



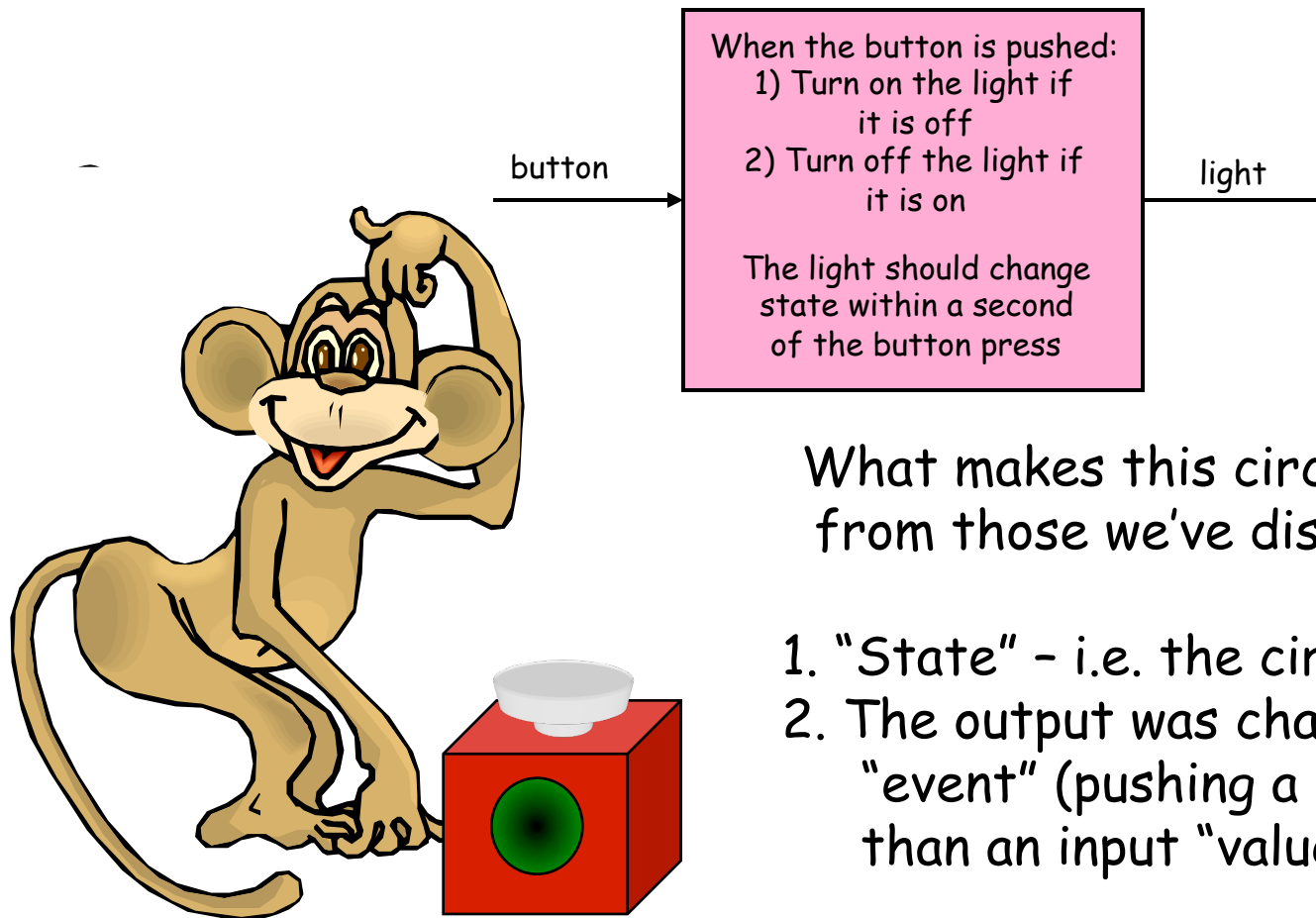
# Sequential Logic

- Digital state: the D-Register
- Timing constraints for D-Registers
- Specifying registers in Verilog
- Blocking and nonblocking assignments
- Verilog execution semantics: concurrency & nondeterminism
- Examples

**Reminder: Lab #2 due Thursday!**

# Something We Can't Build (Yet)

What if you were given the following design specification:

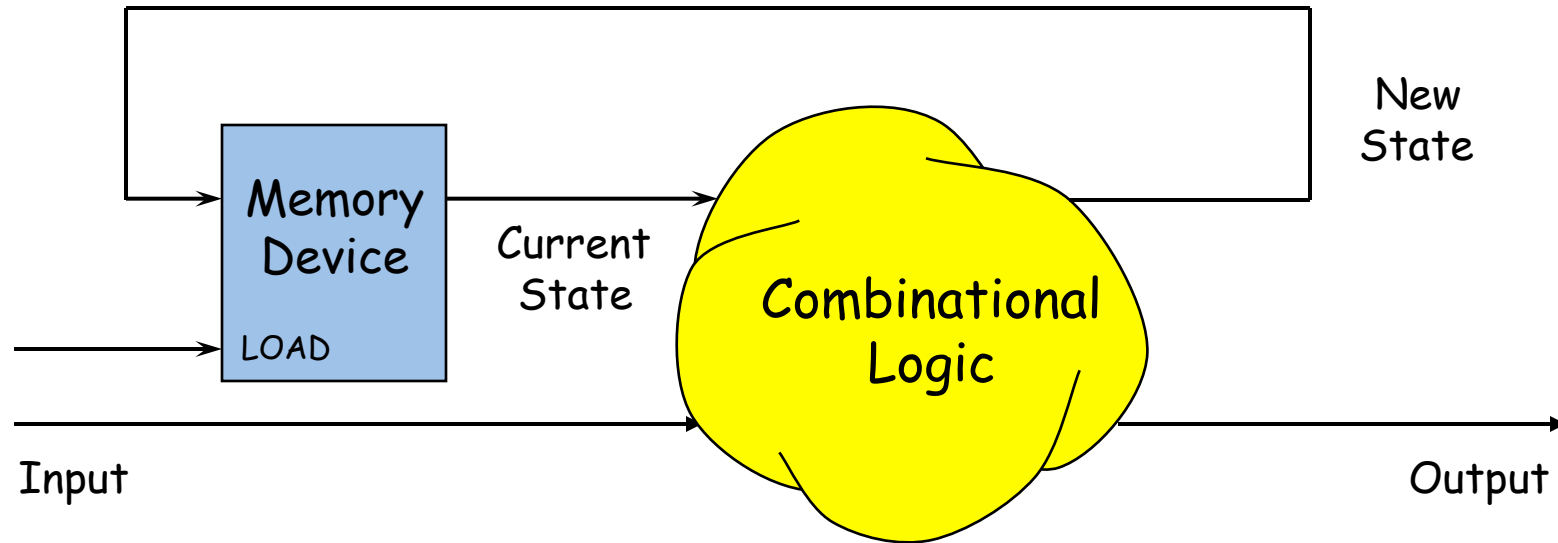


What makes this circuit so different from those we've discussed before?

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*



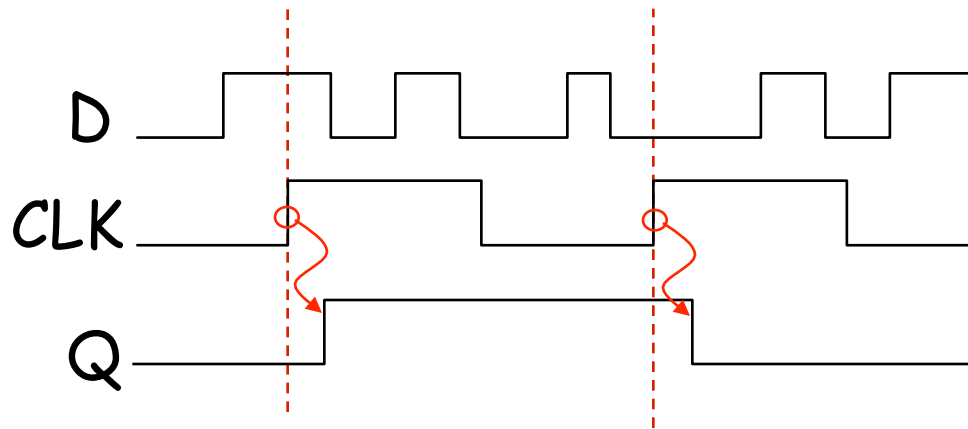
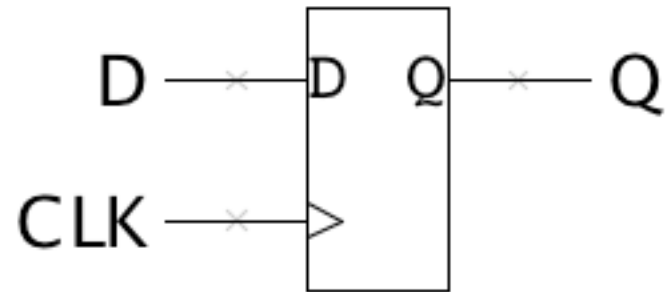
Plan: Build a Sequential Circuit with stored digital STATE -

- 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

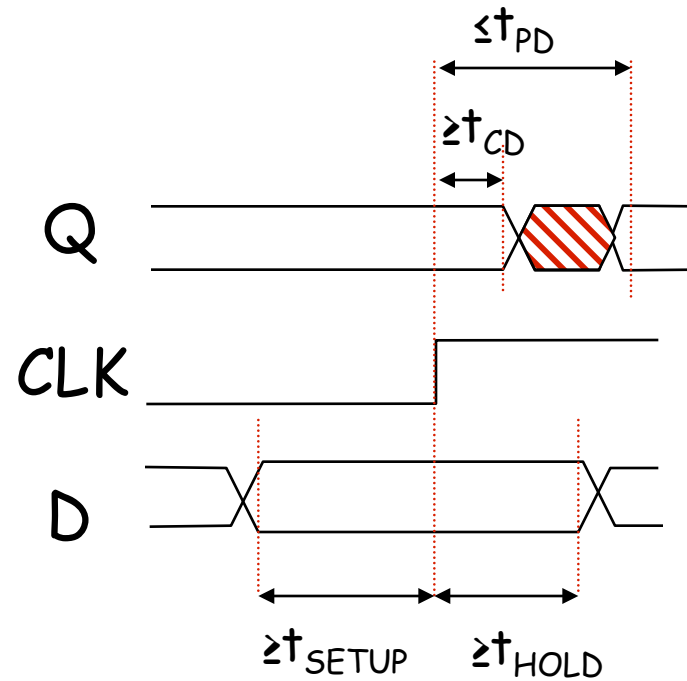
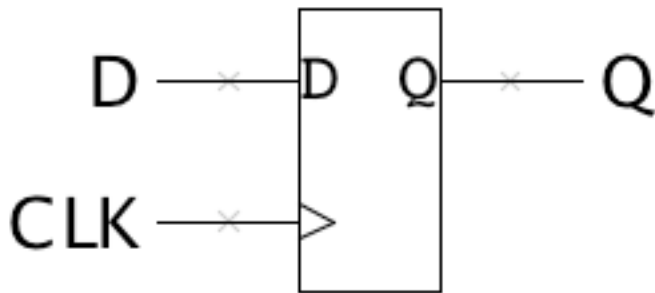
*When Output depends on input and current state, circuit is called a Mealy machine. If Output depends only on the current state, circuit is called a Moore machine.*

# Our next building block: the D register

The edge-triggered D register: *on the rising edge of CLK*, the value of D is saved in the register and then shortly afterwards appears on Q.



# D-Register Timing - I



$t_{PD}$ : maximum propagation delay,  $CLK \rightarrow Q$

$t_{CD}$ : minimum contamination delay,  $CLK \rightarrow Q$

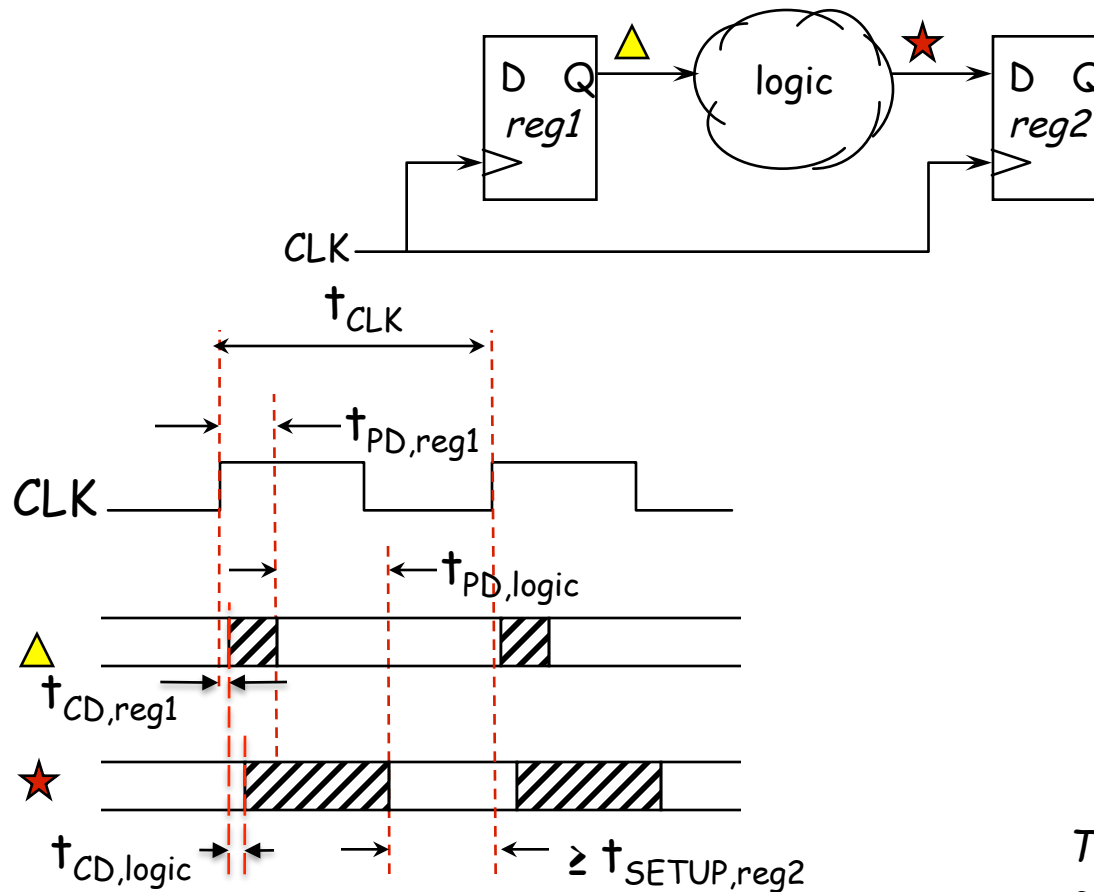
$t_{SETUP}$ : setup time

*How long D must be stable before the rising edge of CLK*

$t_{HOLD}$ : hold time

*How long D must be stable after the rising edge of CLK*

# D-Register Timing - II



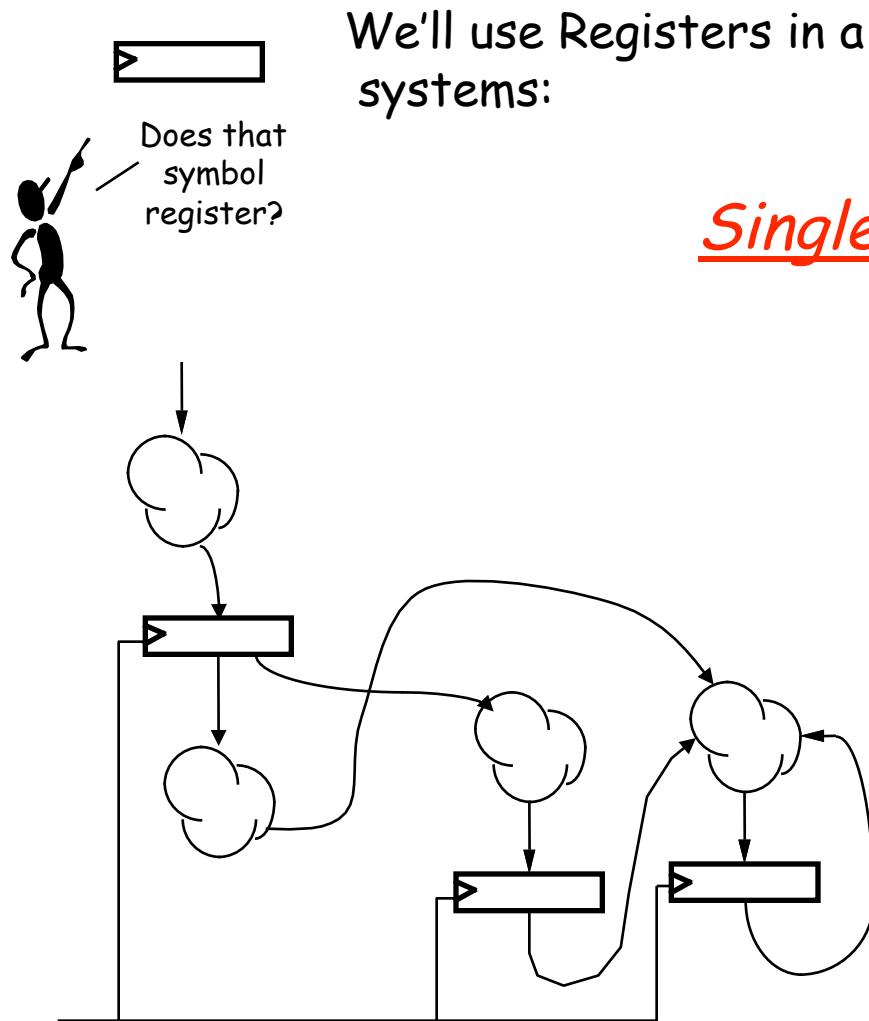
The good news: you can choose  $t_{CLK}$  so that this constraint is satisfied!

$$t_{PD,reg1} + t_{PD,logic} + t_{SETUP,reg2} \leq t_{CLK}$$

The bad news: you have to change your design if this constraint isn't met.

$$t_{CD,reg1} + t_{CD,logic} \geq t_{HOLD,reg2}$$

# Single-clock Synchronous Circuits

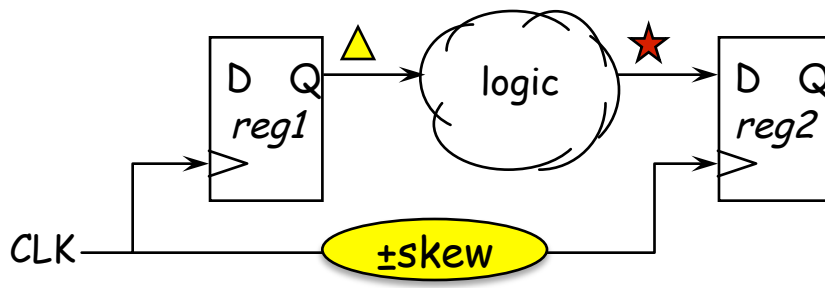


We'll use Registers in a highly constrained way to build digital systems:

## Single-clock Synchronous Discipline

- No combinational cycles
- Single clock signal shared among all clocked devices (one *clock domain*)
- Only care about value of combinational circuits just before rising edge of clock
- Clock period greater than every combinational delay
- Change saved state after noise-inducing logic transitions have stopped!

# D-Register Timing With Skew



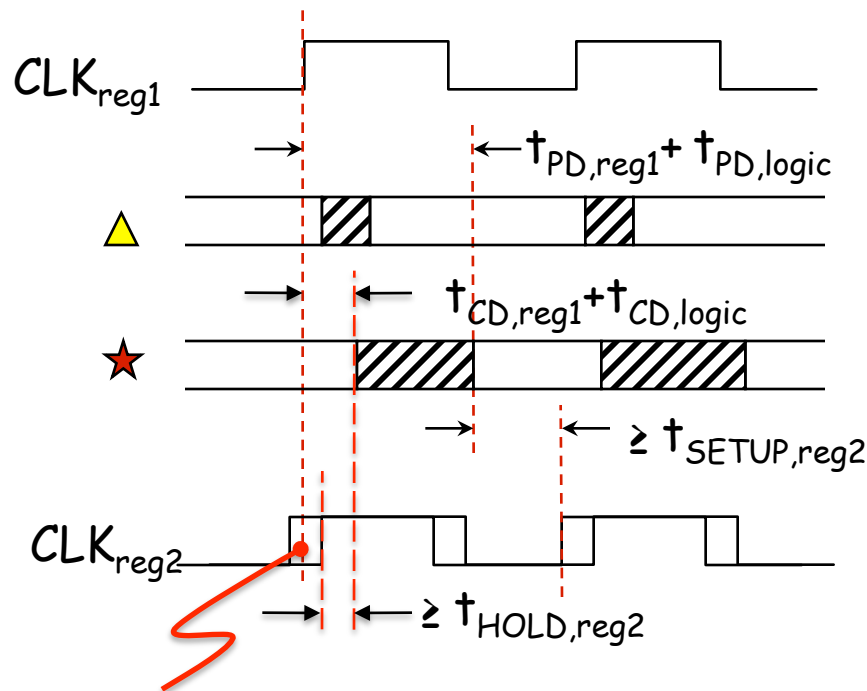
In the real world the clock signal arrives at different registers at different times. The difference in arrival times (pos or neg) is called the *clock skew*  $t_{skew}$ .

We can update our two timing constraints to reflect the worst-case skew:

$$t_{PD,reg1} + t_{PD,logic} + t_{SETUP,reg2} \leq t_{CLK} - t_{skew}$$

$$t_{CD,reg1} + t_{CD,logic} \geq t_{HOLD,reg2} + t_{skew}$$

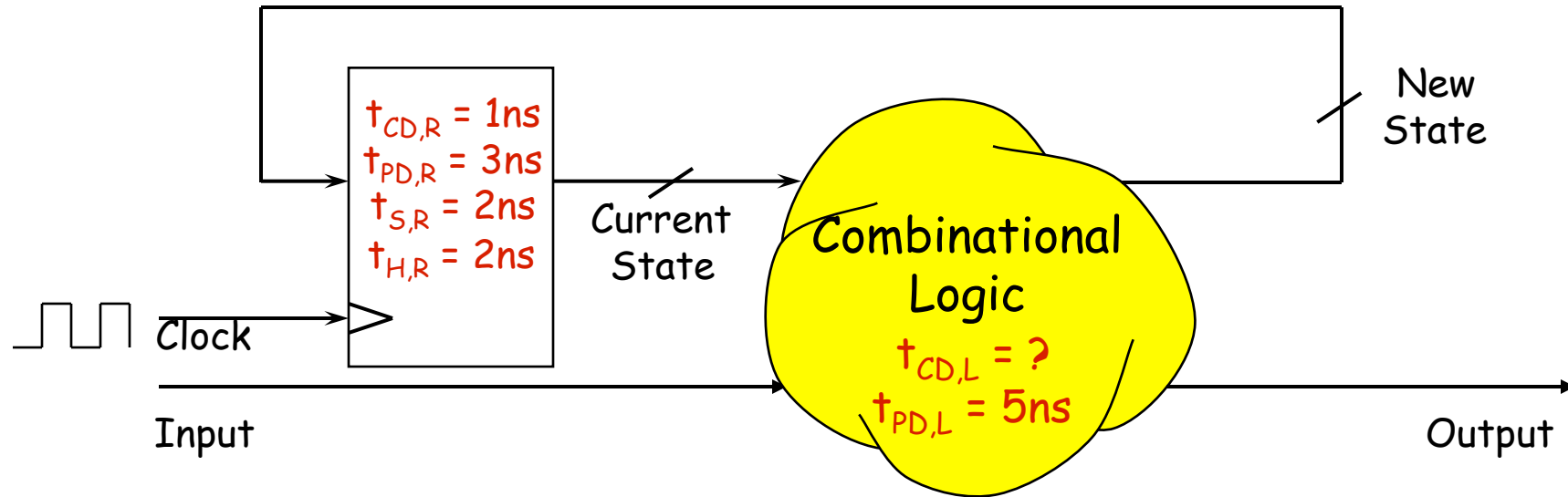
Thus clock skew increases the minimum cycle time of our design and makes it harder to meet register hold times.



*CLK<sub>reg2</sub> rising edge might fall anywhere in this region.*



# Sequential Circuit Timing



## Questions:

- Constraints on  $t_{CD}$  for the logic?  $> 1\text{ ns}$
- Minimum clock period?  $> 10\text{ ns } (t_{PD,R} + t_{PD,L} + t_{SETUP,R})$
- Setup, Hold times for Inputs?
  - $t_{SETUP,Input} = t_{PD,L} + t_{SETUP,R}$
  - $t_{HOLD,Input} = t_{HOLD,R} - t_{CD,L}$

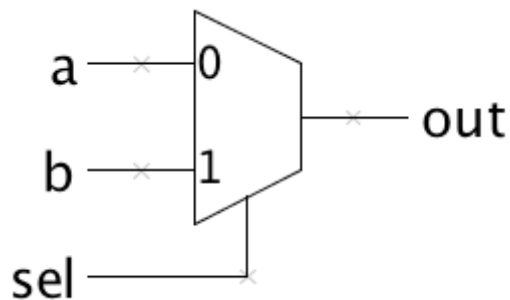
This is a simple *Finite State Machine* ... more on next time!

# The Sequential **always** Block

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

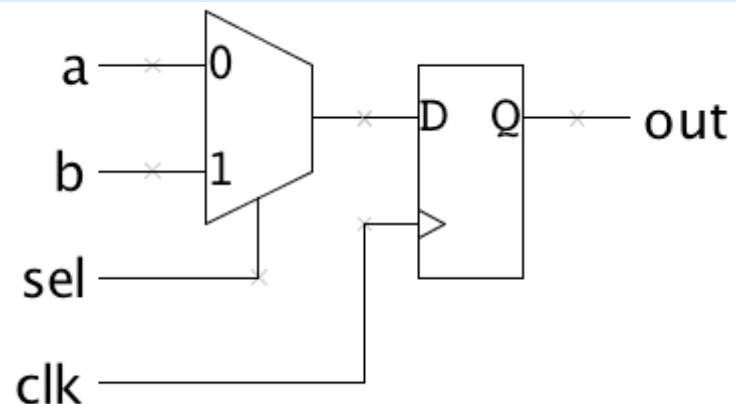
## Combinational

```
module comb(input a, b, sel,
            output reg out);
  always @(*) begin
    if (sel) out = a;
    else out = b;
  end
endmodule
```



## Sequential

```
module seq(input a, b, sel, clk,
            output 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-Register with **synchronous** clear*

```
module dff_sync_clear(  
    input d, clearb, clock,  
    output 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-Register with **asynchronous** clear*

```
module dff_sync_clear(  
    input d, clearb, clock,  
    output 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 with same trigger execute concurrently...

# 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 @(*) begin
  x = a | b;      // 1. evaluate a|b, assign result to x
  y = a ^ b ^ c;  // 2. evaluate a^b^c, assign result to y
  z = b & ~c;     // 3. evaluate b&(~c), assign result to z
end
```

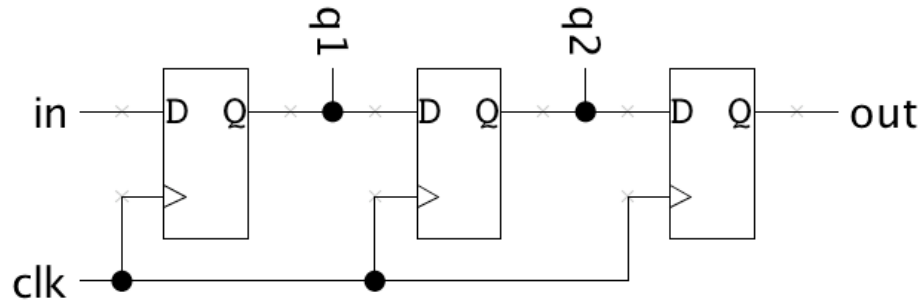
*Nonblocking assignment* (`<=`): all assignments deferred to end of simulation time step after all right-hand sides have been evaluated (even those in other active `always` blocks)

```
always @(*) begin
  x <= a | b;      // 1. evaluate a|b, but defer assignment to x
  y <= a ^ b ^ c;  // 2. evaluate a^b^c, but defer assignment to y
  z <= b & ~c;     // 3. evaluate b&(~c), but defer assignment to z
  // 4. end of time step: assign new values to x, y and z
end
```

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

# Assignment Styles for Sequential Logic

*What we want:  
Register Based  
Digital Delay Line*



Will nonblocking and blocking assignments both produce the desired result? ("old" means value before clock edge, "new" means the value after most recent assignment)

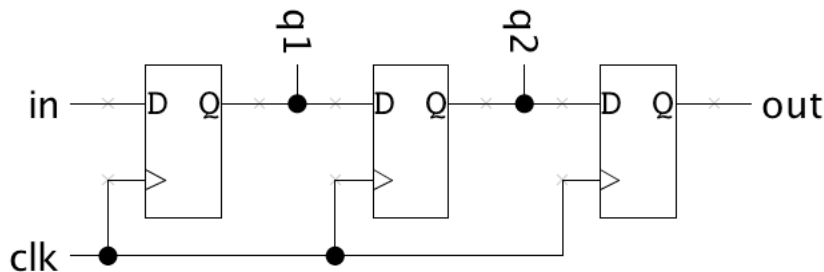
```
module nonblocking(  
    input in, clk,  
    output reg out  
);  
    reg q1, q2;  
    always @(posedge clk) begin  
        q1 <= in;  
        q2 <= q1;    // uses old q1  
        out <= q2;    // uses old q2  
    end  
endmodule
```

```
module blocking(  
    input in, clk,  
    output reg out  
);  
    reg q1, q2;  
    always @(posedge clk) begin  
        q1 = in;  
        q2 = q1;    // uses new q1  
        out = q2;    // uses new q2  
    end  
endmodule
```

# Use Nonblocking for Sequential Logic

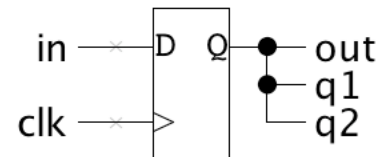
```
always @(posedge clk) begin
    q1 <= in;
    q2 <= q1;    // uses old q1
    out <= q2;    // uses old q2
end
```

“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;    // uses new q1
    out = q2;    // uses new q2
end
```

“At each rising clock edge,  $q1 = in$ .  
After that,  $q2 = q1$ .  
After that,  $out = q2$ .  
Therefore  $out = in$ .”



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

# Verilog Events

IEEE 1364-2001 Verilog Standard: **Section 5.3 The stratified event queue**

The Verilog event queue is logically segmented into five different regions. Events are added to any of the five regions but are only removed from the *active* region.

1. Events that occur at the current simulation time and can be processed in any order. These are the *active* events.
2. Events that occur at the current simulation time, but that shall be processed after all the active events are processed. These are the *inactive* events.
3. Events that have been evaluated during some previous simulation time, but that shall be assigned at this simulation time after all the active and inactive events are processed. These are the *nonblocking assign update* events.
4. Events that shall be processed after all the active, *inactive*, and nonblocking assign update events are processed. These are the *monitor* events.
5. Events that occur at some future simulation time. These are the *future* events. Future events are divided into *future inactive* events, and *future nonblocking assignment update* events.

# Verilog Execution Semantics

```
while (there are events) {  
  if (no active events) {  
    if (there are inactive events) {  
      activate all inactive events;  
    } else if (there are nonblocking assign update events) {  
      activate all nonblocking assign update events;  
    } else if (there are monitor events) {  
      activate all monitor events;  
    } else {  
      advance T to the next event time;  
      activate all inactive events for time T;  
    }  
  }  
  E = any active event;  
  if (E is an update event) {  
    update the modified object;  
    add evaluation events for sensitive processes to event queue;  
  } else { /* E is an evaluation event */  
    evaluate the process;  
    add update events to the event queue;  
  }  
}
```

From Section 5.4 of the  
IEEE 1364-2001 Verilog  
standard.

*Active events can be processed  
in any order, but update events  
from same always block will  
happen in order.*

*A process evaluation might be  
suspended in favor of other  
processes.*

For more info see: [http://www.sunburst-design.com/papers/CummingsSNUG2002Boston\\_NBAwithDelays.pdf](http://www.sunburst-design.com/papers/CummingsSNUG2002Boston_NBAwithDelays.pdf)



# Coding Guidelines

The following helpful guidelines are from the Cummings paper referenced on the previous slide. If followed, they ensure your simulation results will match what the synthesized hardware will do:

1. When modeling sequential logic, use nonblocking assignments.
2. When modeling latches, use nonblocking assignments.
3. When modeling combinational logic with an always block, use blocking assignments.
4. When modeling both sequential and combinational logic within the same always block, use nonblocking assignments.
5. Do not mix blocking and nonblocking assignments in the same always block.
6. Do not make assignments to the same variable from more than one always block.
7. Use \$strobe to display values that have been assigned using nonblocking assignments.
8. Do not make assignments using #0 delays.

# = vs. <= inside always

```
module main;  
  reg a,b,clk;
```

```
  initial begin  
    clk = 0; a = 0; b = 1;  
    #10 clk = 1;  
    #10 $display("a=%d b=%d\n",a,b);  
    $finish;  
  end  
endmodule
```

A

```
always @(posedge clk) begin  
  a = b;    // blocking assignment  
  b = a;    // execute sequentially  
end
```

B

```
always @(posedge clk) begin  
  a <= b;   // non-blocking assignment  
  b <= a;   // eval all RHSs first  
end
```

C

```
always @(posedge clk) a = b;  
always @(posedge clk) b = a;
```

D

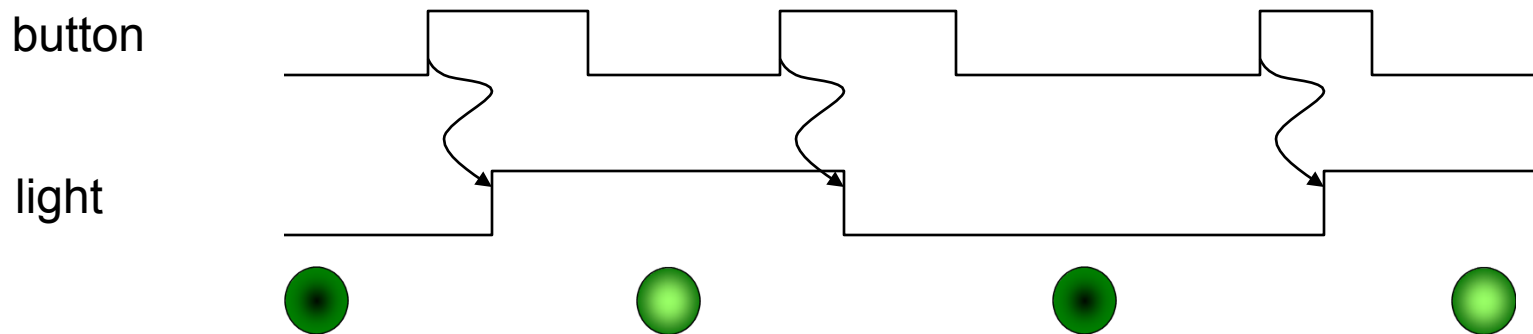
```
always @(posedge clk) a <= b;  
always @(posedge clk) b <= a;
```

E

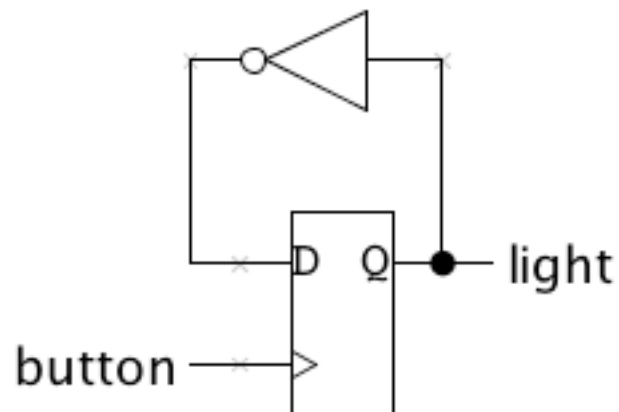
```
always @(posedge clk) begin  
  a <= b;  
  b = a;    // urk! Be consistent!  
end
```

**Rule: always change state using <= (e.g., inside always @(posedge clk)...)**

# Implementation for on/off button



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



# Synchronous on/off button

When designing a system that accepts many inputs it would be hard to have input changes serve as the system clock (which input would we use?). So we'll use a single clock of some fixed frequency and have the inputs control what state changes happen on rising clock edges.

For most of our lab designs we'll use a 27MHz system clock (37ns clock period).

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

# Resetting to a known state

Usually one can't rely on registers powering-on to a particular initial state\*. So most designs have a RESET signal that when asserted initializes all the state to known, mutually consistent initial values.

```
module onoff_sync(input clk, reset, button,
                  output reg light);
    always @ (posedge clk) begin
        if (reset) light <= 0;
        else if (button) light <= ~light;
    end
endmodule
```

\* Actually, our FPGAs will reset all registers to 0 when the device is programmed. But it's nice to be able to press a reset button to return to a known state rather than starting from scratch by reprogramming the device.

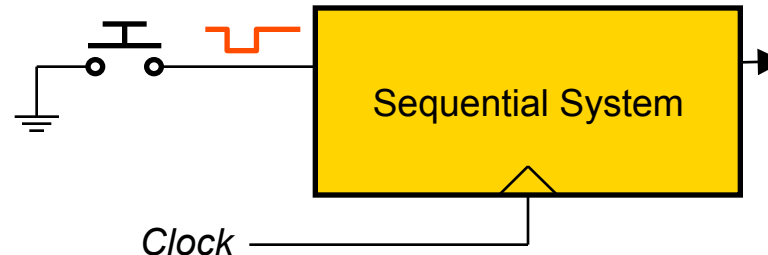
# Clocks are fast, we're slow!

The circuit on the last slide toggles the light on every rising clock edge for which button is 1. But clocks are fast (27MHz!) and our fingers are slow, so how do we press the button for just one clock edge? Answer: we can't, but we can add some state that remembers what button was last clock cycle and then detect the clock cycles when button changes from 0 to 1.

```
module onoff_sync(input clk, reset, button,
                  output reg light);
    reg old_button; // state of button last clk
    always @ (posedge clk) begin
        if (reset)
            begin light <= 0; old_button <= 0; end
        else if (old_button==0 && button==1)
            // button changed from 0 to 1
            light <= ~light;
            old_button <= button;
        end
    endmodule
```

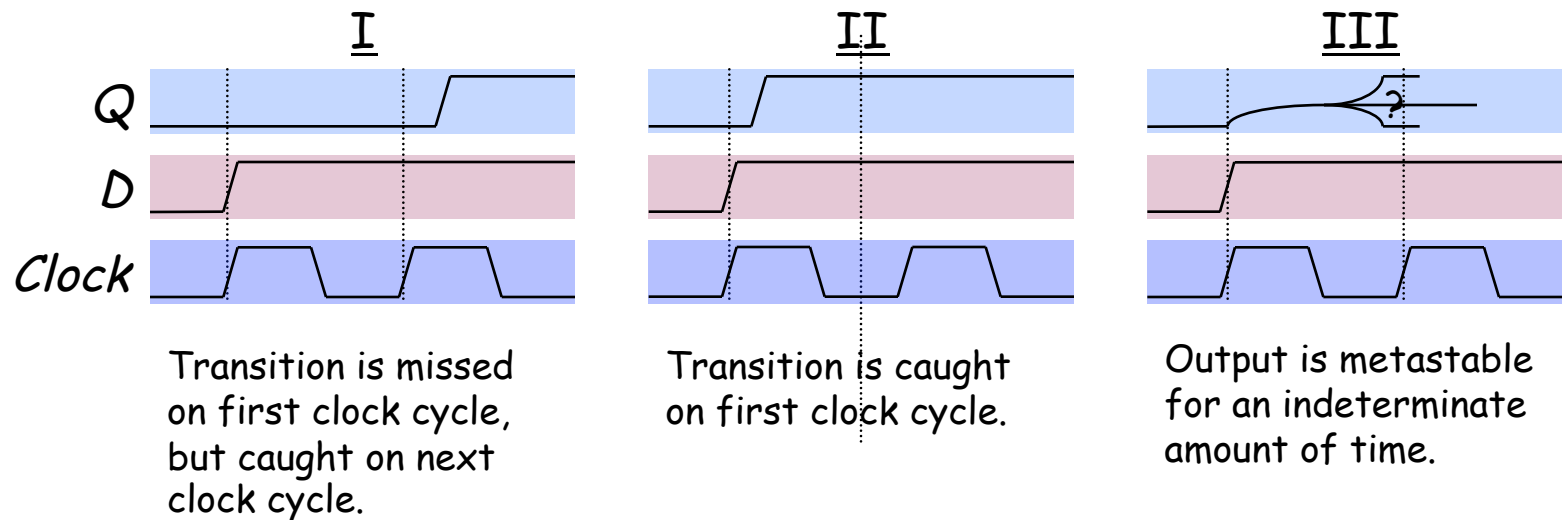
# Asynchronous Inputs in Sequential Systems

What about external signals?



*Can't guarantee setup and hold times will be met!*

When an asynchronous signal causes a setup/hold violation...

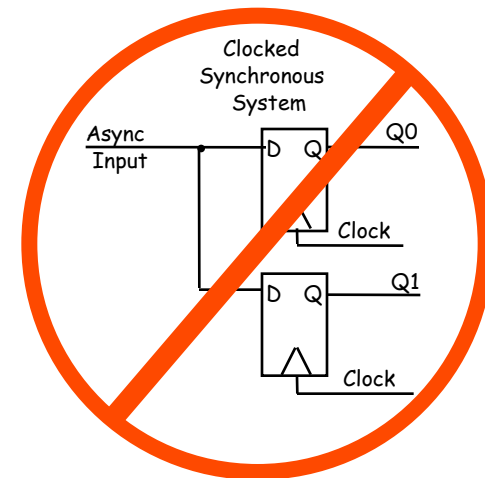
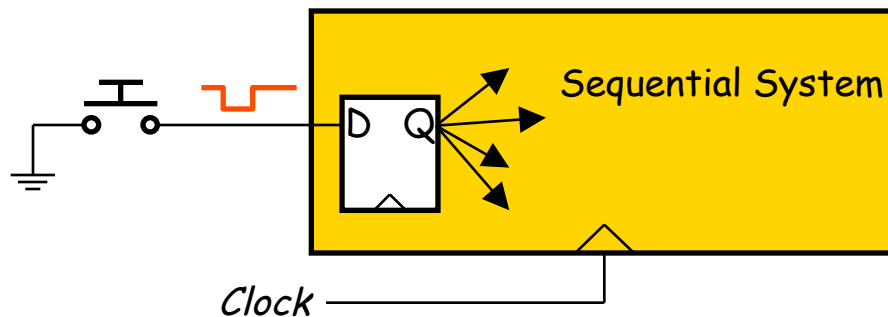


**Q: Which cases are problematic?**

# Asynchronous Inputs in Sequential Systems

*All of them can be*, if more than one happens simultaneously within the same circuit.

*Guideline: ensure that external signals directly feed exactly one flip-flop*

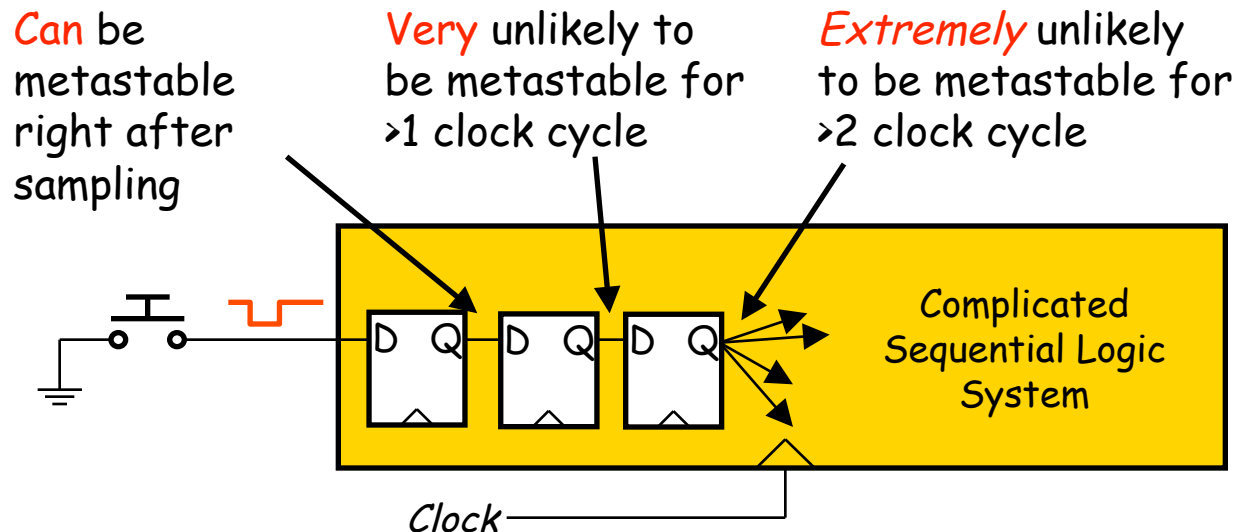


This prevents the possibility of I and II occurring in different places in the circuit, but what about metastability?



# Handling Metastability

- Preventing metastability turns out to be an impossible problem
- High gain of digital devices makes it likely that metastable conditions will resolve themselves quickly
- Solution to metastability: allow time for signals to stabilize

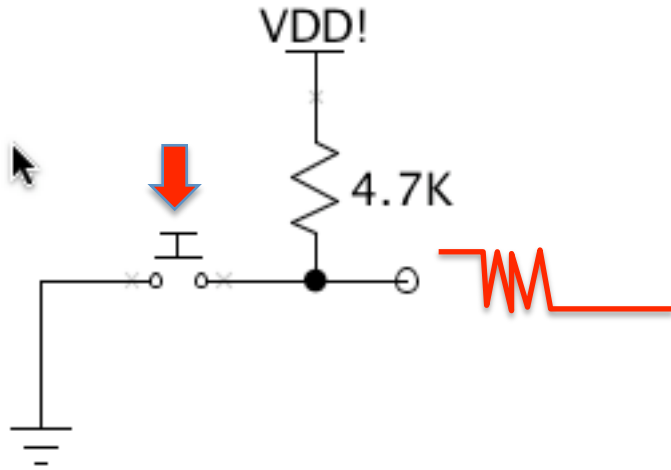


*How many registers are necessary?*

- Depends on many design parameters (clock speed, device speeds, ...)
- In 6.111, a pair of synchronization registers is sufficient

# One last little problem...

Mechanical buttons exhibit contact "bounce" when they change position, leading to multiple output transitions before finally stabilizing in the new position:



We need a debouncing circuit!

```
// Switch Debounce Module
// use your system clock for the clock input
// to produce a synchronous, debounced output
// DELAY = .01 sec with a 27Mhz clock
module debounce #(parameter DELAY=270000)
    (input reset, clock, noisy,
     output reg clean);

    reg [18:0] count;
    reg new;

    always @(posedge clock)
        if (reset) // return to known state
            begin
                count <= 0;
                new <= noisy;
                clean <= noisy;
            end
        else if (noisy != new) // input changed
            begin
                new <= noisy;
                count <= 0;
            end
        else if (count == DELAY) // stable!
            clean <= new;
        else // waiting...
            count <= count+1;

endmodule
```

# On/off button: final answer

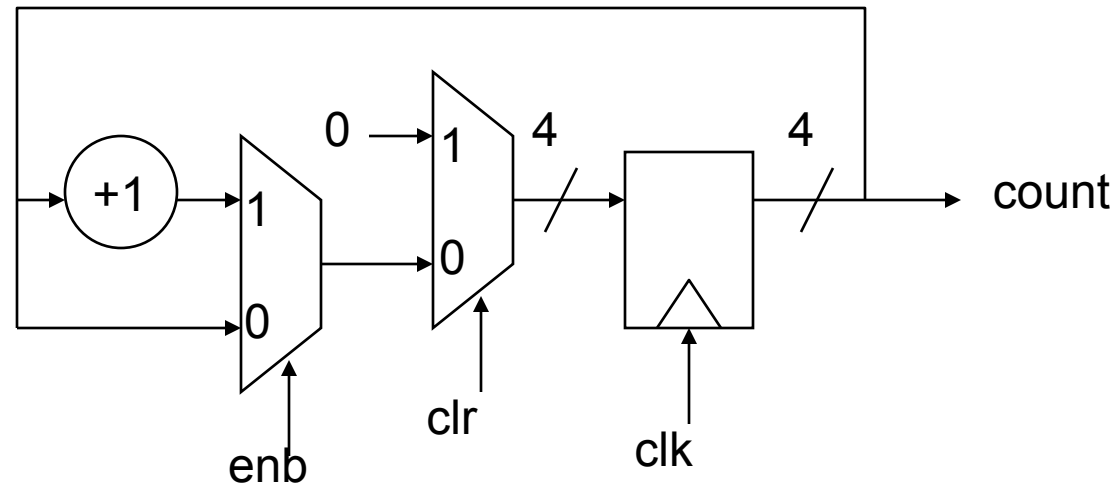
```
module onoff_sync(input clk, reset, button_in,
                  output reg light);
    // synchronizer
    reg button, btemp;
    always @(posedge clk)
        {button, btemp} <= {btemp, button_in};

    // debounce push button
    wire bpressed;
    debounce db1(.clock(clk), .reset(reset),
                .noisy(button), .clean(bpressed));

    reg old_bpressed; // state last clk cycle
    always @ (posedge clk) begin
        if (reset)
            begin light <= 0; old_bpressed <= 0; end
        else if (old_bpressed==0 && bpressed==1)
            // button changed from 0 to 1
            light <= ~light;
            old_bpressed <= bpressed;
        end
    endmodule
```

# Example: A Simple Counter

*Isn't this a lot like  
Exercise 1 in Lab 2?*



```
// 4-bit counter with enable and synchronous clear
module counter(input clk, enb, clr,
               output reg [3:0] count);
    always @(posedge clk) begin
        count <= clr ? 4'b0 : (enb ? count+1 : count);
    end
endmodule
```