EECS150 - Digital Design

Lecture 21 - Metastability, Finite State Machines Revisited

April 6, 2010

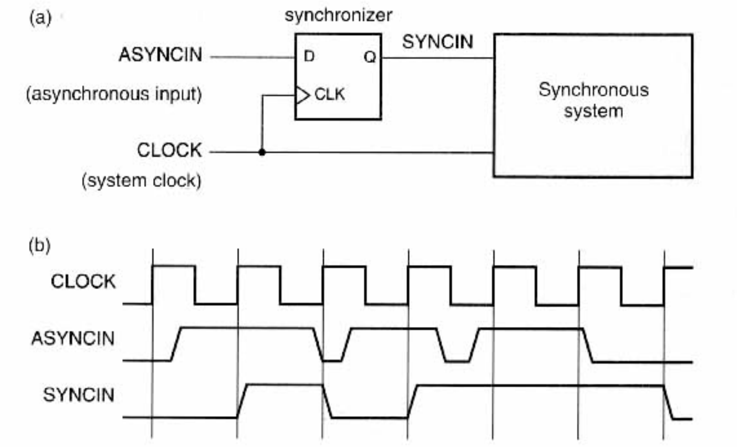
John Wawrzynek

Asynchronous Inputs to Synchronous Systems

- Many synchronous systems need to interface to asynchronous input signals:
  - Consider a computer system running at some clock frequency, say 1GHz with:
    - Interrupts from I/O devices, keystrokes, etc.
    - Data transfers from devices with their own clocks
      - Ethernet has its own 100MHz clock
      - PCI bus transfers, 66MHz standard clock.
  - These signals could have no known timing relationship with the system clock of the CPU.
  - (On FPGAs we can use FIFOs - separate clocks for input and output - as the interface. In general, this is overkill - and too expensive).

"Synchronizer" Circuit

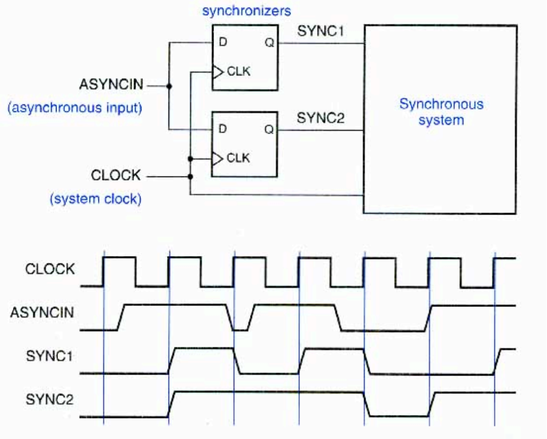
- For a single asynchronous input, we use a simple flip-flop to bring the external input signal into the timing domain of the system clock:



- The D flip-flop samples the asynchronous input at each cycle and produces a synchronous output that meets the setup time of the next stage.

"Synchronizer" Circuit

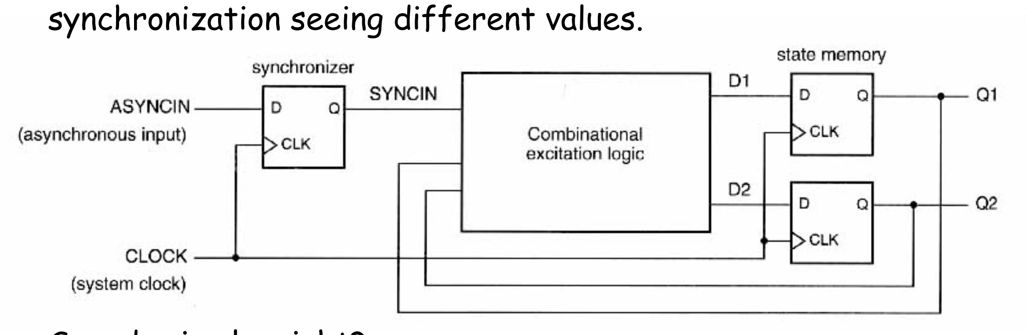
- It is essential for asynchronous inputs to be synchronized at only one place.



- Two flip-flops may not receive the clock and input signals at precisely the same time (clock *and* data skew).
- When the asynchronous changes near the clock edge, one flip-flop may sample input as 1 and the other as 0.

"Synchronizer" Circuit

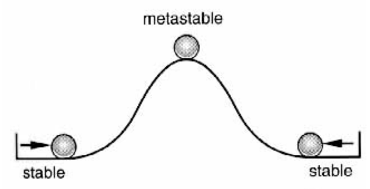
- Single point of synchronization is even more important when input goes to a combinational logic block (ex. FSM)
- The CL block can accidentally hide the fact that the signal is synchronized at multiple points.
- The CL magnifies the chance of the multiple points of synchronization seeing different values.

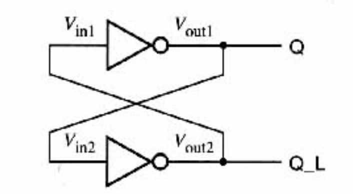


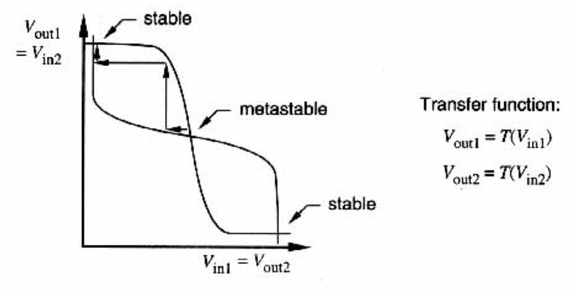
- Sounds simple, right?

Synchronizer Failure & Metastability

- We think of flip-flops having only two stable states - but all have a third *metastable* state halfway between 0 and 1.
- When the setup and hold times of a flip-flop are not met, the flip-flop could be put into the metastable state.
- Noise will be amplified and push the flip-flop one way or other.
- However, in theory, the time to transition to a legal state is unbounded.
- Does this really happen?
- The probability is low, but number of trials is high!





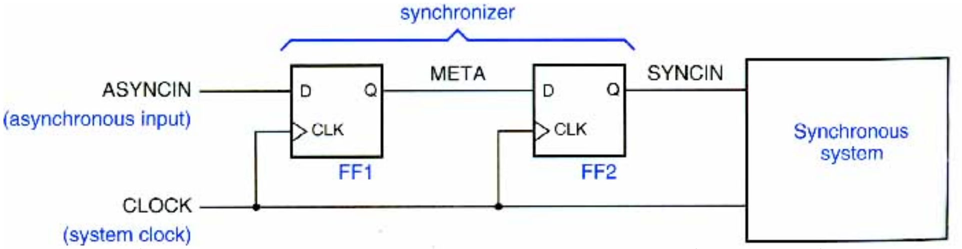


Synchronizer Failure & Metastability

- If the system uses a synchronizer output while the output is still in the metastable state ⇒ synchronizer failure.
- Initial versions of several commercial ICs have suffered from metastability problems - effectively synchronization failure:
  - AMD9513 system timing controller
  - AMD9519 interrupt controller
  - Zilog Z-80 Serial I/O interface
  - Intel 8048 microprocessor
  - AMD 29000 microprocessor
- To avoid synchronizer failure wait long enough before using a synchronizer's output. *"Long enough", according to Wakerly, is so that the mean time between synchronizer failures is several orders of magnitude longer than the designer's expected length of employment!*
- In practice all we can do is reduce the probability of failure to a vanishing small value.

Reliable Synchronizer Design

- **The probability that a flip-flop stays in the metastable state decreases exponentially with time.**
- Therefore, any scheme that delays using the signal can be used to decrease the probability of failure.
- In practice, delaying the signal by a cycle is usually sufficient:



- If the clock period is greater than metastability resolution time plus FF2 setup time, FF2 gets a synchronized version of ASYNCIN.
- Multi-cycle synchronizers (using counters or more cascaded flip-flops) are even better – but often overkill.

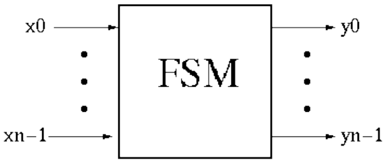
Purely Asynchronous Circuits

- Many researchers (and a few industrial designers) have proposed a variety of circuit design methodologies that **eliminate the need for a globally distributed clock**.
- They cite a variety of important potential advantages over synchronous systems.
- To date, these attempts have remained mainly in Universities.
- A few commercial asynchronous chips/systems have been build.
- Sometimes, asynchronous blocks sometimes appear inside otherwise synchronous systems.
  - Asynchronous techniques have long been employed in DRAM and other memory chips for generation internal control without external clocks. (Precharge/sense-amplifier timing based on address line changes.

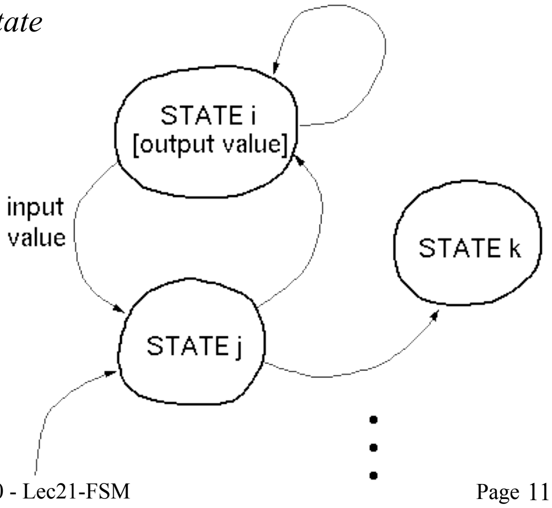
Now on to FSMs

Finite State Machines (FSMs)

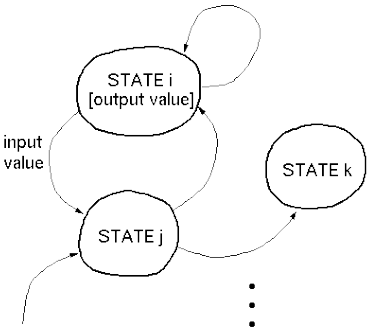
- FSM circuits are a type of *sequential circuit*:
  - output depends on present *and* past inputs
    - effect of past inputs is represented by the current *state*

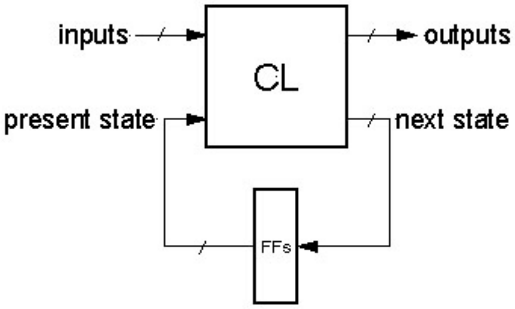


- Behavior is represented by *State Transition Diagram*:
  - traverse one edge per clock cycle.



FSM Implementation





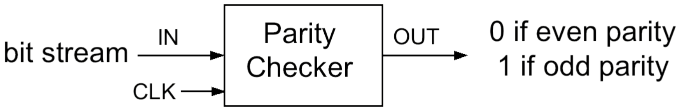
- FFs form *state register*
- number of states ≤ $2^{\text{number of flip-flops}}$
- CL (combinational logic) calculates next state and output
- Remember: The FSM follows exactly one edge per cycle.

So far we have learned how to implement in Verilog. Now we learn how to design “by hand” to the gate level.

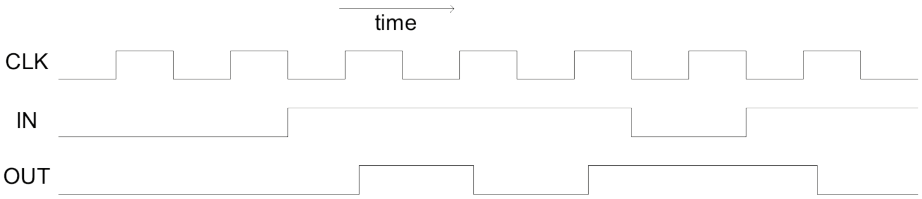
Parity Checker Example

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

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

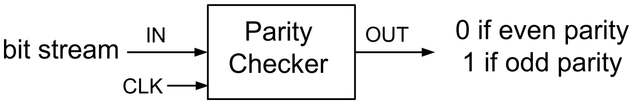


example:	0	0	1	1	1	0	1
	even	even	odd	even	odd	odd	even



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

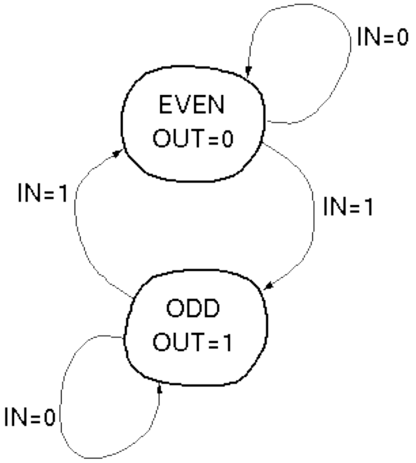
Formal Design Process



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

"State Transition Diagram"

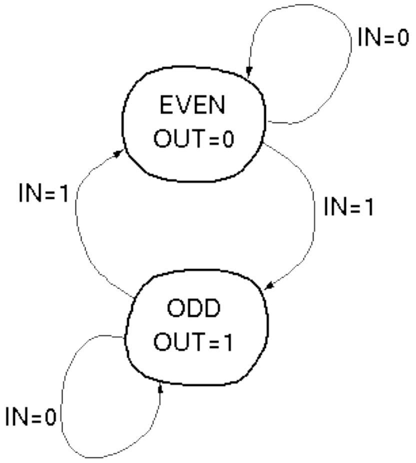
- circuit is in one of two "states".
- transition on each cycle with each new input, over exactly one arc (edge).
- Output depends on which state the circuit is in.



Formal Design Process

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

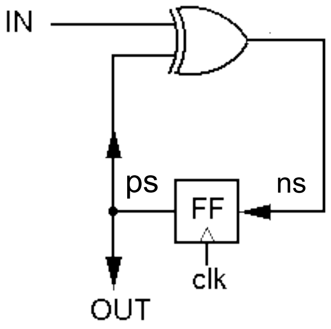
Formal Design Process

Logic equations from table:

OUT = PS

NS = PS xor IN

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



Formal Design Process

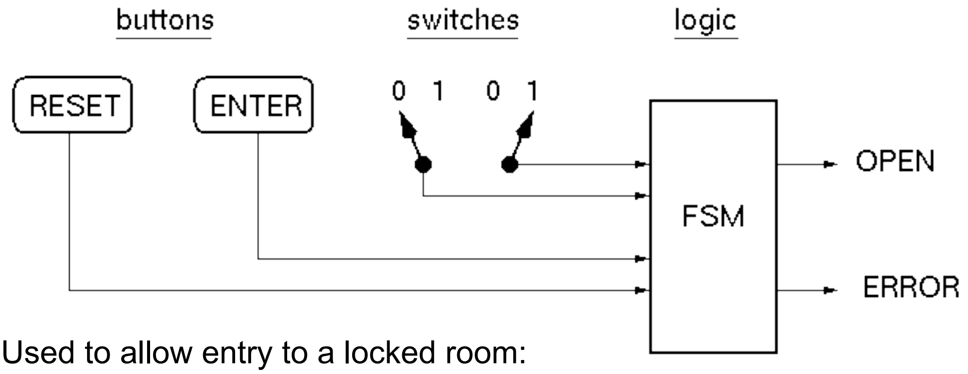
Review of Design Steps:

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:

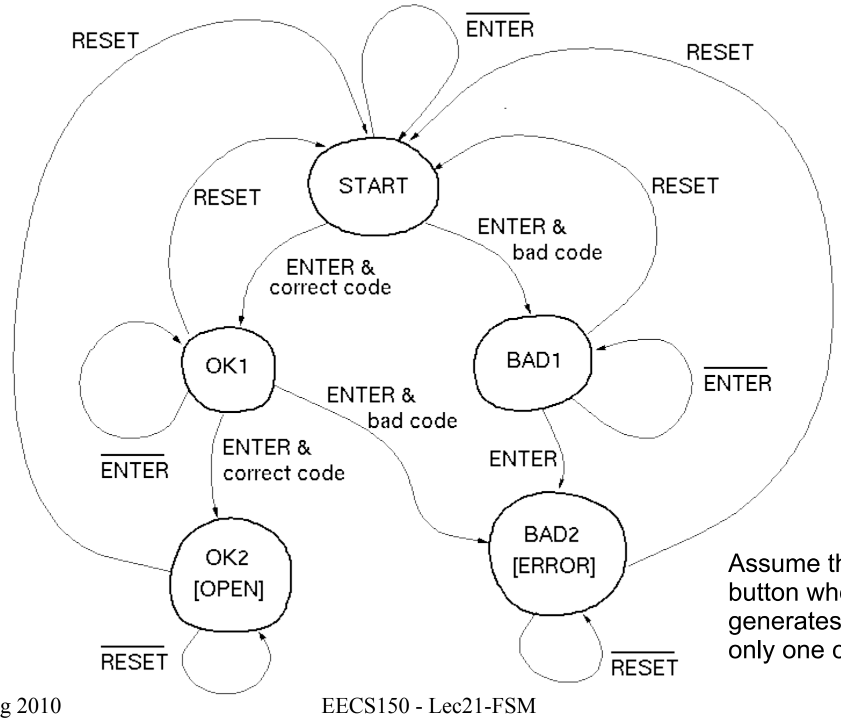
2-bit serial combination. Example 01,11:

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

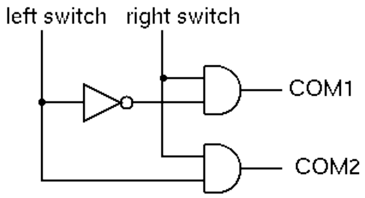

RESET
$\overline{\text{ENTER}}$
RESET
START
RESET
RESET
ENTER &
bad code
ENTER &
correct code
OK1
BAD1
$\overline{\text{ENTER}}$
ENTER &
bad code
ENTER &
correct code
$\overline{\text{ENTER}}$
ENTER
OK2
[OPEN]
BAD2
[ERROR]
$\overline{\text{RESET}}$
$\overline{\text{RESET}}$

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

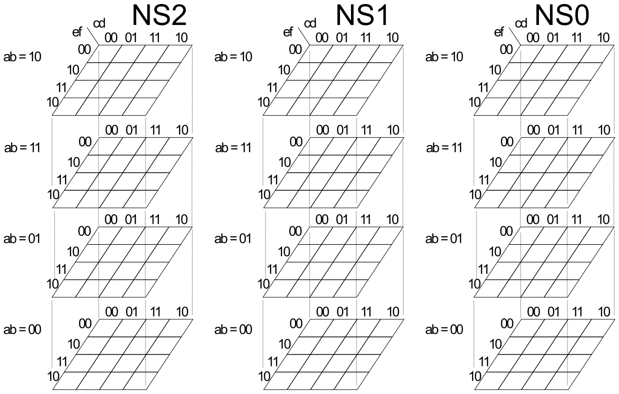
Decoder logic for checking combination (01,11):



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

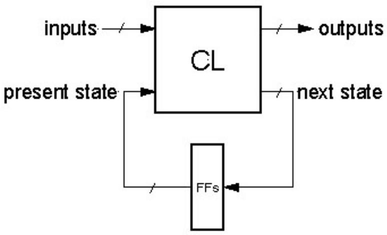
Encoded ST Table

- 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.



- What are the output functions for OPEN and ERROR?

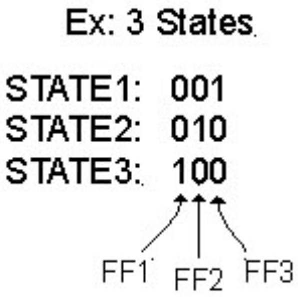
State Encoding



- In general:
  # of possible FSM state = $2^{\text{\# of FFs}}$
  Example:
  state1 = 01, state2 = 11, state3 = 10, state4 = 00
- However, often more than $\log_2$(# of states) FFs are used, to simplify logic at the cost of more FFs.
- Extreme example is one-hot state encoding.

State Encoding

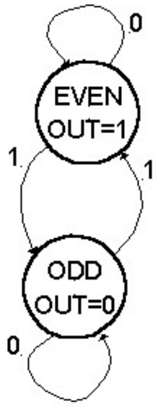
- **One-hot encoding of states.**
- One FF per state.

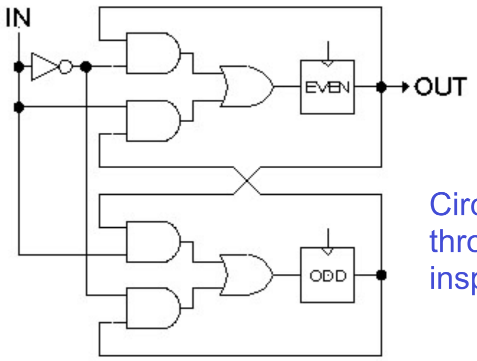


- Why one-hot encoding?
  - Simple design procedure.
    - Circuit matches state transition diagram (example next page).
  - Often can lead to simpler and faster "next state" and output logic.
- Why not do this?
  - Can be costly in terms of FFs for FSMs with large number of states.
- FPGAs are "FF rich",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

One-hot encoded FSM

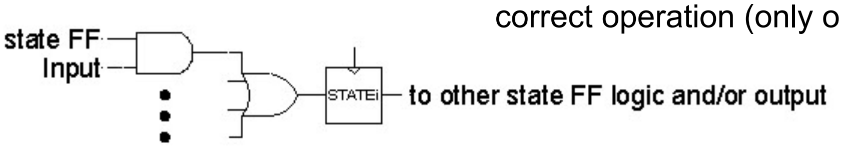
- Even Parity Checker Circuit:





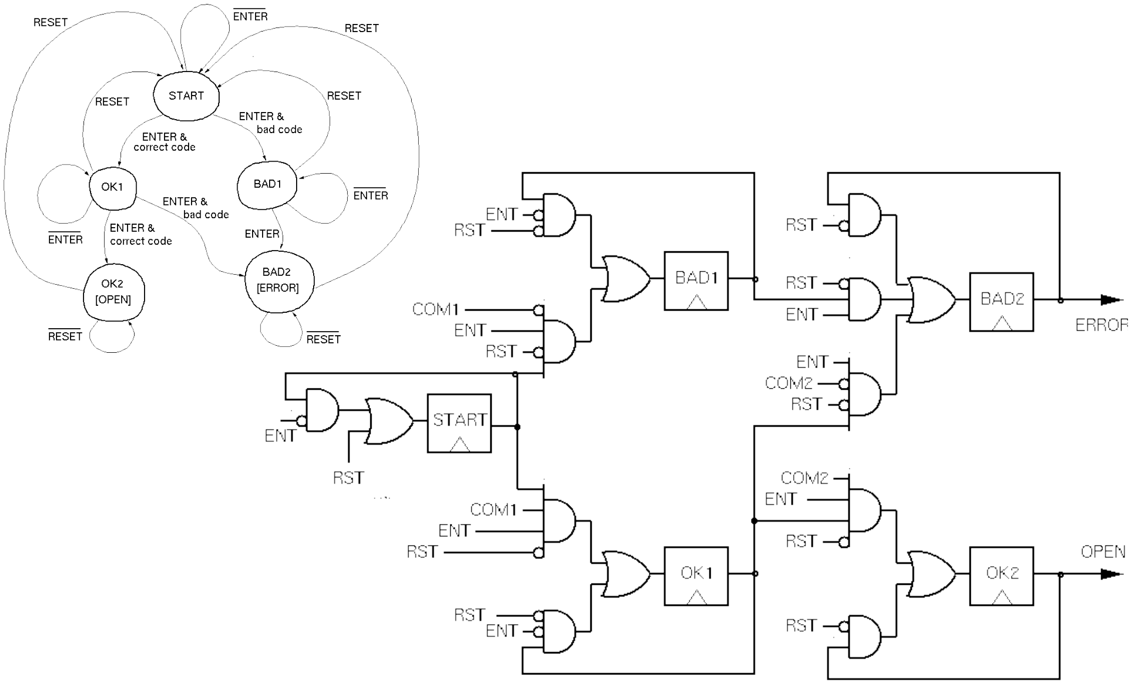
Circuit generated through direct inspection of the STD.

- In General:



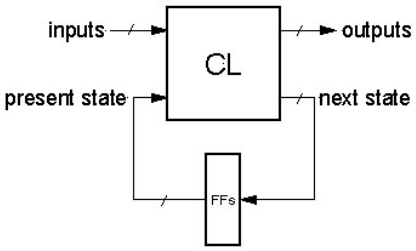
- FFs must be initialized for correct operation (only one 1)

One-hot encoded combination lock

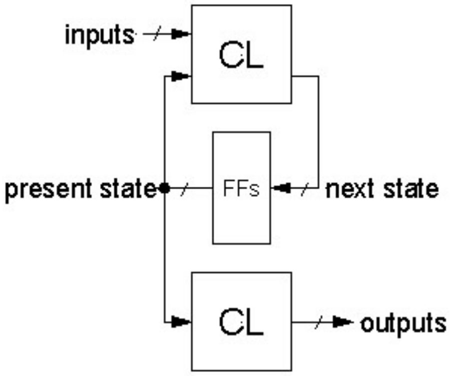


FSM Implementation Notes

- General FSM form:



- All examples so far generate output based only on the present state:

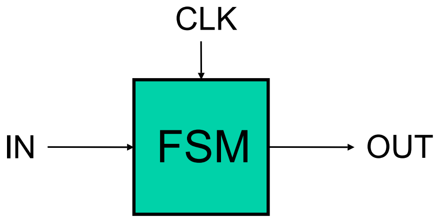


- Commonly name ***Moore Machine*** (If output functions include both present state and input then called a *Mealy Machine*)

Finite State Machines

- **Example: Edge Detector**

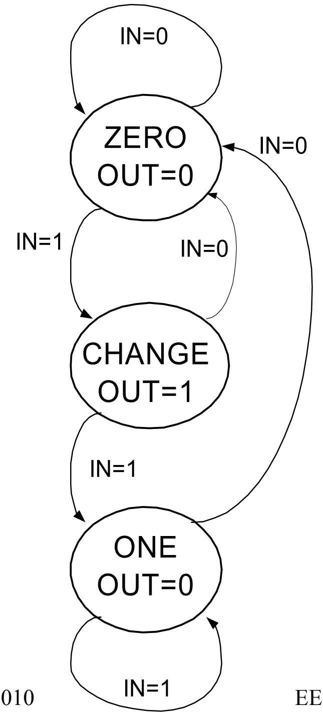
  Bit are received one at a time (one per cycle), such as: 000111010 → *time*



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

  Try two different solutions.

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	-

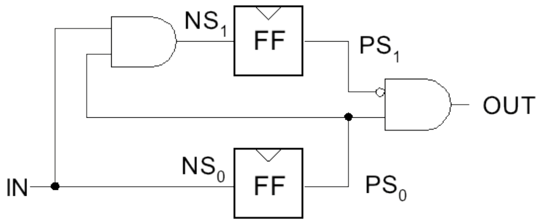
$NS_1 = IN\ PS_0$

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

$NS_0 = IN$

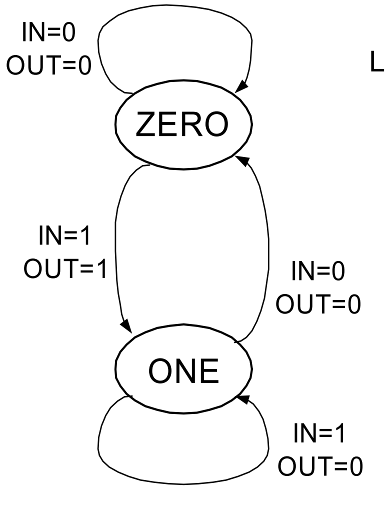
IN \ PS	00	01	11	10
0	0	1	0	-
1	0	1	0	-

$OUT = \overline{PS_1}\ PS_0$



Solution B

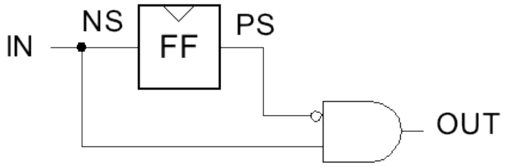
Output depends not only on PS but also on input, IN



Let ZERO=0, ONE=1

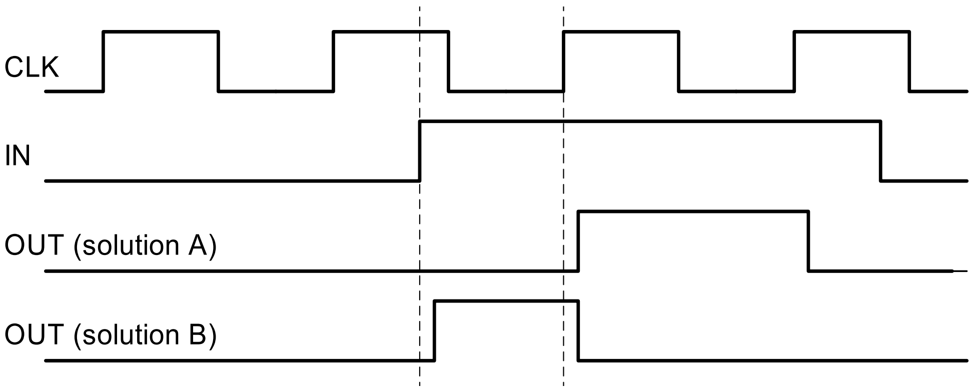
IN	PS	NS	OUT
0	0	0	0
0	1	0	0
1	0	1	1
1	1	1	0

NS = IN, OUT = IN PS’



What’s the *intuition* about this solution?

Edge detector timing diagrams



- Solution A: output follows the clock
- Solution B: output changes with input rising edge and is asynchronous wrt the clock.

FSM Comparison

Solution A

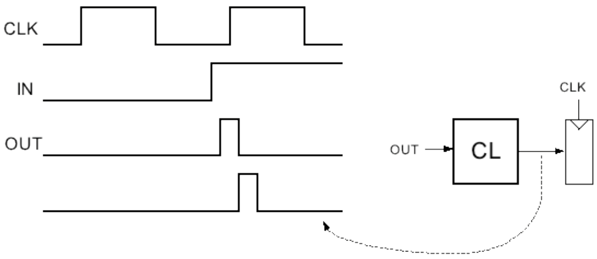
Moore Machine

- output function only of PS
- maybe more states (why?)
- synchronous outputs
  - no glitches
  - one cycle “delay”
  - full cycle of stable output

Solution B

Mealy Machine

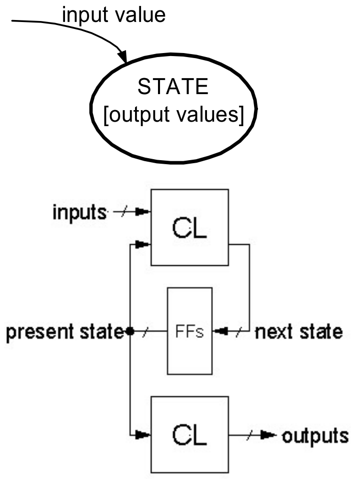
- output function of both PS & input
- maybe fewer states
- asynchronous outputs
  - if input glitches, so does output
  - output immediately available
  - output may not be stable long enough to be useful (below):



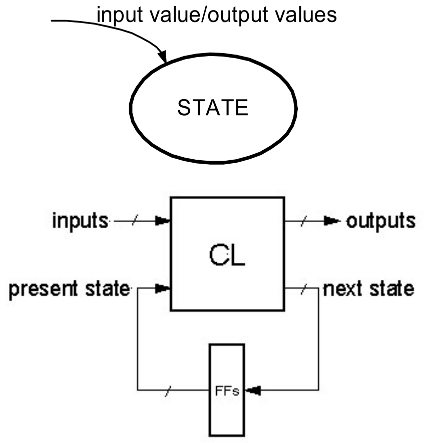
If output of Mealy FSM goes through combinational logic before being registered, the CL might delay the signal and it could be missed by the clock edge.

FSM Recap

Moore Machine



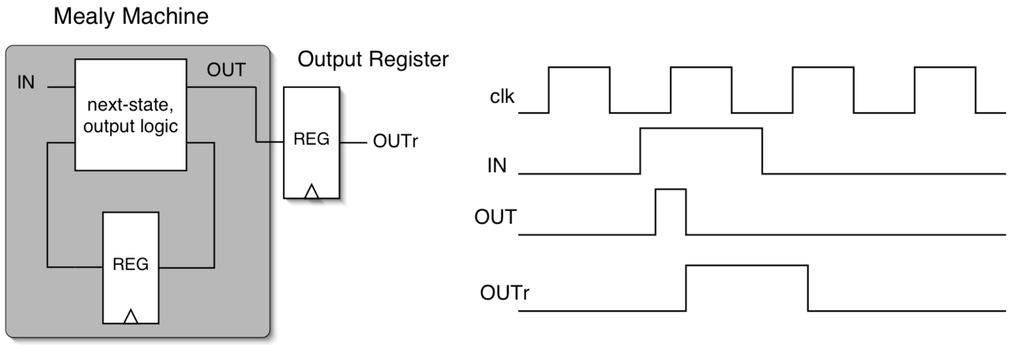
Mealy Machine



Both machine types allow one-hot *implementations.*

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:



General FSM Design Process with Verilog Implementation

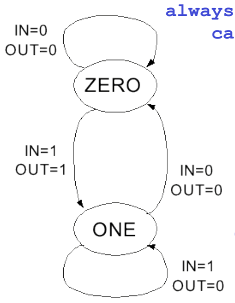
Design Steps:

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 separate always blocks for register assignment and CL logic block.
- ✓ Use case for CL block. Within each case section 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.*

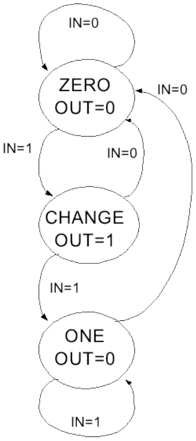
FSMs in Verilog

Mealy Machine



```
always @(posedge clk)
         if (rst) ps <= ZERO;
         else ps <= ns;
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



```
always @(posedge clk)
         if (rst) ps <= ZERO;
         else ps <= ns;
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
```