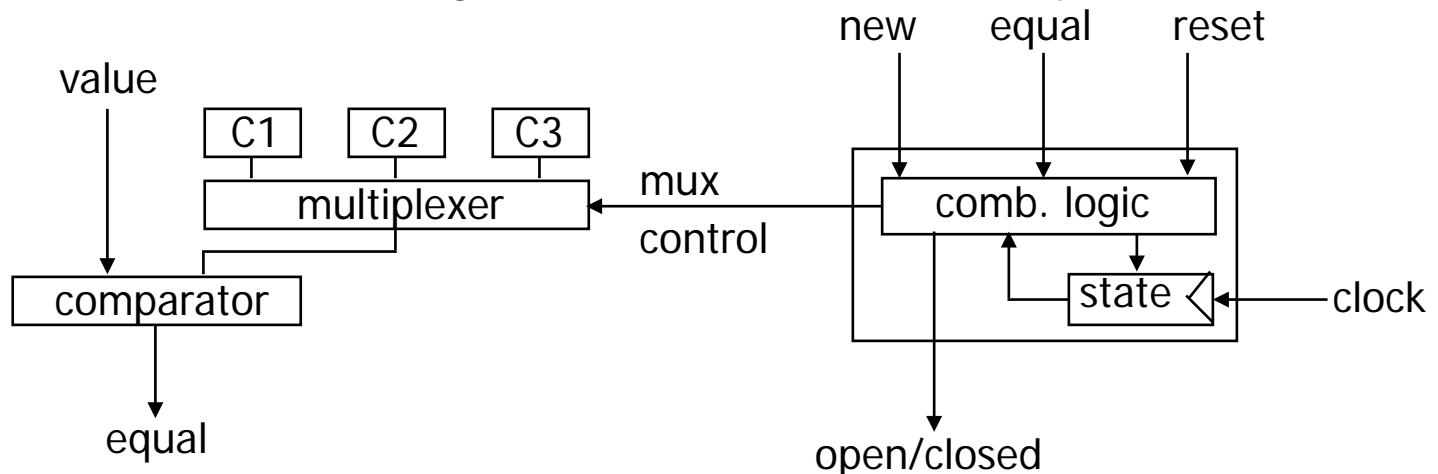


# Sequential logic

- 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
  - simple counters
- Hardware description languages and sequential logic

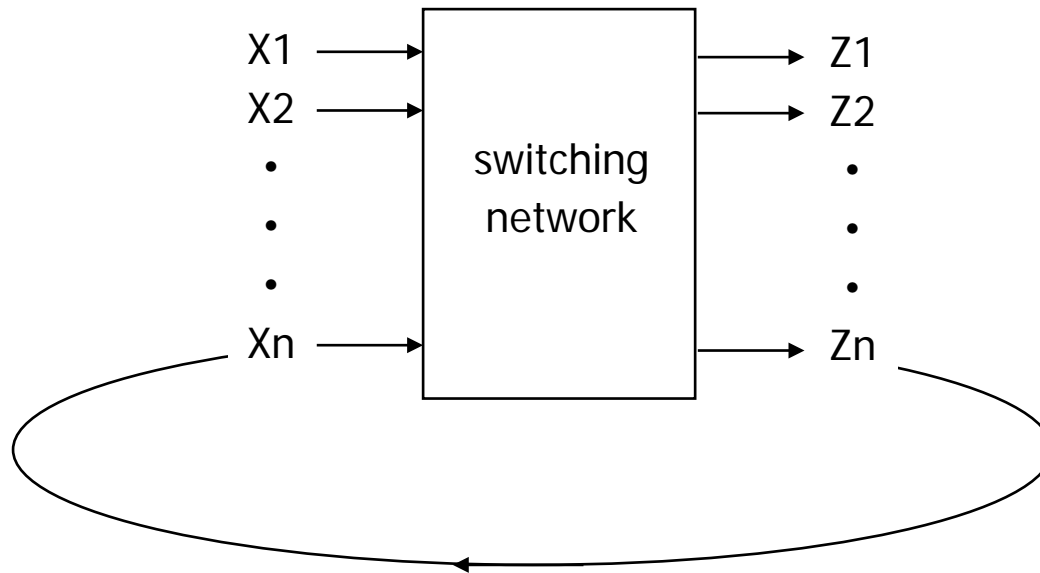
# Sequential circuits

- Circuits with feedback
  - outputs =  $f(\text{inputs}, \text{past inputs}, \text{past outputs})$
  - basis for building "memory" into logic circuits
  - door combination lock is an example of a sequential circuit
    - state is memory
    - state is an "output" and an "input" to combinational logic
    - combination storage elements are also memory



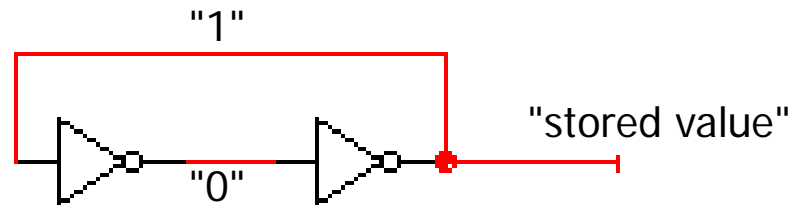
# Circuits with feedback

- How to control feedback?
  - what stops values from cycling around endlessly

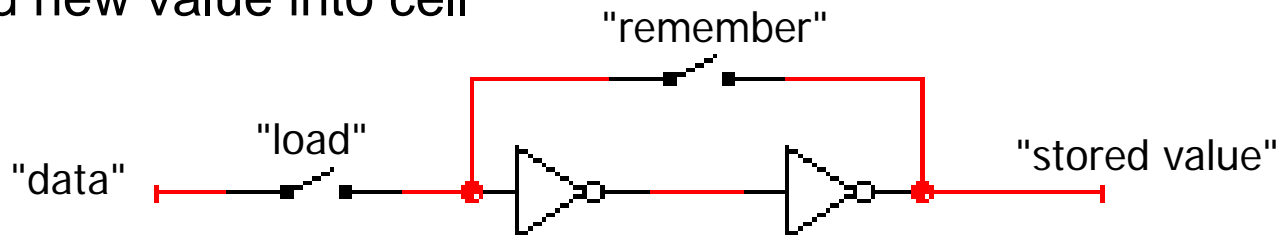


# Simplest circuits with feedback

- Two inverters form a static memory cell
  - will hold value as long as it has power applied



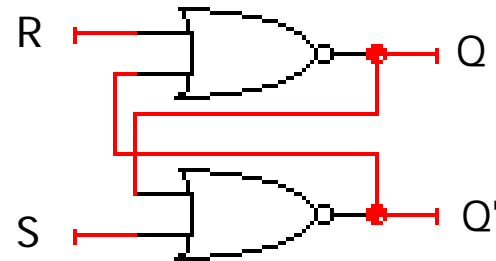
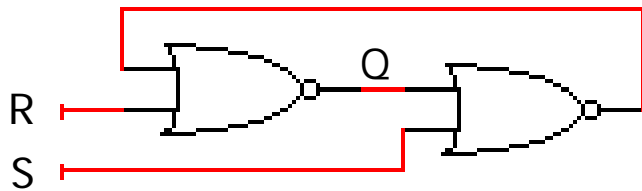
- How to get a new value into the memory cell?
  - selectively break feedback path
  - load new value into cell



# Memory with cross-coupled gates

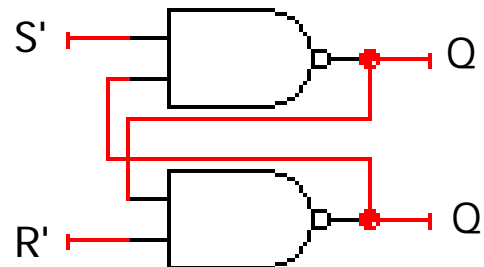
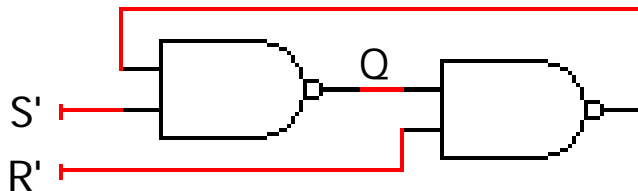
## ■ Cross-coupled NOR gates

- similar to inverter pair, with capability to force output to 0 (reset=1) or 1 (set=1)

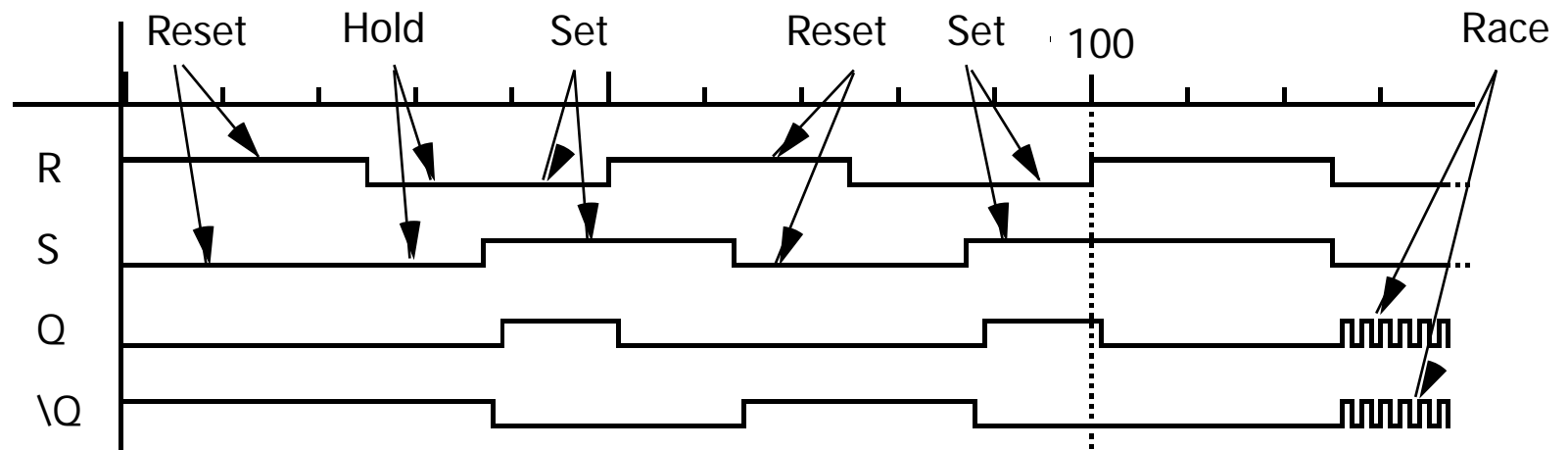
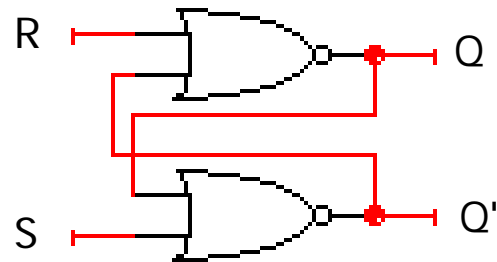


## ■ Cross-coupled NAND gates

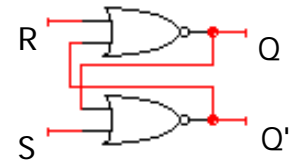
- similar to inverter pair, with capability to force output to 0 (reset=0) or 1 (set=0)



# 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

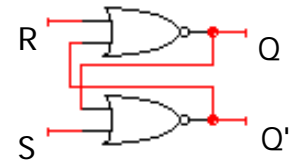
Q Q'  
0 1

Q Q'  
1 0

Q Q'  
0 0

Q Q'  
1 1

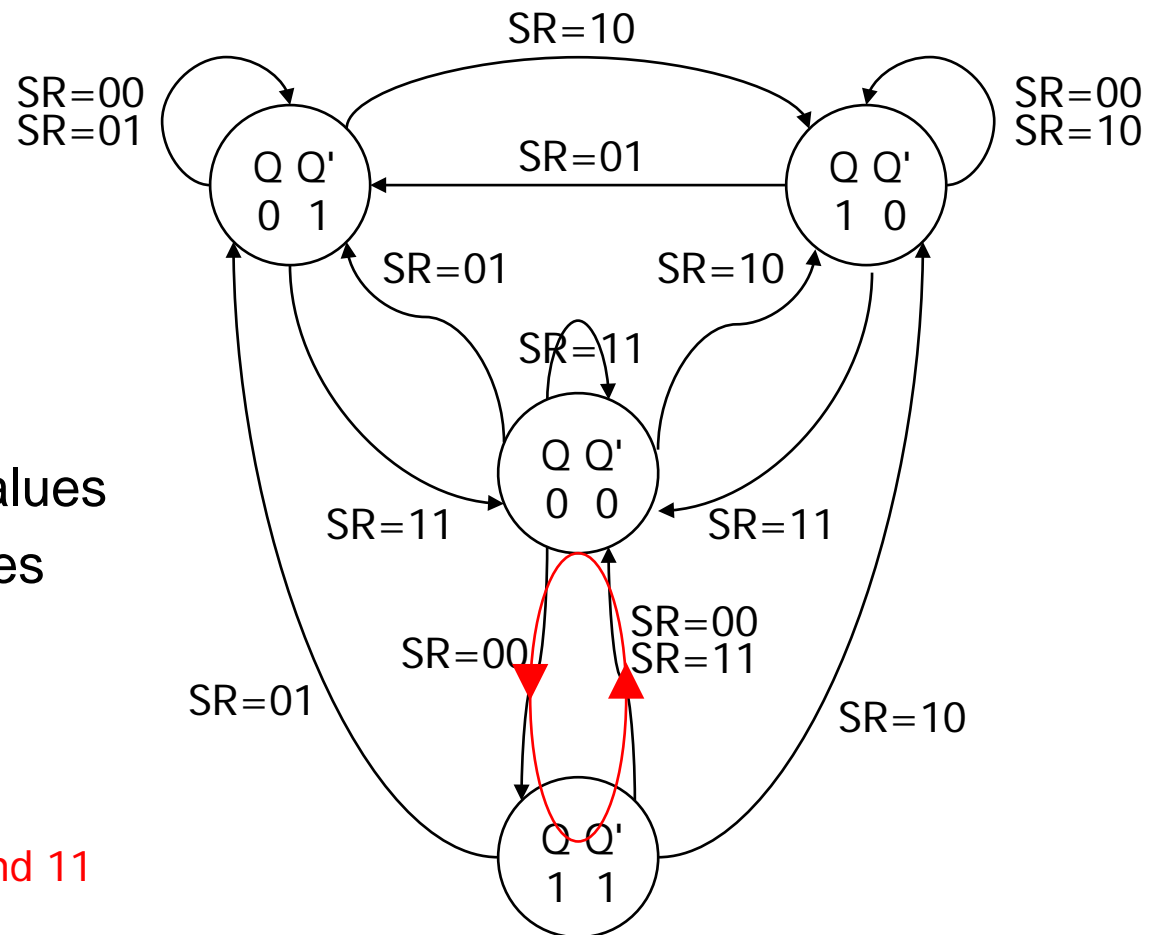
# Theoretical R-S latch behavior



## ■ State diagram

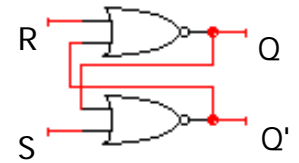
- states: possible values
- transitions: changes based on inputs

possible oscillation  
between states 00 and 11

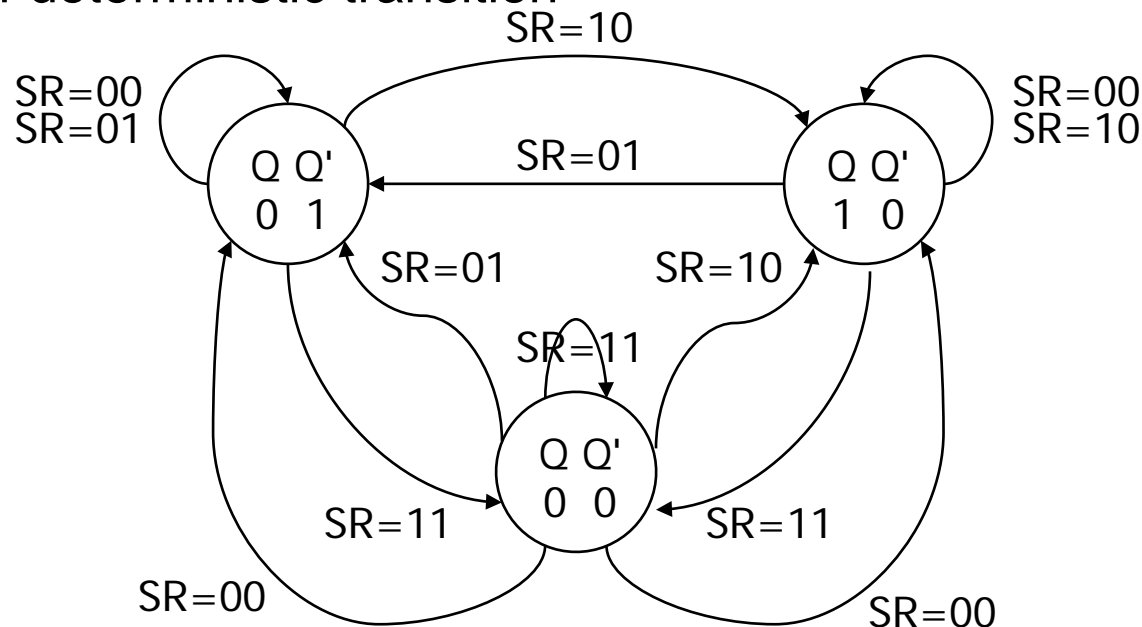




# Observed R-S latch behavior

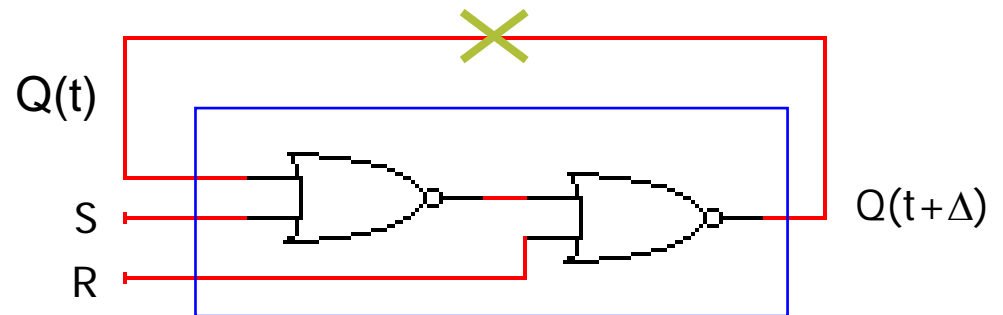
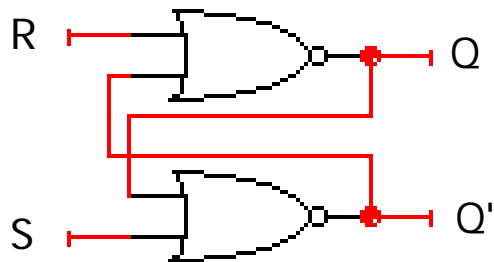


- Very difficult to observe R-S latch in the 1-1 state
  - one of R or S usually changes first
- Ambiguously returns to state 0-1 or 1-0
  - a so-called "race condition"
  - or non-deterministic transition



# R-S latch analysis

## ■ Break feedback path

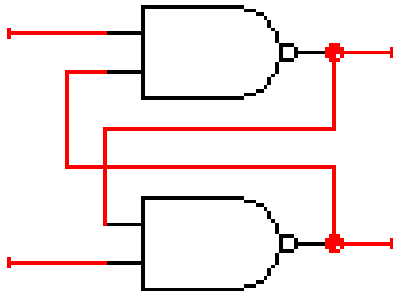


S	R	Q(t)	Q(t+Δ)	
0	0	0	0	hold
0	0	1	1	
0	1	0	0	reset
0	1	1	0	
1	0	0	1	set
1	0	1	1	
1	1	0	X	not allowed
1	1	1	X	

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

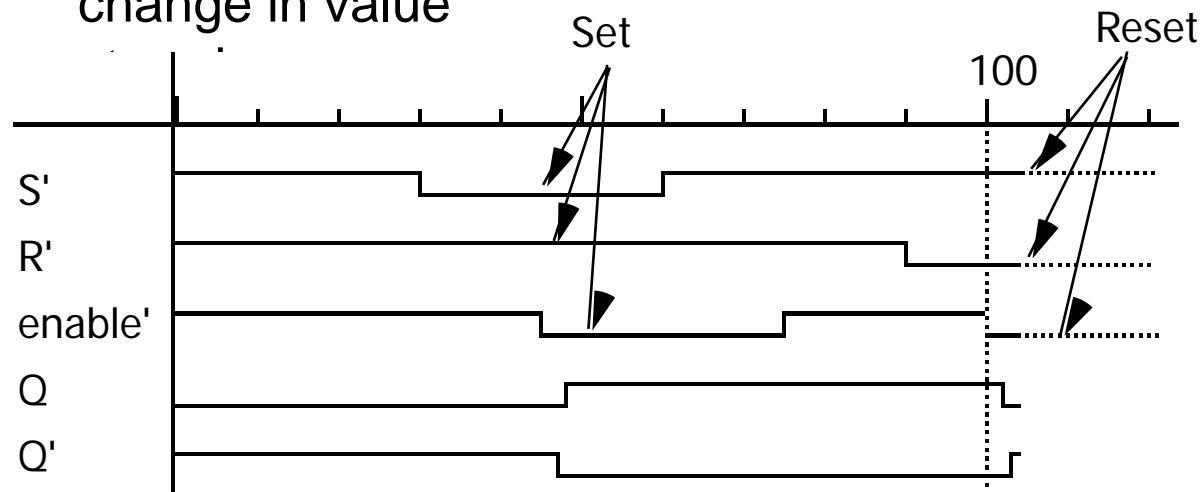
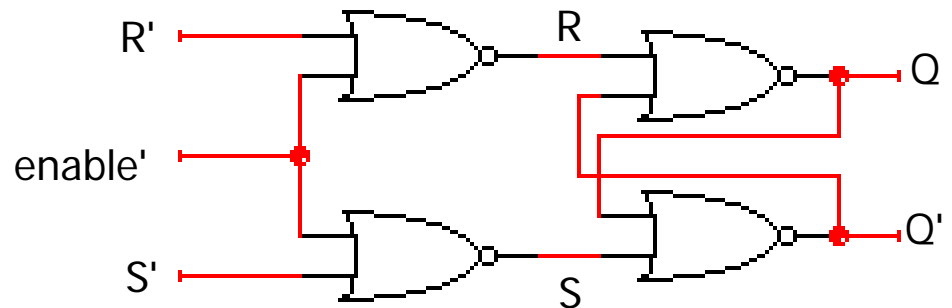
characteristic equation  
 $Q(t+\Delta) = S + R' Q(t)$

# Activity: R-S latch using NAND gates



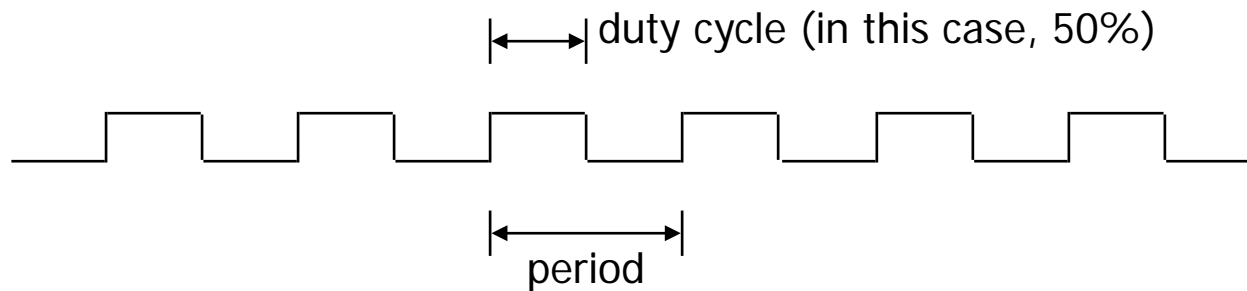
# Gated R-S latch

- Control when R and S inputs matter
  - otherwise, the slightest glitch on R or S while enable is low could cause change in value



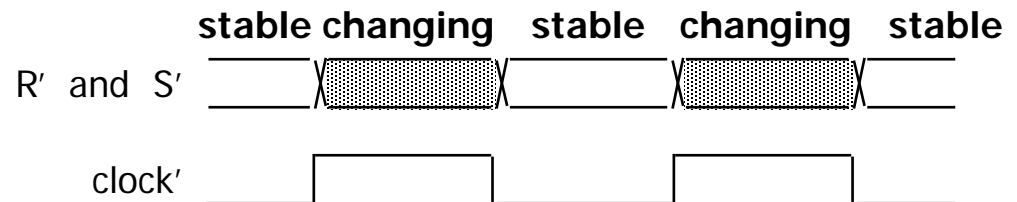
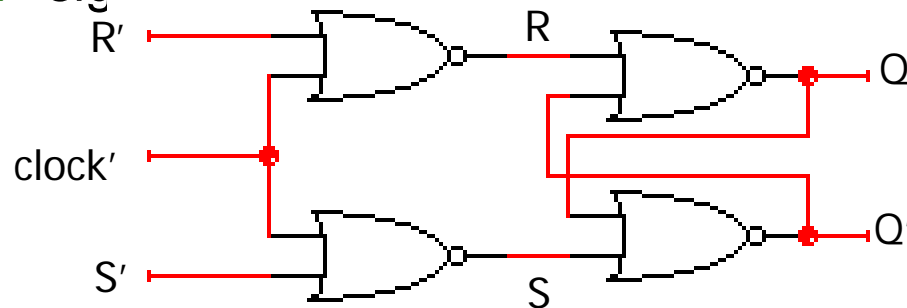
# Clocks

- Used to keep time
  - wait long enough for inputs (R' and S') to settle
  - then allow to have effect on value stored
- Clocks are regular periodic signals
  - period (time between ticks)
  - duty-cycle (time clock is high between ticks - expressed as % of period)



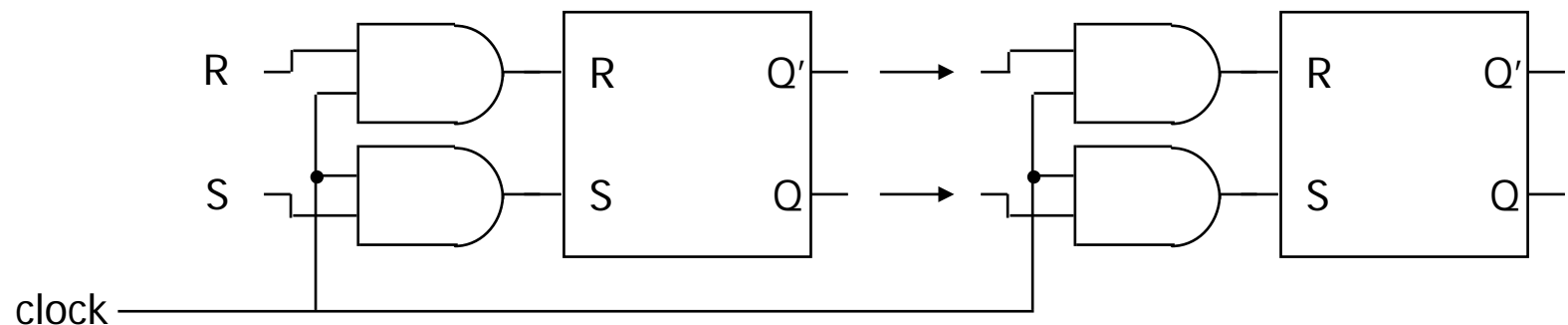
# Clocks (cont'd)

- Controlling an R-S latch with a clock
  - can't let R and S change while clock is active (allowing R and S to pass)
  - only have half of clock period for signal changes to propagate
  - sig



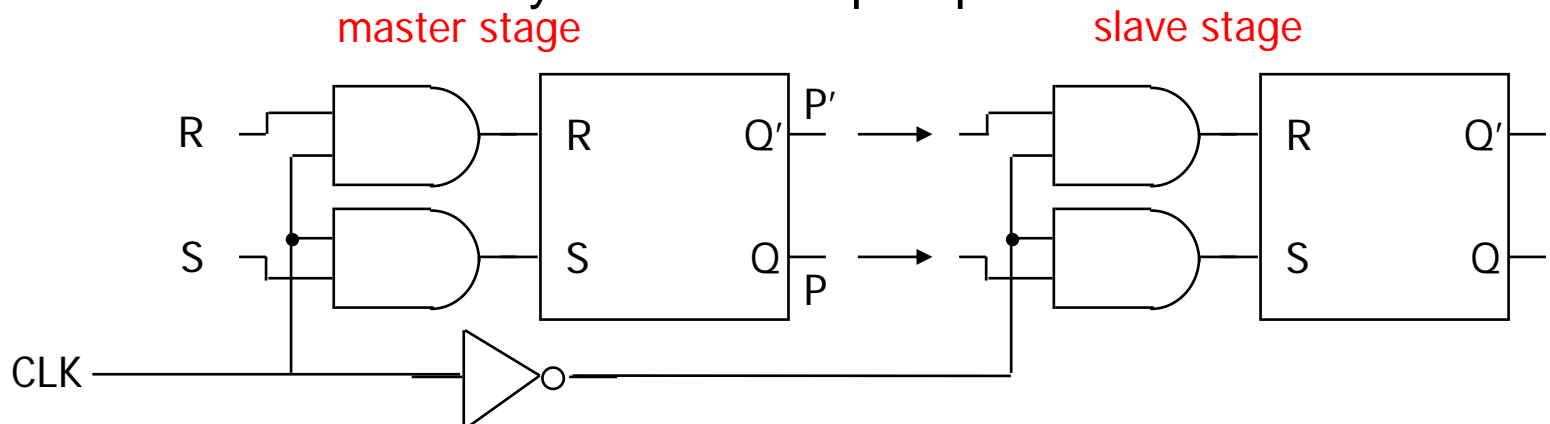
# Cascading latches

- Connect output of one latch to input of another
- How to stop changes from racing through chain?
  - need to be able to control flow of data from one latch to the next
  - move one latch per clock period
  - have to worry about logic between latches (arrows) that is too fast



# Master-slave structure

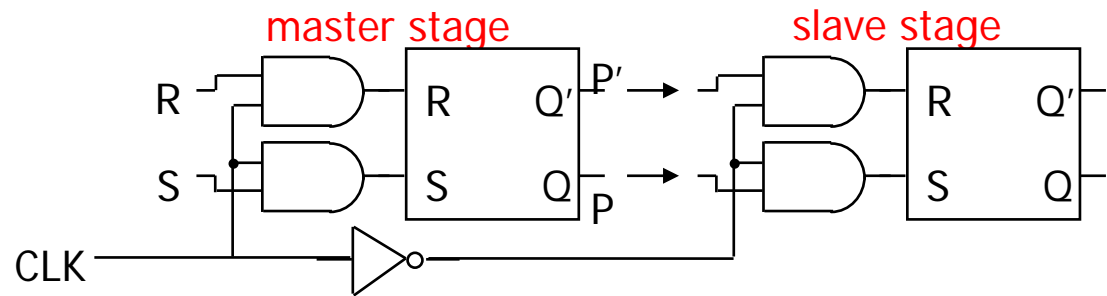
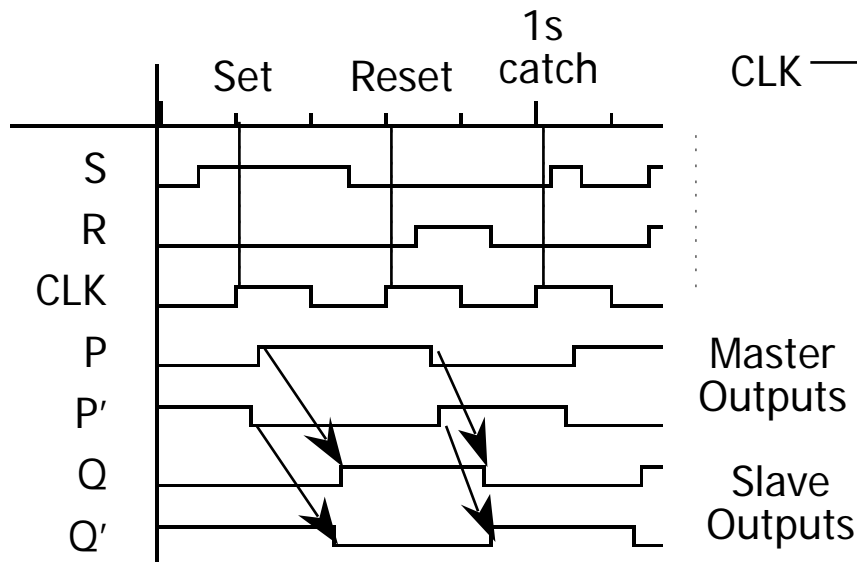
- Break flow by alternating clocks (like an air-lock)
  - use positive clock to latch inputs into one R-S latch
  - use negative clock to change outputs with another R-S latch
- View pair as one basic unit
  - master-slave flip-flop
  - twice as much logic
  - output changes a few gate delays after the falling edge of clock but does not affect any cascaded flip-flops





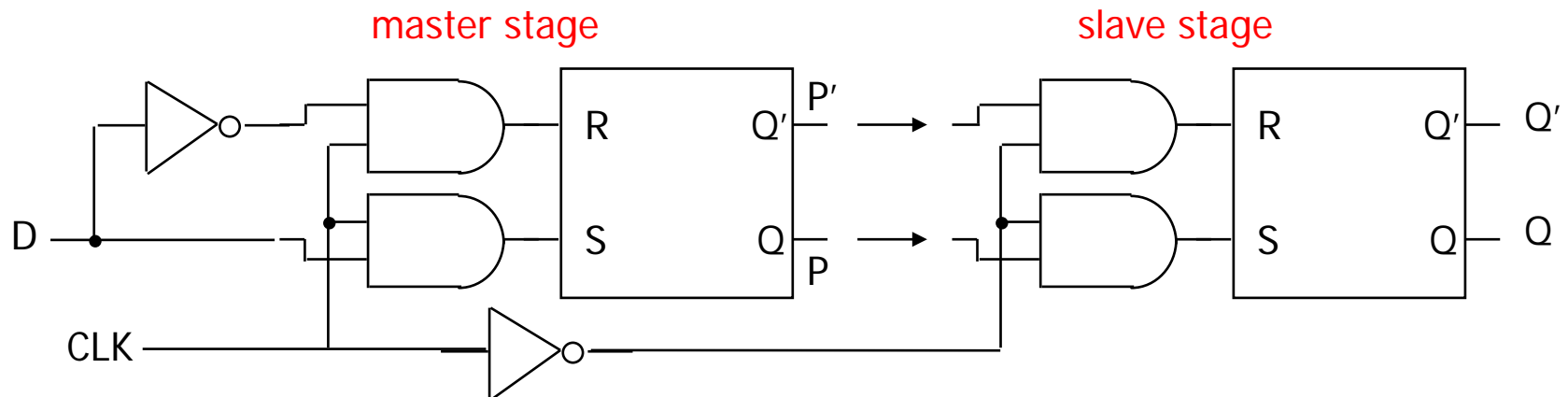
# The 1s catching problem

- 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

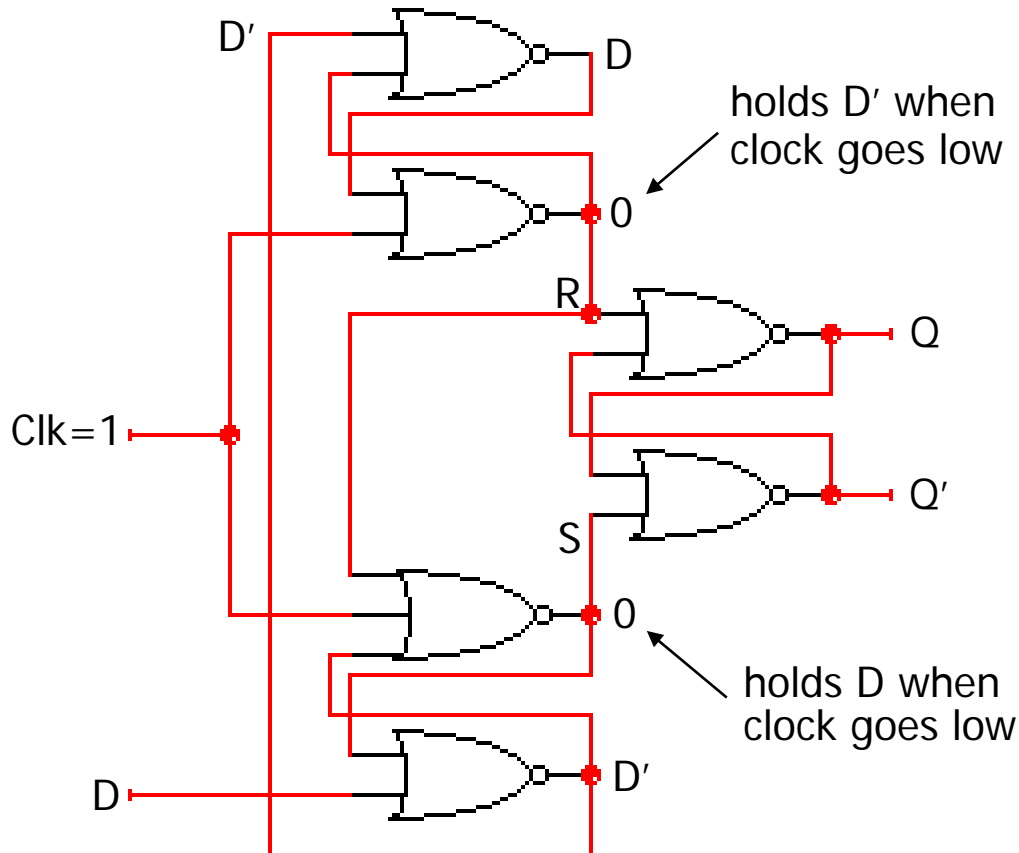
- Make S and R complements of each other
  - ❑ eliminates 1s catching problem
  - ❑ can't just hold previous value  
(must have new value ready every clock period)
  - ❑ value of D just before clock goes low is what is stored in flip-flop
  - ❑ can make R-S flip-flop by adding logic to make  $D = S +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



10 gates

# Edge-triggered flip-flops

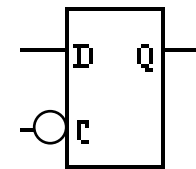
- More efficient solution: only 6 gates
  - sensitive to inputs only near edge of clock signal (not while high)



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

4-5 gate delays

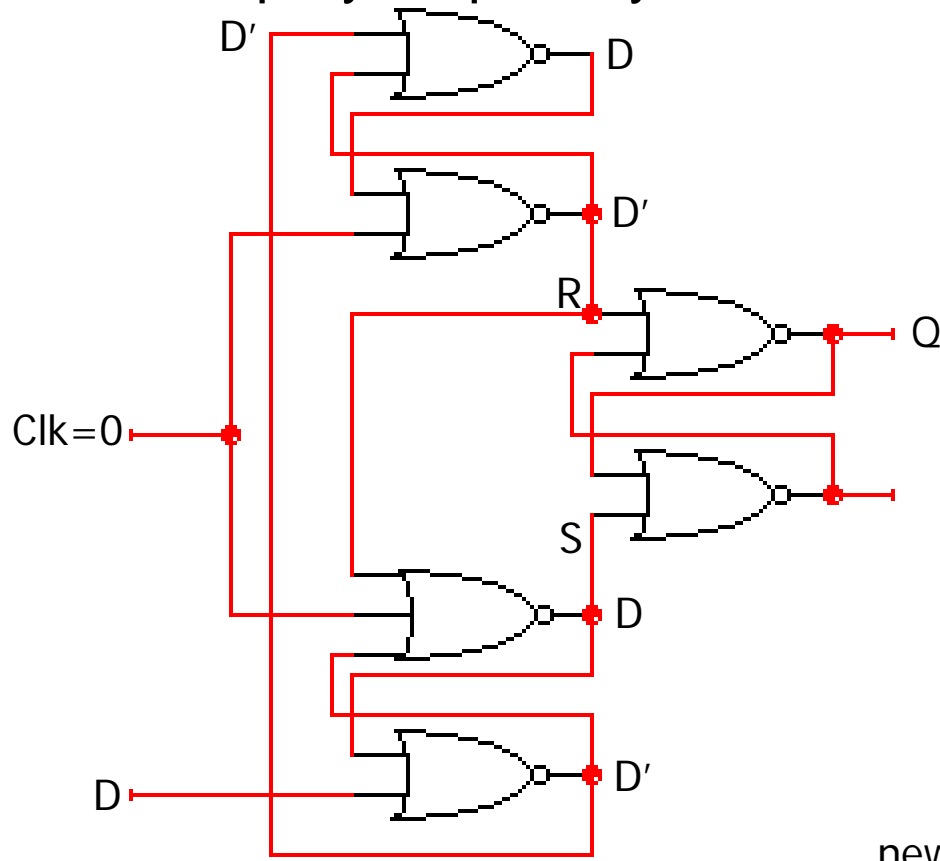
must respect setup and hold time  
constraints to successfully  
capture input



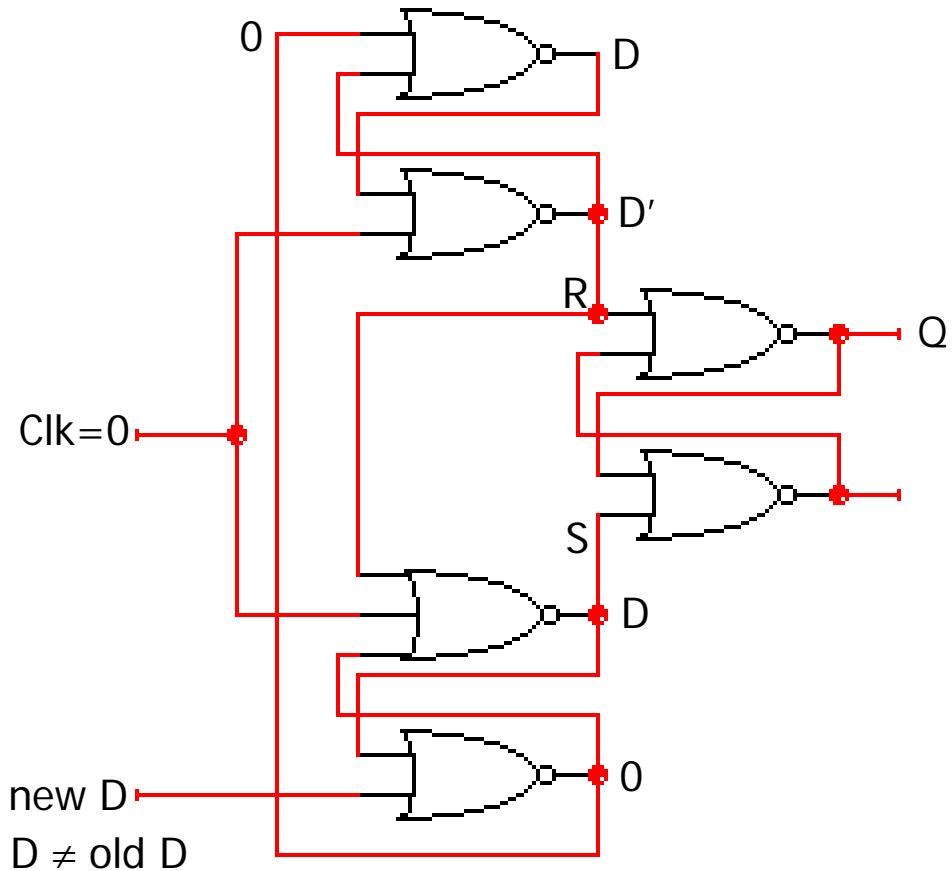
characteristic equation  
 $Q(t+1) = D$

## Edge-triggered flip-flops (cont'd)

- Step-by-step analysis



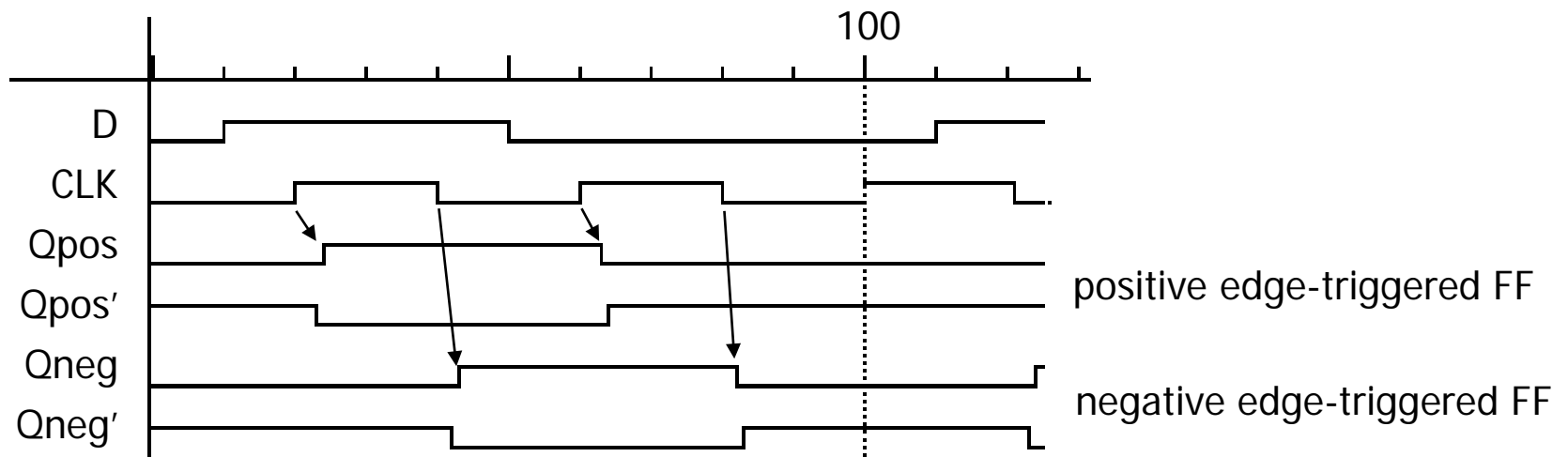
when clock goes high-to-low  
data is latched



when clock is low  
data is held

# Edge-triggered flip-flops (cont'd)

- Positive edge-triggered
  - inputs sampled on rising edge; outputs change after rising edge
- Negative edge-triggered flip-flops
  - inputs sampled on falling edge; outputs change after falling edge



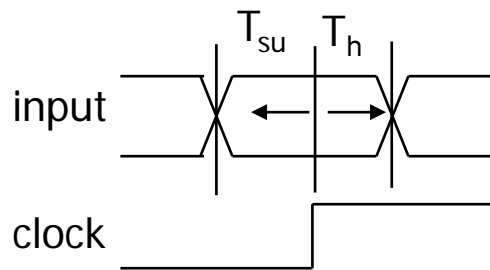
# Timing methodologies

- Rules for interconnecting components and clocks
  - guarantee proper operation of system when strictly followed
- Approach depends on building blocks used for memory elements
  - we'll focus on systems with edge-triggered flip-flops
    - found in programmable logic devices
  - many custom integrated circuits focus on level-sensitive latches
- Basic rules for correct timing:
  - (1) correct inputs, with respect to time, are provided to the flip-flops
  - (2) no flip-flop changes state more than once per clocking event

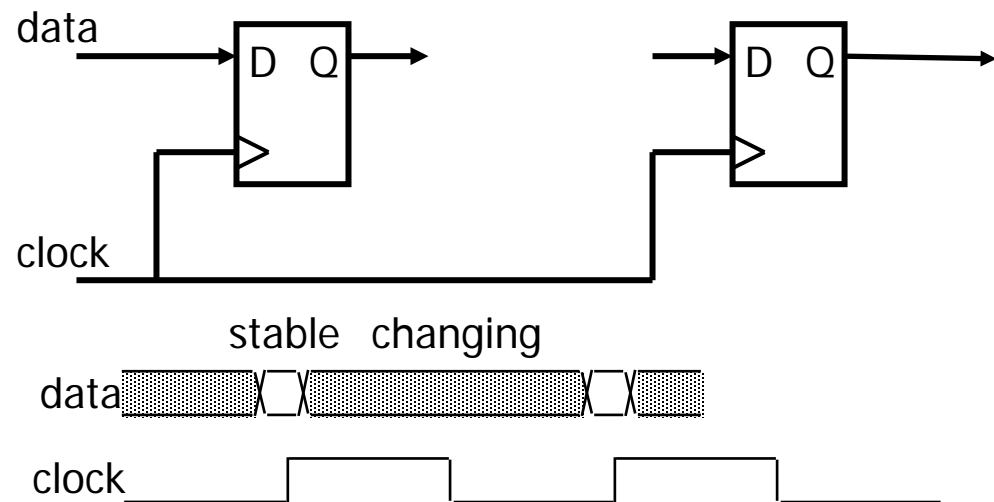
# Timing methodologies (cont'd)

## ■ Definition of terms

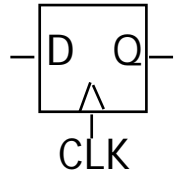
- clock: periodic event, causes state of memory element to change  
can be rising edge or falling edge or high level or low level
- setup time: minimum time before the clocking event by which the input must be stable ( $T_{su}$ )
- hold time: minimum time after the clocking event until which the input must remain stable ( $T_h$ )



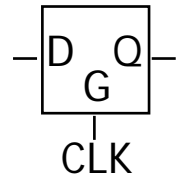
there is a timing "window" around the clocking event during which the input must remain stable and unchanged in order to be recognized



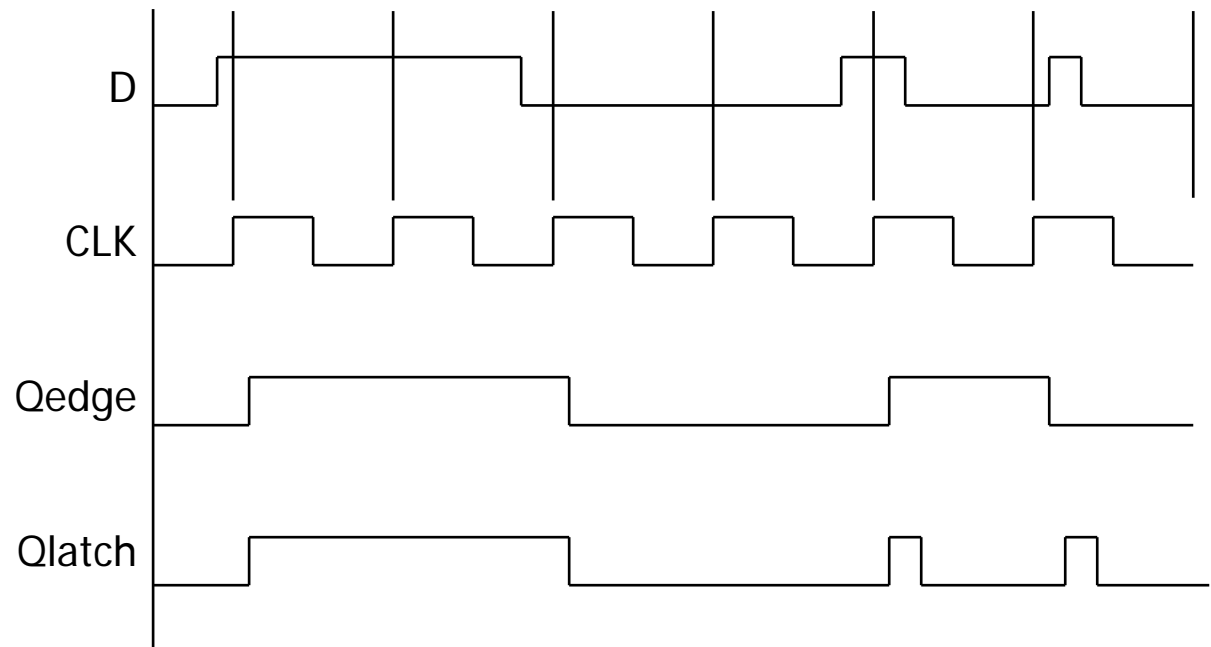
# Comparison of latches and flip-flops



positive  
edge-triggered  
flip-flop



transparent  
(level-sensitive)  
latch



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

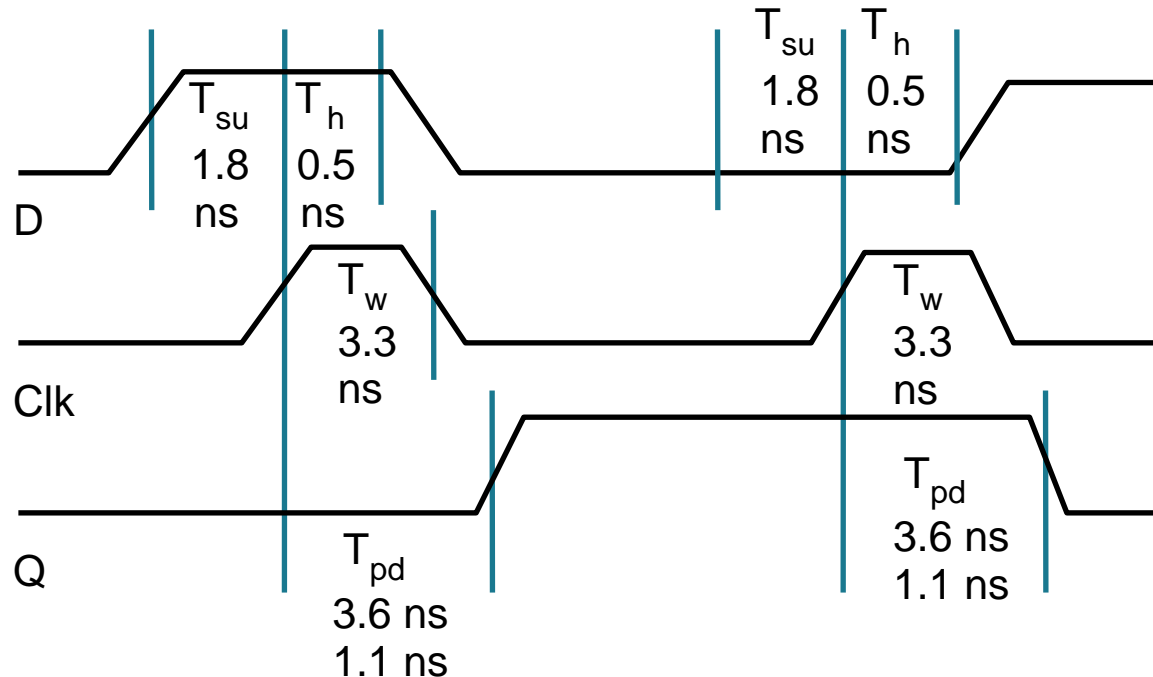


# Comparison of latches and flip-flops (cont'd)

<u>Type</u>	<u>When inputs are sampled</u>	<u>When output is valid</u>
unclocked latch	always	propagation delay from input change
level-sensitive latch	clock high ( $T_{su}/T_h$ around falling edge of clock)	propagation delay from input change or clock edge (whichever is later)
master-slave flip-flop	clock high ( $T_{su}/T_h$ around falling edge of clock)	propagation delay from falling edge of clock
negative edge-triggered flip-flop	clock hi-to-lo transition ( $T_{su}/T_h$ around falling edge of clock)	propagation delay from falling edge of clock

# 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)

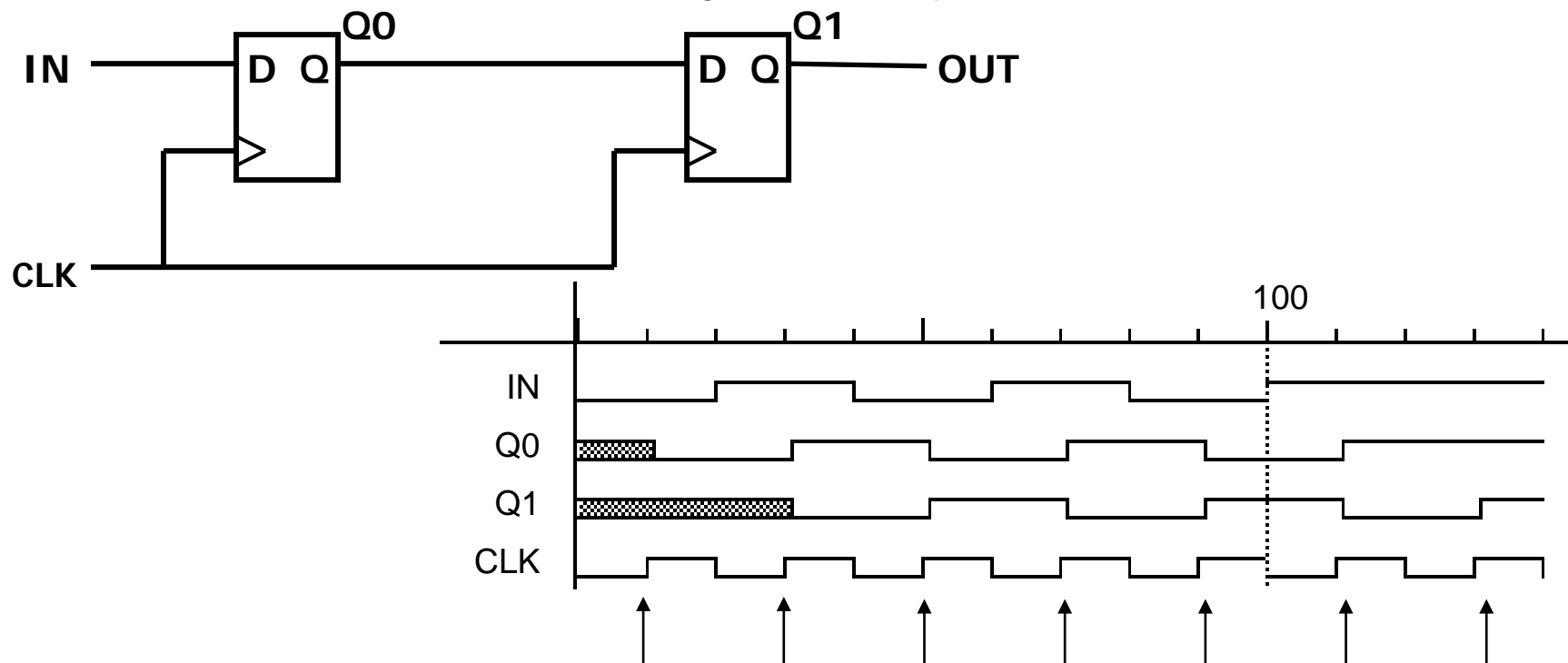


all measurements are made from the clocking event (the rising edge of the clock)

# Cascading edge-triggered flip-flops

## ■ Shift register

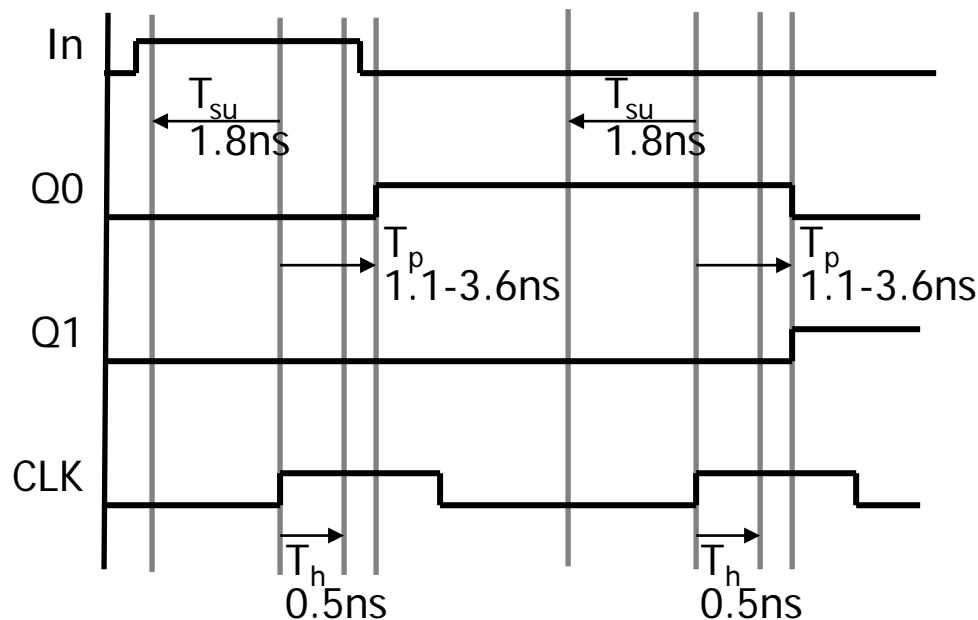
- new value goes into first stage
- while previous value of first stage goes into second stage
- consider setup/hold/propagation delays (prop must be  $>$  hold)



# 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



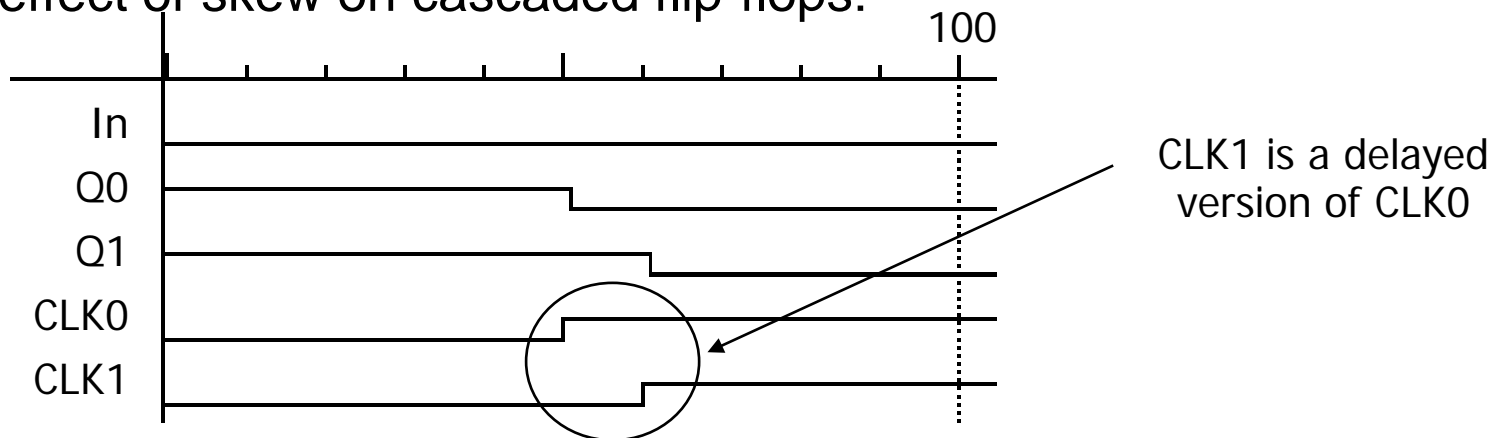
timing constraints  
guarantee proper  
operation of  
cascaded components

assumes infinitely fast  
distribution of the clock

# Clock skew

## ■ The problem

- correct behavior assumes next state of all storage elements determined by all storage elements at the same time
- this is difficult in high-performance systems because time for clock to arrive at flip-flop is comparable to delays through logic
- effect of skew on cascaded flip-flops:



original state:  $IN = 0$ ,  $Q0 = 1$ ,  $Q1 = 1$

due to skew, next state becomes:  $Q0 = 0$ ,  $Q1 = 0$ , and not  $Q0 = 0$ ,  $Q1 = 1$

# Summary of latches and flip-flops

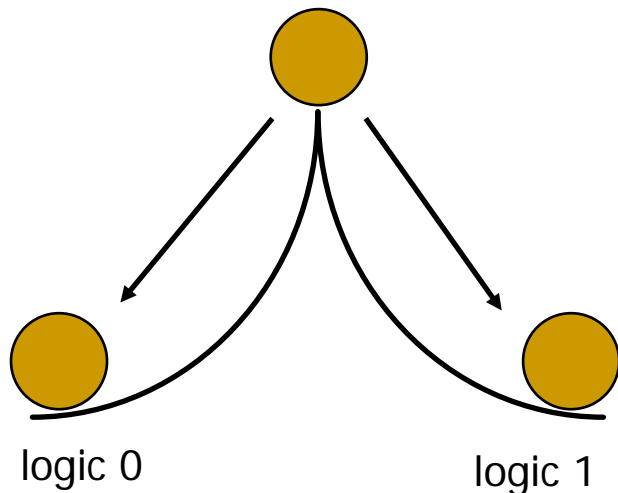
- Development of D-FF
  - level-sensitive used in custom integrated circuits
    - can be made with 4 switches
  - edge-triggered used in programmable logic devices
  - good choice for data storage register
- Historically J-K FF was popular but now never used
  - similar to R-S but with 1-1 being used to toggle output (complement state)
  - good in days of TTL/SSI (more complex input function:  $D = J Q' + K' Q$ )
  - 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

# Metastability and asynchronous inputs

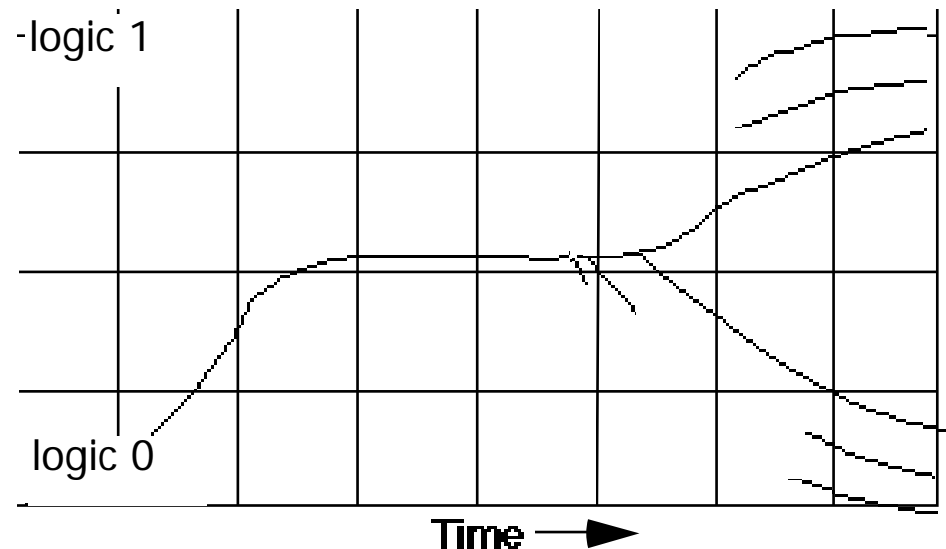
- Clocked synchronous circuits
  - inputs, state, and outputs sampled or changed in relation to a common reference signal (called the clock)
  - e.g., master/slave, edge-triggered
- Asynchronous circuits
  - inputs, state, and outputs sampled or changed independently of a common reference signal (glitches/hazards a major concern)
  - e.g., R-S latch
- Asynchronous inputs to synchronous circuits
  - inputs can change at any time, will not meet setup/hold times
  - dangerous, synchronous inputs are greatly preferred
  - cannot be avoided (e.g., reset signal, memory wait, user input)

# 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

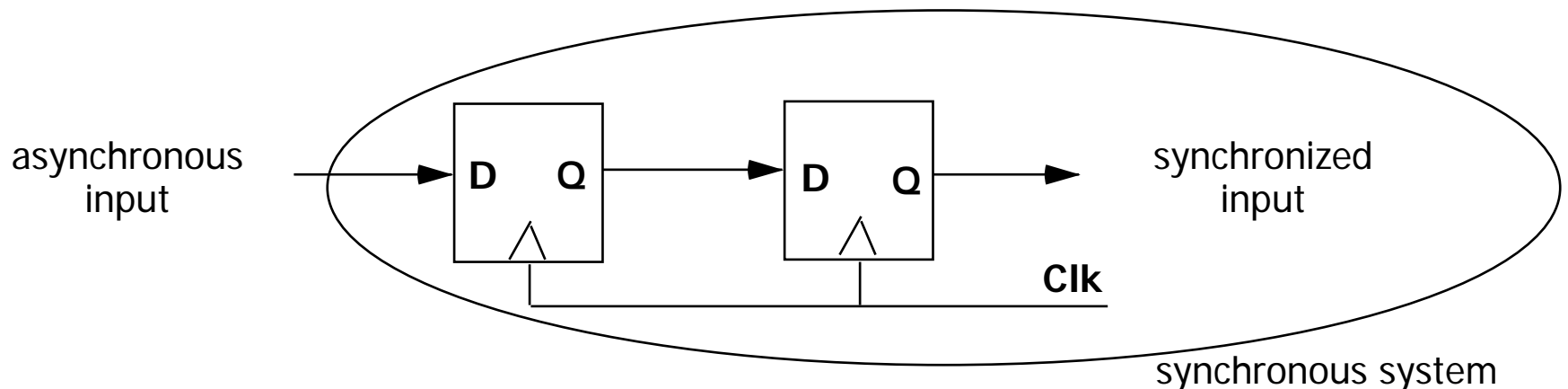


oscilloscope traces demonstrating  
synchronizer failure and eventual  
decay to steady state



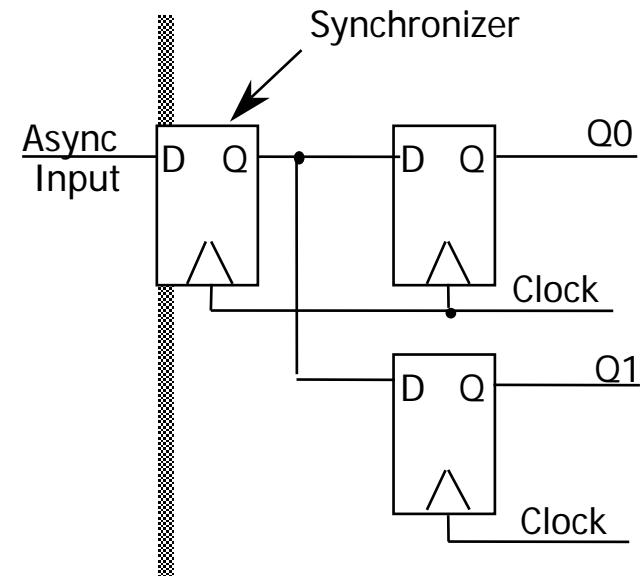
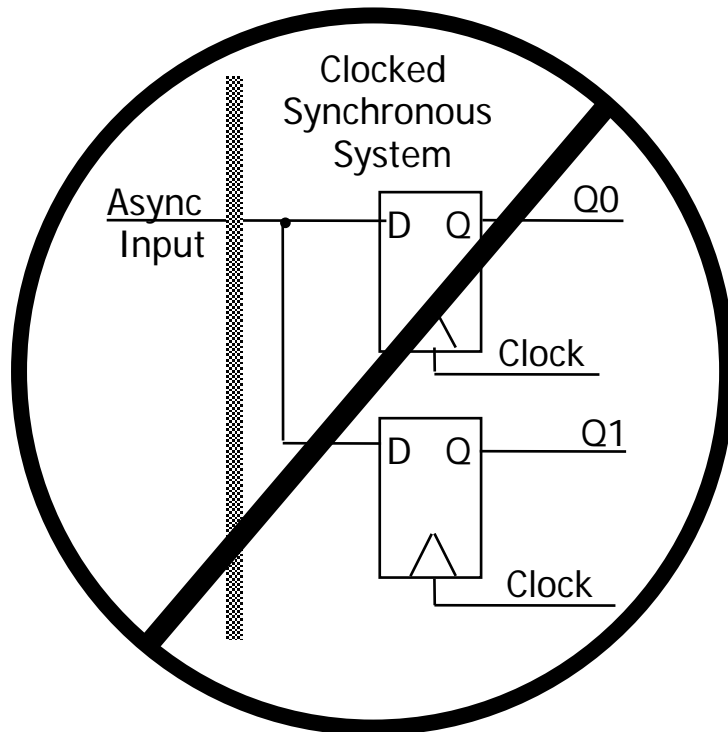
# Dealing with synchronization failure

- 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)



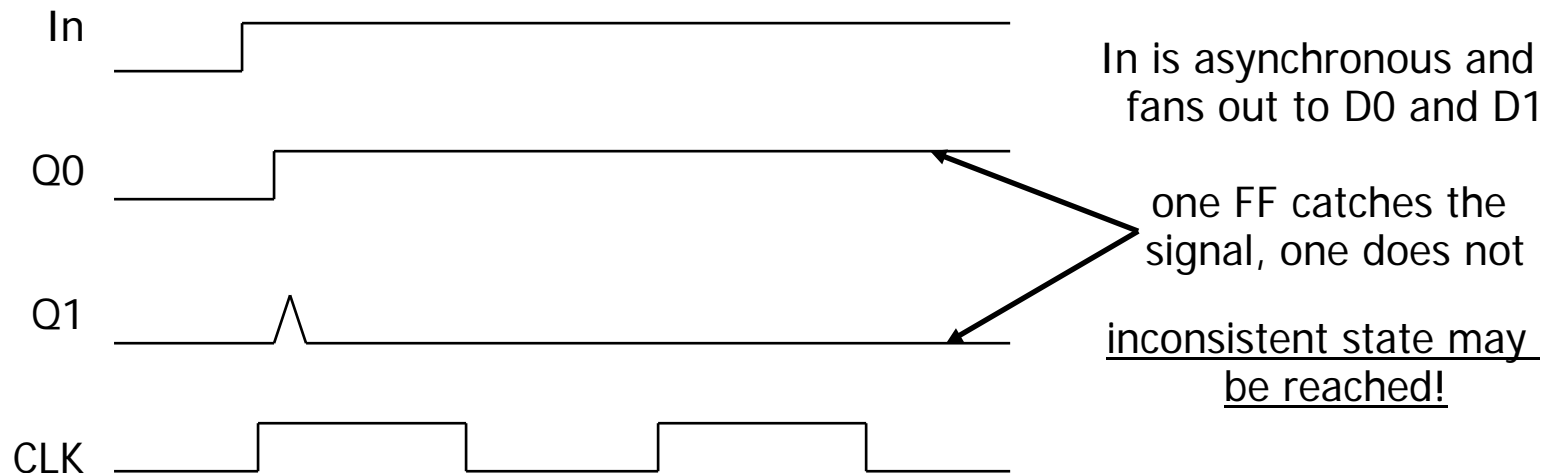
# Handling asynchronous inputs

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



# Handling asynchronous inputs (cont'd)

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

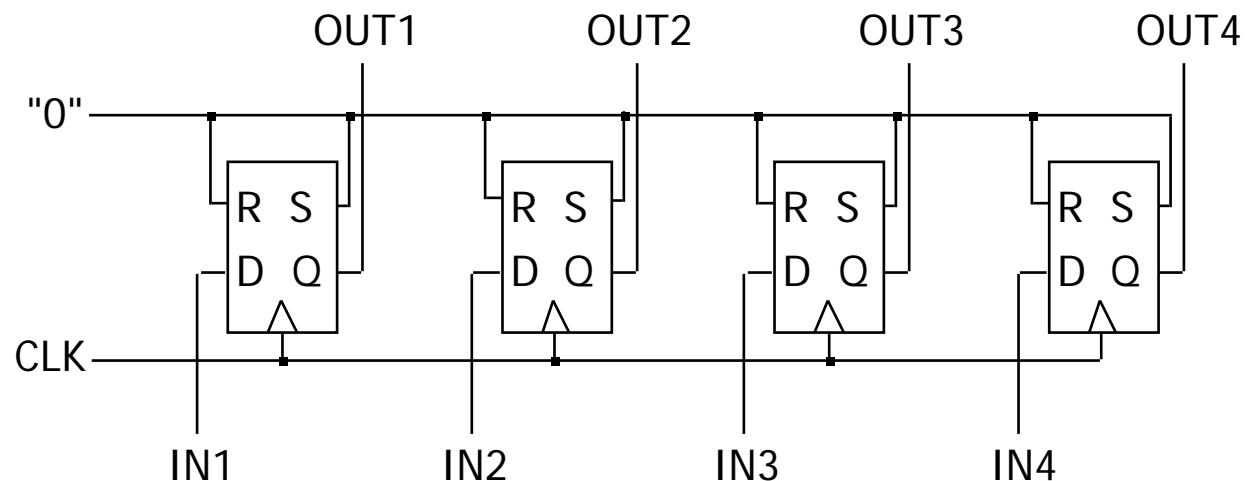


# Flip-flop features

- Reset (set state to 0) – R
  - synchronous:  $D_{\text{new}} = R' \cdot D_{\text{old}}$  (when next clock edge arrives)
  - asynchronous: doesn't wait for clock, quick but dangerous
- Preset or set (set state to 1) – S (or sometimes P)
  - synchronous:  $D_{\text{new}} = D_{\text{old}} + S$  (when next clock edge arrives)
  - asynchronous: doesn't wait for clock, quick but dangerous
- Both reset and preset
  - $D_{\text{new}} = R' \cdot D_{\text{old}} + S$  (set-dominant)
  - $D_{\text{new}} = R' \cdot D_{\text{old}} + R'S$  (reset-dominant)
- Selective input capability (input enable or load) – LD or EN
  - multiplexor at input:  $D_{\text{new}} = LD' \cdot Q + LD \cdot D_{\text{old}}$
  - load may or may not override reset/set (usually R/S have priority)
- Complementary outputs – Q and Q'

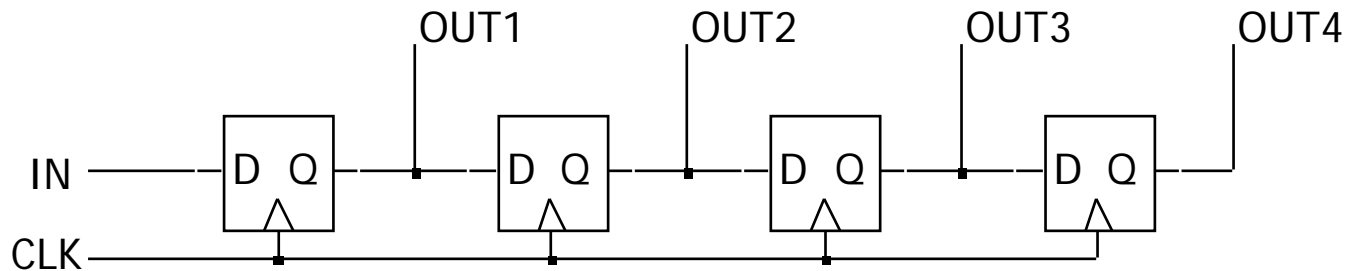
# Registers

- Collections of flip-flops with similar controls and logic
  - stored values somehow related (for example, form binary value)
  - share clock, reset, and set lines
  - similar logic at each stage
- Examples
  - shift registers
  - counters



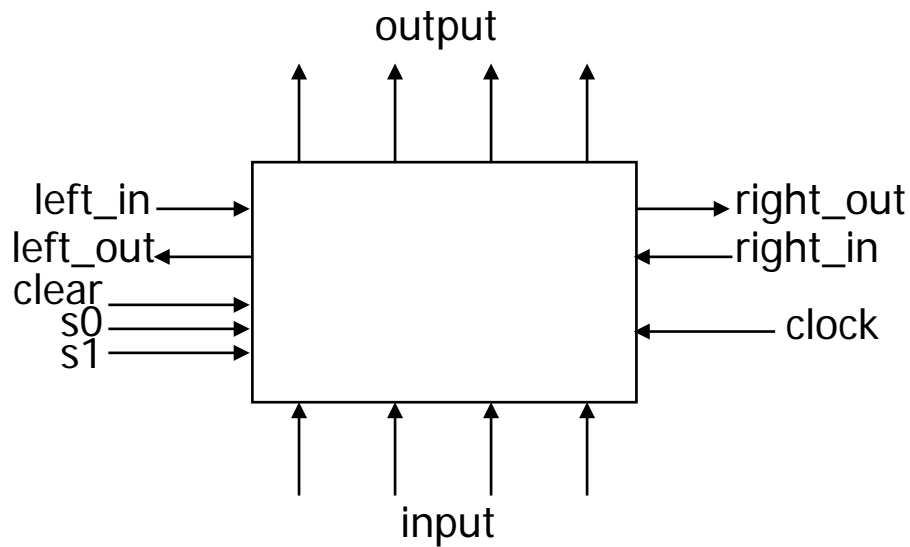
# Shift register

- Holds samples of input
  - store last 4 input values in sequence
  - 4-bit shift register:



# Universal shift register

- Holds 4 values
  - serial or parallel inputs
  - serial or parallel outputs
  - permits shift left or right
  - shift in new values from left or right



clear sets the register contents and output to 0

s1 and s0 determine the shift function

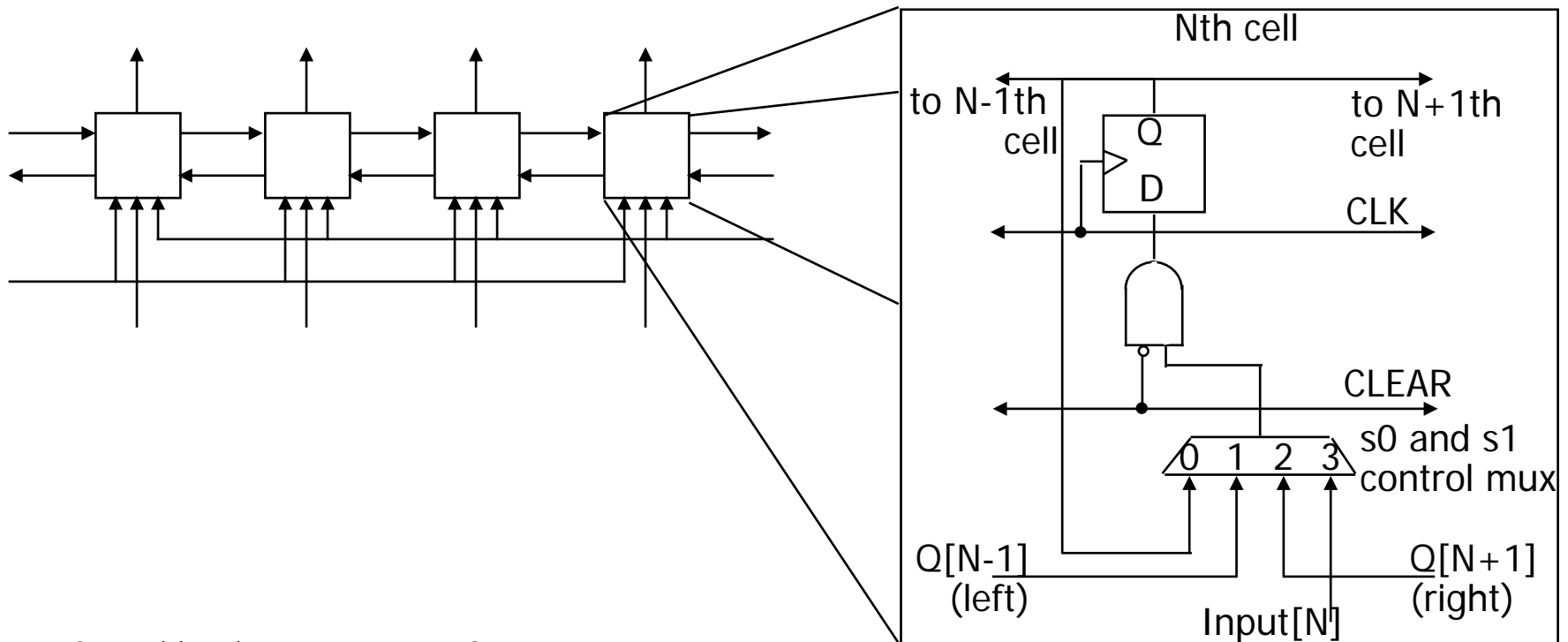
s0	s1	function
0	0	hold state
0	1	shift right
1	0	shift left
1	1	load new input

# Design of universal shift register

- Consider one of the four flip-flops

- new value at next clock cycle:

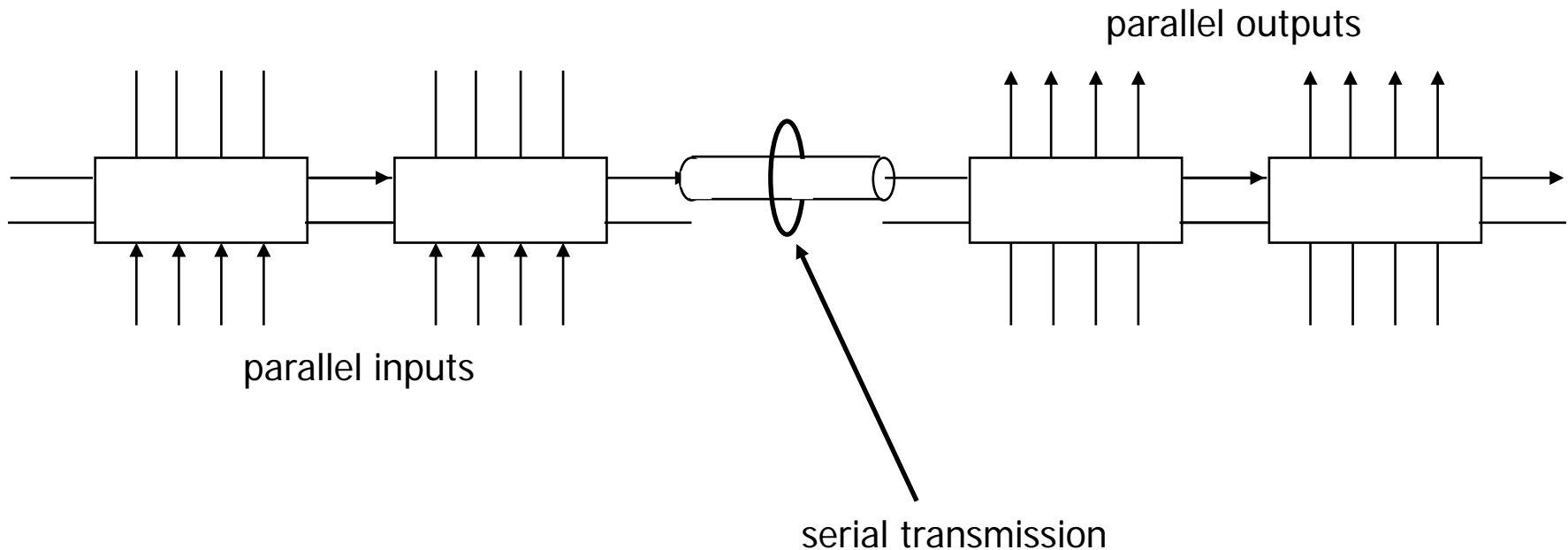
clear	s0	s1	new value
1	—	—	0
0	0	0	output
0	0	1	output value of FF to left (shift right)
0	1	0	output value of FF to right (shift left)
0	1	1	input





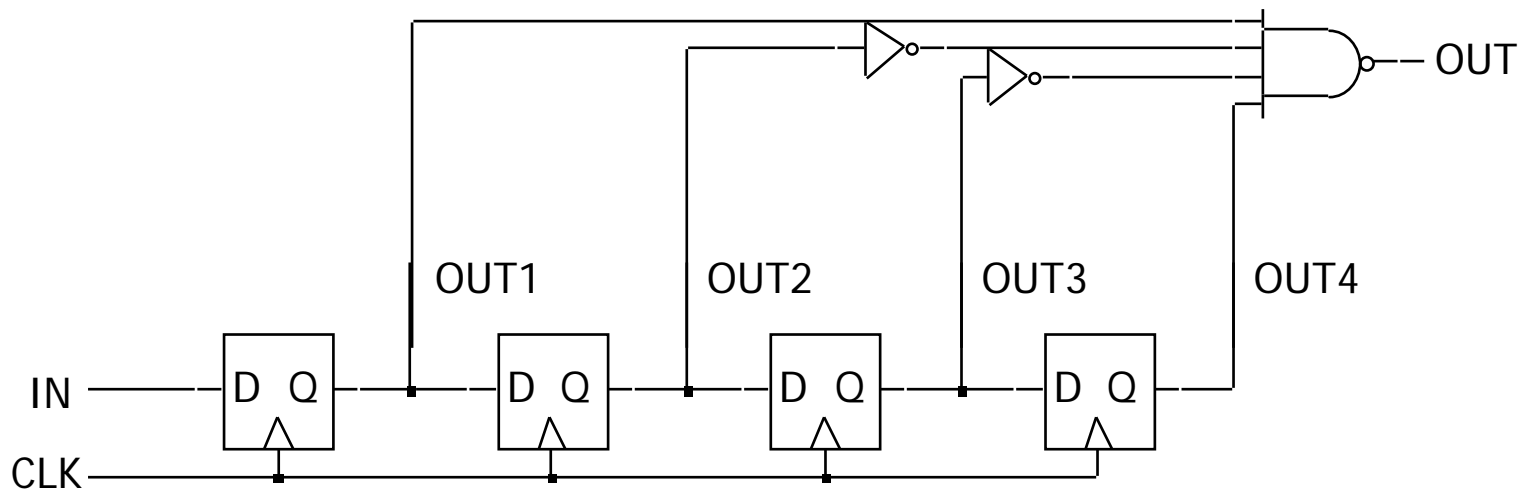
# Shift register application

- Parallel-to-serial conversion for serial transmission



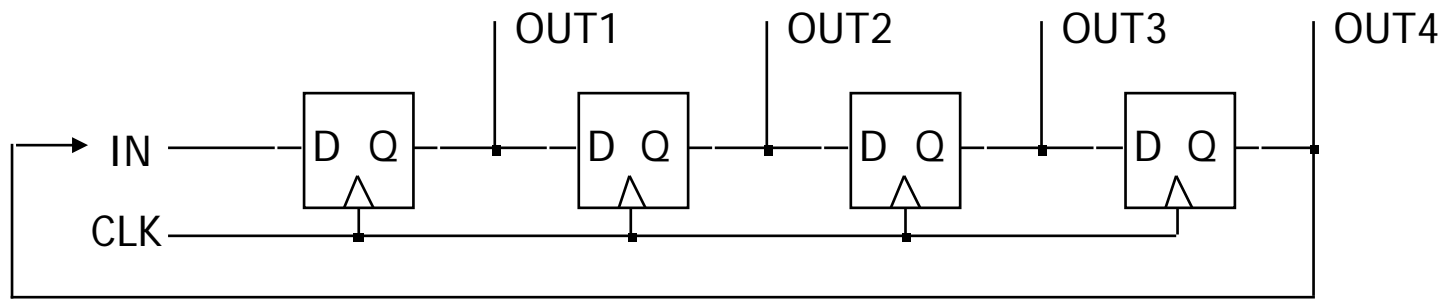
# Pattern recognizer

- Combinational function of input samples
  - in this case, recognizing the pattern 1001 on the single input signal



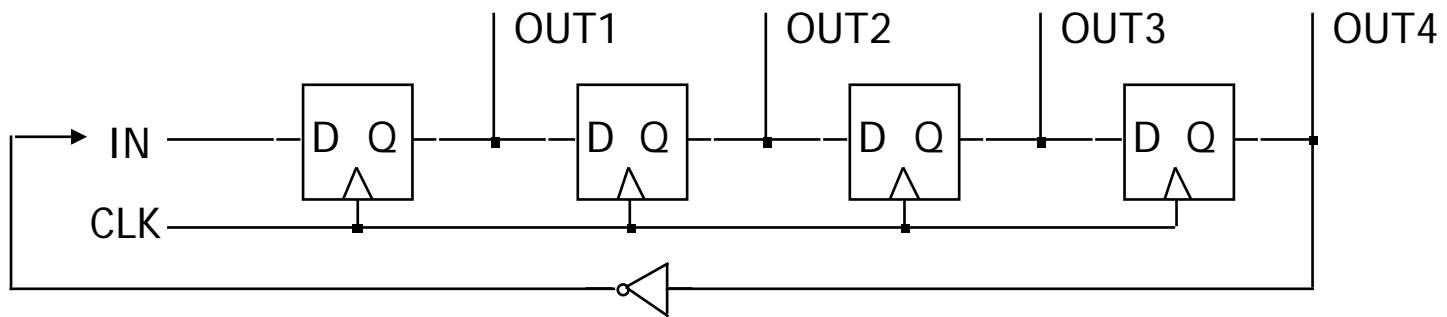
# Counters

- Sequences through a fixed set of patterns
  - in this case, 1000, 0100, 0010, 0001
  - if one of the patterns is its initial state (by loading or set/reset)



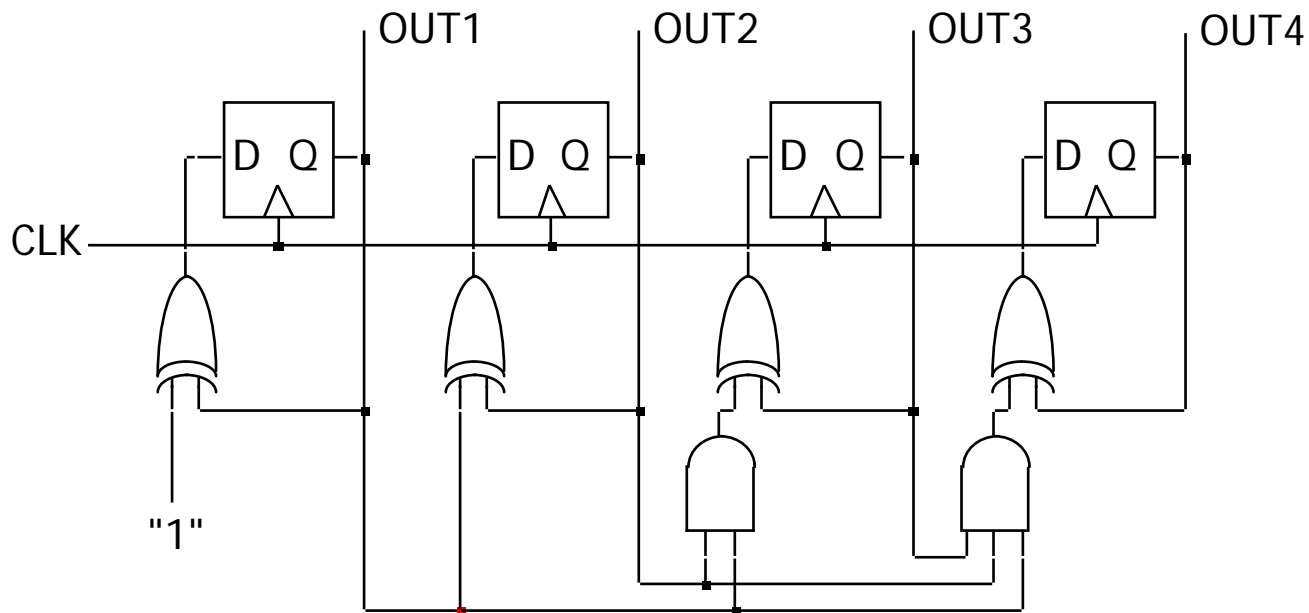
# Activity

- How does this counter work?



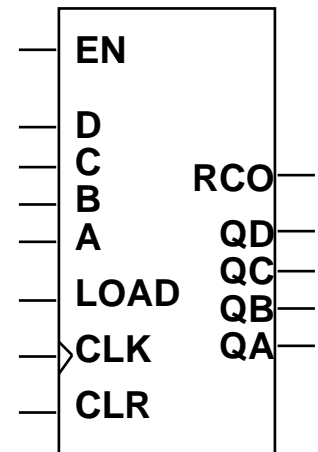
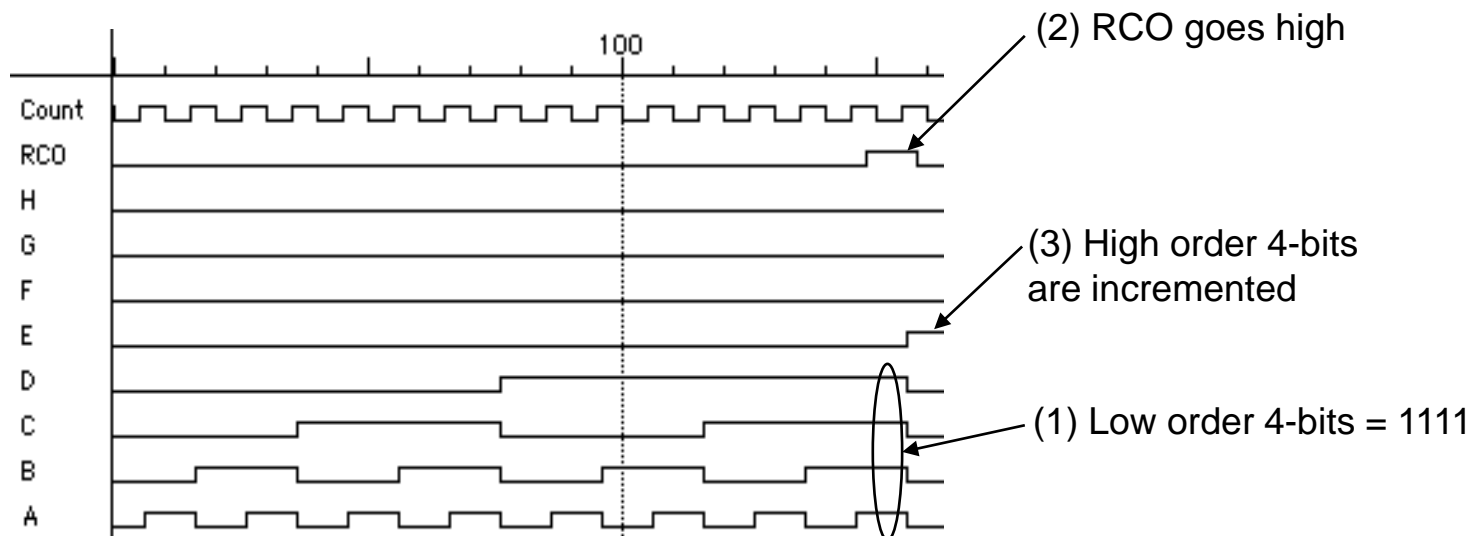
# Binary counter

- Logic between registers (not just multiplexer)
  - XOR decides when bit should be toggled
  - always for low-order bit,  
only when first bit is true for second bit,  
and so on



# Four-bit binary synchronous up-counter

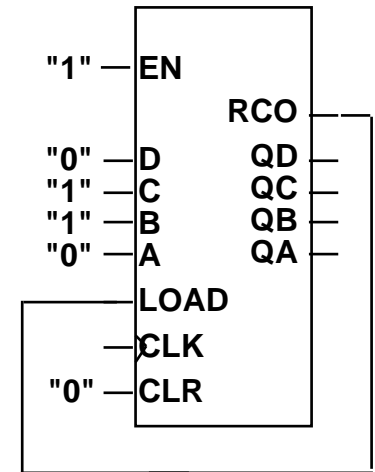
- Standard component with many applications
  - positive edge-triggered FFs w/ synchronous load and clear inputs
  - parallel load data from D, C, B, A
  - enable inputs: must be asserted to enable counting
  - RCO: ripple-carry out used for cascading counters
    - high when counter is in its highest state 1111
    - implemented using an AND gate



# Offset counters

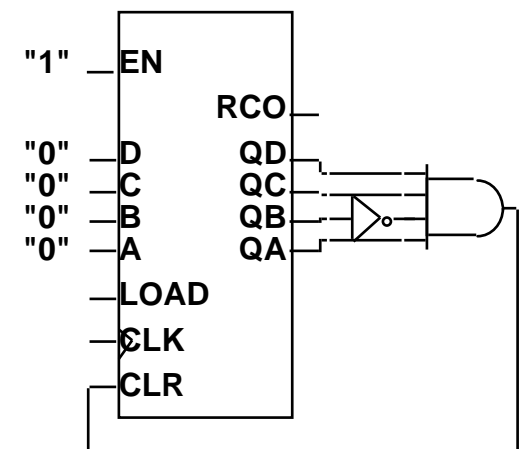
- Starting offset counters – use of synchronous load

- e.g., 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1111, 0110, ...



- Ending offset counter – comparator for ending value

- e.g., 0000, 0001, 0010, ..., 1100, 1101, 0000



- Combinations of the above (start and stop value)

# Hardware Description Languages and Sequential Logic

- Flip-flops
  - representation of clocks - timing of state changes
  - asynchronous vs. synchronous
- Shift registers
- Simple counters



# Flip-flop in Verilog

- Use always block's sensitivity list to wait for clock edge

```
module dff (clk, d, q);  
  
    input  clk, d;  
    output q;  
    reg    q;  
  
    always @(posedge clk)  
        q = d;  
  
endmodule
```

# More Flip-flops

- Synchronous/asynchronous reset/set
  - single thread that waits for the clock
  - three parallel threads – only one of which waits for the clock

## Synchronous

```
module dff (clk, s, r, d, q);
    input  clk, s, r, d;
    output q;
    reg    q;

    always @(posedge clk)
        if (r)      q = 1'b0;
        else if (s) q = 1'b1;
        else       q = d;

endmodule
```

## Asynchronous

```
module dff (clk, s, r, d, q);
    input  clk, s, r, d;
    output q;
    reg    q;

    always @(posedge r)
        q = 1'b0;
    always @(posedge s)
        q = 1'b1;
    always @(posedge clk)
        q = d;

endmodule
```

# Incorrect Flip-flop in Verilog

- Use always block's sensitivity list to wait for clock to change

```
module dff (clk, d, q);
```

```
    input  clk, d;
```

```
    output q;
```

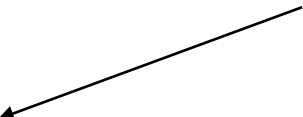
```
    reg    q;
```

```
    always @(clk)
```

```
        q = d;
```

```
endmodule
```

Not correct! Q will  
change whenever the  
clock changes, not  
just on an edge.



# Blocking and Non-Blocking Assignments

- Blocking assignments ( $X=A$ )
  - completes the assignment before continuing on to next statement
- Non-blocking assignments ( $X<=A$ )
  - completes in zero time and doesn't change the value of the target until a blocking point (delay/wait) is encountered
- Example: swap

```
always @(posedge CLK)
begin
    temp = B;
    B = A;
    A = temp;
end
```

```
always @(posedge CLK)
begin
    A <= B;
    B <= A;
end
```

# Register-transfer-level (RTL) Assignment

- Non-blocking assignment is also known as an RTL assignment
  - if used in an always block triggered by a clock edge
  - all flip-flops change together

```
// B,C,D all get the value of A
always @(posedge clk)
begin
    B = A;
    C = B;
    D = C;
end
```

```
// implements a shift register too
always @(posedge clk)
begin
    B <= A;
    C <= B;
    D <= C;
end
```

# Mobius Counter in Verilog

```
initial
begin
    A = 1'b0;
    B = 1'b0;
    C = 1'b0;
    D = 1'b0;
end

always @(posedge clk)
begin
    A <= ~D;
    B <= A;
    C <= B;
    D <= C;
end
```

# Binary Counter in Verilog

```
module binary_counter (clk, c8, c4, c2, c1);
```

```
    input  clk;
    output c8, c4, c2, c1;
```

```
    reg [3:0] count;
```

```
    initial begin
        count = 0;
    end
```

```
    always @(posedge clk) begin
        count = count + 4'b0001;
    end
```

```
    assign c8 = count[3];
    assign c4 = count[2];
    assign c2 = count[1];
    assign c1 = count[0];
```

```
endmodule
```

```
module binary_counter (clk, c8, c4, c2, c1, rco);
```

```
    input  clk;
    output c8, c4, c2, c1, rco;
```

```
    reg [3:0] count;
    reg rco;
```

```
    initial begin . . . end
```

```
    always @(posedge clk) begin . . . end
```

```
    assign c8 = count[3];
    assign c4 = count[2];
    assign c2 = count[1];
    assign c1 = count[0];
    assign rco = (count == 4b'1111);
```

```
endmodule
```

# Sequential logic summary

- Fundamental building block of circuits with state
  - latch and flip-flop
  - R-S latch, R-S master/slave, D master/slave, edge-triggered D flip-flop
- Timing methodologies
  - use of clocks
  - cascaded FFs work because propagation delays exceed hold times
  - beware of clock skew
- Asynchronous inputs and their dangers
  - synchronizer failure: what it is and how to minimize its impact
- Basic registers
  - shift registers
  - counters
- Hardware description languages and sequential logic