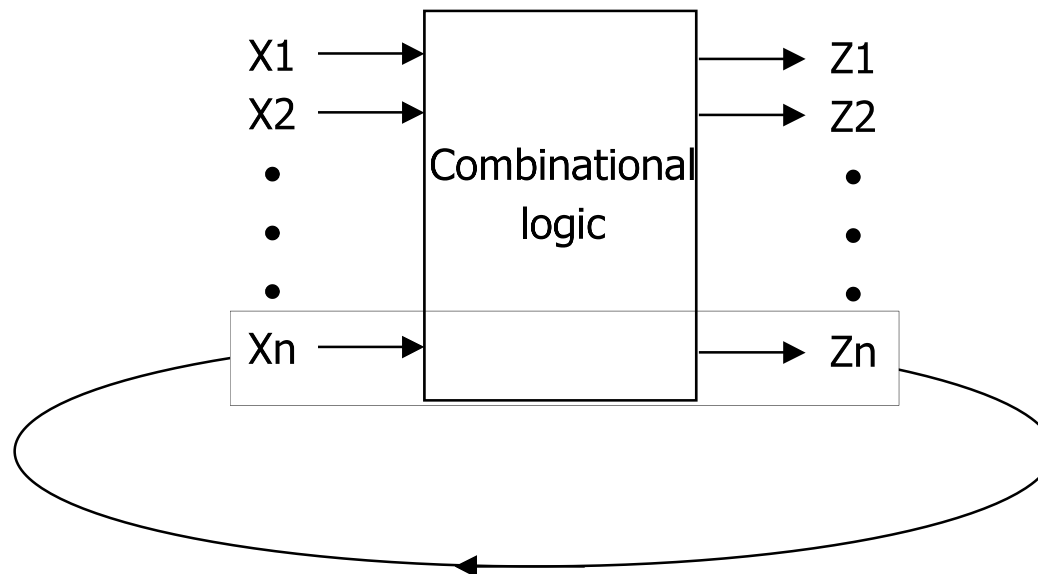
Sequential logic

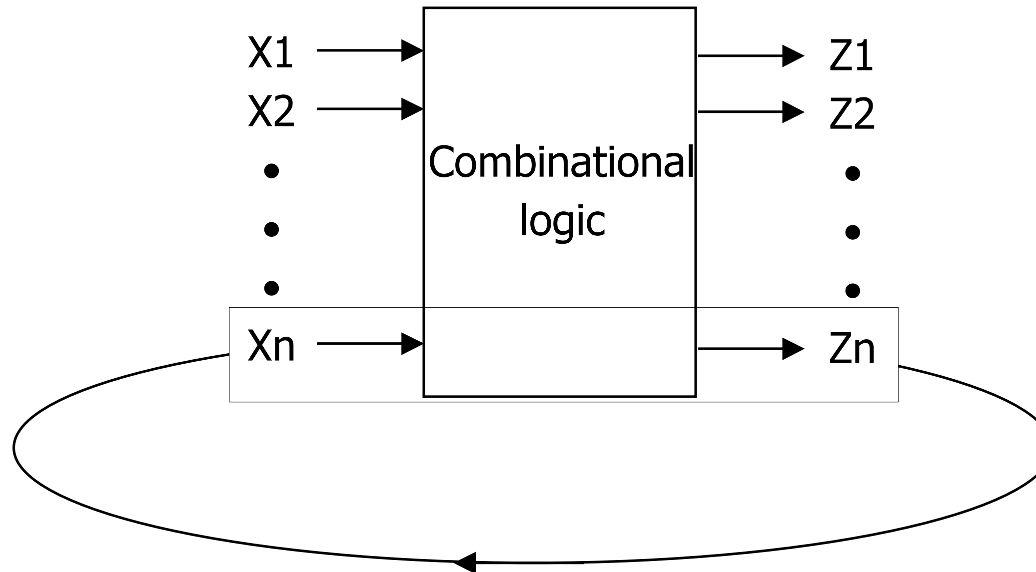
- Sequential circuits
  - simple circuits with feedback
  - latches
  - edge-triggered flip-flops
- Timing methodologies
  - cascading flip-flops for proper operation
  - clock skew
- Asynchronous inputs
  - metastability and synchronization
- Basic registers
  - shift registers

Sequential logic

- Up until now, we've built combinational circuits: outputs are just a function of the inputs.
- Now, we get into circuits with feedback.



Sequential logic diagram



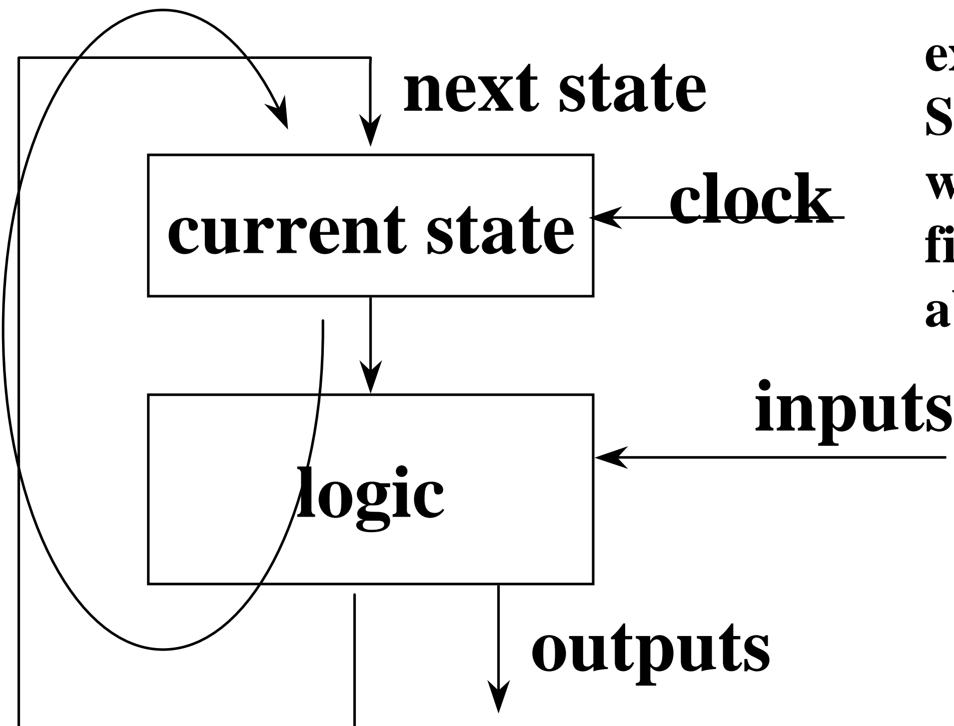
- But, how do we know that the outputs Z1..Zn will stabilize?
- Isn't possible that the outputs will endlessly change, never reaching a stable state?
- Yes, this can happen!
  - We will have to make sure that perpetual oscillation doesn't happen.
  - Or that we actually WANT it.

Sequential logic: what's the point?

- But, feedback seems to make things more complicated (e.g.: need to make sure things don't oscillate). So what's the point?
- Well, feeding outputs back into inputs lets us do interesting things, like…
  - oscillating circuits
  - **memory, state**
- Let's see an example of how this can be useful.

Sequential circuits

- Circuits with feedback
  - outputs = f(inputs, past inputs) = f(inputs, current state)
    - next state = f(inputs, current state)
  - State is distillation of knowledge about the past. Limited to the number of bits of memory. n bits of memory = 2^n possible states.
  - mutex is inherently sequential
    - same inputs can have different outputs -- depends on order of arrival.

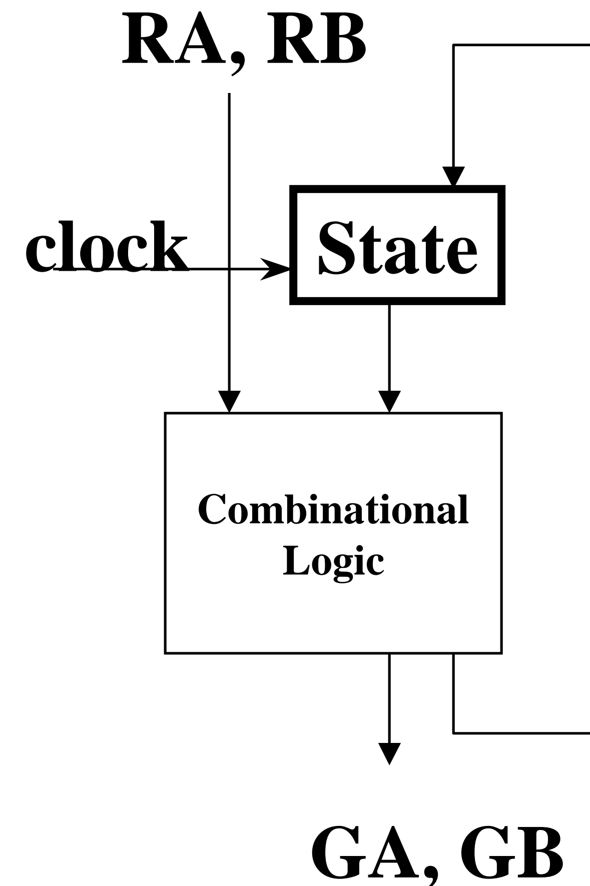


example: mutex
State remembers which request came first. Don't care about exactly when

Mutex Example (from Intro)

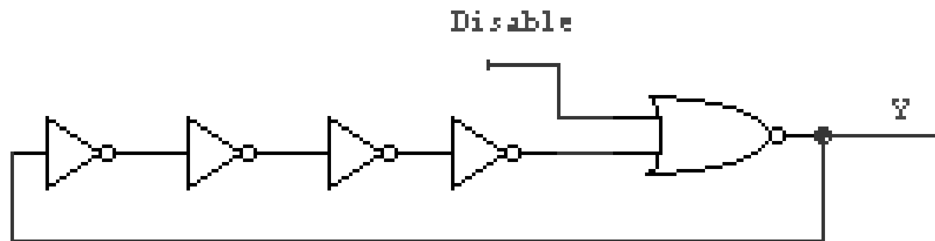
- Point: The key to synchronous sequential logic is the storage element: latches and flip-flops.

Input (RA,RB)	State	Next State	Output (GA,GB)
00	Wait	Wait	00
01	Wait	B Was First	01
10	Wait	A Was First	10
11	Wait	A Was First	10
00	A Was First	Wait	00
01	A Was First	B Was First	01
10	A Was First	A Was First	10
11	A Was First	A Was First	10
00	B Was First	Wait	00
01	B Was First	B Was First	01
10	B Was First	A Was First	10
11	B Was First	B Was First	01



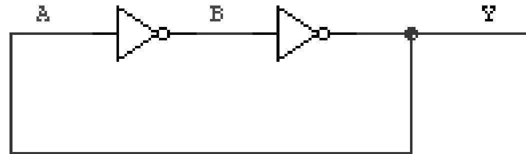
Experiment a little

- Let's start experimenting with feedback. In assignment 4, you already built a circuit with feedback:



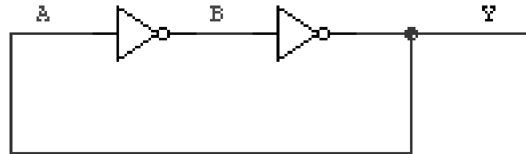
- This circuits oscillates because there are an odd number of inversions in the feedback.
- So… what happens if we only have an even number of inversions in the feedback? Let's take a look.

Cascaded inverters



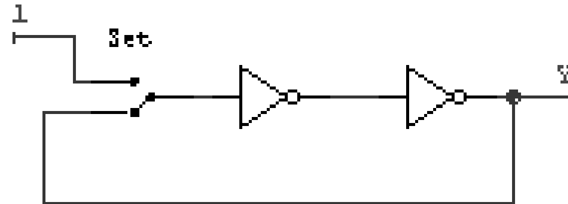
- **IF** A is "1", then B is "0", which forces A again to "1", which forces B again to "0", and so on. Thus, the ouput Y is "1", and stays "1" forever. This is a steady state (contrast this to the oscillator you did in your assignment).
- Similarly, **IF** A is "0", then Y will stay at "0" forever.
- Wow, this looks like a bit of memory (if you ignore the magical **IF**...)
- But, wait a second… How can this circuit store a value forever, it doesn't seem to be using any power: after all we are not applying any inputs...
- This is a common fallacy! Remember, we don't draw the power supplies, but they are ASSUMED to be there.

Cascaded inverters (cont'd)



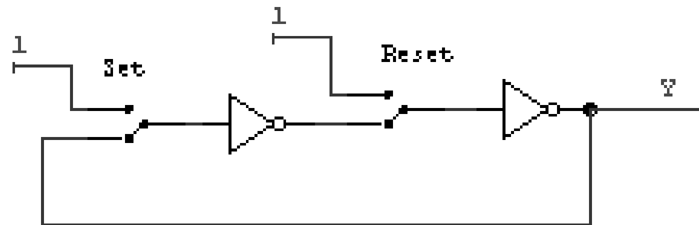
- In describing this circuit, we used the magical **IF**. But what happens if you build this in the lab? Which state will it go in?
- Is there a possibility that A and B will just remain undefined (maybe both will stay at 2.5 Volts...)
- Well, In theory this circuit has an undefined behavior. But if you build it, it WILL go into one of the two states. Why?
  - Say the circuit lands in an undefined state, maybe A=2.5 Volts, and B = 2.5 Volts.
  - The only way you'll stay in this state is if you are in perfect equilibrium.
  - But then, any perturbation in A or B will cause your circuit to spiral towards one of the two states.
- What's worse is that we **don't know** what state it will go in!
- That's not good… We need a way to guarantee that we land in the state we **WANT**.

Cascaded inverters with "Set"

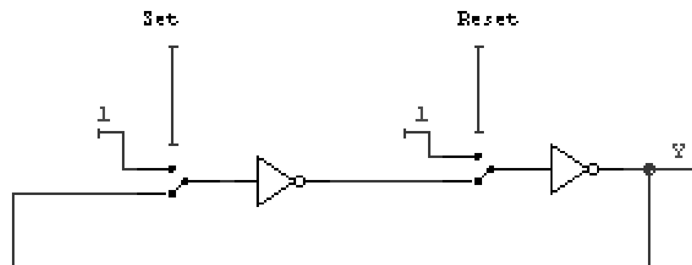


- Think of the switch as a push button, with the default state as shown. When you activate the button, it connects a "1" to the input of the first inverter. When you release the button, it bounces back to the default state.
- When you push the button, the loop is broken. The effect is that it sets Y to "1". When we release the button, the value of "1" at the output remains there.
- We'll call this a "Set" (we're setting the output…). This is good, but this means we can only store a "1". What about a "0"?

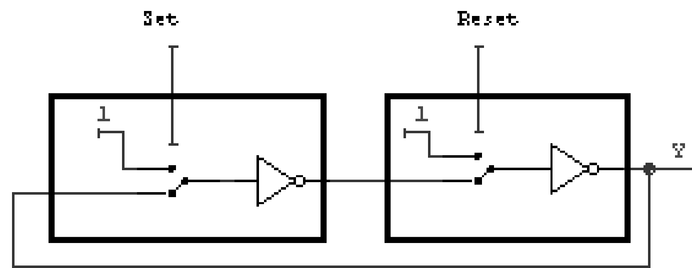
Cascaded inverters with “Set” and “Reset”



- Add a “Reset” switch, which makes the output Y go to 0. Again, when the switch is released, Y stays at 0.
- What happens if none of the switches are pressed? Well, the value of Y that was there before will stay there. This is called a “Hold”.
- So, we have a one bit of memory that we can set, reset, or just leave as is.
- Now, pushing switches is not a scalable solution. We need to control our memory cell with digital signals:

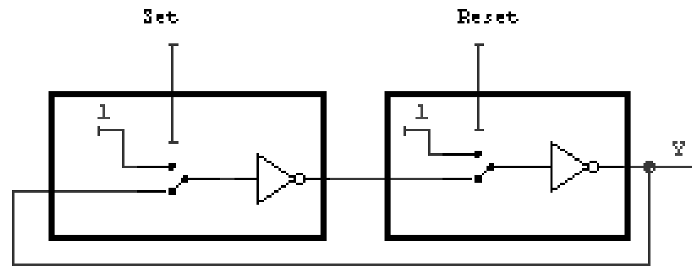


Cascaded inverters with "Set" and "Reset"

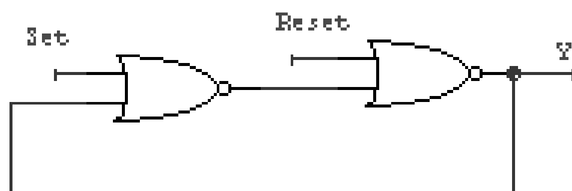


- Now, take a look at the black boxes above. Each has two inputs. When the first input is 1, the output should go to 0, whereas when the first input is 0, then the output should be the negation of the second input. What is that?

Cascaded inverters with "Set" and "Reset"

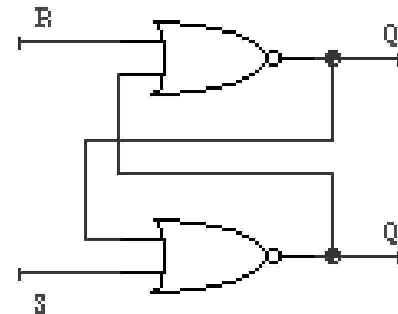


- The black boxes are NORs! So we can redraw the circuit with NORs instead of the black boxes:

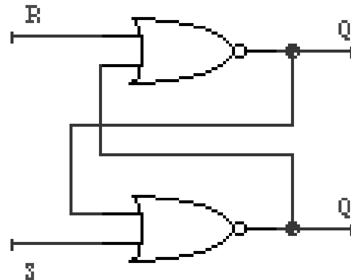


Rearranging

and renaming



Cross-coupled NOR gates



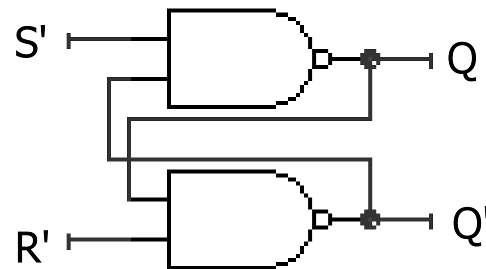
- This is called an R-S latch. We can build a table that shows how this circuit behaves:

S	R	Q
0	0	hold
0	1	0
1	0	1
1	1	???

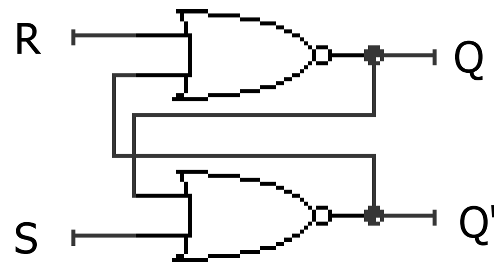
- What happens in the 1, 1 state? At first, both Y and Y' will be 0 (that's already a problem, since that means we can't call them Y and Y'...). But, then, if you "release" the S and R inputs (thus, setting S=0, and R=0), Y and Y' will oscillate. Forever!
- So… **We DISALLOW this state.**

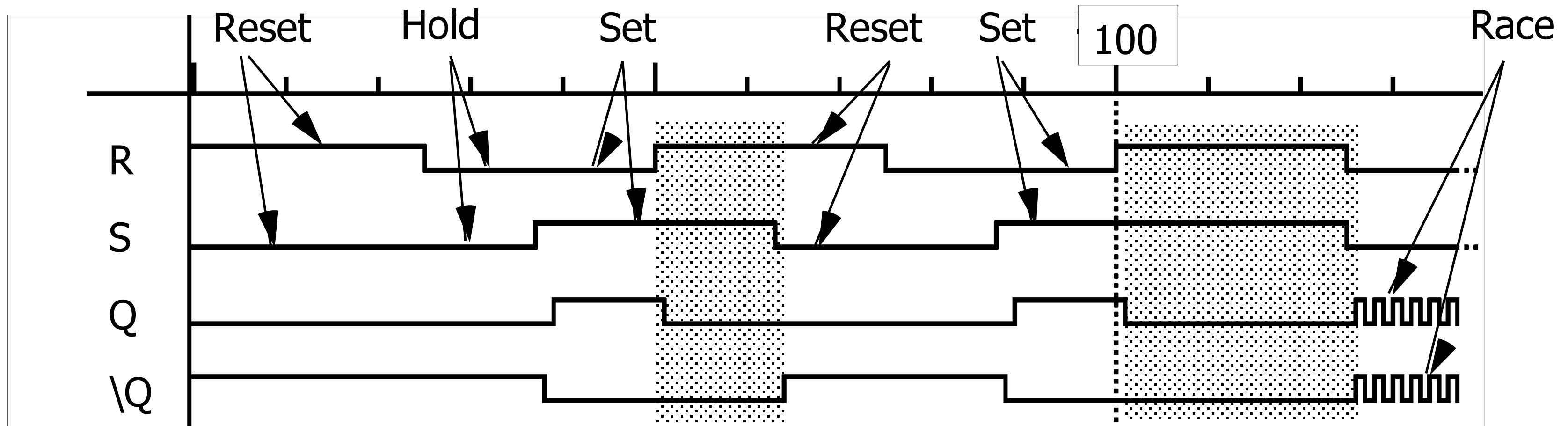
Cross coupled NAND gates

- We can do the use NAND gates aswell to build a latch
- Note the polarity of the inputs: S’ and R’.
- You should convince yourself that you can derive this circuit in the same way that we derived the circuit for cross coupled NOR gates.



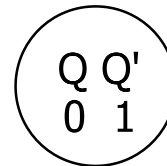
Timing behavior

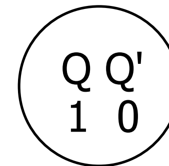




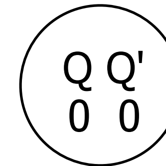
State behavior or R-S latch

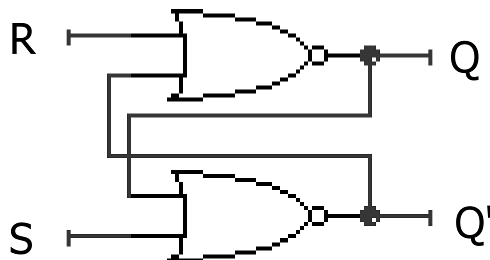
- Truth table of R-S latch behavior

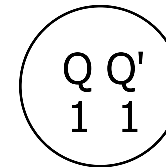




S	R	Q
0	0	hold
0	1	0
1	0	1
1	1	unstable

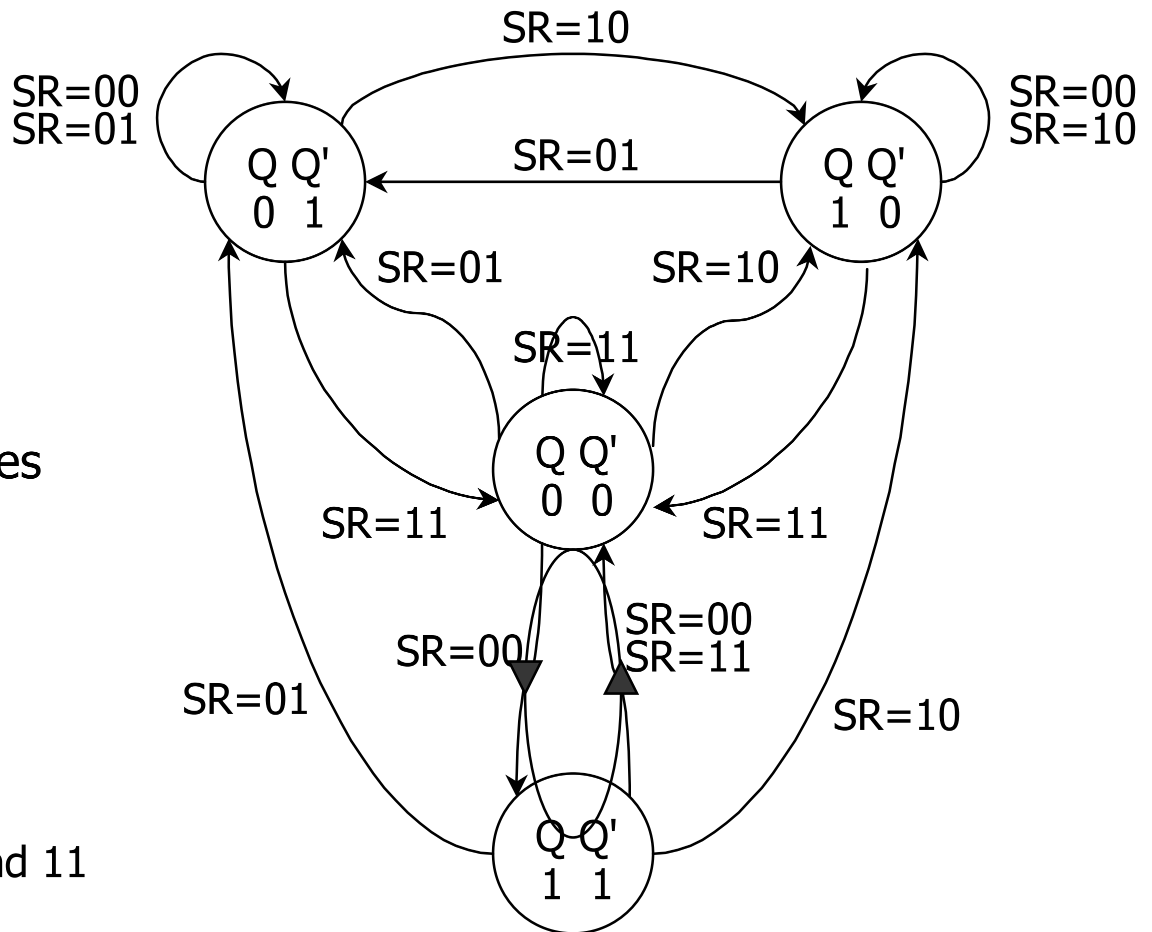






Theoretical R-S latch behavior

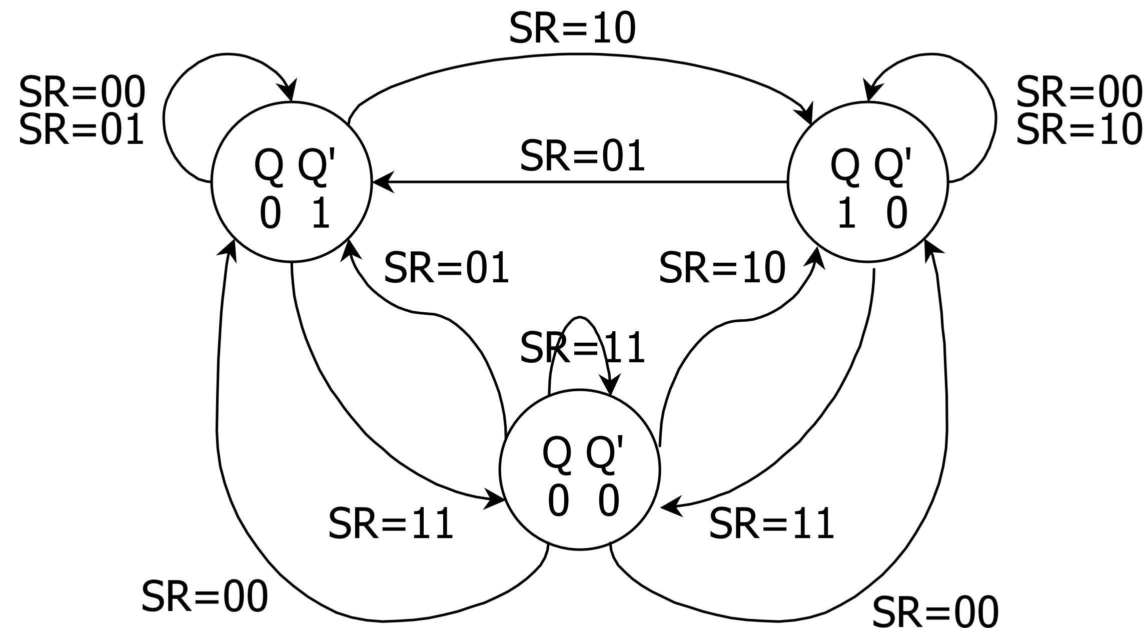
- State diagram
  - states: possible values
  - transitions: changes based on inputs



possible oscillation
between states 00 and 11

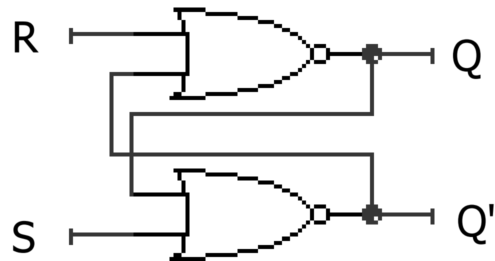
Observed R-S latch behavior

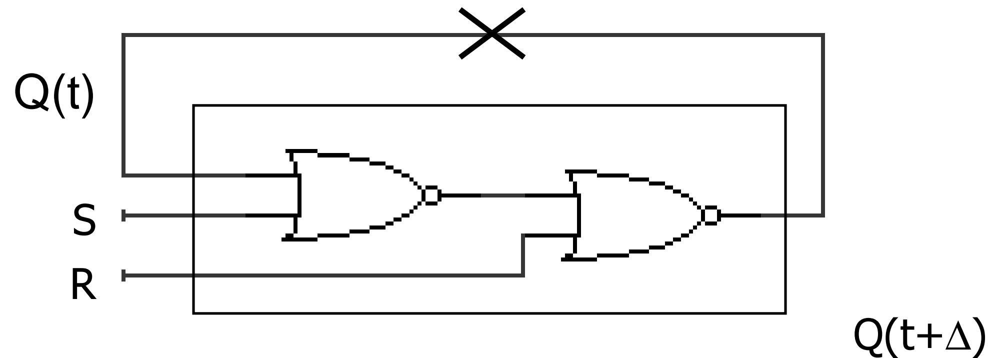
- Very difficult to observe R-S latch in the 1-1 state
  - one of R or S usually changes first
- Ambiguously returns to state 0-1 or 1-0
  - a so-called "race condition"
  - or non-deterministic transition



R-S latch analysis

- Break feedback path to write equations





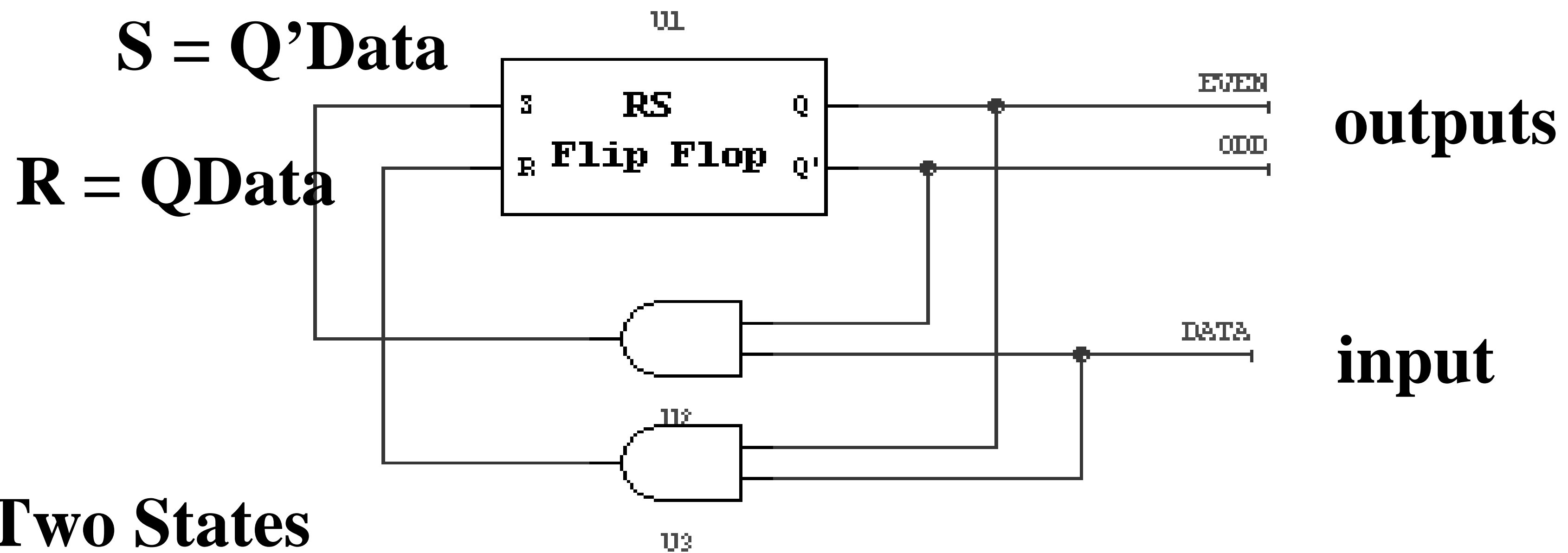
S	R	Q(t)	Q(t+Δ)	
0	0	0	0	hold
0	0	1	1	
0	1	0	0	reset
0	1	1	0	
1	0	0	1	set
1	0	1	1	
1	1	0	X	not allowed
1	1	1	X	

			S	S
	0	0	X	1
Q(t)	1	0	X	1
		R	R	

characteristic equation
$Q(t+\Delta) = S + R' Q(t)$

- **Glitch on RS causes state transition**
- **Can't have S,R = 1 at same time**

A Two-State System: Parity Checker



<u>Two States</u>

Q = 0: Total so far is ODD

Q = 1: Total so far is EVEN

Characteristic Equation: $Q(t+\Delta t) = S + R'Q(t)$

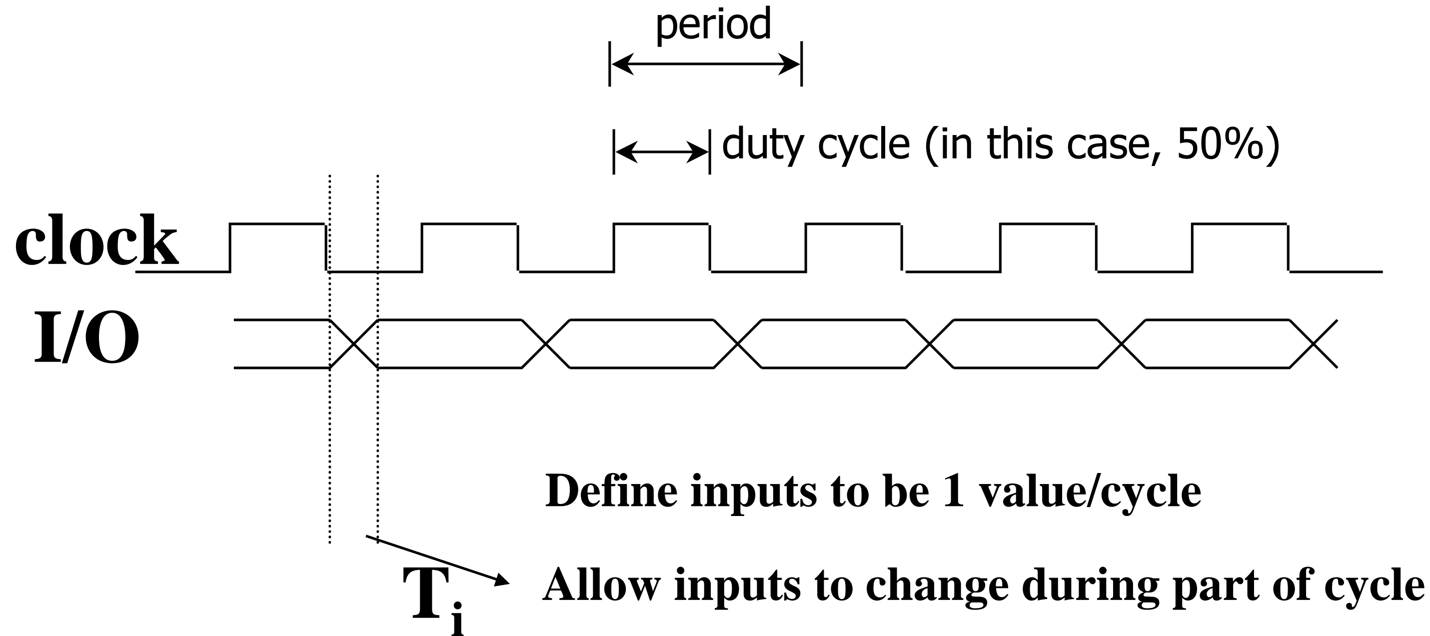
If the Data = 0: R,S = 0,0 No change in state (hold)!

If the Data = 1: $Q(t+\Delta t) = \sim Q(t)$

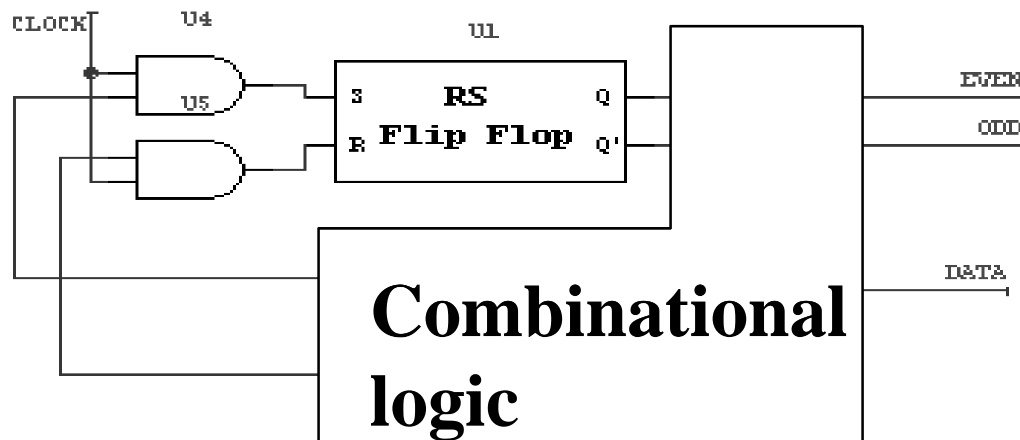
Problems? Continuous cycling and glitches.

Clocks

- Timekeeper
  - Use to prevent cycling
  - Define time boundaries between data values on inputs
  - Define time boundaries between successive states
- Clocks are regular periodic signals
  - period (time between ticks)
  - duty-cycle (time clock is high between ticks - expressed as % of period)



Using the Clock



outputs

input

Problems?
Fast Circuits
Glitches

When Clock Low: RS = 00 (HOLD)

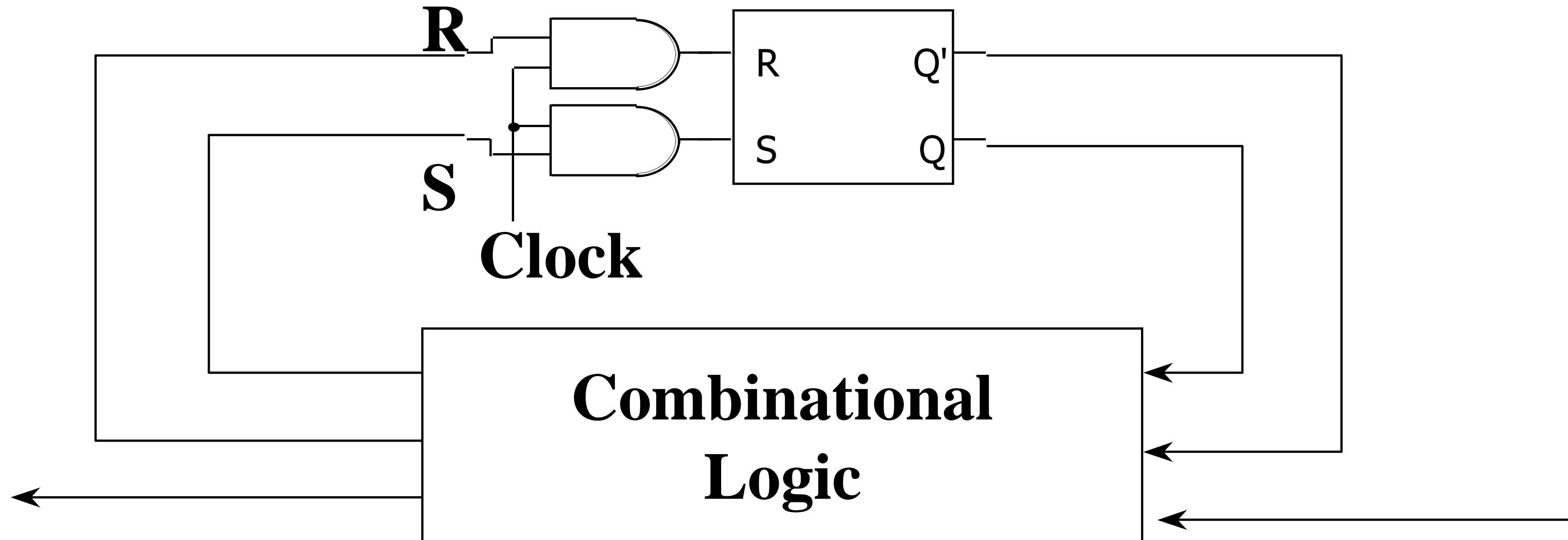
When Clock Hi:

- **Characteristic Equation: Q $(t+\Delta t)$ = S + R'Q(t)**
- **If the Data = 0: R,S = 0, No change in state (hold)!**
- **If the Data = 1: $Q(t+\Delta t) = !Q(t)$**

<u>**Works Great If one decision/cycle**</u>

- **No Wraparound:** $\Delta t > T_{hi}$
- **No Missed Transitions:** $\Delta t < T_{hi} + T_{low}$

A Two-State System

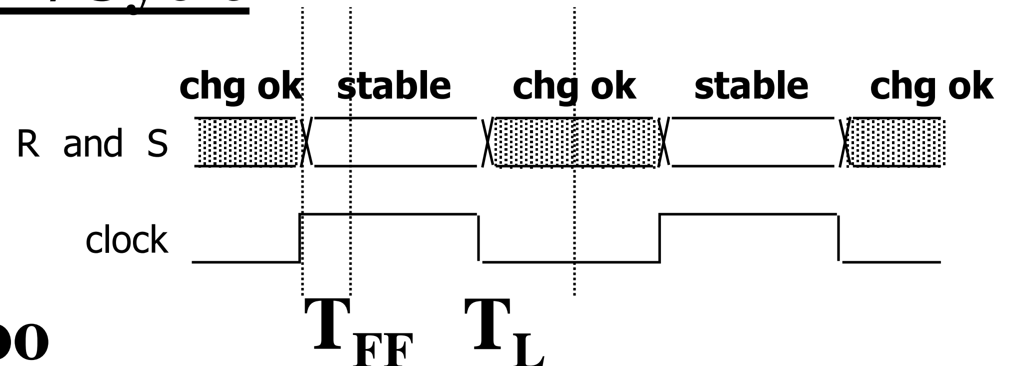


Want One State Decision/Cycle

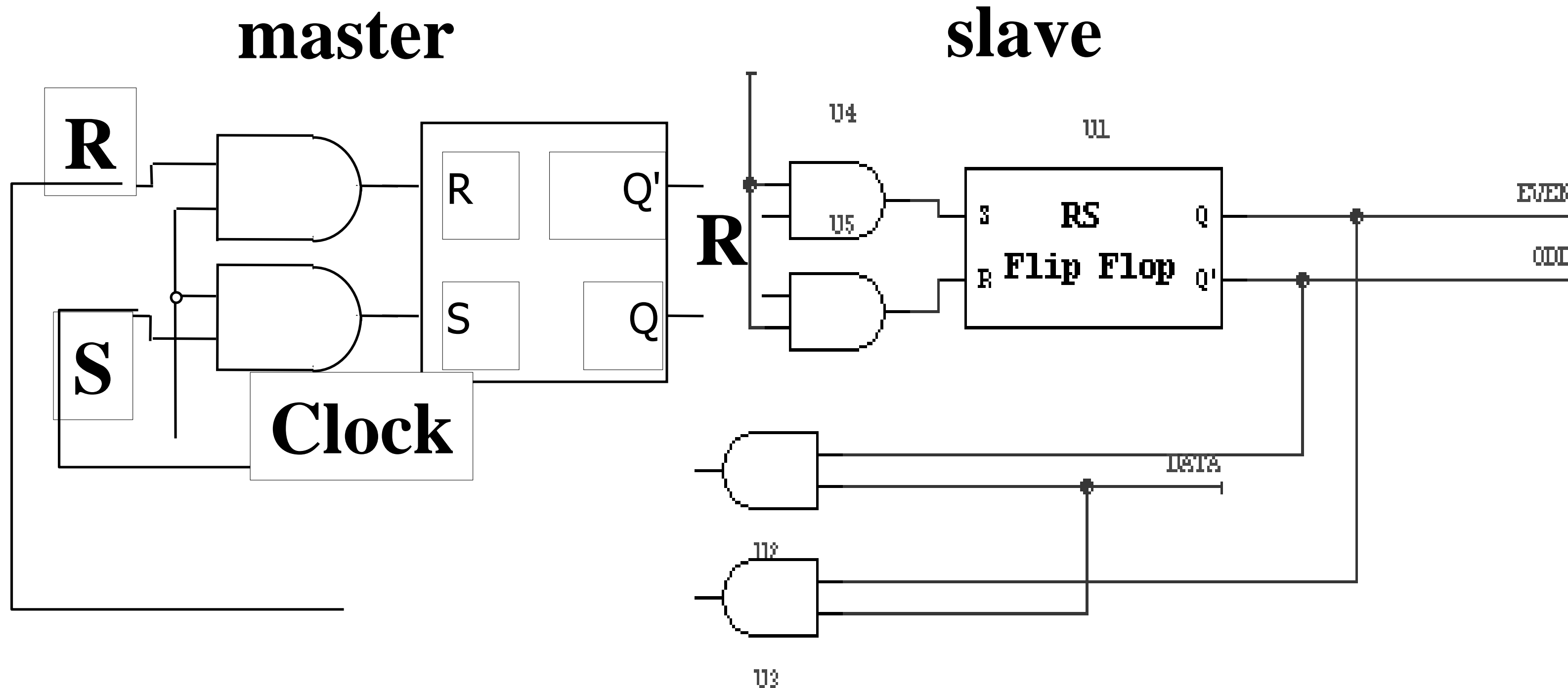
$T_{FF} + T_L > T_{hi}$

$T_{FF} + T_L < T_{hi} + T_{low}$

- Difficult to guarantee
- Worry about inputs too



Preventing Wraparound

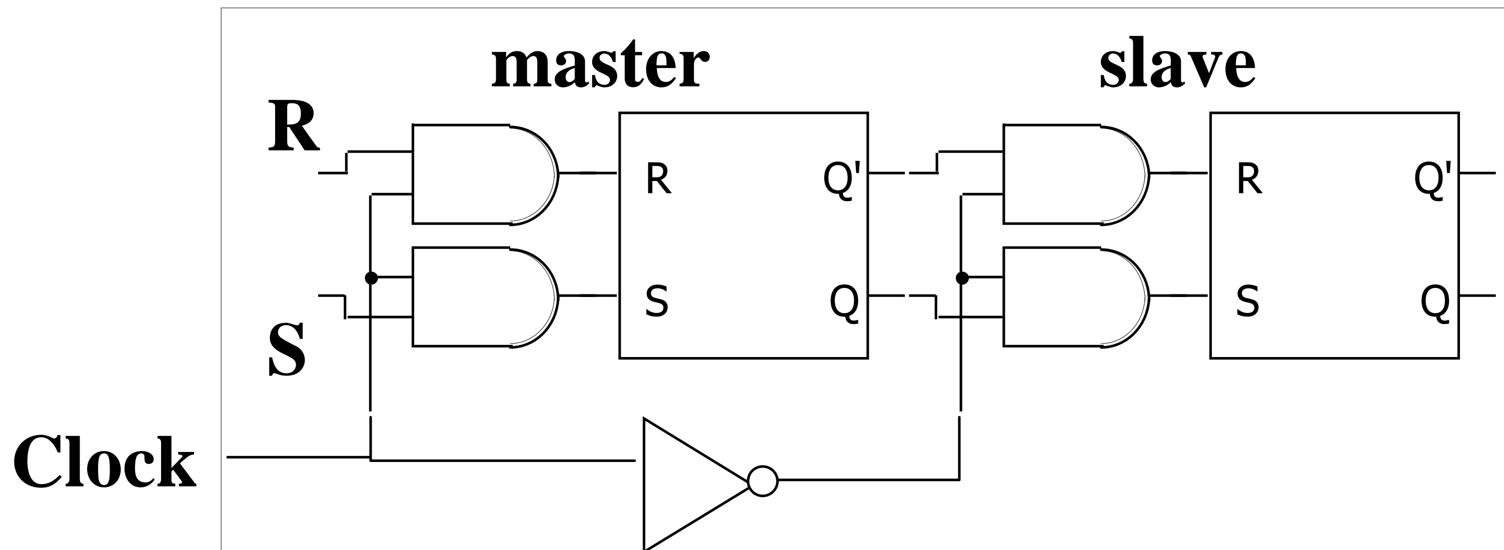


Clock = 0: master holds state(t), slave gets state(t)

Clock = 1: master gets state(t+1)
slave holds state(t)

Clock 1→0: master holds state(t+1)
slave gets state(t+1)

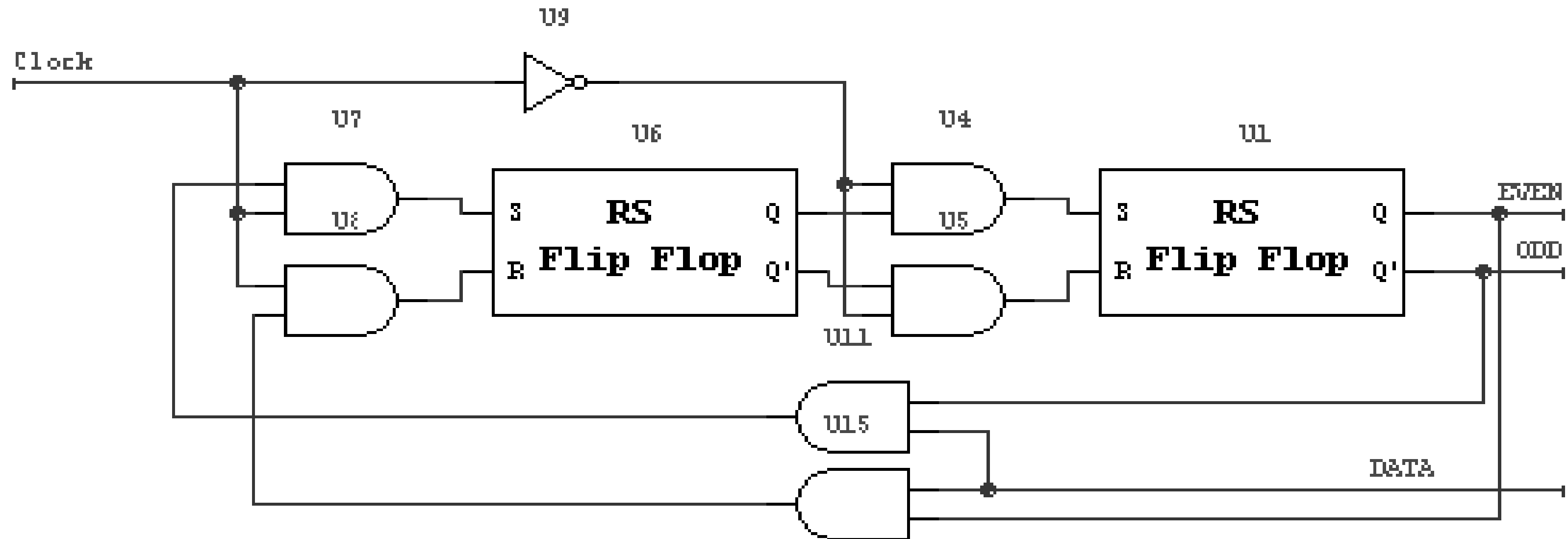
Master Slave Flip-Flop

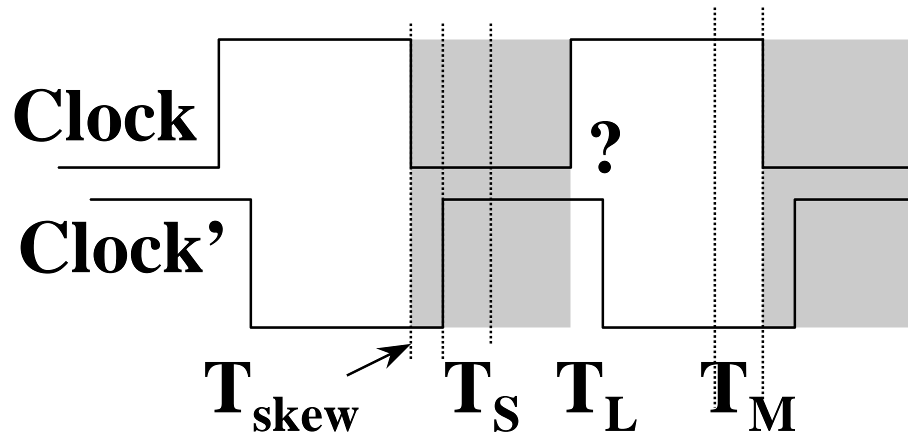


Changes on inputs can’t propagate to outputs until next clock cycle begins.

- **No looping possible**
- **Combinational logic can’t be too fast**
- **Extra hardware**
- **Decide next state before input gate closes**

Two State System with Master-Slave





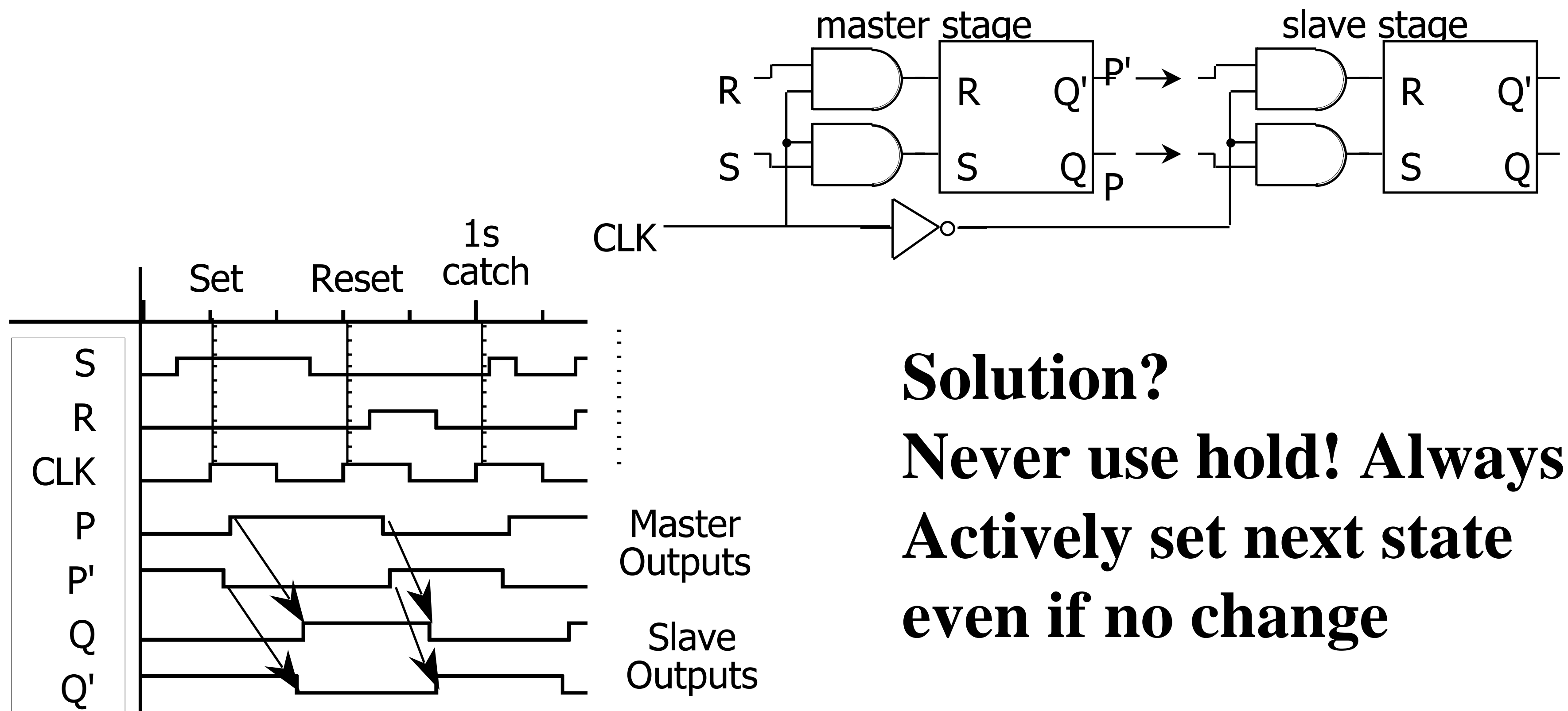
One Transition/Cycle if:

$T_S + T_L + T_M + T_{skew} < T_{low} + T_{hi}$

- Easier to guarantee
- Worry about inputs!
- Still worry about hazards!

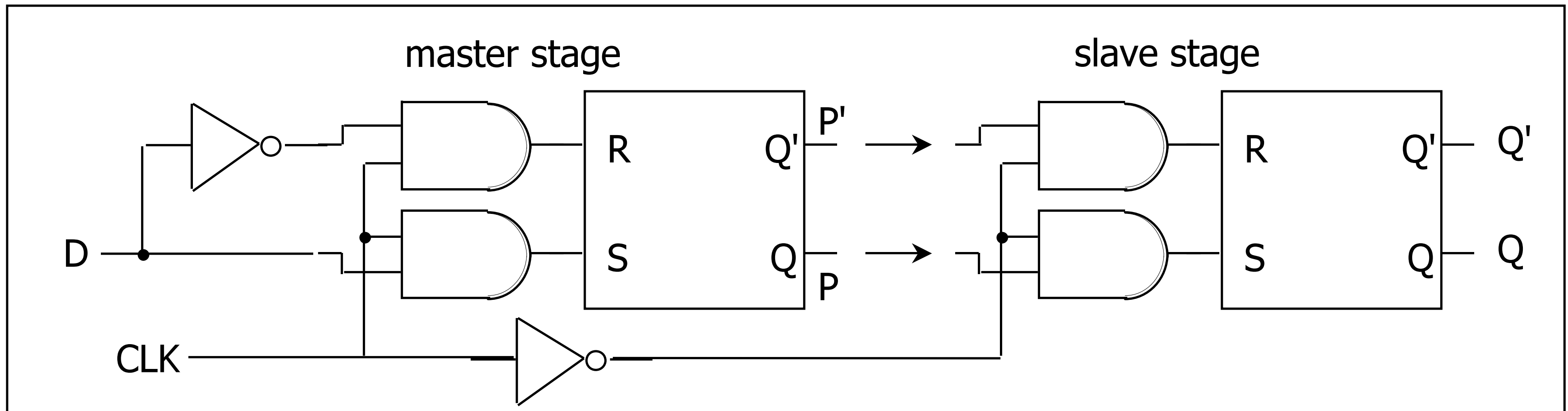
The 1s catching problem

- In first R-S stage of master-slave FF
  - 0-1-0 glitch on R or S while clock is high is "caught" by master stage
  - leads to constraints on logic to be hazard-free



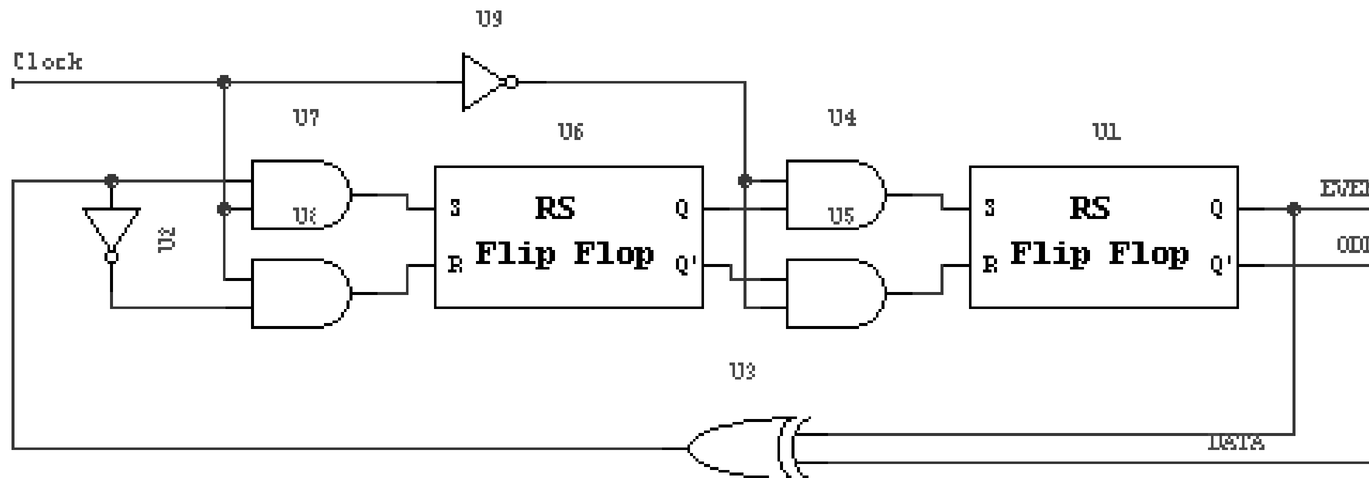
D flip-flop

- Make S and R complements of each other
  - eliminates 1s catching problem, glitches eventually settle on R or S
  - can't just hold previous value (must have new value ready every clock period)
  - value of D just before clock goes low is what is stored in flip-flop
  - can make R-S flip-flop by adding logic to make D = S + R' Q



10 gates

Final Design for Parity Checker w/ D-FF

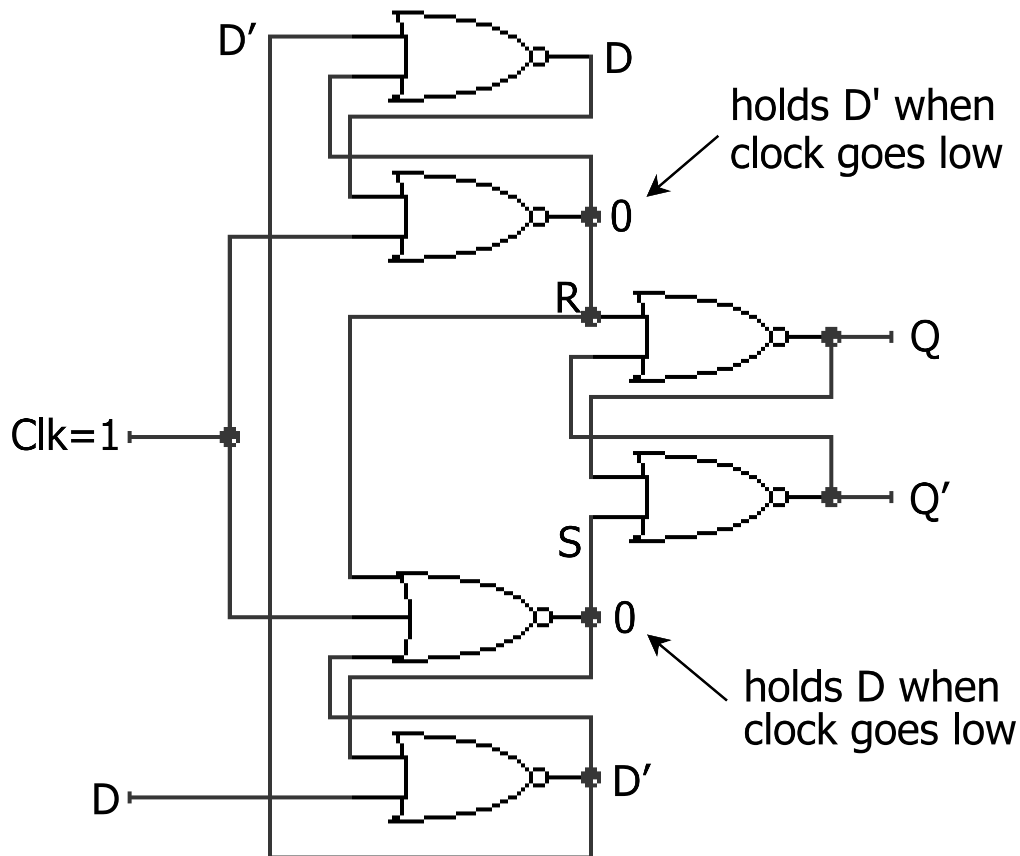


In this case, D-FF approach is less efficient!

Can we make this D-FF simpler??

Edge-triggered flip-flops

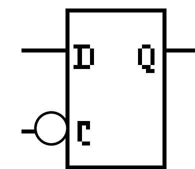
- More efficient solution: only 6 gates
  - sensitive to inputs only near edge of clock signal (not while high)



negative edge-triggered D flip-flop (D-FF)

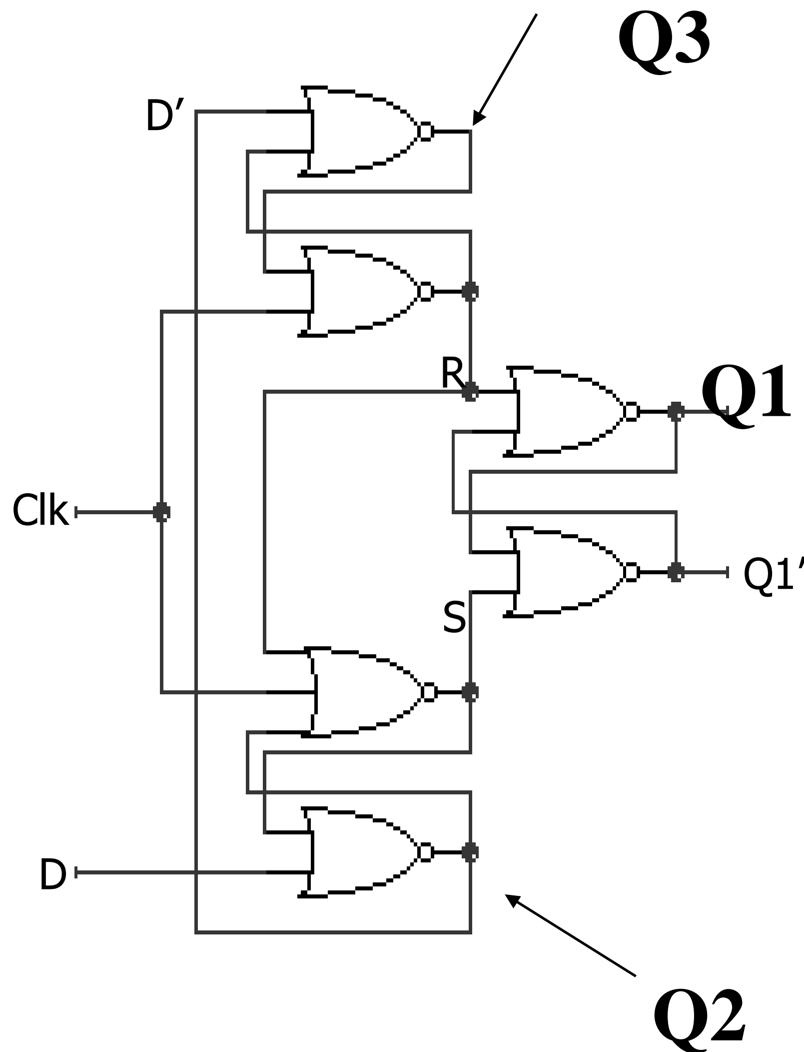
4-5 gate delays

must respect setup and hold time constraints to successfully capture input



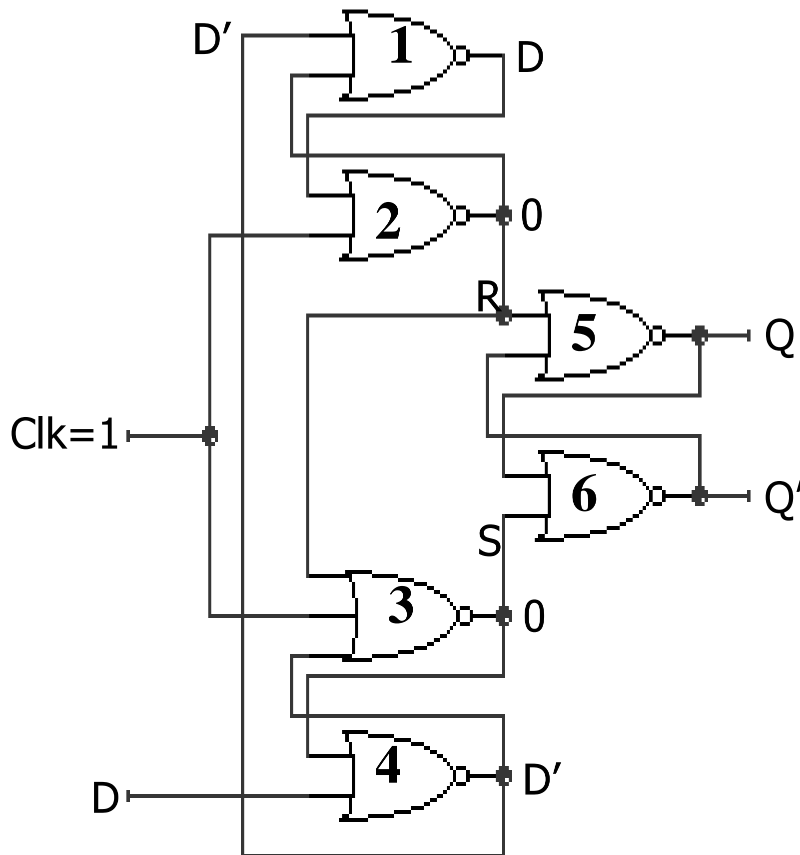
characteristic equation
$Q(t+1) = D$

Functional Analysis



Clock	Values	Comment
1	Q1 = HOLD	Force R,S to 0
1	Q2 = D′	FF2 waiting to latch
1	Q3 = D	FF3 waiting to latch
1→0	Q1 = D	After 1 gate delay R,S = D,D′
1→0	Q2 = D′	FF2 latches
1→0	Q3 = D	FF2 latches
0 (D → D*)	Q2 = 0 or D′	
0 (D→ D*)	Q3 = D	Not a function of D*
0 (D→D*)	Q1 = D (orig.)	Not a function of D*

D-FF Timing

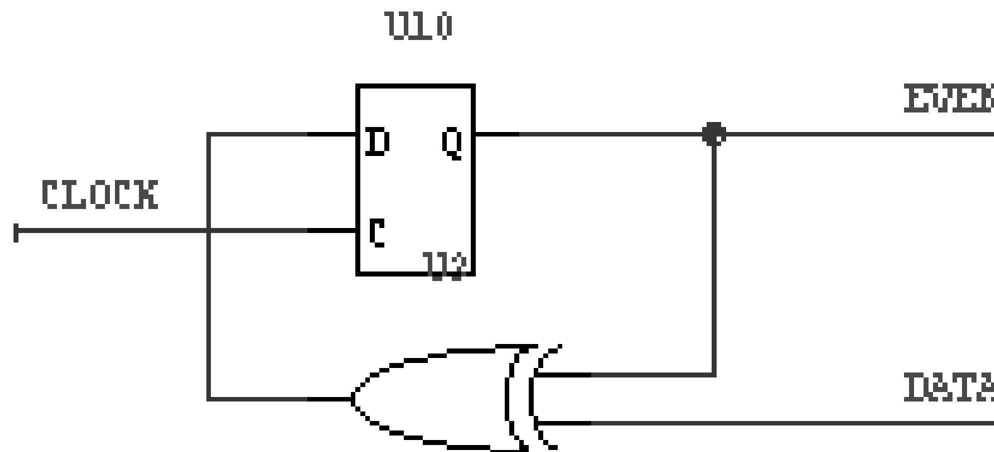


<u>Setup time:</u> Time data must be stable before negedge of clock to ensure proper latching. Where does D wait for falling edge of clock?
at G2 and G3. $T_{su} = 2$

<u>Hold Time:</u> How long after falling edge of clock is D blocked (okay for D to change) G4 and G3 inputs must be stable. Delay from Clock to Blocked is 1 (2 and 3).
$T_h = 1$

<u>Propagation Delay</u>: Time from Clock to new Q
Gates 2,3 then gates 5,6. $T_p = 2$

Timing Example



Minimum Clock Cycle?

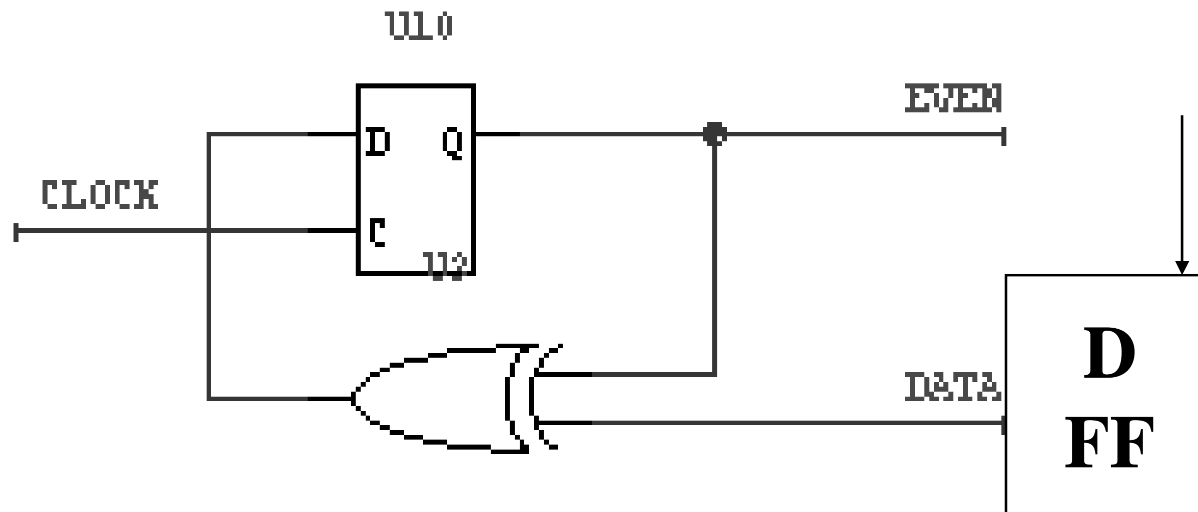
Tclock = Tprop + Txor + Tsu

Minimum Txor?

Txor = 0

the D-FF is inherently safe because Tprop > Thold

Dealing with System Inputs



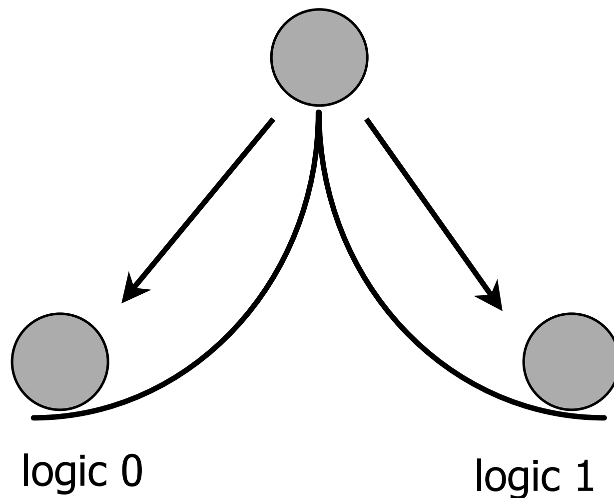
How do we make sure inputs don’t violate Tsu,Thold?

If inside the system – it is our design problem

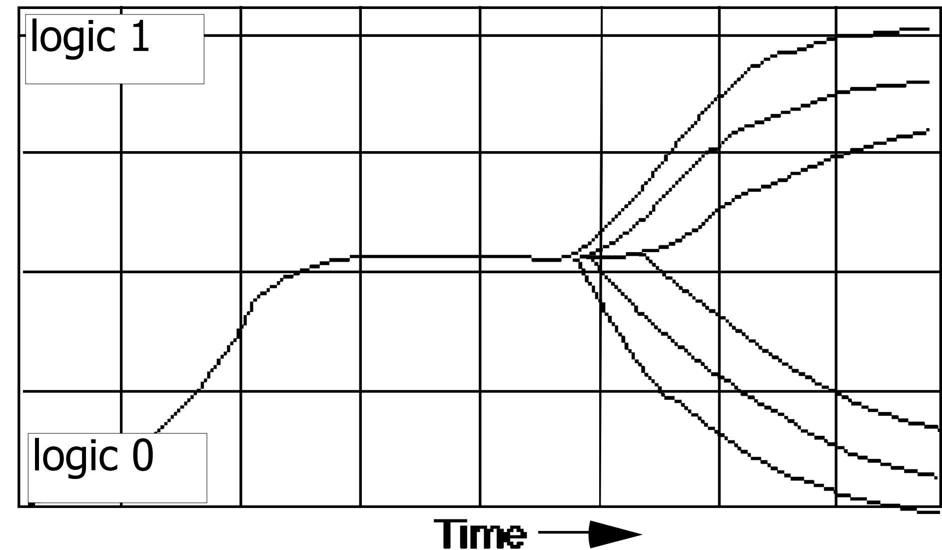
If from outside the system….add a synchronizer

Synchronization failure

- Occurs when FF input changes close to clock edge
  - the FF may enter a metastable state – neither a logic 0 nor 1 –
  - it may stay in this state an indefinite amount of time
  - this is not likely in practice but has some probability



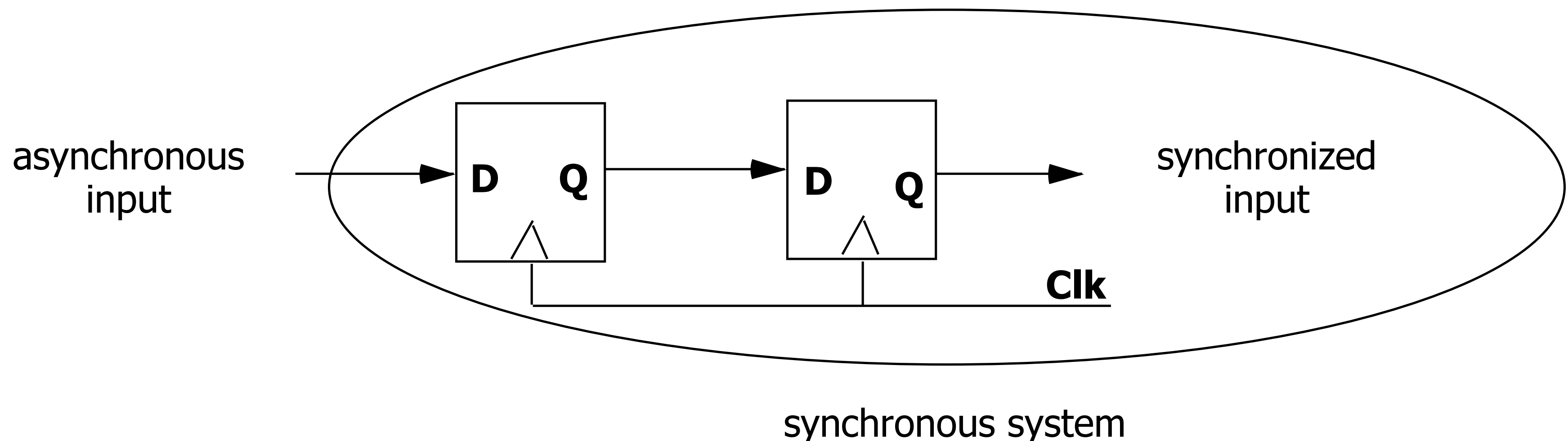
small, but non-zero probability that the FF output will get stuck in an in-between state



oscilloscope traces demonstrating synchronizer failure and eventual decay to steady state

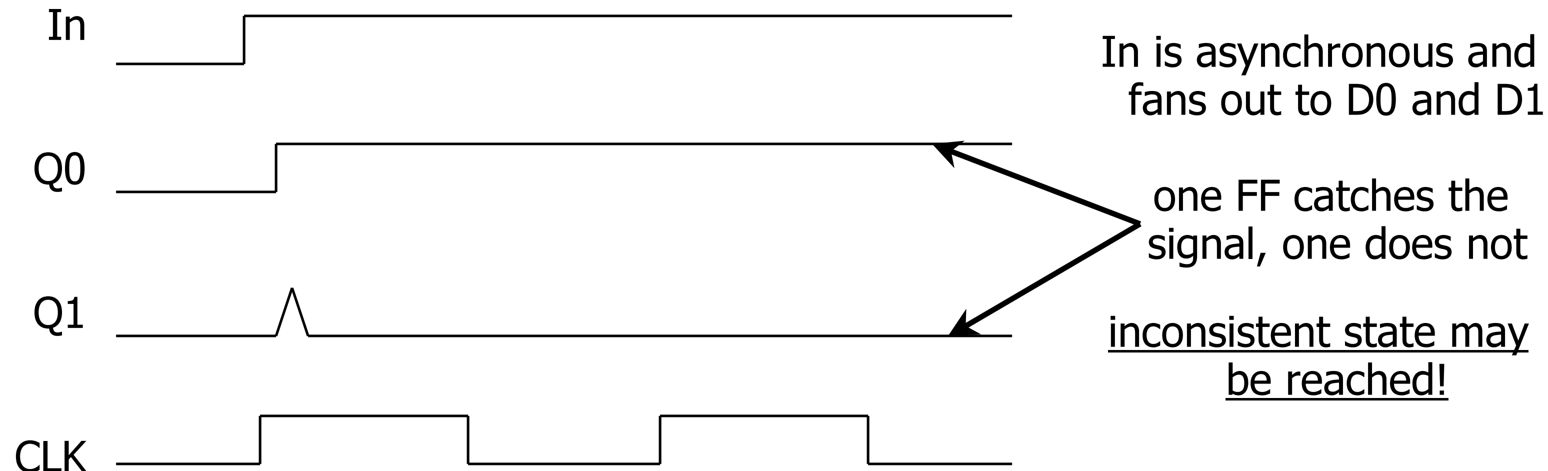
Dealing with synchronization failure

- Probability of failure can never be reduced to 0, but it can be reduced
  - (1) slow down the system clock
    this gives the synchronizer more time to decay into a steady state;
    synchronizer failure becomes a big problem for very high speed systems
  - (2) use fastest possible logic technology in the synchronizer
    this makes for a very sharp "peak" upon which to balance
  - (3) cascade two synchronizers
    this effectively synchronizes twice (both would have to fail)



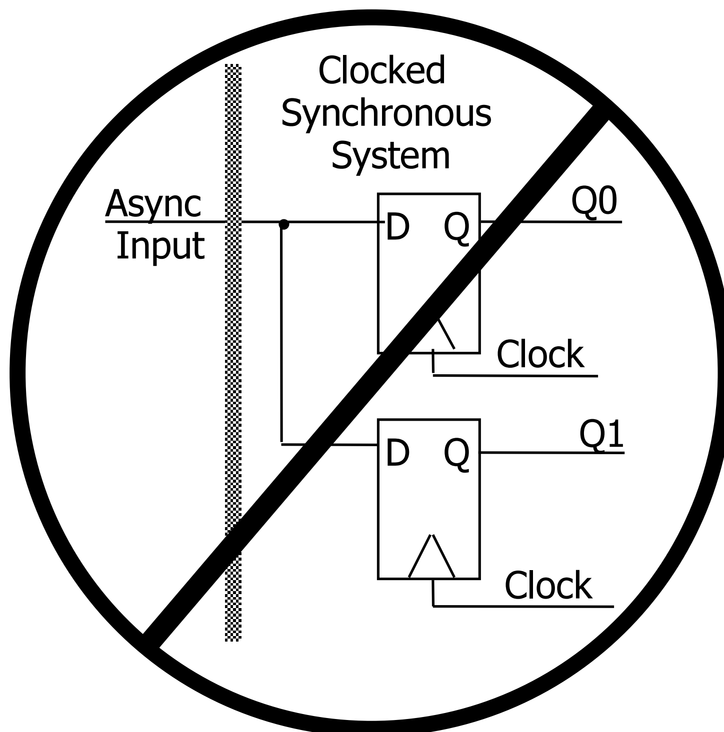
Handling asynchronous inputs (cont'd)

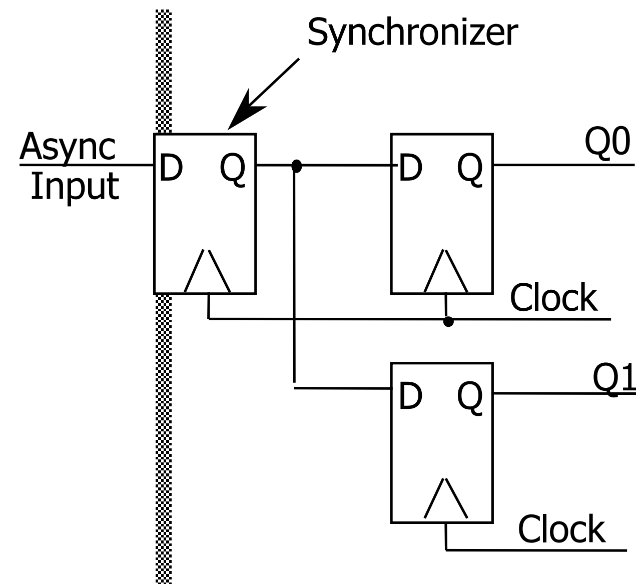
- What can go wrong?
  - input changes too close to clock edge (violating setup time constraint)



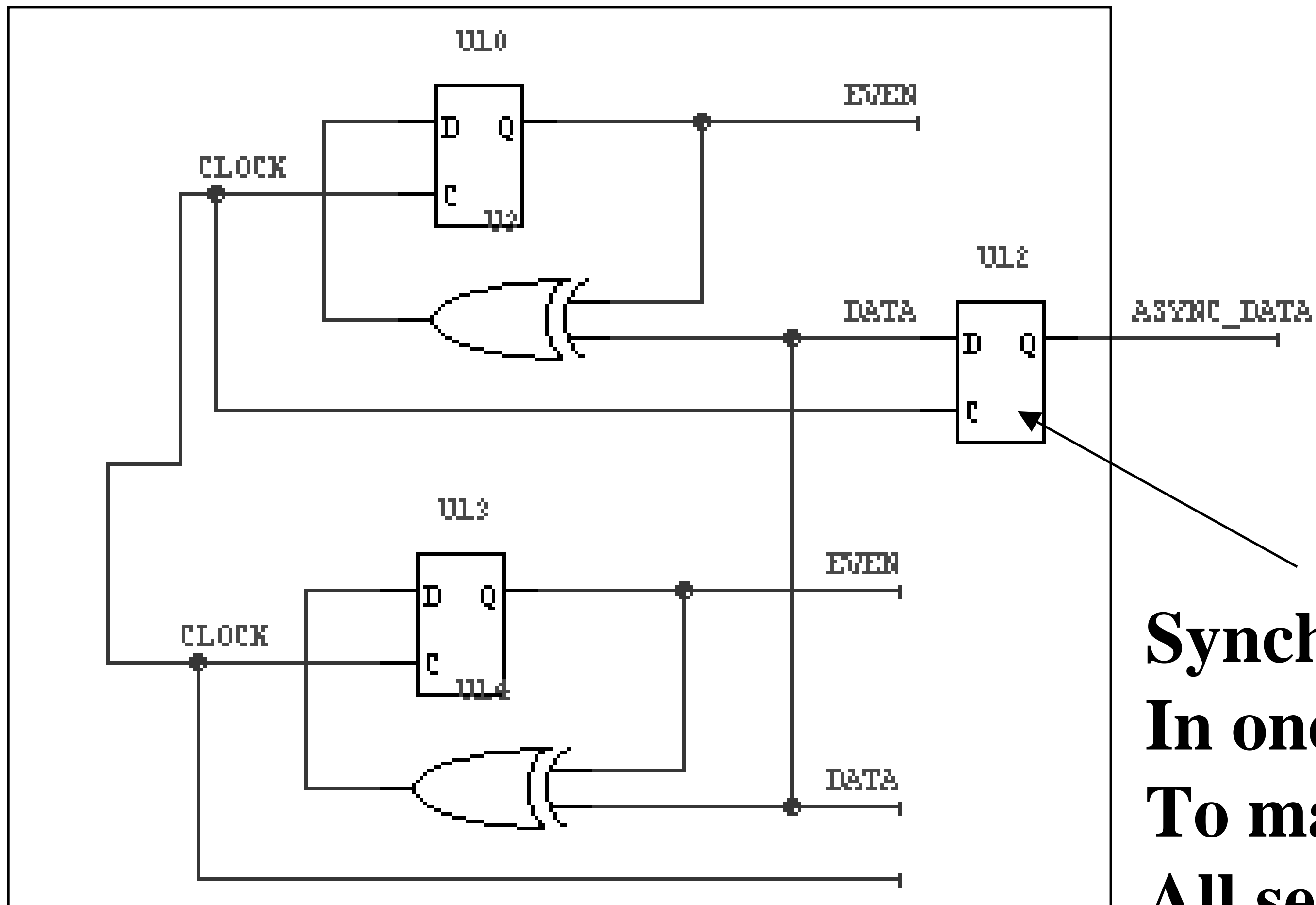
Handling asynchronous inputs

- Never allow asynchronous inputs to fan-out to more than one flip-flop
  - synchronize as soon as possible and then treat as synchronous signal



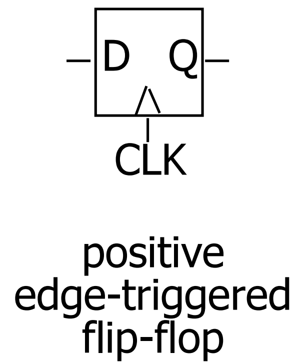


Synchronizing Asynchronous Inputs

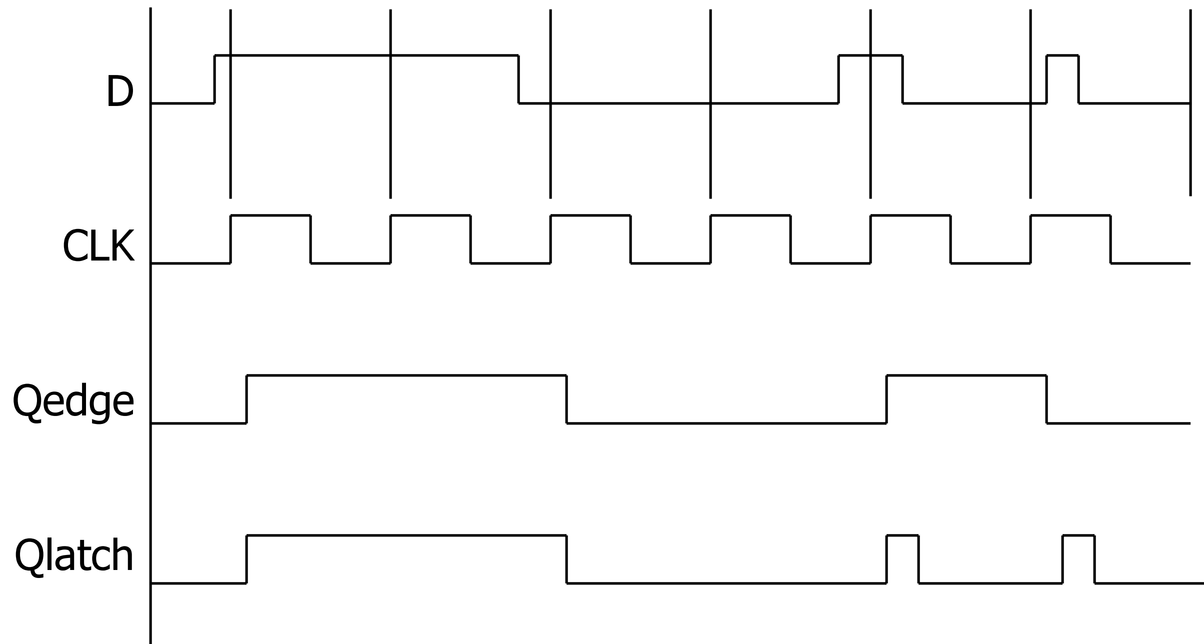


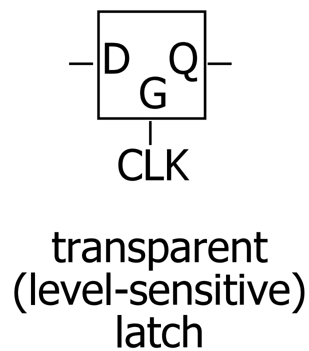
Synchronize
In one place
To make sure
All see same
input

Comparison of latches and flip-flops



positive
edge-triggered
flip-flop





transparent
(level-sensitive)
latch

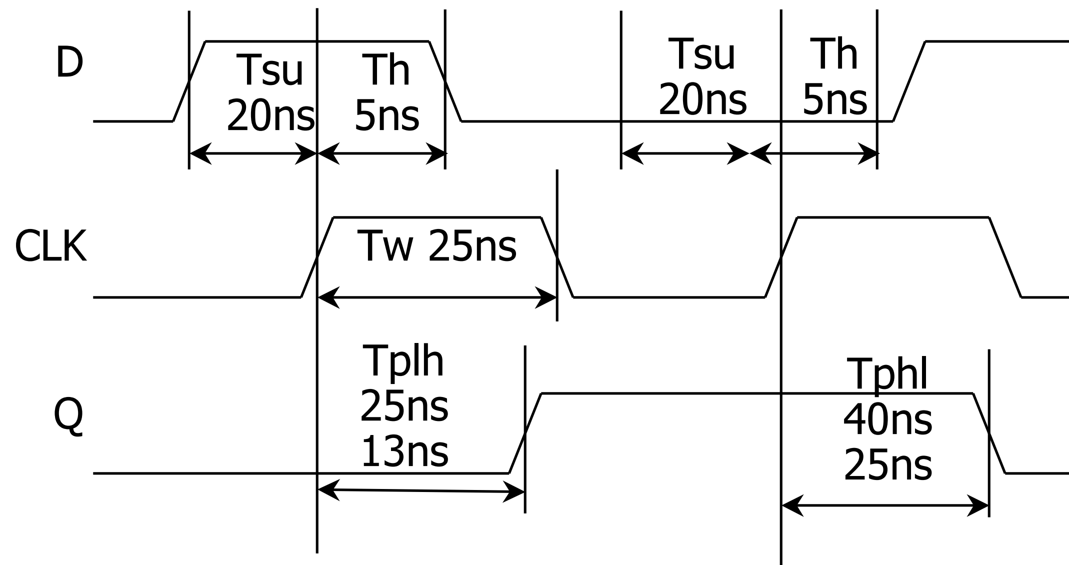
behavior is the same unless input changes
while the clock is high

Comparison of latches and flip-flops (cont'd)

Type	When inputs are sampled	When output is valid
unclocked latch	always	propagation delay from input change
level-sensitive latch	clock high (Tsu/Th around falling edge of clock)	propagation delay from input change or clock edge (whichever is later)
master-slave flip-flop	clock high (Tsu/Th around falling edge of clock)	propagation delay from falling edge of clock
negative edge-triggered flip-flop	clock hi-to-lo transition (Tsu/Th around falling edge of clock)	propagation delay from falling edge of clock
Asynch set/reset	Used to set/reset latch	propagation delay from rising edge

Typical timing specifications

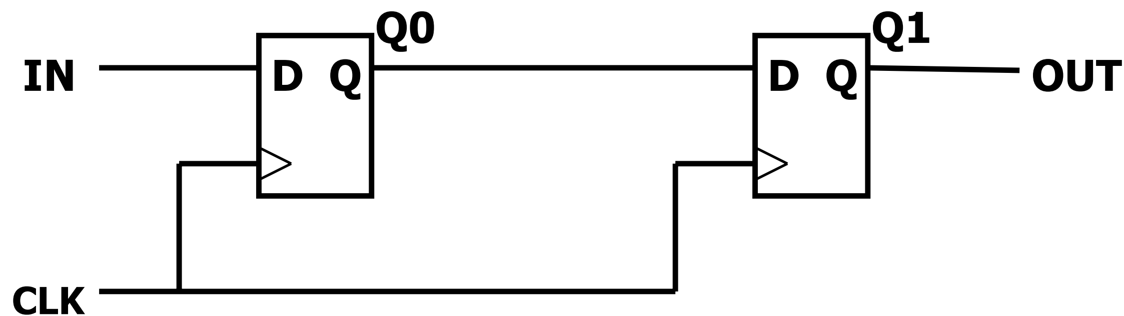
- Positive edge-triggered D flip-flop
  - setup and hold times
  - minimum clock width
  - propagation delays (low to high, high to low, max and typical)

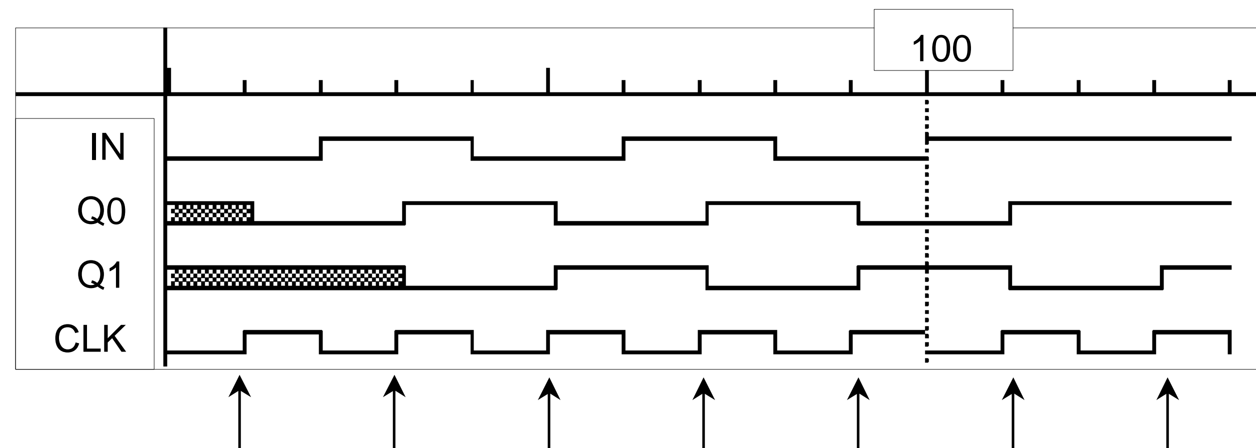


all measurements are made from the clocking event that is, the rising edge of the clock

Cascading edge-triggered flip-flops

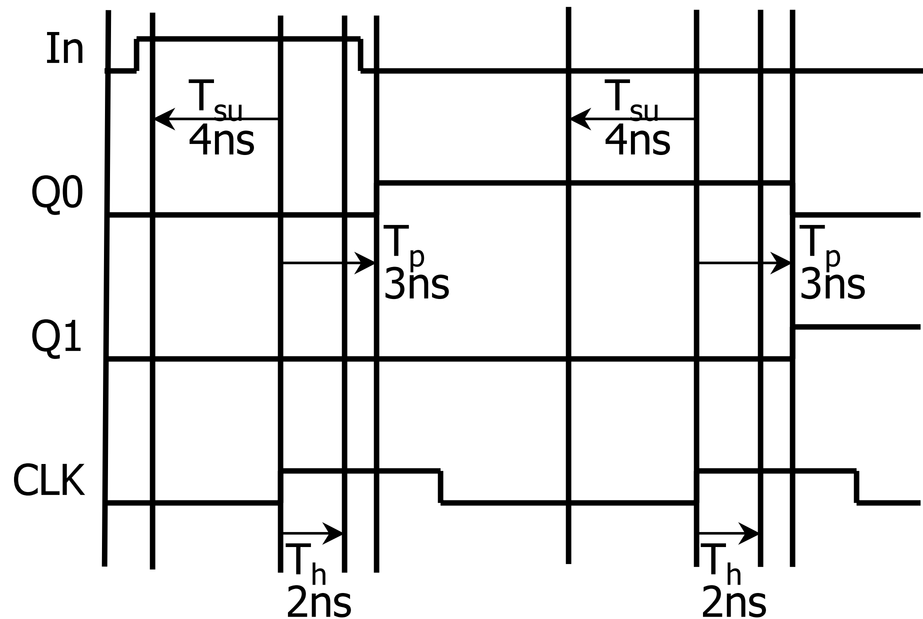
- Shift register
  - new value goes into first stage
  - while previous value of first stage goes into second stage
  - consider setup/hold/propagation delays (prop must be > hold)





Cascading edge-triggered flip-flops (cont'd)

- Why this works
  - propagation delays exceed hold times
  - clock width constraint exceeds setup time
  - this guarantees following stage will latch current value before it changes to new value



timing constraints guarantee proper operation of cascaded components

assumes infinitely fast distribution of the clock

Summary of latches and flip-flops

- Development of D-FF
  - level-sensitive used in custom integrated circuits
    - can be made with 4 switches
  - edge-triggered used in programmable logic devices
  - good choice for data storage register
- Historically J-K FF was popular but now never used
  - similar to R-S but with 1-1 being used to toggle output (complement state)
  - good in days of TTL/SSI (more complex input function: Q+ = JQ' + K'Q
  - not a good choice for PALs/PLAs as it requires 2 inputs
  - can always be implemented using D-FF
- Preset and clear inputs are highly desirable on flip-flops
  - used at start-up or to reset system to a known state
- T-Flip Flop: Q+ = Q xor T
- Terminology:
  - Level-Sensitive = Latch
  - Edge Triggered = Register