EECS 151/251A
Spring 2023
Digital Design and Integrated Circuits

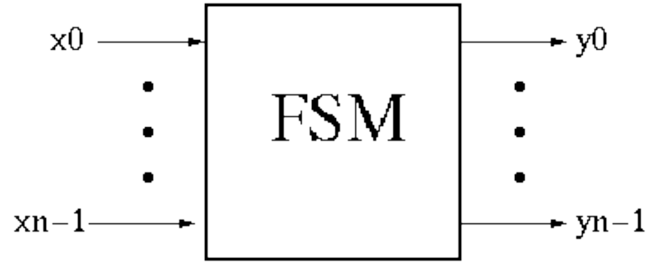
Instructor:
John Wawrzynek

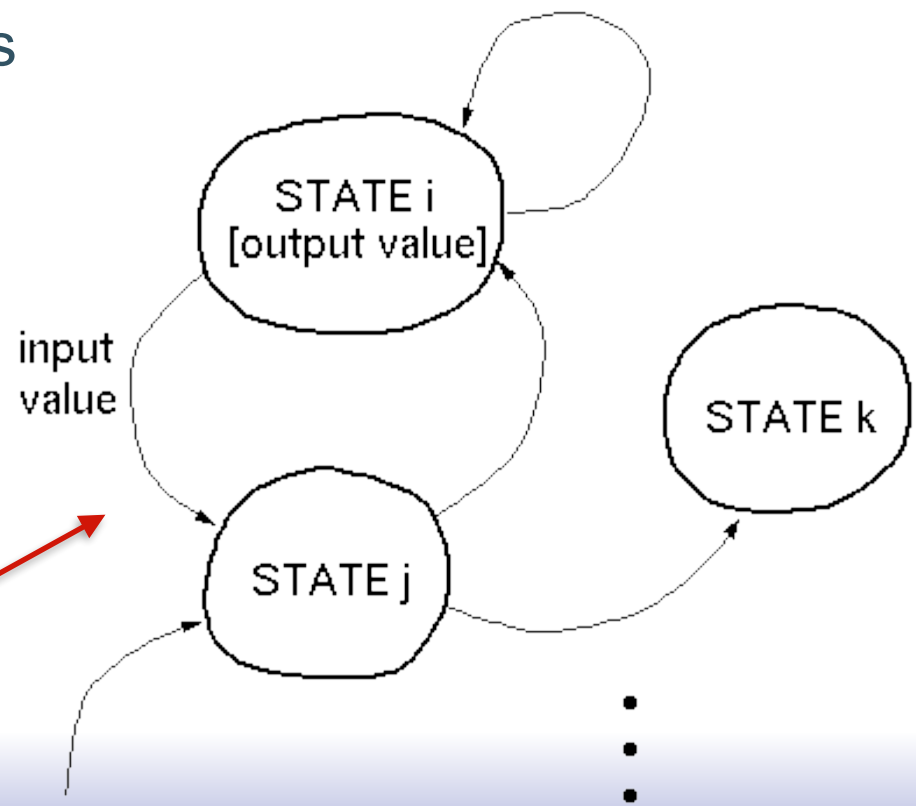
Lecture 8: Finite State Machines part 2

Finite State Machines continued

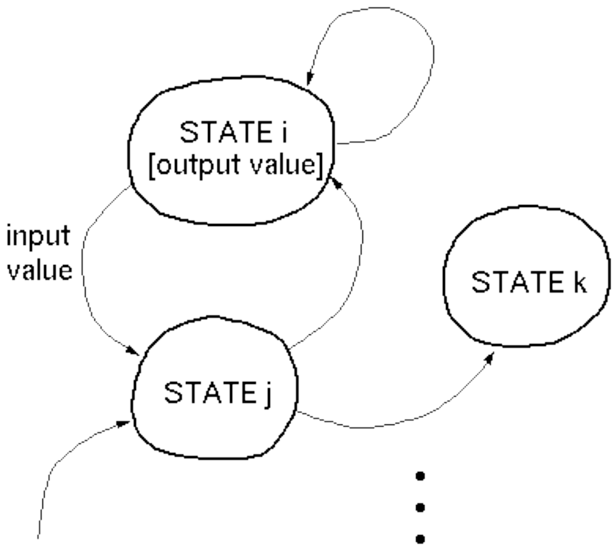
Finite State Machines (FSMs)

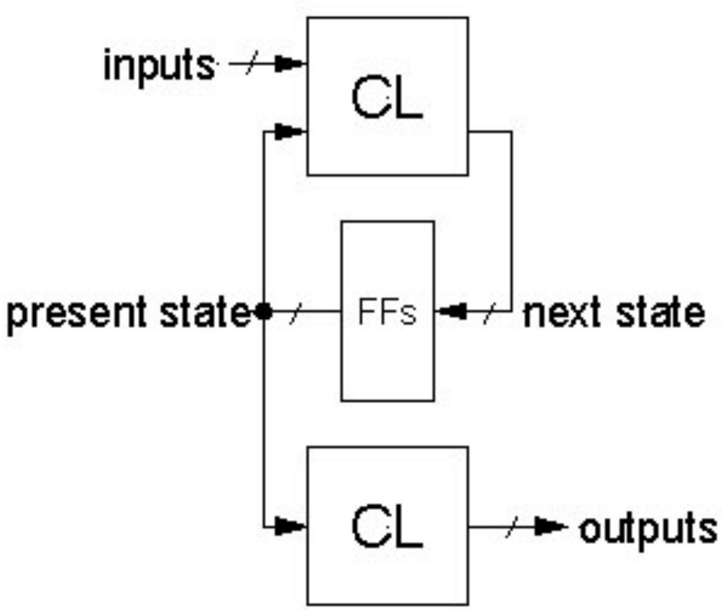
- **FSMs:**
  - Can model behavior of *any sequential circuit*
  - Useful representation for designing sequential circuits
  - As with all sequential circuits: output depends on present *and* past inputs
    - effect of past inputs represented by the current *state*
- Behavior is represented by ***State Transition Diagram***:
  - traverse one edge per clock cycle.





FSM Implementation





- Flip-flops form *state register*
- number of states ≤ $2^{\text{number of flip-flops}}$
- CL (combinational logic) calculates next state and output
- Remember: The FSM follows exactly one edge per cycle.

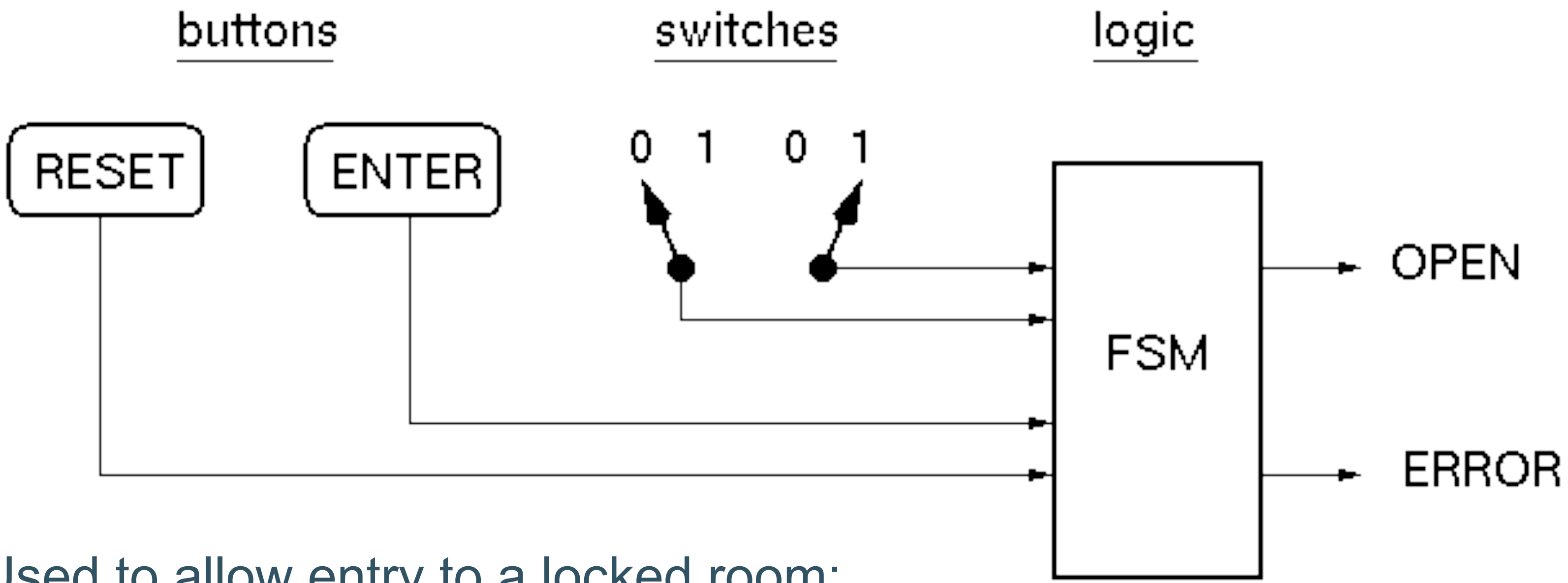
Later we will learn how to implement in Verilog. Now we learn how to design "by hand" to the gate level.

"Formal" By-hand Design Process

Review of Design Steps:

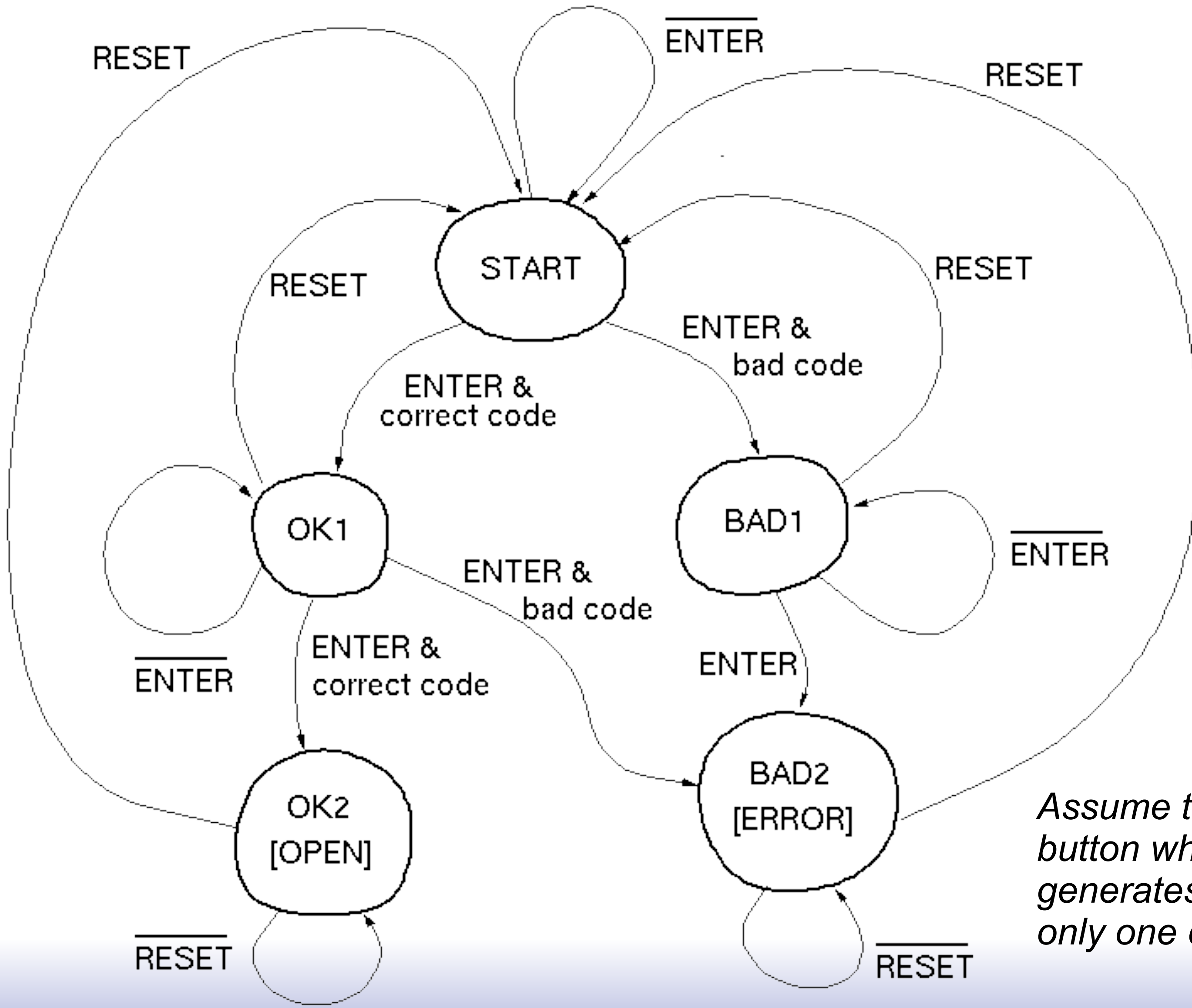
1. *Specify **circuit function** (English)*
2. *Draw **state transition diagram***
3. *Write down **symbolic state transition table***
4. *Write down **encoded state transition table***
5. *Derive **logic equations***
6. *Derive **circuit diagram***
   - *Register to hold state*
   - *Combinational Logic for Next State and Outputs*

Combination Lock Example



- Used to allow entry to a locked room:
  - 2-bit serial combination. Example 01,11:
    1. Set switches to 01, press ENTER
    2. Set switches to 11, press ENTER
    3. OPEN is asserted (OPEN=1).
       - If wrong code, ERROR is asserted (after second combo word entry).
       - Press Reset at anytime to try again.

Combinational Lock STD

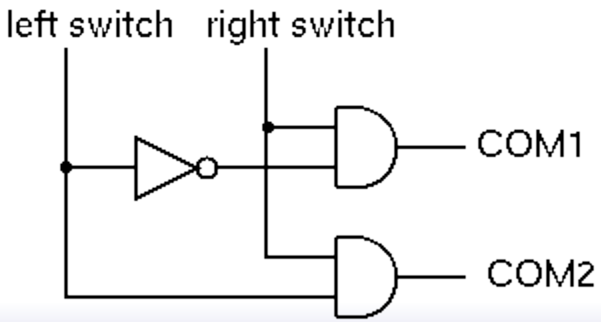


Assume the ENTER button when pressed generates a pulse for only one clock cycle.

Symbolic State Transition Table

RESET	ENTER	COM1	COM2	Preset State	Next State	OPEN	ERROR
0	0	*	*	START	START	0	0
0	1	0	*	START	BAD1	0	0
0	1	1	*	START	OK1	0	0
0	0	*	*	OK1	OK1	0	0
0	1	*	0	OK1	BAD2	0	0
0	1	*	1	OK1	OK2	0	0
0	*	*	*	OK2	OK2	1	0
0	0	*	*	BAD1	BAD1	0	0
0	1	*	*	BAD1	BAD2	0	0
0	*	*	*	BAD2	BAD2	0	1
1	*	*	*	*	START	0	0

Decoder logic for checking combination (01,11):



** represents "wild card" - expands to all combinations*

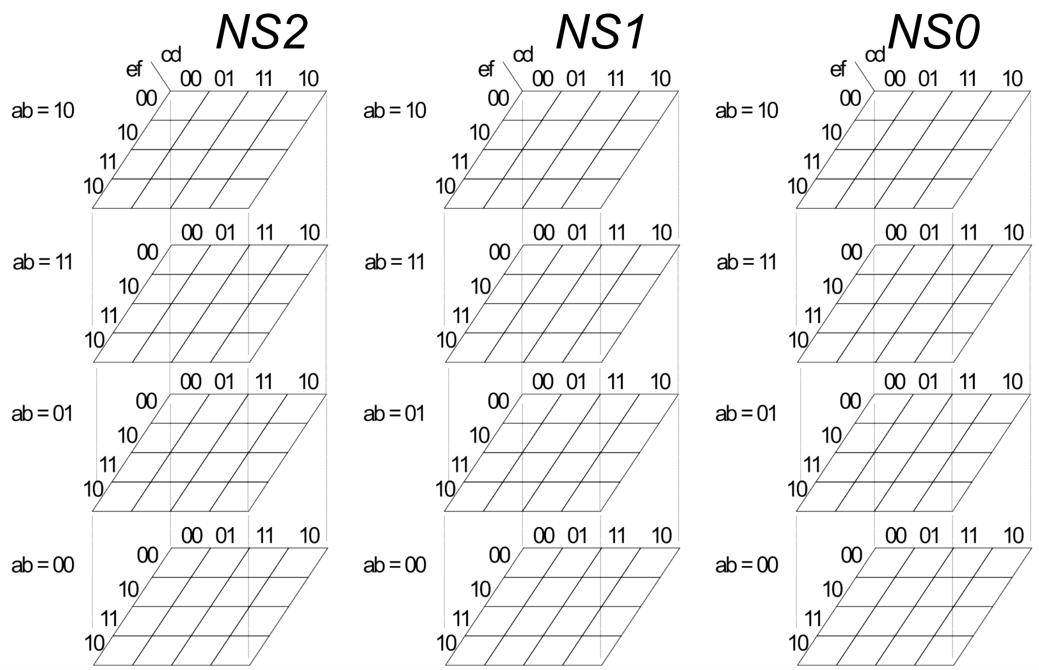
Encoded ST Table

ENTER	COM1	COM2	PS2	PS1	PS0	NS2	NS1	NS0
0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0
0	1	1	0	0	0	0	0	0
1	0	0	0	0	0	1	0	0
1	0	1	0	0	0	1	0	0
1	1	0	0	0	0	0	0	1
1	1	1	0	0	0	0	0	1
0	0	0	0	0	1	0	0	1
0	0	1	0	0	1	0	0	1
0	1	0	0	0	1	0	0	1
0	1	1	0	0	1	0	0	1
1	0	0	0	0	1	1	0	1
1	1	0	0	0	1	1	0	1
1	0	1	0	0	1	0	1	1
1	1	1	0	0	1	0	1	1
0	0	0	0	1	1	0	1	1
0	0	1	0	1	1	0	1	1
0	1	0	0	1	1	0	1	1
0	1	1	0	1	1	0	1	1
1	0	0	0	1	1	0	1	1
1	0	1	0	1	1	0	1	1
1	1	0	0	1	1	0	1	1
1	1	1	0	1	1	0	1	1
0	0	0	1	0	0	1	0	0
0	0	1	1	0	0	1	0	0
0	1	0	1	0	0	1	0	0
0	1	1	1	0	0	1	0	0
1	0	0	1	0	0	1	0	1
1	0	1	1	0	0	1	0	1
1	1	0	1	0	0	1	0	1
1	1	1	1	0	0	1	0	1
0	0	0	1	0	1	1	0	1
0	0	1	1	0	1	1	0	1
0	1	0	1	0	1	1	0	1
0	1	1	1	0	1	1	0	1
1	0	0	1	0	1	1	0	1
1	0	1	1	0	1	1	0	1
1	1	0	1	0	1	1	0	1
1	1	1	1	0	1	1	0	1

- *Assign states:*

START=000, OK1=001, OK2=011
BAD1=100, BAD2=101

- *Omit reset. Assume that primitive flip-flops has reset input.*
- *Rows not shown have don't cares in output, corresponding to invalid PS values.*

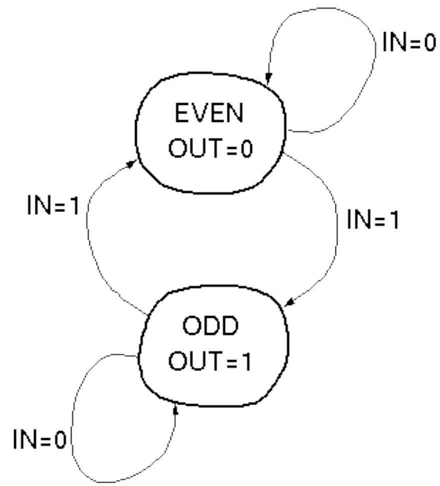


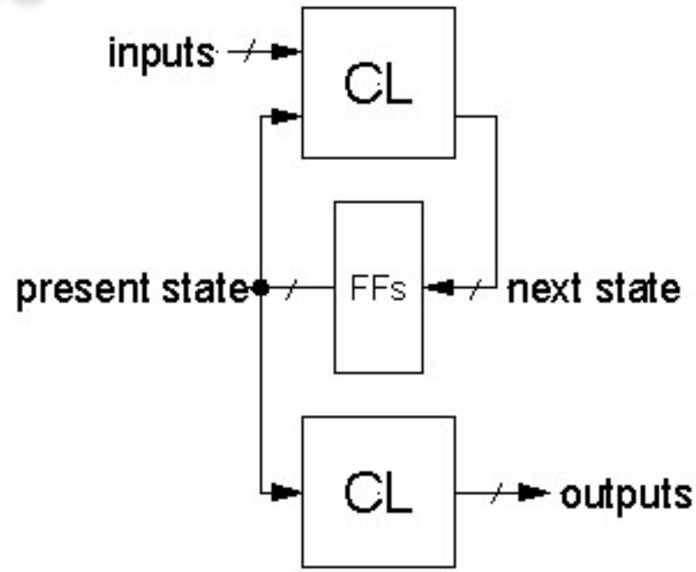
- *What are the output functions for OPEN and ERROR?*

Moore Versus Mealy Machines

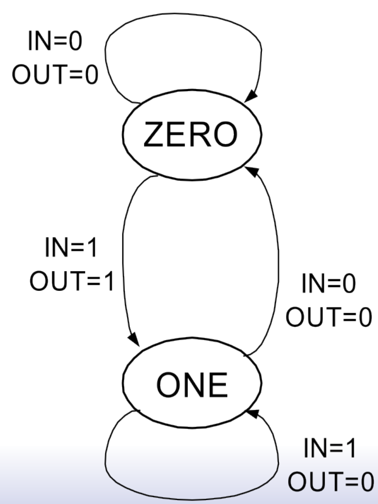
FSM Implementation Notes

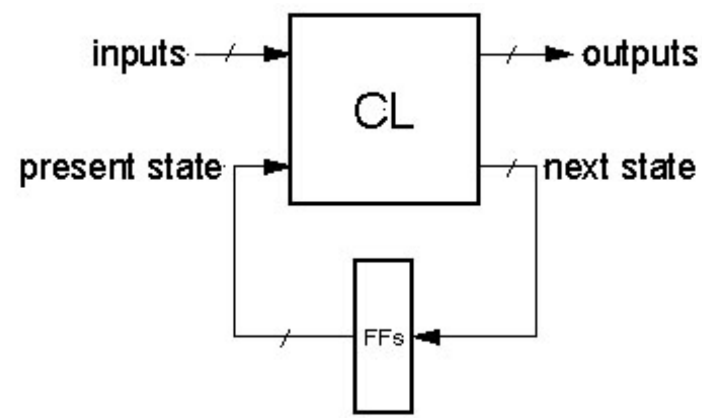
- All examples so far generate output based only on the present state, commonly called a "*Moore Machine*":





- If output functions include both present state and input then called a "*Mealy Machine*":

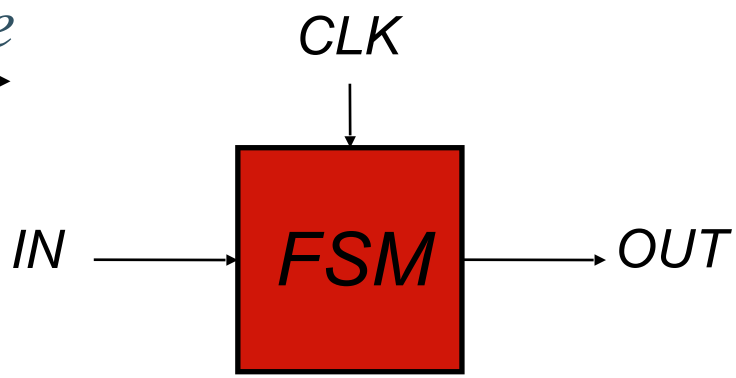




Finite State Machines

- **Example: Edge Detector**

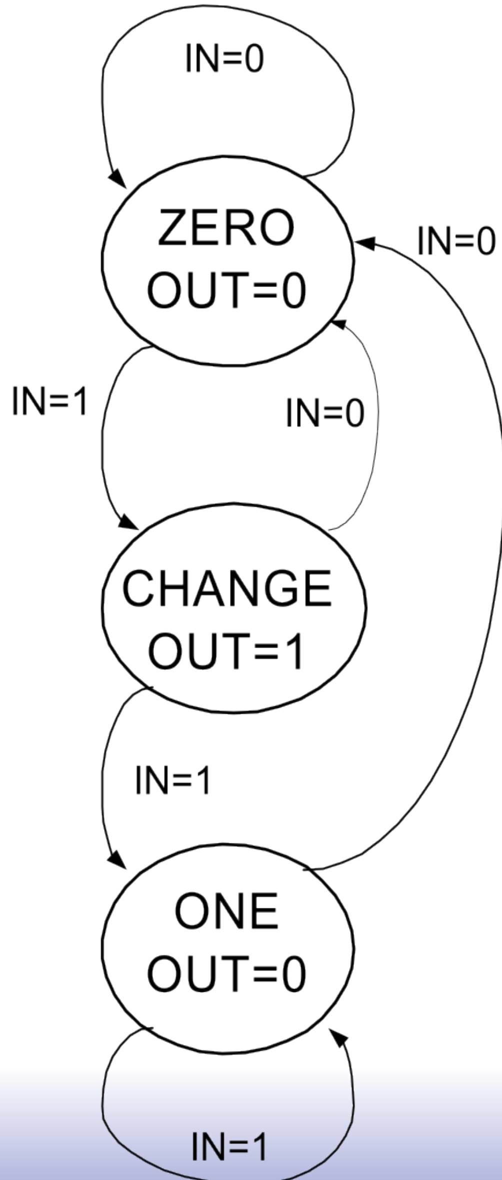
  Bit are received one at a time (one per cycle), such as: 000111010 *time* →



  Design a circuit that asserts its output for one cycle when the input bit stream changes from 0 to 1.

  We'll try two different solutions: Moore then Mealy.

State Transition Diagram Solution A - Moore



	IN	PS	NS	OUT
ZERO	0	00	00	0
	1	00	01	0
CHANGE	0	01	00	1
	1	01	11	1
ONE	0	11	00	0
	1	11	11	0

Solution A, circuit derivation

	IN	PS	NS	OUT
ZERO	0	00	00	0
	1	00	01	0
CHANGE	0	01	00	1
	1	01	11	1
ONE	0	11	00	0
	1	11	11	0

IN \ PS	00	01	11	10
0	0	0	0	-
1	0	1	1	-

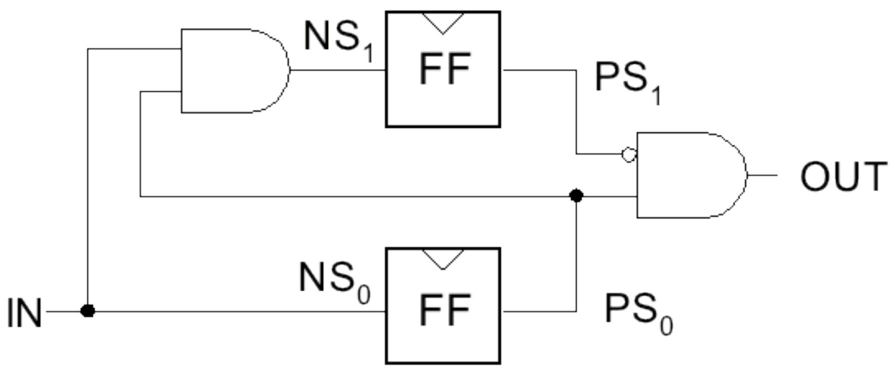
$NS_1 = IN\ PS_0$

IN \ PS	00	01	11	10
0	0	0	0	-
1	1	1	1	-

$NS_0 = IN$

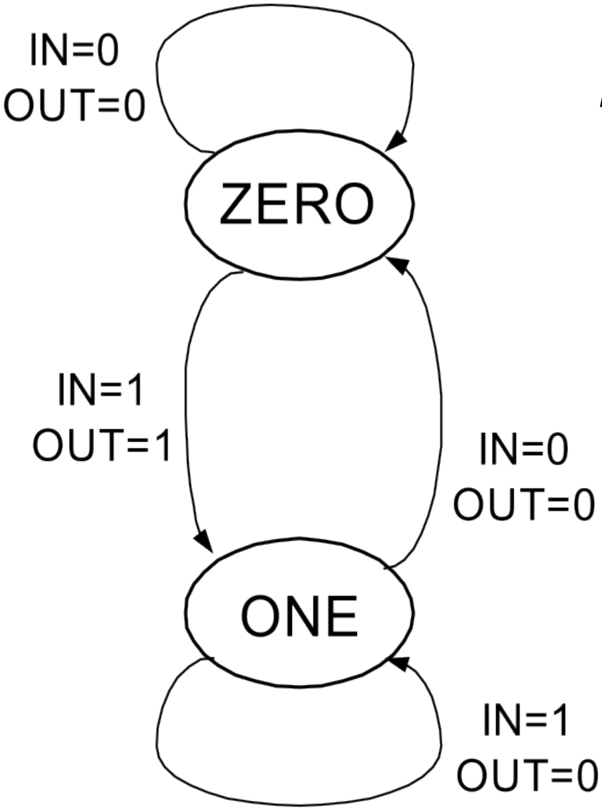
IN \ PS	00	01	11	10
0	0	1	0	-
1	0	1	0	-

$OUT = \overline{PS_1}\ PS_0$



Solution B - Mealy

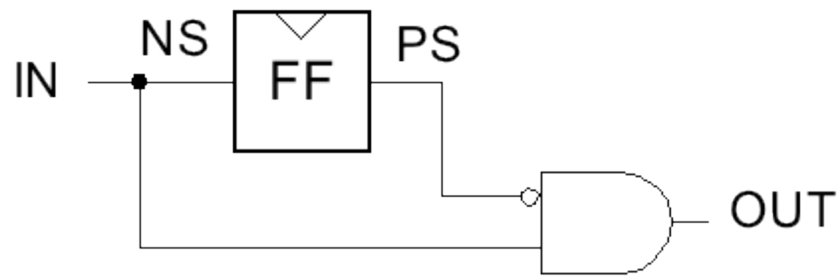
Output depends not only on PS but also on input, IN



Let ZERO=0, ONE=1

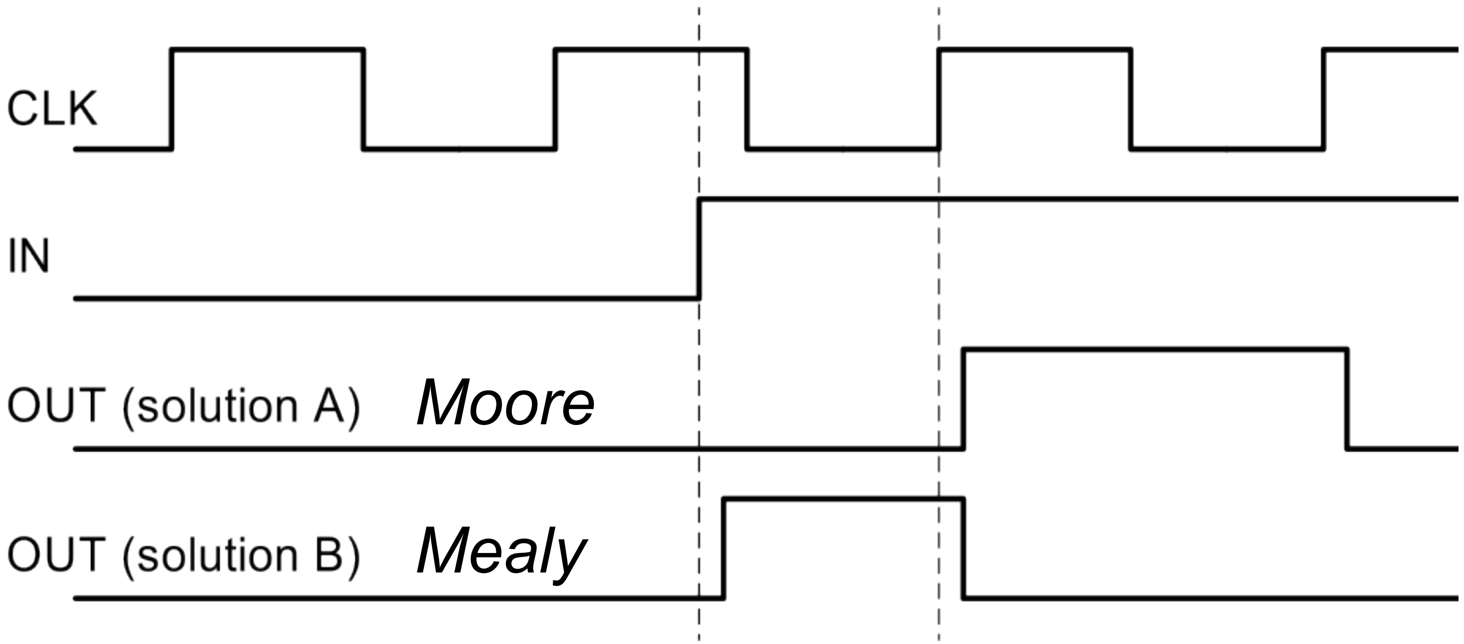
IN	PS	NS	OUT
0	0	0	0
0	1	0	0
1	0	1	1
1	1	1	0

NS = IN, OUT = IN PS'



What's the intuition about this solution?

Edge detector timing diagrams



- *Solution A: both edges of output follow the clock*
- *Solution B: output rises with input rising edge and is asynchronous wrt the clock, output fails synchronous with next clock edge*

FSM Comparison

Solution A

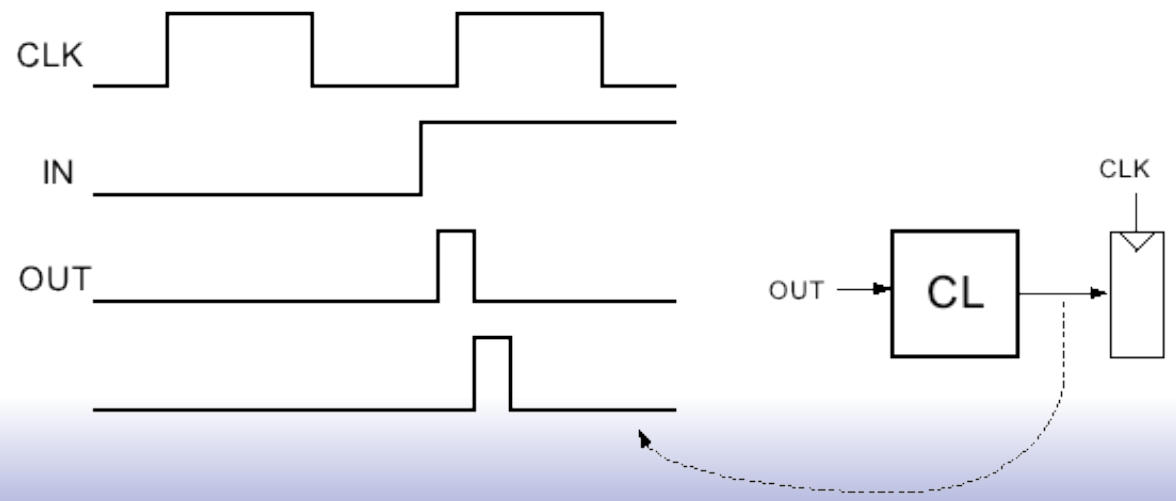
Moore Machine

- output function only of PS
- maybe more states
- synchronous outputs
  - Input glitches not send at output
  - one cycle "delay"
  - full cycle of stable output

Solution B

Mealy Machine

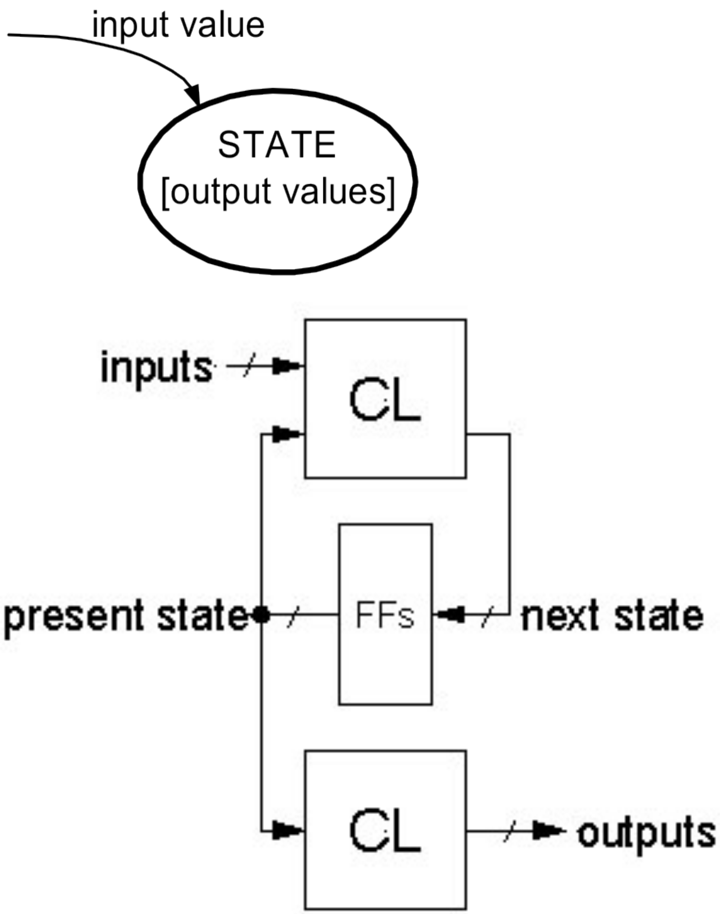
- output function of both PS & input
- maybe fewer states
- asynchronous outputs
- – if input glitches, so does output
- – output immediately available
- – output may not be stable long enough to be useful (below):



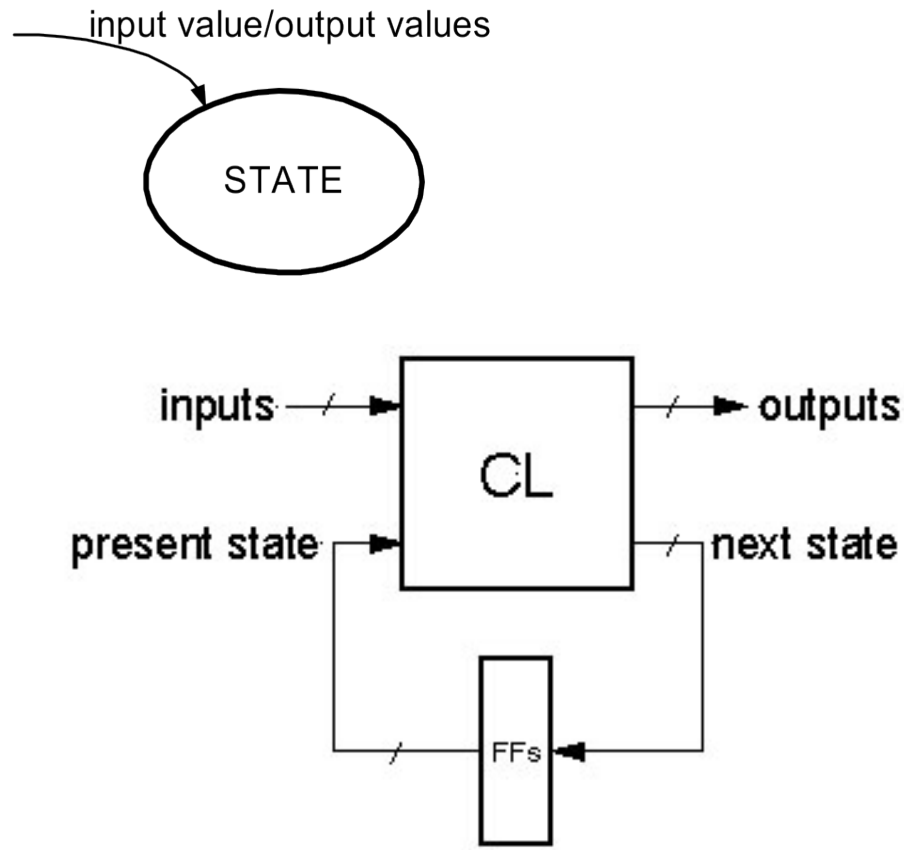
If output of Mealy FSM goes through combinational logic before being registered, the CL might delay the signal and it could be missed by the clock edge (or violate set-up time requirement)

FSM Moore and Mealy Implementation Review

Moore Machine

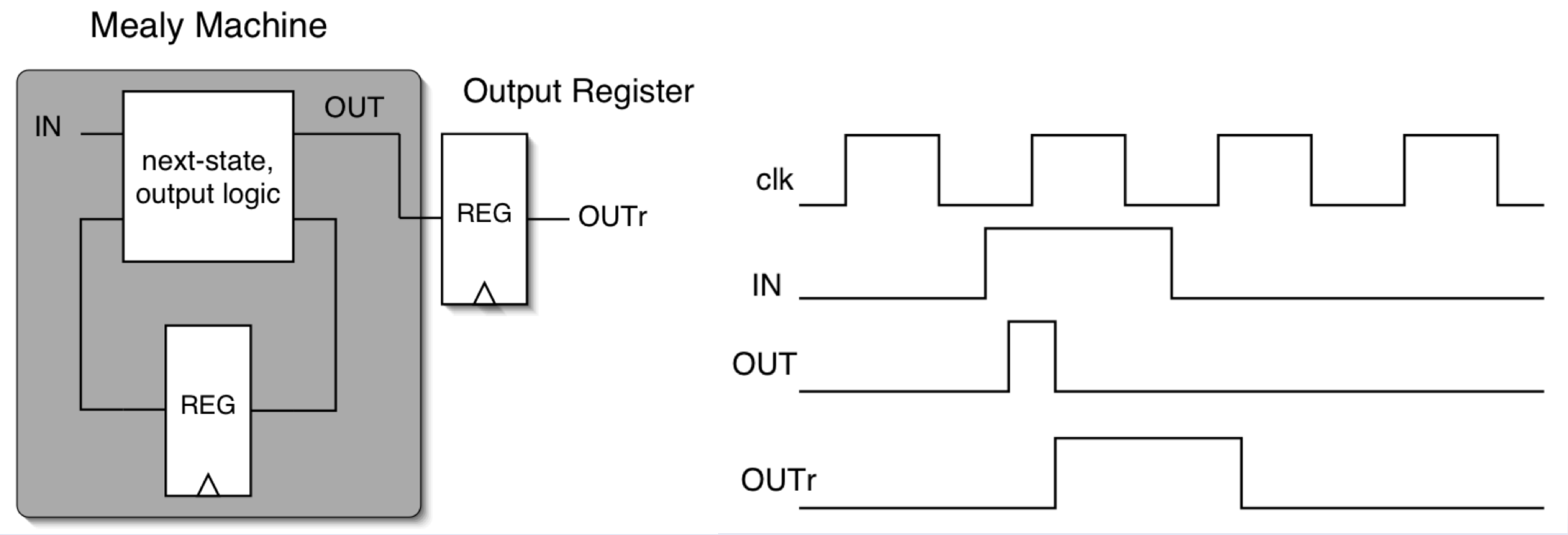


Mealy Machine



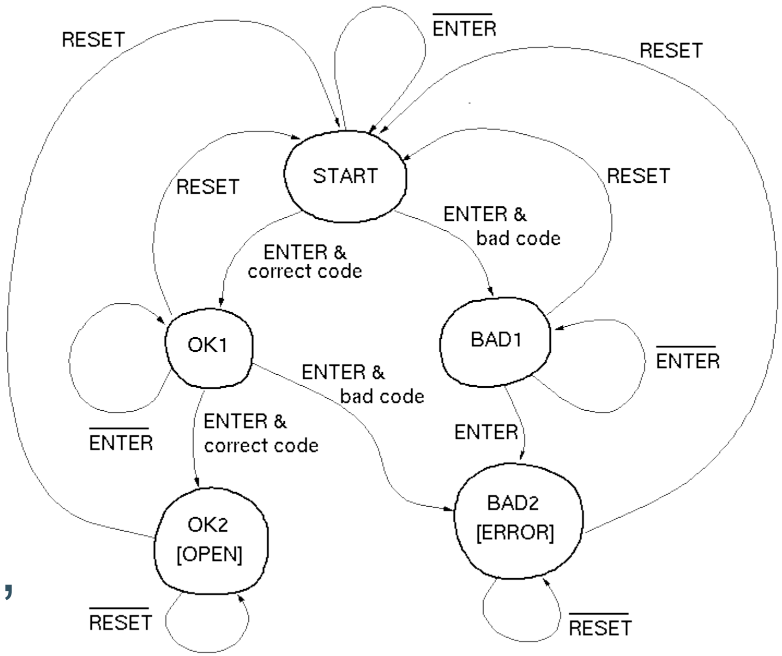
Final Notes on Moore versus Mealy

1. A given state machine *could* have *both* Moore and Mealy style outputs. Nothing wrong with this, but you need to be aware of the timing differences between the two types.
2. The output timing behavior of the Moore machine can be achieved in a Mealy machine by “registering” the Mealy output values:



State Assignment

- When FSM implemented with gate logic, number of gates will depend on mapping between symbolic state names and binary encodings
- Ex: combination lock FSM
  - 5 states, 3 bits
  - my assignment START=000, OK1=001, OK2=011, BAD1=100, BAD2=101
  - only one of 6720 3-bit
- 5 states = 8 choices for first state, 7 for second, 6 for third, 5 for forth, 4 for last = 6720 different encodings

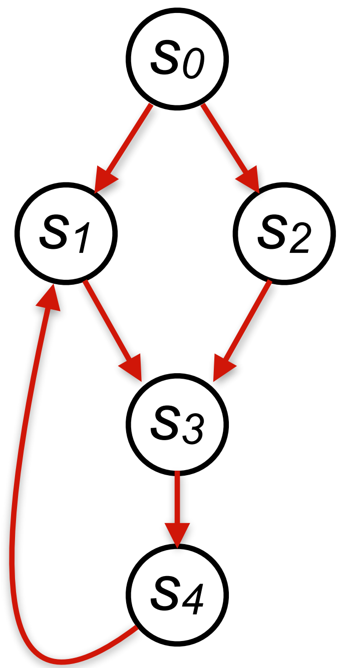


State Assignment

Pencil & Paper Heuristic Methods

State Maps: similar in concept to K-maps

If state X transitions to state Y, then assign "close" assignments to X and Y



State Name	Assignment Q_2	Q_1	Q_0
S_0	0	0	0
S_1	1	0	1
S_2	1	1	1
S_3	0	1	0
S_4	0	1	1

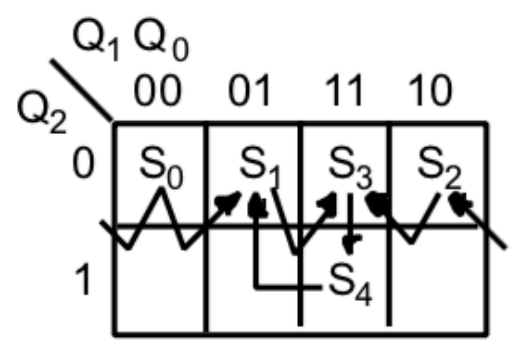
Assignment

Q_2 \ $Q_1 Q_0$	00	01	11	10
0	S_0		S_4	S_3
1		S_1	S_2	

State Map

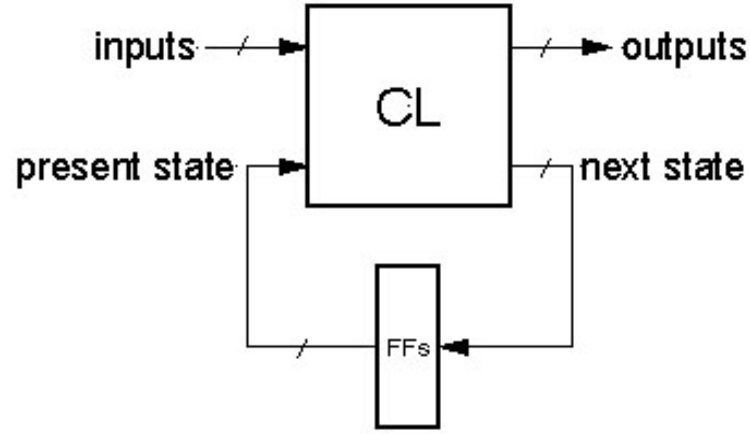
State Name	Assignment Q_2	Q_1	Q_0
S_0	0	0	0
S_1	0	0	1
S_2	0	1	0
S_3	0	1	1
S_4	1	1	1

Assignment



State Map

State Encoding



- In general:

  # of possible FSM states = $2^{\text{\# of Flip-flops}}$

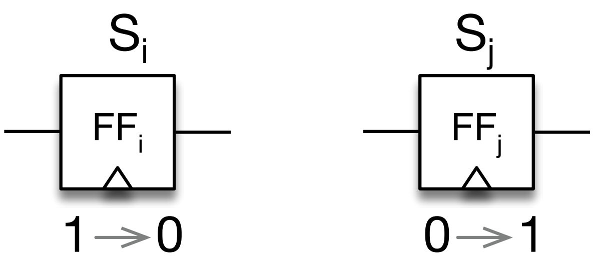
  Example:

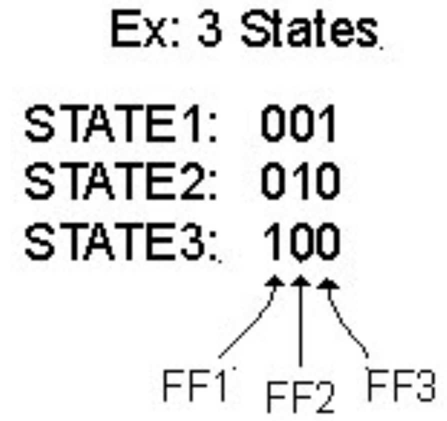
  state1 = 01, state2 = 11, state3 = 10, state4 = 00

- However, sometimes more than $\log_2$(# of states) FFs are used, to simplify logic at the cost of more FFs.
- Extreme example is <u>one-hot state encoding</u>.

State Encoding

- **One-hot encoding of states.**
- *One FF per state.*
- A single 1 moves from FF_i to FF_j
- representing state transition from S_i to S_j



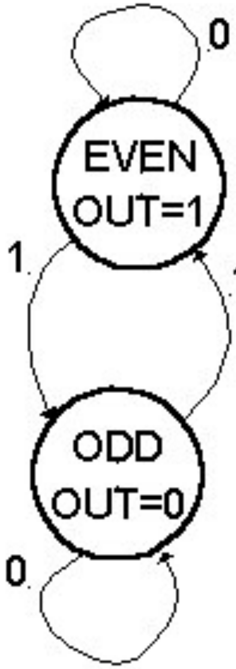


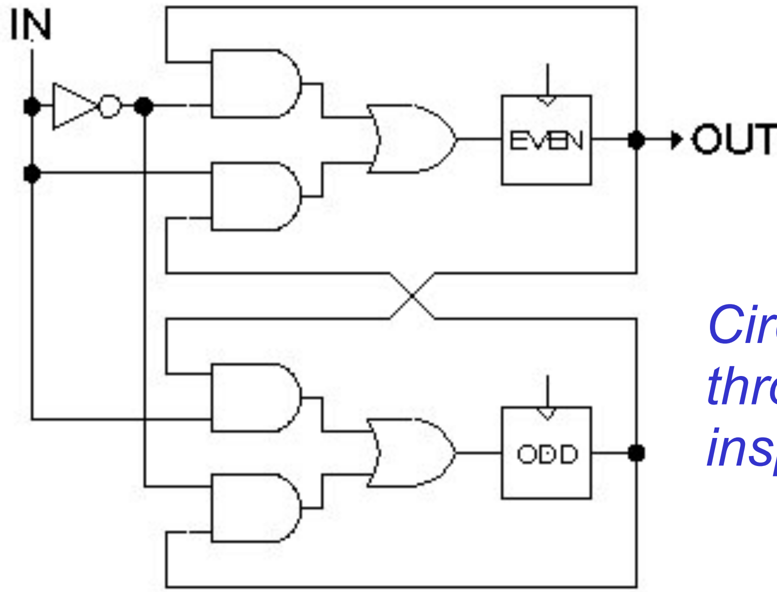
- Why one-hot encoding?
  - Simple design procedure.
    - Circuit matches state transition diagram (example next page).
  - Often can lead to simpler and faster "next state" and output logic.
- Why not do this?
  - Can be costly in terms of Flip-flops for FSMs with large number of states.
- FPGAs are "Flip-flop rich", therefore one-hot state machine encoding is often a good approach.

One-hot encoded FSM

Think about moving a single token from state to state.

- Even Parity Checker Circuit:

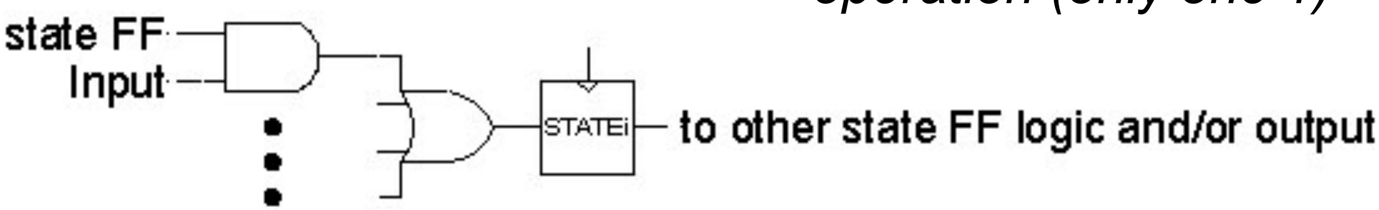




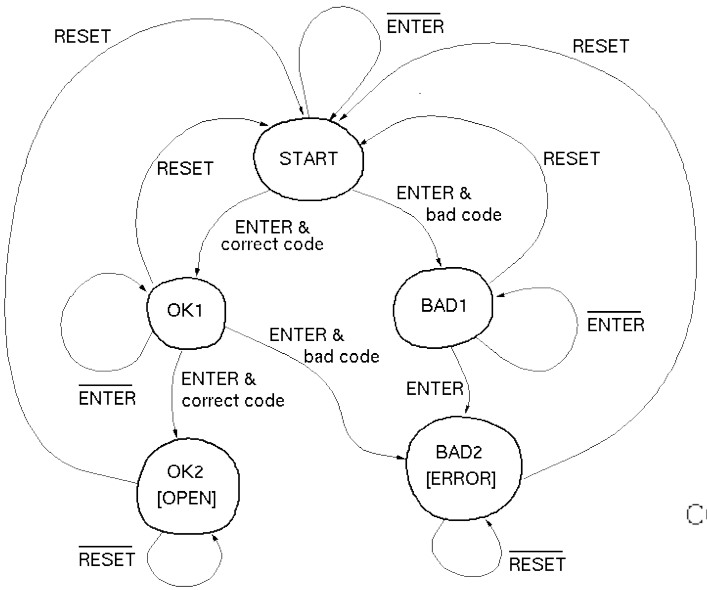
Circuit generated through direct inspection of the STD.

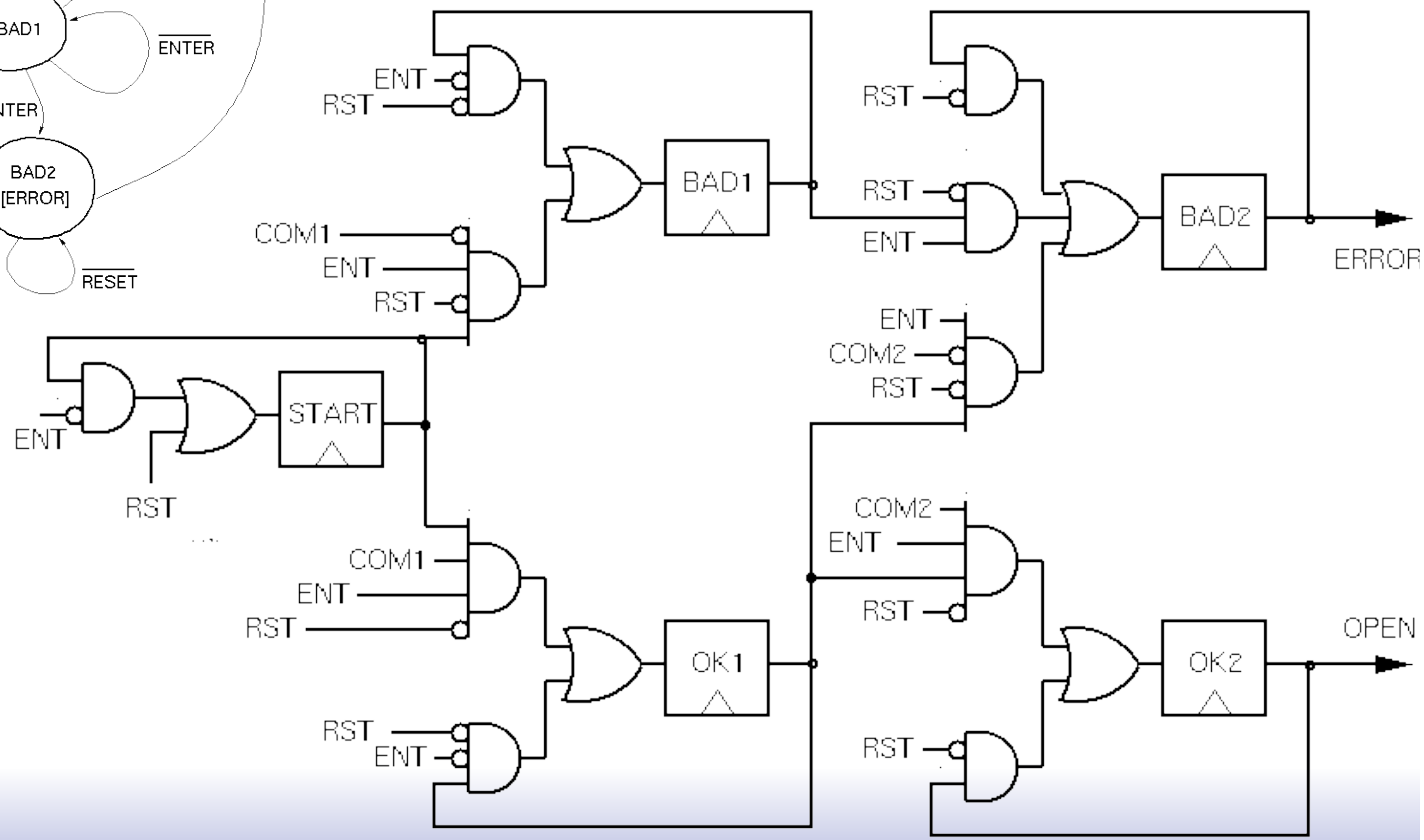
- In General:

- *FFs must be initialized for correct operation (only one 1)*



One-hot encoded combination lock





FSMs in Verilog

General FSM Design Process with Verilog Implementation

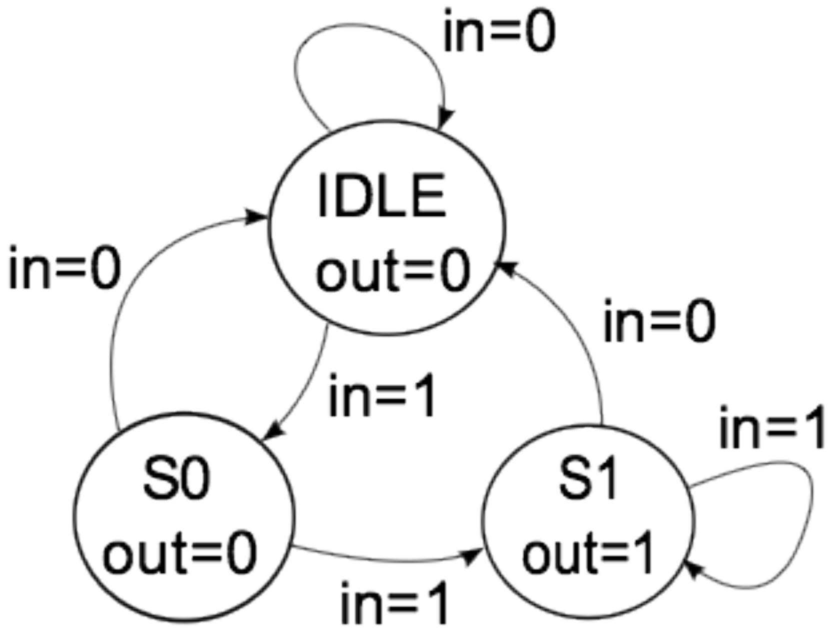
Design Steps:

1. *Specify **circuit function** (English)*
2. *Draw **state transition diagram***
3. *Write down **symbolic state transition table***
4. *Assign encodings (bit patterns) to symbolic states*
5. *Code as Verilog behavioral description*

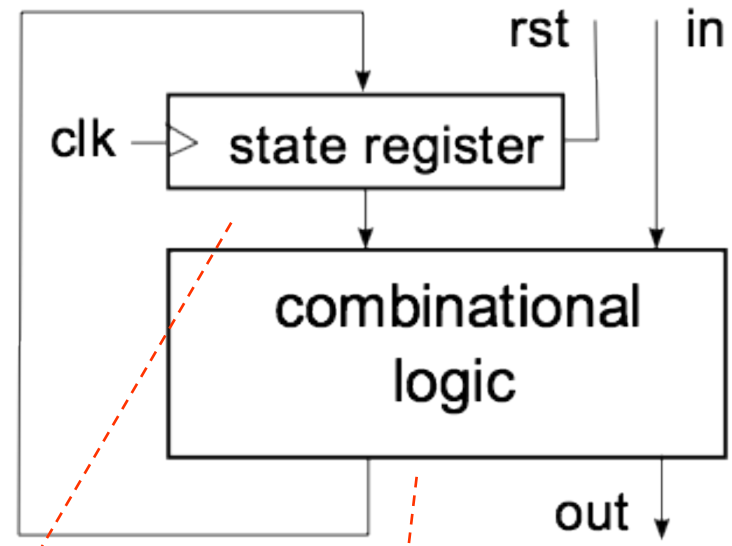
- ✓ *Use parameters to represent encoded states.*
- ✓ *Use register instances for present-state plus CL logic for next-state and outputs.*
- ✓ *Use case for CL block. Within each case section (state) assign all outputs and next state value based on inputs. Note: For Moore style machine make outputs dependent only on state not dependent on inputs.*

Finite State Machine in Verilog

State Transition Diagram



Implementation Circuit Diagram



Holds a symbol to keep track of which bubble the FSM is in.

CL functions to determine output value and next state based on input and current state.

out = f(in, current state)

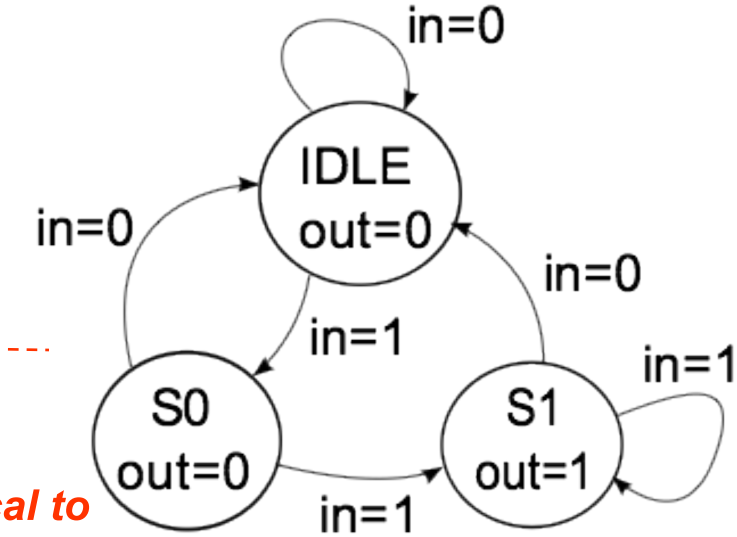
next state = f(in, current state)

Finite State Machines

```
module FSM1(clk, rst, in, out);
input clk, rst;
input in;
output out;
```

Must use reset to force to initial state.

reset not always shown in STD




```
// Defined state encoding:
localparam IDLE = 2'b00;
localparam S0 = 2'b01;
localparam S1 = 2’b10;
```

Constants local to this module.

```
reg out;
reg [1:0] next_state;
wire [1:0] present_state;
```

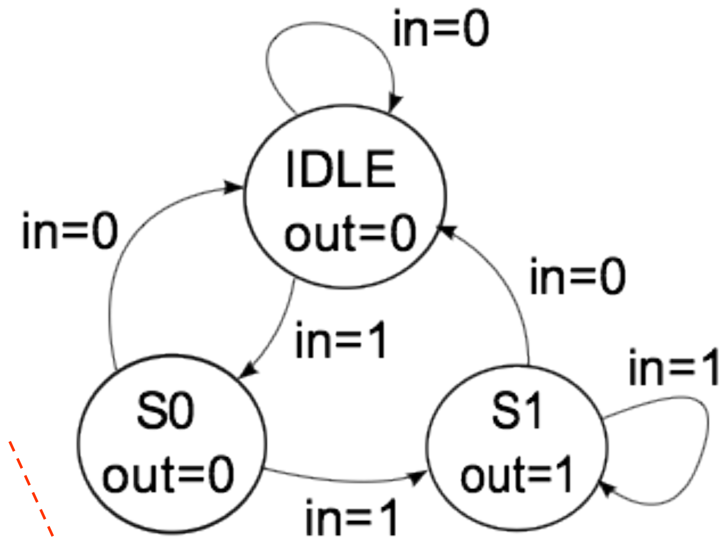
out not a register, but assigned in always block

Combinational logic signals for transition.

```
// state register
REGISTER_R #(.N(2), .INIT(IDLE)) state
(.q(present_state), .d(next_state), .rst(rst));
```

An always block should be used for combination logic part of FSM. Next state and output generation.

FSMs (cont.)




```
// always block for combinational logic portion
always @(present_state or in)
case (present_state)
// For each state def output and next
  IDLE    : begin
             out = 1'b0;
             if (in == 1'b1) next_state = S0;
             else next_state = IDLE;
           end
  S0      : begin
             out = 1'b0;
             if (in == 1'b1) next_state = S1;
             else next_state = IDLE;
           end
  S1      : begin
             out = 1'b1;
             if (in == 1'b1) next_state = S1;
             else next_state = IDLE;
         end
  default: begin
             next_state = IDLE;
             out = 1'b0;
           end
endcase
endmodule
```

Each state becomes a case clause.

For each state define:

Output value(s)

State transition

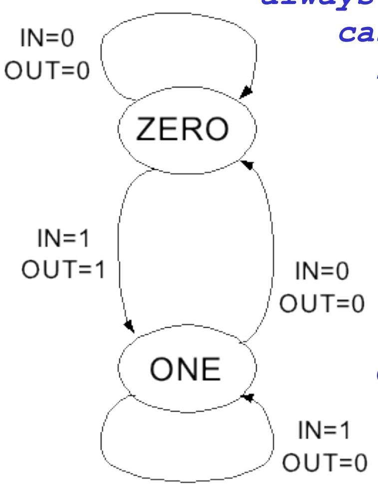
Use "default" to cover unassigned state. Usually unconditionally transition to reset state.

Mealy or Moore?

Edge Detector Example

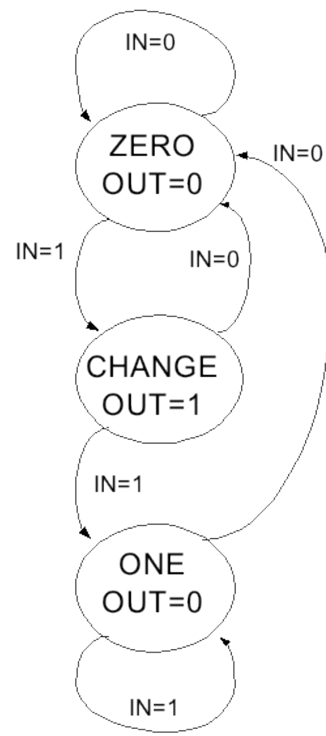
Mealy Machine

```
REGISTER_R #(.INIT(ZERO) state
(.q(ps), .d(ns), .rst(rst));

always @(ps in)
    case (ps)
      ZERO: if (in) begin
                out = 1'b1;
                ns = ONE;
            end
              else begin
                out = 1'b0;
                ns = ZERO;
              end
      ONE: if (in) begin
              out = 1'b0;
              ns = ONE;
            end
            else begin
              out = 1'b0;
              ns = ZERO;
            end
      default: begin
                 out = 1'bx;
                 ns = default;
               end
```



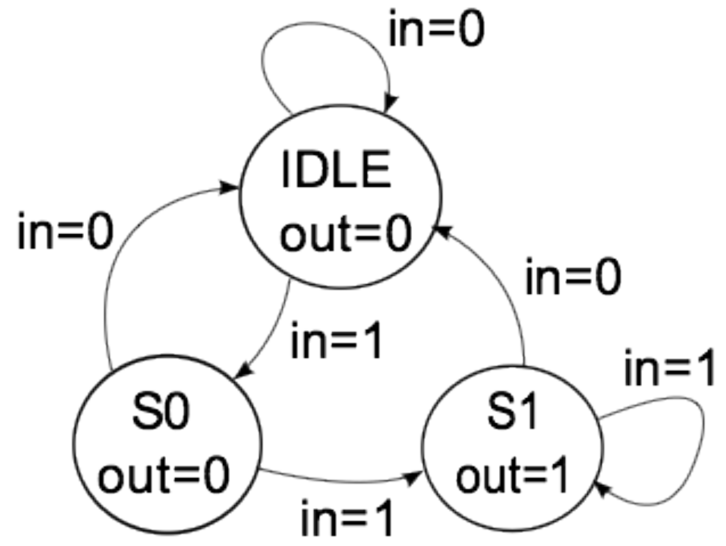
Moore Machine

```
REGISTER_R #(.N(2), .INIT(ZERO)) state
(.q(ps), .d(ns), .rst(rst));

always @(ps in)
    case (ps)
      ZERO: begin
                out = 1'b0;
                if (in) ns = CHANGE;
                  else ns = ZERO;
            end
      CHANGE: begin
                 out = 1'b1;
                 if (in) ns = ONE;
                 else ns = ZERO;
               end
        ONE: begin
                 out = 1'b0;
                 if (in) ns = ONE;
                 else ns = ZERO;
      default: begin
                 out = 1'bx;
                 ns = default;
               end
```



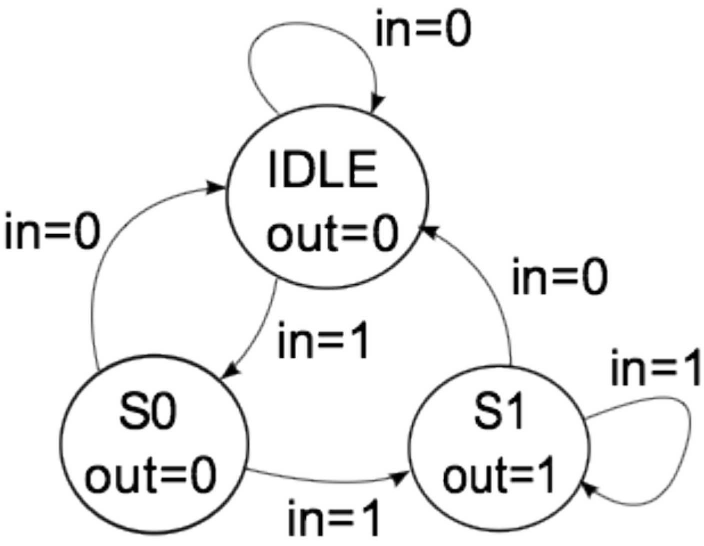
FSM CL block (original)

```
always @(present_state or in)
 case (present_state)
  IDLE    : begin
              out = 1'b0;
              if (in == 1'b1) next_state = S0;
              else next_state = IDLE;
            end
  S0      : begin
              out = 1'b0;
              if (in == 1'b1) next_state = S1;
              else next_state = IDLE;
            end
  S1      : begin
              out = 1'b1;
              if (in == 1'b1) next_state = S1;
              else next_state = IDLE;

             end
  default: begin
               next_state = IDLE;
               out = 1'b0;
             end
 endcase
endmodule
```



The sequential semantics of the blocking assignment allows variables to be multiply assigned within a single always block.

FSM CL block rewritten




```
always @*
  begin
   next_state = IDLE;
   out = 1'b0;
   case (state)
    IDLE    : if (in == 1'b1) next_state = S0;
    S0      : if (in == 1'b1) next_state = S1;
    S1      : begin
               out = 1'b1;
               if (in == 1'b1) next_state = S1;
              end
    default: ;
   endcase
  end
Endmodule
```

*** for sensitivity list**

Normal values: used unless specified below.

Within case only need to specify exceptions to the normal values.

Note: The use of "blocking assignments" allow signal values to be "rewritten", simplifying the specification.