

ENGR 210 / CSCI B441
“Digital Design”

Flip Flops + Sequential Logic

Andrew Lukefahr

Course Website

fangs-bootcamp.github.io

Write that down!

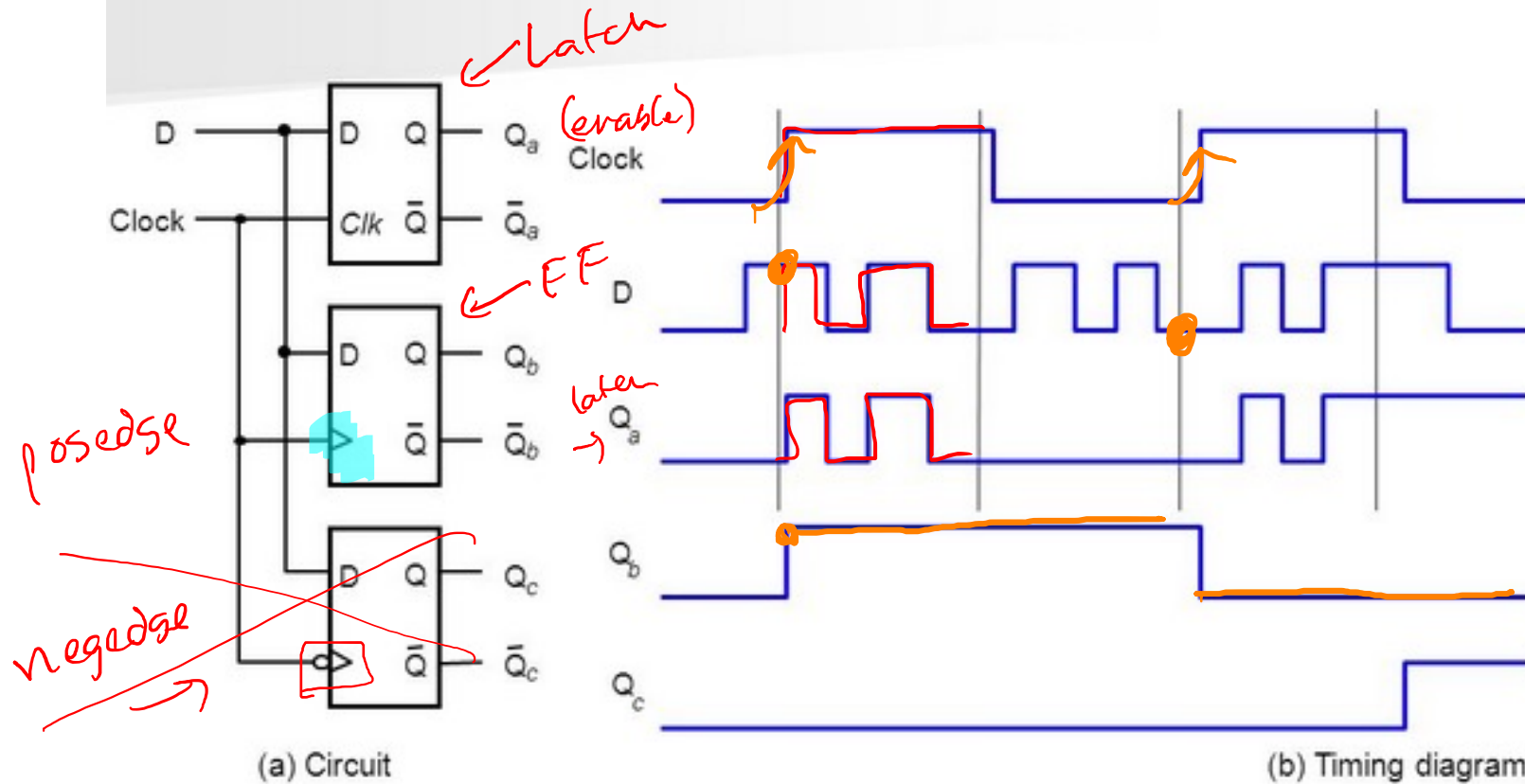
Announcements

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

D Latch versus D Flip-Flop

Latch → output follows input (D) when enable (clk) is high

FF → ~~not~~ output follows input only on rising edge of enable (clk)



Comparison of level-sensitive and edge-triggered devices

Defaults

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

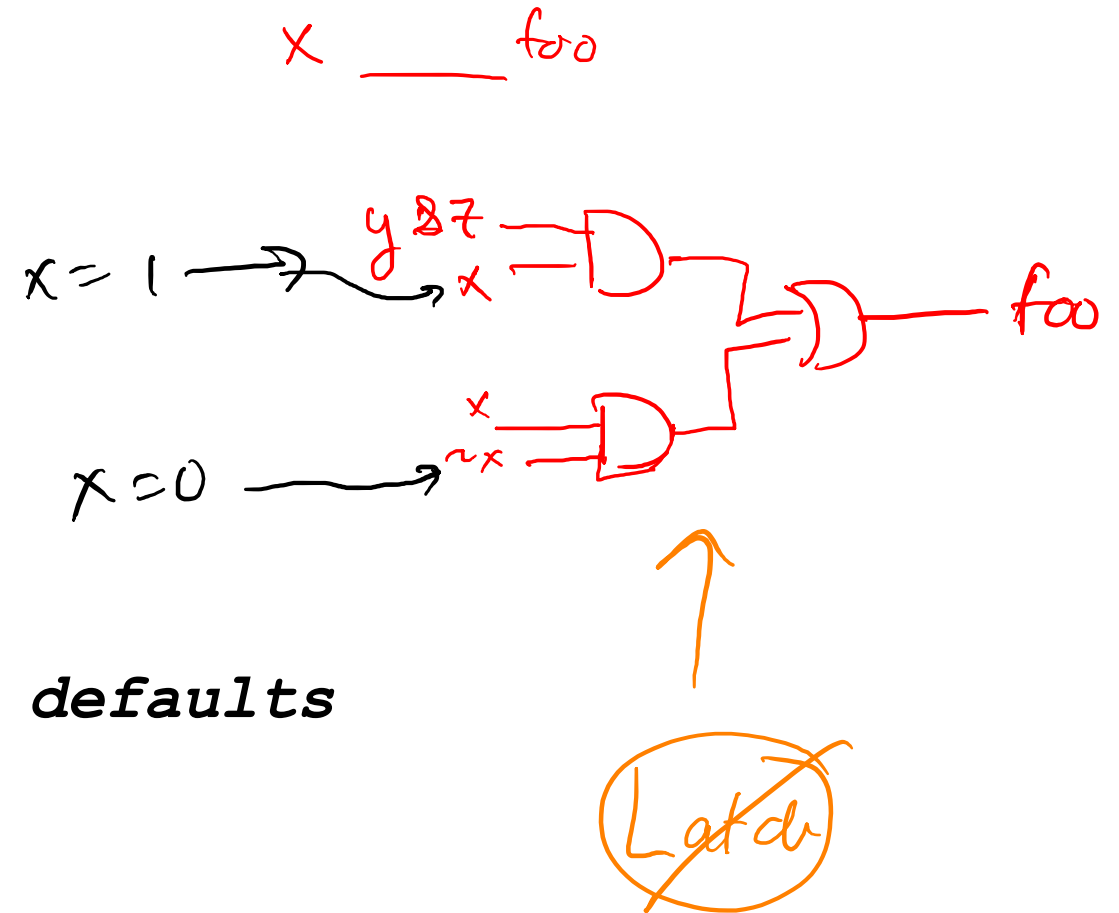
```
always_comb begin
```

```
    foo = x; bar = x; //good: defaults
```

```
    if (x) foo = y & z; //
```

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

```
end
```



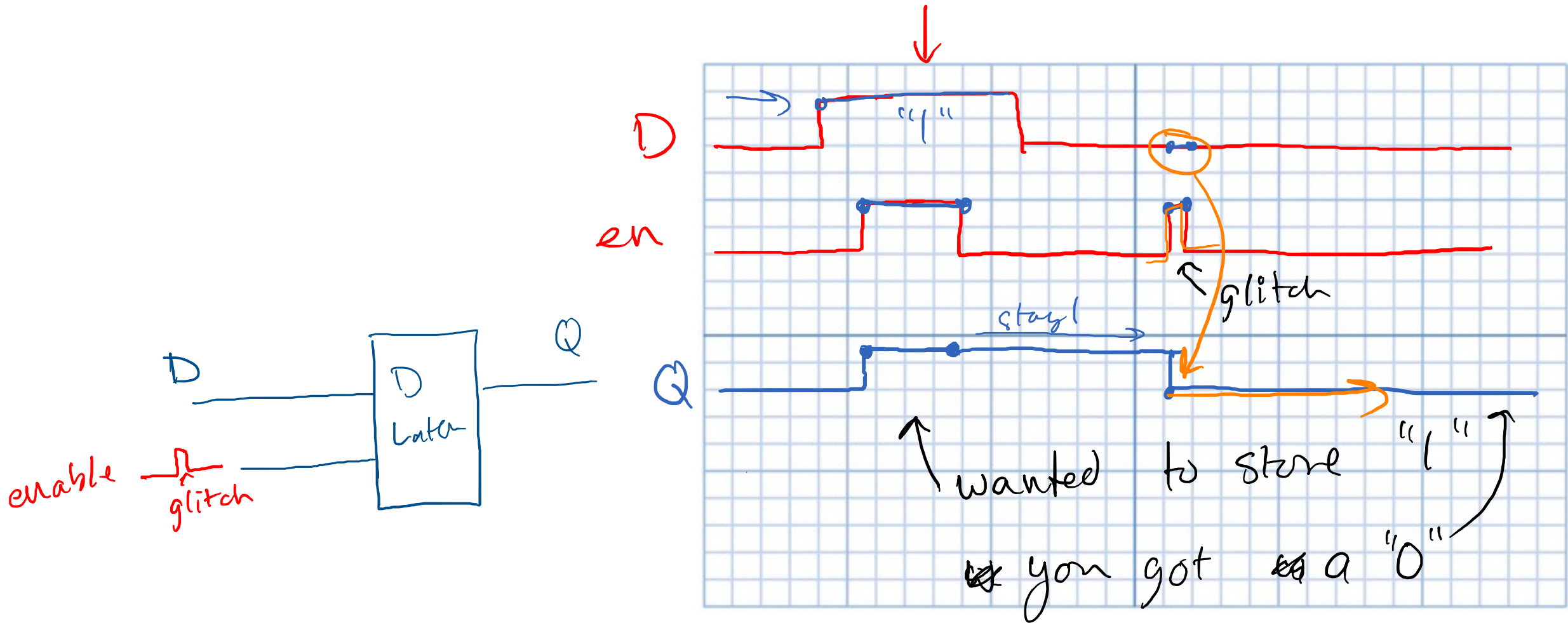
What if `x == 0`? `foo = bar = x`!

Always specify defaults for `always_comb`!

"Warnings: Inferrius Later"

Always specify
defaults for
always_comb!

Glitches on D-Latches



Flip-Flop in Verilog

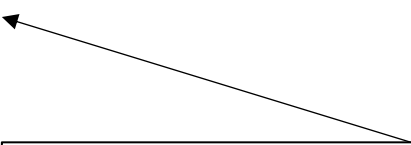
```
module d_ff (  
    input          d,    //data  
    input          clk,  //clock  
    output logic q      //output register  
);  
  
    always_ff @( posedge clk )  
    begin  
        q <= d; //non-blocking assign  
    end  
  
endmodule
```


BLOCKING (=) FOR
always_comb

NON-BLOCKING (<=) for
always_ff

D-FlipFlop w/Clock

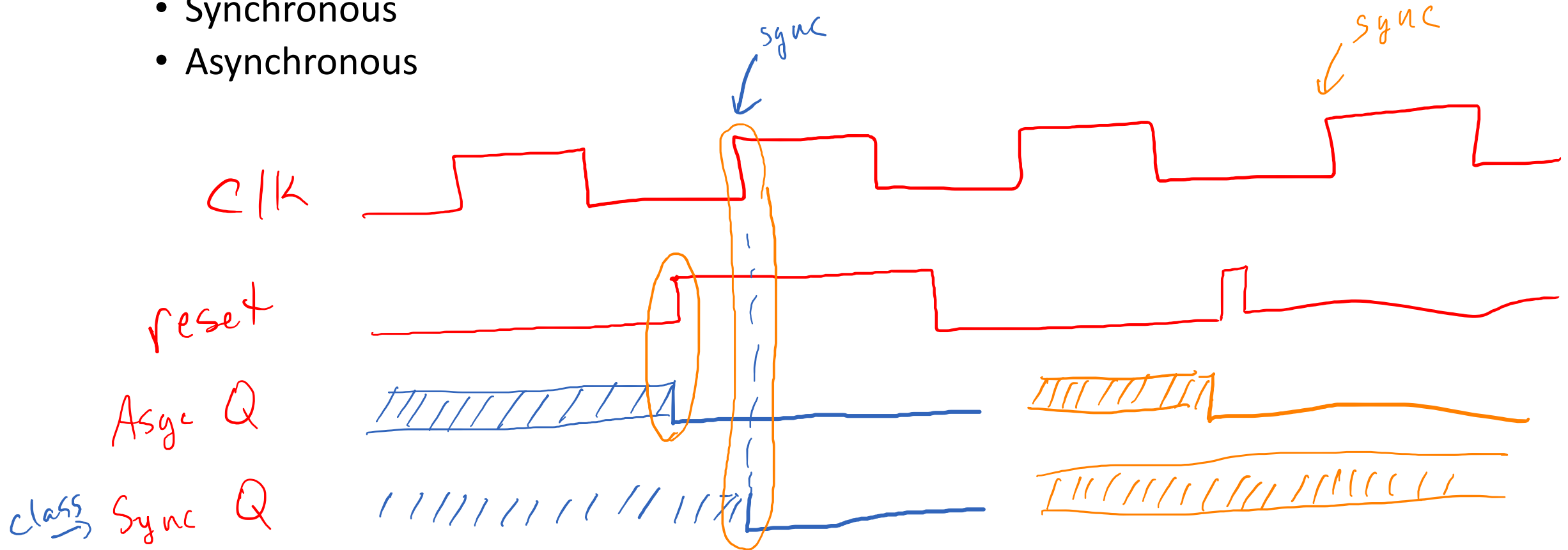
```
module d_ff (  
    input d,           //data  
    input clk,         //clock  
    output logic q      //output  
);  
  
    always_ff @( posedge clk )  
    begin  
        q <= d; //non-blocking assign  
    end  
endmodule
```



What is q before posedge clk?

D-FF's with Reset

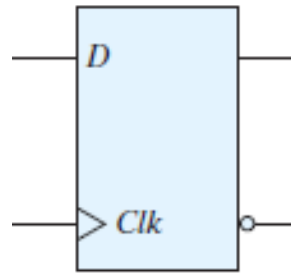
- Two different ways to build in a reset
 - Synchronous
 - Asynchronous



D-FF's with Reset

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

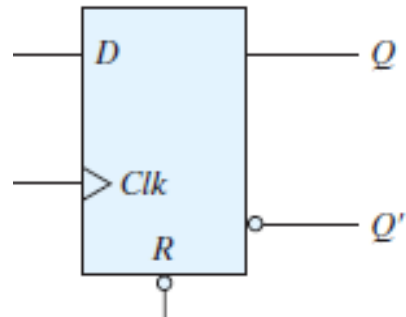
Verilog models of D flip-flop



Edge triggered D flip-flop:

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

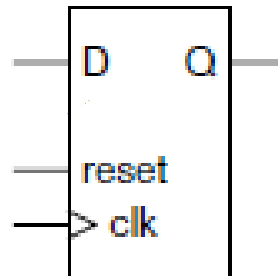
↖ No reset
FF



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: ↖

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

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
```



```

module counter(
    input          clk,    //clock
    input          rst,    //reset
    output logic   out     //output
);

    logic [3:0] Q;
    logic [3:0] sum;

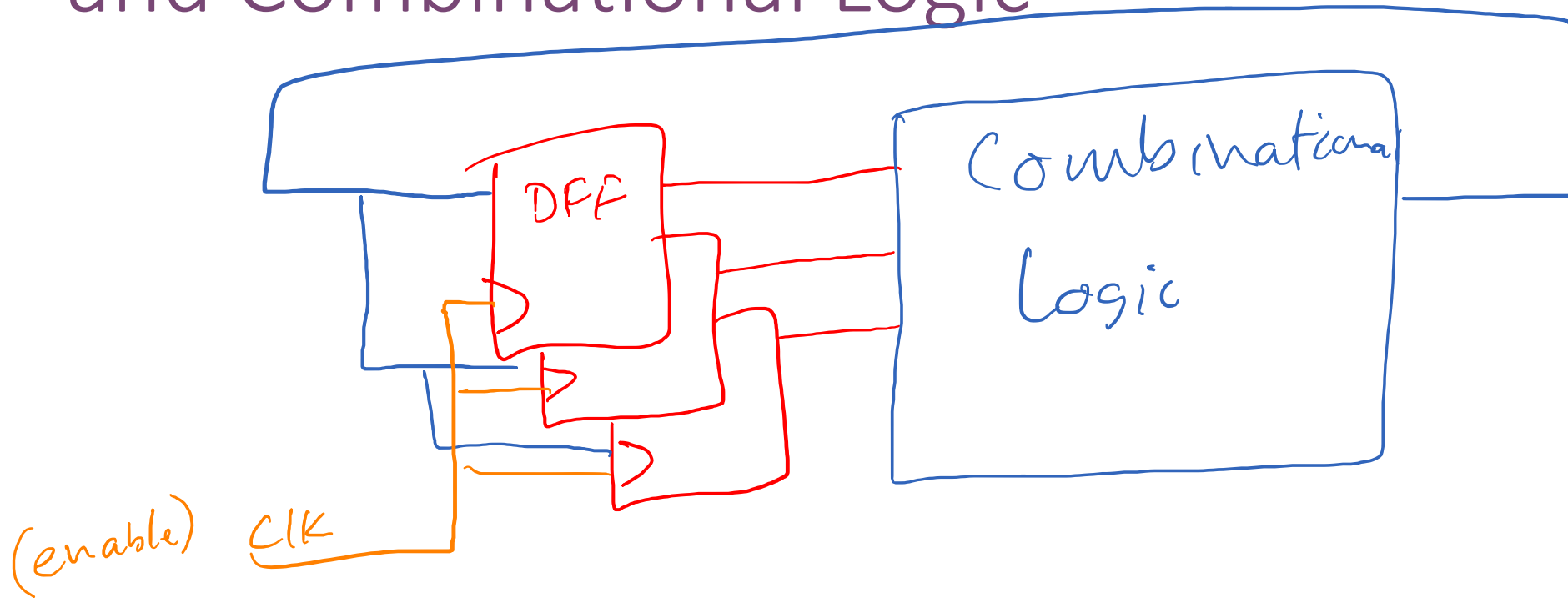
    always_ff @( posedge clk ) // <- sequential logic
    begin
        if (rst) Q <= 4'h0;
        else     Q <= sum;    //non-blocking
    end

    always_comb begin // <- combinational logic
        sum = Q + 4'h1;    //blocking
        out = sum[3];
    end

endmodule

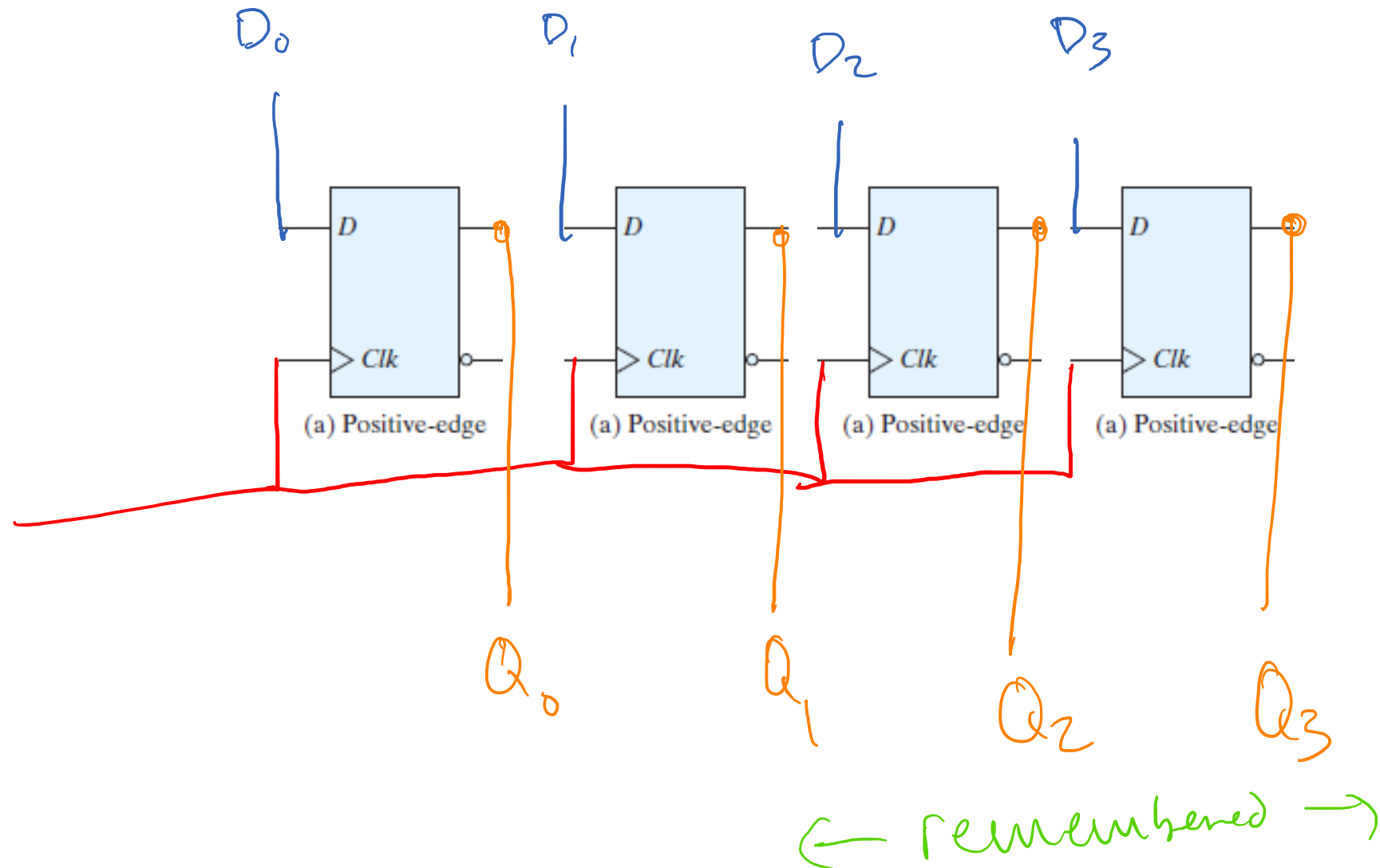
```

Sequential Logic uses both Flip-Flops and Combinational Logic



Registers

Switches



4-bit Register in Verilog

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

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

```
endmodule
```

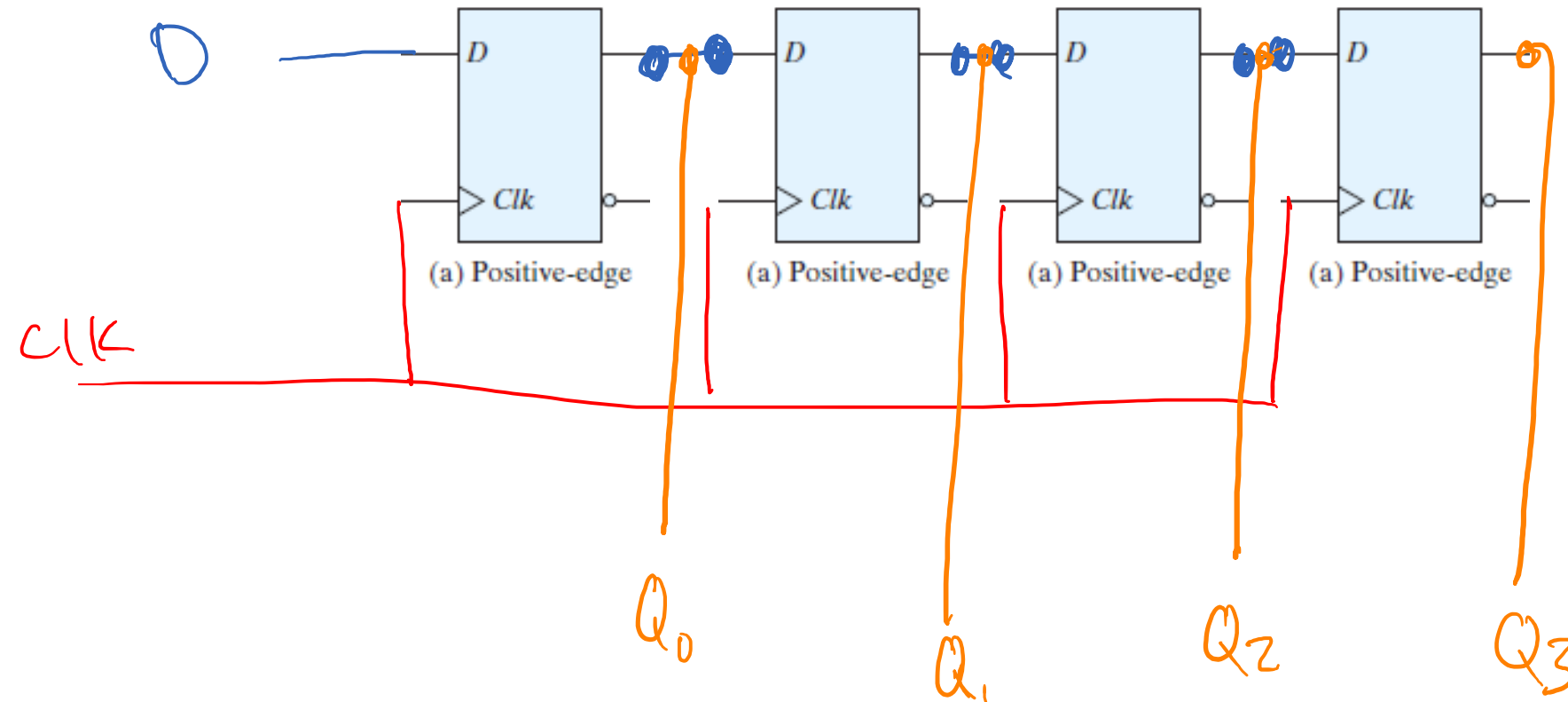
4-bit Register in Verilog

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

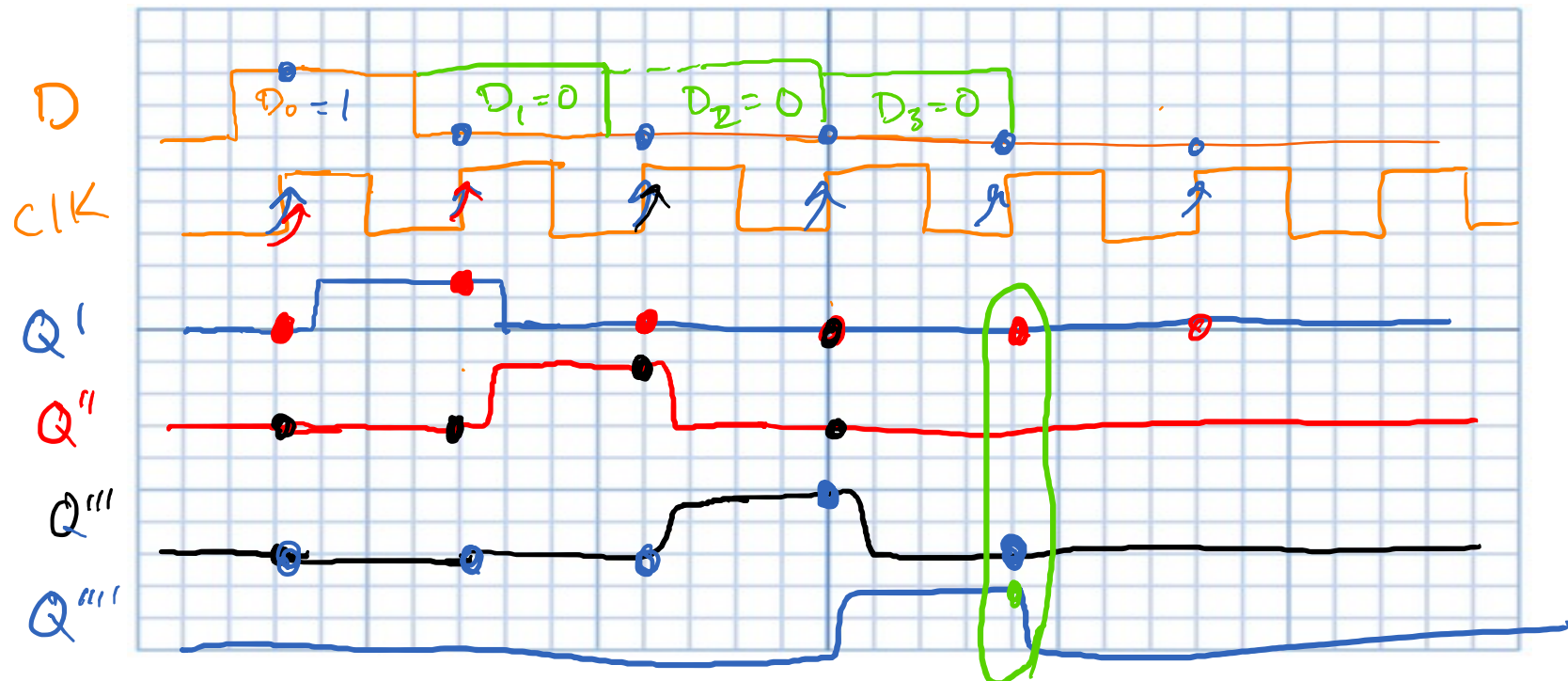
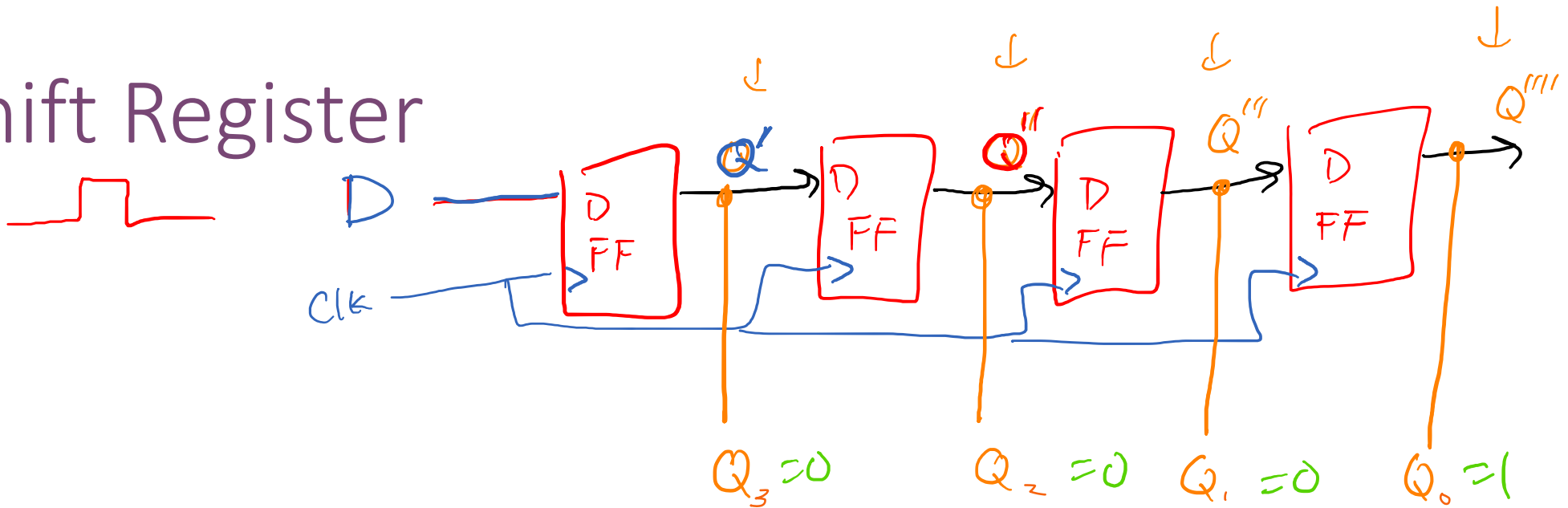
    always_ff @( posedge clk )
    begin
        q <= d; //non-blocking assign
    end

endmodule
```

D Flip-Flops as Shift Registers



Shift Register



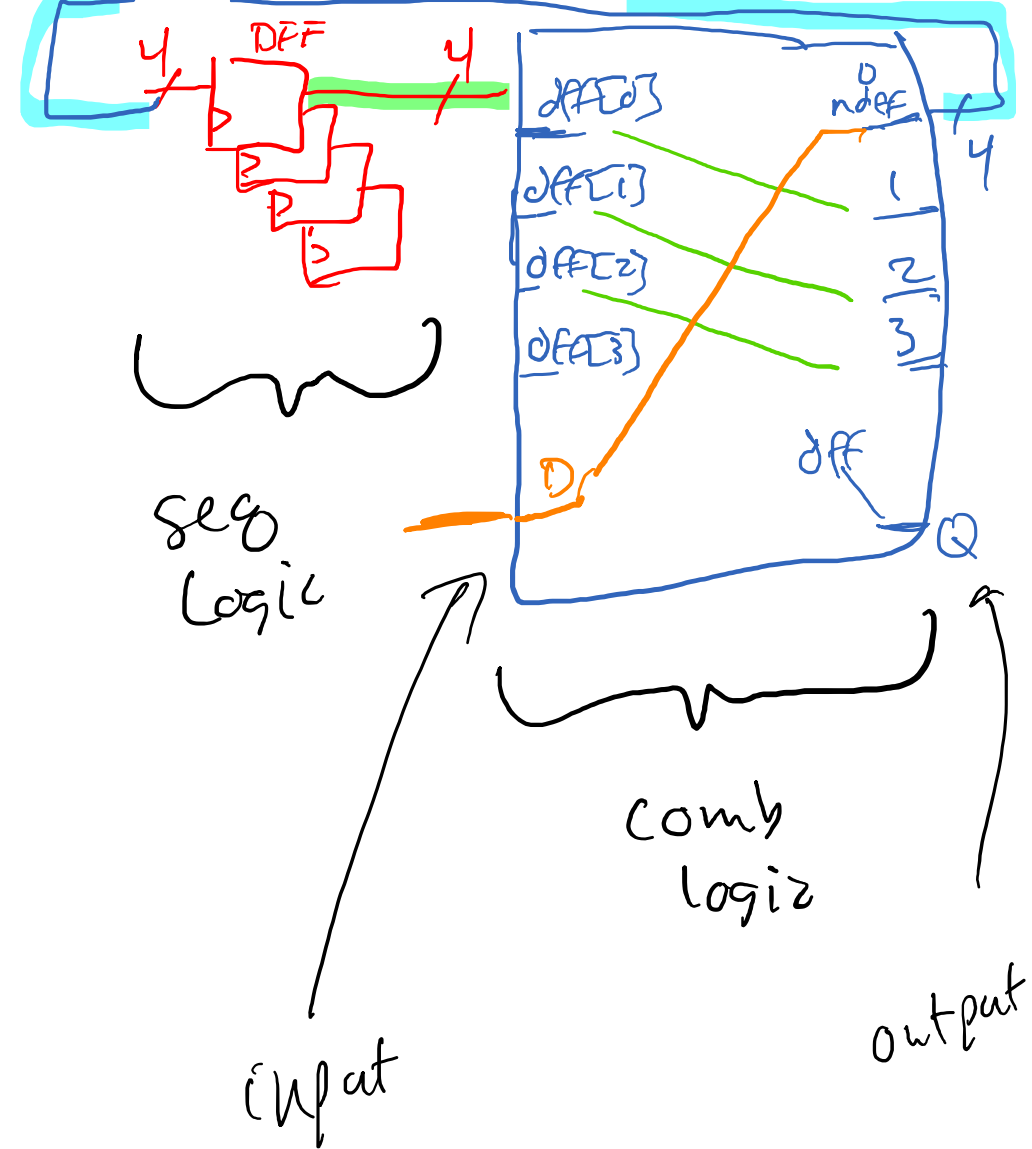
Shift-Register in Verilog

```

module shift_register (
    input clk, rst, D,
    output [3:0] Q );
    logic [3:0] dff // state
    logic [3:0] ndff // next state
    always-ff @(posedge clk) begin
        if (rst) begin
            dff <= 4'h0;
        end else begin
            dff <= ndff;
        end
    end
end

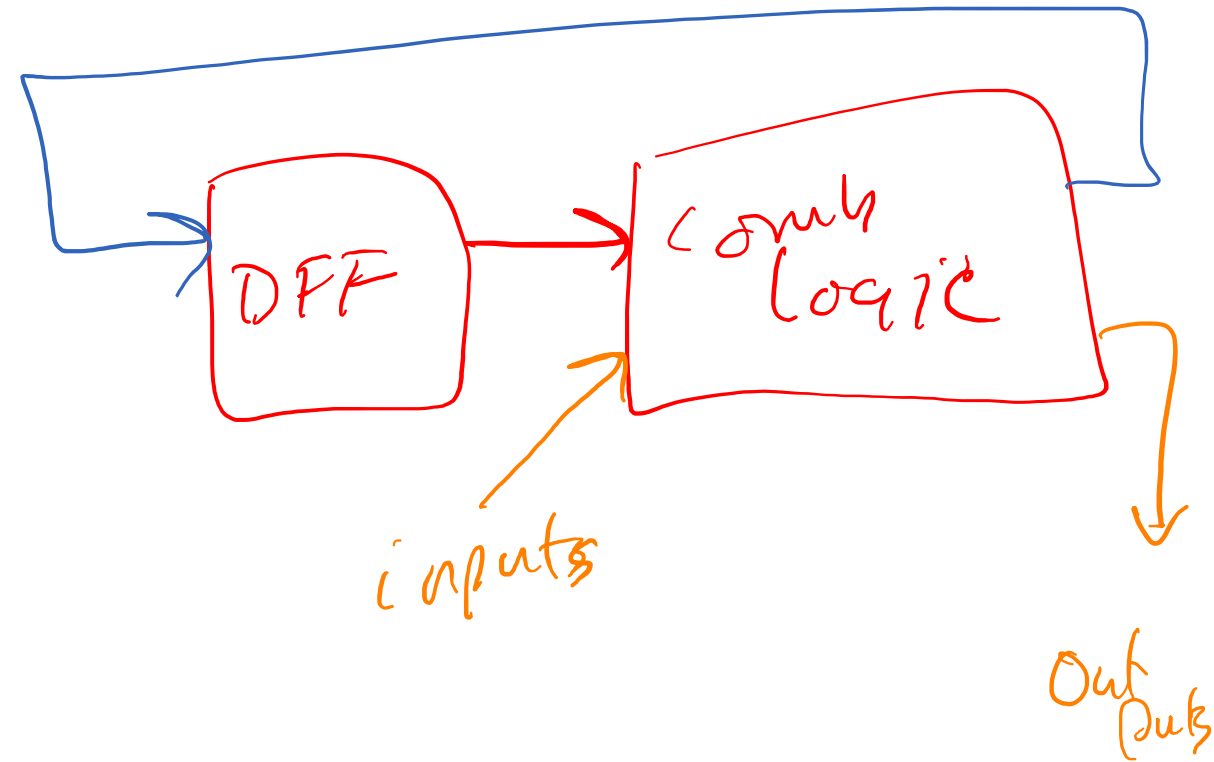
    always-comb begin
        ndff = { dff[2:0], D };
    end
    assign Q = dff;
endmodule

```



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
```



Inputs can affect output or state

```

module counter(
    input clk, rst
    input          out_fast, //faster output
    output logic  out      //output
);
    logic [3:0] Q;
    logic [3:0] sum;

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

    always_comb begin
        sum = Q + 4'h1;
        out = sum[3];
    end

endmodule

```

```

module counter(
    input clk, rst
    input          out_fast, //faster output
    output logic out //output
);
    logic [3:0] Q;
    logic [3:0] sum;

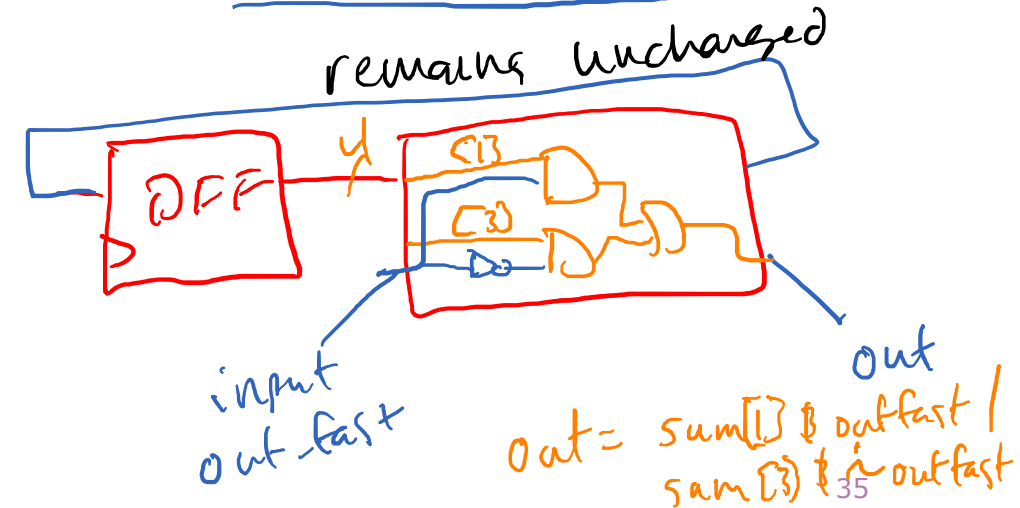
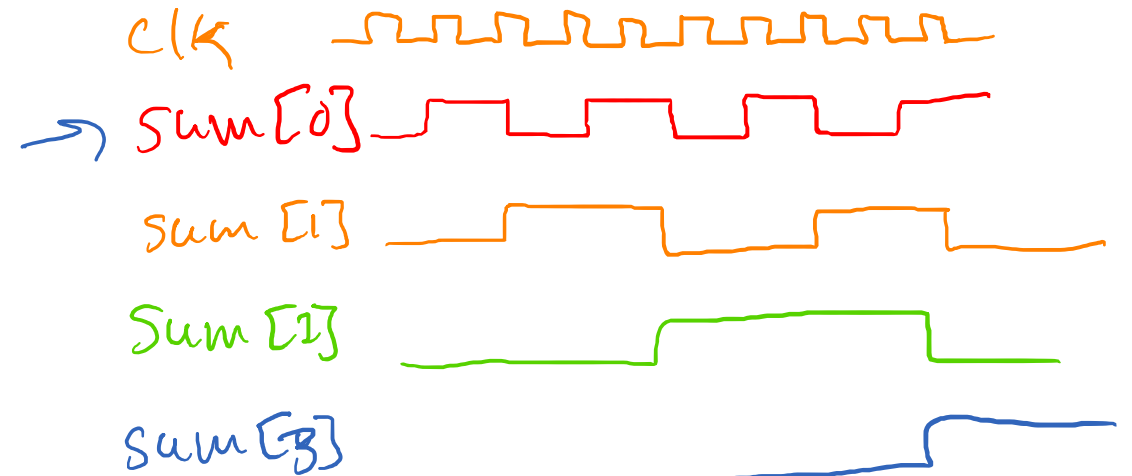
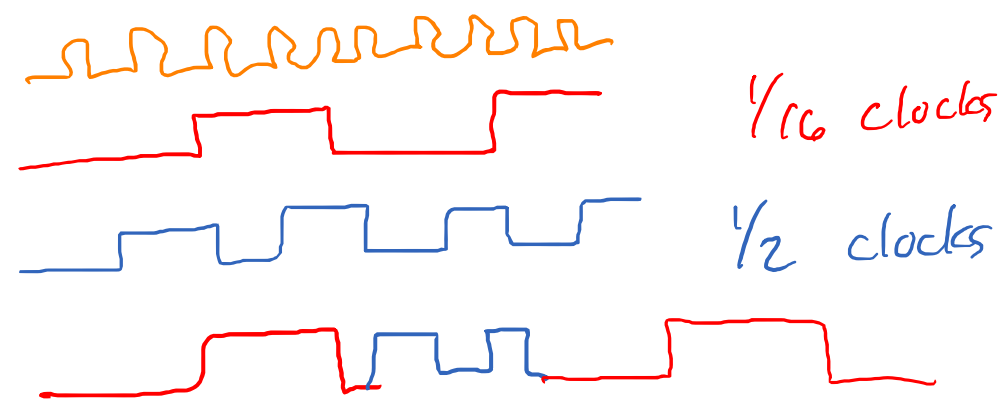
    always_ff @( posedge clk ) begin
        if (rst) Q <= 4'h0;
        else    Q <= sum;
    end

    always_comb begin
        sum = Q + 4'h1;
        ✓ out = sum[3]; //default
        ✓ if (out_fast) out = sum[1];
    end
endmodule

```

out_fast=0

out_fast=1



$$out = sum[1] \& out_fast \vee sum[3] \& \neg out_fast$$

```

module counter(
    input clk, rst
    input          out_fast, //faster output
    output logic   out      //output
);

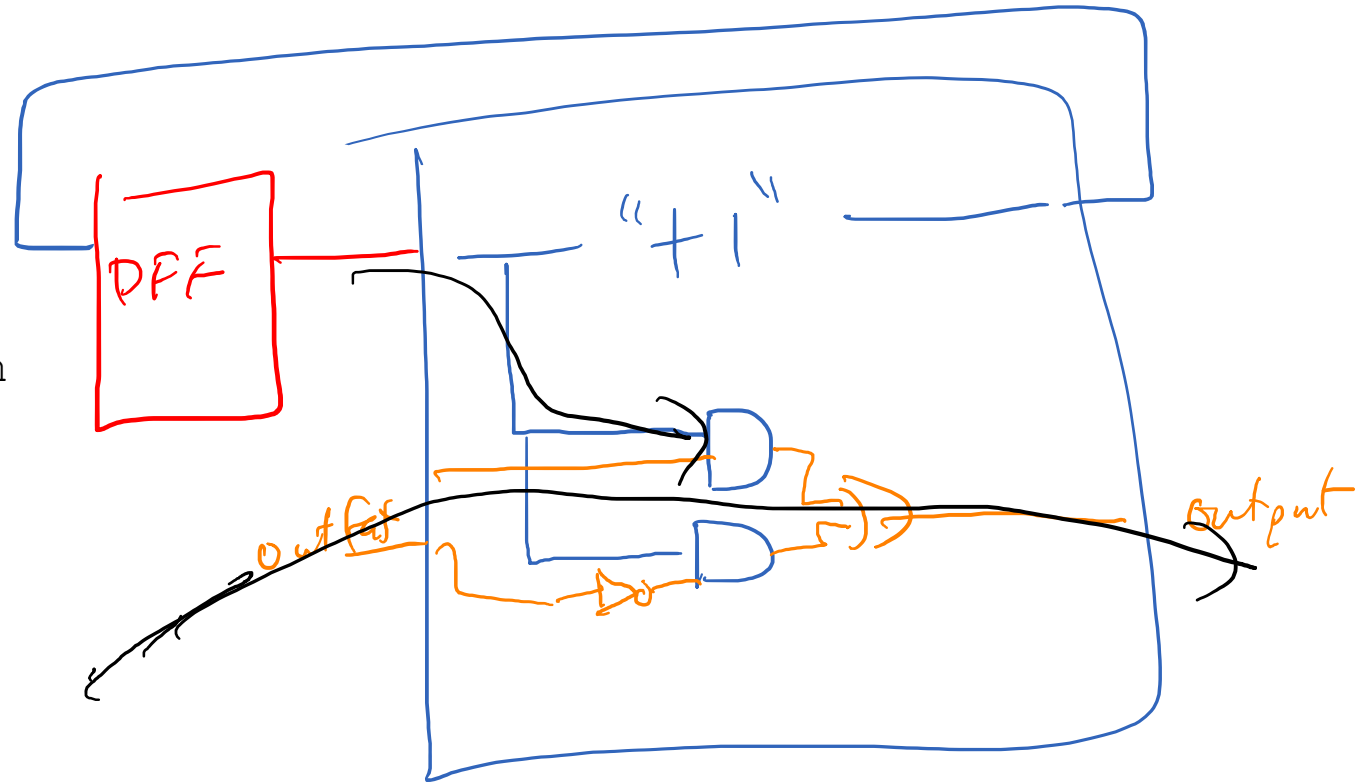
    logic [3:0] Q;
    logic [4:0] sum;

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

    always_comb begin
        sum = Q + 4'h1;
        out = sum[3];
        if (out_fast) out = sum[4];
    end

endmodule

```



```

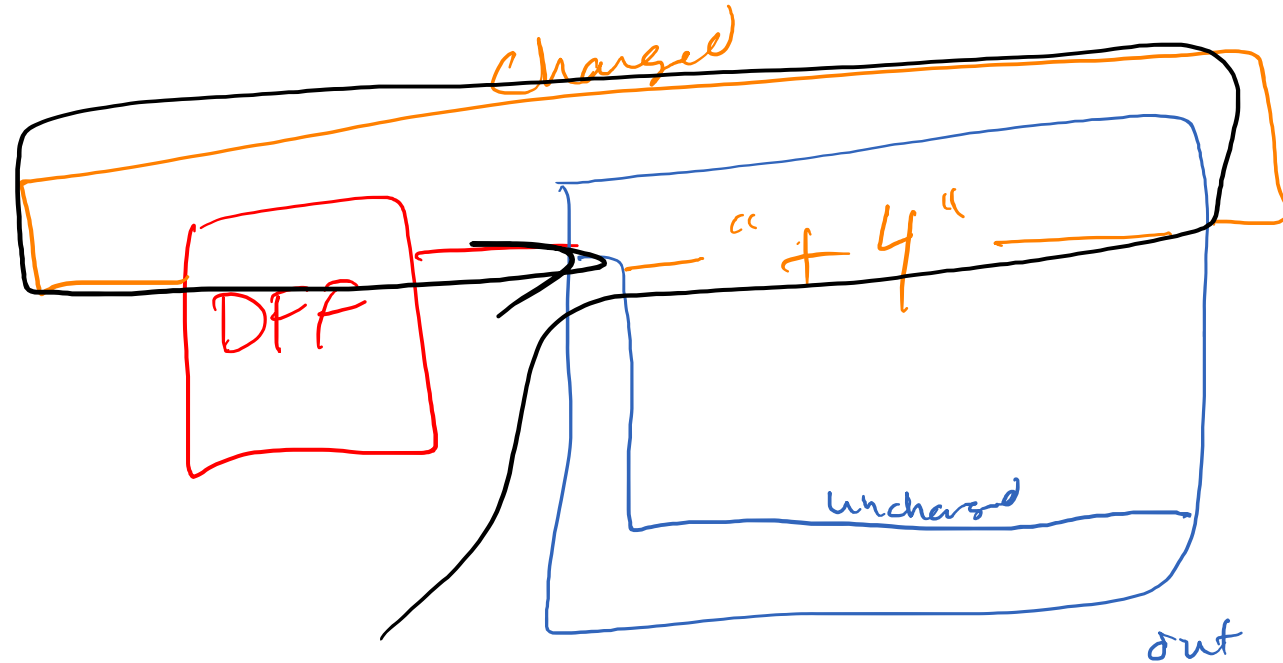
module counter(
    input clk, rst
    input          out_fast, //faster output
    output logic   out      //output
);
    logic [3:0] Q;
    logic [4:0] sum;

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

    always_comb begin
        sum = Q + 4'h1;
        out = sum[3];
        ↪ if (out_fast) sum = {0,Q} + 5'h4;
    end

endmodule

```



Next Time

- Finite State Machines (FSMs)