ALU

ENGR 210 / CSCI B441
"Digital Design"

Flip-Flops

Andrew Lukefahr

Announcements

- P7 Saturating Counter is out
- P8 – Elevator Controller is out
  - This one is hard.

UPDATE: ‘wire’ vs ‘logic’

SystemVerilog (NEW) Rules:

Just use ‘logic’*

* ***EXCEPT***

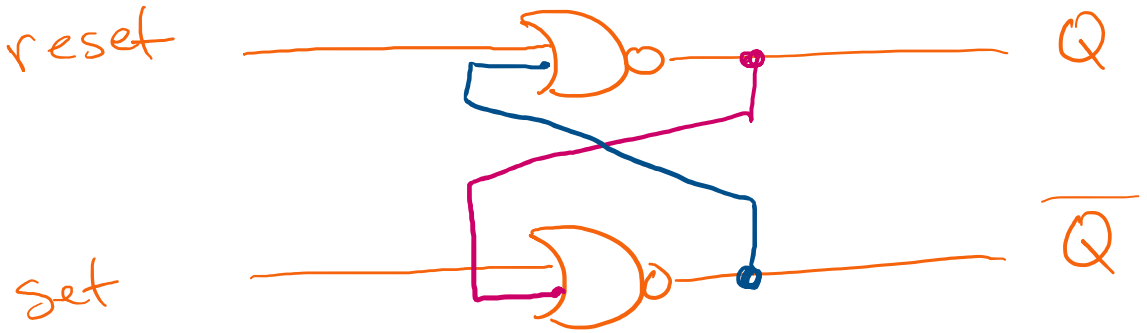
```
logic foo = a & b;  (BAD - Initial values of a & b only)

wire foo = a & b;   (OK)

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

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

BLOCKING (=) FOR `always_comb`

NON-BLOCKING (<=) for `always_ff`

Last Time: SR Latch

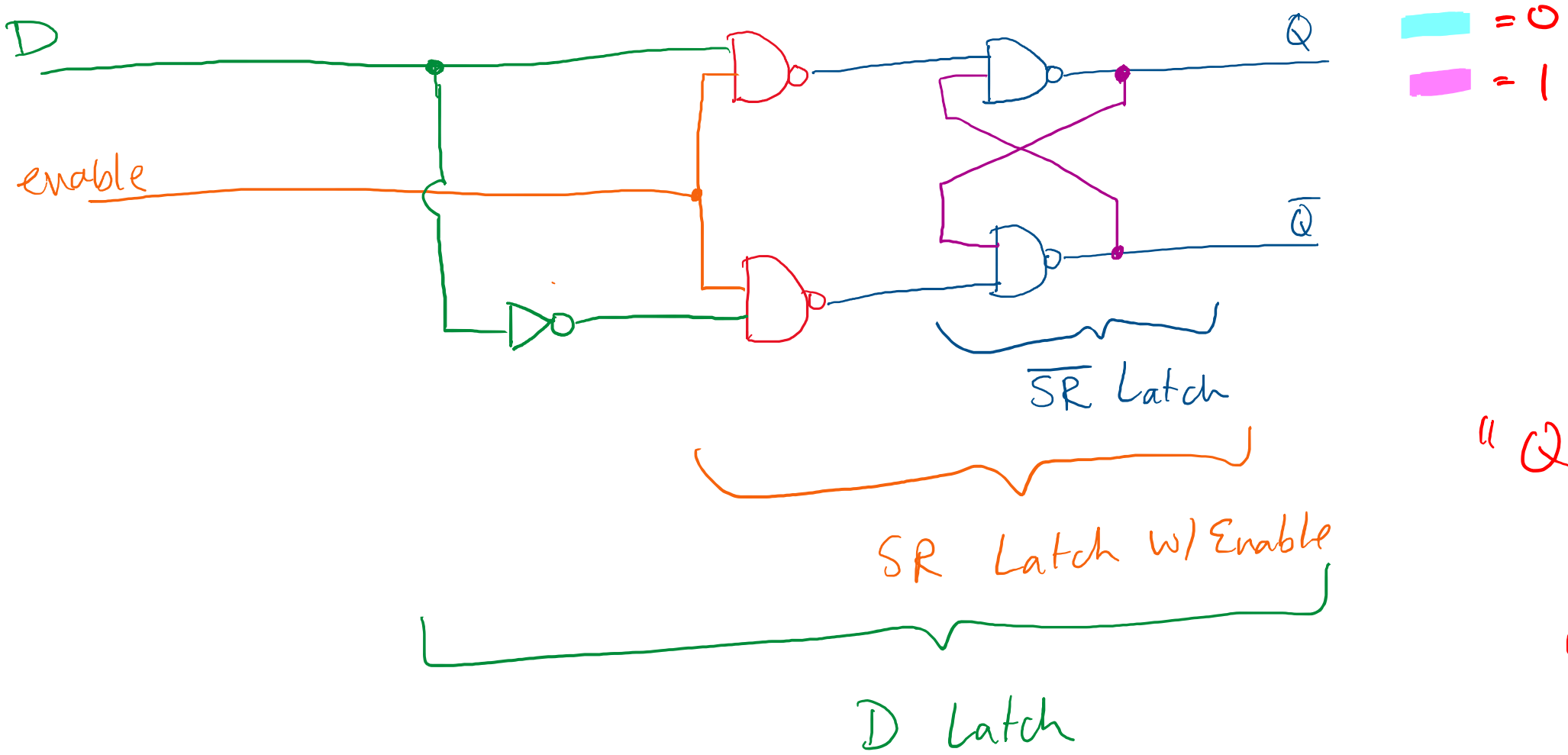


Set	reset	Q	$\overline{Q}$
0	1	0	1
0	0	0	1
1	0	1	0
0	0	1	0

same inputs, different output!

⇒ Internal state!

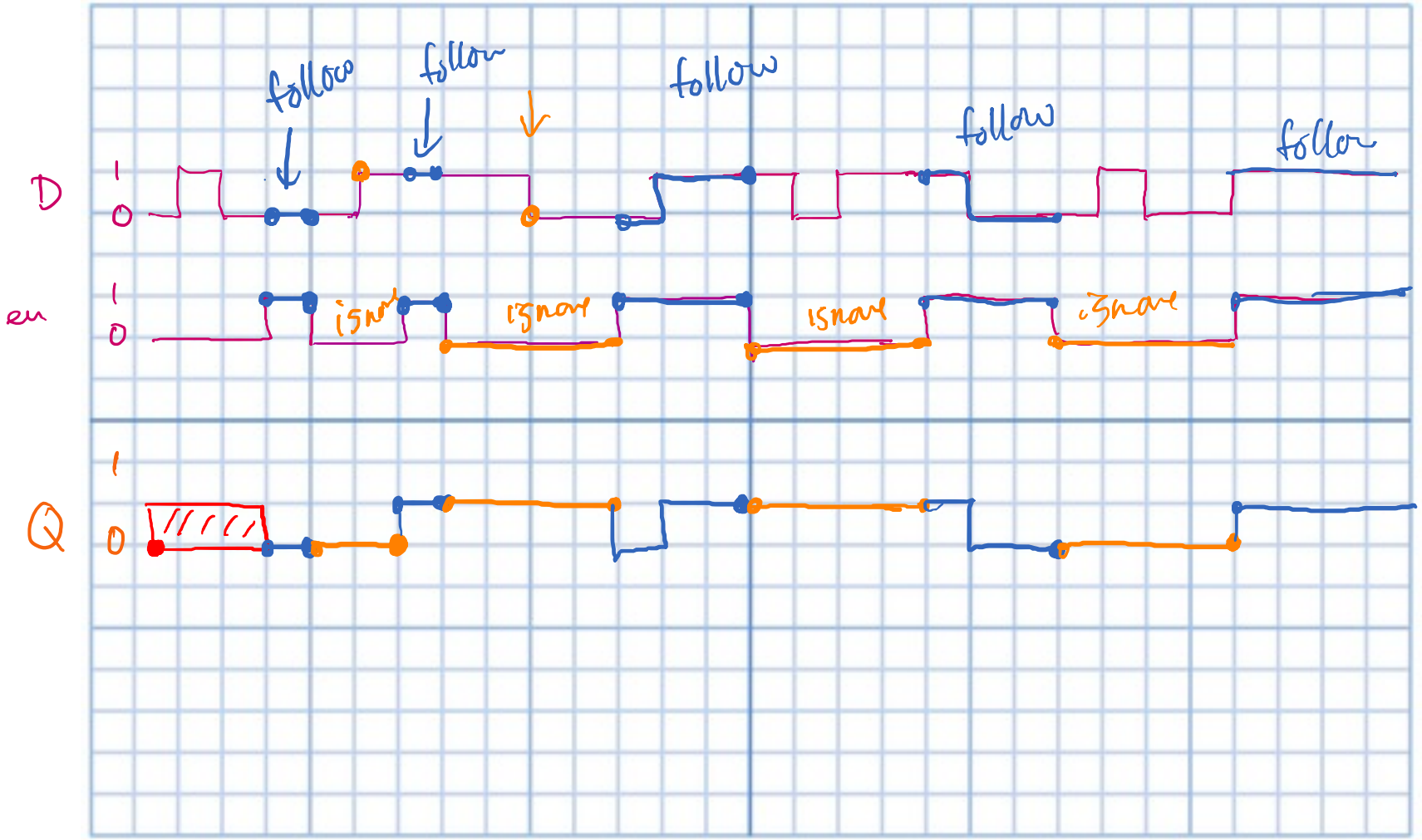
D-Latch



"Q follows D ~~when~~ when en = 1, otherwise doesn't change"

Inputs to D Latches

Output Follows Input when Enable=1

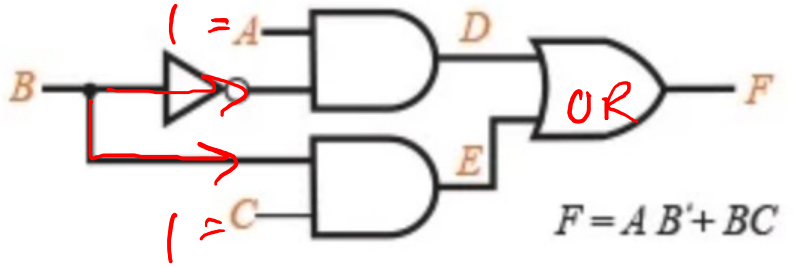


Glitches

→ unintended, short, errors in boolean logic

→ caused by gate delays

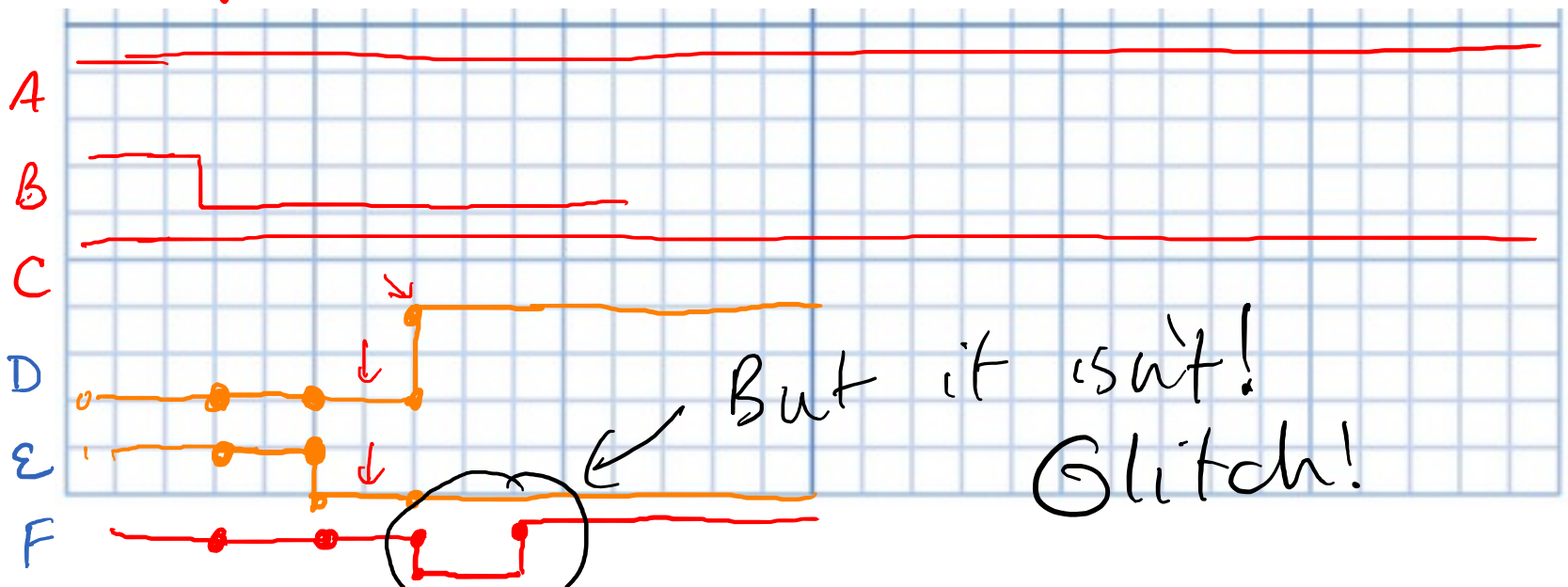
- Assume 10ps / gate.
- A=1, C=1, B falls
- What is F?



← should always be true!

A=1 B=1 C=1

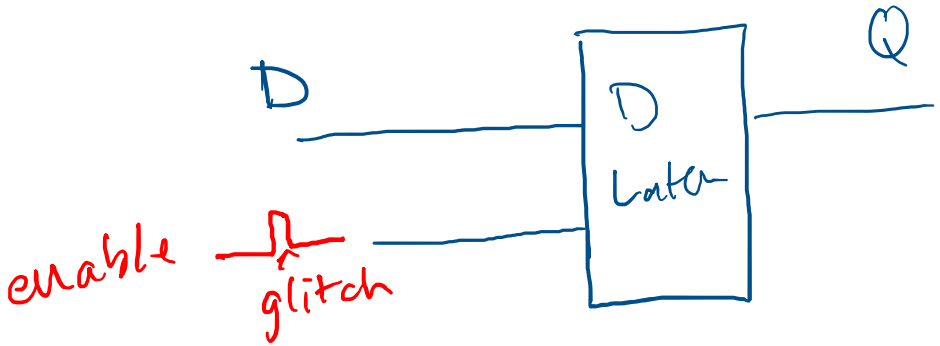
10ps

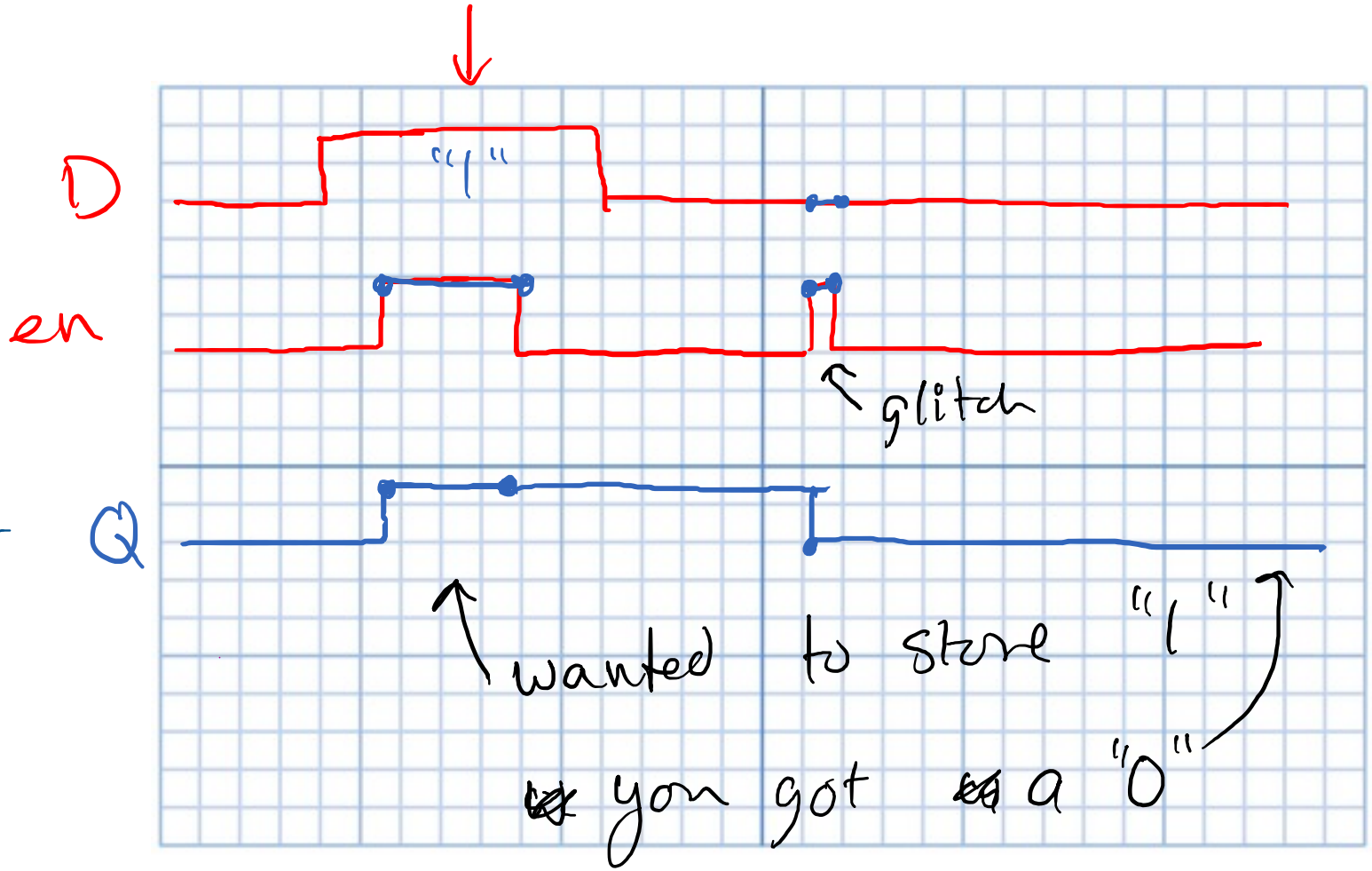


But it isn't!

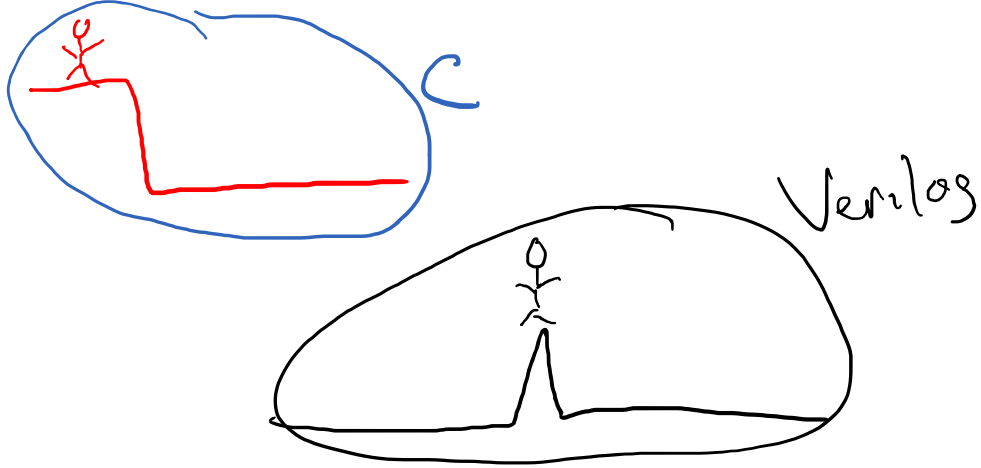
Glitch!

Glitches on D-Latches

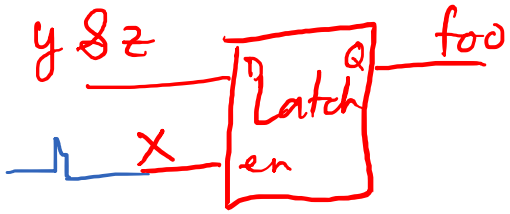




Inferred Latches




```
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
```

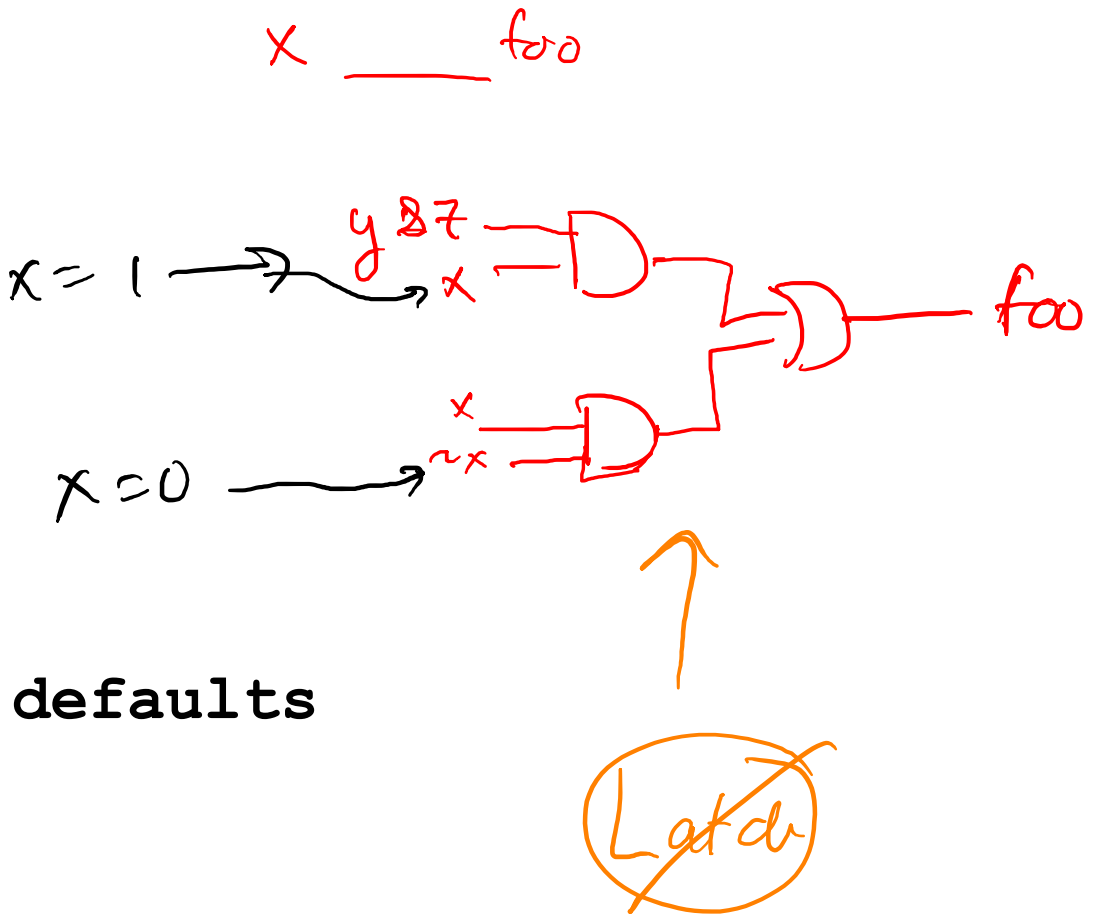


guess Latch?

What if `x == 0`?

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
```

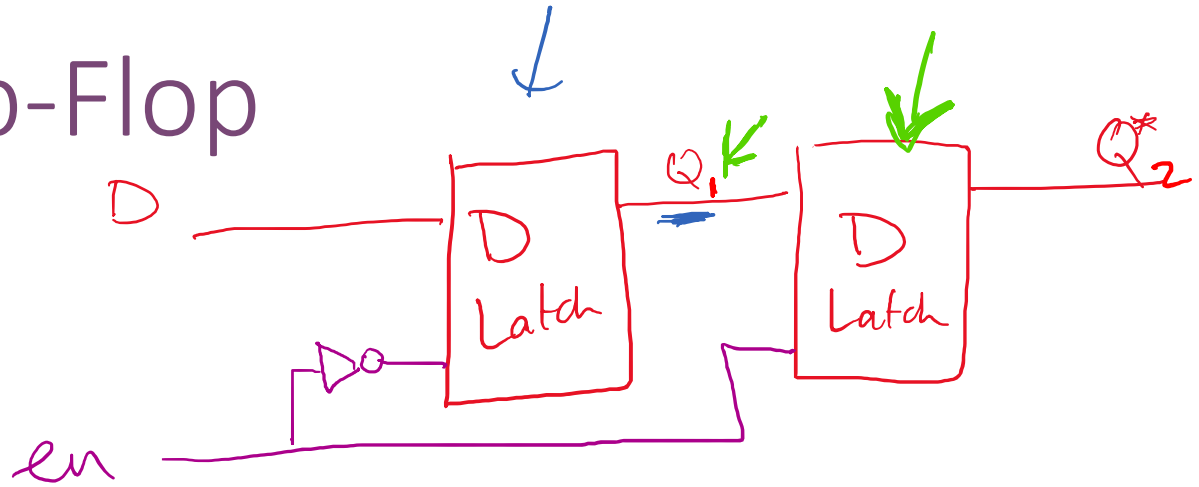


What if `x == 0`? `foo = bar = x`!

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

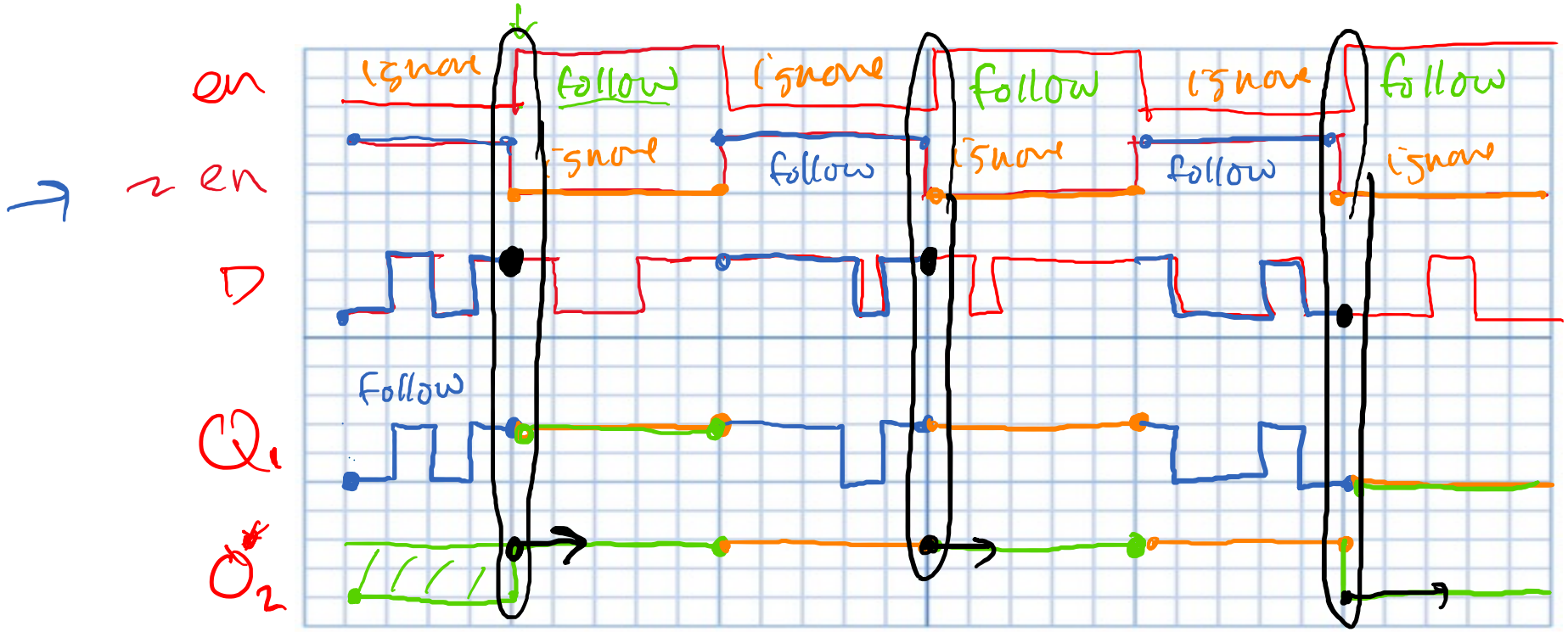
Always specify defaults for **`always_comb`**!

D Flip-Flop



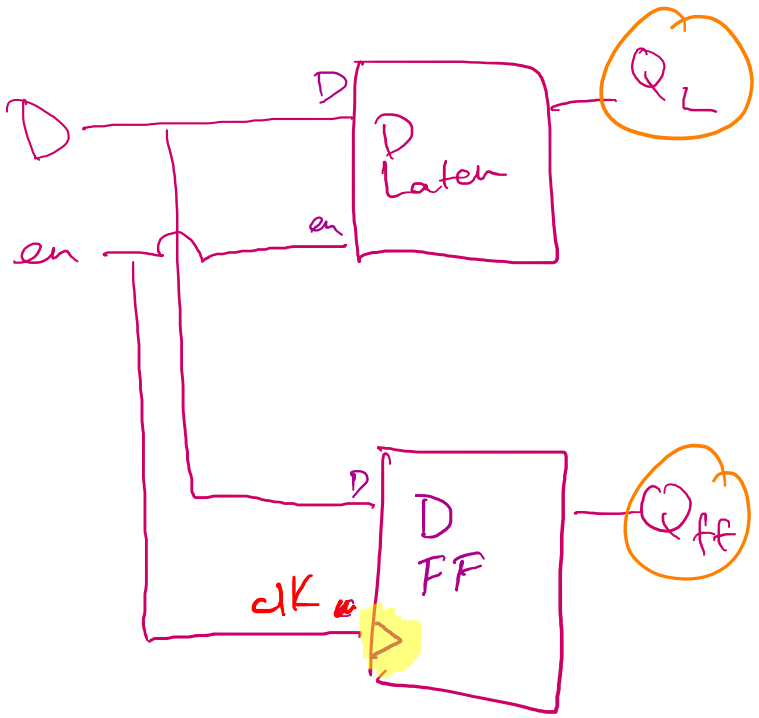
* no gate delays

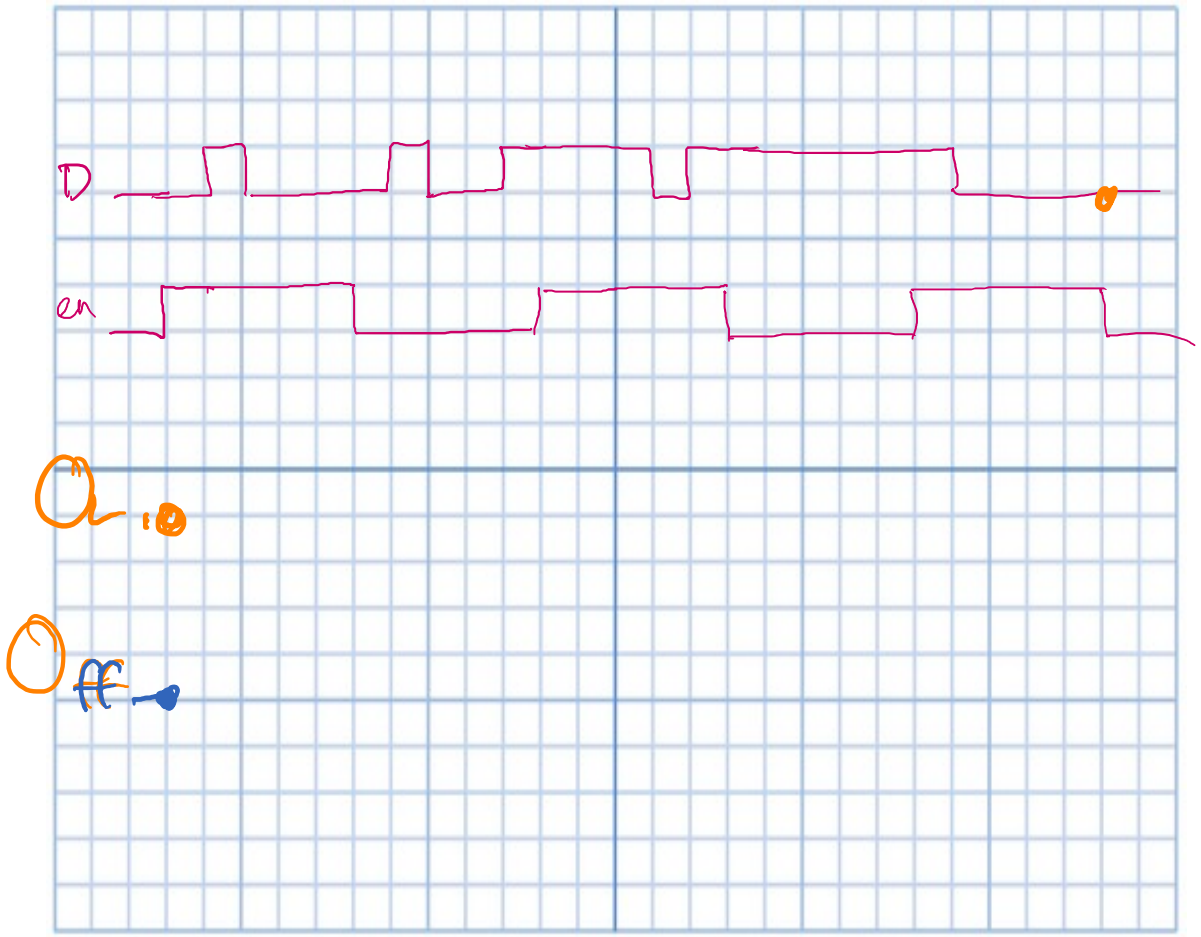
D latch:
Q follows D when en=1



Levels vs. Edges

D Flip-Flop vs. D Latch





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
                  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
                  qn <= ~d; //optional
      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 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.

x=0

always_ff @clk

x=2 x <= 2; 2

y=1 y <= x+1 0+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. 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

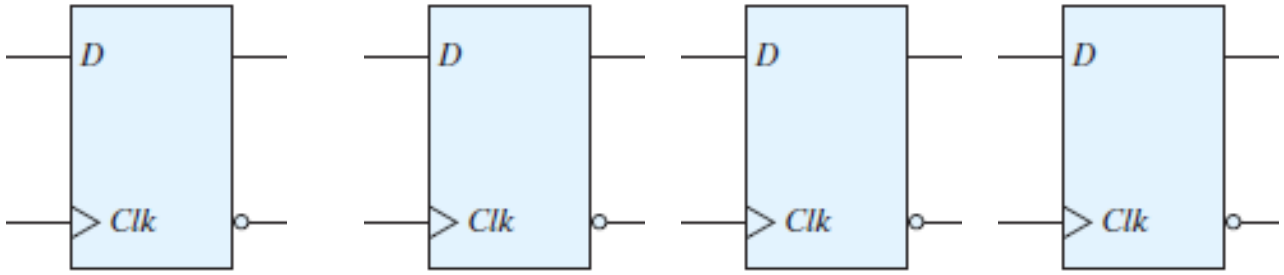
```
module d_ff (
    input d,            //data
    input clk,          //clock
    output logic q      //reg-isters hold state
);

    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

Registers



(a) Positive-edge (a) Positive-edge (a) Positive-edge (a) Positive-edge

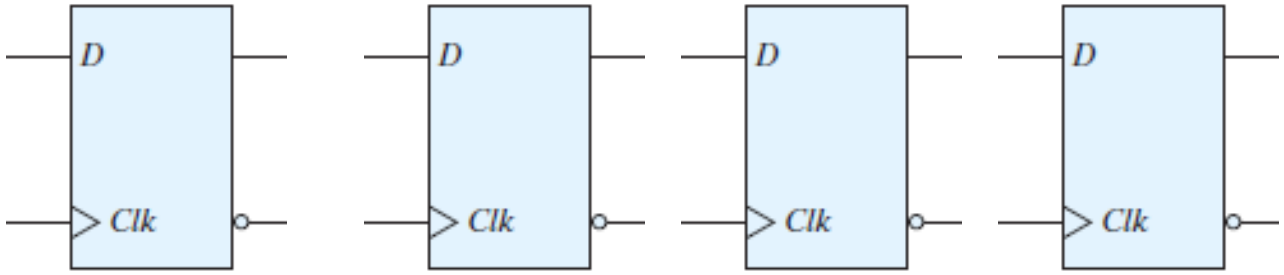
4-bit Register in Verilog

```
module d_ff (
    input              d,    //data
    input              clk,  //clock
    output logic       q     //output register
);

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

endmodule
```

4-bit Register in Verilog

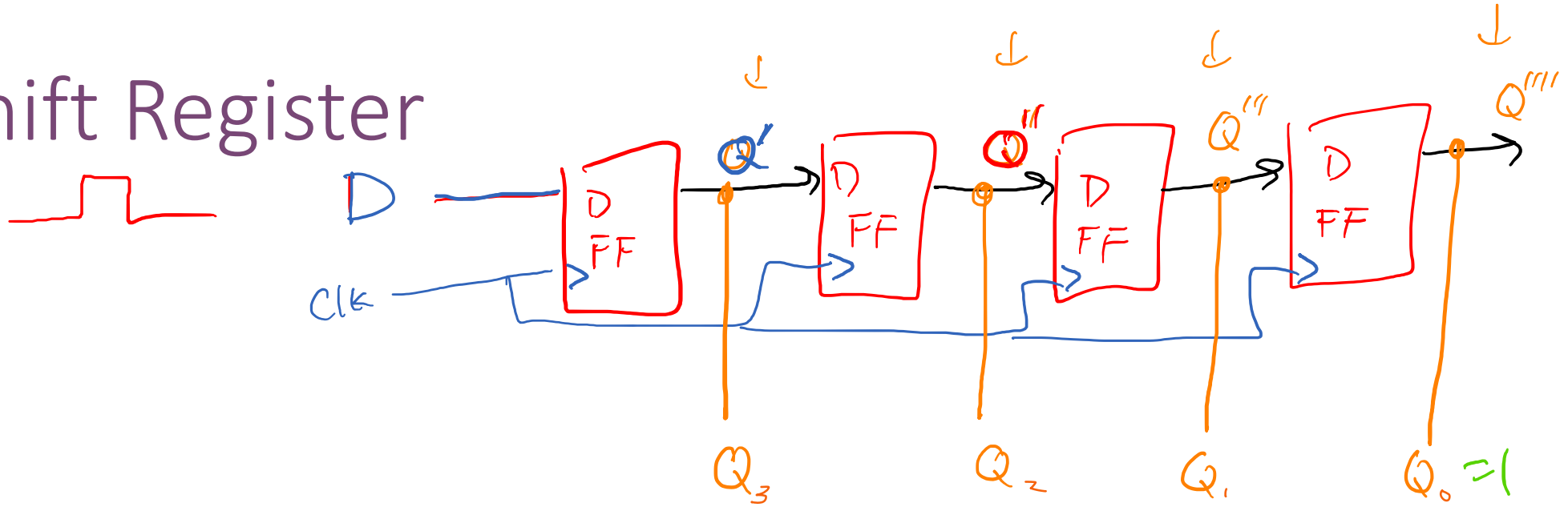
```
module d_ff (
    input       [3:0]   d,   //data
    input               clk, //clock
    output logic [3:0] q    //output register
);

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

endmodule
```

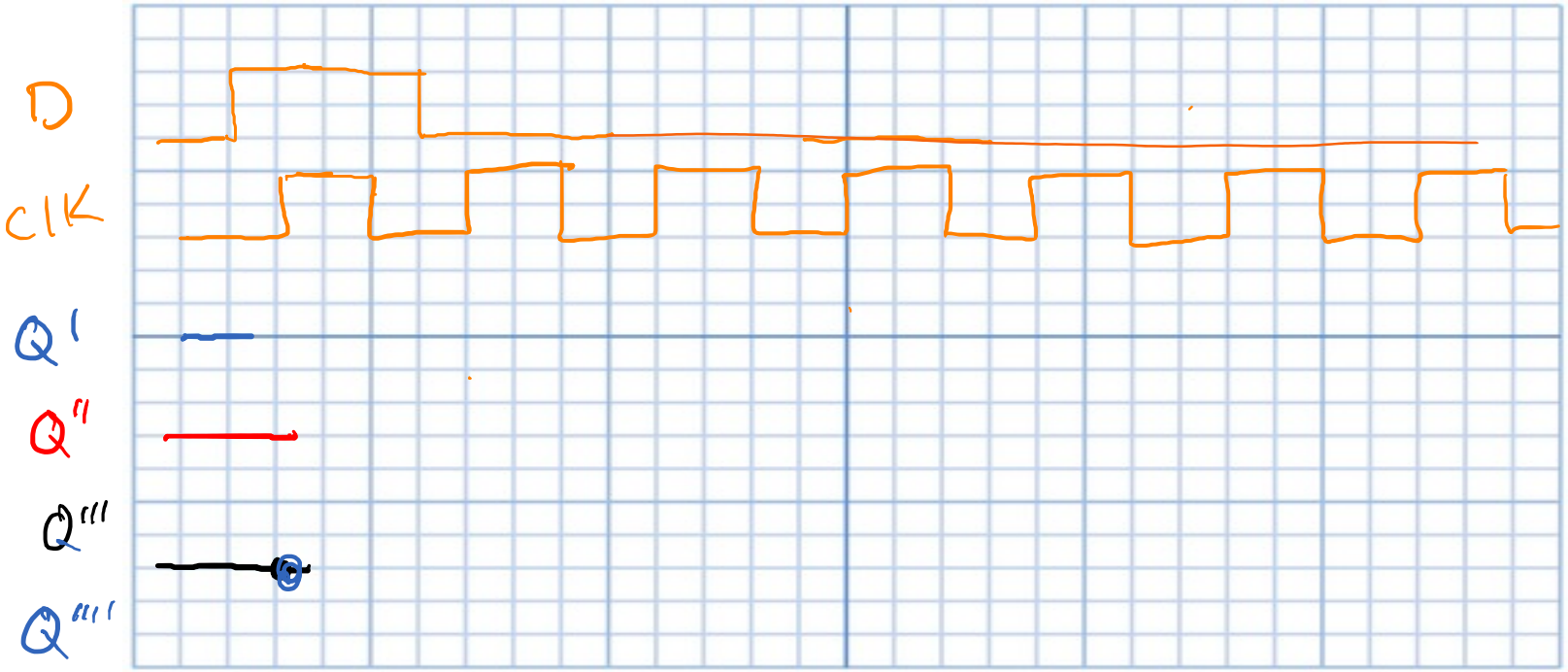
D Flip-Flops as Shift Registers



(a) Positive-edge (a) Positive-edge (a) Positive-edge (a) Positive-edge

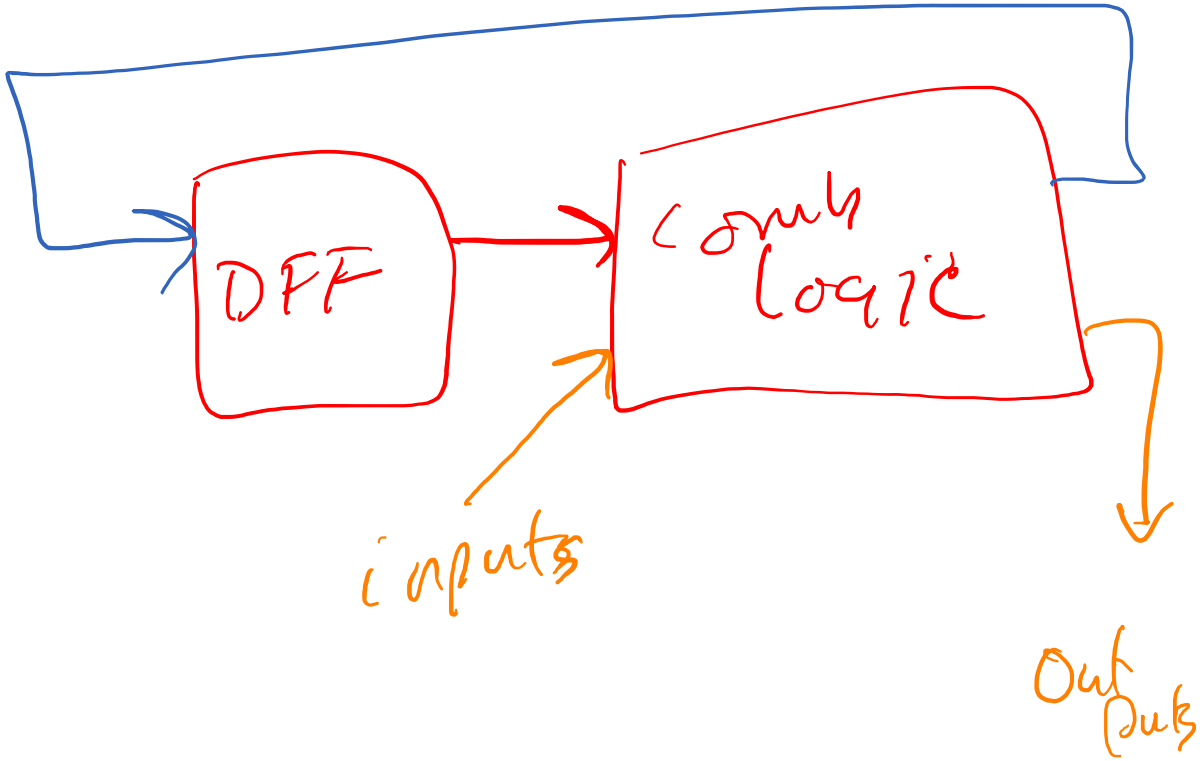
Shift Register





Shift-Register in Verilog

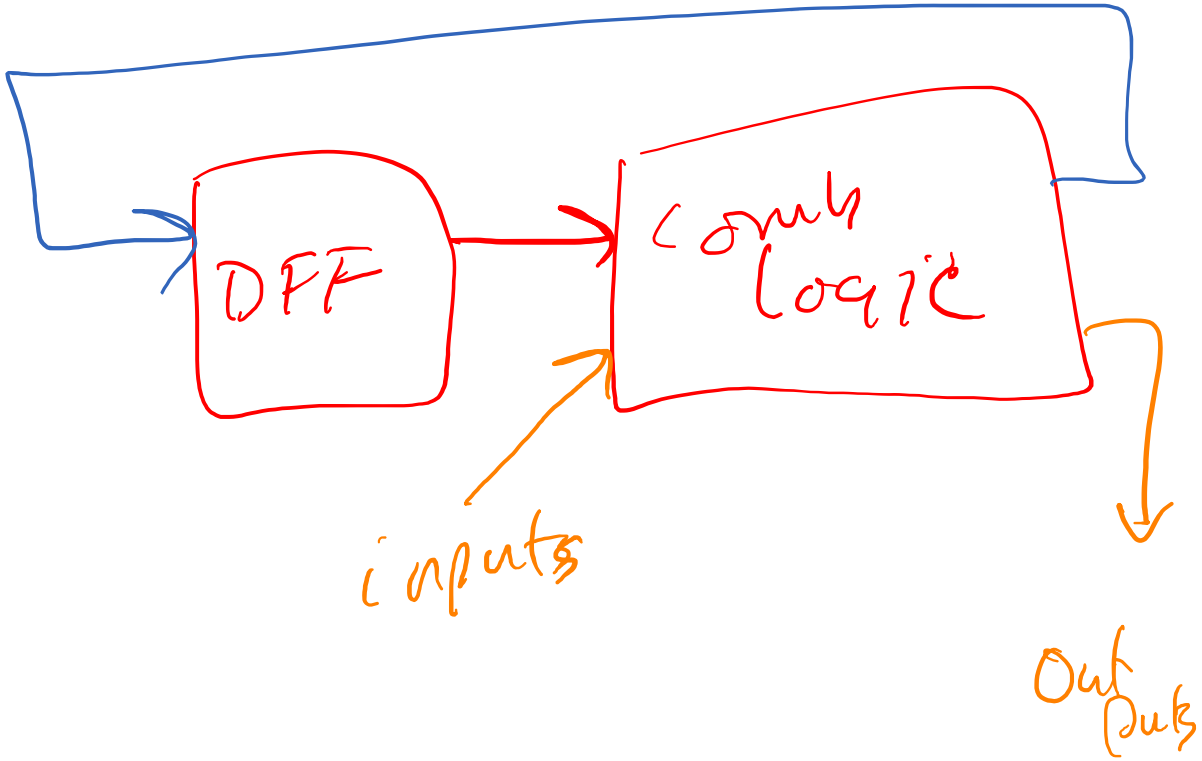
```
module shift_register (
    input clk, rst, D,
    output [3:0] Q );
```




```
endmodule
```

Shift-Register in Verilog

```
module shift_register (
    input clk, rst, D,
    output [3:0] Q );

    logic [3:0] dff;
    logic [3:0] next_dff;

    always_ff(@posedge clk) begin
        if (rst) dff <= 4'h0;
        else     dff <= next_dff;
    end

    always_comb
        next_dff = { dff[2:0], D};

    assign Q = dff;

endmodule
```



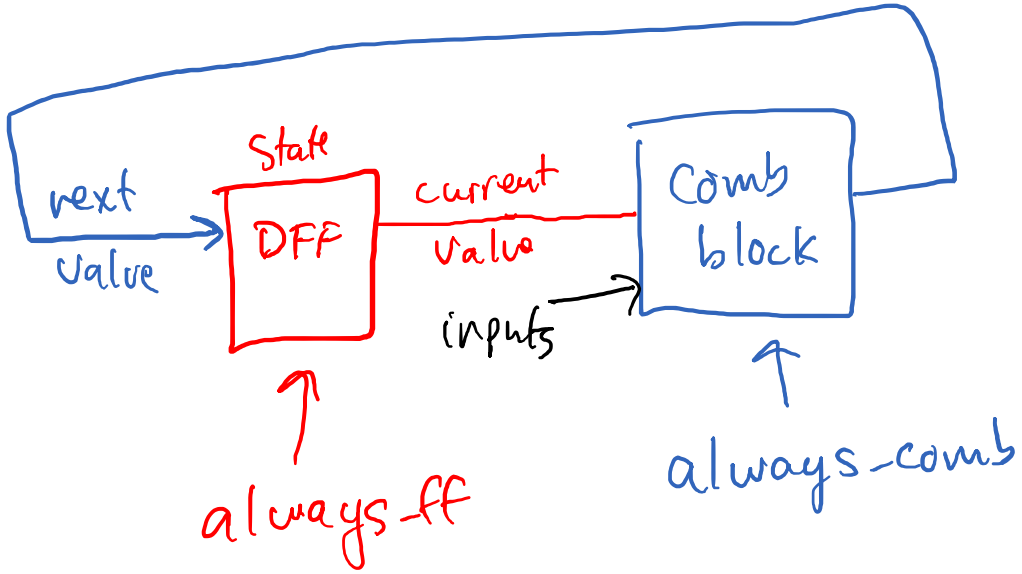
Shift-Register in Verilog

```
module shift_register (
   input clk, rst, D,
   output [3:0] Q );

   logic [3:0] dff;
   logic [3:0] next_dff;

   always_ff(@posedge clk) begin
      if (rst) dff <= 4'h0;
      else     dff <= next_dff;
   end

   always_comb
      next_dff = { dff[2:0], D};

   assign Q = dff;

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

Next Time

- State machines