Latches +
Flip Flops

Andrew Lukefahr

`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?

Assume 4-bit addition

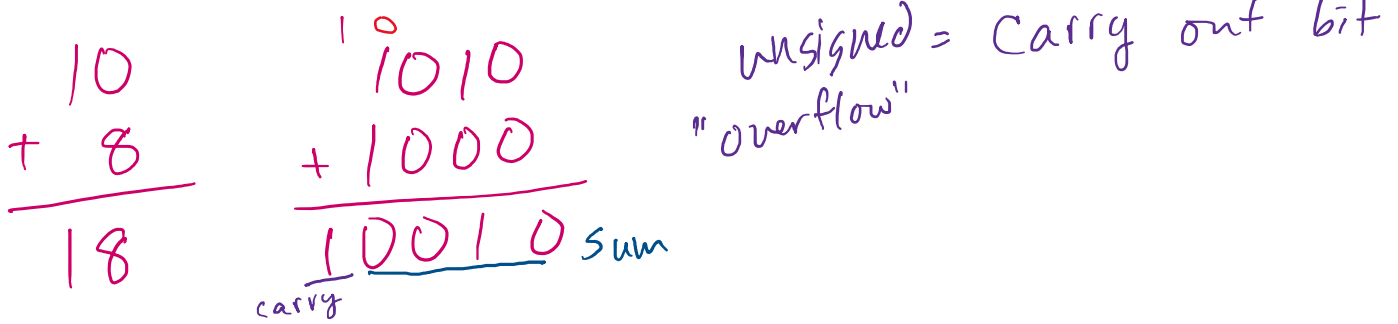
- Unsigned

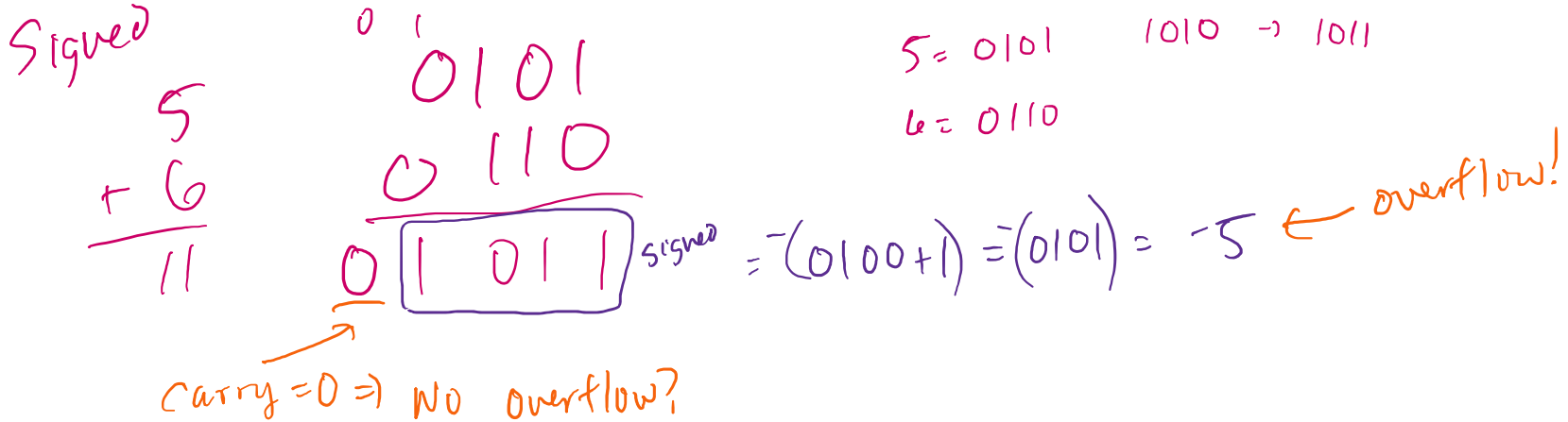
$$\begin{array}{r} 10 \\ +\ 8 \\ \hline 18 \end{array}$$

- Signed

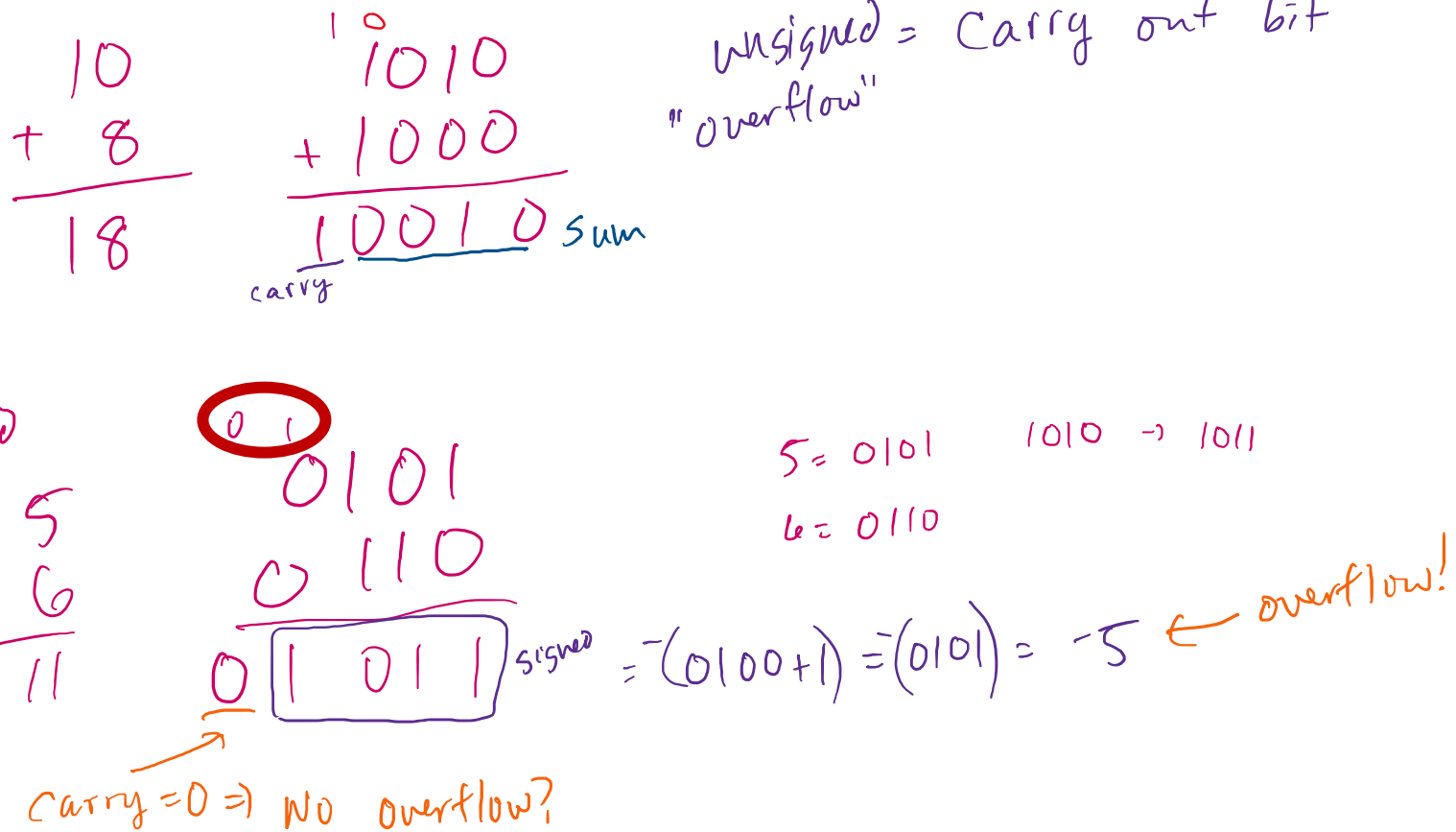
$$\begin{array}{r} 5 \\ +\ 6 \\ \hline 11 \end{array}$$

Overflow for signed numbers?

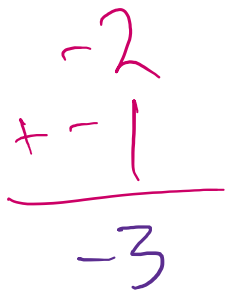




Overflow for signed numbers?

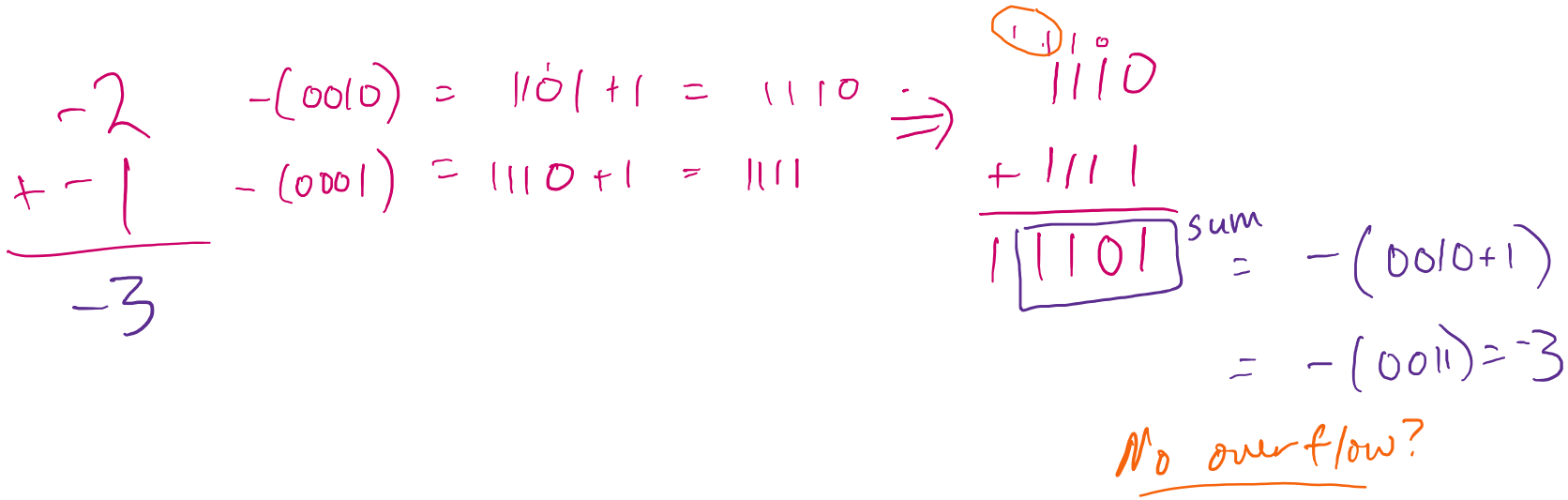


Overflow for signed numbers?




```
   7
+  7
-------
 +14  ?
```

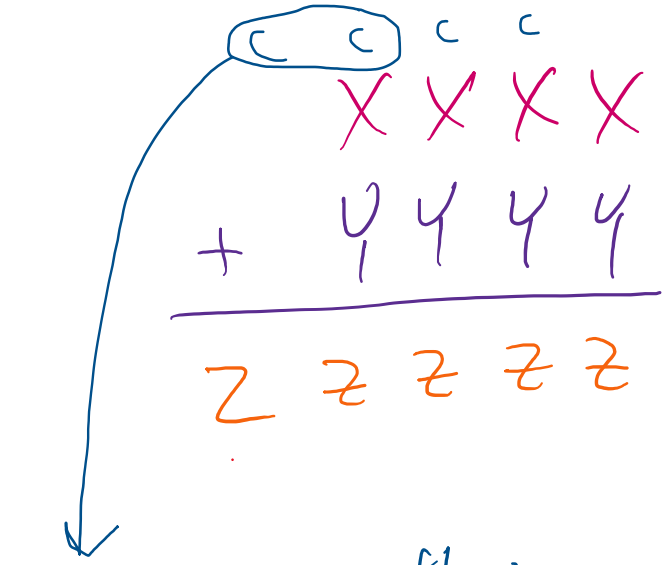
Overflow for signed numbers?




```
                    0 1 0 0 0 <- overflow
   7       0100     00100
+  7    + 0100   + 00100
-----   ------     ------
   +1              01000
```

Overflow for signed numbers

Stopped here!



Same = no overflow

different = overflow

$\Downarrow$

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.

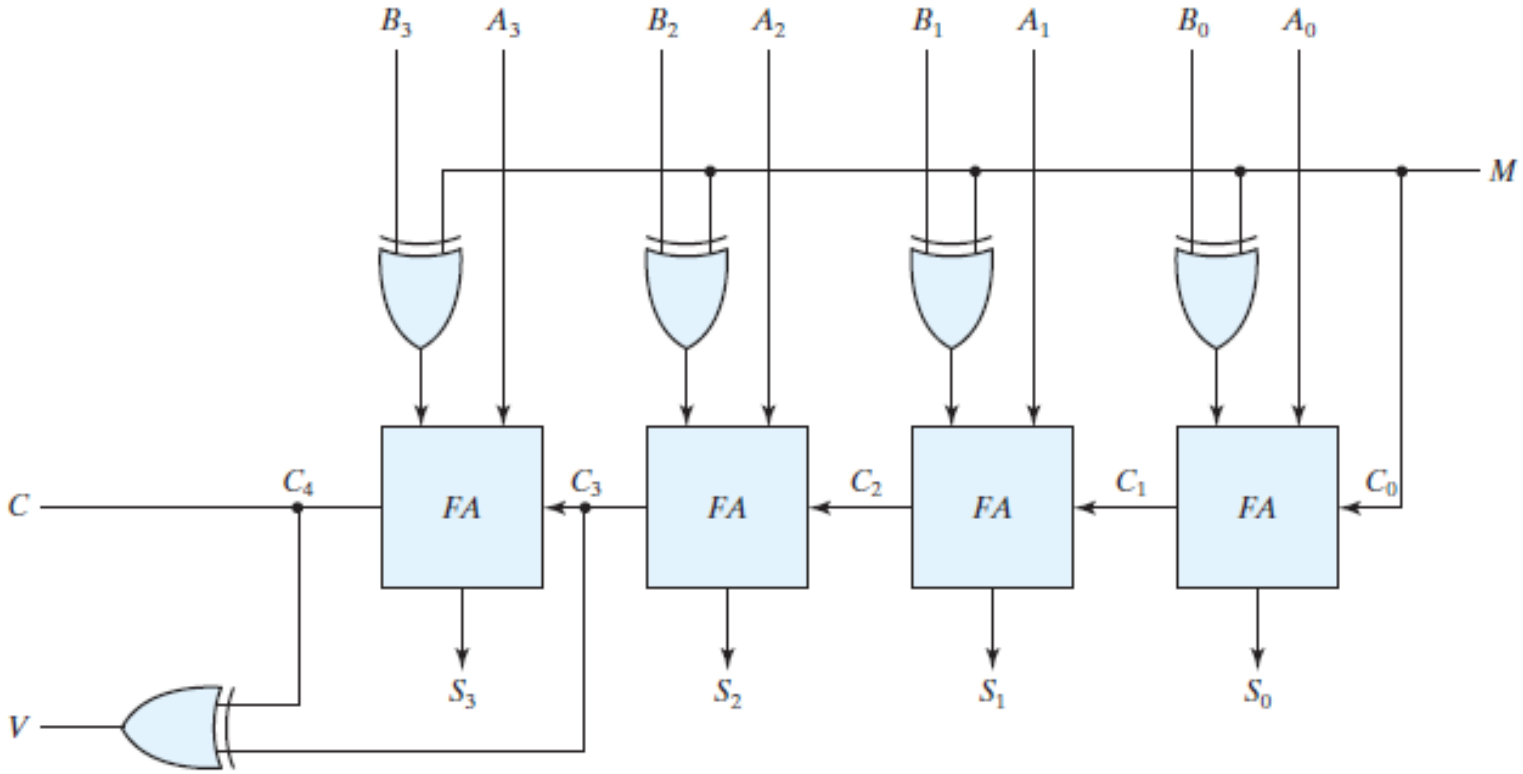
Overflow detection

- 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.

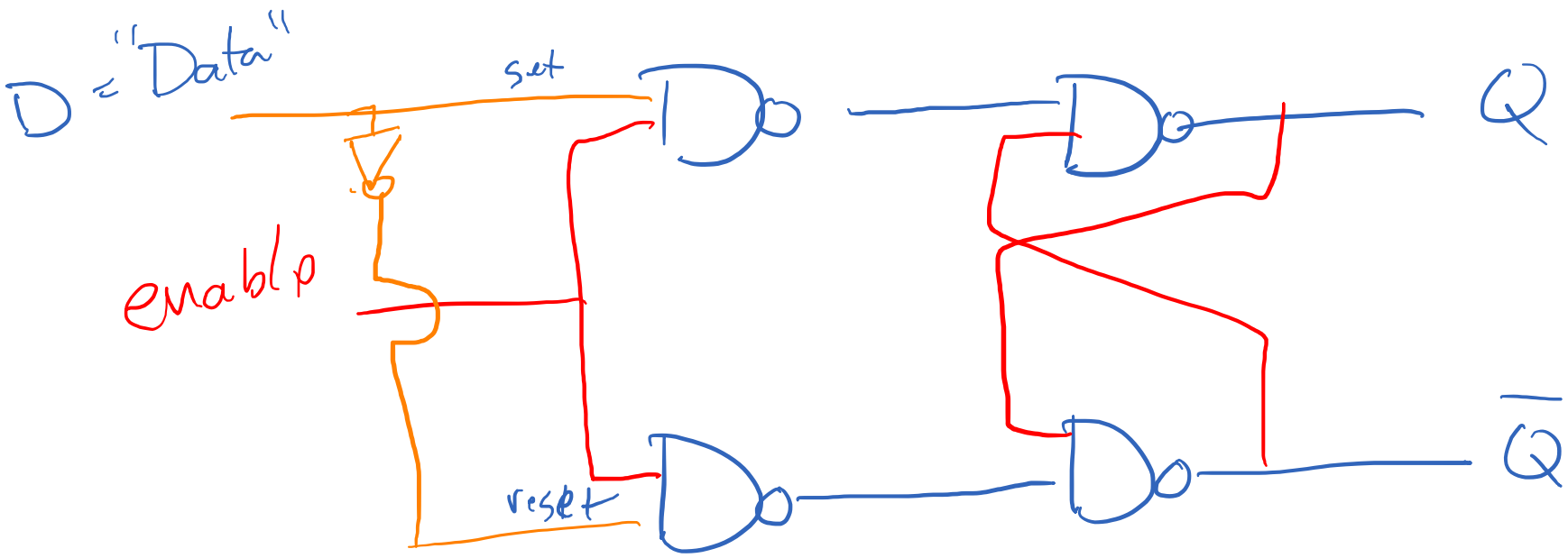
  If these two carries are not equal, an overflow has occurred.
- If the two carries are applied to an exclusive-OR gate, an overflow is detected when the output of the gate is equal to 1.

Adder with overflow detection



"Data" Latch

D-Latch

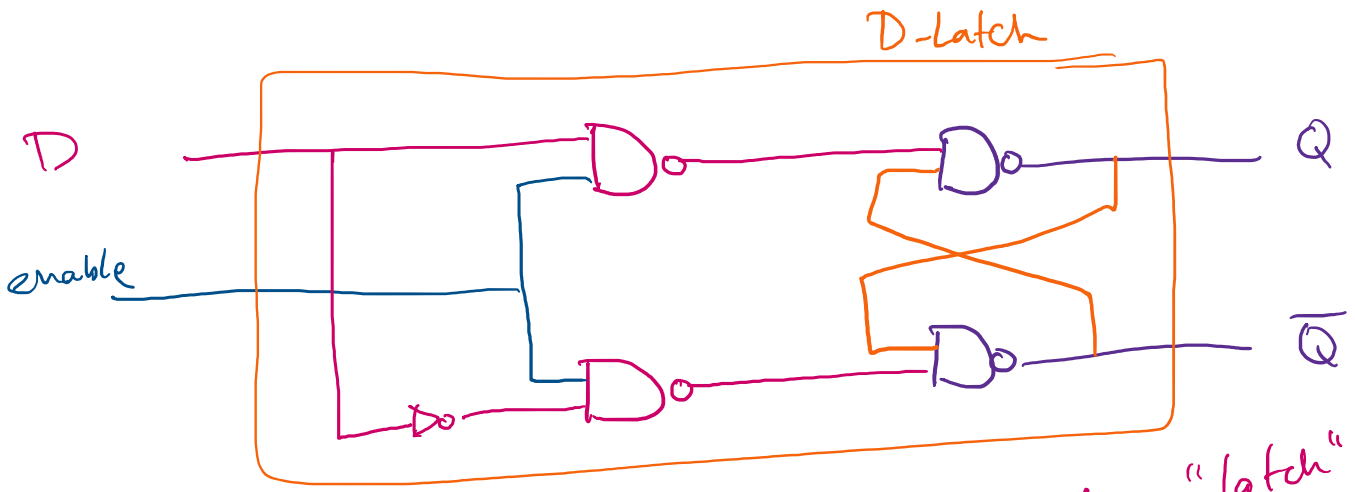


D-Latch

"Data" Latch

= 1

= 0

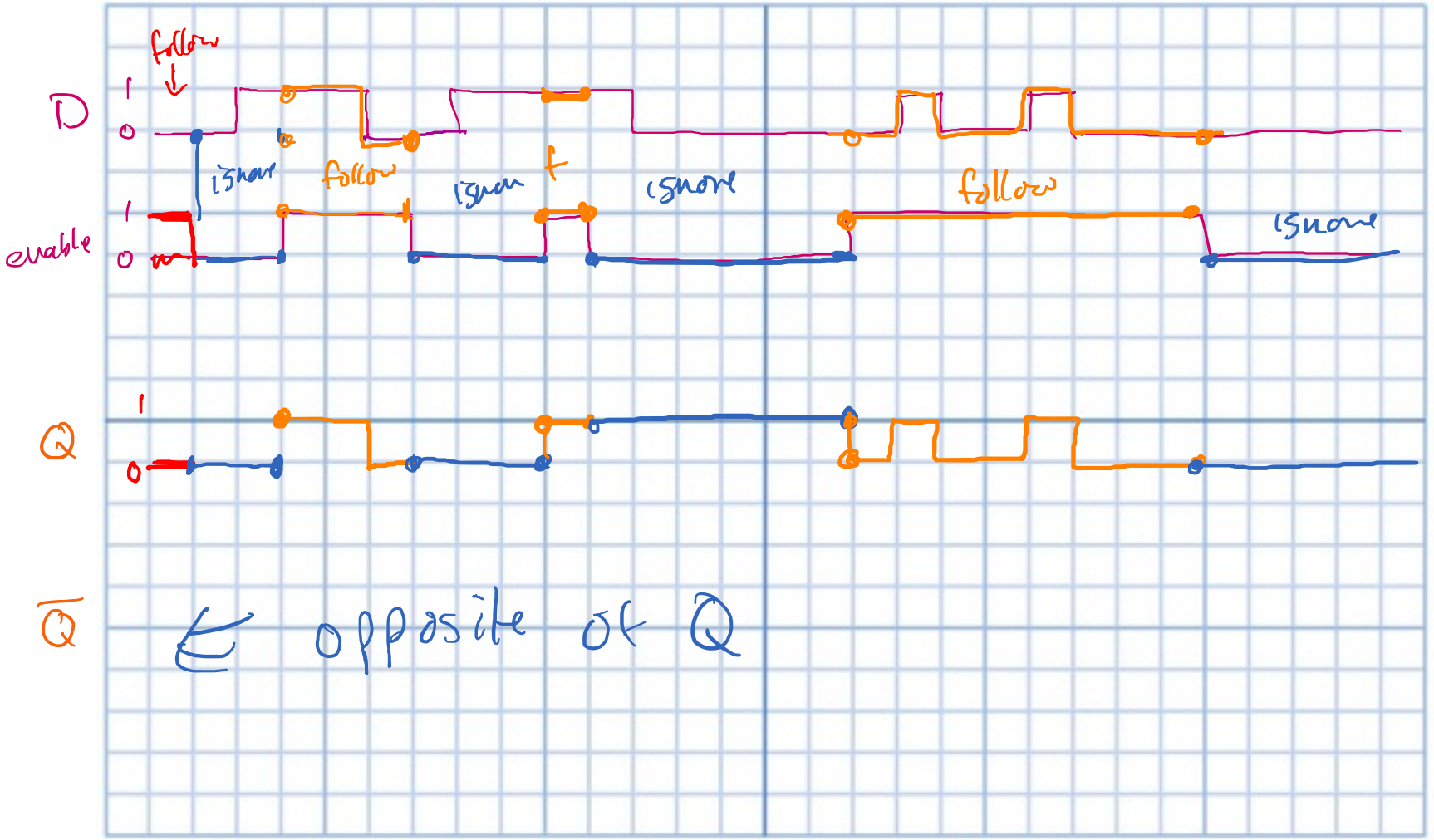


D	en	Q	$\overline{Q}$
0	0	Q	$\overline{Q}$
1	0	Q	$\overline{Q}$
0	1	0	1
1	1	1	0

← circuit to "latch" a value

* assume negligable gate delays

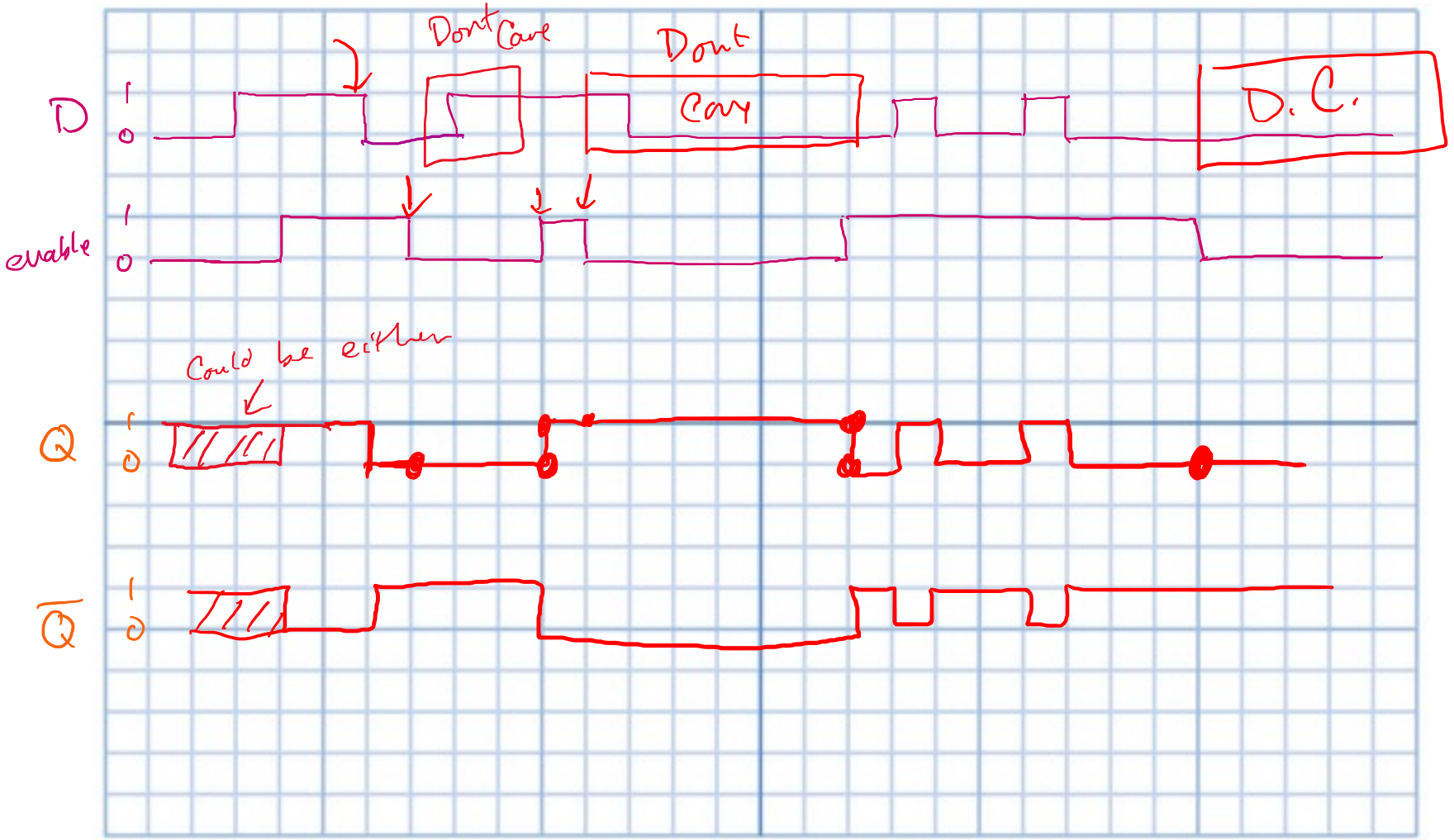
Inputs to D Latches



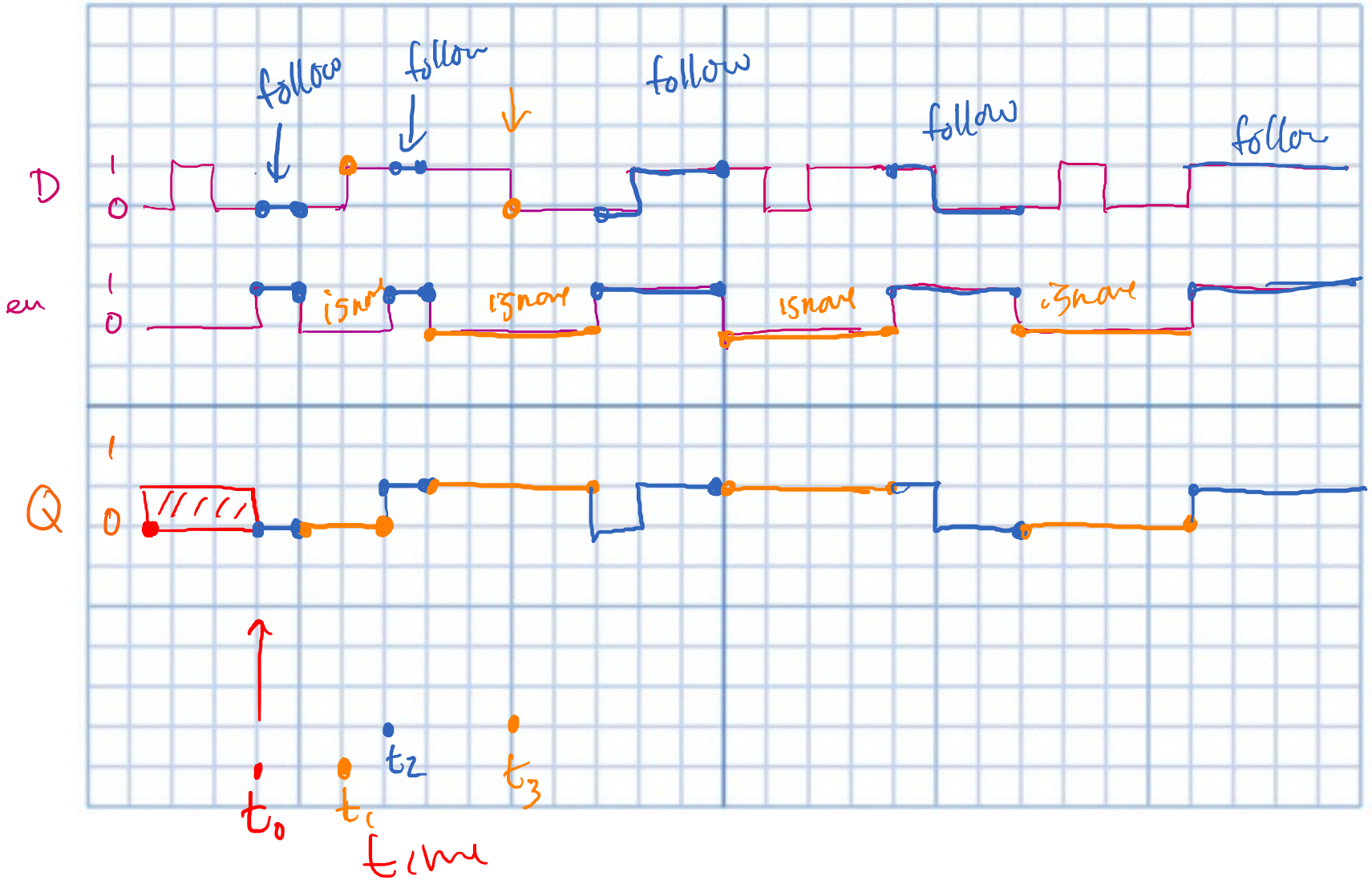
Q follows D when enable = 1, otherwise Q ignores D.

Inputs to D Latches

* assume negligable gate delays



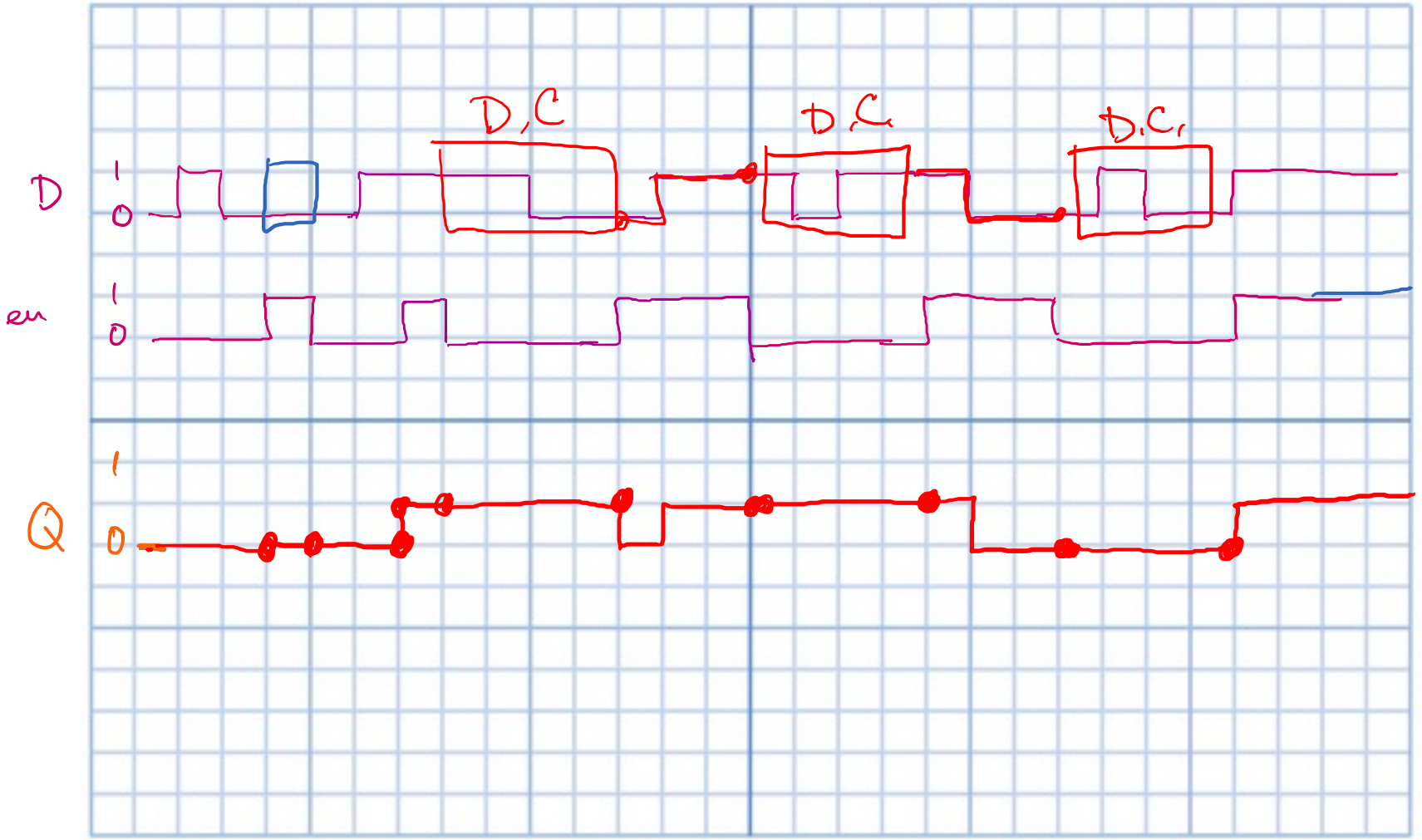
Inputs to D Latches



Inputs to D Latches

if en = 1, Q = D

if en = 0, Q = Q

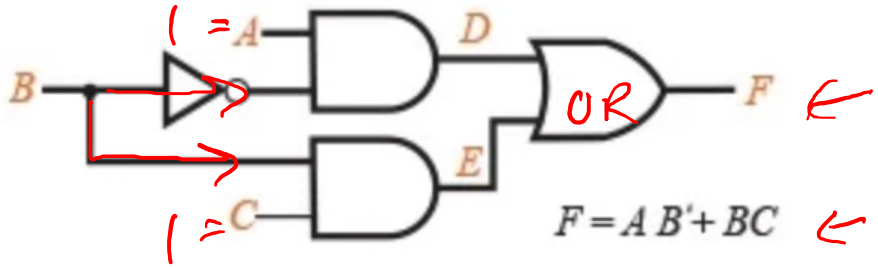


Glitches

→ unintended, short, errors in boolean logic

→ caused by gate delays

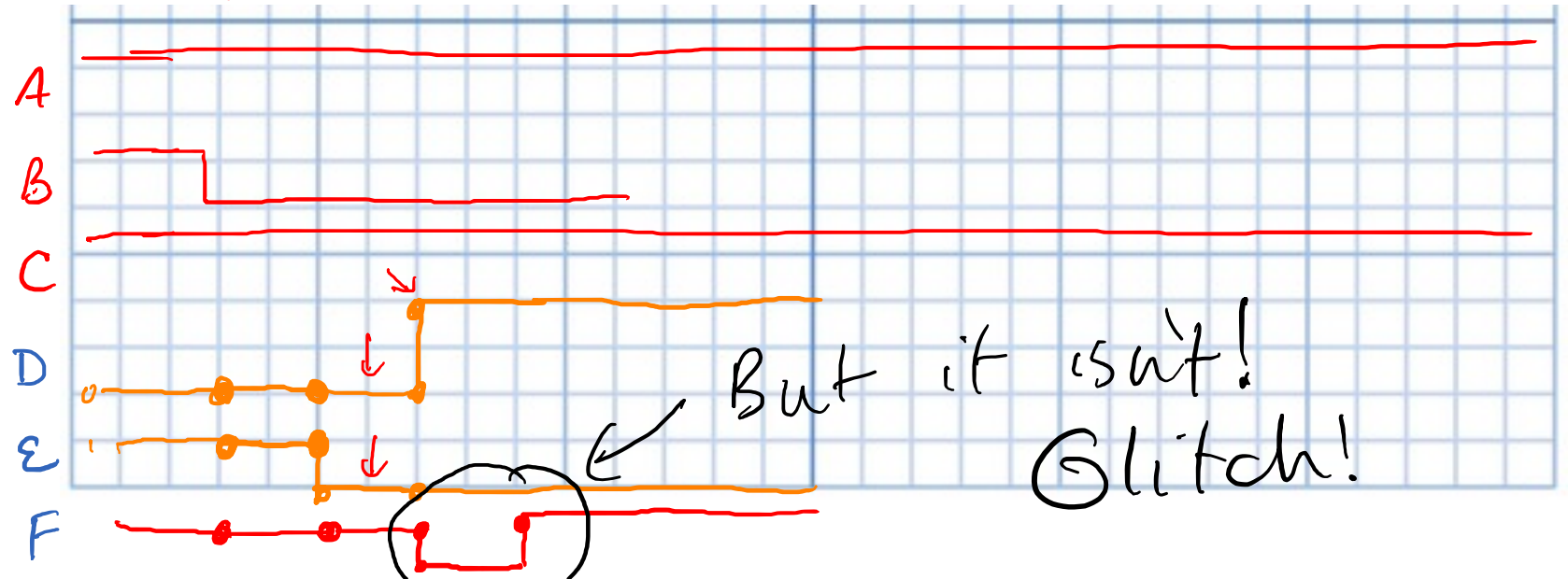
- Assume 10ps / gate.
- A=1, C=1, B falls
- What is F?



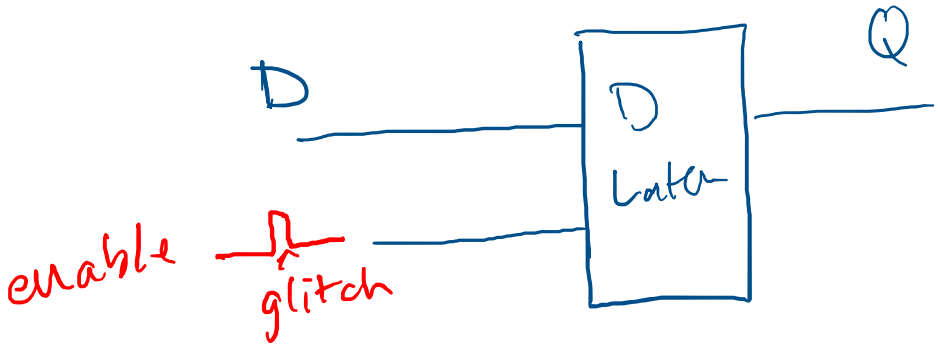
← should always be true!

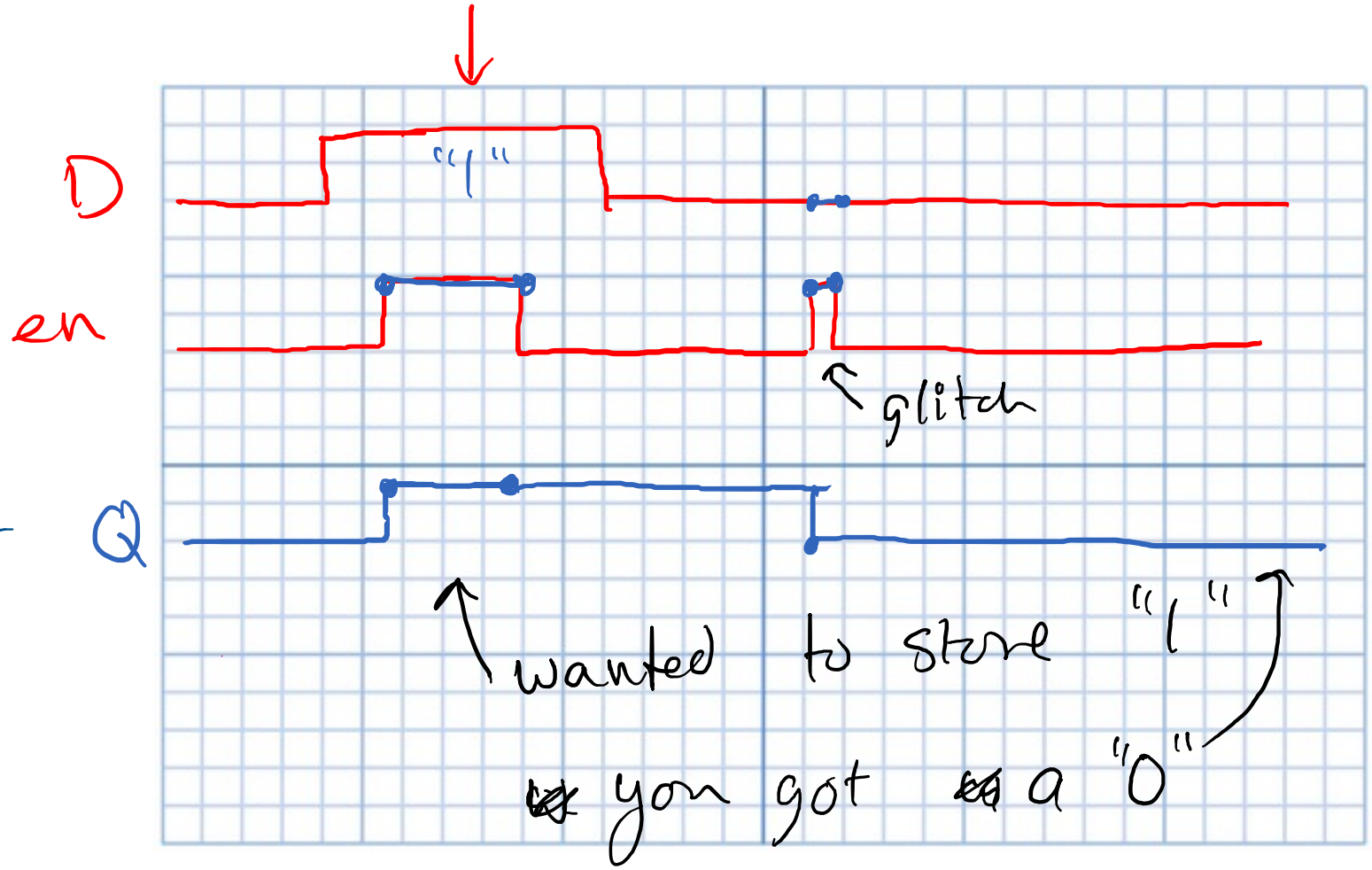
A=1 B=1 C=1

10ps



Glitches on D-Latches





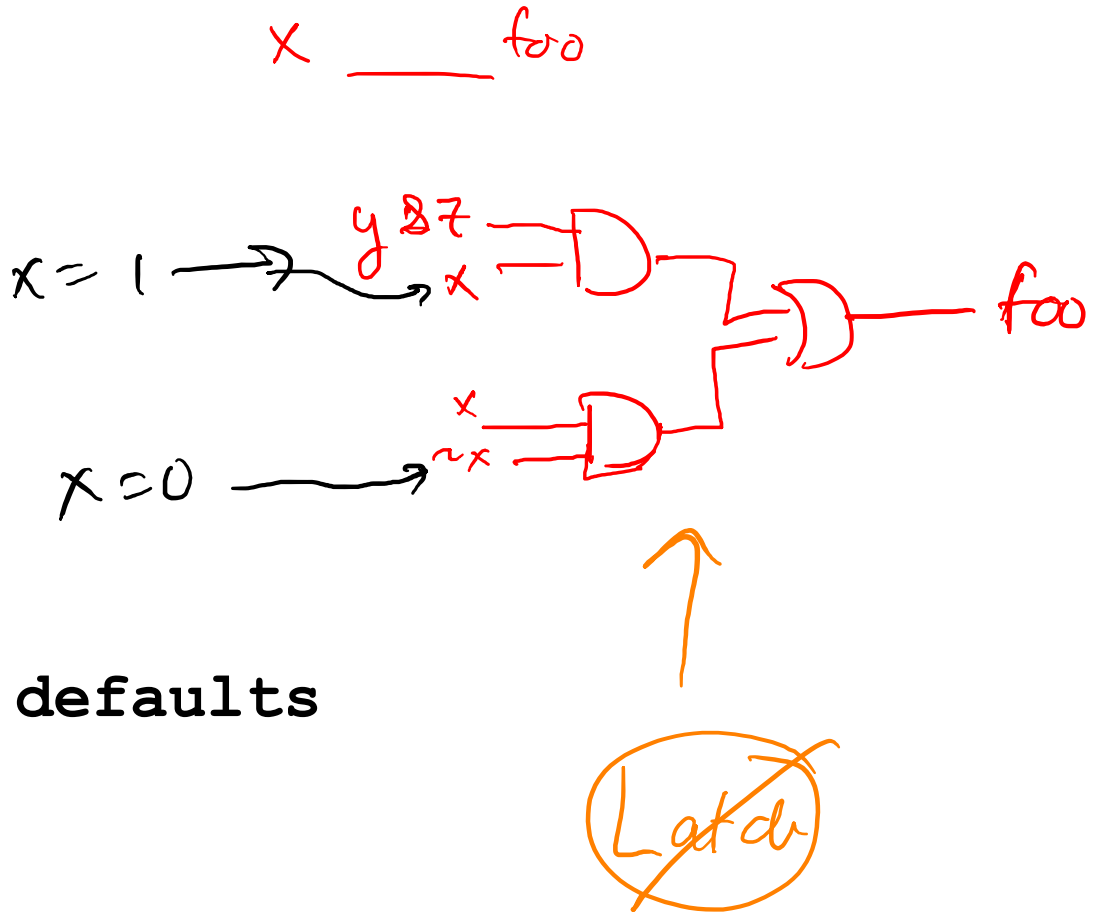
What's wrong here?

```
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`? `foo = bar = x`!

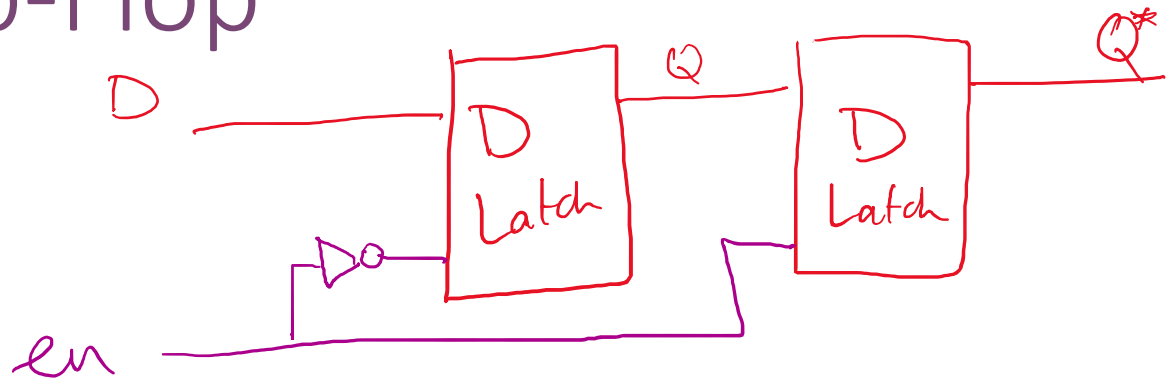
Always specify defaults for **`always_comb`**!

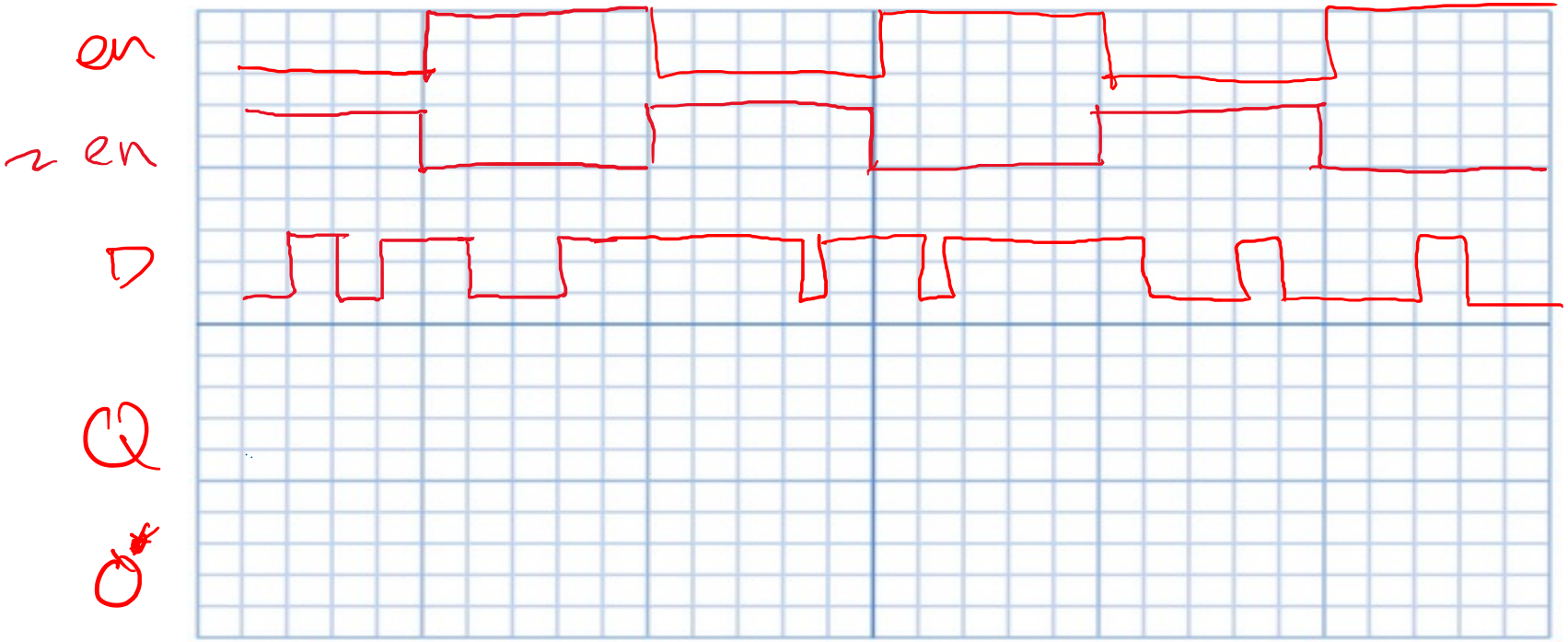
Always specify defaults for **`always_comb`**!

Always specify defaults for `always_comb`!

D Flip-Flop

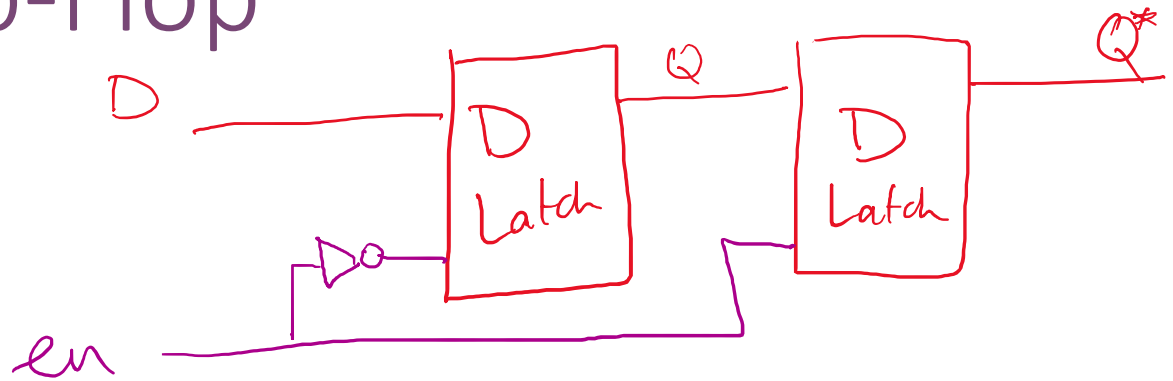
* no gate delays

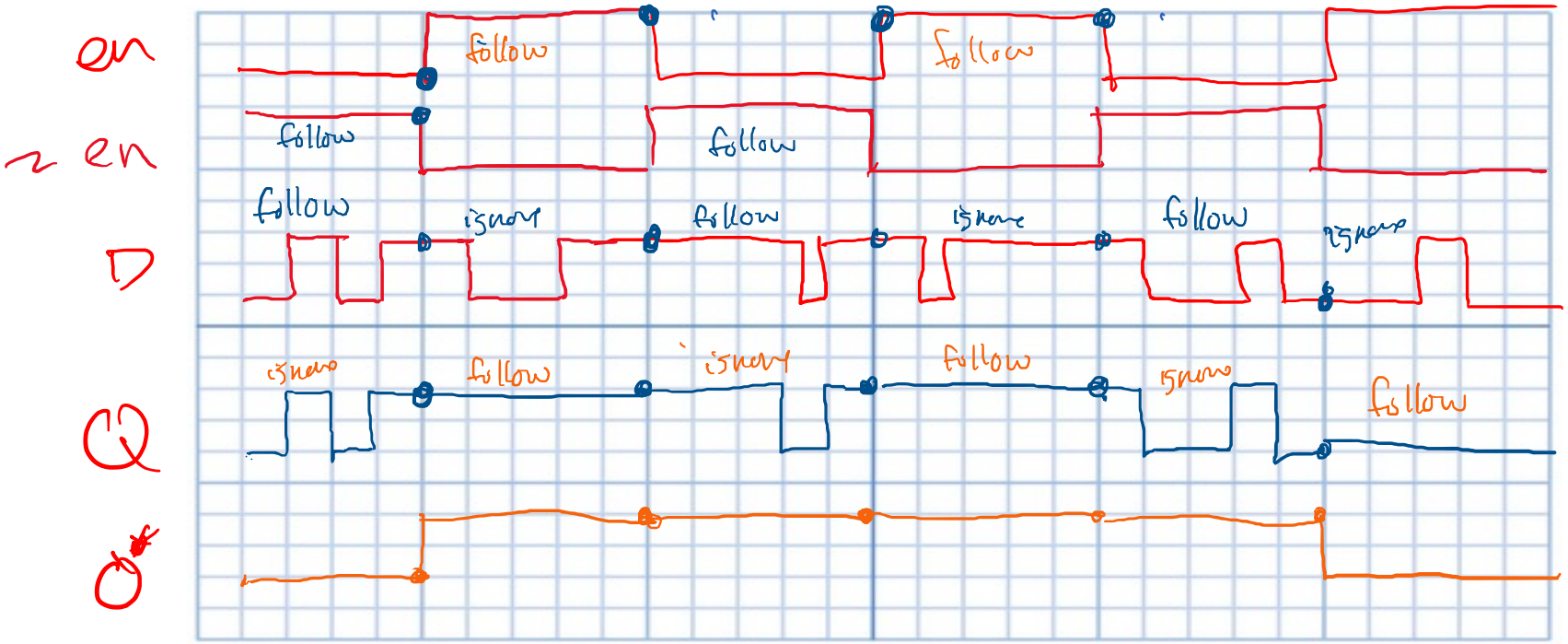




D Flip-Flop

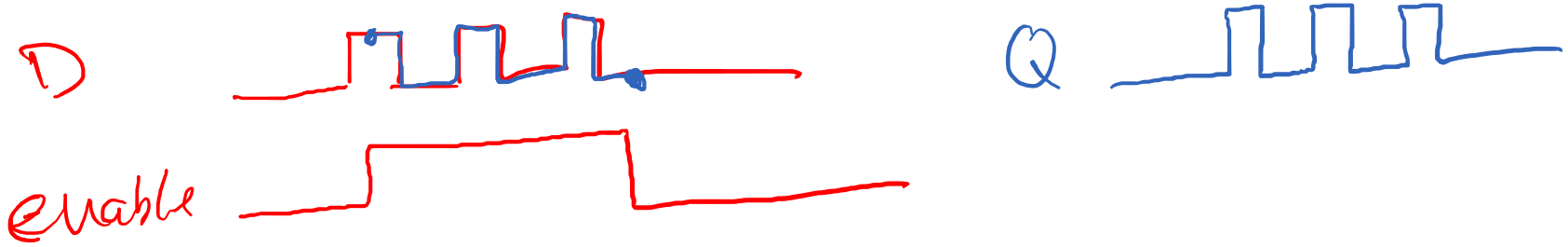
* no gate delays





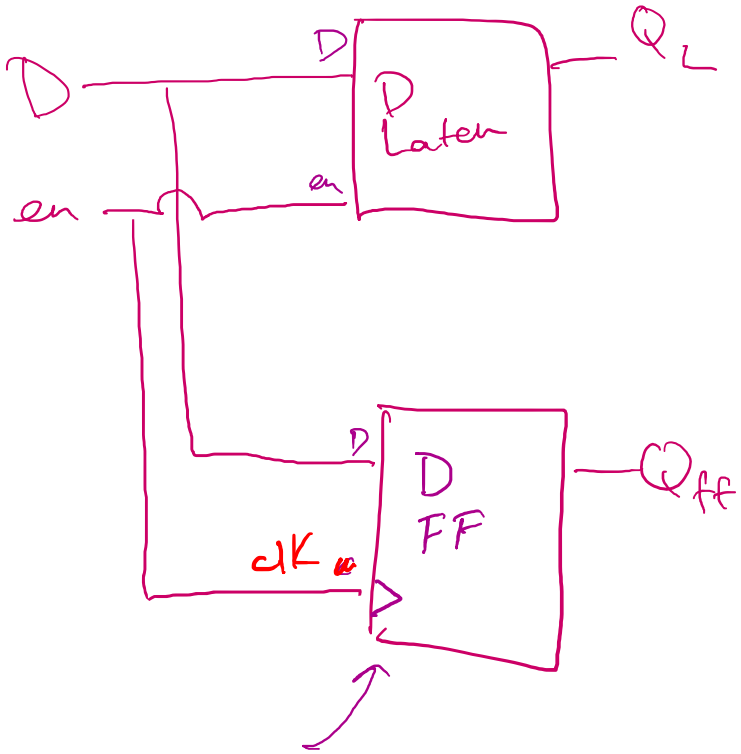
Levels vs. Edges

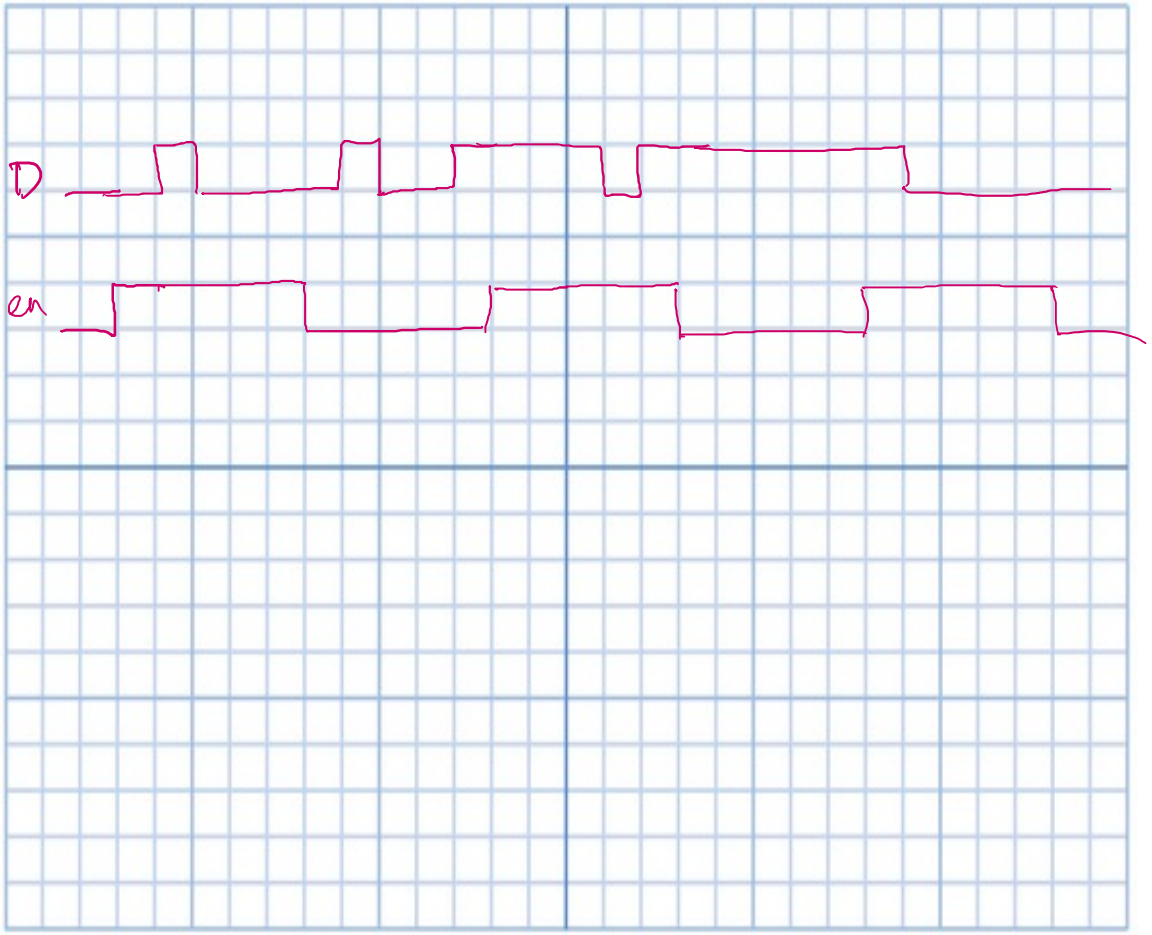
D Latch → Q follows D whenever enable is 1



D flip-flop → Q follows D on rising edge of enable

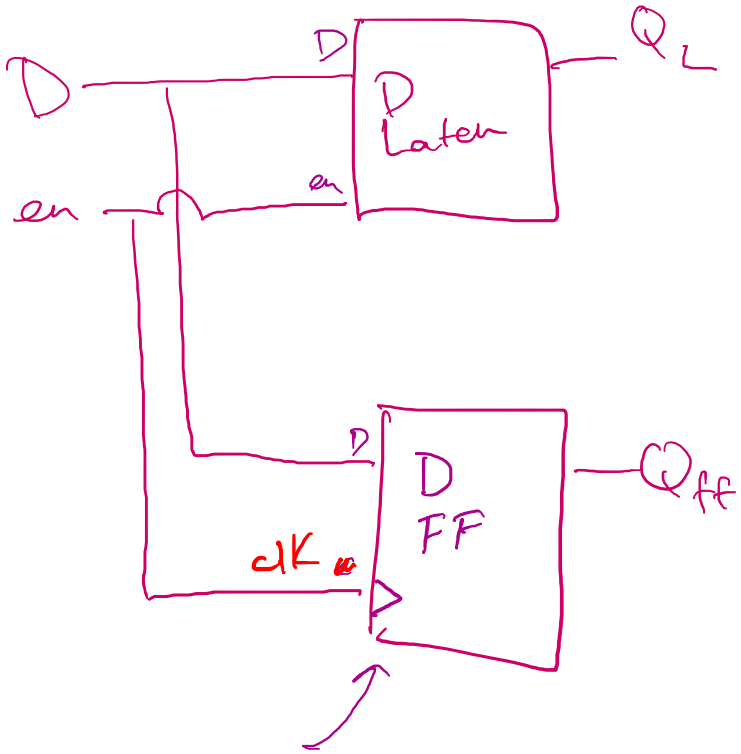
D Flip-Flop vs. D Latch

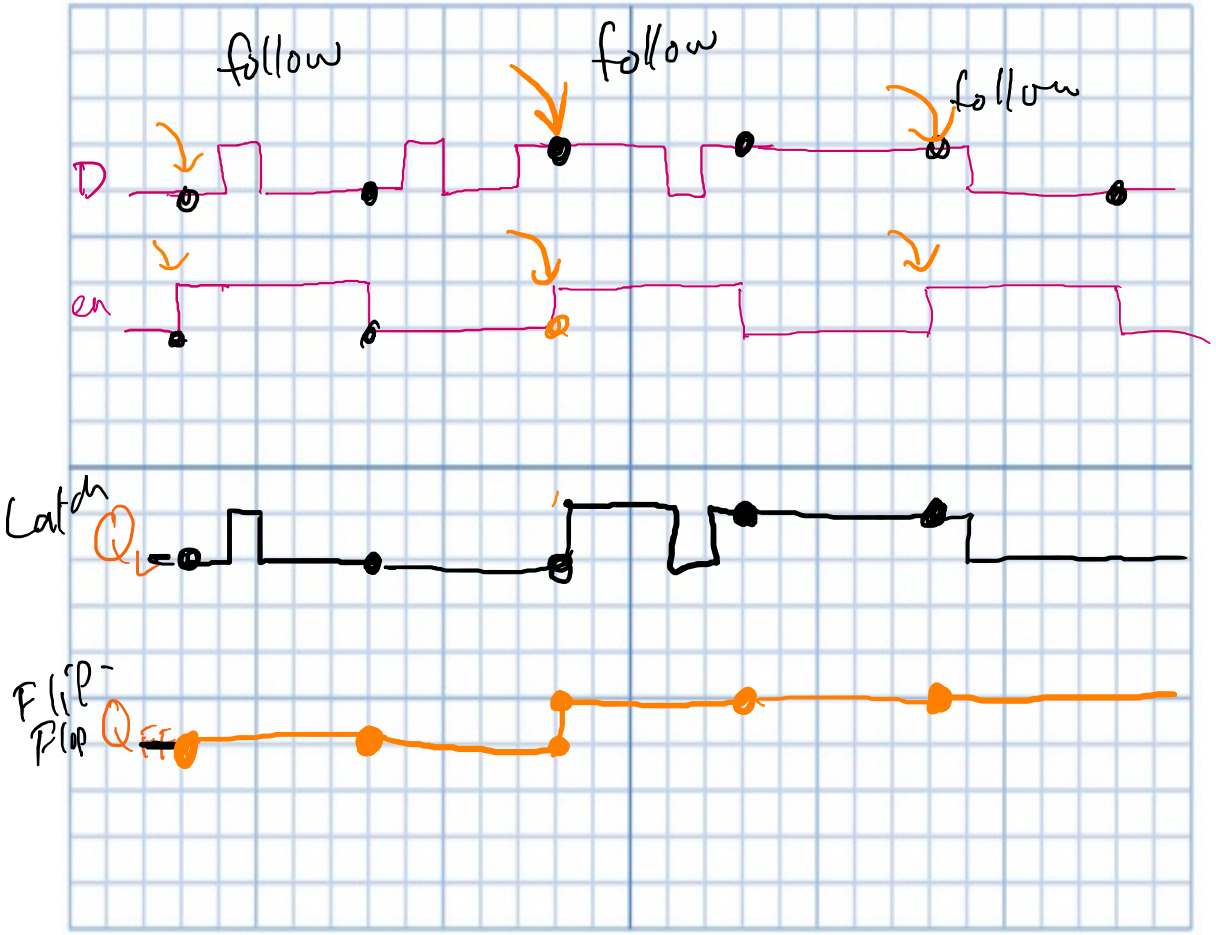




the " > " symbol tells you it is Flip-Flop

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 en-able
    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
```

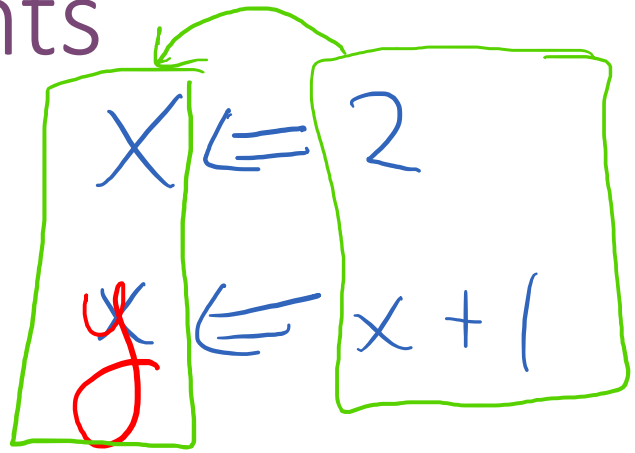
D Flip-Flop w/ Clock

```
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
```

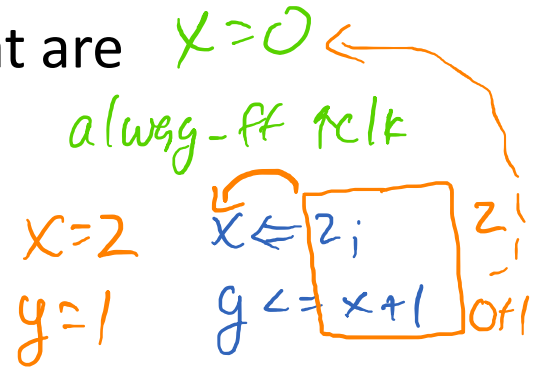
Blocking vs. NonBlocking Assignments

- 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.

Blocking vs. NonBlocking Assignments



- Non-blocking assignments (<= 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
    x = a + 1;
    y = x + 1;
    z = z + 1;
end
```

bad in always_comb

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

a=1 x = 1+1 = 2 ⇐
y = 2+1 = 3
z = 0+1 = 1 ⇐

x=2, z=1 x=2;
y=3;
z = 1+1 = 2

x=2
y=3
z=2+1=3

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

1+1 0+1 0+1

2+1=3 1+1=2

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

a=1, clk↑

x=2
y=1
z=1

clk↑

x=2
y=3
z=2

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

$q \rightarrow d \rightarrow q_{new} \rightarrow d_{new} \rightarrow q_{new_2}$

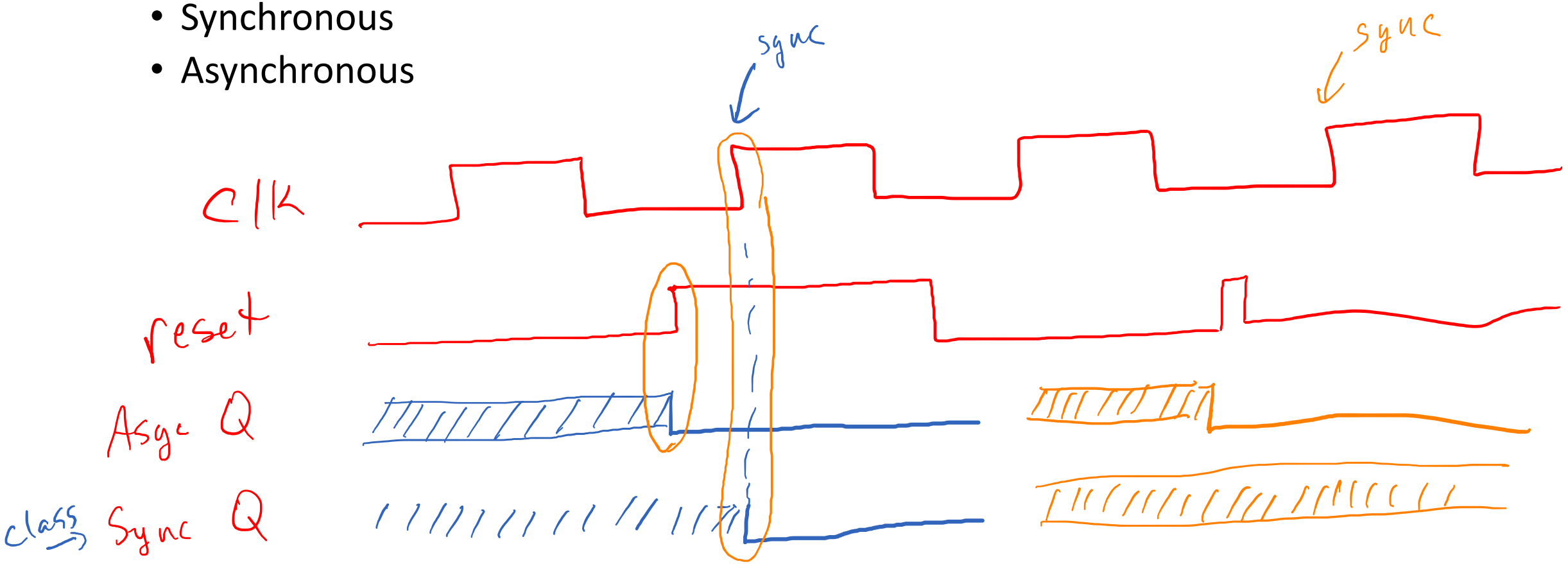
```
module d_ff (
     input d,              //data
     input clk,            //clock
     output reg q          //reg-isters hold state
);

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

endmodule
```

logic

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!

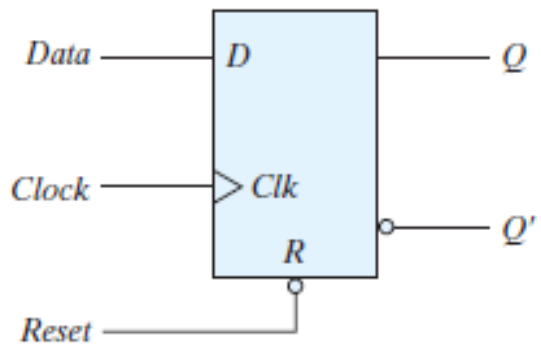
Direct inputs

Some flip-flops have asynchronous inputs that are used to force the flip-flop to a particular state independently of the clock.

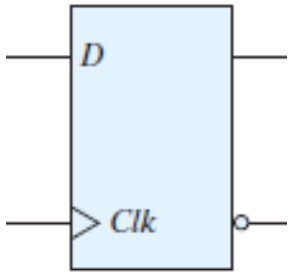
The input that sets the flip-flop to 1 is called *preset* or *direct set* .

The input that clears the flip-flop to 0 is called *clear* or *direct reset* .

When power is turned on in a digital system, the state of the flip-flops is unknown. The direct inputs are useful for bringing all flip-flops in the system to a known starting state prior to the clocked operation.



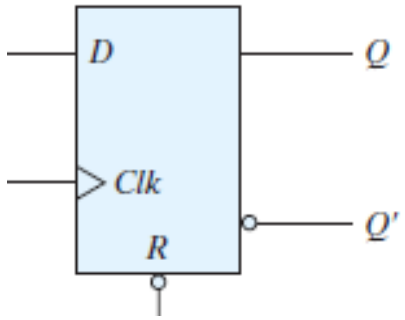
Verilog models of D flip-flop



Edge triggered D flip-flop:

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

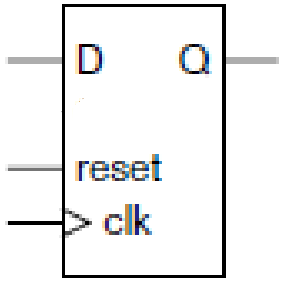
No reset ff



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;
```

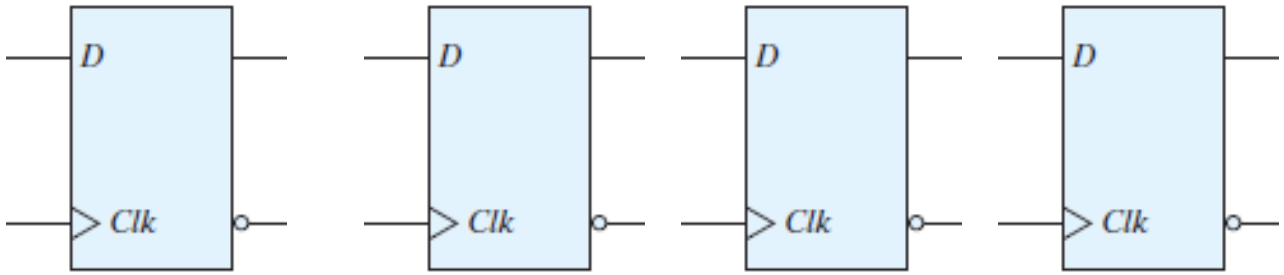
Not used in class



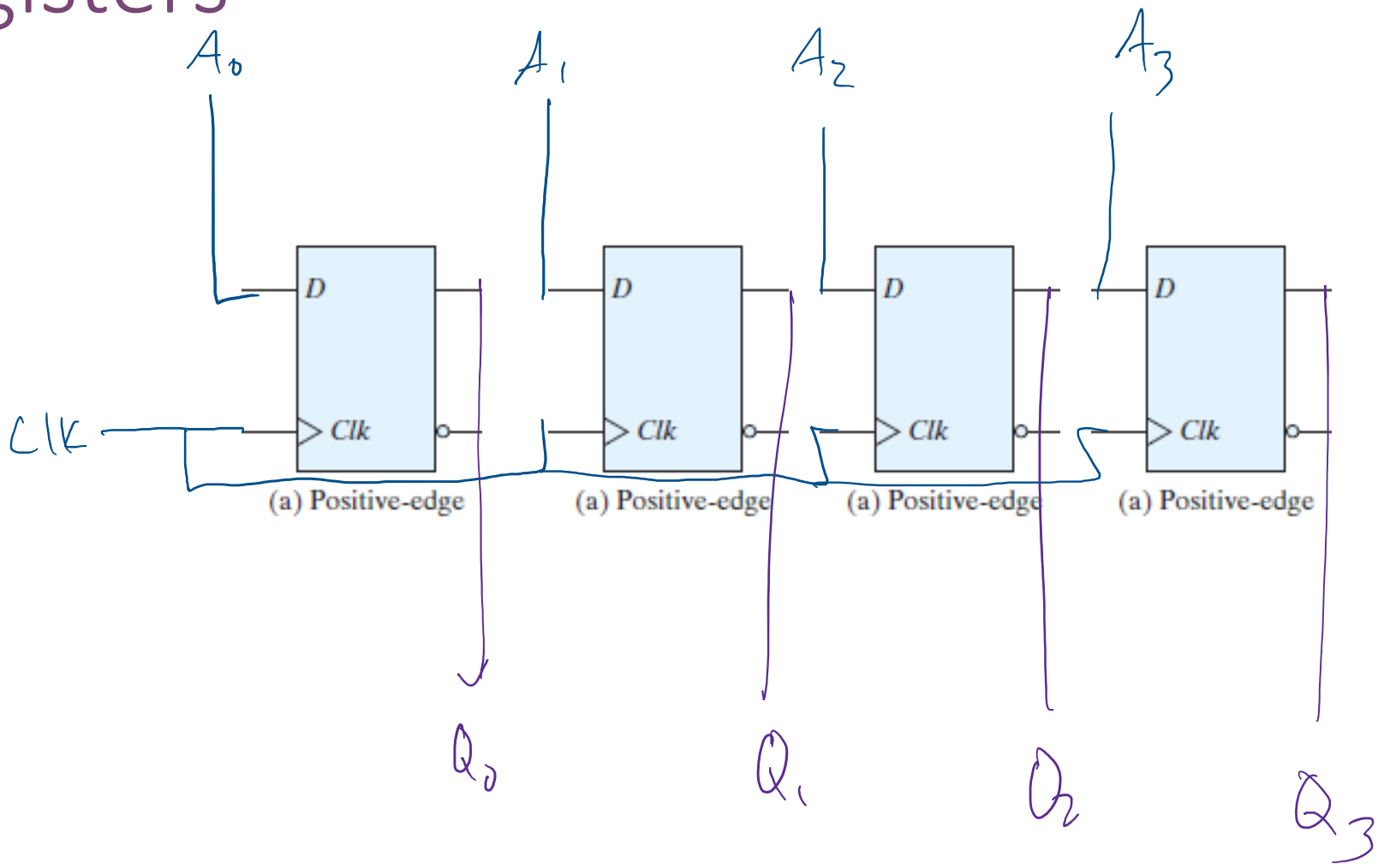
Edge triggered, synchronous reset, clock enable D flip-flop:

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

Registers


D
Clk
(a) Positive-edge
D
Clk
(a) Positive-edge
D
Clk
(a) Positive-edge
D
Clk
(a) Positive-edge

Registers



registers in a CPU?

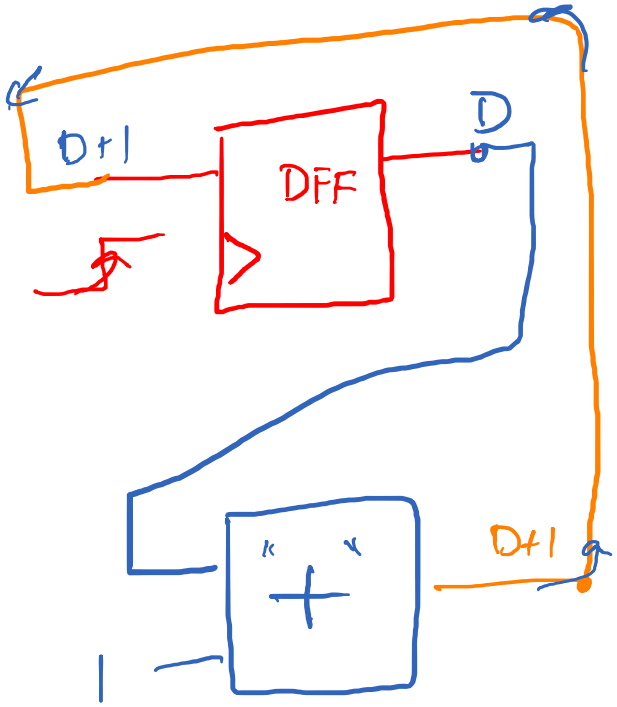
4-bit Register in Verilog

```
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
```

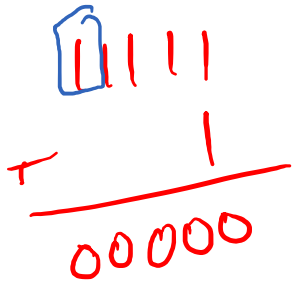
4-bit Register in Verilog

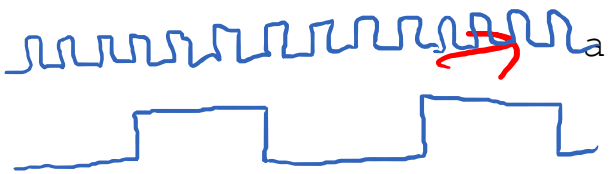
```
module d_ff (
    input      [3:0]   d,   //data
    input              clk, //clock
    output logic [3:0] q    //output register
);

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

endmodule
```

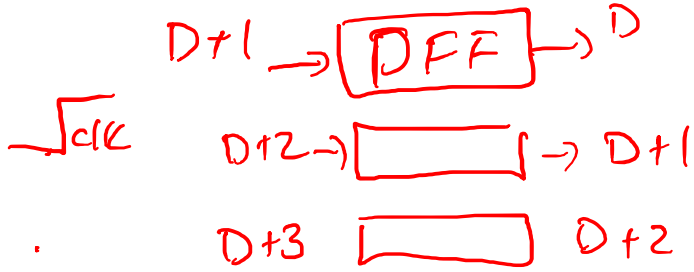
What does this module do?



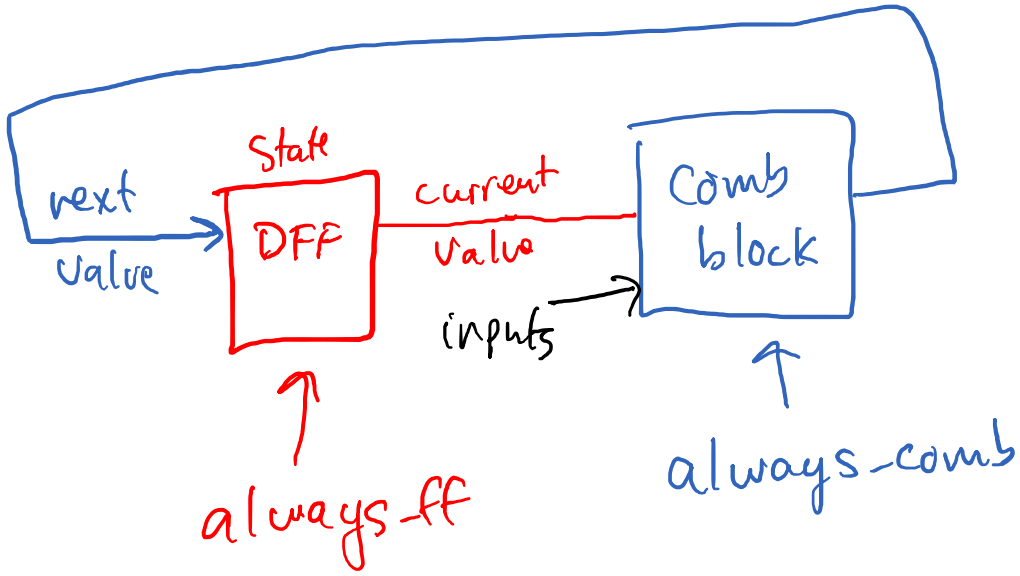

```
module mystery(
    input clk,          //clock
    input rst,          //reset
    output logic out    //output
);
    logic [3:0] D;
    wire [4:0] sum;

    always_ff @( posedge clk ) // <- sequential logic
    begin
        if (rst) D <= 4'h0;
        else     D <= sum;     //non-blocking
    end

    always_comb // <- combinational logic
    {out,sum} = {0,D} + 5'h1;  //blocking

endmodule
```







What does this module do?

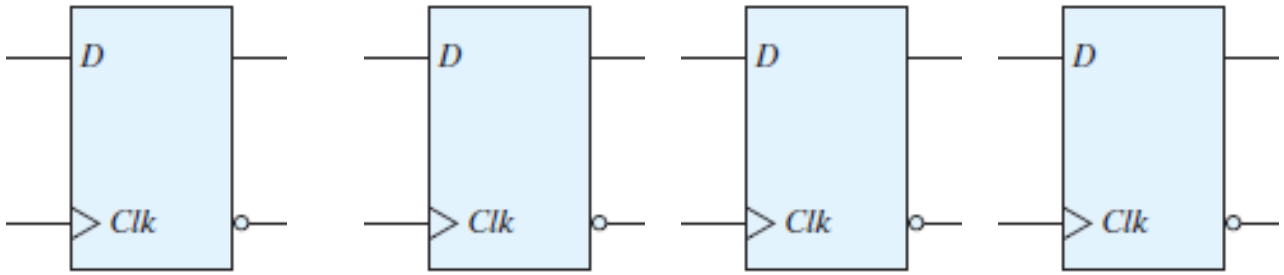



```
module counter(
    input clk,          //clock
    input rst,          //reset
    output logic out    //output
);
    logic [3:0] D;
    wire [4:0] sum;

    always_ff @( posedge clk ) // <- sequential logic
    begin
        if (rst) D <= 4'h0;
        else     D <= sum;    //non-blocking
    end

    always_comb // <- combinational logic
        {out,sum} = {0,D} + 5'h1;  //blocking

endmodule
```

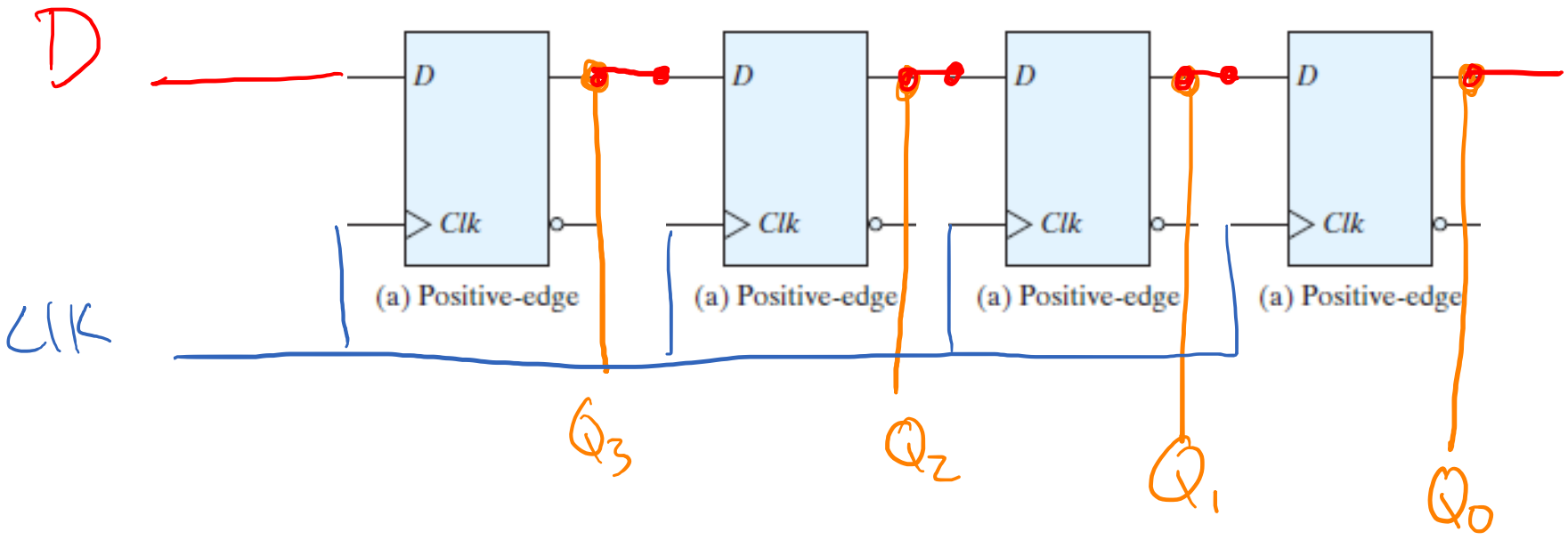
uses FF's

uses AND, OR, NOT Gates

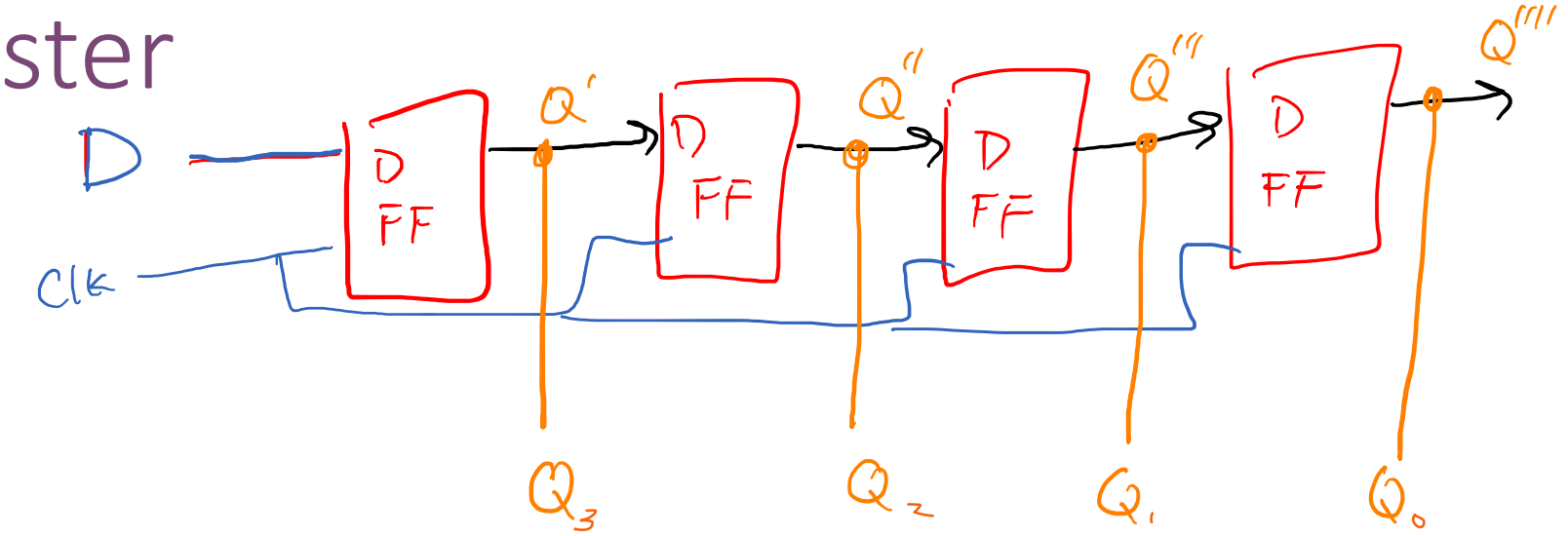
D Flip-Flops as Shift Registers

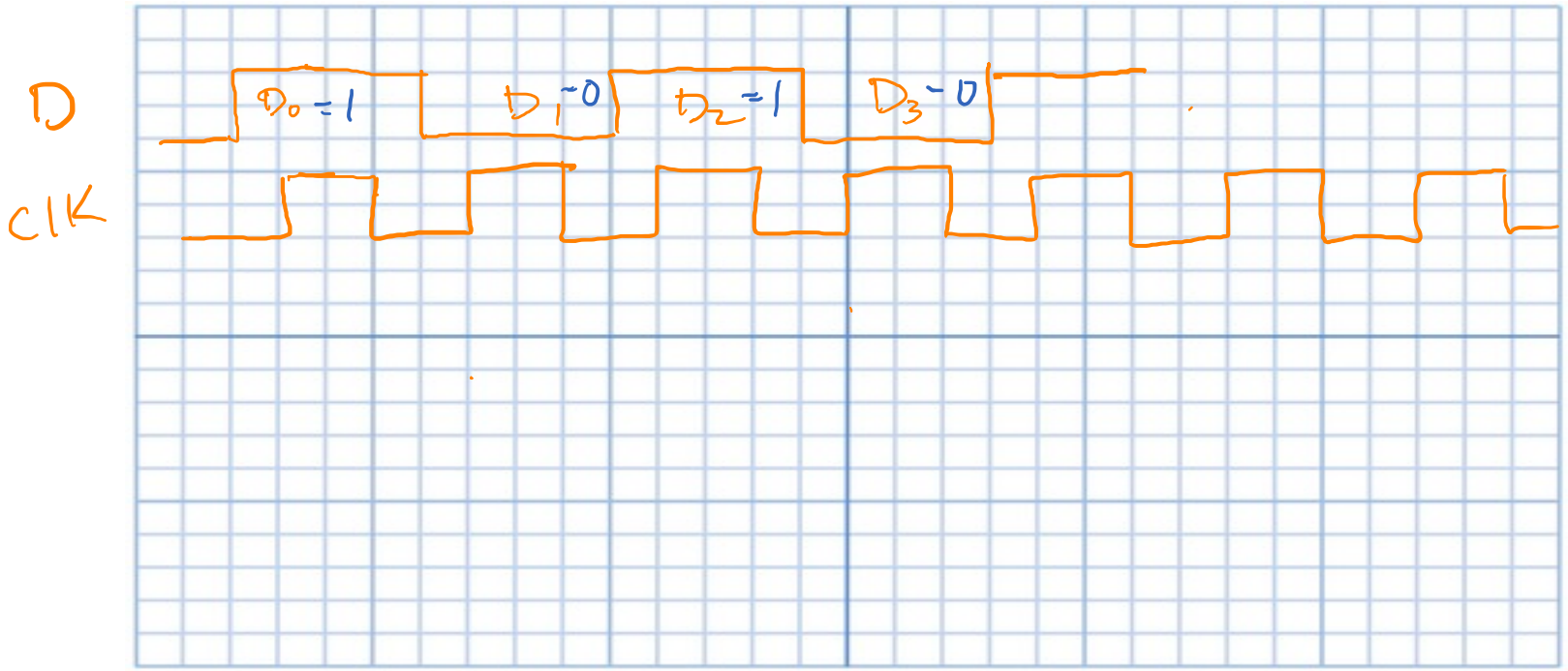


D Flip-Flops as Shift Registers



Shift Register





Shift Register