EECS 151/251A
Spring 2021
Digital Design and Integrated Circuits

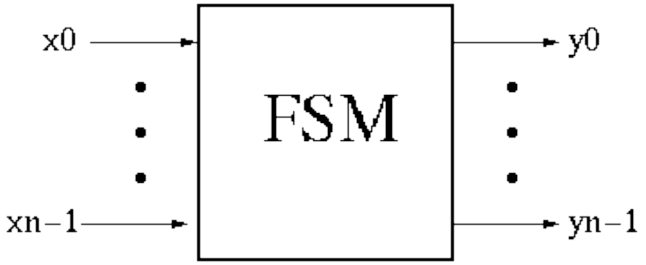
Instructor:
John Wawrzynek

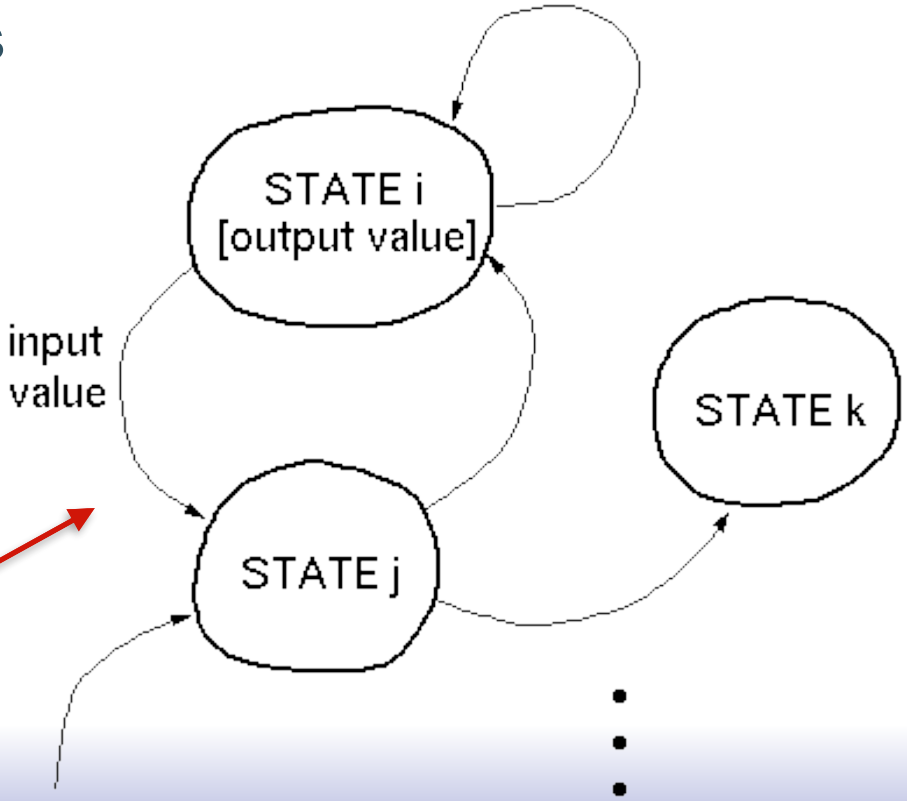
Lecture 7: FSMs Part 1

Finite State Machines

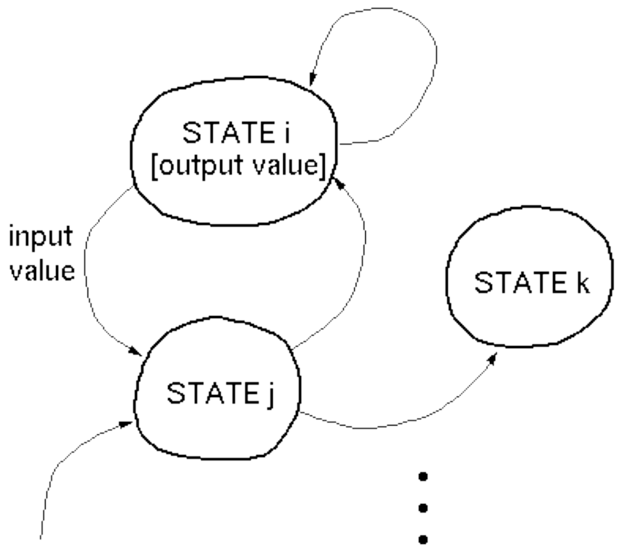
Finite State Machines (FSMs)

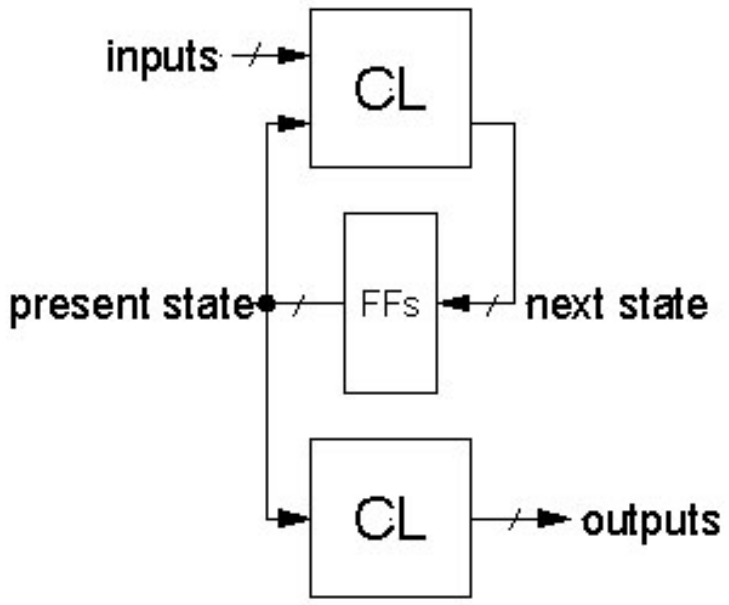
- **FSMs:**
  - Can model behavior of *any sequential circuit*
  - Useful representation for designing sequential circuits
  - As with all sequential circuits: output depends on present *and* past inputs
    - effect of past inputs represented by the current *state*
- Behavior is represented by ***State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





FSM Implementation





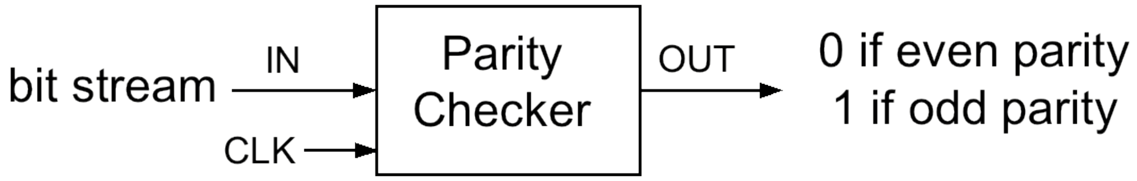
- Flip-flops form *state register*
- number of states ≤ $2^{\text{number of flip-flops}}$
- CL (combinational logic) calculates next state and output
- Remember: The FSM follows exactly one edge per cycle.

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

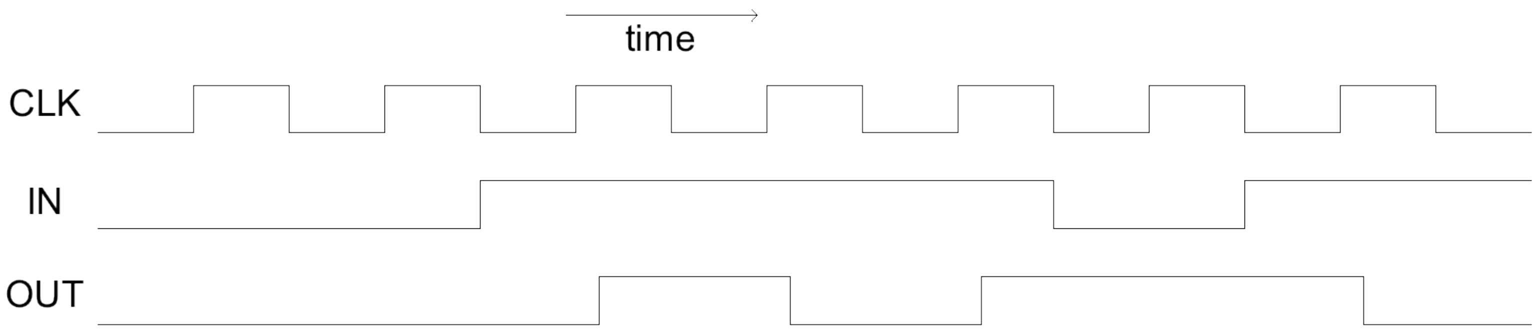
Parity Checker: FSM Example

A string of bits has "even parity" if the number of 1's in the string is even.

- Design a circuit that accepts a infinite serial stream of bits, and outputs a 0 if the parity thus far is even and outputs a 1 if odd:

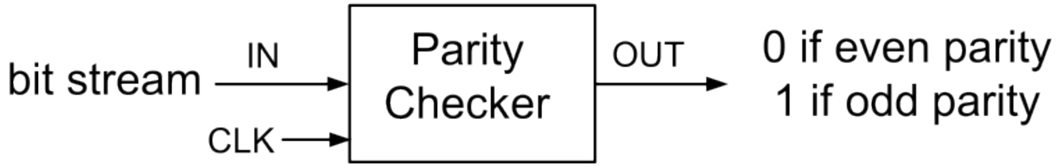


example:	0	0	1	1	1	0	1
	even	even	odd	even	odd	odd	even



Next we take this example through the "formal design process". But first, can you guess a circuit that performs this function?

By-hand Design Process (a)

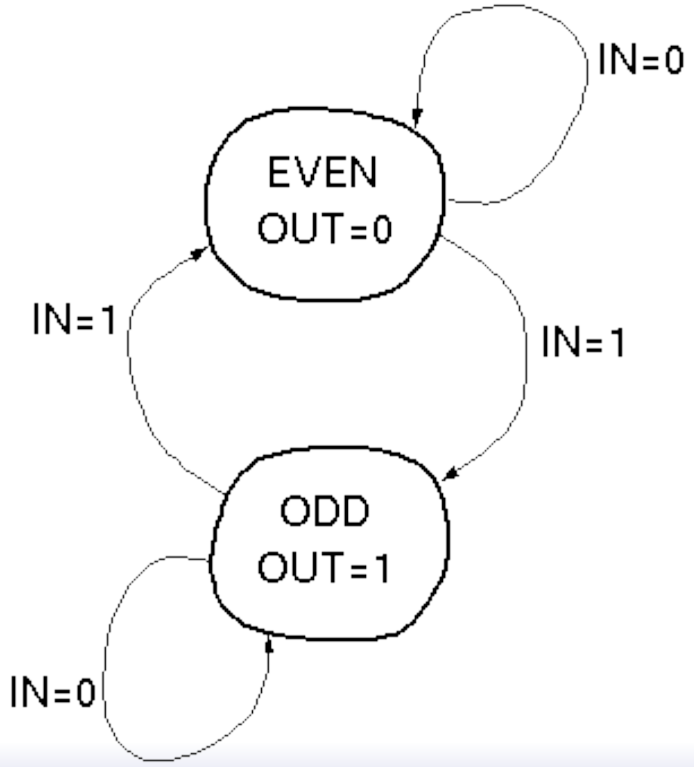


example:	0	0	1	1	1	0	1
	even	even	odd	even	odd	odd	even

time →

"State Transition Diagram"

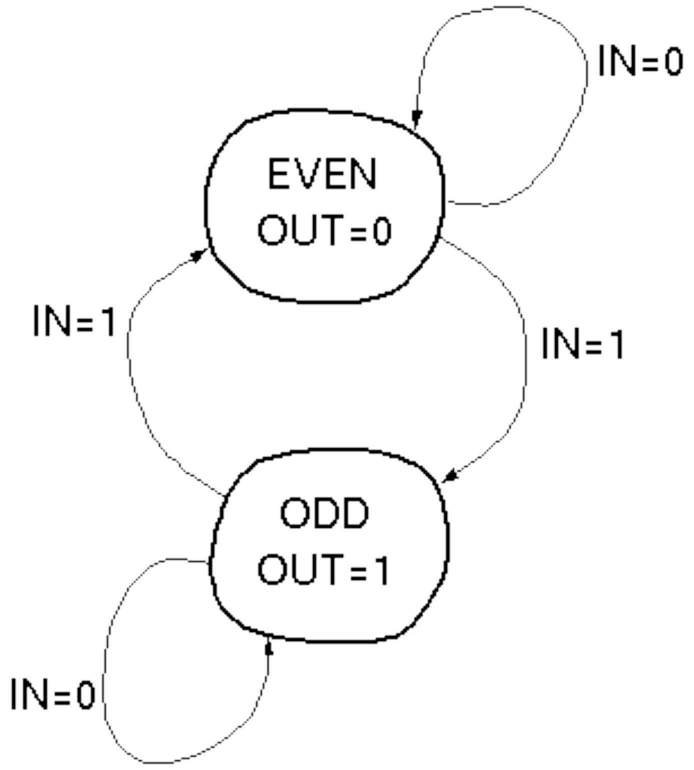
- circuit is in one of two "states".
- transition on each cycle with each new input, over exactly one arc (edge).
- Output depends on which state the circuit is in.



By-hand Design Process (b)

State Transition Table:

present state	*OUT*	*IN*	*next state*
EVEN	*0*	*0*	*EVEN*
EVEN	*0*	*1*	*ODD*
ODD	*1*	*0*	*ODD*
ODD	*1*	*1*	*EVEN*



Invent a code to represent states:

Let 0 = EVEN state, 1 = ODD state

present state (ps)	*OUT*	*IN*	*next state (ns)*
0	*0*	*0*	*0*
0	*0*	*1*	*1*
1	*1*	*0*	*1*
1	*1*	*1*	*0*

Derive logic equations from table (how?):

OUT = PS

NS = PS xor IN

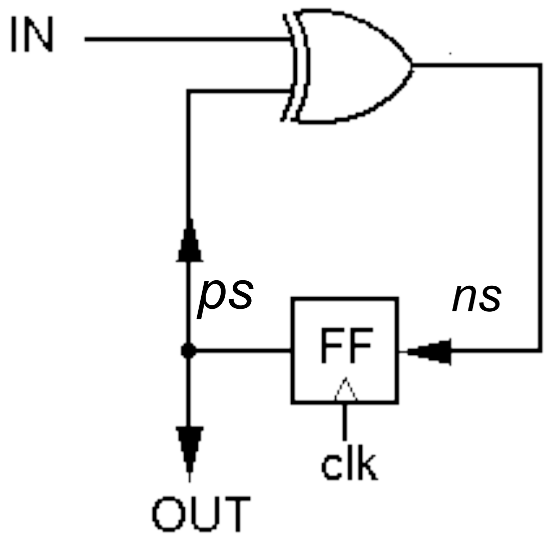
By-hand Design Process (c)

Logic equations from table:

OUT = PS

NS = PS xor IN

- Circuit Diagram:
  - XOR gate for NS calculation
  - Flip-Flop to hold present state
  - no logic needed for output in this example.



"Formal" By-hand Design Process

Review of Design Steps:

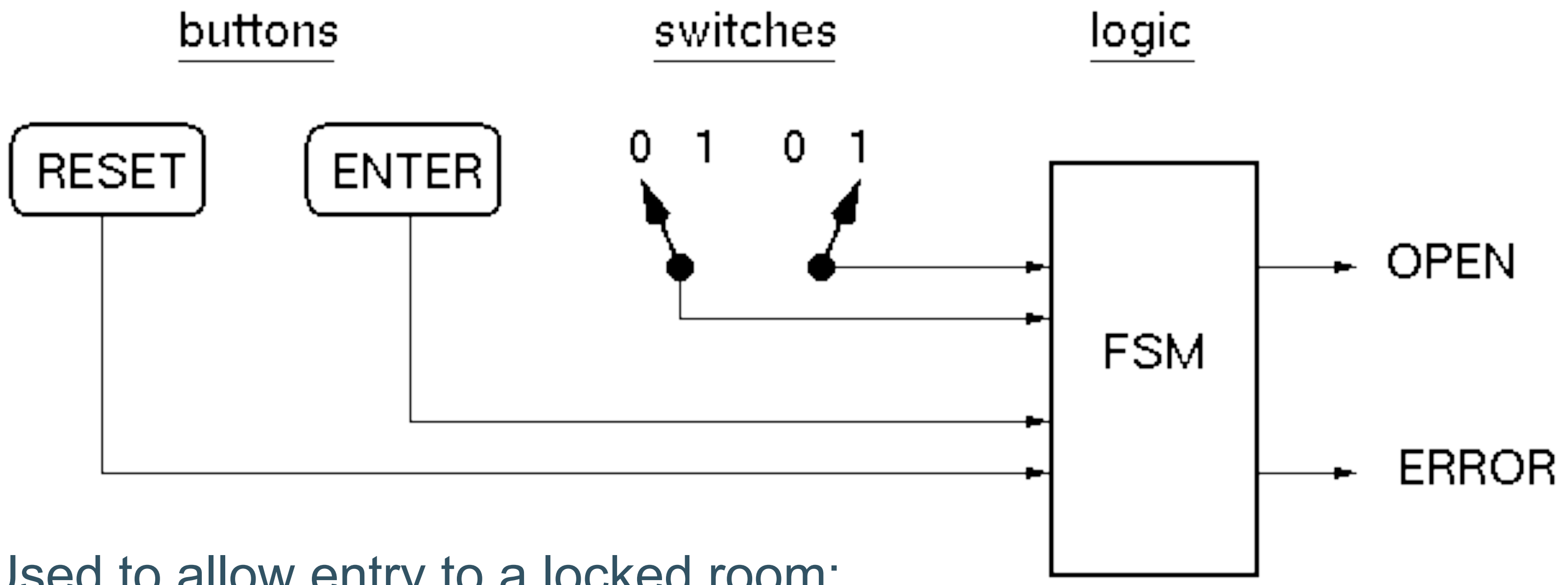
1. *Specify* ***circuit function*** *(English)*
2. *Draw* ***state transition diagram***
3. *Write down* ***symbolic state transition table***
4. *Write down* ***encoded state transition table***
5. *Derive* ***logic equations***
6. *Derive* ***circuit diagram***

   Register to hold state

   Combinational Logic for Next State and Outputs

Another FSM Design Example

Combination Lock Example



- Used to allow entry to a locked room:

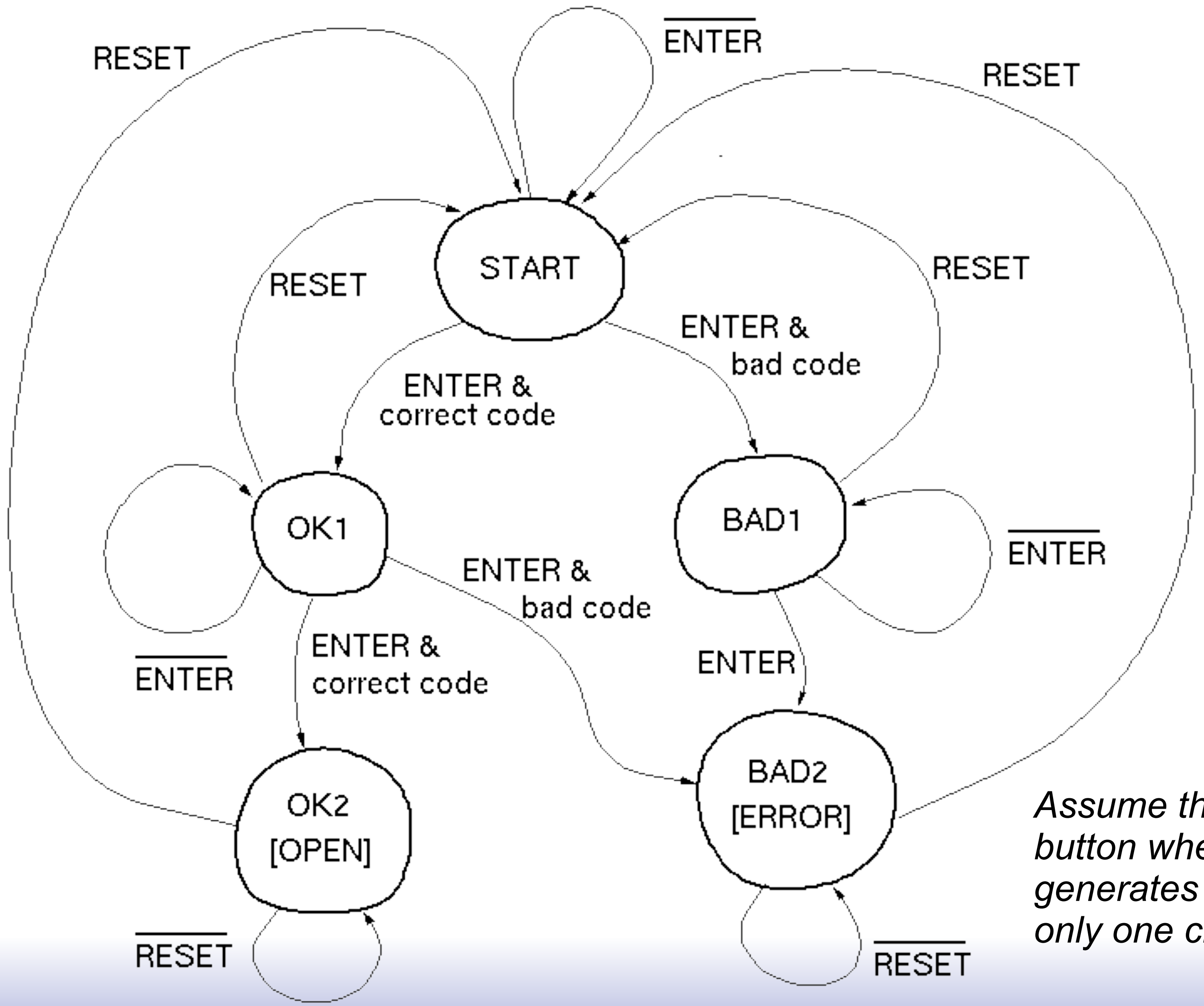
  2-bit serial combination. Example 01,11:

  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

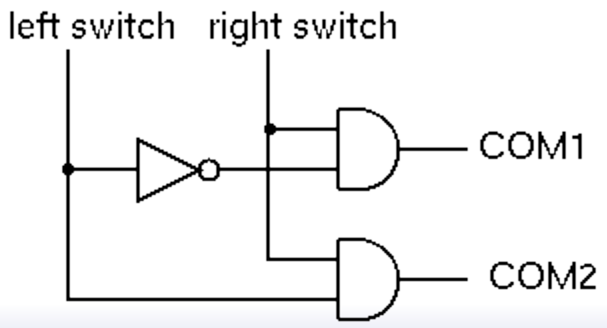


Assume the ENTER *button when pressed generates a pulse for only one clock cycle.*

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):



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

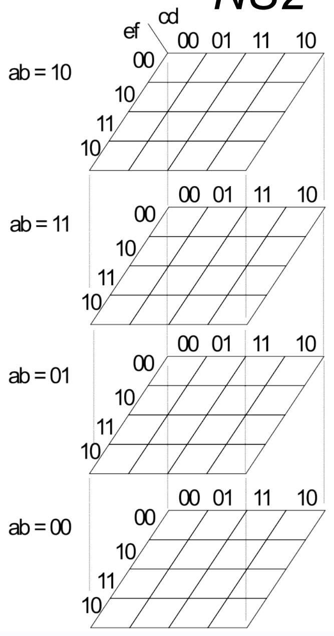
- *Assign states:*

START=000, OK1=001, OK2=011

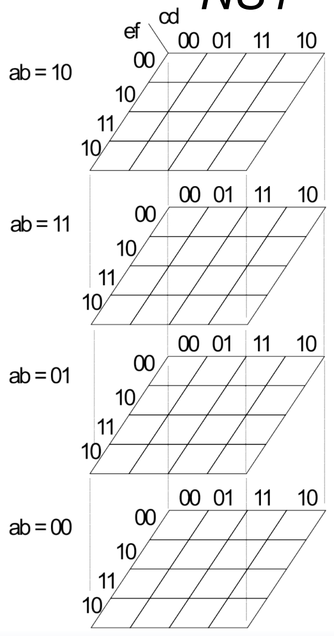
BAD1=100, BAD2=101

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

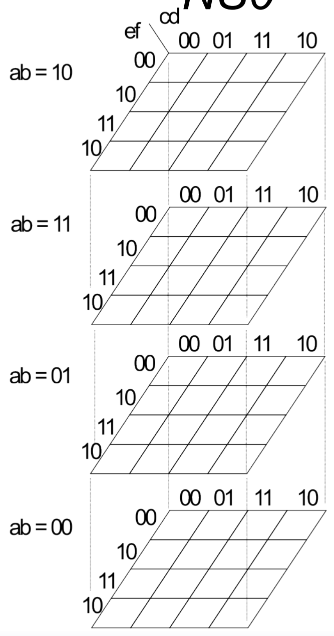
NS2



NS1



NS0

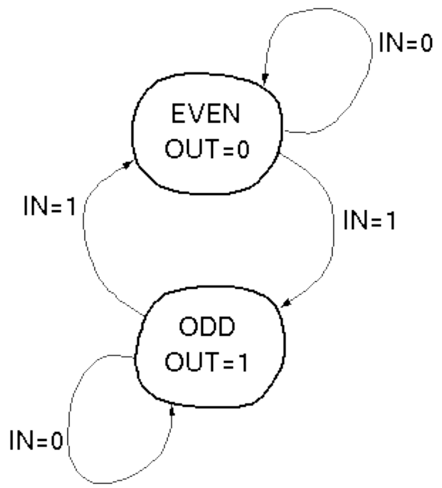


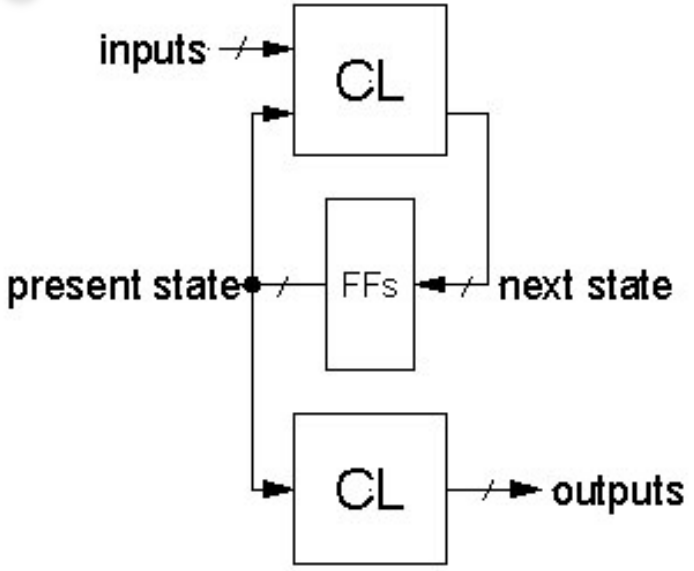
- *What are the output functions for OPEN and ERROR?*

Moore Versus Mealy Machines

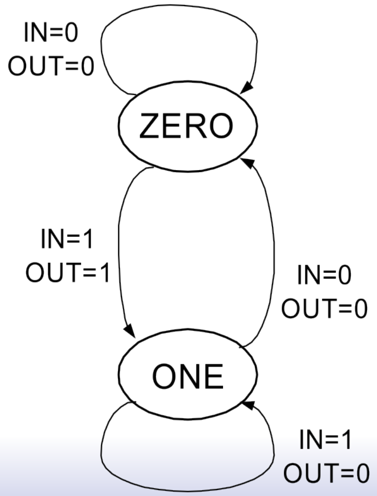
FSM Implementation Notes

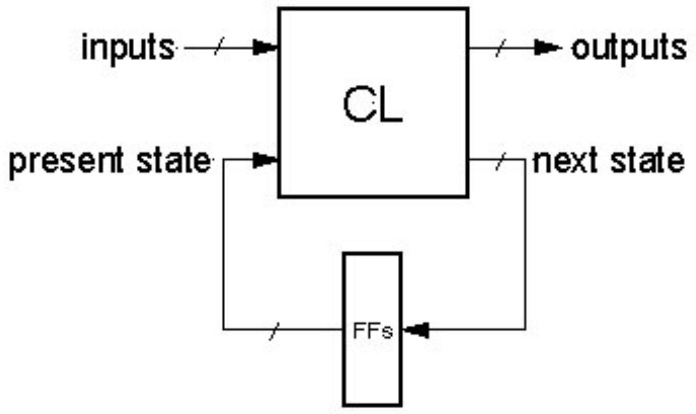
- All examples so far generate output based only on the present state, commonly called a "Moore Machine":





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



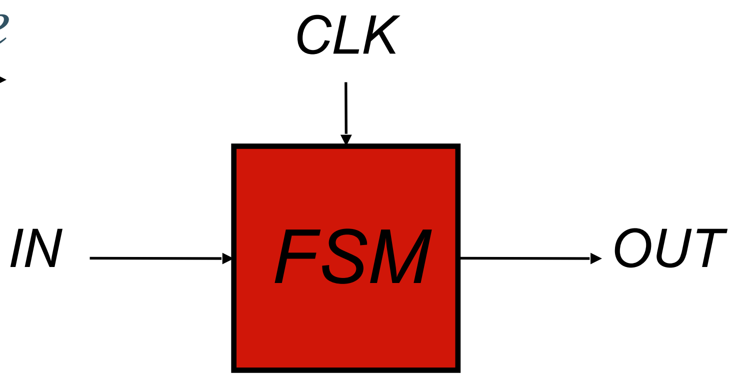


Finite State Machines

- **Example: Edge Detector**

  Bit are received one at a time (one per cycle),

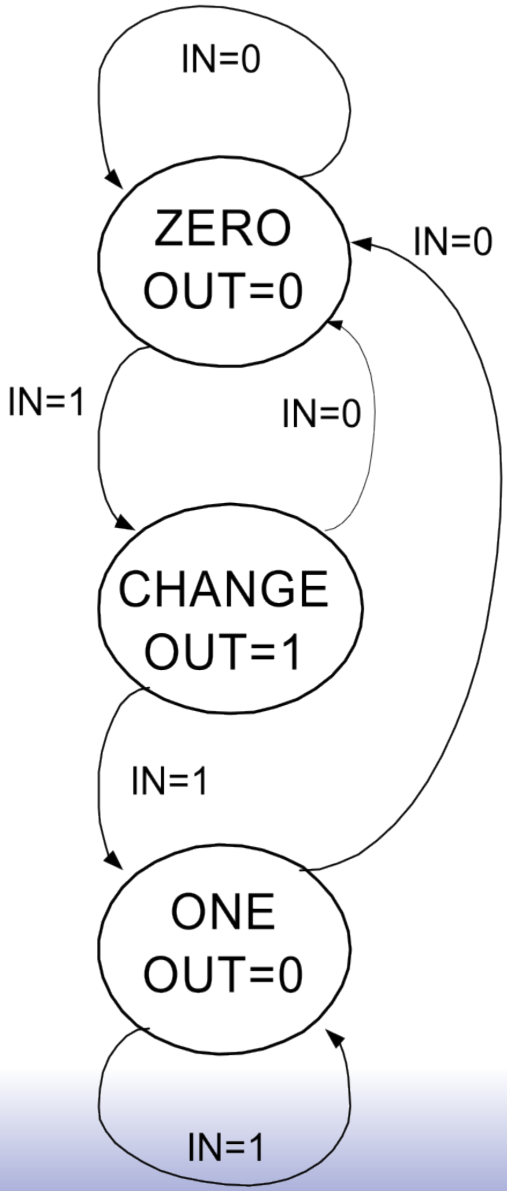
  such as: 000111010 *time* $\longrightarrow$



  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



	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	-

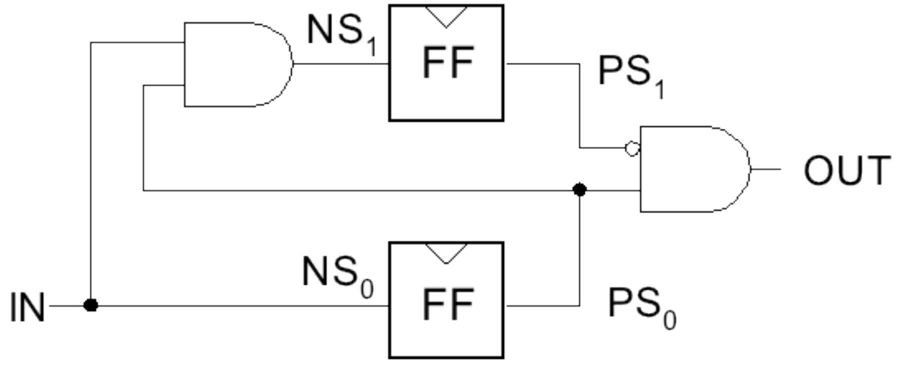
$NS_1 = IN\ PS_0$

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

$NS_0 = IN$

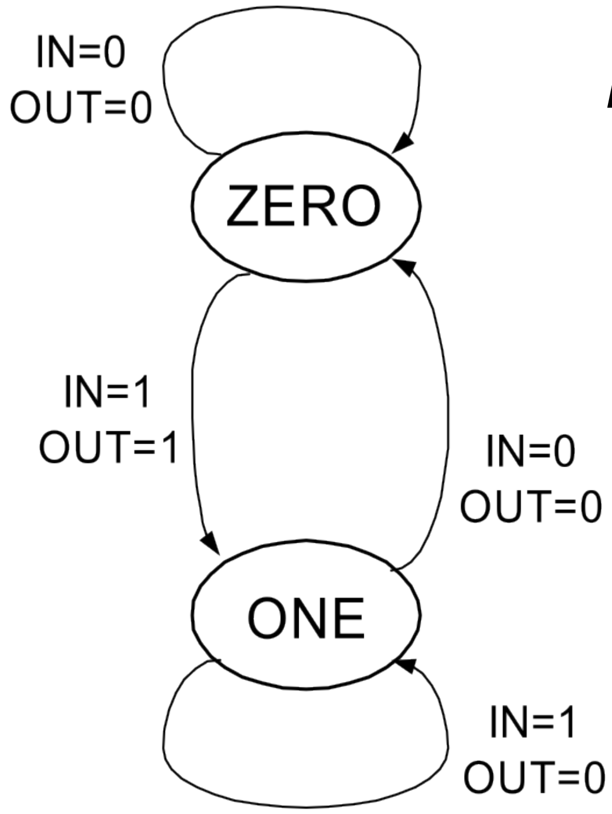
IN \ PS	00	01	11	10
0	0	1	0	-
1	0	1	0	-

$OUT = \overline{PS_1}\ PS_0$



Solution B

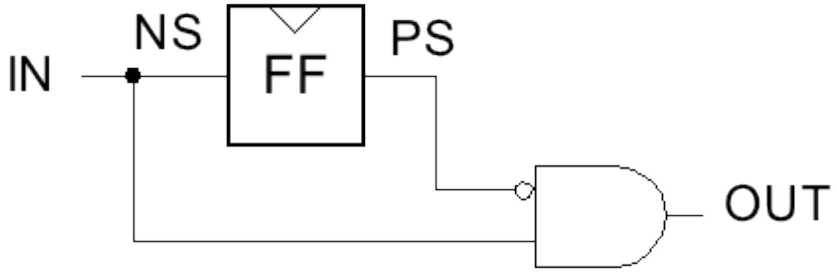
Output depends not only on PS but also on input, IN



Let ZERO=0, ONE=1

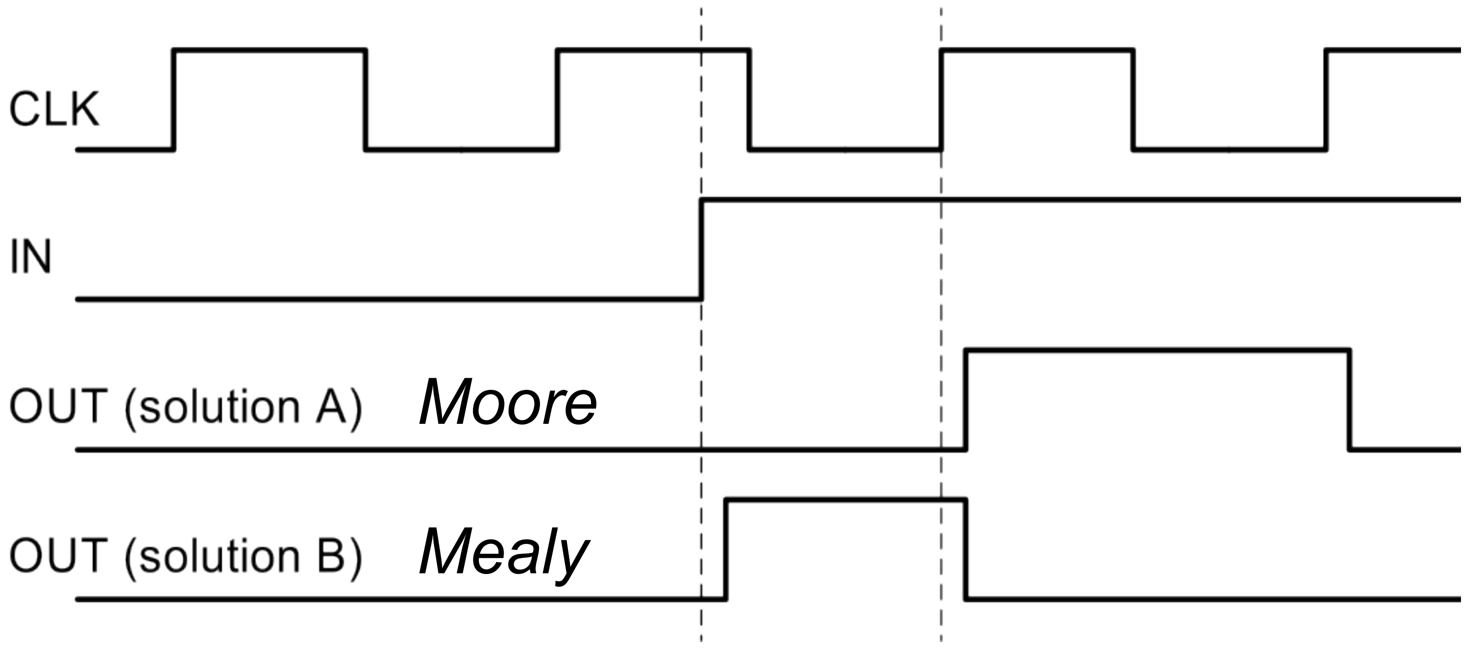
IN	*PS*	*NS*	*OUT*
0	*0*	*0*	*0*
0	*1*	*0*	*0*
1	*0*	*1*	*1*
1	*1*	*1*	*0*

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

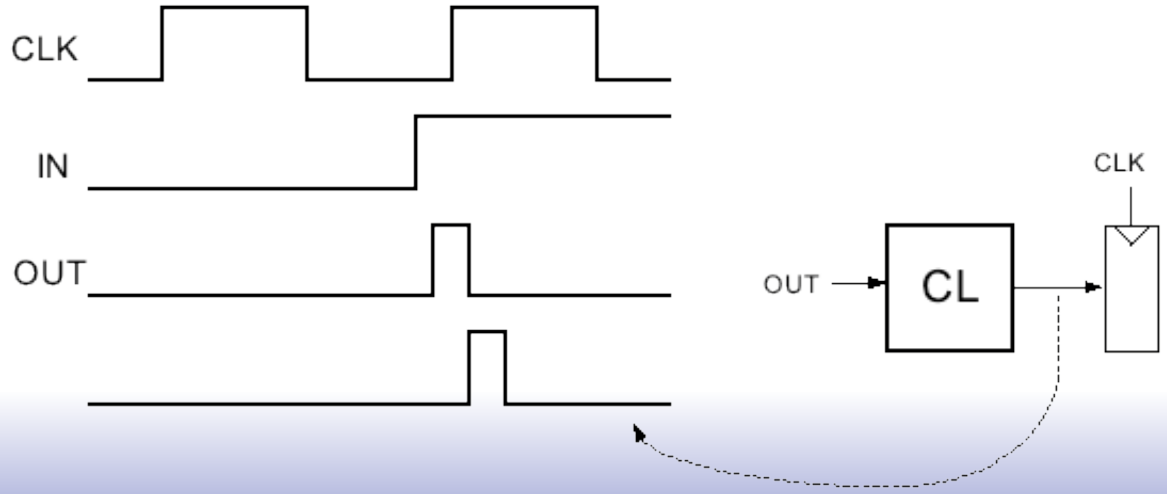
Moore Machine

- output function only of PS
- maybe <u>more</u> states (why?)
- synchronous outputs
  - Input glitches not send at output
  - one cycle "delay"
  - full cycle of stable output

Solution B

Mealy Machine

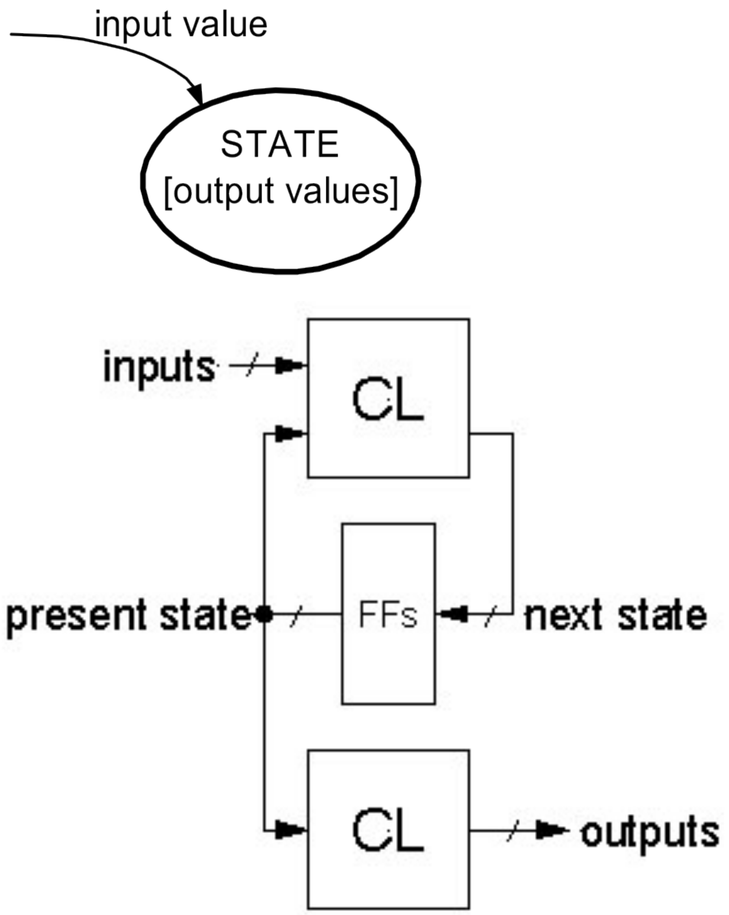
- *output function of both PS & input*
- *maybe fewer states*
- *asynchronous outputs*
- *if input glitches, so does output*
- *output immediately available*
- *output may not be stable long enough to be useful (below):*



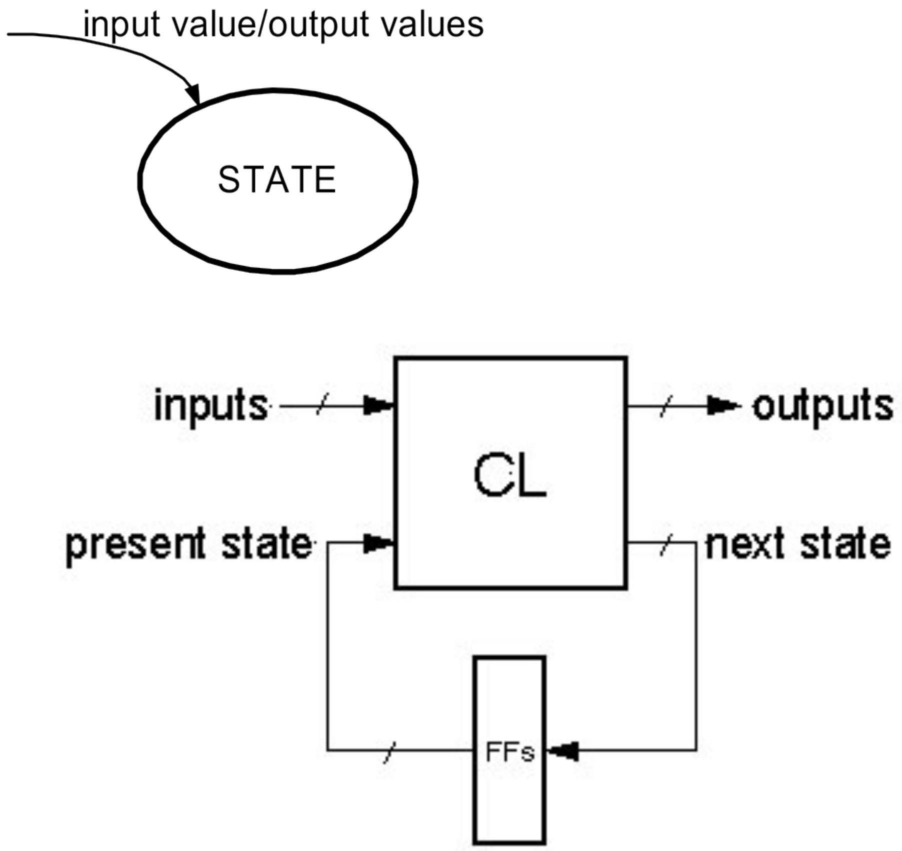
If output of Mealy FSM goes through combinational logic before being registered, the CL might delay the signal and it could be missed by the clock edge (or violate set-up time requirement)

FSM Moore and Mealy Review

Moore Machine

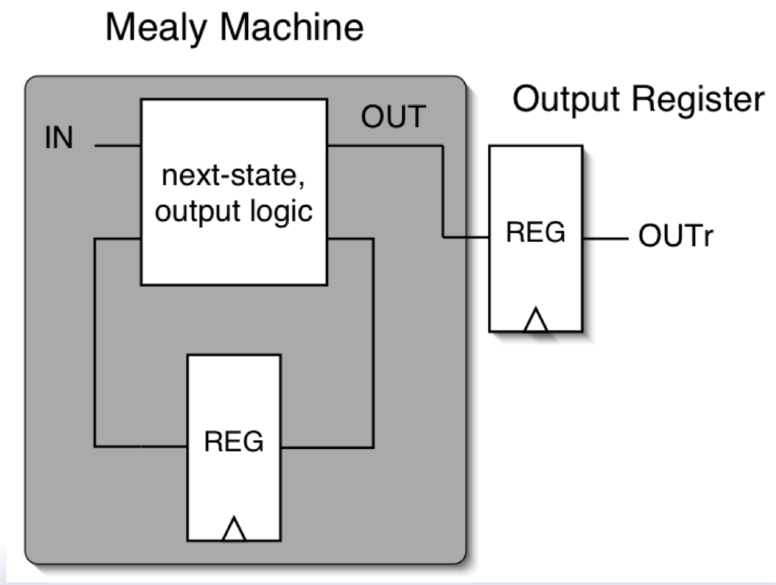


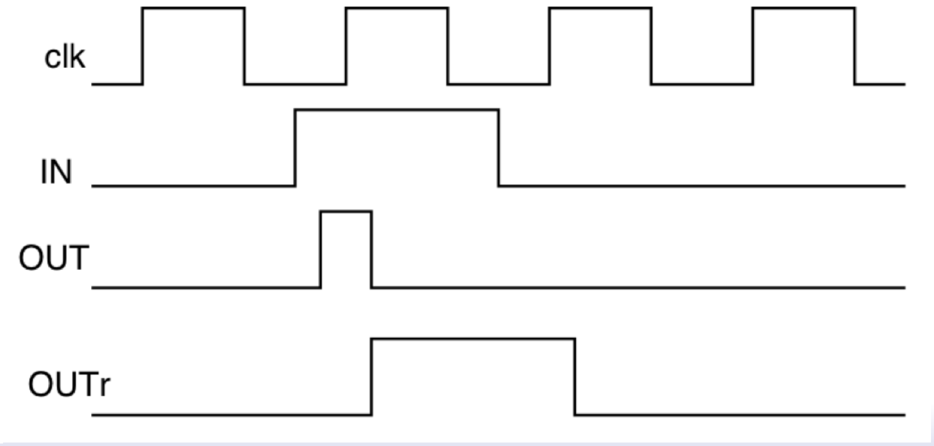
Mealy Machine



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:


Mealy Machine
IN
next-state, output logic
OUT
REG
Output Register
REG
OUTr


clk
IN
OUT
OUTr