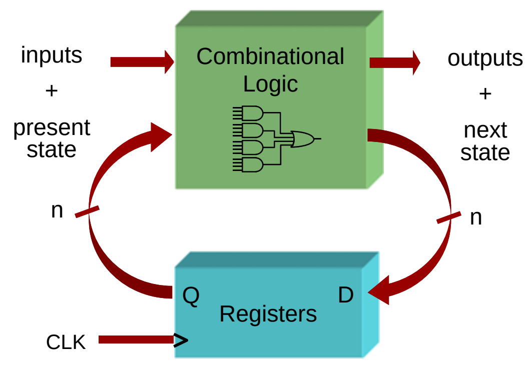
Finite State Machines

- Design methodology for sequential logic
  -- identify distinct states
  -- create state transition diagram
  -- choose state encoding
  -- write combinational Verilog for next-state logic
  -- write combinational Verilog for output signals
- Lots of examples

Reminder: Lab #2 due tonight!

Finite State Machines

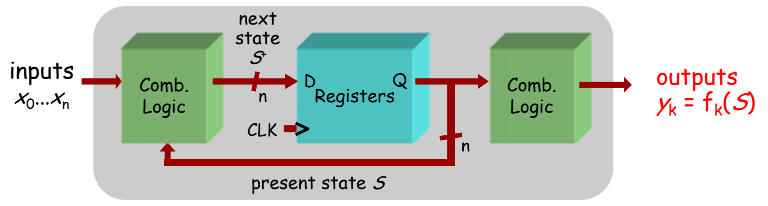
- Finite State Machines (FSMs) are a useful abstraction for sequential circuits with centralized "states" of operation
- At each clock edge, combinational logic computes *outputs* and *next state* as a function of *inputs* and *present state*



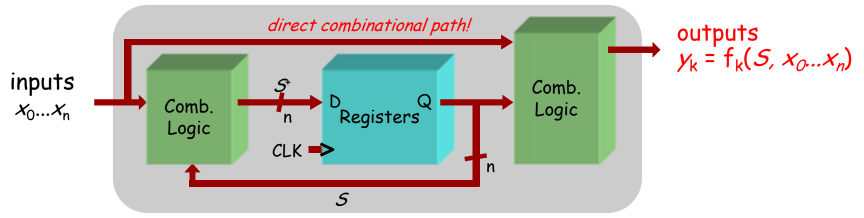
Two Types of FSMs

Moore and Mealy FSMs : different output generation

- Moore FSM:

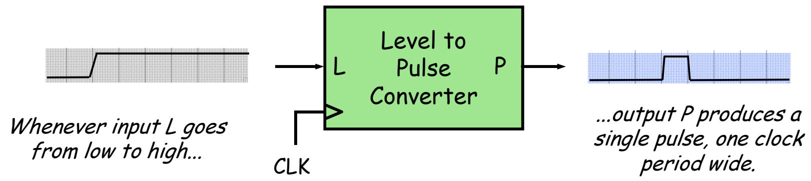


- Mealy FSM:



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

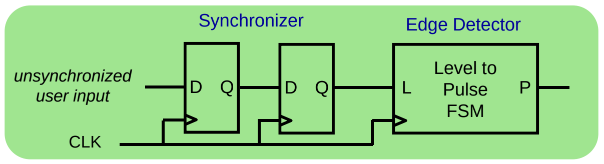


Whenever input L goes from low to high...

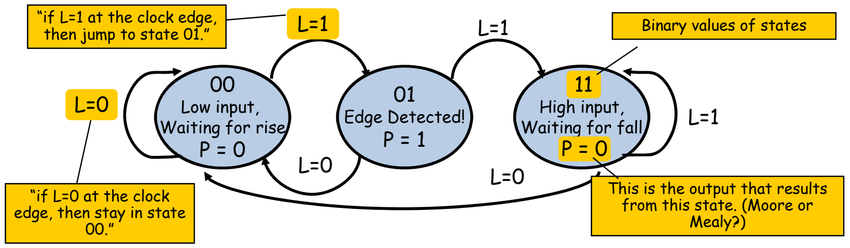
...output P produces a single pulse, one clock period wide.

Step 1: State Transition Diagram

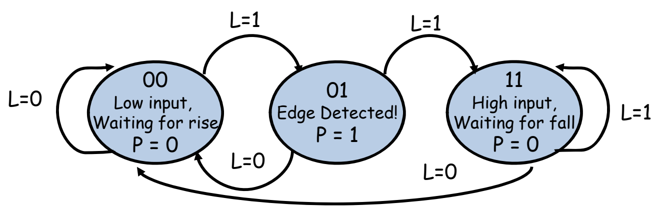
- Block diagram of desired system:



- 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?)

Valid State Transition Diagrams


L=1
L=1
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

- Arcs leaving a state are mutually exclusive, i.e., for any combination input values there's at most one applicable arc
- Arcs leaving a state are collectively exhaustive, i.e., for any combination of input values there's at least one applicable arc
- So for each state: for any combination of input values there's exactly one applicable arc
- Often a starting state is specified
- Each state specifies values for all outputs (Moore)

Choosing State Representation

Choice #1: binary encoding

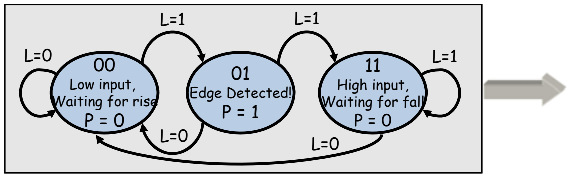
For N states, use ceil($\log_2 N$) bits to encode the state with each state represented by a unique combination of the bits.
Tradeoffs: most efficient use of state registers, but requires more complicated combinational logic to detect when in a particular state.

Choice #2: "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 often much less combinational logic since state decoding is trivial.

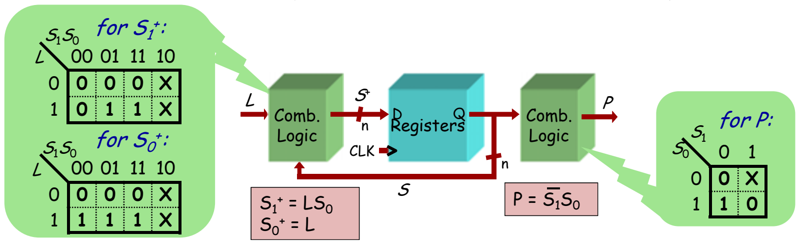
Step 2: Logic Derivation

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

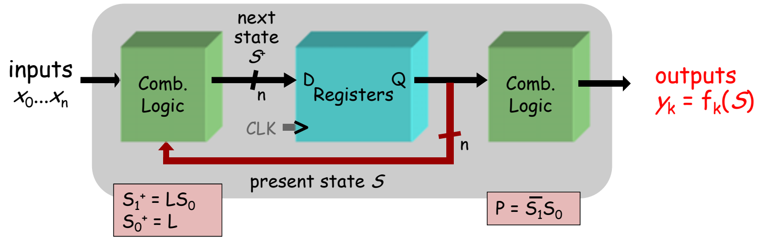

L=1
L=1
L=0
00 Low input, Waiting for rise P = 0
01 Edge Detected! P = 1
11 High input, Waiting for fall P = 0
L=1
L=0
L=0

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

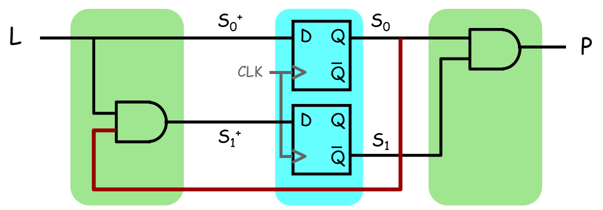
- Combinational logic may be derived using Karnaugh maps



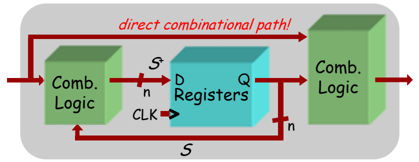
Moore Level-to-Pulse Converter



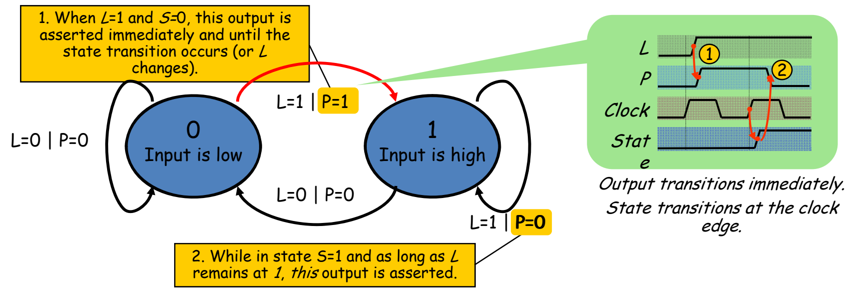
Moore FSM circuit implementation of level-to-pulse converter:



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

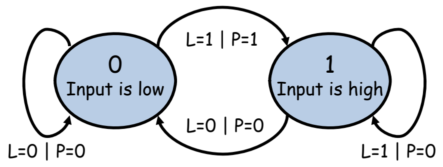


1. When *L*=1 and *S*=0, this output is asserted immediately and until the state transition occurs (or *L* changes).

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

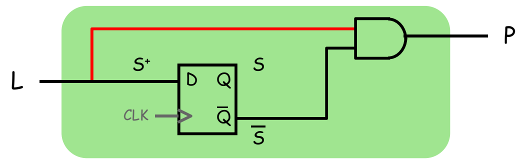
Output transitions immediately. State transitions at the clock edge.

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:

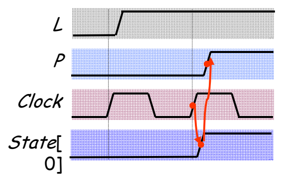


- FSM's state simply remembers the previous value of L
- Circuit benefits from the Mealy FSM's implicit single-cycle assertion of outputs during state transitions

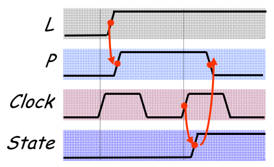
Moore/Mealy Trade-Offs

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

<u>Moore: delayed assertion of P</u>



<u>Mealy: immediate assertion of P</u>



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

Example: Intersection Traffic Lights

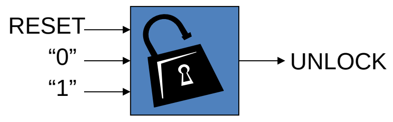
- Design a controller for the traffic lights at the intersection of two streets – two sets of traffic lights, one for each of the streets.
- Step 1: Draw starting state transition diagram. Just handle the usual green-yellow-red cycle for both streets. How many states? Well, how many different combinations of the two sets of lights are needed?
- Step 2: add support for a walk button and walk lights to your state transition diagram.
- Step 3: add support for a traffic sensor for each of the streets – when the sensor detects traffic the green cycle for that street is extended.

Example to be worked collaboratively on the board…

FSM Example

GOAL:

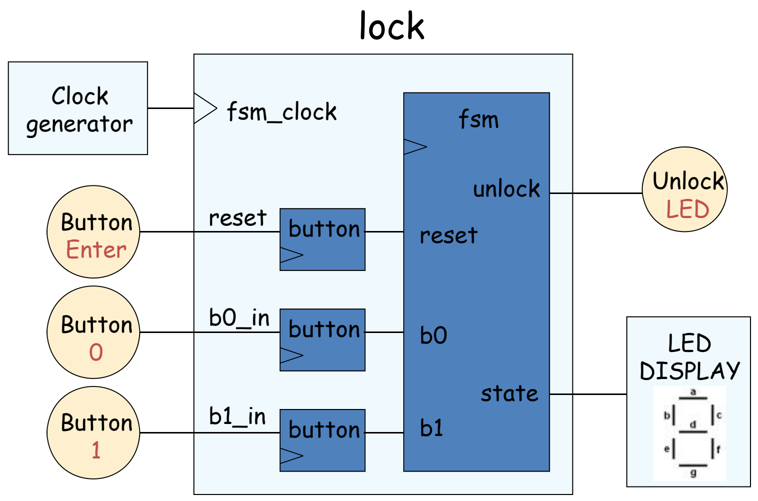
Build an electronic combination lock with a reset button, two number buttons (0 and 1), and an unlock output. The combination should be 01011.



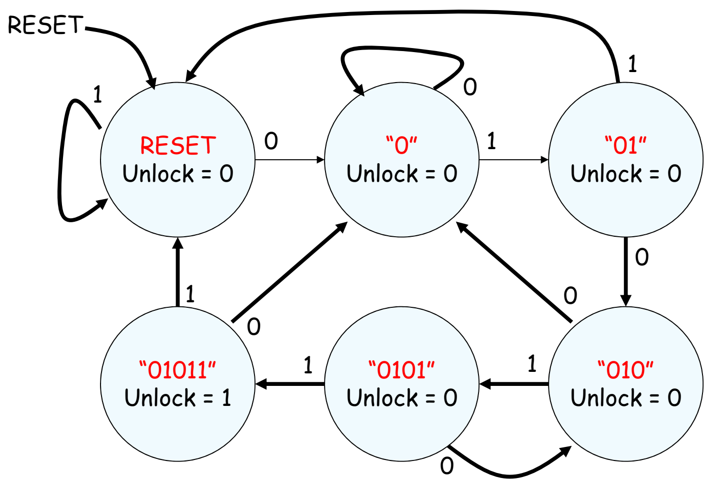
STEPS:

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

Step 1A: Block Diagram



Step 1B: State transition diagram



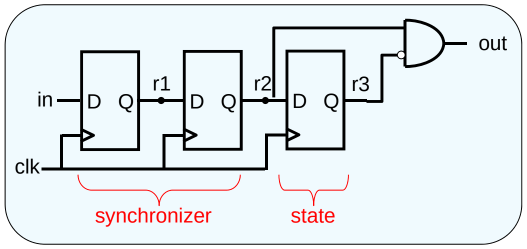
6 states → 3 bits

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
    always @(posedge clk) state <= next_state;

    // generate output
    assign out = ???;

    // debugging?
endmodule
```

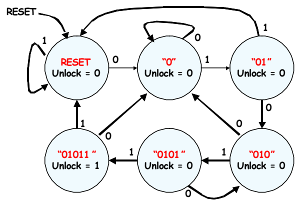
Step 2A: Synchronize buttons

```
// 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
```



Step 2B: state transition diagram

```
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;
```



Step 2C: generate output

```
// it's a Moore machine!  Output only depends on current state

assign out = (state == S_01011);
```

Step 2D: debugging?

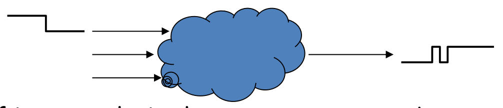
```
// hmmm.  What would be useful to know?  Current state?

assign hex_display = {1'b0,state[2:0]};
```

Step 2: final Verilog implementation

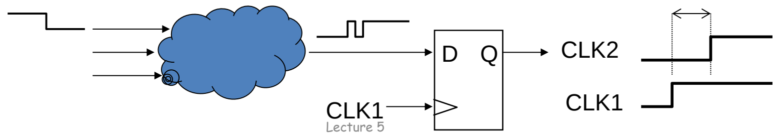
```
module lock(input clk,reset_in,b0_in,b1_in,
            output out, output [3:0] hex_display);

  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
  assign hex_display = {1'b0,state};      // debugging
endmodule
```

Where should CLK come from?

- Option 1: external crystal
  - Stable, known frequency, typically 50% duty cycle
- Option 2: internal signals
  - Option 2A: output of combinational logic



    - No! If inputs to logic change, output may make several transitions before settling to final value → several rising edges, not just one! Hard to design away output glitches…
  - Option 2B: output of a register
    - Okay, but timing of CLK2 won't line up with CLK1


D
Q
CLK1
CLK2
CLK1