

*ALU*

ENGR 210 / CSCI B441  
“Digital Design”

# Flip-Flops

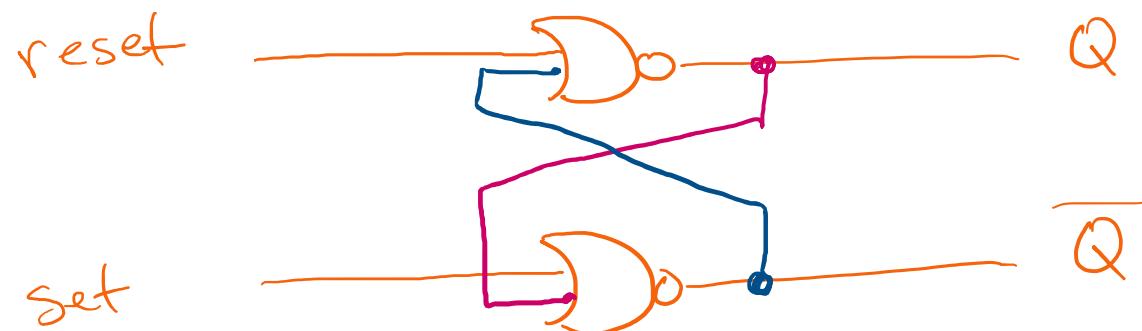
Andrew Lukefahr

# Announcements

⚠ Password 2-3-5

- P2 due Friday
- P3 will add “demo” requirement. Will need hardware (below).
- Labs/Office Hours ✓
  - Will remain “Virtual” for now
  - You will have access to Luddy 4111
  - You can work from there or from home
- Hardware
  - Still working on that, hopefully pickup on Friday.

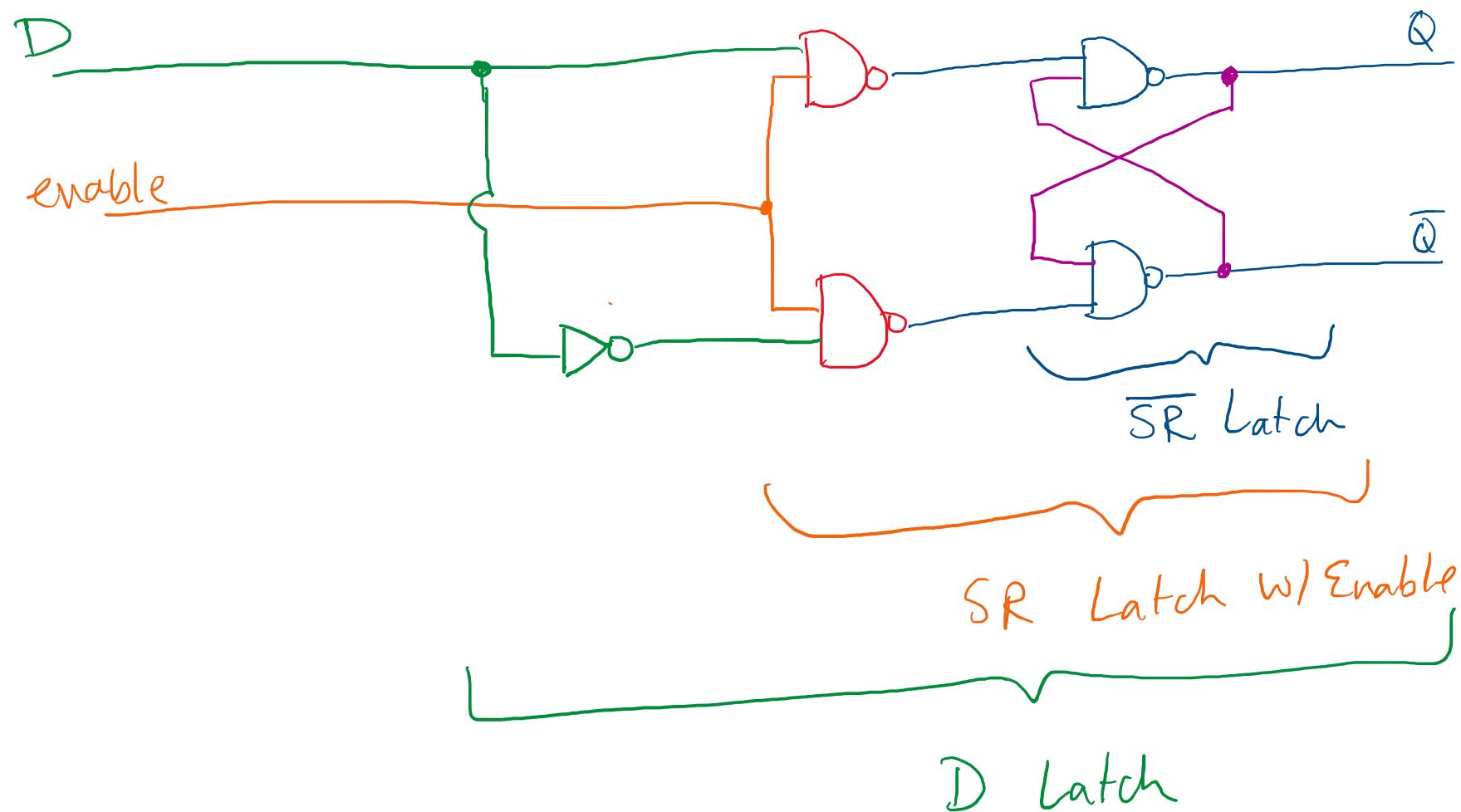
# Last Time: SR Latch



<u>Set</u>	<u>reset</u>	$Q$	$\bar{Q}$
0	1	0	1
0	0	0	1
1	0	1	0
0	0	1	0

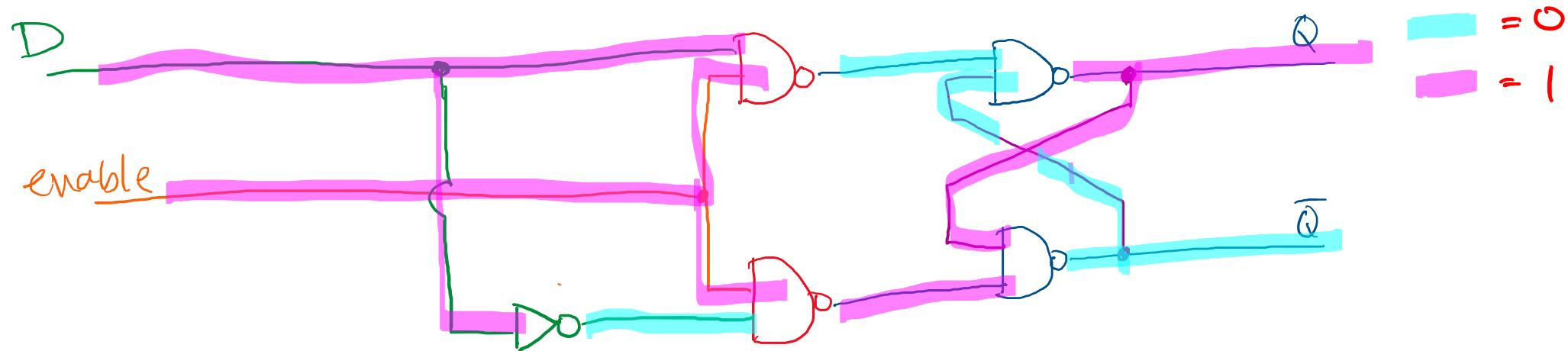
Same inputs, different output!  
⇒ Internal state!

# D-Latch



"Q follows D  
~~when~~ when  
 $\text{en} = 1$ ,  
otherwise  
doesn't  
change"

# D-Latch

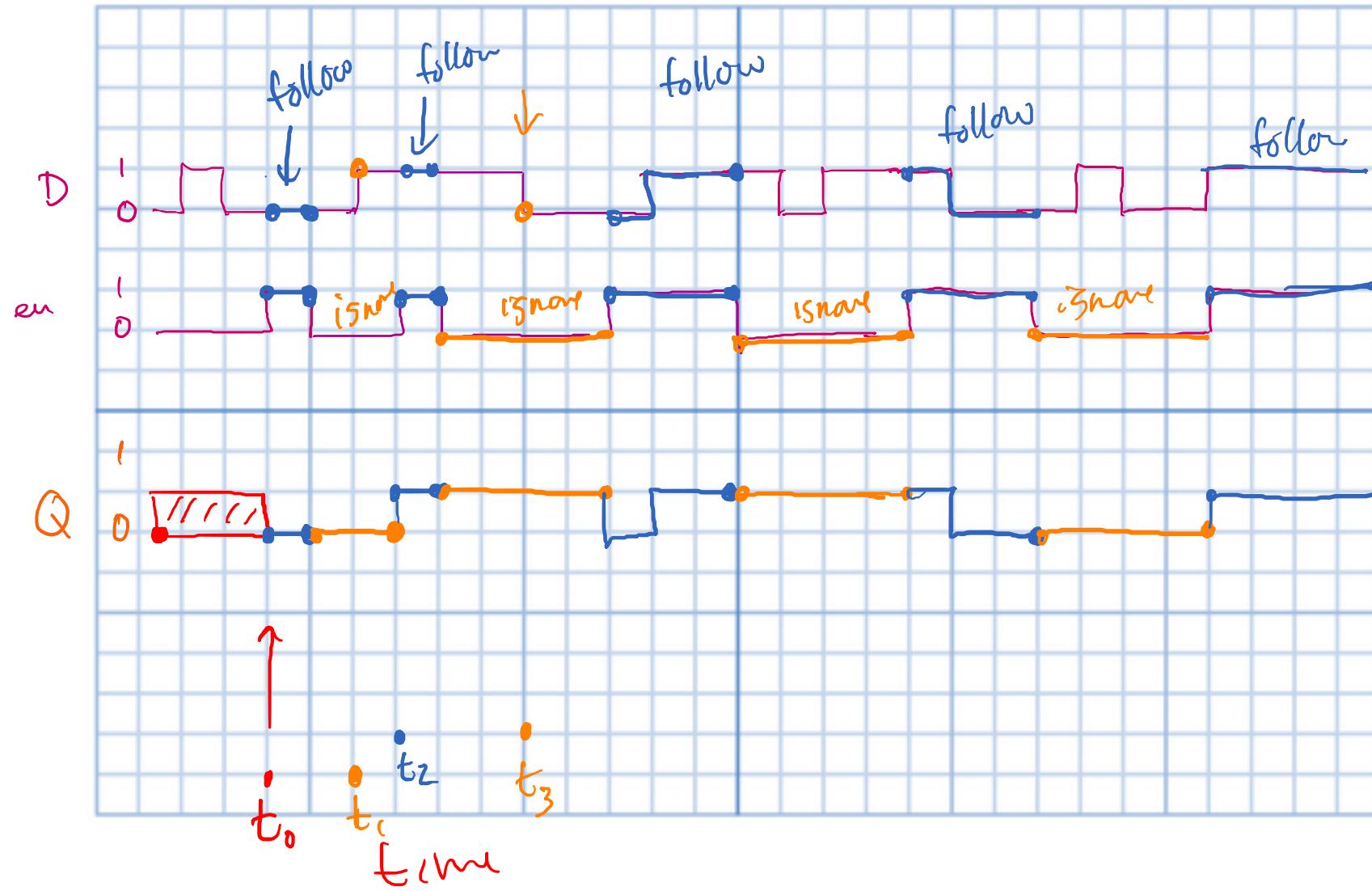


<u>D</u>	<u>enable</u>
0	0
0	1
1	0
1	1

<u>Q</u>	<u><math>\bar{Q}</math></u>
Q	$\bar{Q}$
0	1
Q	$\bar{Q}$
1	0

"might be on  
the test!"

# Inputs to D Latches



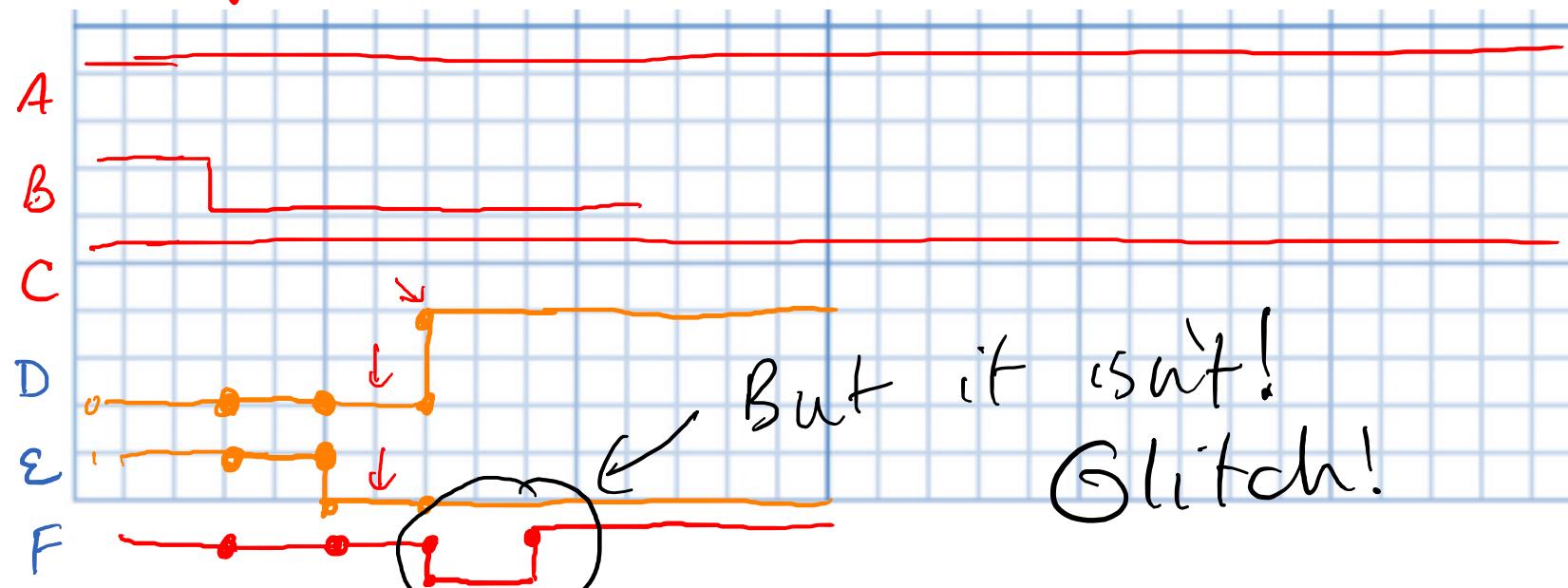
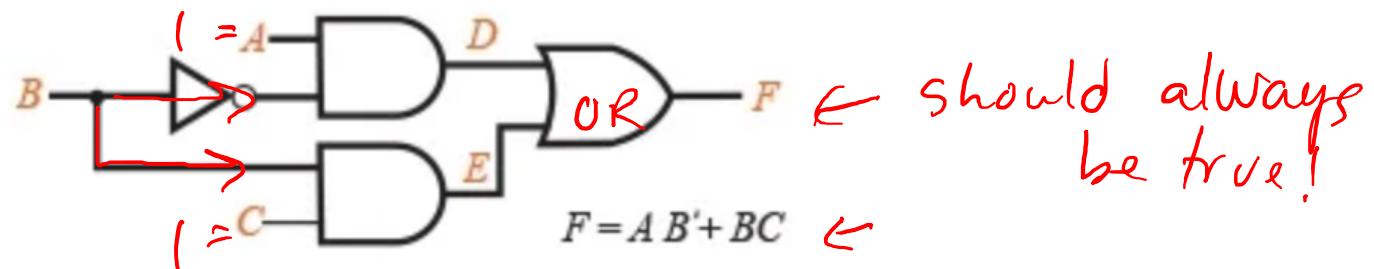
# Glitches

→ unintended, short, errors in  
boolean logic

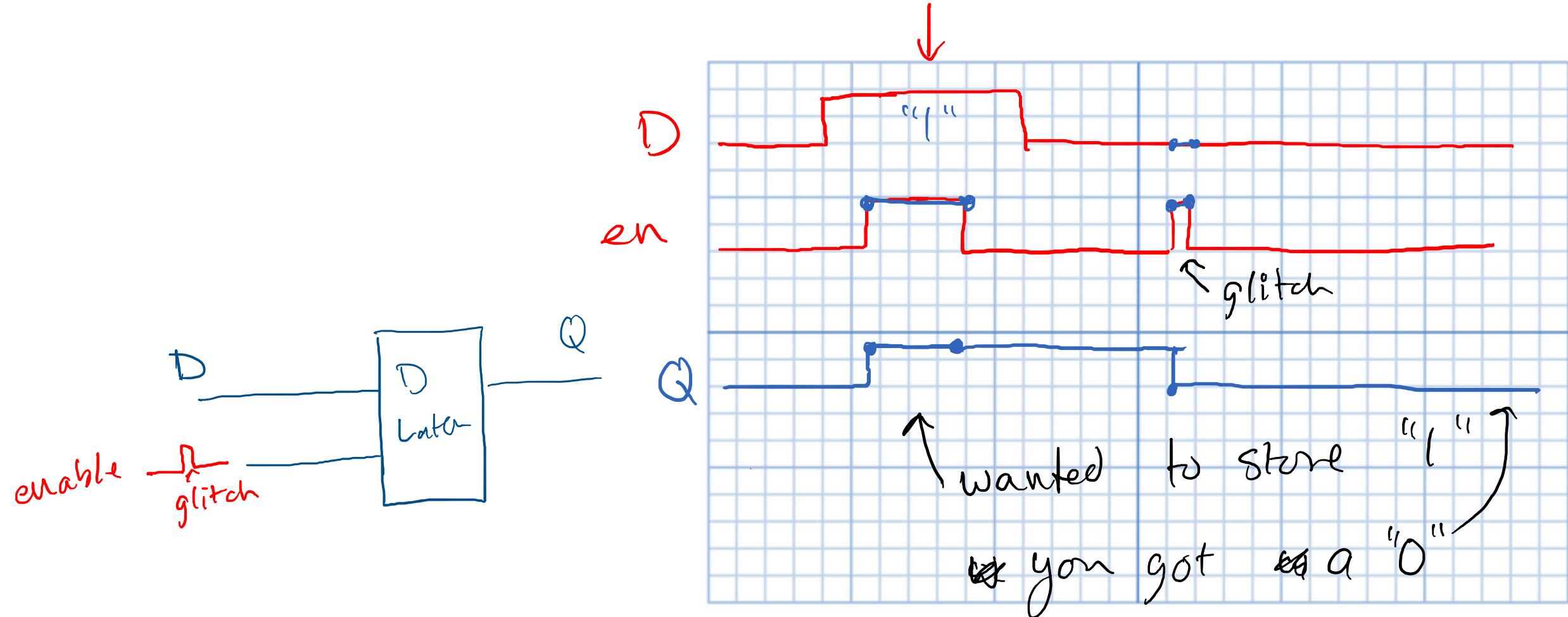
→ caused by gate delays

- Assume 10ps / gate.
- $A=1$ ,  $C=1$ ,  $B$  falls
- What is  $F$ ? ↗

$$A=1 \quad B=1 \quad C=1 \\ \downarrow \qquad \downarrow \qquad \downarrow \\ 10\text{ps}$$



# Glitches on D-Latches

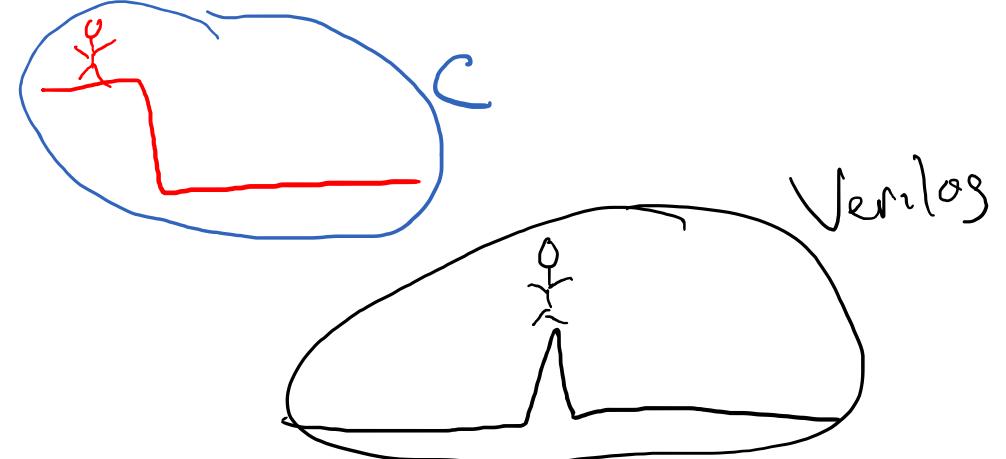


# Inferred Latches

```
wire x, y, z;  
reg foo, bar ;
```

→ always\_comb begin  
    if x [foo = y & z;] //bad:  
    if (x) bar = y | z; // what if ~x?  
end

What if  $x == 0$ ?



↑  
guess  
latch?

# Inferred Latches

```
wire x, y, z;  
reg foo, bar ;  
  
always_comb begin  
    if (x) foo = y & z; //bad:  
    if (x) bar = y | z; // what if ~x?  
end
```

# Defaults

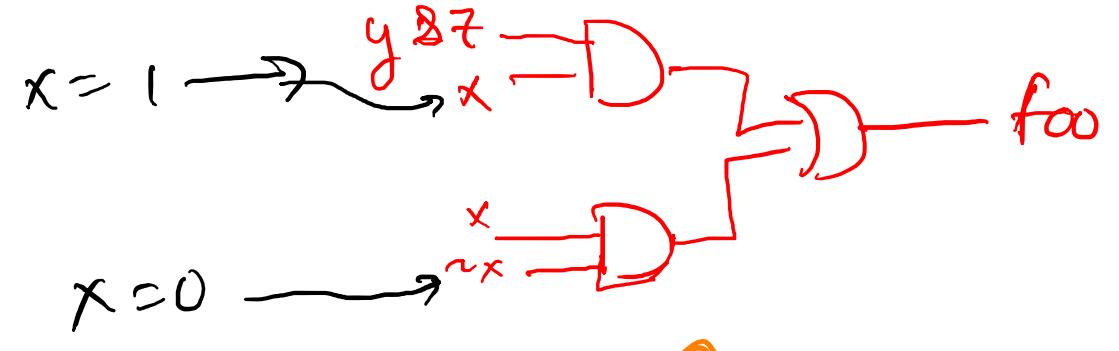
```
wire x, y, z;  
reg foo, bar ;
```

```
always_comb begin  
    foo = x; bar = x; //good: defaults  
    if (x) foo = y & z; //
```

```
    if (x) bar = y | z; //
```

```
end
```

x foo

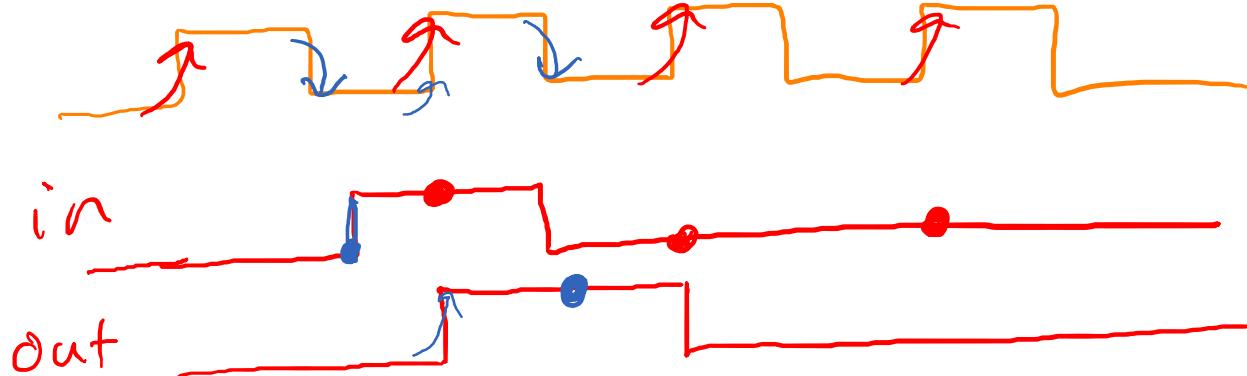


What if  $x == 0$ ?  $foo = bar = x!$

Always specify defaults for always\_comb!

Verilog

module  
(@posedge)



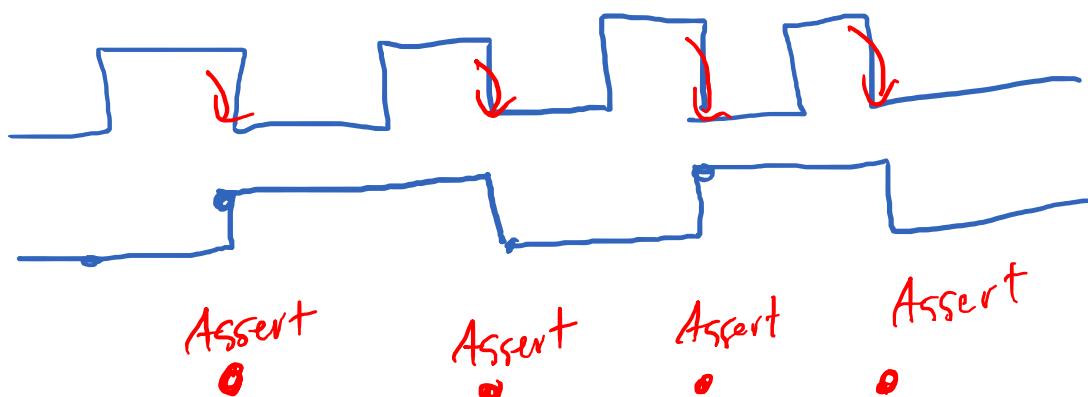
Always specify defaults for  
**always\_comb**!

Test bench

@ negedge

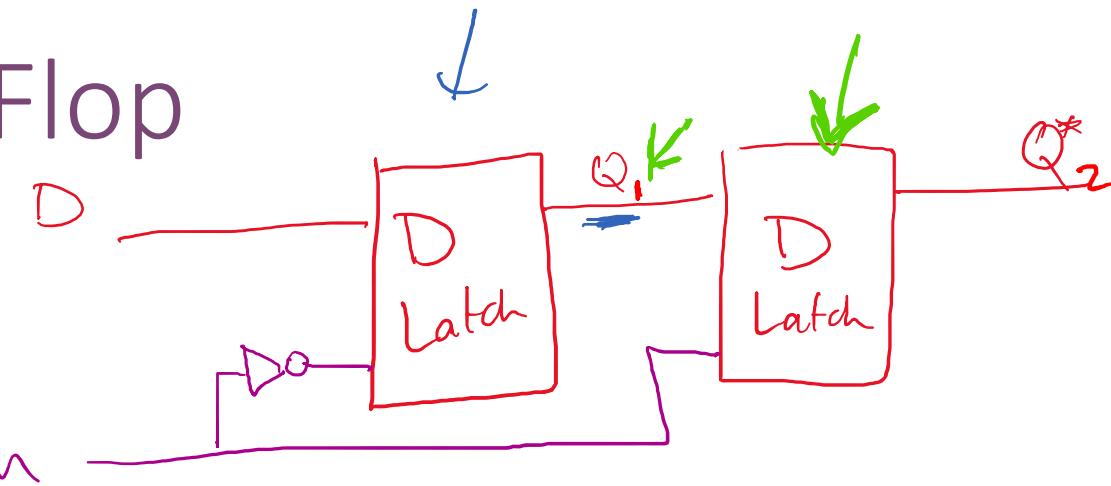
inputs

assert  
outputs



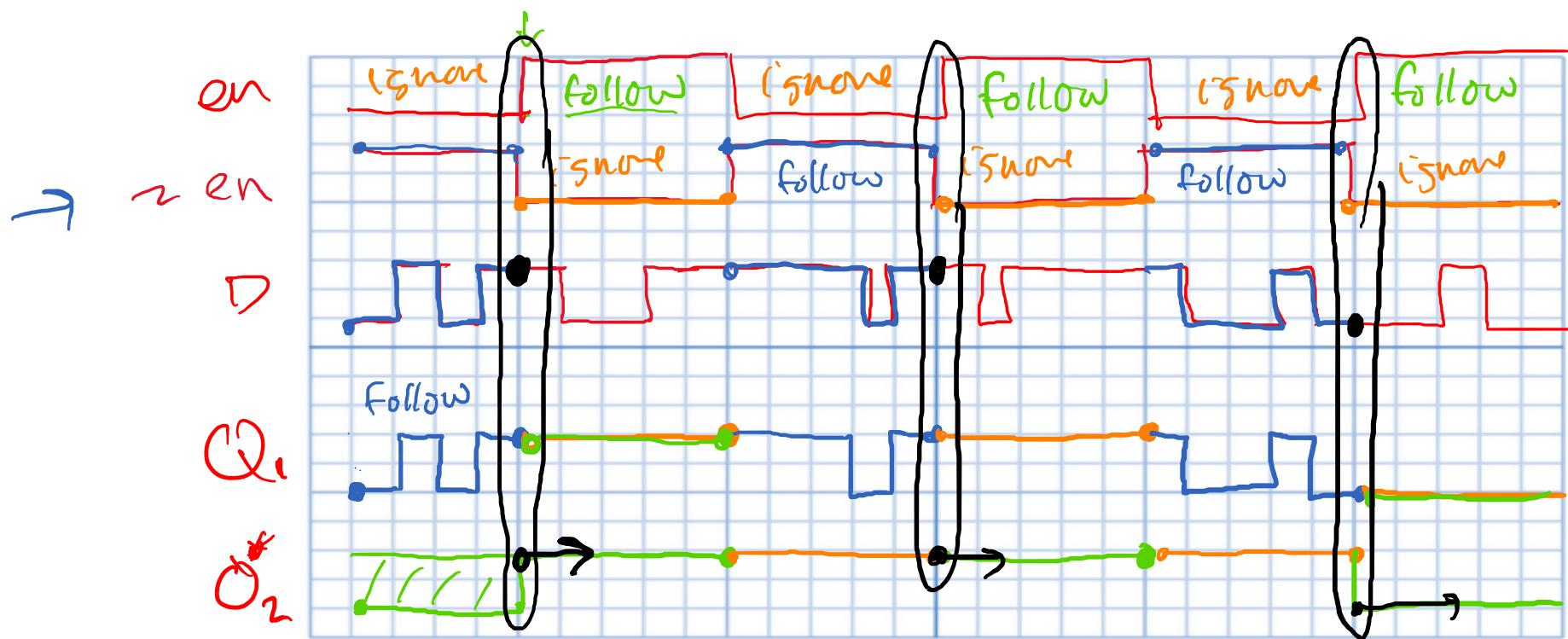
Always specify  
defaults for  
always\_comb!

# D Flip-Flop



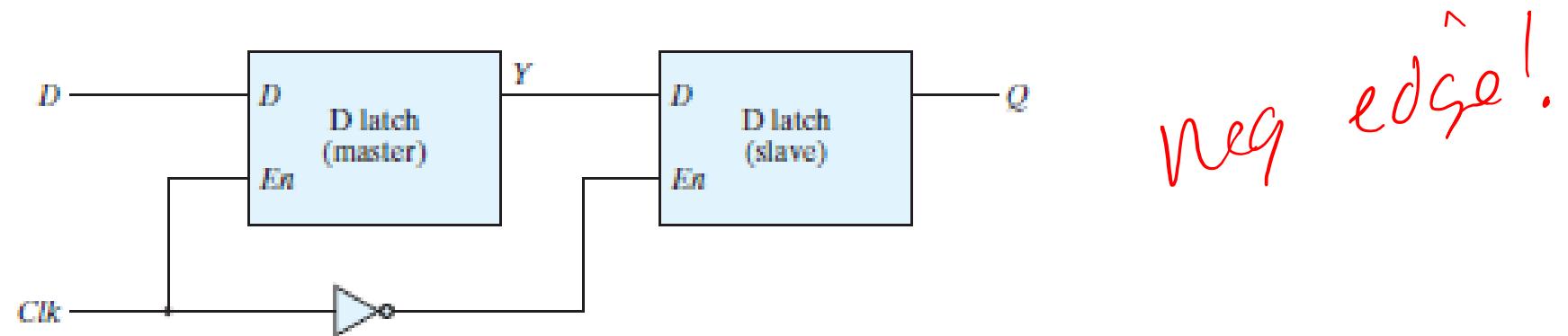
\* no gate delays

D latch:  
 $Q$  follows  $D$  when  
 $en = 1$



## Master-slave D flip-flop

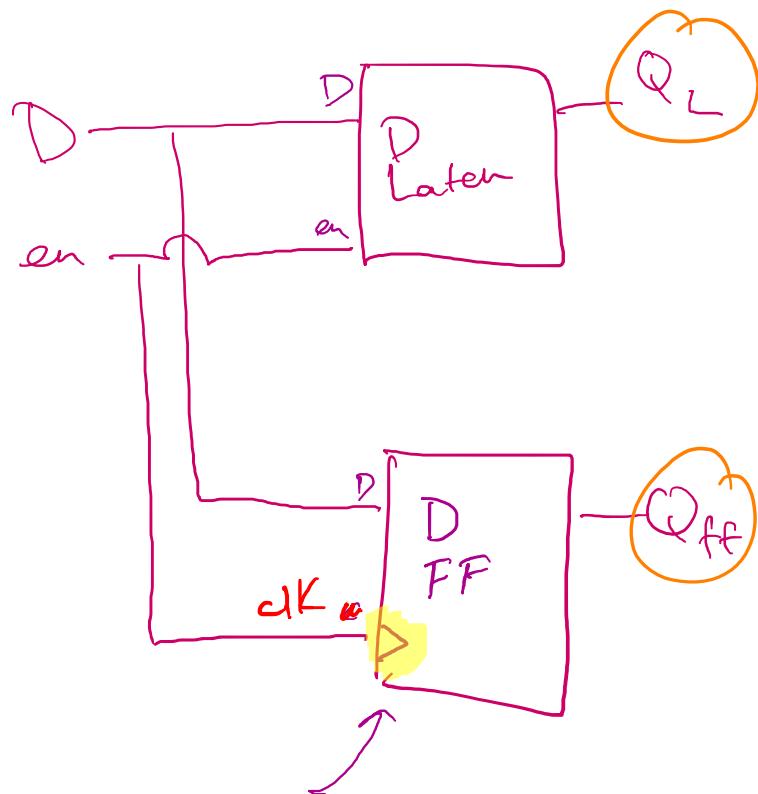
One way to construct a flip-flop is to employ two latches in a special configuration that isolates the output of the flip-flop and prevents it from being affected while the input to the flip-flop is changing.



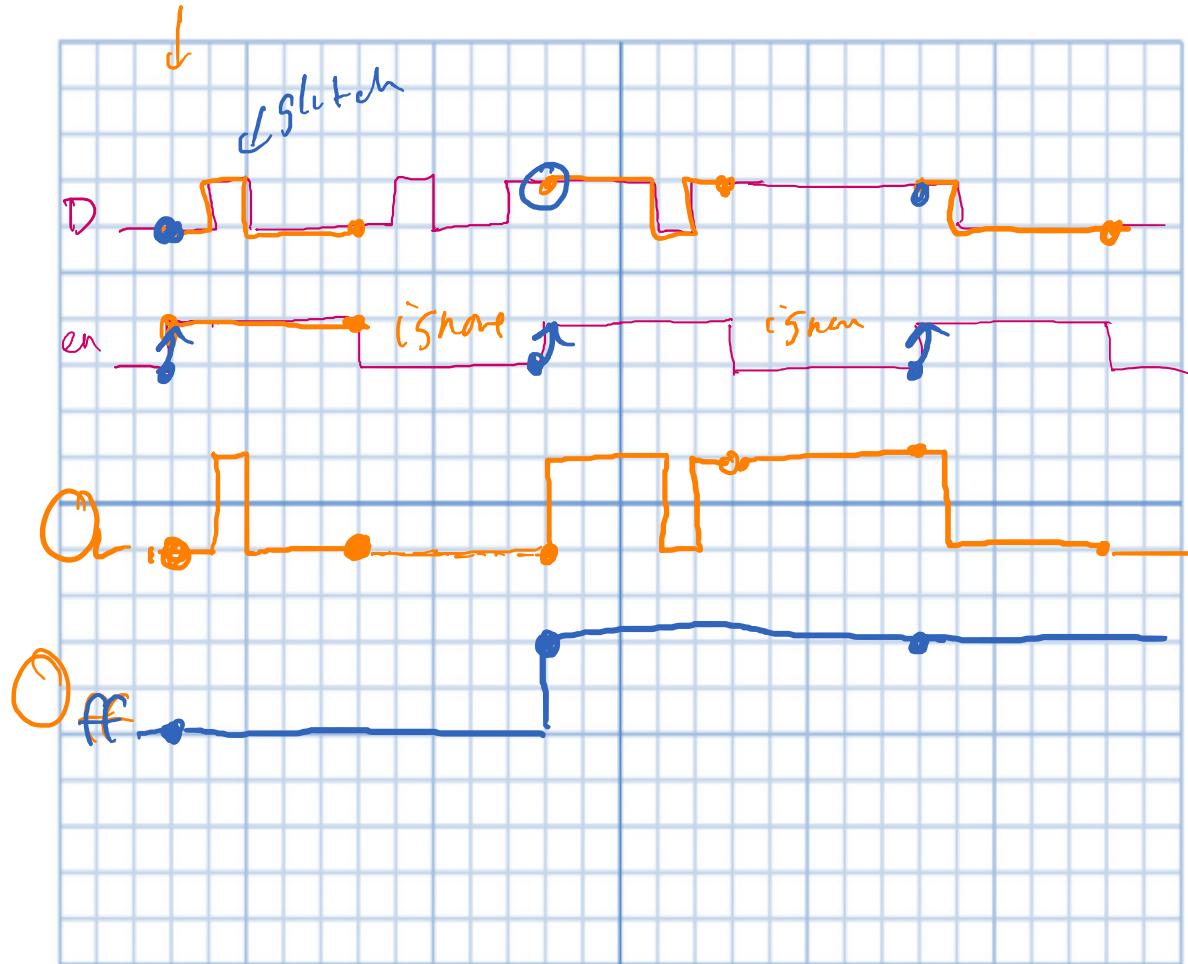
When the *Clk* is at the logic-1 level, the data from the external *D* input are transferred to the master. The slave is disabled and any change in the input changes the master output at *Y*, but cannot affect the slave output.

When the *Clk* returns to 0, the master is disabled and is isolated from the *D* input. At the same time, the slave is enabled and the value of *Y* is transferred to the output of the flip-flop at *Q*. Thus, a change in the output of the flip-flop can be triggered only by and during the ***transition of the clock from 1 to 0***.

# D Flip-Flop vs. D Latch



the " > " symbol tells you  
it is Flip-Flop



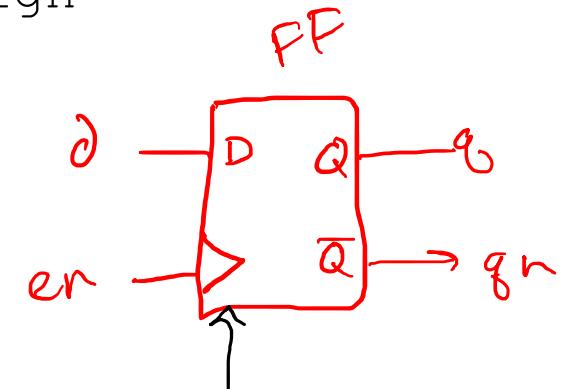
Follows when  
 $en = 1$   
otherwise is none

# D Flip-Flop in Verilog

```
module d_ff (
    input d,           //data
    input en,          //enable
    output reg q      //reg-isters hold state
);
    always_ff@posedge en begin
        q <= d; //non-blocking assign
        qn <= ~d; //optional
    end
endmodule
```

Diagram illustrating the timing behavior of the enable signal (en) for the D flip-flop:

- The enable signal (en) is shown as a waveform with two transitions.
- A red arrow points to the first rising edge labeled "posedge".
- A blue arrow points to the falling edge labeled "negedge".
- Annotations indicate that the register holds its state during the low period of the enable signal.



# D Flip-Flop w/ Clock

```
module d_ff (
    input d, //data
    input clk, //clock
    output reg q //reg-isters hold state
);

    always_ff@(posedge clk)
    begin
        q <= d; //non-blocking assign
        q_n <= ~d; //optional
    end

endmodule
```

CLk100MHz



# Blocking vs. NonBlocking Assignments

- Blocking Assignments ( $=$  in Verilog)
  - Execute in the order they are listed in a sequential block;
  - Upon execution, they immediately update the result of the assignment before the next statement can be executed.

```
x = 2  
y = x + 1
```

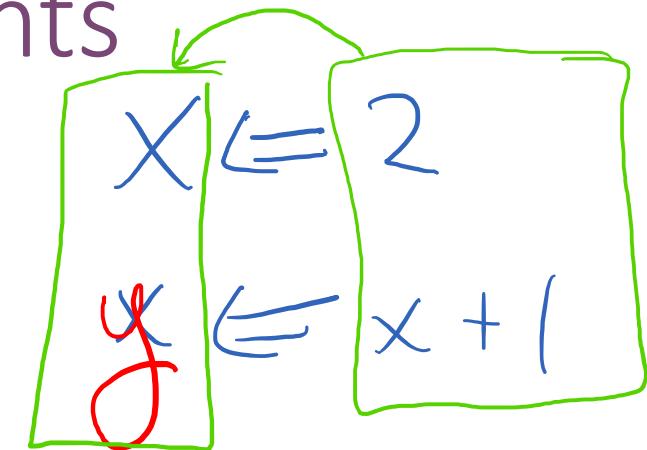
```
x = 2;  
y = x + 1;  
→ x = 2  
      y = 3
```

LHS      RHS  
 $x \Leftarrow 2$

# Blocking vs. NonBlocking Assignments

- Non-blocking assignments ( $\Leftarrow$  in Verilog):

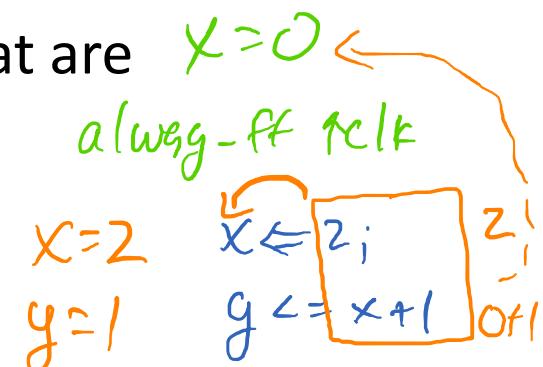
- Execute concurrently



- Evaluate the expression of all right-hand sides of each statement in the list of statements before assigning the left-hand sides.

- Consequently, there is no interaction between the result of any assignment and the evaluation of an expression affecting another assignment.

- Nonblocking procedural assignments be used for all variables that are assigned a value within an edge-sensitive cyclic behavior.



# Blocking vs. NonBlocking

```
always_comb
begin
    LHS
     $\rightarrow x = a + 1;$ 
     $y = x + 1;$ 
     $\rightarrow z = z + 1;$ 
     $\text{bad in}$ 
     $\text{always\_comb}$ 
end
```

start  $x=0, y=0, z=0, a=0$

$$\begin{aligned} a=1 & \quad x=1+1=2 \leftarrow \\ & \quad y=2+1=3 \\ & \quad z=0+1=1 \leftarrow \\ x=2, & z=1 \quad x=2; \\ & y=3; \\ & z=1+1=2 \end{aligned}$$

```
always_ff @ (posedge clk)
```

```
begin
```

```
 $x \leq a + 1;$ 
 $y \leq x + 1;$ 
 $z \leq z + 1;$ 
```

```
end
```

$1+1$	$a=1+1=2$
$0+1$	$z+1=3$
$0+1$	$1+1=2$

start:  $x=0, y=0, z=0, a=0$

$a=1, \text{clk} \uparrow$

$$\begin{aligned} x &= 2 \\ y &= 1 \\ z &= 1 \end{aligned}$$

$\text{clk} \uparrow$

$$\begin{aligned} x &= 2 \\ y &= 3 \\ z &= 2 \end{aligned}$$

# Blocking vs. Non-Blocking Assignments

- ONLY USE BLOCKING ( $=$ ) FOR COMBINATIONAL LOGIC
  - always\_comb
- ONLY USE NON-BLOCKING ( $<=$ ) FOR SEQUENTIAL LOGIC
  - always\_ff
- Disregard what you see/find on the Internet!

**BLOCKING (=) FOR  
always\_comb**

never  
hold state  
No flip flops!  
+ defaults!

**NON-BLOCKING ( $\leq$ ) for  
always\_ff**

← always for  
flip flops  
(always hold state)

# D-FlipFlop w/Clock

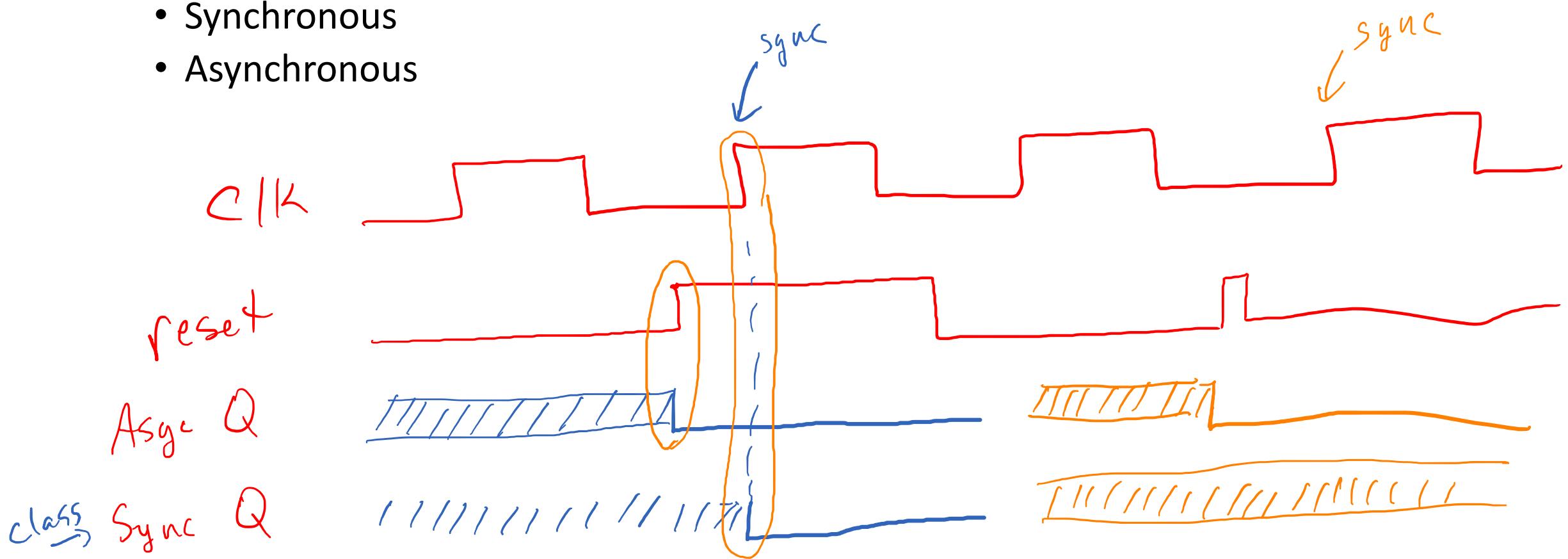
$$q \rightarrow d \rightarrow q_{\text{new}} \rightarrow d_{\text{new}} \rightarrow q_{\text{new}_2}$$

```
module d_ff (
    input d,           //data
    input clk,        //clock
    output logic q   //reg-isters hold state
) ;  
  
    always_ff @ (posedge clk)
    begin
        q <= d; //non-blocking assign
    end
endmodule
```

What is q before posedge clk?

# D-FF's with Reset

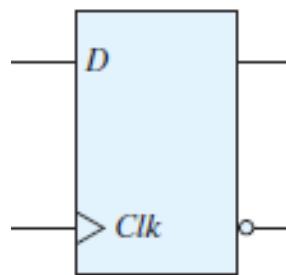
- Two different ways to build in a reset
  - Synchronous
  - Asynchronous



# D-FF's with Reset

- Two different ways to build in a reset
  - Synchronous
  - Asynchronous

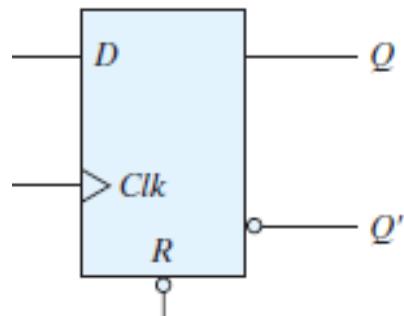
## Verilog models of D flip-flop



Edge triggered D flip-flop:

```
logic Q;  
always_ff @ (posedge clk)  
    Q <= D;
```

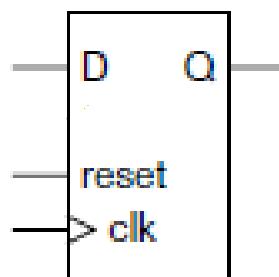
No reset  
ff



Edge triggered, asynchronous reset D flip-flop:

```
logic Q;  
always_ff @ (posedge clk, negedge rst)  
    if (~rst) Q <= 1'b0; //asynch. reset  
    else Q <= D;
```

Not used  
in class



Edge triggered, synchronous reset, clock enable D flip-flop: C

```
logic Q;  
always_ff @ (posedge clk)  
    if (reset) Q <= 1'b0; // synch. reset  
    else Q <= d;
```

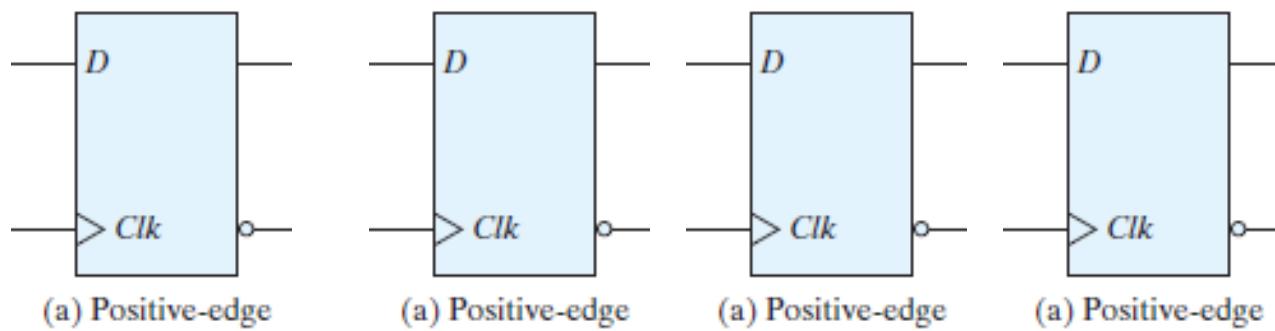
## Blocking Procedural Assignments

---

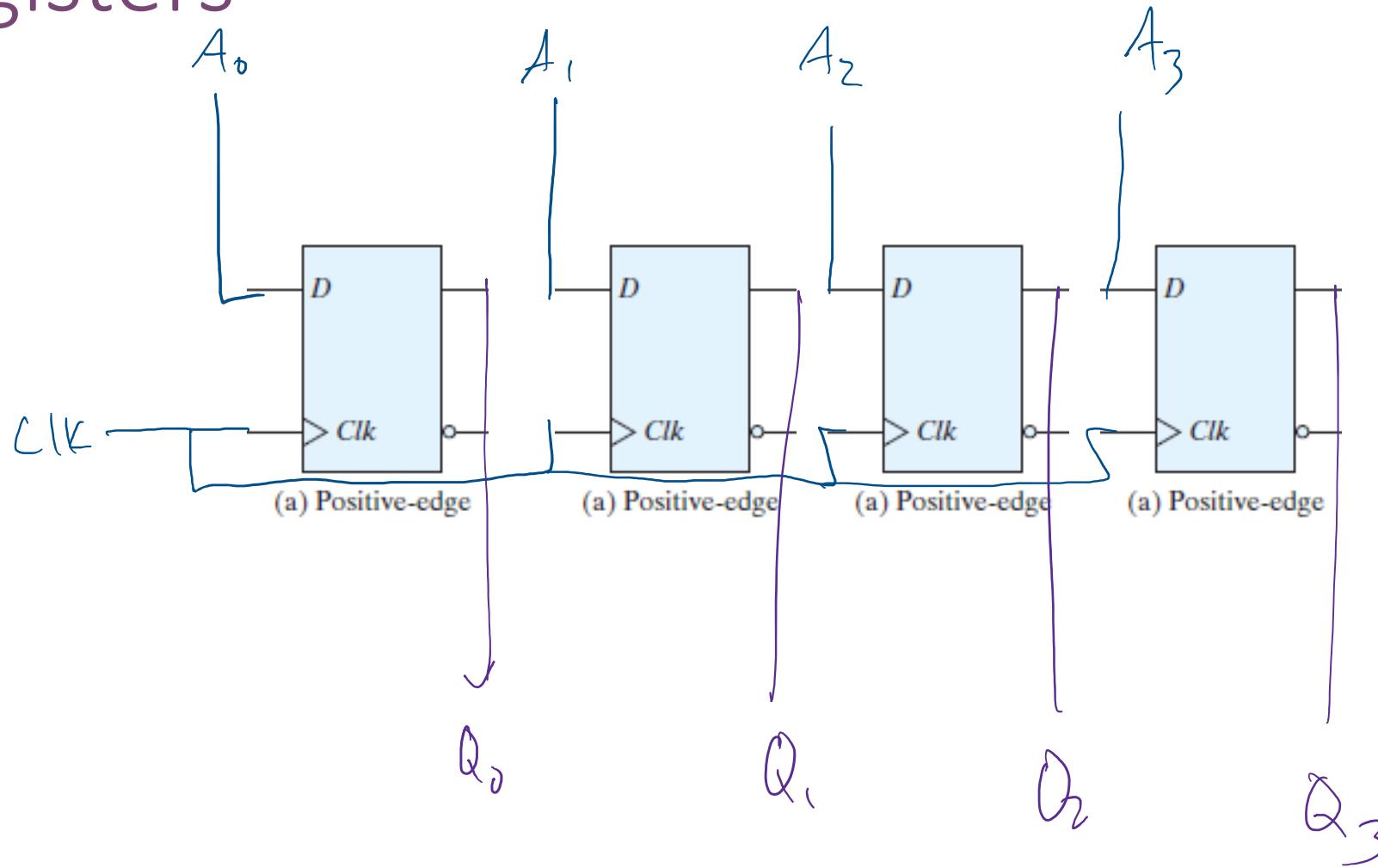
Blocking procedural assignments ( symbol = ):

- Execute sequentially in the order that they are listed in a sequential block;
- When they execute, they have an immediate effect on the contents of memory before the next statement can be executed.
- Used to model level sensitive behavior, such as combinational logic.

# Registers



# Registers



registers in a CPU?

# 4-bit Register in Verilog

```
module d_ff (
    input          d,    //data
    input          clk,   //clock
    output         q     //output register
) ;

    always_ff @(posedge clk)
    begin
        q <= d; //non-blocking assign
    end

endmodule
```

# 4-bit Register in Verilog

```
module d_ff (
    input      [3:0]  d,    //data
    input                  clk,   //clock
    output logic [3:0] q     //output register
) ;

    always_ff @(posedge clk)
    begin
        q <= d; //non-blocking assign
    end

endmodule
```

# What does this module do?

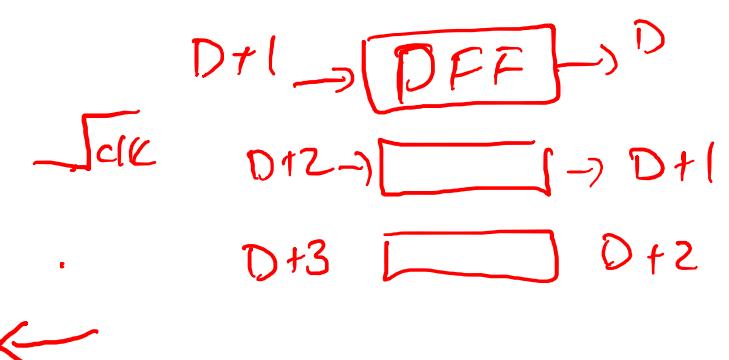
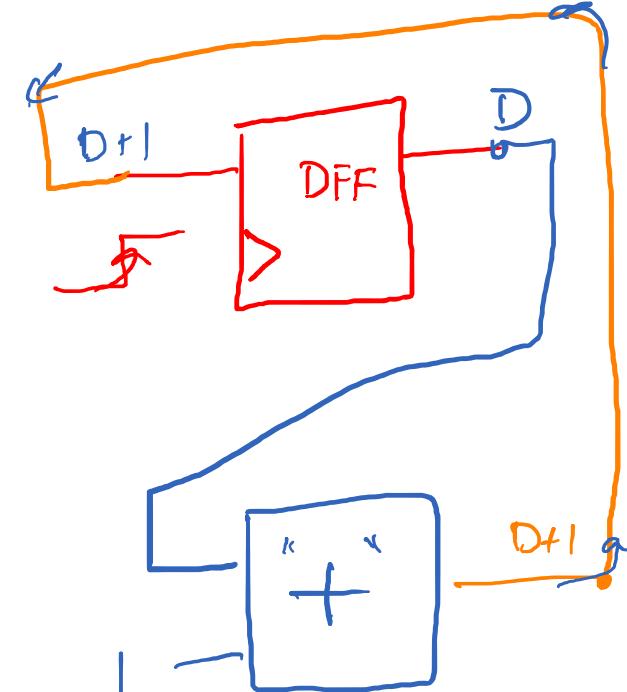
```

module mystery(
    input clk,           //clock
    input rst,           //reset
    output logic out    //output
);
    logic [3:0] D;
    wire [4:0] sum;
    always_ff @ ( posedge clk ) // <- sequential logic
    begin
        if (rst) D <= 4'h0;
        else D <= sum;           //non-blocking
    end
    always_comb // <- combinational logic
    begin
        {out, sum} = {0, D} + 5'h1; //blocking
    end
endmodule

```

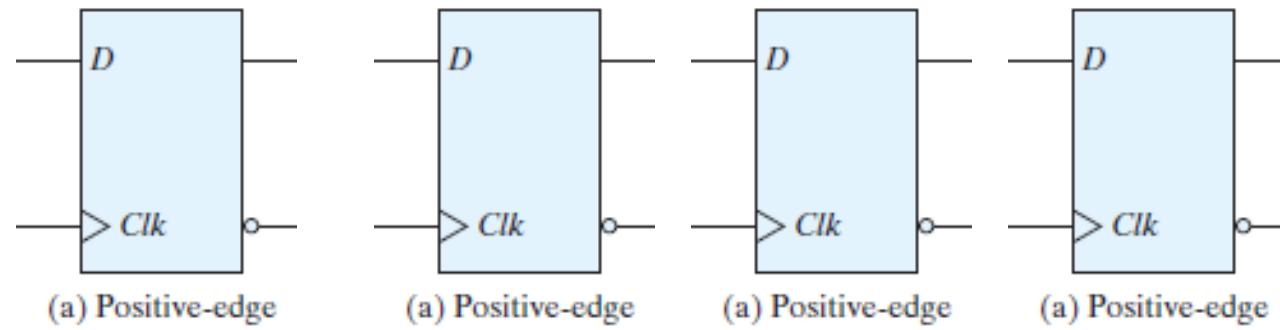
**Annotations:**

- Initial State:** A red arrow points from a state diagram (00000) to the module. The state diagram shows a 5-bit register with bits 0, 1, 2, 3, 4 all set to 0.
- Clock:** A blue waveform labeled "clk" shows a square wave with a period of 4 clock cycles. A red arrow points from this waveform to the "posedge clk" sensitivity in the sequential logic block.
- Reset:** A blue waveform labeled "rst" shows a single pulse at the start. A red arrow points from this waveform to the "rst" input of the sequential logic block.
- Sum Calculation:** A blue box labeled "sum" contains a circled "5'+" symbol, indicating a 5-bit addition operation. A red arrow points from this box to the assignment statement in the combinational logic block.
- Output:** A blue waveform labeled "out" shows a sequence of values: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. A red arrow points from this waveform to the "out" output port.
- Data Types:** Labels "1bit", "4bit", "5bit", and "5bit" are placed near the wires to indicate their widths: 1 bit for the least significant bit of the sum, 4 bits for the rest of the sum, and 5 bits for both the sum and the output.

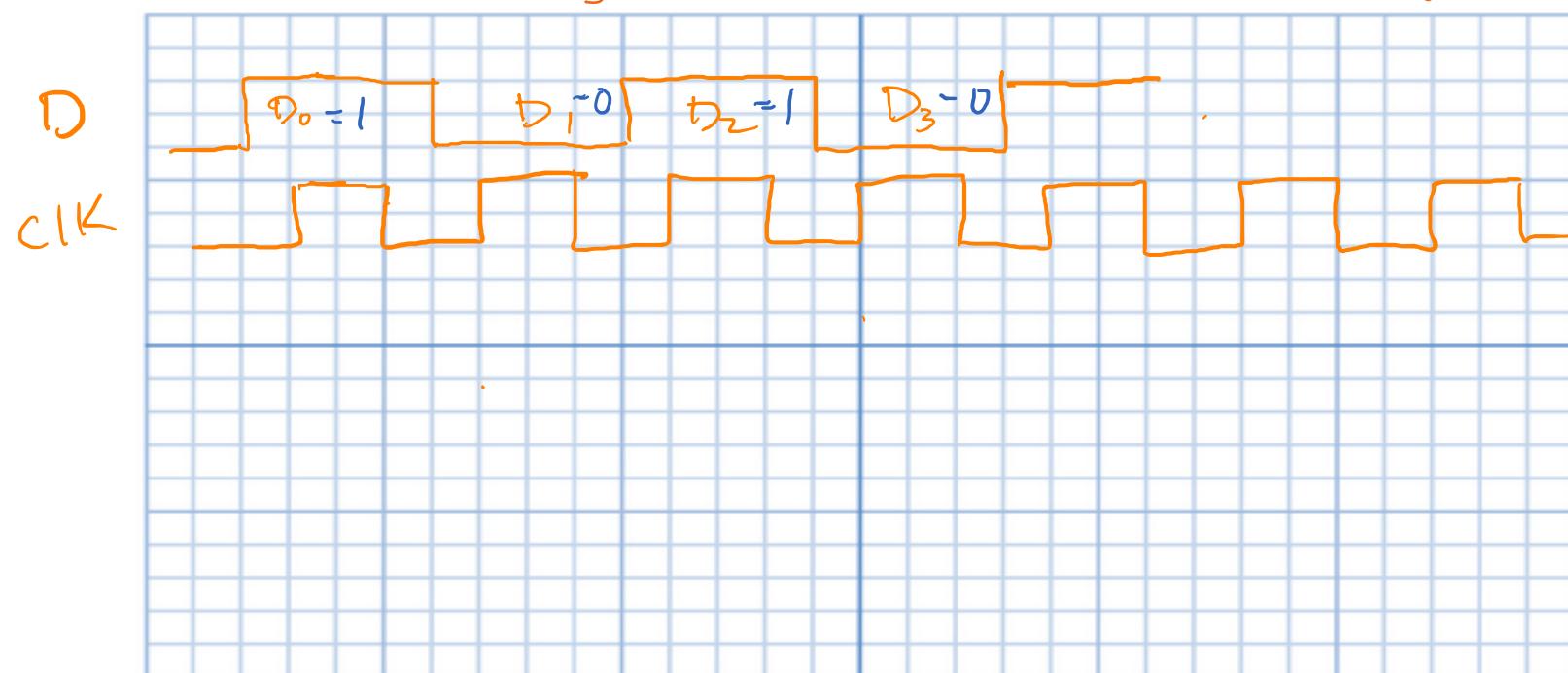
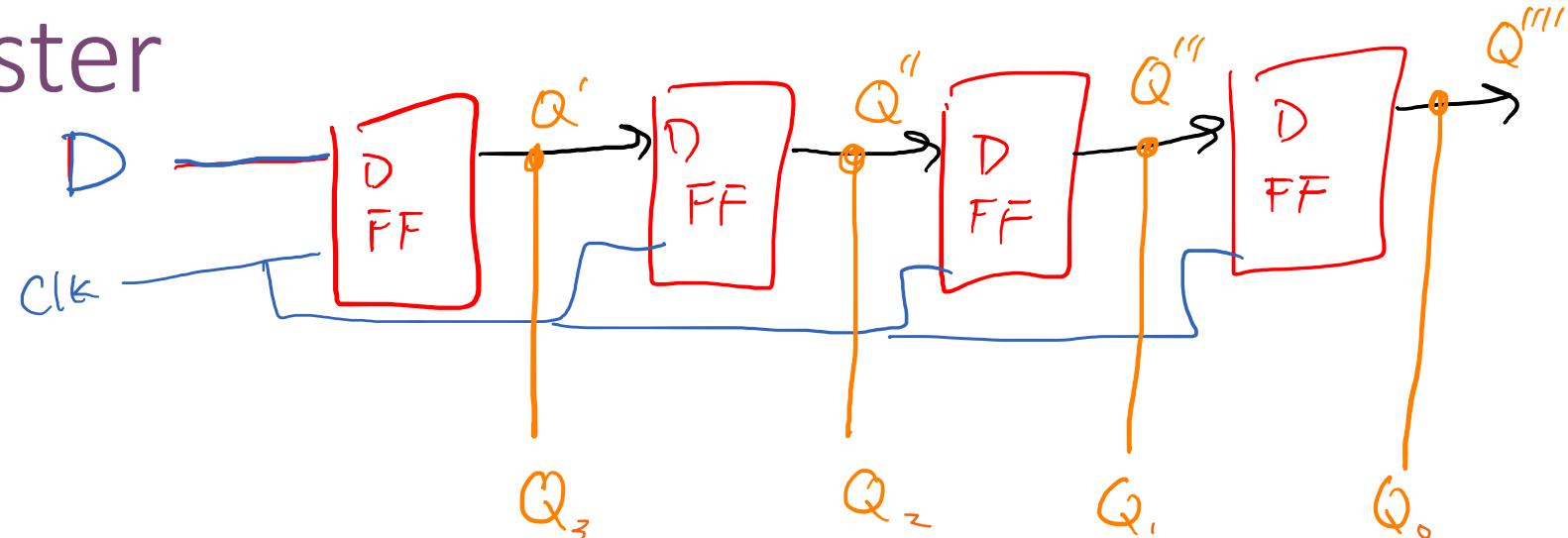


Start here!

# D Flip-Flops as Shift Registers



# Shift Register



# Next Time

- State machines