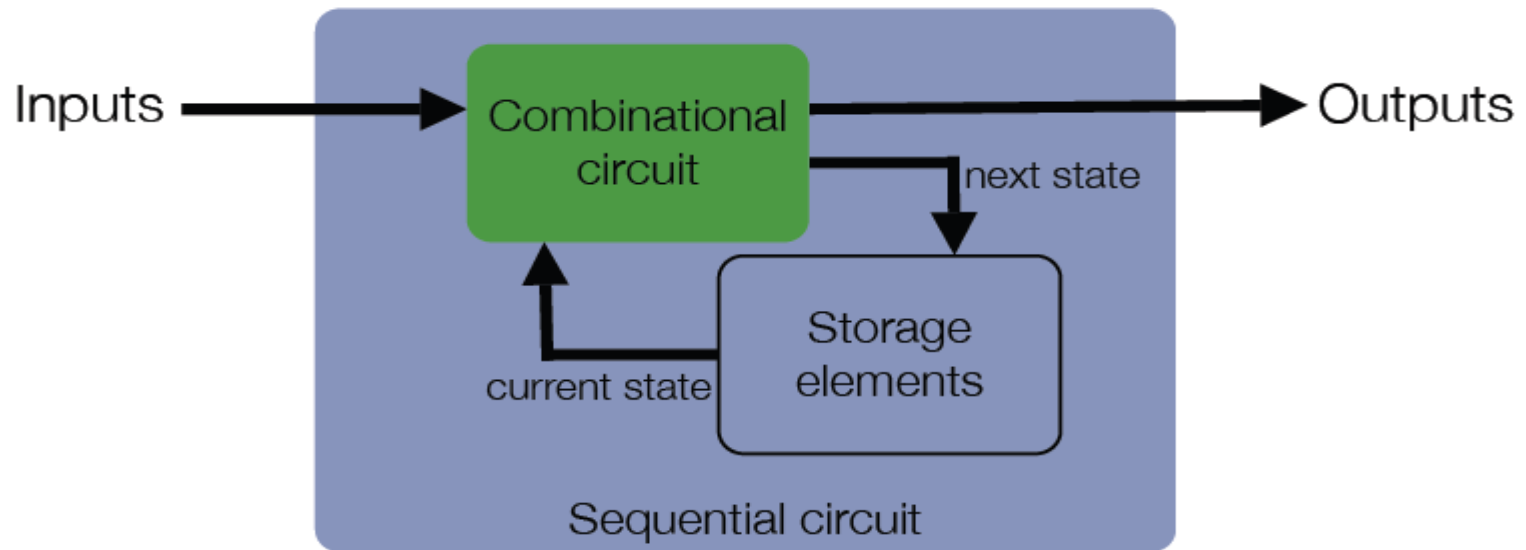
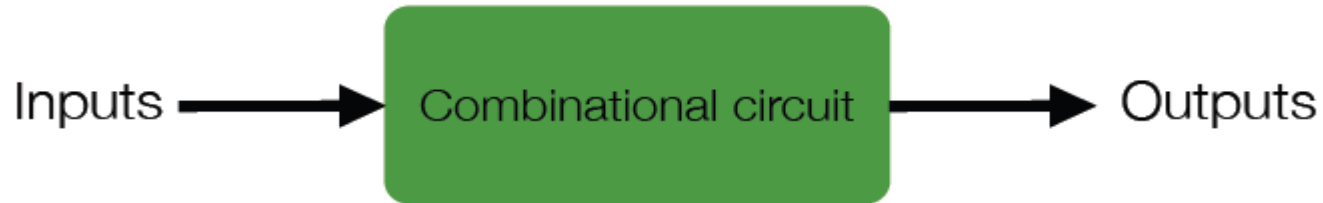
- Introduction
- Latches and Flip-Flops
- Synchronous Logic Design
- Finite State Machines
- Timing of Sequential Logic
- Parallelism

Application Software	
Operating Systems	
Architecture	
Micro-architecture	
Logic	
Digital Circuits	
Analog Circuits	
Devices	
Physics	

Introduction

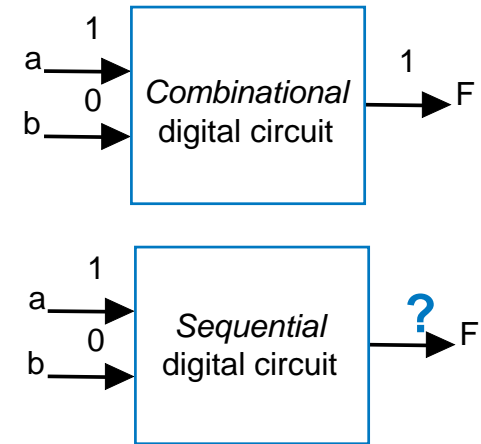
- Outputs of sequential logic depend on current *and* prior input values – it has ***memory***.
- Some definitions:
 - **State**: all the information about a circuit necessary to explain its future behavior
 - **Latches and flip-flops**: state elements that store one bit of state
 - **Synchronous sequential circuits**: combinational logic followed by a bank of flip-flops

Combinational vs Sequential Circuits



Sequential Circuits

- 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



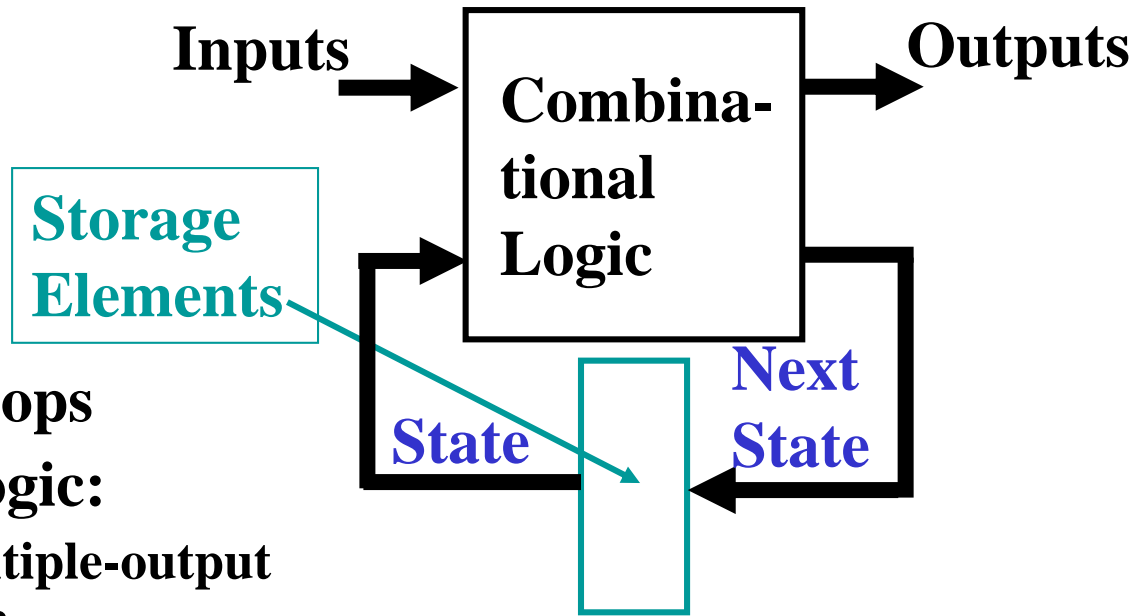
*Must know
sequence of
past inputs to
know output*



Introduction to Sequential Circuits

- A Sequential circuit contains:

- Storage elements: Latches or Flip-Flops
- Combinational Logic:
 - Implements a multiple-output switching function
 - Inputs are signals from the outside.
 - Outputs are signals to the outside.
 - Other inputs, State or Present State, are signals from storage elements.
 - The remaining outputs, Next State are inputs to storage elements.

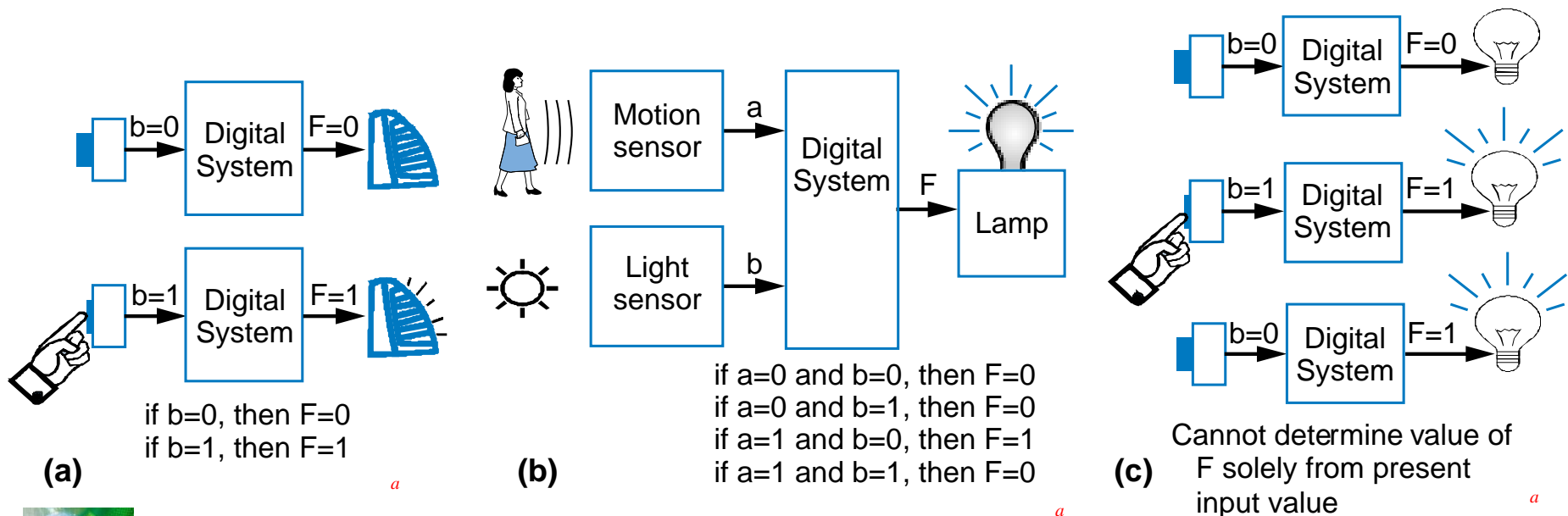


Introduction

- Let's learn to design digital circuits, starting with a simple form of circuit:

- **Combinational circuit**

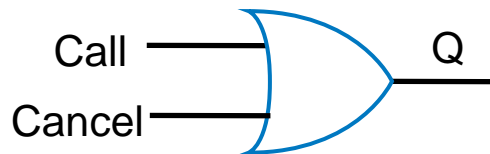
- Outputs depend solely on the present combination of the circuit inputs' values
- Vs. sequential circuit: Has "memory" that impacts outputs too



Storing One Bit – Flip-Flops

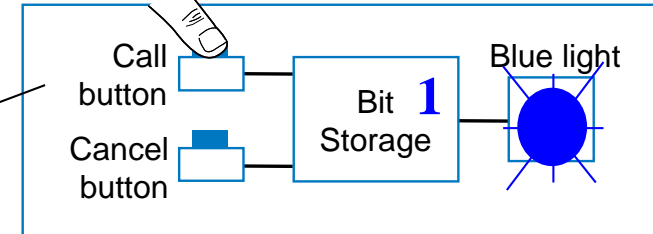
Example Requiring Bit Storage

- 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?

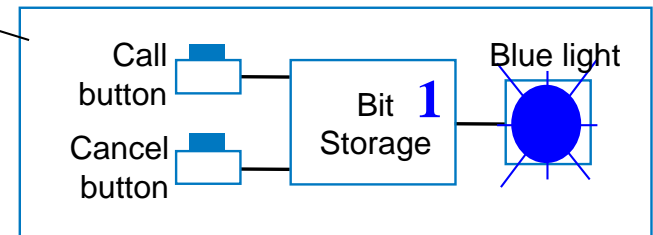


Doesn't work. $Q=1$ when $\text{Call}=1$, but doesn't stay 1 when Call returns to 0

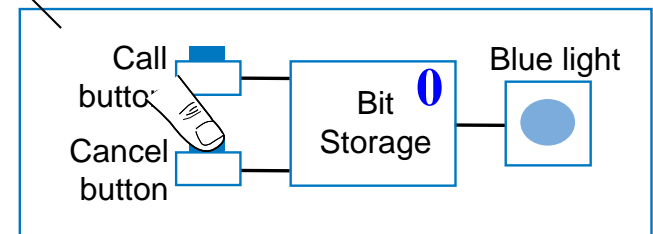
Need some form of "feedback" in the circuit



1. Call button pressed – light turns on



2. Call button released – light stays on



3. Cancel button pressed – light turns off



Sequential Circuits – Examples

- Digital Camera – Stores Pictures
- Traffic Light Controller
- Door Bell ??

- Feedback Sequential Circuits
 - Ordinary gates and feedback loops to obtain memory
- Clocked Synchronous State Machine
 - Uses above building blocks with a controlling clock

Sequential Circuits

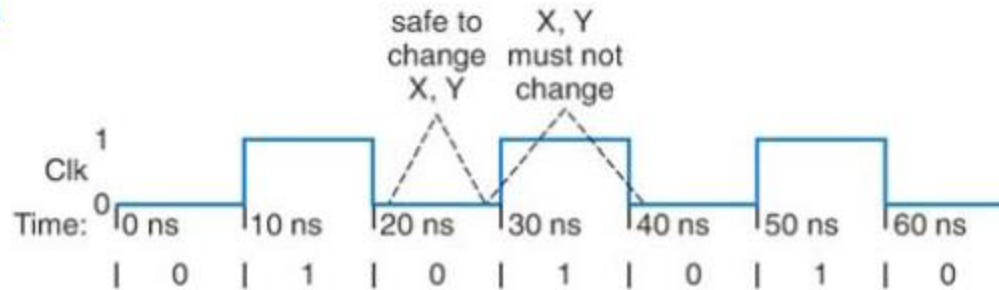
- Give sequence to events
- Have memory (short-term)
- Use feedback from output to input to store information

Latches and Flip-Flops

- Latches
 - Sequential device that can change its output at any time
- Flip-Flop
 - Samples its inputs and changes outputs only when a clocking signal is changing

Clocks

- Period
 - Time after which the signal repeats itself
 - Time between successive 1's
- Clock Cycle
 - One segment of time where the clock is 1 and then 0 or vice-versa
- Frequency
 - Number of cycles per second – $1/\text{clock period}$



rising edges

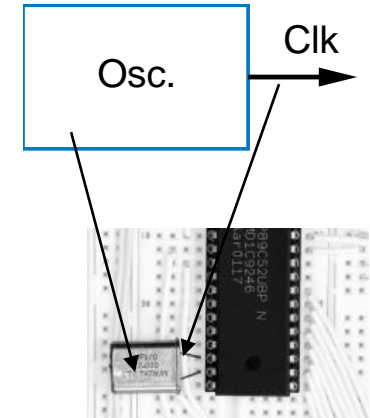
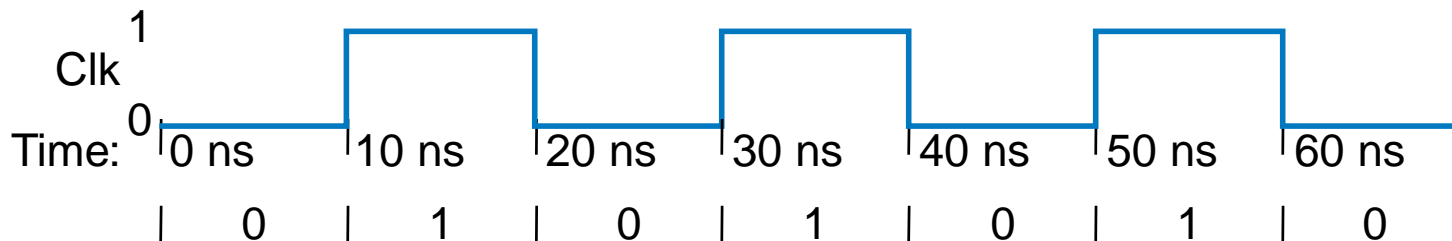


falling edges



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" = $1/20 \text{ ns} = 50 \text{ MHz}$
 - "Cycle" is duration of 1 period (20 ns); below shows 3.5 cycles



Period/Freq shortcut: Remember $1 \text{ ns} \rightarrow 1 \text{ GHz}$

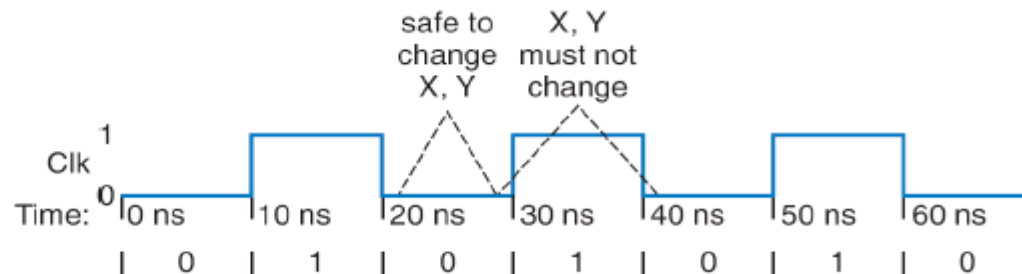
Freq.	Period
100 GHz	0.01 ns
10 GHz	0.1 ns
1 GHz	1 ns
100 MHz	10 ns
10 MHz	100 ns



Clocks

Clocks

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



- **Clock** -- Pulsing signal for enabling latches; ticks like a clock
- **Synchronous** circuit: sequential circuit with a clock
- **Clock period**: time btwn pulse starts
 - Above signal: period = 20 ns
- **Clock cycle**: one such time interval
 - Above signal shows 3.5 clock cycles
- **Clock duty cycle**: time clock is high
 - 50% in this case
- **Clock frequency**: $1/\text{period}$
 - Above : $\text{freq} = 1 / 20\text{ns} = 50\text{MHz}$;

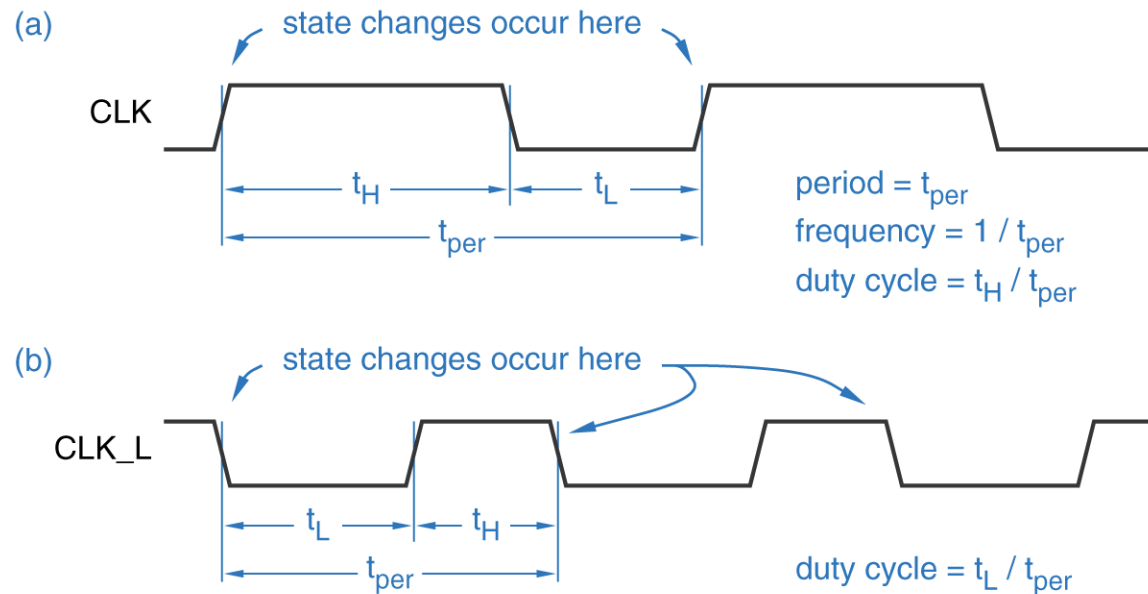
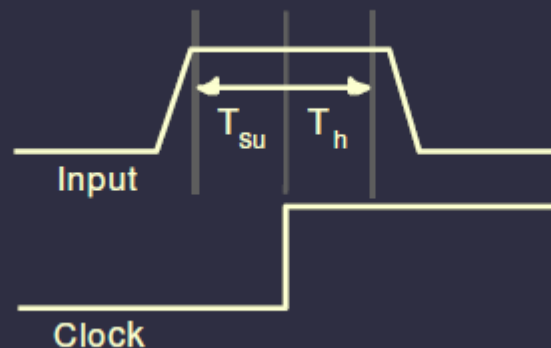


Figure 7-1

Clock signals: (a) active high; (b) active low.

Clocking Terms

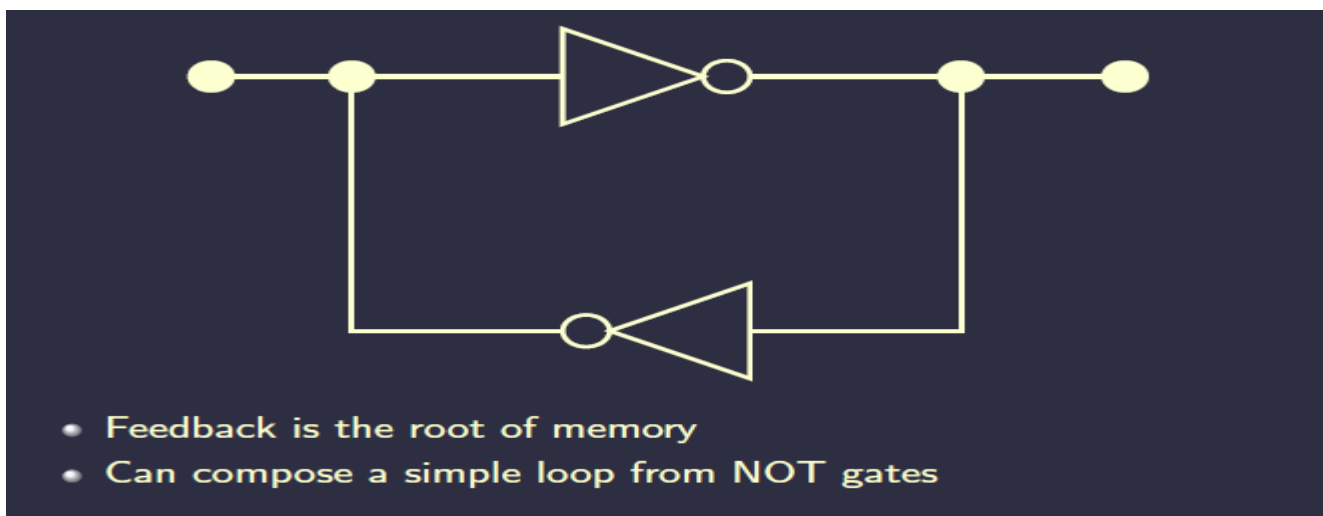
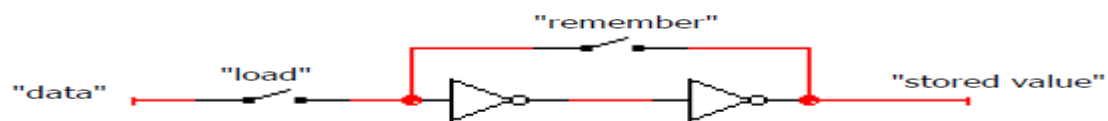


- Clock – Rising edge, falling edge, high level, low level, period
- Setup time: Minimum time before clocking event by which input must be stable (T_{SU})
- Hold time: Minimum time after clocking event for which input must remain stable (T_H)
- Window: From setup time to hold time

State Elements

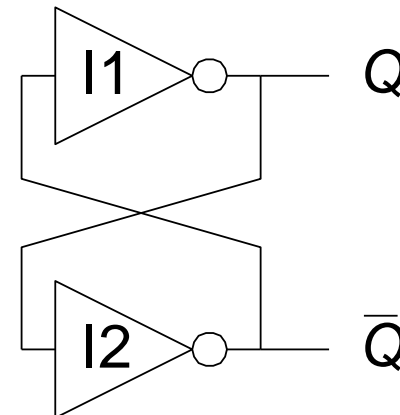
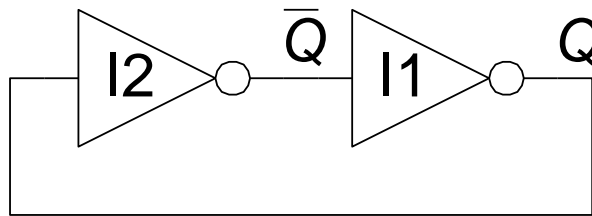
- The state of a circuit influences its future behavior
- State elements store state
 - Bistable circuit
 - SR Latch
 - D Latch
 - D Flip-flop

Sequential Circuits



Bistable Circuit

- Fundamental building block of other state elements
- Two outputs: Q , \bar{Q}
- No inputs

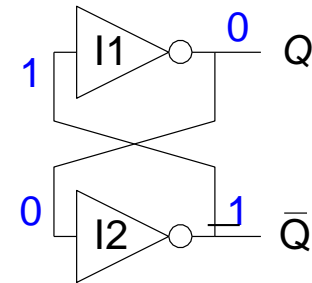


Bistable Circuit Analysis

- Consider the two possible cases:

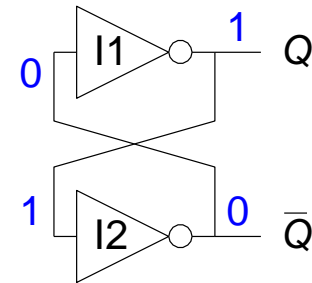
– $Q = 0$:

then $\bar{Q} = 1$, $Q = 0$ (consistent)



– $Q = 1$:

then $\bar{Q} = 0$, $Q = 1$ (consistent)



- Stores 1 bit of state in the state variable, Q (or \bar{Q})
- But there are **no inputs to control the state**

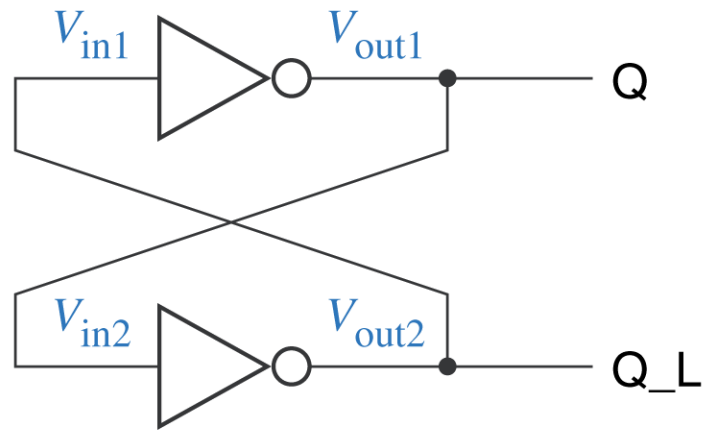


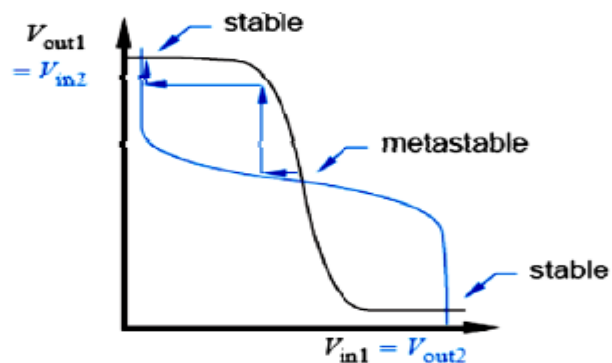
Figure 7-2

A pair of inverters forming a bistable element.

Bistable Circuits

- Meta-stable behavior

Figure 7-3
Transfer functions for
inverters in a bistable
feedback loop.



Transfer function:

$$V_{out1} = T(V_{in1})$$

$$V_{out2} = T(V_{in2})$$

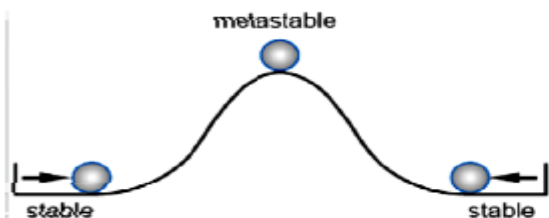
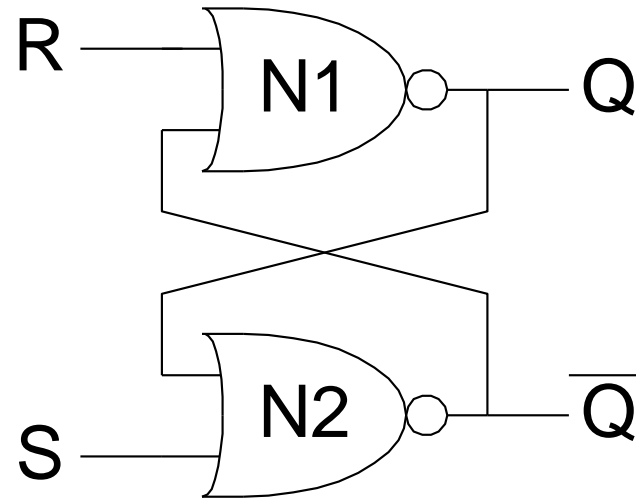


Figure 7-4
Ball and hill analogy for
metastable behavior.

SR (Set/Reset) Latch

- SR Latch

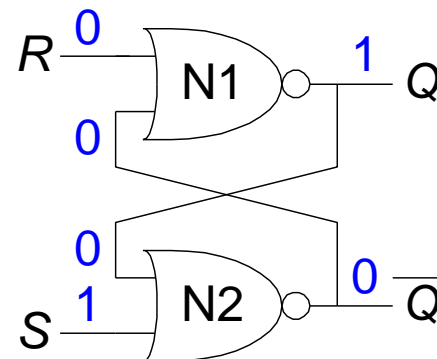


- Consider the four possible cases:
 - $S = 1, R = 0$
 - $S = 0, R = 1$
 - $S = 0, R = 0$
 - $S = 1, R = 1$

SR Latch Analysis

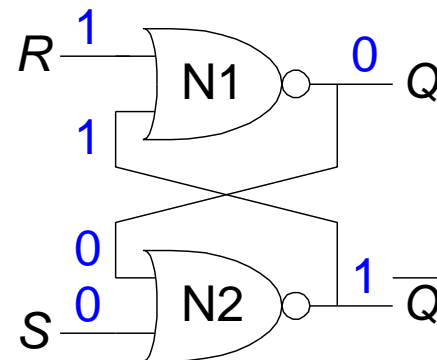
– $S = 1, R = 0$:

then $Q = 1$ and $\bar{Q} = 0$



– $S = 0, R = 1$:

then $Q = 0$ and $\bar{Q} = 1$

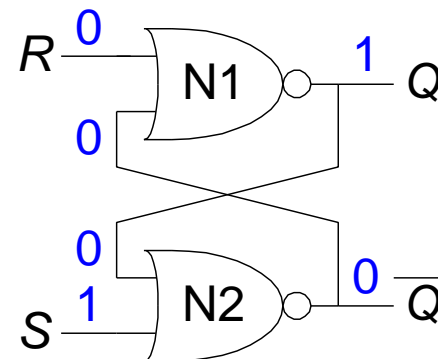


SR Latch Analysis

– $S = 1, R = 0$:

then $Q = 1$ and $\bar{Q} = 0$

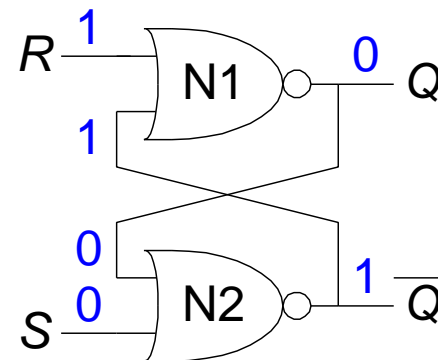
Set the output



– $S = 0, R = 1$:

then $Q = 0$ and $\bar{Q} = 1$

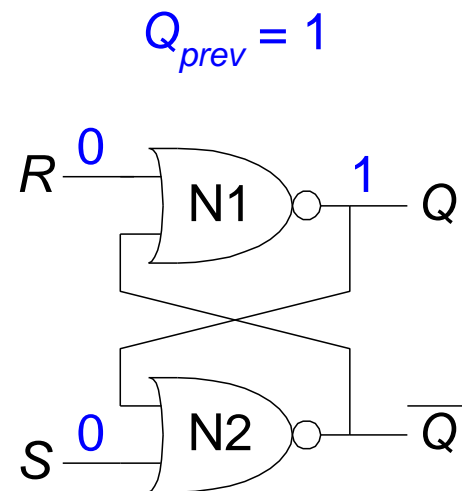
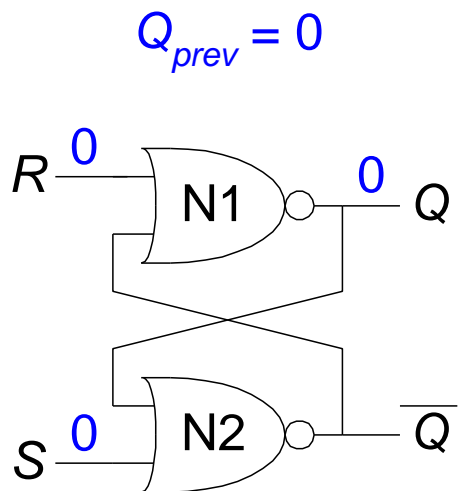
Reset the output



SR Latch Analysis

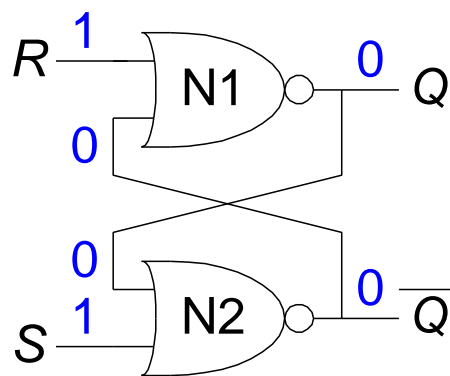
– $S = 0, R = 0$:

then $Q = Q_{prev}$



– $S = 1, R = 1$:

then $Q = 0, \bar{Q} = 0$

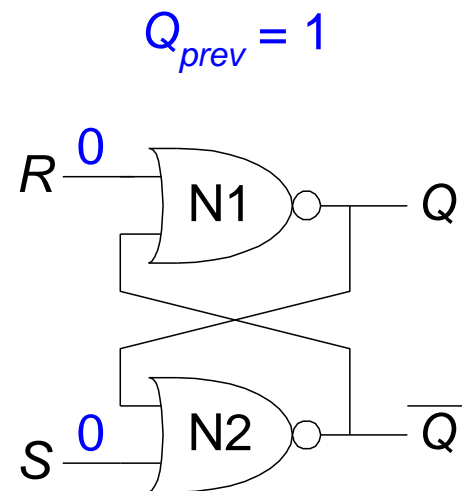
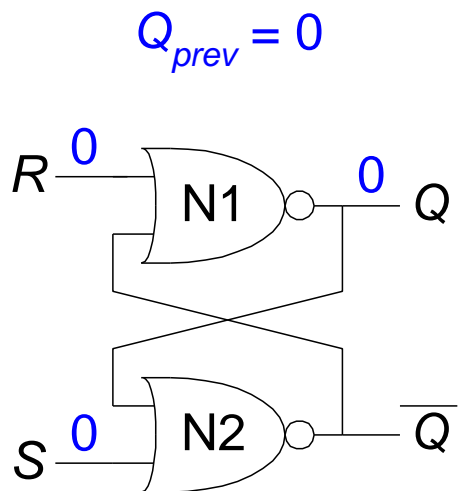


SR Latch Analysis

– $S = 0, R = 0$:

then $Q = Q_{prev}$

Memory!

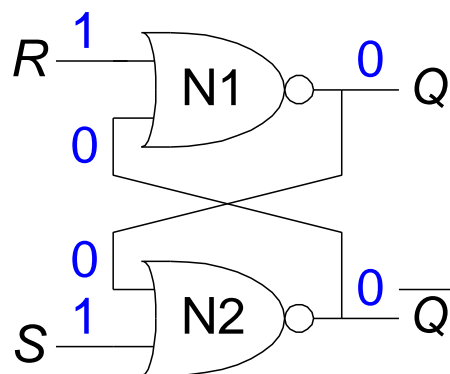


– $S = 1, R = 1$:

then $Q = 0, \bar{Q} = 0$

Invalid State

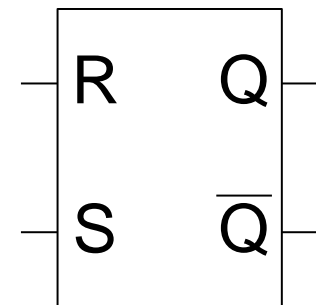
$\bar{Q} \neq \text{NOT } Q$



SR Latch Symbol

- SR stands for Set/Reset Latch
 - Stores one bit of state (Q)
- Control what value is being stored with S , R inputs
 - **Set:** Make the output 1
($S = 1$, $R = 0$, $Q = 1$)
 - **Reset:** Make the output 0
($S = 0$, $R = 1$, $Q = 0$)

SR Latch
Symbol



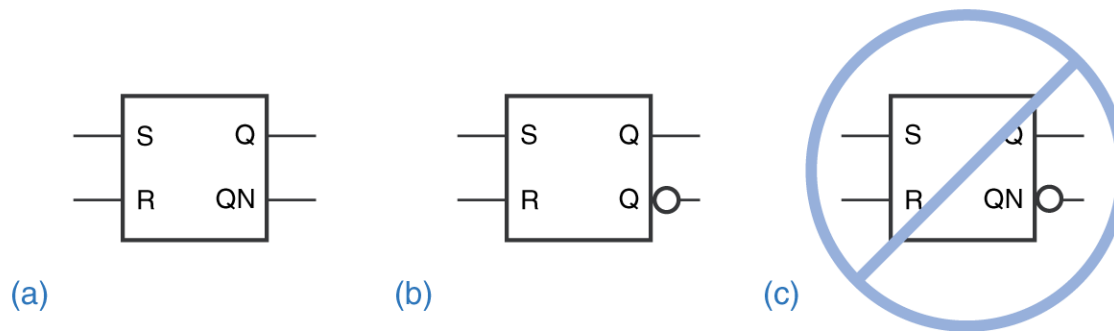
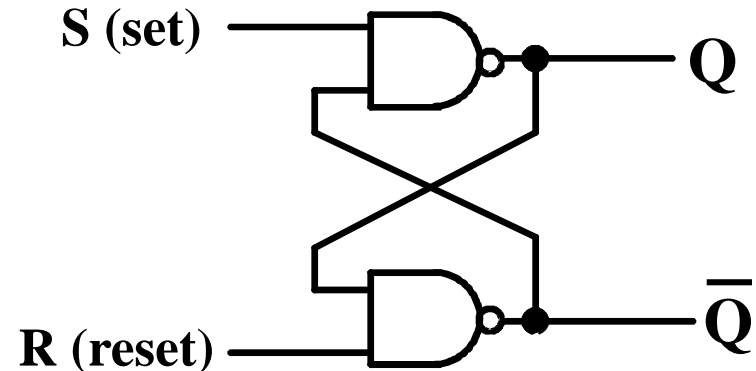


Figure 7-7

Symbols for an S-R latch: (a) without bubble; (b) preferred for bubble-to-bubble design; (c) incorrect because of double negation.

Basic (NAND) \bar{S} – \bar{R} Latch

- “Cross-Coupling”
two NAND gates gives
the \bar{S} - \bar{R} Latch:



- Which has the time
sequence behavior:

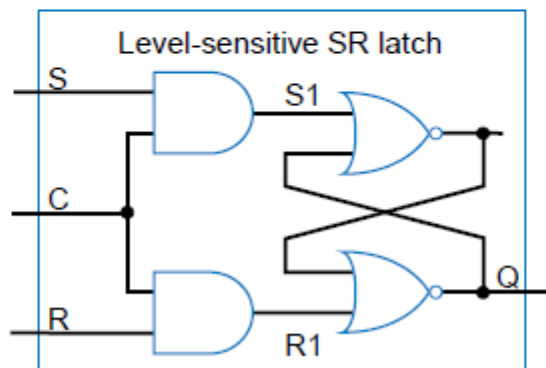
Time

R	S	Q	\bar{Q}	Comment
1	1	?	?	Stored state unknown
1	0	1	0	“Set” Q to 1
1	1	1	0	Now Q “remembers” 1
0	1	0	1	“Reset” Q to 0
1	1	0	1	Now Q “remembers” 0
0	0	1	1	Both go high
1	1	?	?	Unstable!

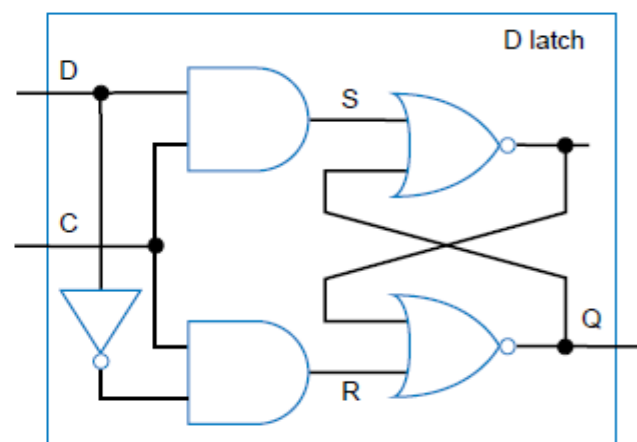
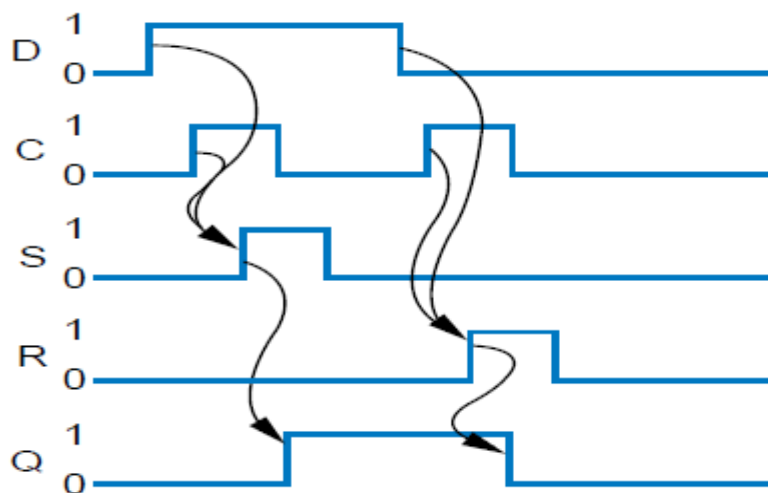
- $S = 0, R = 0$ is
forbidden as
input pattern

Level-Sensitive SR Latch

- Add input “C”
 - Change C to 1 only after S and R are stable
 - C is usually a clock (CLK)



Level-Sensitive D Latch

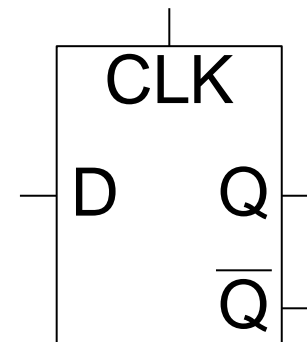


- SR latch requires careful design so $SR=11$ never occurs
- D latch relieves designer of that burden
 - Inserted inverter ensures R always opposite of S

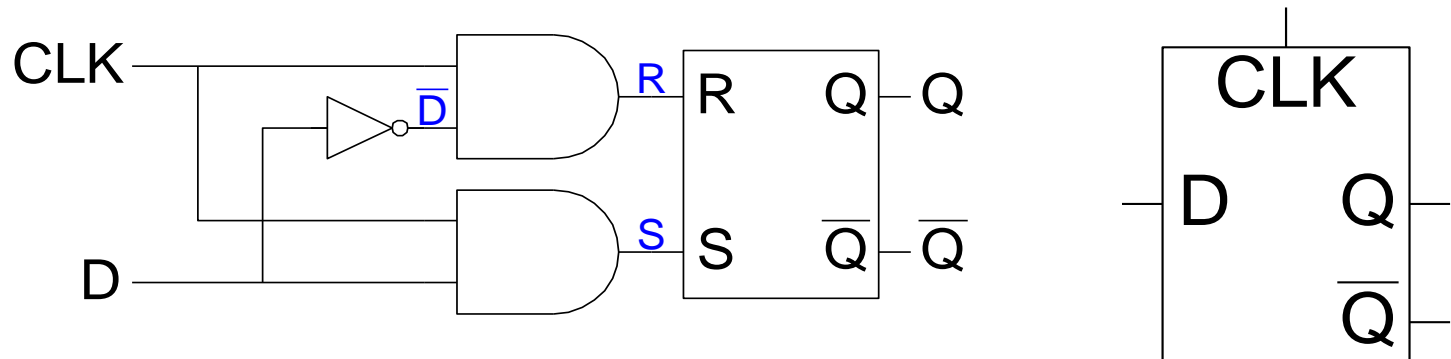
D Latch

- Two inputs: CLK , D
 - CLK : controls *when* the output changes
 - D (the data input): controls *what* the output changes to
- Function
 - When $CLK = 1$,
 D passes through to Q (*transparent*)
 - When $CLK = 0$,
 Q holds its previous value (*opaque*)
- Avoids invalid case when
 $Q \neq \text{NOT } \bar{Q}$

D Latch
Symbol

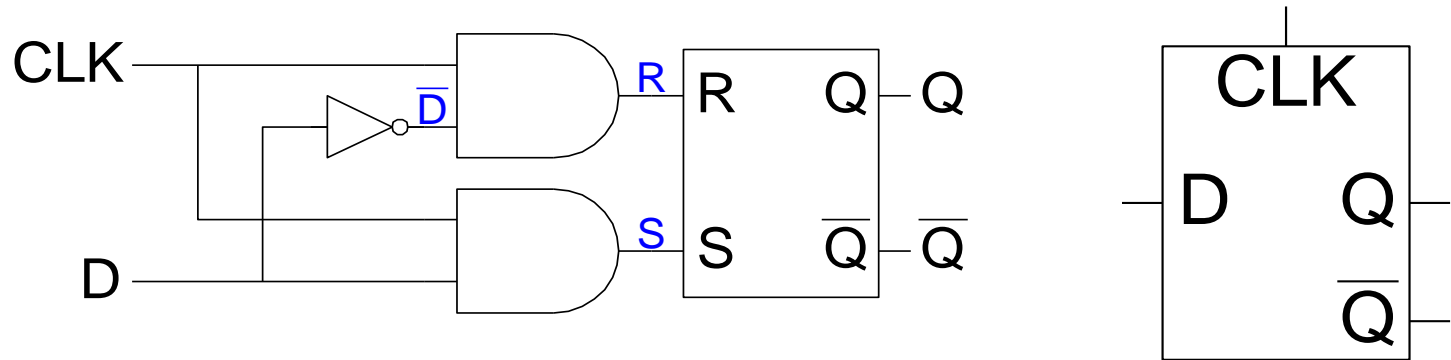


D Latch Internal Circuit



CLK	D	\overline{D}	S	R	Q	\overline{Q}
0	X					
1	0					
1	1					

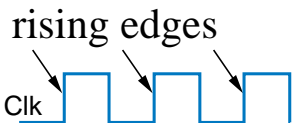
D Latch Internal Circuit



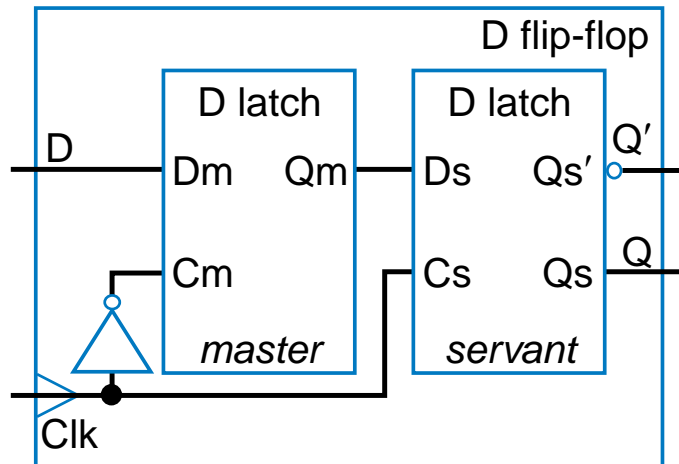
CLK	D	\overline{D}	S	R	Q	\overline{Q}
0	X	X	0	0	Q_{prev}	\overline{Q}_{prev}
1	0	1	0	1	0	1
1	1	0	1	0	1	0

D Flip-Flop

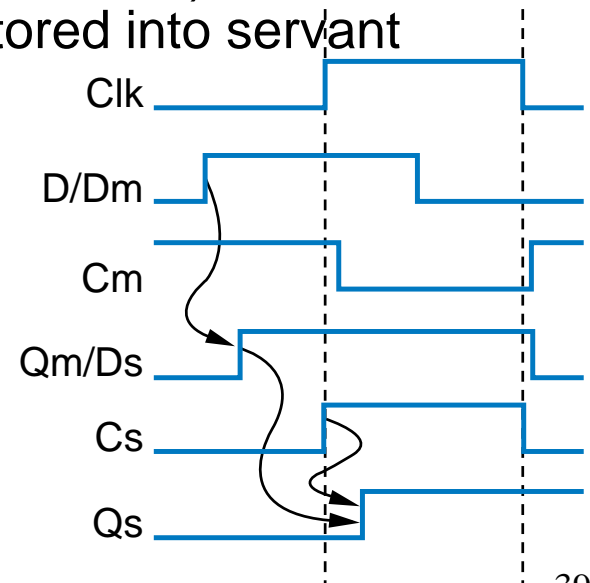
Can we design bit storage that only stores a value on the rising edge of a clock signal?



- **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



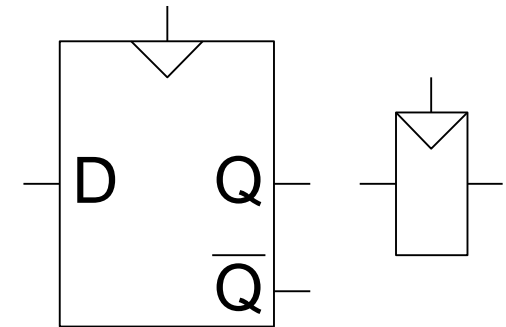
Note:
Hundreds
of different
flip-flop
designs
exist



D Flip-Flop

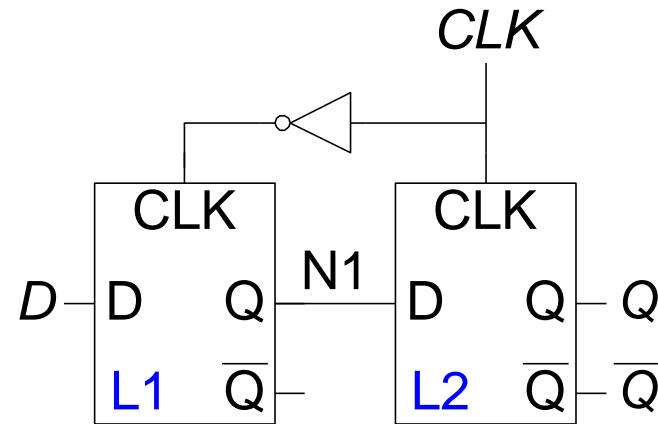
- **Inputs:** CLK , D
- **Function**
 - Samples D on rising edge of CLK
 - When CLK rises from 0 to 1, D passes through to Q
 - Otherwise, Q holds its previous value
 - Q changes only on rising edge of CLK
- Called *edge-triggered*
- Activated on the clock edge

D Flip-Flop Symbols



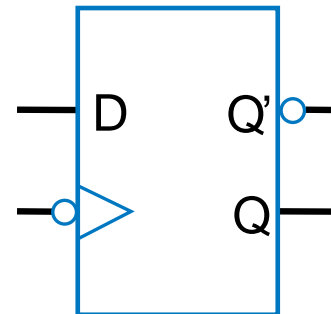
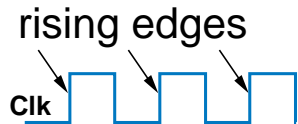
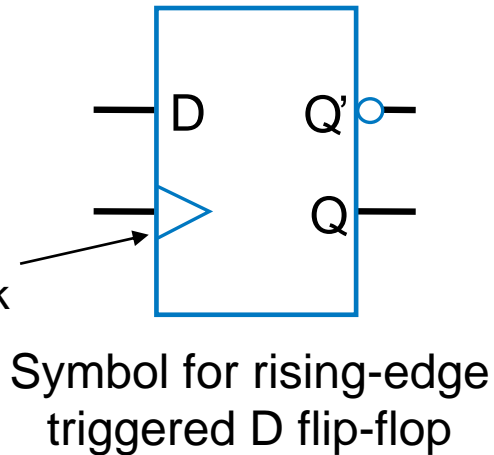
D Flip-Flop Internal Circuit

- Two back-to-back latches (L1 and L2) controlled by complementary clocks
- When $CLK = 0$
 - L1 is transparent
 - L2 is opaque
 - D passes through to N1
- When $CLK = 1$
 - L2 is transparent
 - L1 is opaque
 - N1 passes through to Q
- Thus, on the edge of the clock (when CLK rises from $0 \rightarrow 1$)
 - D passes through to Q



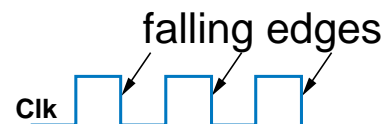
D Flip-Flop

The triangle means edge-triggered clock input



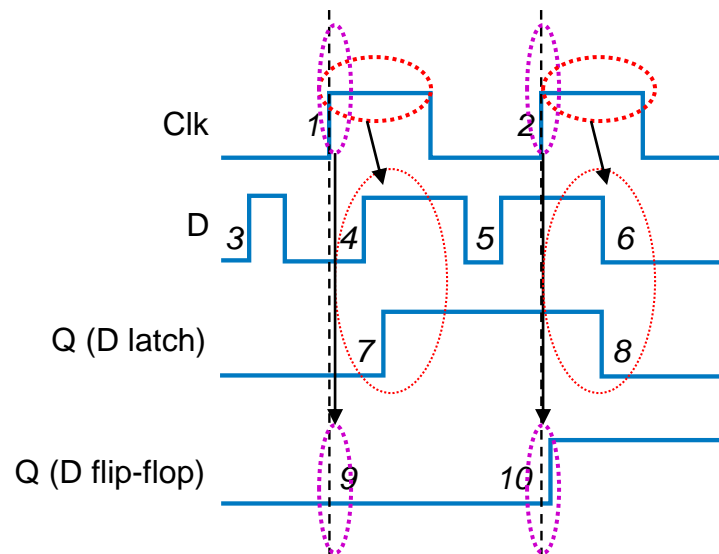
Internal design: Just invert servant clock rather than master

Symbol for falling-edge triggered D 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
- Comparing behavior of latch and flip-flop:

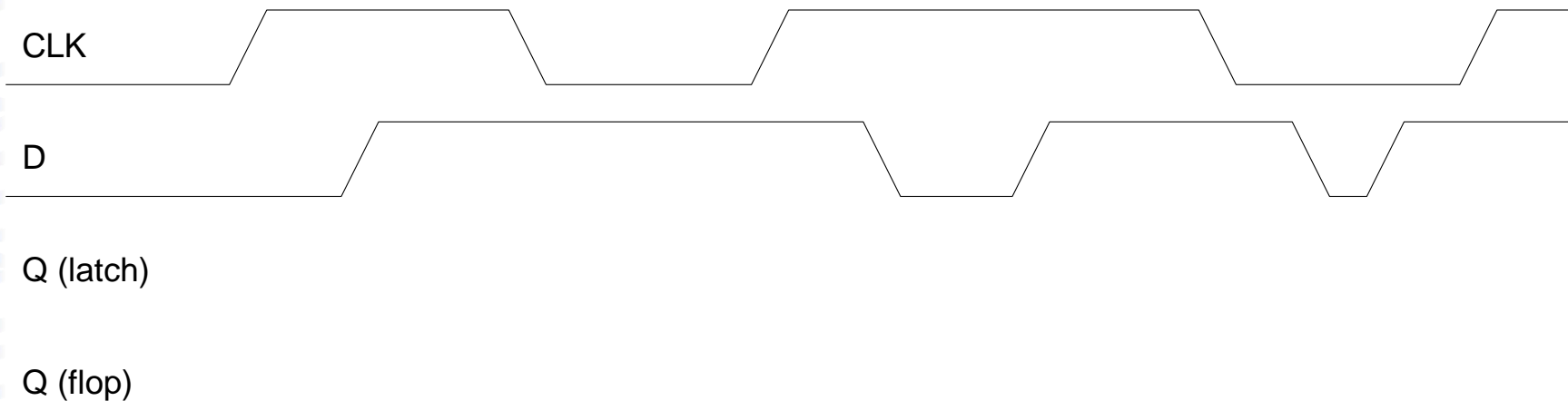
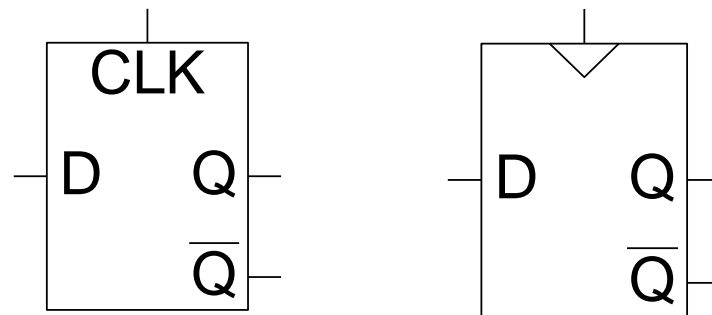


*Latch follows D
while Clk is 1*

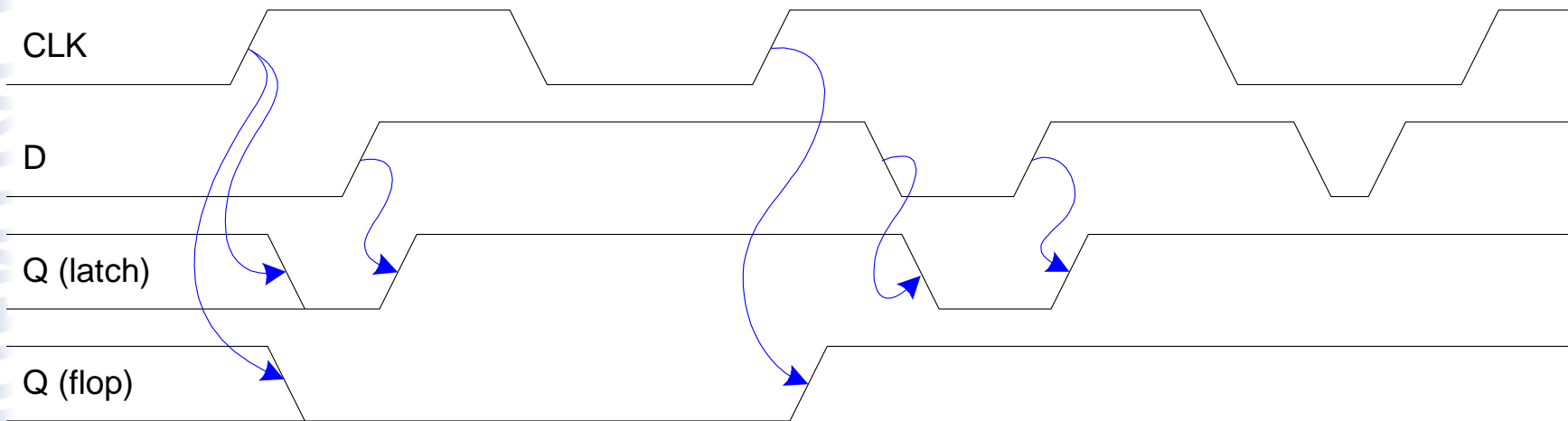
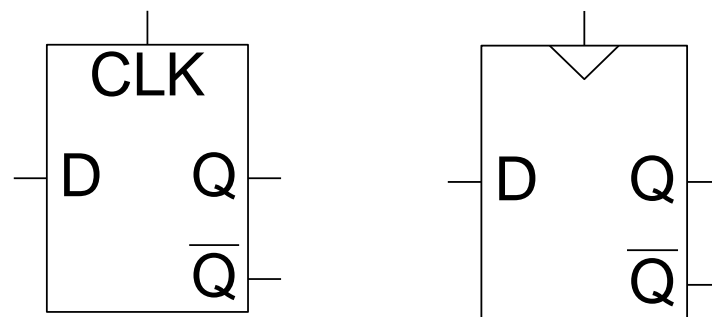
*Flip-flop only loads D
during Clk rising edge*



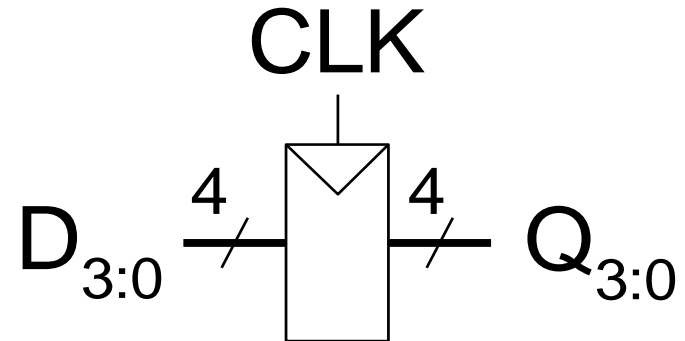
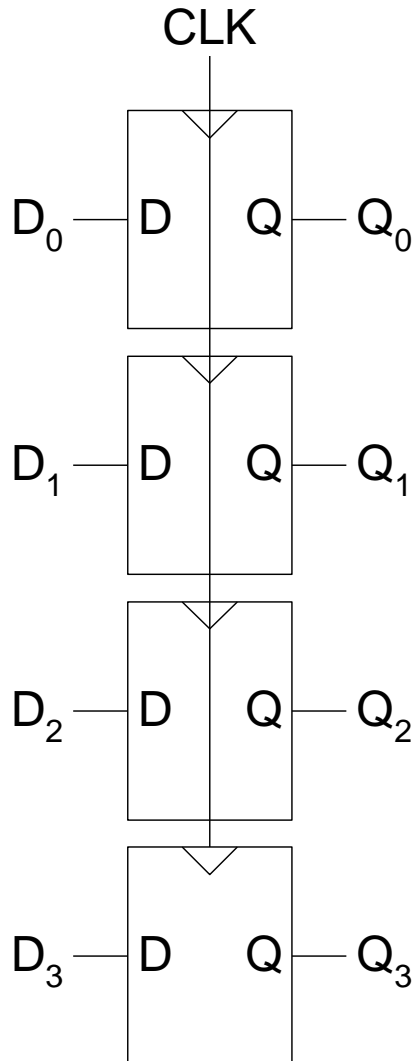
D Latch vs. D Flip-Flop



D Latch vs. D Flip-Flop



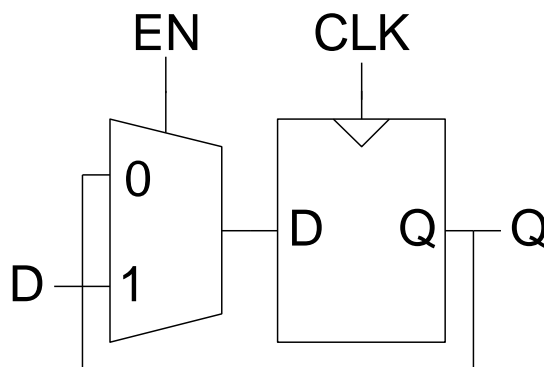
Registers



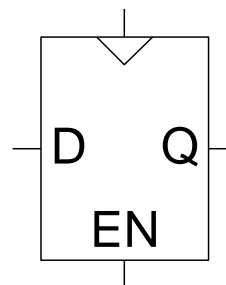
Enabled Flip-Flops

- **Inputs:** CLK , D , EN
 - The enable input (EN) controls when new data (D) is stored
- **Function**
 - $EN = 1$: D passes through to Q on the clock edge
 - $EN = 0$: the flip-flop retains its previous state

Internal
Circuit



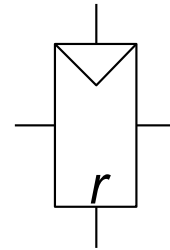
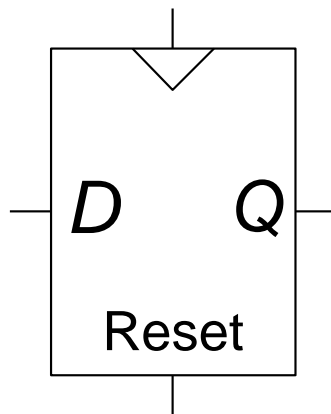
Symbol



Resettable Flip-Flops

- **Inputs:** CLK , D , $Reset$
- **Function:**
 - $Reset = 1$: Q is forced to 0
 - $Reset = 0$: flip-flop behaves as ordinary D flip-flop

Symbols

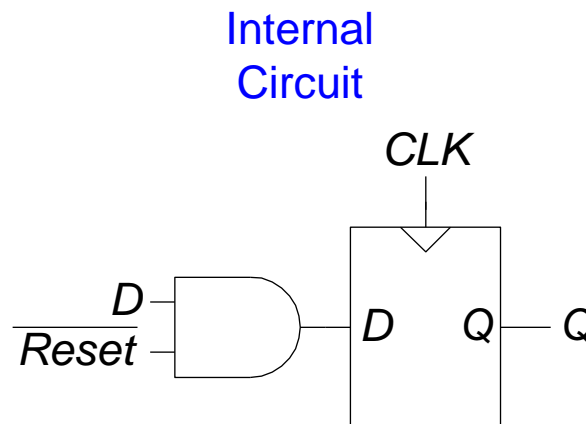


Resettable Flip-Flops

- Two types:
 - **Synchronous:** resets at the clock edge only
 - **Asynchronous:** resets immediately when $Reset = 1$
- Asynchronously resettable flip-flop requires changing the internal circuitry of the flip-flop
- Synchronously resettable flip-flop?

Resettable Flip-Flops

- Two types:
 - **Synchronous:** resets at the clock edge only
 - **Asynchronous:** resets immediately when $Reset = 1$
- Asynchronously resettable flip-flop requires changing the internal circuitry of the flip-flop
- Synchronously resettable flip-flop?



Other Flip-Flop Types

- **J-K and T flip-flops**
 - **Behavior**
 - **Implementation**

J-K Flip-flop

■ Behavior

- Same as S-R flip-flop with J analogous to S and K analogous to R
- Except that $J = K = 1$ is allowed, and
- For $J = K = 1$, the flip-flop changes to the *opposite state*

■ Behavior

- Same as S-R flip-flop with J analogous to S and K analogous to R
- Except that $J = K = 1$ is allowed, and
- For $J = K = 1$, the flip-flop changes to the *opposite state*

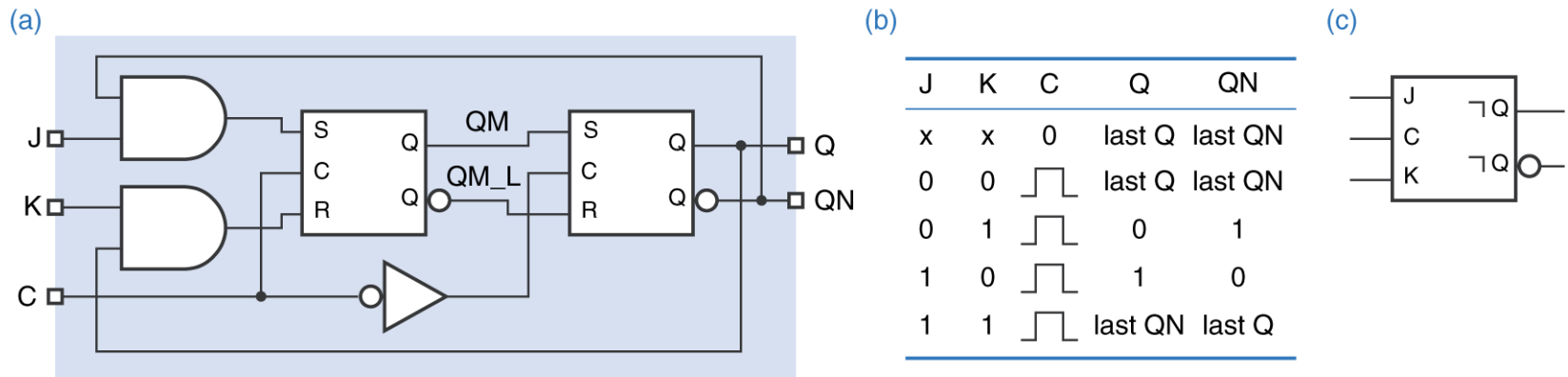


Figure 7-26

Master/slave J-K flip-flop: (a) circuit design using S-R latches; (b) function table; (c) logic symbol.

- The key states are $J=K=1$, for either output state (set or reset).
- If $Q = 1$ ("Set") and $J = K = 1$, output of the OR = 0, so the ff will reset.
- Likewise, if $Q = 0$ ("Reset") and $J = K = 1$, OR = 1, and the ff will be set.
- (For $J=1$ and $K=0$, or $J=0$ and $K=1$) normal set or reset occurs.)
- Then for $J = K = 1$, when the clock ticks $Q \rightarrow$ the opposite state.

T Flip-flop

- **Behavior**
 - **Has a single input T**
 - **For $T = 0$, no change to state**
 - **For $T = 1$, changes to opposite state**
- **Same as a J-K flip-flop with $J = K = T$**
- **Cannot be initialized to a known state using the T input**
 - **Reset (asynchronous or synchronous) essential**

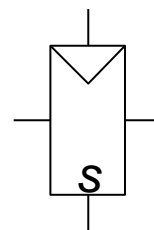
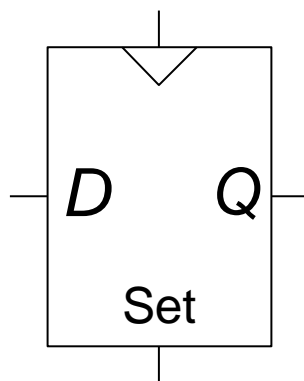
Flip-Flops

- D FF – ALU, registers, shift registers
- J-K FF – Control functions, status indicators
- T FF – Counters

Settable Flip-Flops

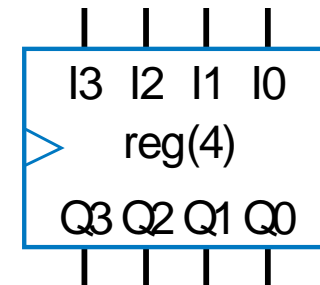
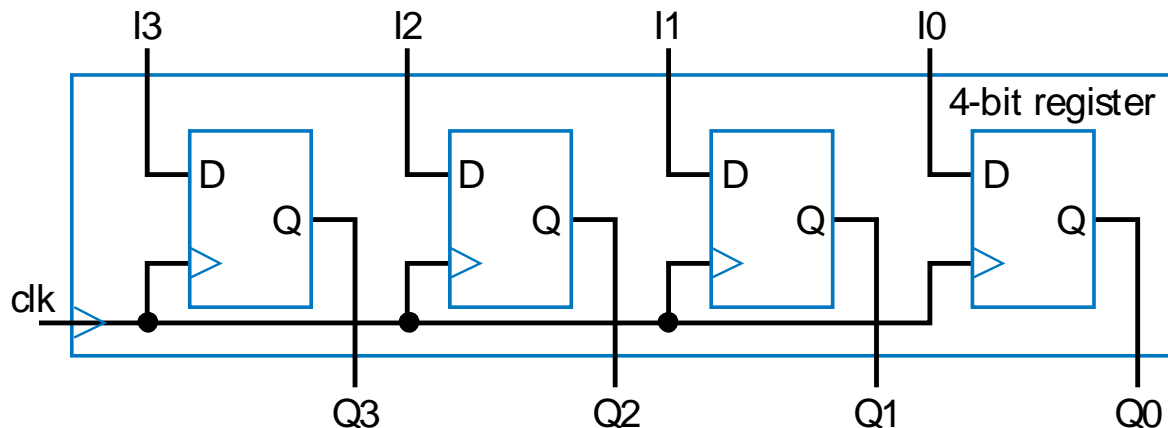
- Inputs: CLK , D , Set
- Function:
 - $Set = 1$: Q is set to 1
 - $Set = 0$: the flip-flop behaves as ordinary D flip-flop

Symbols



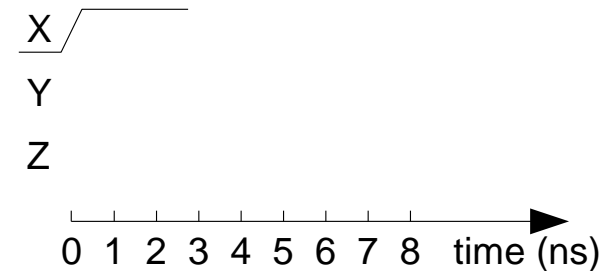
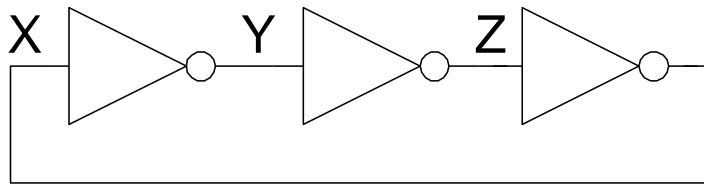
Basic Register

- 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



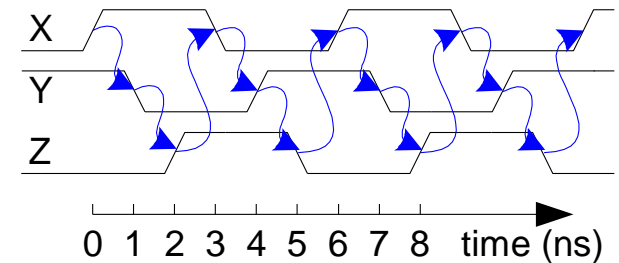
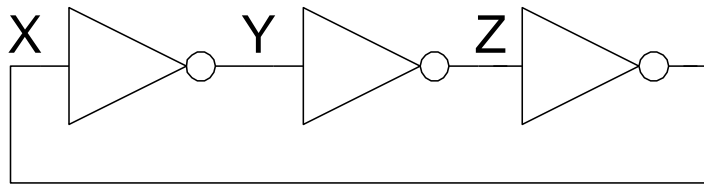
Sequential Logic

- Sequential circuits: all circuits that aren't combinational
- A problematic circuit:



Sequential Logic

- Sequential circuits: all circuits that aren't combinational
- A problematic circuit:



- No inputs and 1-3 outputs
- Astable circuit, oscillates
- Period depends on inverter delay
- It has a *cyclic path*: output fed back to input

Synchronous Sequential Logic Design

- Breaks cyclic paths by **inserting registers**
- Registers contain **state** of the system
- State changes at clock edge: system **synchronized** to the clock
- **Rules** of synchronous sequential circuit composition:
 - Every circuit element is either a register or a combinational circuit
 - At least one circuit element is a register
 - All registers receive the same clock signal
 - Every cyclic path contains at least one register
- Two common synchronous sequential circuits
 - Finite State Machines (FSMs)
 - Pipelines



Part 2

Finite State Machine

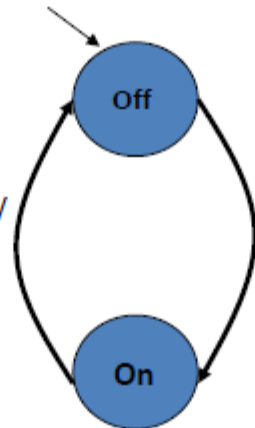
Finite State Machine (Adapted from various sources)

represented by arrows between the states

e.g. Light Switch



Switch down/
open circuit



Name of event
that causes
transition

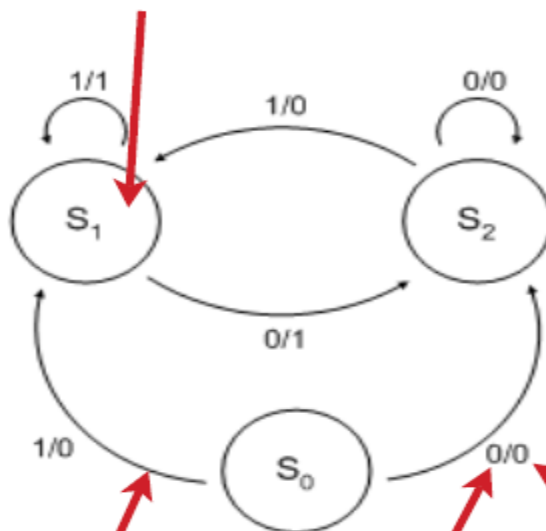
Switch up/
Close circuit

Action performed
when transition
occurs

- **Event:** Something that happens to the object **instantaneously**. It is often implemented as an operation on an object.
- **State:** Defines behavior and attribute values held by an object.
- **State transition:** A change of state (in response to an event).

State Machine

one or more states, indicated by nodes



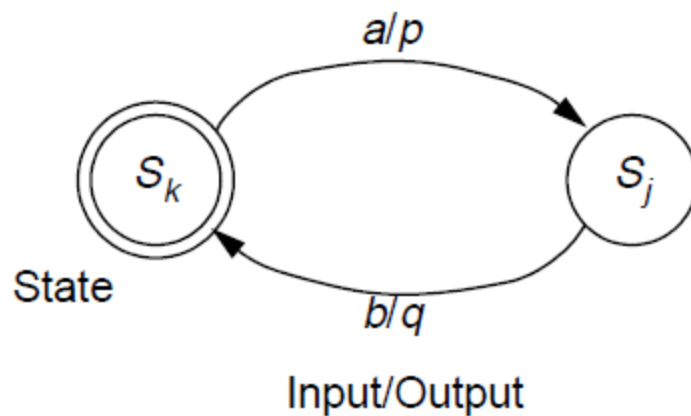
edges between states

machine output at transition

input value that triggers transition on edge

State Machine

- **Circles:** represent the machine states
 - Labelled with a binary encoded number or S_k reflecting state.
- **Directed arcs:** represent the transitions between states
 - Labelled with input/output for that state transition.



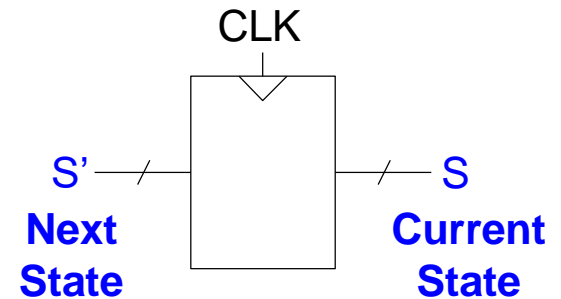
Input: $x(t) \in \{a, b\}$
 Output: $z(t) \in \{p, q\}$
 State: $s(t) \in \{S_k, S_j\}$
 Initial state: $s(0) = S_k$

Finite State Machine (FSM)

- Consists of:

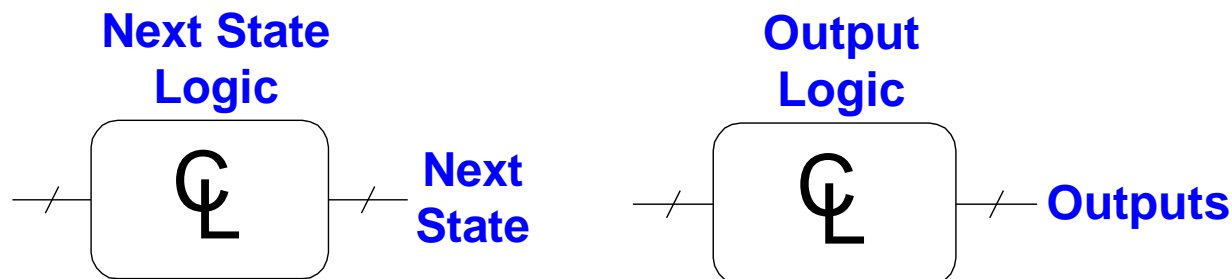
- State register**

- Stores current state
 - Loads next state at clock edge



- Combinational logic**

- Computes the next state
 - Computes the outputs



Moore and Mealy Models

- **Sequential Circuits or Sequential Machines are also called *Finite State Machines* (FSMs). Two formal models exist:**

- **Moore Model**

- **Named after E.F. Moore**
- **Outputs are a function ONLY of states**
- **Usually specified on the states.**

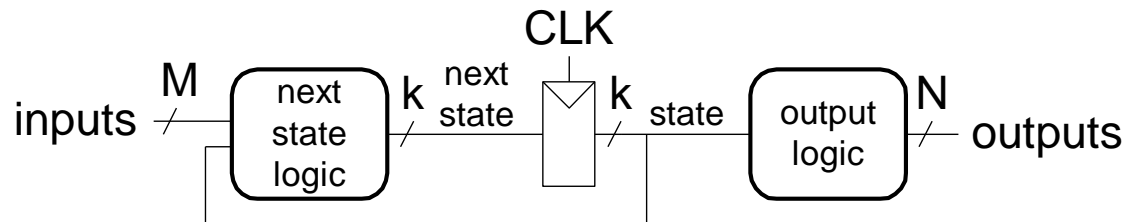
- **Mealy Model**

- **Named after G. Mealy**
- **Outputs are a function of inputs AND states**
- **Usually specified on the state transition arcs.**

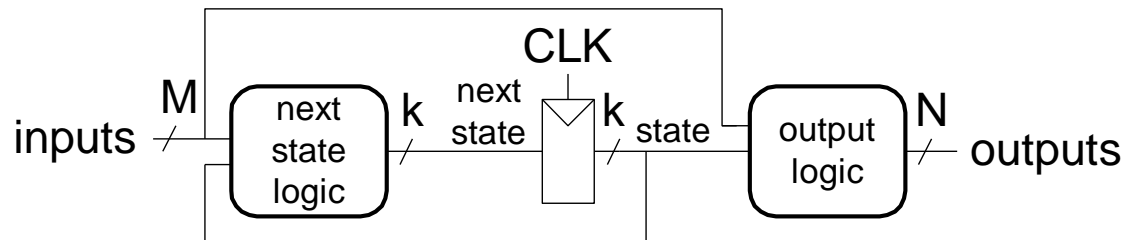
Finite State Machines (FSMs)

- Next state determined by current state and inputs
- Two types of finite state machines differ in output logic:
 - **Moore FSM:** outputs depend only on current state
 - **Mealy FSM:** outputs depend on current state *and* inputs

Moore FSM

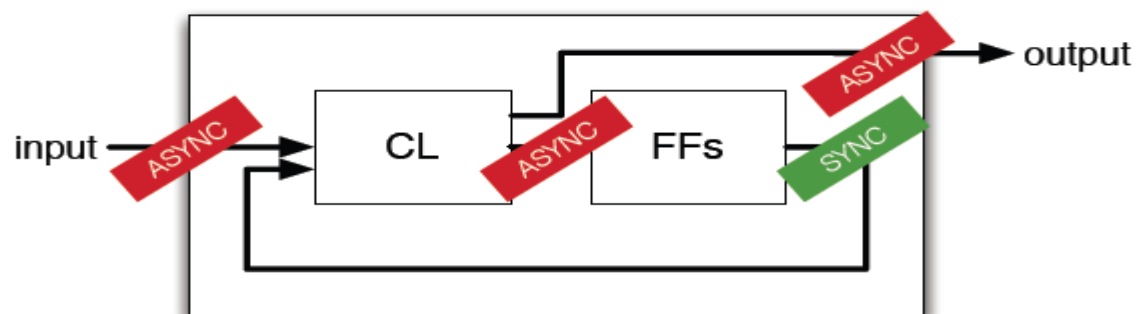


Mealy FSM

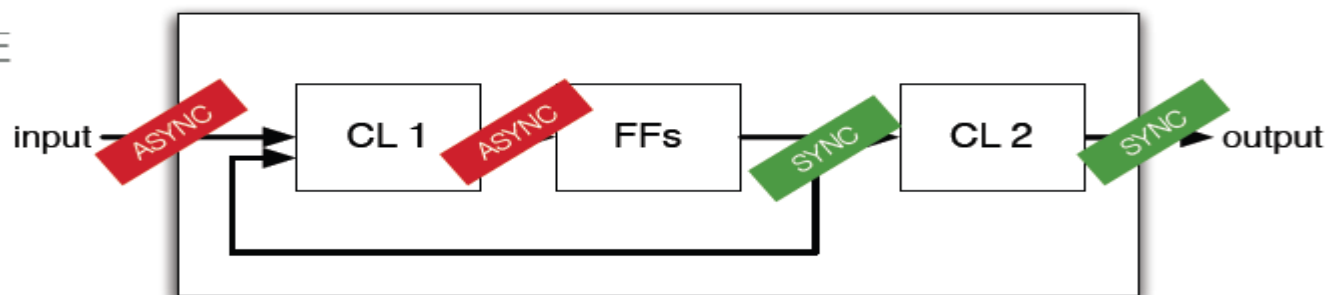


FSM Characteristics

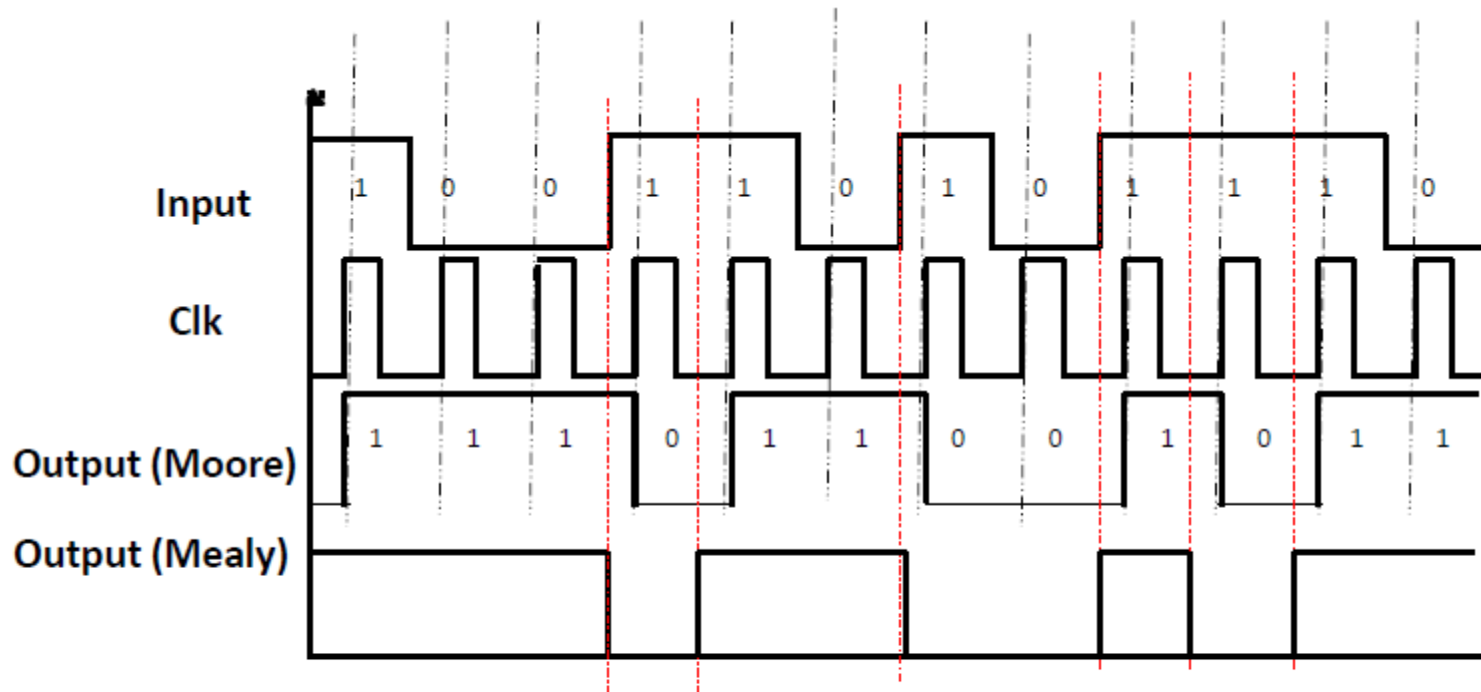
MEALY



MOORE



Moore and Mealy State Machine (Adapted from various sources)



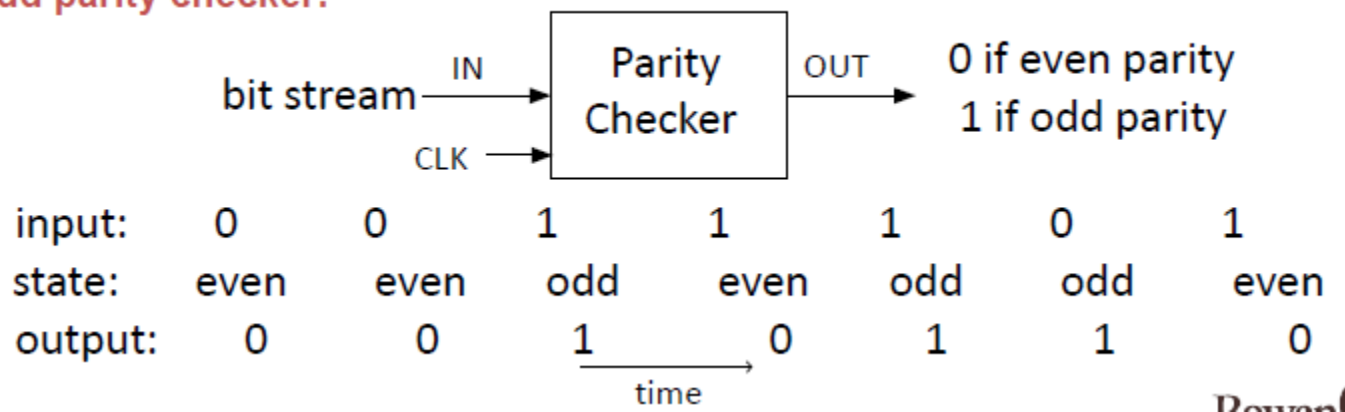
Moore: the output change is **synchronous** with the enabling clock edge.

Mealy: the output changes **asynchronously** with the enabling clock edge.

Finite State Machine – Parity Checker (Adapted from various sources)

Parity checker counts the number of 1's in a bit-serial input stream. If the checker asserts its output when the input stream contains an odd number of 1's, it is called an **odd parity checker**. If it asserts its output when it has seen an even number of 1's, it is an **even parity checker**.

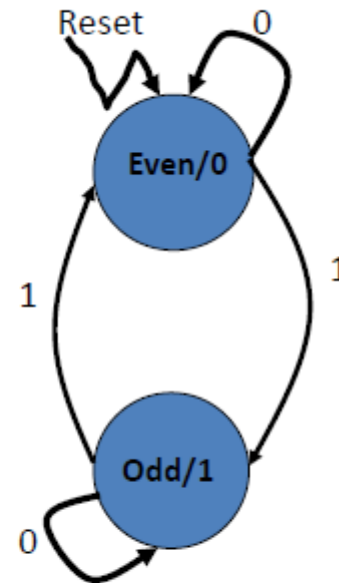
Odd parity checker:



Finite State Machine – Parity Checker (Adapted from various)

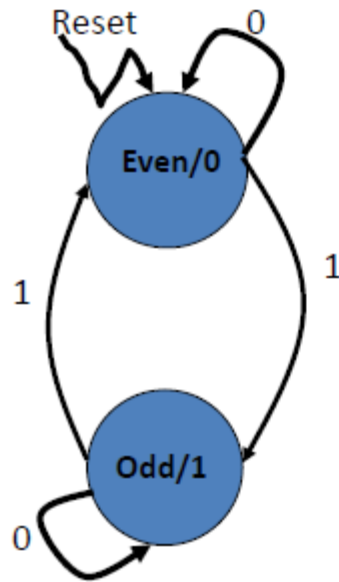
• State Diagram

- States: parity checker is in one of two states (even or odd).
- Initial state: even.
- Inputs: 0 or 1. Inputs cause state transitions.
- Outputs: depends on which state the parity checker is in. 0 for even and 1 for odd. (Moore machine: output is associate with state)



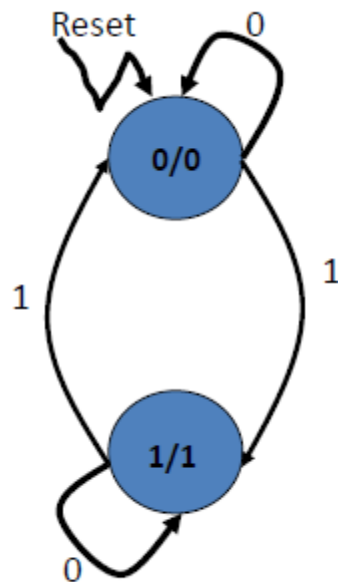
Finite State Machine – Parity Checker (Adapted from various)

FSM may be presented by state transition table.



Present State	Input	Next State	Output
Even	0	Even	0
Even	1	Odd	0
Odd	0	Odd	1
Odd	1	Even	1

Finite State Machine – Parity Checker (Adapted from various)



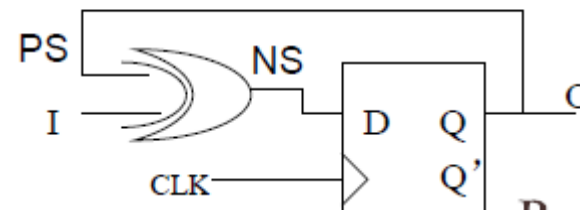
Binary coding: Even (0); Odd (1)

Present State (PS)	Input (I)	Next State (NS)	Output (O)
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

$$NS = PS \text{ XOR } I$$

(exclusive disjunction of Present State and Input)

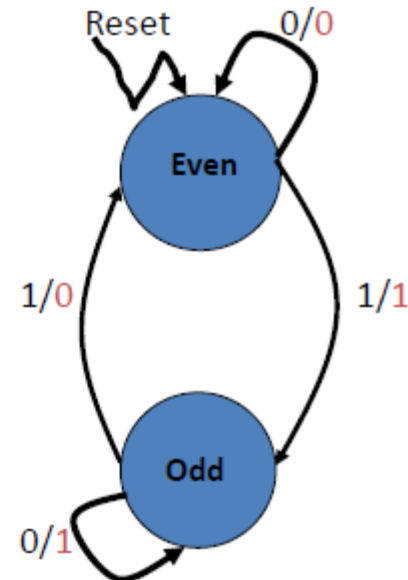
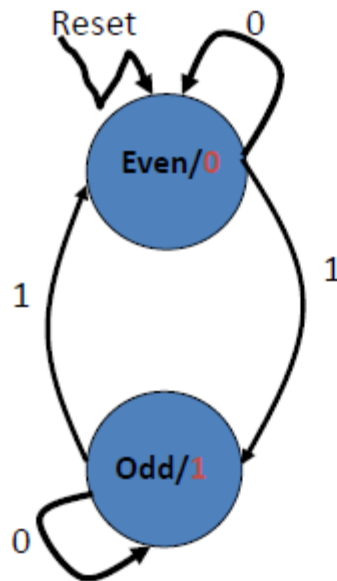
Implement the parity checker using D-flip flop:



Moore and Mealy State Machine (Adapted from various sources)

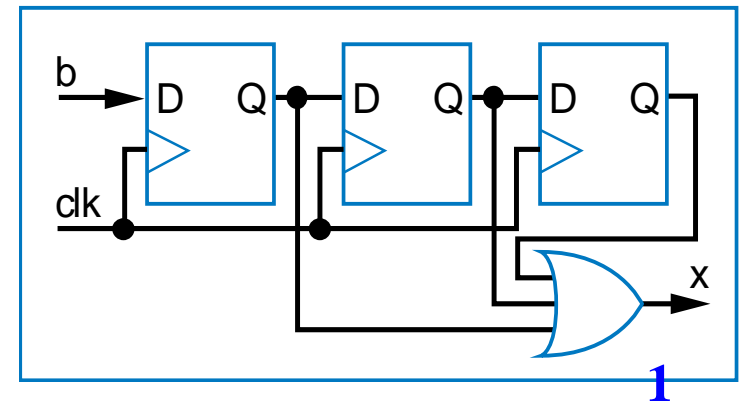
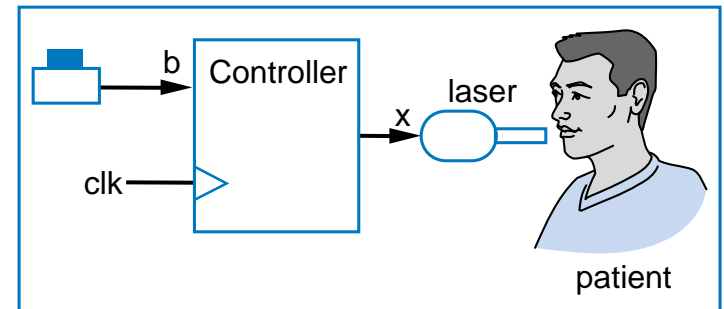
Moore Machine: Output is associated with the state and hence appears after the state transition takes place.

Mealy Machine: Output is associated with the state transition, and appears before the state transition is completed (by the next clock pulse).



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*

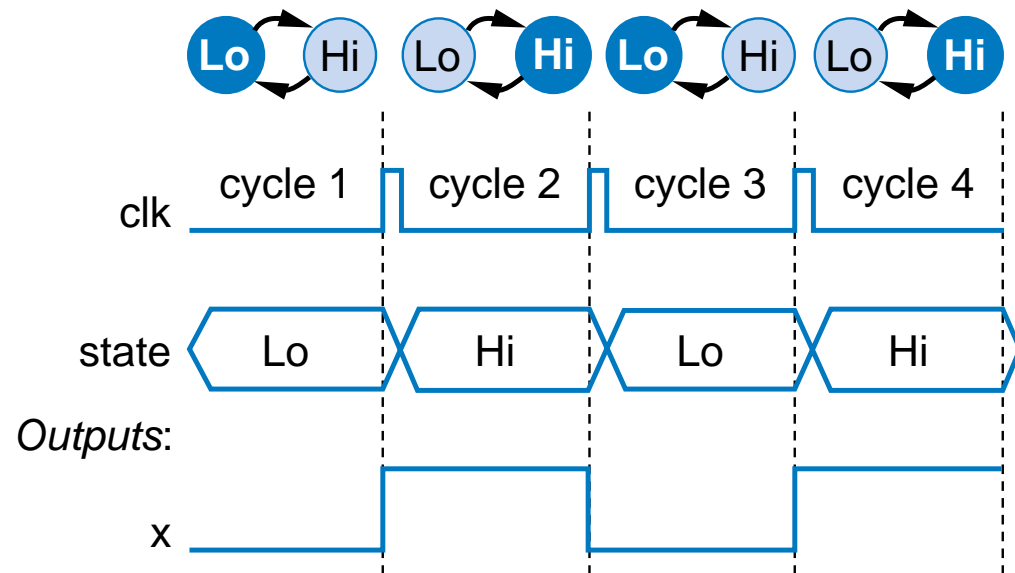
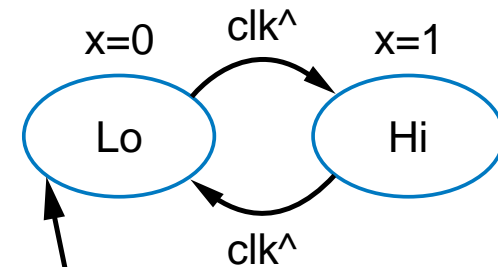
Step	Description
Step 1: Capture behavior	Capture the function Create a truth table or equations, whichever is most natural for the given problem , to describe the desired behavior of each output of the combinational logic.
Step 2: Convert to circuit	2A: Create equations This substep is only necessary if you captured the function using a truth table instead 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.)



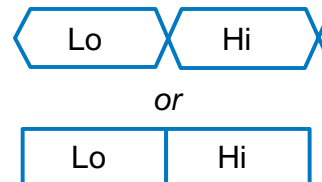
Capturing Sequential Circuit Behavior as FSM

- 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^\wedge)
 - Arrow points to initial state (when circuit first starts)

Outputs: x



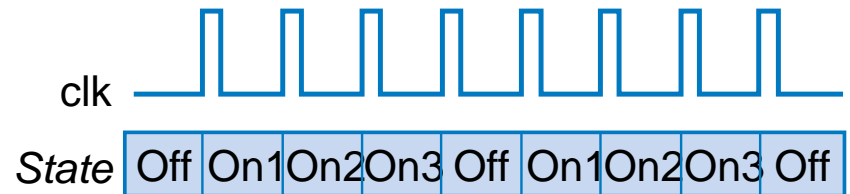
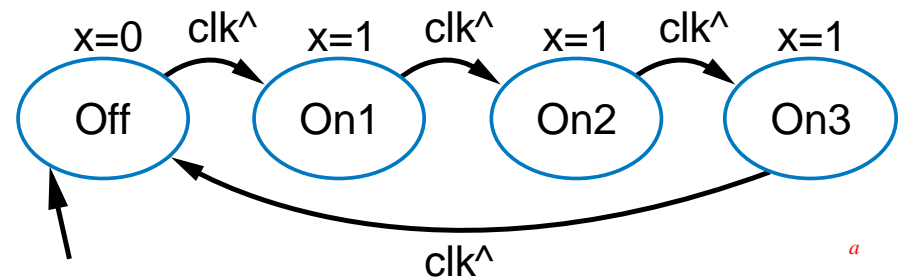
Depicting multi-bit or other info in a timing diagram



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

Outputs: x

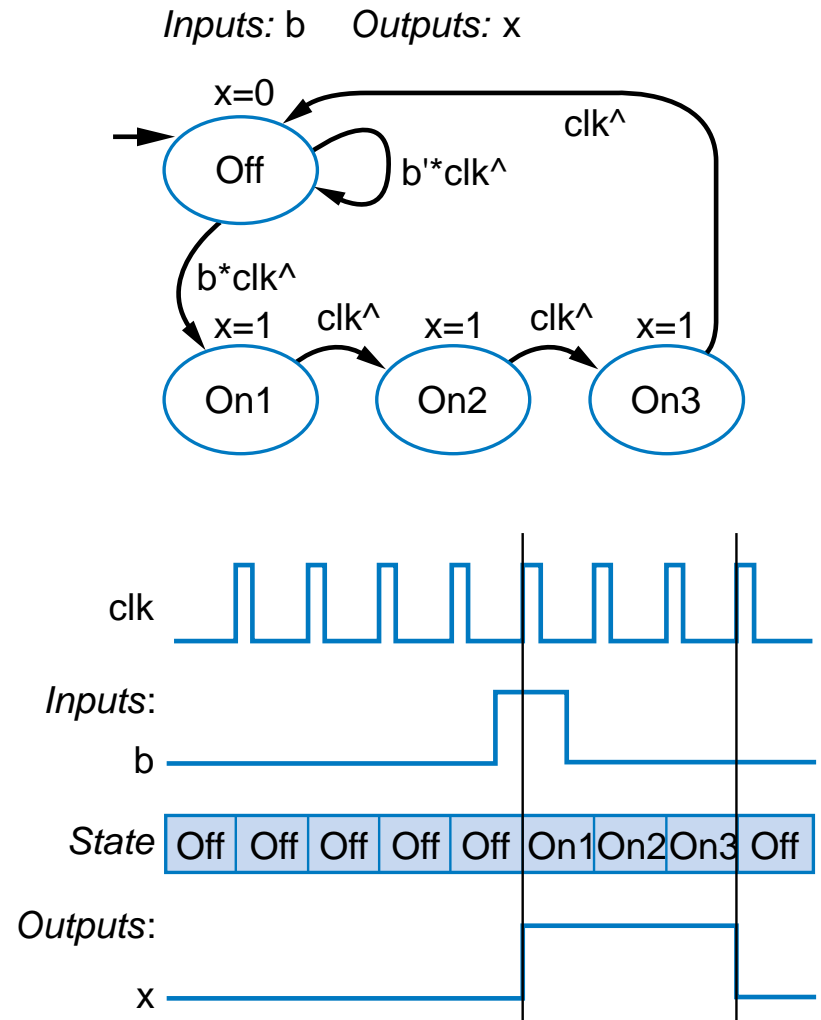


Outputs:



Three-Cycles High System with Button Input

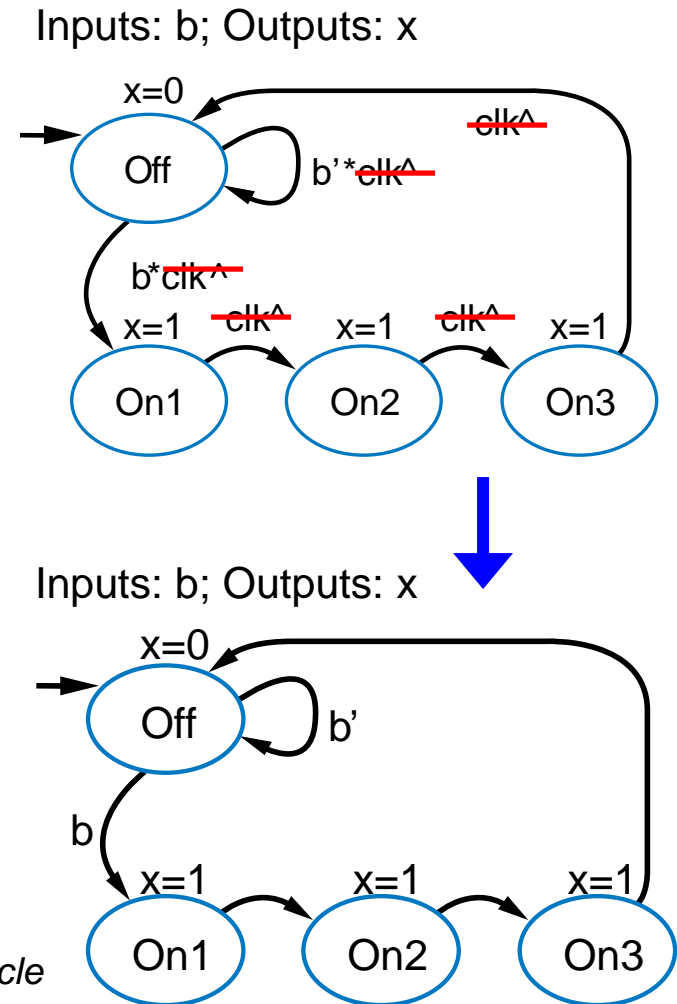
- Four states
- Wait in “Off” while b is 0 ($b' \cdot \text{clk}^\wedge$)
- When b is 1 ($b \cdot \text{clk}^\wedge$), transition to On1
 - Sets $x=1$
 - Next two clock edges, transition to On2, then On3
- So $x=1$ for three cycles after button pressed



FSM Simplification: Rising Clock Edges Implicit

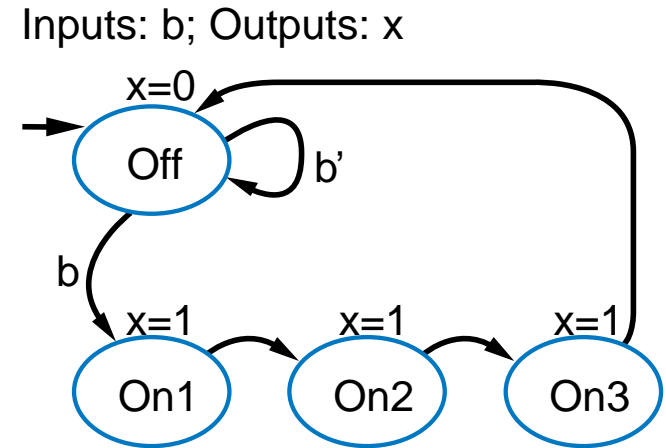
- Every edge ANDed with rising clock edge
- What if we wanted a transition *without* a rising edge
 - We don't consider such asynchronous FSMs – less common, and advanced topic
 - Only consider **synchronous** FSMs – rising edge on *every* transition

Note: Transition with no associated condition thus transitions 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 $x=1$



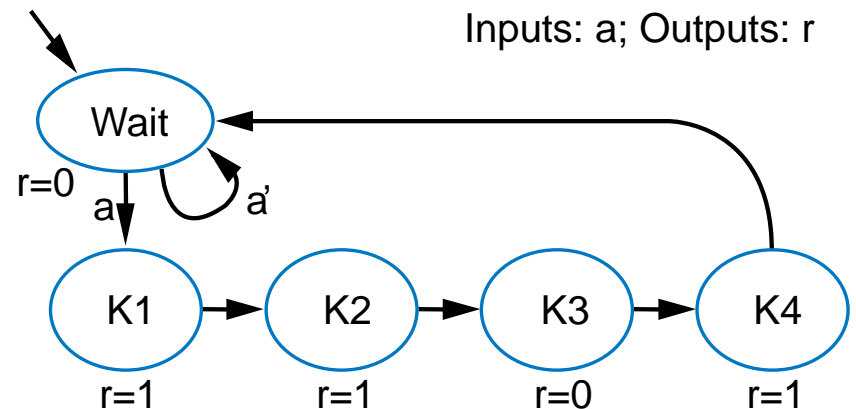
We often draw FSM graphically, known as **state diagram**

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



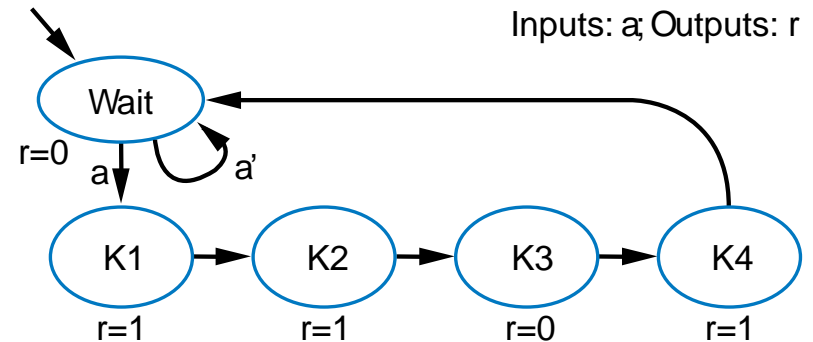
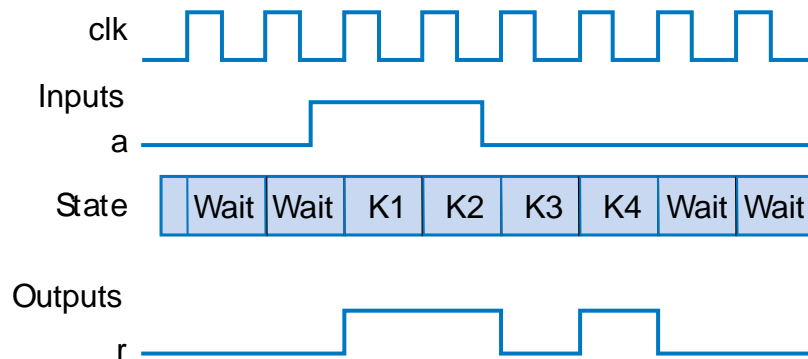
FSM Example: Secure Car Key

- 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)

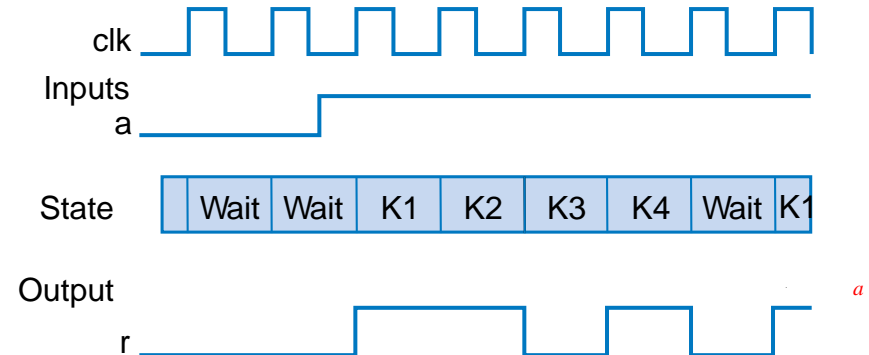


FSM Example: Secure Car Key (cont.)

- 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:



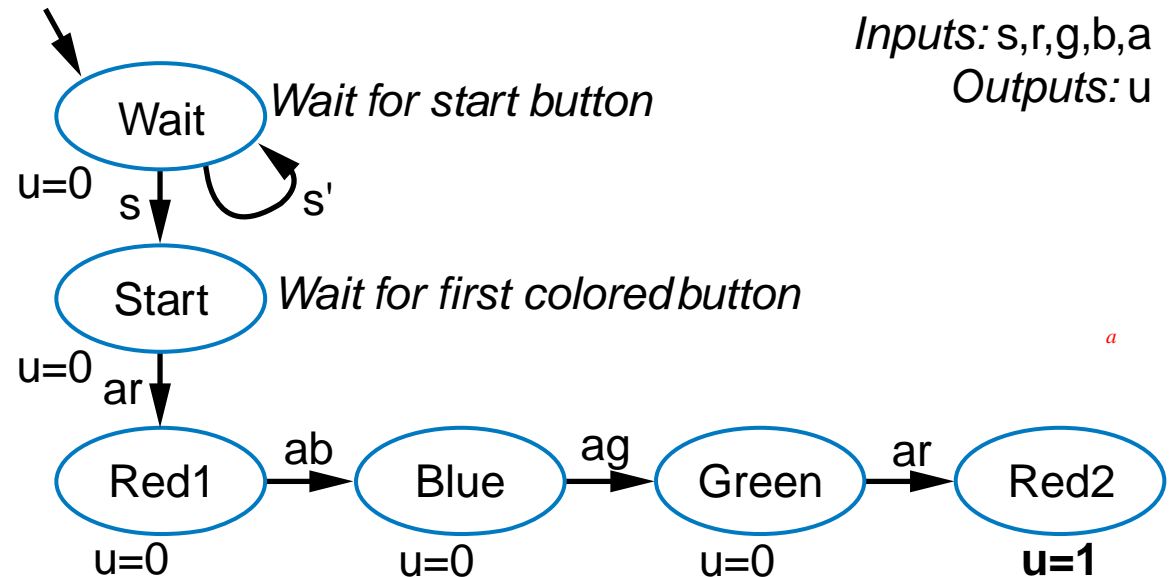
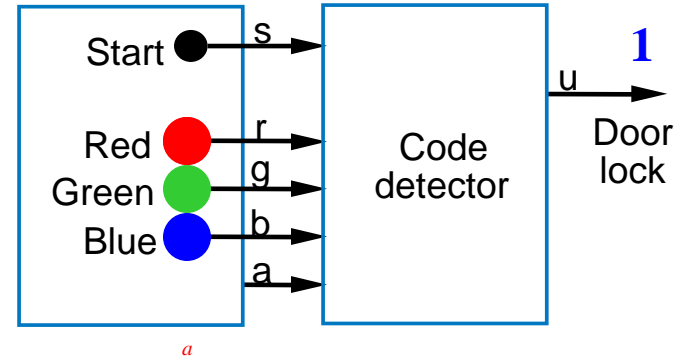
How To Capture Desired Behavior as FSM

- *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.



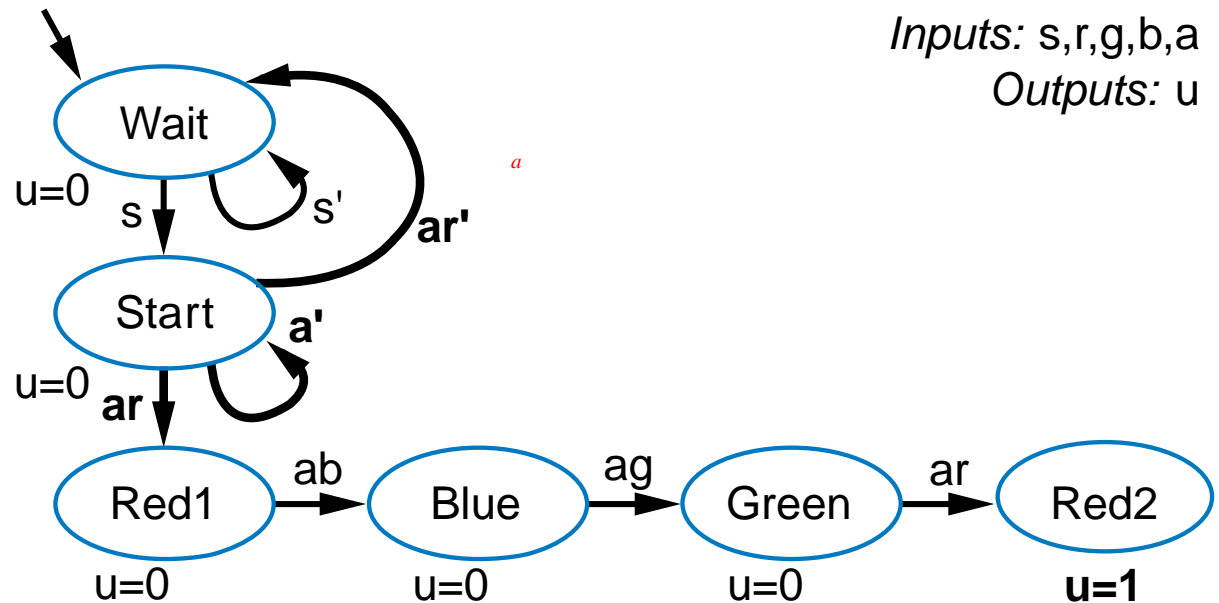
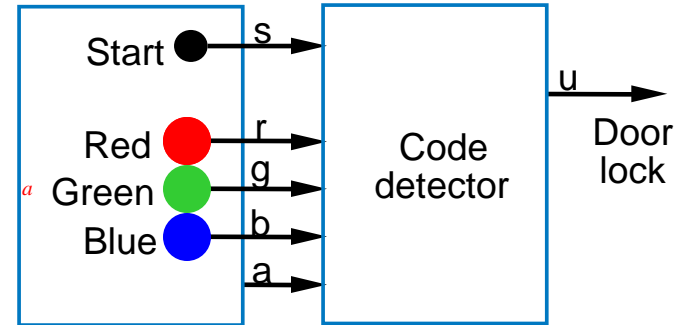
FSM Capture Example: Code Detector

- 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

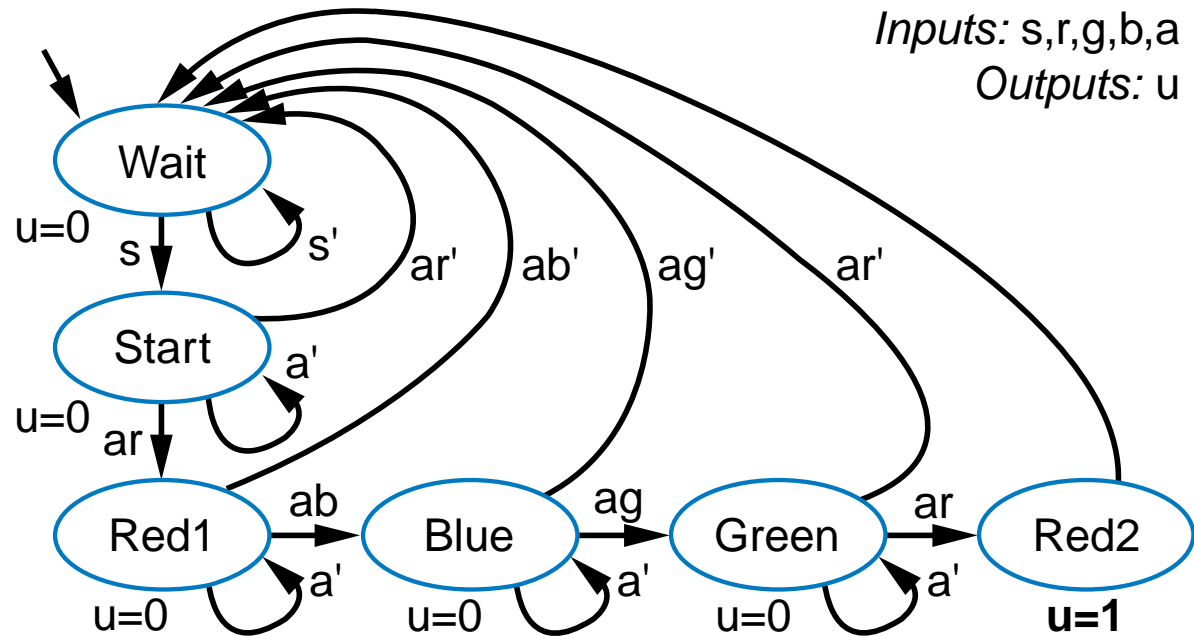
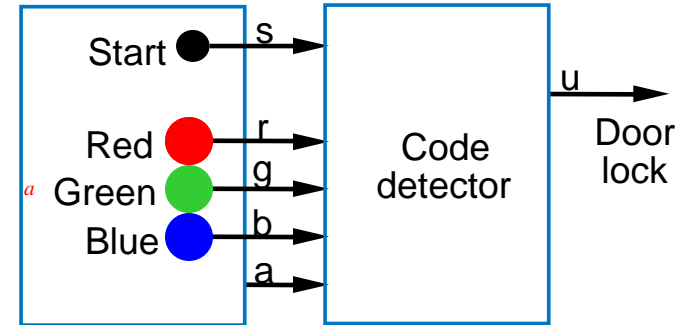
- Capture as FSM
 - *List states*
 - **Create transitions**



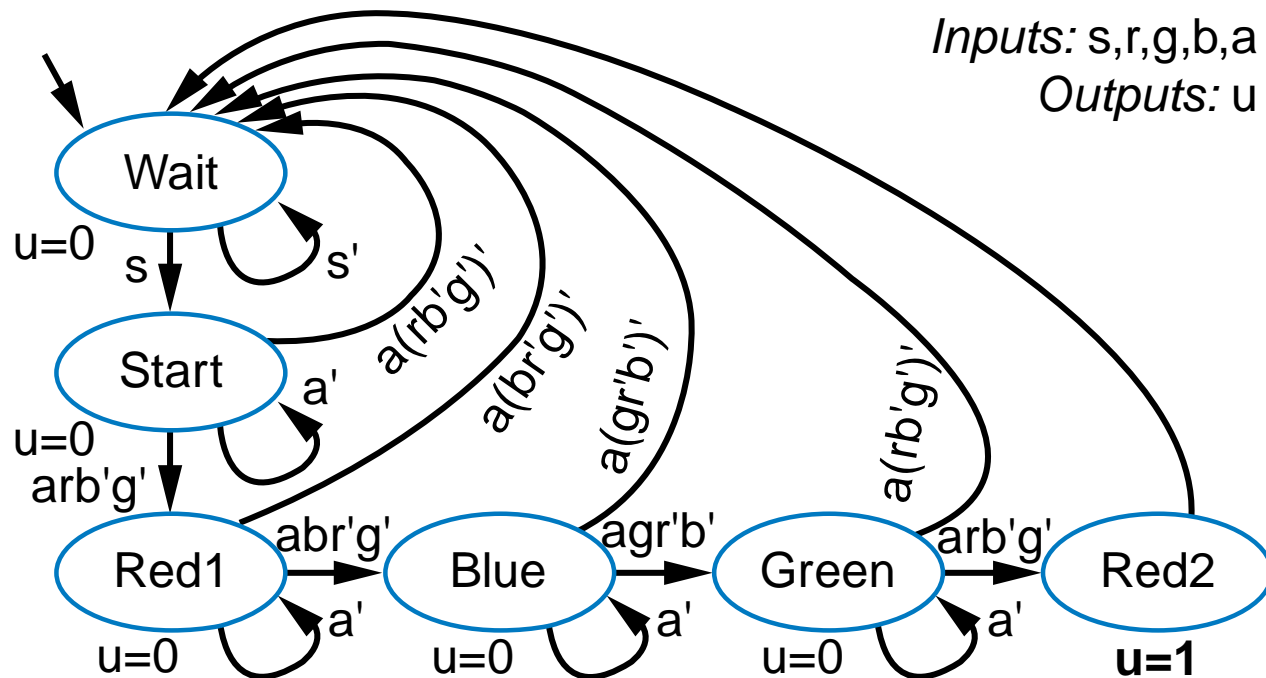
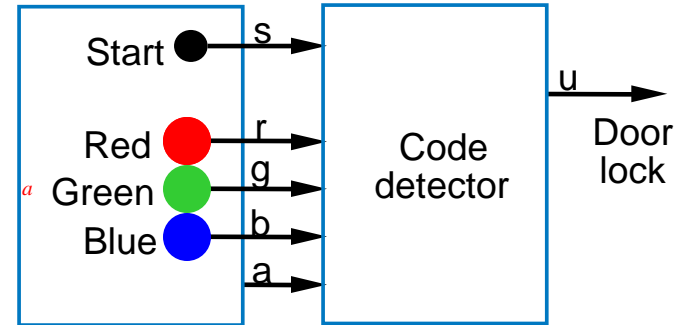
FSM Capture Example: Code Detector

- 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

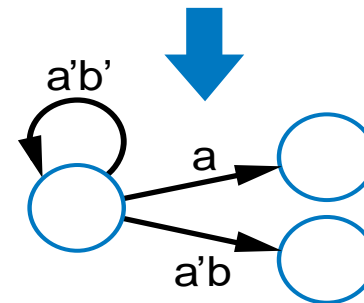
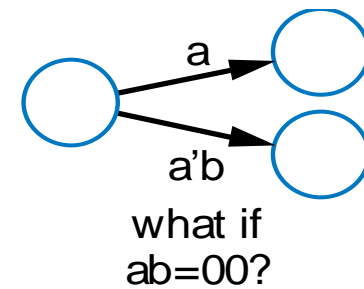
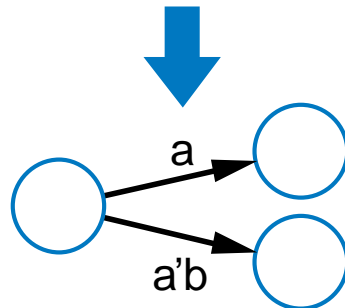
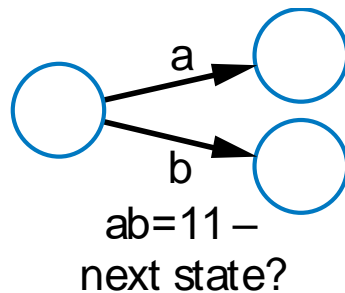


FSM Capture Example: Code Detector



Common Mistakes when Capturing FSMs

- Non-exclusive transitions
- Incomplete transitions



FSM Design Procedure

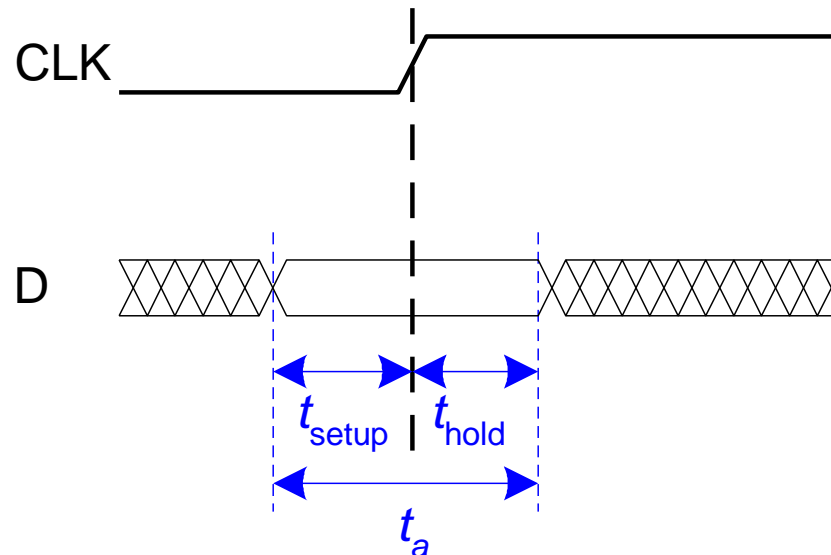
1. Identify inputs and outputs
2. Sketch state transition diagram
3. Write state transition table
4. Select state encodings
5. For Moore machine:
 1. Rewrite state transition table with state encodings
 2. Write output table
6. For a Mealy machine:
 1. Rewrite combined state transition and output table with state encodings
7. Write Boolean equations for next state and output logic
8. Sketch the circuit schematic

Timing

- Flip-flop samples D at clock edge
- D must be stable when sampled
- Similar to a photograph, D must be stable around clock edge
- If not, metastability can occur

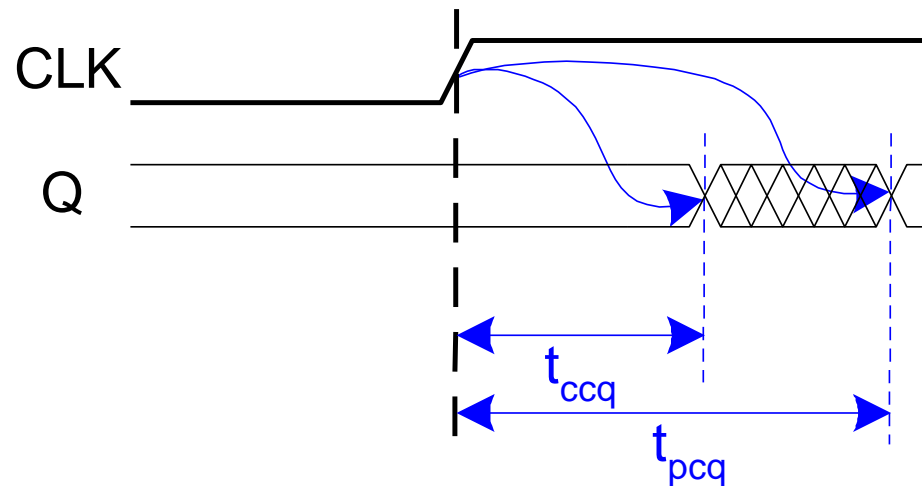
Input Timing Constraints

- **Setup time:** t_{setup} = time *before* clock edge data must be stable (i.e. not changing)
- **Hold time:** t_{hold} = time *after* clock edge data must be stable
- **Aperture time:** t_a = time *around* clock edge data must be stable ($t_a = t_{\text{setup}} + t_{\text{hold}}$)



Output Timing Constraints

- **Propagation delay:** t_{pcq} = time after clock edge that the output Q is guaranteed to be stable (i.e., to stop changing)
- **Contamination delay:** t_{ccq} = time after clock edge that Q might be unstable (i.e., start changing)



Dynamic Discipline

- Synchronous sequential circuit inputs must be stable during aperture (setup and hold) time around clock edge
- Specifically, inputs must be stable
 - at least t_{setup} before the clock edge
 - at least until t_{hold} after the clock edge

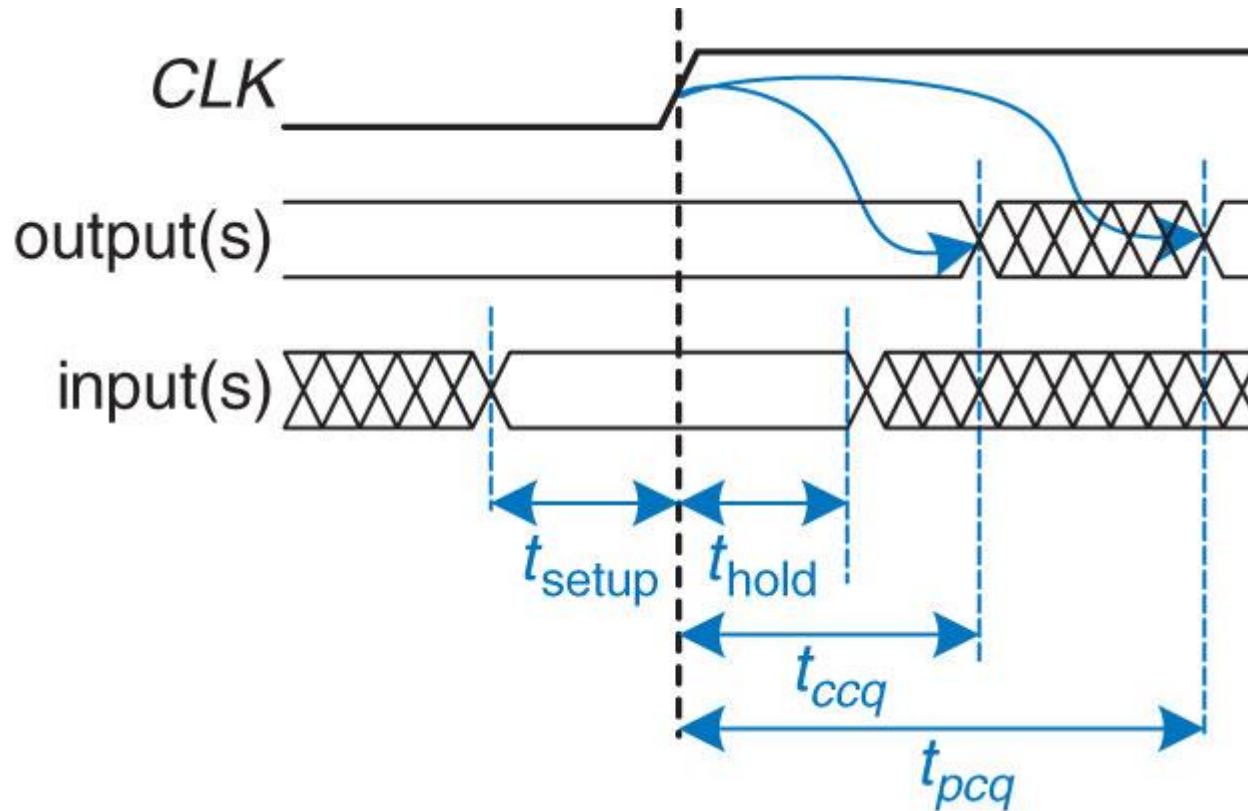
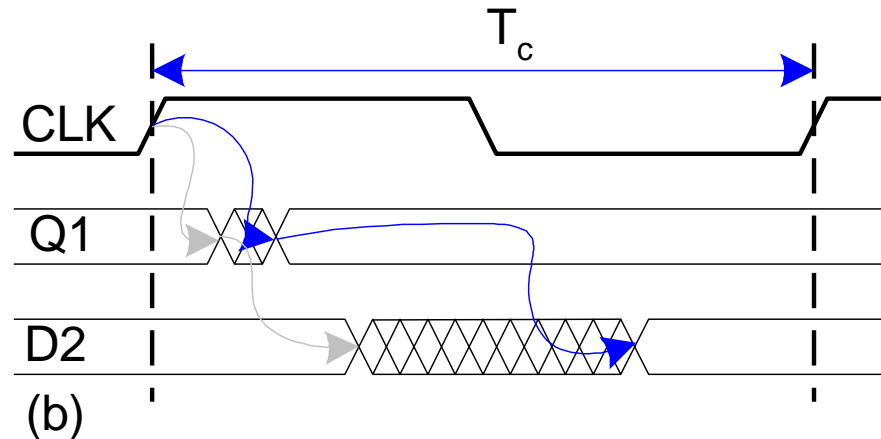
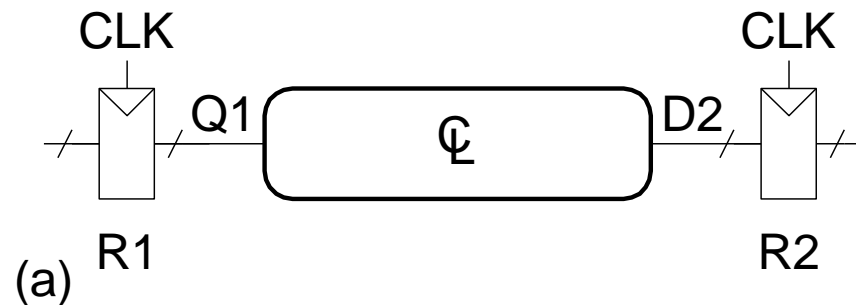


Figure 3.37 Timing specification for synchronous sequential circuit

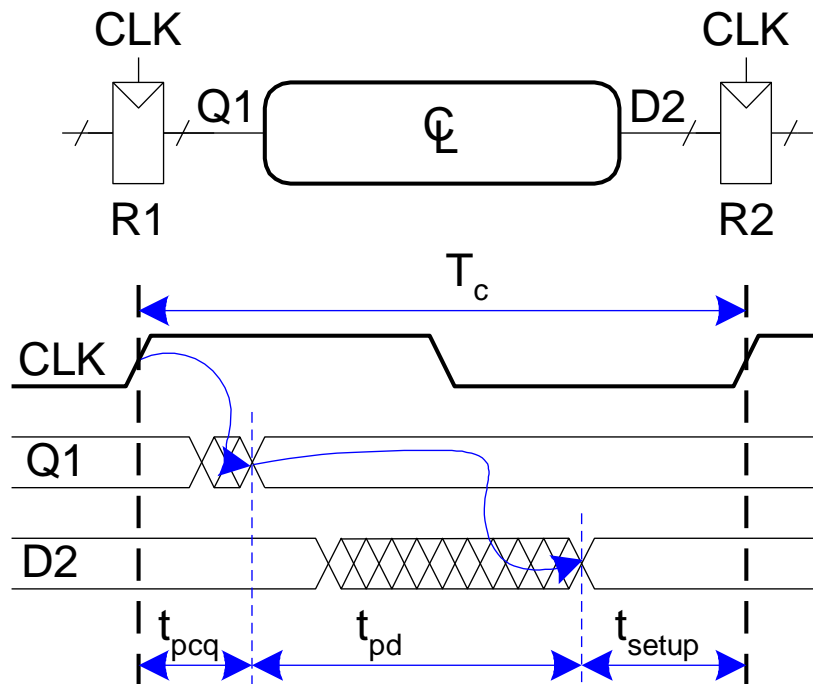
Dynamic Discipline

- The delay between registers has a **minimum** and **maximum** delay, dependent on the delays of the circuit elements



Setup Time Constraint

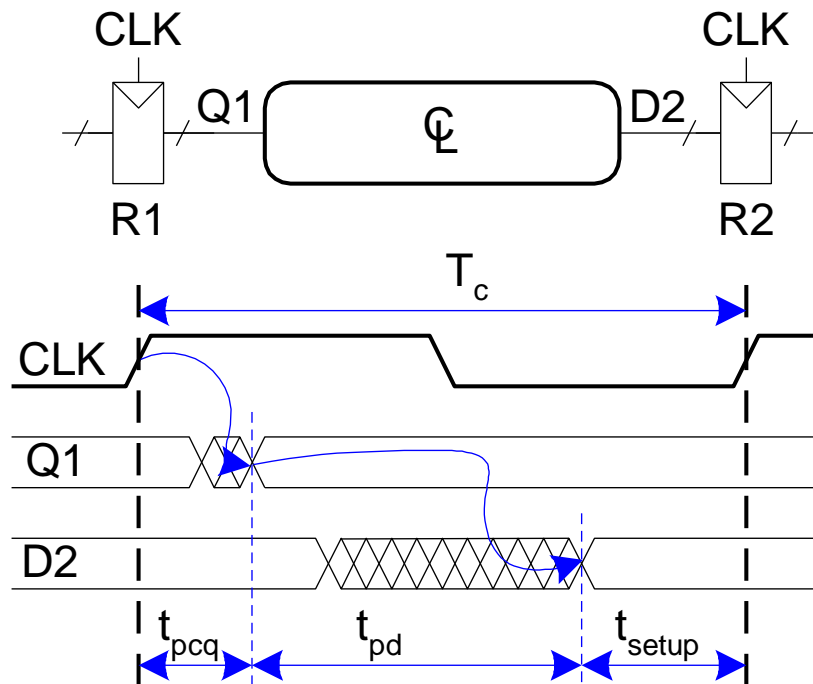
- Depends on the **maximum** delay from register R1 through combinational logic to R2
- The input to register R2 must be stable at least t_{setup} before clock edge



$$T_c \geq$$

Setup Time Constraint

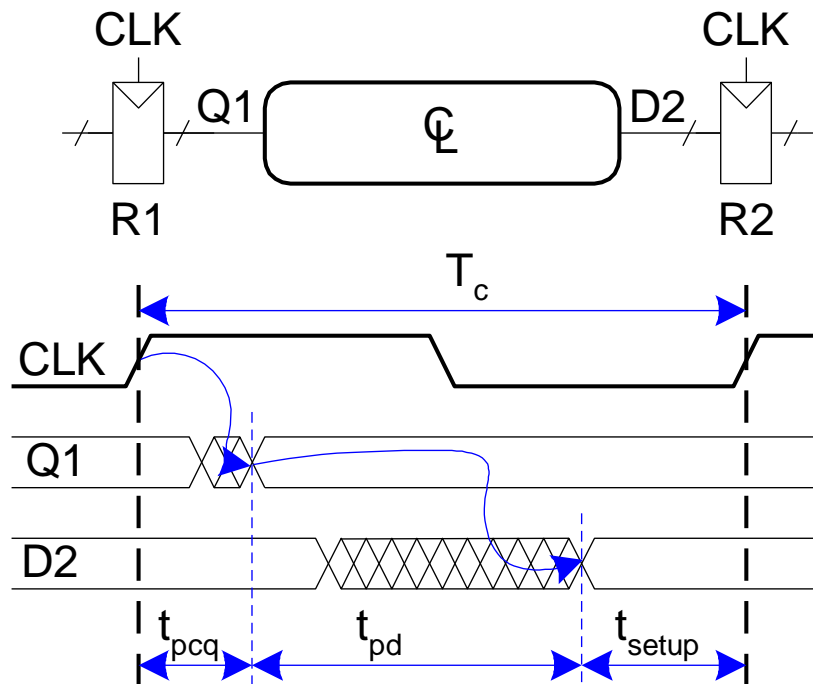
- Depends on the **maximum** delay from register R1 through combinational logic to R2
- The input to register R2 must be stable at least t_{setup} before clock edge



$$T_c \geq t_{pcq} + t_{pd} + t_{\text{setup}}$$
$$t_{pd} \leq$$

Setup Time Constraint

- Depends on the **maximum** delay from register R1 through combinational logic to R2
- The input to register R2 must be stable at least t_{setup} before clock edge

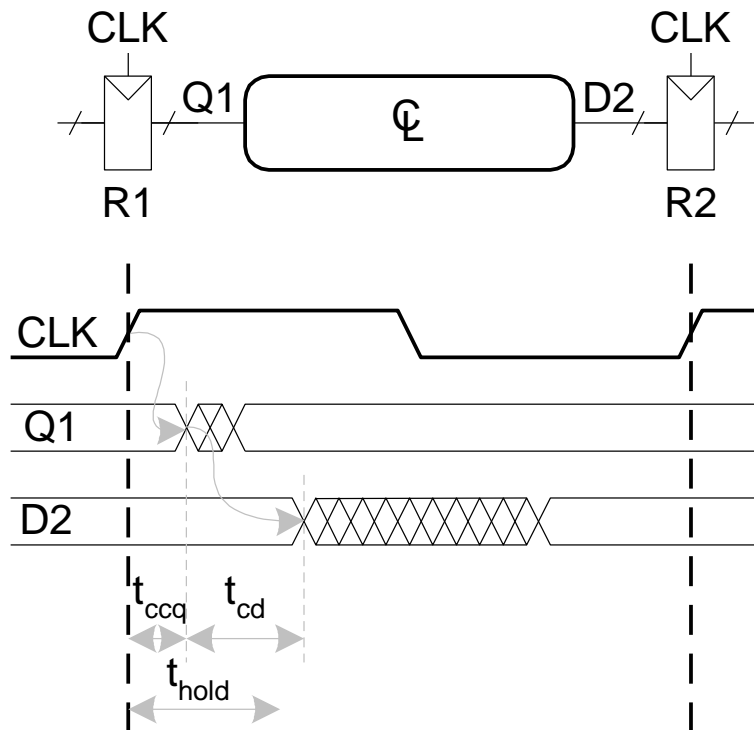


$$T_c \geq t_{pcq} + t_{pd} + t_{\text{setup}}$$
$$t_{pd} \leq T_c - (t_{pcq} + t_{\text{setup}})$$

$(t_{pcq} + t_{\text{setup}})$: sequencing overhead

Hold Time Constraint

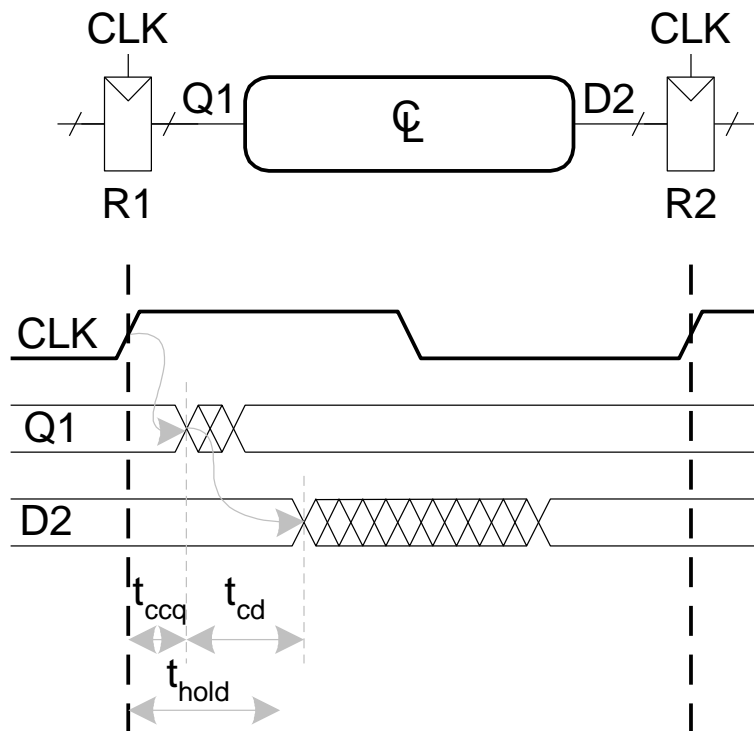
- Depends on the **minimum** delay from register R1 through the combinational logic to R2
- The input to register R2 must be stable for at least t_{hold} after the clock edge



$$t_{\text{hold}} <$$

Hold Time Constraint

- Depends on the **minimum** delay from register R1 through the combinational logic to R2
- The input to register R2 must be stable for at least t_{hold} after the clock edge

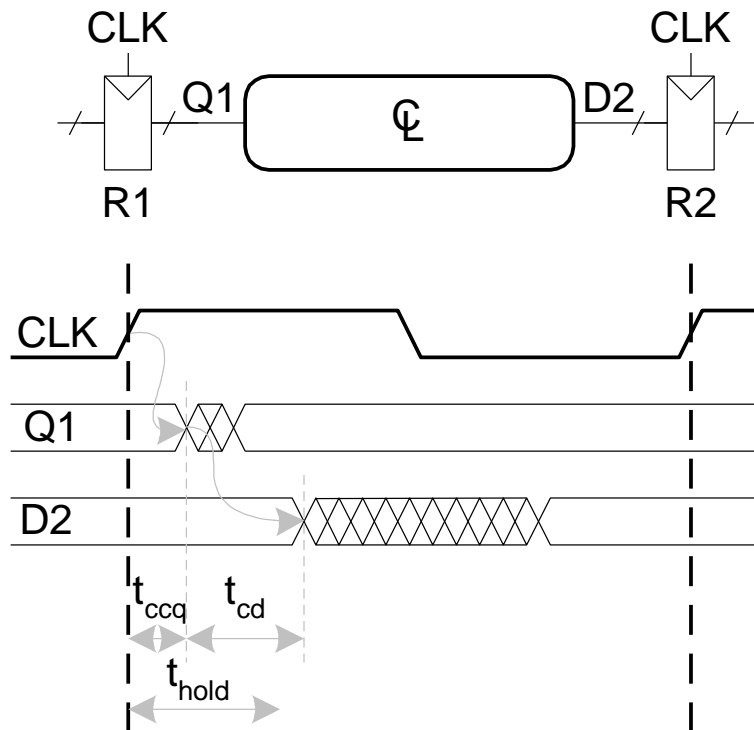


$$t_{\text{hold}} < t_{\text{ccq}} + t_{\text{cd}}$$

$$t_{\text{cd}} >$$

Hold Time Constraint

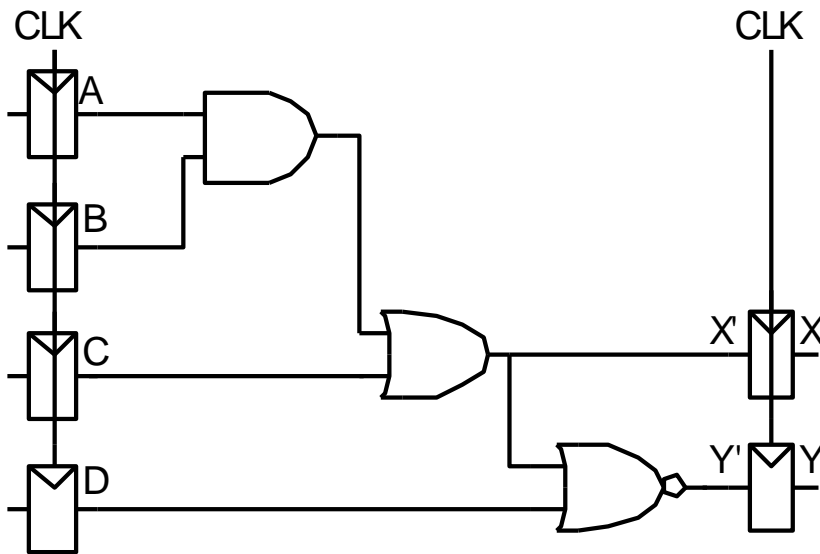
- Depends on the **minimum** delay from register R1 through the combinational logic to R2
- The input to register R2 must be stable for at least t_{hold} after the clock edge



$$t_{\text{hold}} < t_{ccq} + t_{cd}$$

$$t_{cd} > t_{\text{hold}} - t_{ccq}$$

Timing Analysis



$$t_{pd} =$$

$$t_{cd} =$$

Setup time constraint:

$$T_c \geq$$

$$f_c =$$

Timing Characteristics

$$t_{ccq} = 30 \text{ ps}$$

$$t_{pcq} = 50 \text{ ps}$$

$$t_{\text{setup}} = 60 \text{ ps}$$

$$t_{\text{hold}} = 70 \text{ ps}$$

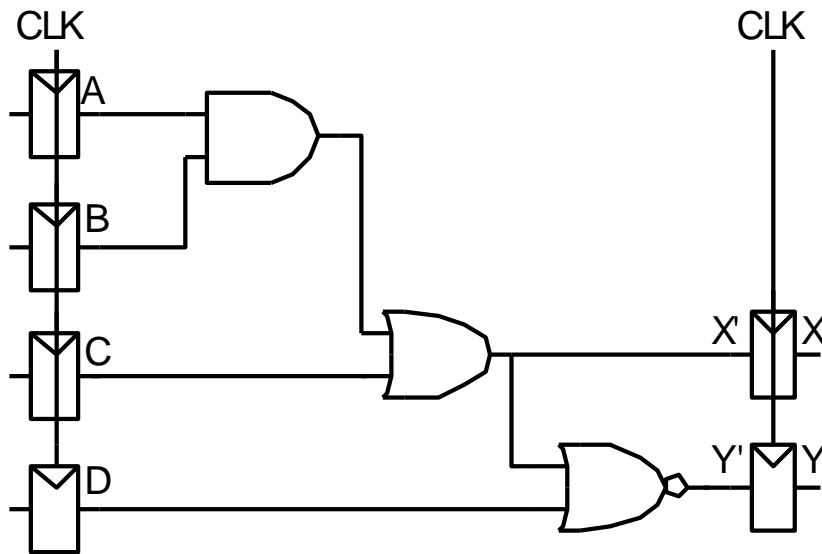
per gate

$$\left[\begin{array}{l} t_{pd} = 35 \text{ ps} \\ t_{cd} = 25 \text{ ps} \end{array} \right.$$

Hold time constraint:

$$t_{ccq} + t_{cd} > t_{\text{hold}} ?$$

Timing Analysis



$$t_{pd} = 3 \times 35 \text{ ps} = 105 \text{ ps}$$

$$t_{cd} = 25 \text{ ps}$$

Setup time constraint:

$$T_c \geq (50 + 105 + 60) \text{ ps} = 215 \text{ ps}$$

$$f_c = 1/T_c = 4.65 \text{ GHz}$$

Timing Characteristics

$$t_{ccq} = 30 \text{ ps}$$

$$t_{pcq} = 50 \text{ ps}$$

$$t_{\text{setup}} = 60 \text{ ps}$$

$$t_{\text{hold}} = 70 \text{ ps}$$

per gate

$$\left[\begin{array}{l} t_{pd} = 35 \text{ ps} \\ t_{cd} = 25 \text{ ps} \end{array} \right.$$

Hold time constraint:

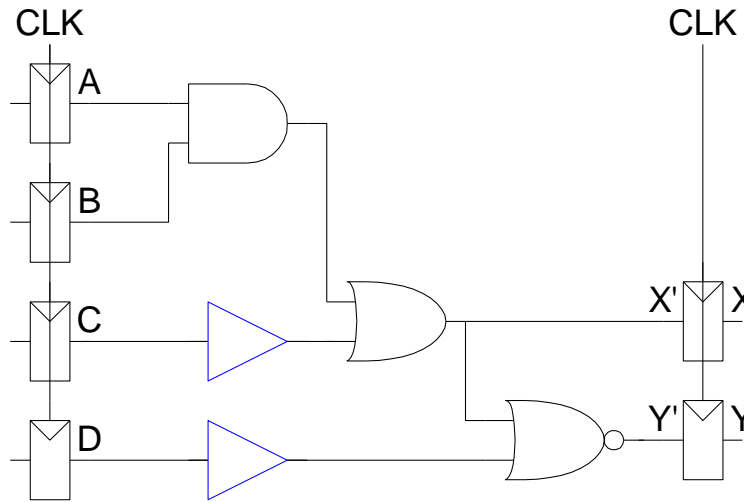
$$t_{ccq} + t_{cd} > t_{\text{hold}} ?$$

$$(30 + 25) \text{ ps} > 70 \text{ ps} ? \text{ No!}$$



Timing Analysis

Add buffers to the short paths:



$$t_{pd} =$$

$$t_{cd} =$$

Setup time constraint:

$$T_c \geq$$

$$f_c =$$

Timing Characteristics

$$t_{ccq} = 30 \text{ ps}$$

$$t_{pcq} = 50 \text{ ps}$$

$$t_{\text{setup}} = 60 \text{ ps}$$

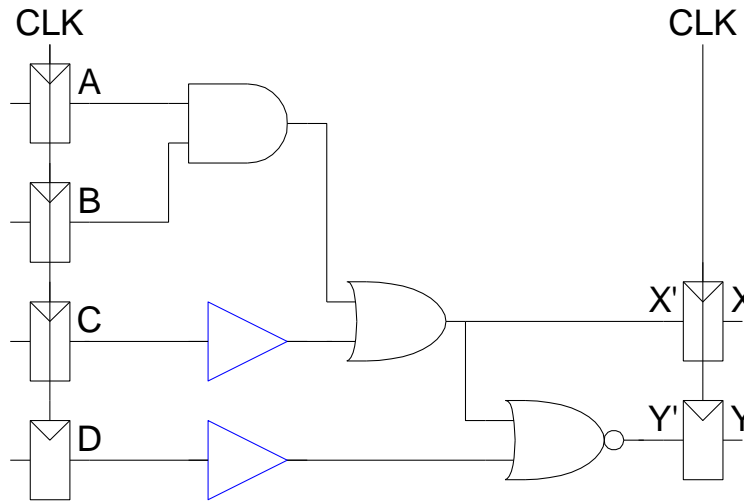
$$t_{\text{hold}} = 70 \text{ ps}$$

$$\text{per gate} \left[\begin{array}{l} t_{pd} = 35 \text{ ps} \\ t_{cd} = 25 \text{ ps} \end{array} \right.$$

Hold time constraint:

$$t_{ccq} + t_{cd} > t_{\text{hold}} ?$$

Add buffers to the short paths:



$$t_{pd} = 3 \times 35 \text{ ps} = 105 \text{ ps}$$

$$t_{cd} = 2 \times 25 \text{ ps} = 50 \text{ ps}$$

Setup time constraint:

$$T_c \geq (50 + 105 + 60) \text{ ps} = 215 \text{ ps}$$

$$f_c = 1/T_c = 4.65 \text{ GHz}$$

Timing Characteristics

$$t_{ccq} = 30 \text{ ps}$$

$$t_{pcq} = 50 \text{ ps}$$

$$t_{\text{setup}} = 60 \text{ ps}$$

$$t_{\text{hold}} = 70 \text{ ps}$$

$$\text{per gate} \left[\begin{array}{ll} t_{pd} & = 35 \text{ ps} \\ t_{cd} & = 25 \text{ ps} \end{array} \right.$$

Hold time constraint:

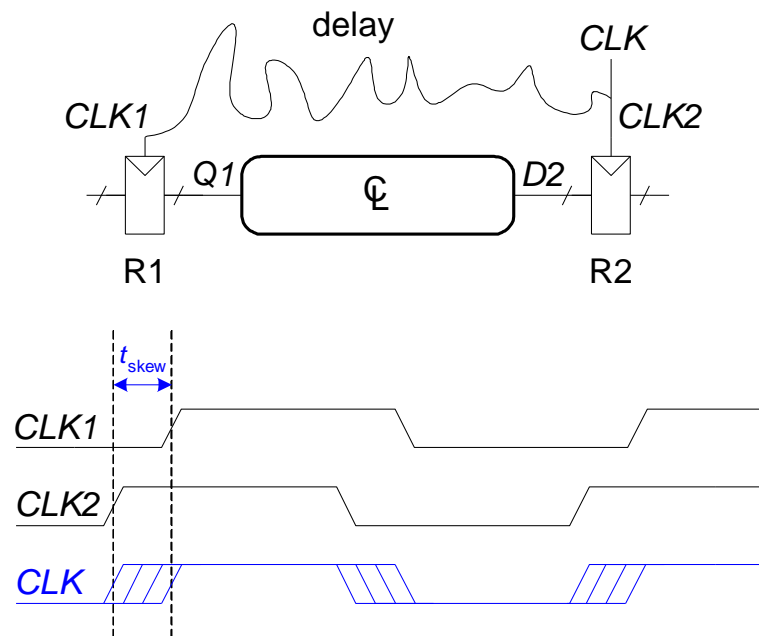
$$t_{ccq} + t_{cd} > t_{hold} ?$$

$(30 + 50) \text{ ps} > 70 \text{ ps}$? **Yes!**



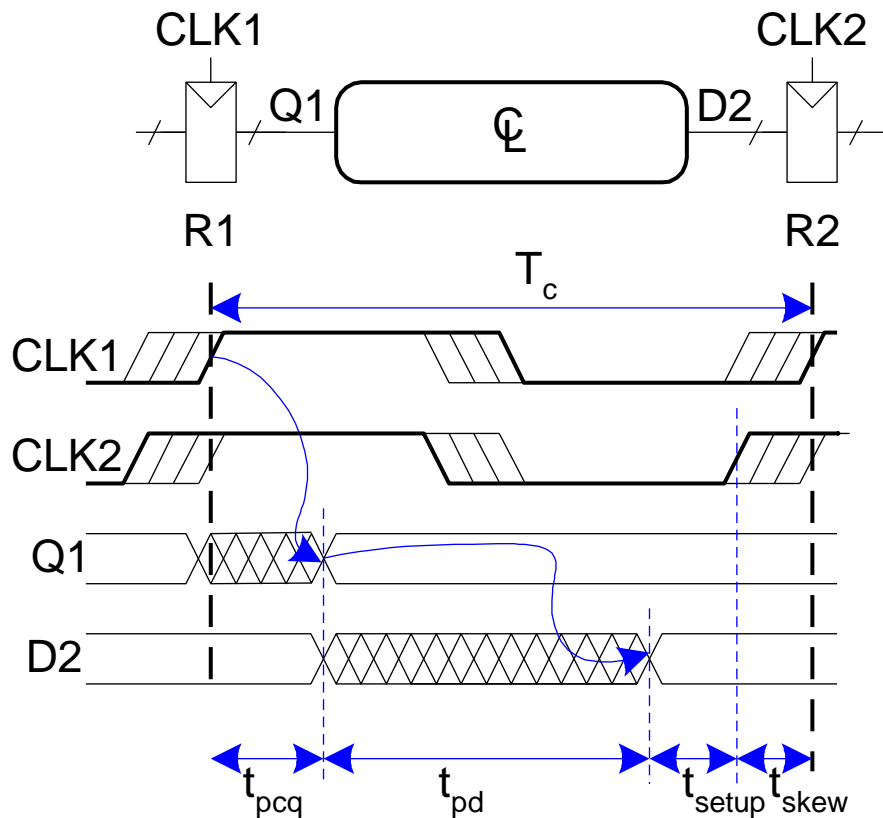
Clock Skew

- The clock doesn't arrive at all registers at same time
- **Skew**: difference between two clock edges
- Perform **worst case analysis** to guarantee dynamic discipline is not violated for any register – many registers in a system!



Setup Time Constraint with Skew

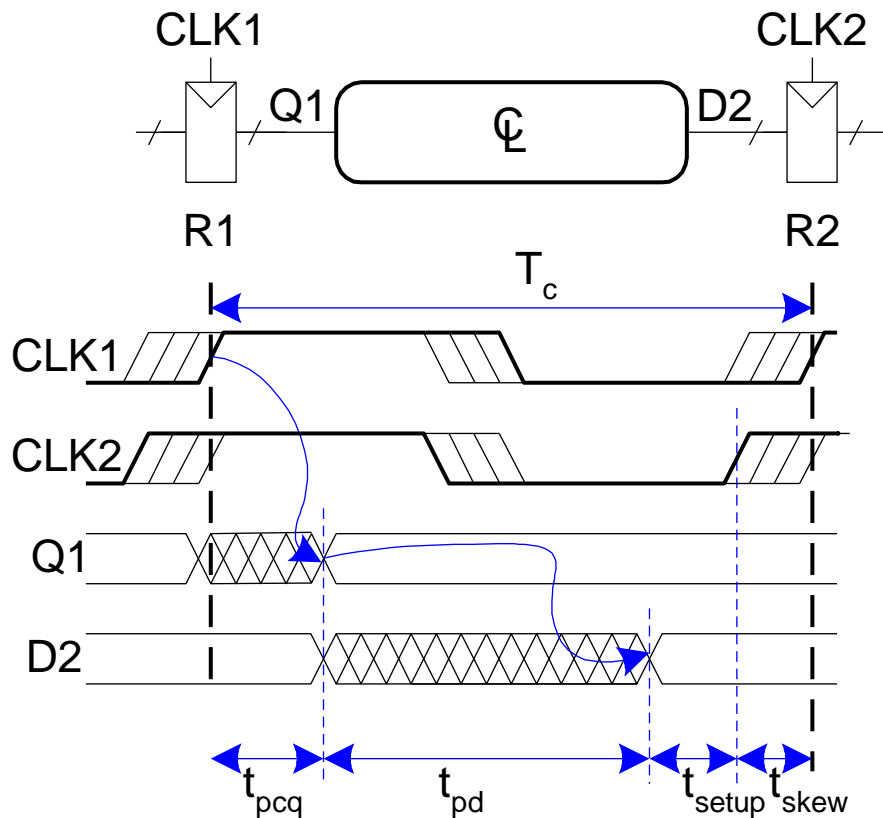
- In the worst case, CLK2 is earlier than CLK1



$$T_c \geq$$

Setup Time Constraint with Skew

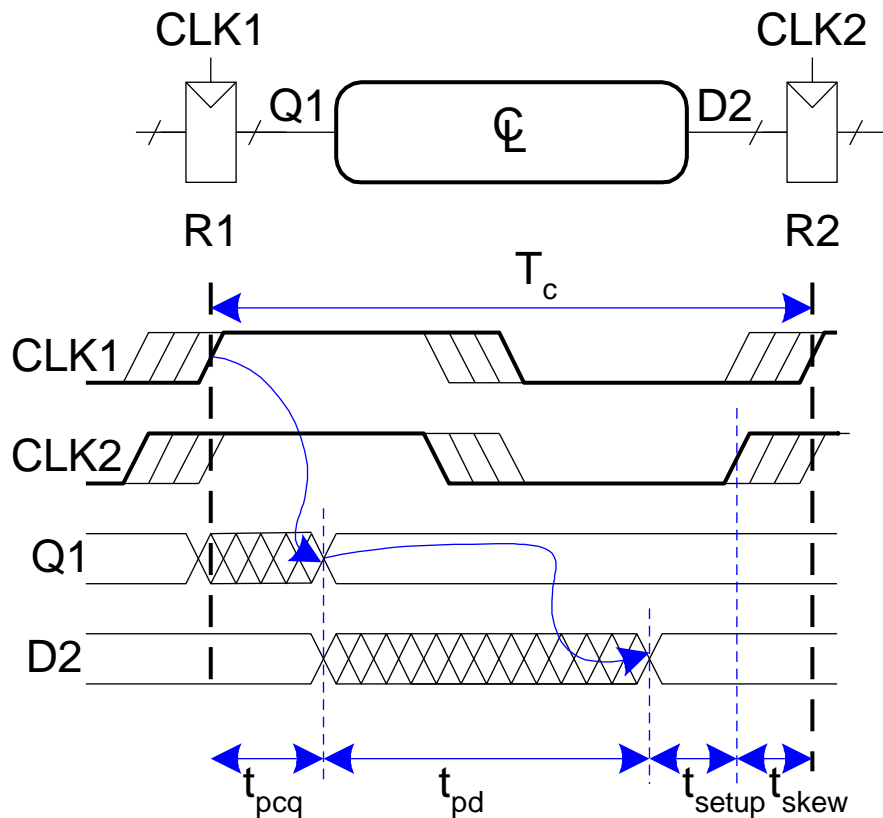
- In the worst case, CLK2 is earlier than CLK1



$$T_c \geq t_{pcq} + t_{pd} + t_{setup} + t_{skew}$$
$$t_{pd} \leq$$

Setup Time Constraint with Skew

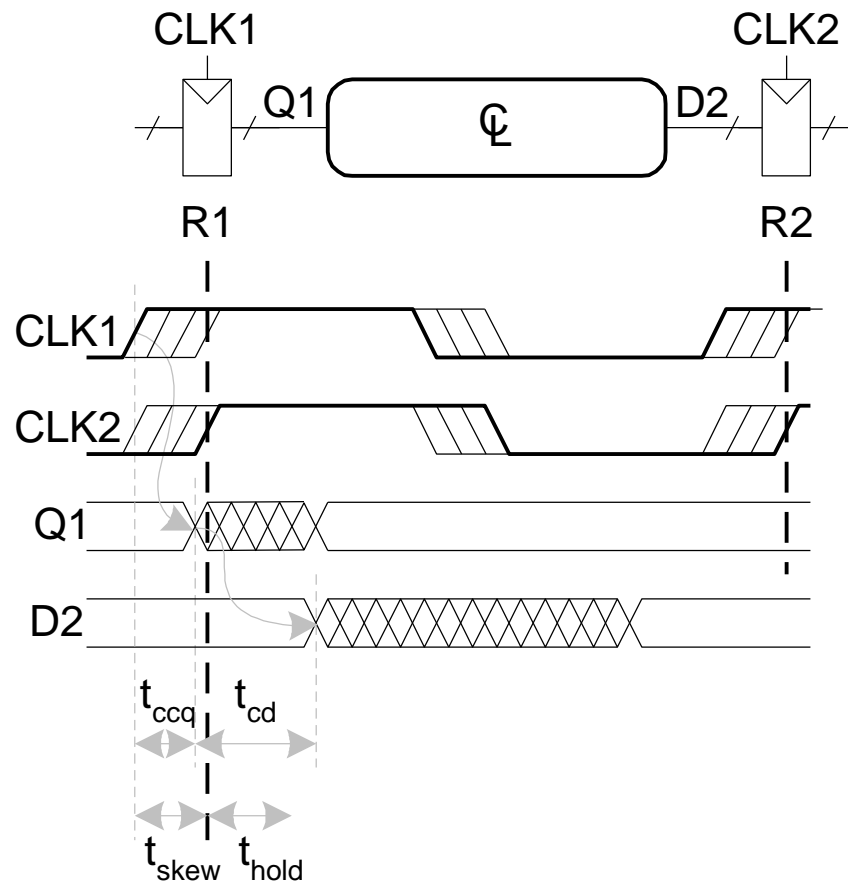
- In the worst case, CLK2 is earlier than CLK1



$$T_c \geq t_{pcq} + t_{pd} + t_{setup} + t_{skew}$$
$$t_{pd} \leq T_c - (t_{pcq} + t_{setup} + t_{skew})$$

Hold Time Constraint with Skew

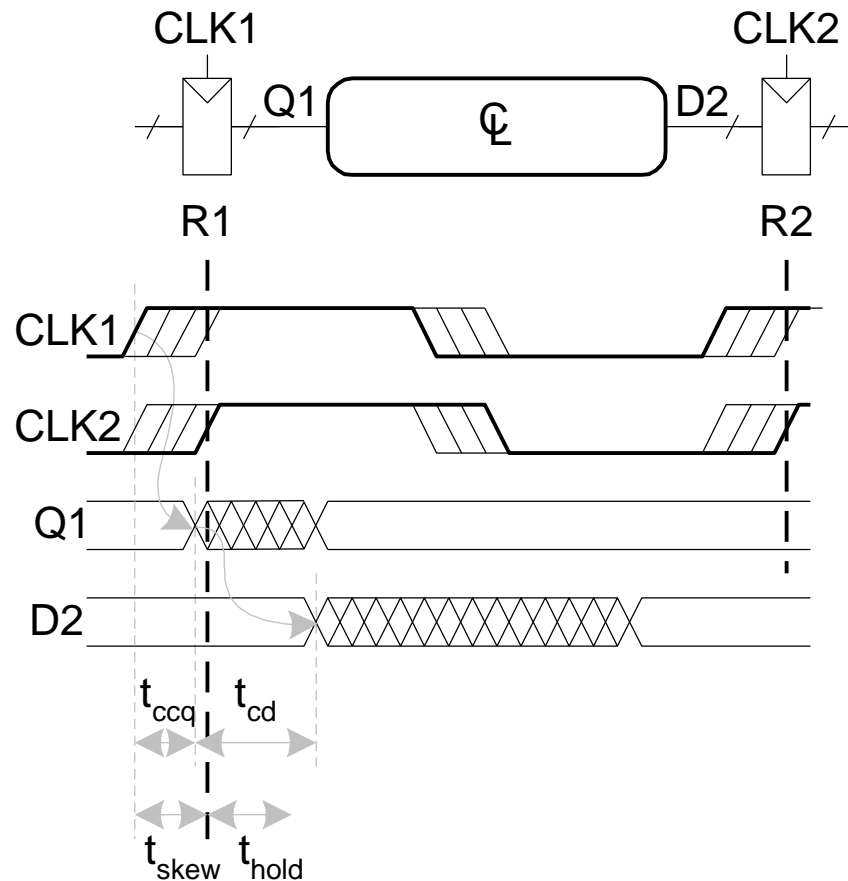
- In the worst case, CLK2 is later than CLK1



$$t_{ccq} + t_{cd} >$$

Hold Time Constraint with Skew

- In the worst case, CLK2 is later than CLK1

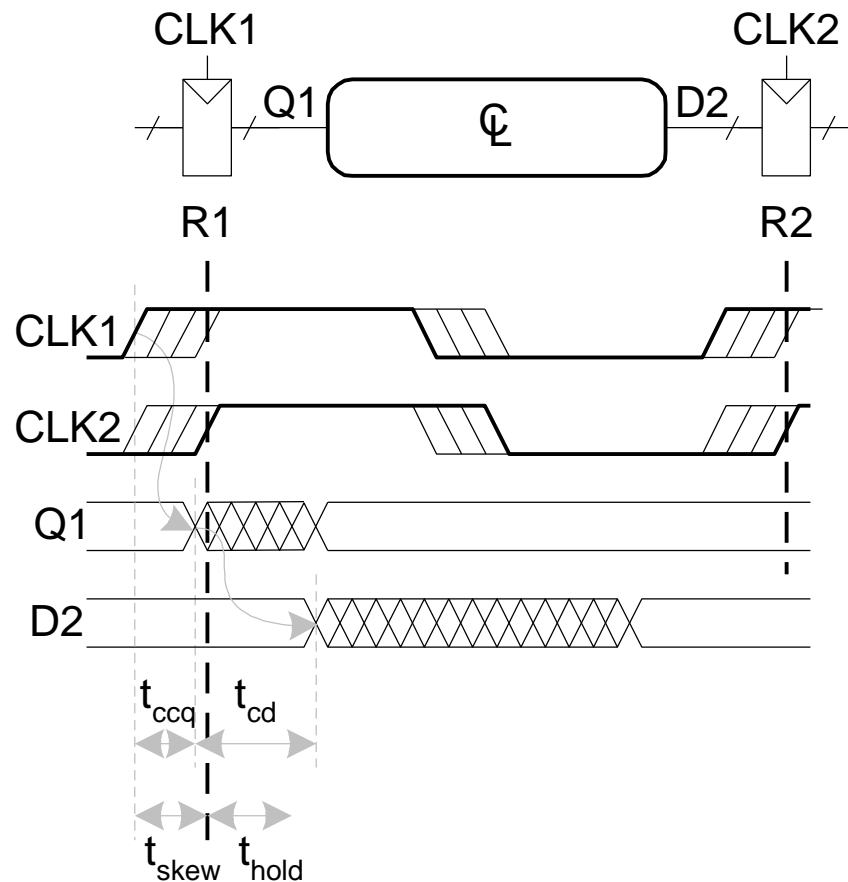


$$t_{ccq} + t_{cd} > t_{hold} + t_{skew}$$

$$t_{cd} >$$

Hold Time Constraint with Skew

- In the worst case, CLK2 is later than CLK1

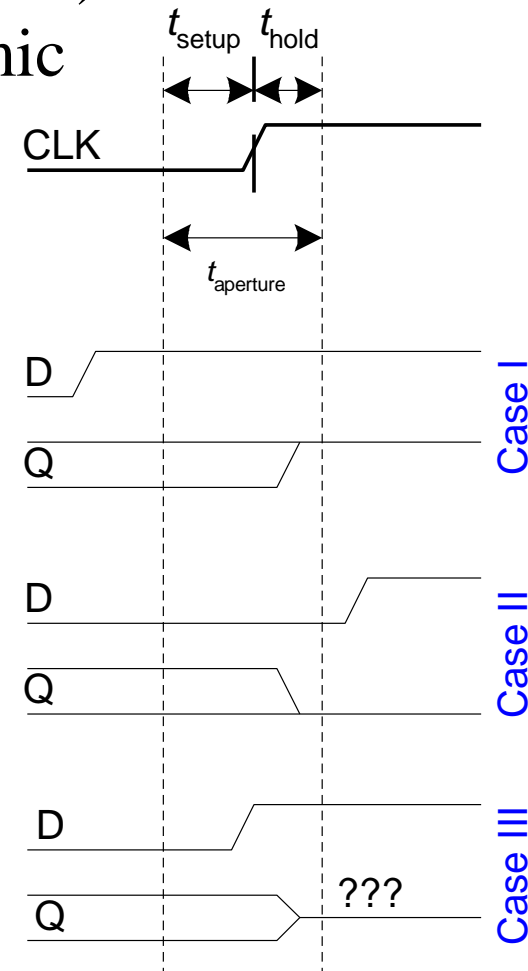
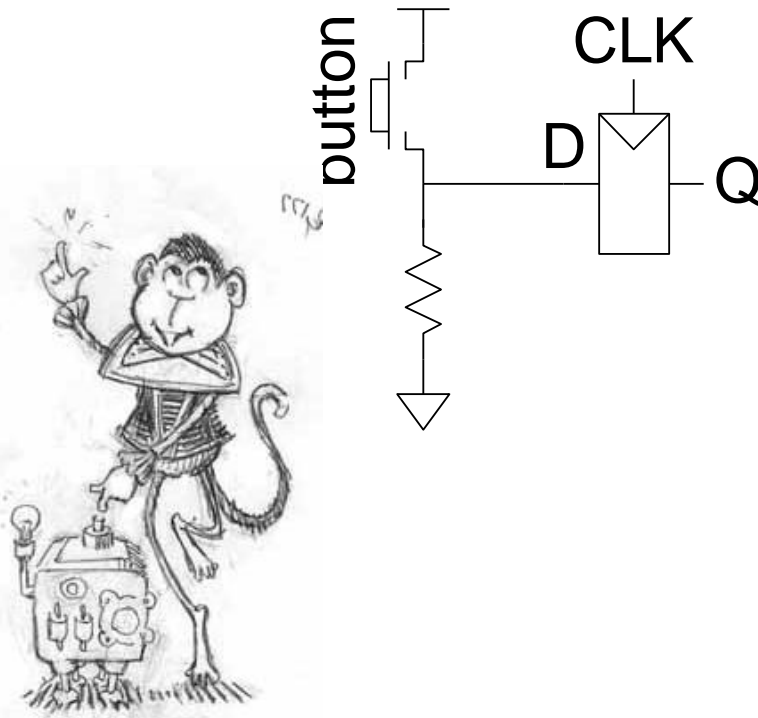


$$t_{ccq} + t_{cd} > t_{hold} + t_{skew}$$

$$t_{cd} > t_{hold} + t_{skew} - t_{ccq}$$

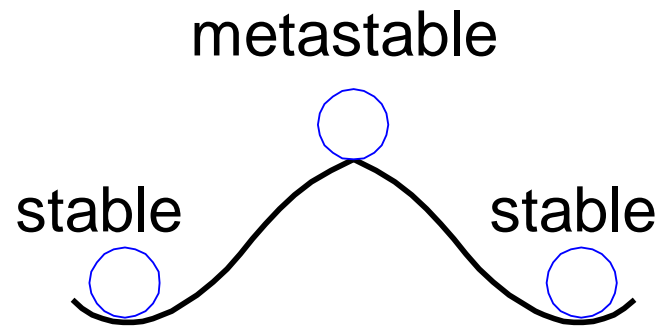
Violating the Dynamic Discipline

- Asynchronous (for example, user) inputs might violate the dynamic discipline



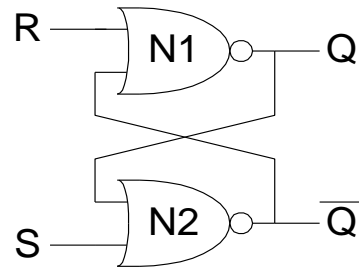
Metastability

- **Bistable devices:** two stable states, and a metastable state between them
- **Flip-flop:** two stable states (1 and 0) and one metastable state
- If flip-flop lands in metastable state, could stay there for an undetermined amount of time



Flip-Flop Internals

- Flip-flop has **feedback**: if Q is somewhere between 1 and 0, cross-coupled gates drive output to either rail (1 or 0)



- Metastable signal**: if it hasn't resolved to 1 or 0
- If flip-flop input changes at random time, **probability** that output Q is metastable after waiting some time, t :

$$P(t_{\text{res}} > t) = (T_0/T_c) e^{-t/\tau}$$

t_{res} : time to resolve to 1 or 0

T_0, τ : properties of the circuit



Metastability

- **Intuitively:**

T_0/T_c : probability input changes at a bad time (during aperture)

$$P(t_{\text{res}} > t) = (T_0/T_c) e^{-t/\tau}$$

τ : time constant for how fast flip-flop moves away from metastability

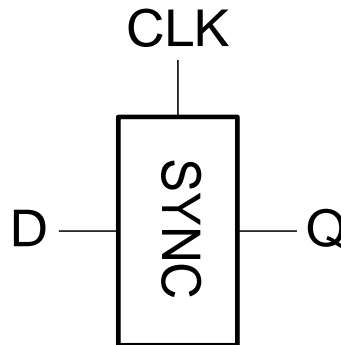
$$P(t_{\text{res}} > t) = (T_0/T_c) e^{-t/\tau}$$

- In short, if flip-flop samples metastable input, if you wait long enough (t), the output will have resolved to 1 or 0 with high probability.



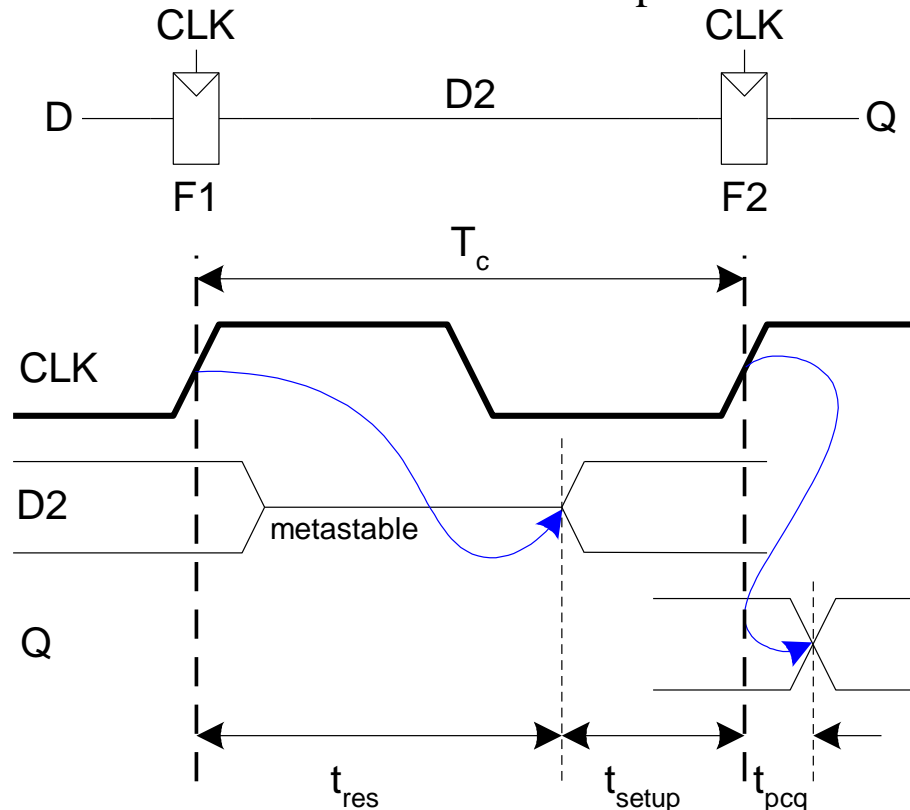
Synchronizers

- **Asynchronous inputs are inevitable** (user interfaces, systems with different clocks interacting, etc.)
- **Synchronizer goal:** make the probability of failure (the output Q still being metastable) low
- Synchronizer cannot make the probability of failure 0



Synchronizer Internals

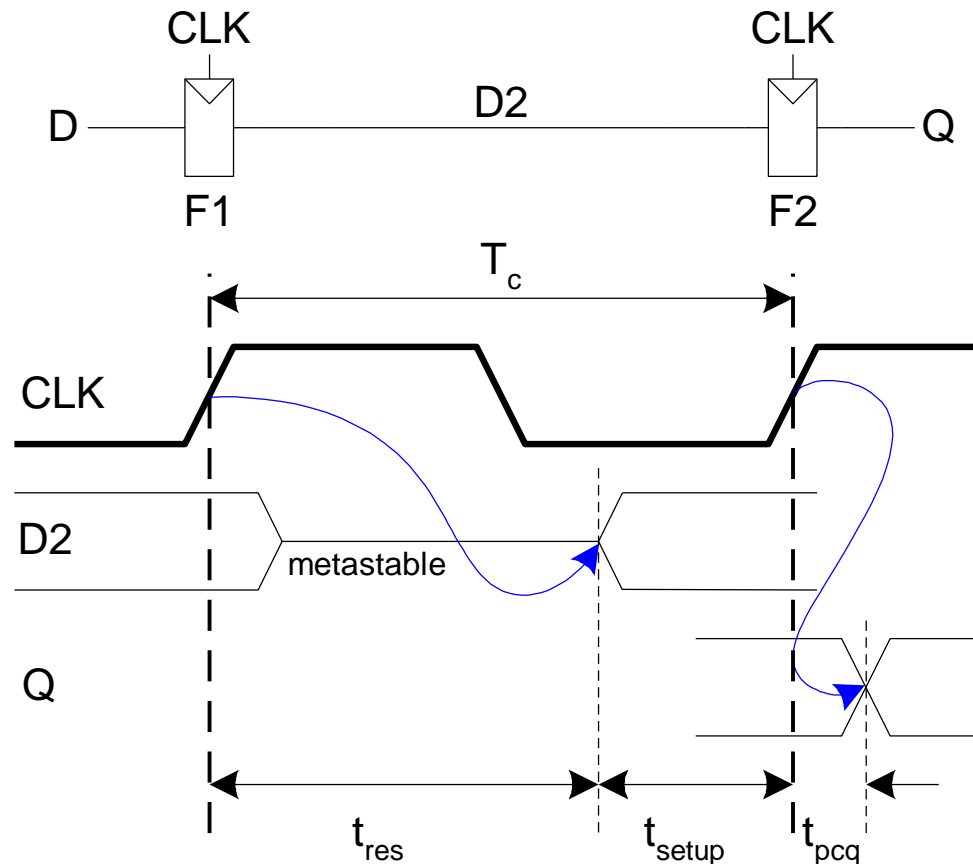
- Synchronizer: built with two back-to-back flip-flops
- Suppose D is transitioning when sampled by F1
- Internal signal D2 has $(T_c - t_{\text{setup}})$ time to resolve to 1 or 0



Synchronizer Probability of Failure

For each sample, probability of failure is:

$$\mathbf{P(\text{failure})} = (T_0/T_c) \mathbf{e}^{-(T_c - t_{\text{setup}})/\tau}$$



Synchronizer Mean Time Between Failures

- If asynchronous input changes once per second, probability of failure per second is $P(\text{failure})$.
- If input changes N times per second, probability of failure per second is:

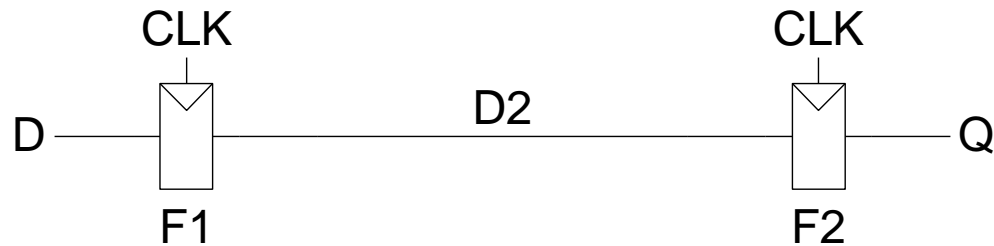
$$P(\text{failure})/\text{second} = (NT_0/T_c) e^{-(T_c - t_{\text{setup}})/\tau}$$

- Synchronizer fails, on average, $1/[P(\text{failure})/\text{second}]$
- Called *mean time between failures*, MTBF:

$$\text{MTBF} = 1/[P(\text{failure})/\text{second}] = (T_c/NT_0) e^{(T_c - t_{\text{setup}})/\tau}$$

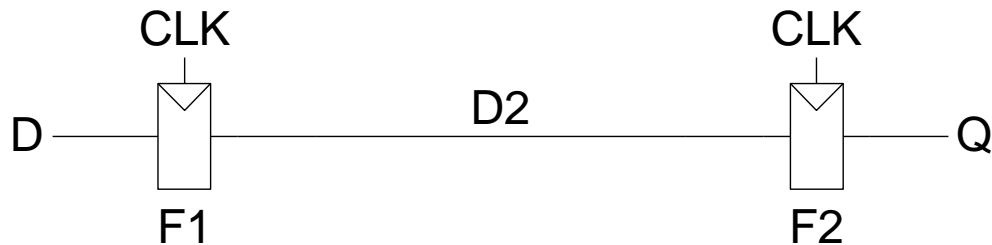


Example Synchronizer



- Suppose: $T_c = 1/500 \text{ MHz} = 2 \text{ ns}$ $\tau = 200 \text{ ps}$
 $T_0 = 150 \text{ ps}$ $t_{\text{setup}} = 100 \text{ ps}$
 $N = 10 \text{ events per second}$
- What is the probability of failure? MTBF?

Example Synchronizer



- Suppose: $T_c = 1/500 \text{ MHz} = 2 \text{ ns}$ $\tau = 200 \text{ ps}$
 $T_0 = 150 \text{ ps}$ $t_{\text{setup}} = 100 \text{ ps}$
 $N = 10 \text{ events per second}$

- What is the probability of failure? MTBF?

$$P(\text{failure}) = (150 \text{ ps} / 2 \text{ ns}) e^{-(1.9 \text{ ns}) / 200 \text{ ps}}$$
$$= 5.6 \times 10^{-6}$$

$$P(\text{failure}) / \text{second} = 10 \times (5.6 \times 10^{-6})$$
$$= 5.6 \times 10^{-5} / \text{second}$$

$$\text{MTBF} = 1 / [P(\text{failure}) / \text{second}] \approx 5 \text{ hours}$$



Parallelism

- **Two types of parallelism:**
 - **Spatial parallelism**
 - duplicate hardware performs multiple tasks at once
 - **Temporal parallelism**
 - task is broken into multiple stages
 - also called pipelining
 - for example, an assembly line

Parallelism Definitions

- **Token:** Group of inputs processed to produce group of outputs
- **Latency:** Time for one token to pass from start to end
- **Throughput:** Number of tokens produced per unit time

Parallelism increases throughput

Parallelism Example

- Ben Bitdiddle bakes cookies to celebrate traffic light controller installation
- 5 minutes to roll cookies
- 15 minutes to bake
- What is the latency and throughput without parallelism?

Parallelism Example

- Ben Bitdiddle bakes cookies to celebrate traffic light controller installation
- 5 minutes to roll cookies
- 15 minutes to bake
- What is the latency and throughput without parallelism?

Latency = 5 + 15 = 20 minutes = **1/3 hour**

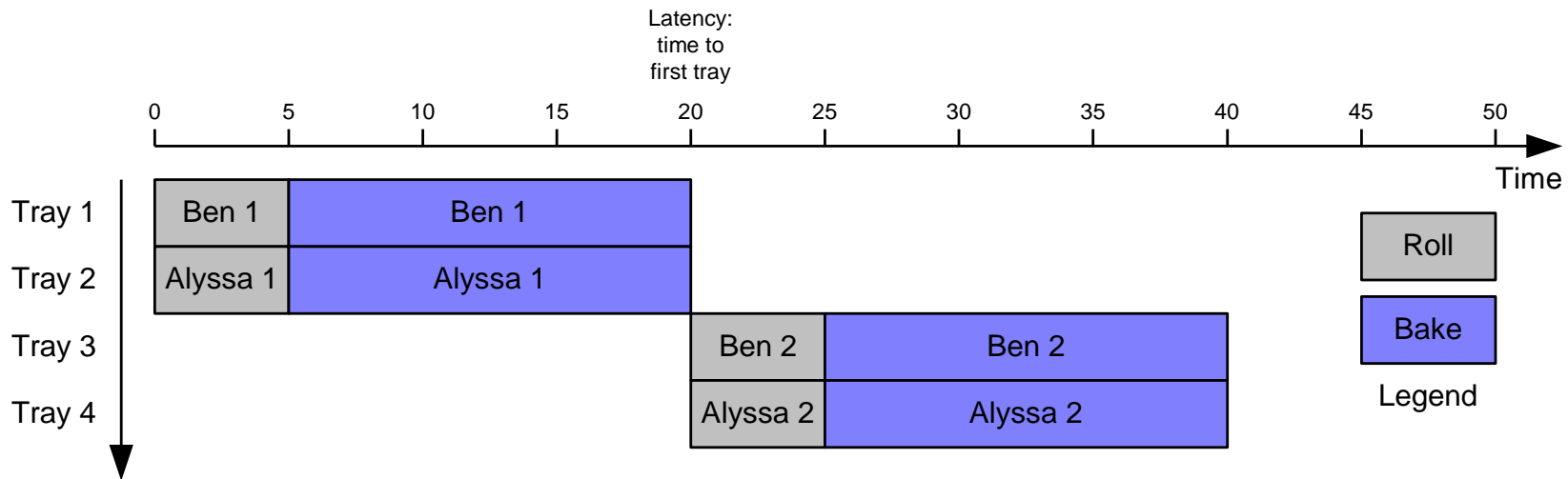
Throughput = 1 tray/ 1/3 hour = **3 trays/hour**

Parallelism Example

- What is the latency and throughput if Ben uses parallelism?
 - **Spatial parallelism:** Ben asks Allysa P. Hacker to help, using her own oven
 - **Temporal parallelism:**
 - two stages: rolling and baking
 - He uses two trays
 - While first batch is baking, he rolls the second batch, etc.



Spatial Parallelism

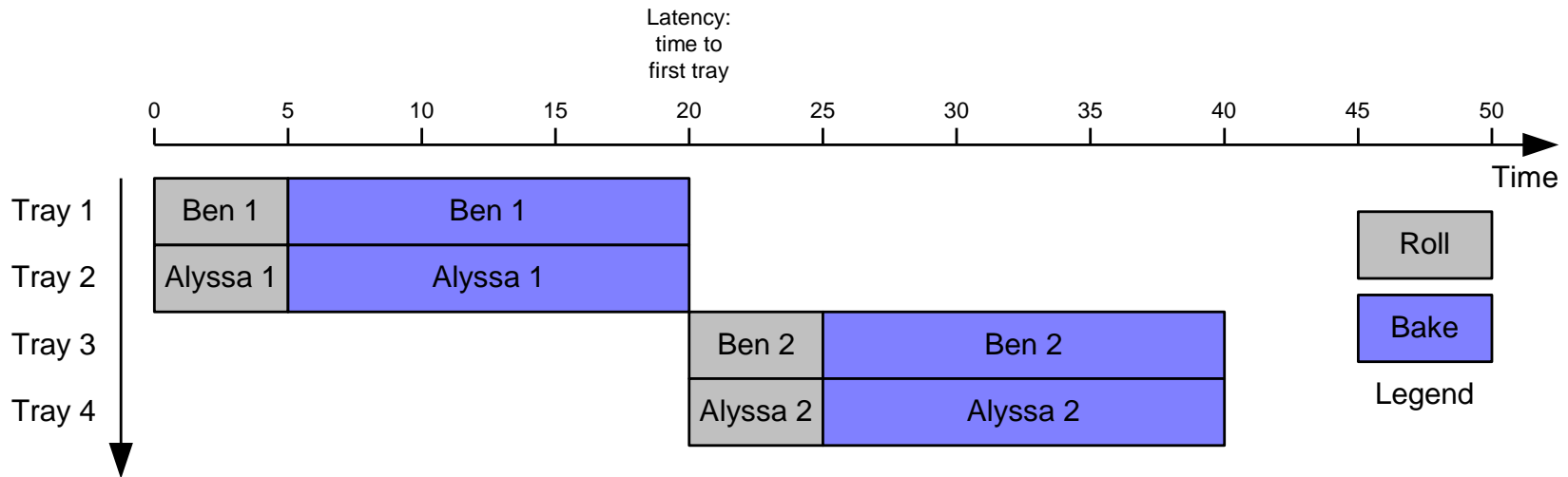


Latency = ?

Throughput = ?



Spatial Parallelism

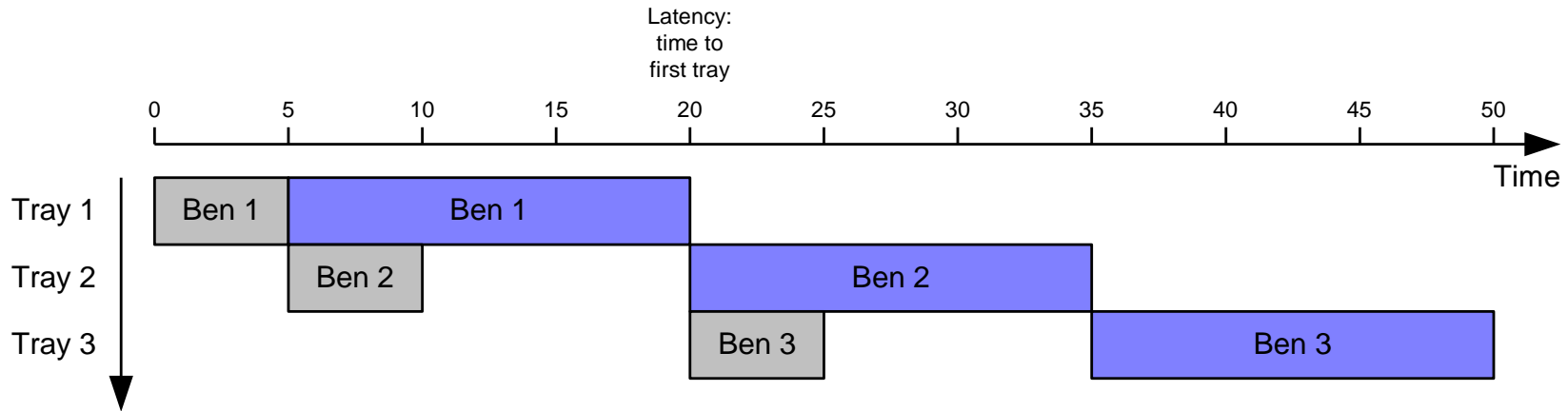


Latency = 5 + 15 = 20 minutes = **1/3 hour**

Throughput = 2 trays/ 1/3 hour = **6 trays/hour**

Temporal Parallelism

Temporal
Parallelism

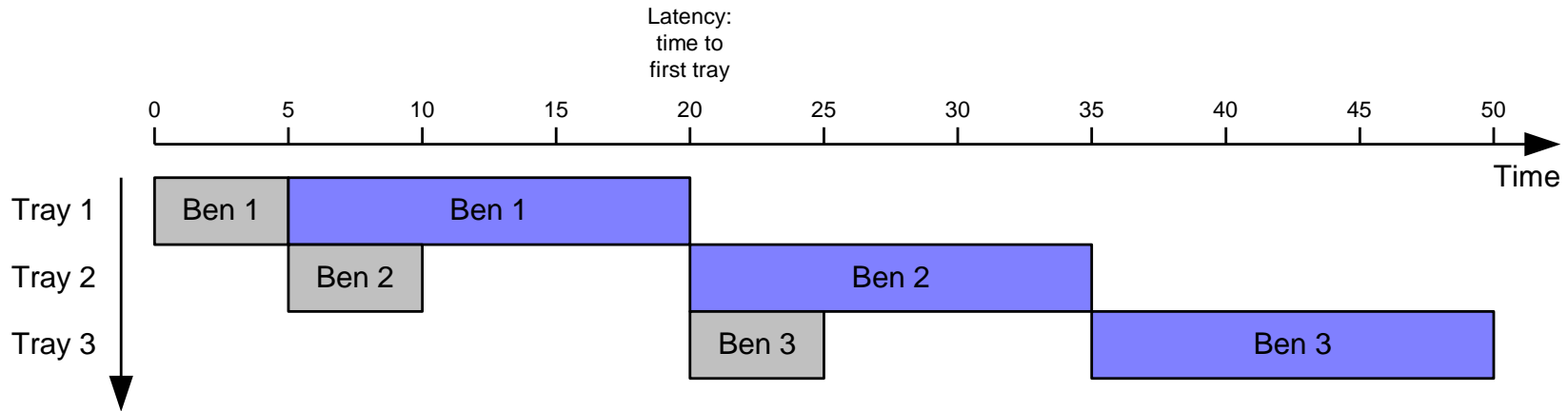


Latency = ?

Throughput = ?

Temporal Parallelism

Temporal
Parallelism



Latency = 5 + 15 = 20 minutes = **1/3 hour**

Throughput = 1 trays/ 1/4 hour = **4 trays/hour**

Using both techniques, the throughput would be **8 trays/hour**