I. 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;                         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
```

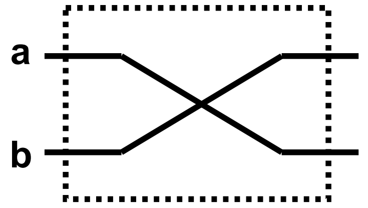
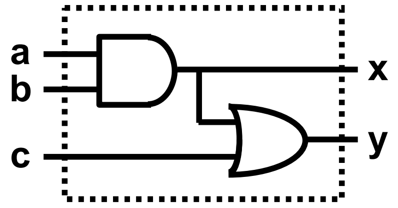
- *Nonblocking assignment:* all assignments deferred until all right-hand sides have been evaluated (end of simulation timestep)

```
always @ (a or b or c)
begin
   x <= a | b;                        1. Evaluate a | b but defer assignment of x
   y <= a ^ b ^ c;                    2. Evaluate a^b^c  but defer assignment of y
   z <= b & ~c;                       3. Evaluate b&(~c) but defer assignment of z
end                                   4. Assign x, y, and z with their new values
```

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

Why two ways of assigning values?

Conceptual need for two kinds of assignment (in `always` blocks):

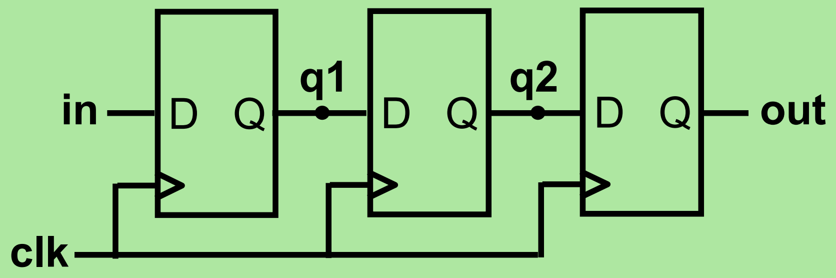
Blocking:
Evaluation and assignment are immediate
	~~`a = b`~~
	~~`b = a`~~
		`x = a & b`
		`y = x | c`

Non-Blocking:
Assignment is postponed until all r.h.s. evaluations are done
	`a <= b`
	`b <= a`
		~~`x <= a & b`~~
		~~`y <= x | c`~~

When to use: (only in always blocks!)	Sequential Circuits	Combinational Circuits

Assignment Styles for Sequential Logic

Flip-Flop Based Digital Delay Line



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

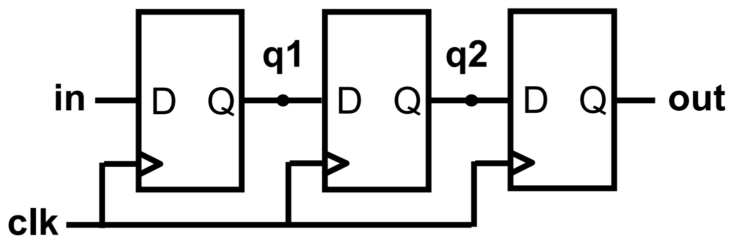
```
module nonblocking(in, clk, out);
  input in, clk;
  output out;
  reg q1, q2, out;
  always @ (posedge clk)
  begin
    q1 <= in;
    q2 <= q1;
    out <= q2;
  end
endmodule
```

```
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

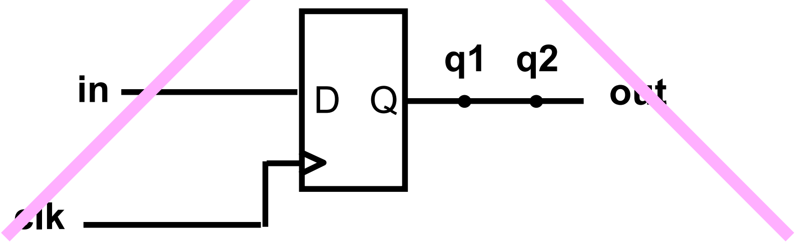
```
always @ (posedge clk)
begin
   q1 <= in;
   q2 <= q1;
   out <= q2;
end
```

"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*; Finally *out* = *in*."~~



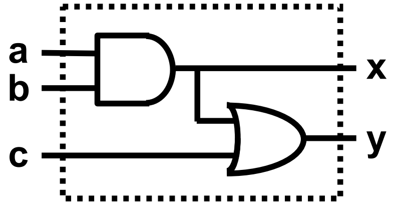
- Blocking assignments do not reflect the intrinsic behavior of multi-stage sequential logic
- Guideline: use nonblocking assignments for sequential `always` blocks

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

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



~~***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~~	

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

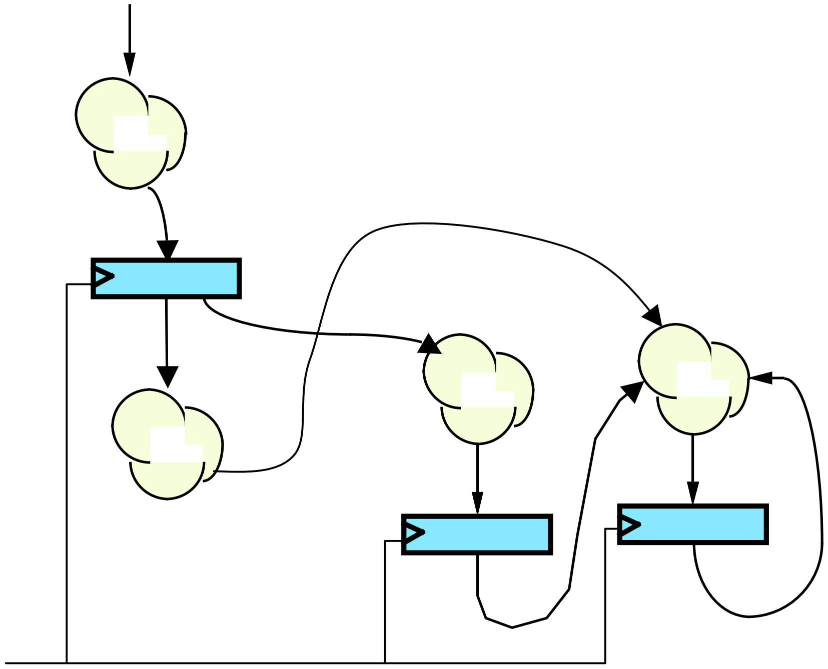
- Nonblocking assignments do not reflect the intrinsic behavior of multi-stage combinational logic
- While nonblocking assignments can be hacked to simulate correctly (expand the sensitivity list), it's not elegant
- **Guideline: use blocking assignments for combinational `always` blocks**

II. 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.

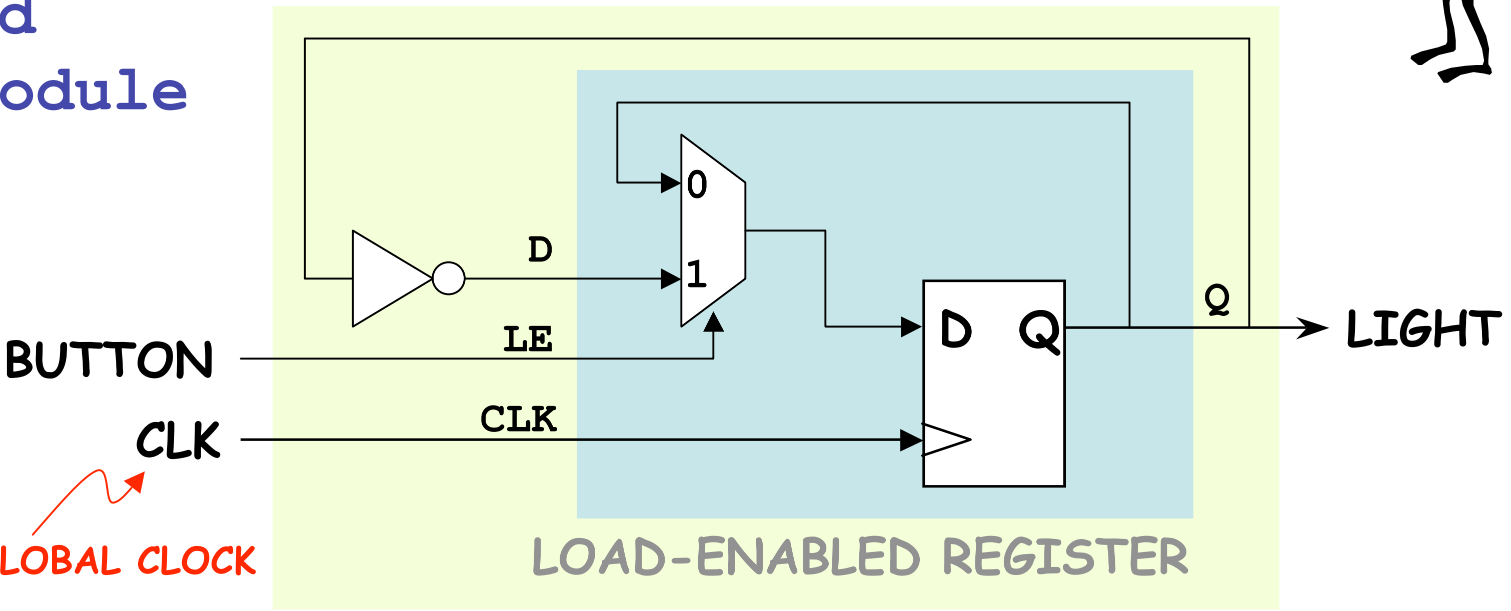
Single-clock Synchronous Discipline:

- 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!



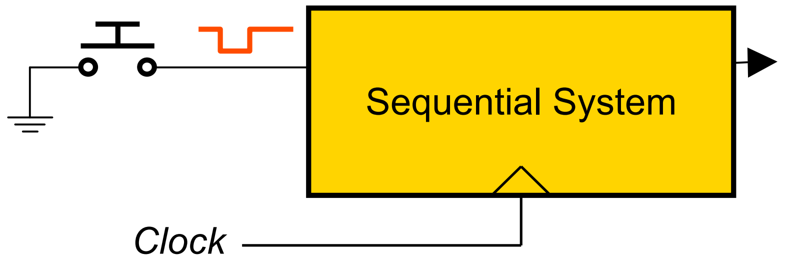
Clocked circuit for on/off button

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



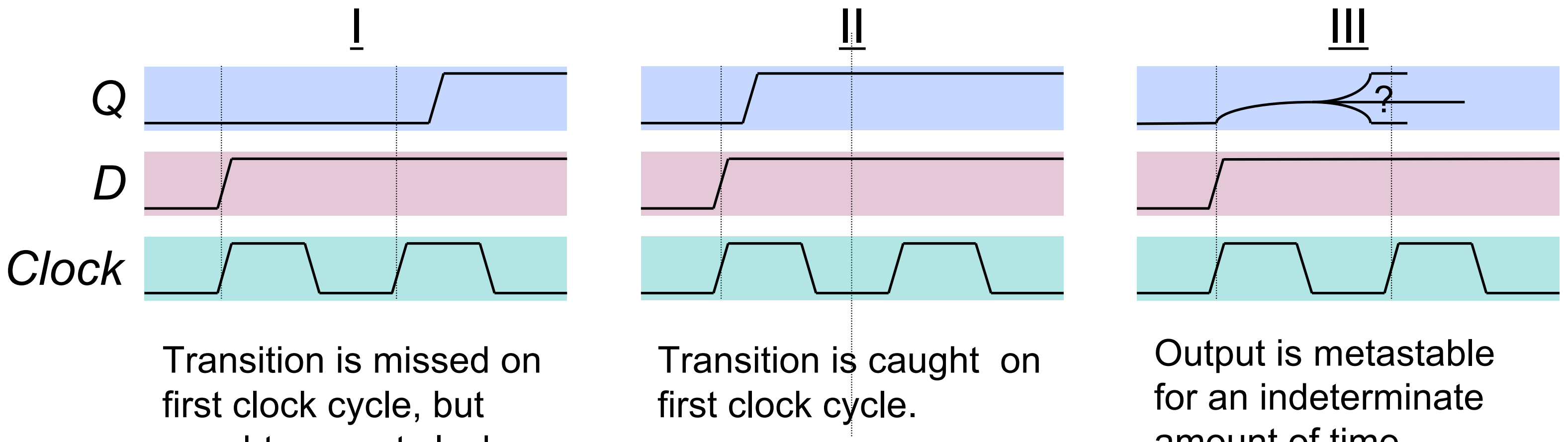
Asynchronous Inputs in Sequential Systems

What about external signals?



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

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



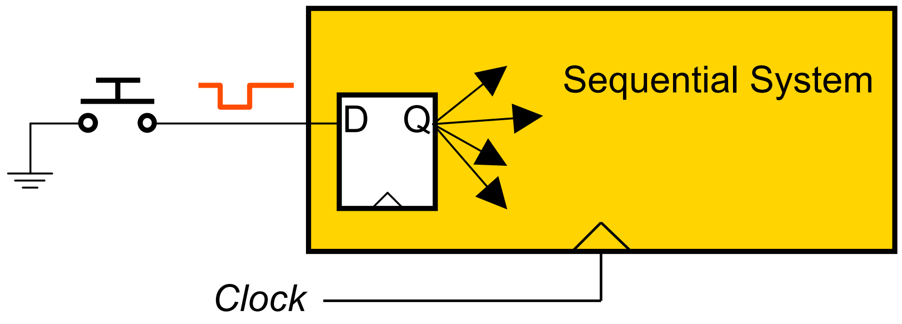
I	II	III
Transition is missed on first clock cycle, but caught on next clock cycle.	Transition is caught on first clock cycle.	Output is metastable for an indeterminate amount of time.

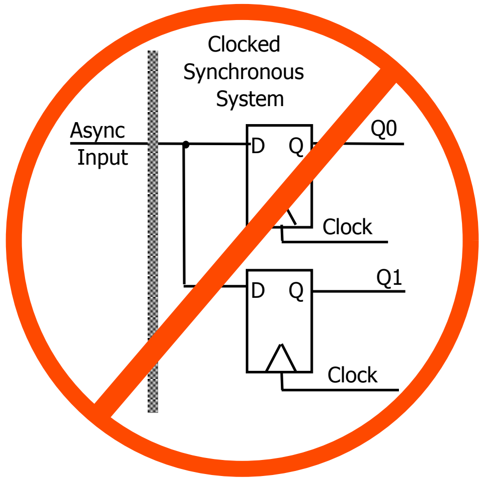
Q: Which cases are problematic?

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*

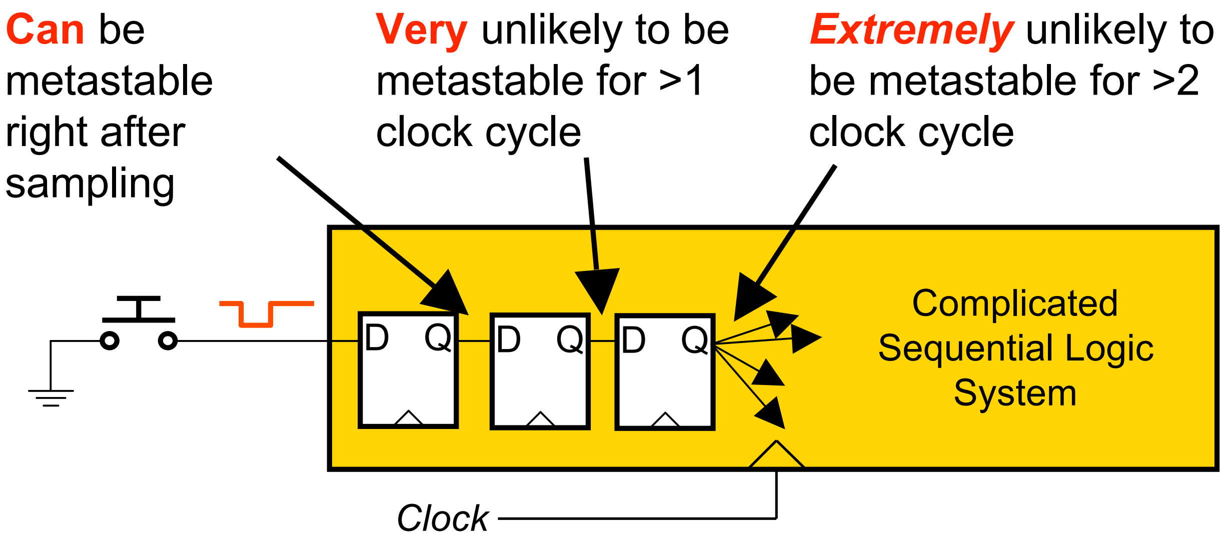




This prevents the possibility of I and II occurring in different places in the circuit, but what about metastability?

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

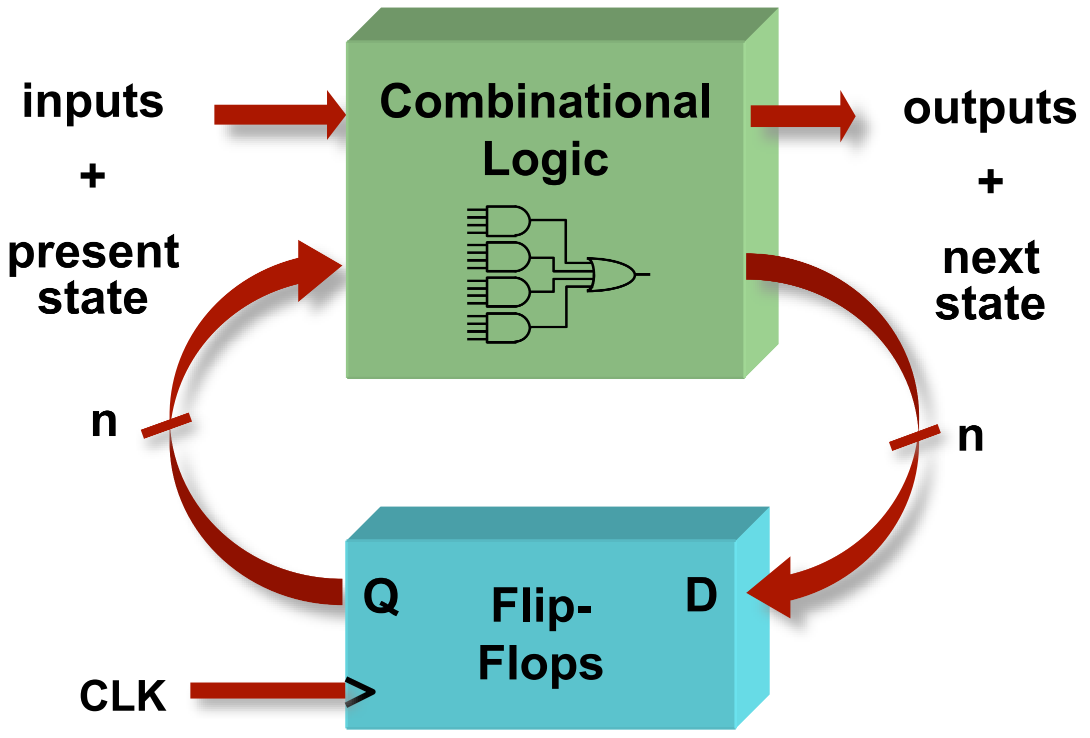

Can be metastable right after sampling
Very unlikely to be metastable for >1 clock cycle
Extremely unlikely to be metastable for >2 clock cycle
D Q
D Q
D Q
Complicated Sequential Logic System
Clock

How many registers are necessary?

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

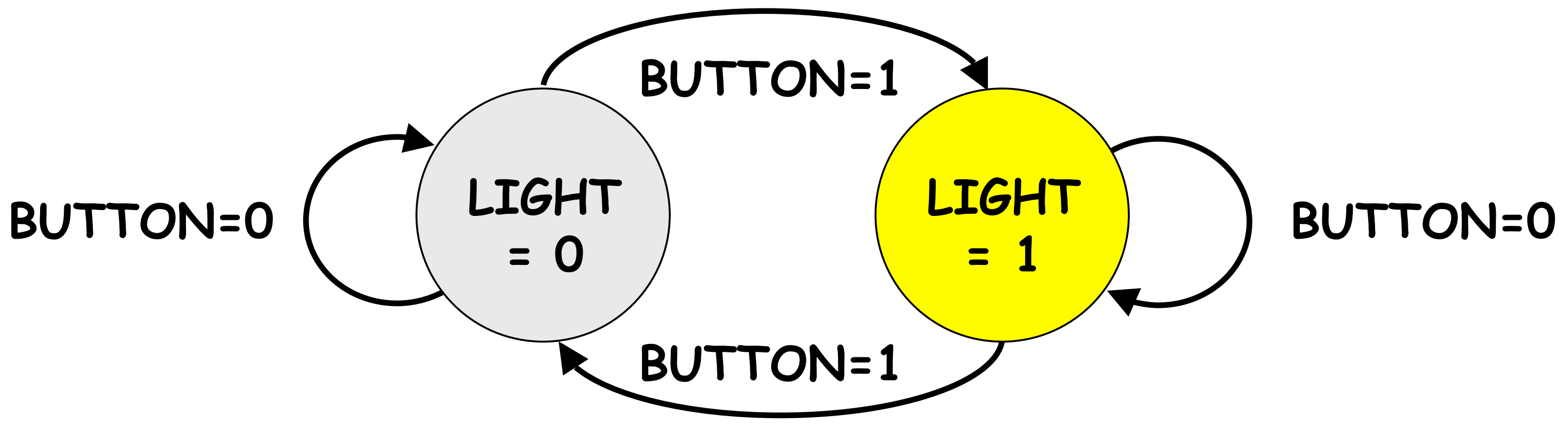
III. <u>Finite State Machines</u>

- 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*

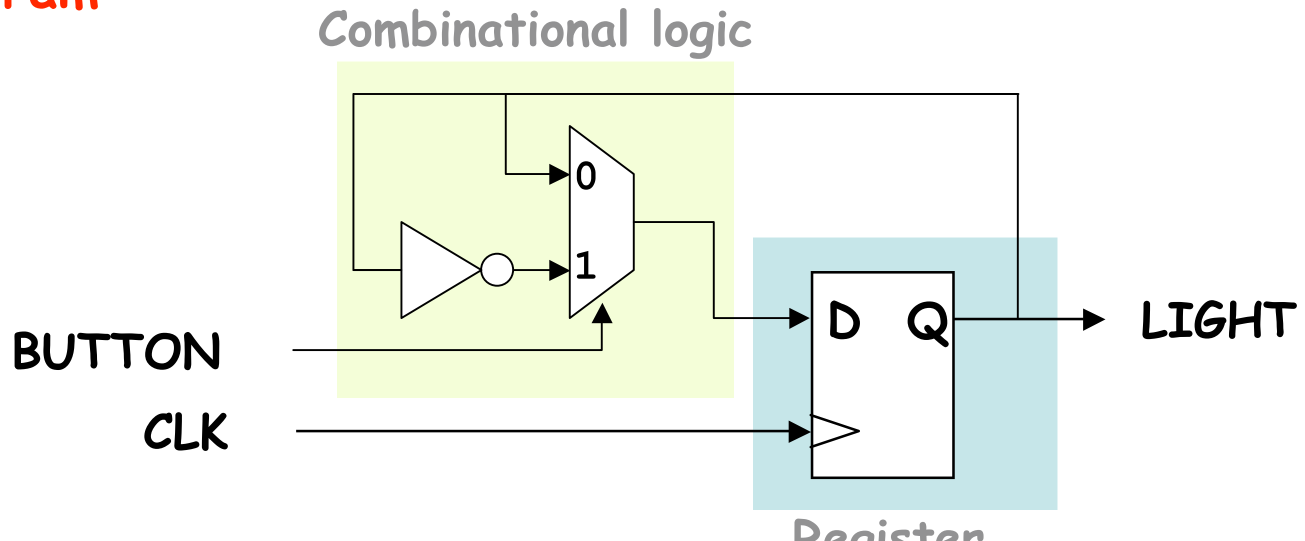


Example 1: Light Switch

- State transition diagram

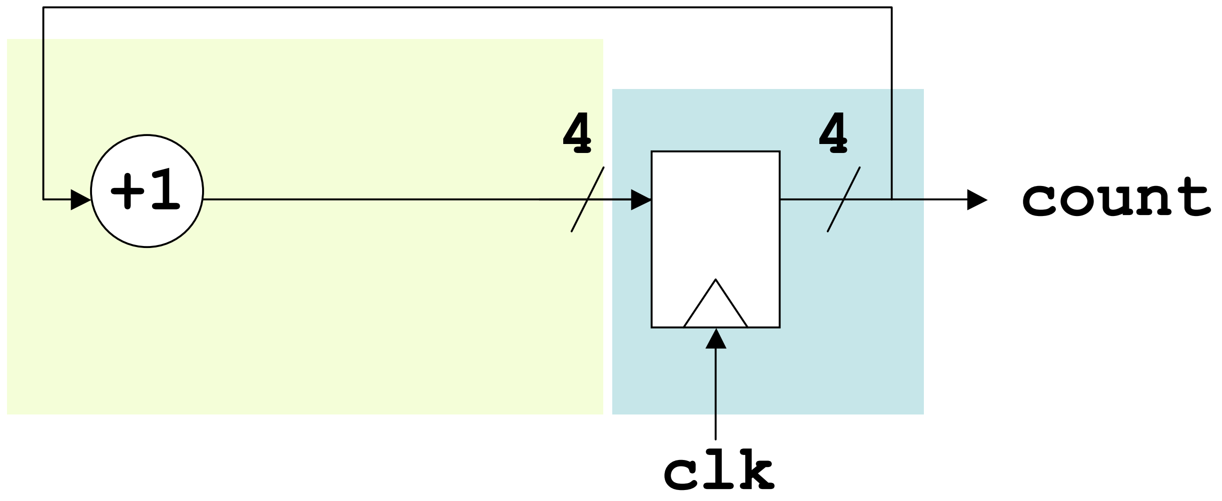


- Logic diagram



Example 2: 4-bit Counter

- Logic diagram

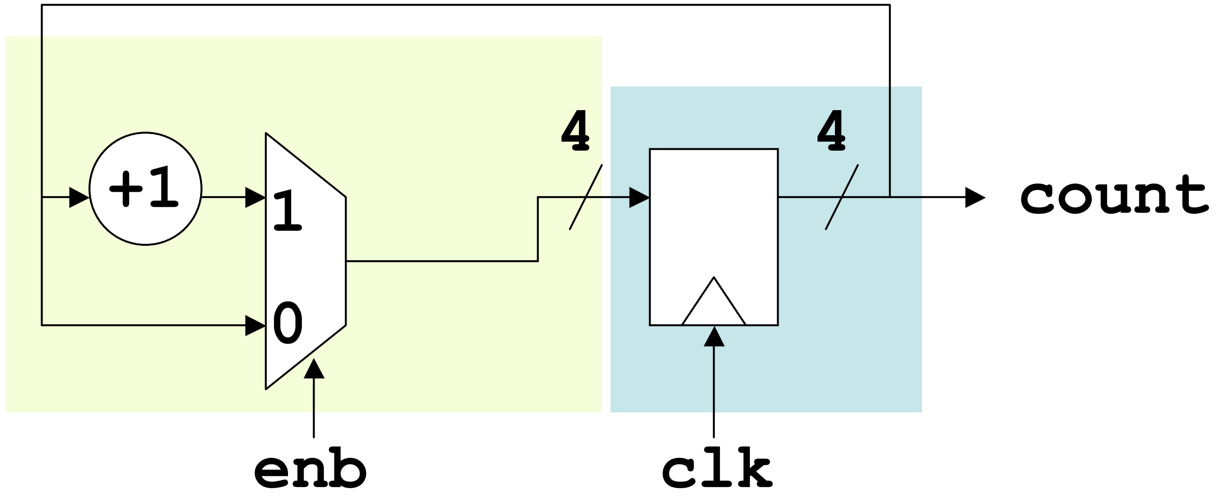


- Verilog

```
# 4-bit counter
module counter(clk, count);
  input clk;
  output [3:0] count;
  reg [3:0] count;

  always @ (posedge clk) begin
    count <= count+1;
  end
endmodule
```

Example 2: 4-bit Counter

- Logic diagram



- Verilog

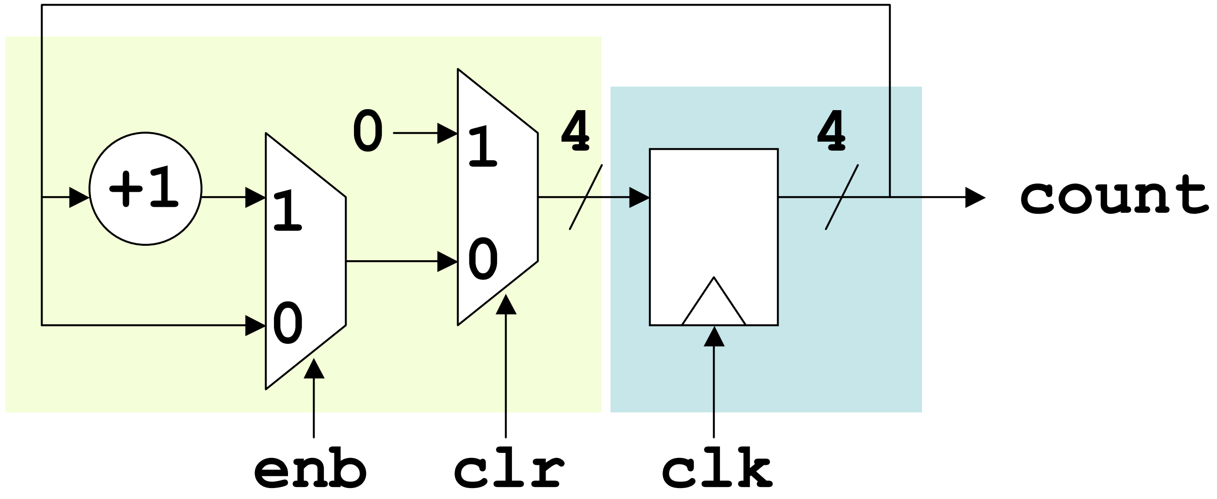
```
# 4-bit counter with enable
module counter(clk,enb,count);
  input clk,enb;
  output [3:0] count;
  reg [3:0] count;

  always @ (posedge clk) begin
    count <= enb ? count+1 : count;
  end
endmodule
```

Could I use the following instead?

```
if (enb) count <= count+1;
```

Example 2: 4-bit Counter

- Logic diagram



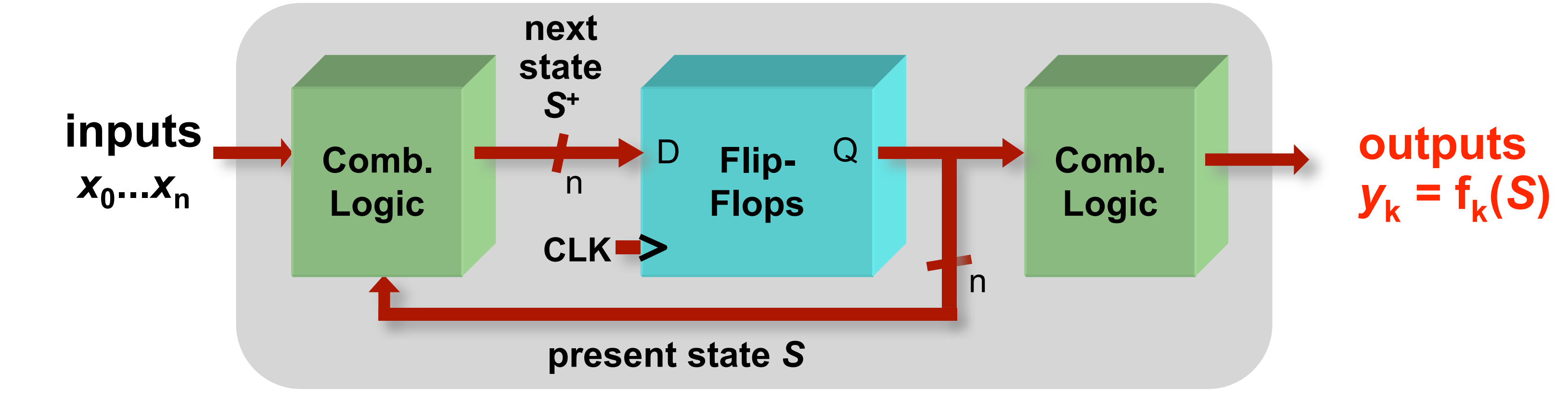
- Verilog

```
# 4-bit counter with enable and synchronous clear
module counter(clk,enb,clr,count);
  input clk,enb,clr;
  output [3:0] count;
  reg [3:0] count;

  always @ (posedge clk) begin
    count <= clr ? 4'b0 : (enb ? count+1 : count);
  end
endmodule
```

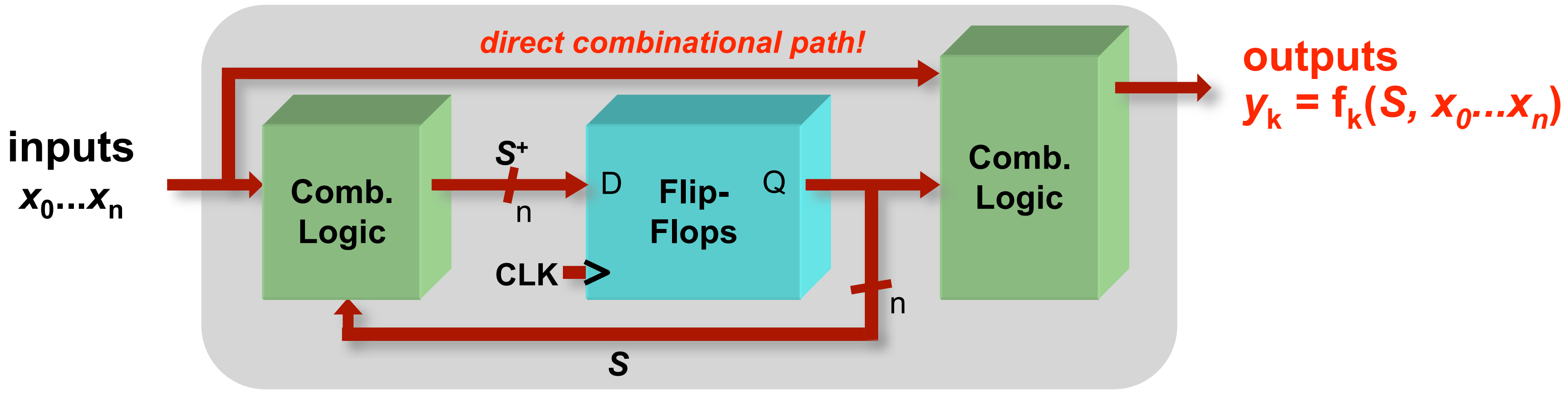
Two Types of FSMs

Moore and Mealy FSMs : different output generation

- **Moore FSM:**

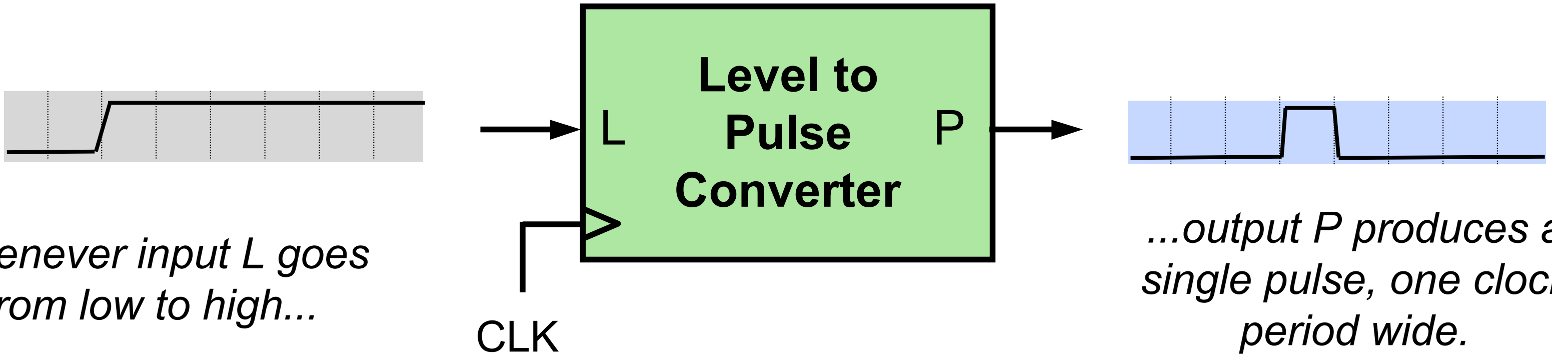


- **Mealy FSM:**



Design Example: Level-to-Pulse

- A level-to-pulse converter produces a single-cycle pulse each time its input goes high.
- 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

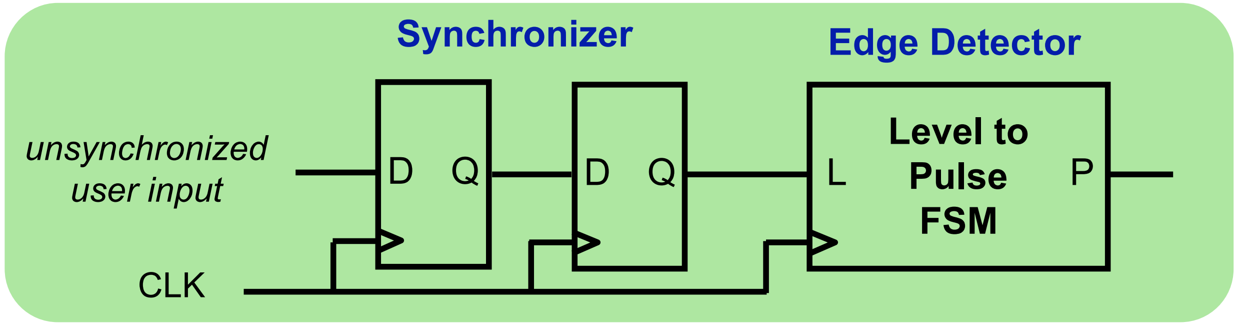


Whenever input L goes from low to high...

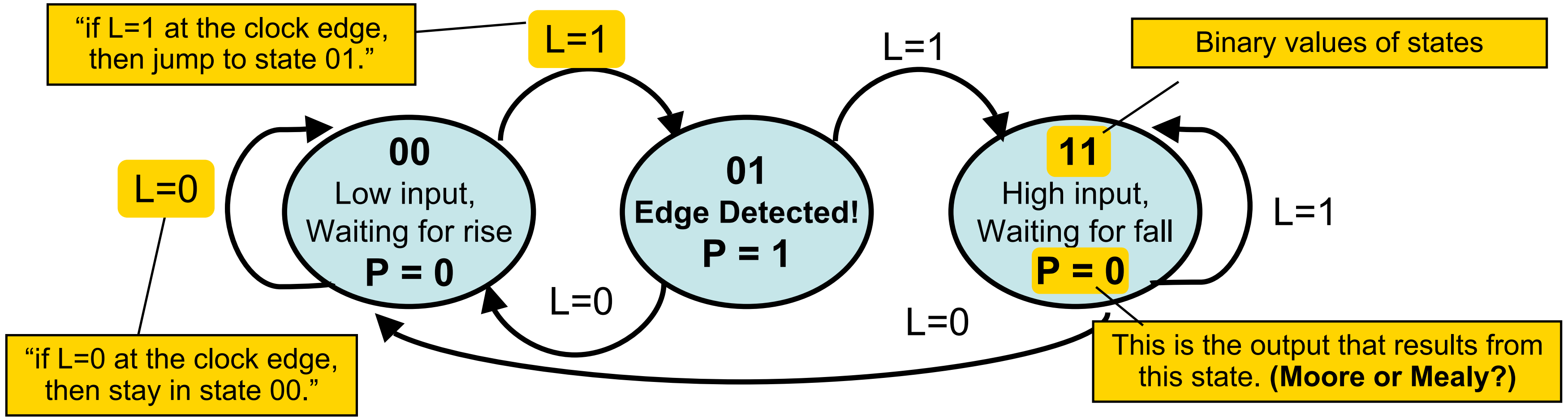
...output P produces a single pulse, one clock period wide.

Step 1: State Transition Diagram

- Block diagram of desired system:

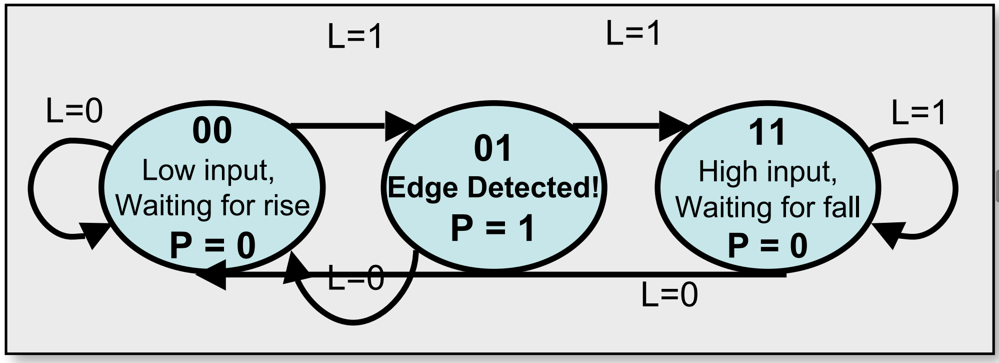


- State transition diagram is a useful FSM representation and design aid:


"if L=1 at the clock edge, then jump to state 01."
L=1
L=1
Binary values of states
L=0
00
Low input, Waiting for rise
P = 0
01
Edge Detected!
P = 1
11
High input, Waiting for fall
P = 0
L=1
L=0
L=0
"if L=0 at the clock edge, then stay in state 00."
This is the output that results from this state. (Moore or Mealy?)

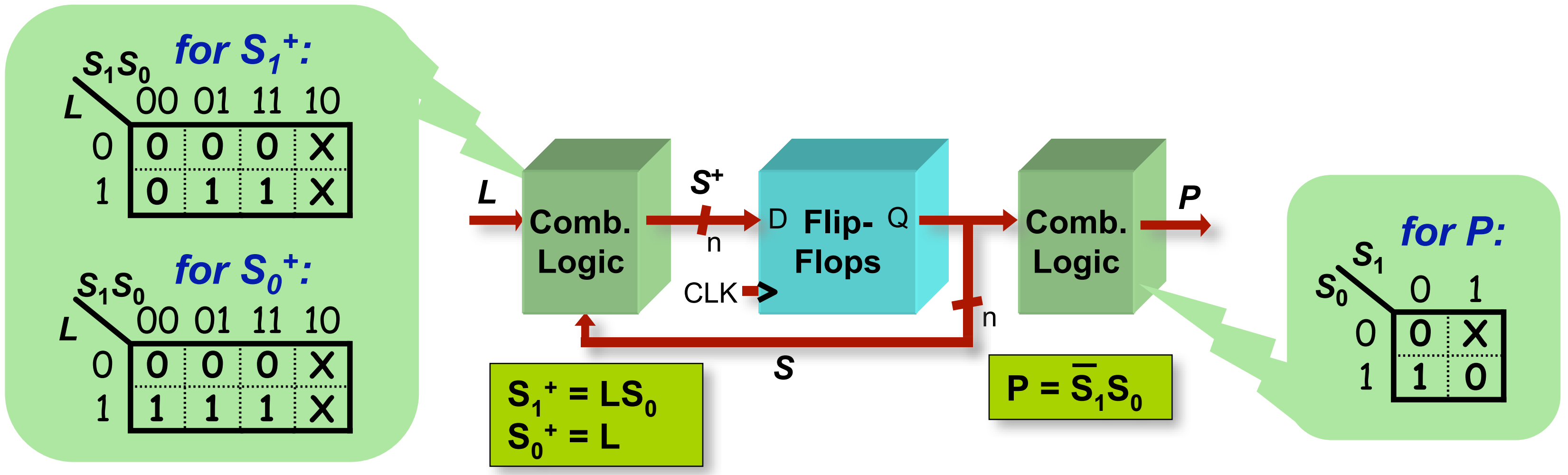
Step 2: Logic Derivation

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



Curren t 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

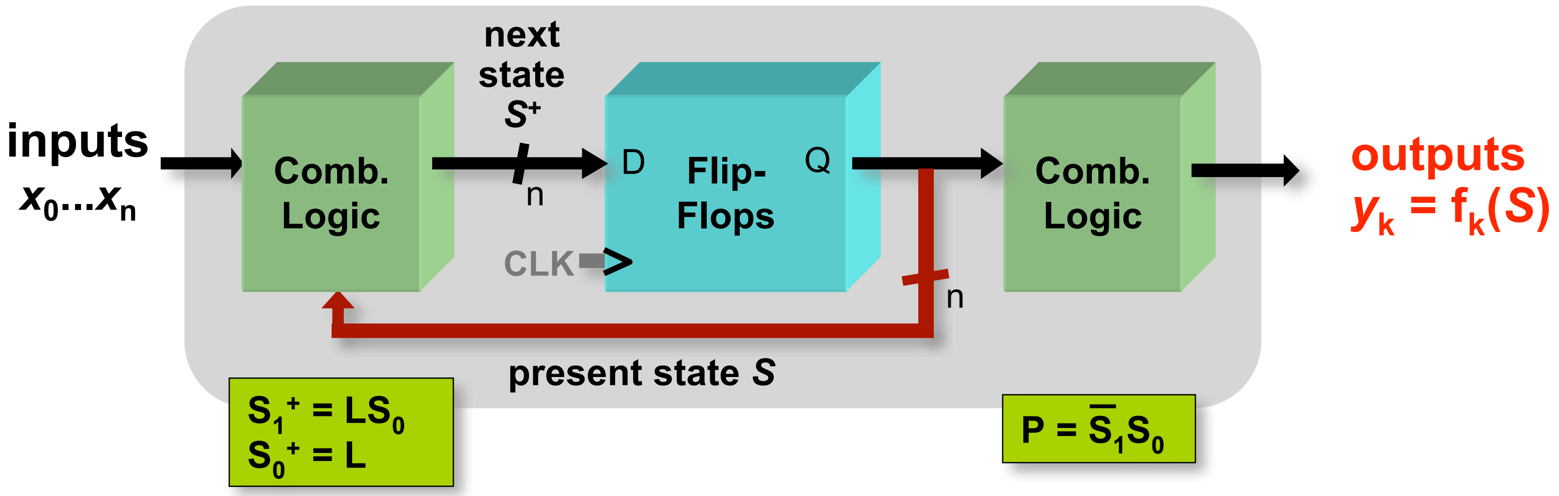
- Combinational logic may be derived using Karnaugh maps



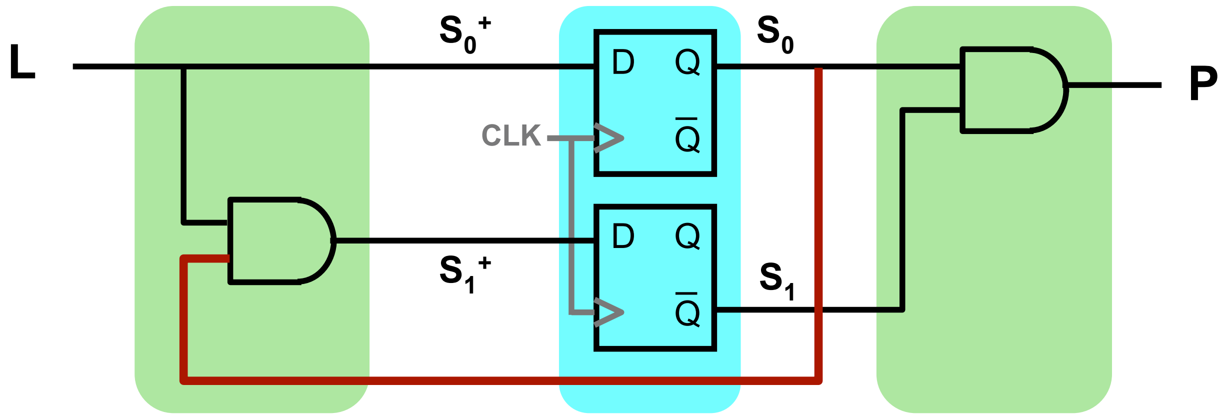
$S_1^+ = LS_0$
$S_0^+ = L$

$P = \overline{S}_1 S_0$

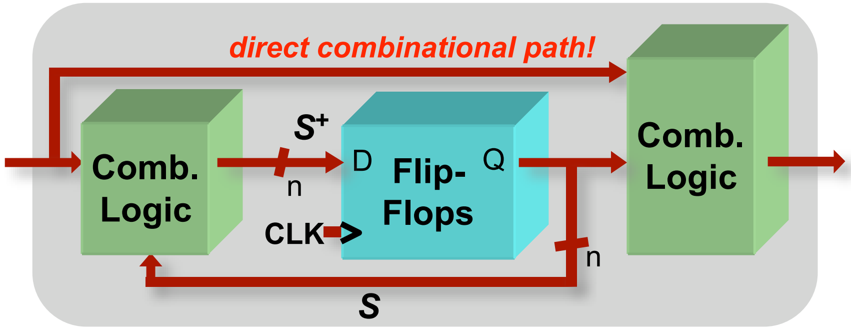
Moore Level-to-Pulse Converter



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

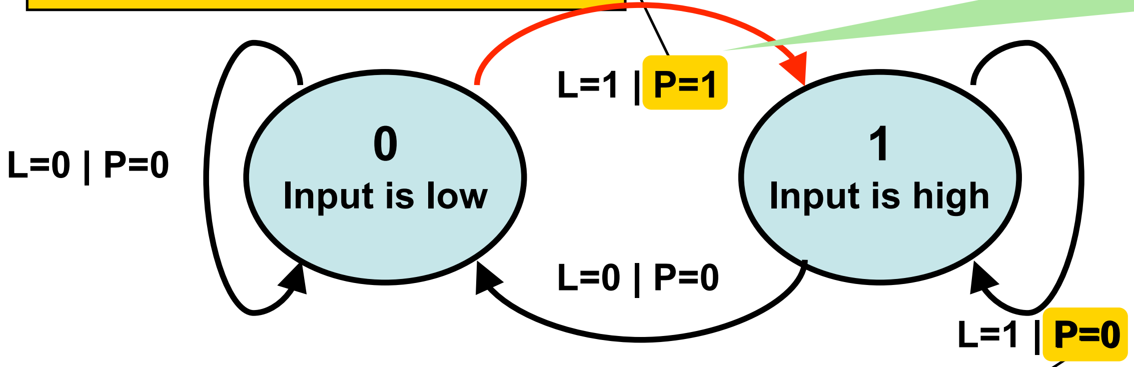


Design of a Mealy Level-to-Pulse

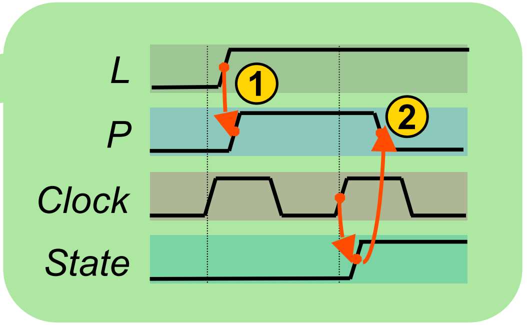


- **Since outputs are determined by state *and* inputs, Mealy FSMs may need fewer states than Moore FSM implementations**

1. When *L*=1 and *S*=0, this output is asserted **immediately** and **until** the state transition occurs (or *L* changes).

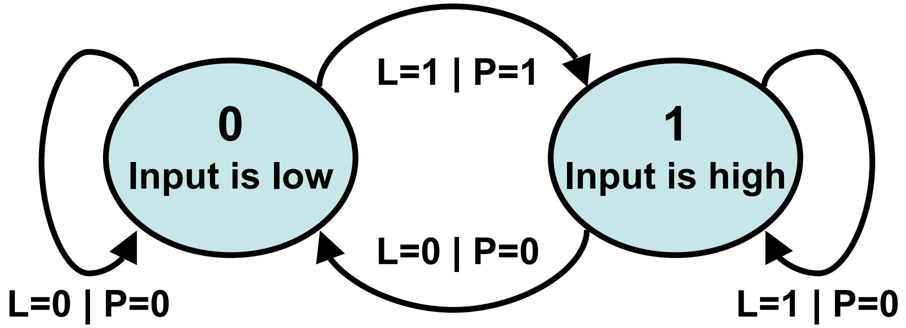


2. **While in state** S=1 and as long as *L* remains at *1*, *this* output is asserted.



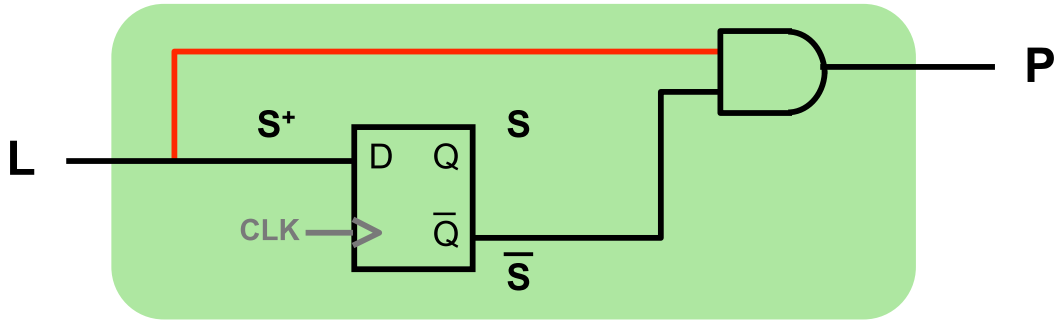
Output transitions immediately.
State transitions at the clock edge.

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:

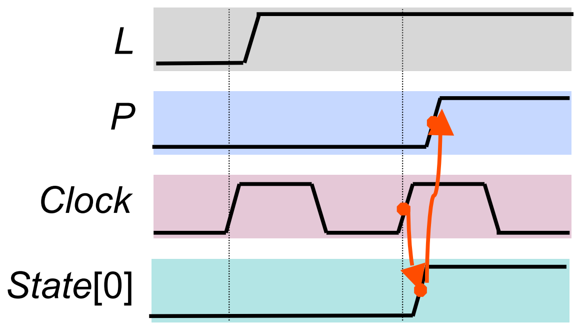


- FSM's state simply remembers the previous value of L
- Circuit benefits from the Mealy FSM's implicit single-cycle assertion of outputs during state transitions

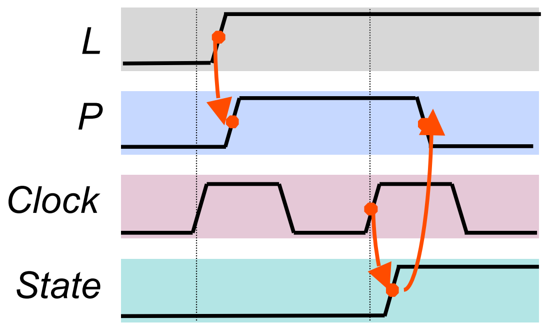
Moore/Mealy Trade-Offs

- How are they different?
  - Moore: outputs = f(state) only
  - Mealy outputs = f(state *and input*)
  - Mealy outputs generally occur one cycle earlier than a Moore:

Moore: delayed assertion of P

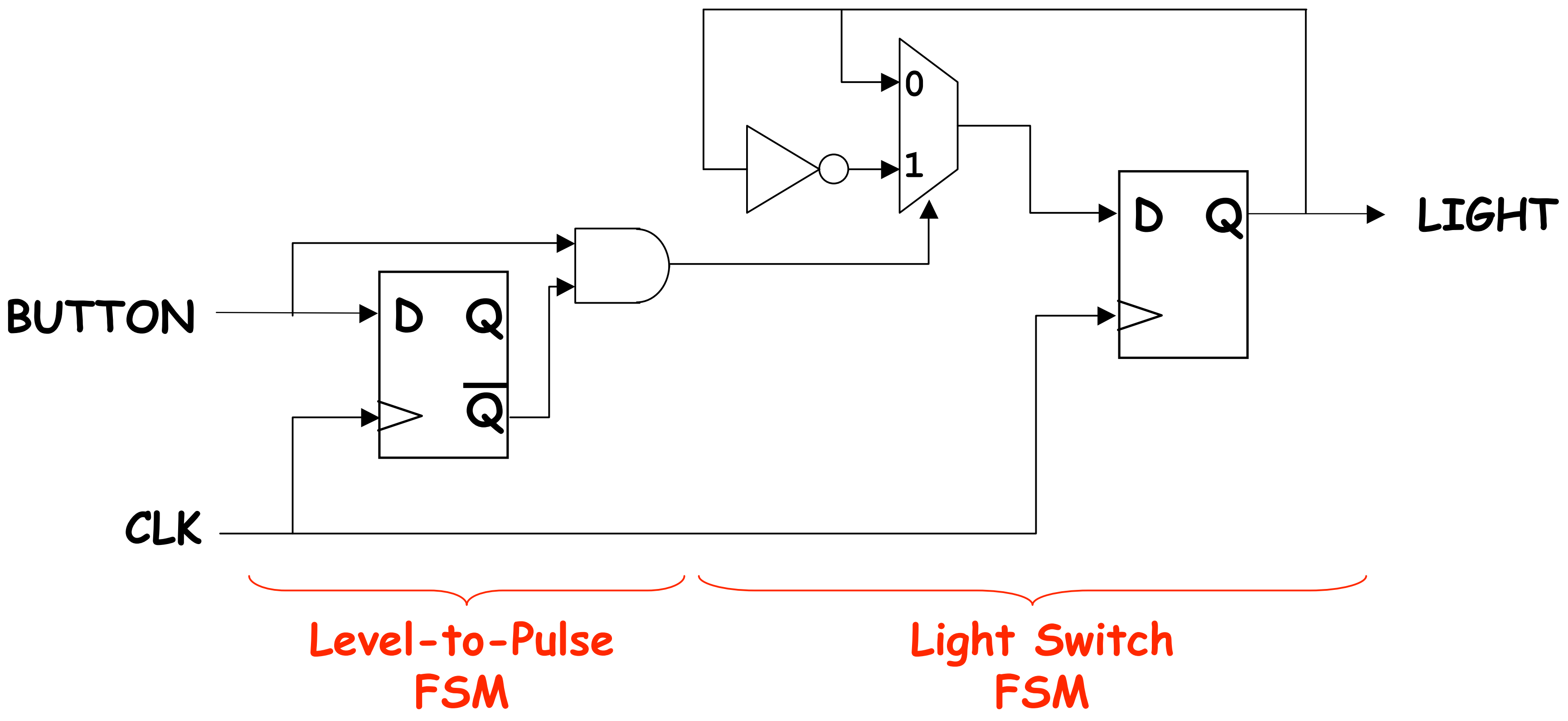


Mealy: immediate assertion of P



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

Light Switch Revisited


0
1
D
Q
LIGHT
BUTTON
D
Q
Q̅
CLK
Level-to-Pulse FSM
Light Switch FSM