08 Memory elements

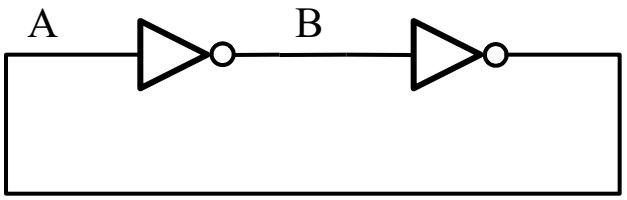
- Disclaimer
- Introduction to “memory” / metastability
- Setup and hold times
- Edge-triggered D flip-flop
    - Analysis / Setup & hold & clk2Q times
- Verilog for memory elements
- How to read waveforms
- Some Design rules for Verilog

Disclaimer

- This is the “memory element” used in conjunction with combinational logic design
  - These small memory elements are laid out together with combinational logic
- For larger memory blocks, many other possible technologies. They involve tradeoffs between simplicity, throughput, density, etc.
  - Register Arrays (RA)
  - embedded SRAM
  - embedded DRAM (eDRAM) – needs controller
  - Flash (non-volatile)
- Theory: everything can be abstracted to D-FFs.

A basic bistable element:

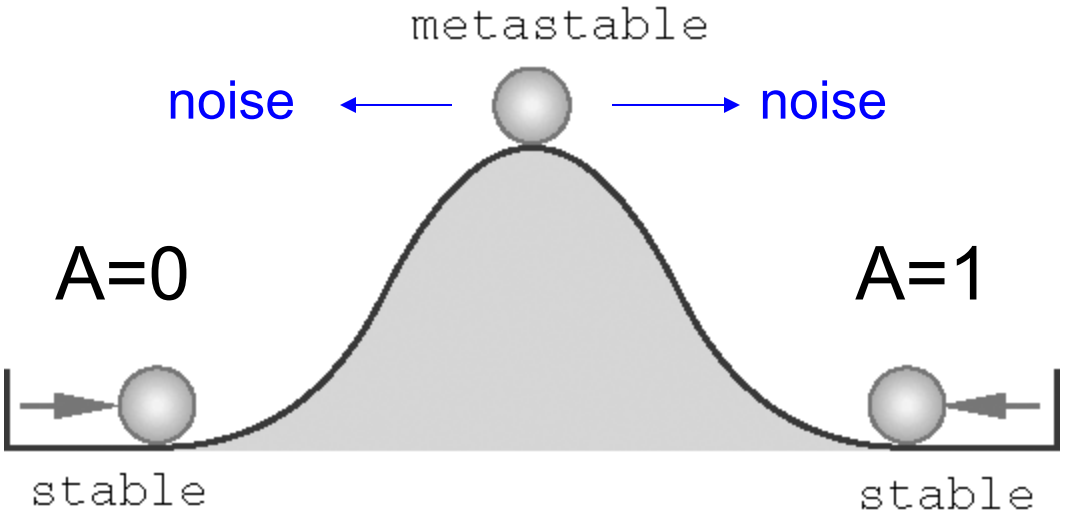
[BV]



2 possible states after power-up:

A=1, B=0 or A=0, B=1

Model: drop at top of hill, settle to A=0 or 1.

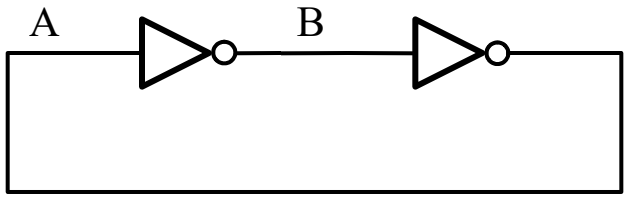


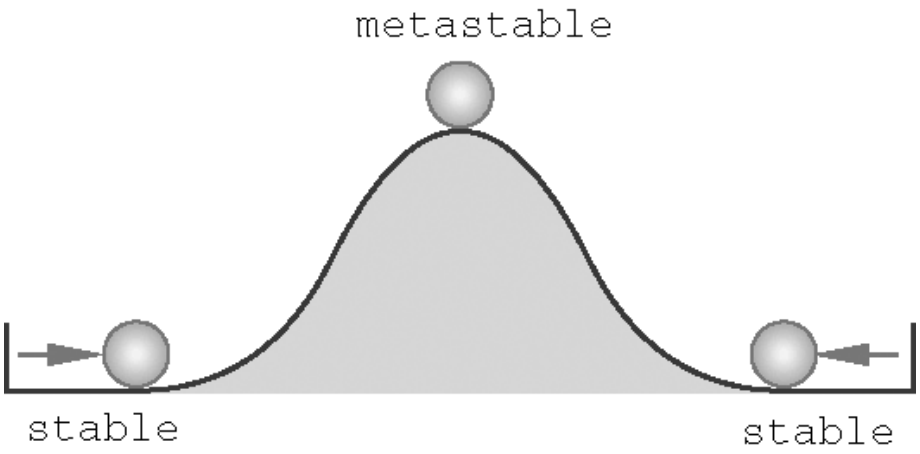
[W]

Metastability:

- outputs of gates eventually settle
- longer the wait, higher probability to settle
- while not settling, A=maybe, B=maybe
- will not worry about not settling in this course yet

[BV]

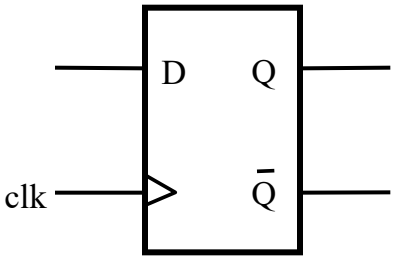




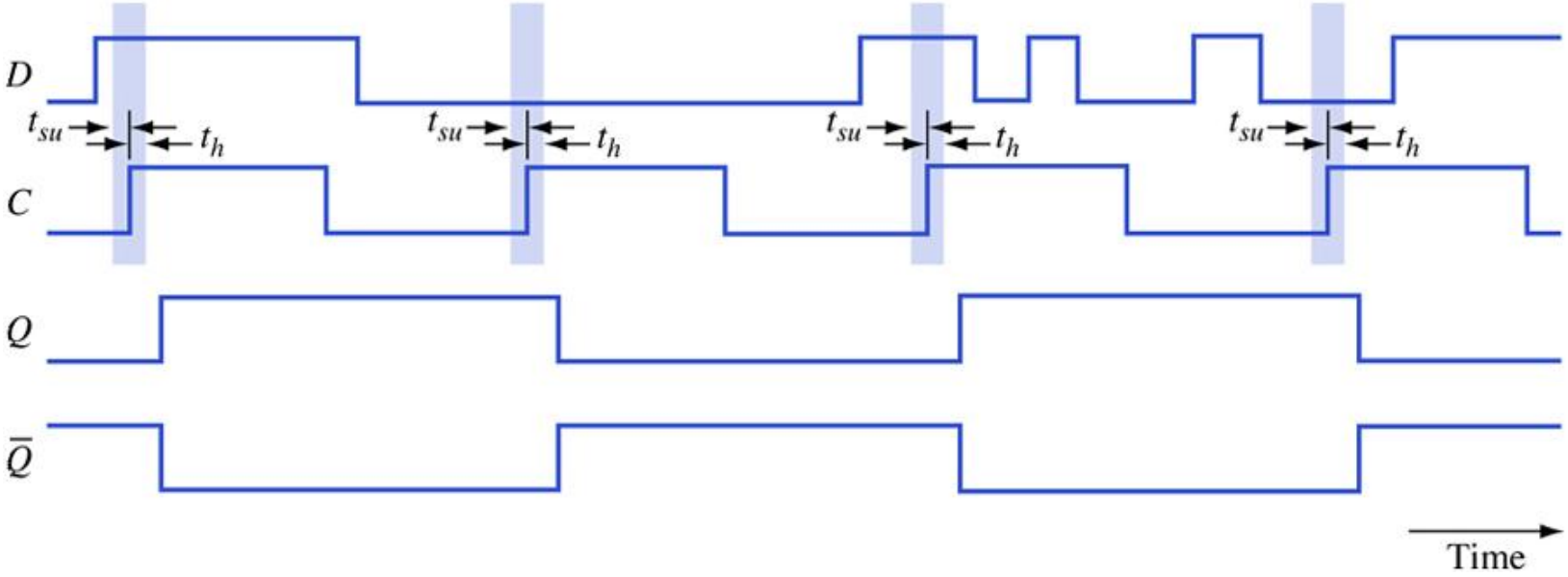
[W]

Edge-triggered D flip-flop (D-flop):

Review:

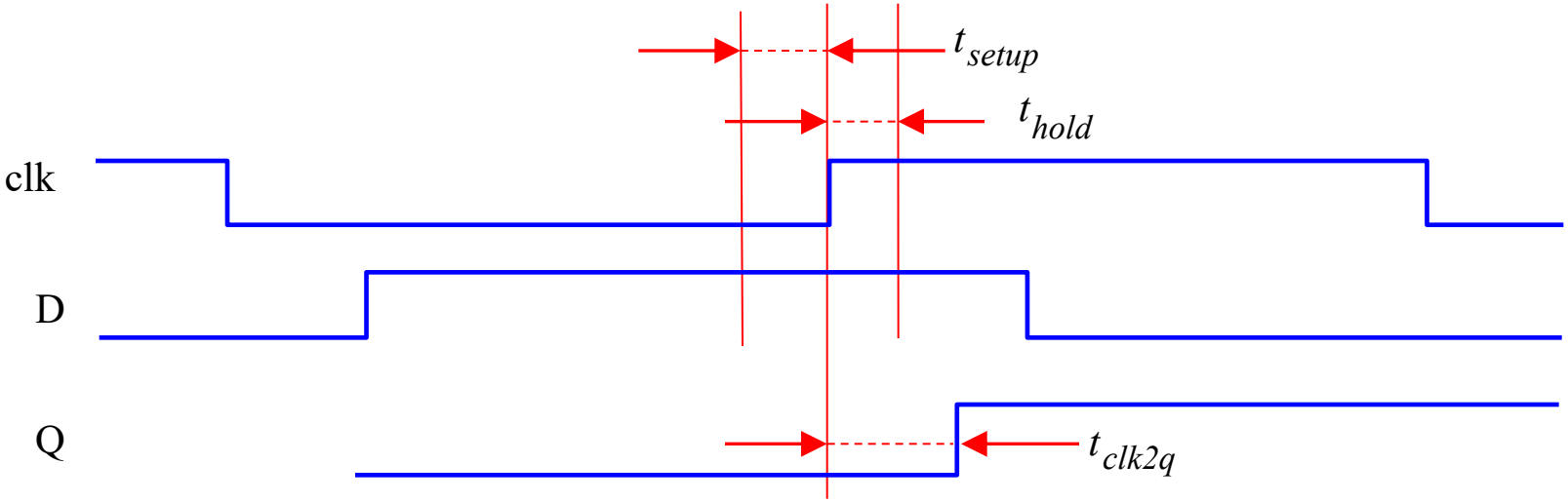


Description: At positive edge of clock, sample D. Wait a bit, show the sample at Q. Repeat.



[BV]

Setup, hold, and clk2q times:



D must be stable in time window around posedge clk.

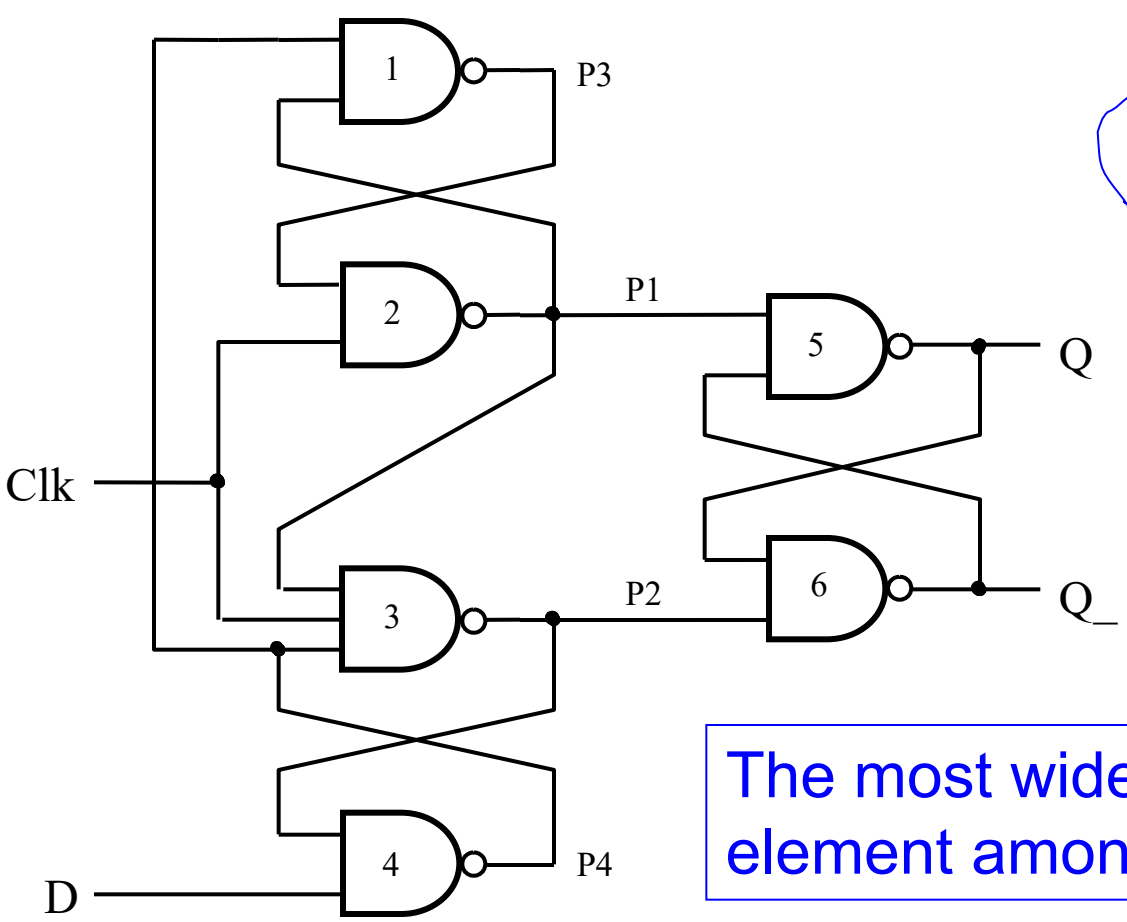
- D must be stable for setup time (t_{su}) before posedge clk
- D must be stable for hold time (t_h) after posedge clk
- Q changes after clk-to-Q time (t_{clk2q}) after posedge clk

t_{clk2q} is sometimes called t_{pDff} (flip-flop prop. delay)

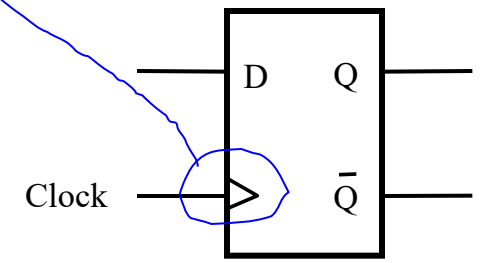
What if setup or hold time is violated? → Q becomes metastable!

[BV]

Edge-triggered D flip-flop (D-flop):



notch (>) here means edge-trigged



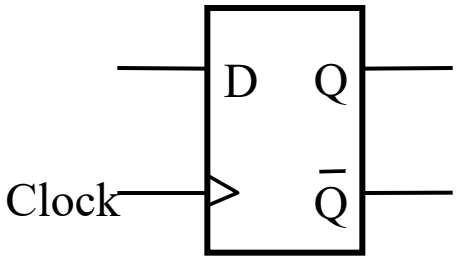
Graphical symbol

The most widely-used memory element among logic designers ☺

Q will ONLY sample value of D at positive edge of Clk.
Value of D around posedge Clk will be copied to Q.

[BV]

Edge-triggered D flip-flop (D-flop):



Graphical symbol

Clk	D	Q	Q_
0	x	No change	
1	x	No change	
↑	0	0	1
↑	1	1	0

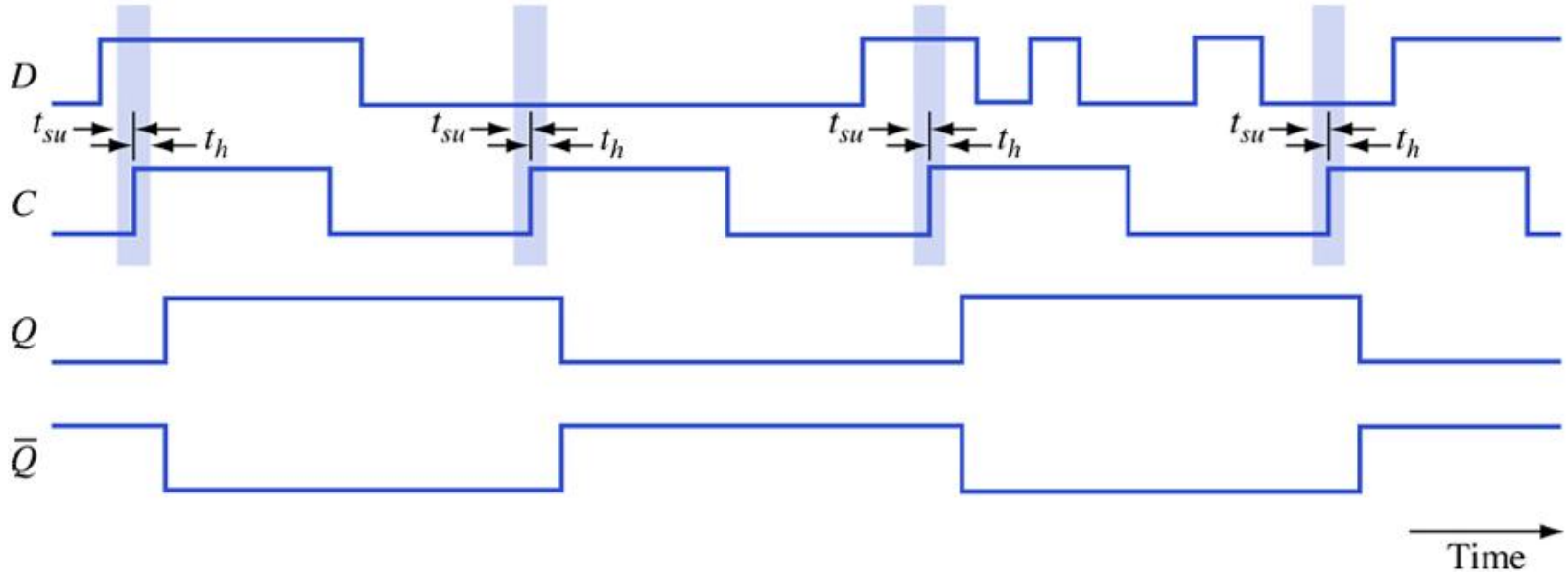
Truth table

This symbol means rising edge (positive edge) of the signal specified.

Q will ONLY sample value of D at positive edge of Clk.
Value of D around posedge Clk will be copied to Q.

[BV]

Edge sensitive behavior:



D will influence Q only at posedge clk.
After clk2Q delay, D that was *sampled* at posedge clk will appear at Q output.

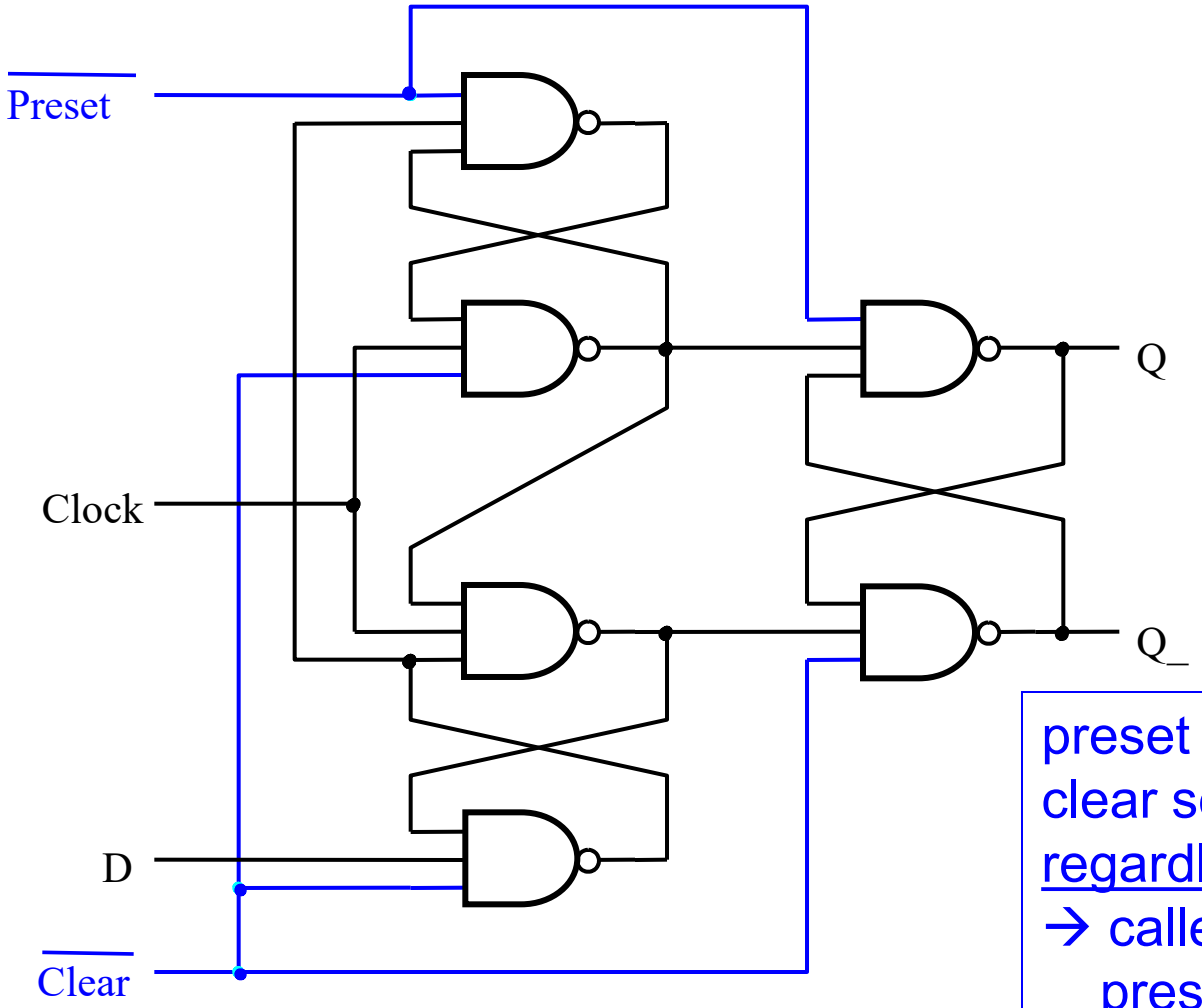
- This kind of memory element is called edge sensitive or flip-flop or simply flop.

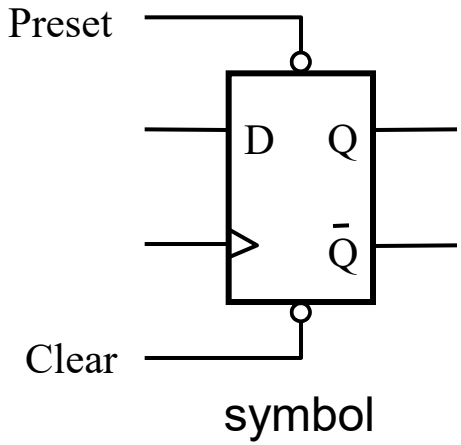
Q is immune to “noise” on D except at setup-hold window

[G]

D-flop with asynchronous preset and clear:

Startup state of Q is unknown, like any memory element:
We want to add ability to force Q to 0 or 1 at startup.



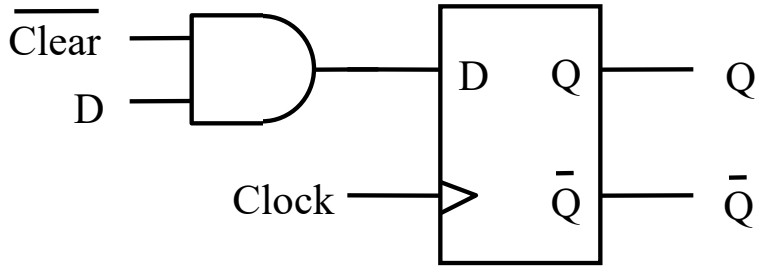


preset sets Q to 1
clear sets Q to 0
regardless of clk
→ called asynchronous preset and clear

[BV]

D-flop with synchronous clear:

We want to clear Q to 0, but only after posedge of clock.



[BV]

Assert clear signal during reset, and Q will be 0 after the next posedge clk.

This is a different design than asynchronous method:

- Less complex design in D-flop → smaller setup-hold-clk2q
- With asynchronous preset/clear, you have to be careful when preset and clear are deasserted with respect to posedge clk, otherwise metastability can happen.

Verilog for memory elements:

- Focus on D-flop with Q output
- We usually don't want Q_ output

Functional description: at positive edge of clock, sample the value of d. After some clk2q delay, q takes value of d.

```
module dflop (d, clk, q);
      input d, clk;
      output q;
      reg q;

      always @(posedge clk)
            q <=  #1  d;

endmodule
```

NEW SENSITIVITY LIST
code inside **always** block will only execute at posedge clk

→ matches functional description of D-flop

Verilog for memory elements:

- Focus on D-flop with Q output
- We usually don't want Q_ output

Functional description: at positive edge of clock, sample the value of d. After some clk2q delay, q takes value of d.

```
module dflop (d, clk, q);
    input d, clk;
    output q;
    reg q;

    always @(posedge clk)
        q <= #1 d;

endmodule
```

Just think about this whole thing as assignment (=) but for use with flip-flop inside
always @(posedge clk) only.

Treat as if this is =

Verilog for memory elements:

- Focus on D-flop with Q output

Functional description: at positive edge of clock, sample the value of d. After some clk2q delay, q takes value of d.

```
module dflop (d, clk, q);
      input d, clk;
      output q;
      reg q;

      always @(posedge clk)
            q <= #1  d;

endmodule
```

#delay (delay = some number)

In this example, #1 means
It will take 1 time unit delay
before D is assigned to Q.

(Our Verilog **simulator** has
the default value of 1nS for
1 time unit)
→ so clk2q time is 1nS here.

Verilog for memory elements:

- Focus on D-flop with Q output

```
`ifdef RTL_SIM
  `define CLK2Q #1
`else
  `define CLK2Q
`endif
```

```
module dflop (d, clk, q);
      input d, clk;
      output q;
      reg q;

      always @(posedge clk)
            q <=  `CLK2Q  d;

endmodule
```

When your design is going to be turned into a real chip:

(Depends on chip-lead)

1. Use substitution `CLK2Q – suppresses warnings but you need this `define chunk for every module

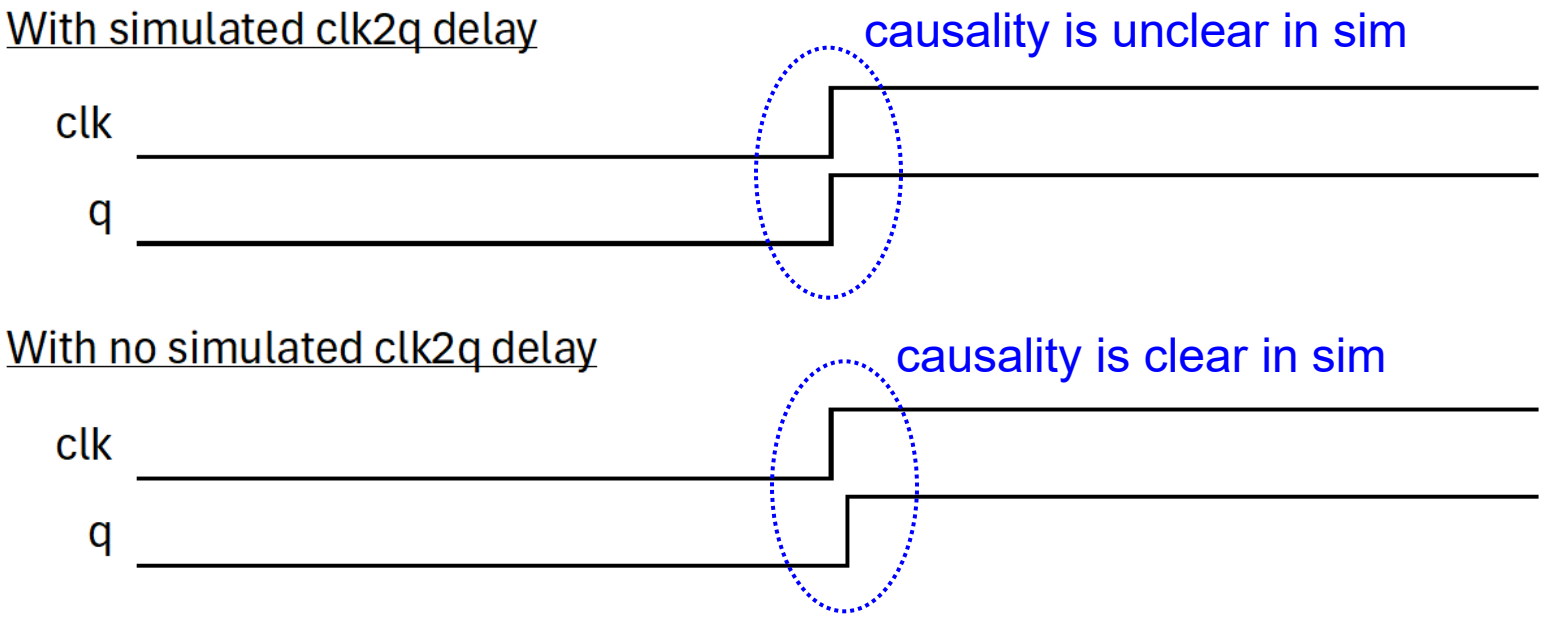
2. Use #1 and waive the warnings in synthesis. – universal delay element suppression in synthesis.

Verilog for memory elements:

Why do we want a small delay in simulation?

- real circuits actually have some positive clk2q

- having positive clk2q makes identifying bugs in simulation a lot easier. Eliminates “snap-at-wrong-edge” problem in waveform viewing.



Verilog for memory elements:

- Focus on D-flop with Q output

Functional description: at positive edge of clock, sample the value of d. After some clk2q delay, q takes value of d.

```
module dflop (d, clk, q);
      input d, clk;
      output q;
      reg q;

      always @(posedge clk)
            q <= #1 d;

endmodule
```

Non-blocking assignment

<= is the *non-blocking assignment* operator.

= is the *blocking assignment* operator.

Use <= when modeling flops.
Use = when modeling combinational logic.

Verilog for memory elements:

- Adding synchronous reset

```
module dflop(d, clk, q, reset);
  input d, clk, reset;
  output q;
  reg q;

  always @(posedge clk)
    if (reset)
      q <= #1 `RESET_VALUE;
    else
      q <= #1 d;
endmodule
```

Good View: This is a synchronous reset D-Flop with reset value 0

Bad View: This is a 2-to-1 MUX in front of a regular D-Flop

Verilog for memory elements:

- Adding asynchronous reset

```
module dflop(d, clk, q, reset);
  input d, clk, reset;
  output q;
  reg q;

  always @(posedge clk or posedge reset)
    if (reset)
      q <= #1 `RESET_VALUE;
    else
      q <= #1 d;
endmodule
```

View: This is an asynchronous reset D-Flop with reset value

Verilog for memory elements:

- Oh! So many types of reset signals:
- some reset signals are asserted high: `reset, rst`
- some reset signals are asserted low:

  `resetL, resetn, reset_, rstL, rstn, rst_`
- adjust sensitivity list accordingly:

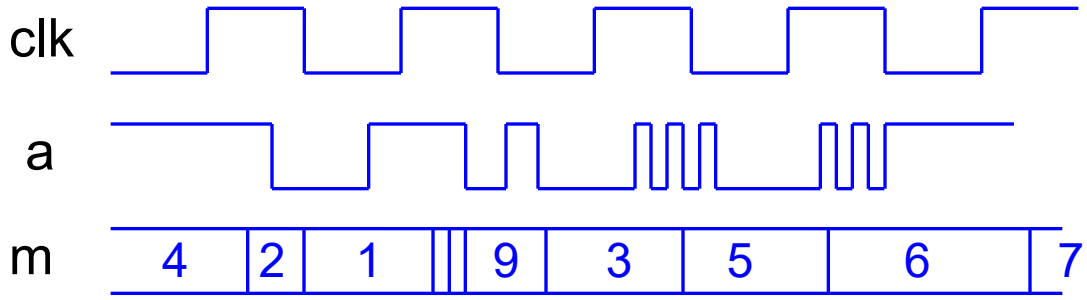
```
module dflop(d, clk, q, resetL);
  input d, clk, resetL;
  output q;
  reg q;

  always @(posedge clk or negedge resetL)
    if (reset)
      q <= #1 `RESET_VALUE;
    else
      q <= #1 d;
endmodule
```

Recommendation:

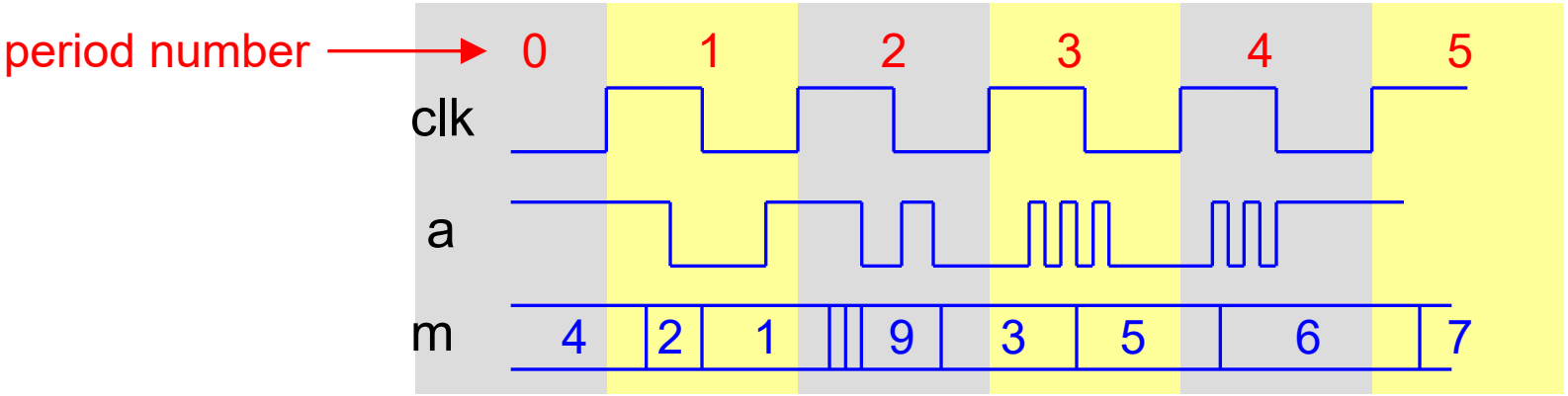
If it's your own design, avoid asserted low logic as much as possible.

Use it only if you are interfacing with vendor/3rd-party modules.

How to read waveforms:

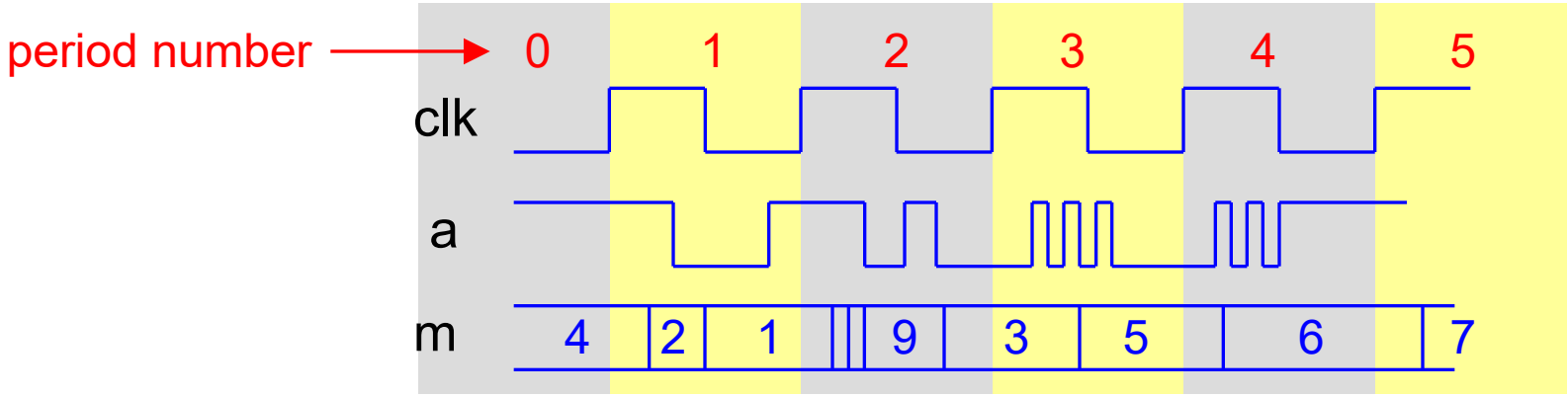


- clock period begins and ends on posedge clk
- clock period numbers are arbitrary
- read value of signals at the end of clock period



How to read waveforms:

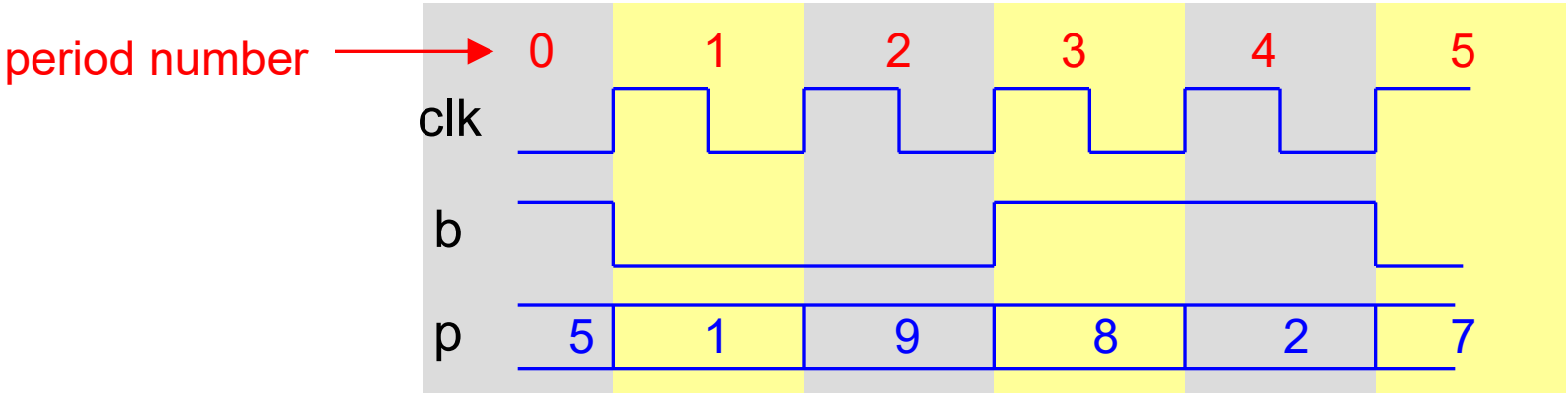
- read value of signals at the end of clock period (at posedge clk)



clk #	0	1	2	3	4	5
a	1	1	0	0	1	?
m	4	1	3	5	6	?

How to read waveforms:

- some data sheets draw this:

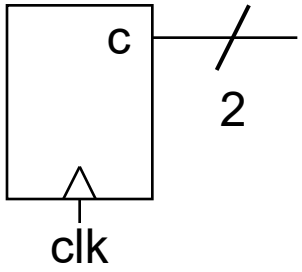


- just assume signal runs a bit past posedge clk

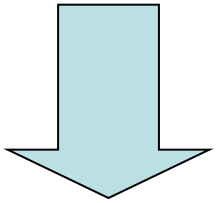
clk #	0	1	2	3	4	5
b	1	0	0	1	1	?
p	5	1	9	8	2	?

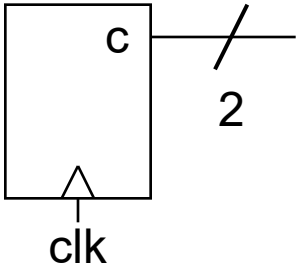
How to read waveforms:

- sometimes even the table is omitted:



clk #	1	2	3	4	5	6	7	8	…
c	0	1	2	3	0	1	2	3	…





c: 0, 1, 2, 3, 0, 1, 2, 3, …

DESIGN RULE 1

Use = (blocking assignment) to model combinational logic.

Use <= (non-blocking assignment) to model memory elements.

DESIGN RULE 2

Avoid mixing combinational logic and memory elements in the same always block.

Separate combinational logic and memory into different always blocks.

Rules are meant to be broken

Some more experienced designers sometimes violate Rule 2 (myself included), **BUT**

They know exactly what the circuit will do, and they never break Rule 1

What headache if I break the rules?

- Simulation and actual hardware outputs MAY NOT MATCH

  ➔ Design is no good if it works only in simulation and not in actual hardware

- Code is hard to debug to work in simulation

- Actual hardware is even harder to debug

Blocking vs. non-blocking assignments:

Blocking assignments (=)

= evaluate and assign value immediately

BEFORE executing next line of code.

Non-blocking assignments (<=)

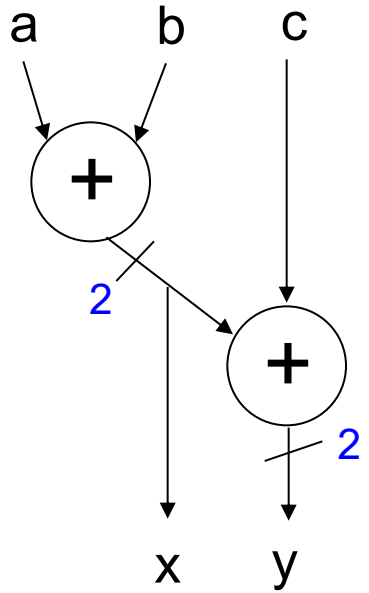
<= evaluate immediately but

assign value at the end

of the current simulation time.

Example (combinational logic):

We want to generate a+b and a+b+c.
We name them x and y.

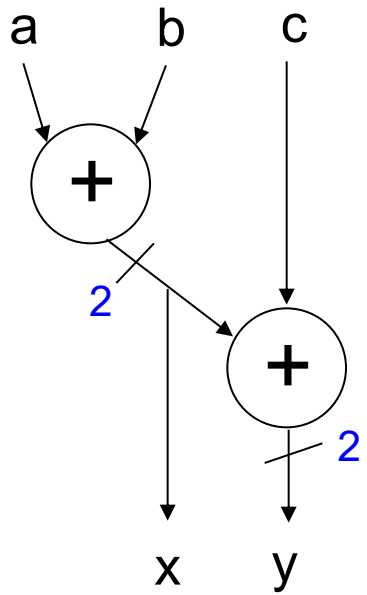


correct code fragment:

```
always @ (a or b or c) begin
        x = a + b;    ← evaluate a+b and assign it to x before next line
        y = x + c;    ← y will then have the correct value, (a + b) + c
end
```

Example (combinational logic):

We want to generate a+b and a+b+c.
We name them x and y.



bad code fragment:

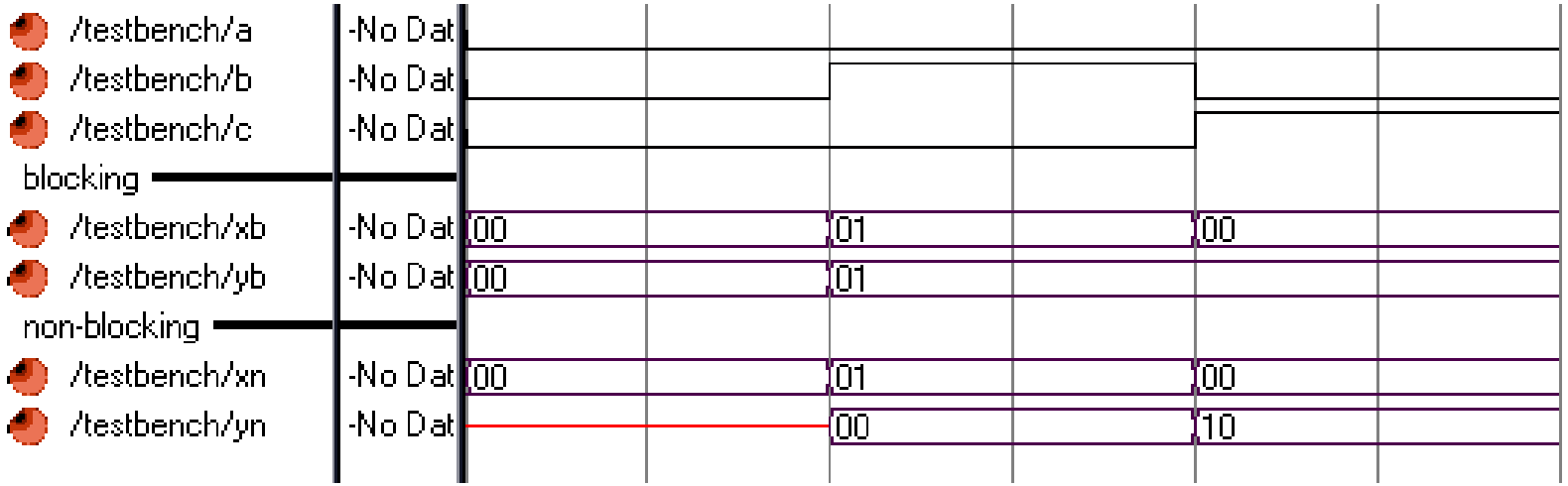
BUG!

```
always @ (a or b or c) begin
        x <= a + b;    ← evaluate a+b but wait to assign to x later
        y <= x + c;    ← evaluate y to be (old_value_of_x) + c
end     ← x and y are assigned here
```

Test code and simulation for example:

```
module block_nonblock (a, b, c, xb, yb, xn, yn);
   input a, b, c;
   output [1:0] xb, yb, xn, yn;    // xb, yb = x and y from blocking assgn
   reg [1:0]    xb, yb, xn, yn;    // xn, yn = x and y for non-blocking assgn.

        always @(a or b or c) begin
                xb = a + b;        // blocking assignment
                yb = xb + c;
        end

        always @(a or b or c) begin
                xn <= a + b;       // non-blocking assignment
                yn <= xn + c;
        end

endmodule
```

Test code and simulation for example:



a	b	c	xb (a+b)	yb (a+b+c)	old_x	xn (a+b)	yn (old_x +c)
0	0	0	00	00	xx	00	xx
0	1	0	01	01	00	01	00
0	0	1	00	01	01	00	10