

Intro to Digital Design

L5: Finite State Machines

Instructor: Naomi Alterman

Teaching Assistants:

Derek de Leuw

Isabel Froelich

Kevin Hernandez

Sathvik Kanuri

Aadithya Manoj

Administrivia

- ❖ Quiz 1 today, grades out Friday
 - Both the quiz and solutions will be added to the question bank on the course website
- ❖ Lab 5 – Verilog implementation of FSMs
 - Step up in difficulty from Labs 1-4 (worth 100 points)
 - Bonus points for minimal logic
 - Simplification through *design* (Verilog does the rest)

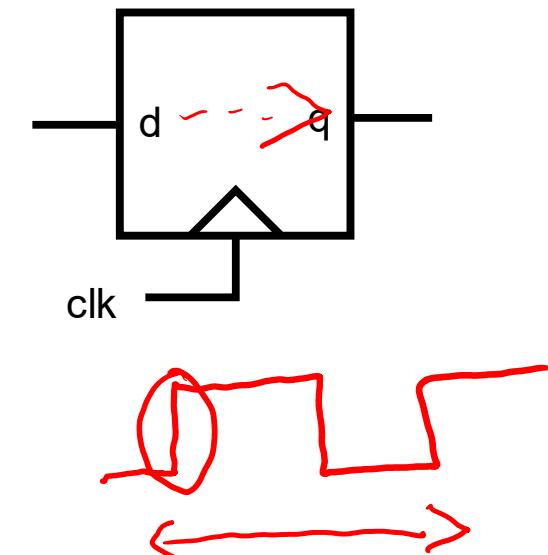
Outline

- ❖ Sequential Logic in Verilog
- ❖ Finite State Machines
- ❖ FSMs in Verilog

Reminder: Flip Flops

- ❖ A single bit of memory
- ❖ Copy d to q on the rising edge of the clock signal

```
module DFF (q, d, clk);
    output logic q; // q is state-holding
    input logic d, reset, clk;
    always ff @ (posedge clk) begin
        q <= d;
    end
endmodule
```



Reminder: “`always_comb`” blocks

- ❖ Verilog requires us to wrap control flow statements in an **`always_comb` block**
 - Block defines the full set of circuits that *may* drive the value on a `logic` variable
 - Idea: the last assignment in an always block to a given variable is the result that gets used
- ❖ But I promised there were more species of “`always`” block...

Sequential “always” blocks

- ❖ General `always` blocks:

- “Always” running, but its output only gets sampled when signals in the *sensitivity list* change:

`always` `@ (posedge clk)`

- ❖ `always_ff`:

- Forces SystemVerilog to use flip flops as the stateful element in the circuit
 - **Only** use `always_ff` for sequential logic in this class, never `always`

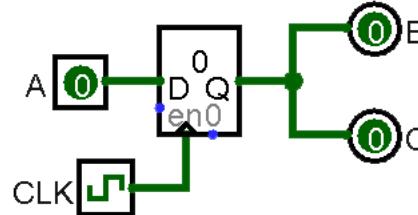
`always_ff` `@ (posedge clk)`



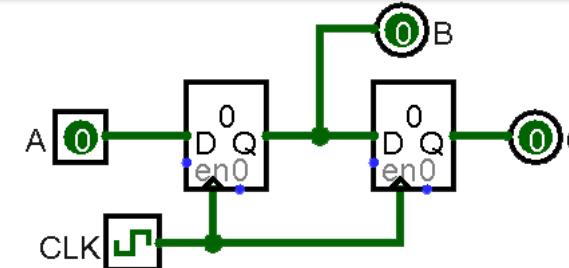
Blocking vs. Nonblocking

- ❖ **Blocking statement (=):** statements executed sequentially
 - Resembles programming languages
- ❖ **Nonblocking statement ($<=$):** statements executed “in parallel”
 - Resembles hardware
- ❖ Example:

```
always_ff @ (posedge clk)
begin
    b = a;
    c = b;
end
```



```
always_ff @ (posedge clk)
begin
    b <= a;
    c <= b;
end
```



SystemVerilog Coding Guidelines

- 1) When modeling sequential logic with an `always_ff` block, use *nonblocking* assignments (`<=`)
- 2) When modeling combinational logic with an `always_comb` block, use *blocking* assignments (`=`)
- 3) When modeling both sequential and combinational logic within the same `always_ff` block, use *nonblocking* assignments
- 4) Do not mix *blocking* and *nonblocking* assignments in the same `always_*` block
- 5) Do not make assignments to the same variable from more than one `always_*` block

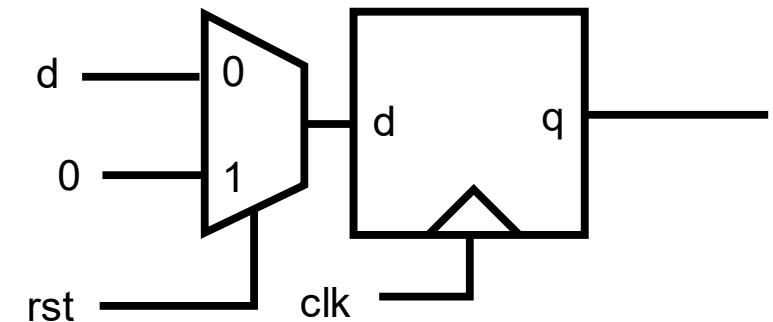
Verilog: Reset Functionality

- ❖ Option 1: synchronous reset

```
module DFFR (q, d, reset, clk);
    output logic q; // q is state-holding
    input logic d, reset, clk;

    always_ff @(posedge clk)
        if (reset)
            q <= 0;           // on reset, set to 0
        else
            q <= d;           // otherwise pass d to q

endmodule
```



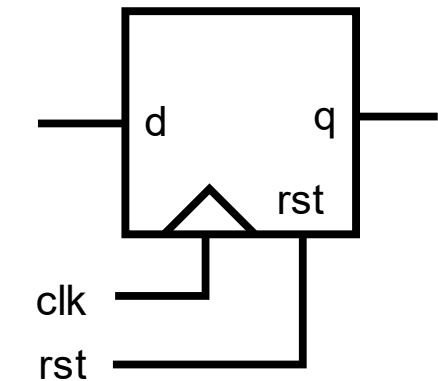
Verilog: Reset Functionality

- ❖ Option 2: asynchronous reset

```
module DFFaR (q, d, reset, clk);
    output logic q; // q is state-holding
    input logic d, reset, clk;

    always_ff @ (posedge clk or posedge reset)
        if (reset)
            q <= 0;           // on reset, set to 0
        else
            q <= d;           // otherwise pass d to q

endmodule
```



Verilog: Simulated Clock, three ways

- ❖ In our testbenches we need to generate a clock signal
 - Use “forever” block to loop testbench code for whole simulation

Explicit Edges:

```
initial begin
    clk = 0;
    forever begin
        #50    clk = 1;
        #50    clk = 0;
    end
end
```

Clock period → #100

Toggle:

```
initial begin
    clk = 0;
    forever begin
        #50    clk = ~clk;
    end
end
```

Parameterized clock period:

```
parameter period = 100;
initial begin
    clk = 0;
    forever begin
        #(period/2)    clk = ~clk;
    end
end
```

Verilog Testbench with Clock

```
module D_FF_testbench;
  logic CLK, reset, d;
  logic q;

  parameter PERIOD = 100;

  D_FF dut (.q, .d, .reset, .CLK); // Instantiate the D_FF

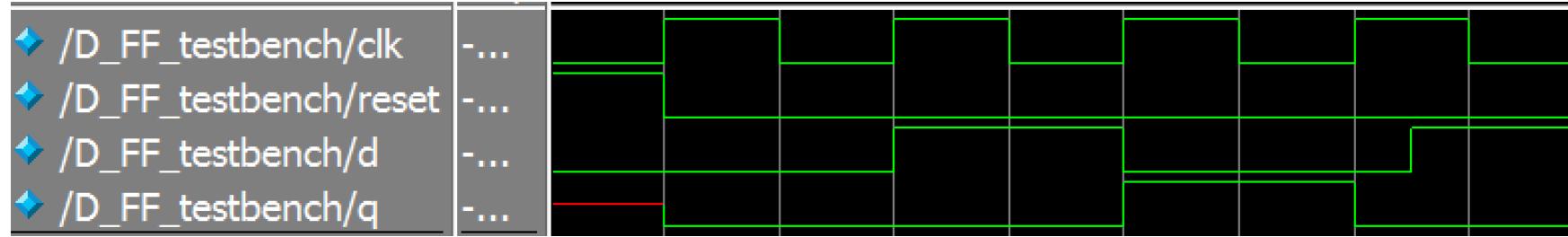
  initial CLK <= 0; // Set up clock
  always #(PERIOD/2) CLK = ~CLK;
  forever
  initial begin // Set up signals
    d <= 0; reset <= 1;
    @ (posedge CLK); reset <= 0;
    @ (posedge CLK); d <= 1;
    @ (posedge CLK); d <= 0;
    @ (posedge CLK); #(PERIOD/4) d <= 1;
    @ (posedge CLK);
    $stop(); // end the simulation
  end
endmodule
```

clk
generator

Simulation Timing Controls

- ❖ **Delay:** `#<time>`
 - Delays by a specific amount of simulation time
 - Can do calculations in `<time>`
 - Examples: `#(PERIOD/4), #50`
- ❖ **Edge-sensitive:** `@(<pos/negedge> signal)`
 - Delays next statement until specified transition on signal
 - Example: `@(posedge CLK)`
- ❖ **Level-sensitive Event:** `wait(<expression>)`
 - Delays next statement until `<expression>` evaluates to TRUE
 - Example: `wait(enable == 1)`

ModelSim Waveforms



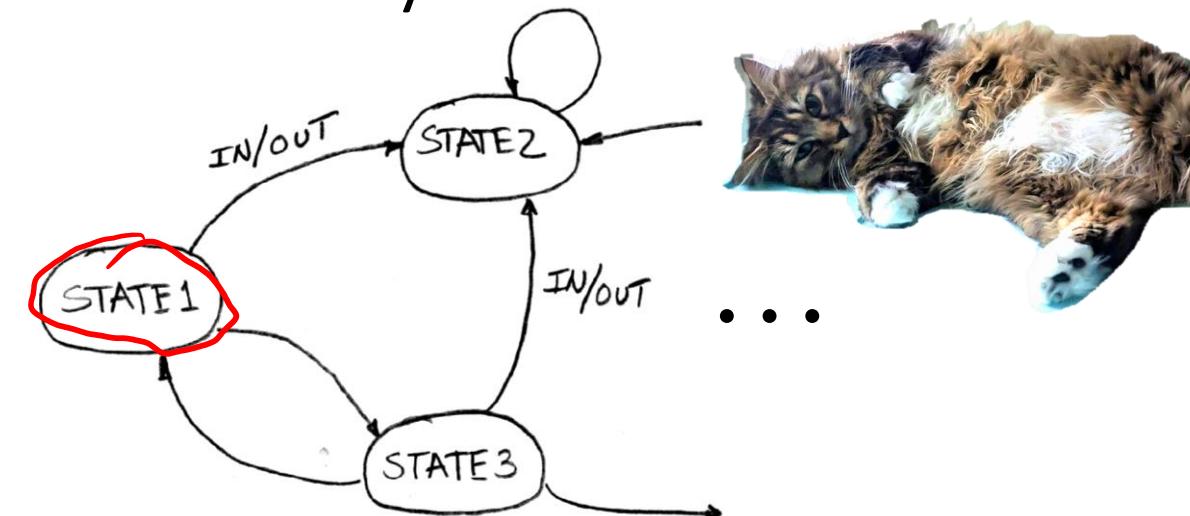
```
initial begin
    d <= 0; reset <= 1;
    @ (posedge CLK);           reset <= 0;
    @ (posedge CLK); d <= 1;
    @ (posedge CLK); d <= 0;
    @ (posedge CLK); #(PERIOD/4) d <= 1;
    @ (posedge CLK);
    $stop();
end
```

Outline

- ❖ Sequential Logic in Verilog
- ❖ **Finite State Machines**
- ❖ FSMs in Verilog

Finite State Machines (FSMs)

- ❖ A convenient way to conceptualize computation over time
 - Function can be represented with a *state transition diagram*
 - The state represents “what step we’re at” in a procedure
 - The transitions represent time advancing (in units of clock cycles)
 - You’ve seen these before in CSE311
- ❖ **New for CSE369:** Implement FSMs in hardware as synchronous digital systems
 - Flip-flops/registers hold “state”
 - Controller (state update, I/O) implemented in combinational logic



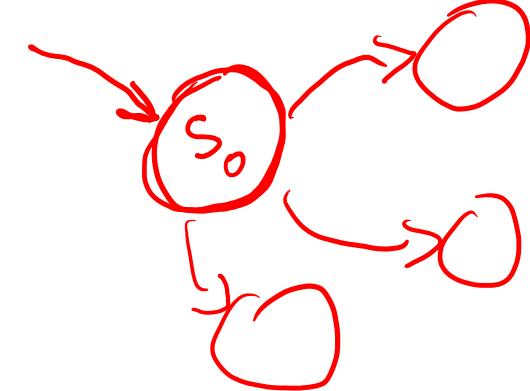
FSMs, philosophically

- ❖ George Mealy defined¹ data like this:
 1. There is the real world, as it is
 2. There are the mental conceptions of that world, that exist in our heads
 3. There are *fragments* of those conceptions, and that is what we call data **data**
- ❖ He also invented “Mealy Machine” FSMs, but that’s where his head was at 

¹G. H. Mealy, "Another Look at Data," AFIPS, pp. 525-534, 1967 Proceedings of the Fall Joint Computer Conference, 1967.
<http://doi.ieeecomputersociety.org/10.1109/AFIPS.1967.112>

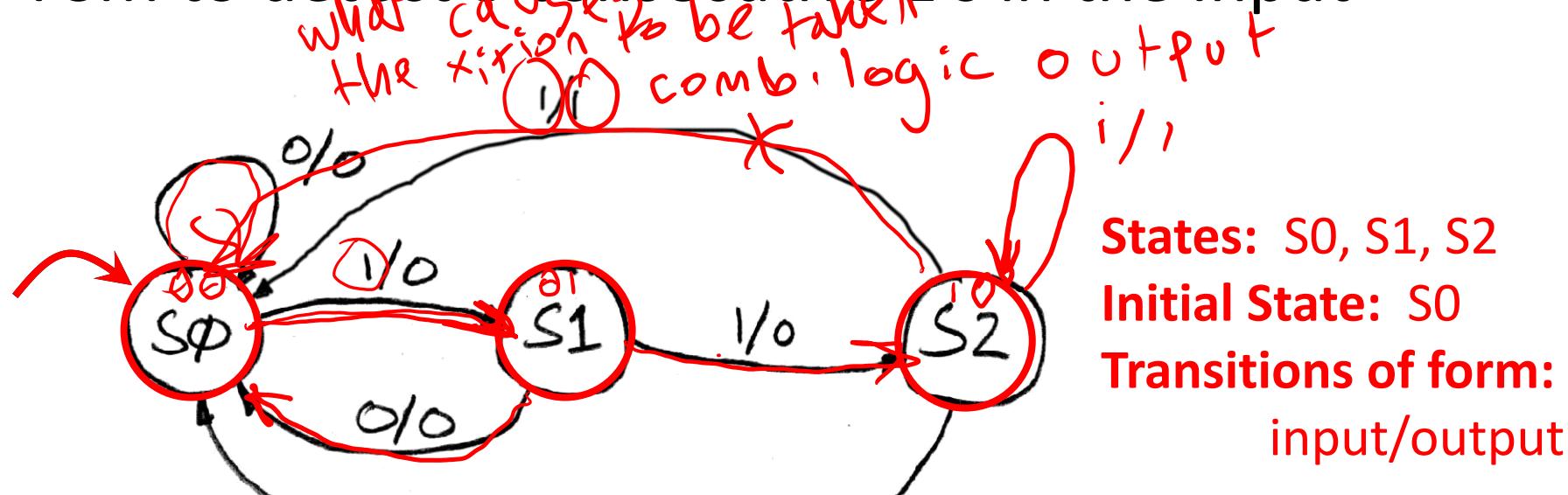
State Diagrams

- ❖ Our digital logic state diagrams are defined by:
 - A set of *states* S (circles)
 - A *transition function* that maps from the current input and current state to the output and the next state (arrows between states)
 - An *initial state* s_0 (only arrow not between states)
- ❖ State transitions are controlled by the clock:
 - On each clock cycle the machine checks the inputs and generates a new state (could be same) and new output
- ❖ **Note:** We cover Mealy machines here; Moore machines put outputs on states, not transitions



Example: Buggy 3 Ones FSM

- ❖ FSM to detect 3 consecutive 1's in the Input



States: S0, S1, S2

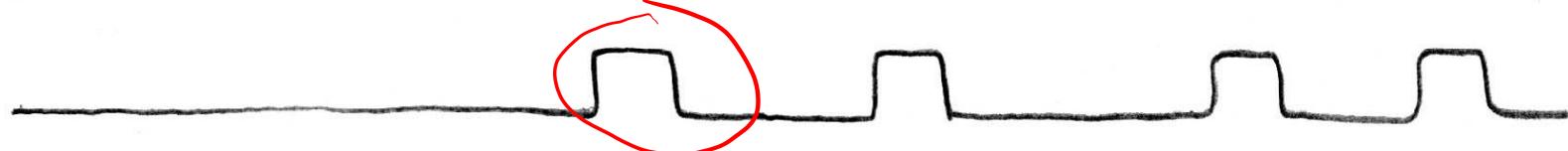
Initial State: S0

Transitions of form: input/output

INPUT

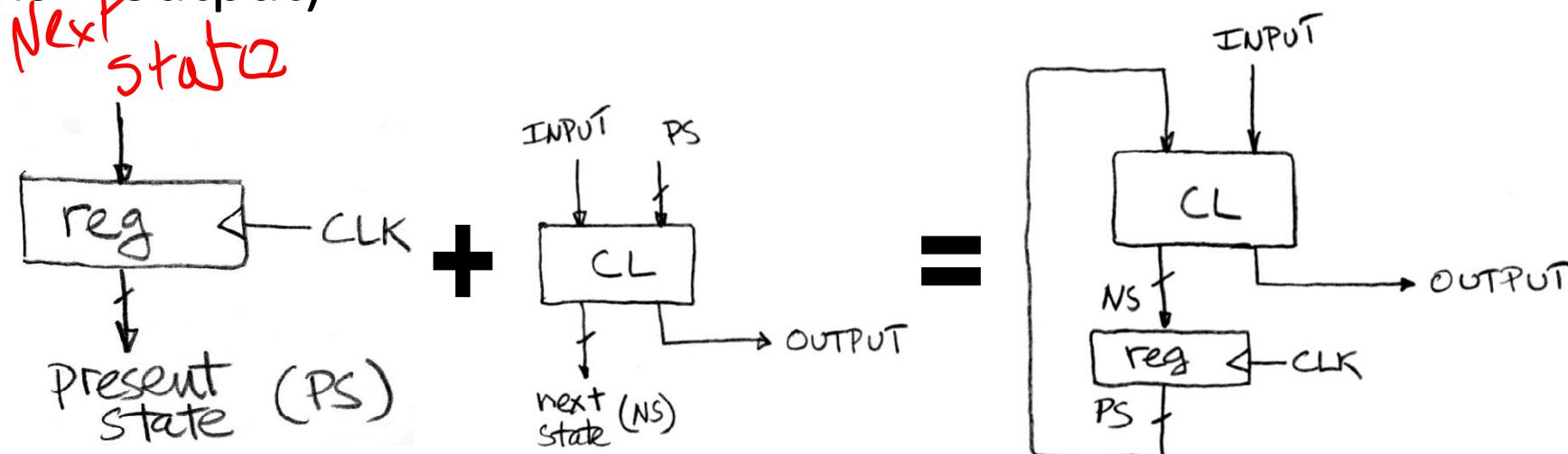


OUTPUT



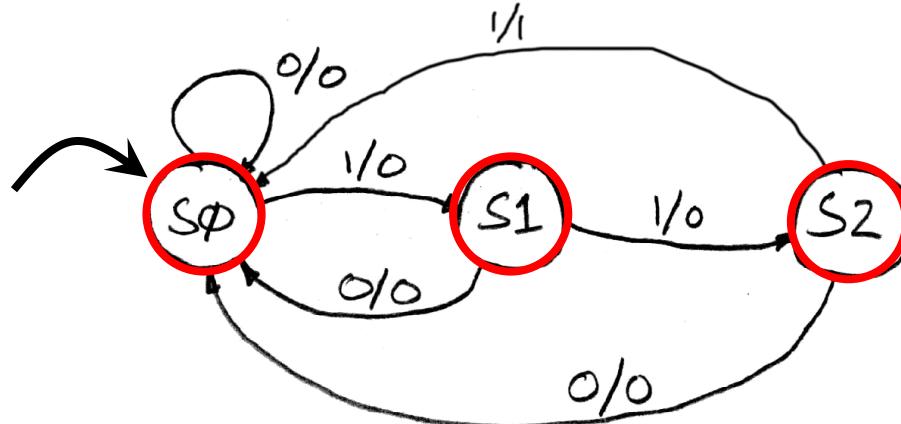
Hardware Implementation of FSM

- ❖ Register holds a representation of the FSM's state
 - Must assign a *unique* bit pattern for each state
 - Output is *present/current state* (PS/CS)
 - Input is *next state* (NS)
- ❖ Combinational Logic implements transition function (state transitions + output)



FSM: Combinational Logic

- ❖ Read off transitions into Truth Table!
 - **Inputs:** Present State (PS) and Input (In)
 - **Outputs:** Next State (NS) and Output (Out)



PS	In	NS	Out
00	0	00	0
00	1	01	0
01	0	00	0
01	1	10	0
10	0	00	0
10	1	00	1

↑↑ ↑

- ❖ Implement logic for *EACH* output (2 for NS, 1 for Out)



FSM: Logic Simplification

NS₁ NS₀

PS	In	NS	Out
00	0	00	0
00	1	01	0
01	0	00	0
01	1	10	0
10	0	00	0
10	1	00	1
11	0	XX	X
11	1	XX	X

$$NS_1 = PS_0 \cdot In$$

PS

In	00	01	11	10
0	0	0	X	0
1	0	0	X	0

PS

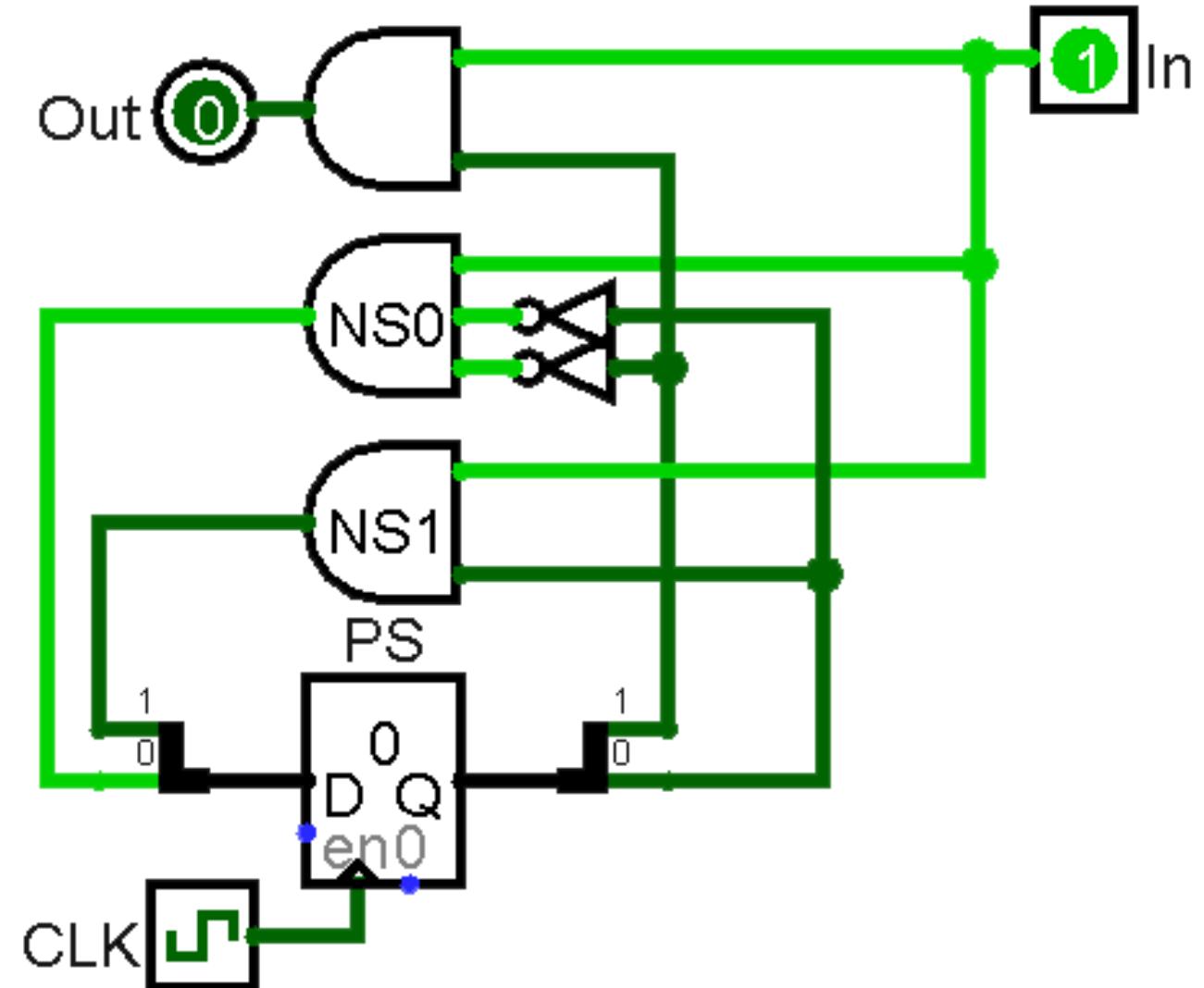
In	00	01	11	10
0				
1				

PS

In	00	01	11	10
0				
1				

FSM: Implementation

- ❖ $NS_1 = PS_0 \cdot In$
- ❖ $NS_0 = \overline{PS_1} \cdot \overline{PS_0} \cdot In$
- ❖ $Out = PS_1 \cdot In$
- ❖ How do we test the FSM?
 - “Take” every *transition* that we care about!



State Diagram Properties

- ❖ For S states, how many state bits do I use?

$$\lceil \log_2(S) \rceil$$

- ❖ For I inputs, what is the *maximum* number of transition arrows on the state diagram?

$$S(2^I)$$

- Can sometimes combine transition arrows
- Can sometimes omit transitions (don't cares)
- ❖ For s state bits and I inputs, how big is the truth table?

$$2^{\frac{s+I}{s+I}}$$