

ENGR 210 / CSCI B441

Addition / Subtraction II

Andrew Lukefahr

Course Website

fangs-bootcamp.github.io

Write that down!

Announcements

- P2 is out! Due **next** Friday

• C and V bits updates ↪
 ↑ ↪
 carry overflow

$$Add = 1 + 1 = \underline{2}$$

$$\begin{array}{r} Q = 0000\ 0001 \\ S = 0001\ 0001 \\ \hline R = 0000\ 0000 \end{array}$$

$C = 0$ for everything but
Addition & Subtraction

$V =$ defined for Addition & Sub
→ undefined for all other
operations

→ AG will use $V=0$ for
all cases other than
Addition & Subtraction

wire vs logic

- **wire** ↪

- Only used with 'assign' and module outputs
- Boolean combination of inputs
- Can never hold state

- **logic**

- Used with 'always' and module outputs
- Can be Boolean combination of inputs
- Can also hold state

wire w;
assign w = 'h0;

// w = 1 if x == 1
0 if x == 2
1 if x == 3

always_comb blocks with if

```
module decoder (
    input [1:0] sel,
    output logic [3:0] out
);

    always_comb begin
        if (sel == 2'b00) begin
            out = 4'b0001;
        end else if (sel == 2'b01) begin
            out = 4'b0010;
        end else if (sel == 2'b10) begin
            out = 4'b0100;
        end else if (sel == 2'b11) begin
            out = 4'b1000;
        end
    end
endmodule
```

always_comb with case

```
module decoder (
    input [1:0] sel,
    output logic [3:0] out
);
```

→ *always_comb begin*

case(sel)

 2'b00: out=4'b0001;
 2'b01: out=4'b0010;
 2'b10: out=4'b0100;
 2'b11: out=4'b1000;

endcase default: out = 4'b0000;

end

"switch" in C

```
endmodule
```

always_comb with case

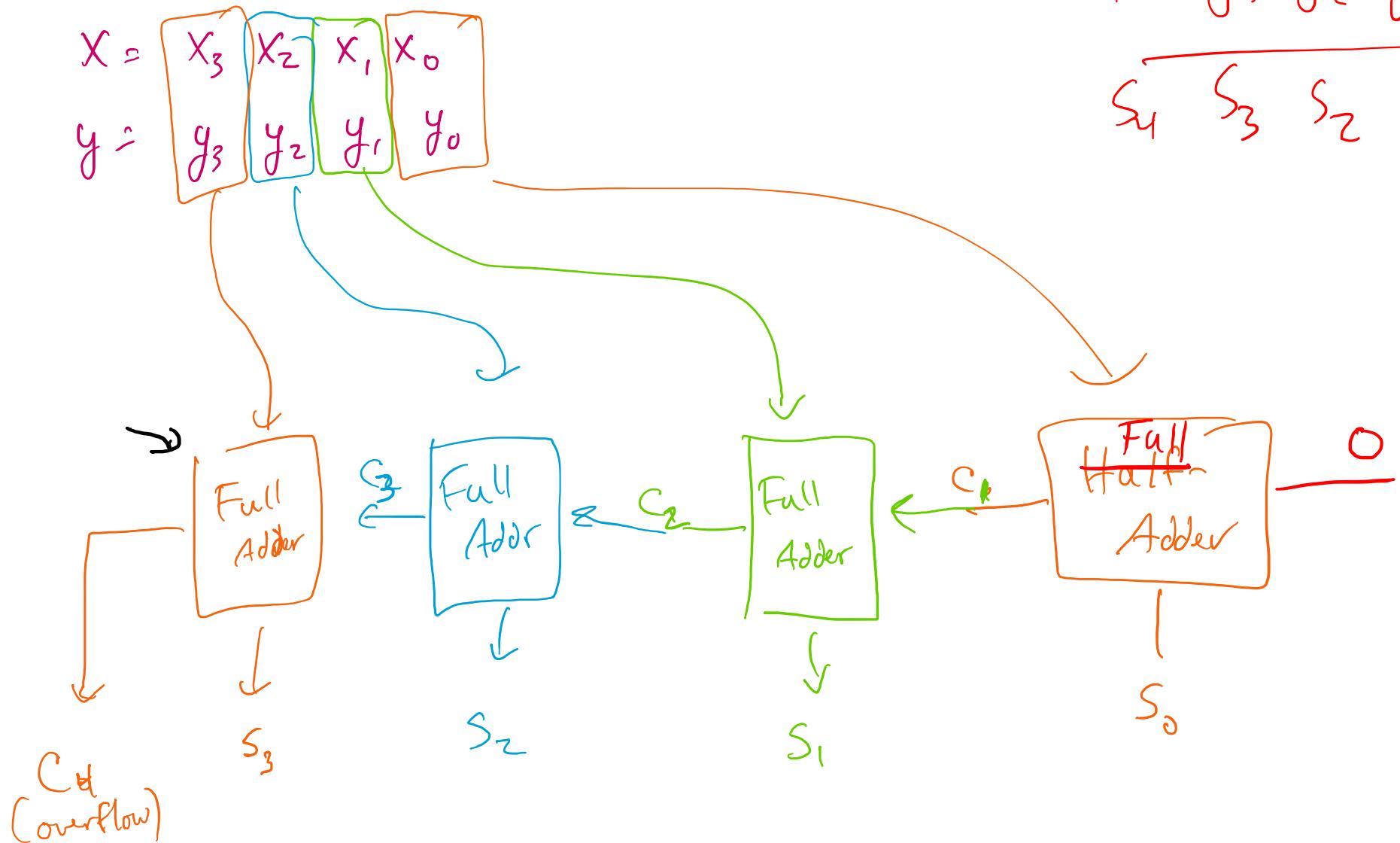
```
module decoder (
    input [1:0] sel,
    output logic [3:0] out
);

always_comb begin
    out = 4'b0000; //default
    case(sel)
        2'b00: out=4'b0001;
        2'b01: out=4'b0010;
        2'b10: out=4'b0100;
                                // what about sel==2'b11?
    endcase
end

endmodule
```

Always specify
defaults for
always_comb!

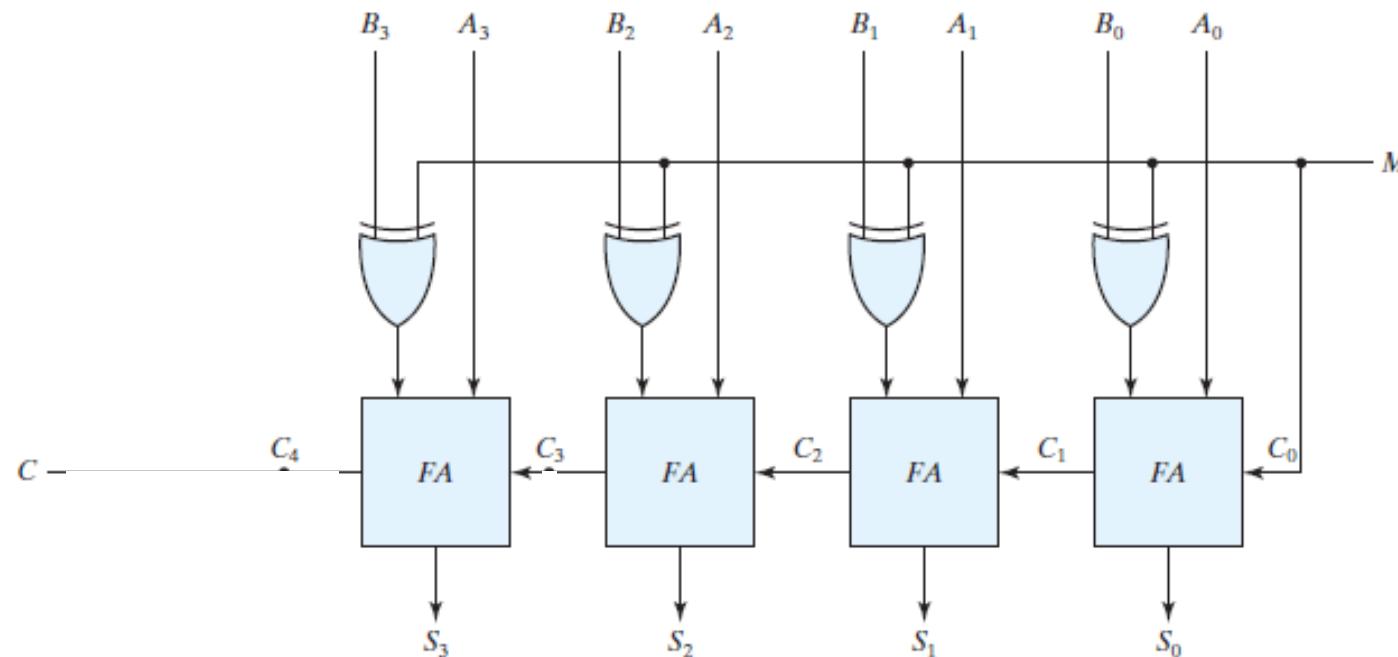
Ripple-Carry Adder



$$\begin{aligned} X + Y &= X_3 X_2 X_1 X_0 \\ &\quad + Y_3 Y_2 Y_1 Y_0 \\ &\hline S_4 & S_3 & S_2 & S_1 & S_0 \end{aligned}$$

Adder/Subtractor

- Mode input:
 - If $M = 0$, then $S = A + B$, the circuit performs addition
 - If $M = 1$, then $S = A + \bar{B} + 1$, the circuit performs subtraction



Overflow for signed numbers?

- Unsigned

Assume 4-bit addition

$$\begin{array}{r} 10 \\ + 8 \\ \hline 18 \end{array}$$

- Signed

$$\begin{array}{r} 5 \\ + 6 \\ \hline 11 \end{array}$$

Overflow for signed numbers?

$$\begin{array}{r} 10 \\ + 8 \\ \hline 18 \end{array}$$

$$\begin{array}{r} 1010 \\ + 1000 \\ \hline \text{carry } \underline{10010} \end{array} \text{ sum}$$

unsigned = carry out bit
"overflow"

$$\begin{array}{r} \text{Signed} \\ 5 \\ + 6 \\ \hline 11 \end{array}$$

carry = 0 \Rightarrow no overflow?

$$5 = 0101 \quad 1010 \rightarrow 1011$$

$$6 = 0110$$

$$= (0100 +) = (0101) = -5 \leftarrow \text{overflow!}$$

Overflow for signed numbers?

$$\begin{array}{r} -2 \\ + -1 \\ \hline -3 \end{array}$$

$$\begin{array}{r} +2 \\ + -1 \\ \hline +1 \end{array}$$

Overflow for signed numbers?

$$\begin{array}{r} -2 \\ + -1 \\ \hline -3 \end{array} \quad \begin{aligned} -(0010) &= 1101 + 1 = 1110 \Rightarrow \\ -(0001) &= 1110 + 1 = 1111 \end{aligned}$$
$$\begin{array}{r} 1110 \\ + 1111 \\ \hline \boxed{11101}^{\text{sum}} \end{array} = \begin{aligned} -(0010+1) \\ = -(0011) = -3 \end{aligned}$$

No overflow?

$$\begin{array}{r} +2. \quad 0010. \quad 0010 \\ + -1. \quad +- (0001) \quad + \quad 1111 \\ \hline +1 \quad \quad \quad \quad \quad 10001 \end{array} \leftarrow \text{no overflow}$$

stopped
here!

Overflow for signed numbers

$$\begin{array}{r} \text{XXXX} \\ + \text{YYYY} \\ \hline \text{ZZZZ} \end{array}$$

Same = no overflow

different = overflow



XOR (C_4, C_5)

Signed numbers
Only!

unsigned = regular
carry

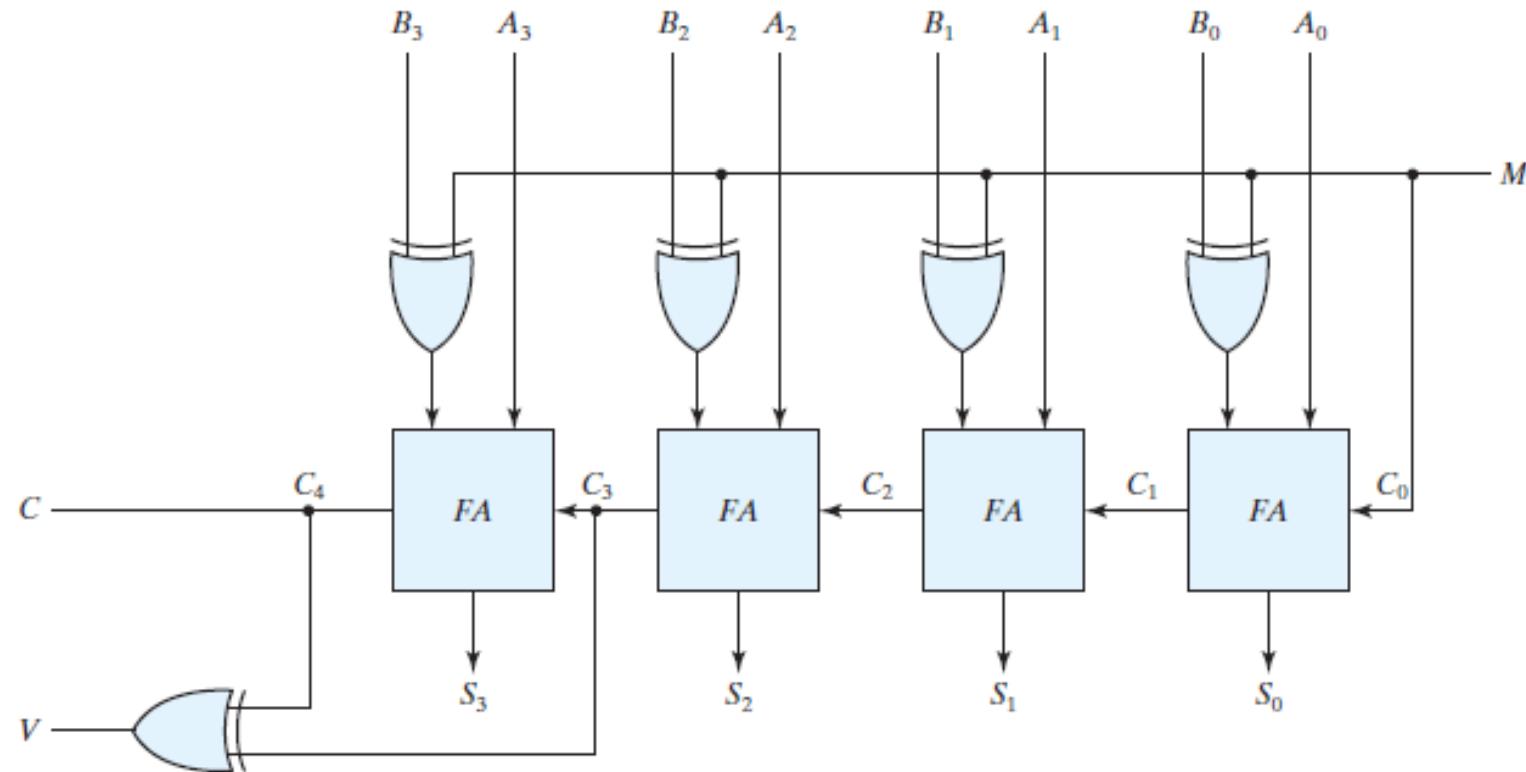
Overflow detection

- When two numbers with n digits each are added and the sum is a number occupying $n + 1$ digits, we say that an overflow occurred.
- The detection of an overflow after the addition of two binary numbers depends on whether the numbers are considered to be signed or unsigned.
- When two unsigned numbers are added, an overflow is detected from the end carry out of the most significant position.
- In case of signed numbers, two details are important:
 - the leftmost bit always represents the sign,
 - negative numbers are in 2's-complement form.
- When two signed numbers are added:
 - the sign bit is treated as part of the number
 - the end carry does not indicate an overflow.

Overflow detection

- An overflow cannot occur after an addition if one number is positive and the other is negative, since adding a positive number to a negative number produces a result whose magnitude is smaller than the larger of the two original numbers.
- An overflow may occur if the two numbers added are both positive or both negative.
- An overflow condition can be detected by observing the carry into the sign bit position and the carry out of the sign bit position.
 - If these two carries are equal, there was no overflow.
 - If these two carries are not equal, an overflow has occurred.
- If the two carries are applied to an exclusive-OR gate, an overflow is detected when the output of the gate is equal to 1.

Adder with overflow detection



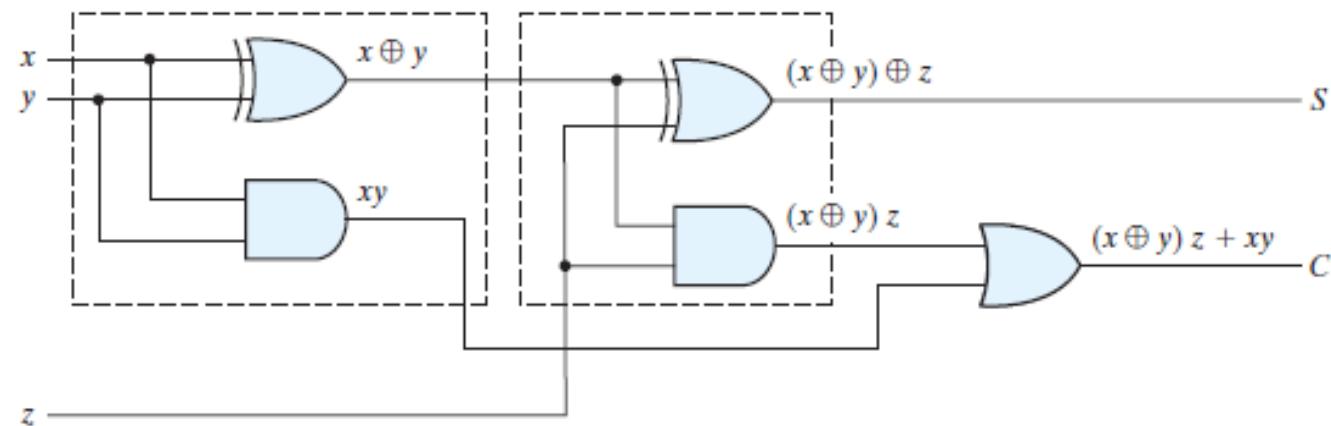
Gate Delay

- Gates are not magic, they are physical
- Takes time for changes flow through
- Assume 5ps (5E-12) / gate

- How fast can we update our adder?

Full Adder Gate Delay

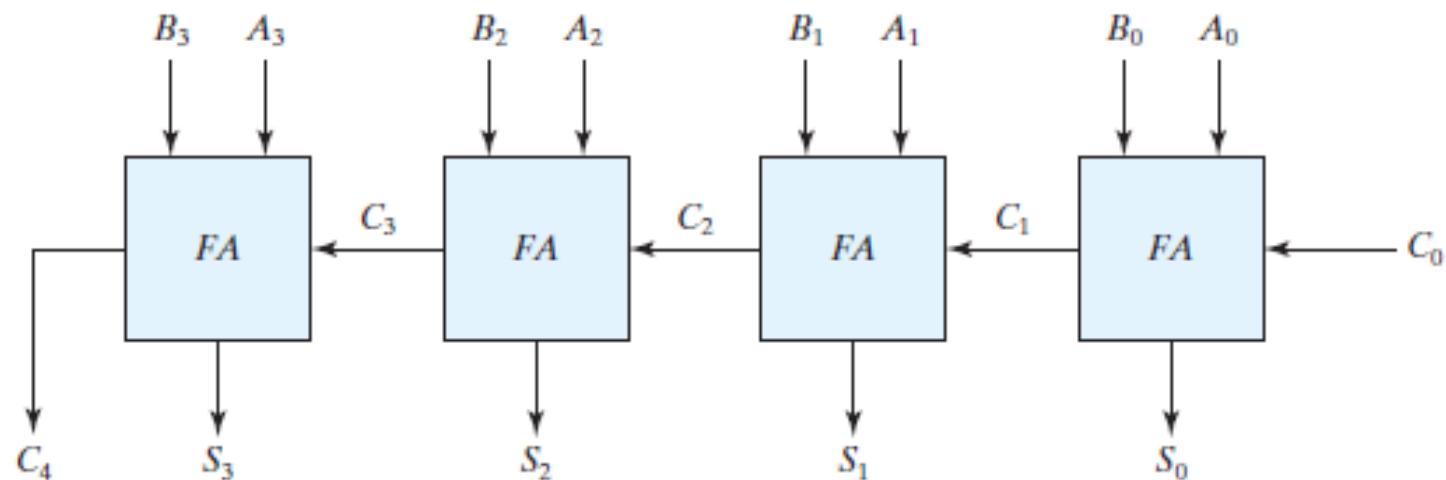
- Assume 5ps/gate



- What is the total delay on s ? on c ?

Ripple-Carry Gate Delays

- What is the total delay here?



Adder Gate Delays

- What is the total delay for:
 - 1-bit addition:
 - 4-bit addition:
 - 8-bit addition:
 - 16-bit addition:
 - 32-bit addition:
 - 64-bit addition:

Adder Gate Delays

- What is the total delay for:

- 1-bit addition:

15 ps

- 4-bit addition:

60 ps

- 8-bit addition:

120 ps

- 16-bit addition:

240 ps

- 32-bit addition:

480 ps

- 64-bit addition:

960 ps = ~ 1 GHz

Faster Adder Options?

- What can be done to build a faster 64-bit adder?

WARNING: MAJOR TOPIC SHIFT

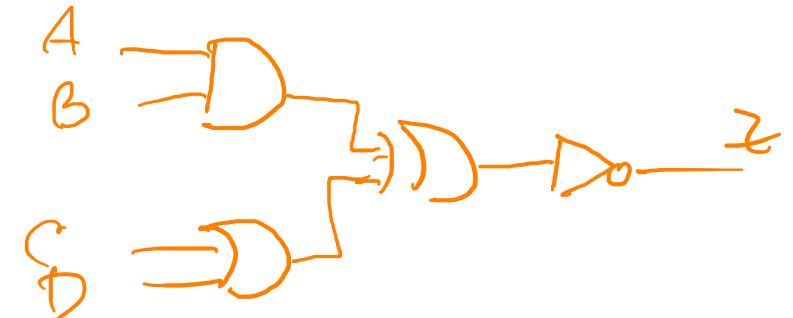
SEQUENTIAL LOGIC

Stopped here!

Sequential vs. Combinational

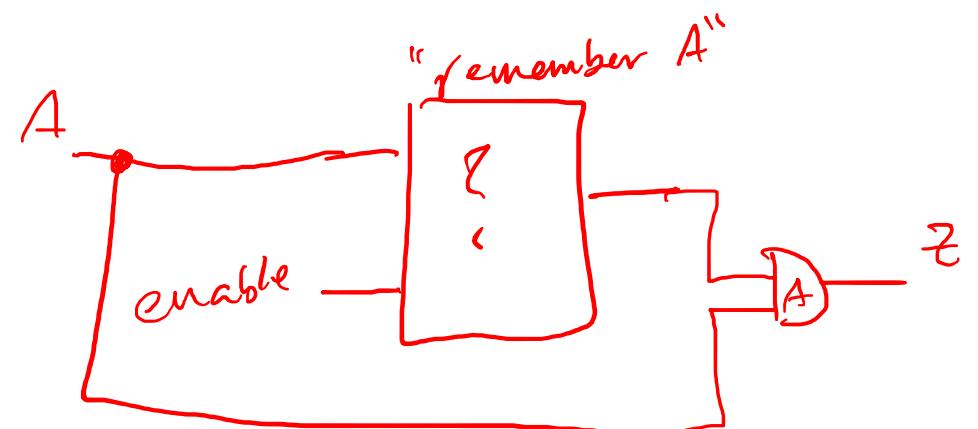
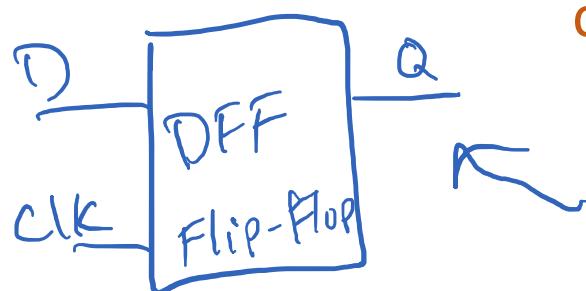
- Combinational Logic

- The output is a combination of the **current inputs only**



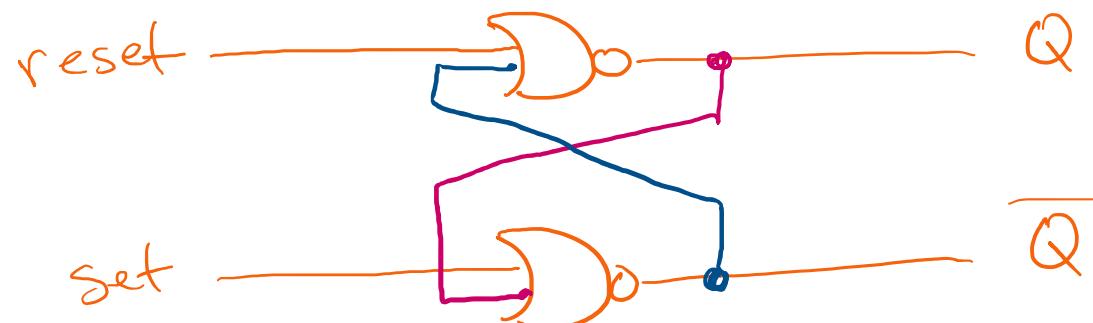
- Sequential Logic

- The output is a combination of the **current and past inputs**



SR Latch

 = 1
 = 0

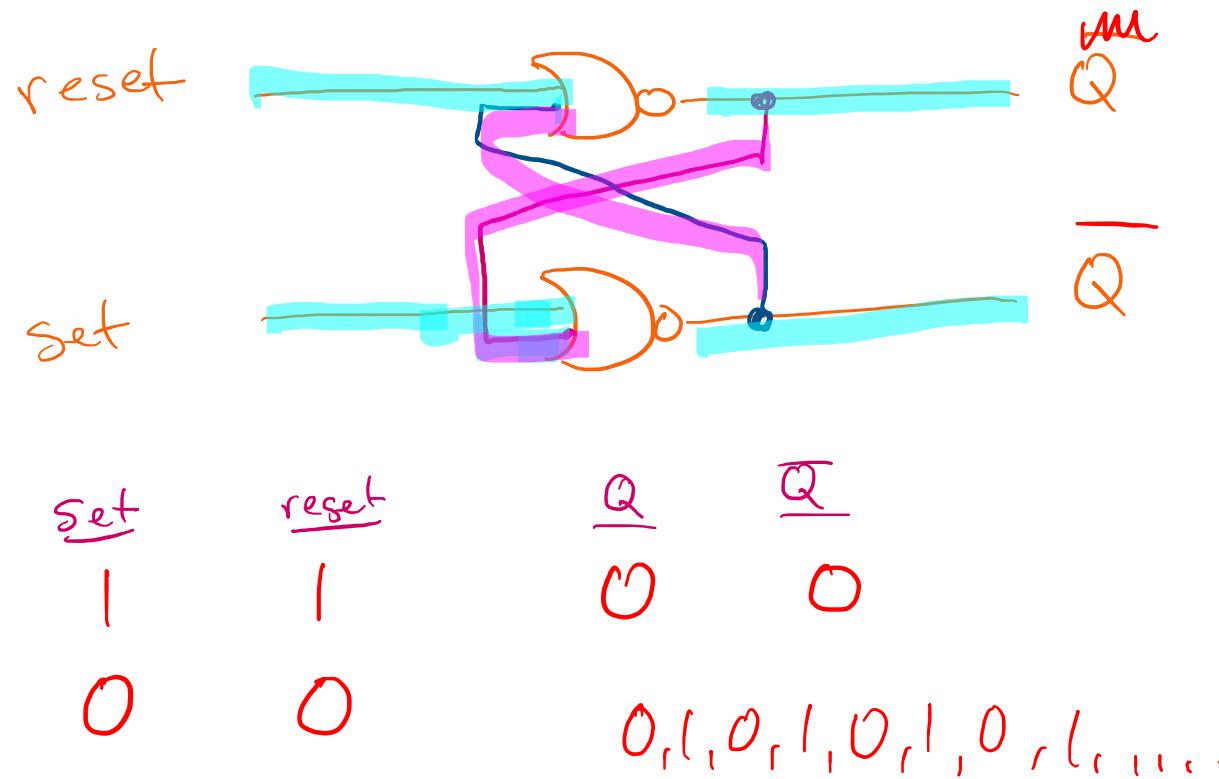


<u>reset</u>	<u>Set</u>	<u>Q</u>	<u>\bar{Q}</u>
0	1	1	0
0	0	1	0
1	0	0	1
0	0	0	1

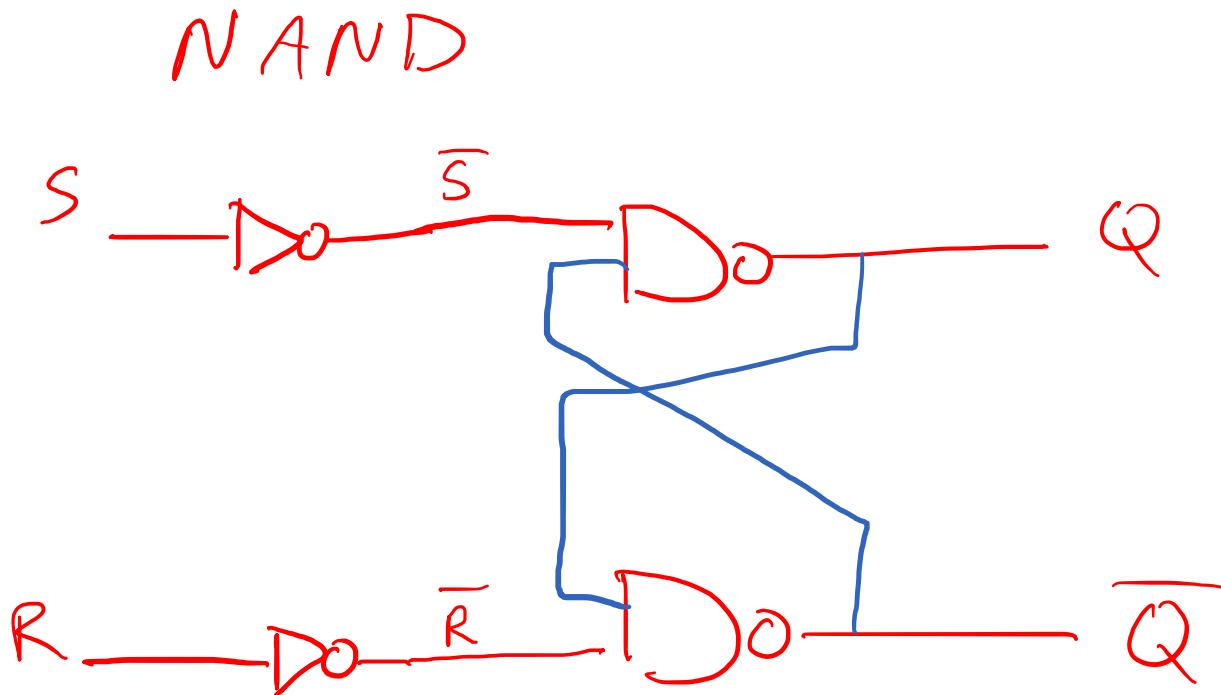
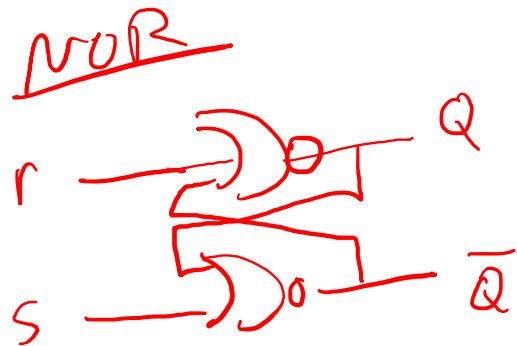
Same inputs,
different output!
 \Rightarrow Internal
state!

SR Latch w/ S=1 & R=1

 = 1
 = 0



SR Latch w/NAND gates

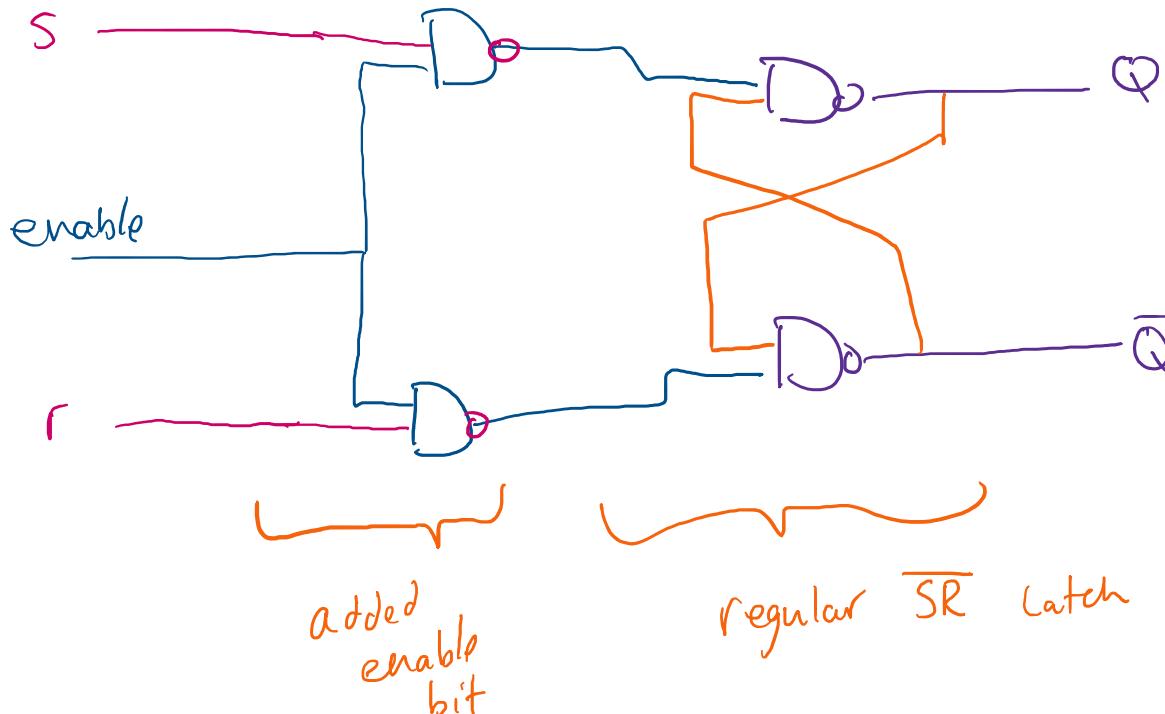


→ better setup for ~~⇒~~ Flip-Flops

→ easier for me to draw

SR Latch with Enable

Prevent changes in S & R from changing circuit output

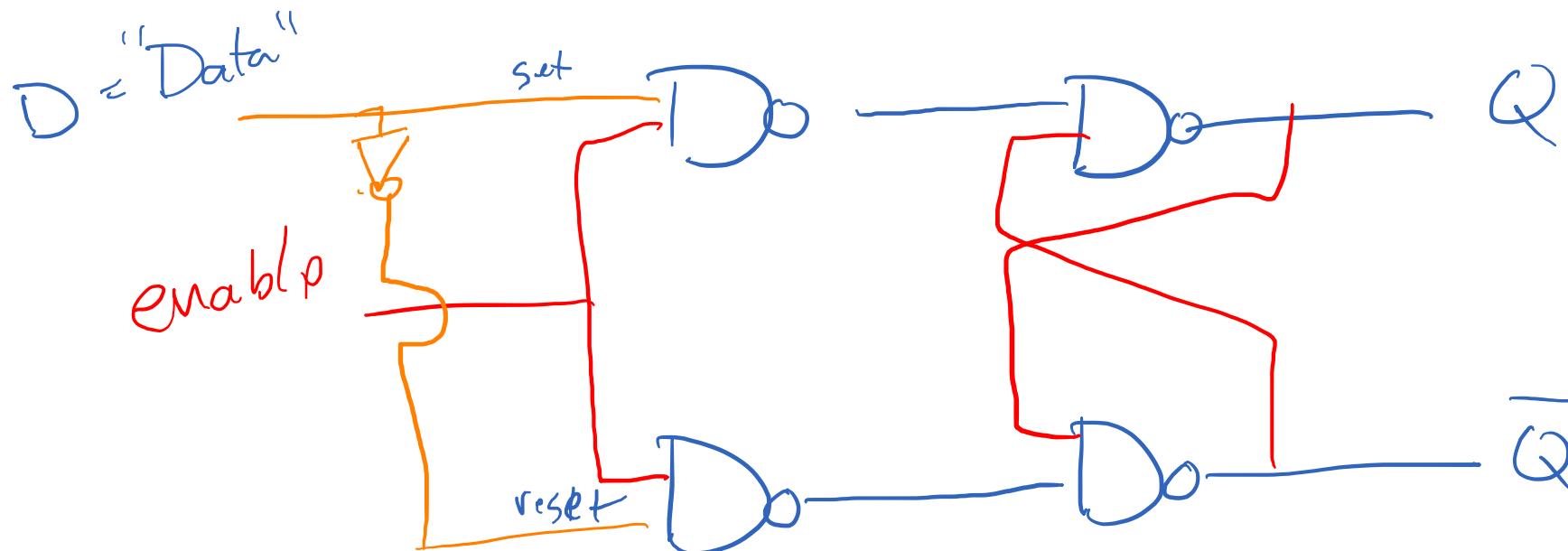


S	R	E	Q	\bar{Q}
x	x	0	Q	\bar{Q}
1	0	1	1	0
0	1	1	0	1

* assume
no $S=1$
 $r=1$

D-Latch

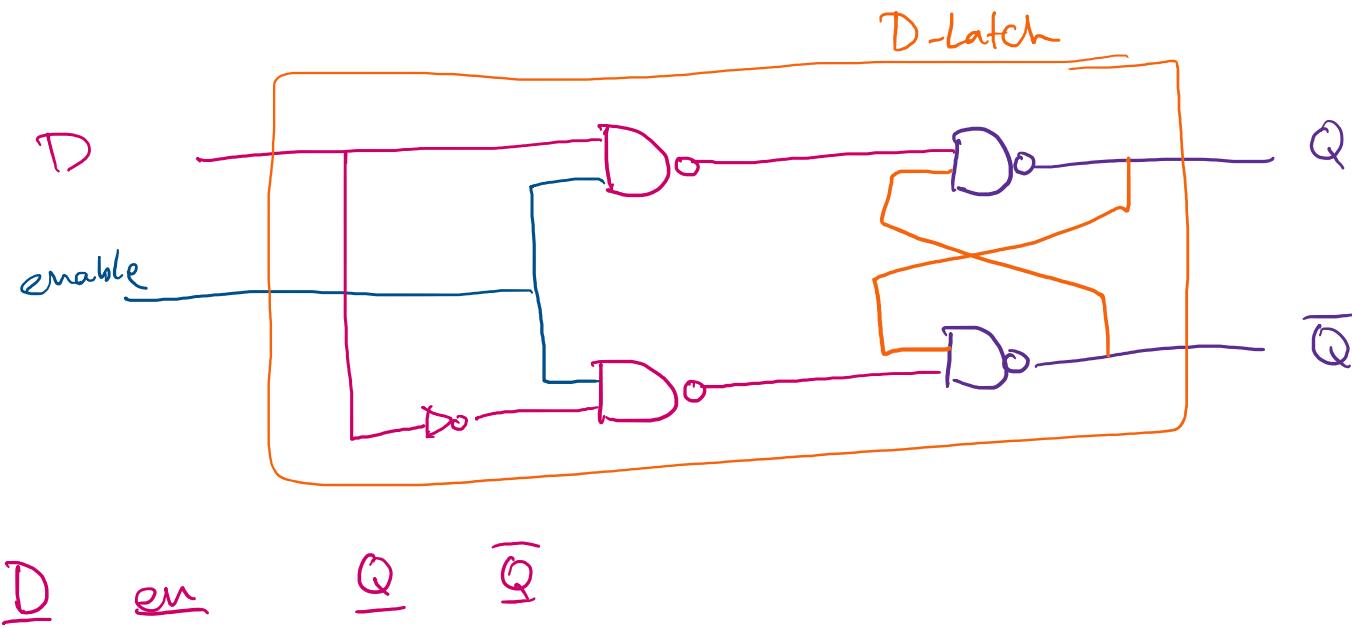
“Data” Latch



D-Latch

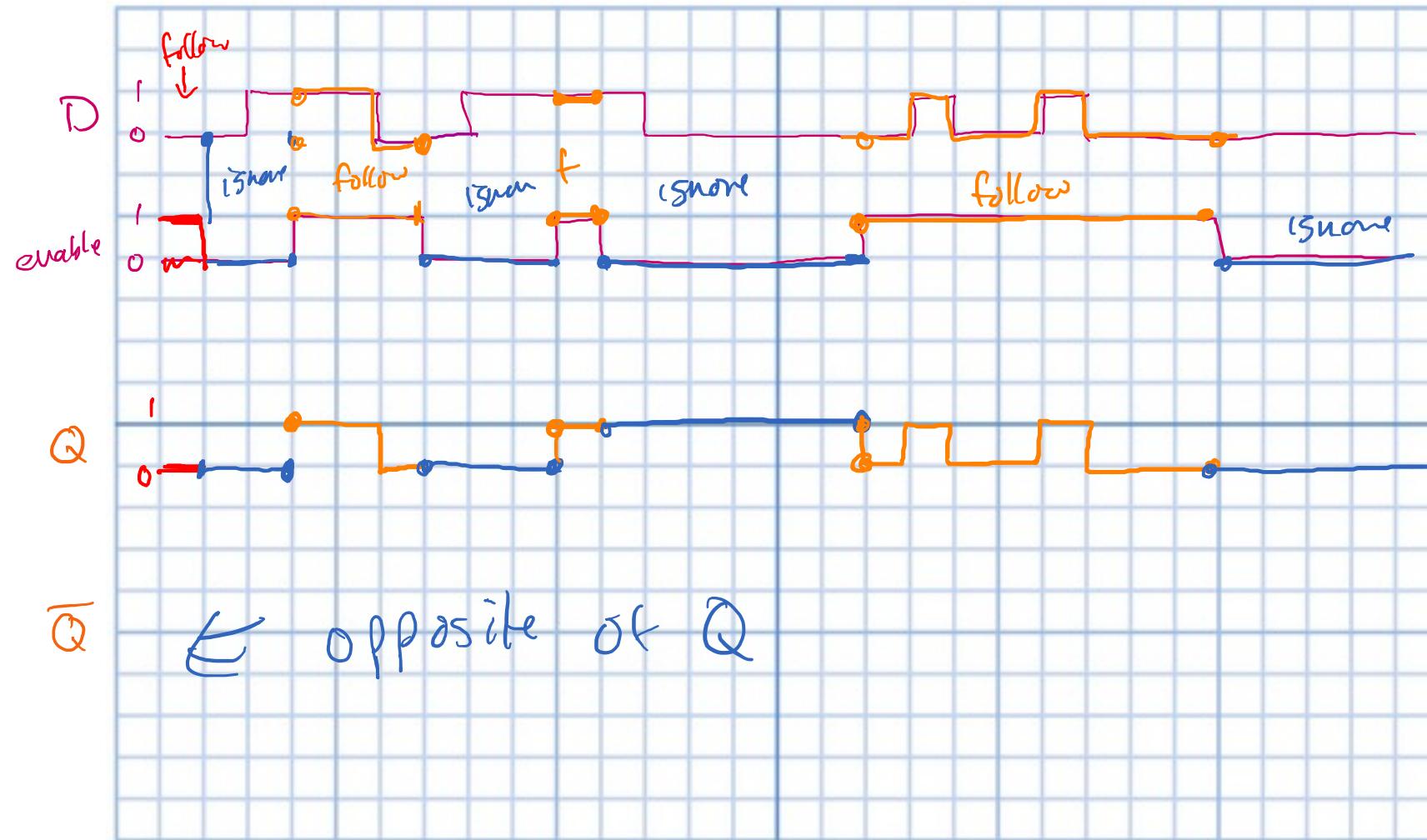
 = 1

 = 0



Inputs to D Latches

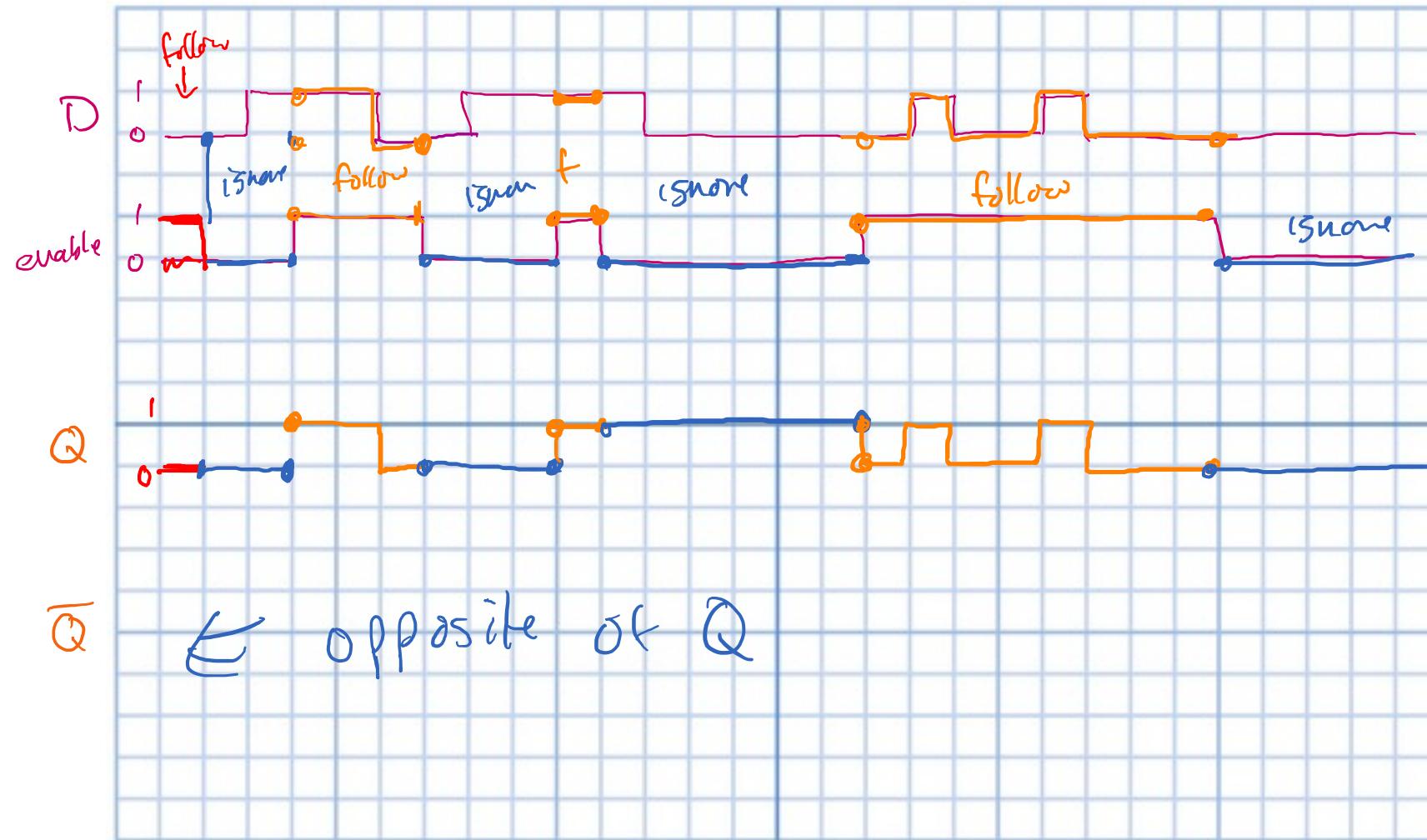
+ assume negligible gate delays



Q follows D when enable = 1,
otherwise \bar{Q} ignores D.

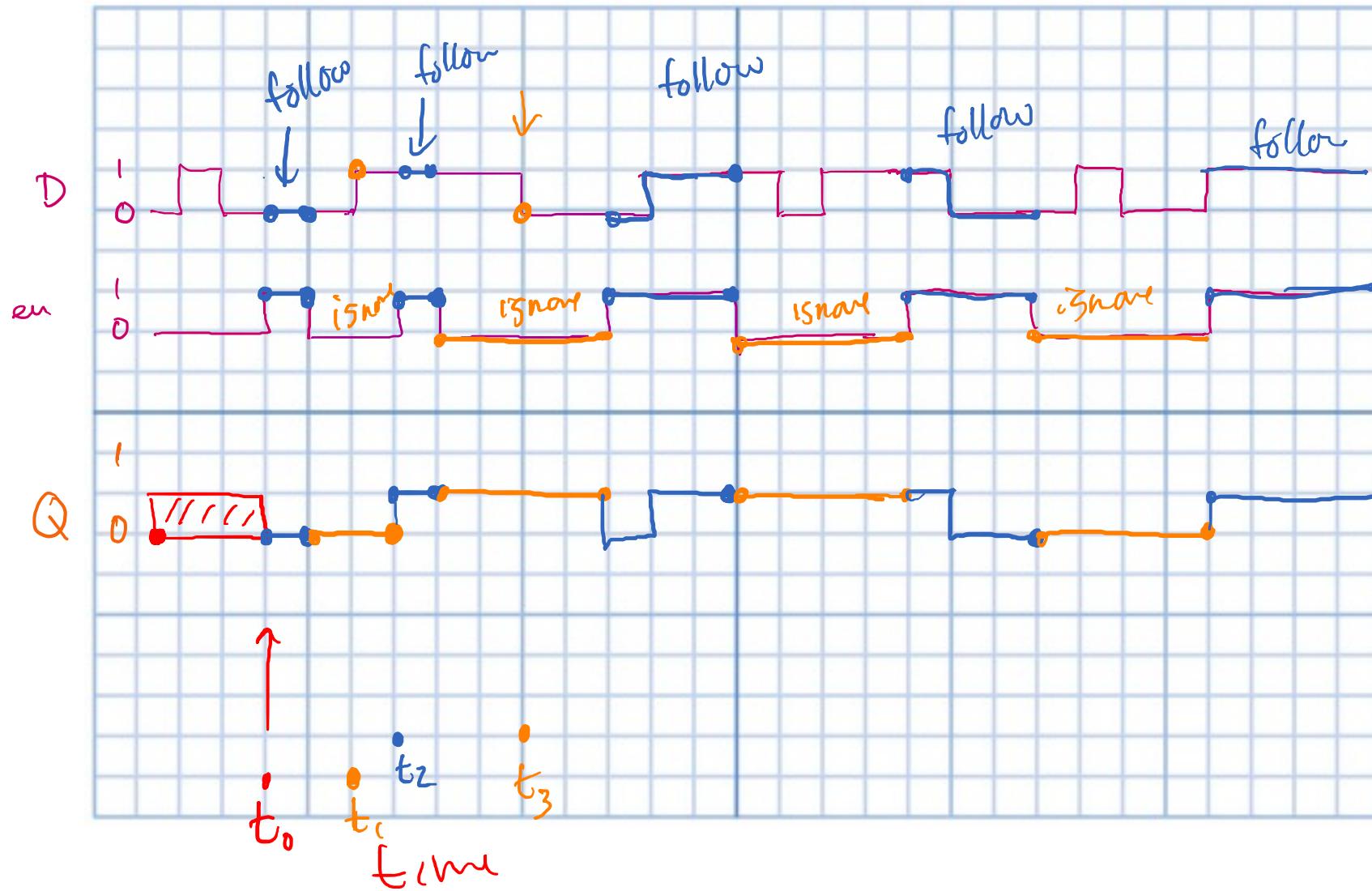
Inputs to D Latches

+ assume negligible gate delays



Q follows D when enable = 1,
otherwise \bar{Q} ignores D.

Inputs to D Latches



Glitches

→ unintended, short, errors in
boolean logic

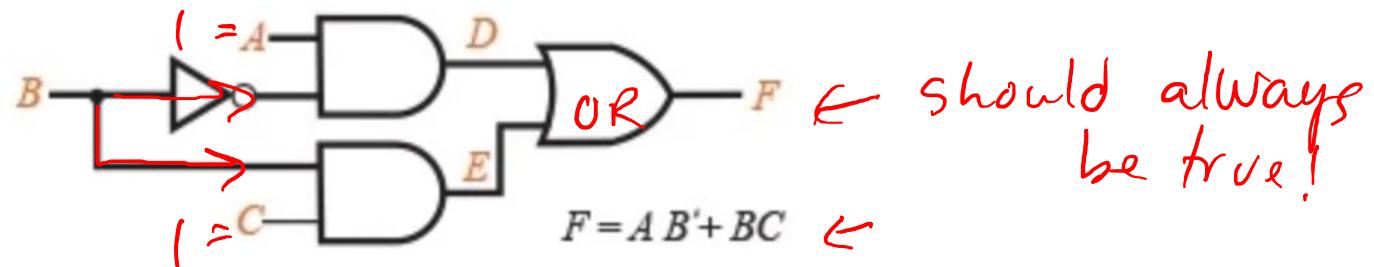
→ caused by gate delays

- Assume 10ps / gate.

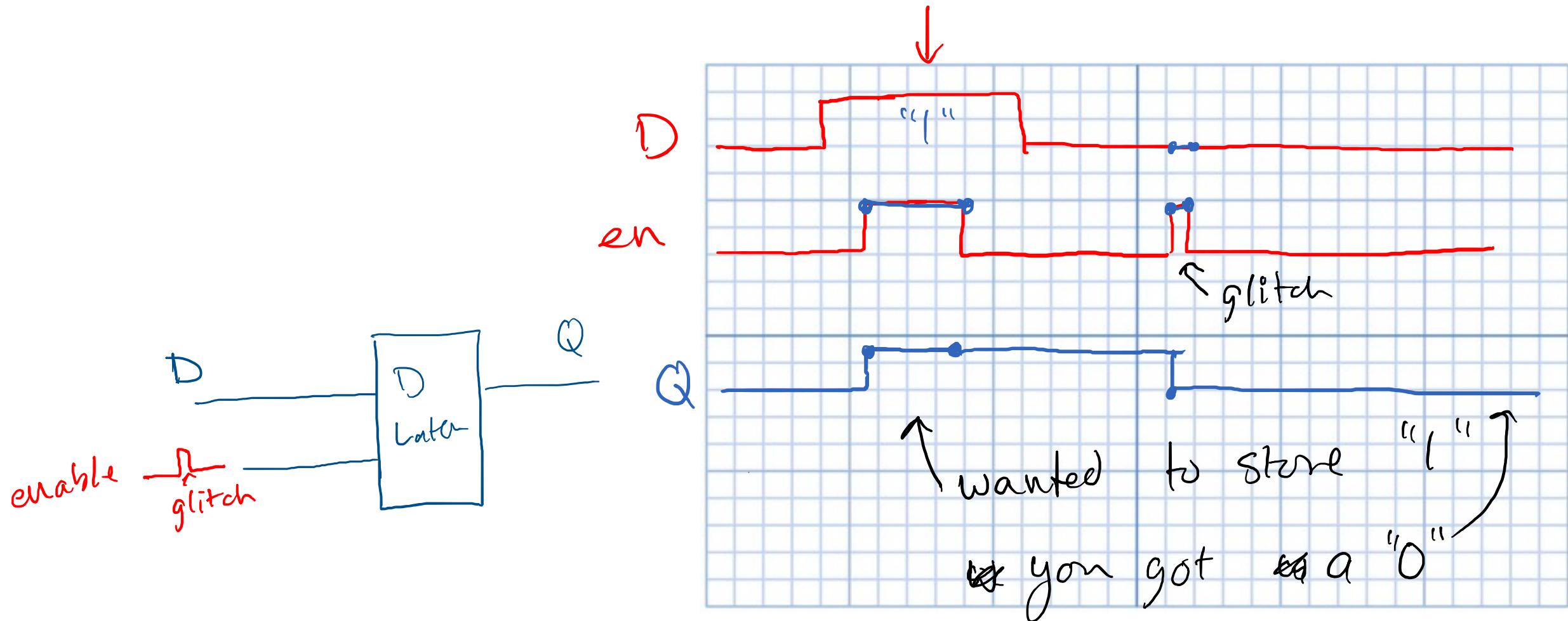
- $A=1$, $C=1$, B falls

- 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
```

Defaults

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

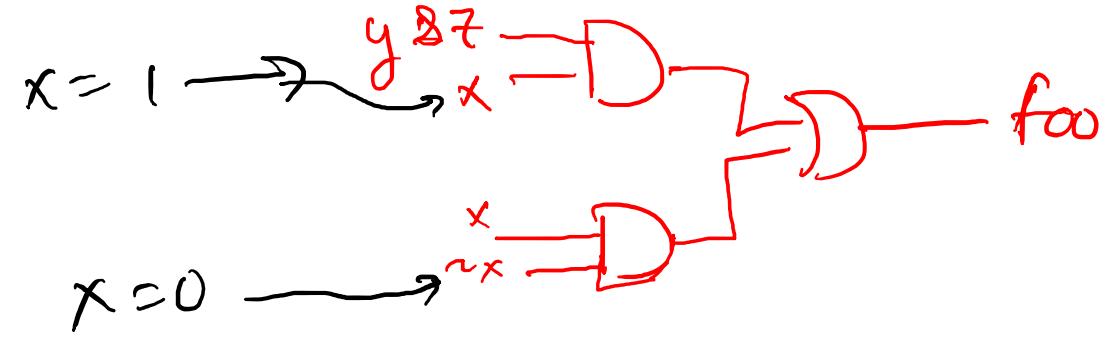
```
always_comb begin
```

→ **foo = x; bar = x; //good: defaults**

if (x) foo = y & z; //
if (x) bar = y | z ; //

```
end
```

x _____ foo



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

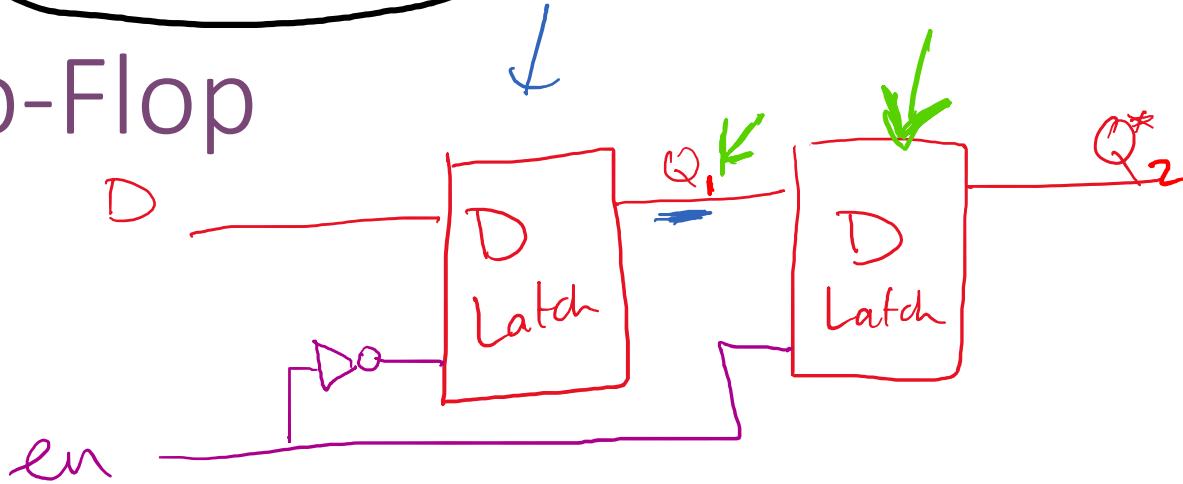
Always specify defaults for *always_comb*!

Always specify defaults for
always_comb!

Always specify
defaults for
always_comb!

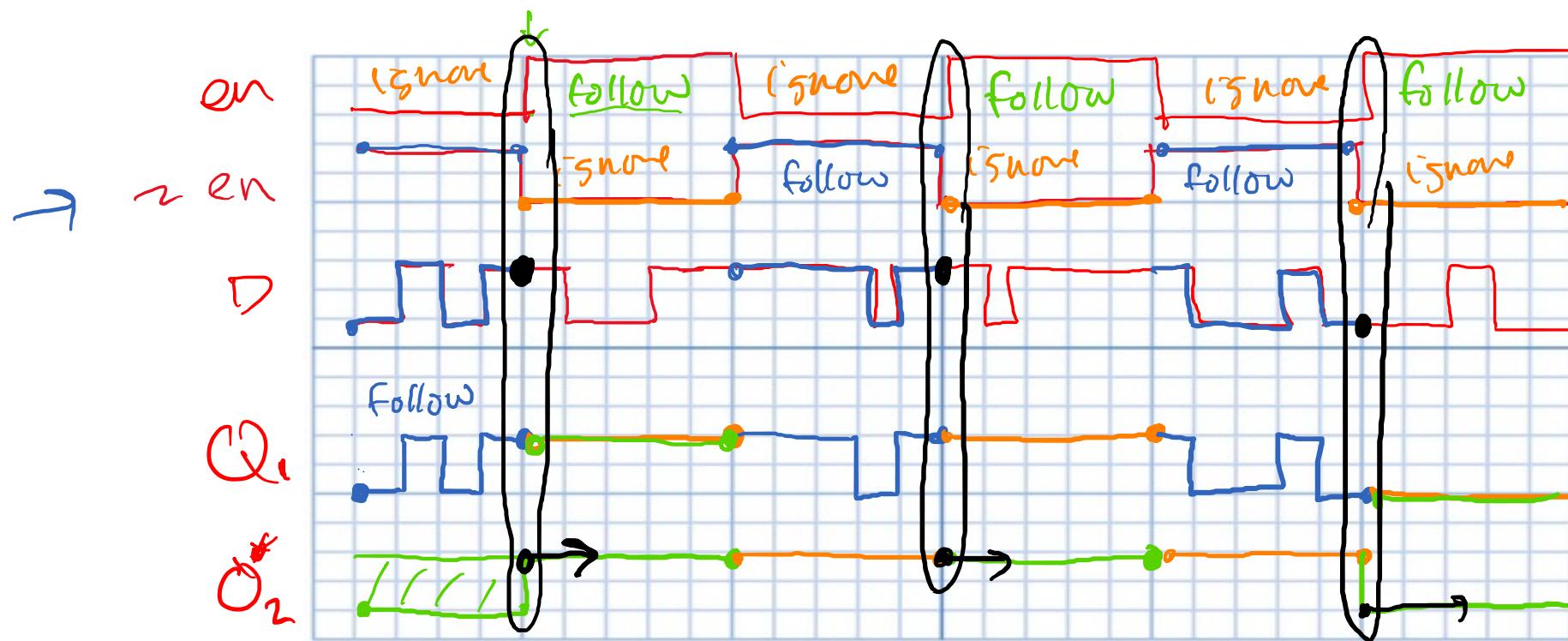
stopped here!

D Flip-Flop



* no gate delays

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



Levels vs. Edges

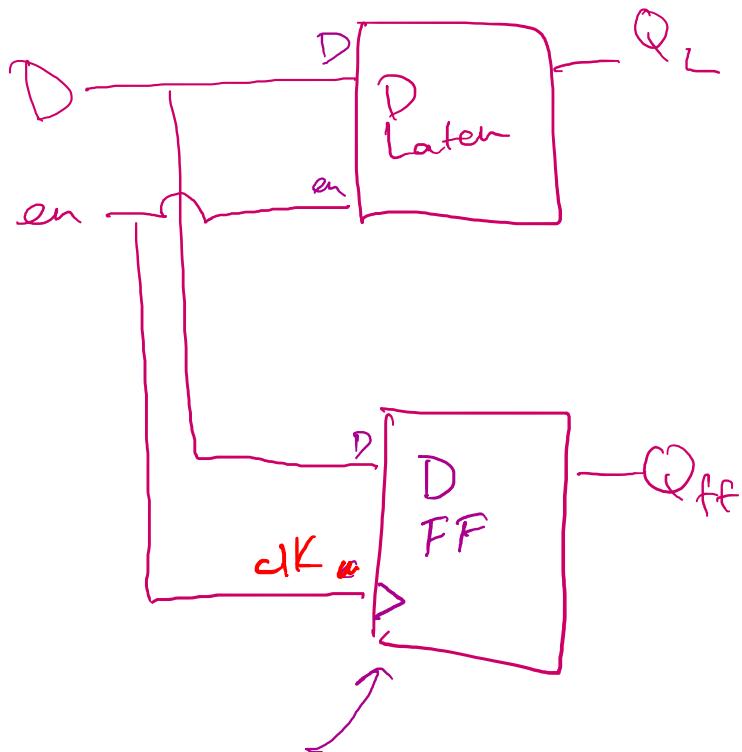
D latch \rightarrow Q follows D whenever enable is 1



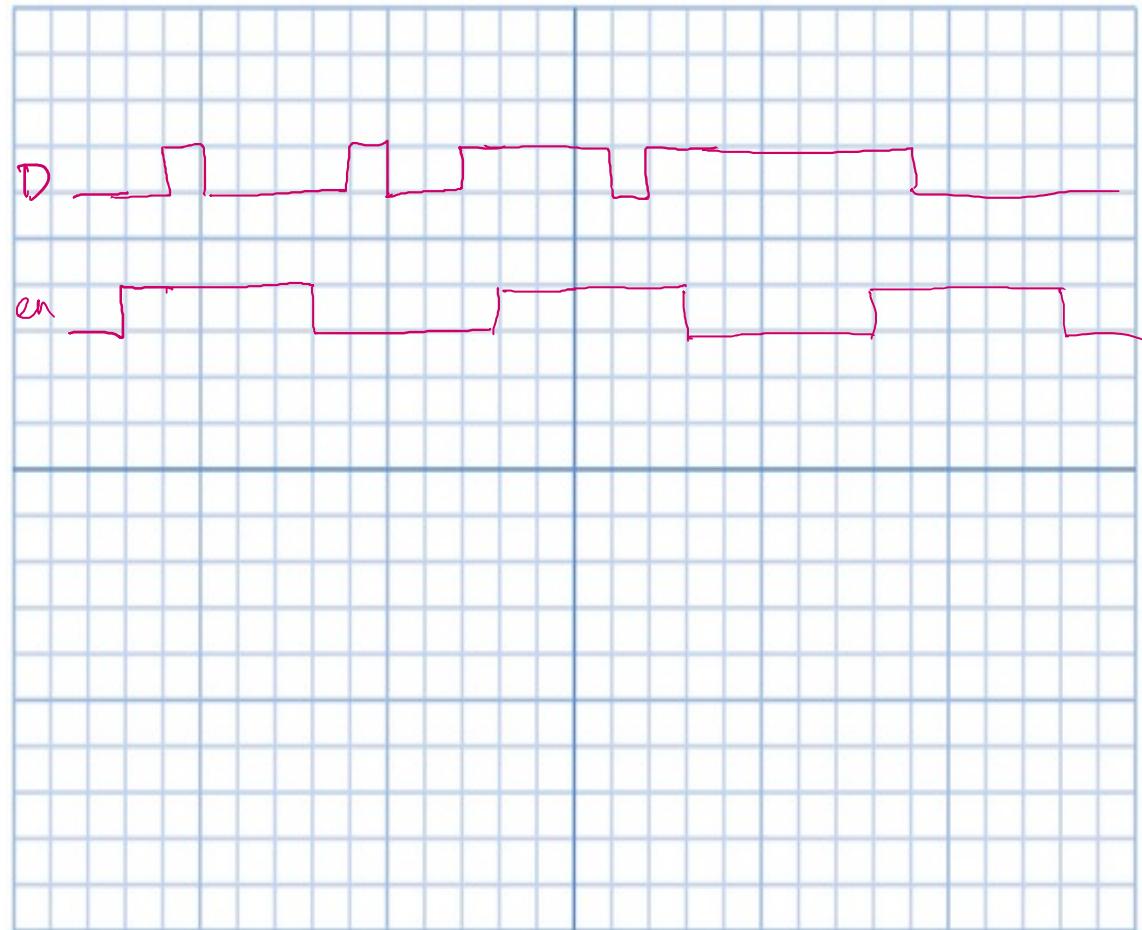
D flip-flop \rightarrow Q follows D on rising edge of enable



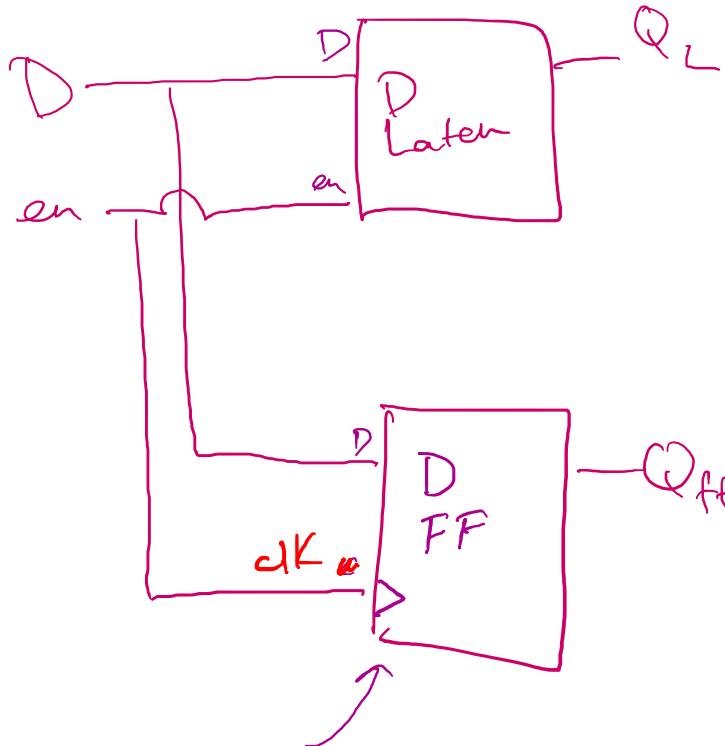
D Flip-Flop vs. D Latch



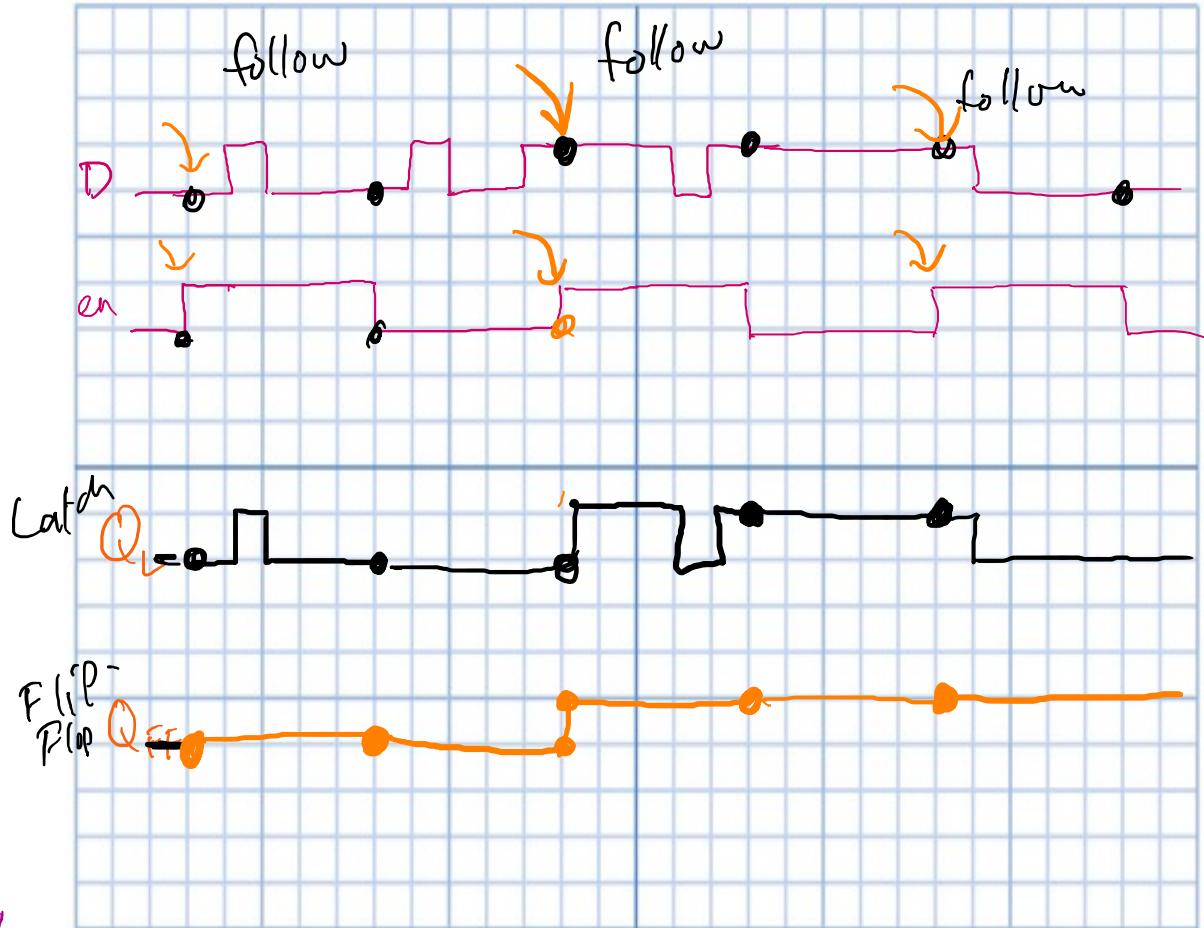
The " > " symbol tells you
it is Flip-Flop



D Flip-Flop vs. D Latch

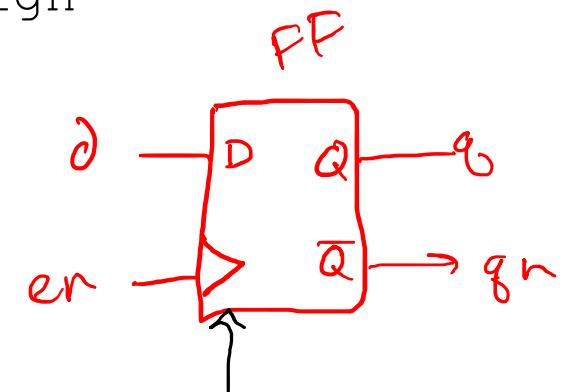
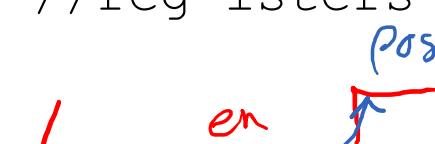


The " > " symbol tells you
it is Flip-Flop



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) //positive edge of enable
    begin
        q <= d; //non-blocking assign
        qn <= ~d; //optional
    end
endmodule
```



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.

$x = 2$
 $y = x + 1$

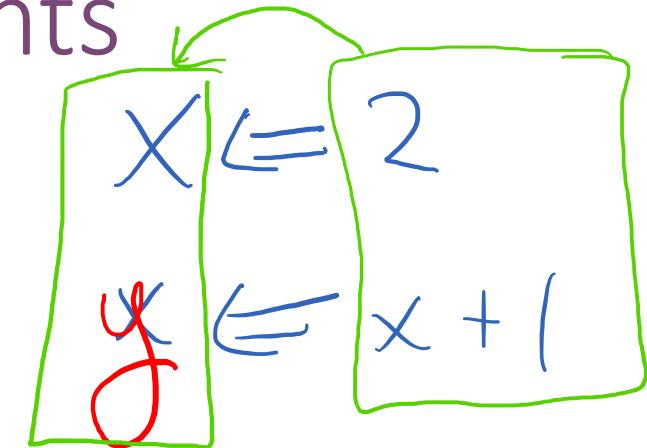
$x = 2 ;$
 $y = x + 1 ;$
 $\rightarrow x = 2$
 $y = 3$

LHS RHS
 $x \Leftarrow 2$

Blocking vs. NonBlocking Assignments

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

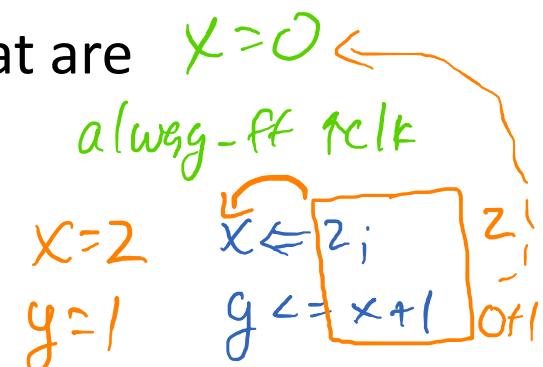
- Execute concurrently



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

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

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

```
begin
```

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

```
end
```

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

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

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

$$\begin{aligned} x &= 2 \\ y &= 3 \\ z &= 1 \end{aligned}$$

$\text{clk} \uparrow$

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

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

+ defaults!

never
hold state

No flip flops!

NON-BLOCKING ($\leftarrow=$) for

always_ff

← always for

flip flops

(always hold state)

D-FlipFlop w/Clock

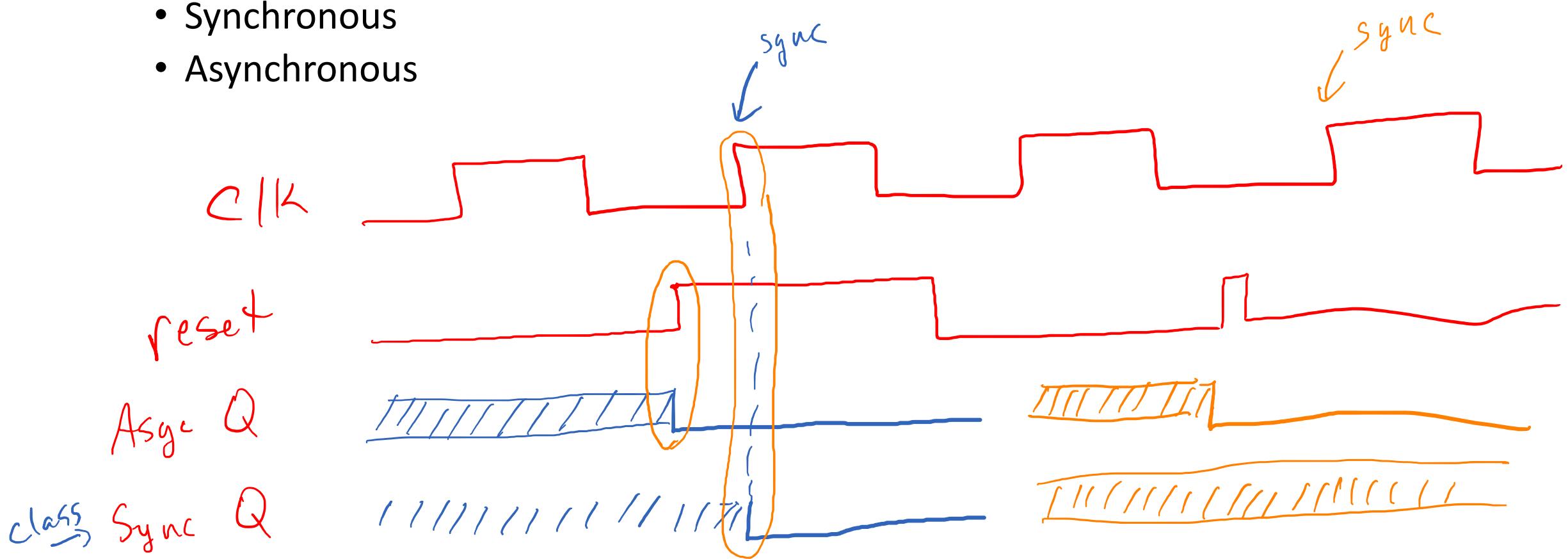
$$q \rightarrow d \rightarrow q_{\text{new}} \rightarrow d_{\text{new}} \rightarrow q_{\text{new}_2}$$

```
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

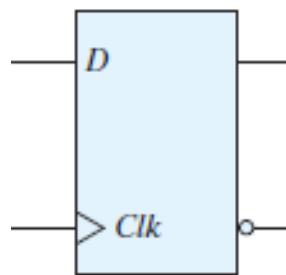
- Two different ways to build in a reset
 - Synchronous
 - Asynchronous



D-FF's with Reset

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

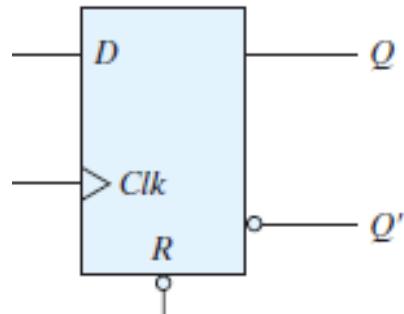
Verilog models of D flip-flop



Edge triggered D flip-flop:

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

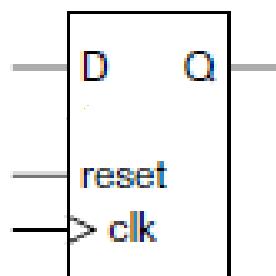
No reset
ff



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;
```