ALUs + Latches

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

```
wire w;
assign w = 'h0;
```

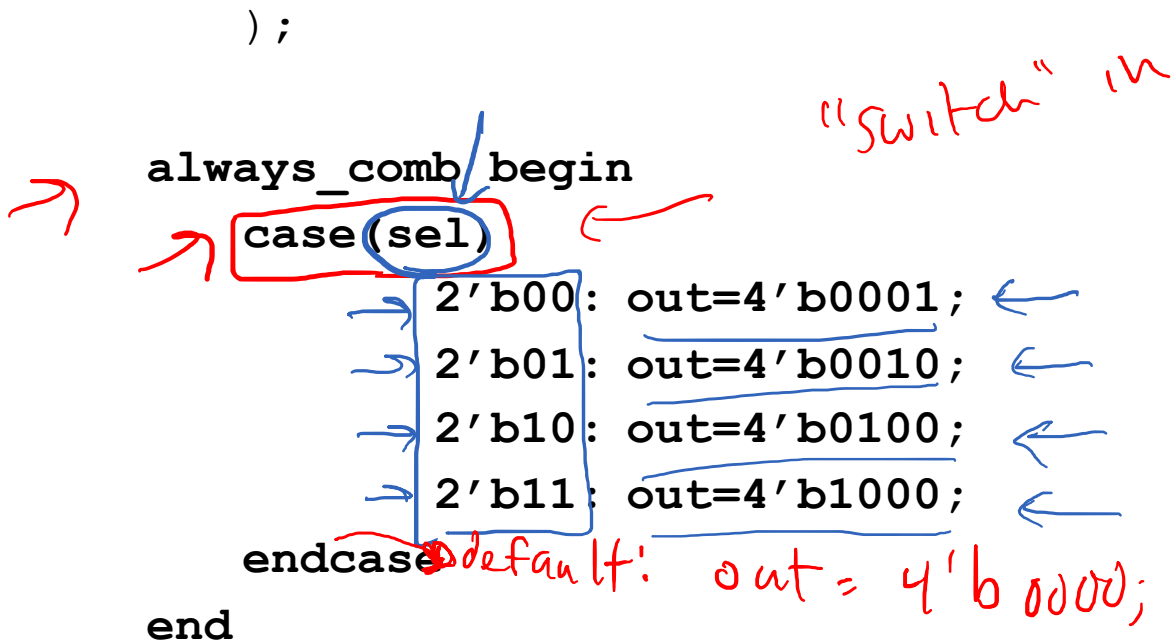
```
// w = 1 if x == 1
//     0 if x == 2
//     1 if x == 3
```

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
    end

endmodule
```



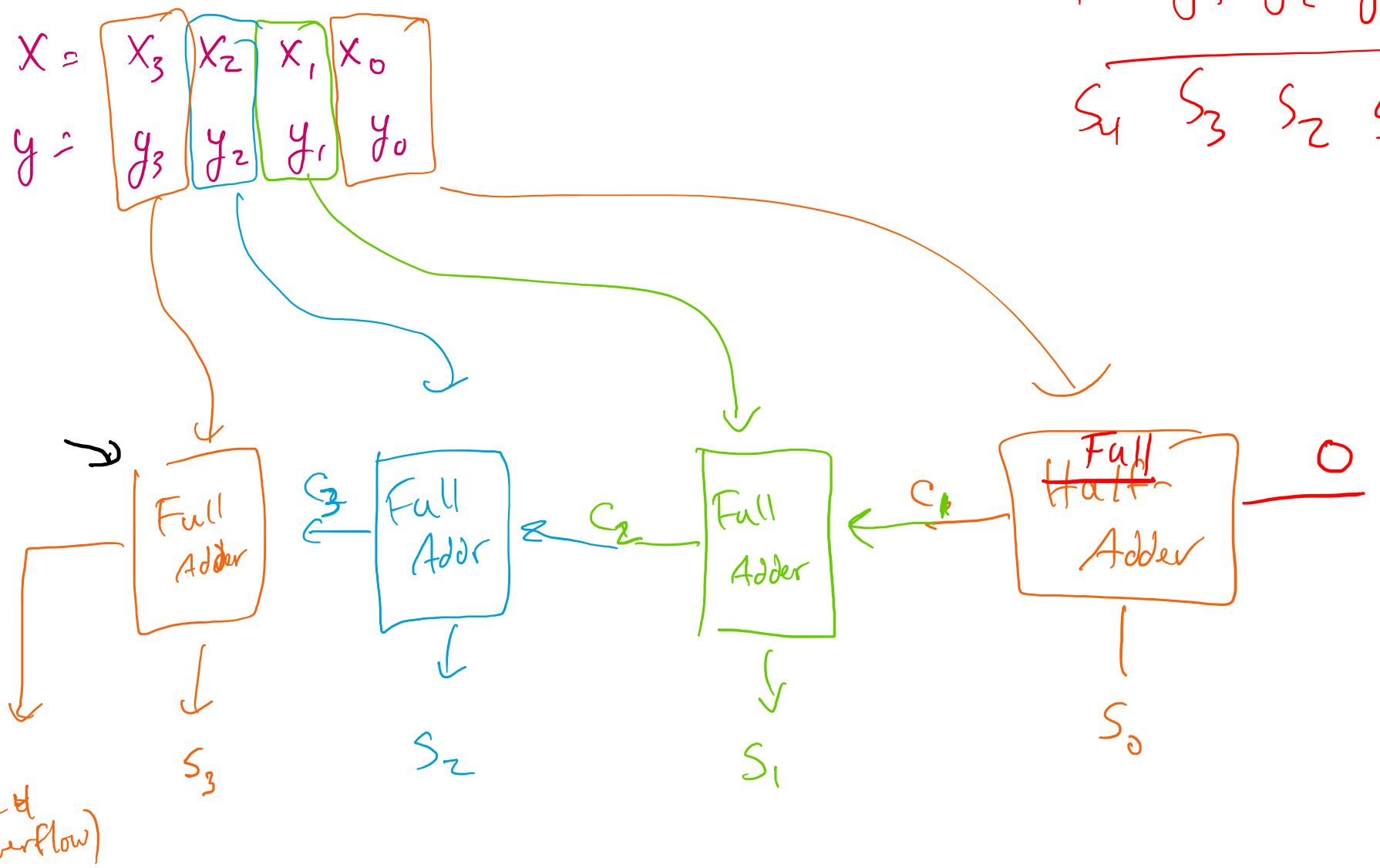
always_comb with case

```
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
```

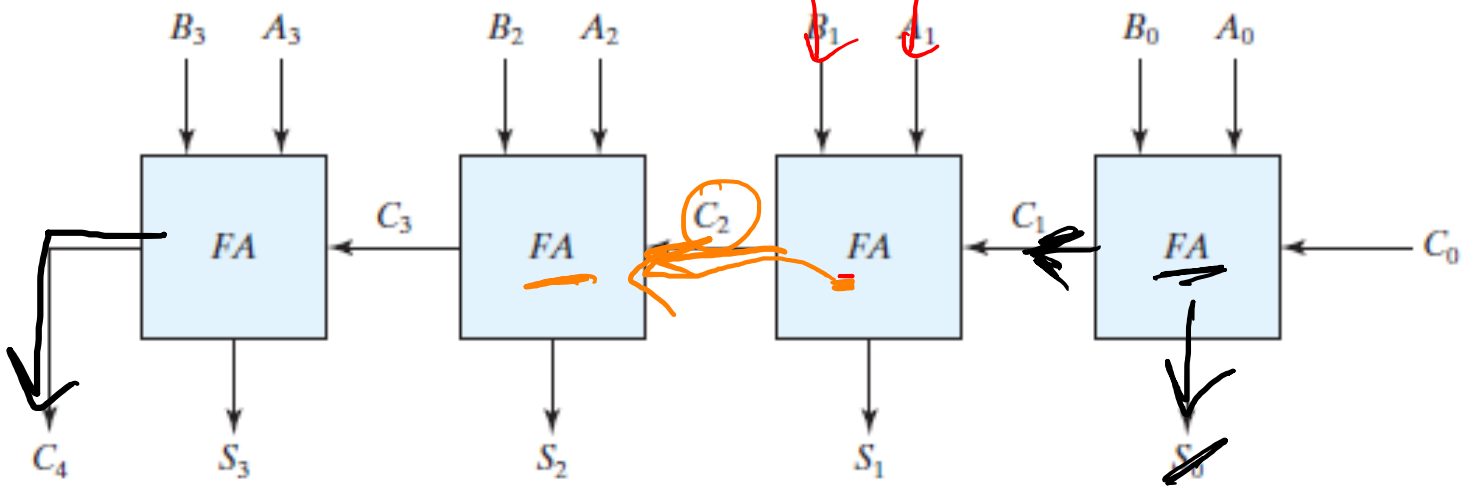
Always specify defaults for **always_comb**!

Ripple-Carry Adder

$X + Y = X_3\ X_2\ X_1\ X_0 + Y_3\ Y_2\ Y_1\ Y_0 = S_4\ S_3\ S_2\ S_1\ S_0$



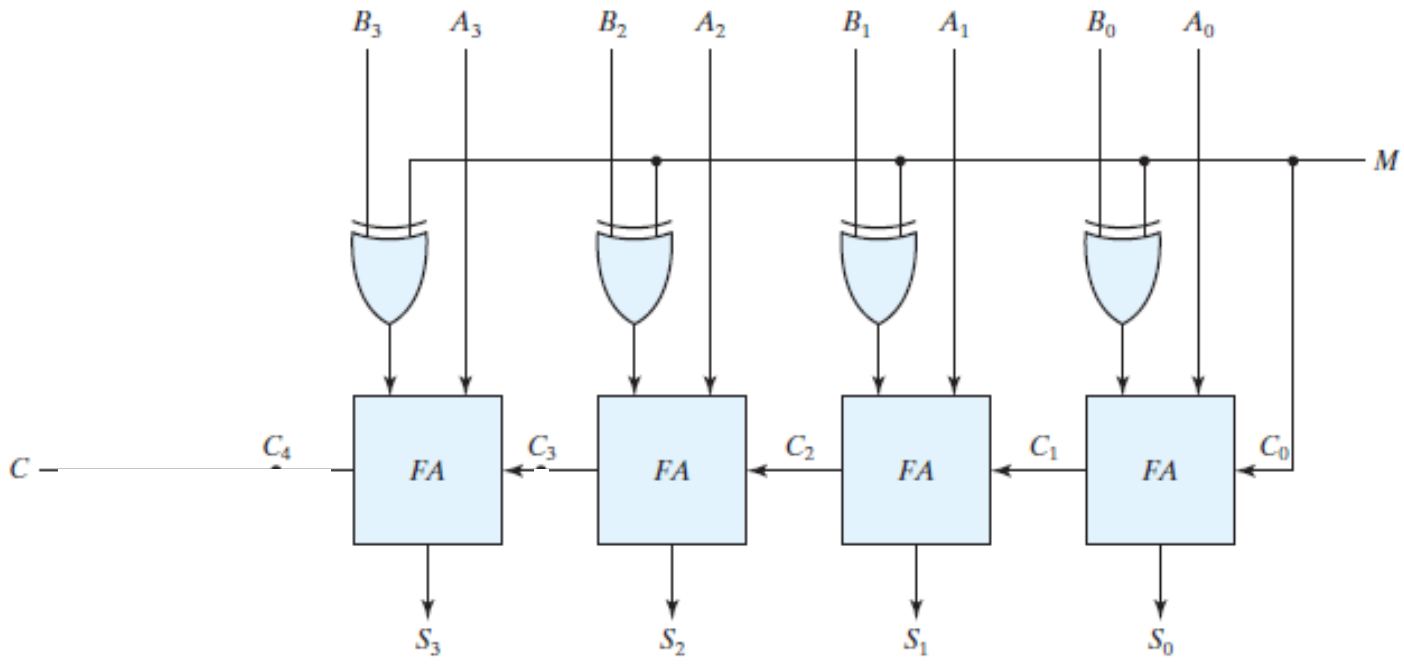
Ripple-Carry Adder




```
module addr (
input [3:0] a;
input [3:0] b,
output [4:0] z
);
[4:0] c;
FA fa0(.a(a[0])
.b(b[0]),
.cin('h0),
.sum(z[0]),
.cout(c[1]),
FA fa1 ( a[1]
b[1]
c[1]
z[0]
c[2]);
assign z[4] = c[4];
```

Adder/Subtractor

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



Overflow for signed numbers?

Assume 4-bit addition

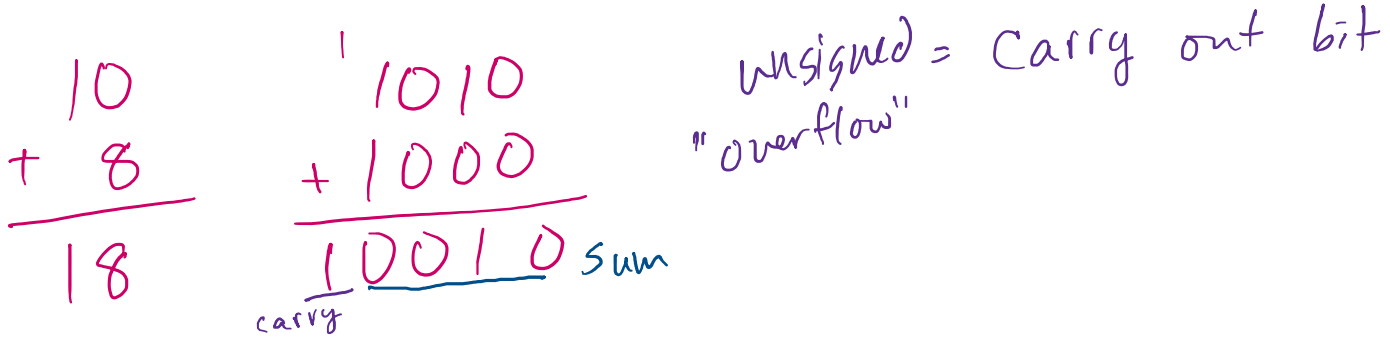
- Unsigned

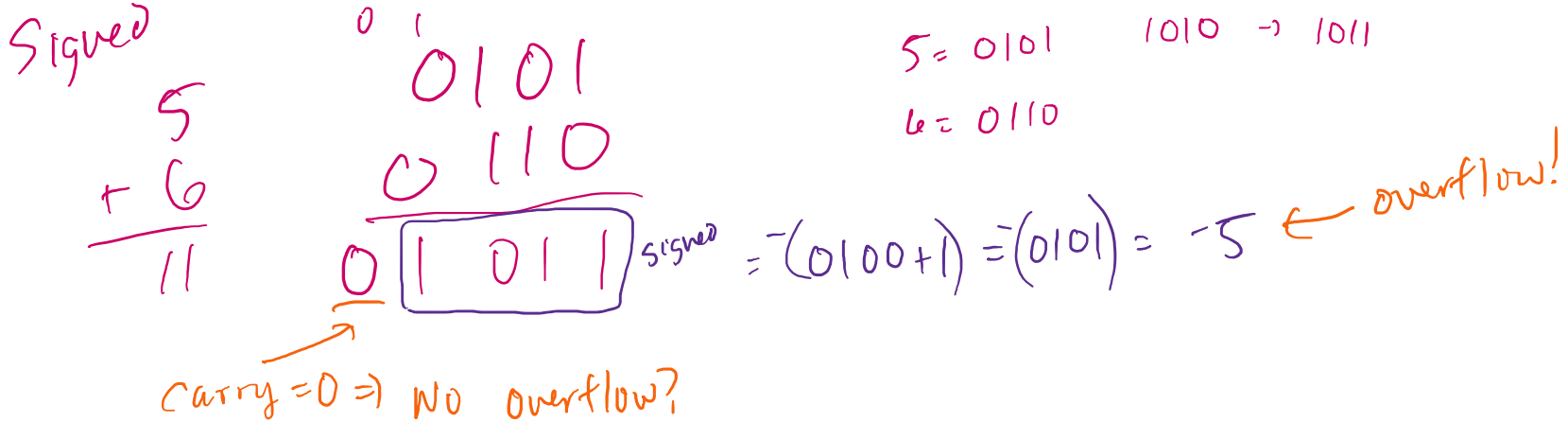
$$\begin{array}{r} 10 \\ +\ 8 \\ \hline 18 \end{array}$$

- Signed

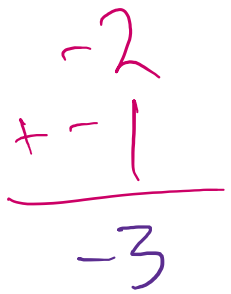
$$\begin{array}{r} 5 \\ +\ 6 \\ \hline 11 \end{array}$$

Overflow for signed numbers?





Overflow for signed numbers?




```
  +2
+ -1
-------
   +1
```

Overflow for signed numbers?

-2 $\quad -(0010) = 1101 + 1 = 1110$

$+-1$ $\quad -(0001) = 1110 + 1 = 1111$

-3

$\Rightarrow$

$$\begin{array}{r} {}^{1}{}^{1}{}^{1}1110 \\ +\ 1111 \\ \hline 11101 \end{array}$$

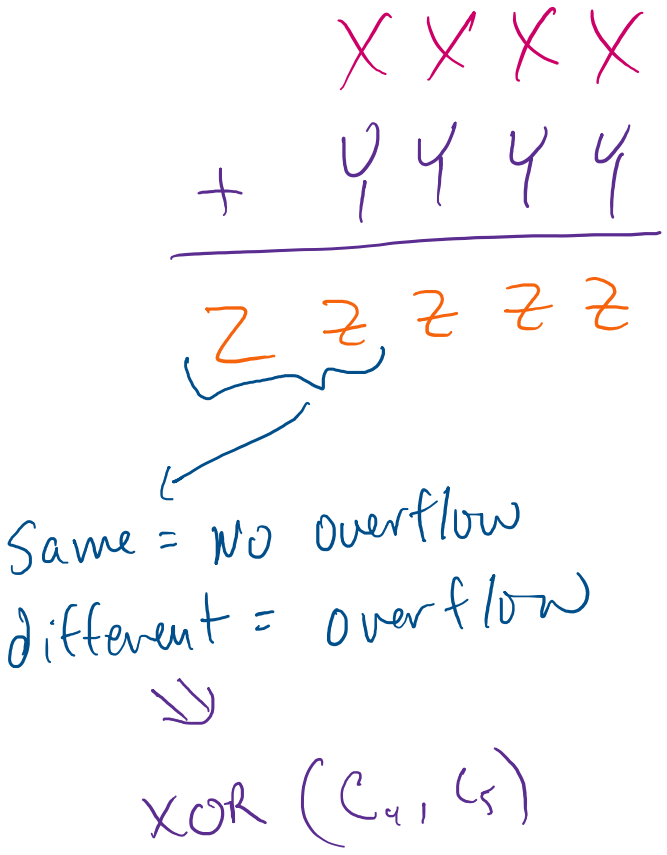
sum $= -(0010+1)$

$= -(0011) = -3$

No overflow?

```
  +2.      0010.    0010
+ -1.   +-(0001) + 1111
-------  ------    ----
   +1              10001 <- no overflow
```

Overflow for signed numbers



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.

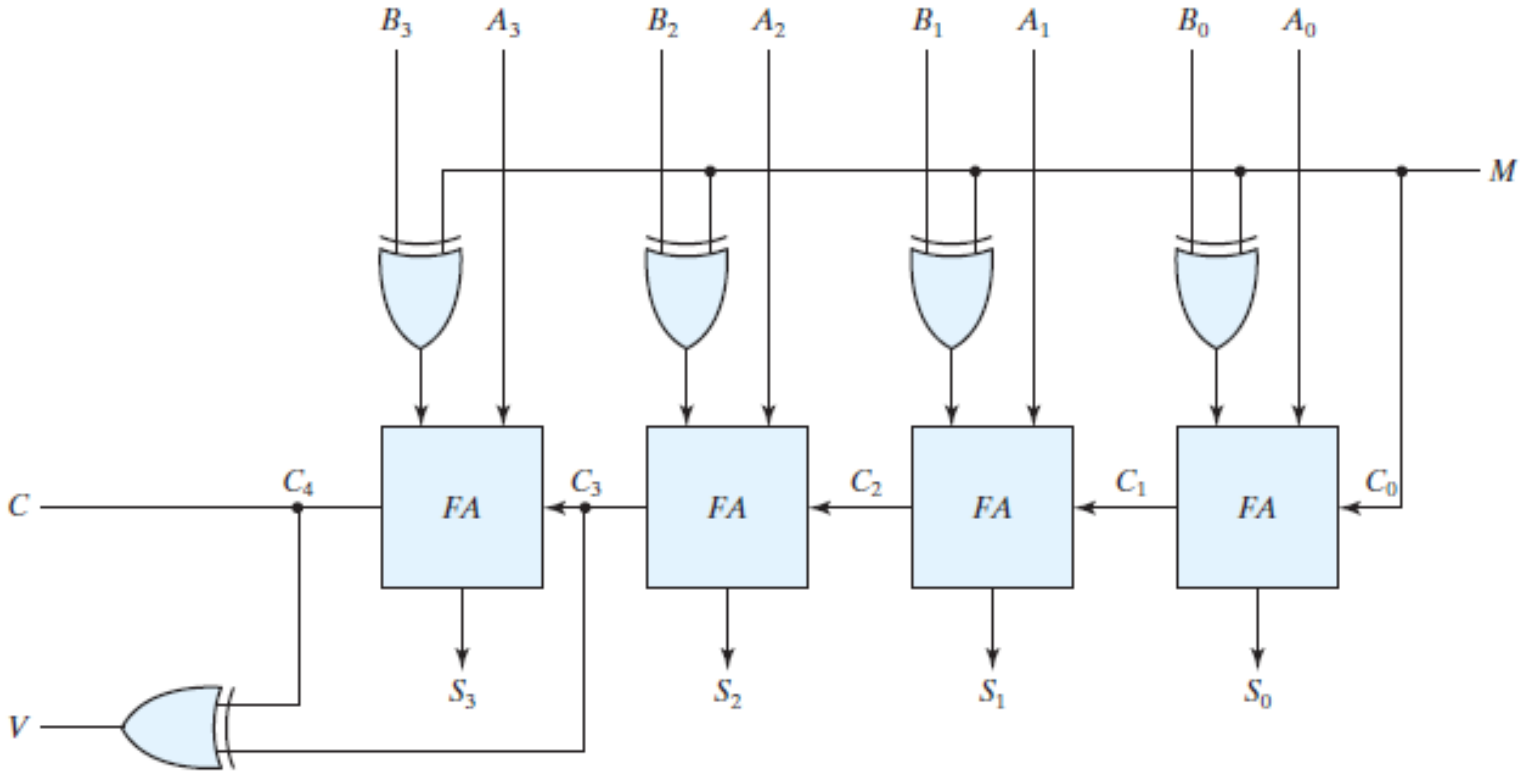
Overflow detection

- 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



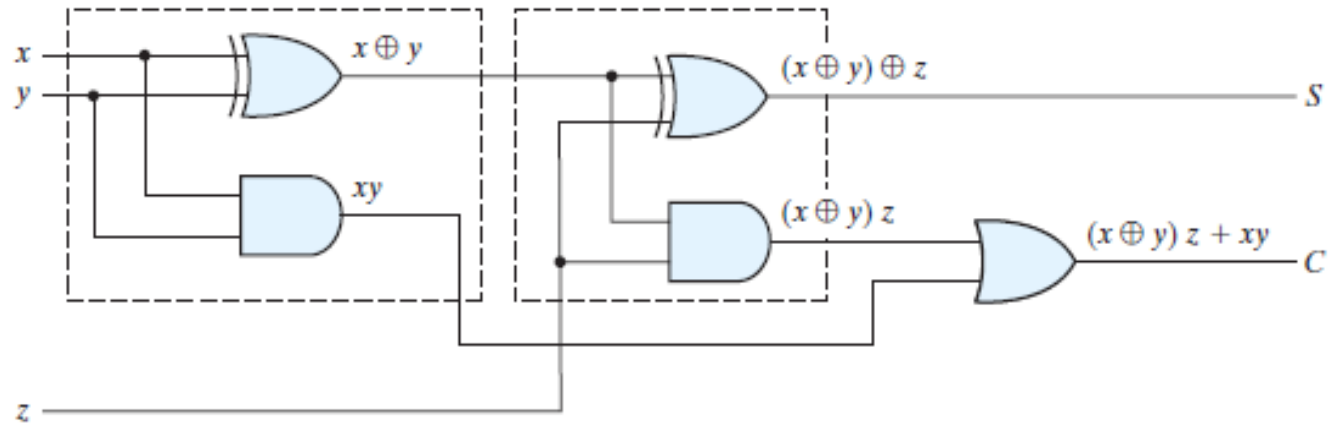
Gate Delay

- 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

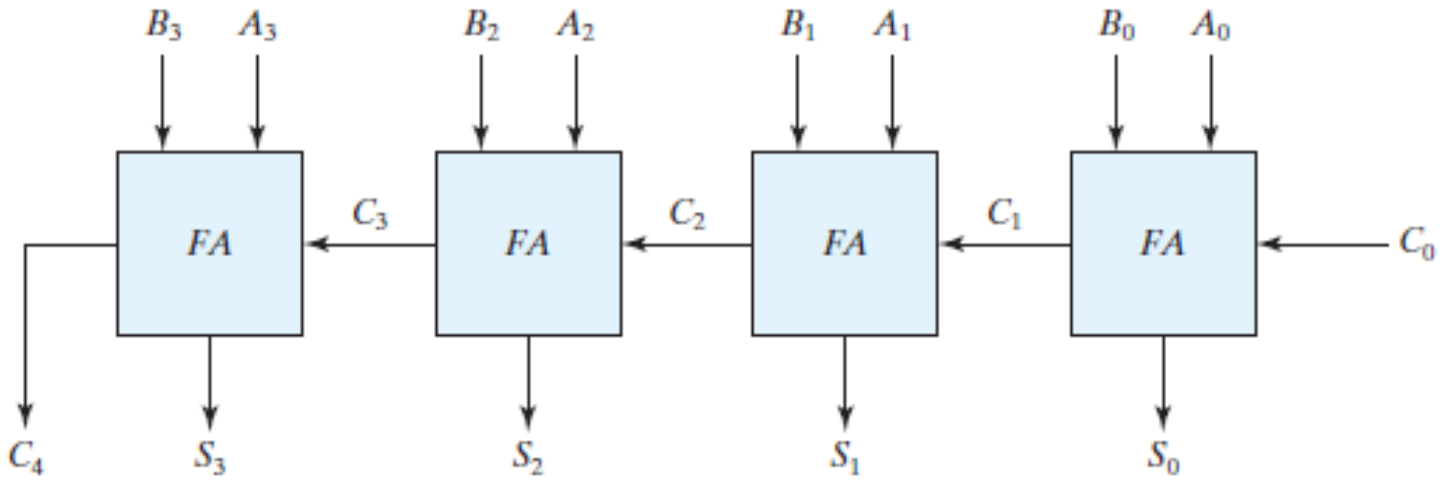
- Assume 5ps/gate



- 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: 15ps
- 4-bit addition: 60ps
- 8-bit addition: 120ps
- 16-bit addition: 240ps
- 32-bit addition: 480ps
- 64-bit addition: 960ps = ~1 GHz

Faster Adder Options?

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

- Google “Carry Look-Ahead Adder”

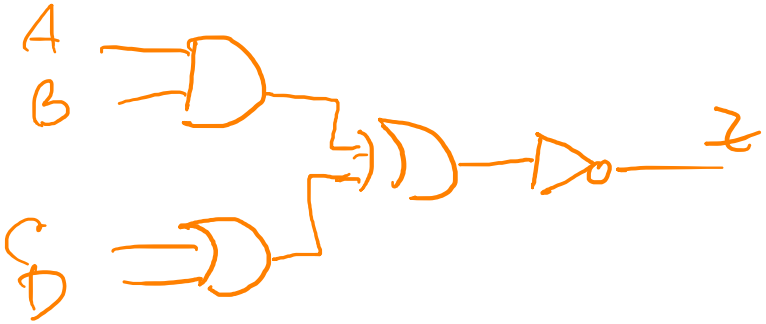
WARNING: MAJOR TOPIC SHIFT

SEQUENTIAL LOGIC

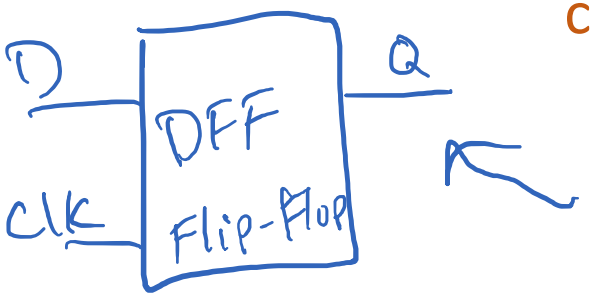
Sequential vs. Combinational

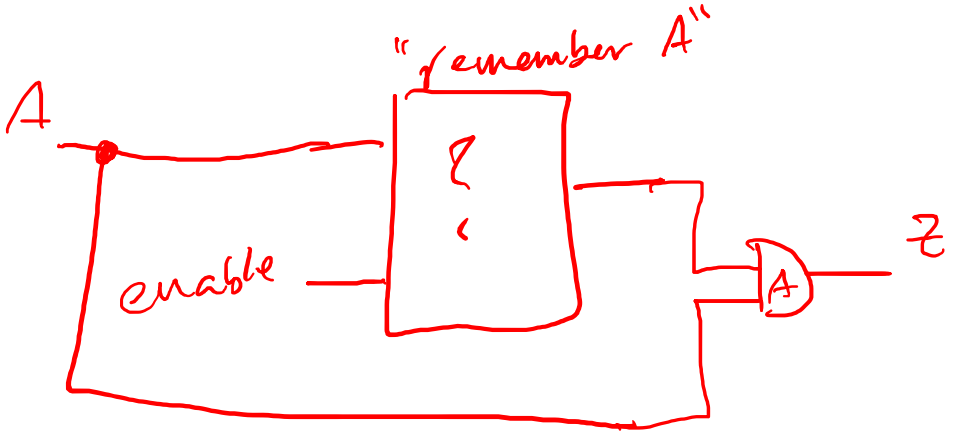
Stopped here!

- 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

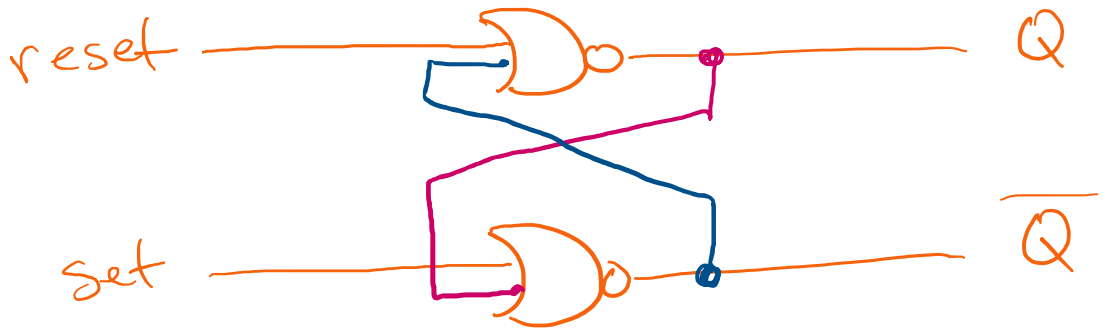




SR Latch

= 1

= 0

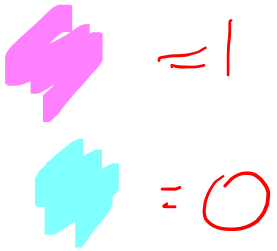


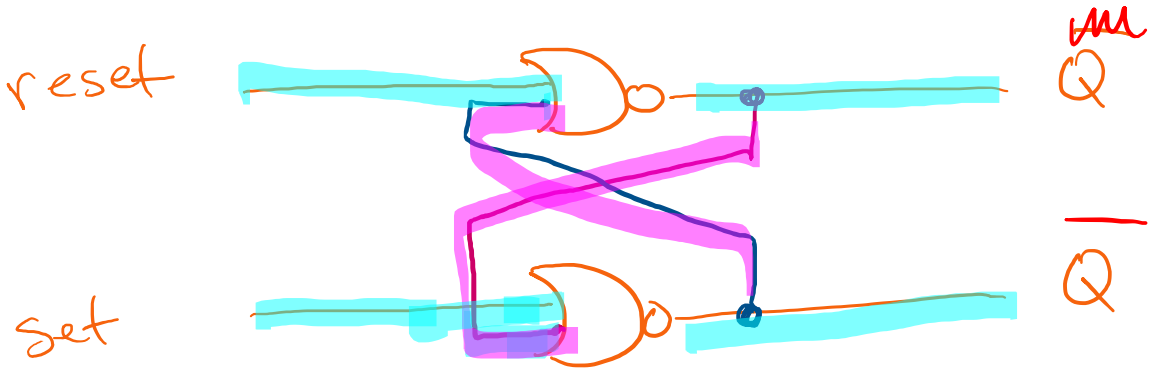
reset	set	Q	$\overline{Q}$
0	1	1	0
0	0	1	0
1	0	0	1
0	0	0	1

same inputs, different output!

⇒ Internal state!

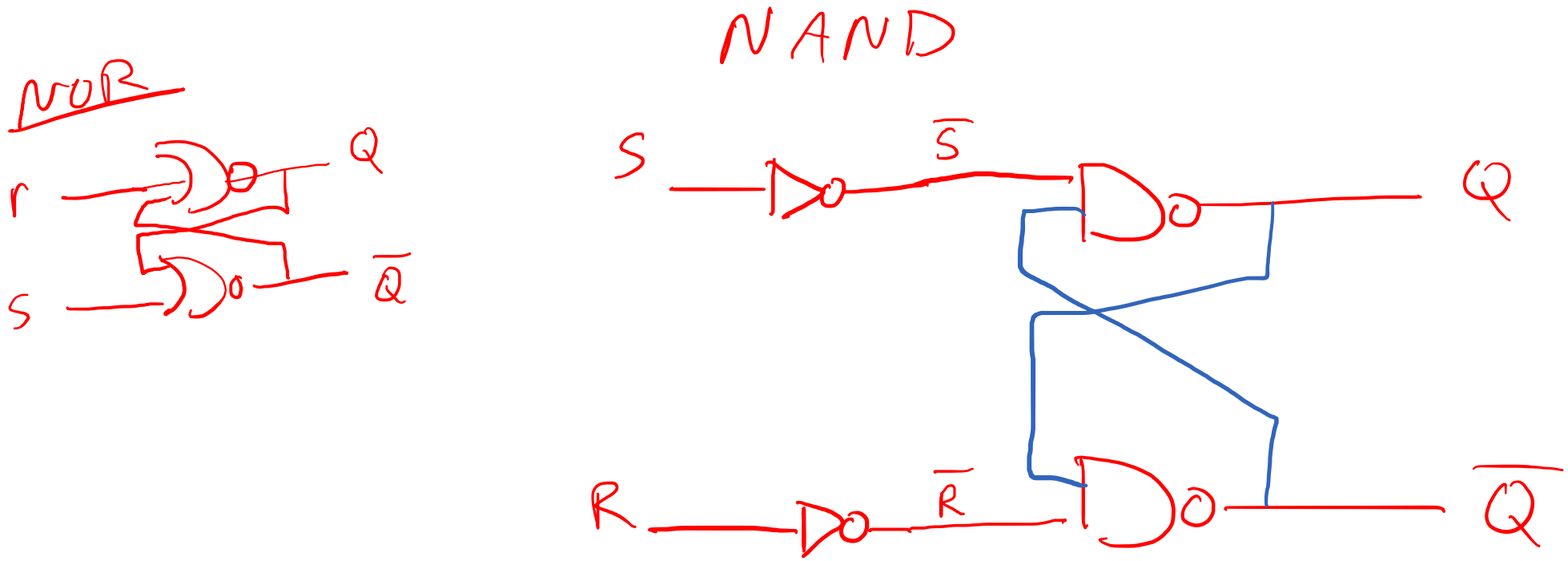
SR Latch w/ S=1 & R=1





set	reset	Q	$\overline{Q}$
1	1	0	0
0	0	0,1,0,1,0,1,0,1,...	

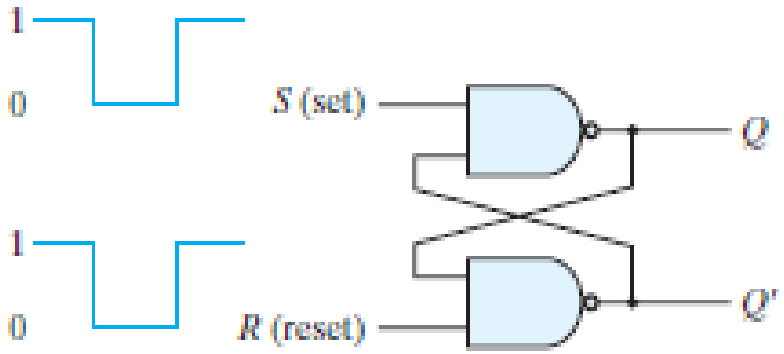
SR Latch w/NAND gates



→ better setup for Flip-Flops

→ easier for me to draw

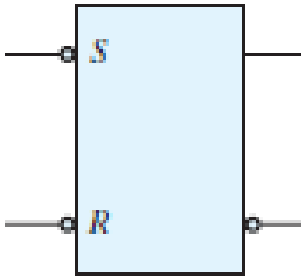
SR Latch with NAND gates



(a) Logic diagram

S	R	Q	Q'	
1	0	0	1	
1	1	0	1	(after $S = 1, R = 0$)
0	1	1	0	
1	1	1	0	(after $S = 0, R = 1$)
0	0	1	1	(forbidden)

(b) Function table



$\overline{SR}$

The latch operates with both inputs normally at 1.

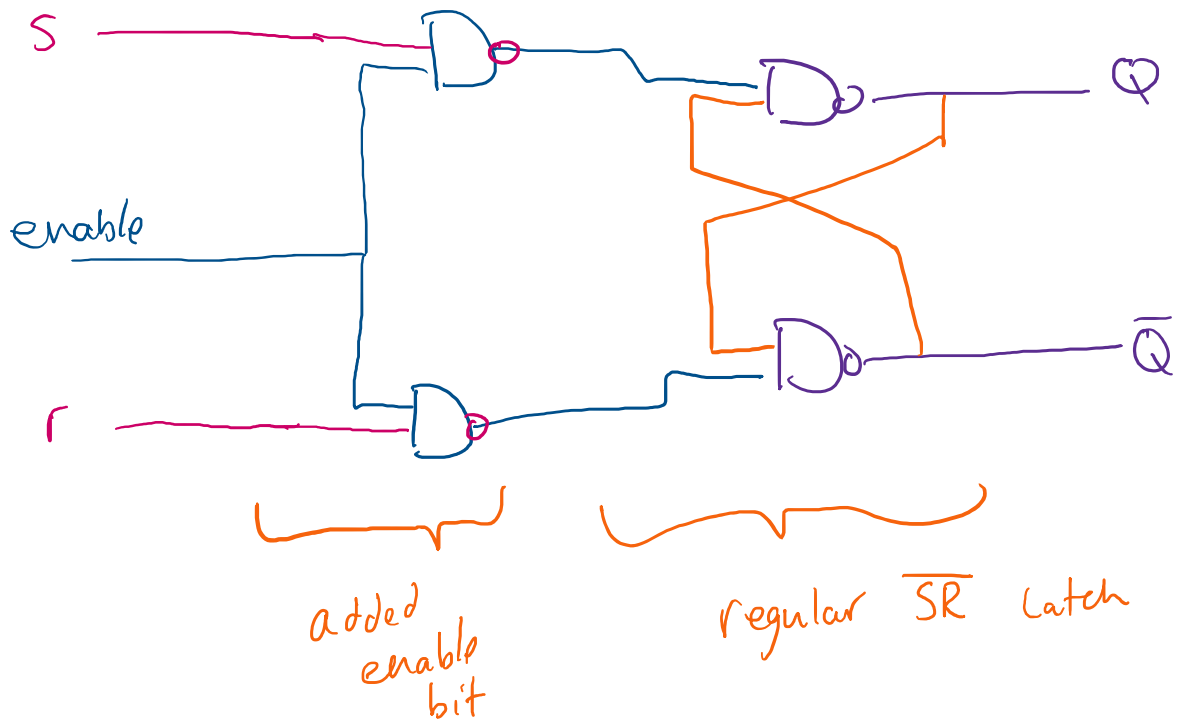
The application of 0 to the *S* input causes output *Q* to go to 1, putting the latch in the set state. When the *S* input goes back to 1, the circuit remains in the set state.

After both inputs go back to 1, we are allowed to change the state of the latch by placing a 0 in the *R* input. This action causes the circuit to go to the reset state and stay there even after both inputs return to 1.

The condition that is forbidden for the NAND latch is both inputs being equal to 0 at the same time, an input combination that should be avoided.

SR Latch with Enable

Prevent changes in S & r from changing circuit output

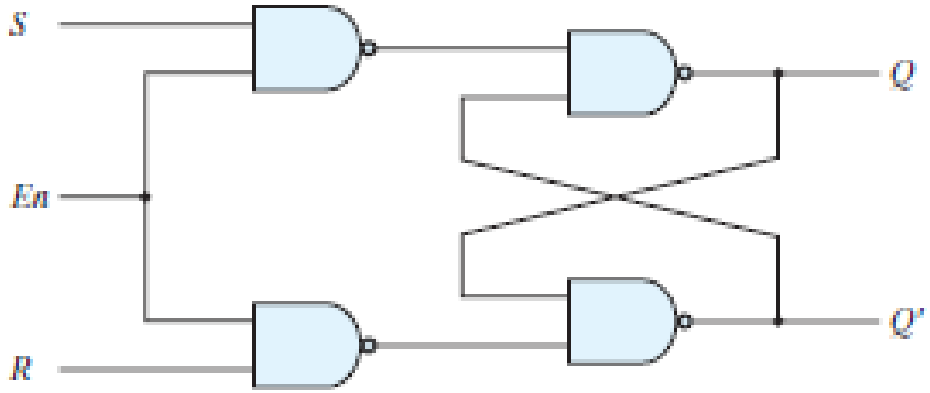


S	R	E	Q	$\overline{Q}$
x	x	0	Q	$\overline{Q}$
1	0	1	1	0
0	1	1	0	1

* assume no S=1 & r=1

SR Latch with control input

The operation of the basic *SR* latch can be modified by providing an additional input signal that controls *when* the state of the latch can be changed by determining whether *S* and *R* (or *S* and *R*) can affect the circuit.



(a) Logic diagram

En	*S*	*R*	Next state of *Q*
0	X	X	No change
1	0	0	No change
1	0	1	*Q* = 0; reset state
1	1	0	*Q* = 1; set state
1	1	1	Indeterminate

(b) Function table

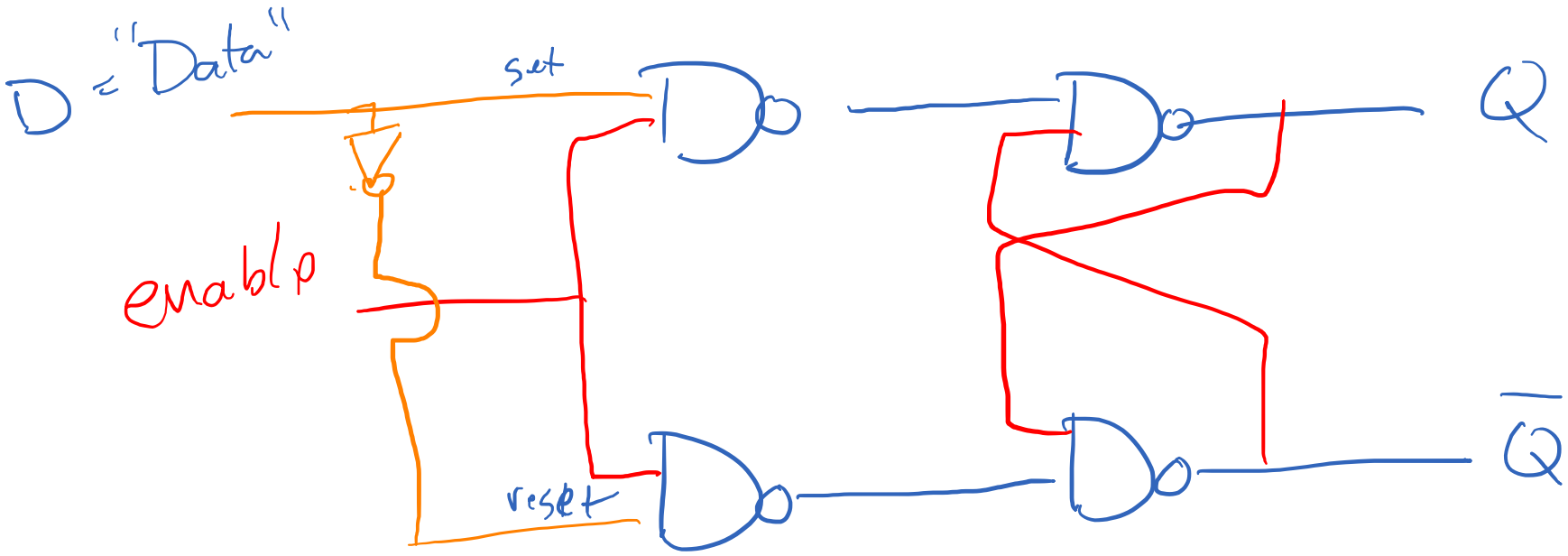
The control input *En* acts as an *enable* signal for the other two inputs. The outputs of the NAND gates stay at the logic *1* level as long as the enable signal remains at *0*. This is the quiescent condition for the *SR* latch.

When the enable input goes to *1*, information from the *S* or *R* input is allowed to affect the latch. The set state is reached with *S* = *1*, *R* = *0*, and *En* = *1*.

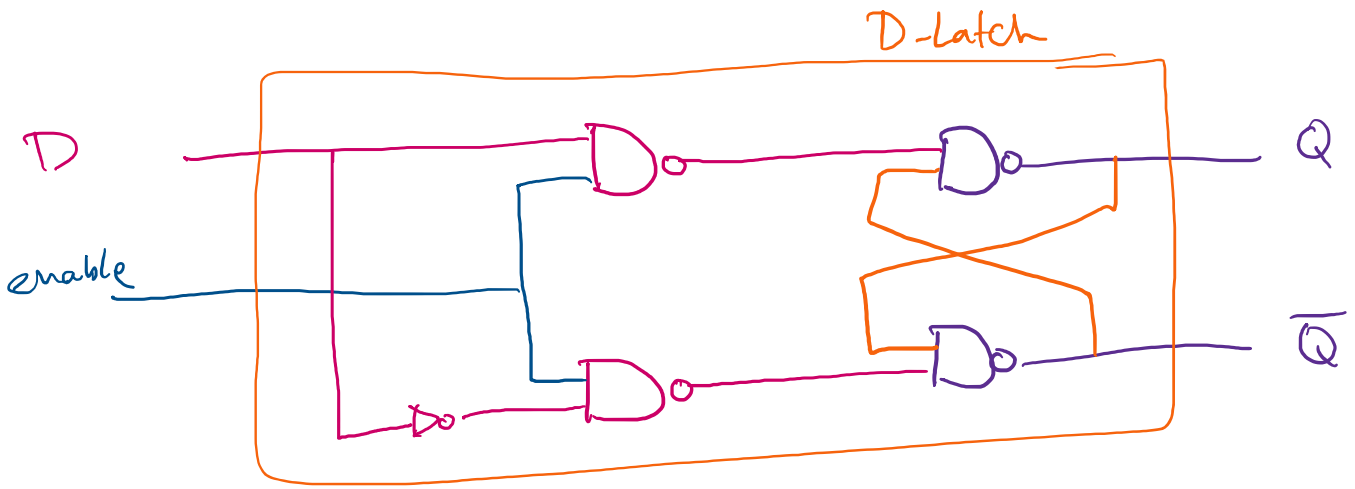
To change to the reset state, the inputs must be *S* = 0, *R* = 1, and *En* = 1.

"Data" Latch

D-Latch



D-Latch



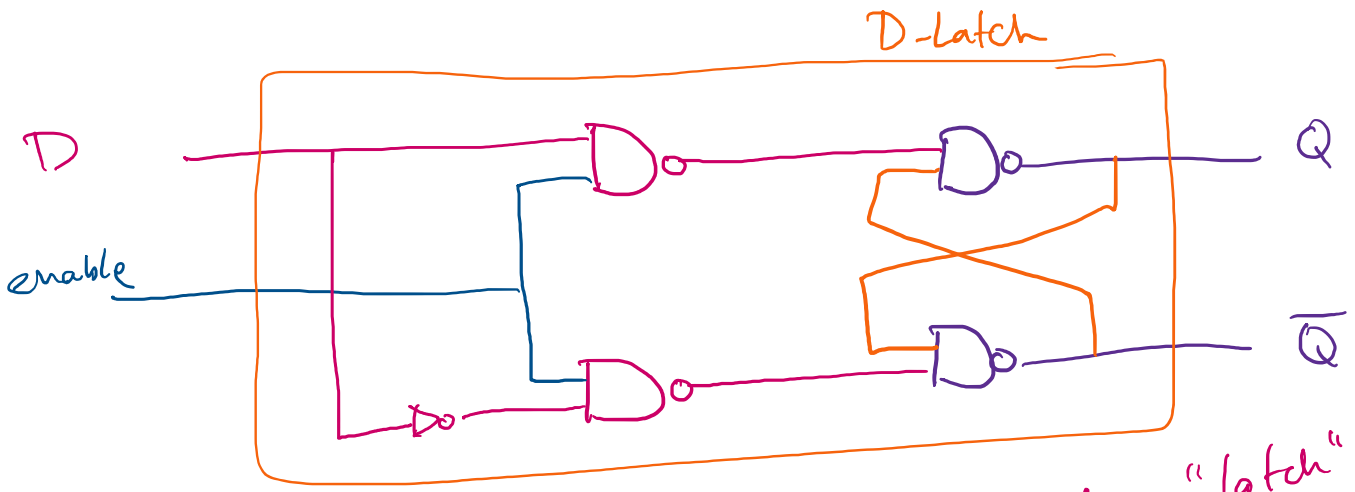
D	en	Q	$\overline{Q}$

D-Latch

"Data" Latch

= 1

= 0

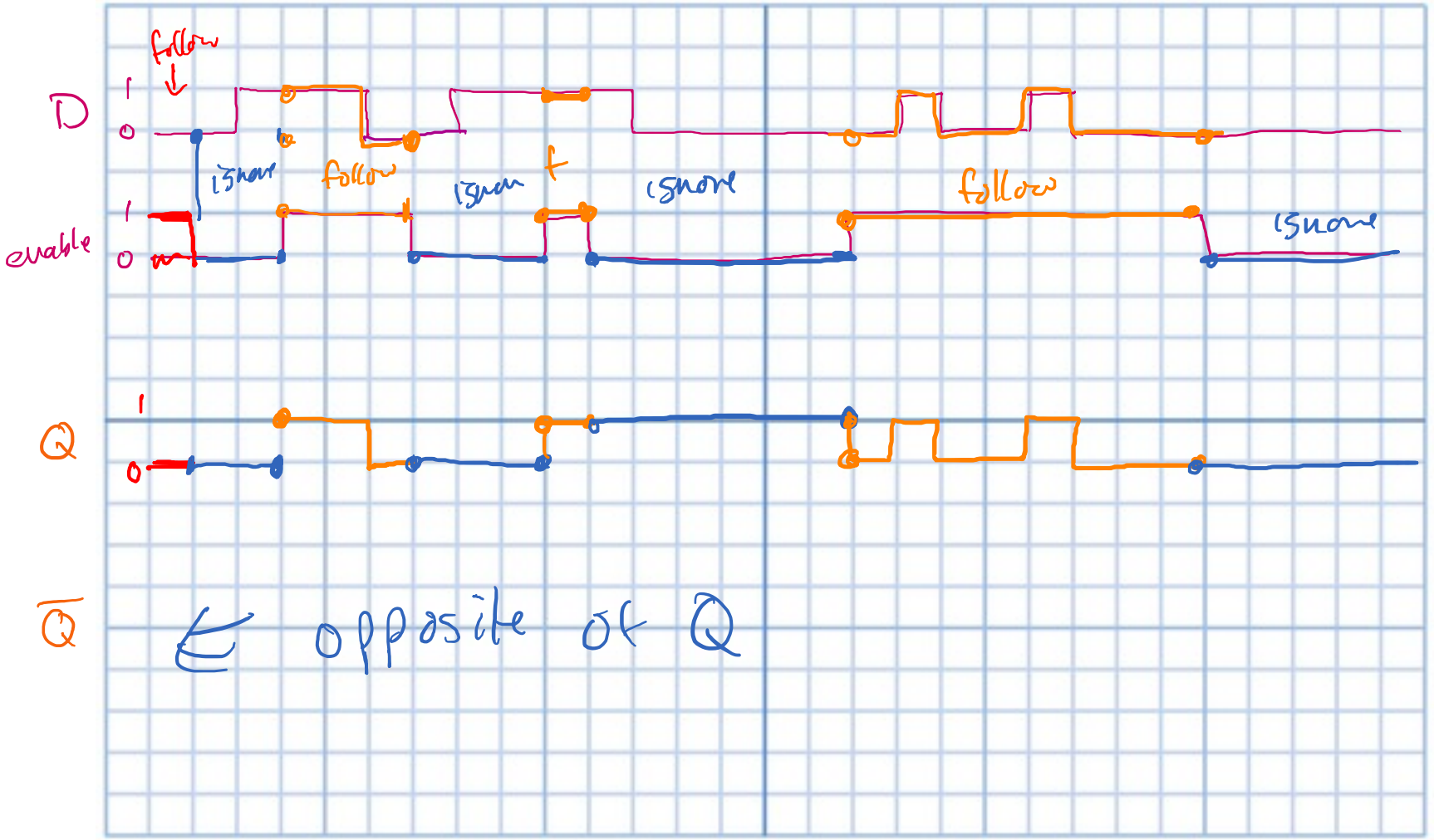


D	en	Q	$\overline{Q}$
0	0	Q	$\overline{Q}$
1	0	Q	$\overline{Q}$
0	1	0	1
1	1	1	0

← circuit to "latch" a value

* assume negligable gate delays

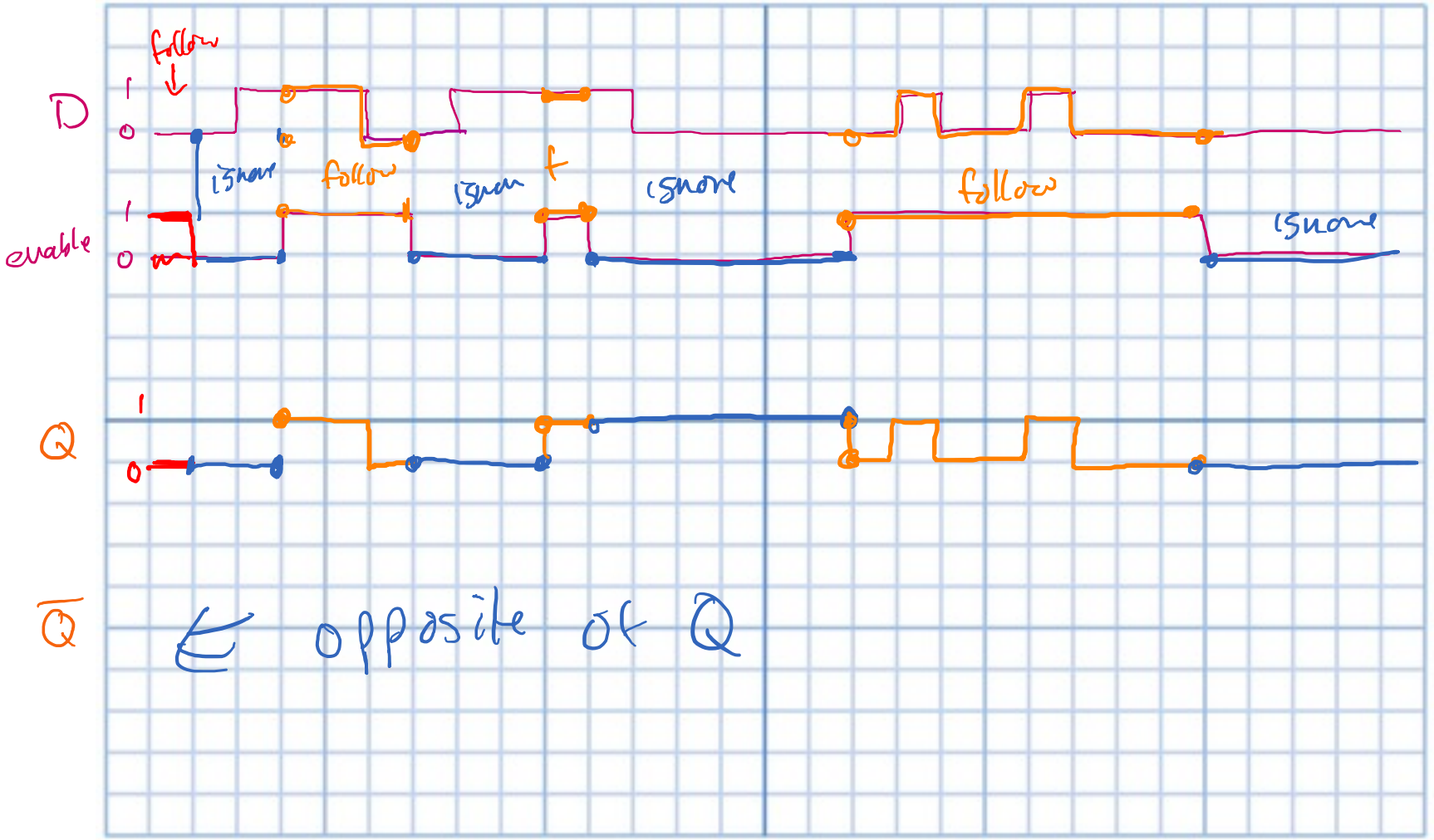
Inputs to D Latches



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

* assume negligable gate delays

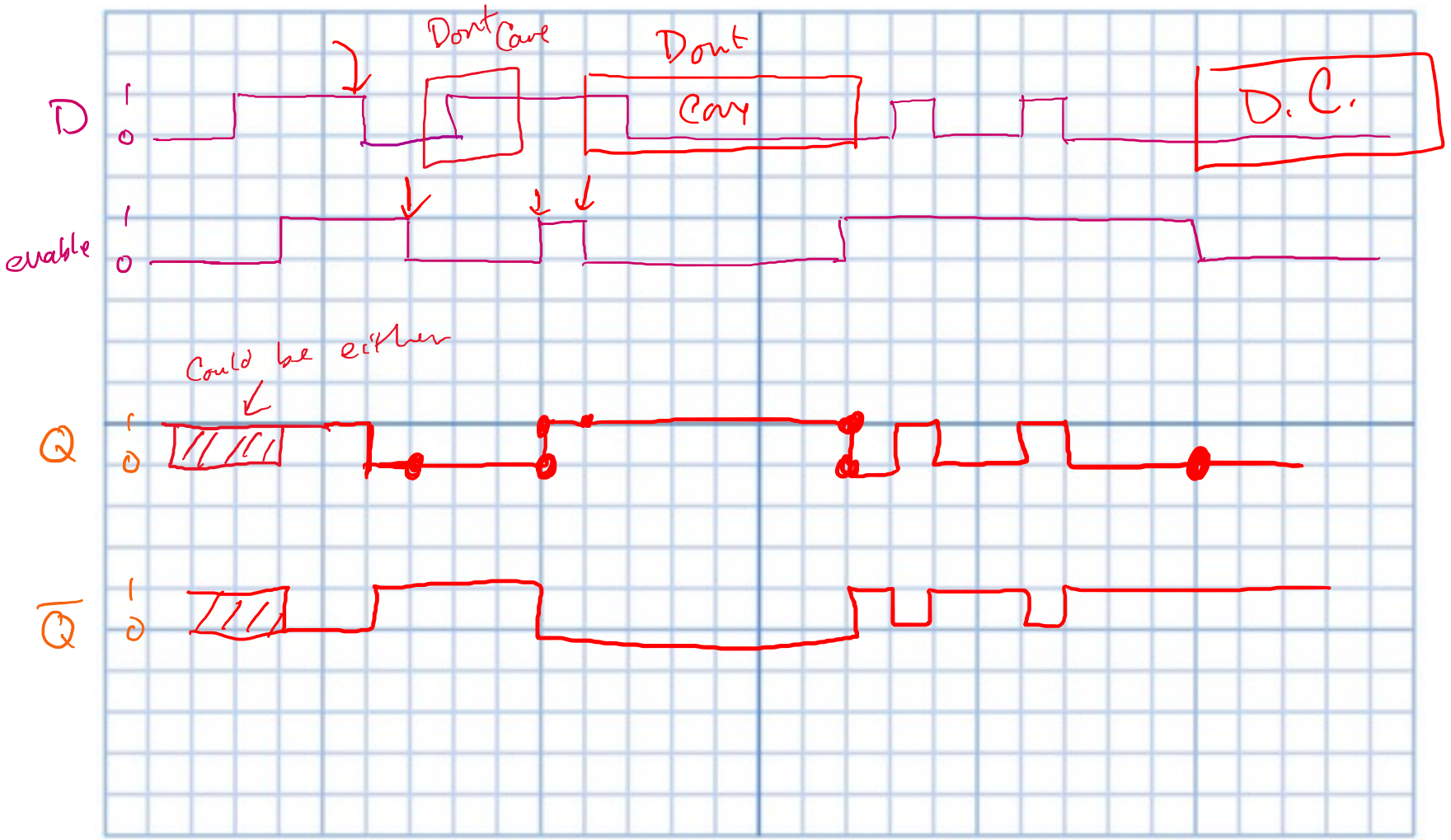
Inputs to D Latches



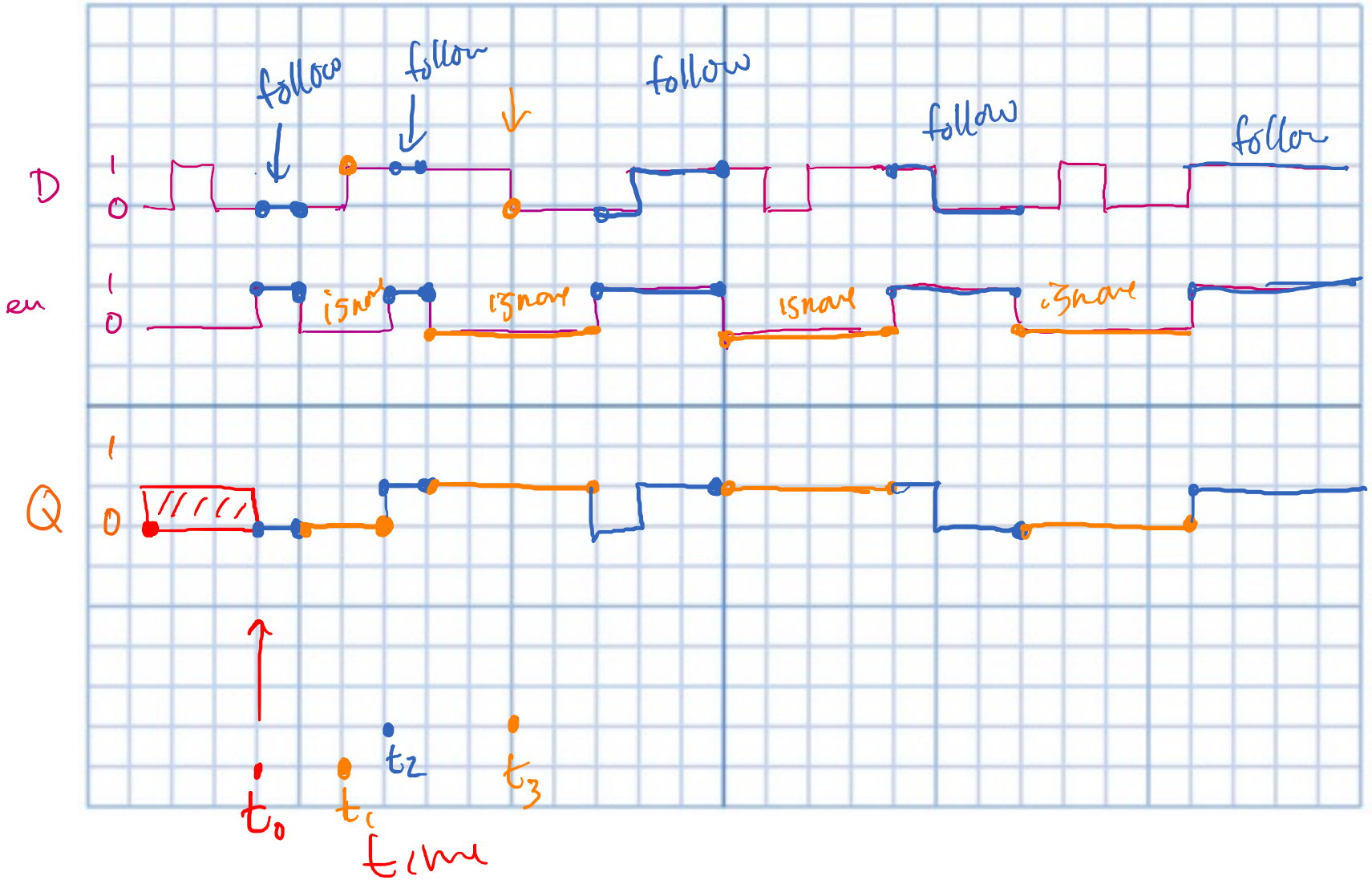
Q follows D when enable = 1, otherwise Q ignores D.

Inputs to D Latches

* assume negligable gate delays



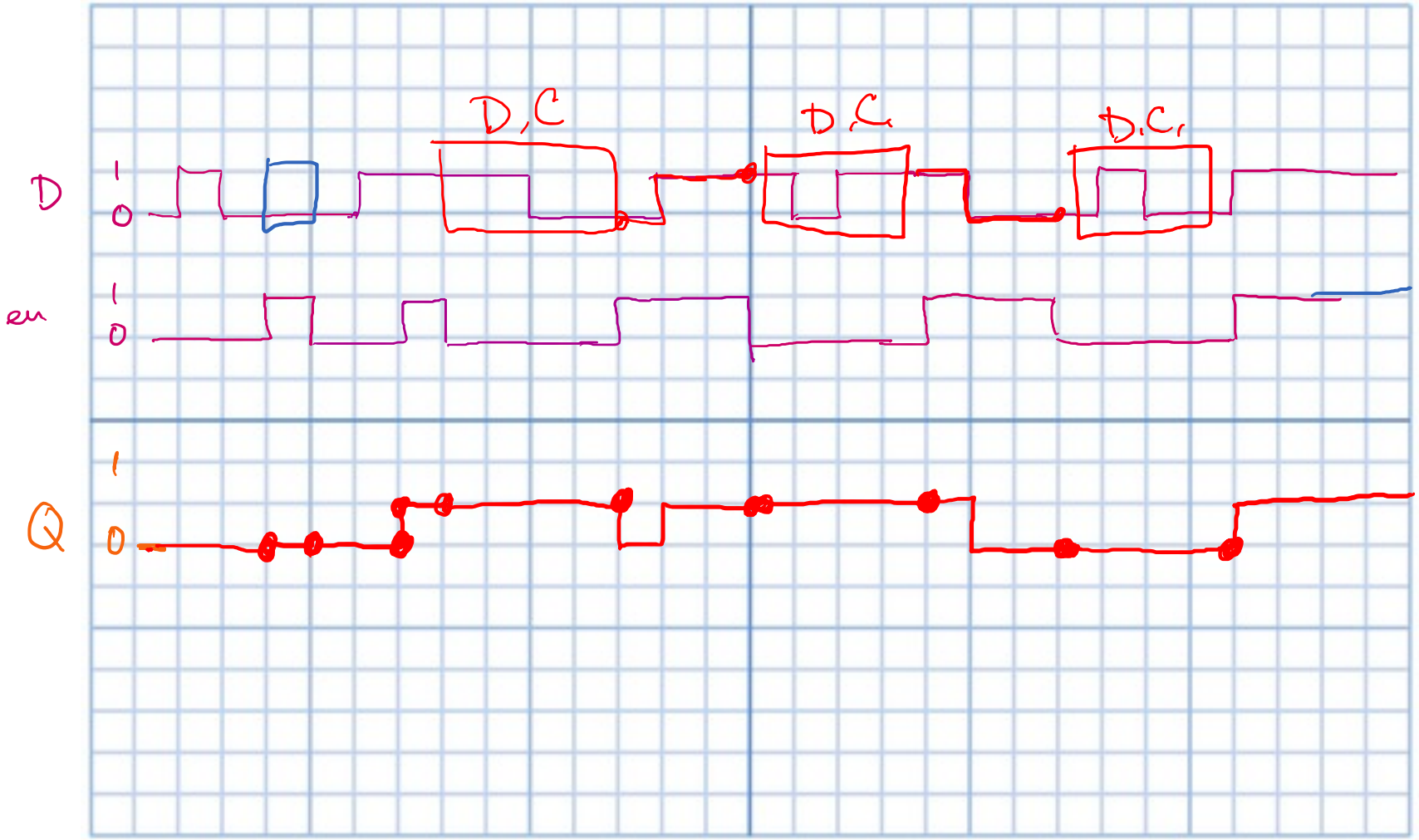
Inputs to D Latches



Inputs to D Latches

if en = 1, Q = D

if en = 0, Q = Q

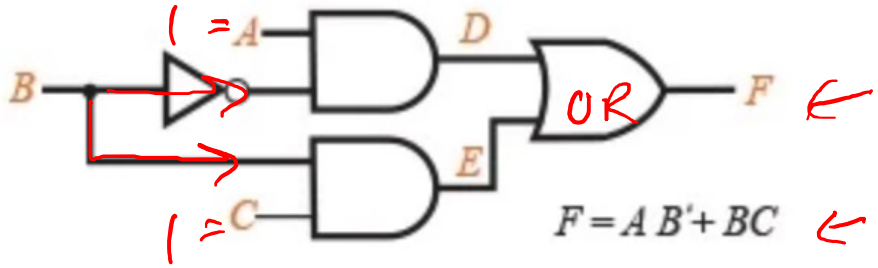


Glitches

→ unintended, short, errors in boolean logic

→ caused by gate delays

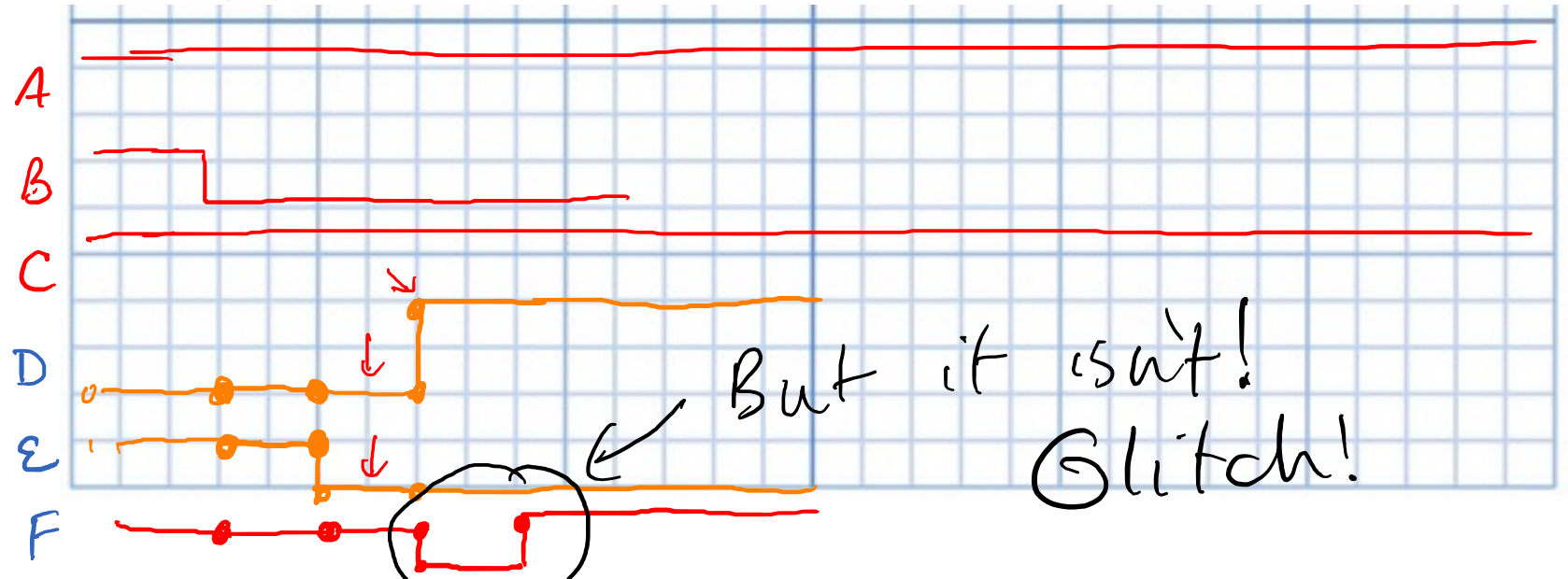
- Assume 10ps / gate.
- A=1, C=1, B falls
- What is F?



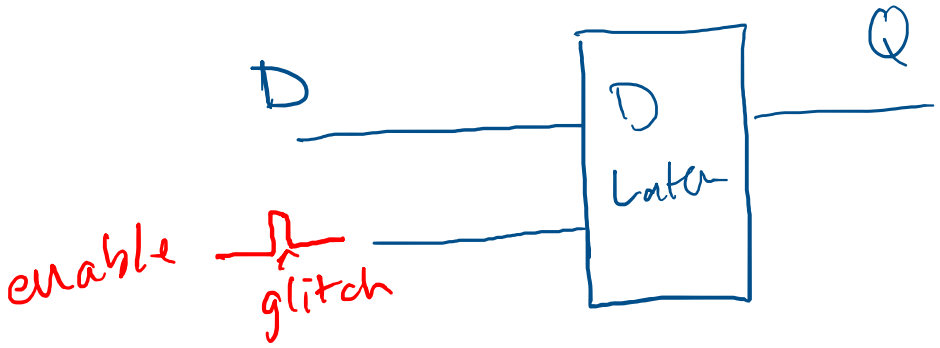
← should always be true!

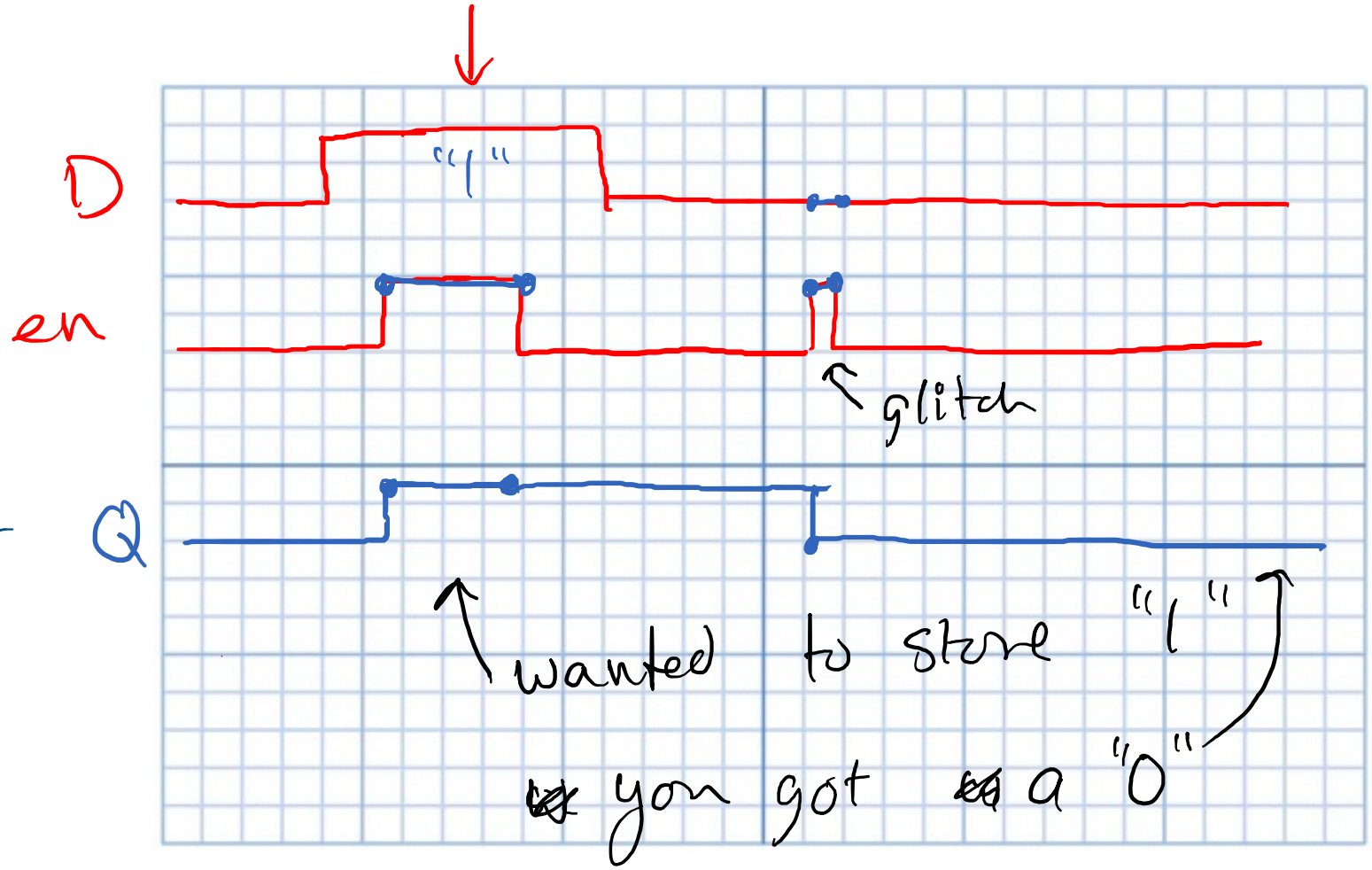
A=1 B=1 C=1

10ps



Glitches on D-Latches





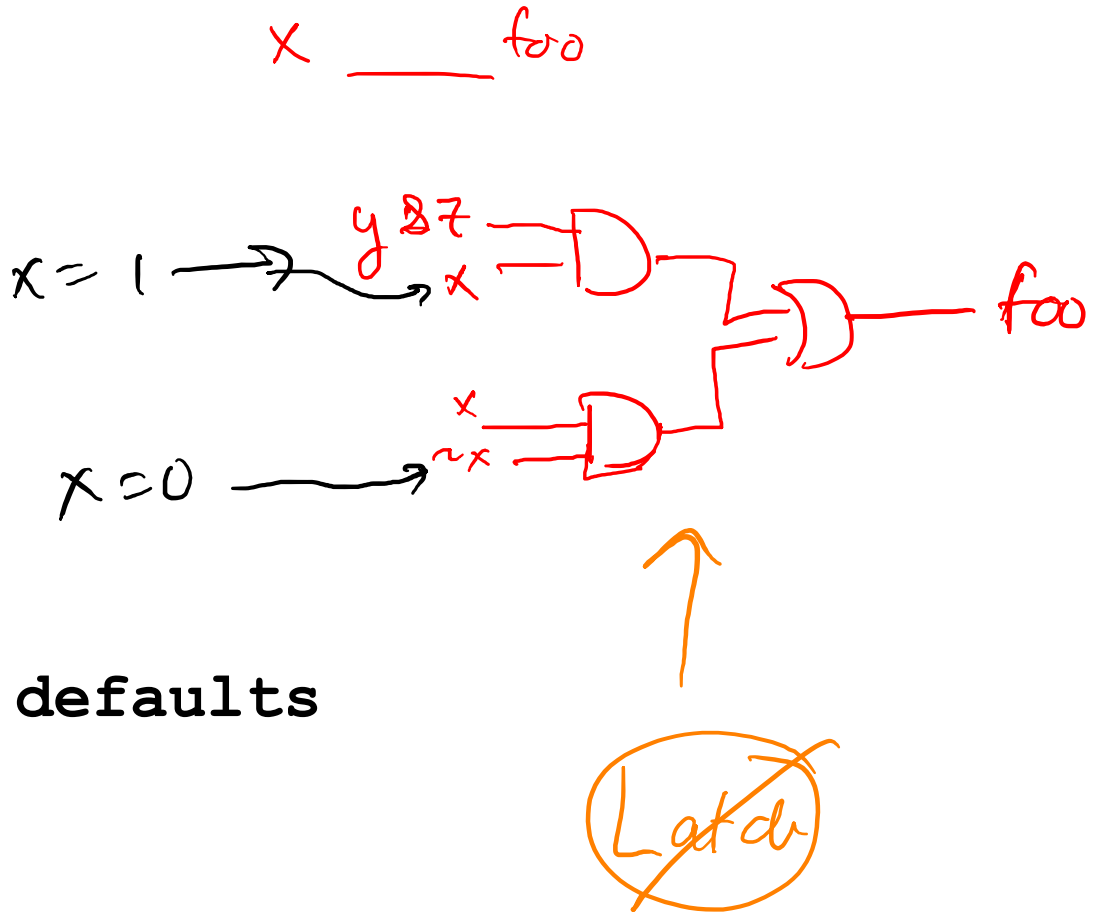
What's wrong here?

```
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`!

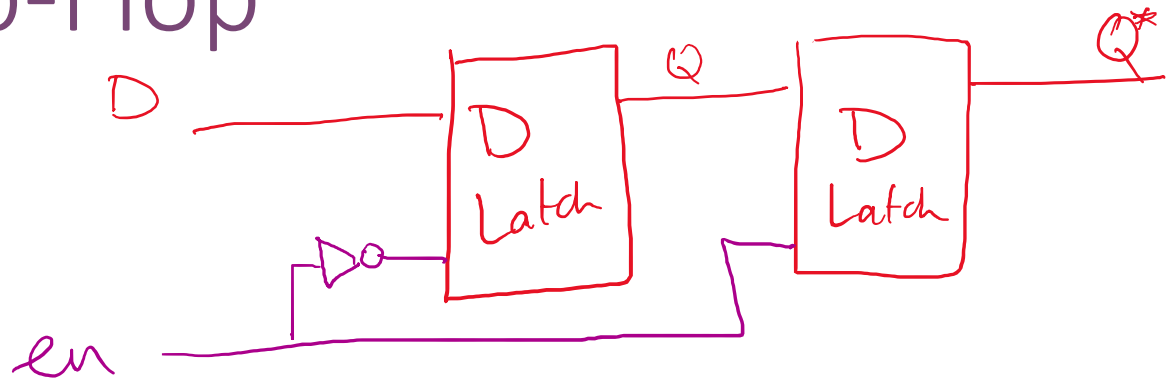
Always specify defaults for **`always_comb`**!

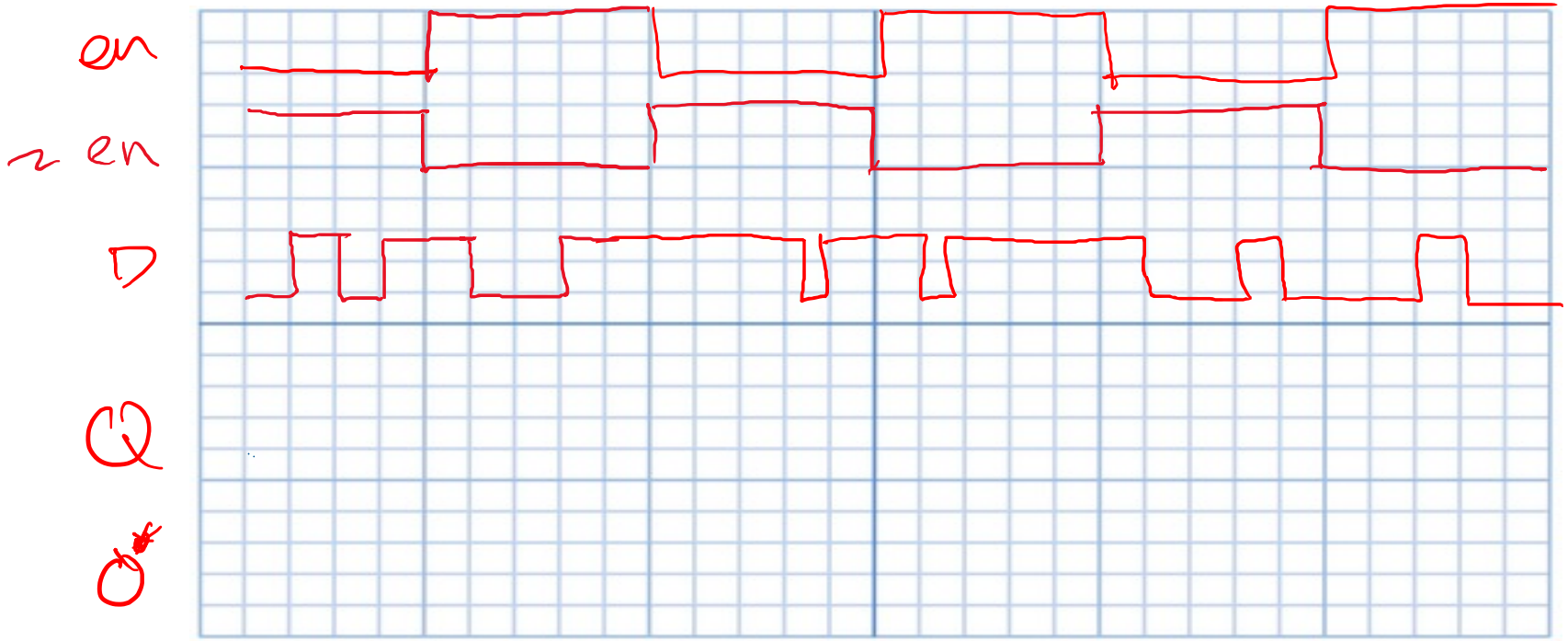
Always specify defaults for **`always_comb`**!

Always specify defaults for `always_comb`!

D Flip-Flop

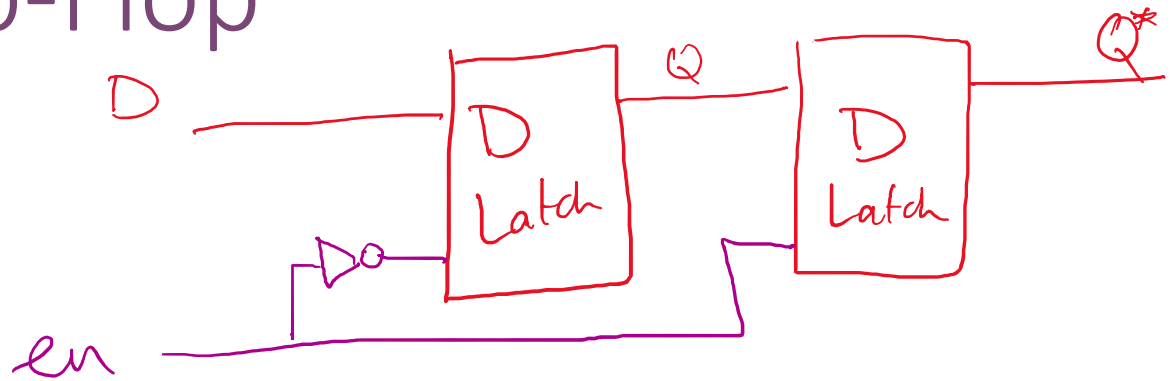
* no gate delays

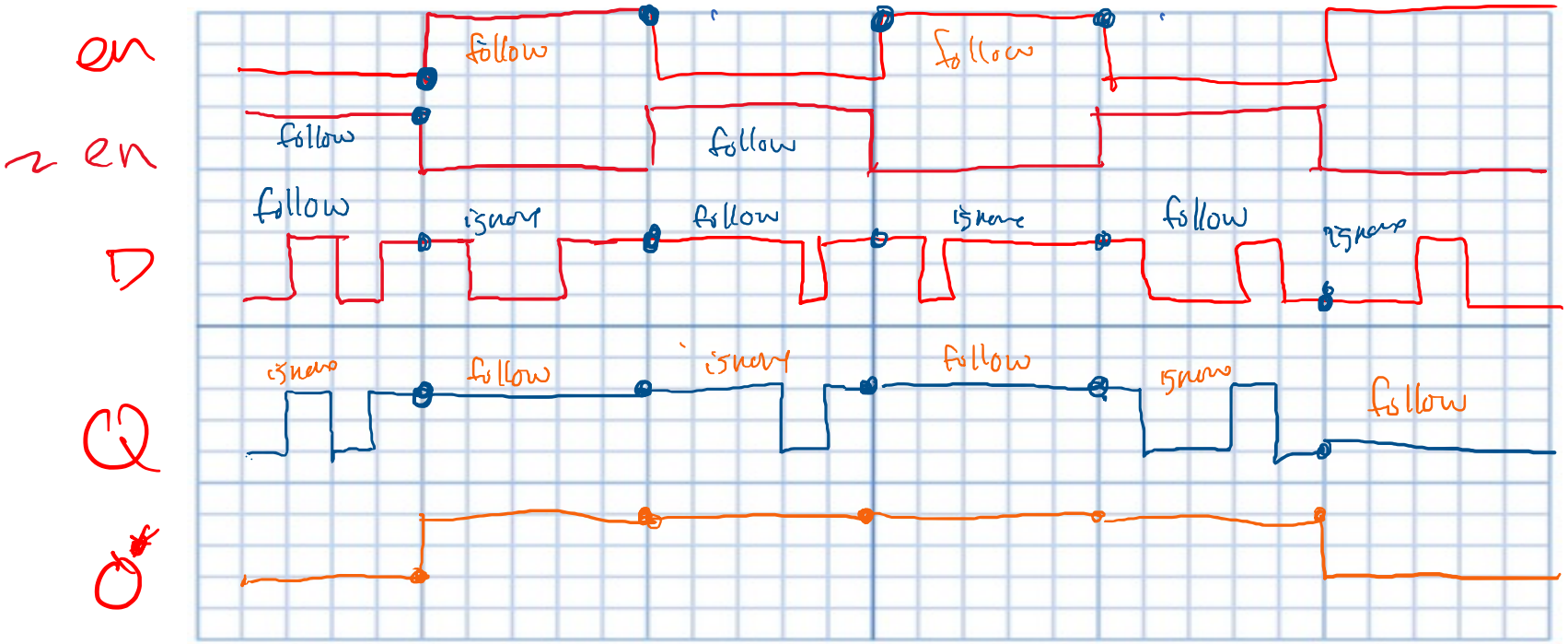




D Flip-Flop

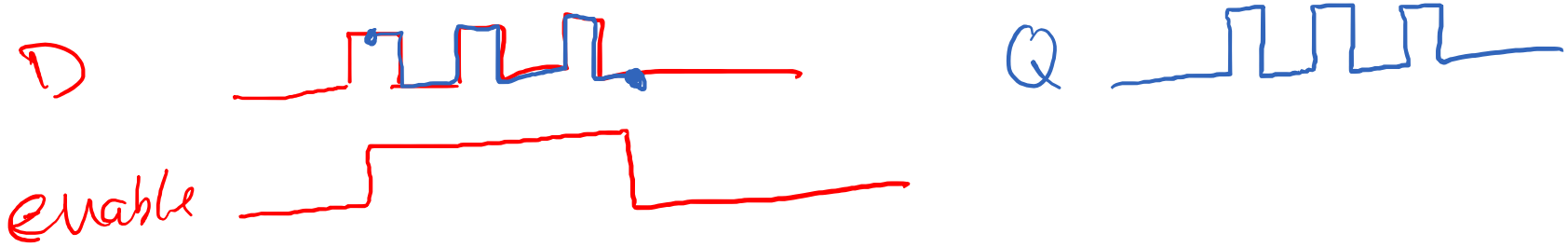
* no gate delays





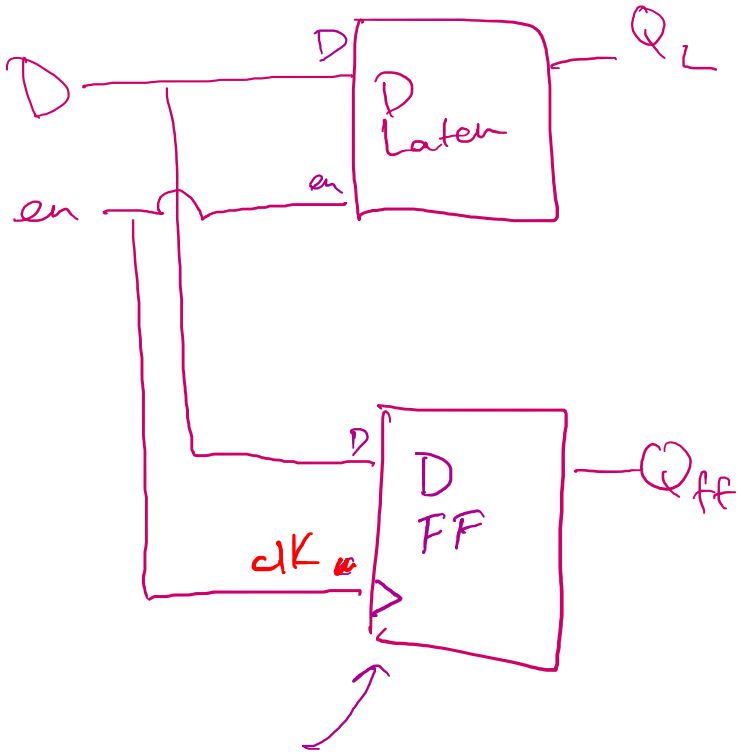
Levels vs. Edges

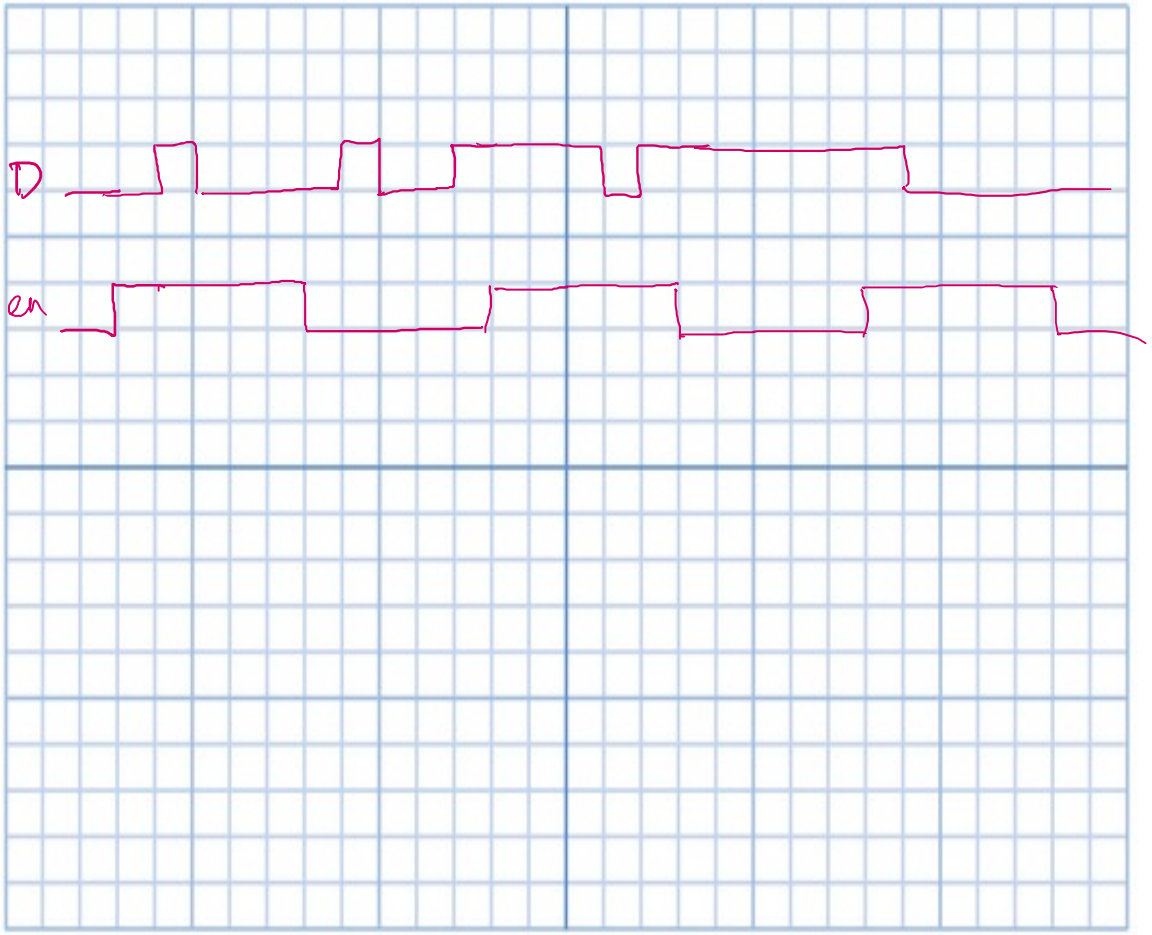
D Latch → Q follows D whenever enable is 1



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

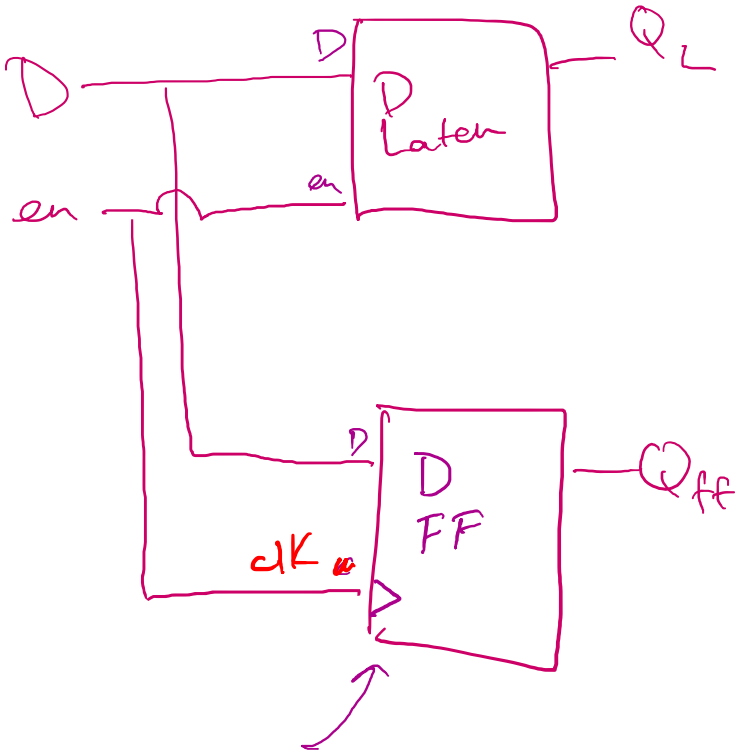
D Flip-Flop vs. D Latch

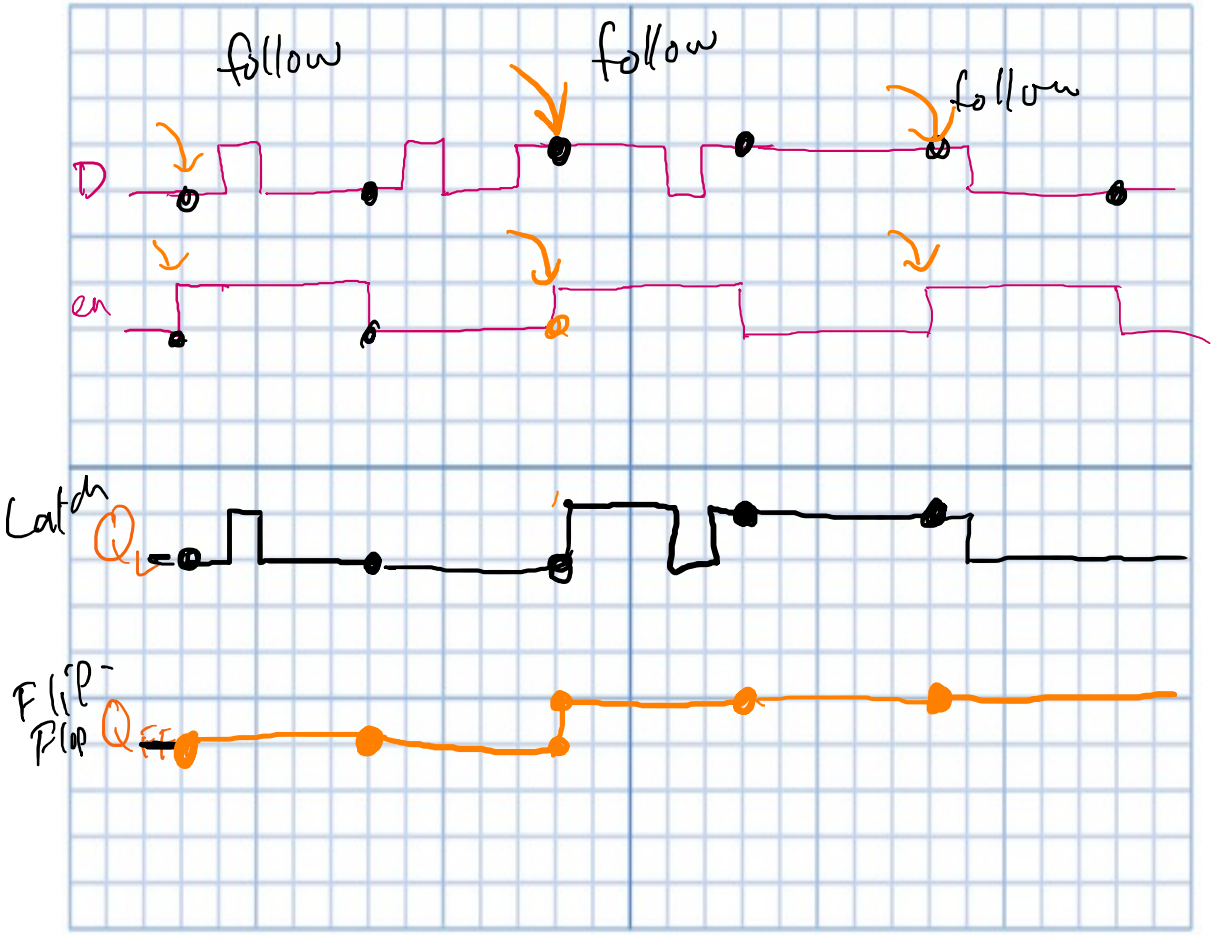




the " > " Symbol tells you it is Flip-Flop

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
```

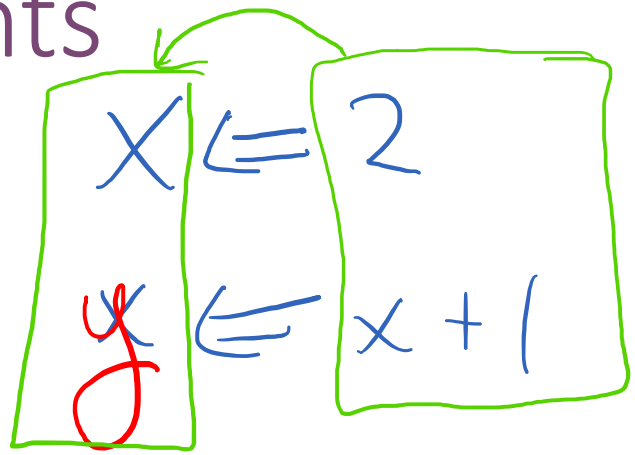
D Flip-Flop w/ Clock

```
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
```

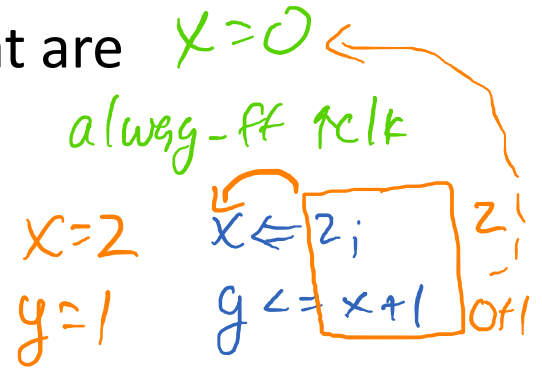
Blocking vs. NonBlocking Assignments

- 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 | 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

never hold state

+ defaults!

No Flip Flops!

NON-BLOCKING (<=) for always_ff

always for Flip Flops (always hold state)

D-FlipFlop w/Clock

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

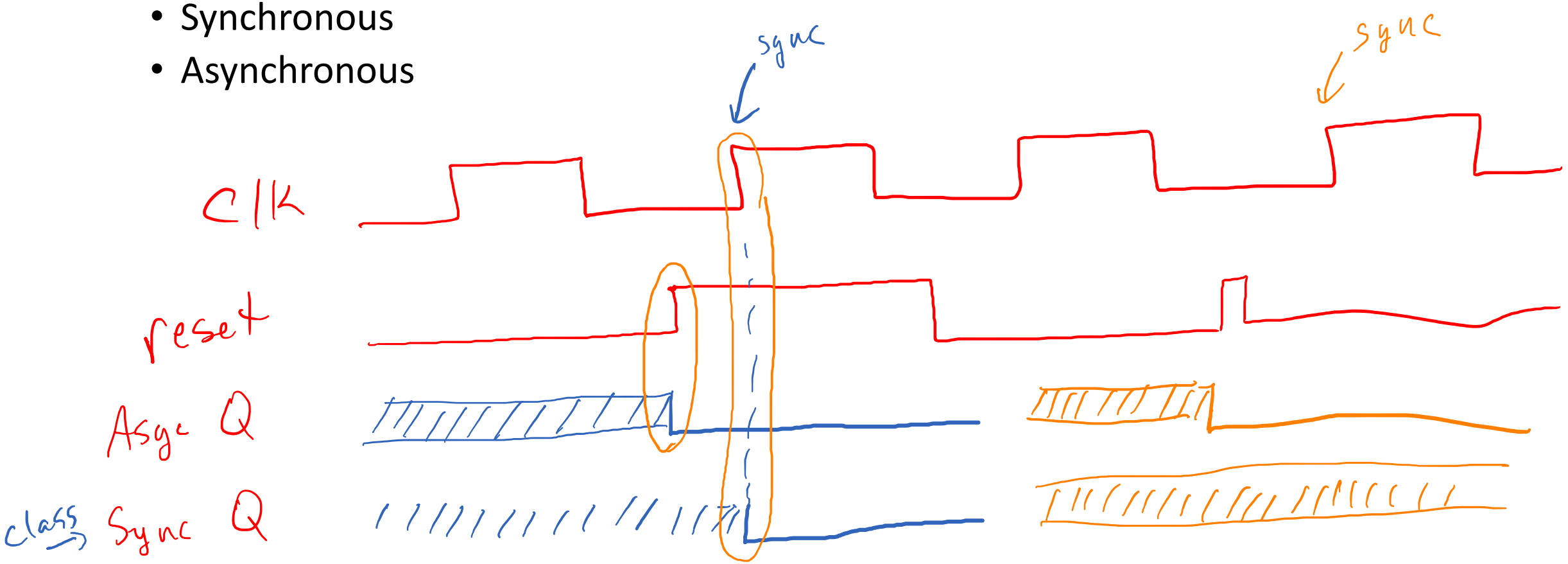
```
module d_ff (
     input d,              //data
     input clk,            //clock
     output reg q          //reg-isters hold state
);

      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

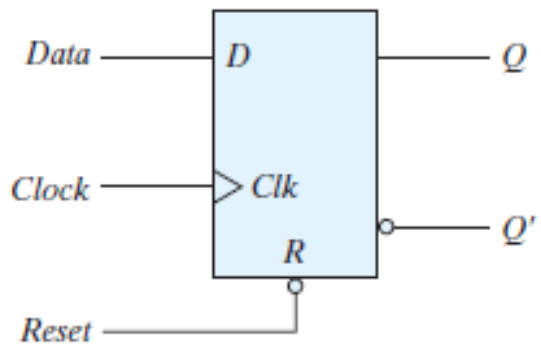
Direct inputs

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

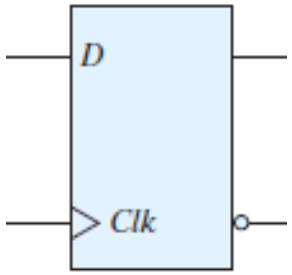
The input that sets the flip-flop to 1 is called *preset* or *direct set* .

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.



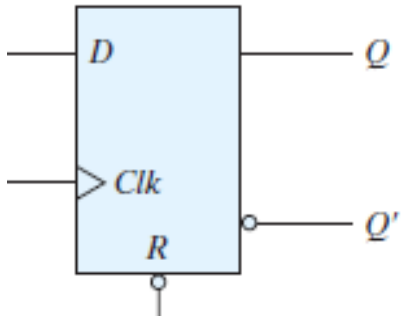
Verilog models of D flip-flop



Edge triggered D flip-flop:

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

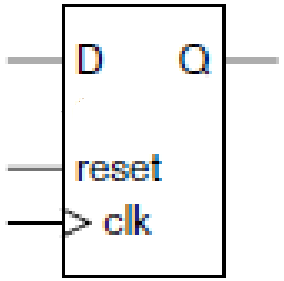
No reset ff



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;
```