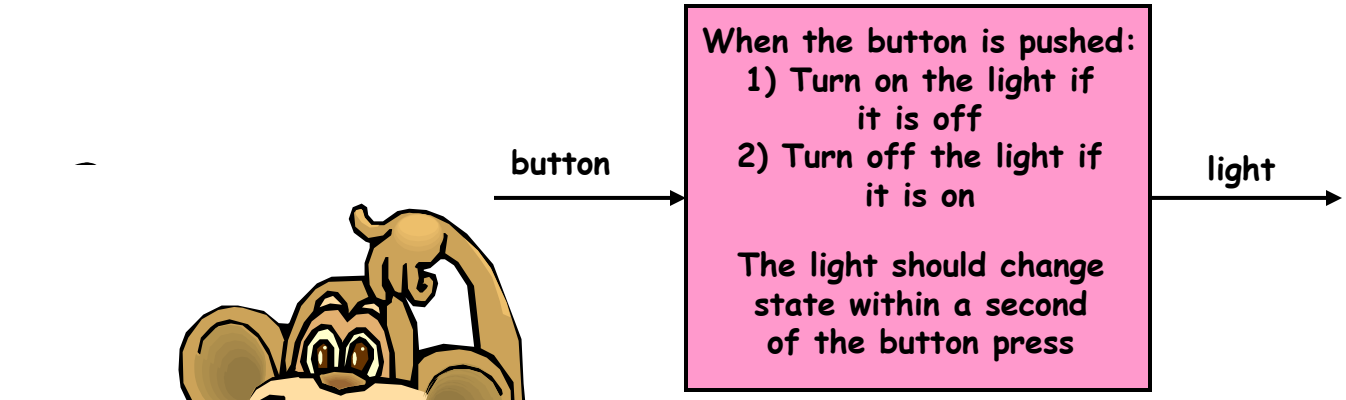
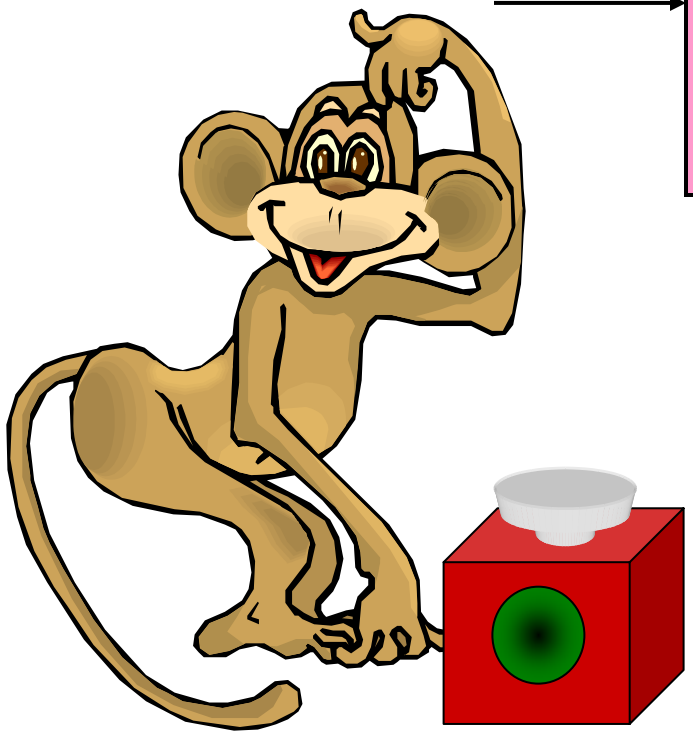
Something We Can't Build (Yet)

What if you were given the following design specification:



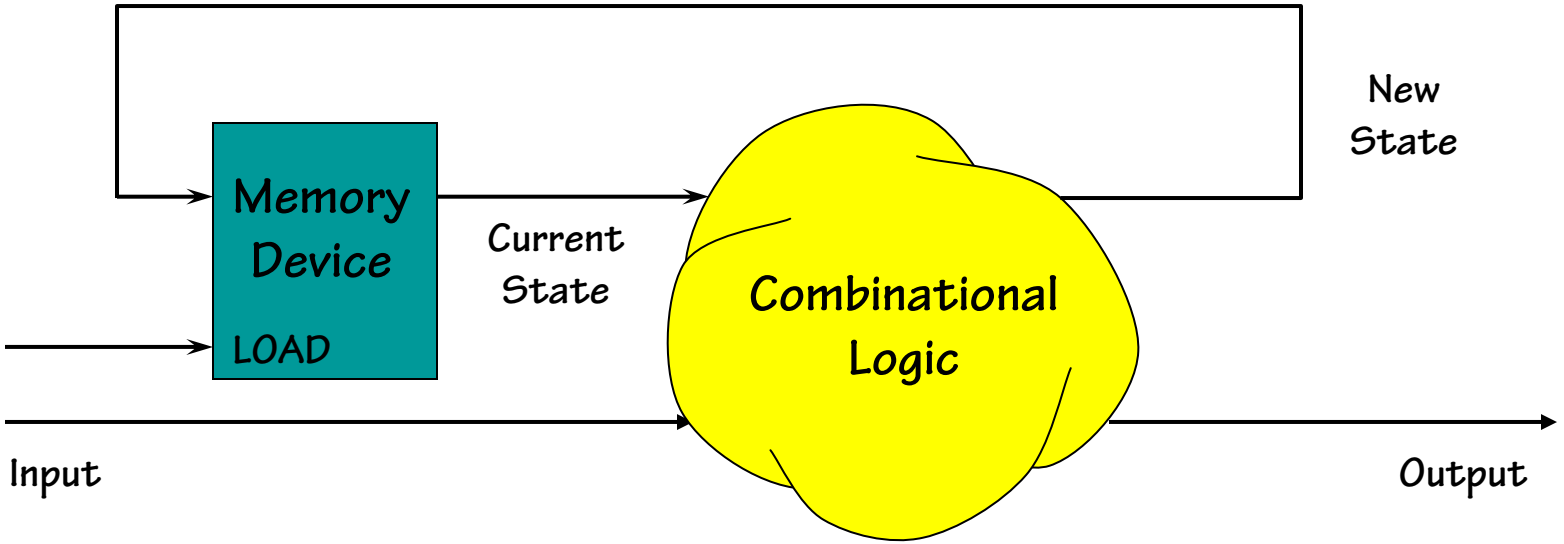


What makes this circuit so different from those we've discussed before?

1. "State" – i.e. the circuit has memory
2. The output was changed by a input "event" (pushing a button) rather than an input "value"

Digital State

One model of what we'd like to build



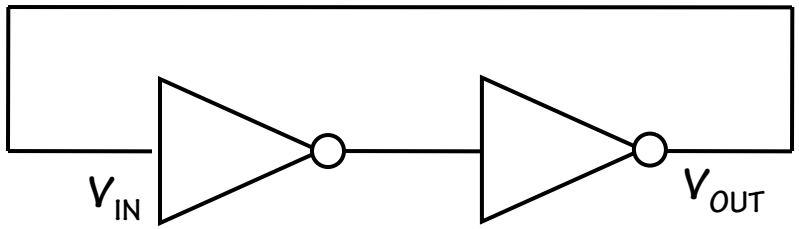
Plan: Build a Sequential Circuit with stored digital STATE –

- Memory stores CURRENT state, produced at output
- Combinational Logic computes
  - NEXT state (from input, current state)
  - OUTPUT bit (from input, current state)
- State changes on LOAD control input

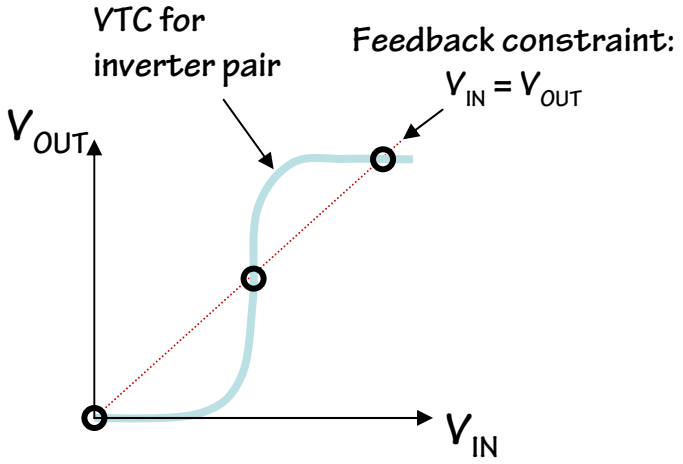
Storage: Using Feedback

IDEA: use positive feedback to maintain storage indefinitely.

Our logic gates are built to restore marginal signal levels, so noise shouldn't be a problem!



Result: a bistable storage element



Not affected by noise

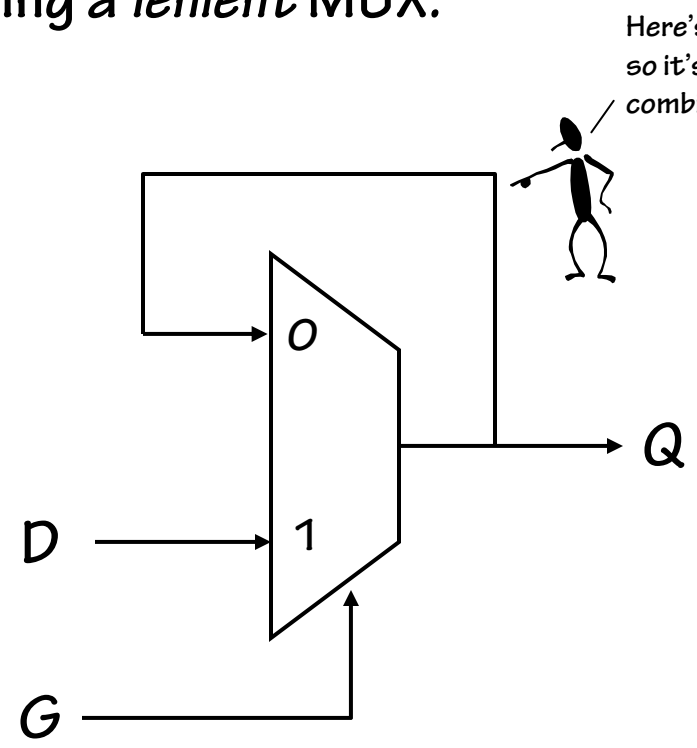
Three solutions:

- two end-points are stable
- middle point is unstable

We'll get back to this!

Settable Storage Element

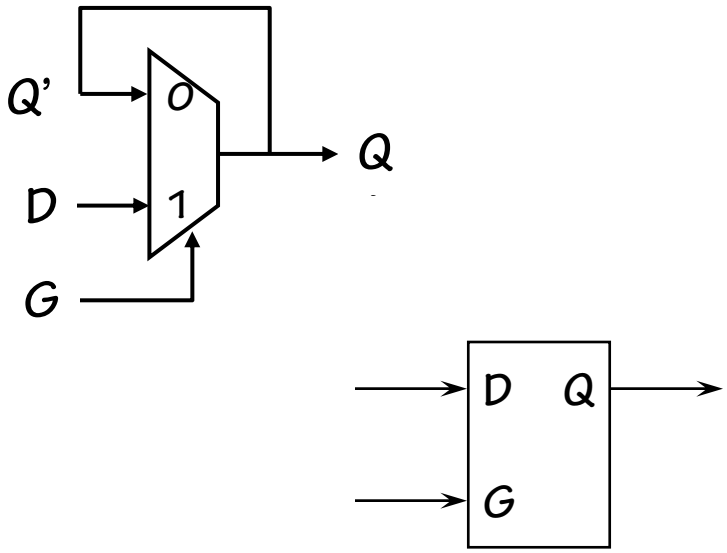
It's easy to build a settable storage element (called a latch) using a *lenient* MUX:



"state" signal appears as both input and output

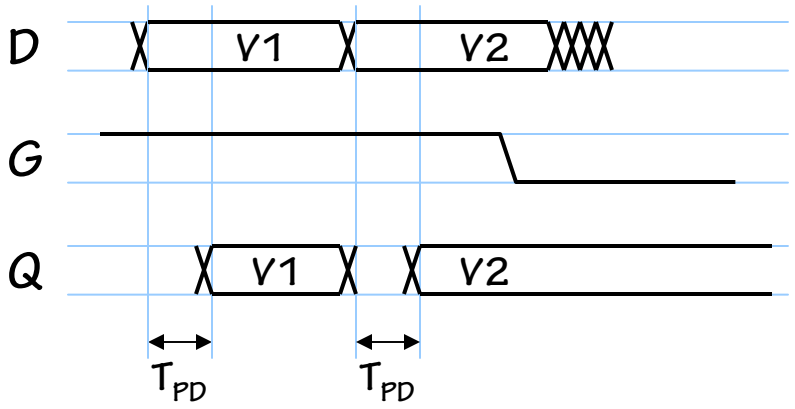
G	D	Q_{IN}	Q_{OUT}	
0	--	0	0	Q stable
0	--	1	1	
1	0	--	0	Q follows D
1	1	--	1	

New Device: D Latch



G=1: Q follows D

G=0: Q holds

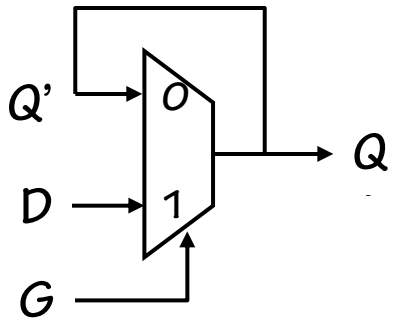


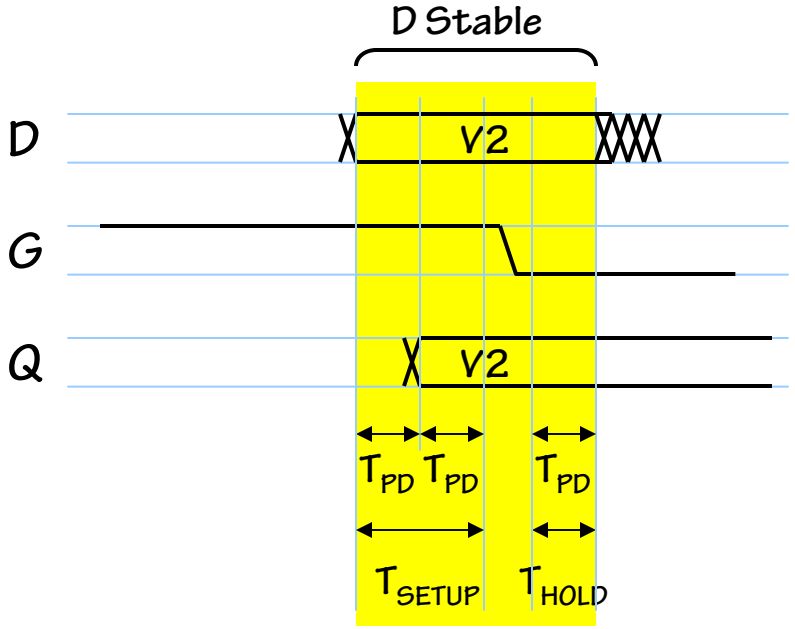
BUT… A change in D or G contaminates Q, hence Q' … how can this possibly work?

G=1: Q Follows D, *independently of Q'*

G=0: Q Holds stable Q', *independently of D*

D-Latch timing





To reliably latch V2:

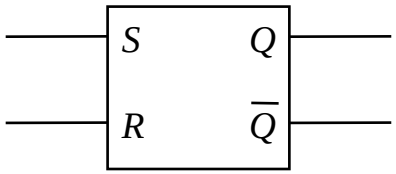
- Apply V2 to D, holding G=1
- After T_{PD}, V2 appears at Q=Q'
- After another T_{PD}, Q' & D both valid for T_{PD}; *will hold Q=V2 independently of G*
- Set G=0, while Q' & D hold Q=D
- After another T_{PD}, G=0 and Q' are sufficient to hold Q=V2 *independently of D*

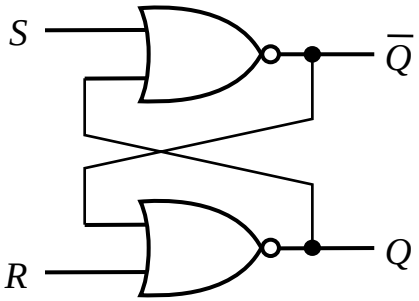
Dynamic Discipline **for our latch:**

$T_{SETUP} = 2T_{PD}$: interval *prior to G* transition for which D must be stable & valid

$T_{HOLD} = T_{PD}$: interval *following G* transition for which D must be stable & valid

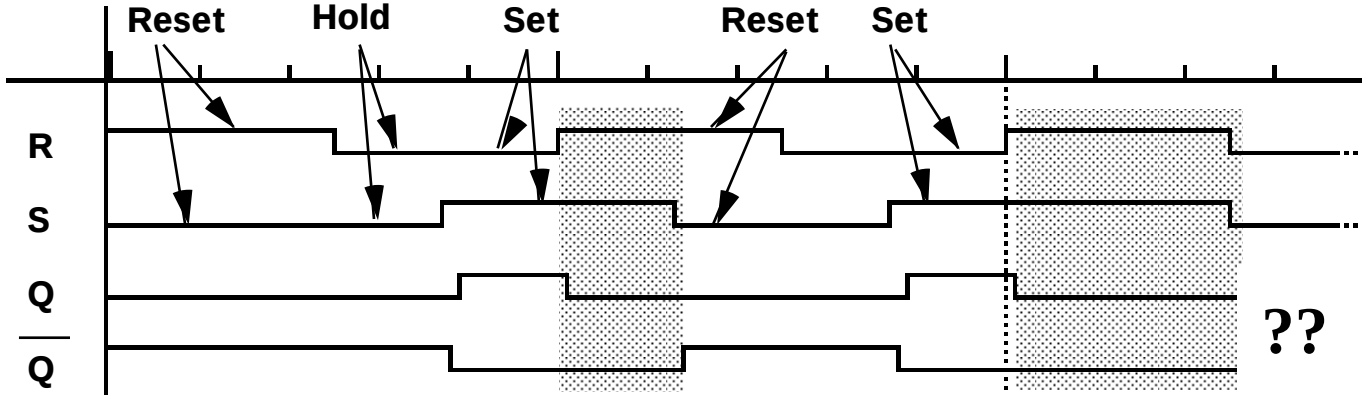
NOR-based Set-Reset (SR) Flipflop





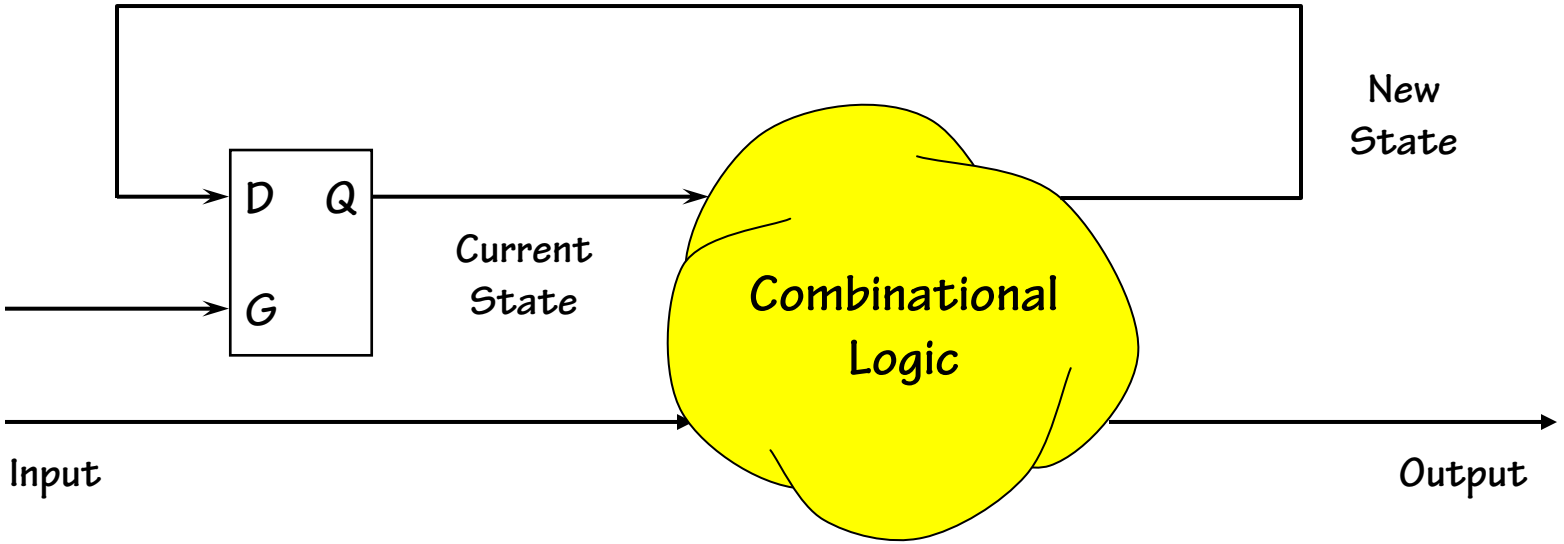
S	R	Q	$\overline{Q}$
0	0	Q	$\overline{Q}$
1	0	1	0
0	1	0	1
1	1	0	0

Forbidden State



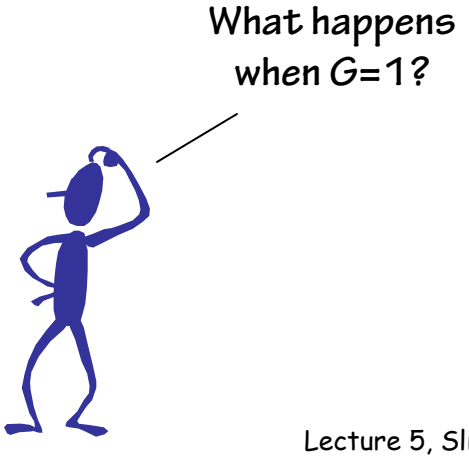
Flip-flop refers to a bi-stable element

Lets try using the D-Latch…

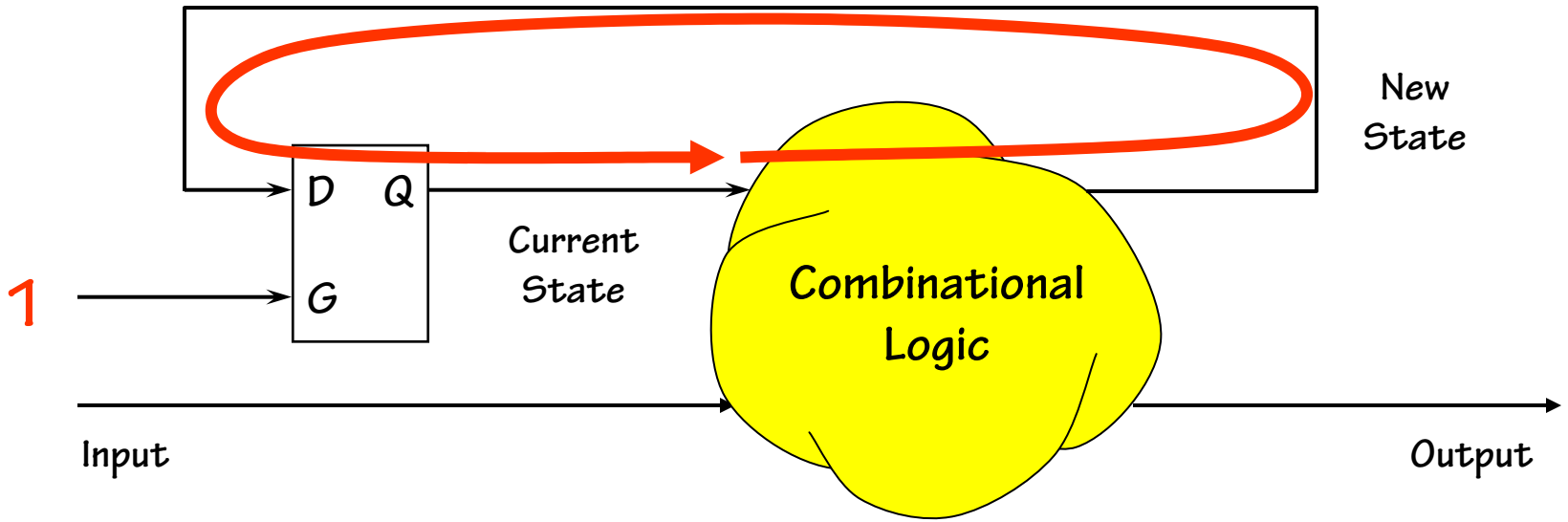


Plan: Build a Sequential Circuit with one bit of STATE –

- Single latch holds CURRENT state
- Combinational Logic computes
  - NEXT state (from input, current state)
  - OUTPUT bit (from input, current state)
- State changes when G = 1 (briefly!)



Combinational Cycles



When G=1, latch is *Transparent…*

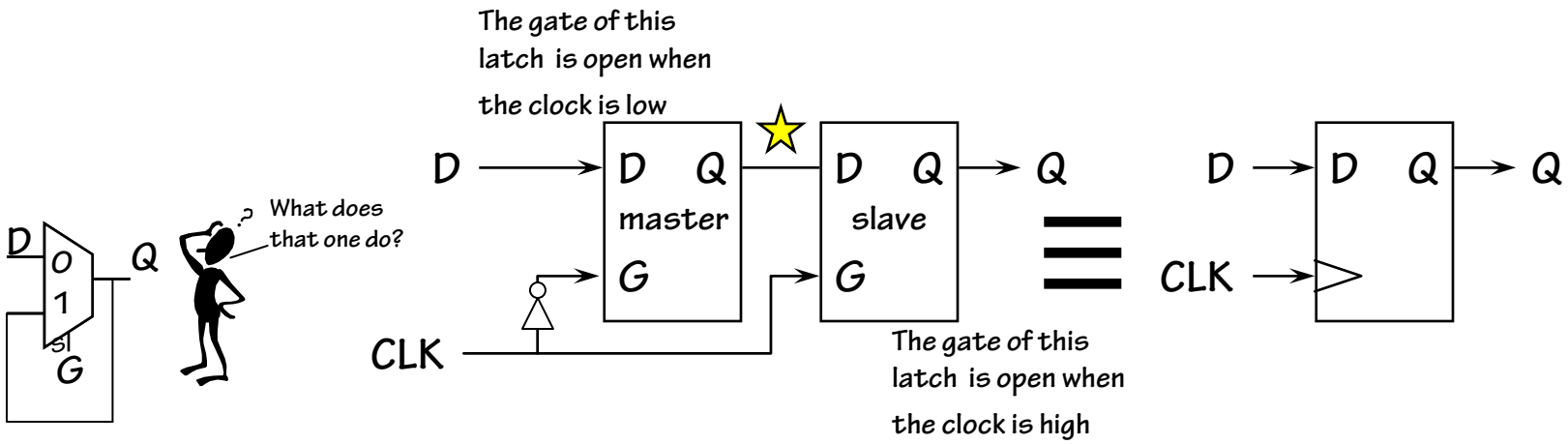
… provides a combinational path from D to Q.

Can't work without tricky timing constrants on G=1 pulse:

- Must fit within contamination delay of logic
- Must accommodate latch setup, hold times

Want to signal an INSTANT, not an INTERVAL…

Edge-triggered D-Register



Observations:

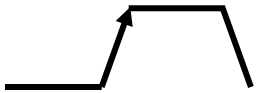
- only one latch "transparent" at any time:
  - master closed when slave is open
  - slave closed when master is open

  → no combinational path through flip flop

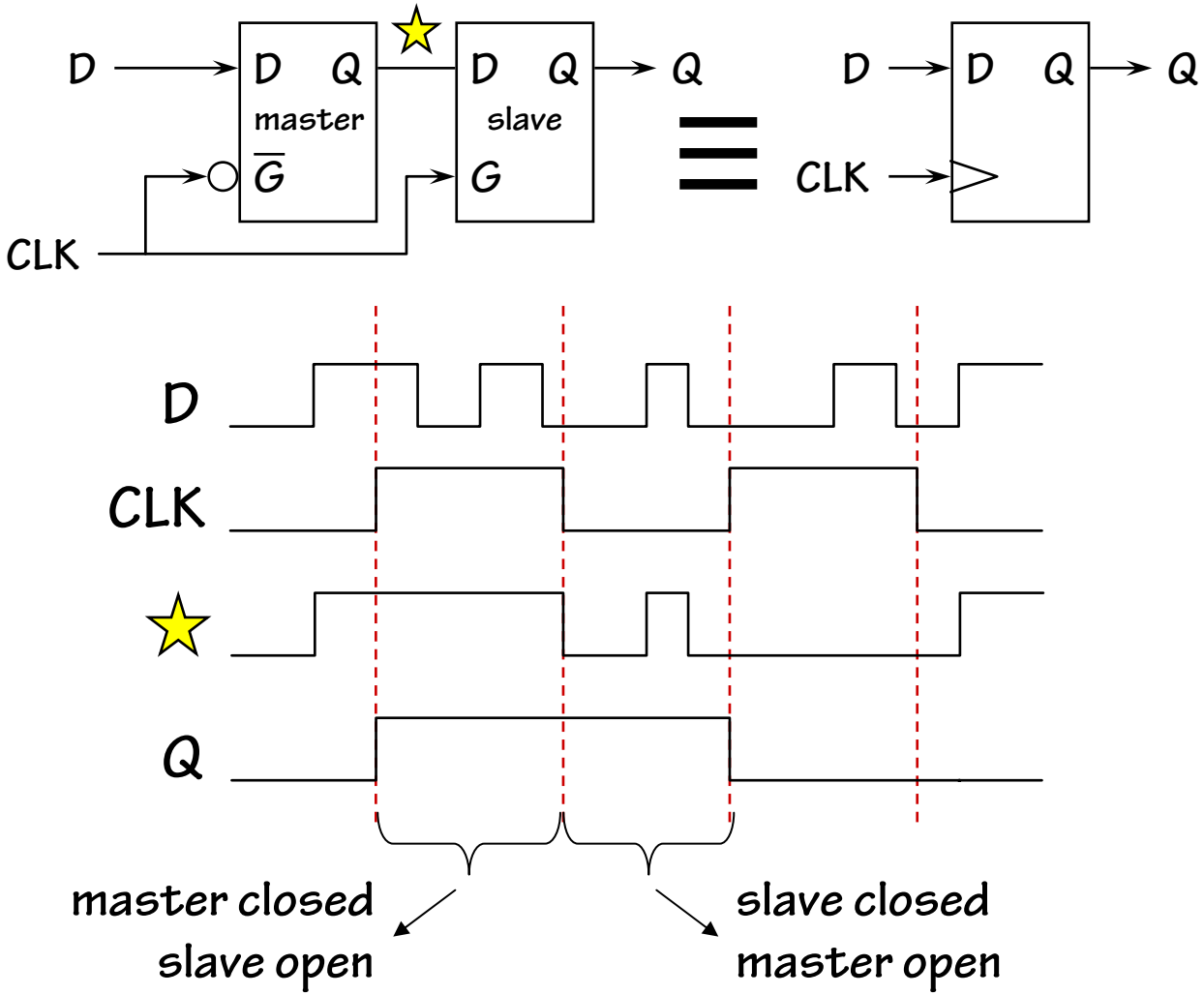
  (the feedback path in one of the master or slave latches is always active)

- Q only changes shortly after 0 → 1 transition of CLK, so flip flop appears to be "triggered" by rising edge of CLK

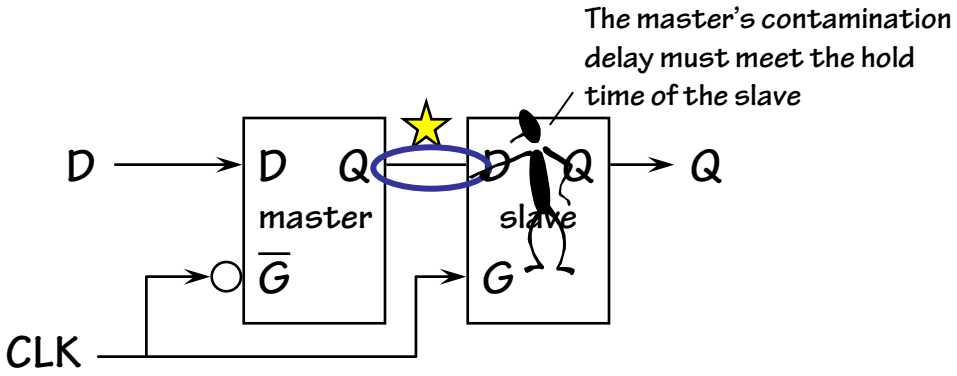
Transitions mark *instants*, not intervals



D-Register Waveforms



Um, about that hold time…

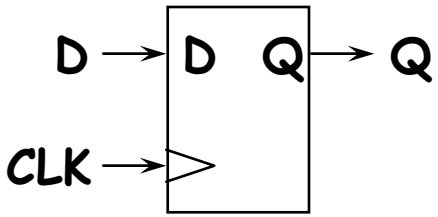


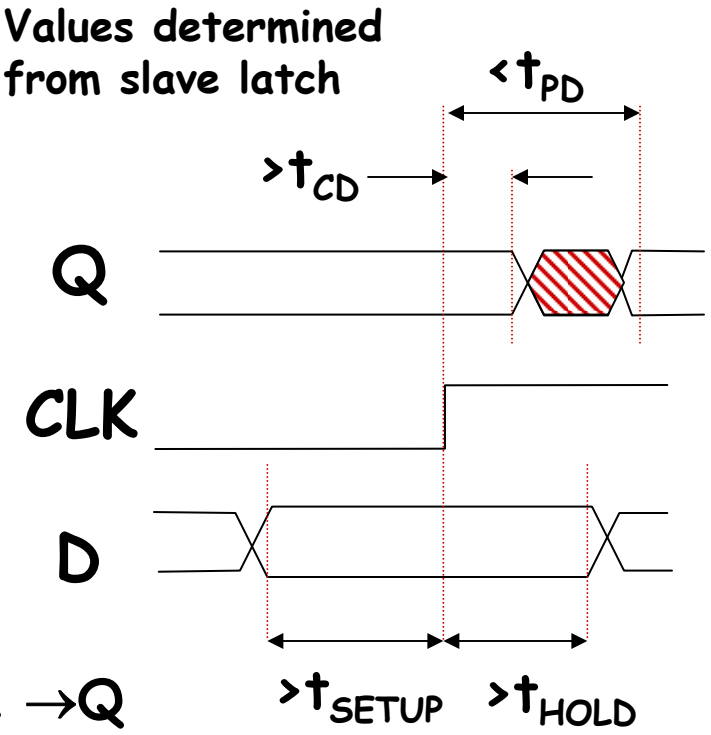
Consider HOLD TIME requirement for slave:

- Negative (1 →0) clock transition → slave freezes data:
  - SHOULD be no output glitch, since master held constant data; BUT
  - master output contaminated by change in G input!
- HOLD TIME of slave not met, UNLESS we assume sufficient contamination delay in the path to its D input!

Accumulated t_{CD} thru inverter, G → Q path of master must cover slave t_{HOLD} for this design to work!

D-Register Timing - I





t_{PD}: maximum propagation delay, CLK →Q

t_{CD}: minimum contamination delay, CLK →Q

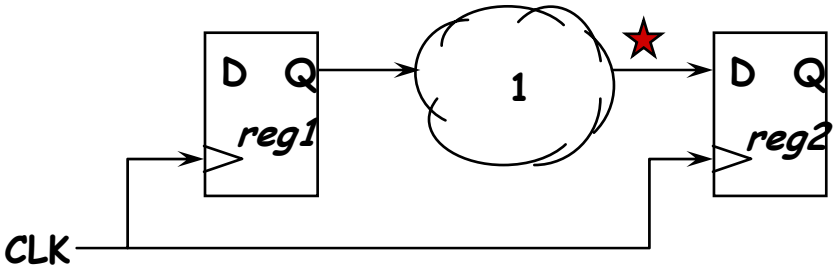
t_{SETUP}: setup time

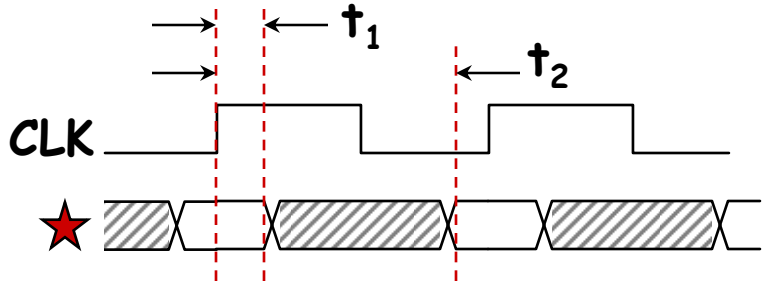
guarantee that D has propagated through feedback path before master closes

t_{HOLD}: hold time

guarantee master is closed and data is stable before allowing D to change

D-Register Timing - II





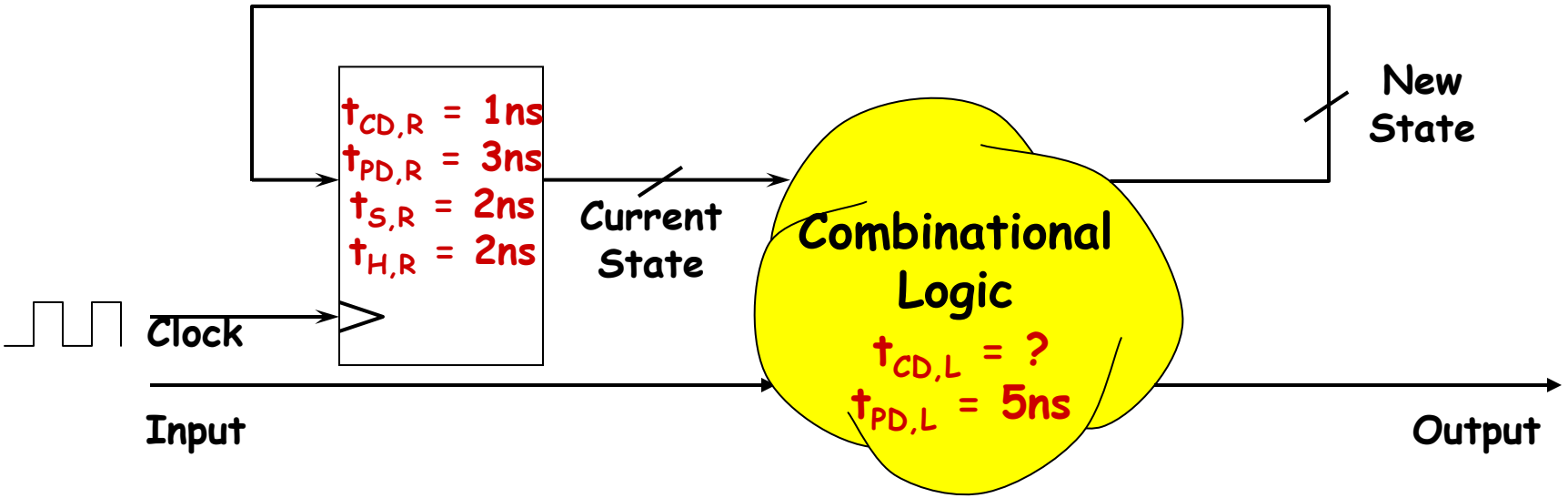
$t_1 = t_{CD,reg1} + t_{CD,1} > t_{HOLD,reg2}$

$t_2 = t_{PD,reg1} + t_{PD,1} < t_{CLK} - t_{SETUP,reg2}$

Questions for register-based designs:

- how much time for useful work (i.e. for combinational logic delay)?
- does it help to guarantee a minimum t_{CD}? How about designing registers so that $t_{CD,reg} > t_{HOLD,reg}$?
- what happens if CLK signal doesn't arrive at the two registers at exactly the same time (a phenomenon known as "clock skew")?

Sequential Circuit Timing



Questions:

- Constraints on T_{CD} for the logic? > 1 ns
- Minimum clock period? > 10 ns ($T_{PD,R}+T_{PD,L}+T_{S,R}$)
- Setup, Hold times for Inputs? $T_S = T_{PD,L} + T_{S,R}$, $T_H = T_{H,R} - T_{CD,L}$

This is a simple *Finite State Machine* … more on next time!

The Sequential `always` Block

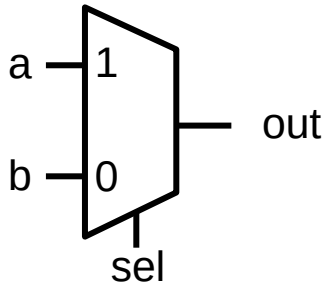
- Edge-triggered circuits are described using a sequential `always` block

Combinational

```
module combinational(a, b, sel,
                             out);
  input a, b;
  input sel;
  output out;
  reg out;

  always @ (a or b or sel)
  begin
    if (sel) out = a;
    else out = b;
  end

endmodule
```

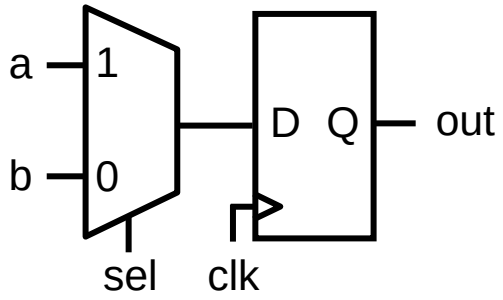


Sequential

```
module sequential(a, b, sel,
                        clk, out);
  input a, b;
  input sel, clk;
  output out;
  reg out;

  always @ (posedge clk)
  begin
    if (sel) out <= a;
    else out <= b;
  end

endmodule
```



Importance of the Sensitivity List

- The use of `posedge` and `negedge` makes an `always` block sequential (edge-triggered)
- Unlike a combinational `always` block, the sensitivity list **does** determine behavior for synthesis!

D Flip-flop with synchronous clear

```
module dff_sync_clear(d, clearb,
clock, q);
input d, clearb, clock;
output q;
reg q;
always @ (posedge clock)
begin
  if (!clearb) q <= 1'b0;
  else q <= d;
end
endmodule
```

`always` block entered only at each positive clock edge

D Flip-flop with asynchronous clear

```
module dff_async_clear(d, clearb, clock, q);
input d, clearb, clock;
output q;
reg q;

always @ (negedge clearb or posedge clock)
begin
  if (!clearb) q <= 1'b0;
  else q <= d;
end
endmodule
```

`always` block entered immediately when (active-low) clearb is asserted

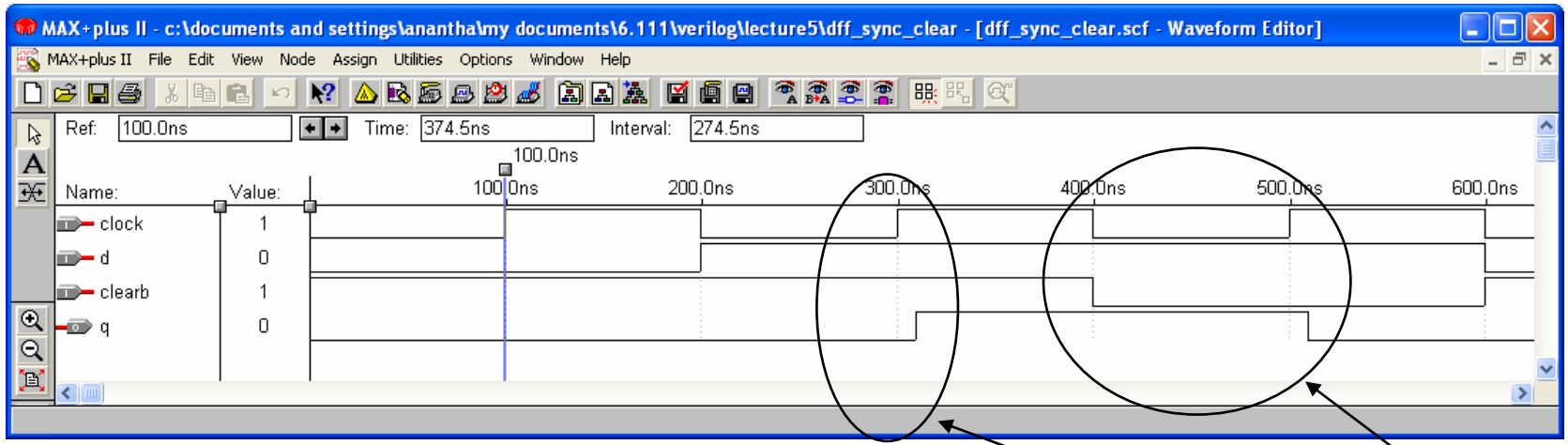
Note: The following is **incorrect** syntax: `always @ (clear or negedge clock)`

If one signal in the sensitivity list uses posedge/negedge, then all signals must.

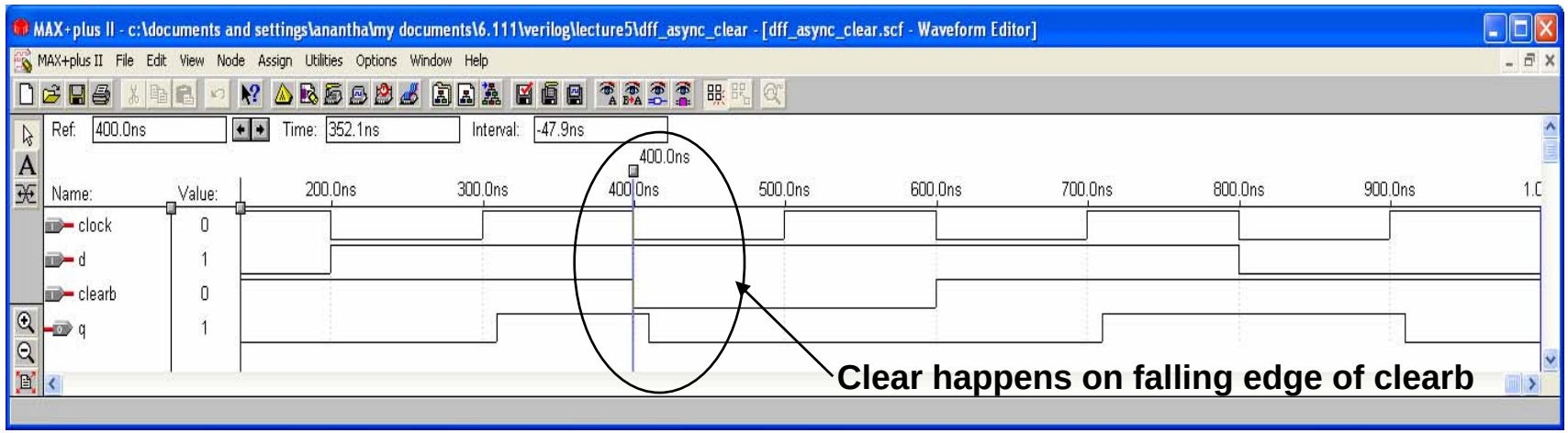
- **Assign any signal or variable from only one always block, Be wary of race conditions: always blocks execute in parallel**

Simulation

- **DFF with Synchronous Clear**



- **DFF with Asynchronous Clear**



Blocking vs. Nonblocking Assignments

- Verilog supports two types of assignments within `always` blocks, with subtly different behaviors.
- *Blocking assignment:* evaluation and assignment are immediate

```
always @ (a or b or c)
begin
  x = a | b;
  y = a ^ b ^ c;
  z = b & ~c;
end
```

1. Evaluate *a* | *b*, assign result to *x*
2. Evaluate *a*^*b*^*c*, assign result to *y*
3. Evaluate *b*&(~*c*), assign result to *z*

- *Nonblocking assignment:* all assignments deferred until all right-hand sides have been evaluated (end of simulation timestep)

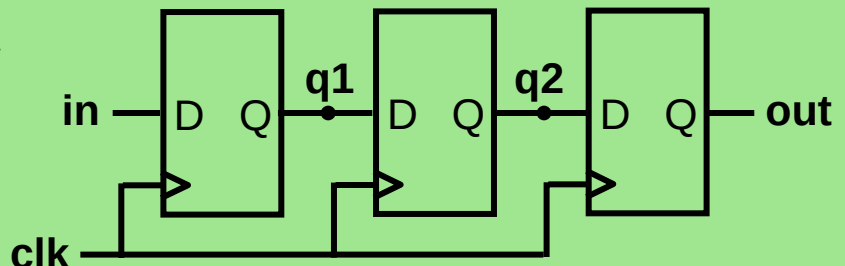
```
always @ (a or b or c)
begin
  x <= a | b;
  y <= a ^ b ^ c;
  z <= b & ~c;
end
```

1. Evaluate *a* | *b* but defer assignment of *x*
2. Evaluate *a*^*b*^*c* but defer assignment of *y*
3. Evaluate *b*&(~*c*) but defer assignment of *z*
4. Assign *x*, *y*, and *z* with their new values

- Sometimes, as above, both produce the same result. Sometimes, not!

Assignment Styles for Sequential Logic

Flip-Flop Based Digital Delay Line



- **Will nonblocking and blocking assignments both produce the desired result?**

```
module nonblocking(in, clk, out);
  input in, clk;
  output out;
  reg q1, q2, out;

  always @ (posedge clk)
  begin
    q1 <= in;
    q2 <= q1;
    out <= q2;
  end

endmodule
```

```
module blocking(in, clk, out);
  input in, clk;
  output out;
  reg q1, q2, out;

  always @ (posedge clk)
  begin
    q1 = in;
    q2 = q1;
    out = q2;
  end

endmodule
```

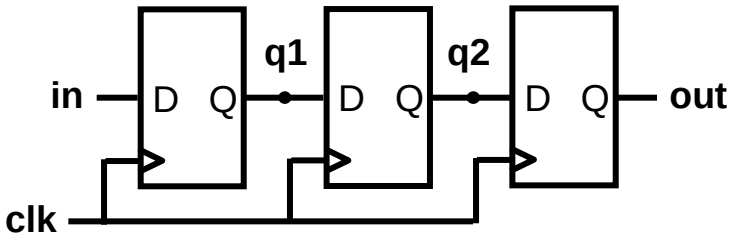
Use Nonblocking for Sequential Logic

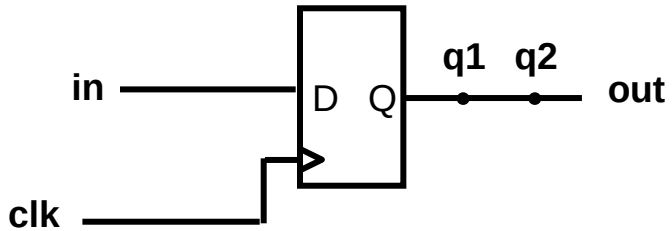
```
always @ (posedge clk)
begin
  q1 <= in;
  q2 <= q1;
  out <= q2;
end
```

"At each rising clock edge, *q1*, *q2*, and *out* simultaneously receive the old values of *in*, *q1*, and *q2*."

```
always @ (posedge clk)
begin
  q1 = in;
  q2 = q1;
  out = q2;
end
```

"At each rising clock edge, *q1* = *in*.
After that, *q2* = *q1* = *in*.
After that, *out* = *q2* = *q1* = *in*.
Therefore *out* = *in*."

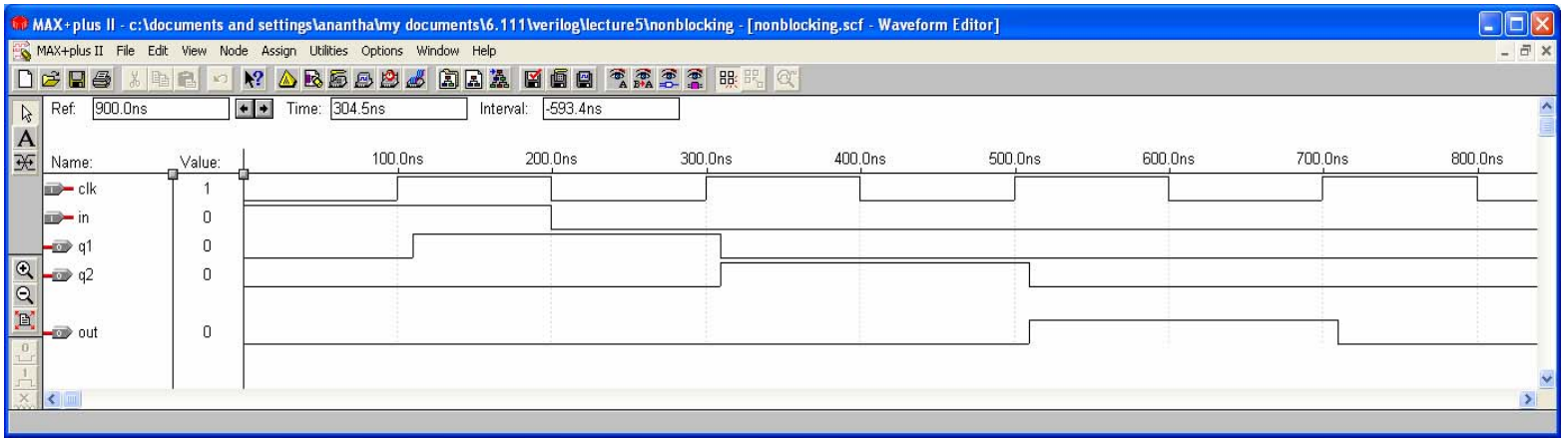




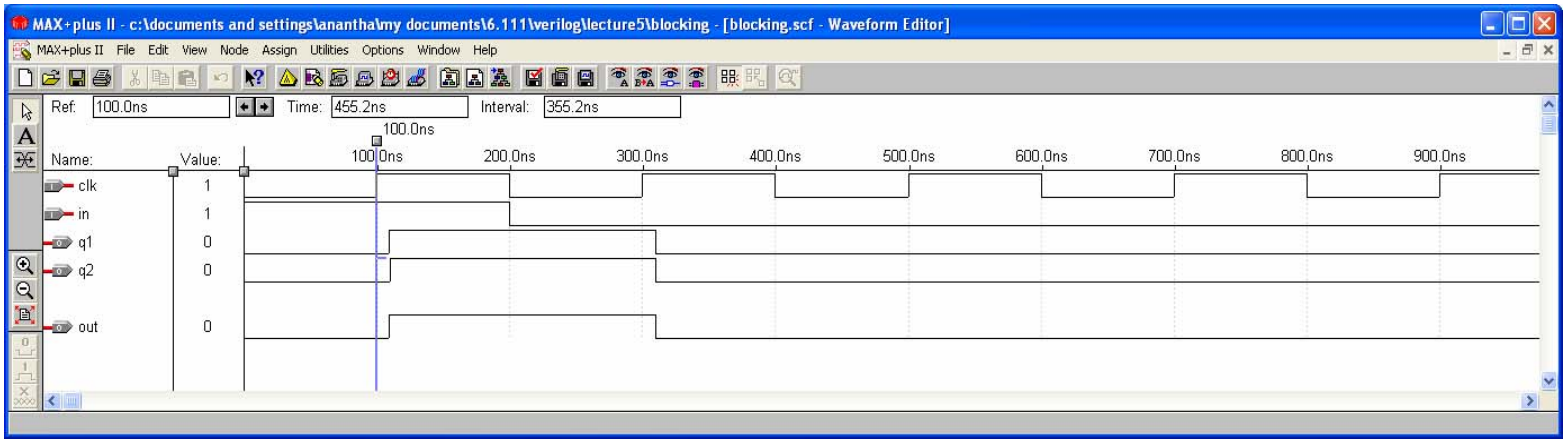
- Blocking assignments do not reflect the intrinsic behavior of multi-stage sequential logic
- Guideline: use nonblocking assignments for sequential `always` blocks

Simulation

- **Non-blocking Simulation**

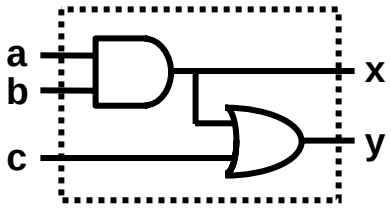


- **Blocking Simulation**



Use Blocking for Combinational Logic

Blocking Behavior	a b c x y
(Given) Initial Condition	1 1 0 1 1
a changes; `always` block triggered	0 1 0 1 1
`x = a & b;`	0 1 0 0 1
`y = x | c;`	0 1 0 0 0




```
module blocking(a,b,c,x,y);
  input a,b,c;
  output x,y;
  reg x,y;

  always @ (a or b or c)
  begin
    x = a & b;
    y = x | c;
  end

endmodule
```

Nonblocking Behavior	a b c x y	*Deferred*
(Given) Initial Condition	1 1 0 1 1	
a changes; `always` block triggered	0 1 0 1 1	
`x <= a & b;`	0 1 0 1 1	x<=0
`y <= x | c;`	0 1 0 1 1	x<=0, y<=1
Assignment completion	0 1 0 0 1	

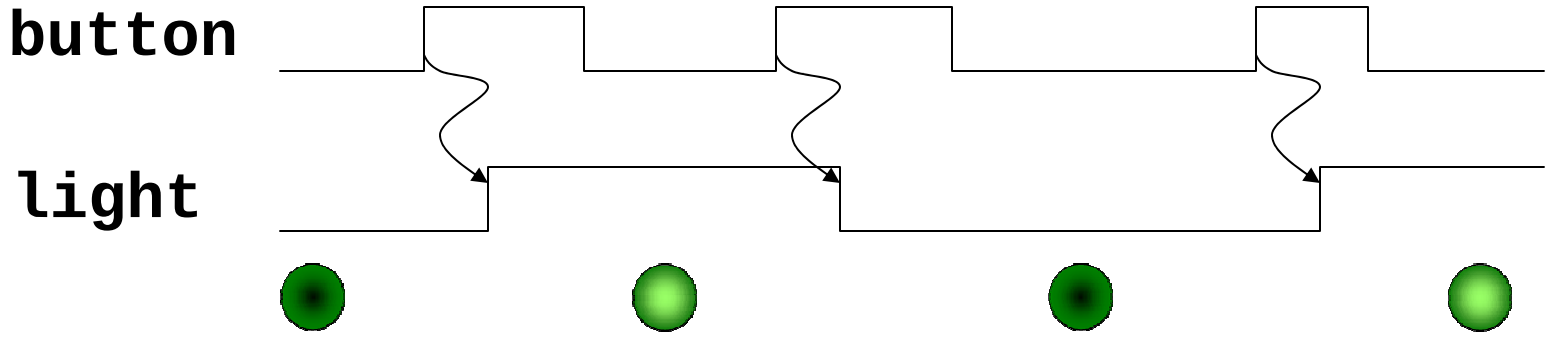
```
module nonblocking(a,b,c,x,y);
  input a,b,c;
  output x,y;
  reg x,y;

  always @ (a or b or c)
  begin
    x <= a & b;
    y <= x | c;
  end

endmodule
```

- Nonblocking and blocking assignments will synthesize correctly. Will both styles simulate correctly?
- Nonblocking assignments do not reflect the intrinsic behavior of multi-stage combinational logic
- While nonblocking assignments can be hacked to simulate correctly (expand the sensitivity list), it's not elegant
- Guideline: use blocking assignments for combinational `always` blocks

Implementation for on/off button




```
module onoff(button,light);
  input button;
  output light;
  reg light;
  always @ (posedge button)
  begin
    light <= ~light;
  end
endmodule
```