Chapter 6.
Sequential logic design

This is the beginning of the second part of this course, sequential logic.

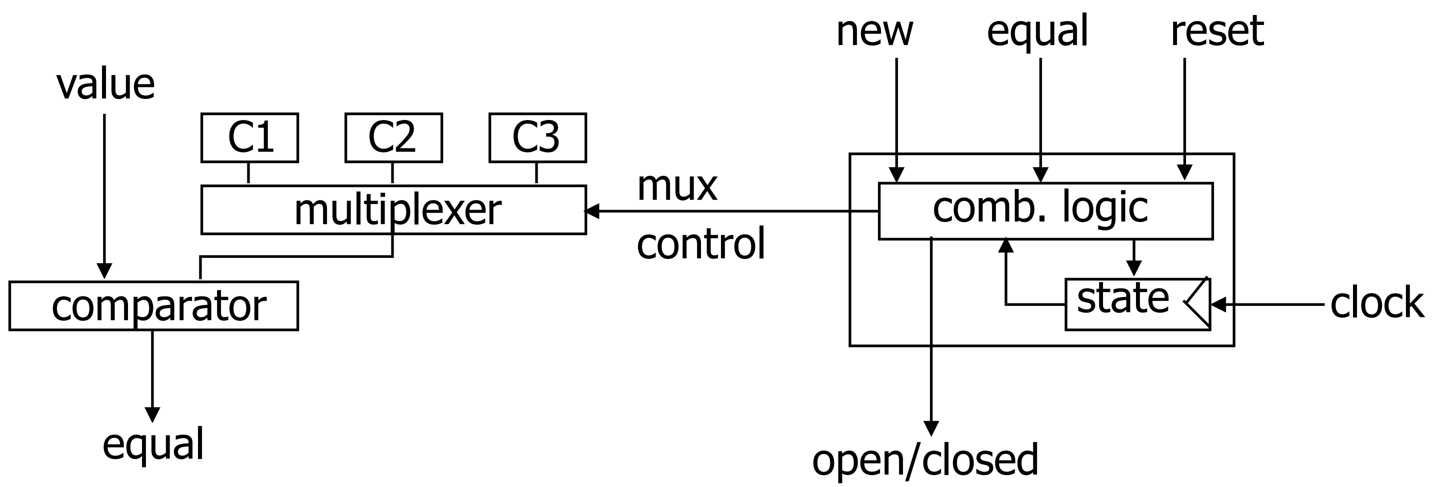
Sequential logic

- **Sequential circuits**
  - simple circuits with feedback
  - latches
  - edge-triggered flip-flops
- **Basic registers**
  - shift registers
  - simple counters

Again, sequential logic circuits are quite different from combinational logic. In general, sequential systems are more difficult to design. We will discuss some basic issues in the sequential logic systems in this chapter.

Sequential logic circuits (SLCs)

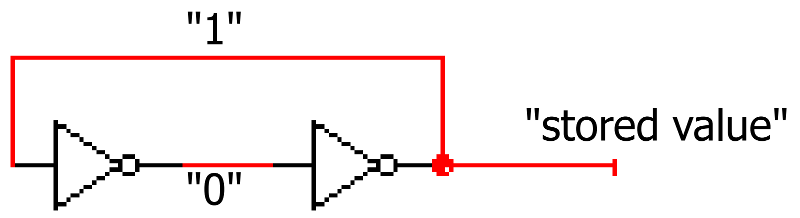
- **Circuits with feedback**
  - outputs = f(inputs, past inputs, past outputs)
  - basis for building "memory" into logic circuits
  - door combination lock is an example of a sequential circuit
    - state is memory
    - state is an "output" and an "input" to combinational logic
    - combination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



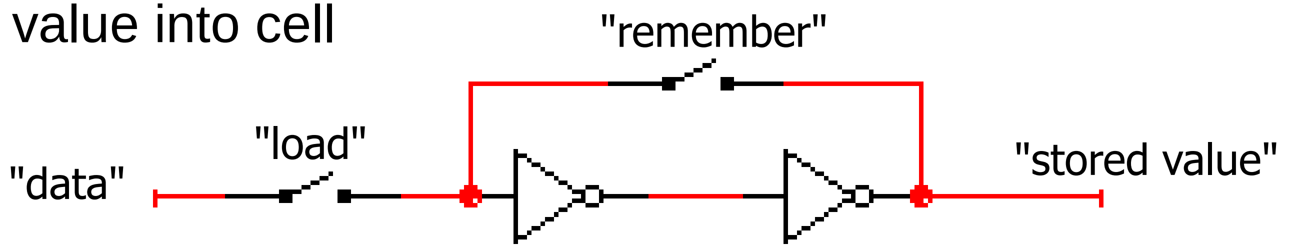
In most sequential logic circuits (SLCs), there are storage parts or memory elements. Recall that in the door lock system, three numbers should be stored and compared. What is the key element here?

Simplest circuits with feedback

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



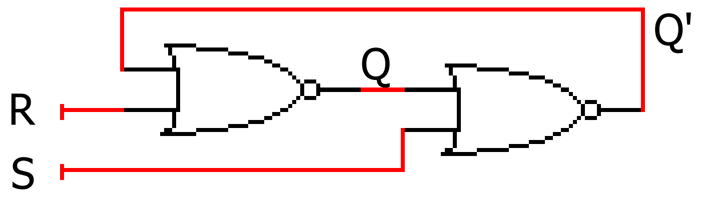
- **How to get a new value into the memory cell?**
  - selectively break feedback path
  - load new value into cell

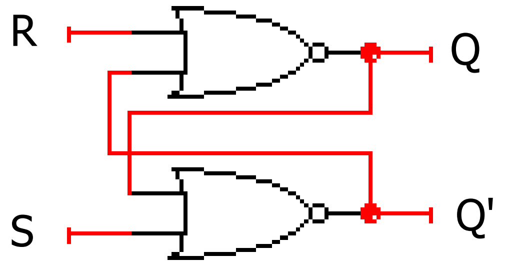


Let’s consider a simple memory element first. What if we place two inverters before a stored value and suppose the stored value is 1. Note that there is a feedback from the stored value to the first inverter. Another more sophisticated option is to have a selection function between a stored value and a new value. If “load” is enabled, the new “data” will be put into the stored value.

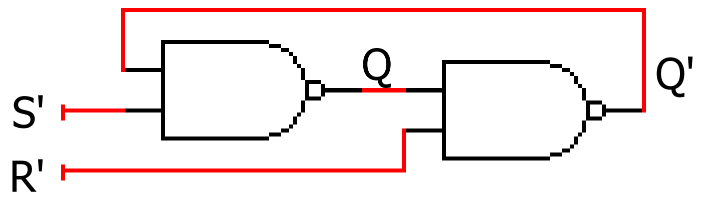
Memory with cross-coupled gates

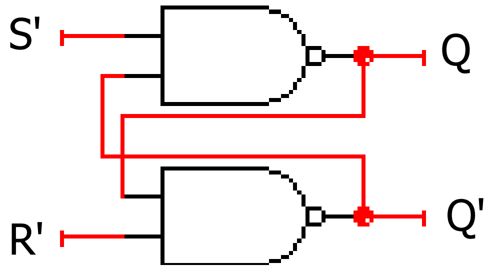
- **Cross-coupled NOR gates (R-S Latch)**
  - similar to inverter pair, with capability to force output to 0 (reset=1) or 1 (set=1)





- **Cross-coupled NAND gates**
  - similar to inverter pair, with capability to force output to 0 (R'=0) or 1 (S'=0)

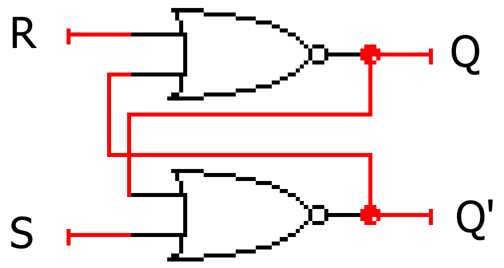


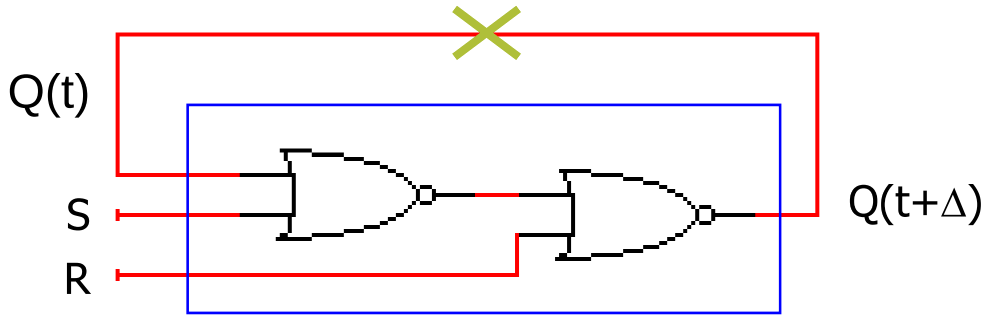


Let's consider this feedback in the cases of cross-connectivity between other gate types. With Nor or Nand gates, we can have more control inputs. R and S denote reset and set inputs, respectively. Q is the output value of interest and its inverse is denoted by Q'. For a NOR gate, think of a case when a single input can determine the output of the gate. Suppose R is 1, which makes Q 0. Basically, we assume R and S are opposite for update.

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	

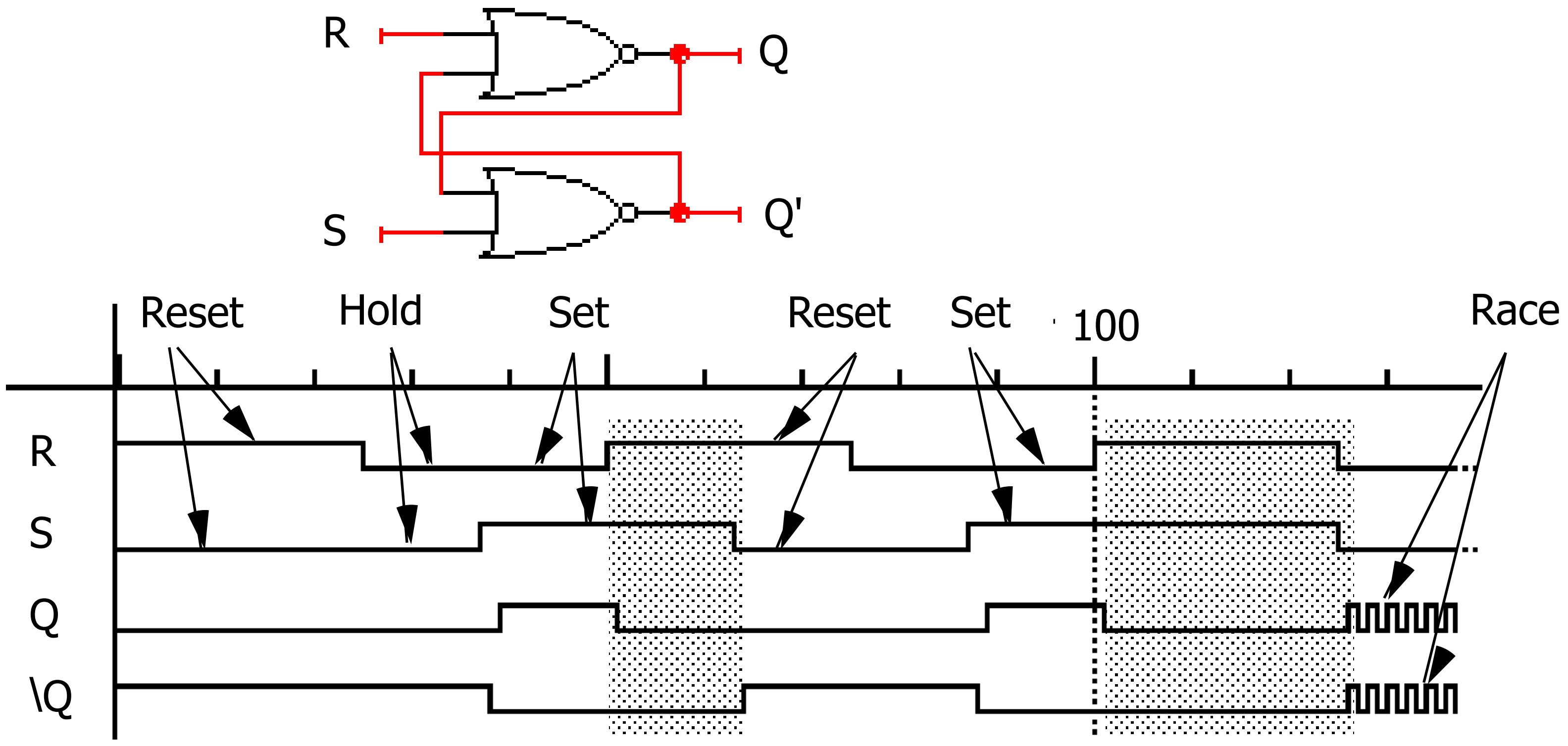
$$\begin{array}{c|c|c|c|c|} \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{2}{c}{S} \\ \cline{2-5} & 0 & 0 & X & 1 \\ \cline{2-5} Q(t) & 1 & 0 & X & 1 \\ \cline{2-5} \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{2}{c}{R} & \multicolumn{1}{c}{} \end{array}$$

characteristic equation

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

To analyze the timing, let's assume the feedback loop is cut off. That is, consider Q(t) and Q(t+ Δ) separately. Here Δ is the delay incurred by the latch. When you look at the truth table, you can notice something different. What is it?

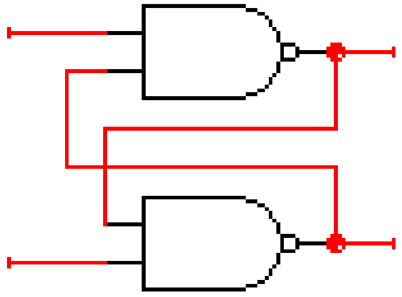
Timing behavior



If R=1 and S=0, Q is 0 and Q' is 1 (recall the meaning of reset R). If R=0 and S=1, Q is 1. What if R=S=0, then Q and Q' will remain with the previous values.

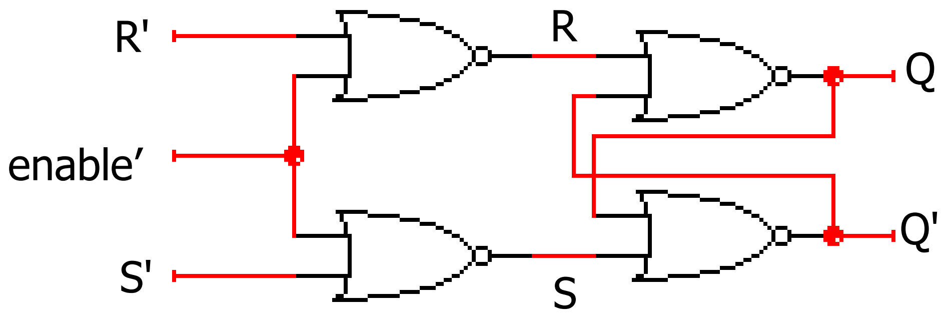
The problem happens when R=S=1. Are Q and Q' just 0s? The more significant problem arises when R=S=1 and then R=S=0, which will yield an oscillation as follows. Q and Q' will be 1 together. But then Q and Q' should be changed to 0 again. Then this cycle continues.

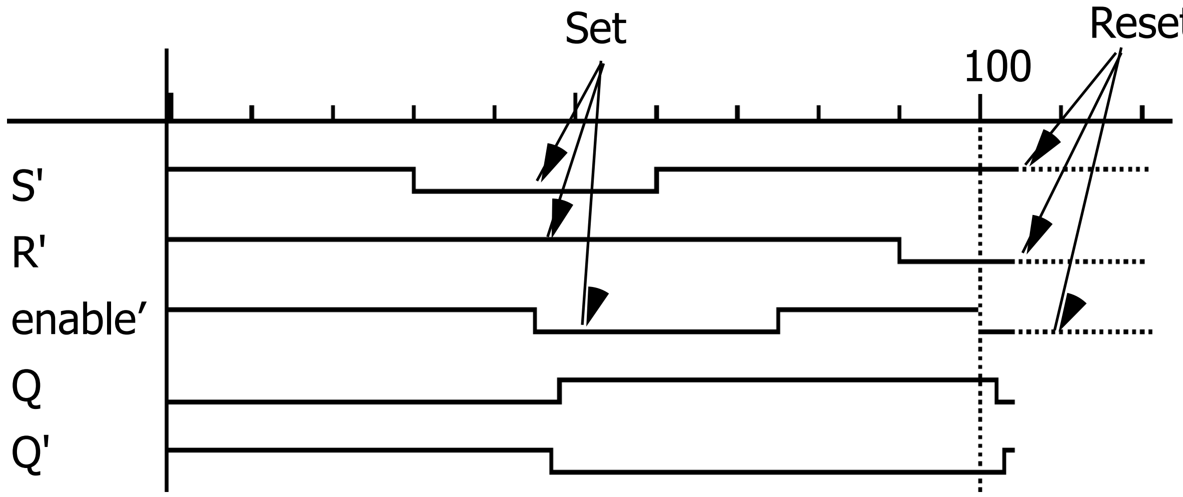
Activity: R-S latch using NAND gates



Gated R-S latch

- **Enable controls when R and S inputs matter**
  - otherwise, the slightest glitch on R or S could cause change in value stored

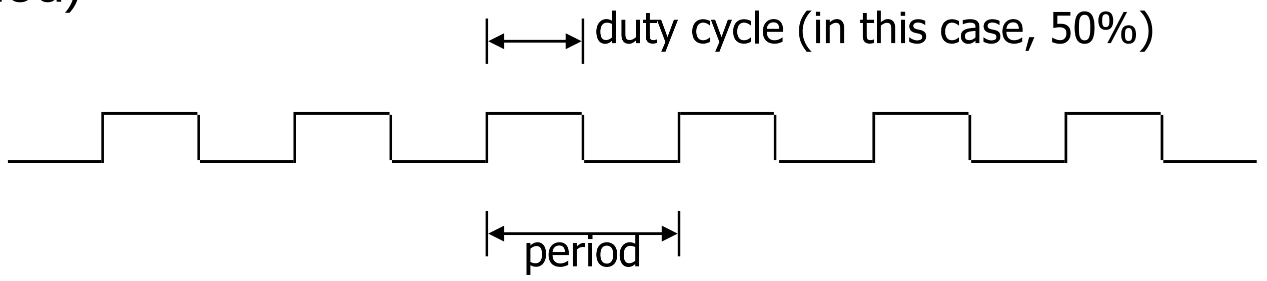




The next version is a gated R-S latch. In the gated (level-sensitive) R-S latch, the R-S values are handled cautiously. When enable' is high, then R and S are always zeroes (there is no glitch or fluctuation in R and S). So R' and S' are meaningful only when enable' is low. This waveform shows when enable is high and S is set (or S' is low), Q will become true.

Clocks

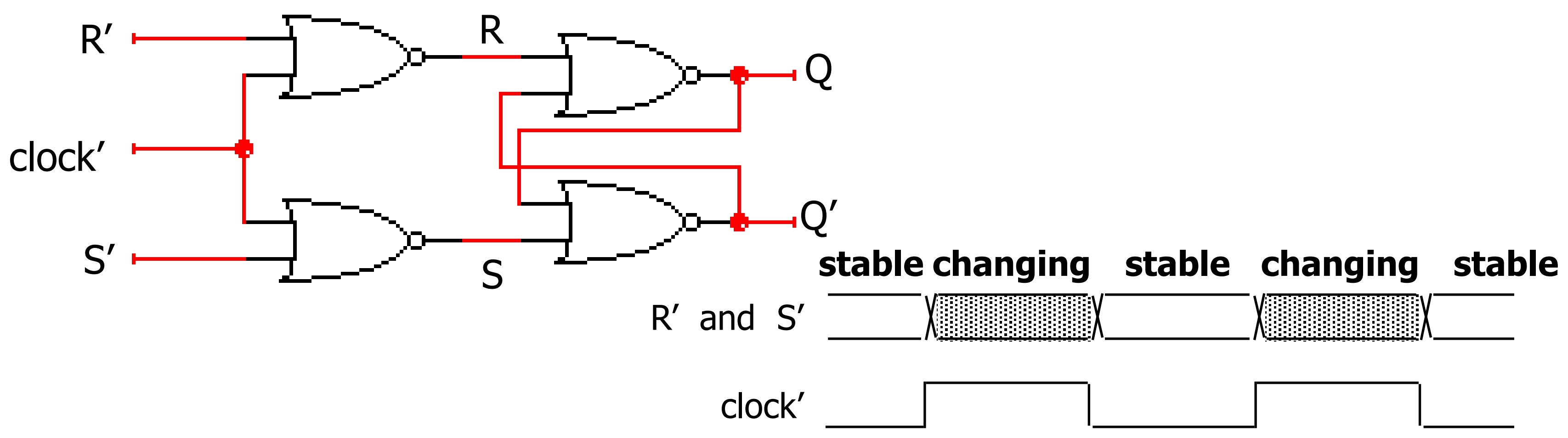
- **Used to keep time**
  - wait long enough for inputs (R' and S') to settle
  - then allow to have effect on value stored
- **Clocks are regular periodic signals**
  - period (time between ticks)
  - duty-cycle (time clock is high between ticks - expressed as % of period)



A clock is an important element in sequential circuits. The enable signal in the previous slide serves as kind of a clock. Once enable is asserted, it should remain high until the input stimulates the output fully. Normally, a clock is periodically alternating between high and low. The beginning of each period is called a clock tick. And the duty cycle is defined as the ratio of High voltage interval to the period.

Clocks (cont'd)

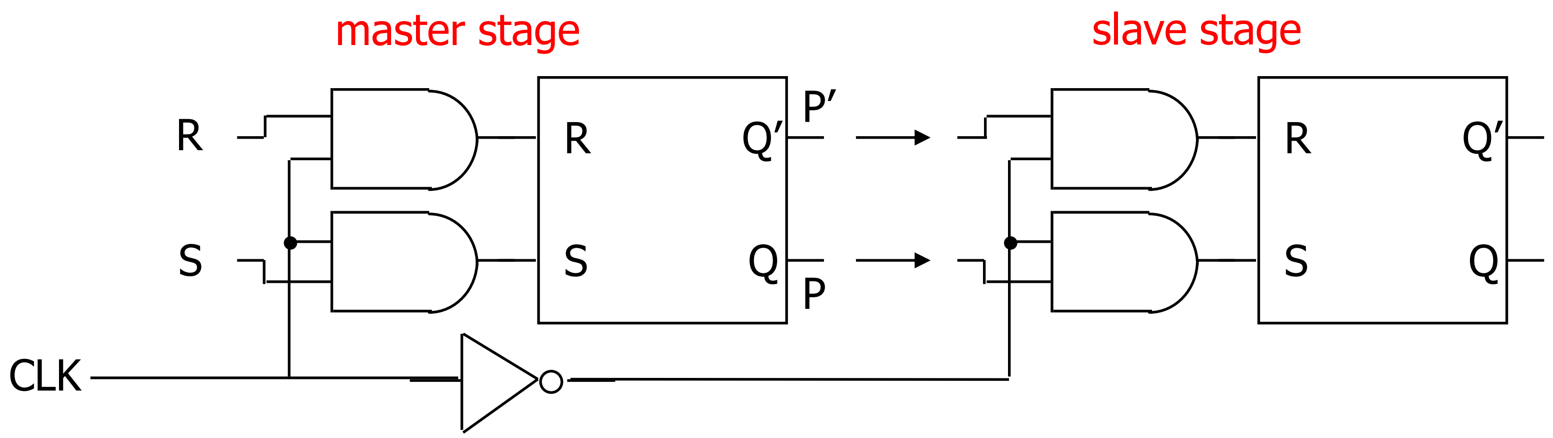
- **Controlling an R-S latch with a clock**
  - Change R and S while clock' is 1 (inject new input)
    - only have half of clock period for signal changes to propagate
  - Keep R and S stable while clock' is 0 (allowing R and S to pass)
    - signals must be stable for the other half of clock period



Let's control the gated (or level-sensitive) R-S latch with a clock. While clock' is 0, R' and S' should sustain their values which will update Q and Q' during that interval. While clock' is high, R' and S' can be changed to a new value (for next operation); in the meantime, Q and Q' will not change since R and S are 00.

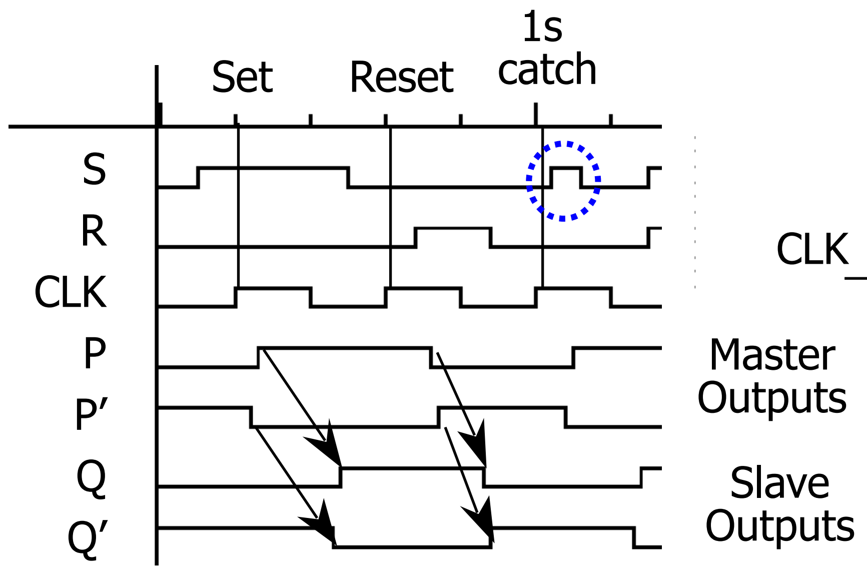
Master-slave structure

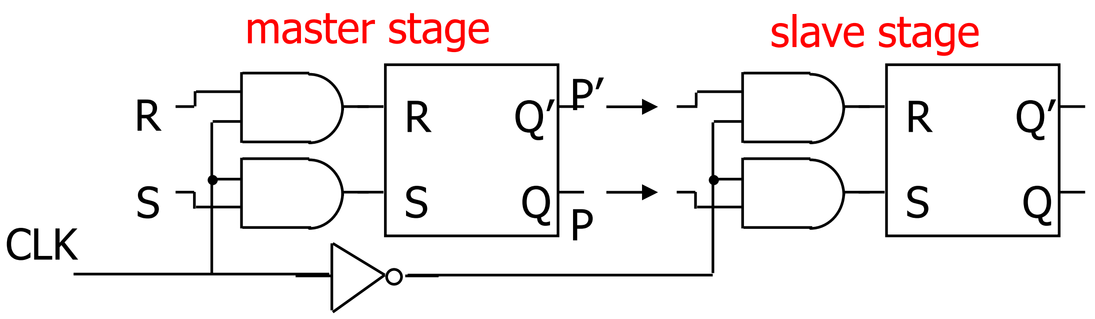
- **Break flow by alternating clocks**
  - use positive clock to latch inputs into one R-S latch
  - use negative clock to change outputs with another R-S latch
- **View pair as one basic unit**
  - master-slave flip-flop
  - twice as much logic
  - output changes a few gate delays after the falling edge of clock



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

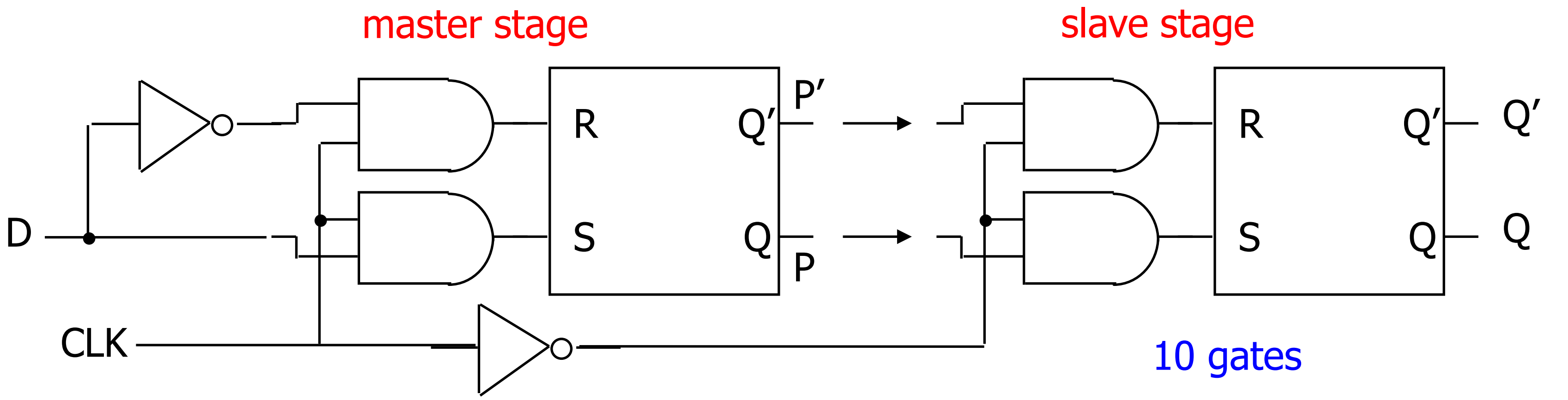




What would be the problem of the inverted clock signals for the pair of R-S latches? While the clock is high, suppose there is a glitch is in the very first S (0-1-0), P and P' will change, which in turn will affect the slave latch when the clock becomes low. This is called the 1s catching problem.

D flip-flop

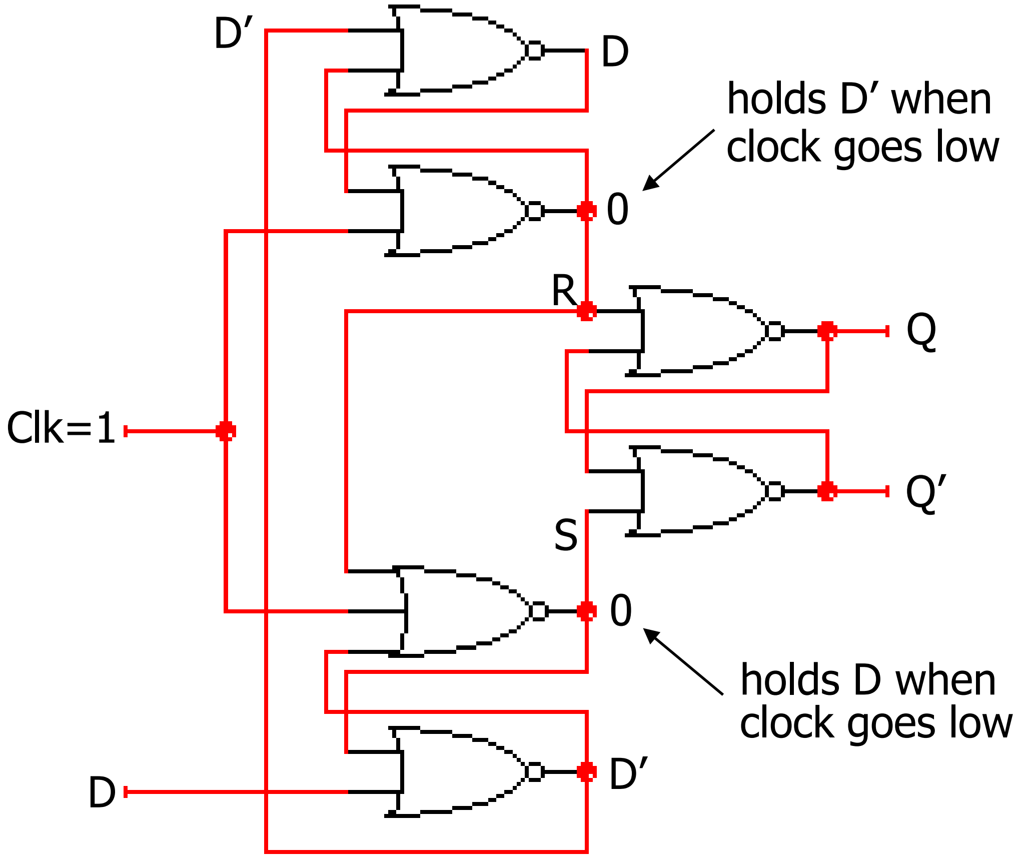
- **Make S and R complements of each other**
  - eliminates 1s catching problem
  - can't just hold previous value
    (must have new value ready every clock period)
  - value of D just before clock goes low is what is stored in flip-flop
    - negative edge-triggered



To eliminate the 1s catching problem, we have to make S and R have opposite values; so we use the complementary values from the same input, which is called D flip-flop. D is an abbreviation for data. So R and S can be either 01 and 10 only. Now we cannot use 00 to maintain the same value in the latch. How many gates here? Each R-S latch has two NOR gates.

(Negative) Edge-triggered D flip-flops (FFs)

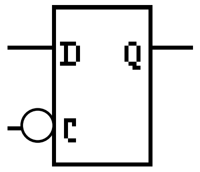
- **More efficient solution: only 6 gates**
  - sensitive to inputs only near edge of clock signal (not while high)



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

4-5 gate delays

must respect setup and hold time constraints to successfully capture input



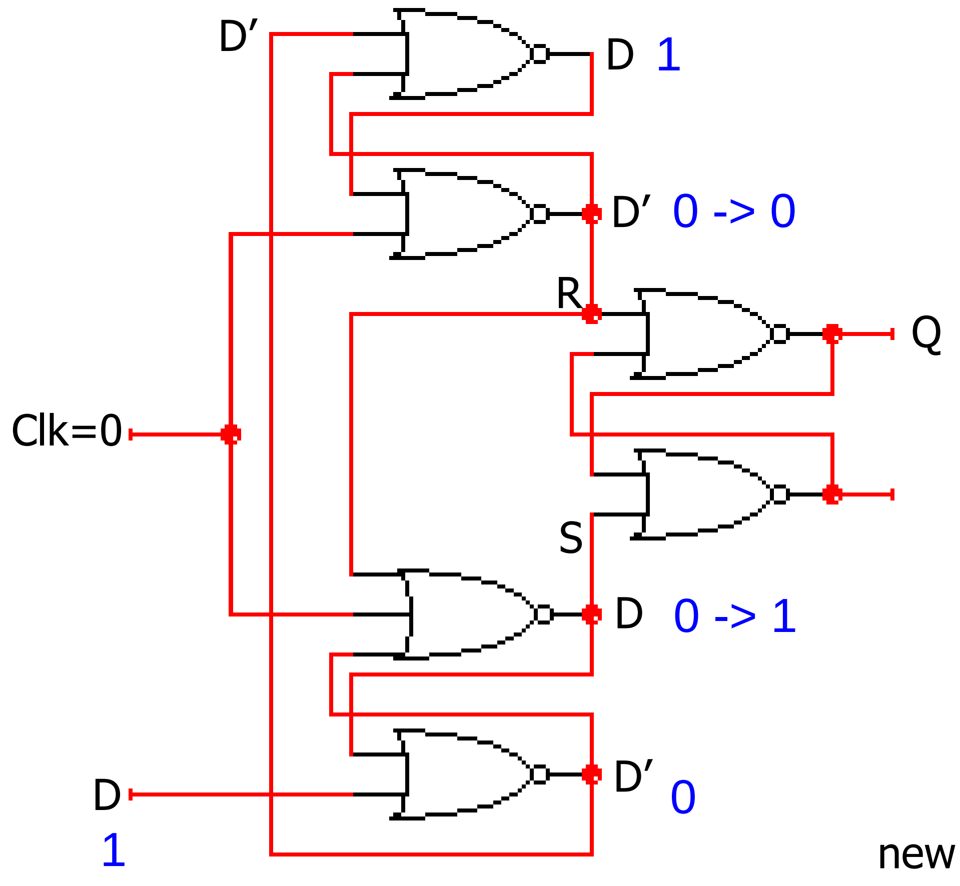
characteristic equation

Q(t+1) = D

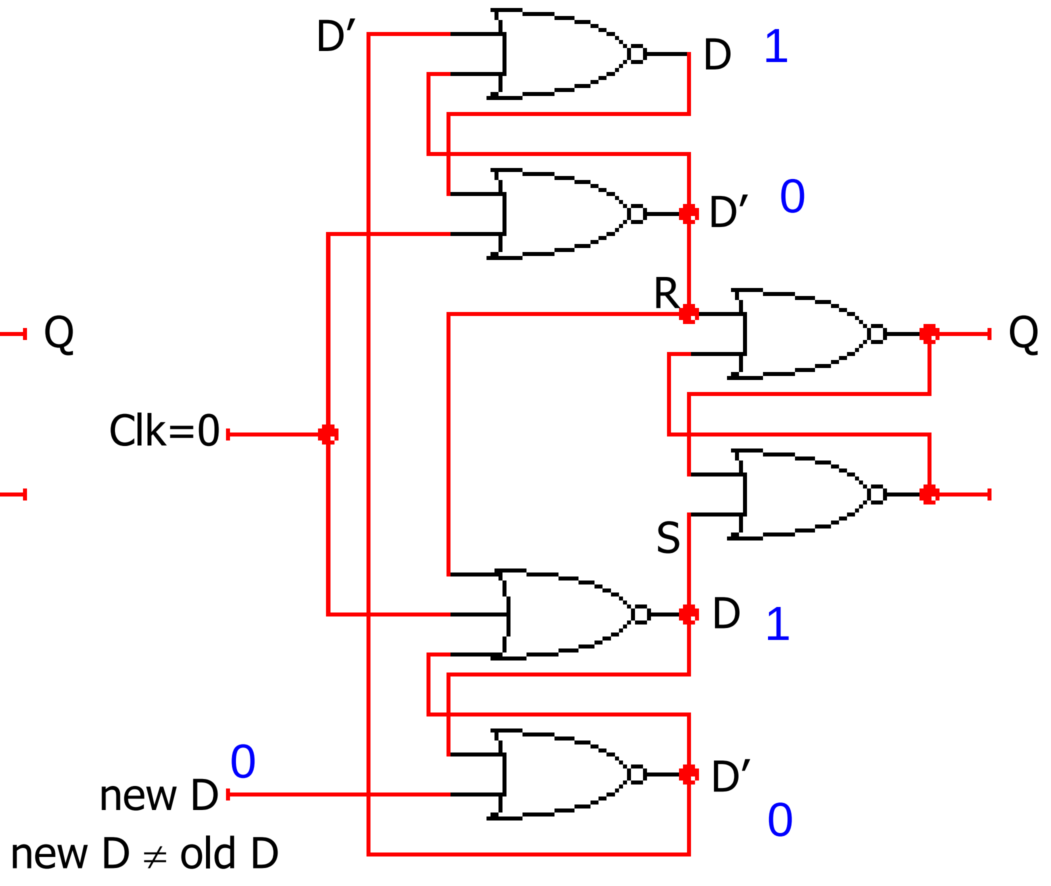
The other realization to solve the 1s catching problem is to use a clock edge to trigger the change of the flip-flop’s value. While the clock is high, the second top NOR and the second bottom NOR gates will be 0, which keeps the old values of Q and Q’

Negative Edge-triggered D flip-flops (cont'd)

- **Step-by-step analysis**



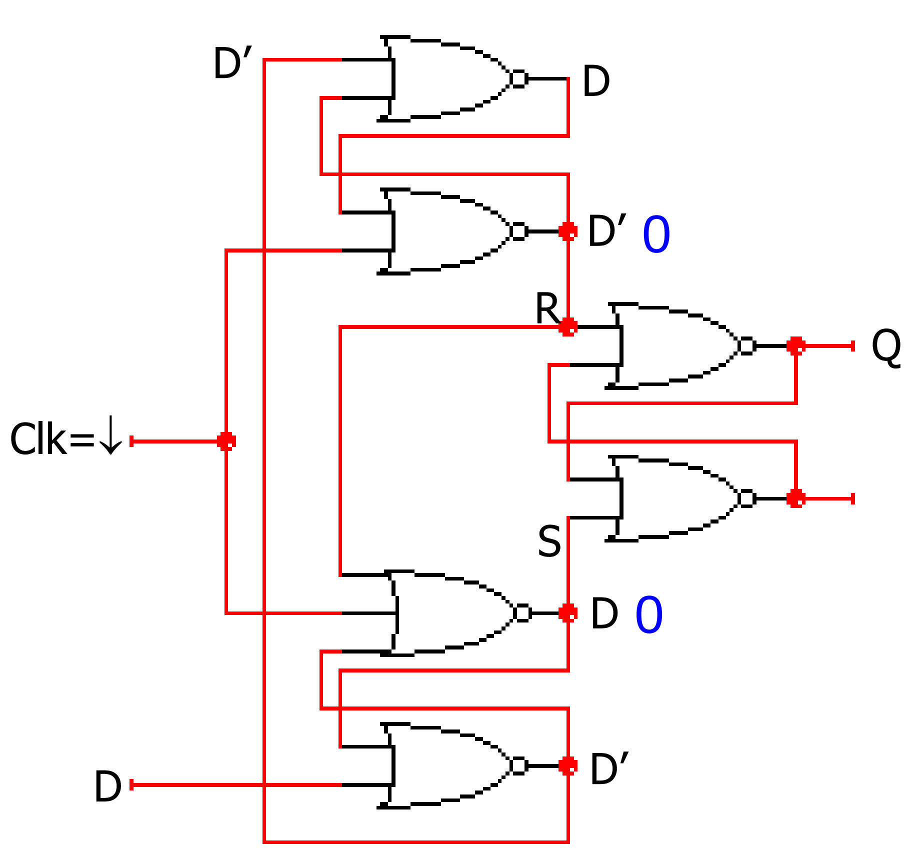
when clock goes high-to-low
data is latched

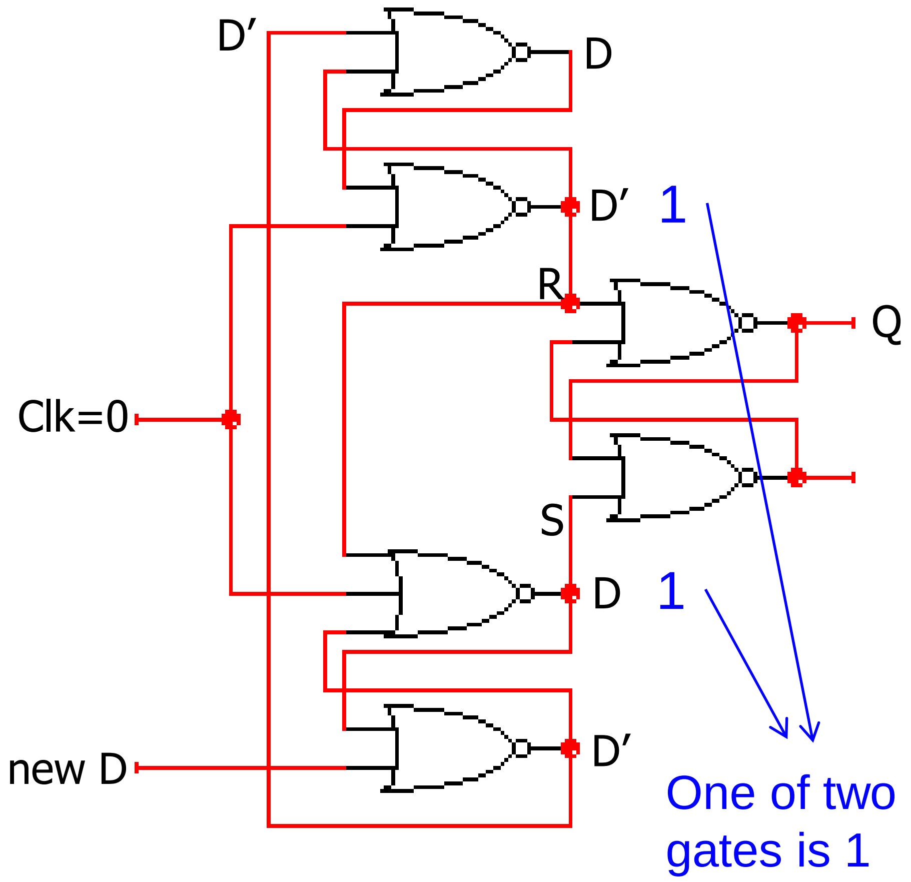


when clock is low
data is held

If clock goes from 1 to 0, initially S and R are 0. Then, depending on D's value, S or R will be changed, which in turn will set or reset Q. The numbers in blue shows the case when D is 1. After that, D's change makes no effect. E.g. new D is 0 now, D (second bottom NOR) was 1, D'(bottom) is 0. So D (top) is still 1; R and S are not changed.

How edge-triggered?

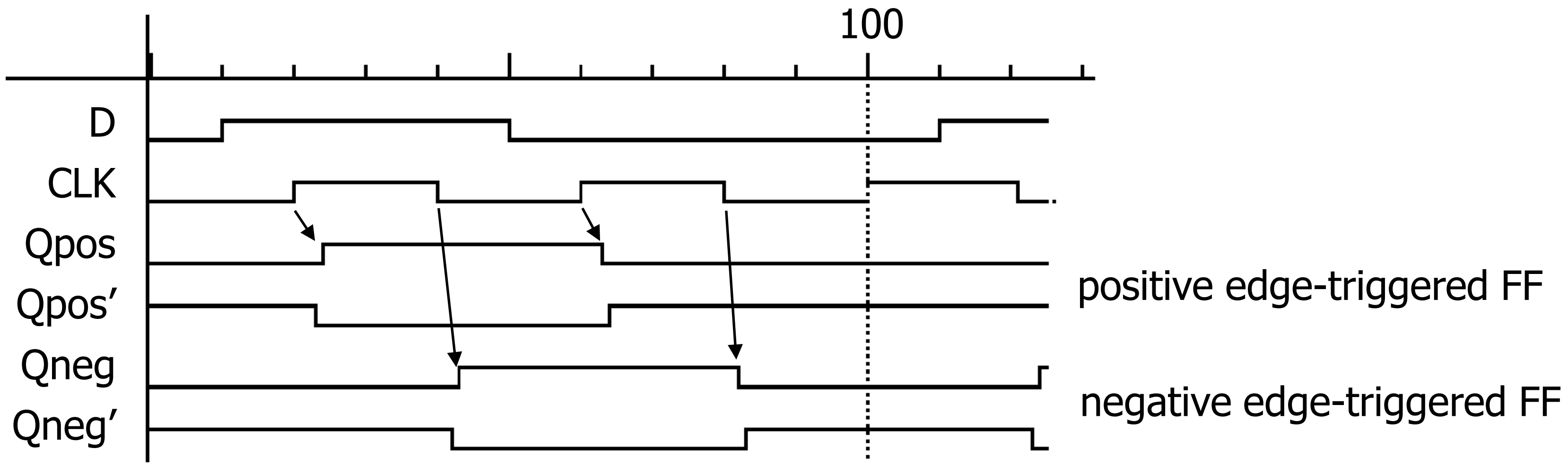




Right after the clock goes from 1 to 0, the second top and second bottom NOR gates are open to the D input. As soon as the input is latched to the Q, either of these NOR gates will be 1, which blocks the new input from entering.

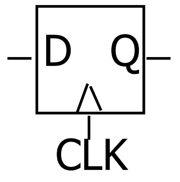
Edge-triggered D flip-flops (cont'd)

- **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

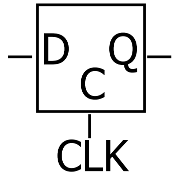


There are two kinds of flip-flops which are triggered by the two edges of the signals: rising edge and falling edge. The previous slide shows a negative edge-triggered FF. If we add an inverter to the clock, that FF is turned into a positive edge-triggered FF. Typically, latches are level triggered and simpler. FFs are mostly edge triggered and more complicated.

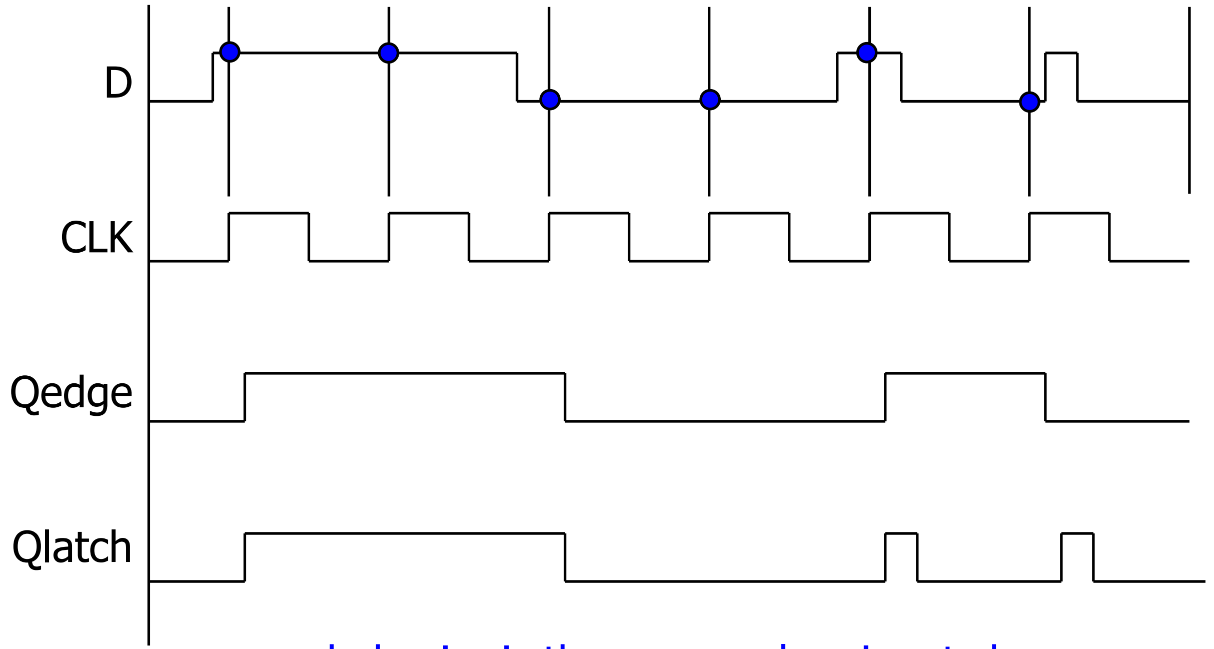
Comparison of latches and flip-flops (FFs)



positive
edge-triggered
flip-flop



transparent
(level-sensitive)
latch

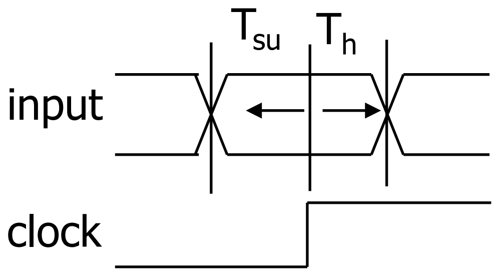


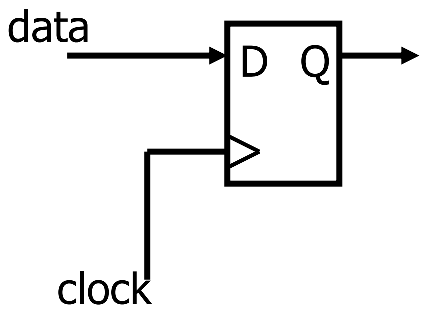
behavior is the same unless input changes
while the clock is high

Again, in FFs, the data value only at the rising edge (or falling edge) is critical (see blue dots). Meanwhile, most latches are sensitive to D value changes as long as the clock is high. Typically, the clock input of a FF is depicted by a triangle.

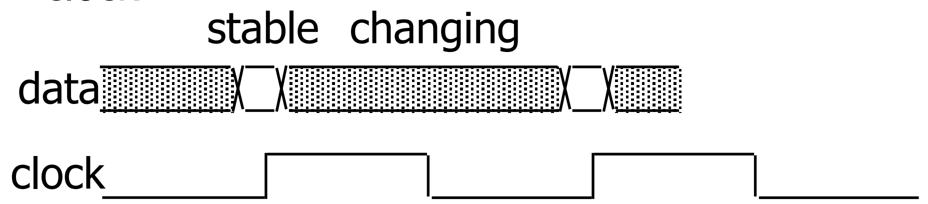
Timing constraints

- **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 (Tsu)
  - hold time: minimum time after the clocking event until which the input must remain stable (Th)





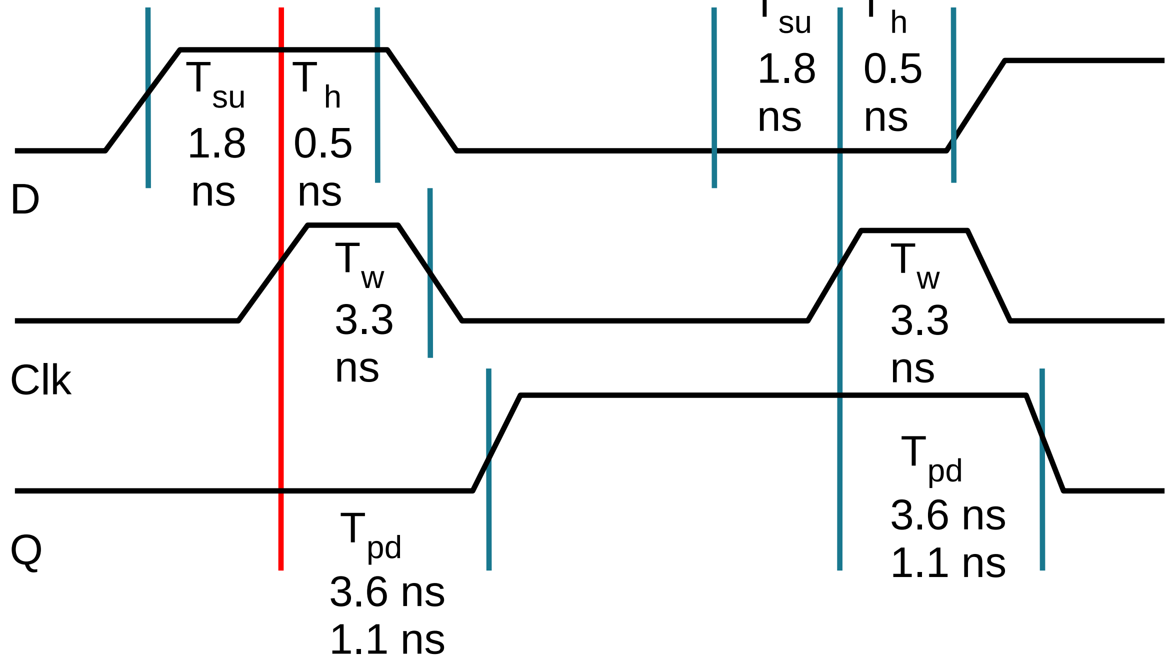
there is a timing "window" around the clocking event during which the input must remain stable and unchanged in order to be recognized



Let's go over terminologies first. We already know what is a clock. For each edge of a clock signal, there are some timing constraints. Suppose the positive edge of a clock signal triggers a circuit. Then the data input should be stable for an interval which is Tsu+Th, which are dependent on transistor circuit delay.

Typical timing specifications

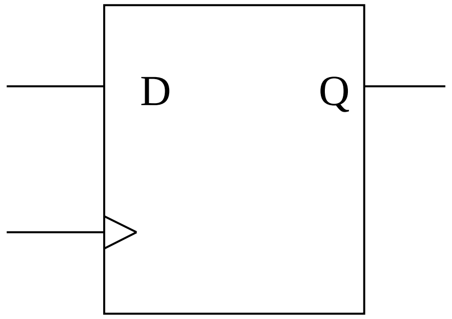
- **Positive edge-triggered D flip-flop**
  - setup and hold times
  - minimum clock width (Tw)
  - propagation delays (low to high, high to low, max and typical)



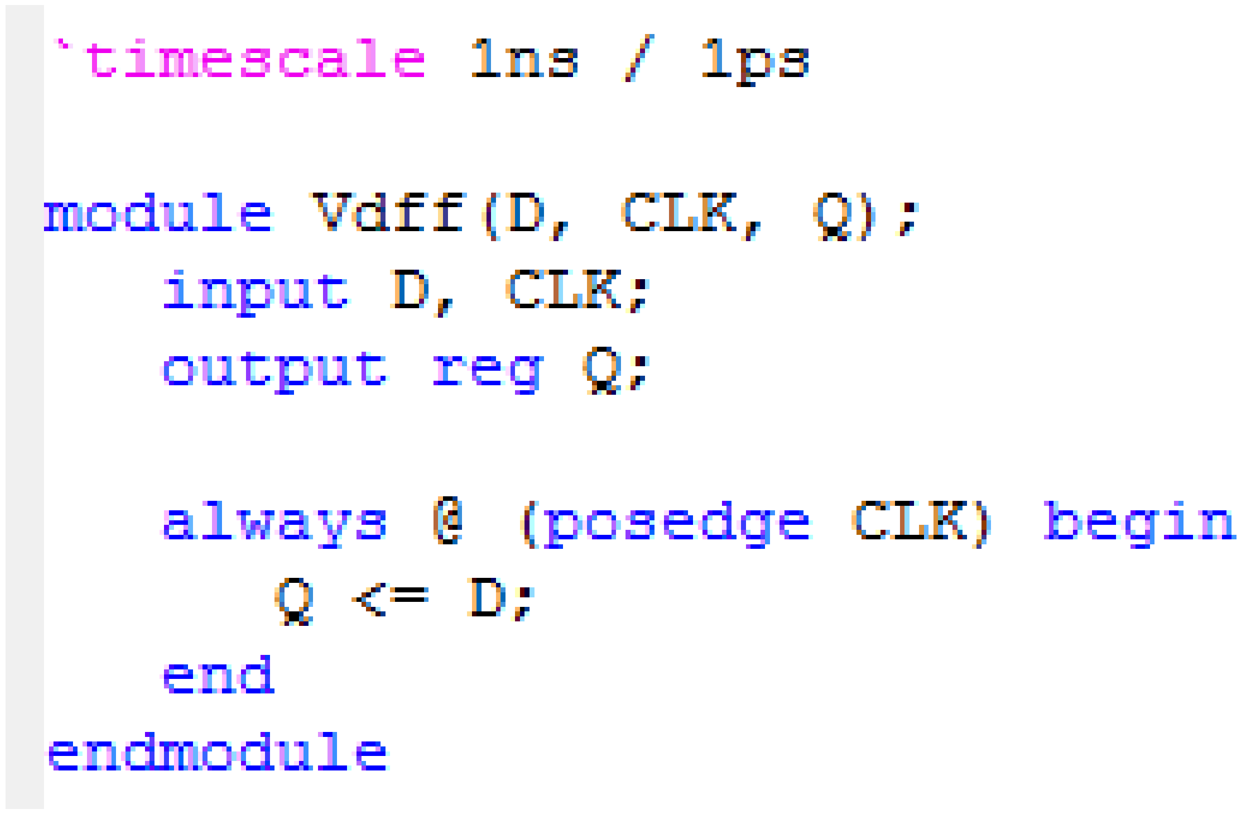
all measurements are made from the clocking event (the rising edge of the clock)

This slide illustrates the timing where the rising edge of the clock signal is the reference. Tpd is the propagation delay between the rising edge of the clock (event trigger) and the change in the output. Tw should be long enough to ensure that D will change Q

Verilog behavioral model of an edge-triggered D flip-flop



- **Use "posedge" attribute (built into Verilog)**
  - posedge SIG is true at the positive edge of SIG

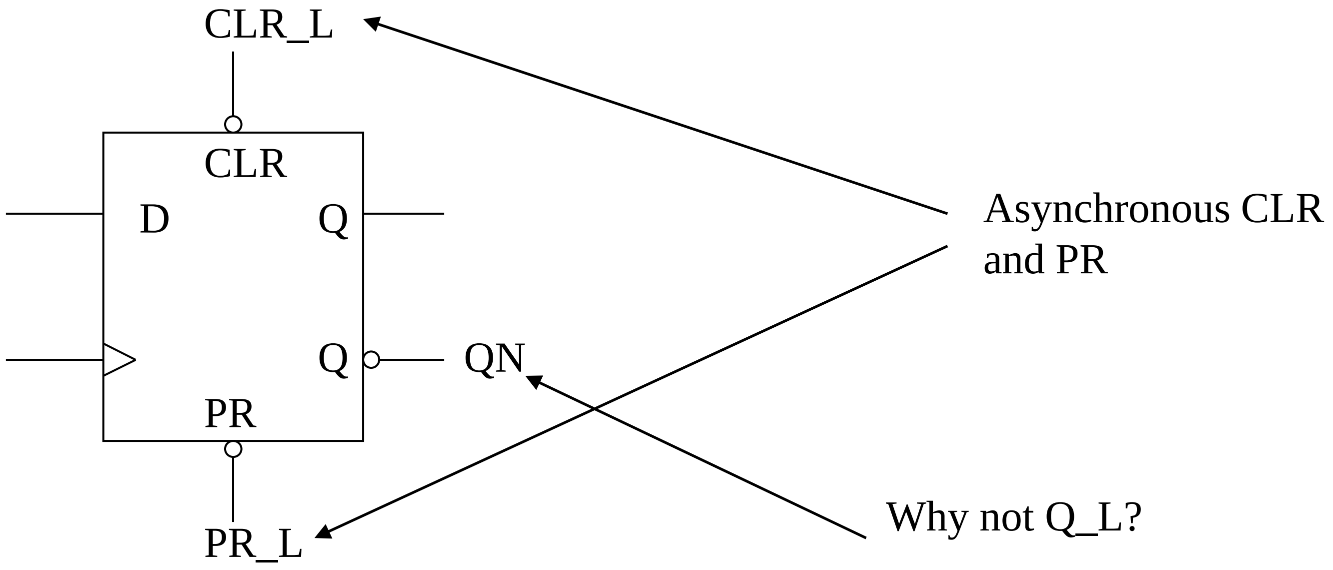


```
`timescale 1ns / 1ps

module Vdff(D, CLK, Q);
    input D, CLK;
    output reg Q;

    always @ (posedge CLK) begin
        Q <= D;
    end
endmodule
```

74x74-like D flip-flop with preset and clear



```
`timescale 1ns / 1ps

module vdff74(Q,QN,D,CLK,PR_L,CLR_L);
   output reg Q, QN;
   input D, CLK, PR_L, CLR_L;

   wire PR = !PR_L;
   wire CLR = !CLR_L;

   always @(posedge CLK or posedge PR or posedge CLR) begin
      if (CLR) begin
         if(PR) begin
            Q <= 0;
            QN <= 0;
         end
      end
      else if (CLR) begin
         Q <= 0;
         QN <= 1;
      end
      else if (PR) begin
         Q <= 1;
         QN <= 0;
      end
      else if (CLK) begin
         Q <= D;
         QN <= !D;
      end
   end
endmodule
```

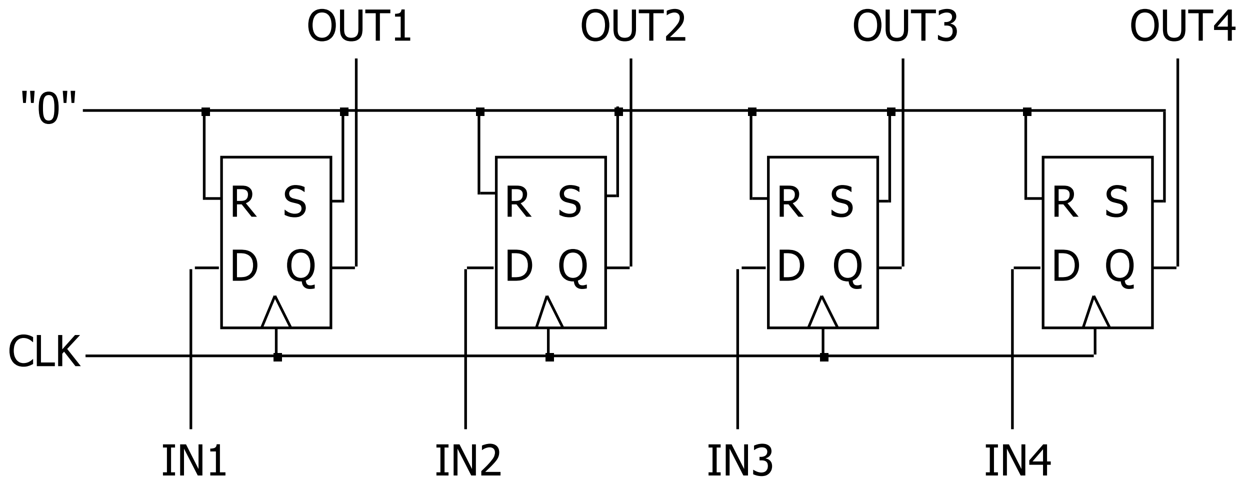
Summary of latches and flip-flops

- **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
- **Preset and clear inputs are highly desirable on flip-flops**
  - used at start-up or to reset system to a known state

D-FFs can be either level-sensitive or edge-triggered. For maintenance purposes, preset and clear inputs are desirable for FFs, which will be discussed later. Preset inputs initialize the values in FFs. Clear inputs will reset the values of FFs to 0s.

Registers

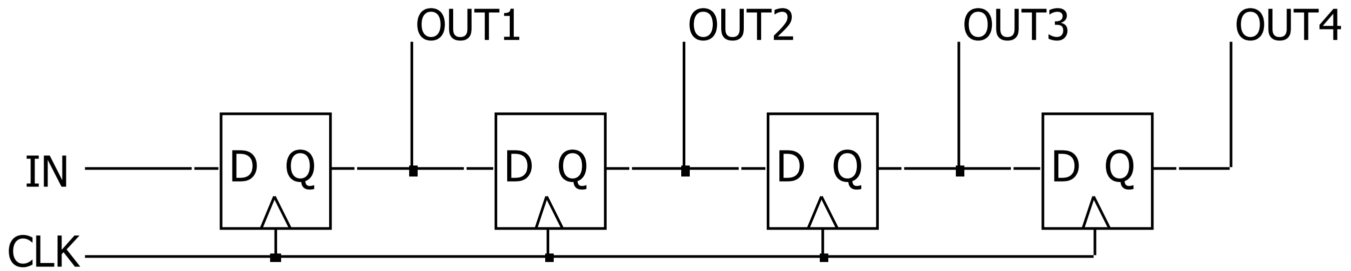
- **Collections of flip-flops with similar controls and logic**
  - stored values somehow related (for example, form binary value)
  - share clock, reset, and set lines
  - similar logic at each stage
- **Examples**
  - shift registers
  - counters



From now on, we will look at a collection of FFs. The first memory element to look at is a register. A register is normally defined as a group of FFs with coordinated controls or shared controls. Examples of controls are clock, reset, set and so on. In this case, we can read/write 4 bits in parallel.

Shift register

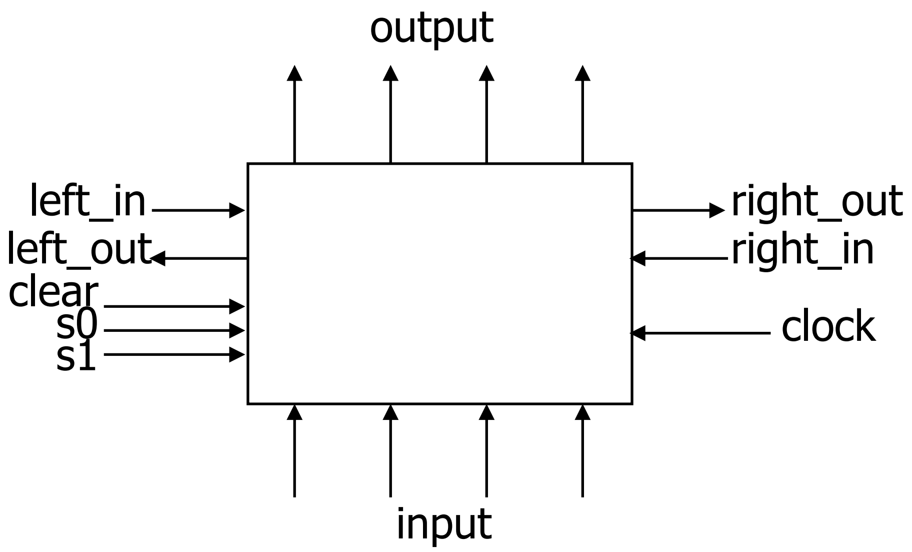
- **Holds samples of input**
  - store last 4 input values in sequence
  - 4-bit shift register:



One of the relatively simple registers is a shift register. Here one bit is shifted (or moved to right) to the next FF at each clock tick (its positive edge). At each positive edge, the stored value will come out and move to the next element.

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

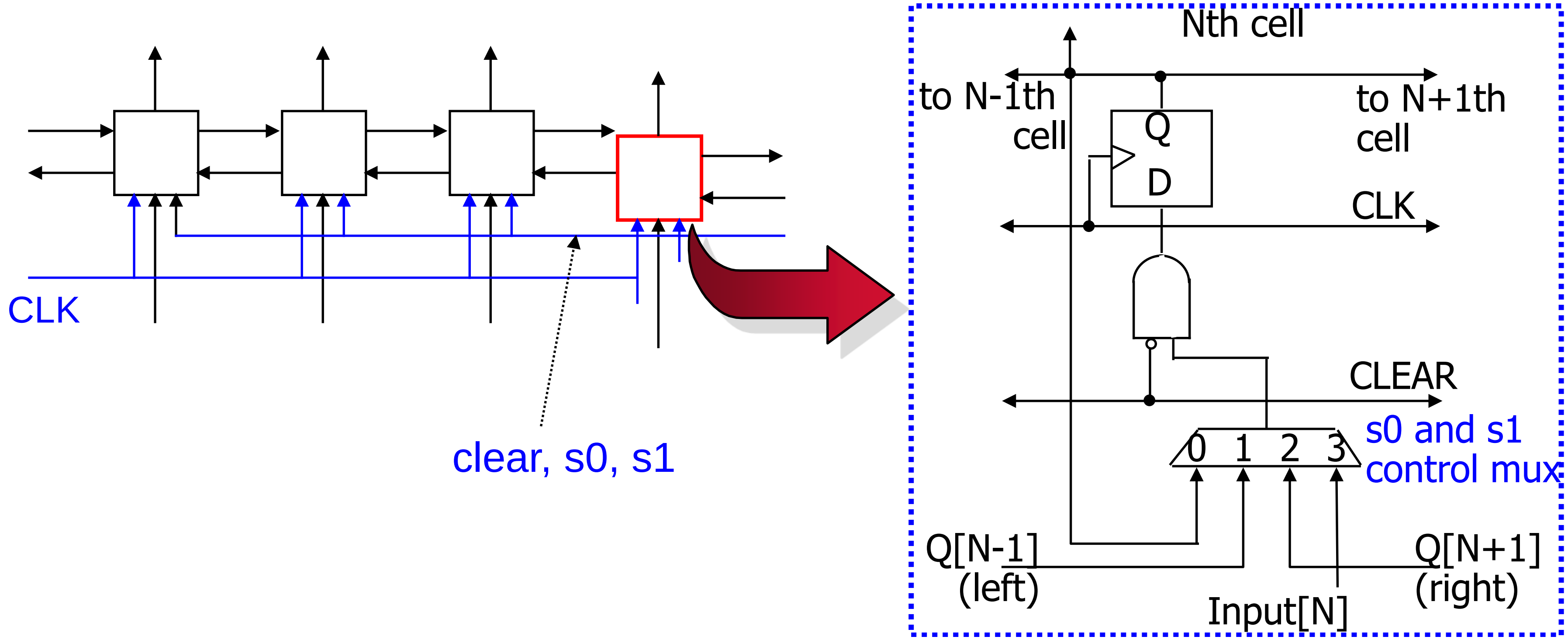
s0	s1	function
0	0	hold state
0	1	shift right
1	0	shift left
1	1	load new input

The shift register in the previous slide goes only from left to right. Here we want to design a generic or multi-purpose shift register with the above functionalities. In addition, we also want to hold the current value without I/O. Overall, we need some control variables.

Design of universal shift register

- **Consider one of the four flip-flops**
  - new value at next clock cycle:

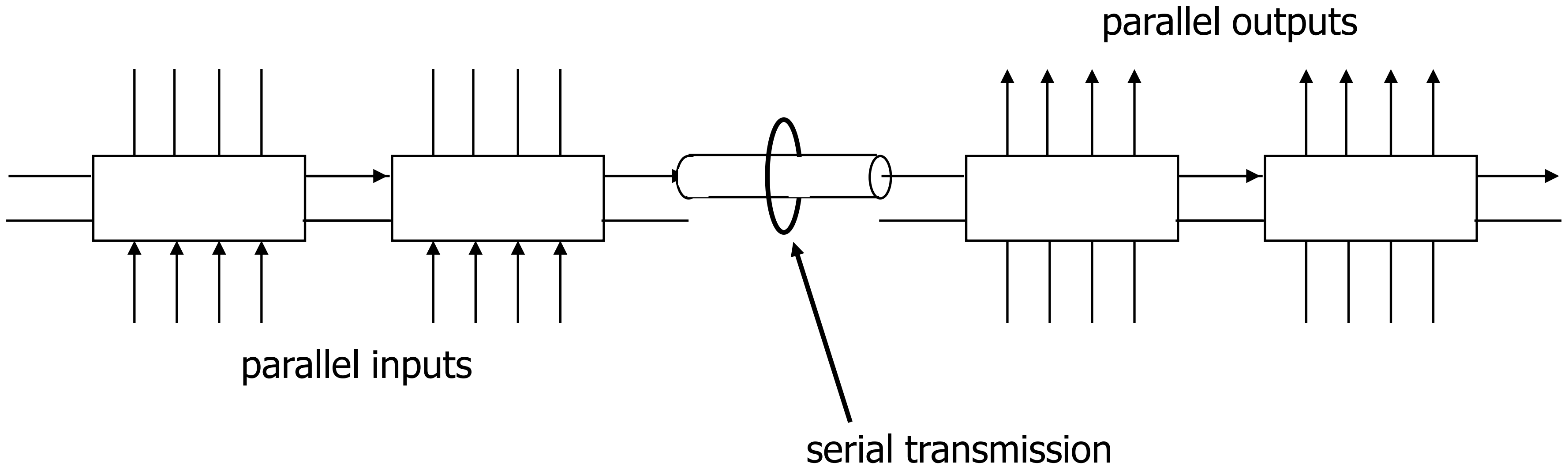
clear	s0	s1	new value
1	–	–	0
0	0	0	output (hold)
0	0	1	output value of left FF (shift right)
0	1	0	output value of right FF(shift left)
0	1	1	input (load)



Each memory module (that stores 1 bit) should be able to perform 5 functions. Note that there are multiple incoming lines and one of them should be selected. This should ring the bell. It will be convenient to use a MUX. Blue wires are about control while black wires are data paths. Here, clear, S1 and S0 are depicted by a single wire for simplicity.

Shift register application

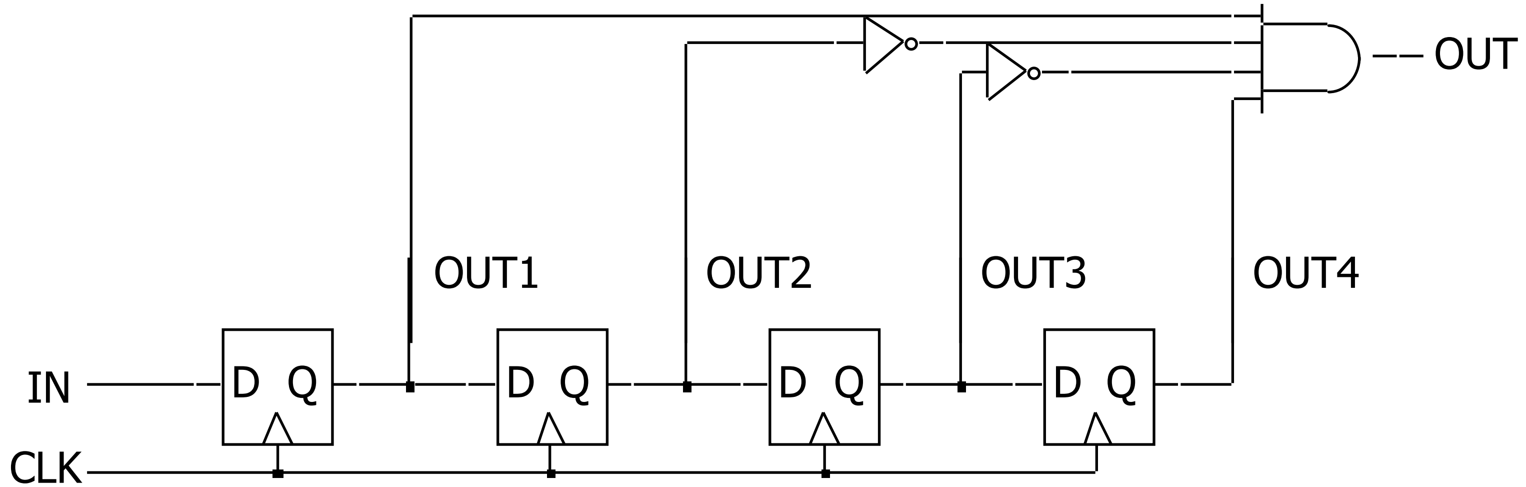
- **Parallel-to-serial conversion for serial transmission**



One of the popular application of the shift register is serial transmission, where information is transmitted over the medium bit-by-bit.

Pattern recognizer

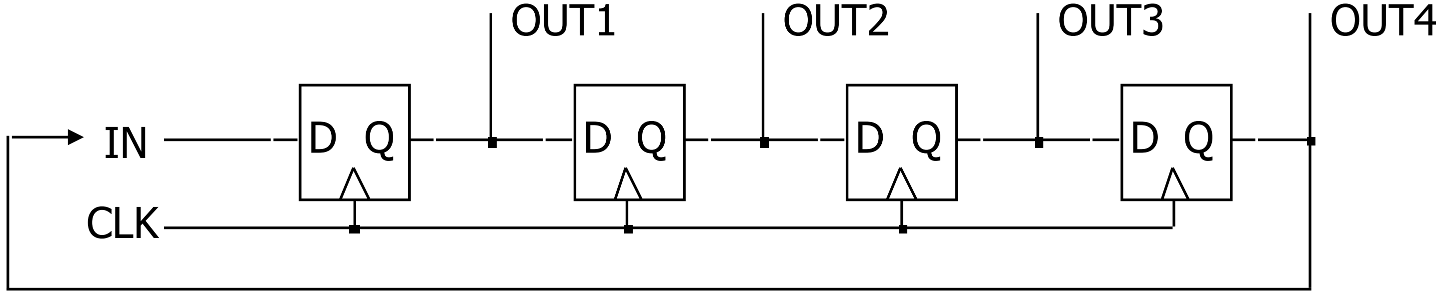
- **Combinational function of input samples**
  - in this case, recognizing the pattern 1001 on the single input signal



Another useful application of shift registers is bit string identification. In this case, bits are shifted from left to right. At any moment, if 4 bits are 1001, then OUT will be true.

Counters

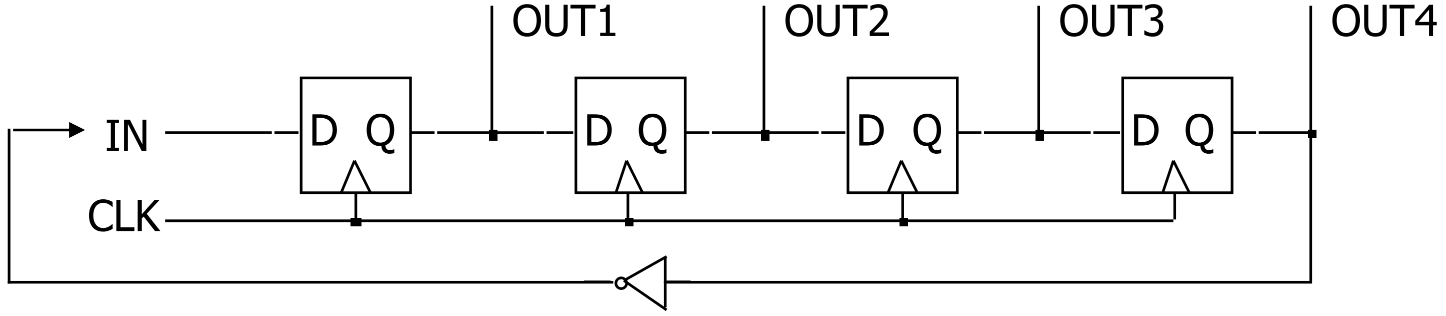
- **Sequences through a fixed set of patterns**
  - in this case, 1000, 0100, 0010, 0001
  - if one of the patterns is its initial state (by loading or set/reset)



If there are multiple patterns that are used for state representation, this register is typically referred to as a counter. Look at the shift register in the slide. Suppose there is a initialization (or preset) logic that stores 1000 in the register, which is not shown here. Then, as the clock ticks, the bits are rotating this ring. That's why it is called a ring counter.

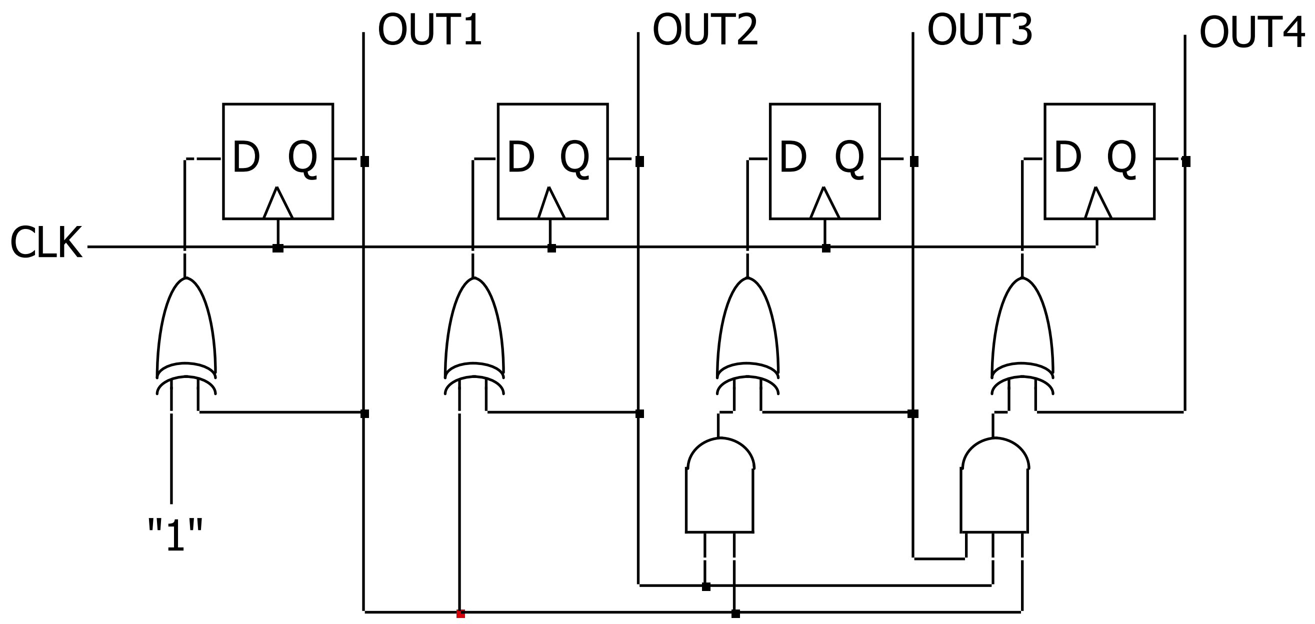
Activity

- **How does this counter work? (initial value: 1000)**



Binary counter

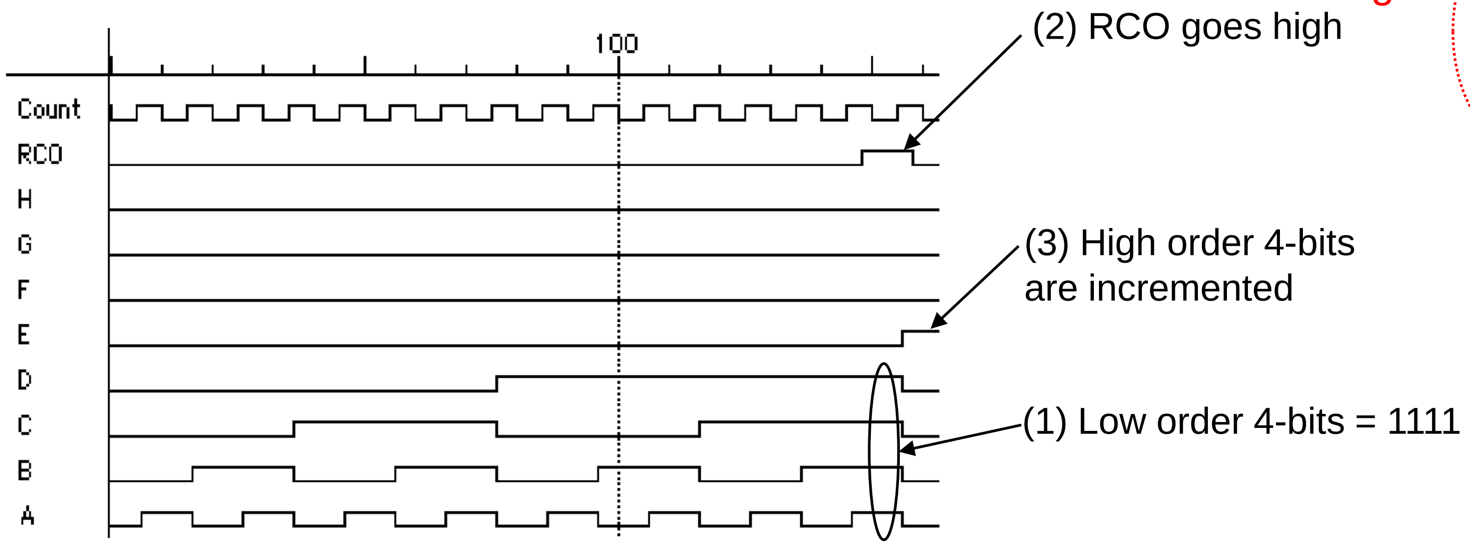
- **Logic between registers (not just multiplexer)**
  - XOR decides when bit should be toggled
  - always for low-order bit,
    only when first bit is true for second bit,
    and so on

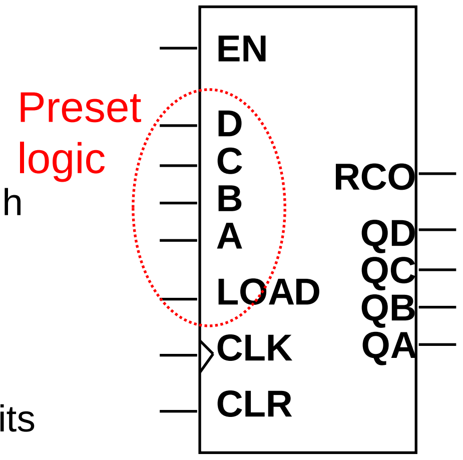


Here is a binary counter; the rule of thumb is that if all lower bits are true, than the upper bit should be toggled. OUT4 is the MSB while OUT1 is the LSB.

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
  - ripple-carry out (RCO) is used for cascading counters
    - high when counter is in its highest state 1111
    - implemented using an AND gate

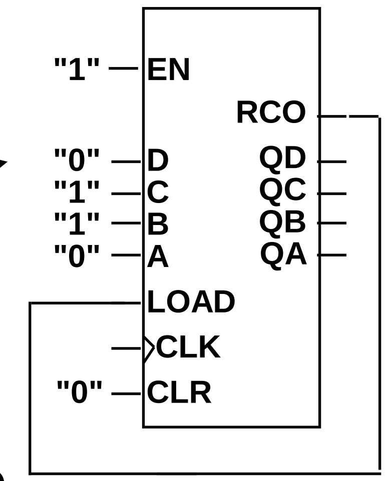




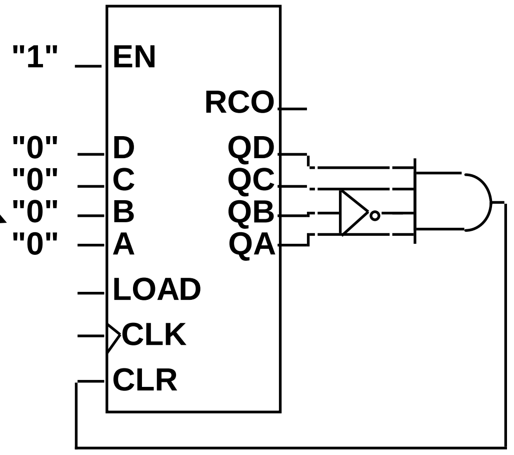
If we use the binary counter in the previous slide as a basic component, we can build many complicated circuits, e.g. a wider binary counter. Here D is the MSB.

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)**

Other examples are here; using the load input, we can control the initial value. Or by using some product term from the stored values, we can configure the ending value of the counter.

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
  - Setup and hold times around the clock edge
- **Basic registers**
  - shift registers
  - counters