ENGR 210 / CSCI B441

Math

Andrew Lukefahr

Announcements

- P6 is due Friday
  - Carry Tip: do 9 bit addition
    - {‘h0,a} + {‘h0,b)
  - Overflow:
    - Check the book

- P7 Saturating Counter is out

Autograder Demo

2-BeltAlarm Task

```
task checkAlarm(
    input kV, stPasV, sbPasV,
    input stDrvV, sbDrvV,
    input alarmV
    );

    k = kV; stPas=stPasV, sbPas=sbPasV;
    stDrv = stDrvV; sbDrv = sbDrvV;
    #10
    assert(alarm == alarmV) else
        $fatal (1, "bad alarm, expected:%b got:%b",
                    alarmV, alarm);
endtask
```

```
module TwoBeltAlarm(
     input k, st_pas, sb_pas,
     input st_drv, sb_drv
     output alarm
);
     logic al_pas, al_drv;

     BeltAlarm ba_drv(k, st_drv, sb_drv, al_drv);
     BeltAlarm ba_pas(.k(k), .p(st_pas),
          .s(sb_pas), .alarm(al_pas));

     assign alarm = al_pas | al_drv;
endmodule
```

‘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 hold state (but doesn’t have to)

UPDATE: ‘wire’ vs ‘logic’

SystemVerilog (NEW) Rules:

Just use ‘logic’*

* ***EXCEPT***

```
logic foo = ‘h42;  (BAD)  (OK)
logic foo = a & b;  (BAD – Initial a & b only)

wire foo = a & b;   (OK)

logic foo;
assign foo = a & b;  (OK)
```

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
```

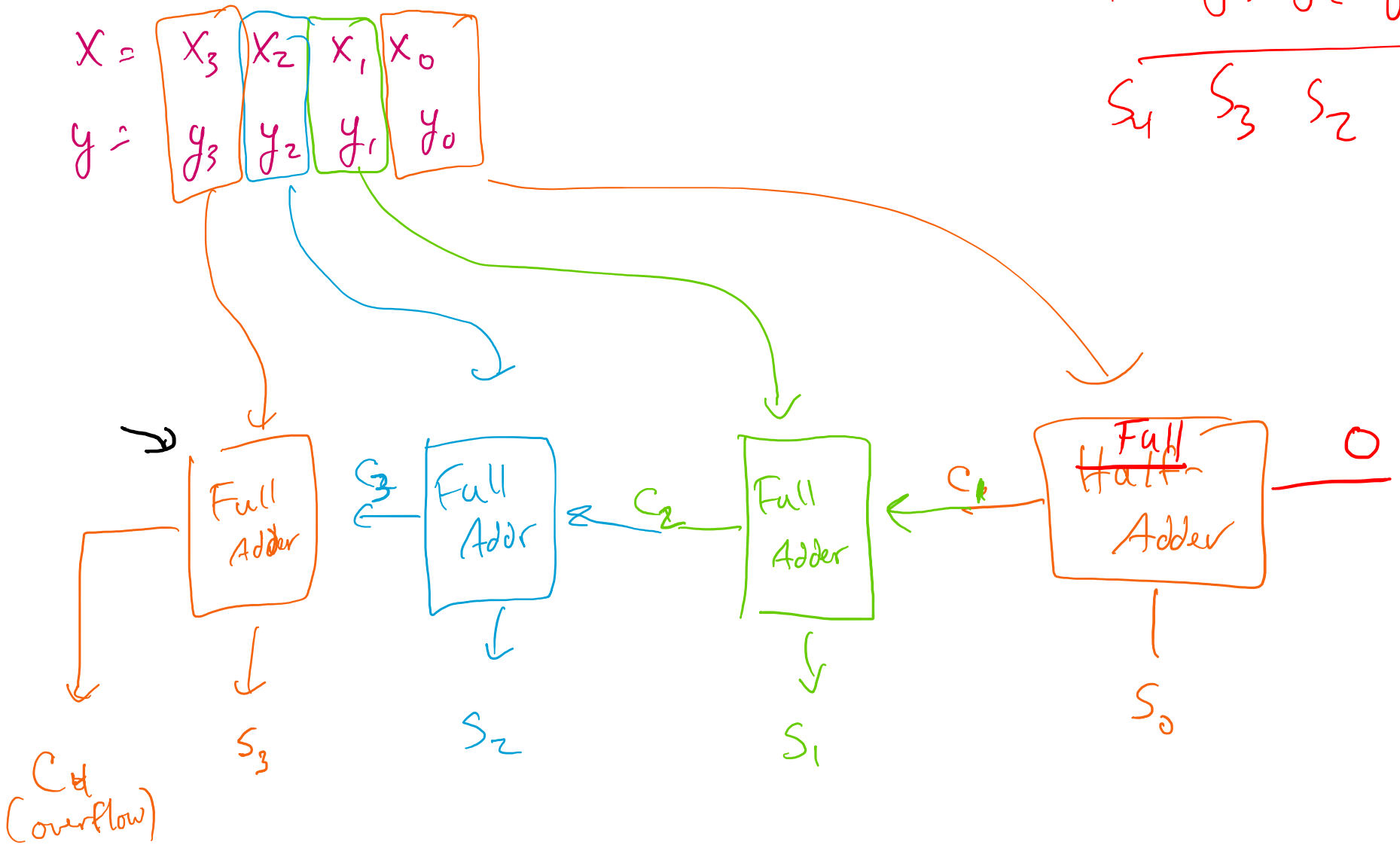
Always specify defaults for **always_comb**!

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

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

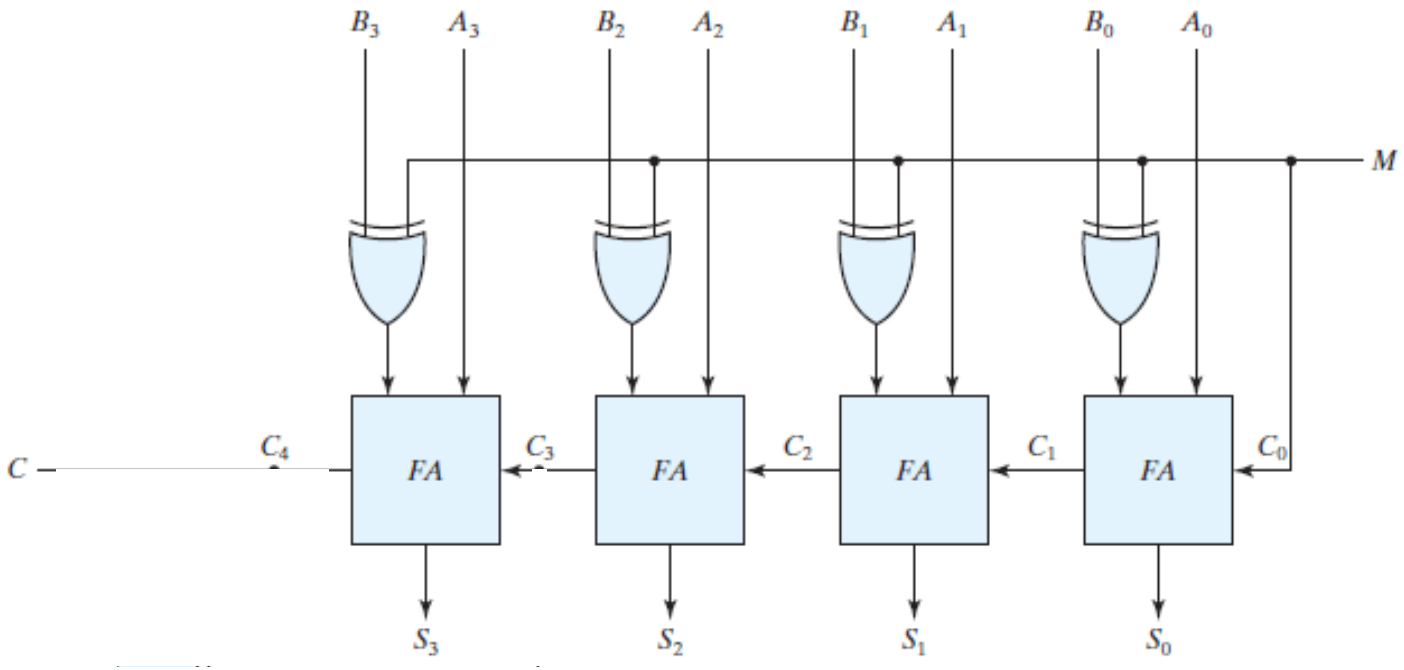
Ripple-Carry Adder

$$
\begin{array}{rcccccc}
x+y & = & x_3 & x_2 & x_1 & x_0 \\
 & + & y_3 & y_2 & y_1 & y_0 \\
\hline
 & & s_4 & s_3 & s_2 & s_1 & s_0
\end{array}
$$



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?

$$\begin{array}{r} 10 \\ +\ 8 \\ \hline 18 \end{array} \qquad \begin{array}{r} {}^{1}1010 \\ +\ 1000 \\ \hline 10010 \end{array}$$

carry

sum

unsigned = carry out bit "overflow"

Signed

$$\begin{array}{r} 5 \\ +\ 6 \\ \hline 11 \end{array} \qquad \begin{array}{r} {}^{0\ 1}0101 \\ 0110 \\ \hline 0\ \boxed{1011} \end{array}$$

signed $= -(0100+1) = -(0101) = -5$ ← overflow!

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

$6 = 0110$

carry $= 0 \Rightarrow$ No overflow?

Overflow for signed numbers?

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

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

Overflow for signed numbers?

carry is same

$$
\begin{array}{rrcrcr}
-2 & -(0010) & = & 1101+1 & = & \boxed{11}10 \\
+-1 & -(0001) & = & 1110+1 & = & +\ 1111 \\
\hline
 & & & & & 10001
\end{array}
$$

$\Rightarrow$ No overflow

$$
\begin{array}{rcl}
 & & 1110 \\
+2 & = & 0010 \\
+-1 & = & +\ 1111 \\
\hline
 & & 10001
\end{array}
$$

carry is same

$\Rightarrow$ No overflow

$$
\begin{array}{lll}
-7 & -(0111) = 1000+1 = & 1001 \\
+\underline{-7} & -(0111) = 1000+1 = & +\,1001 \\
\hline
 & & 10010
\end{array}
$$

carry is Different

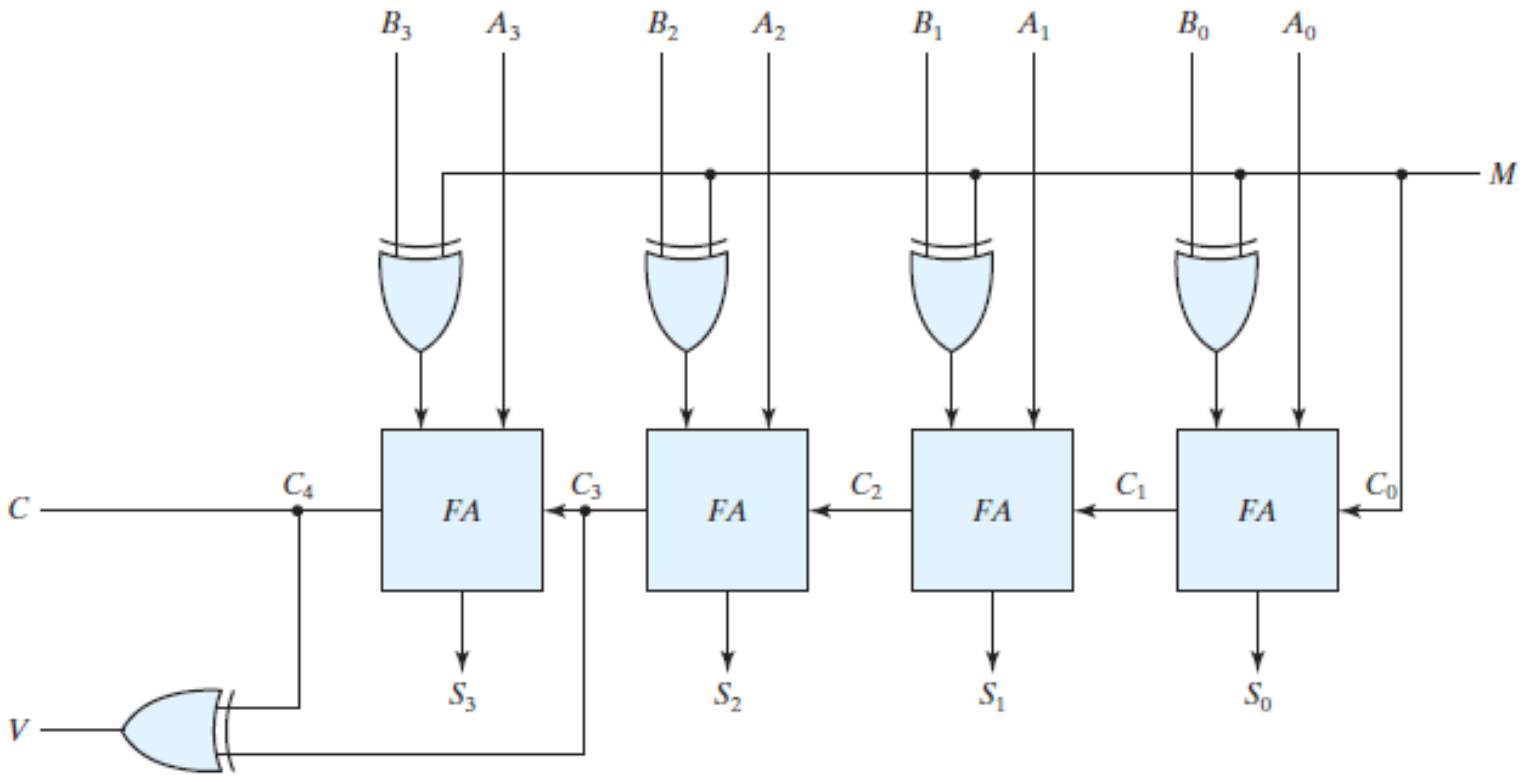
=> overflow

$$
\begin{array}{lll}
 & & 0111 \\
+7 & & 0111 \\
+\underline{+7} & & +\,0111 \\
\hline
 & & 01110
\end{array}
$$

carry is different

=> overflow

Adder with overflow detection



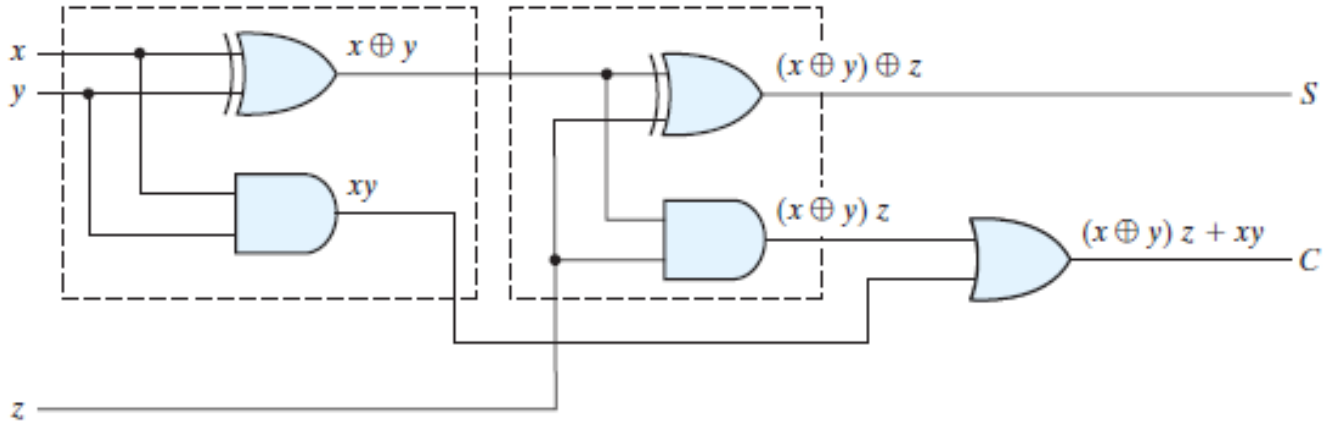
Gate Delay

- 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

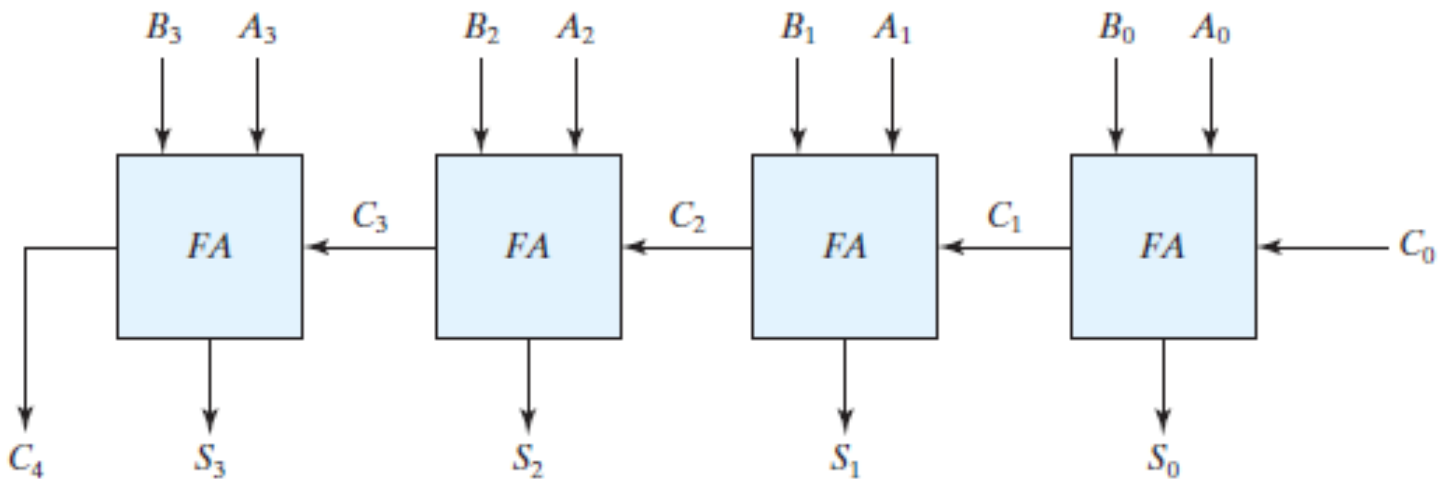
- Assume 5ps/gate



- 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?

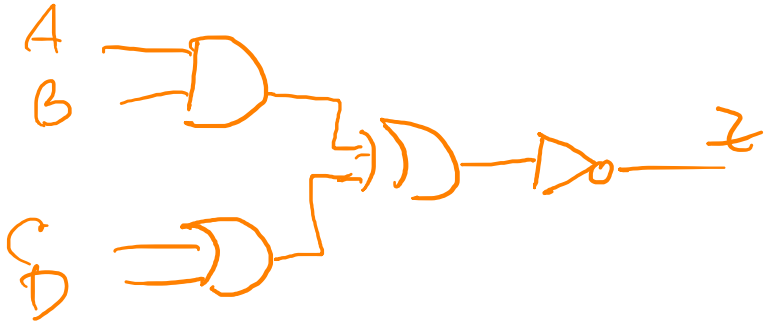
- Google “Carry Look-Ahead Adder”

WARNING: MAJOR TOPIC SHIFT

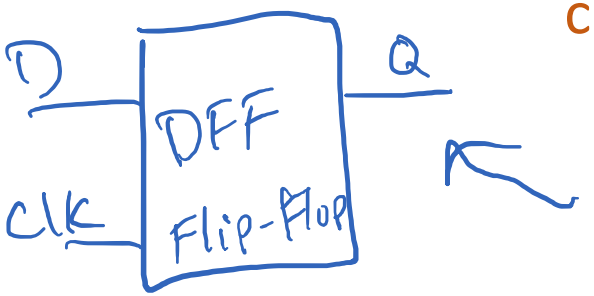
SEQUENTIAL LOGIC

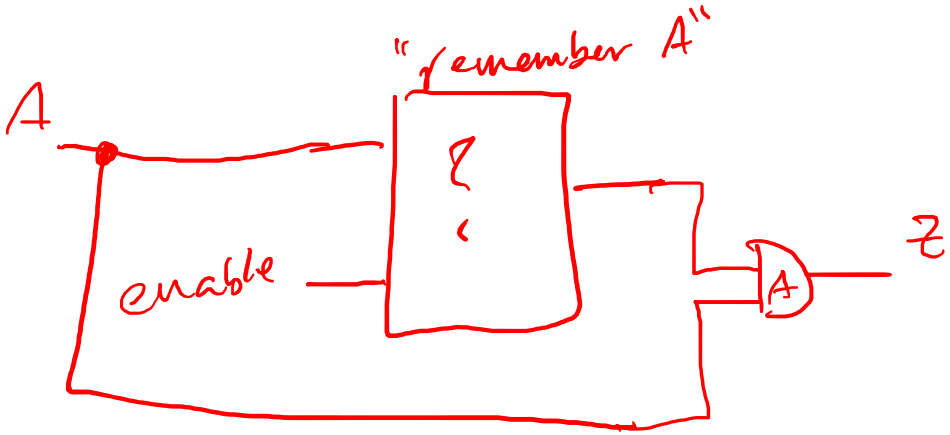
Sequential vs. Combinational

- Combinational Logic
  - The output is a combination of the current inputs only

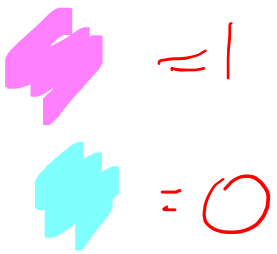


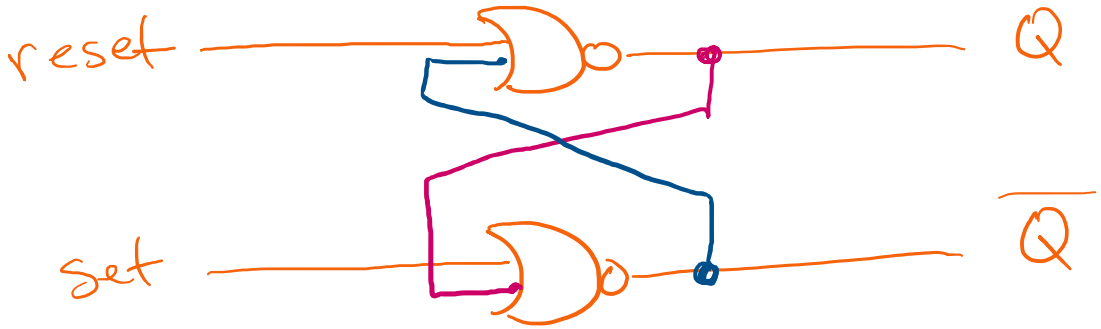
- Sequential Logic
  - The output is a combination of the current **and past** inputs





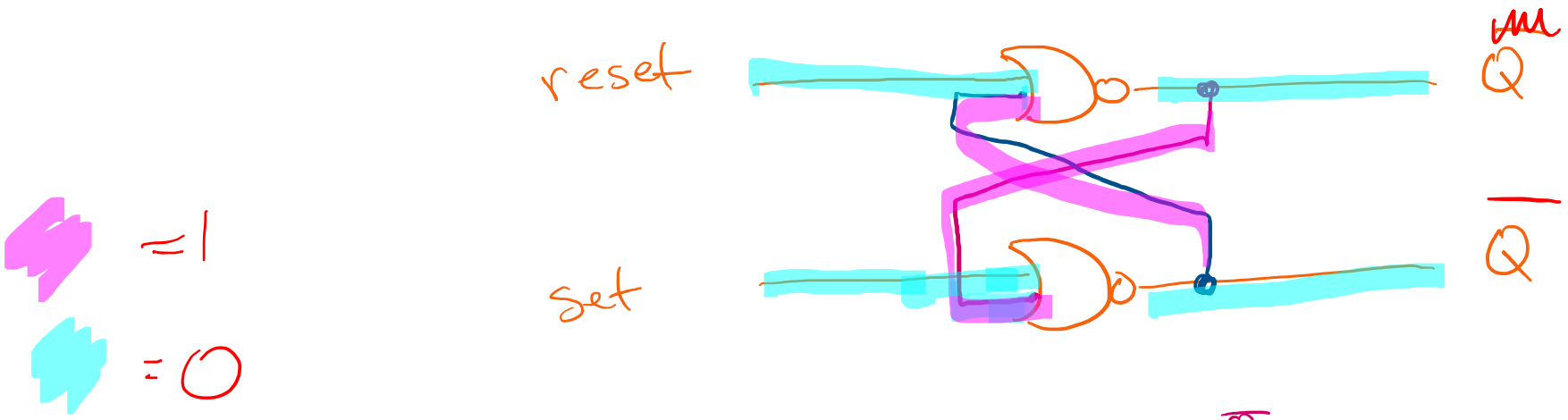
SR Latch





reset	set	Q	$\overline{Q}$

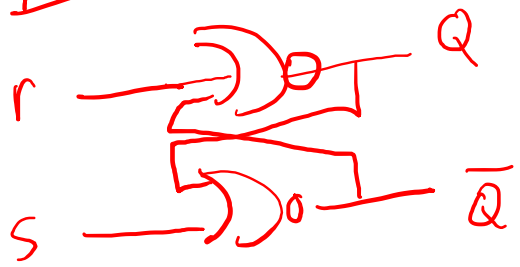
SR Latch w/ S=1 & R=1



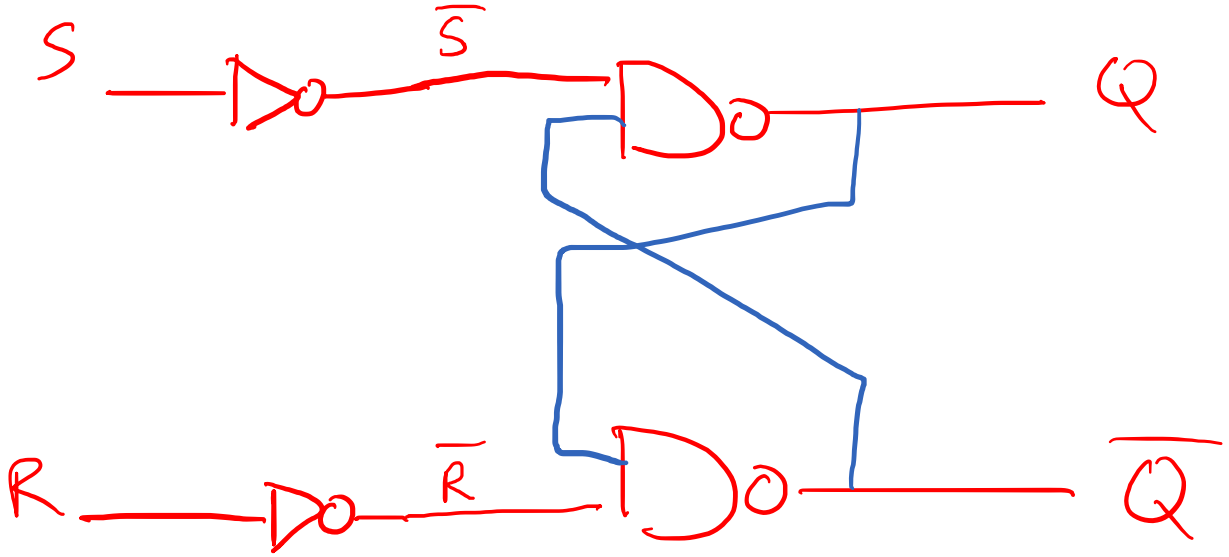
set	reset	Q	$\overline{Q}$
1	1	0	0
0	0	0,1,0,1,0,1,0,1,.....	

SR Latch w/NAND gates

NOR



NAND

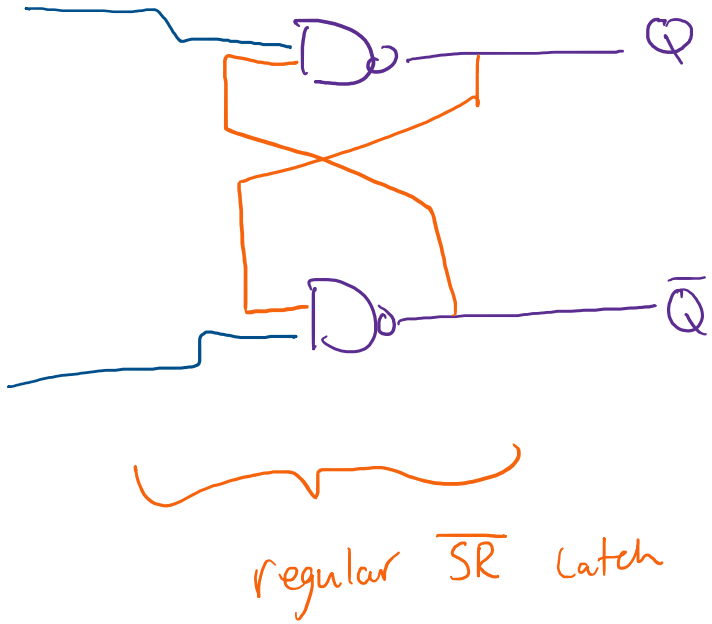


→ better setup for Flip-Flops

→ easier for me to draw

SR Latch with Enable

Prevent changes in S & R from changing circuit output



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

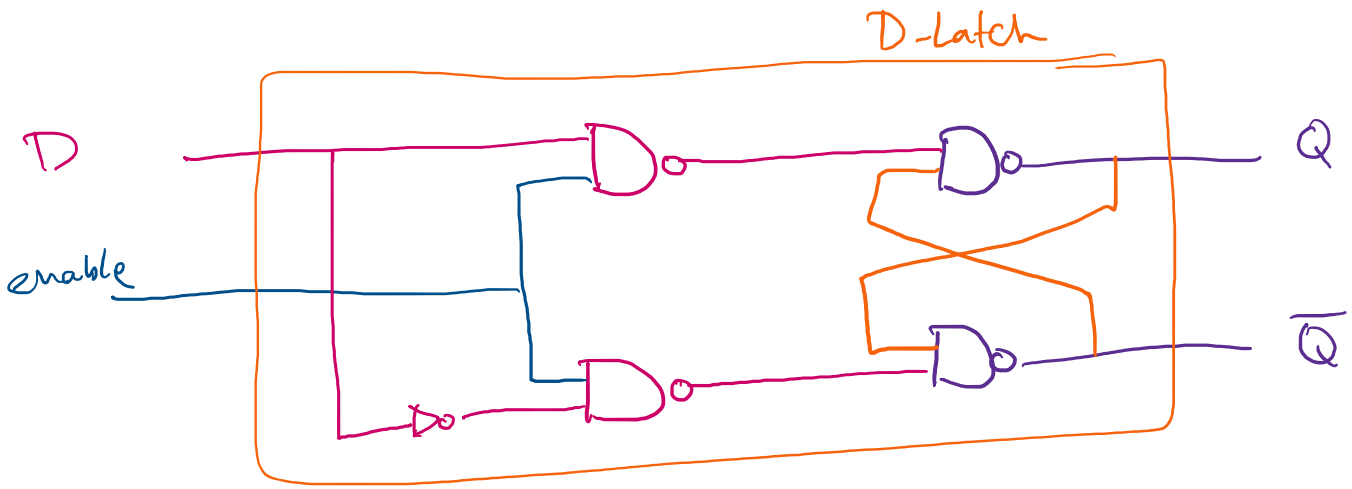
* assume no S=1 & r=1

D-Latch

D-Latch

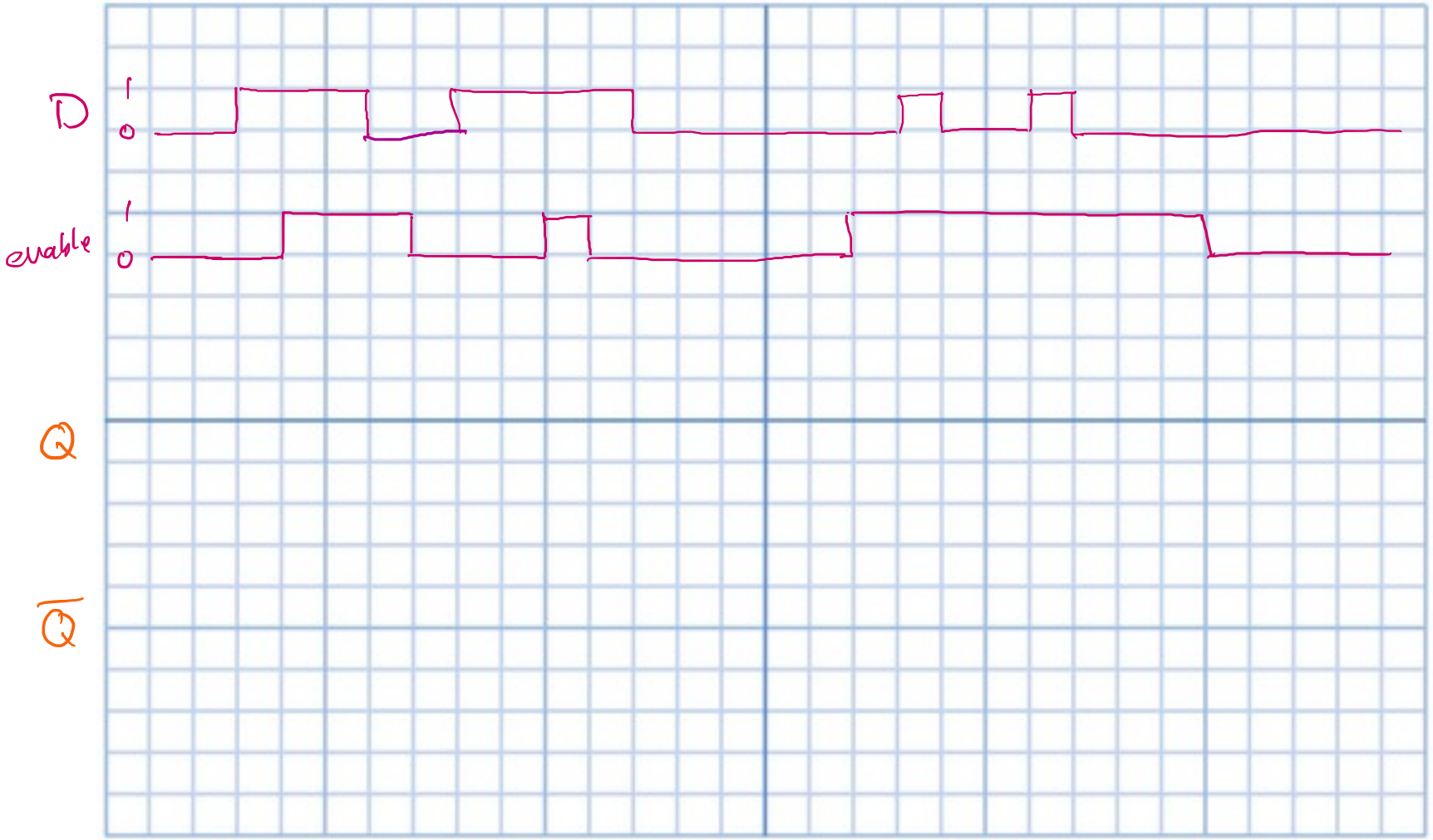
= 1

= 0

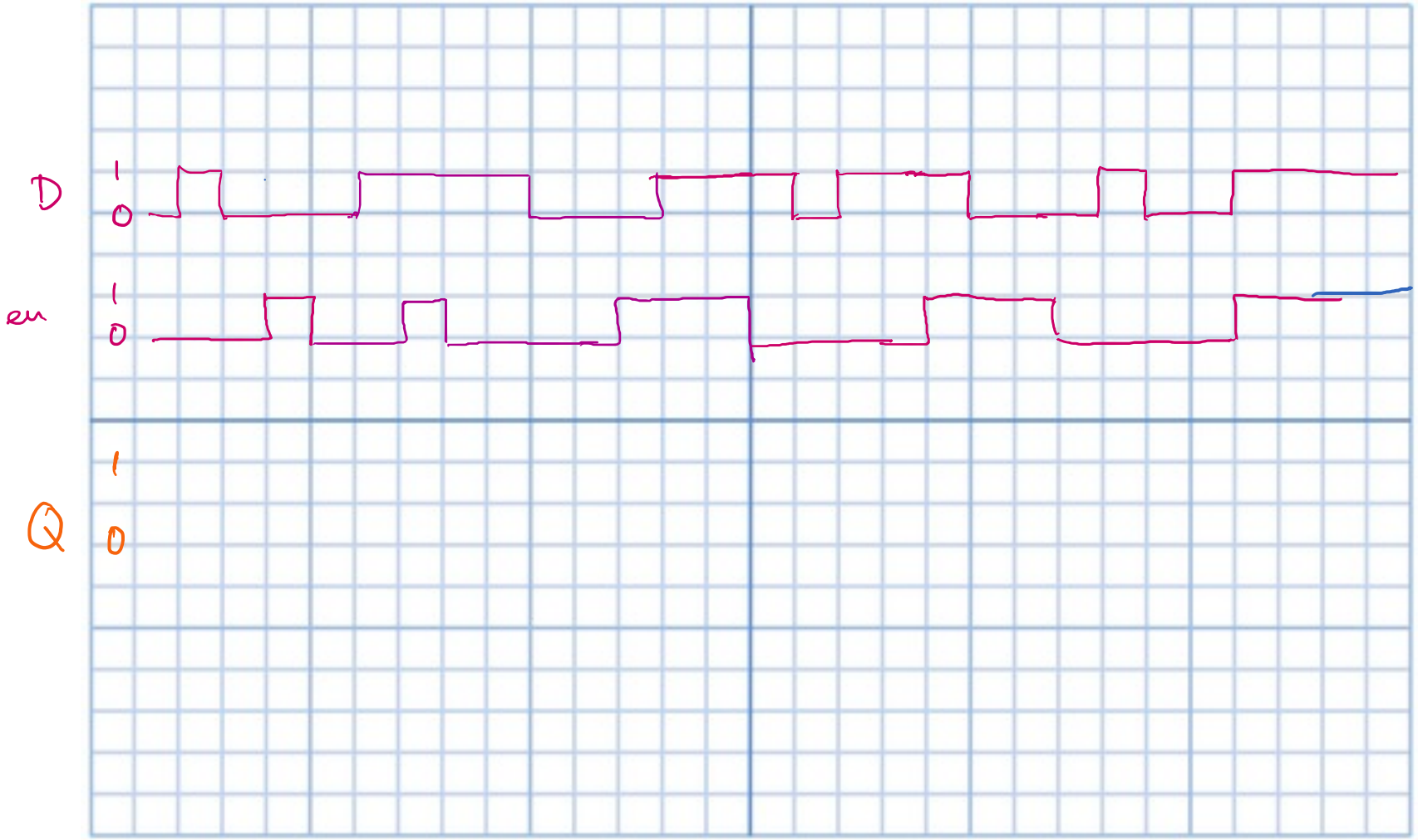


D	en	Q	$\overline{Q}$

Inputs to D Latches



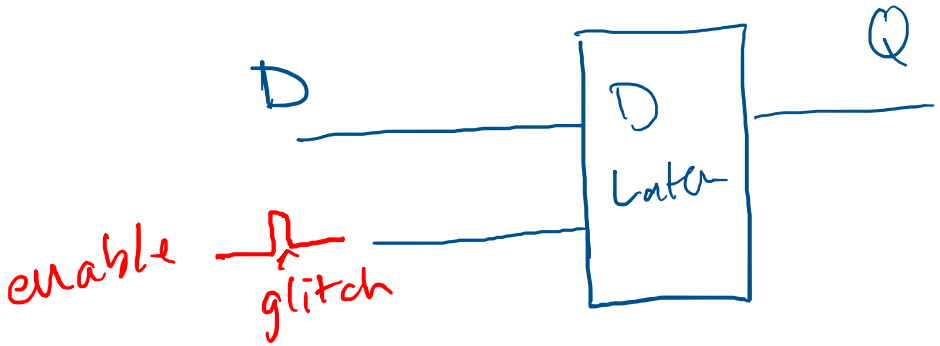
Inputs to D Latches

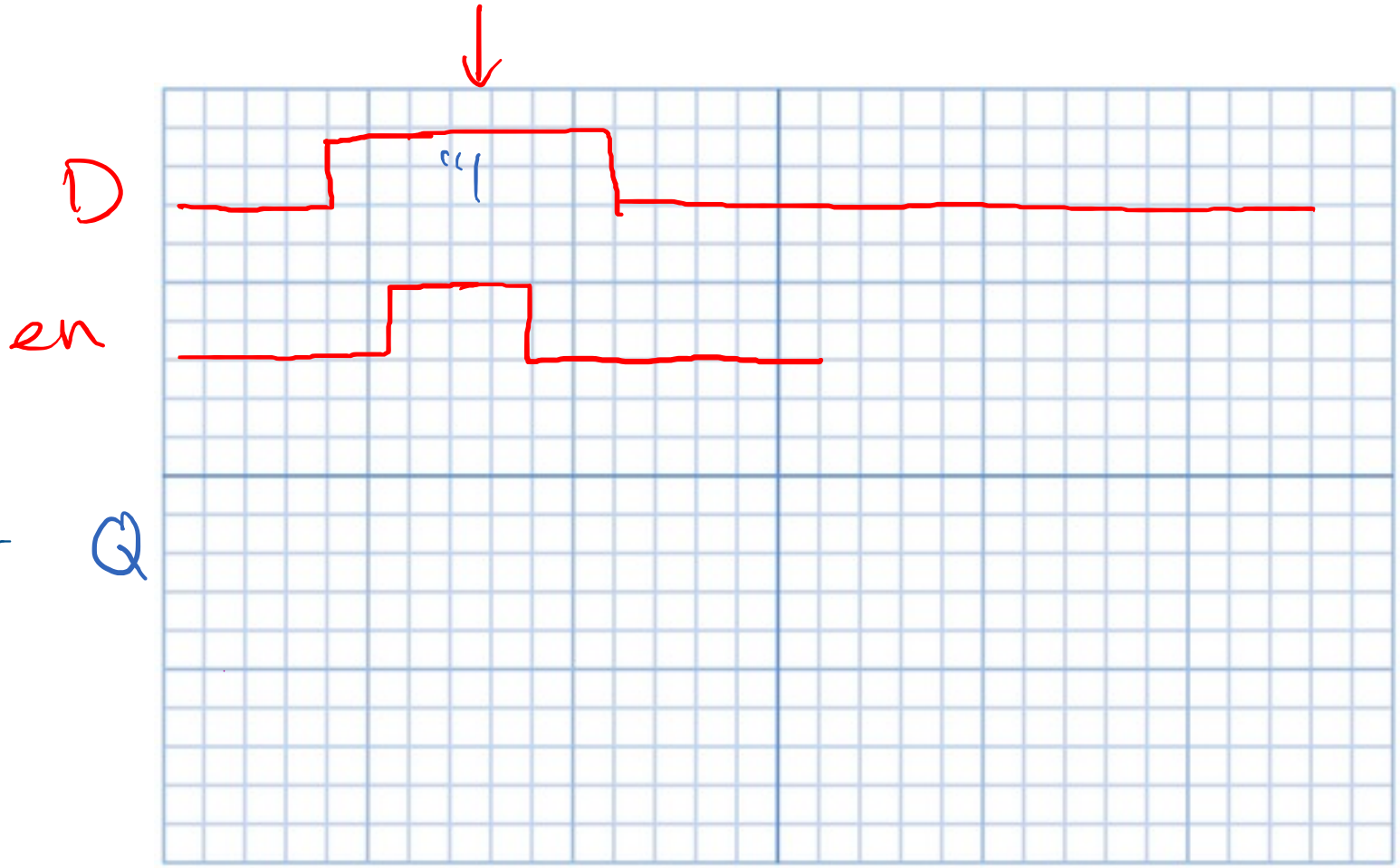


Glitches

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

Glitches on D-Latches





What’s wrong here?

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

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

Inferred Latches

```
logic 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
```

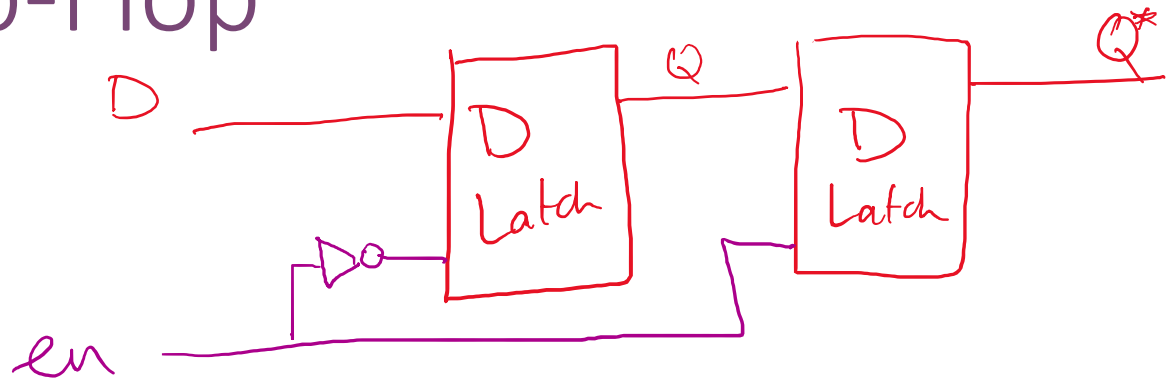
What if `x == 0`? `foo = bar = x`!

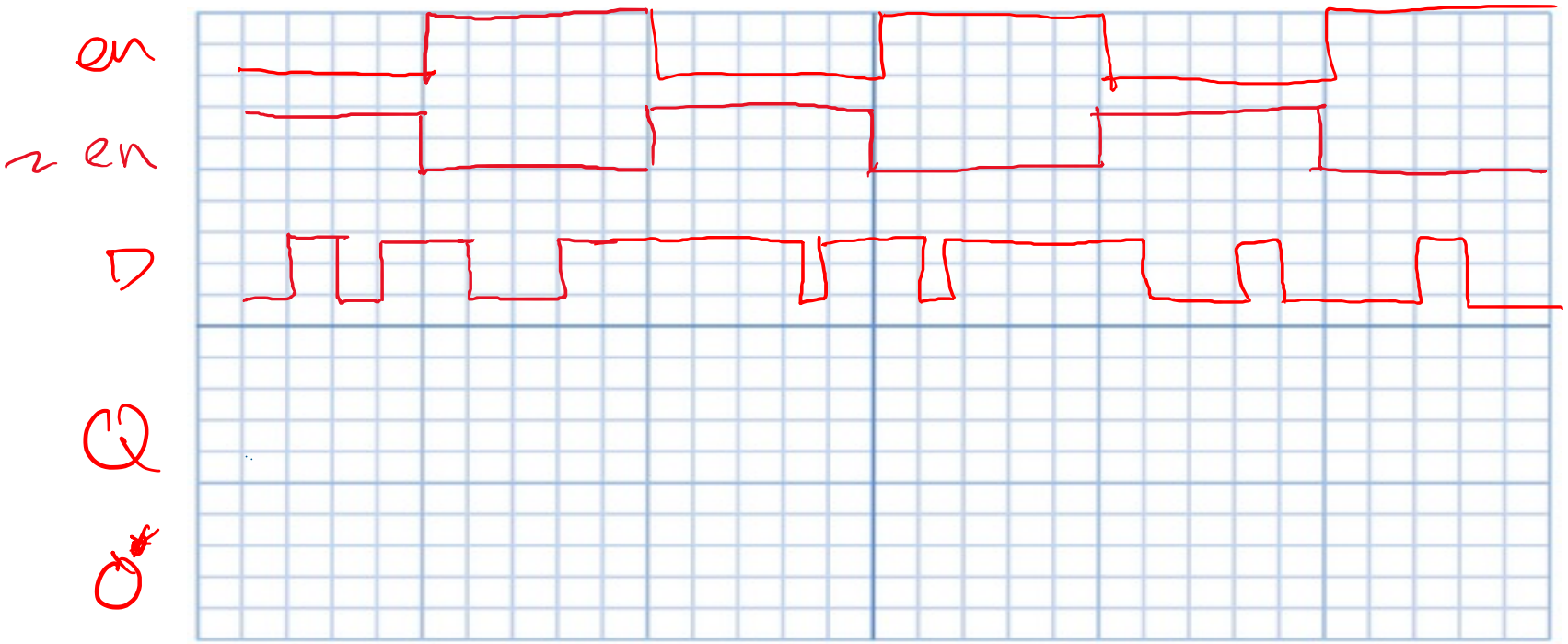
Always specify defaults for **`always_comb`**!

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

D Flip-Flop

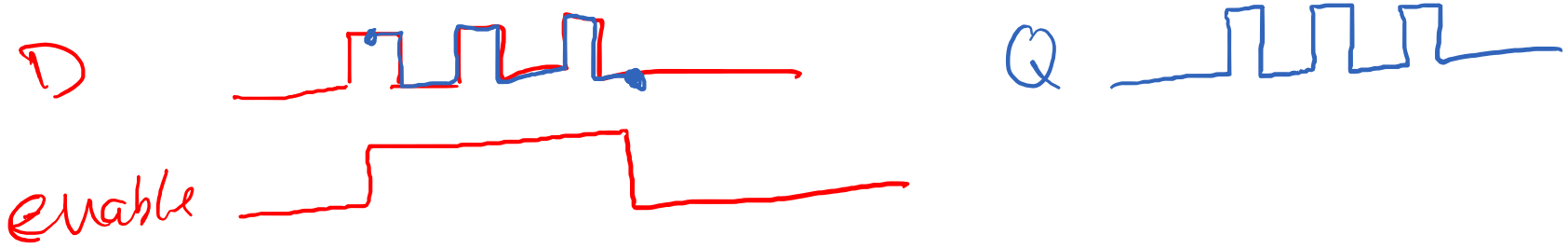
* no gate delays





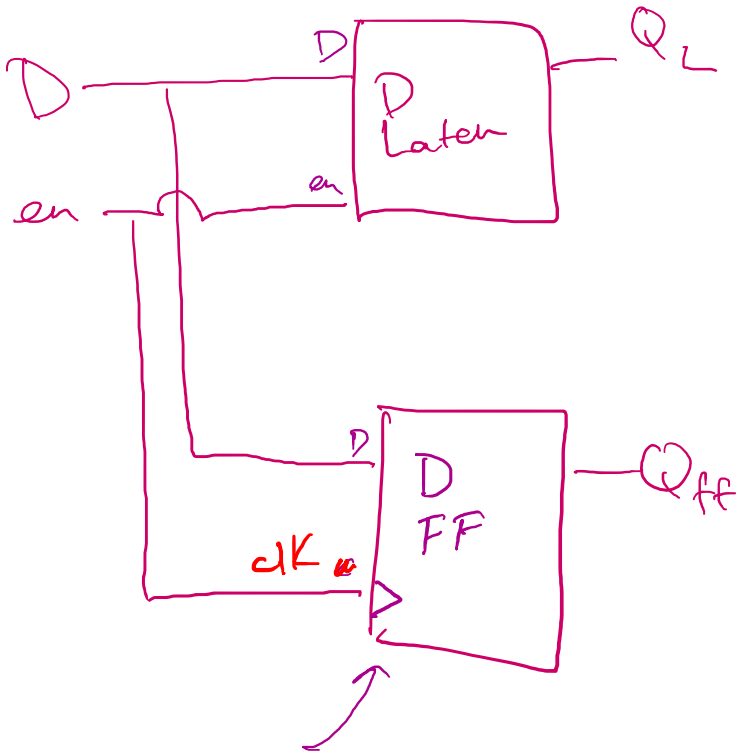
Levels vs. Edges

D Latch → Q follows D whenever enable is 1

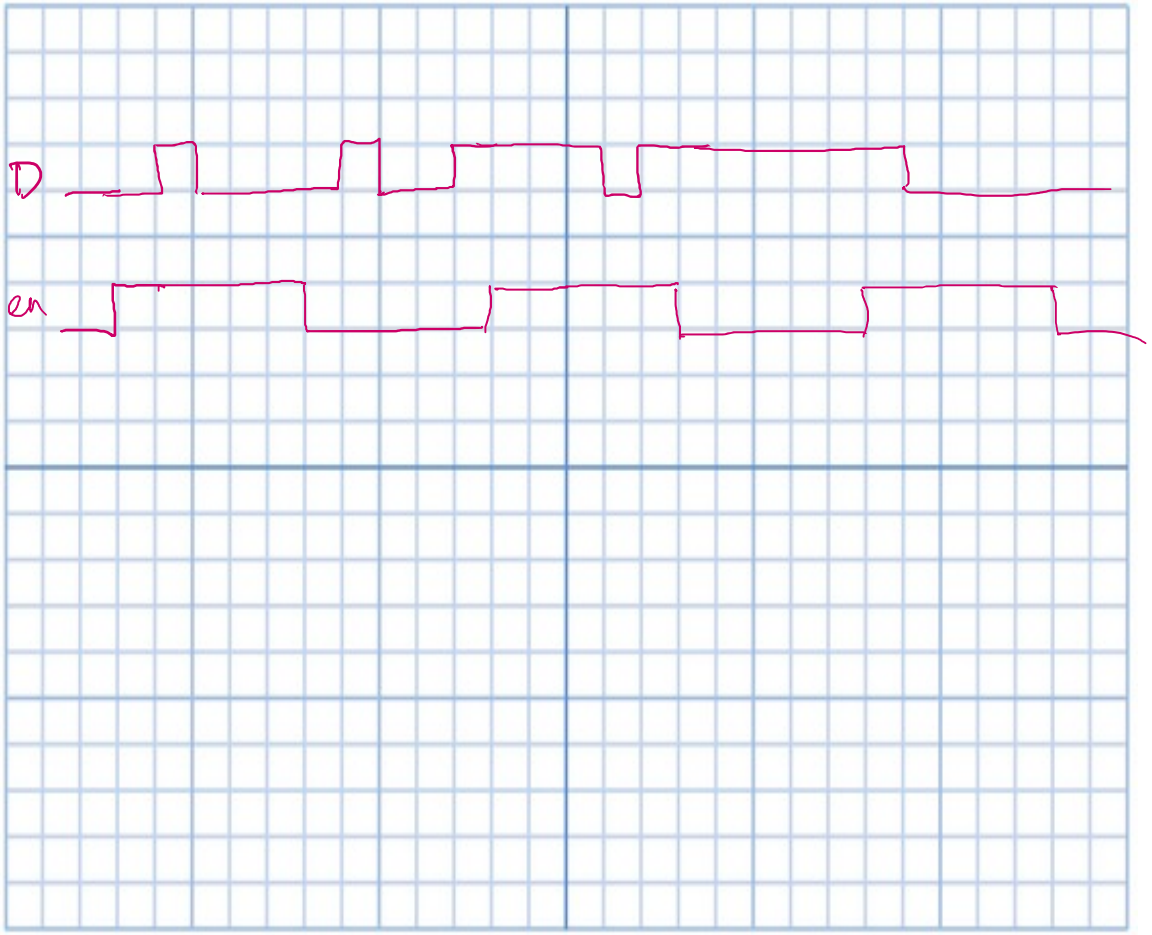


D flip-flop → Q follows D on rising edge of enable

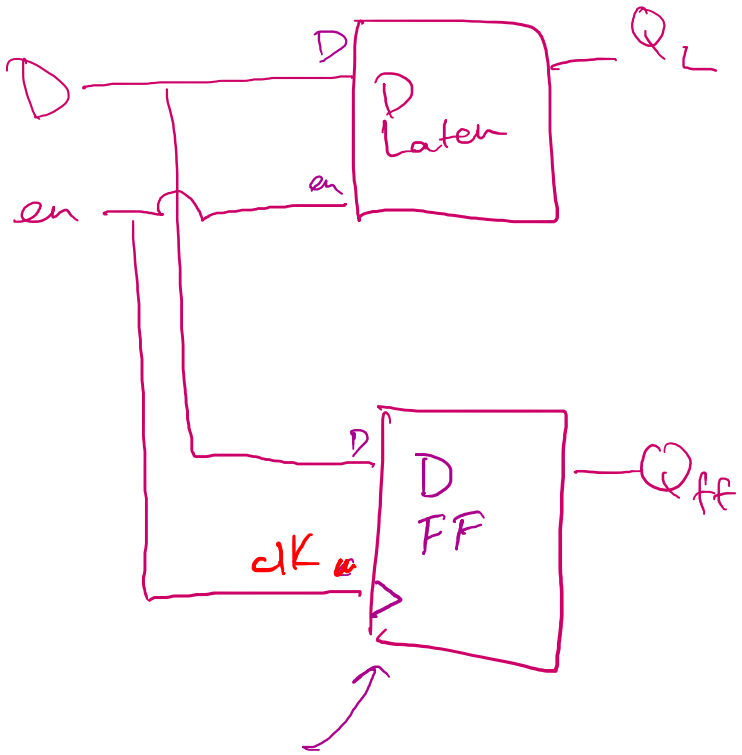
D Flip-Flop vs. D Latch

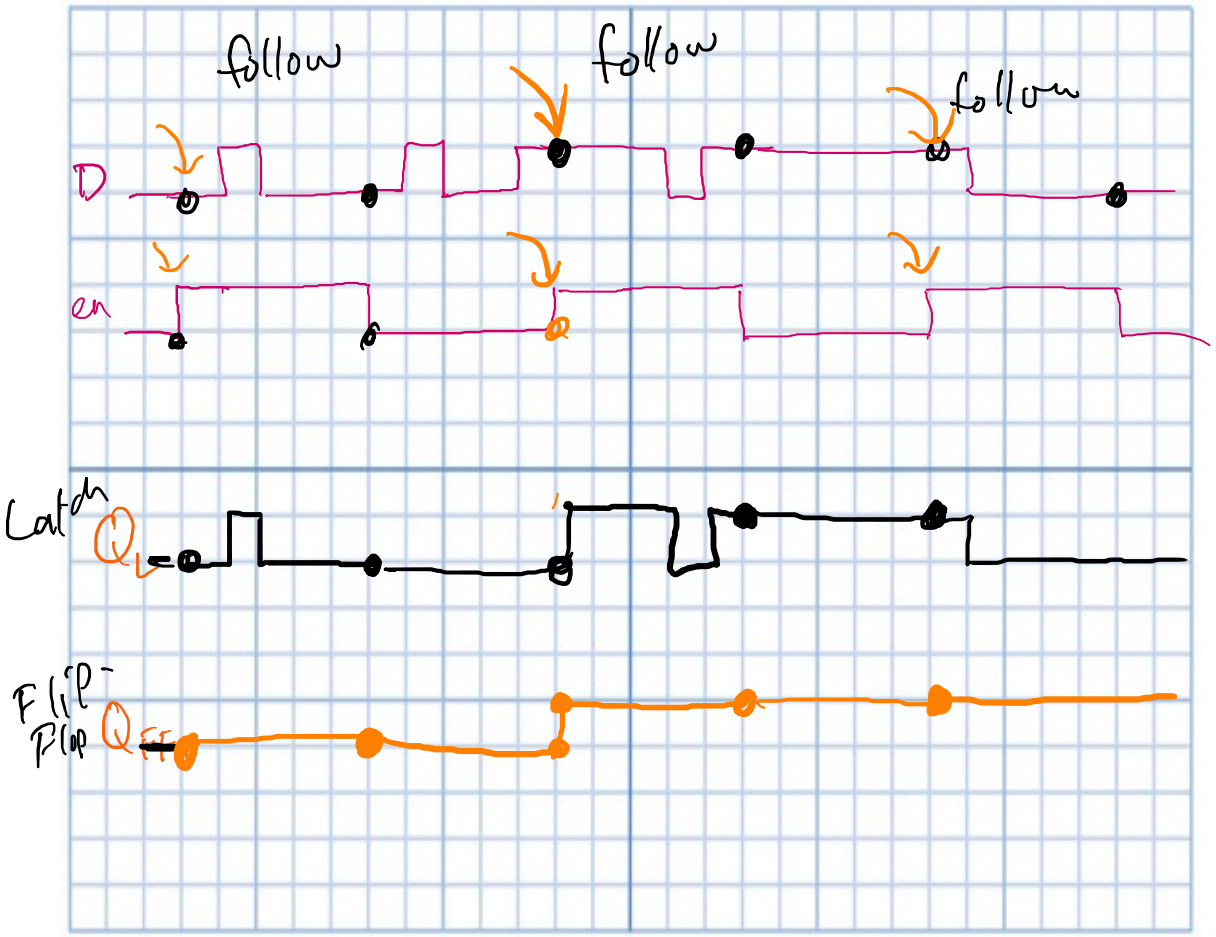


the " > " symbol tells you it is Flip-Flop



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

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 <= 2

Blocking vs. NonBlocking Assignments

x <= 2

y <= x + 1

- Non-blocking assignments (<= in Verilog):
  - Execute <u>concurrently</u>
  - 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.

x=0

always_ff @clk

x=2 x <= 2;

y=1 y <= x+1

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
```

LS RHS

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 1+1=2
0+1 2+1=3
0+1 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

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

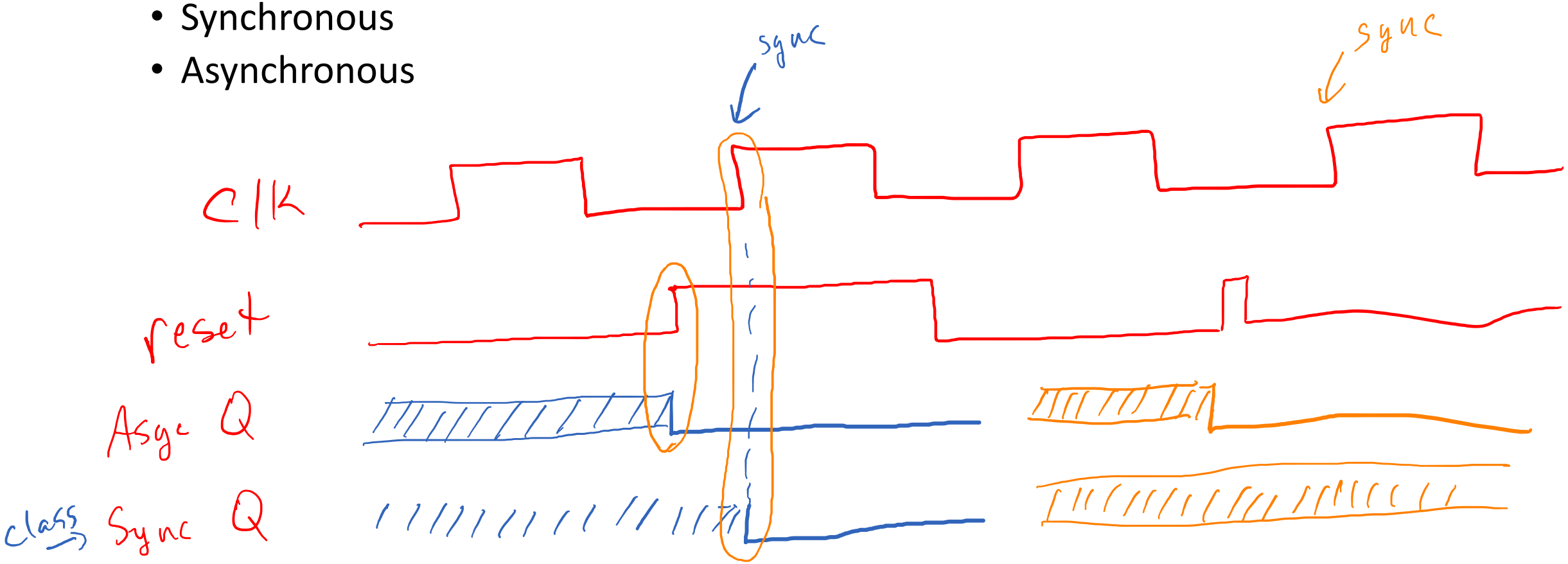
```
module d_ff (
    input d,            //data
    input clk,          //clock
    output reg q        //reg-isters hold state
);

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

endmodule
```

logic

What is q before posedge clk?

D-FF's with Reset

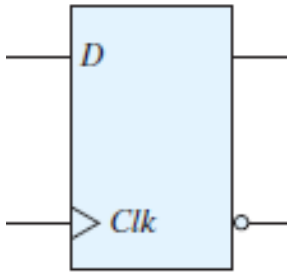
- Two different ways to build in a reset
  - Synchronous
  - Asynchronous



D-FF’s with Reset

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

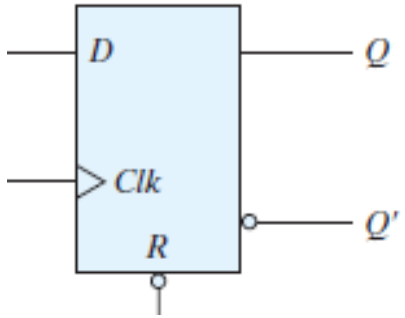
Verilog models of D flip-flop



Edge triggered D flip-flop:

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

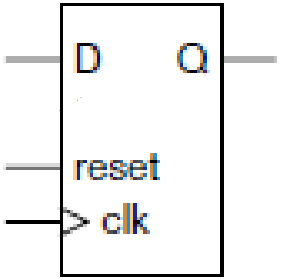
No reset ff



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;
```