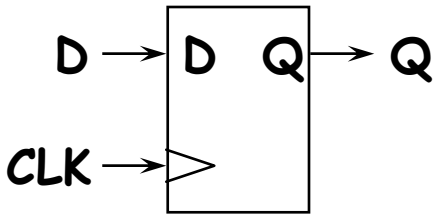
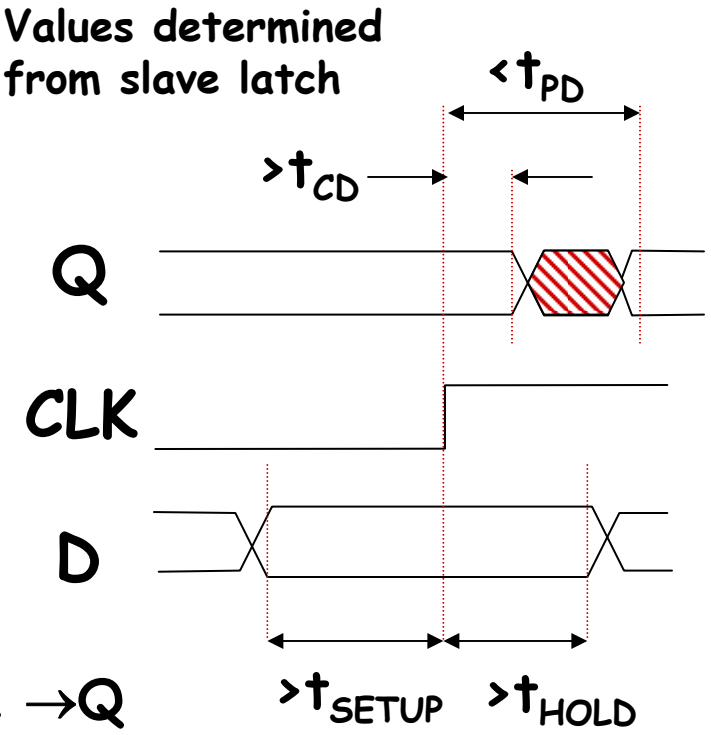
D-Register Timing - I





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

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

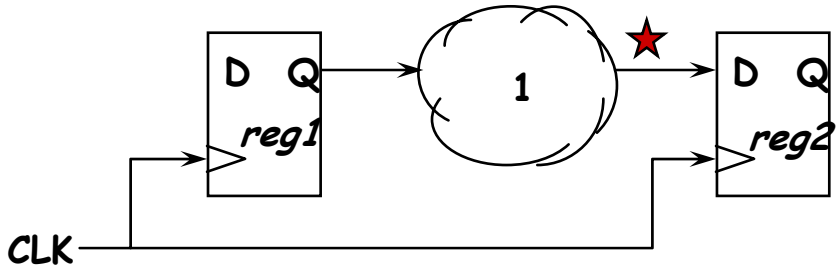
t_{SETUP}: setup time

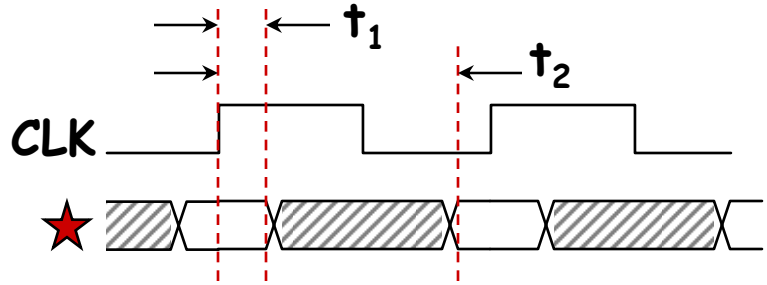
guarantee that D has propagated through feedback path before master closes

t_{HOLD}: hold time

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

D-Register Timing - II





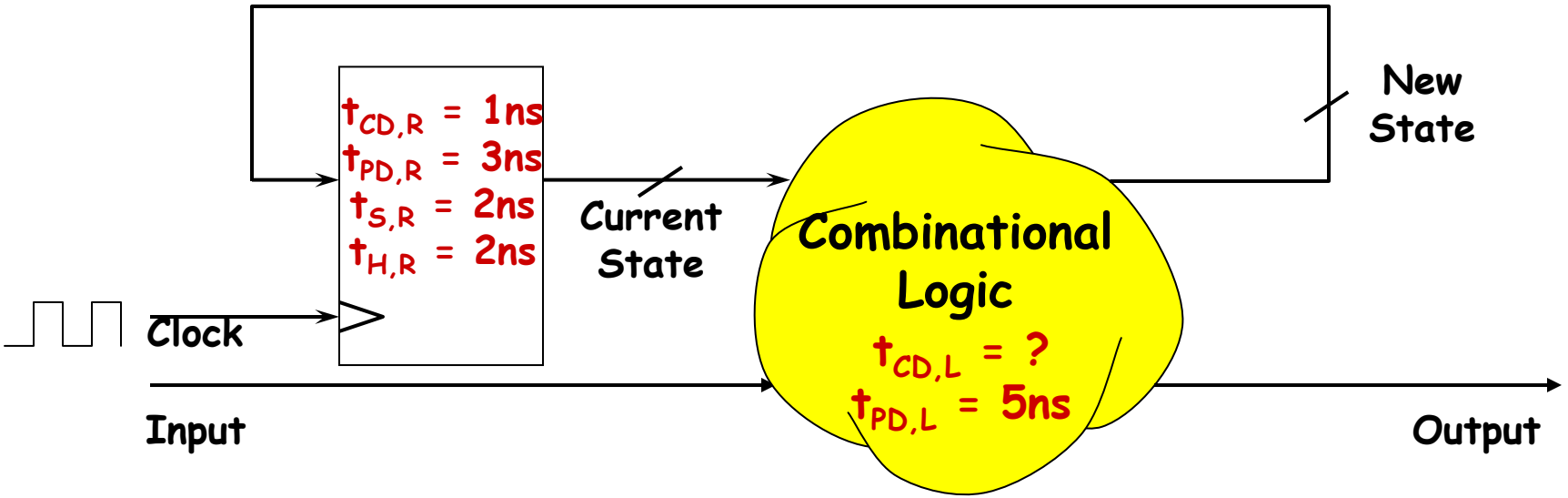
$t_1 = t_{CD,reg1} + t_{CD,1} > t_{HOLD,reg2}$

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

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:

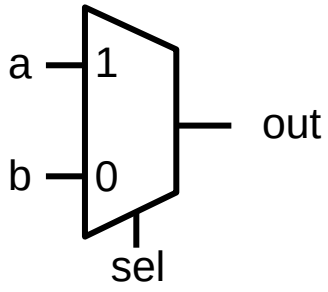
- Constraints on T_{CD} for the logic? > 1 ns
- Minimum clock period? > 10 ns ($T_{PD,R}+T_{PD,L}+T_{S,R}$)
- Setup, Hold times for Inputs? $T_S = T_{PD,L} + T_{S,R}$, $T_H = T_{H,R} - T_{CD,L}$

The Sequential `always` Block

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

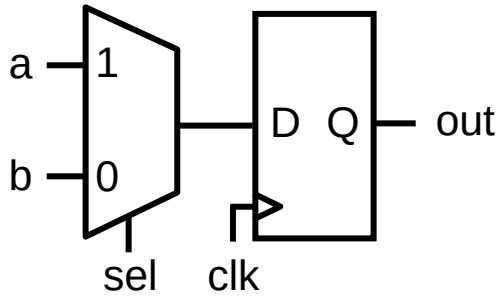
Combinational

```
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

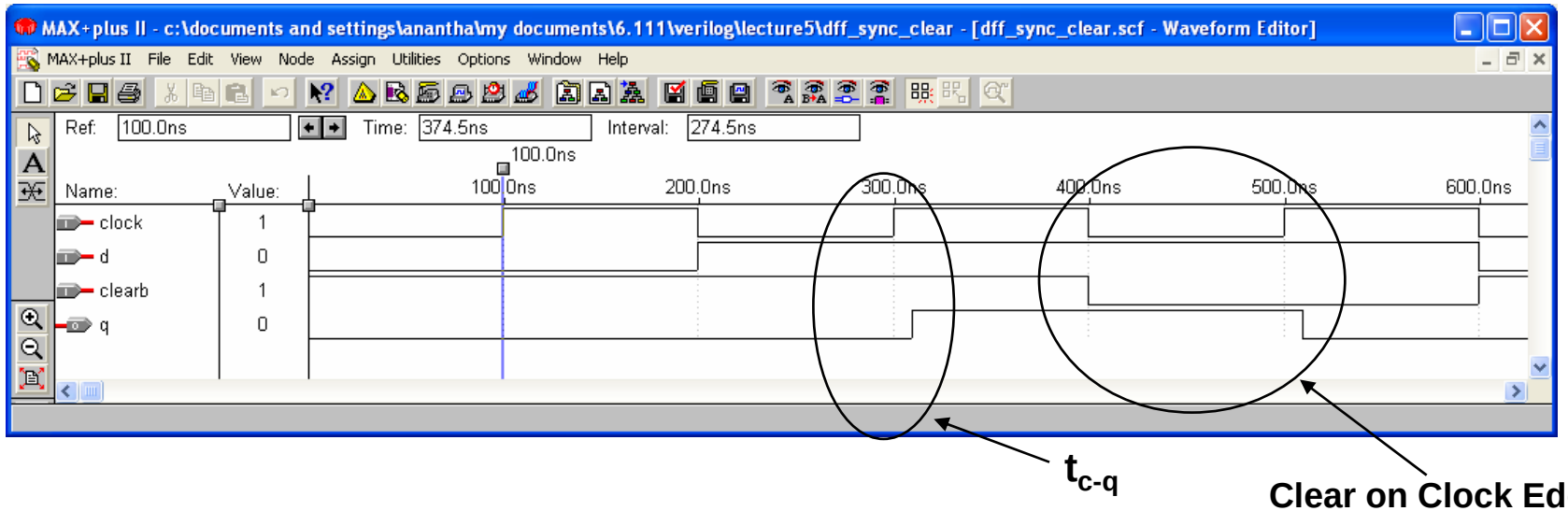
Note: The following is **incorrect** syntax: `always @ (clear or negedge clock)`

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

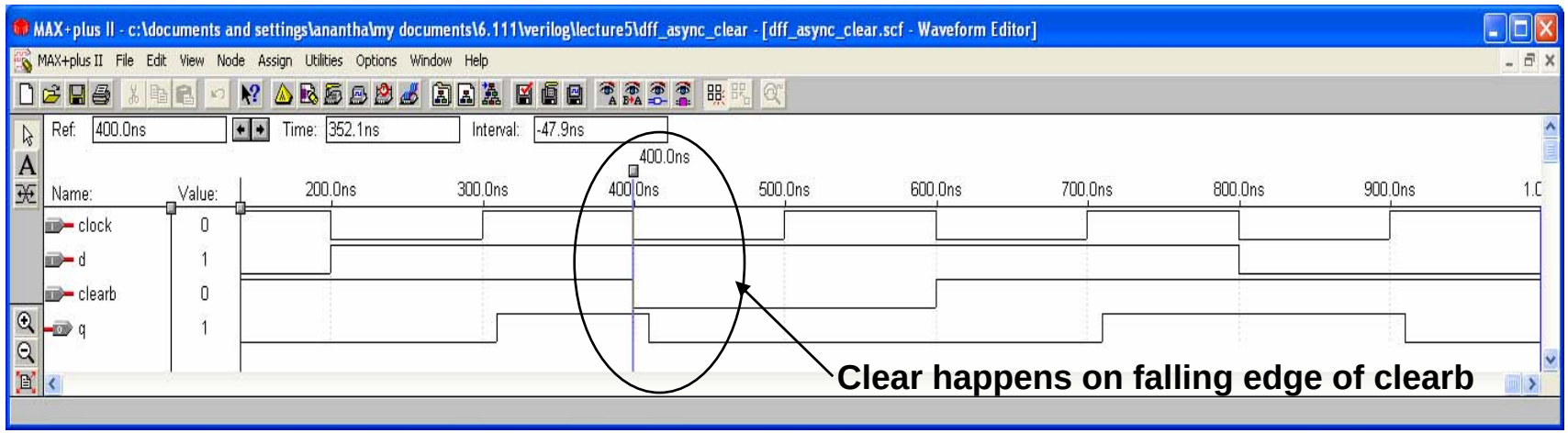
- **Assign any signal or variable from only one always block, Be wary of race conditions: always blocks execute in parallel**

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)

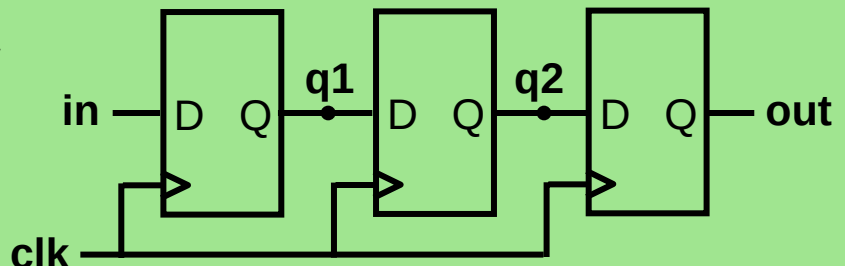
```
always @ (a or b or c)
begin
  x <= a | b;
  y <= a ^ b ^ c;
  z <= b & ~c;
end
```

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
```

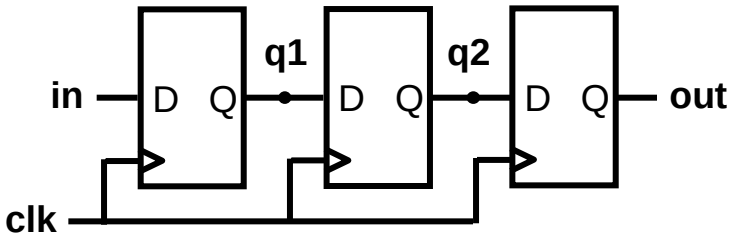
Use Nonblocking for Sequential Logic

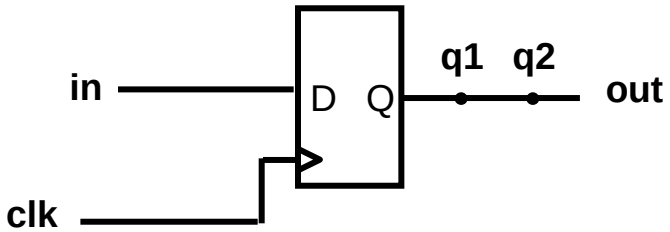
```
always @ (posedge clk)
begin
  q1 <= in;
  q2 <= q1;
  out <= q2;
end
```

"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*."

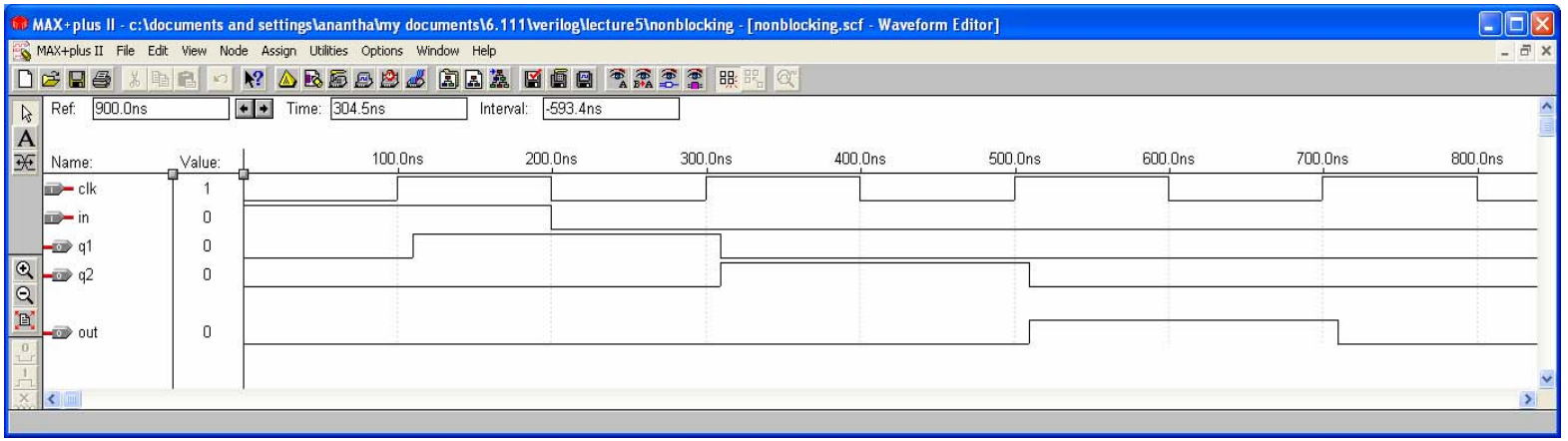




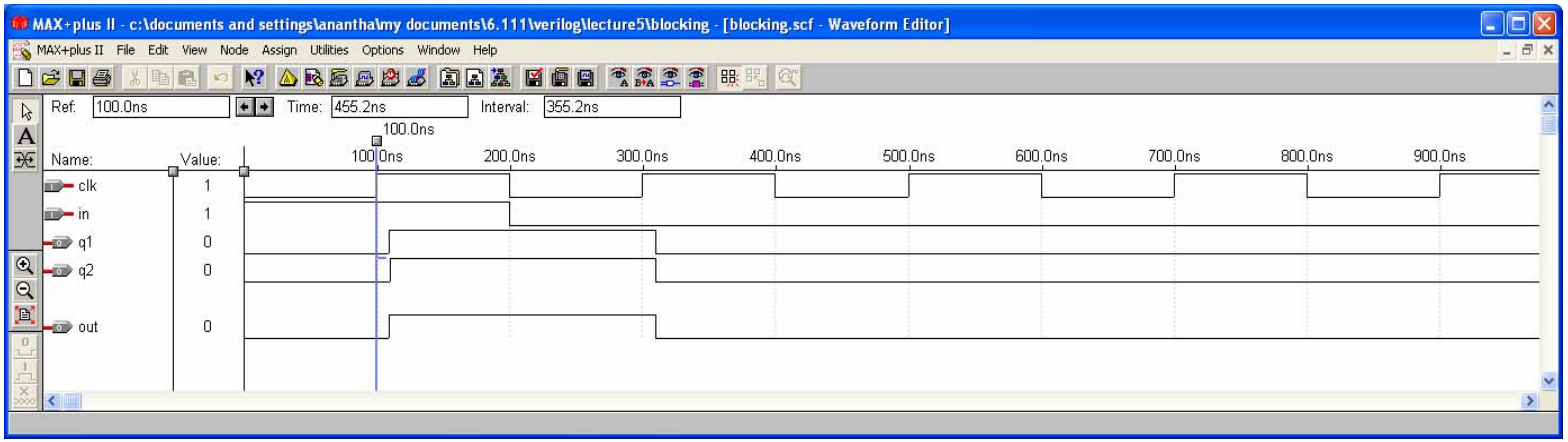
- Blocking assignments do not reflect the intrinsic behavior of multi-stage sequential logic
- Guideline: use nonblocking assignments for sequential `always` blocks

Simulation

- **Non-blocking Simulation**

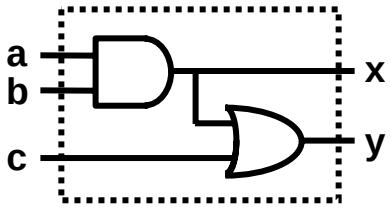


- **Blocking Simulation**



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



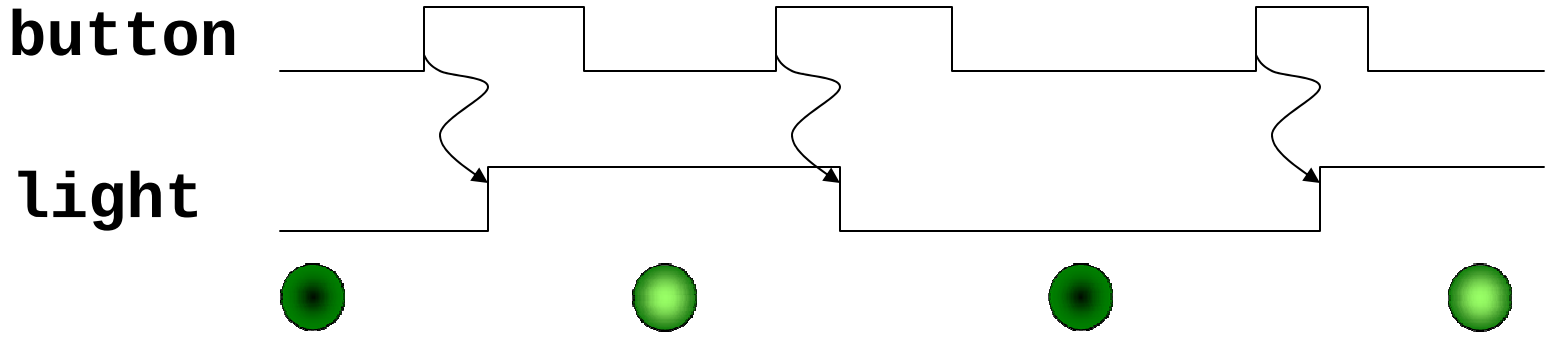

```
module blocking(a,b,c,x,y);
  input a,b,c;
  output x,y;
  reg x,y;

  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	

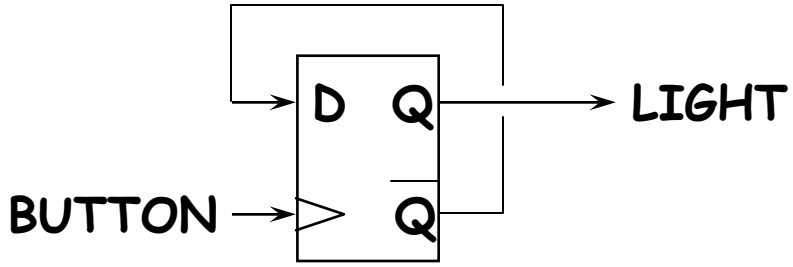
```
module nonblocking(a,b,c,x,y);
  input a,b,c;
  output x,y;
  reg x,y;

  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

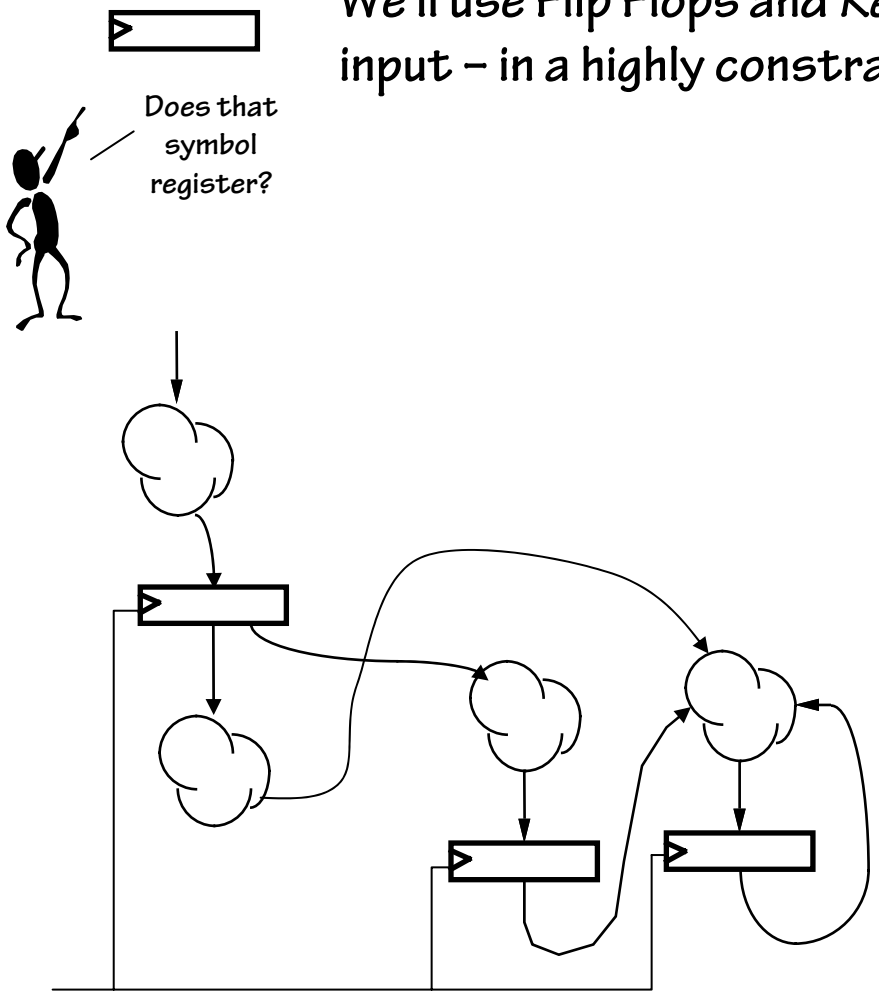
Implementation for on/off button




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



Single-clock Synchronous Circuits



We'll use Flip Flops and *Registers* – groups of FFs sharing a clock input – in a highly constrained way to build digital systems:

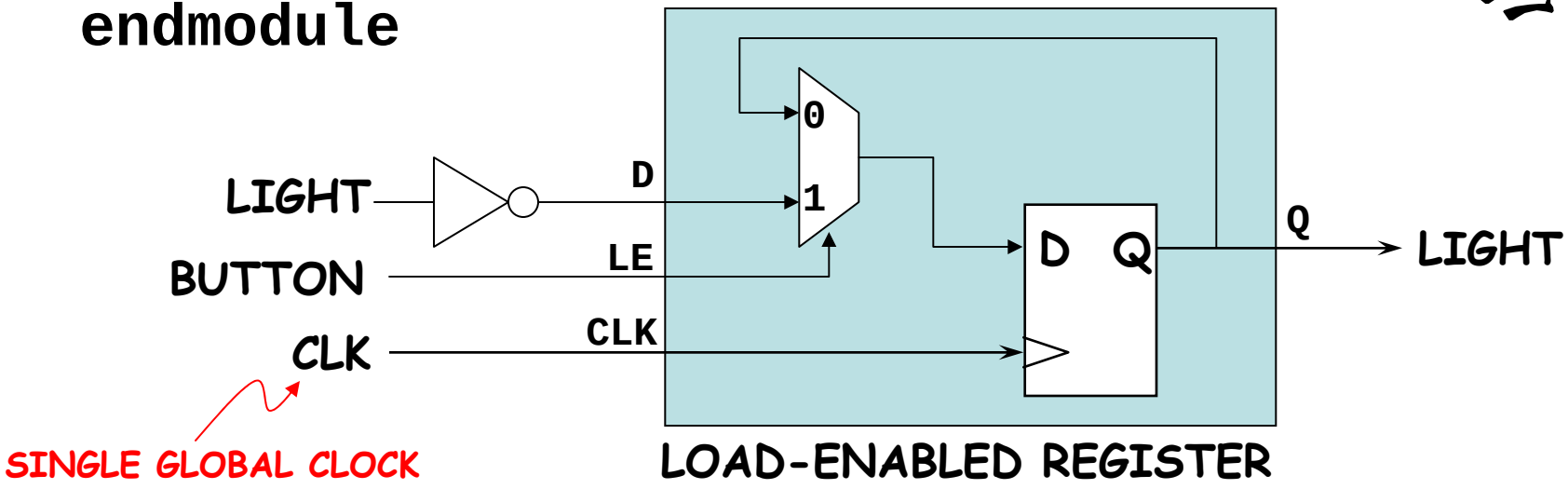
Single-clock Synchronous Discipline

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

Clocked circuit for on/off button

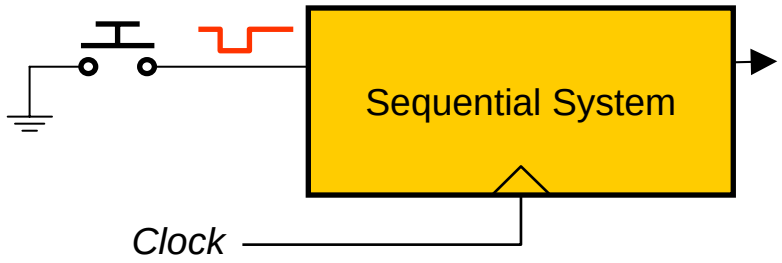
```
module onoff(clk,button,light);
  input clk,button;
  output light;
  reg light;
  always @ (posedge clk)
  begin
    if (button) light <= ~light;
  end
endmodule
```

Does this work with a 1Mhz CLK?



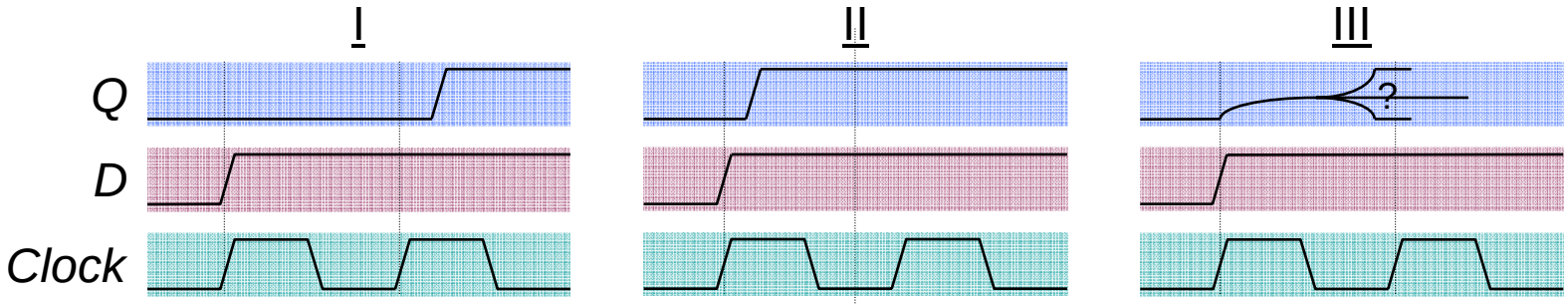
Asynchronous Inputs in Sequential Systems

What about external signals?



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

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



Transition is missed on first clock cycle, but caught on next clock cycle.

Transition is caught on first clock cycle.

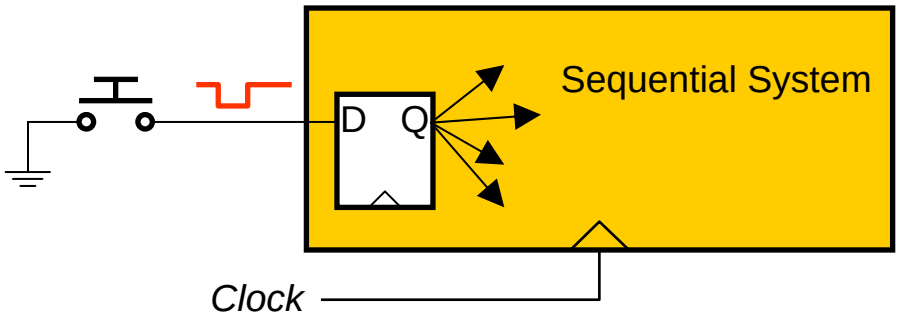
Output is metastable for an indeterminate amount of time.

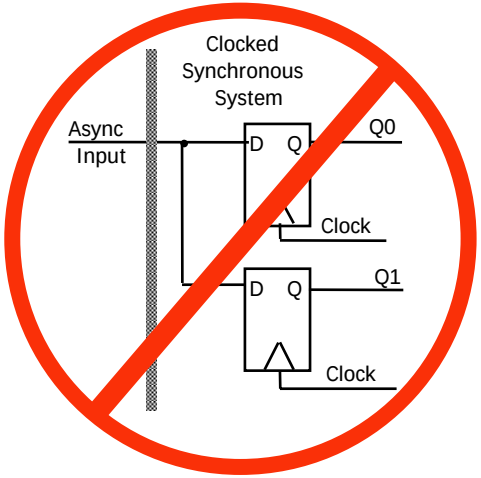
Q: Which cases are problematic?

Asynchronous Inputs in Sequential Systems

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

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

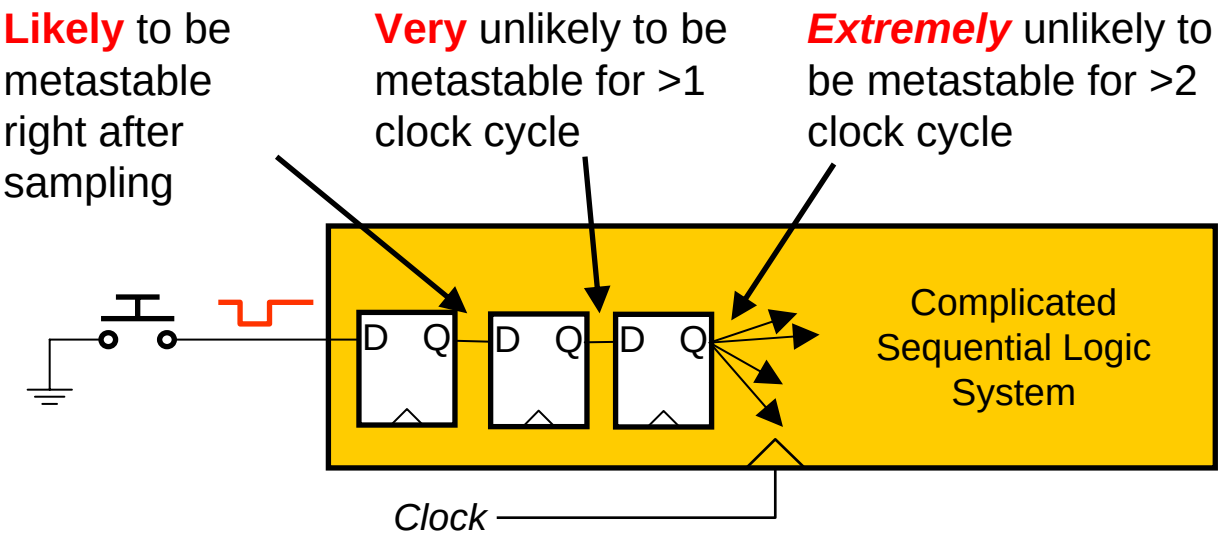




This prevents the possibility of I and II occurring in different places in the circuit, but what about metastability?

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

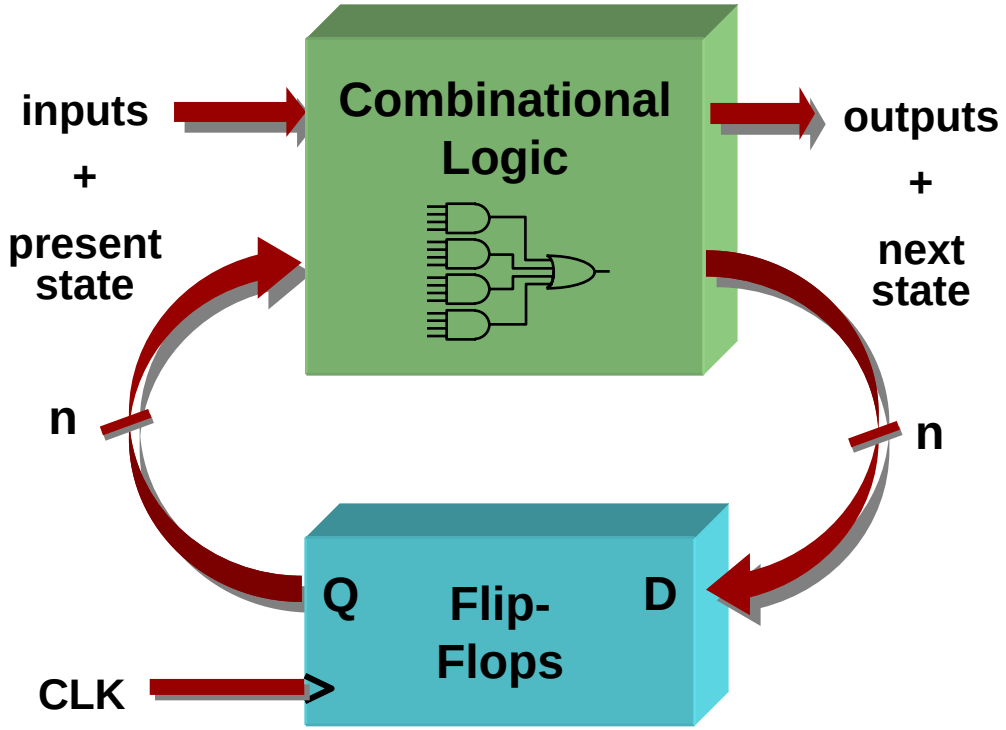


How many registers are necessary?

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

Finite State Machines

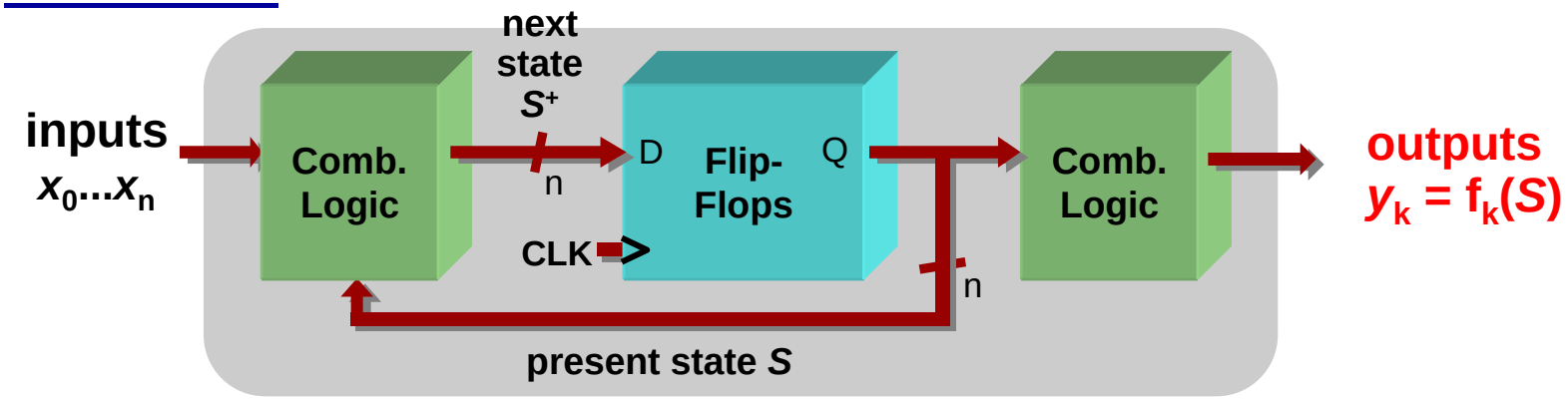
- Finite State Machines (FSMs) are a useful abstraction for sequential circuits with centralized "states" of operation
- At each clock edge, combinational logic computes *outputs* and *next state* as a function of *inputs* and *present state*



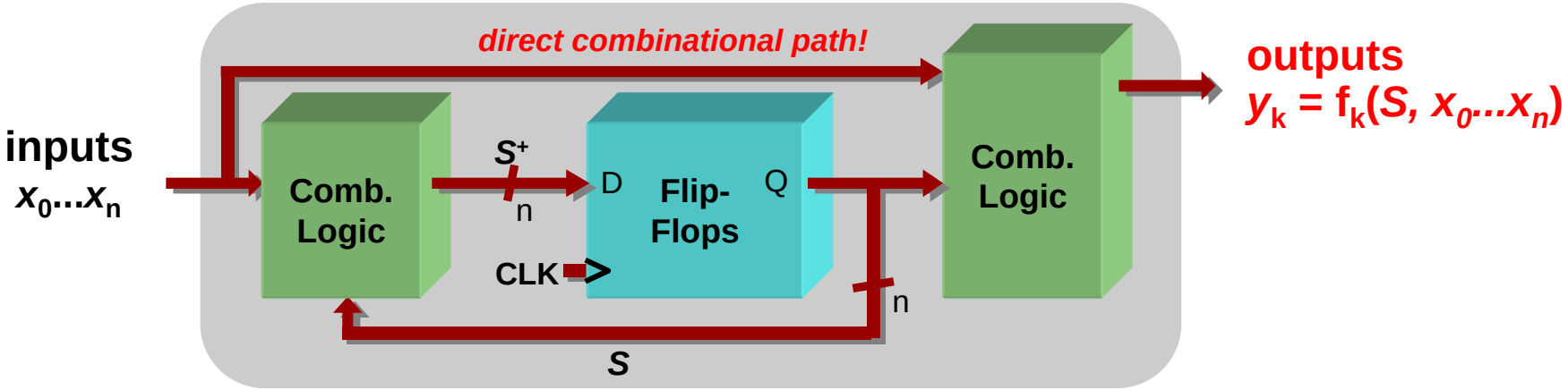
Two Types of FSMs

Moore and **Mealy** FSMs are distinguished by their output generation

Moore FSM:

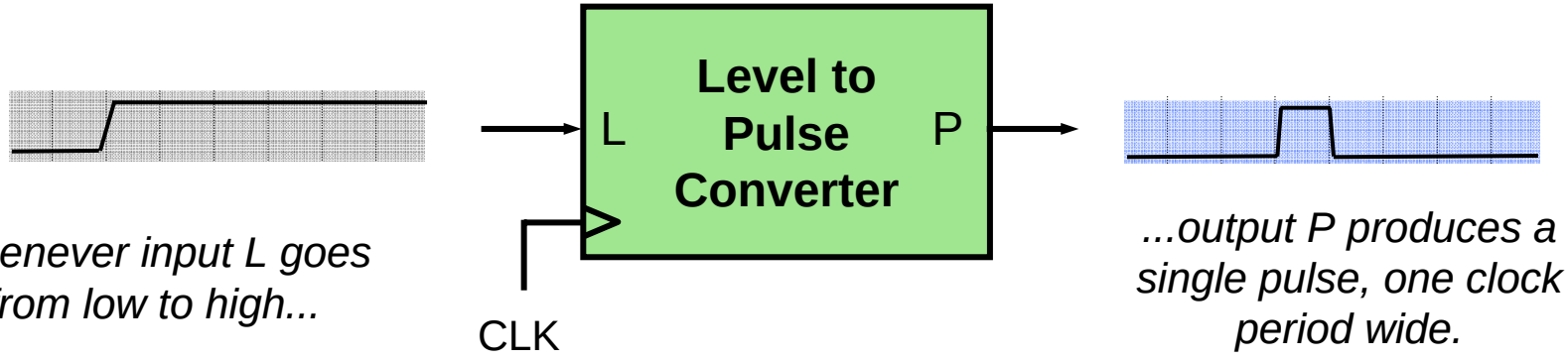


Mealy FSM:



On/off button done right!

- A level-to-pulse converter produces a single-cycle pulse each time its input goes high.
- In other words, it's a synchronous rising-edge detector.
- Sample uses:
  - Buttons and switches pressed by humans for arbitrary periods of time
  - Single-cycle enable signals for counters

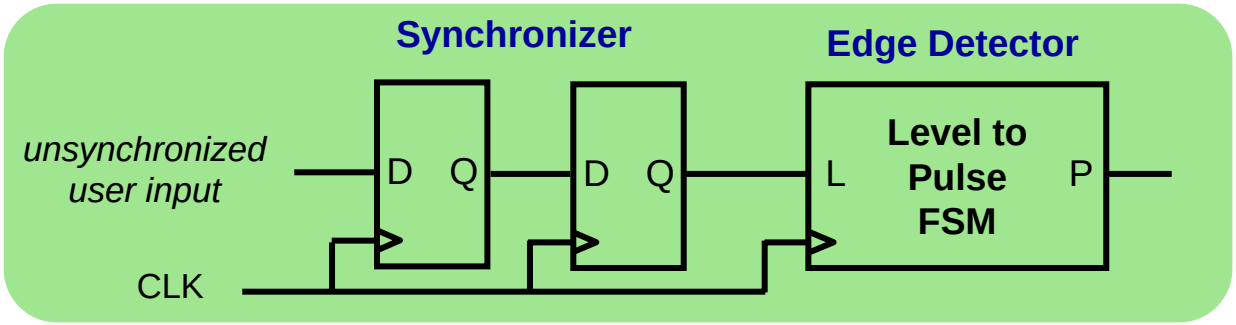


Whenever input L goes from low to high...

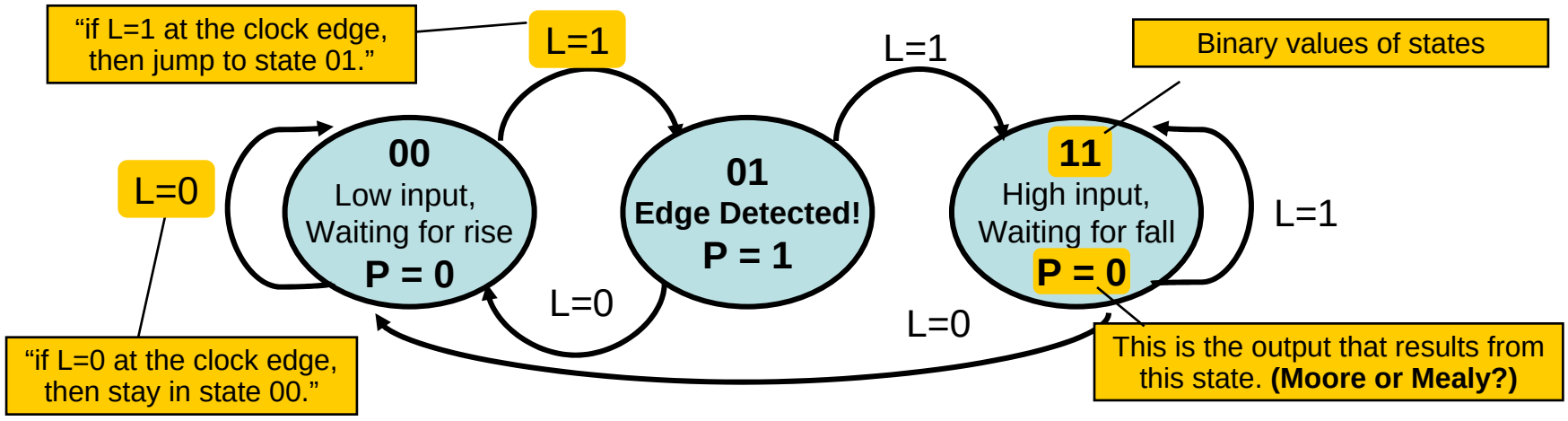
...output P produces a single pulse, one clock period wide.

State Transition Diagrams

- **Block diagram of desired system:**

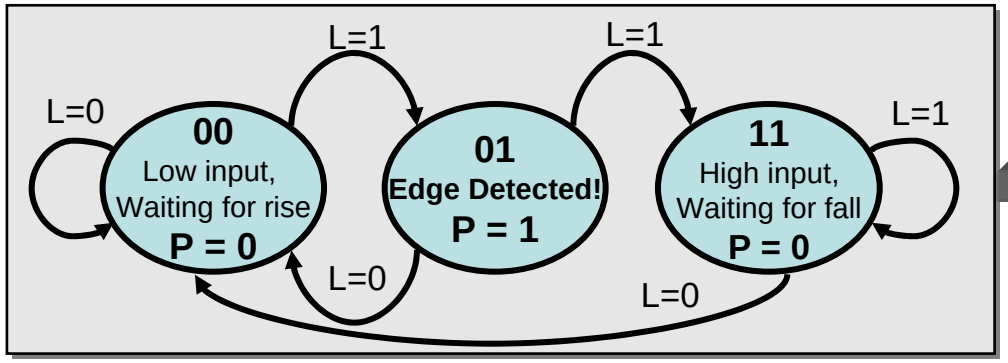


- **State transition diagram is a useful FSM representation and design aid**



Logic Derivation for a Moore FSM

Transition diagram is readily converted to a state transition table (just a truth table)



Current State		In	Next State		Out
S_1	S_0	L	S_1^+	S_0^+	P
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	1
1	1	0	0	0	0
1	1	1	1	1	0

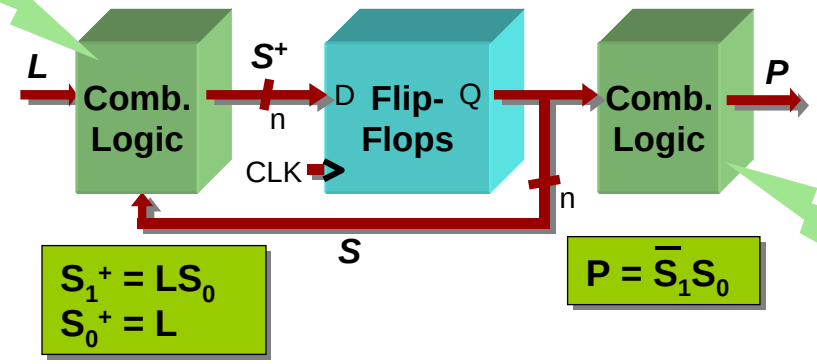
- Combinational logic may be derived using Karnaugh maps

for S_1^+:

L \ S_1S_0	00	01	11	10
0	0	0	0	X
1	0	1	1	X

for S_0^+:

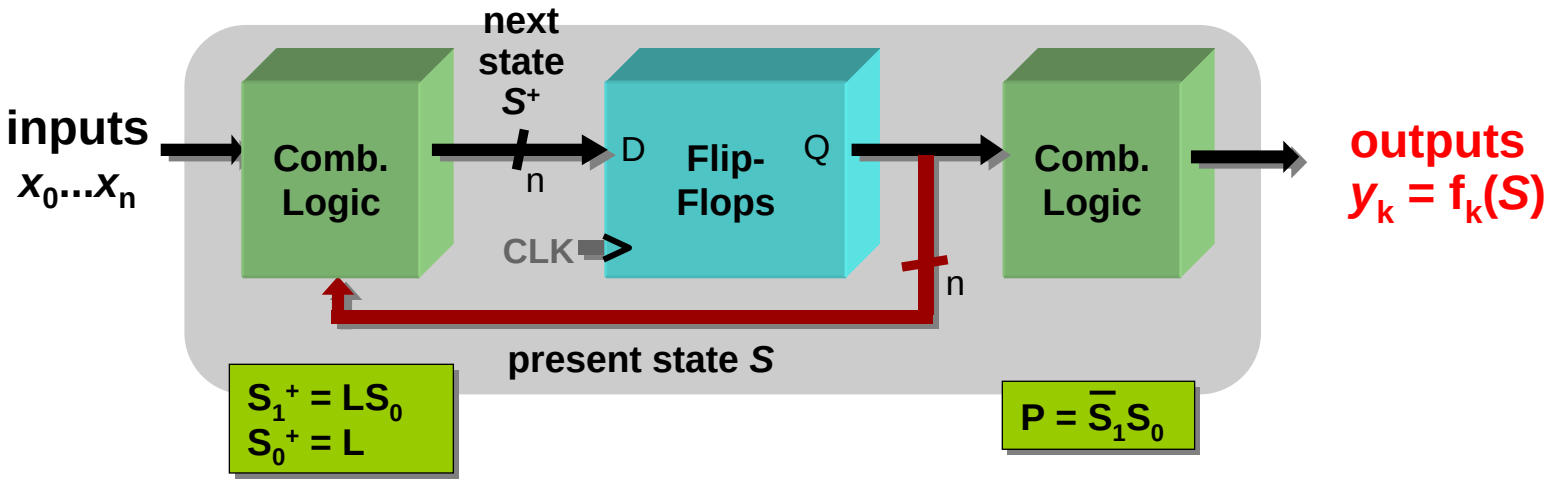
L \ S_1S_0	00	01	11	10
0	0	0	0	X
1	1	1	1	X



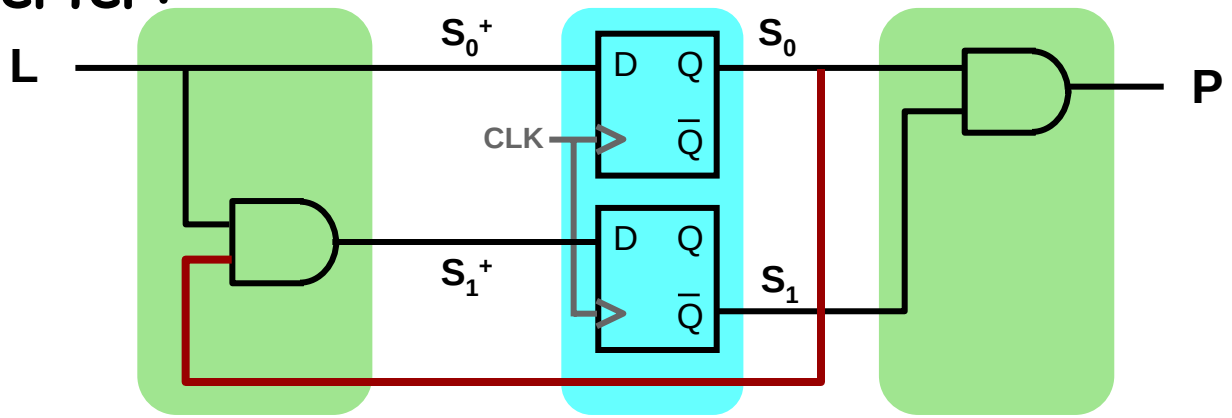
for P:

S_0 \ S_1	0	1
0	0	X
1	1	0

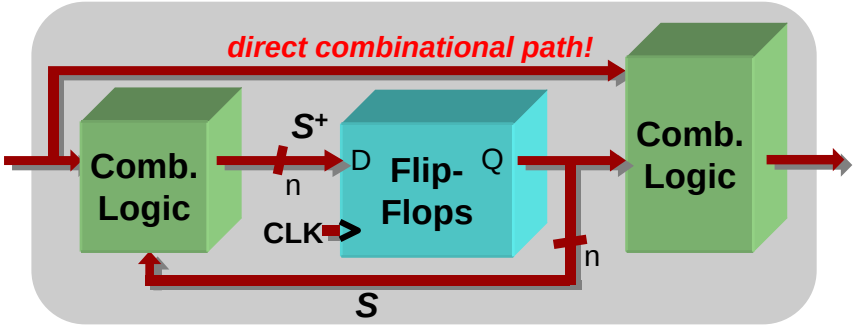
Moore Level-to-Pulse Converter



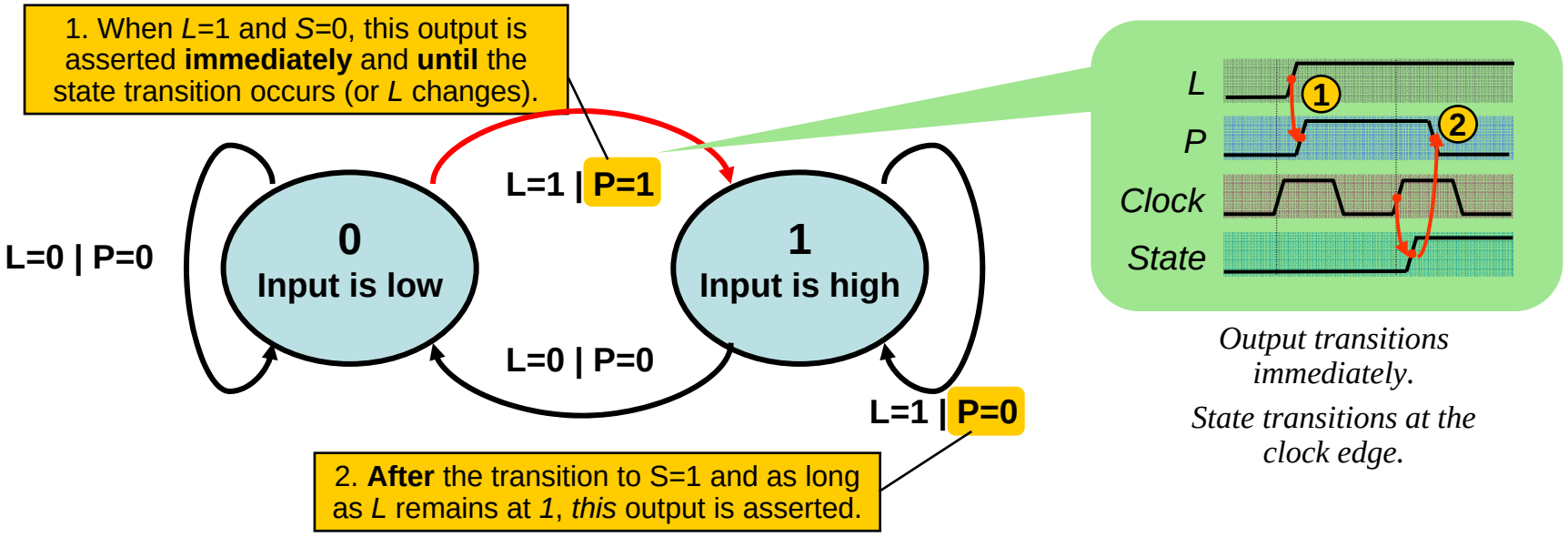
Moore FSM circuit implementation of level-to-pulse converter:



Design of a Mealy Level-to-Pulse



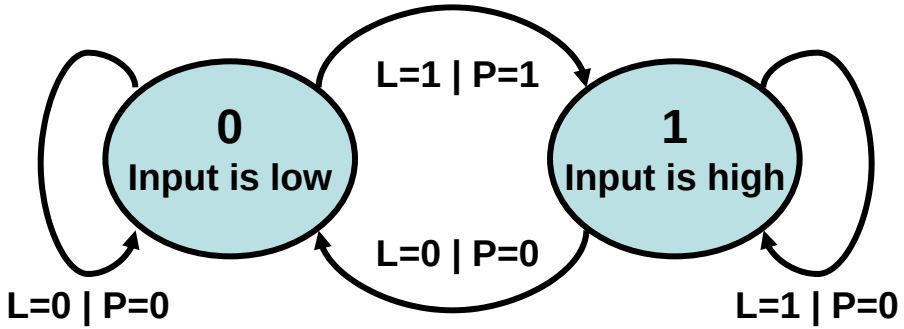
- Since outputs are determined by state *and* inputs, Mealy FSMs may need fewer states than Moore FSM implementations



Output transitions immediately.

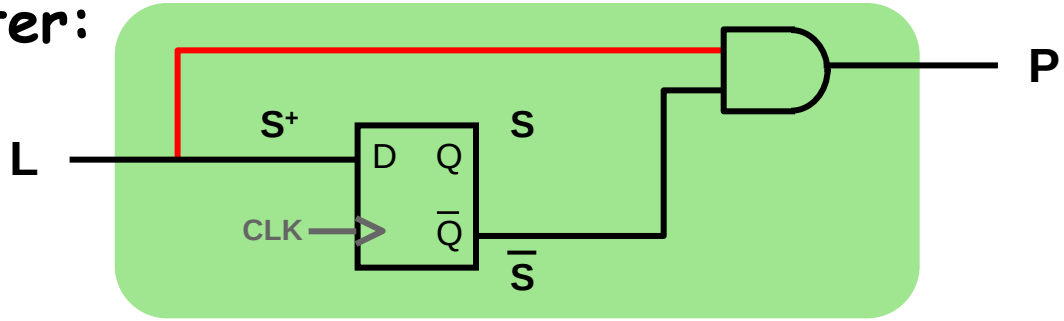
State transitions at the clock edge.

Mealy Level-to-Pulse Converter



Pres. State	In	Next State	Out
S	***L***	***S⁺***	***P***
0	0	0	0
0	1	1	1
1	0	0	0
1	1	1	0

Mealy FSM circuit implementation of level-to-pulse converter:

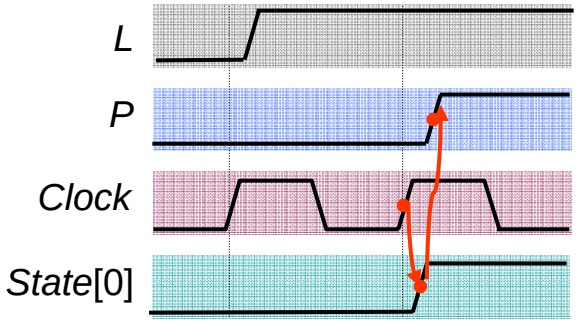


- FSM's state simply remembers the previous value of L
- Circuit benefits from the Mealy FSM's implicit single-cycle assertion of outputs during state transitions

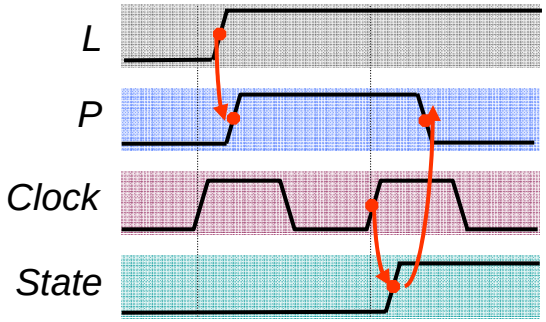
Moore/Mealy Trade-Offs

- Remember that the difference is in the output:
  - Moore outputs are based on state only
  - Mealy outputs are based on state *and input*
  - Therefore, Mealy outputs generally occur one cycle earlier than a Moore:

Moore: delayed assertion of P



Mealy: immediate assertion of P

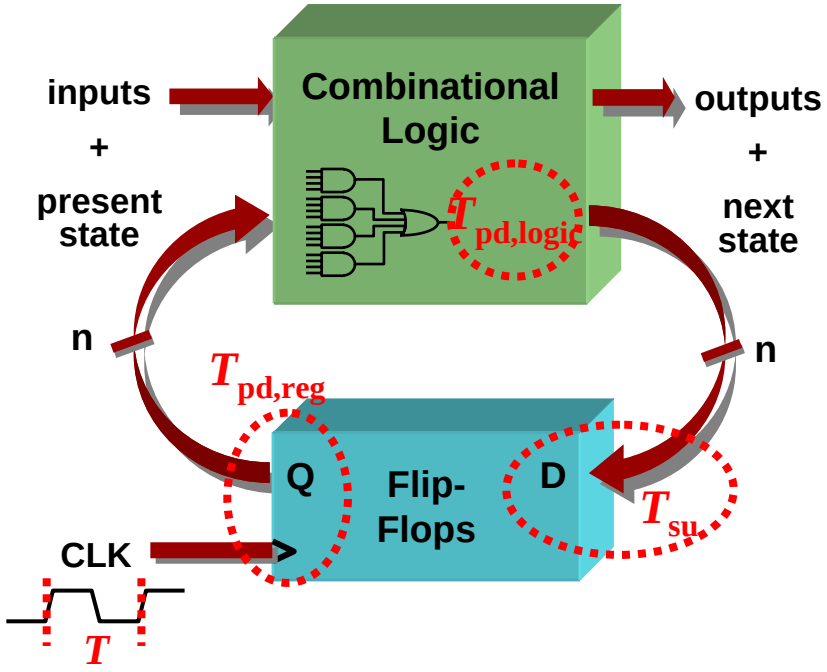


- Compared to a Moore FSM, a Mealy FSM might...
  - Be more difficult to conceptualize and design
  - Have fewer states

FSM Timing Requirements

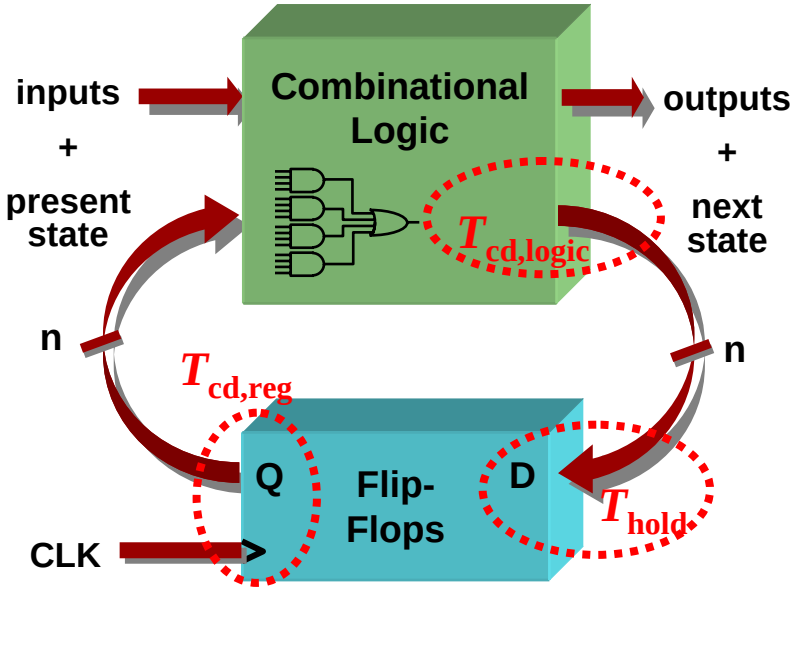
- Timing requirements for FSM are identical to any generic sequential system with feedback

Minimum Clock Period



$$T > T_{pd,reg} + T_{pd,logic} + T_{su}$$

Minimum Delay



$$T_{cd,reg} + T_{cd,logic} > T_{hold}$$

Summary

- Use blocking assignments for combinational `always` blocks
- Use non-blocking assignments for sequential `always` blocks
- Synchronous design methodology usually used in digital circuits
  - Single global clocks to all sequential elements
  - Sequential elements almost always of edge-triggered flavor (design with latches can be tricky)