# CIRCUIT DESIGN WITH HDL

# Course outline

- Chapter 1: Introduction
- Chapter 2: Verilog Language Concepts
- Chapter 3: Structural modeling
- Chapter 4: Behavioral modeling
- Chapter 5: Finite State machines
- Chapter 6: Tasks and Functions
- Chapter 7: Functional Simulation/Verification (Testbench)
- Chapter 8: Synthesis of Combinational and Sequential Logic
- Chapter 9: Post-synthesis design tasks
- Chapter 10: VHDL introduction

# Introduction

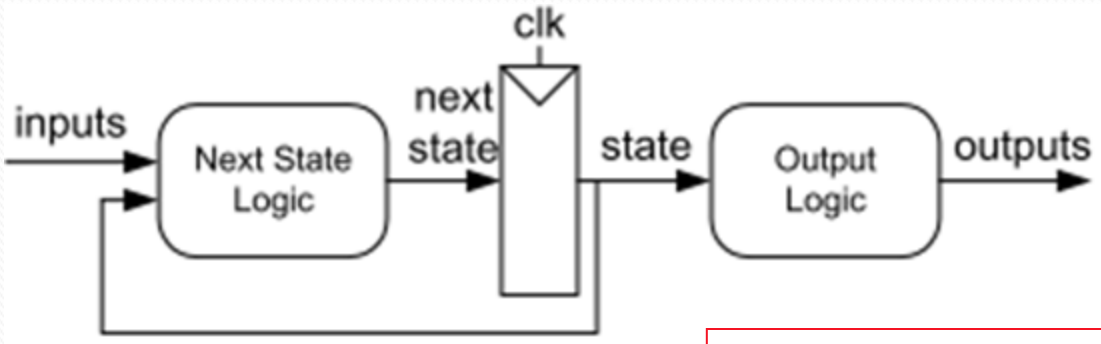
- Finite State Machines (FSMs):
  - A useful abstraction for digital logic circuits with centralized “states” of operation
  - A main control block of any digital logic circuit
- Three basic components:

  - Combinational logic – next states
  - Sequential logic – store states (current states)
  - Output logic

# FSM types

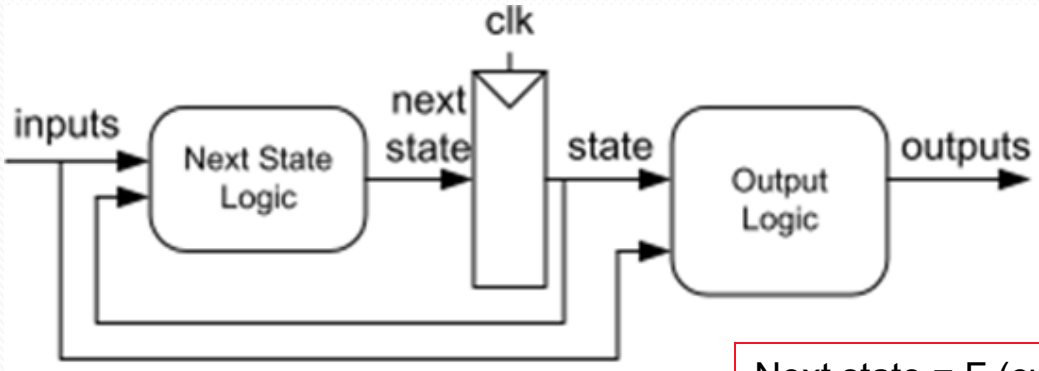
- There are two types of designing FSM

**Moore** type




Next state = F (current state, inputs)
Outputs = G (current state)

**Mealy** type

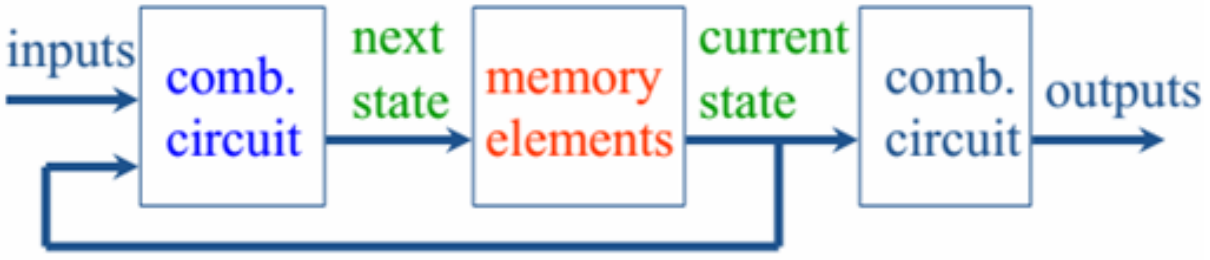



Next state = F (current state, inputs)
Outputs = G (current state, inputs)

# FSM types (con't)

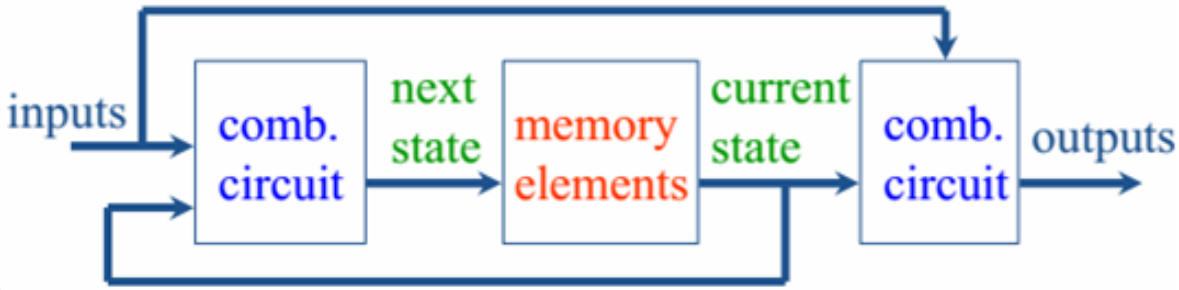
- implement at circuit level

**Moore** type




Next state = F (current state, inputs)
Outputs = G (current state)

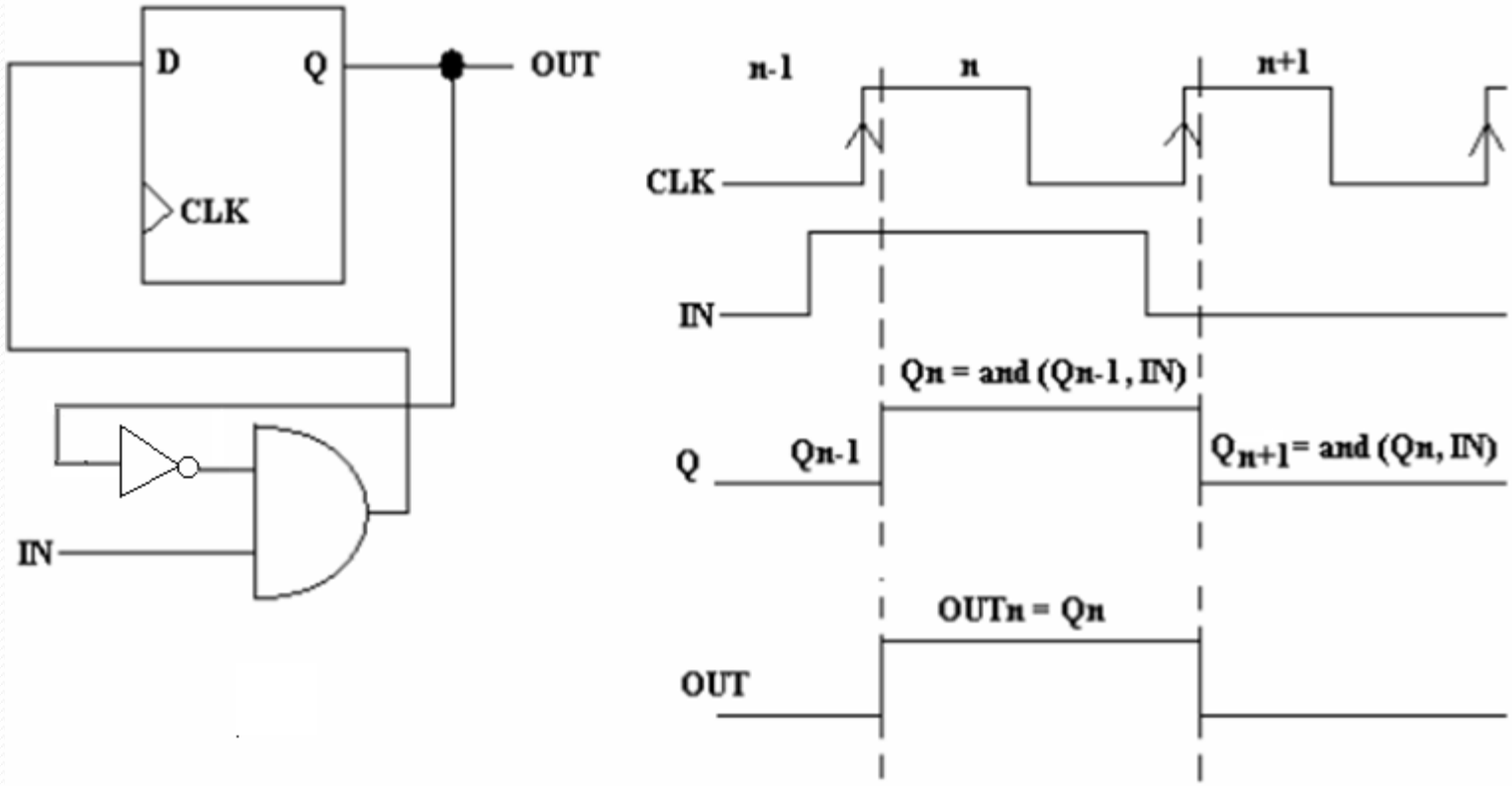
**Mealy** type




Next state = F (current state, inputs)
Outputs = G (current state, inputs)

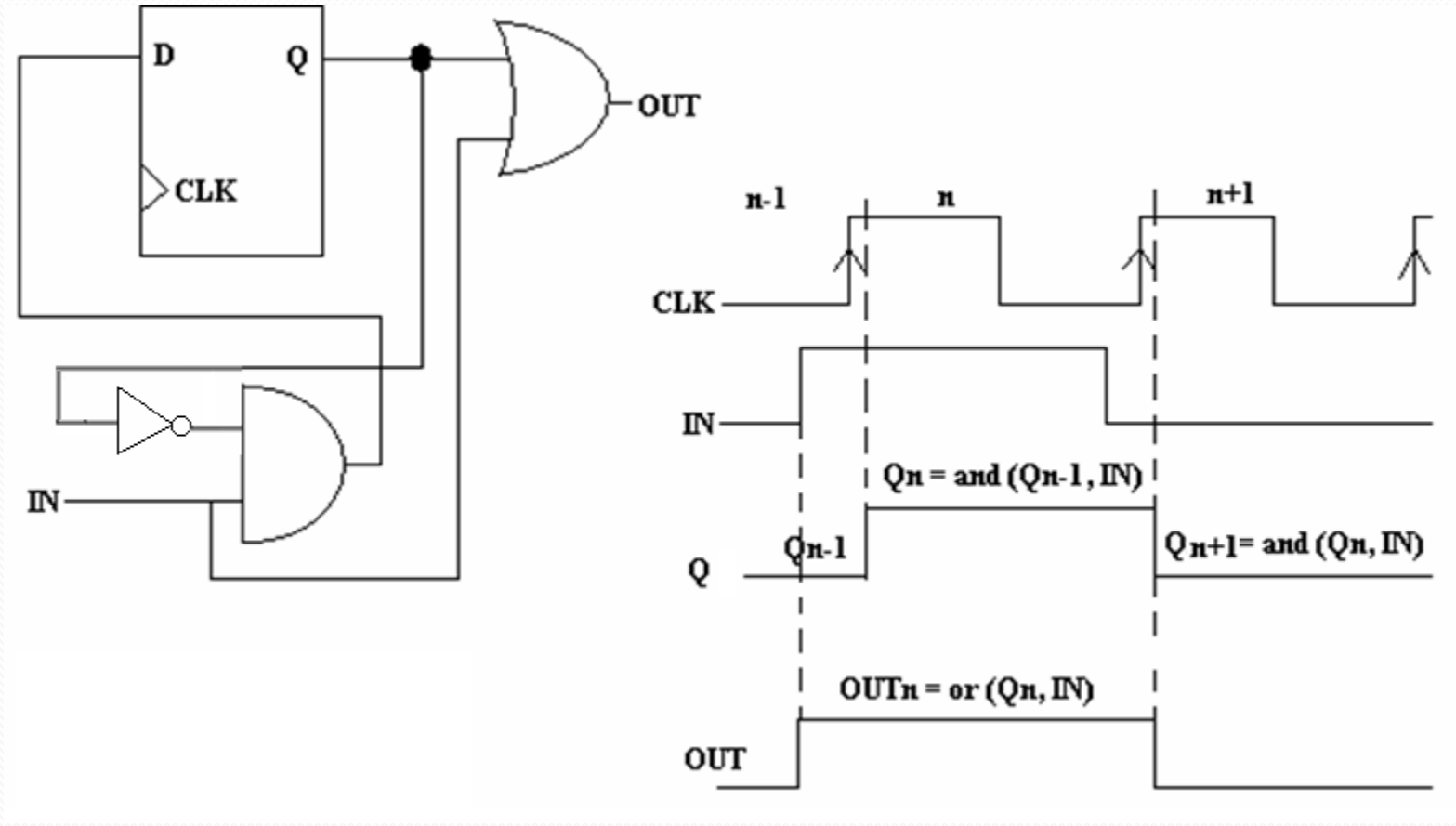
# FSM types (con't)

**Moore** type example



# FSM types (con't)

**Mealy** type example



# Constructing State Machines

- **Current state logic**

  - Register to hold: *always @(posedge/negedge clock)* block

- **Next state logic**

  - State by state case analysis: next state determined by *current state* and given *inputs*

- **Output logic**

  State by state analysis:

  - Moore: output determined by current state only

  - Mealy: output determined by current state and inputs

# Constructing State Machines (con't)

- Model **FSM**:
  - Decide how many states are needed, which transitions are possible from one state to another

    *Using constants declaration like **parameter** or **`define** to define name of states in FSM makes code more readable and easy to manage.*

  - Reset state
  - "always @ (pos/negedge clock)" defines the **current state**
  - *Combinational logic* defines the **next state** and **output** logic

# Constructing State Machines (con't)

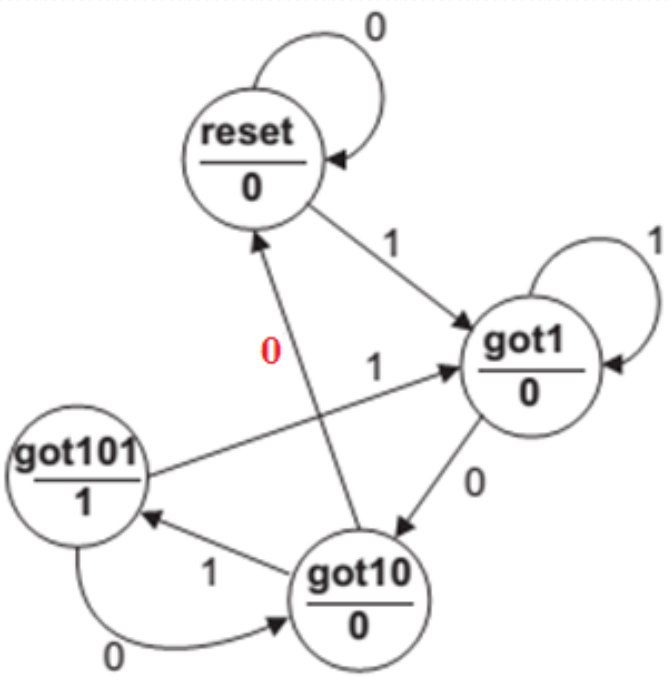
- **Encoding state style**:
  - **Binary** (usually used)
  - **One hot** (usually used)
  - One cold
  - Almost one hot
  - Almost one cold
  - Gray

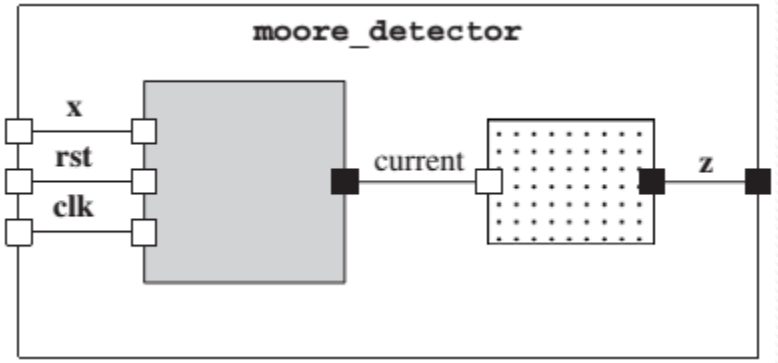
# Constructing State Machines (con't)

- **Encoding state style**:
  - ❖ **Binary encoding**
    - For N states, use ceil ($\log_2 N$) bits to encode the state with each state represented by unique combination of the bits.
    - Tradeoffs: most efficient use of state registers, but requires more complicated combinational logic to detect when in a particular state.
  - ❖ **One-hot encoding**
    - For N states, use **N bits** to encode the state where the bit corresponding to the current state is 1, all the others 0.
    - Tradeoffs: more state registers, but ofter much less combinational logic since state decoding is trivial.

# FSM Coding Style – Simple model

- **Example: 101** detector (**Moore** style)



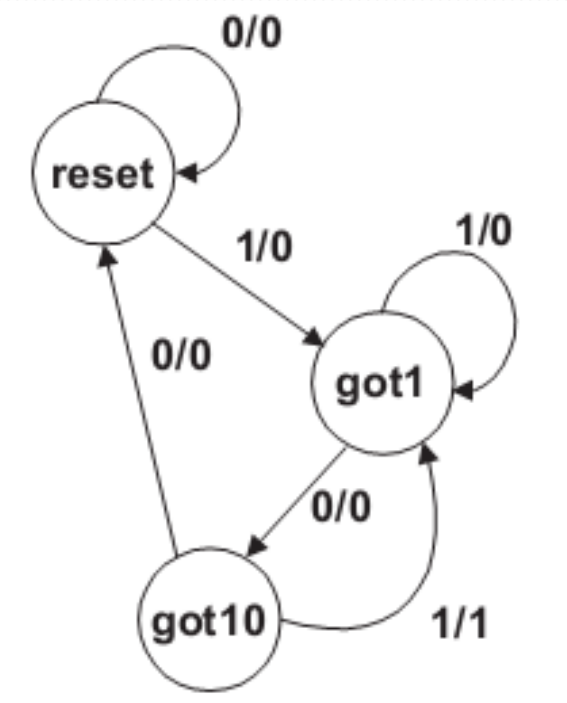
# FSM Coding Style – Simple model

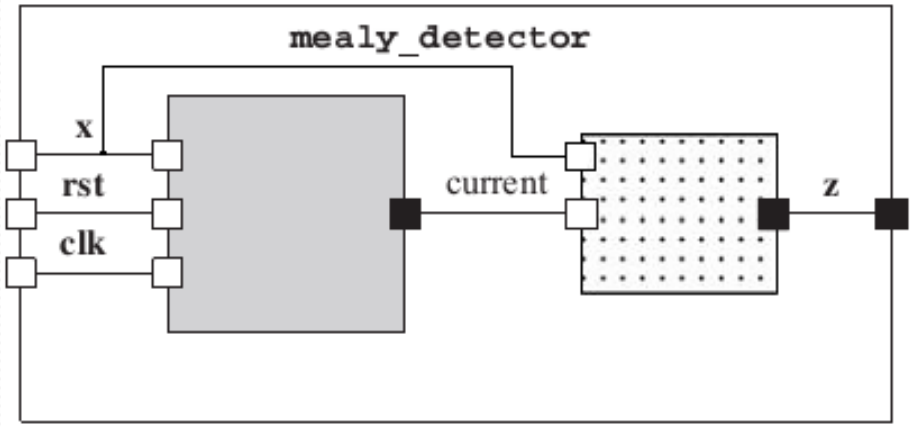
**Example: 101** detector (**Moore** style)

```
module moore_detector (input x, rst, clk, output z);
` define reset      2'b00
` define got1       2'b01
` define got10     2'b10
` define got101   2'b11
reg [1:0] current;
//---Current state and Next state together
always @ (posedge clk )
    if ( rst ) current <= `reset;
    else
        case ( current )
            reset: begin
                    if( x==1'b1 ) current <= `got1;
                    else current <= `reset;
                end
            got1: begin
                    if( x==1'b0 ) current <= `got10;
                    else current <= `got1;
                end
            got10: begin
                    if ( x==1'b1 ) current <= `got101;
                    else current <= `reset;
                end
            got101: begin
                    if (x==1'b1 ) current <= `got1;
                    else current <= `got10;
                end
            default:  current <= `reset;
        endcase
//-------- output ----------
assign z = (current==`got101) ? 1 : 0;
endmodule
```

# FSM Coding Style – Simple model

- **Example: 101** detector (**Mealy** style)



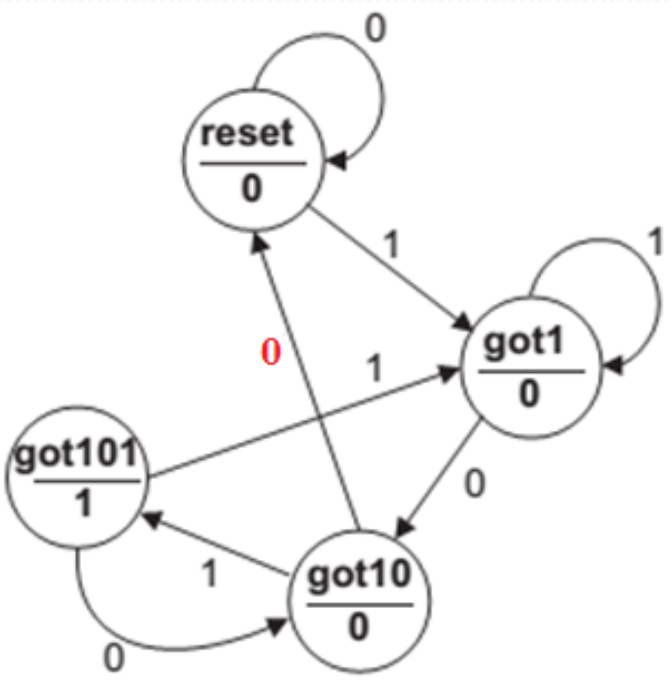
# FSM Coding Style – Simple model

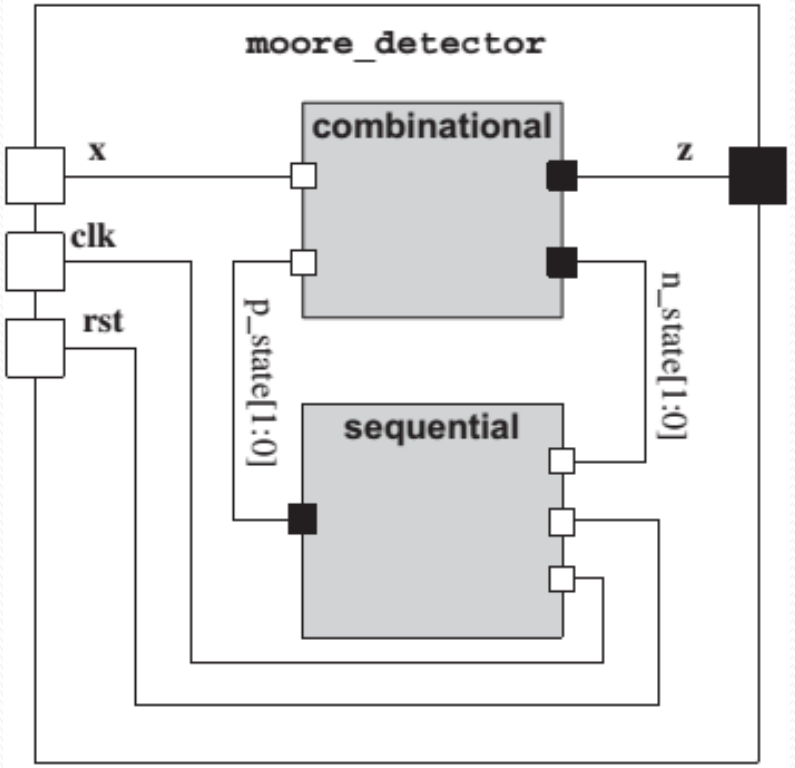
- **Example: 101** detector (**Mealy** style)

```
module mealy_detector (input x, rst, clk, output z);
localparam [1:0] reset = 0, got1 = 1, got10 = 2;
reg [1:0] current;

always @ (posedge clk ) begin
   if (rst) current <= reset;
   else
      case ( current )
            reset:
                  if( x==1'b1 ) current <= got1;
                  else current <= reset;
            got1:
                  if( x==1'b0 ) current <= got10;
                  else current <= got1;
            got10:
                  if( x==1'b1 ) current <= got1;
                  else current <= reset;
            default:
            current <= reset;
      endcase
end

//--------- output ---------------
assign z =
(current==got10 && x==1'b1) ?1'b1 : 1'b0;

endmodule
```

# FSM Coding Style – Huffman model

- **Example: 101** detector (**Moore** style)



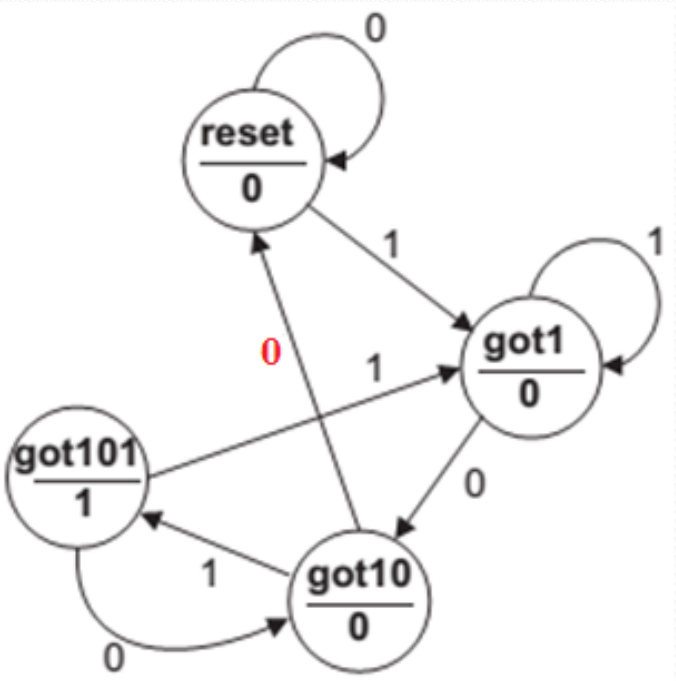
# FSM Coding Style – Huffman model

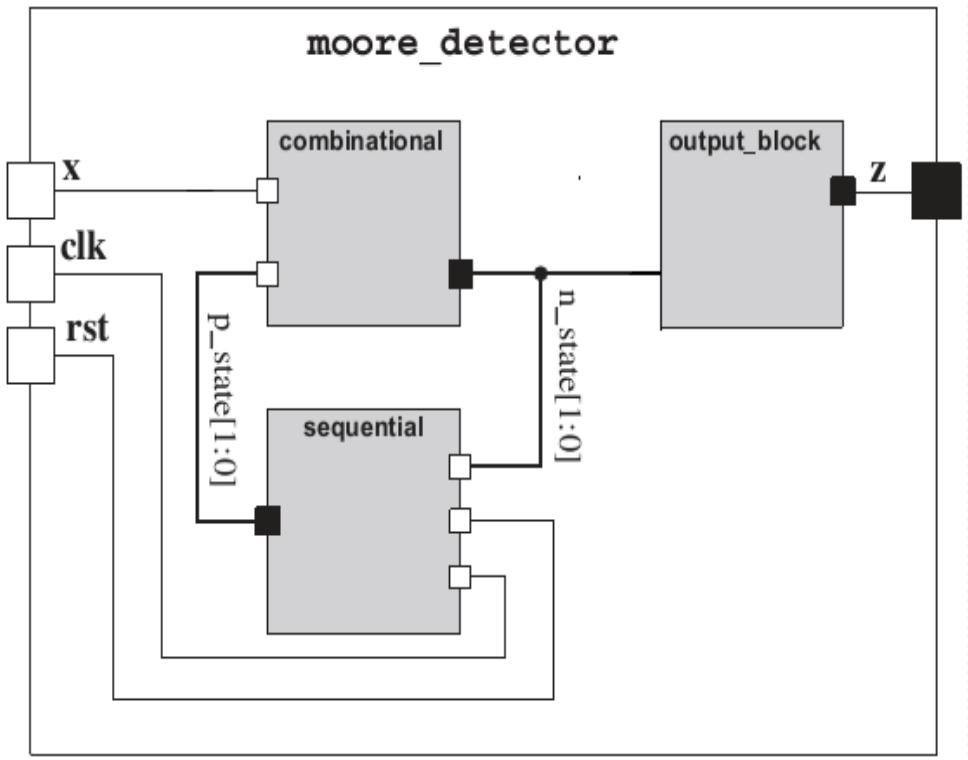
- **Example: 101** detector (**Moore** style)

```
module moore_detector (input x, rst, clk, output reg z);
parameter [1:0] reset=0, got1=1, got10=2, got101=3;
reg [1:0] p_state, n_state;

//---Next state and Output circuit together --------
always @ (*) begin
        case ( p_state )
                reset: begin
                        if( x==1'b1 ) n_state = got1;
                        else n_state = reset;
                        z = 1'b0;
                end
                got1: begin
                        if( x==1'b0 ) n_state = got10;
                        else n_state = got1;
                        z = 1'b0;
                end
                got10: begin
                        if( x==1'b1 ) n_state = got101;
                        else n_state = reset;
                        z = 1'b0;
                end
                got101: begin
                        if( x==1'b1 ) n_state = got1;
                        else n_state = got10;
                        z = 1'b1;
                end
                default: begin
                        n_state = reset;
                        z = 1'b0;
                end
        endcase
end
//--------- current state logic----------------
always@(posedge clk )
        if ( rst )      p_state <= reset;
        else            p_state <= n_state;
endmodule
```

# FSM Coding Style – more modular model

- **Example: 101** detector (**Moore** style)



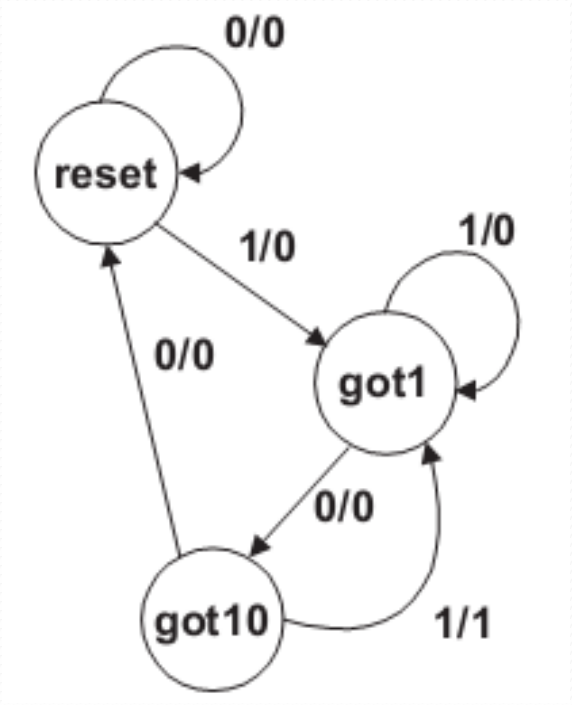
# FSM Coding Style – more modular model

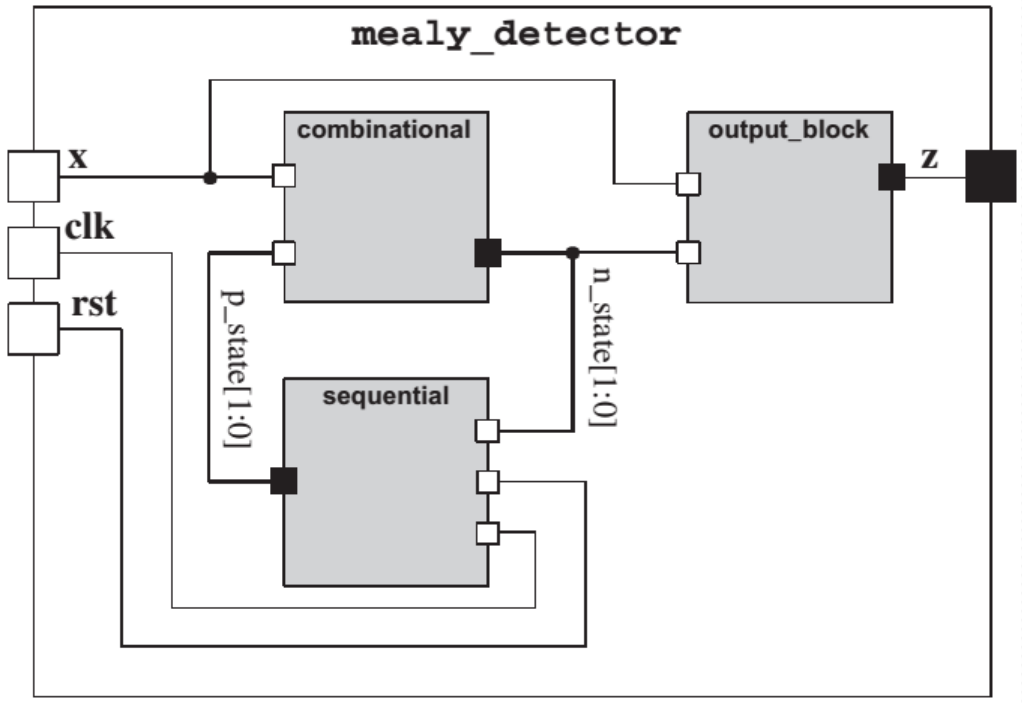
- **Example: 101** detector (**Moore** style)

```
module moore_detector( input x, rst, clk, output z);

localparam [1:0]  reset=0, got1=1, got10=2, got101=3;
reg [1:0] state, next_state;

//-- Combinational Next State Logic ----
always @(*)
  case (state)
    reset:
       if (x == 1'b1) next_state = got1;
       else     next_state = reset;
    got1:
       if (x == 1'b1) next_state = got1;
       else     next_state = got10;
    got10:
       if (x == 1'b1) next_state = got101;
       else next_state = reset;
    got101:
       if (x == 1'b1) next_state = got1;
       else next_state = got10;
    default:
       next_state = reset;
  endcase // case(state)

 //--- State FF Transition (current state)-------
 always @(posedge clock)
   if (reset == 1'b0)
     state <= reset;
   else
     state <= next_state;

 //---- Combinational Output Logic ---
 assign z = (state == got101) ? 1: 0;

 endmodule
```

# FSM Coding Style – more modular model

- **Example: 101** detector (**Mealy** style)



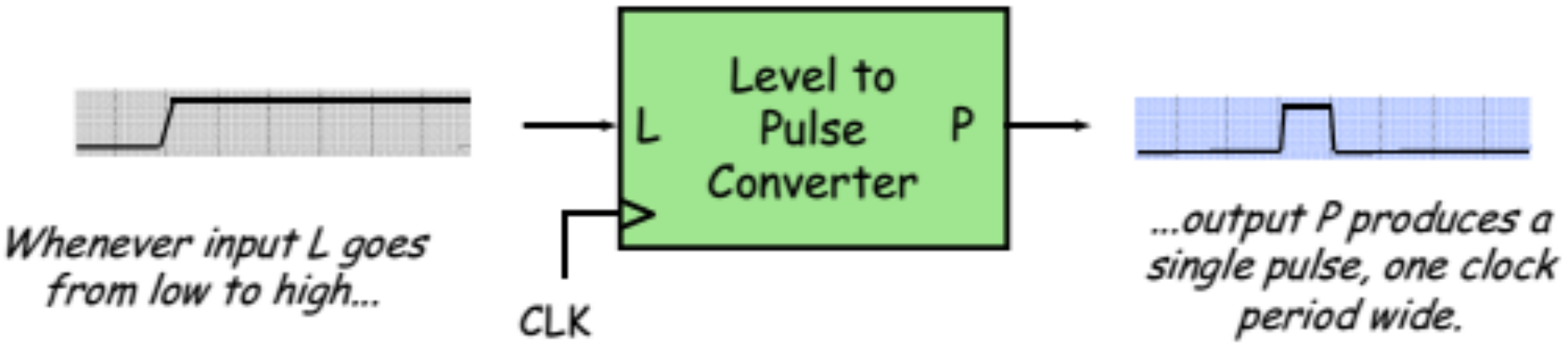
# FSM Coding Style – more modular model

- **Example: 101** detector (**Mealy** style)

```
module moore_detector (input x, rst, clk,
                                    output z);
parameter [1:0] reset=0, got1=1, got10=2;
reg [1:0] state, next_state;

//---Combinational Next State Logic----
always @(*)
  case (state)
    reset:
        if (x) next_state = got1;
        else next_state = reset;
    got1:
        if (x) next_state = got1;
        else next_state = got10;
   got10:
        if (x) next_state = got1;
        else next_state = reset;
    default:
        next_state = reset;
  endcase // case(state)

//----State FF Transition-----
always @(posedge clock)
  if (reset == 1'b0)
    state <= reset;
  else
    state <= next_state;

//---Combinational Output Logic---
assign z =
          (state == got10) && (x == 1);

endmodule
```

# Design Example – Level-to-Pulse

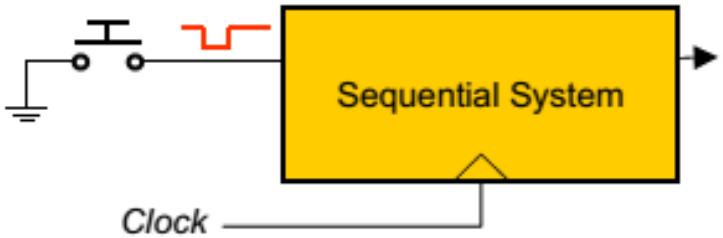
- A level-to-pulse converter produces a single-cycle pulse each time its input goes high.
- It's a synchronous rising-edge detector.
- Sample uses:
  - Buttons and switches pressed by humans for arbitrary periods of time
  - Single-cycle enable signals for counters


*Source: MIT*

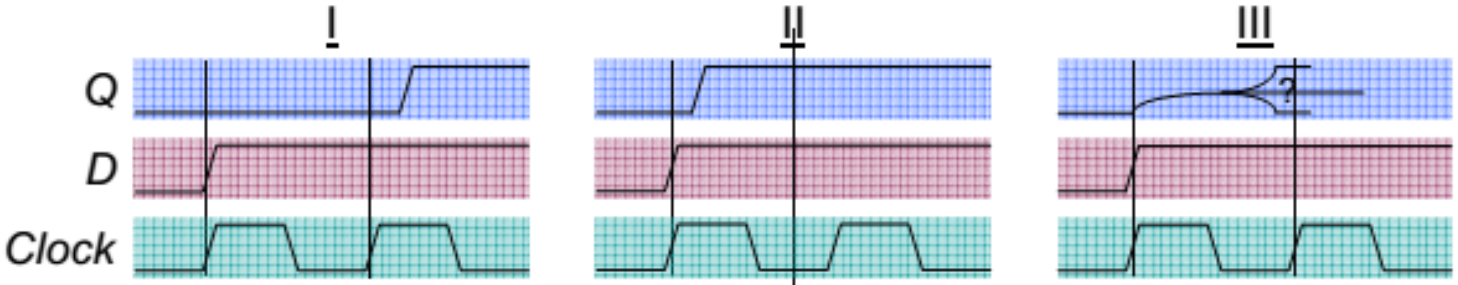
# Asynchronous Inputs in Sequential System

**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.

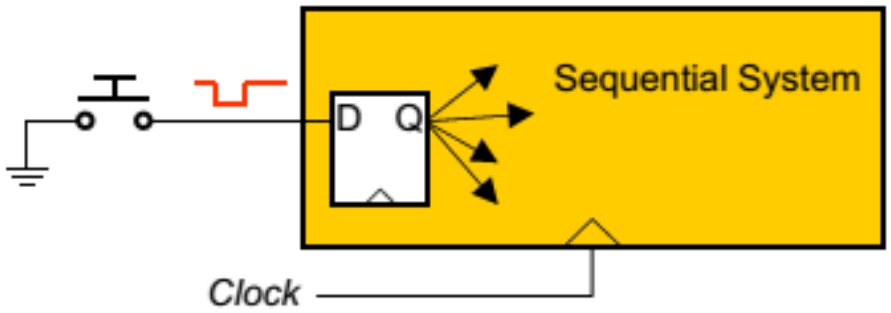
**Q: Which cases are problematic?**

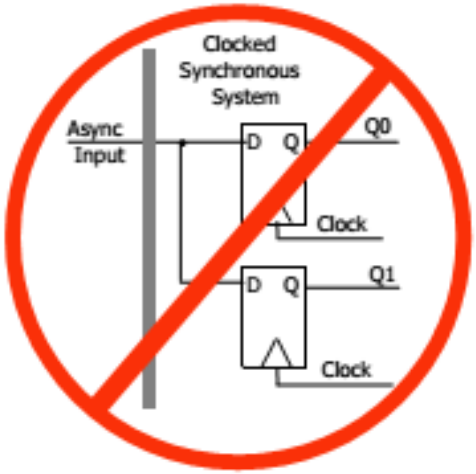
*Source: MIT*

# Asynchronous Inputs in Sequential System

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*



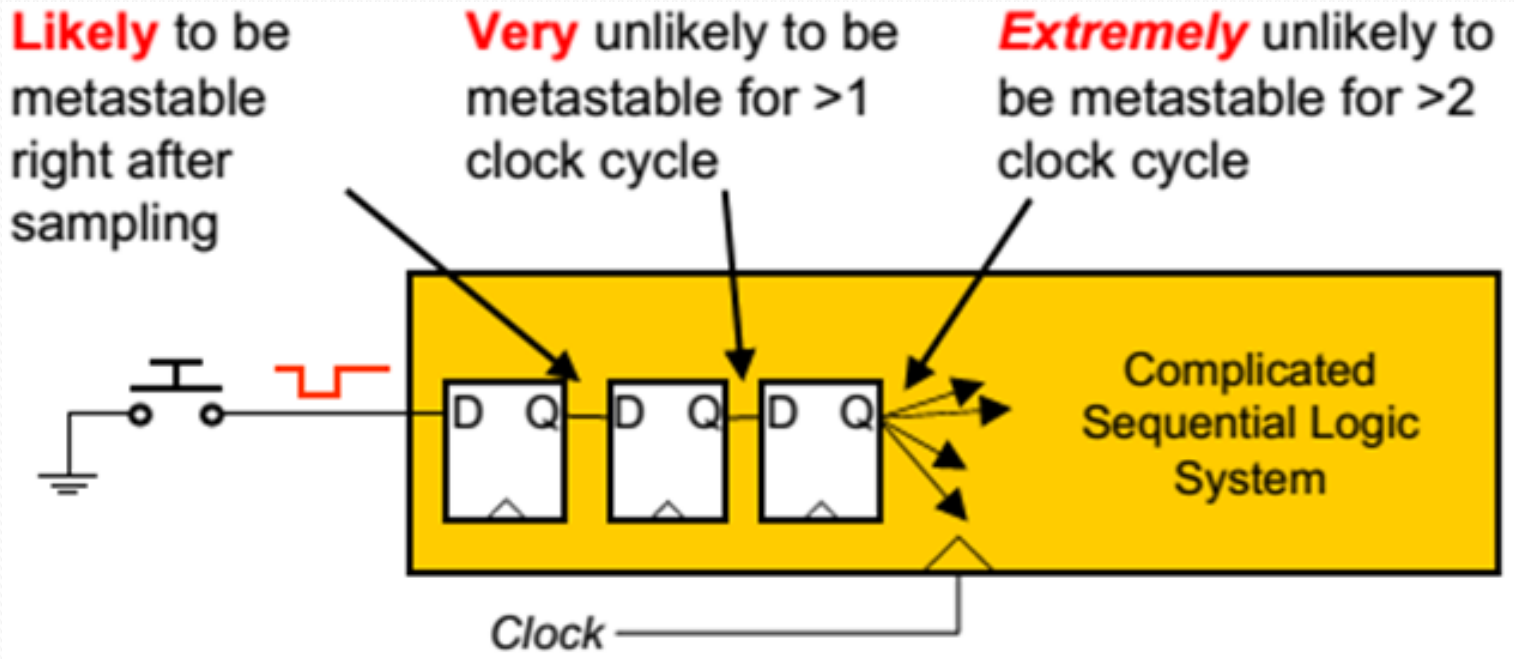




This prevents the possibility of I and II occurring in different places in the circuit, but what about metastability?

*Source: MIT*

# Asynchronous Inputs in Sequential System

- **Preventing metastability turns out to be an impossible problem**
- **Solution to metastability: allow time for signal to stabilize**



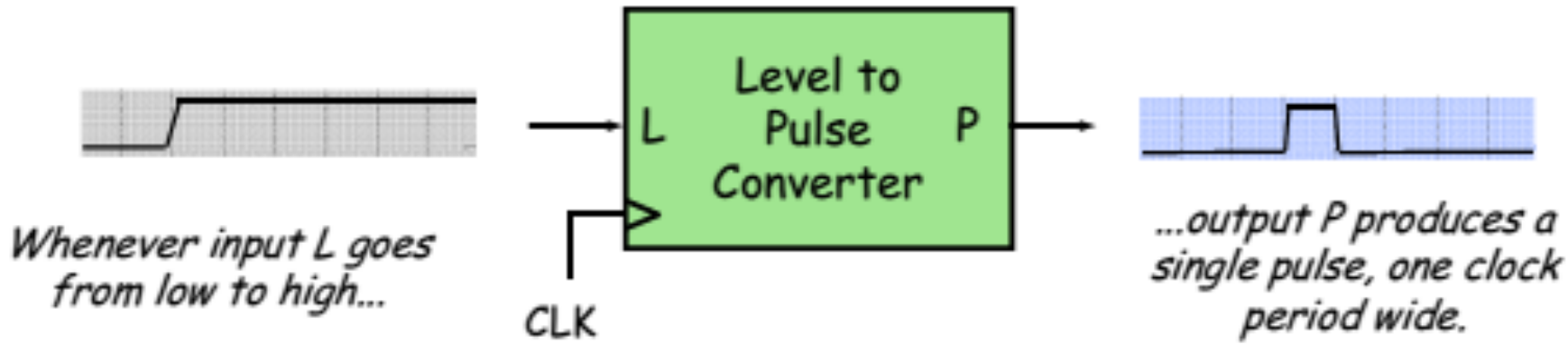

**How many registers are necessary?**

- Depend on many design parameters (clock speed, design speed,…)
- *A pair of synchronization registers is sufficient*

*Source: MIT*

# Design Example – Level-to-Pulse

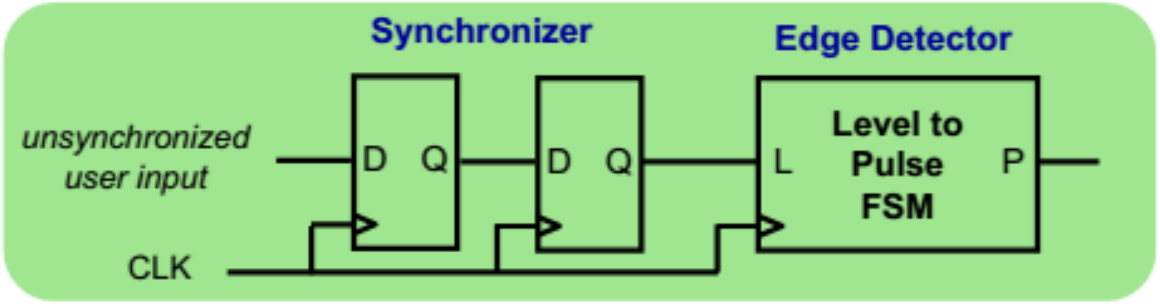
- A level-to-pulse converter produces a single-cycle pulse each time its input goes high.
- It's a synchronous rising-edge detector.
- Sample uses:
  - Buttons and switches pressed by humans for arbitrary periods of time
  - Single-cycle enable signals for counters

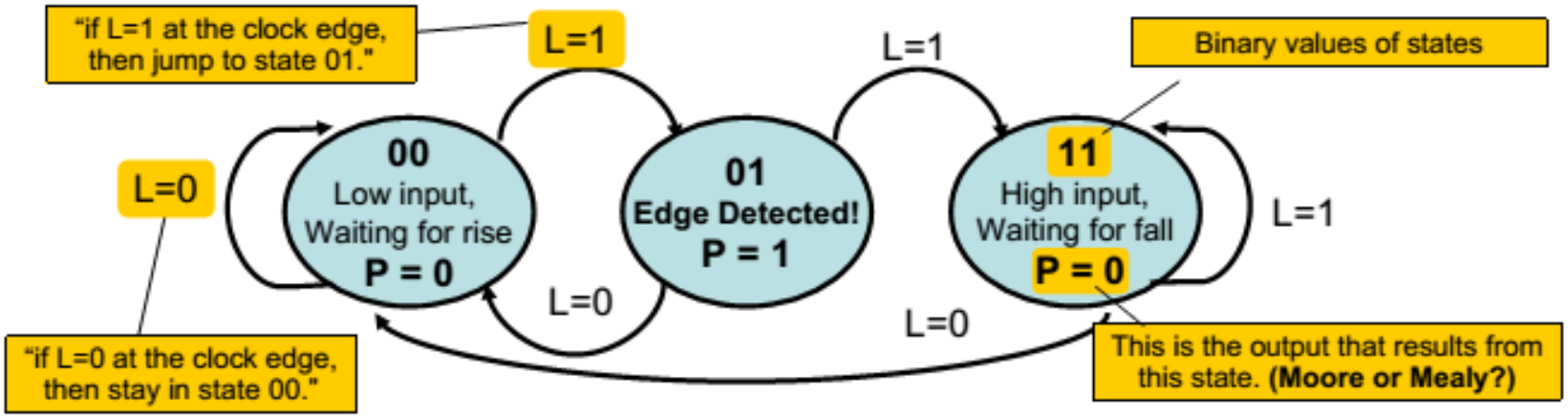

*Source: MIT*

# Design Example (cont.)

- **Block diagram of desired system:**



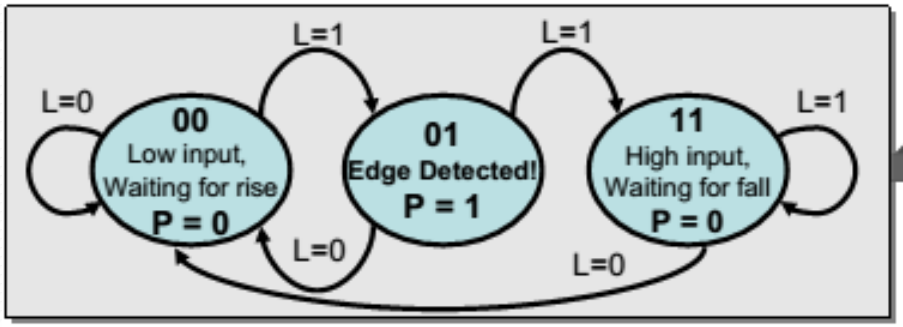

- **State transition diagram is a useful FSM representation and design aid**




*Source: MIT*

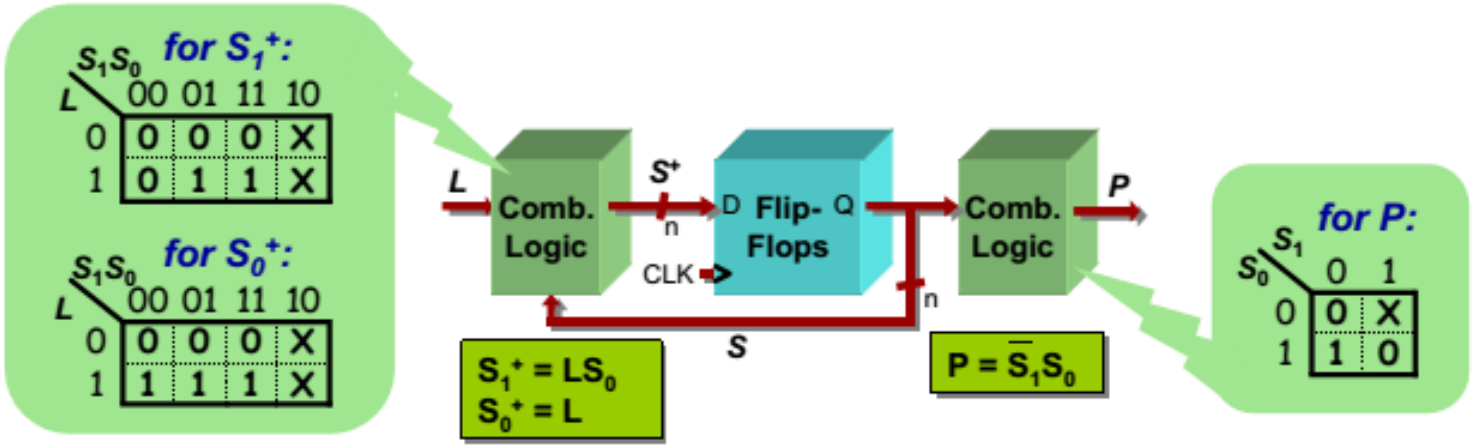
# Design Example (cont.) – **Moore** FSM

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



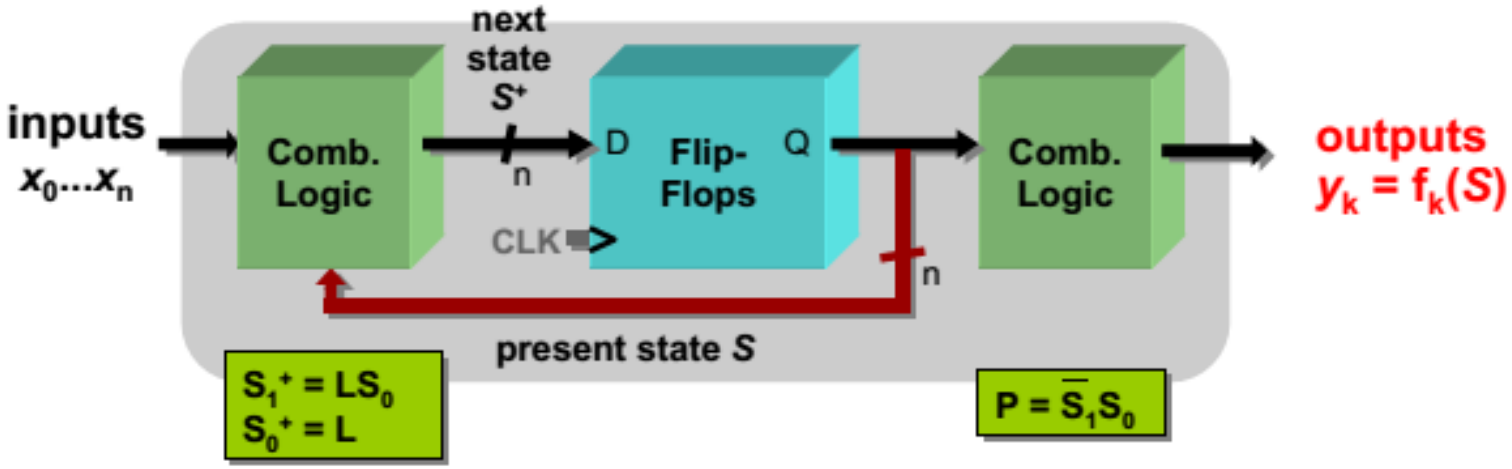

| Current State | | In | Next State | | Out |
|---|---|---|---|---|---|
| $S_1$ | $S_0$ | $L$ | $S_1^+$ | $S_0^+$ | $P$ |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 |

- Combinational logic may be derived using Karnaugh maps

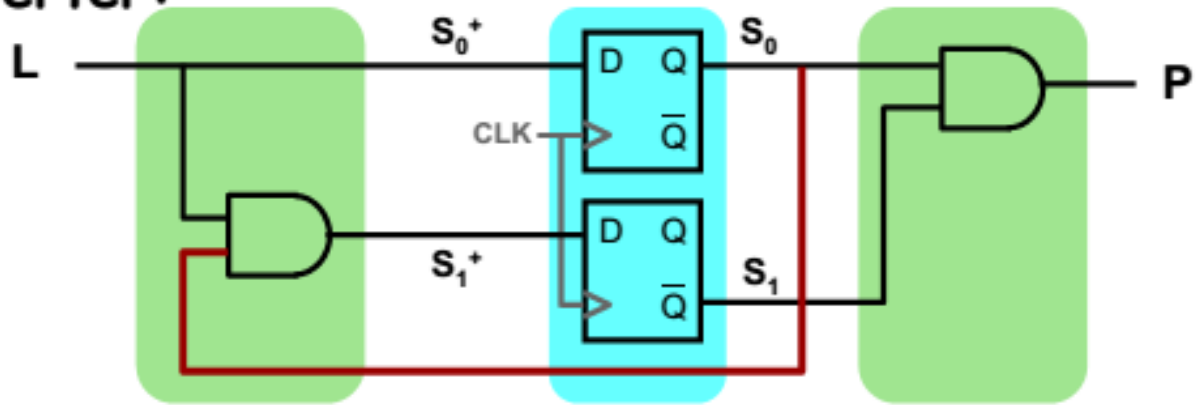



*Source: MIT*

# Design Example (cont.) – **Moore** FSM

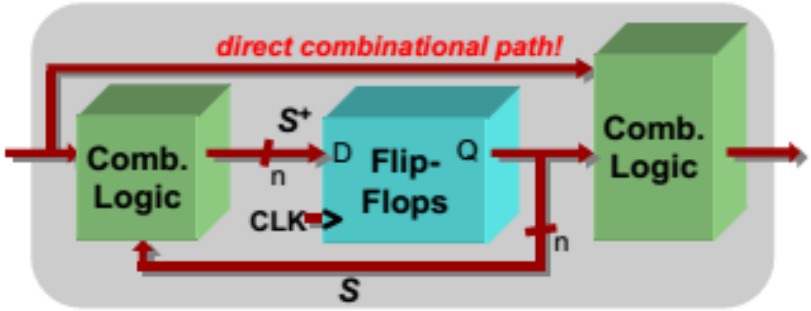



Moore FSM circuit implementation of level-to-pulse converter:

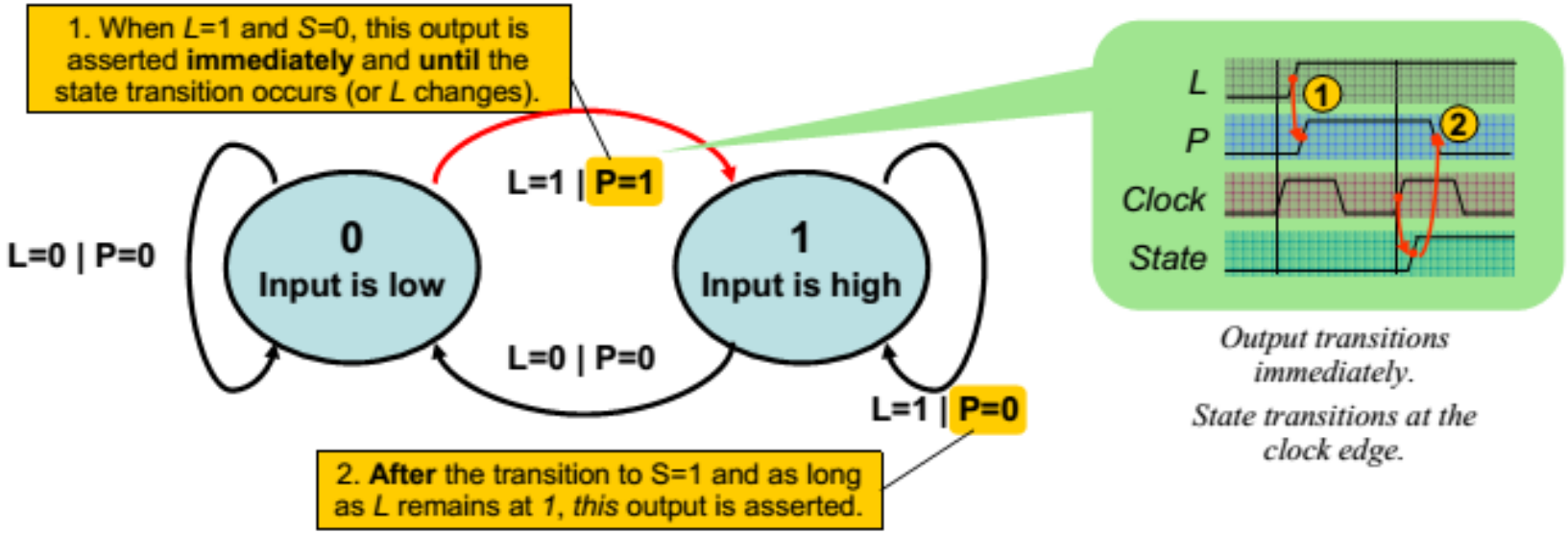



*Source: MIT*

# Design Example (cont.) – **Mealy** FSM

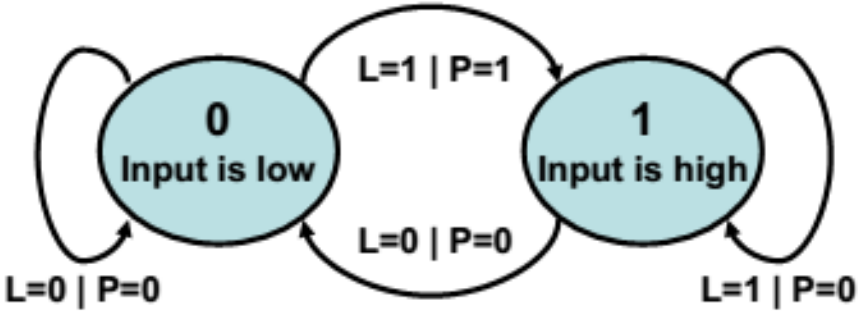



- Since outputs are determined by state *and* inputs, Mealy FSMs may need fewer states than Moore FSM implementations



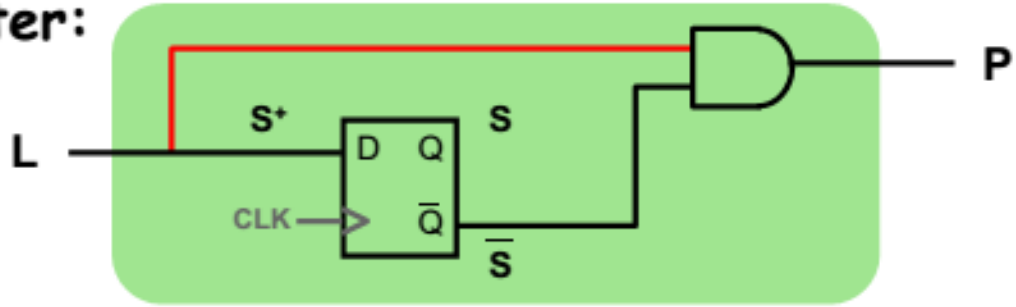

*Source: MIT*

# Design Example (cont.) – **Mealy** FSM




| 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

*Source: MIT*

# Typical Example of FSM

- Objectives:

  - Build an electronic combination lock with a reset button, two number buttons – 0 and 1, and an unlock output.

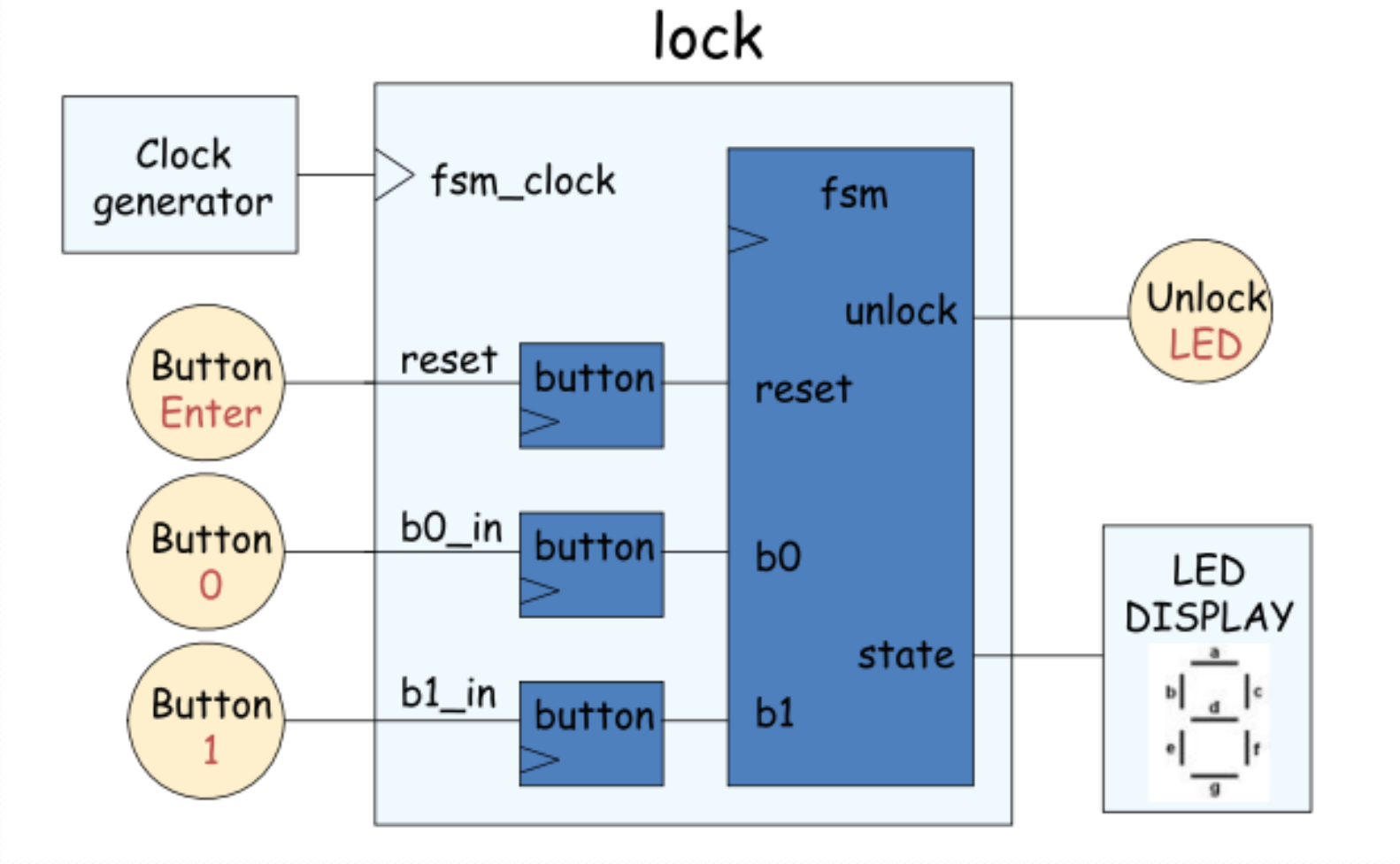
  - The unlock code should be **01011**

- Step:

1. Design lock FSM: block diagram, state transitions
2. Write verilog module(s) for FSM

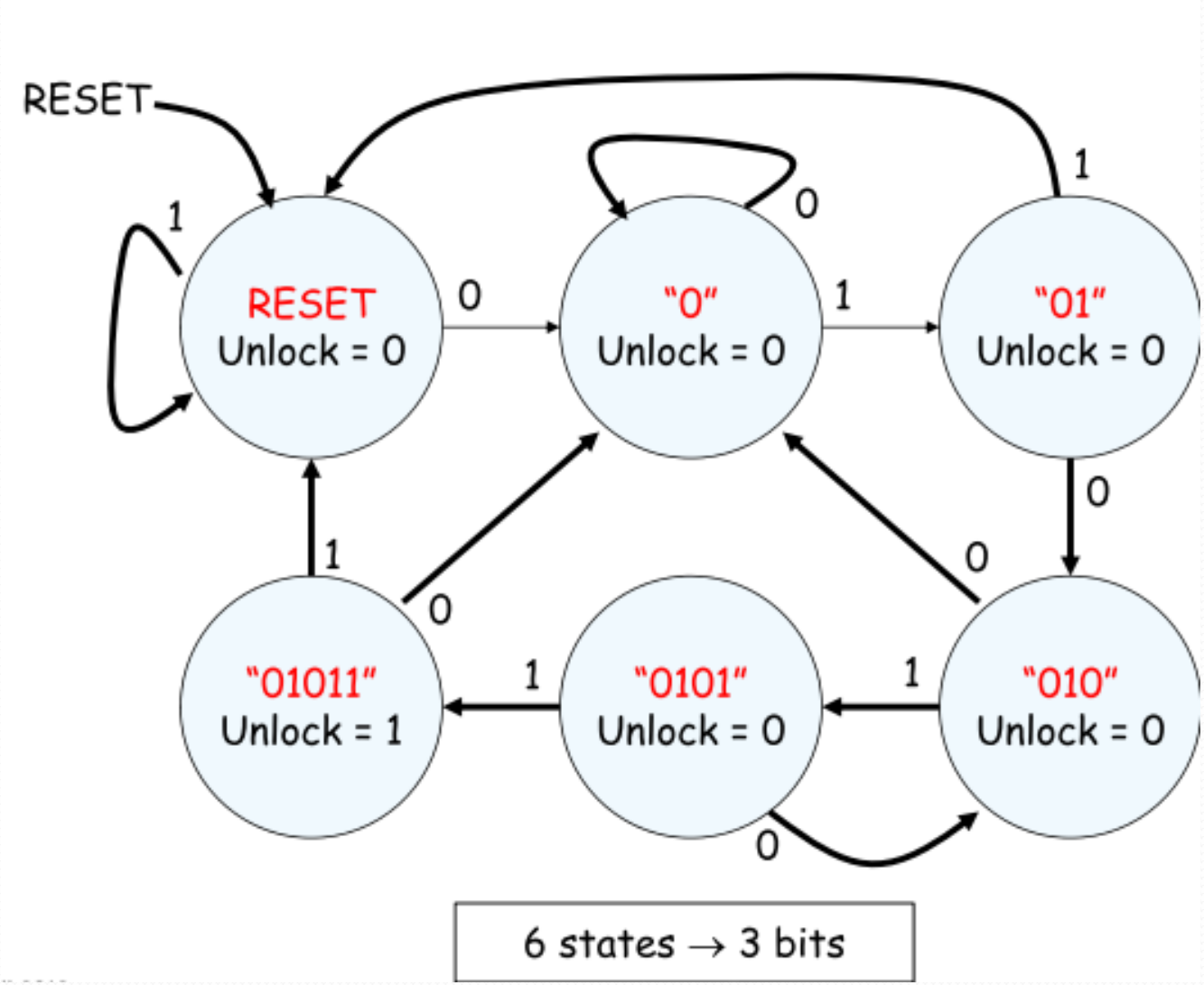

*Source: MIT*

# STEP 1A - Block Diagram




*Source: MIT*

# STEP 1B – State Transition Diagram




*Source: MIT*

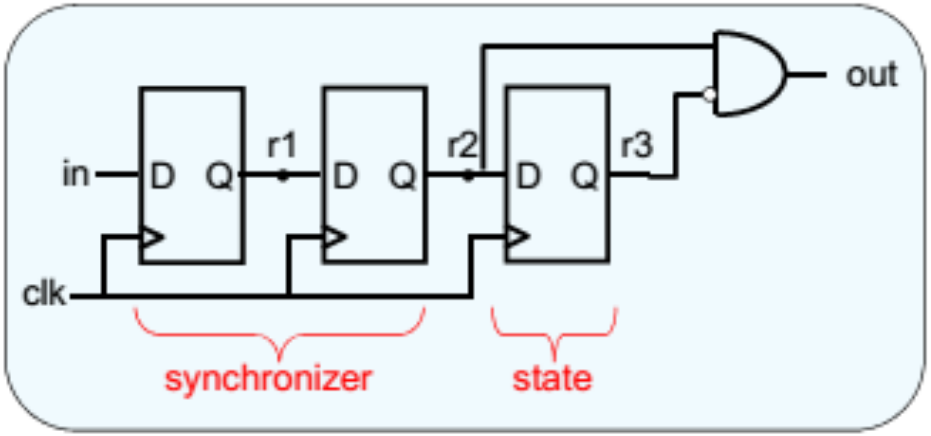
# STEP 2 – Write Verilog

```
module lock(input clk,reset_in,b0_in,b1_in,
            output out);

  // synchronize push buttons, convert to pulses

  // implement state transition diagram
  reg [2:0] state,next_state;
  always @(*) begin
    // combinational logic!
    next_state = ???;
  end

  // current state
  always @(posedge clk) state <= next_state;

  // generate output
  assign out = ???;

  // debugging?
endmodule
```

*Source: MIT*

# STEP 2A – Synchronize Buttons

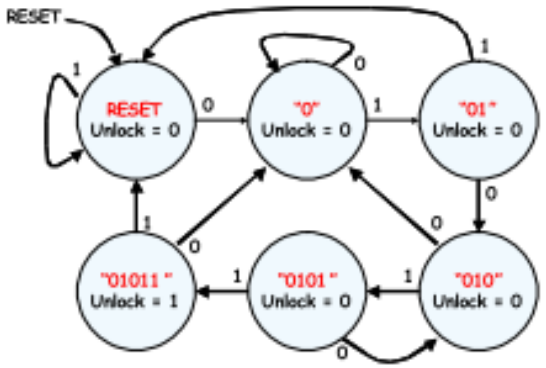
```
// button
// push button synchronizer and level-to-pulse converter
// OUT goes high for one cycle of CLK whenever IN makes a
// low-to-high transition.

module button(
  input clk,in,
  output out
);
  reg r1,r2,r3;
  always @(posedge clk)
  begin
    r1 <= in;    // first reg in synchronizer
    r2 <= r1;    // second reg in synchronizer, output is in sync!
    r3 <= r2;    // remembers previous state of button
  end

  // rising edge = old value is 0, new value is 1
  assign out = ~r3 & r2;
endmodule
```




*Source: MIT*

# STEP 2B – State Transition Diagram and Output

```
parameter S_RESET = 0; // state assignments
parameter S_0 = 1;
parameter S_01 = 2;
parameter S_010 = 3;
parameter S_0101 = 4;
parameter S_01011 = 5;

reg [2:0] state, next_state;
always @(*) begin
  // implement state transition diagram
  if (reset) next_state = S_RESET;
  else case (state)
    S_RESET: next_state = b0 ? S_0    : b1 ? S_RESET : state;
    S_0:     next_state = b0 ? S_0    : b1 ? S_01    : state;
    S_01:    next_state = b0 ? S_010  : b1 ? S_RESET : state;
    S_010:   next_state = b0 ? S_0    : b1 ? S_0101  : state;
    S_0101:  next_state = b0 ? S_010  : b1 ? S_01011 : state;
    S_01011: next_state = b0 ? S_0    : b1 ? S_RESET : state;
    default: next_state = S_RESET;  // handle unused states
  endcase
end

always @(posedge clk) state <= next_state;
```




```
// it's a Moore machine!
assign out = (state == S_01011);
```

*Source: MIT*

# STEP 2 – Final Implementation Verilog

```
module lock(input clk,reset_in,b0_in,b1_in,
            output out );

  wire reset, b0, b1; // synchronize push buttons, convert to pulses
  button b_reset(clk,reset_in,reset);
  button b_0(clk,b0_in,b0);
  button b_1(clk,b1_in,b1);

  parameter S_RESET = 0; parameter S_0 = 1; // state assignments
  parameter S_01 = 2;  parameter S_010 = 3;
  parameter S_0101 = 4;  parameter S_01011 = 5;

  reg [2:0] state,next_state;
  always @(*) begin                          // implement state transition diagram
    if (reset) next_state = S_RESET;
    else case (state)
      S_RESET: next_state = b0 ? S_0    : b1 ? S_RESET : state;
      S_0:     next_state = b0 ? S_0    : b1 ? S_01    : state;
      S_01:    next_state = b0 ? S_010  : b1 ? S_RESET : state;
      S_010:   next_state = b0 ? S_0    : b1 ? S_0101  : state;
      S_0101:  next_state = b0 ? S_010  : b1 ? S_01011 : state;
      S_01011: next_state = b0 ? S_0    : b1 ? S_RESET : state;
      default: next_state = S_RESET;     // handle unused states
    endcase
  end
  always @ (posedge clk) state <= next_state;

  assign out = (state == S_01011);   // assign output: Moore machine
endmodule
```

*Source: MIT*

# Real FSM Security System


*Source: MIT*

# Tips on FSM

- Don't forget to handle the "**default**" for **case** statement
- Use *two different* ***always*** *blocks* for *next state* and *current state* circuit

  Can do it in one big block but not as clear
- Outputs can be a mix of combin. and seq.

  **Moore** machine: Output only depends on current state

  **Mealy** machine: Output only depends on current state and inputs