

Test

ENGR 210 / CSCI B441
“Digital Design”

Latches + Flip Flops

Andrew Lukefahr

Course Website

fangs-bootcamp.github.io

Write that down!

Announcements

- P2: You should be done.
- P3: Get going.
- P4: Up and ready.

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

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!

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?

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

$$\begin{array}{r} ^0\text{1010} \\ + 1000 \\ \hline \underline{10010} \end{array} \text{sum}$$

unsigned = carry out bit
"overflow"

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

$$\begin{array}{r} ^0\text{0101} \\ + 0110 \\ \hline \boxed{0\text{1011}} \end{array} \text{Signed}$$

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

$$6 = 0110$$

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

carry = 0 \Rightarrow NO overflow?

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"

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

$$\begin{array}{r} 0101 \\ + 0110 \\ \hline \boxed{01011} \end{array}$$

carry = 0 \Rightarrow no overflow?

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

$$6 = 0110$$

$$= -(0100 + 1) = -(0101) = -5 \leftarrow \text{overflow!}$$

Overflow for signed numbers?

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

$$\begin{array}{r} 7 \\ + 7 \\ \hline - \\ +14 ? \end{array}$$

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?

$$\begin{array}{r} 7 \\ + 7 \\ \hline +1 \end{array} \quad \begin{array}{r} 0100 \\ + 0100 \\ \hline \end{array} \quad \begin{array}{r} 00100 \\ + 00100 \\ \hline 01000 \end{array}$$

0 1 0 0 0 <- overflow

stopped
here!

Overflow for signed numbers

$$\begin{array}{r} \text{c c} \\ \text{c c} \\ \text{X X X X} \\ + \text{Y Y Y Y} \\ \hline \text{Z Z Z Z} \end{array}$$

Same = no overflow

Different = overflow



XOR (C_4, C_5)

Signed numbers
Only!

unsigned = regular
carry

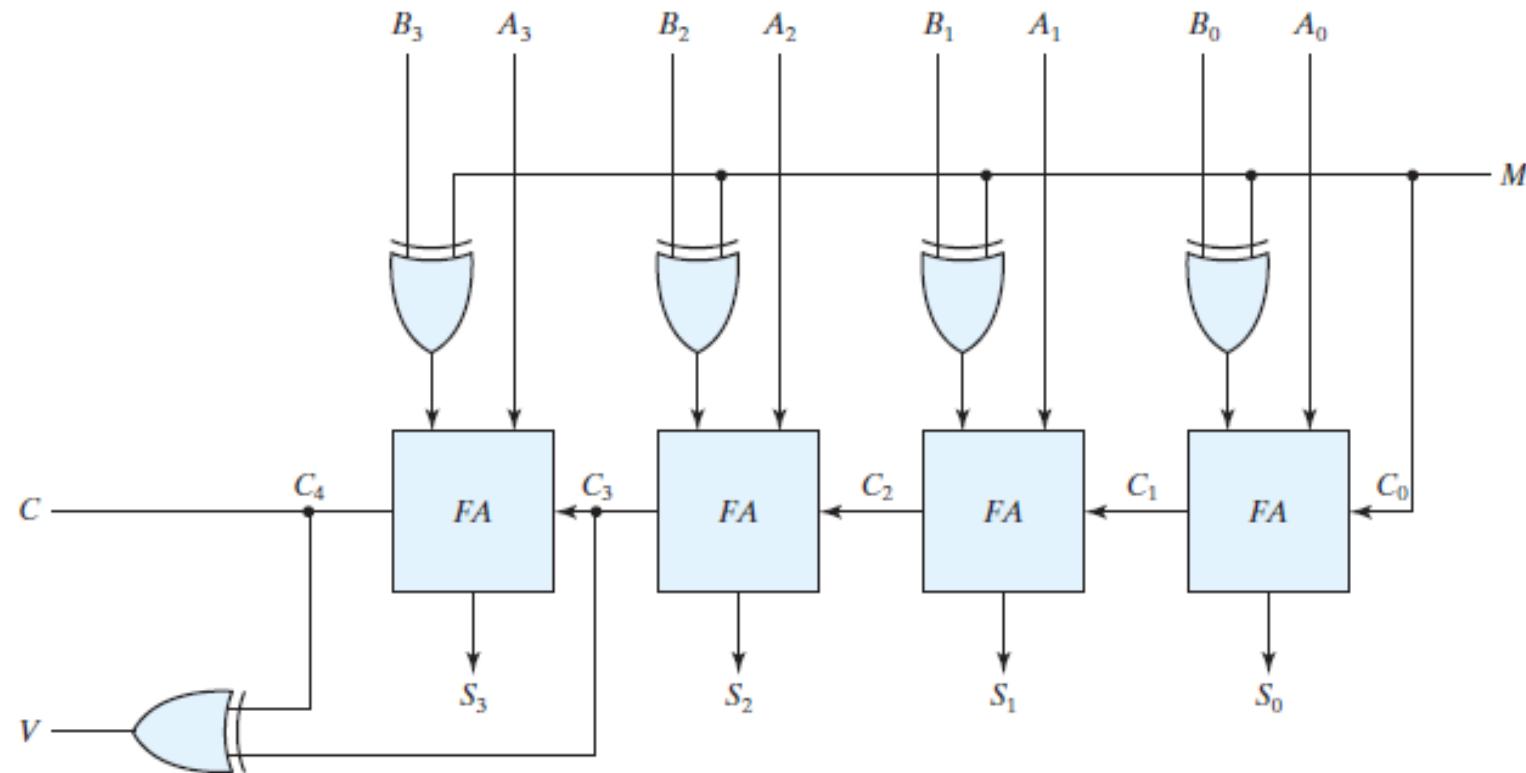
Overflow detection

- 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



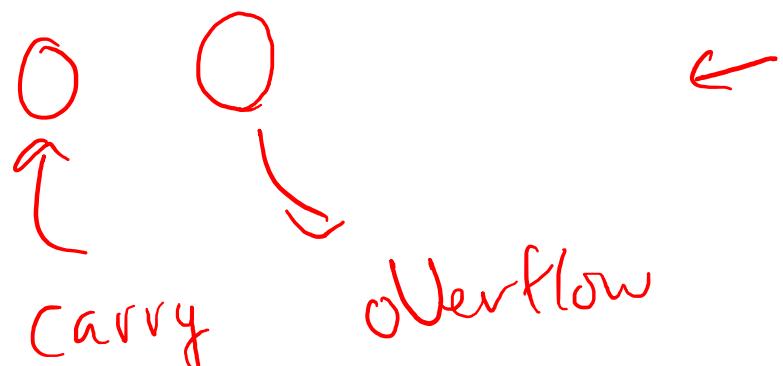
P3 Tips

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

$$a = 0000\ 0001 \quad r = 0000\ 0000$$

$$b = 0000\ 0001 =$$

$$s = \underline{\underline{1010}}$$



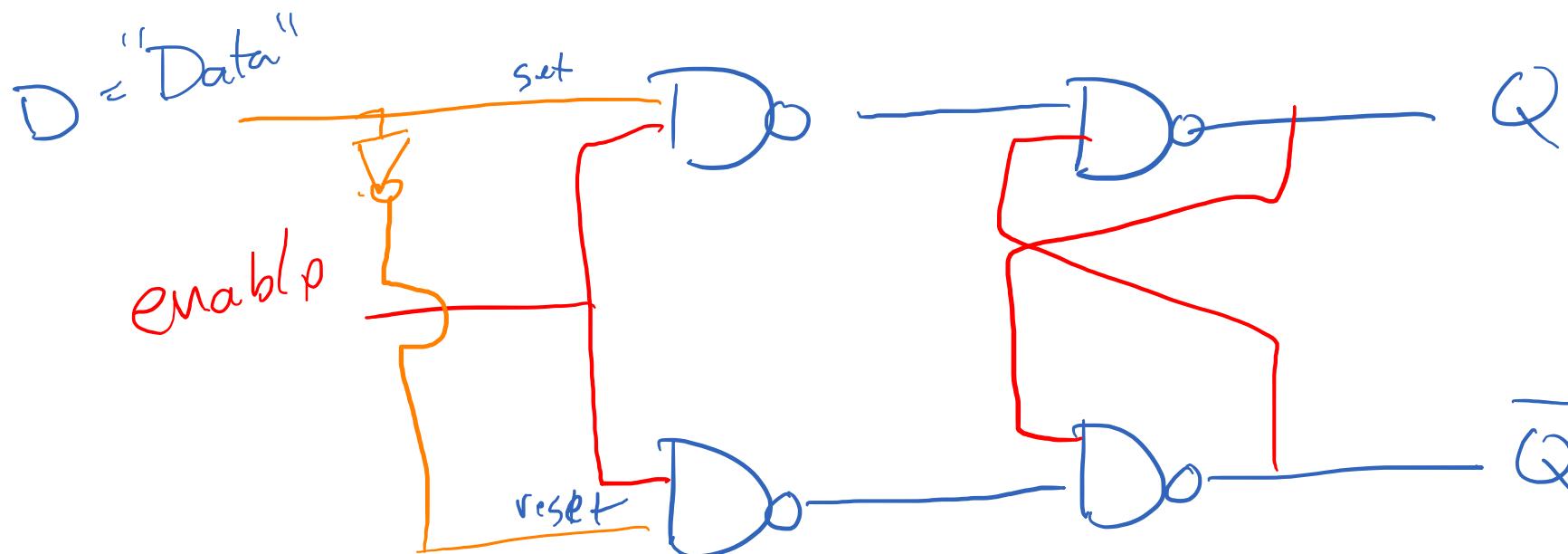
$c = 0$ for everything but
Addition & Subtraction

$\sqrt{ } = \text{defined for Addition \& Sub}$
 $\rightarrow \underline{\text{undefined}} \text{ for all other operations}$

$\rightarrow AG$ will use $\sqrt{ }=0$ for
all cases other than
Addition & Subtraction

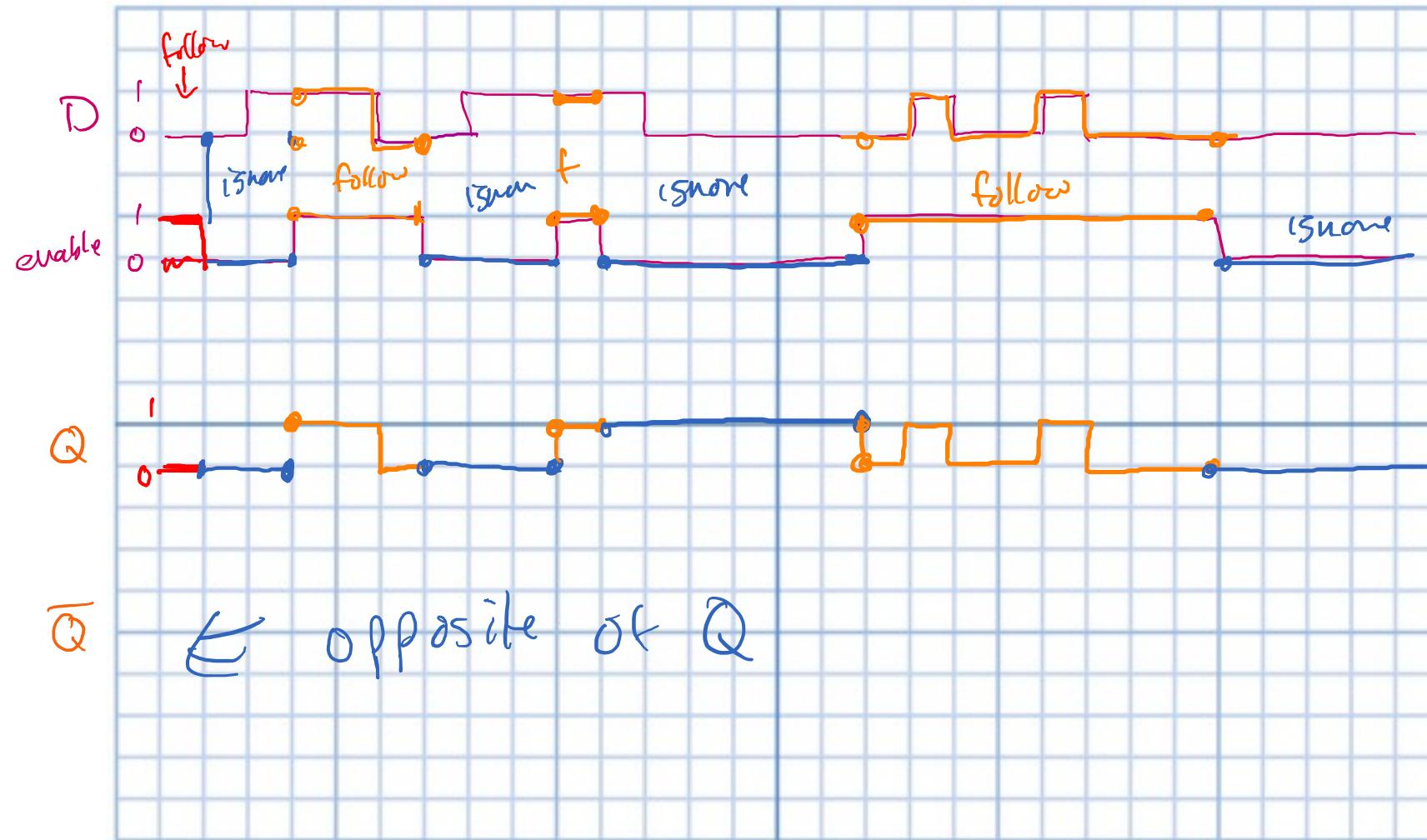
D-Latch

“Data” Latch



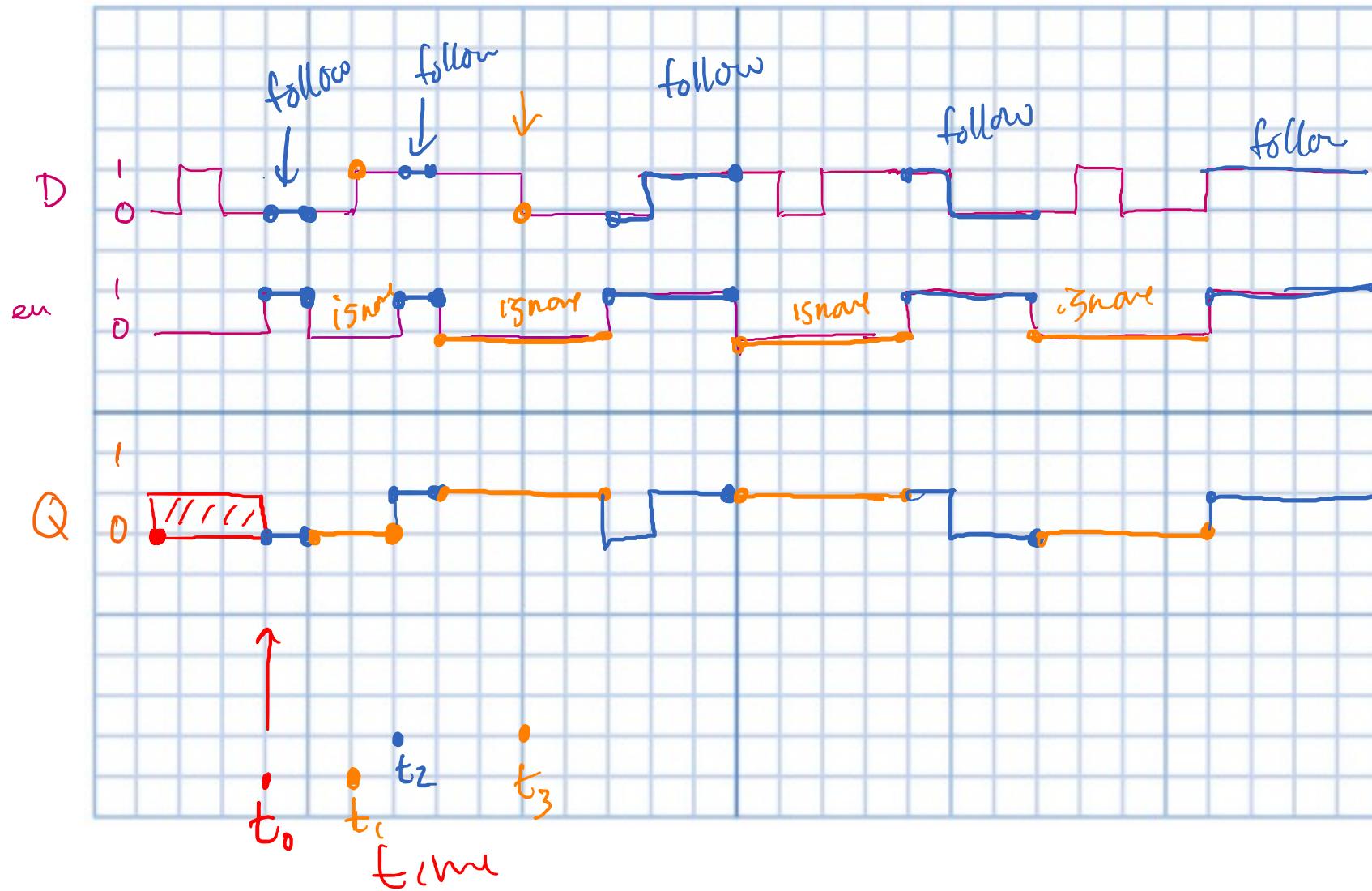
Inputs to D Latches

+ 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

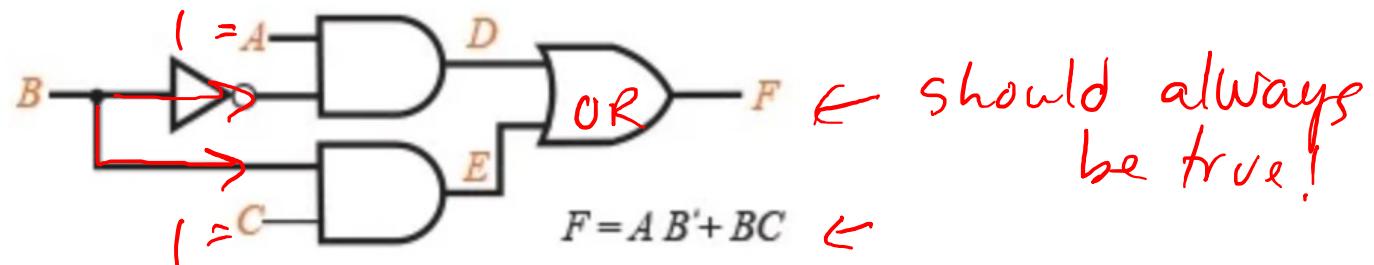
→ caused by gate delays

- Assume 10ps / gate.

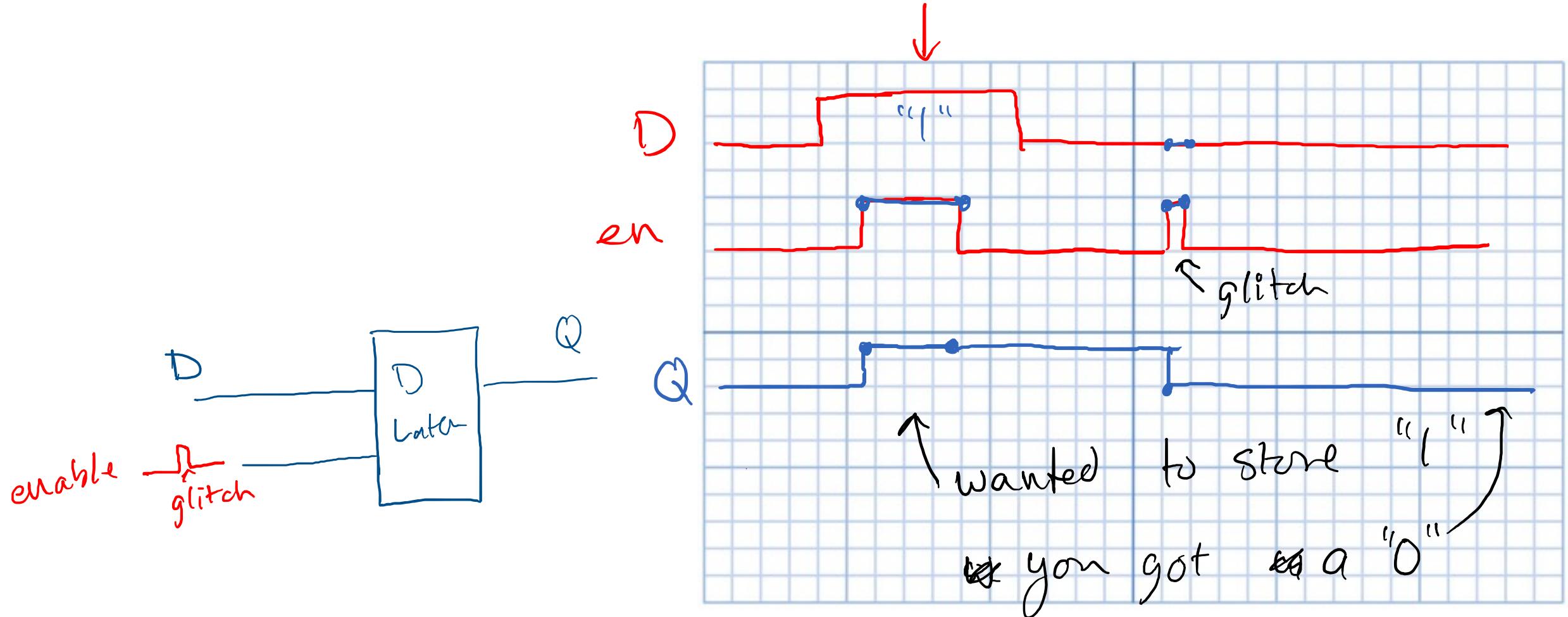
- $A=1$, $C=1$, B falls

- What is F ?

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



Glitches on D-Latches



What's wrong here?

```
wire x, y, z;  
logic foo, bar ;  
  
always_comb begin  
    if (x) foo = y & z;  
    if (x) bar = y | z;  
end
```

Inferred Latches

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

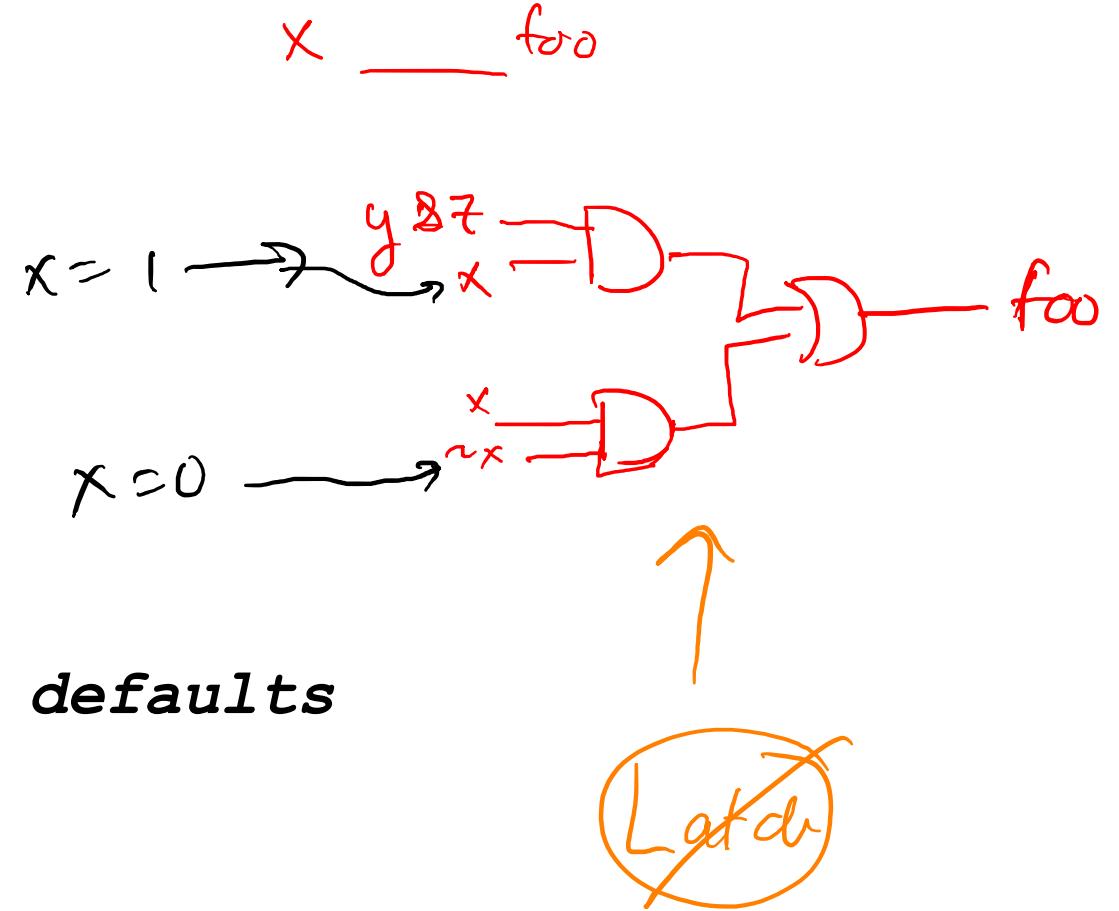
```
always_comb begin  
    foo = x; bar = x; //good: defaults  
    if (x) foo = y & z; //
```

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

```
end
```

What if $x == 0$? $\text{foo} = \text{bar} = x!$

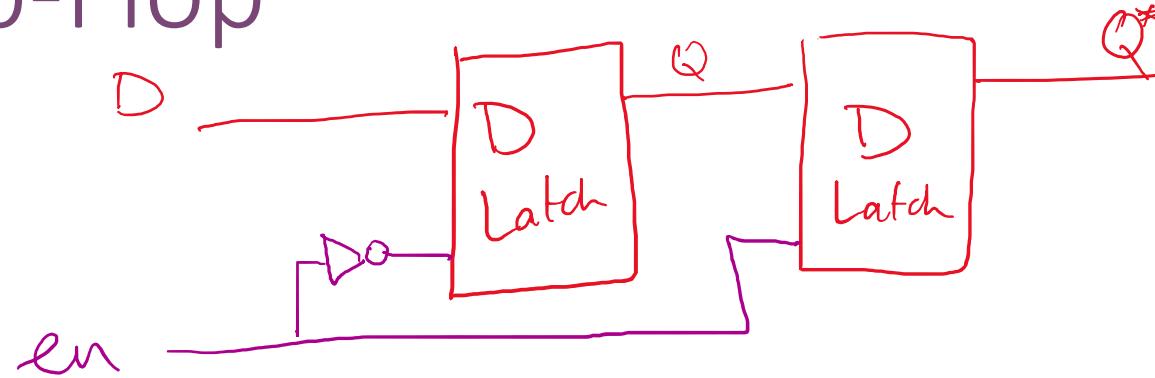
Always specify defaults for always_comb!



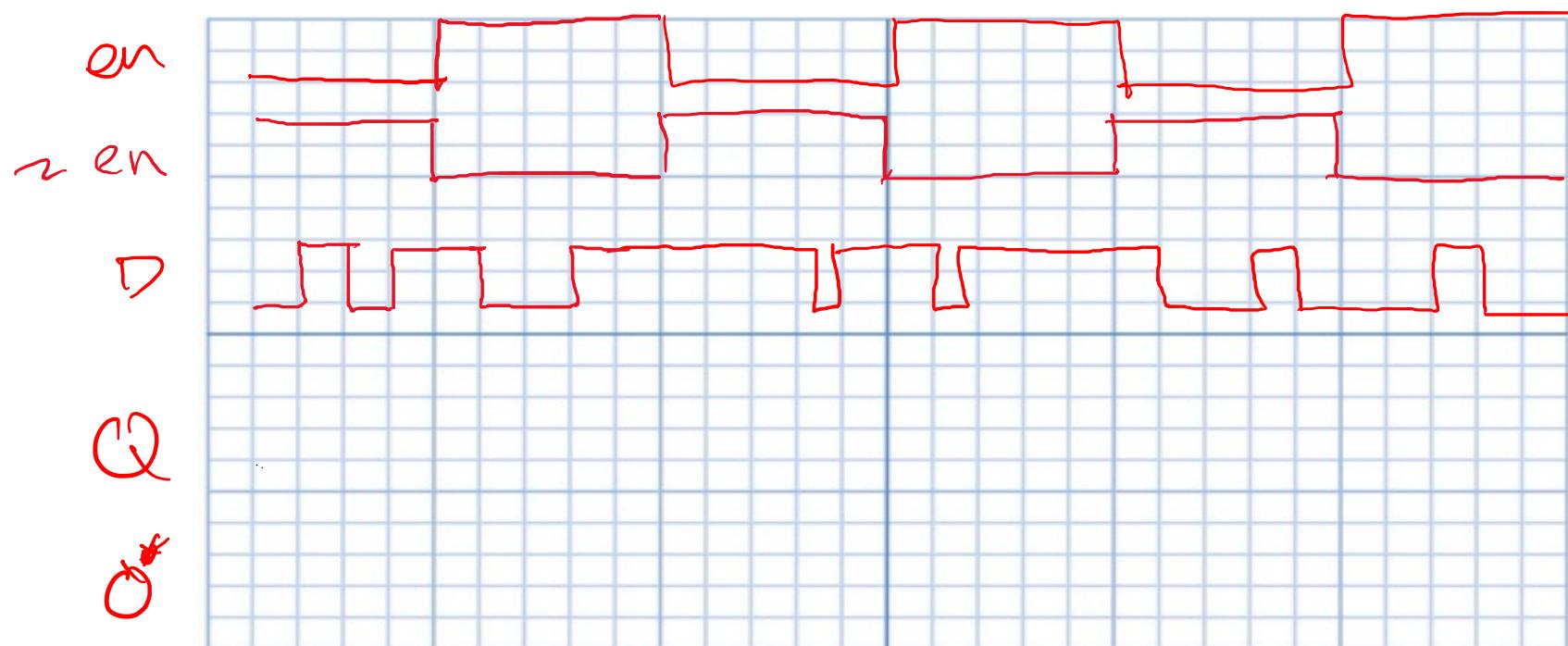
Always specify defaults for
always_comb!

Always specify
defaults for
always_comb!

D Flip-Flop



* no gate delays



Levels vs. Edges

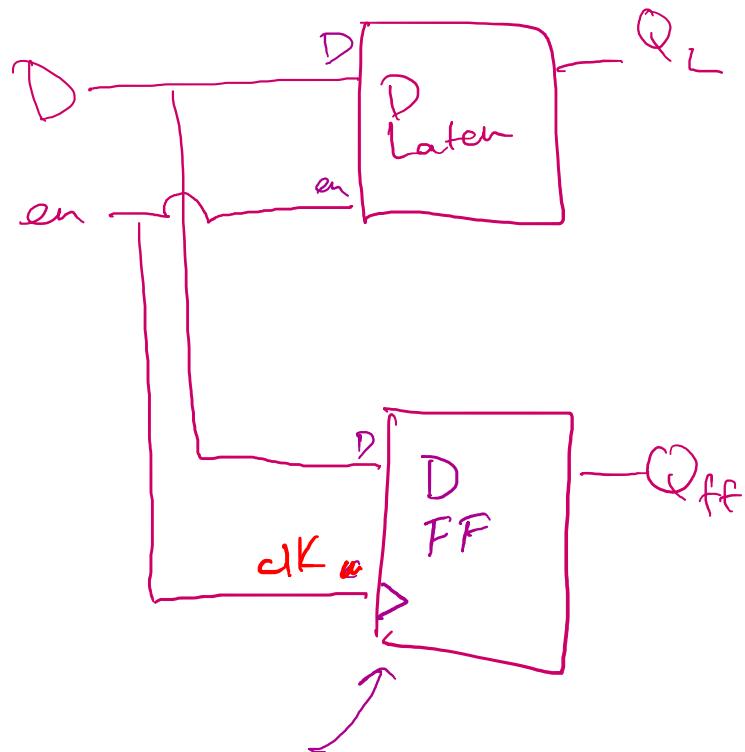
D latch \rightarrow Q follows D whenever enable is 1



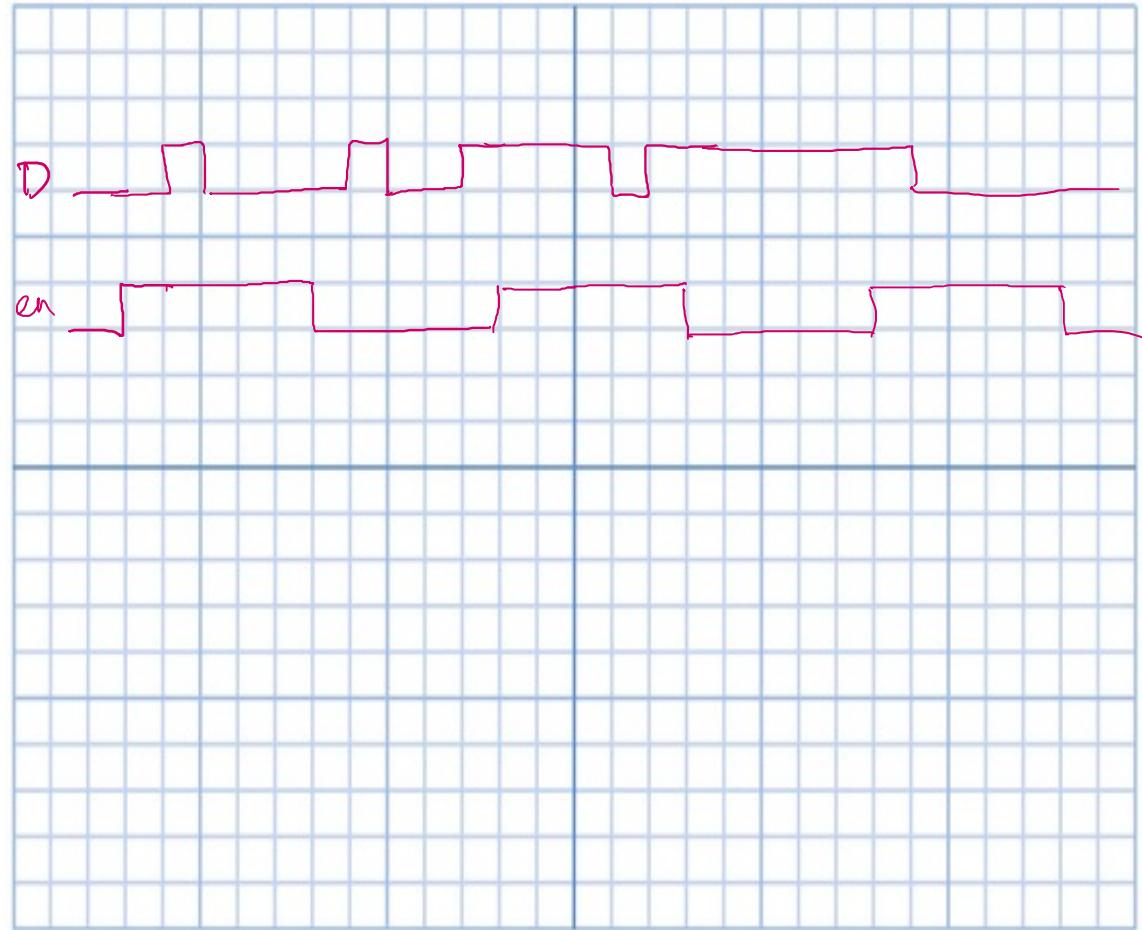
D flip-flop \rightarrow Q follows D on rising edge of enable



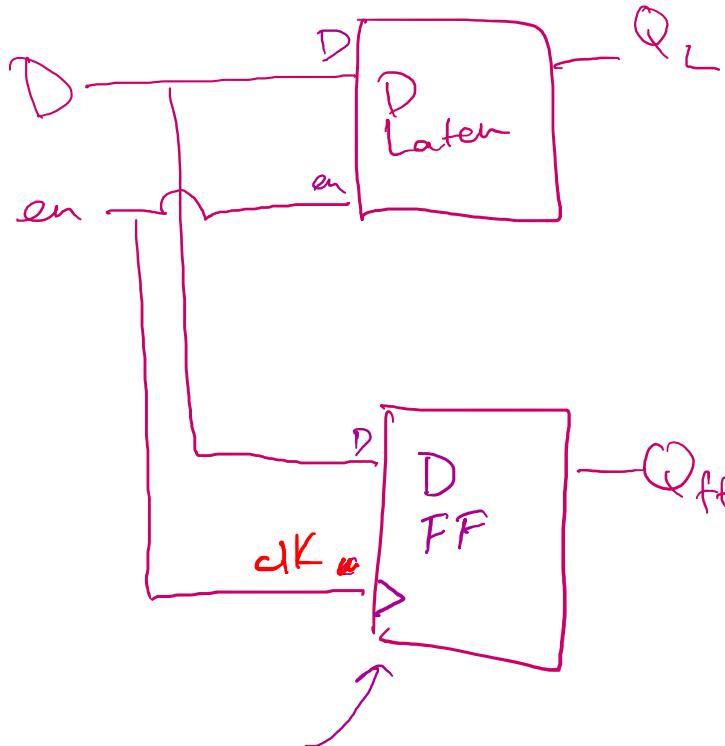
D Flip-Flop vs. D Latch



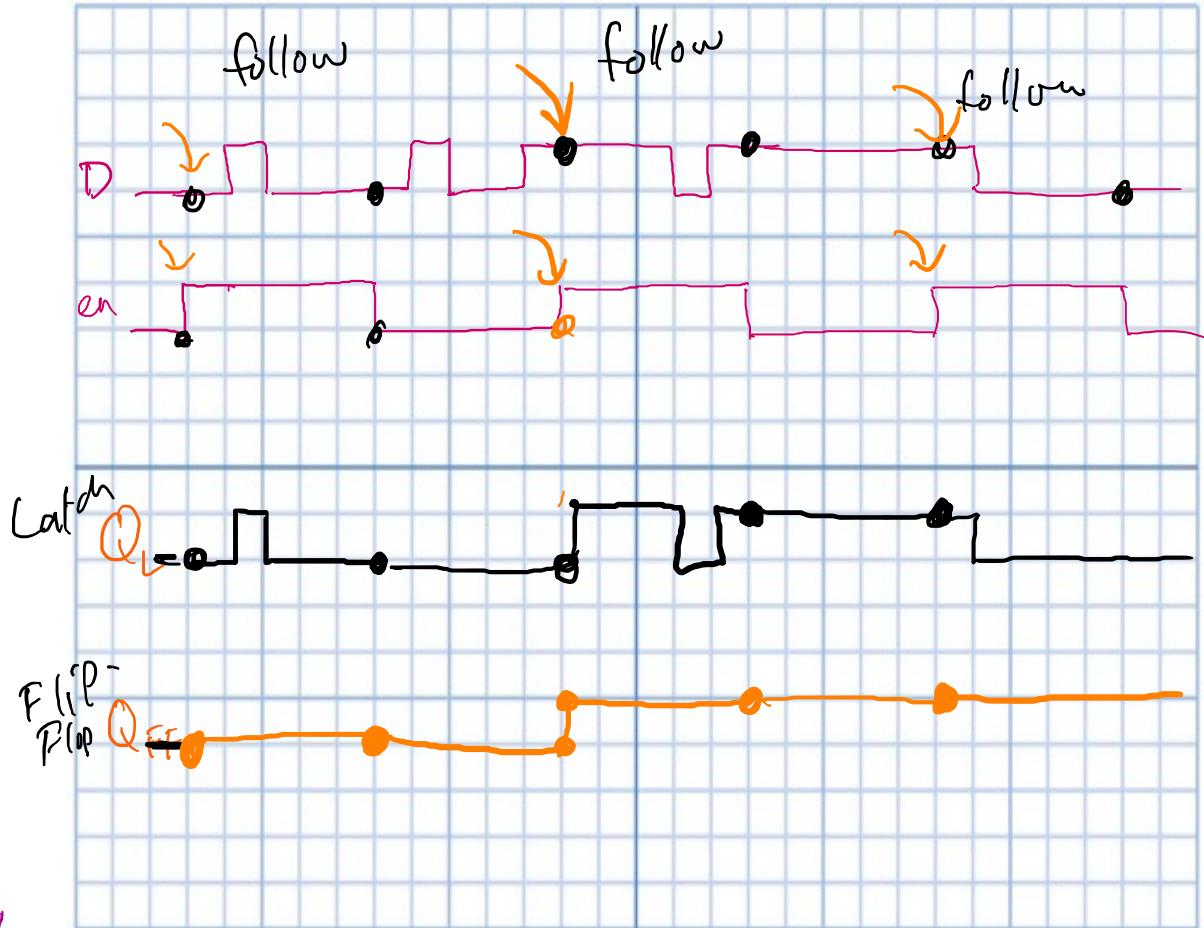
The " $>$ " symbol tells you
it is Flip-Flop



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) //pos-itive edge of enable
begin
    q <= d; //non-blocking assign
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
end

endmodule
```

D Flip-Flop w/ Clock

CLK100MHz



```
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
    end

endmodule
```

Blocking vs. NonBlocking Assignments

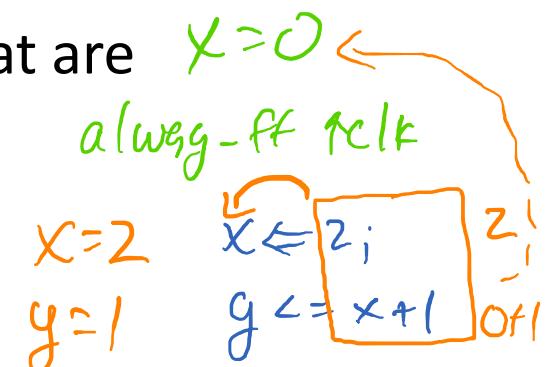
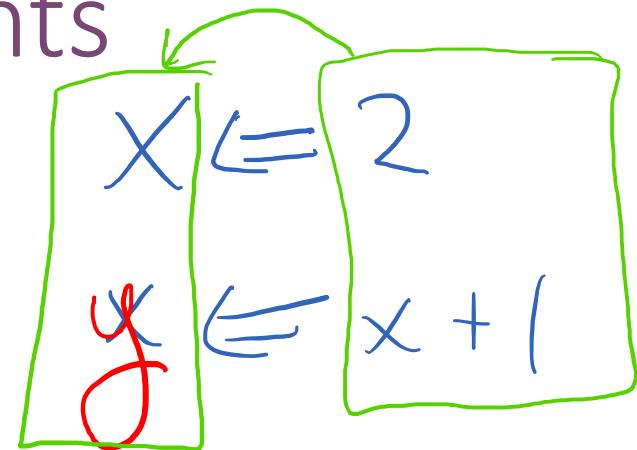
- Blocking Assignments (`=` in Verilog)
 - Execute in the order they are listed in a sequential block;
 - Upon execution, they immediately update the result of the assignment before the next statement can be executed.

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
    x = a + 1;
    y = x + 1;
    z = z + 1;
end
```

```
always_ff @ (posedge clk)
begin
    x <= a + 1;
    y <= x + 1;
    z <= z + 1;
end
```

Blocking vs. NonBlocking

```
always_comb
begin
    IS RHS
     $\rightarrow x = a + 1;$ 
     $\rightarrow y = x + 1;$ 
     $\rightarrow z = z + 1;$ 
bad in
always_comb
end
```

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

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

$$x=2, z=1$$

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

```
always_ff @ (posedge clk)
begin
     $x <= a + 1;$ 
     $y <= x + 1;$ 
     $z <= z + 1;$ 
end
```

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

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

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

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

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

NON-BLOCKING (<=) for

always_ff

D-FlipFlop w/Clock

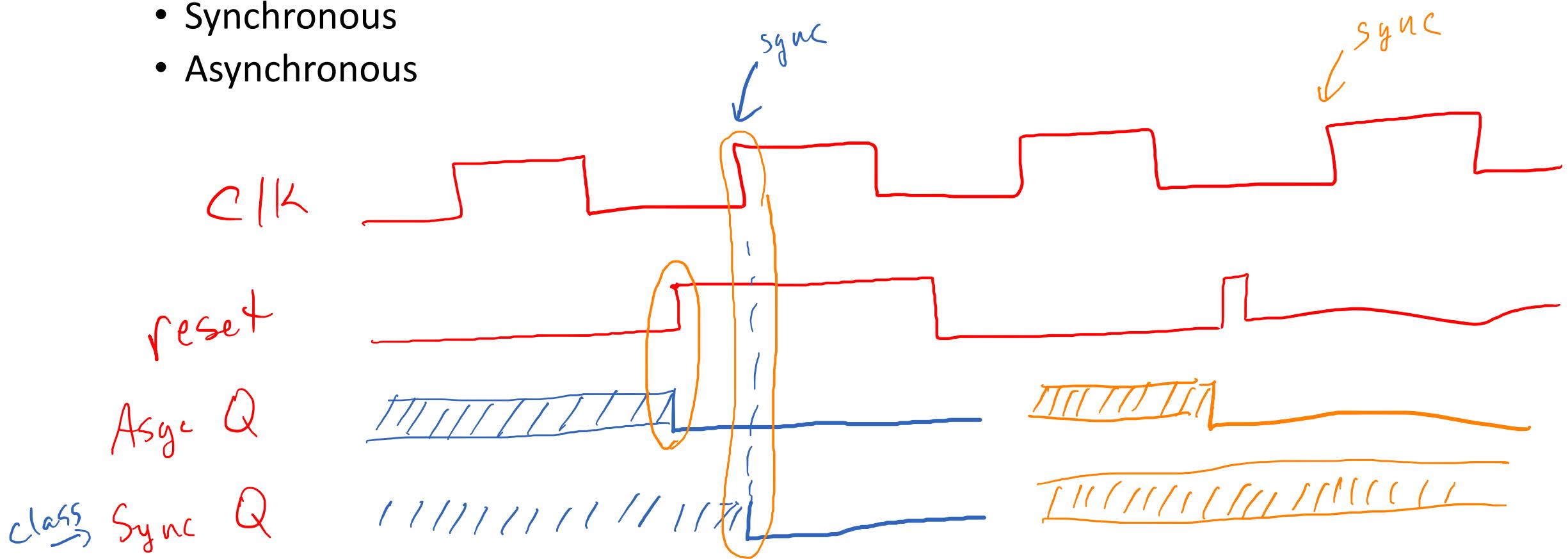
$q \rightarrow d \rightarrow q_{\text{new}} \rightarrow d_{\text{new}} \rightarrow q_{\text{new}_2}$

```
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

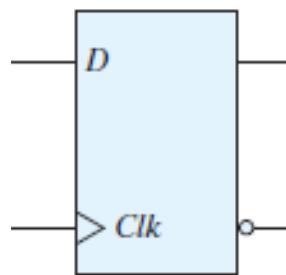
- Two different ways to build in a reset
 - Synchronous
 - Asynchronous



D-FF's with Reset

- Two different ways to build in a reset
 - Synchronous
 - Asynchronous
- We always use synchronous resets for this class!

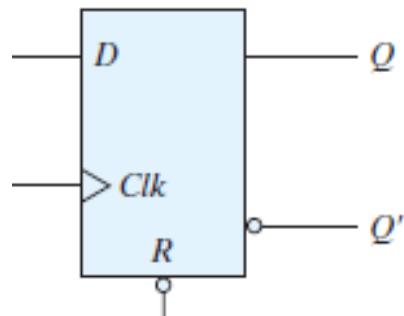
Verilog models of D flip-flop



Edge triggered D flip-flop:

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

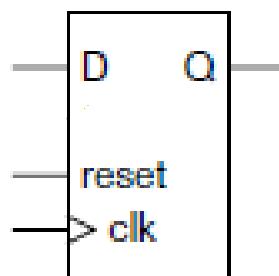
No reset
FF



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;
```

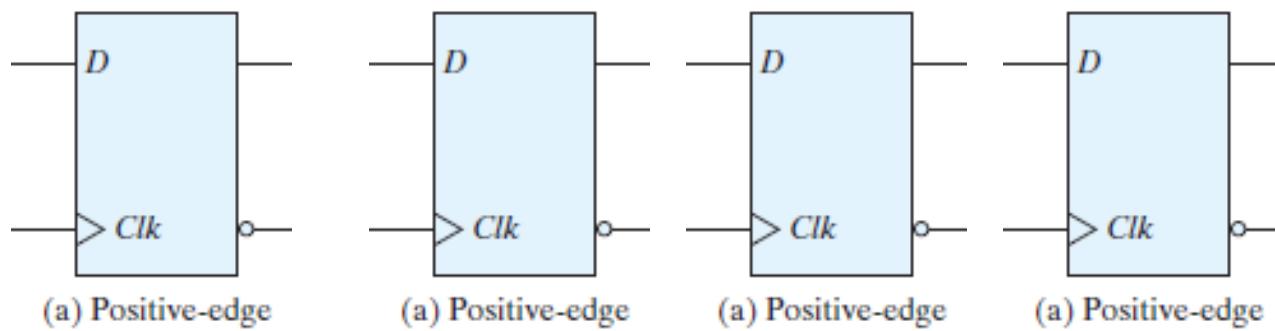
Not used
in class



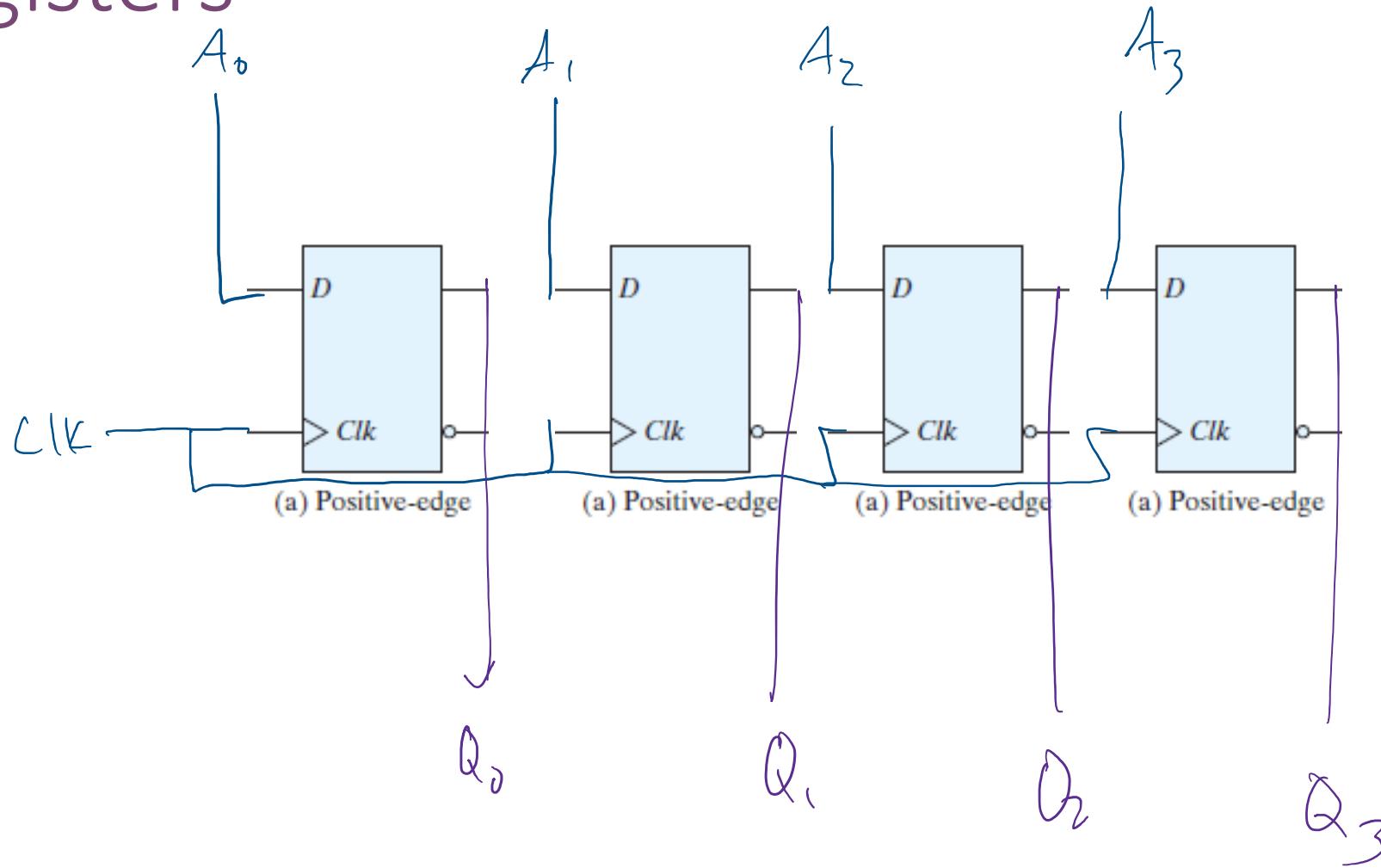
Edge triggered, synchronous reset, clock enable D flip-flop: C

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

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?

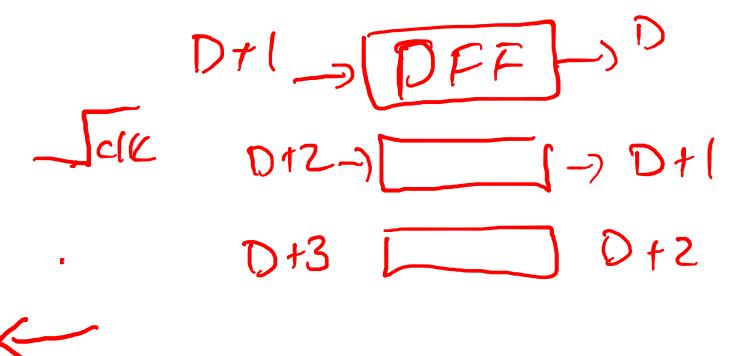
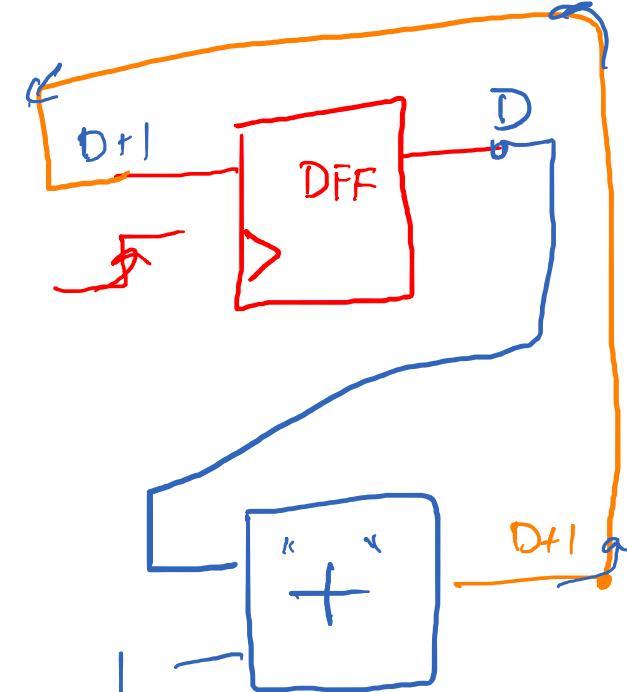
```

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

```

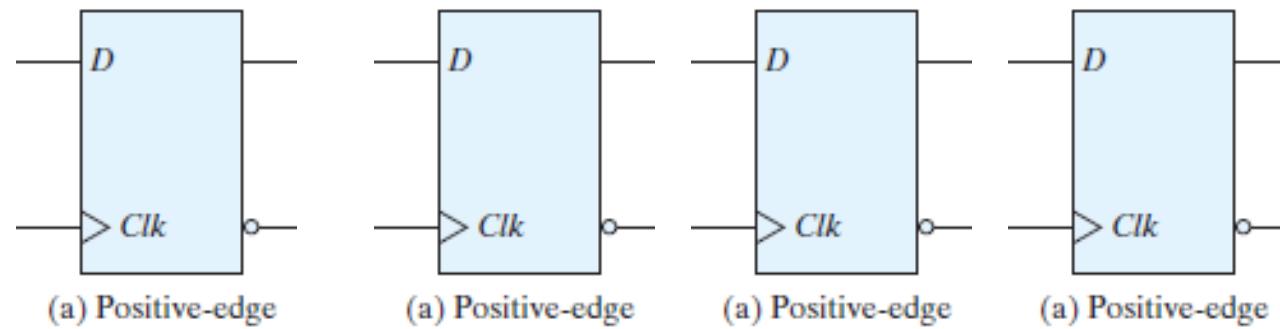
Annotations:

- 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. A red arrow points from this waveform to the "posedge clk" condition in the always_ff block.
- Reset:** A red arrow points from a state diagram (00000) to the "rst" input. The state diagram shows a 5-bit register with bits 0, 1, 2, 3, 4 all set to 0.
- Output:** A blue waveform labeled "out" shows a square wave. A red arrow points from this waveform to the "out" output.
- Data Flow:** A red arrow points from the "sum" wire to the assignment in the always_comb block: $\{out, sum\} = \{0, D\} + 5'h1;$. The "sum" wire is highlighted in red.
- Bit Widths:** Red annotations indicate bit widths: "1bit 4bit" under the first assignment, "5bit + 5bit" under the second assignment, and "5bit" under the output waveform.



Start here!

D Flip-Flops as Shift Registers



Shift Register

