

CSE 140: Components and Design Techniques for Digital Systems

Lecture 7: Sequential Networks

CK Cheng

Dept. of Computer Science and Engineering
University of California, San Diego

Part II: Sequential Networks

- Introduction
 - Sequential circuits
 - Memory hierarchy
 - Basic mechanism of memory
- Basic Building Blocks
 - Latches
 - Flip-Flops
 - Examples of Memory Modules
- Implementation
 - Finite state machine

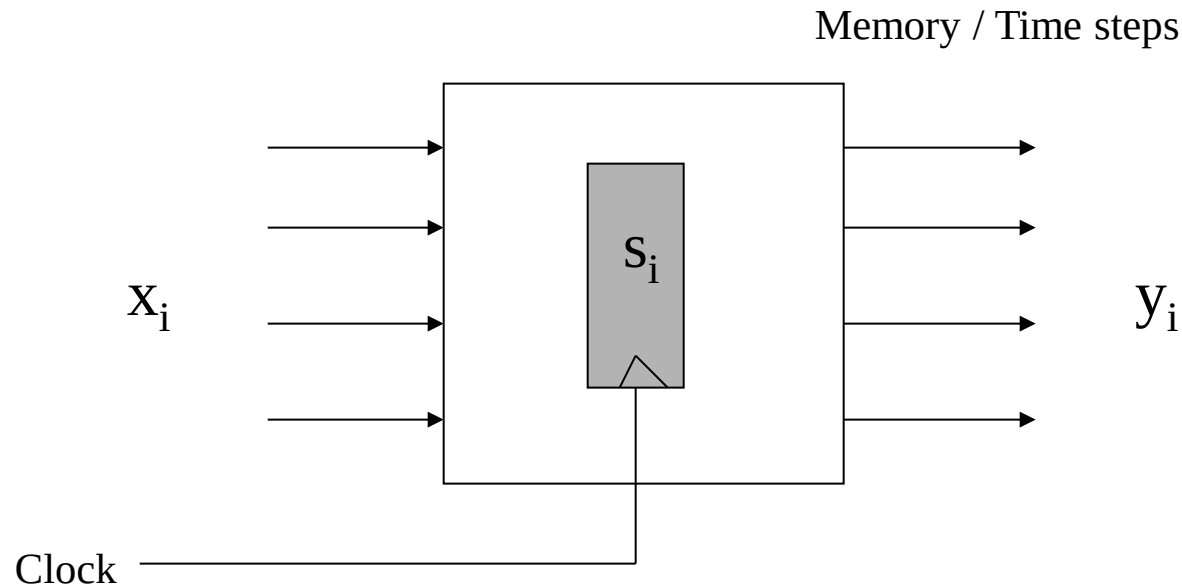


What is a sequential circuit?

“A circuit whose output depends on current inputs and past outputs”

“A circuit with **memory**”

Memory: a key parameter is Time





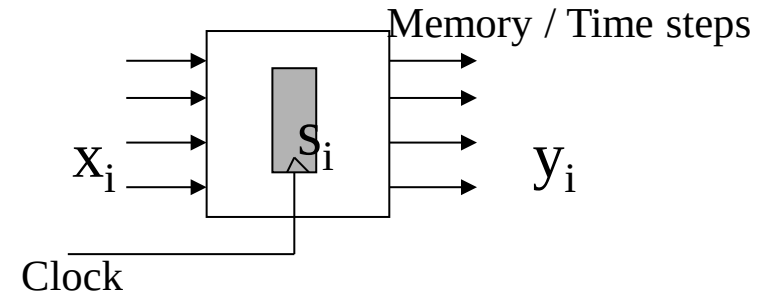
Sequential Networks: Key features

Memory: Flip flops

Specification: Finite State Machines

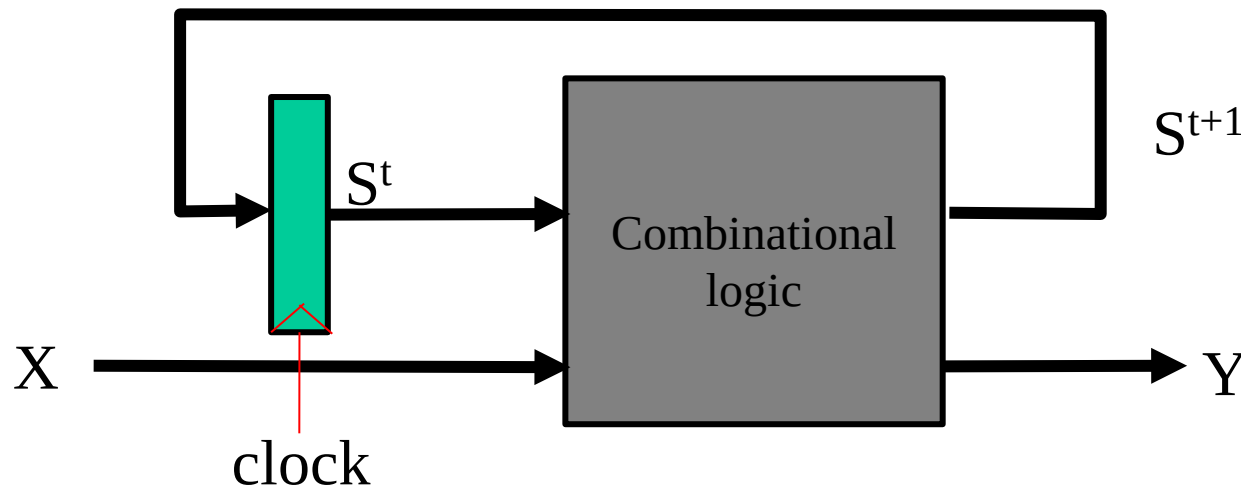
Implementation: Excitation Tables

Main Theme: Timing



Present time = t and next time = $t+1$

Timing constraints to separate the present and next times.



$$y_i = f_i(S^t, X)$$
$$s_i^{t+1} = g_i(S^t, X)$$

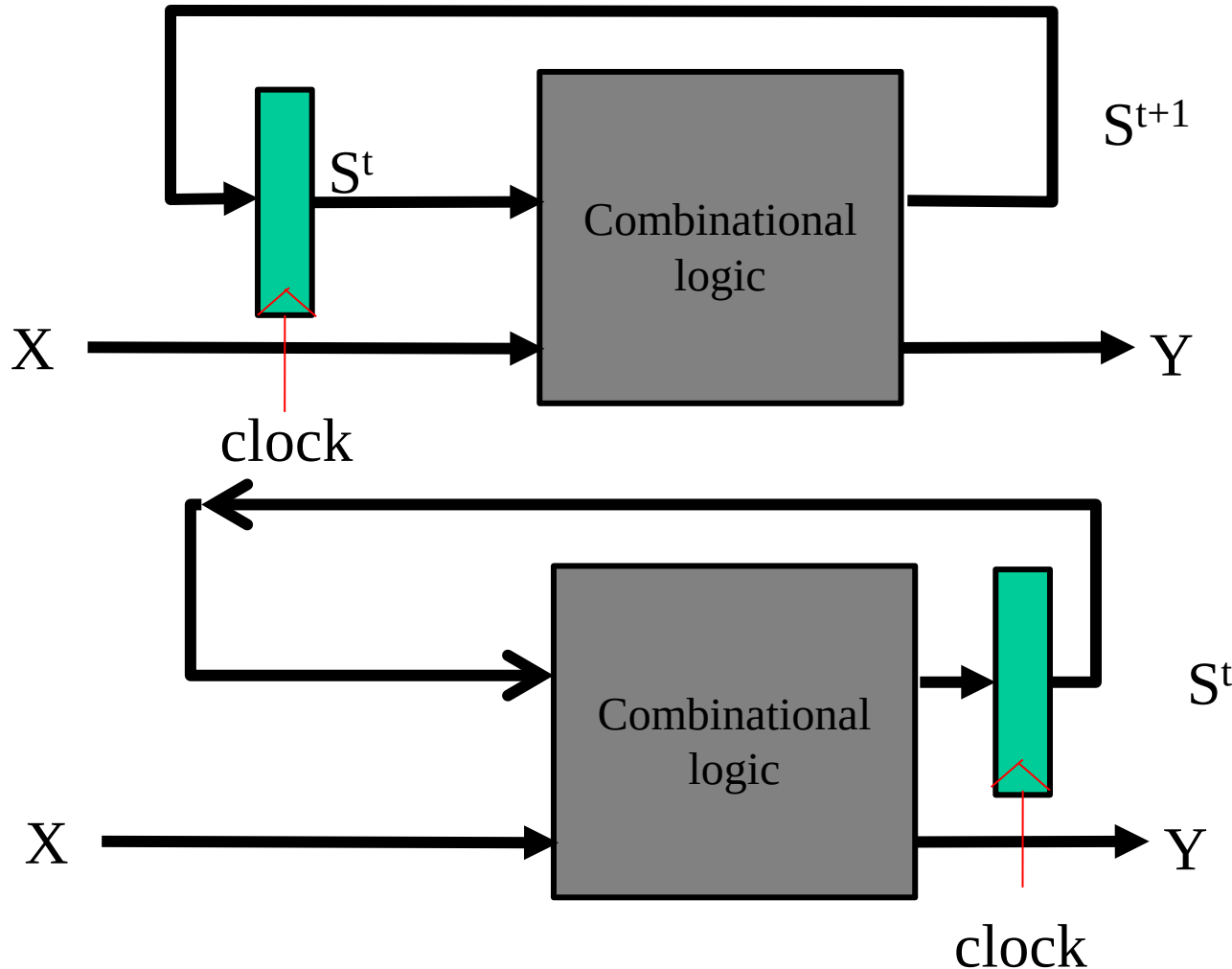


Sequential Networks: Key features

Main Theme: Timing

Present time = t and **next time = $t+1$**

Timing constraints to separate the present and next times.

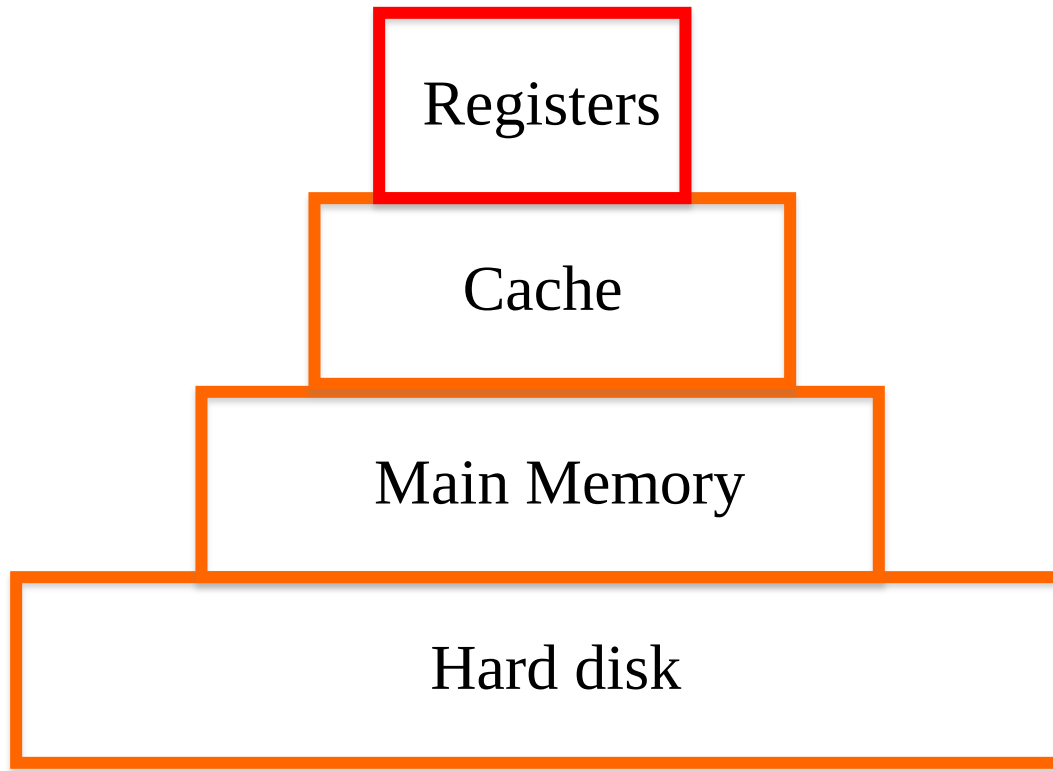


$$y_i = f_i(S^t, X)$$
$$s_i^{t+1} = g_i(S^t, X)$$

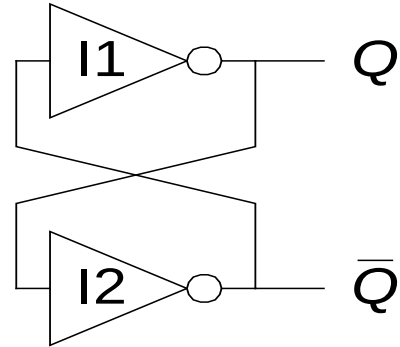
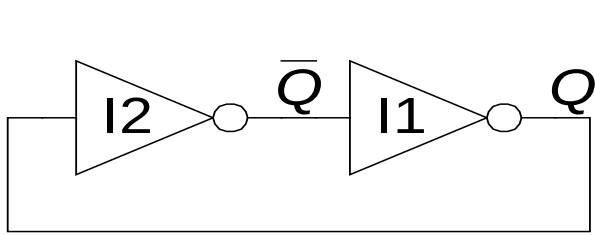
Memory Hierarchy

- What are registers made of?

Flip-Flops, Latches

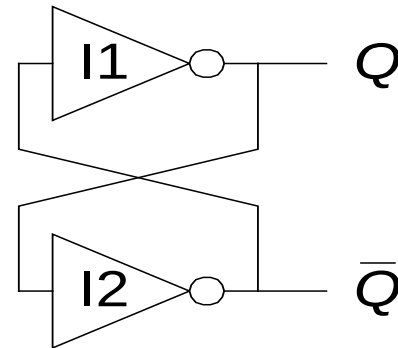
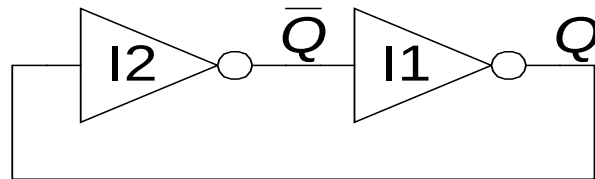


Fundamental Memory Mechanism



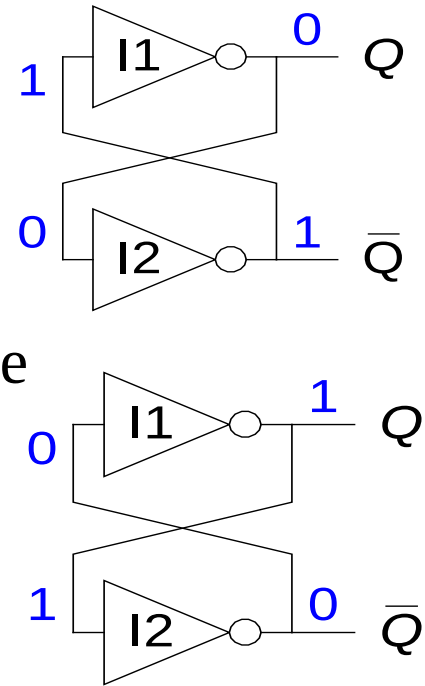
Memory Mechanism: Capacitive Load

- Fundamental building block of sequential circuits
- Two outputs: $\overline{Q}$, Q
- There is a feedback loop!
 - In a typical combinational logic, there is no feedback loop.
- No inputs



Capacitive Loads

- Consider the two possible cases:
 - $Q = 0$: then $Q' = 1$ and $Q = 0$ (consistent)
 - $Q = 1$: then $Q' = 0$ and $Q = 1$ (consistent)
 - Bistable circuit stores 1 bit of state in the state variable, Q (or Q')
 - Hold the value due to capacitive charges and feedback loop strengthening
- But there are **no inputs to control the state**



iClicker

Q. Given a memory component made out of a loop of inverters, the number of inverters in the loop has to be

- A. Even
- B. Odd
- C. No constraints

Basic Building Blocks

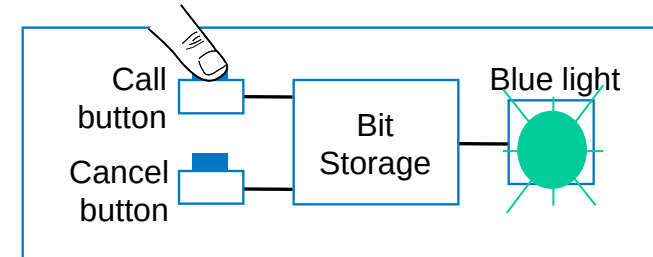
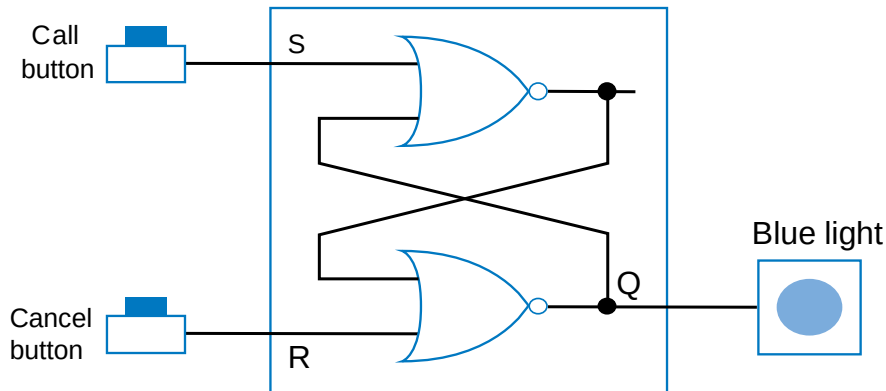
- Latches (Level Sensitive)
 - SR Latches, D Latches
- Flip-Flops (Edge Triggered)
 - D FFs, (JK FFs, T FFs)
- Examples of Memory Modules
 - Registers, Shift Registers, Pattern Recognizers, Counters, FIFOs

Flight attendant call button

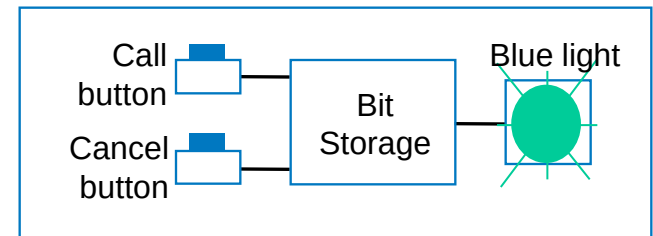
- Flight attendant call button
 - Press call: light turns on
 - **Stays on** after button released
 - Press cancel: light turns off
 - Logic gate circuit to implement this?

- SR latch implementation

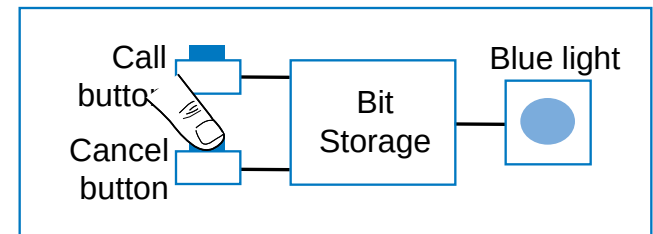
- Call=1 : sets Q to 1 and keeps it at 1
- Cancel=1 : resets Q to 0



1. Call button pressed – light turns on



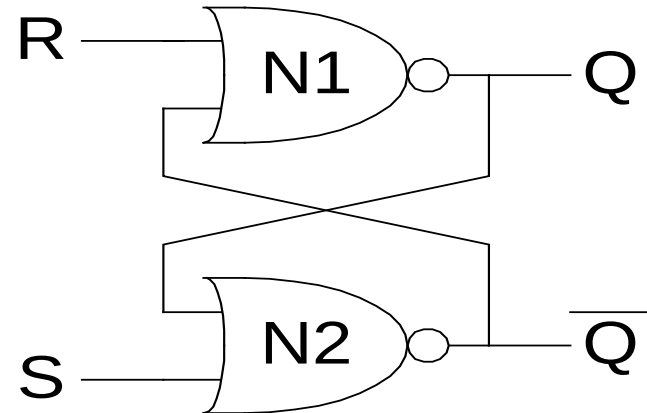
2. Call button released – light **stays on**



3. Cancel button pressed – light turns off

SR (Set/Reset) Latch

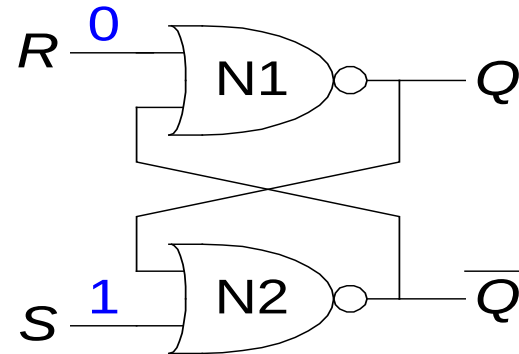
- SR Latch



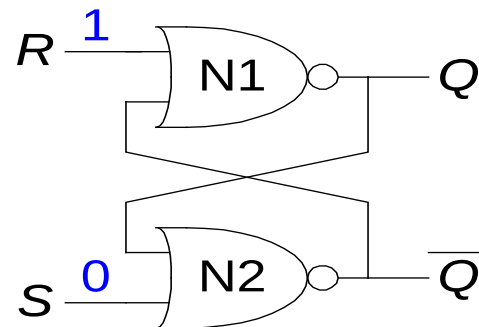
- Consider the four possible cases:
 - $S = 1, R = 0$
 - $S = 0, R = 1$
 - $S = 0, R = 0$
 - $S = 1, R = 1$

SR Latch Analysis

– $S = 1, R = 0$: then $Q = 1$ and $\overline{Q} = 0$



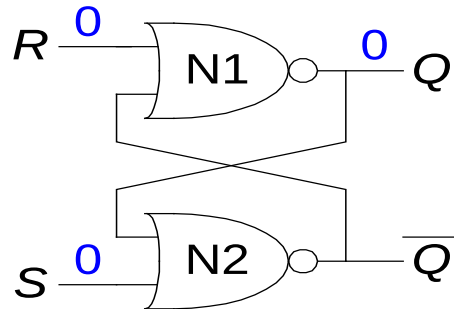
– $S = 0, R = 1$: then $Q = 0$ and $\overline{Q} = 1$



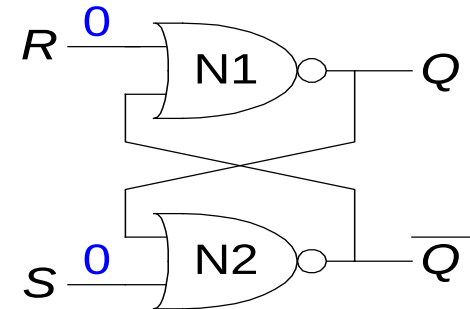
SR Latch Analysis

– $S = 0, R = 0$: then $Q = Q_{prev}$

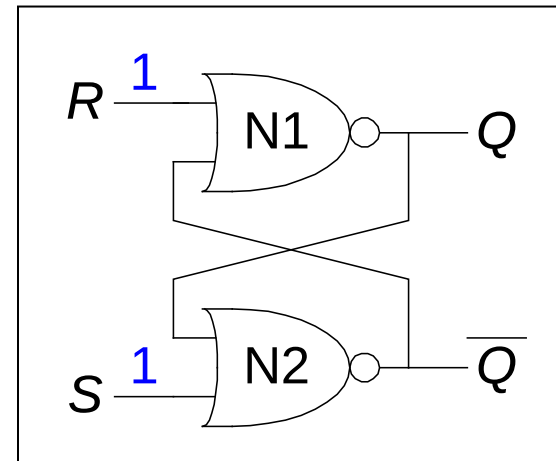
$Q_{prev} = 0$



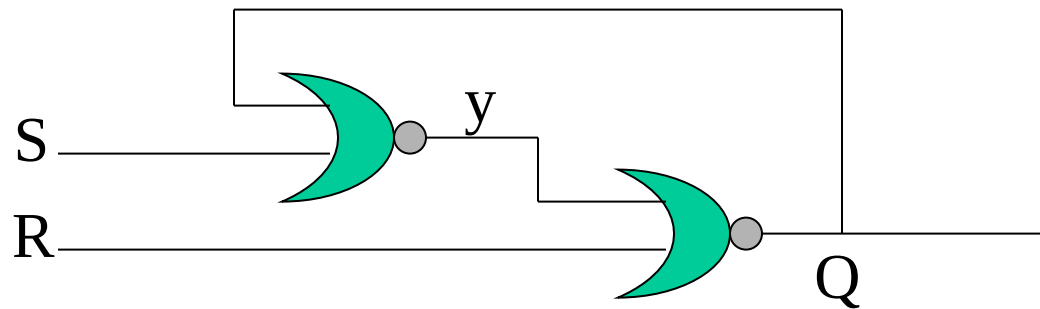
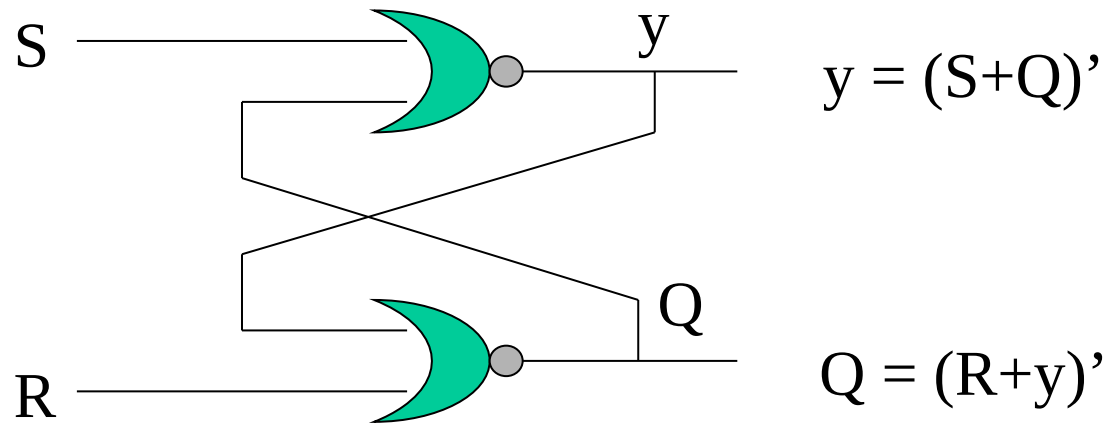
$Q_{prev} = 1$



– $S = 1, R = 1$: then $Q = 0$ and $\overline{Q} = 0$



SR Latch

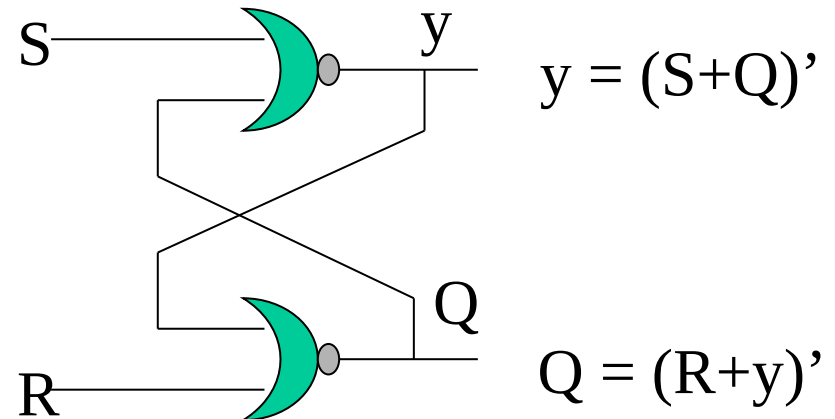


Inputs: S, R

State: (Q, y)

id	S	R	Qt	yt	Qt	yt	Qt	yt	Qt	yt
0	0	0	0	0	1	1	0	0	1	1
1	0	0	0	1	0	1	0	1	0	1
2	0	0	1	0	1	0	1	0	1	0
3	0	0	1	1	0	0	1	1	0	0
4	0	1	0	0	0	1	0	1	0	1
5	0	1	0	1	0	1	0	1	0	1
6	0	1	1	0	0	0	0	1	0	1
7	0	1	1	1	0	0	0	1	0	1
8	1	0	0	0	1	0	1	0	1	0
9	1	0	0	1	0	0	1	0	1	0
10	1	0	1	0	1	0	1	0	1	0
11	1	0	1	1	0	0	1	0	1	0
12	1	1	0	0	0	0	0	0	0	0
13	1	1	0	1	0	0	0	0	0	0
14	1	1	1	0	0	0	0	0	0	0
15	1	1	1	1	0	0	0	0	0	0

Truth table of SR latch
with incremental steps in
time



id	S	R	Qt	yt	Qt	yt	Qt	yt	Qt	yt
0	0	0	0	0	1	1	0	0	1	1
1	0	0	0	1	0	1	0	1	0	1
2	0	0	1	0	1	0	1	0	1	0
3	0	0	1	1	0	0	1	1	0	0
4	0	1	0	0	0	1	0	1	0	1
5	0	1	0	1	0	1	0	1	0	1
6	0	1	1	0	0	0	0	1	0	1
7	0	1	1	1	0	0	0	1	0	1
8	1	0	0	0	1	0	1	0	1	0
9	1	0	0	1	0	0	1	0	1	0
10	1	0	1	0	1	0	1	0	1	0
11	1	0	1	1	0	0	1	0	1	0
12	1	1	0	0	0	0	0	0	0	0
13	1	1	0	1	0	0	0	0	0	0
14	1	1	1	0	0	0	0	0	0	0
15	1	1	1	1	0	0	0	0	0	0

“State Table” of SR latch

Qy\SR	00	01	10	11
00	11	01	10	00
01	01	01	10	00
10	10	01	10	00
11	00	01	10	00

CASES:

SR=01: $(Q,y) = (0,1)$

SR=10: $(Q,y) = (1,0)$

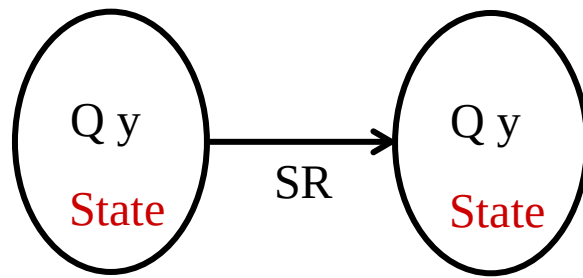
SR=11: $(Q,y) = (0,0)$

SR= 00: (Q,y) does not change if $(Q,y)=(1,0)$ or $(0,1)$

However, when $(Q,y) = (0,0)$ or $(1,1)$, the output keeps changing

Remark: To verify the design, we need to enumerate all combinations.

State Table and State Diagram

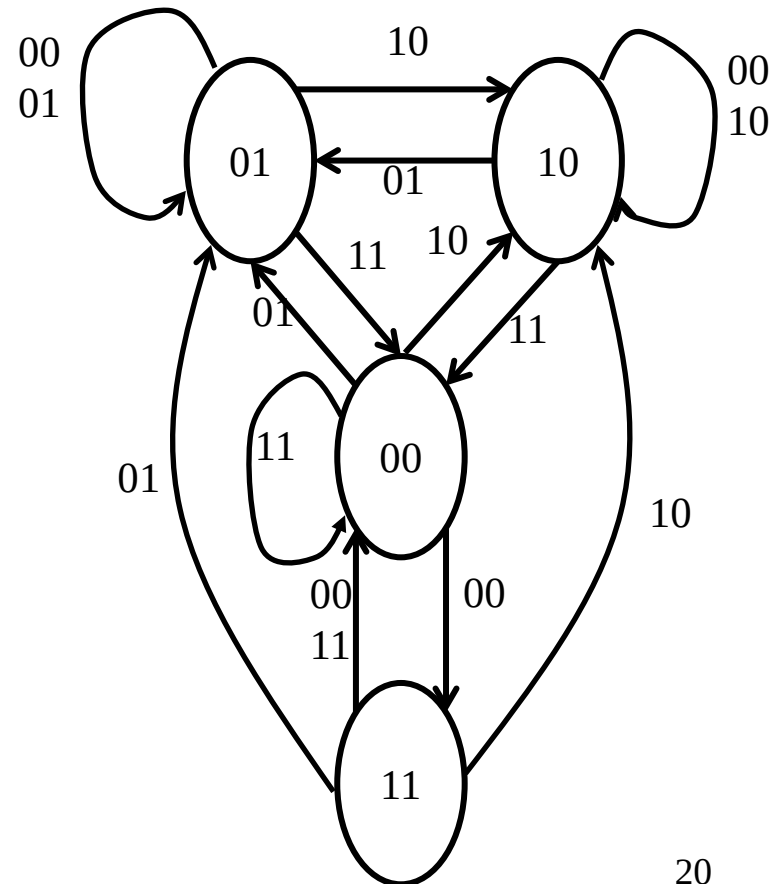


Present State Transition Next State

State diagram

State Table

\SR Qy	00	01	10	11
00	11	01	10	00
01	01	01	10	00
10	10	01	10	00
11	00	01	10	00



CASES:

SR=01: $(Q,y) = (0,1)$

SR=10: $(Q,y) = (1,0)$

SR=11: $(Q,y) = (0,0)$

SR= 00: (Q,y) does not change if $(Q,y)=(1,0)$ or $(0,1)$

However, when $(Q,y) = (0,0)$ or $(1,1)$, the output keeps changing

Q. Suppose that we can set the initial state $(Q,y)=(0,1)$. To avoid the SR latch output from toggling or behaving in an undefined way which input combinations should be avoided:

A. $(S, R) = (0, 0)$

B. $(S, R) = (1, 1)$

C. None of the above

CASES:

SR=01: $(Q,y) = (0,1)$

SR=10: $(Q,y) = (1,0)$

SR=11: $(Q,y) = (0,0)$

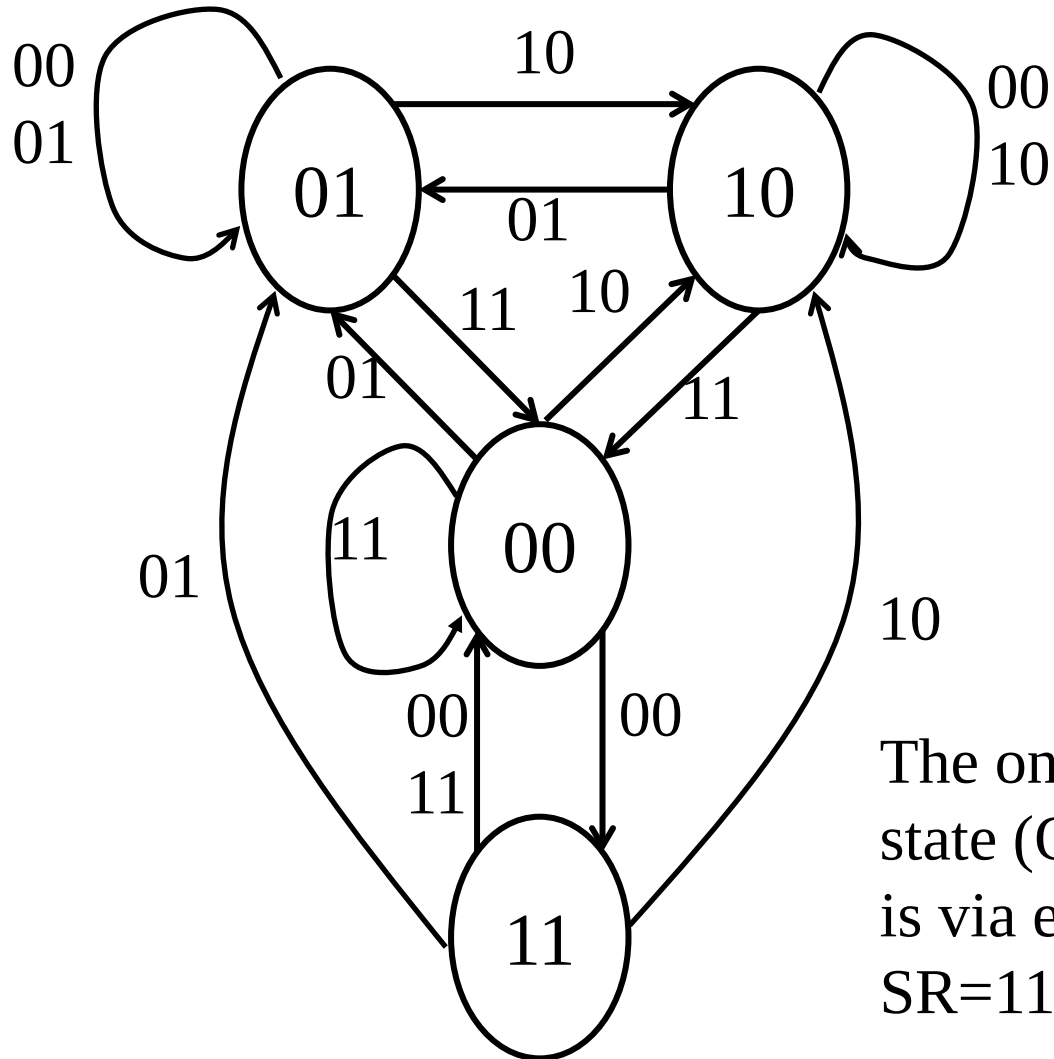
SR= 00: (Q,y) does not change if $(Q,y)=(1,0)$ or $(0,1)$

However, when $(Q,y) = (0,0)$ or $(1,1)$, the output keeps changing

We set the initial state $(Q,y)=(0,1)$ or $(1,0)$. To avoid the state $(Q,y)= (0,0)$ or $(1,1)$, we block the input SR=11.

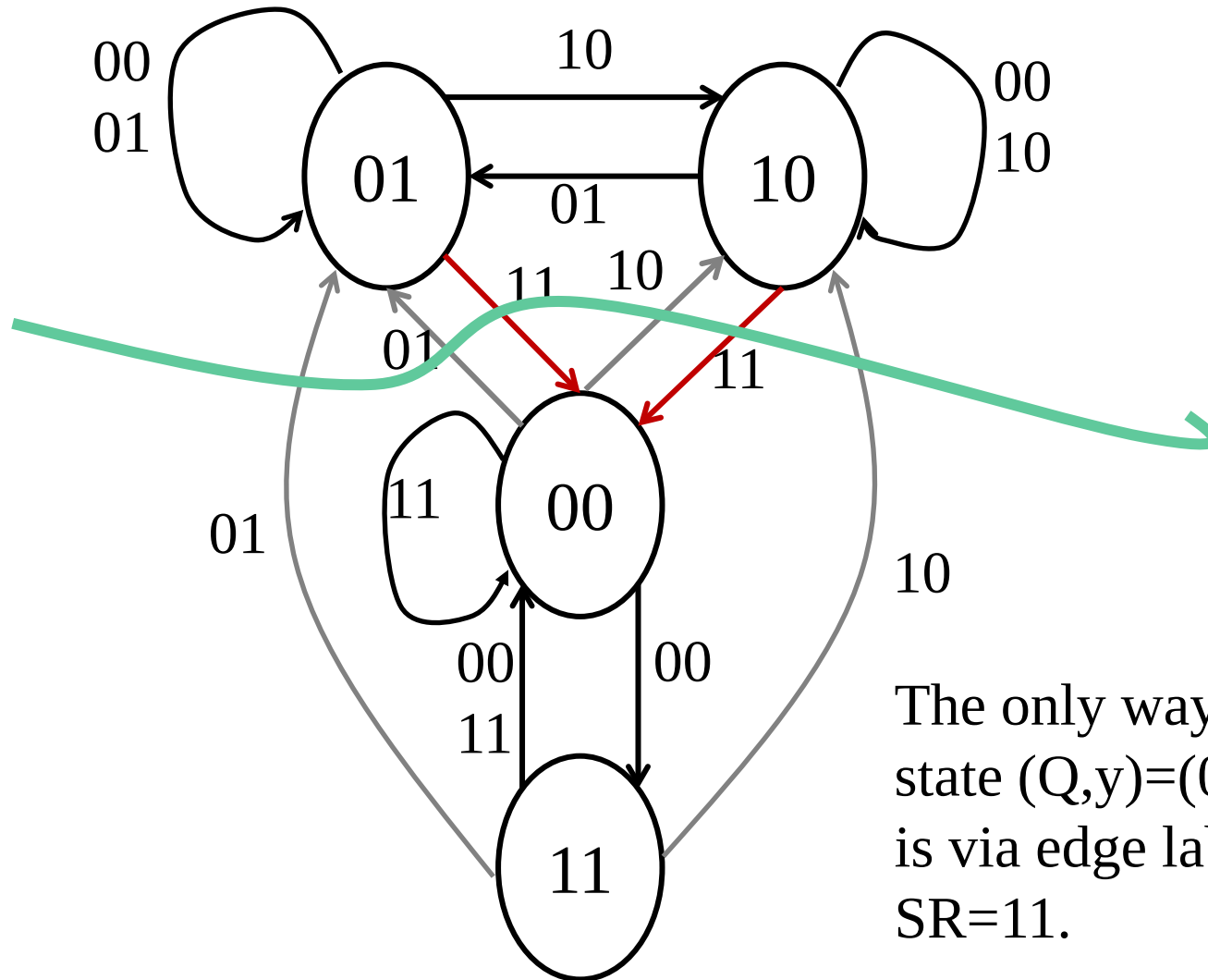
Thus, without input SR=11, the state can only be $(Q,y)=(0,1)$ or $(1,0)$.

SR Latch



The only way to reach state $(Q,y)=(0,0)$ or $(1,1)$ is via edge labeled $SR=11$.

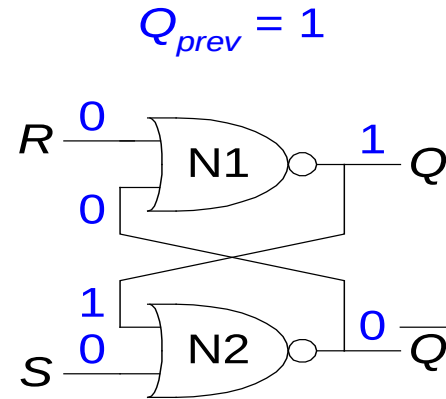
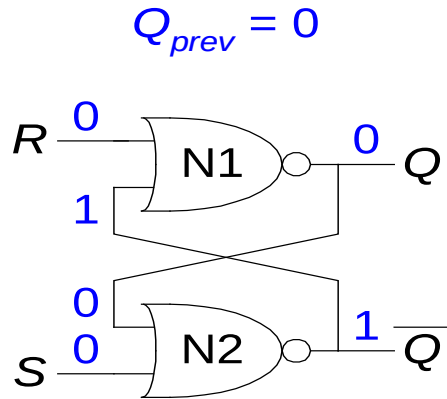
SR Latch



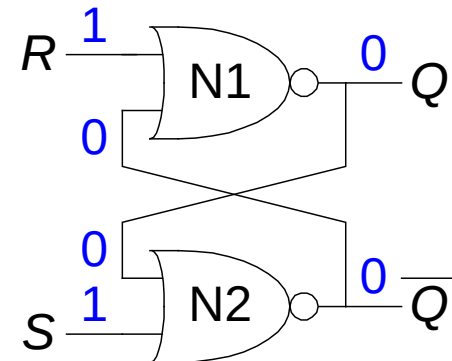
The only way to reach state $(Q,y)=(0,0)$ or $(1,1)$ is via edge labeled $SR=11$.

SR Latch Analysis

- $S = 0, R = 0$: then $Q = Q_{prev}$ and $\bar{Q} = \bar{Q}_{prev}$ (**memory!**)



- $S = 1, R = 1$: then $Q = 0$ and $\bar{Q} = 0$ (**invalid state**: $Q \neq \text{NOT } \bar{Q}$)



SR Latch

CASES

SR=01: $(Q,y) = (0,1)$

SR=10: $(Q,y) = (1,0)$

SR=11: $(Q,y) = (0,0)$

SR = 00: if $(Q,y) = (0,0)$ or $(1,1)$, the output keeps changing

Solutions: Avoid the case that SR = (1,1).

State table

		SR			
		00	01	10	11
PS	inputs				
Q(t)	0	0	0	1	-
	1	1	0	1	-

Characteristic Expression

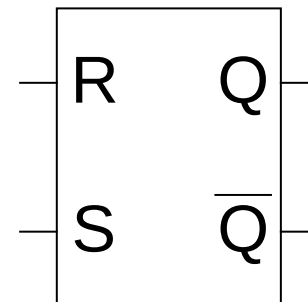
$$Q(t+1) = S(t) + R'(t)Q(t)$$

Q(t+1) NS (next state)

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 ($S = 0$, $R = 1$, $Q = 0$)
- **Must do something to avoid invalid state (when $S = R = 1$)**

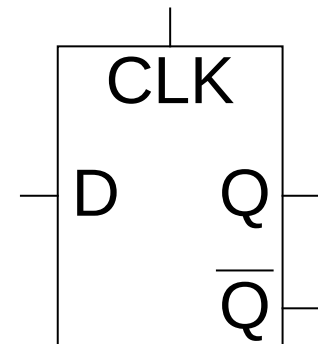
SR Latch
Symbol



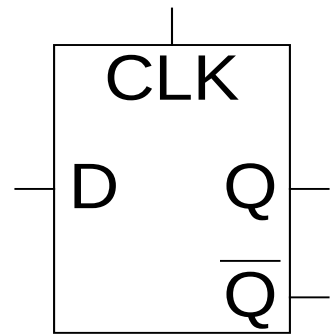
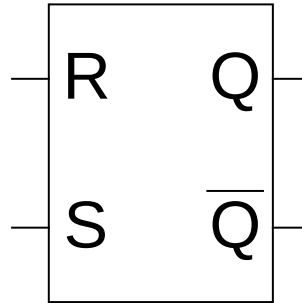
D Latch

- 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 (the latch is *transparent*)
 - When $CLK = 0$, Q holds its previous value (the latch is *opaque*)
- Avoids invalid case when $Q \neq \text{NOT } \bar{Q}$

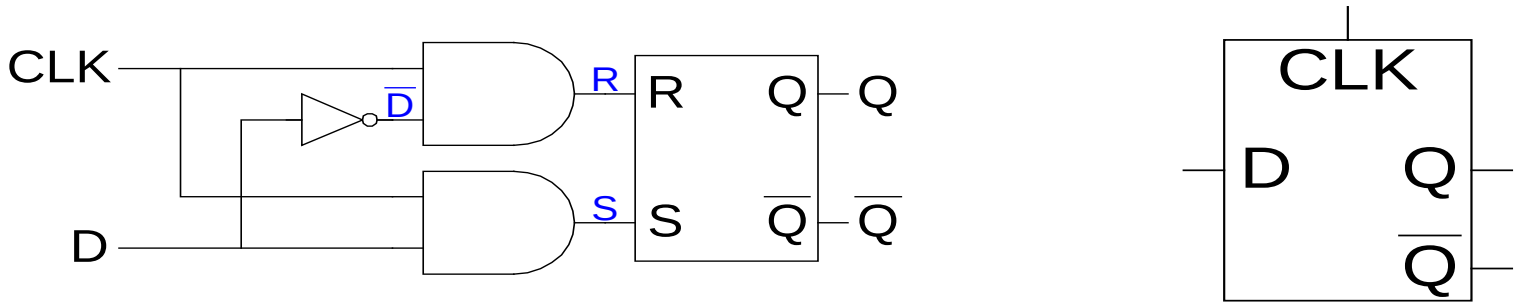
D Latch
Symbol



D Latch Internal Circuit

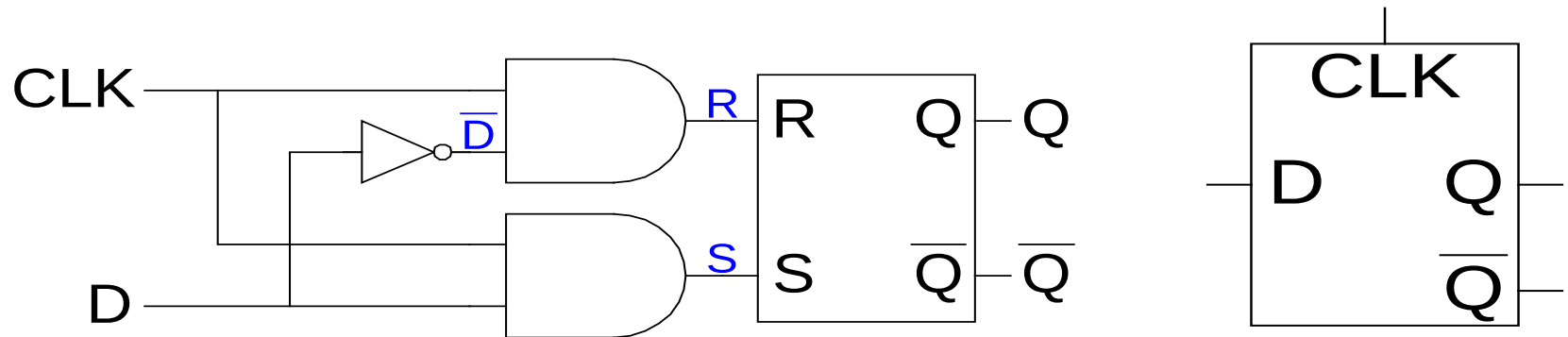


D Latch Internal Circuit



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

D Latch Internal Circuit

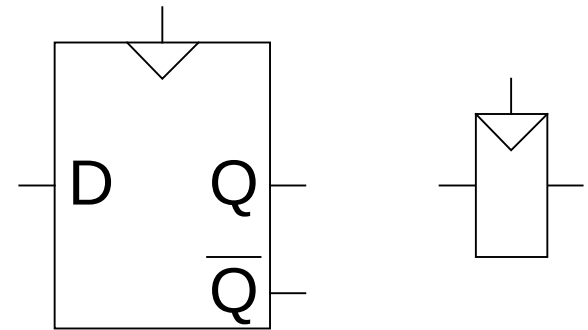


CLK	D	$\overline{D}$	S	R	Q	$\overline{Q}$
0	X	$\overline{X}$	0	0	Q_{prev}	$\overline{Q}_{prev}$
1	0	1	0	1	0	1
1	1	0	1	0	1	0

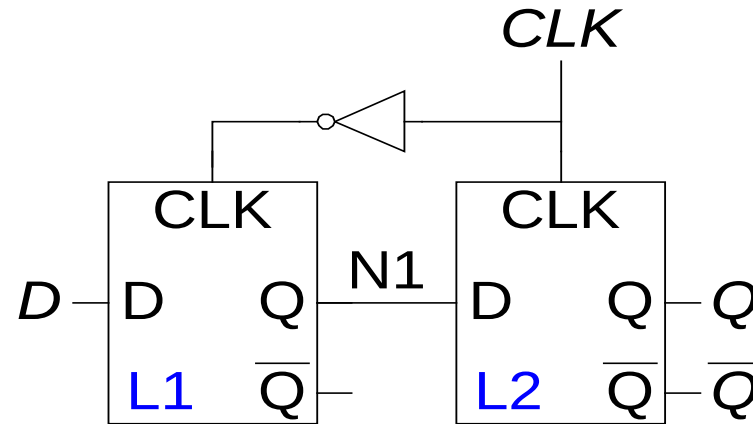
D Flip-Flop

- Two inputs: CLK , D
- *Function*
 - The flip-flop “samples” D on the 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 the rising edge of CLK
- A flip-flop is called an *edge-triggered* device because it is activated on the clock edge

D Flip-Flop Symbols

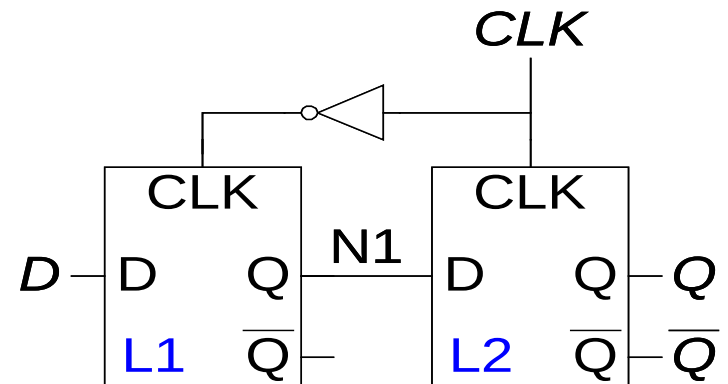


D Flip-Flop Internal Circuit

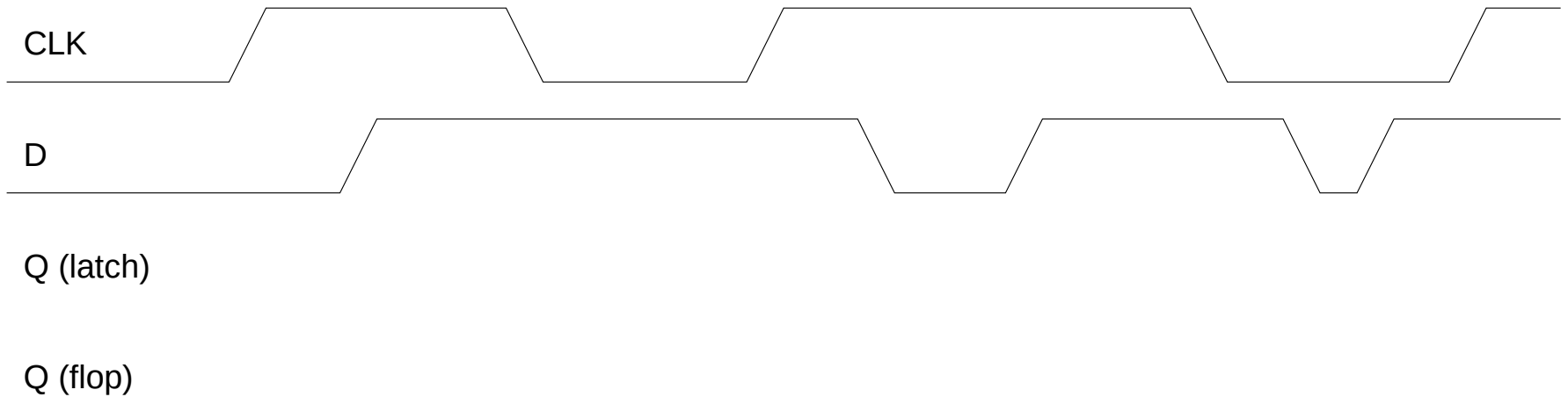
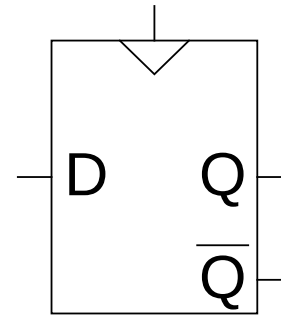
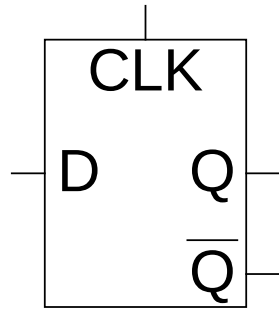


D Flip-Flop Internal Circuit

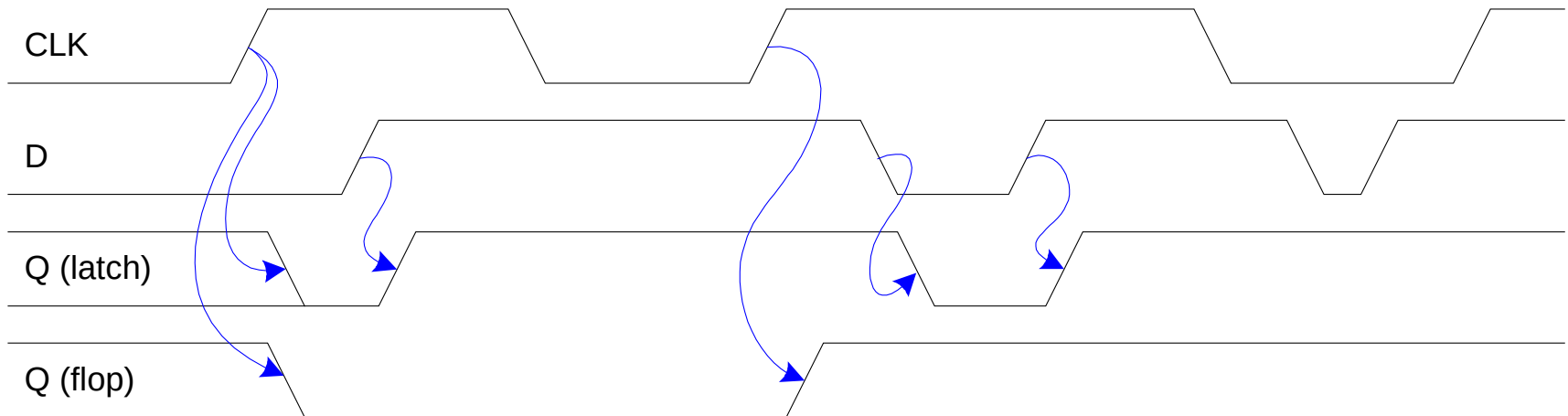
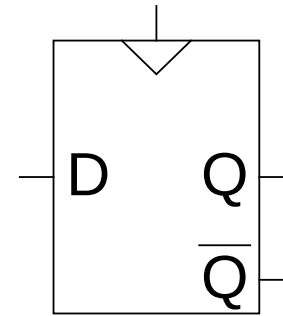
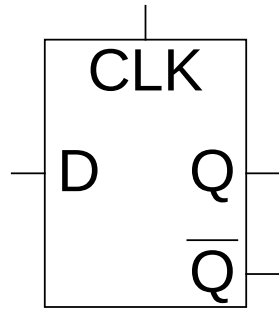
- Two back-to-back latches (L1 and L2) controlled by complementary clocks
- When $CLK = 0$
 - L1 is transparent, L2 is opaque
 - D passes through to N1
- When $CLK = 1$
 - L2 is transparent, L1 is opaque
 - N1 passes through to Q
- Thus, on the edge of the clock (when CLK rises from 0 to 1)
 - D passes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



D Flip-Flop vs. D Latch

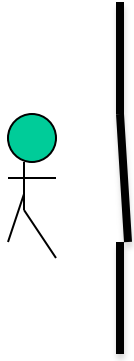


D Flip-Flop vs. D Latch

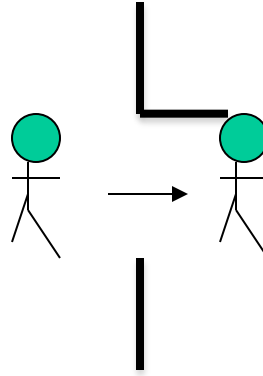


Latch and Flip-flop (two latches)

A latch can be considered as a door

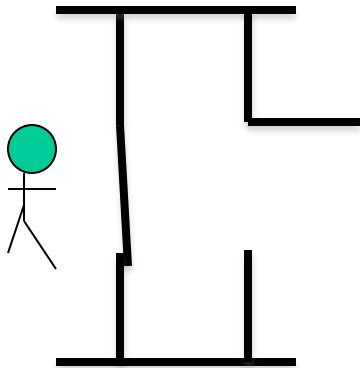


CLK = 0, door is shut

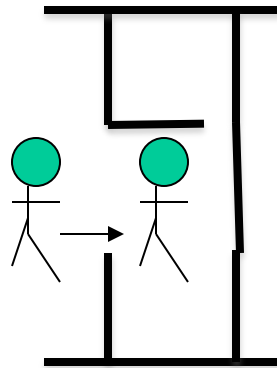


CLK = 1, door is unlocked

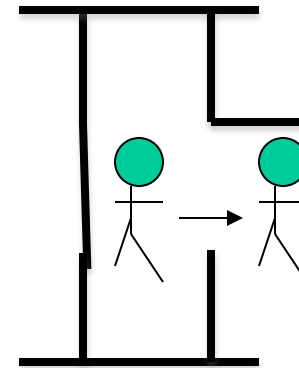
A flip-flop is a two door entrance



CLK = 1

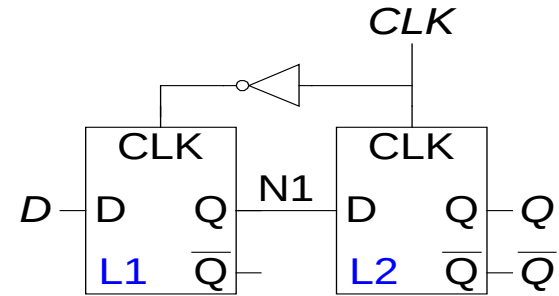
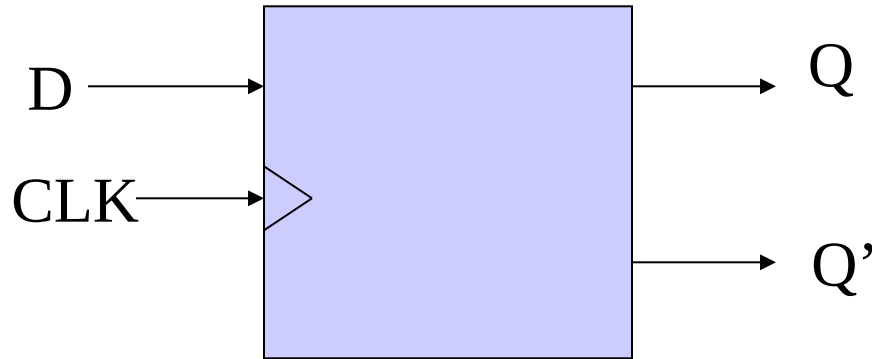


CLK = 0



CLK = 1

D Flip-Flop (Delay)



Id	D	Q(t)	Q(t+1)
0	0	0	0
1	0	1	0
2	1	0	1
3	1	1	1

State table

PS	D	
	0	1
0	0	1
1	0	1

NS= Q(t+1)

Characteristic Expression: $Q(t+1) = D(t)$

iClicker

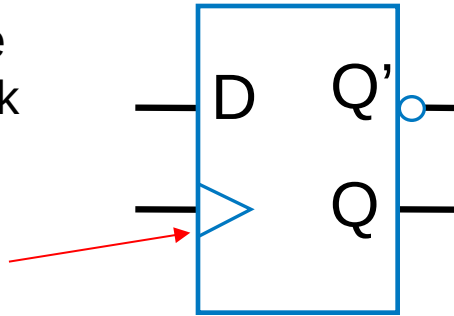
Can D flip-flop serve as a memory component?

A. Yes

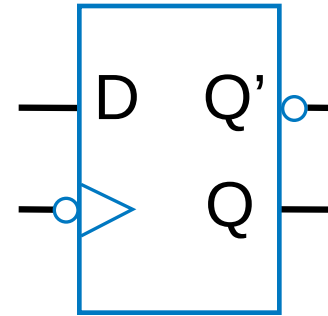
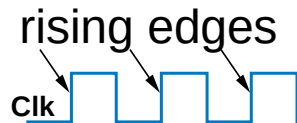
B. No

Rising vs. Falling Edge D Flip-Flop

The triangle means clock input, edge triggered

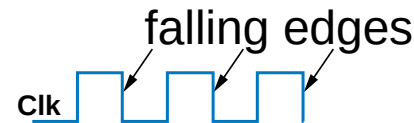


Symbol for rising-edge triggered D flip-flop



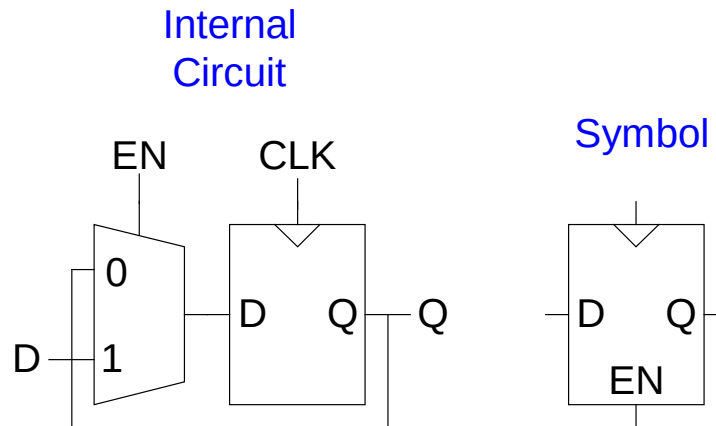
Internal design:
Just invert servant clock rather than master

Symbol for falling-edge triggered D flip-flop



Enabled D-FFs

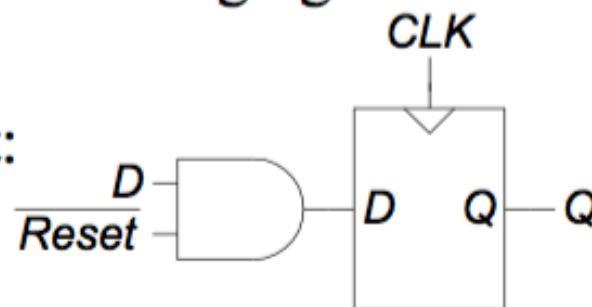
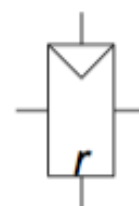
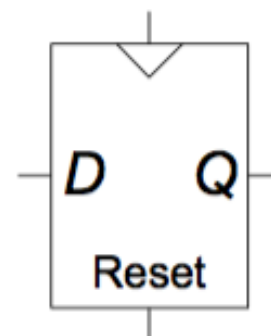
- **Inputs:** CLK , D , EN
 - The enable input (EN) controls when new data (D) is stored
- **Function**
 - $EN = 1$: D passes through to Q on the clock edge
 - $EN = 0$: the flip-flop retains its previous state



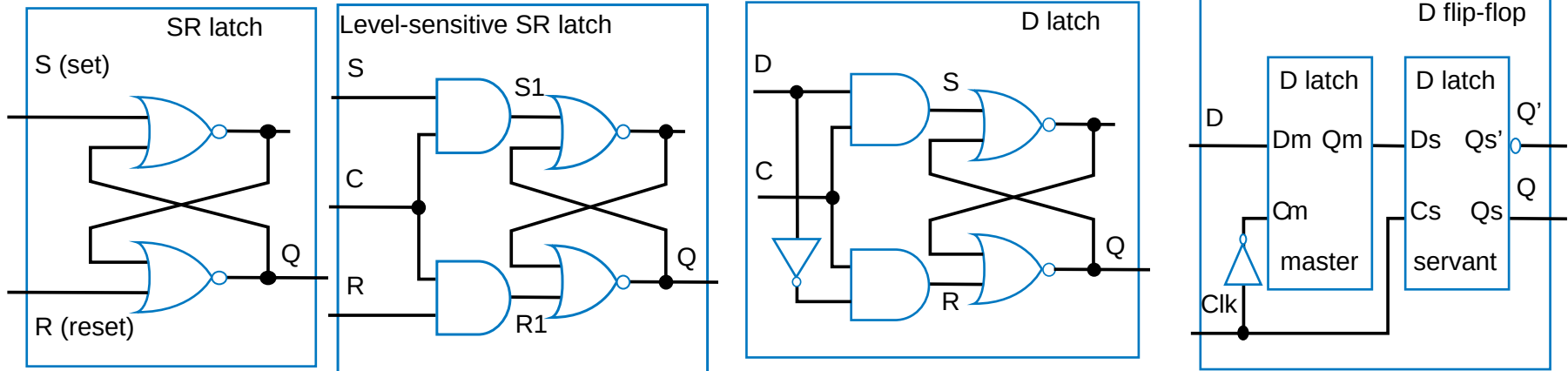
Resettable Flip-Flops

Symbols

- **Inputs:** CLK , D , $Reset$
- **Function:**
 - **$Reset = 1$:** Q is forced to 0
 - **$Reset = 0$:** flip-flop behaves as ordinary D flip-flop
- 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 circuit:
- There are also synch/asynch settable FFs



Bit Storage Overview

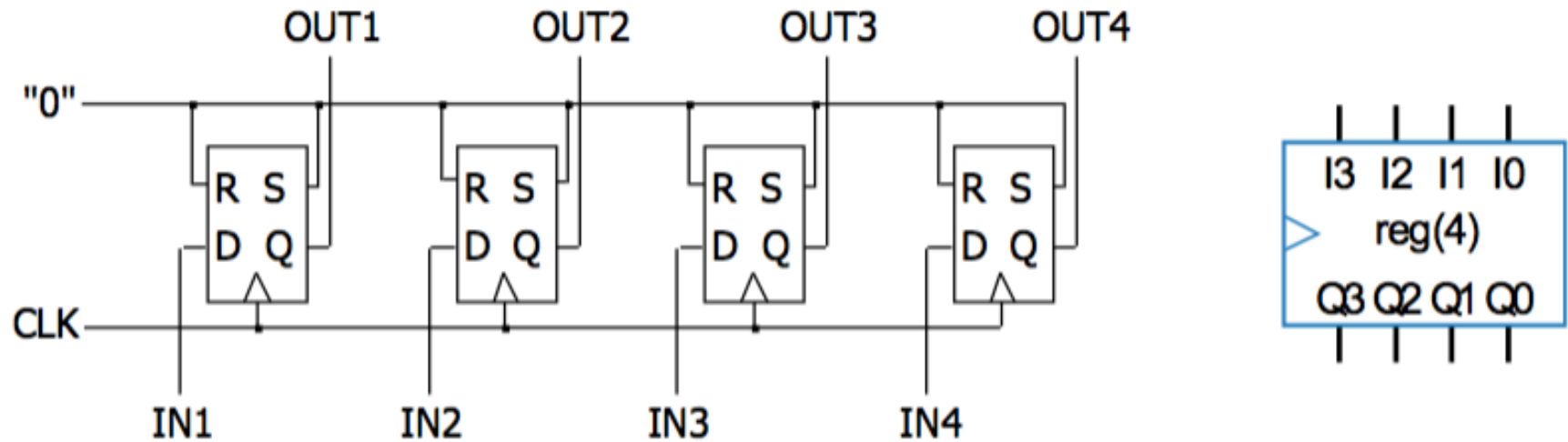


S=1 sets Q to 1, S and R only have effect when C=1.
R=1 resets Q to 0. Problem: SR=11 yield undefined Q.
We can design outside circuit so SR=11 never happens when C=1.
Problem: avoiding SR=11 can be a burden.

SR can't be 11 if D is stable before and while C=1, and will be 11 for only a brief glitch even if D changes while C=1.
*Transition may cross many levels of latches.

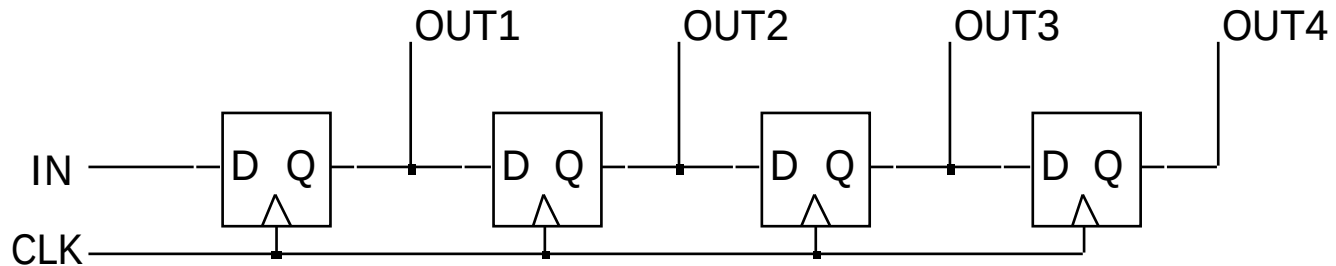
Only loads D value present at rising clock edge, so values can't propagate to other flip-flops during same clock cycle.
*Transition happens between two level of flip-flops.

Building blocks with FFs: Basic Register



Shift register

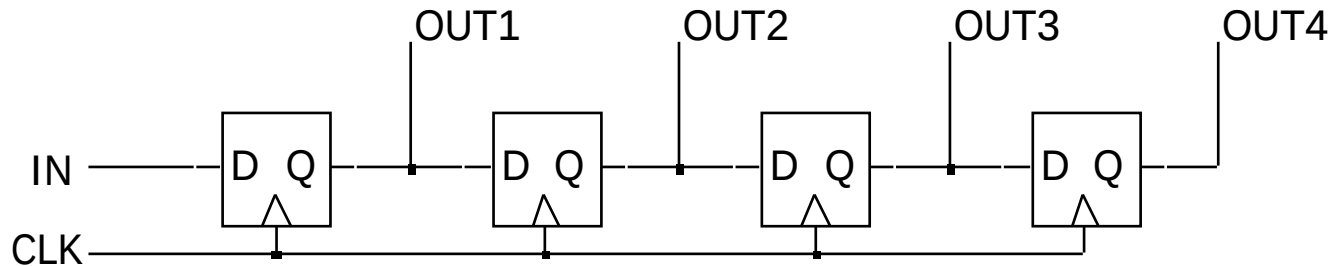
- Holds & shifts samples of input



Time	Input	OUT1	OUT2	OUT3	OUT4
0	1	0	0	0	0
1	0				
2	1				
3	1				
4	0				
5	0				
6	0				

Shift register

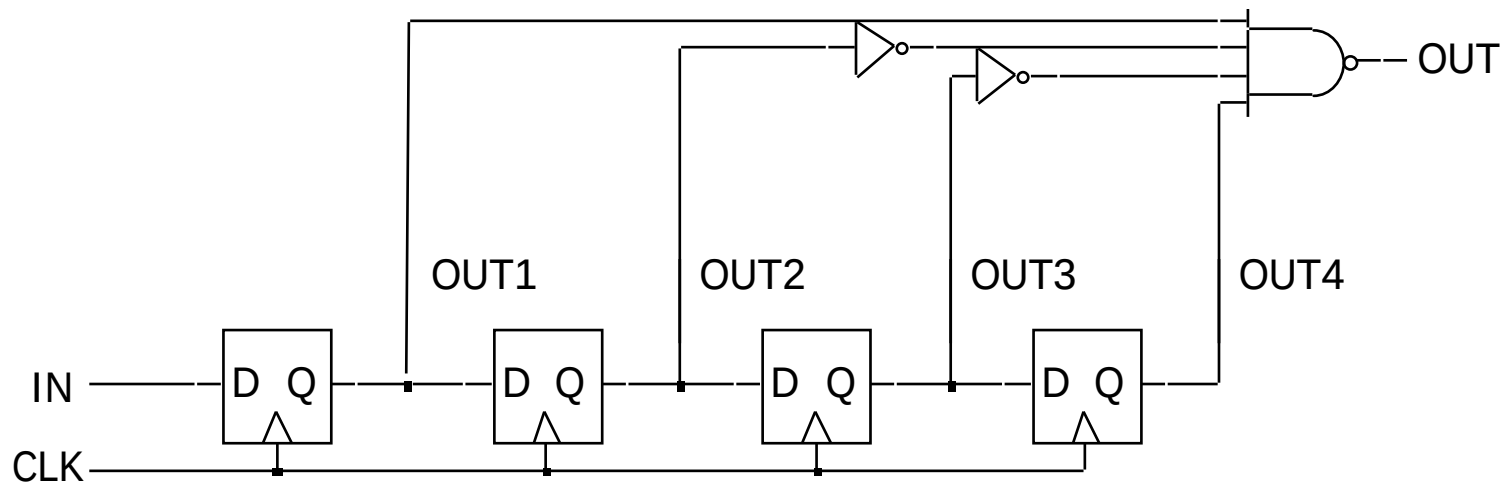
- Holds & shifts samples of input



Time	Input	OUT1	OUT2	OUT3	OUT4
0	1	0	0	0	0
1	0	1	0	0	0
2	1	0	1	0	0
3	1	1	0	1	0
4	0	1	1	0	1
5	0	0	1	1	0
6	0	0	0	1	1

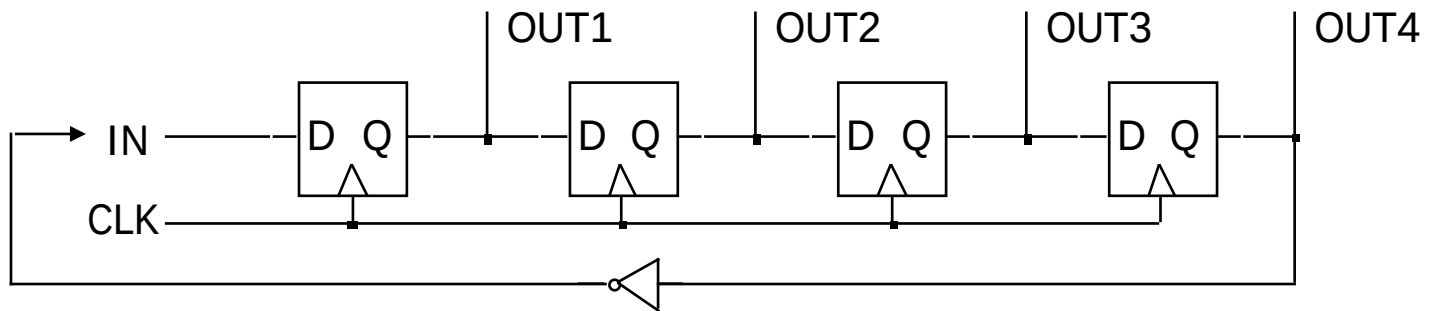
Pattern Recognizer

- Combinational function of input samples



Counters

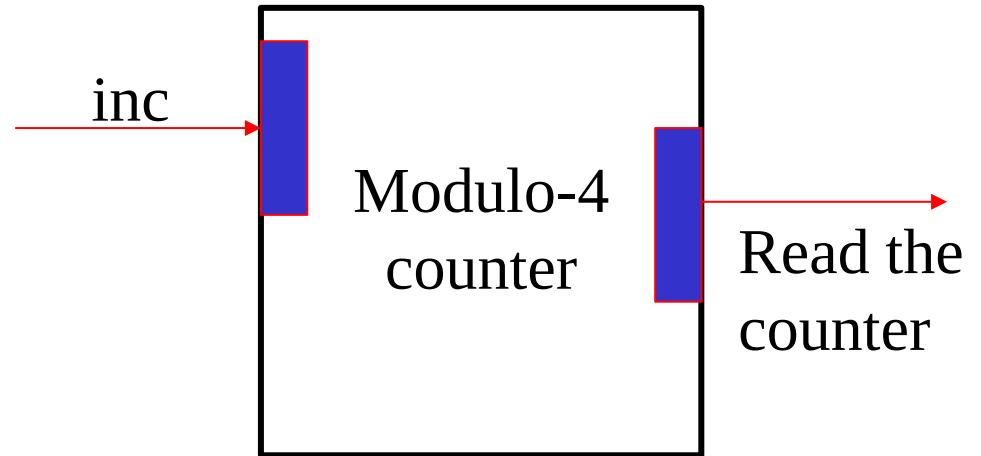
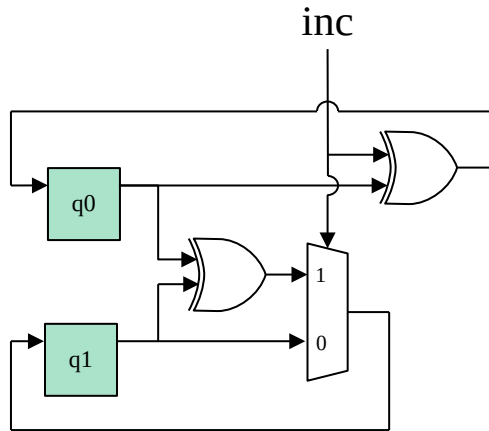
- Sequences through a fixed set of patterns



Describing Sequential Ckts

- State diagrams and next-state tables nor are suitable for describing very large digital designs
 - large circuits must be described in a modular fashion -- as a collection of cooperating FSMs
- BSV is a modern programming language to describe cooperating FSMs
 - We will give various examples of FSMs in BSV

Modulo-4 counter circuit



$$q0^{t+1} = \sim inc \cdot q0^t + inc \cdot \sim q0^t$$

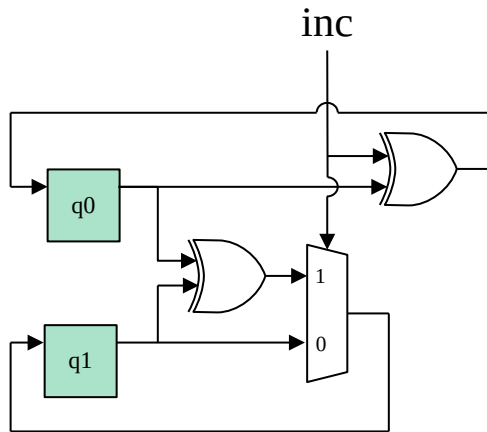
$$q1^{t+1} = \sim inc \cdot q1^t + inc \cdot \sim q1^t \cdot q0^t + inc \cdot q1^t \cdot \sim q0^t$$

“Optimized” logic

$$q0^{t+1} = inc \oplus q0^t$$

$$q1^{t+1} = (inc == 1) ? q0^t \oplus q1^t : q1^t$$

Modulo-4 counter circuit



$$q0^{t+1} = \sim inc \cdot q0^t + inc \cdot \sim q0^t$$

$$q1^{t+1} = \sim inc \cdot q1^t + inc \cdot \sim q1^t \cdot q0^t + inc \cdot q1^t \cdot \sim q0^t$$

“Optimized” logic

$$q0^{t+1} = inc \oplus q0^t$$

$$q1^{t+1} = (inc == 1) ? q0^t \oplus q1^t : q1^t$$

PS\input	inc=0	inc=1
00	00	01
01	01	10
10	10	11
11	11	00

PS: $q1^t q0^t$, NS: $q1^{t+1} q0^{t+1}$

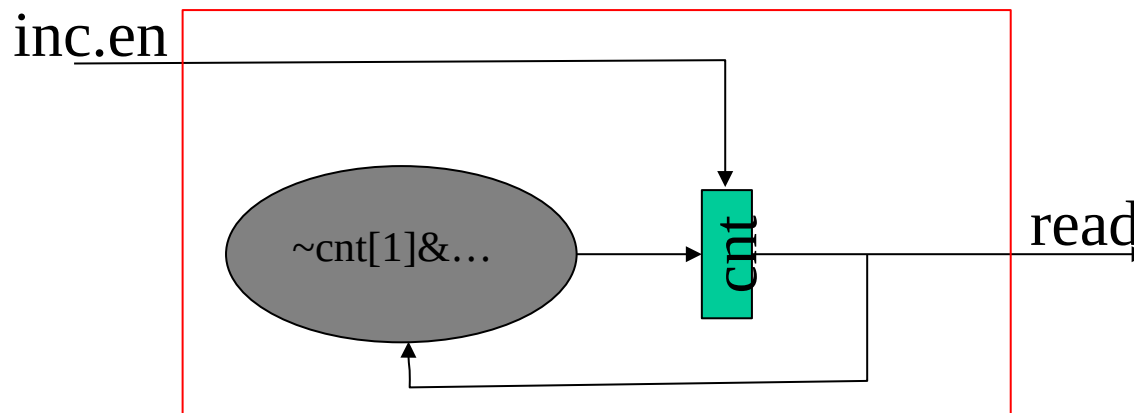
modulo4 counter in BSV

```
module moduloCounter(Counter);  
  Reg#(Bit#(2)) cnt <- mkReg(0);  
  method Action inc;  
    cnt <= {!cnt[1]&cnt[0] | cnt[1]&!cnt[0],  
           !cnt[0]};  
  endmethod  
  method Bit#(2) read;  
    return cnt;  
  endmethod  
endmodule
```

State
specification

Initial
value

An action
to specify
how the
value of
the cnt is
to be set



Interface

- Modulo counter has the following interface, i.e., type

```
interface Counter;  
    method Action inc;  
    method Bit#(2) read;  
endinterface
```

- An interface can have many different implementations
 - For example, the numbers may be represented as Gray code

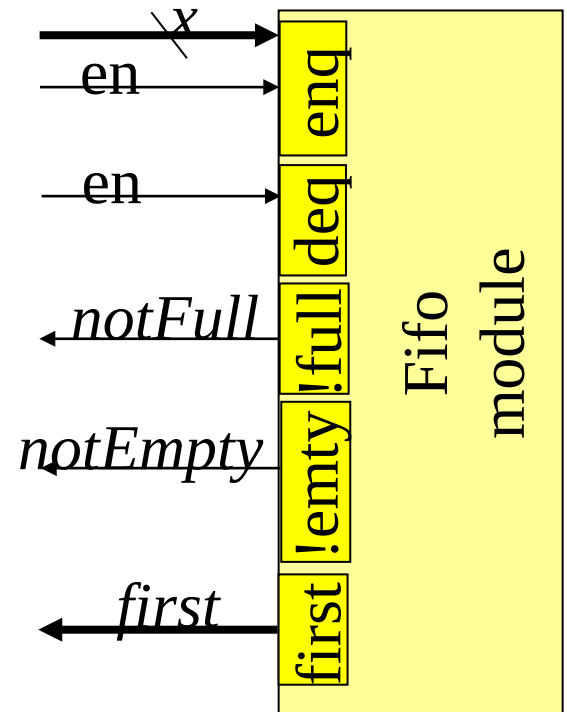
Modules

- A module in BSV is like a class definition in Java or C++
 - It has internal state
 - The internal state can only be read and manipulated by the (interface) methods
 - An action specifies which state elements are to be modified
 - Actions are atomic -- either all the specified state elements are modified or none of them are modified (no partially modified state is visible)

FIFO Interface

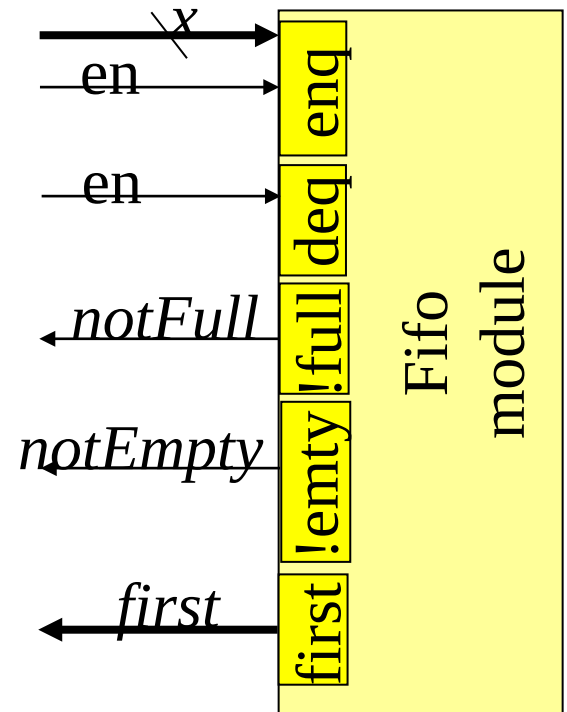
```
interface Fifo#(numeric type size,  
    type t);  
    method Bool notFull;  
    method Bool notEmpty;  
    method Action enq(t x);  
    method Action deq;  
    method t first;  
endinterface
```

- enq should be called only if notFull returns True;
- deq and first should be called only if notEmpty returns True

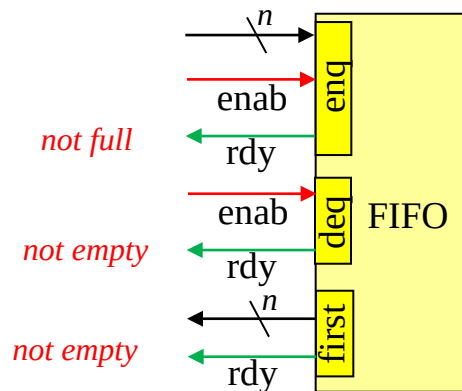


One-Element FIFO Implementation

```
module mkCFFifo (Fifo#(1, t));  
  Reg#(t)    d  <- mkRegU;  
  Reg#(Bool) v  <- mkReg(False);  
  method Bool notFull;  
    return !v;  
  endmethod  
  method Bool notEmpty;  
    return v;  
  endmethod  
  method Action enq(t x);  
    v <= True; d <= x;  
  endmethod  
  method Action deq;  
    v <= False;  
  endmethod  
  method t first;  
    return d;  
  endmethod  
endmodule
```



FIFO Module: methods with guarded interfaces



```
interface Fifo#(numeric type size,  
                type t);  
    method Action enq(t x);  
    method Action deq;  
    method t first;  
endinterface
```

- Every method has a *guard* (rdy wire); the value returned by a value method is meaningful only if its guard is true
- Every action method has an *enable signal* (en wire); an action method is invoked (en is set to true) only if the guard is true
- Guards make it possible to transfer the responsibility of the correct use of a method from the user to the compiler
- Guards are extraordinarily convenient for programming and also enhance modularity of the code

One-Element FIFO Implementation with guards

```
module mkCFFifo (Fifo#(1, t));  
  Reg#(t)      d  <- mkRegU;  
  Reg#(Bool)   v  <- mkReg(False);  
  method Action enq(t x) if (!v); not full  
    v <= True; d <= x;  
  endmethod  
  method Action deq if (v);  
    v <= False;  
  endmethod  
  method t first if (v);  
    return d;  
  endmethod  
endmodule
```

