

Intro to Digital Design

Finite State Machines

Instructor: Justin Hsia

Teaching Assistants:

Caitlyn Rawlings

Emilio Alcantara

Naoto Uemura

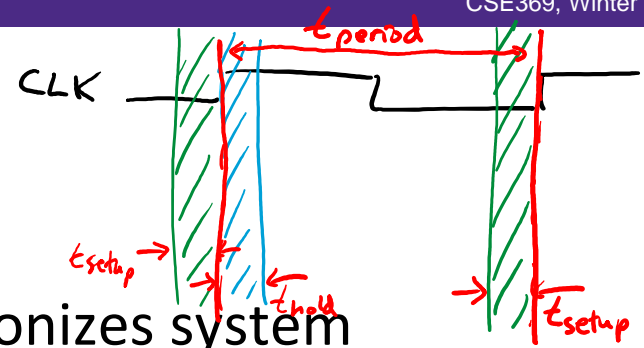
Donovan Clay

Joy Jung

Relevant Course Information

- ❖ Quiz 1 grades should be out on Gradescope tonight
 - 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 110 max possible
 - Simplification through *design* (Verilog does the rest)

Review of Timing Terms



- ❖ **Clock:** steady square wave that synchronizes system
- ❖ **Flip-flop:** one bit of state that samples every rising edge of CLK (positive edge-triggered)
- ❖ **Register:** several bits of state that samples on rising edge of CLK (positive edge-triggered); often has a RESET
- ❖ **Setup Time:** when input must be stable *before* CLK trigger
- ❖ **Hold Time:** when input must be stable *after* CLK trigger
- ❖ **CLK-to-Q Delay:** how long it takes output to change from CLK trigger

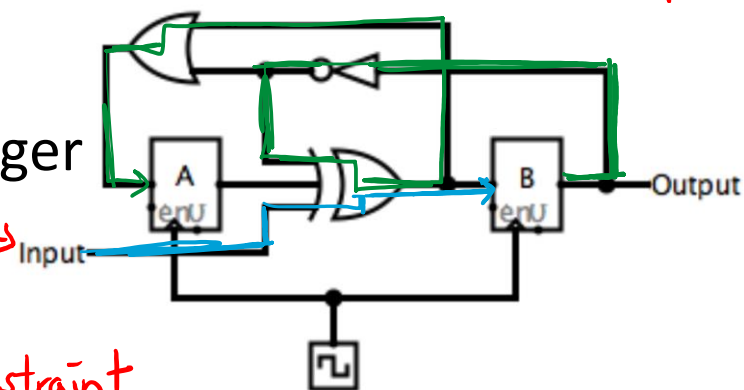
SDS Timing Question (all times in ns)

$$t_{hold} \leq t_{input,i} \leq t_{period} - t_{setup}$$

❖ The circuit below has the following timing parameters

- $t_{period} = 20, t_{setup} = 2$
- $t_{XOR} = t_{OR} = 5, t_{NOT} = 4$
- Input changes 1 ns after clock trigger

— longest path to reg input
— shortest path to reg input



❖ What is the max t_{C2Q} ?

t_{setup} constraint
 $t_{input,n}$ - last time a register input changes

$$t_{C2Q} + t_{NOT} + t_{XOR} + t_{OR} \leq t_{period} - t_{setup}$$

$$4 + 5 + 5 \leq 20 - 2$$

$$t_{C2Q} \leq 4 \text{ ns}$$

❖ If $t_{C2Q} = 3$, what is the max t_{hold} ?

t_{hold} constraint

$$1 \text{ ns} + t_{XOR} \geq t_{hold}$$

$$1 + 5 \geq t_{hold}$$

$t_{input,1}$ - first time a register input changes

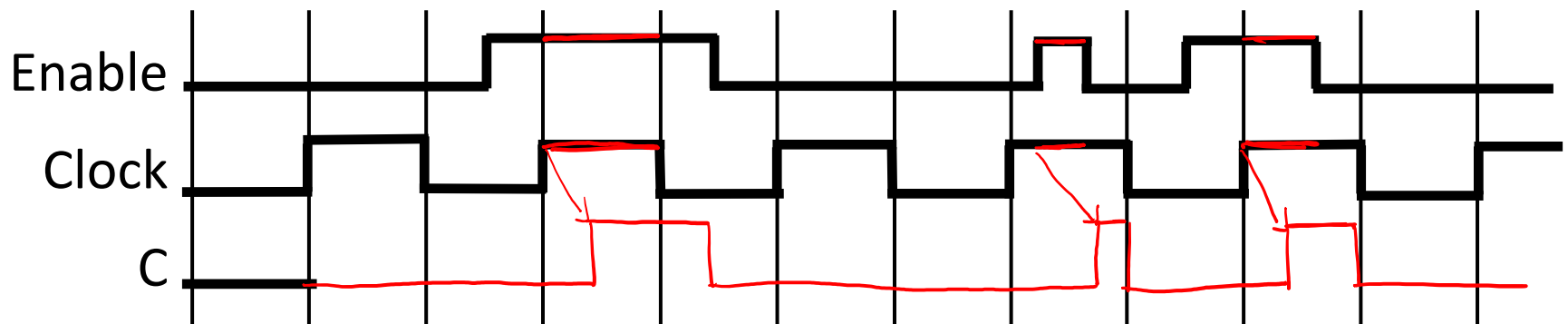
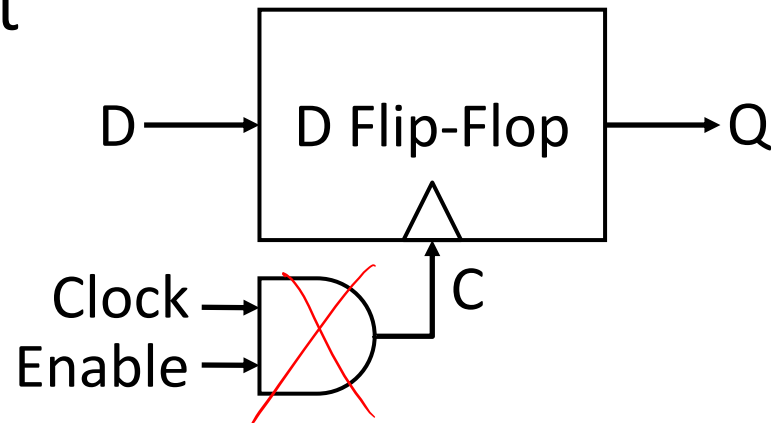
$$t_{hold} \leq 6 \text{ ns}$$

Outline

- ❖ **Flip-Flop Realities**
- ❖ Finite State Machines
- ❖ FSMs in Verilog

Flip-Flop Realities: Gating the Clock

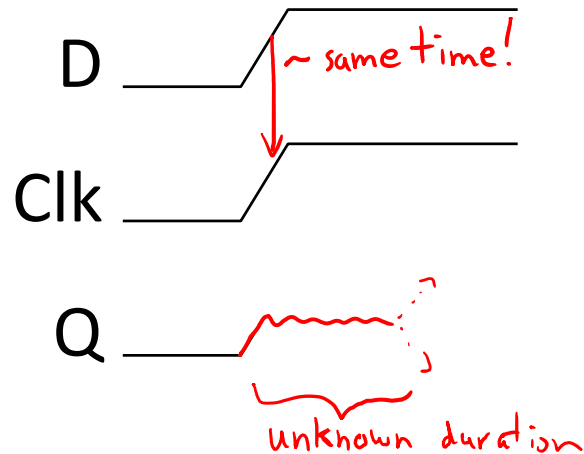
- ❖ Delay can cause part of circuit to get out of sync with rest
 - More timing headaches!
 - Adds to *clock skew*
- ❖ Hard to track non-uniform triggers



- ❖ **NEVER GATE THE CLOCK!!!**

Flip-Flop Realities: External Inputs

- ❖ External inputs aren't synchronized to the clock
 - If not careful, can violate timing constraints
- ❖ What happens if input changes around clock trigger?

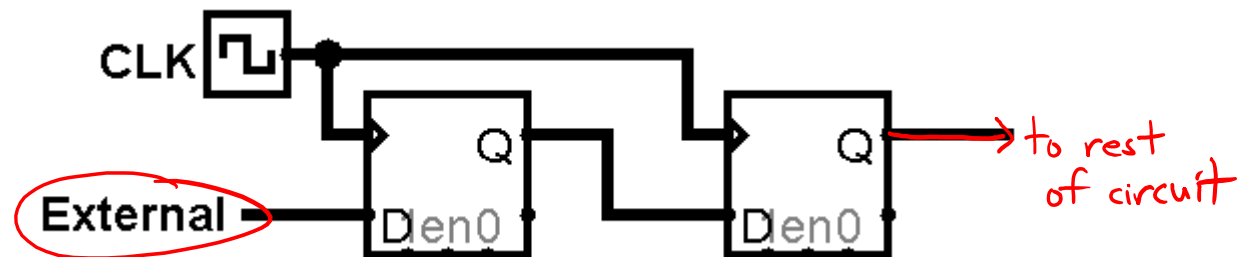


could/will violate setup or hold constraint

"Metastability"

Flip-Flop Realities: Metastability

- ❖ **Metastability** is the ability of a digital system to persist for an unbounded time in an unstable equilibrium or metastable state
 - Circuit may be unable to settle into a stable '0' or '1' logic level within the time required for proper circuit operation
 - Unpredictable behavior or random value
 - https://en.wikipedia.org/wiki/Metastability_in_electronics
- ❖ State elements can help reject transients
 - Longer chains = more rejection, but longer signal delay

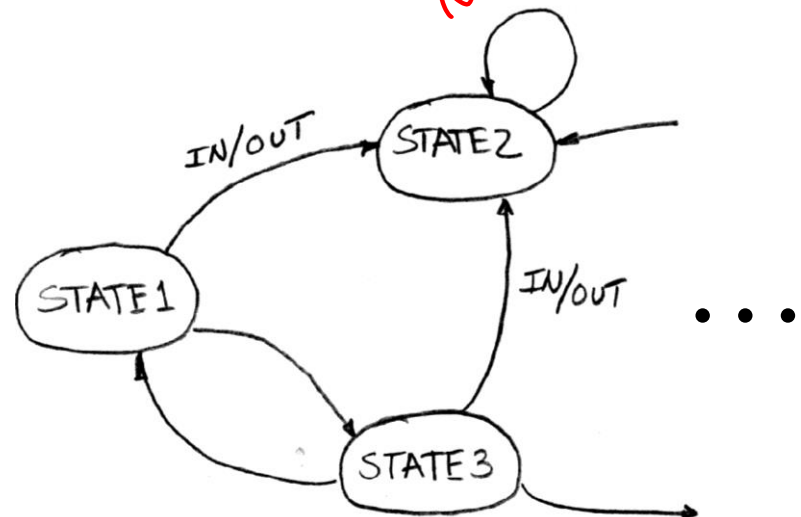


Outline

- ❖ Flip-Flop Realities
- ❖ **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*
 - 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

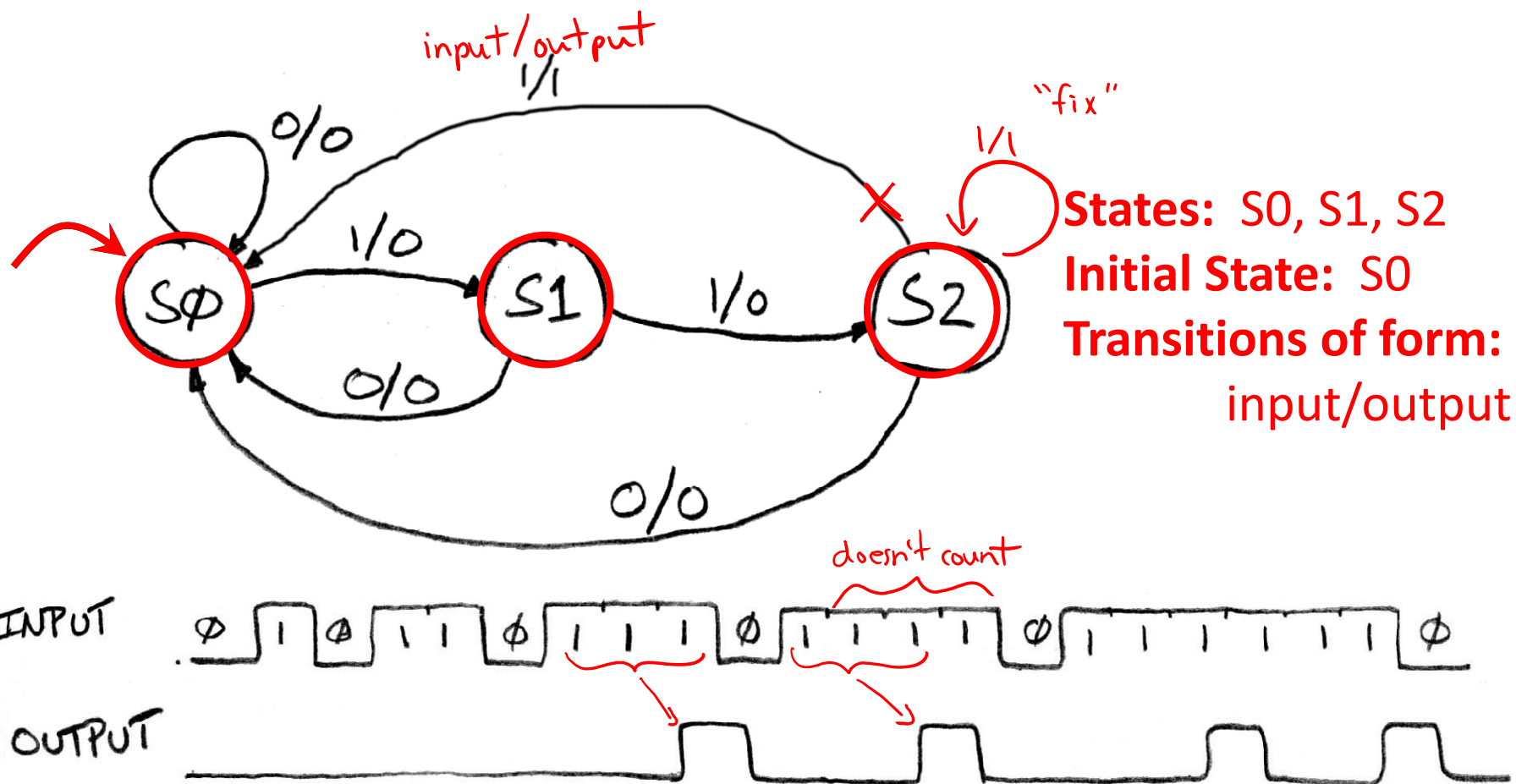


State Diagrams

- ❖ An state diagram (in this class) is defined by:
 - A set of *states* S (circles)
 - An *initial state* s_0 (only arrow not between states)
 - A *transition function* that maps from the current input and current state to the output and the next state (arrows between states)
 - **Note:** We cover Mealy machines here; Moore machines put outputs on states, not transitions
- ❖ 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

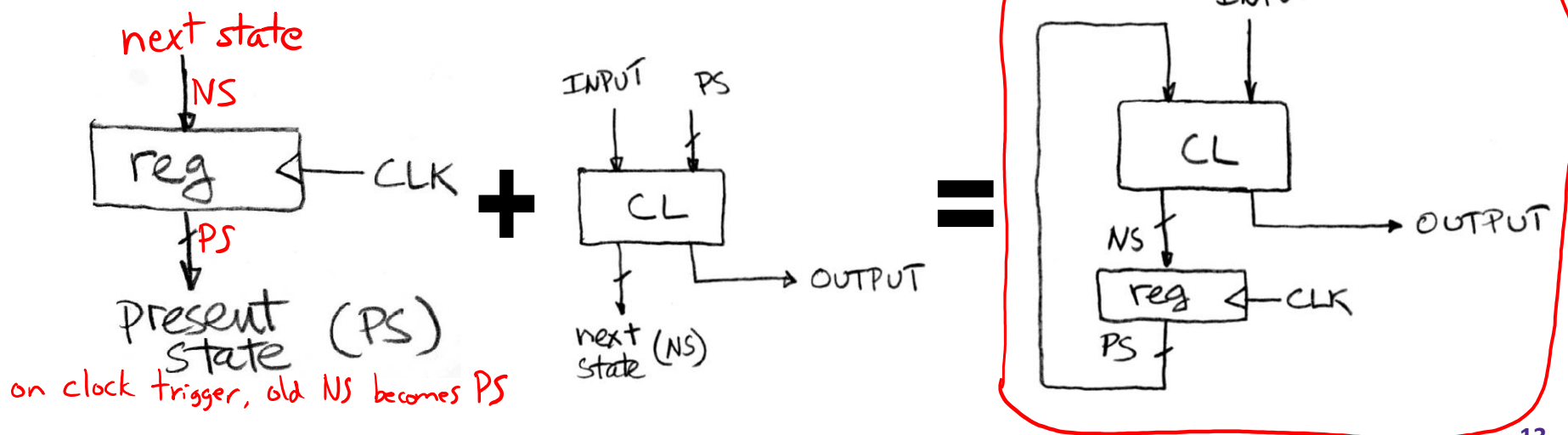
Example: Buggy 3 Ones FSM

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



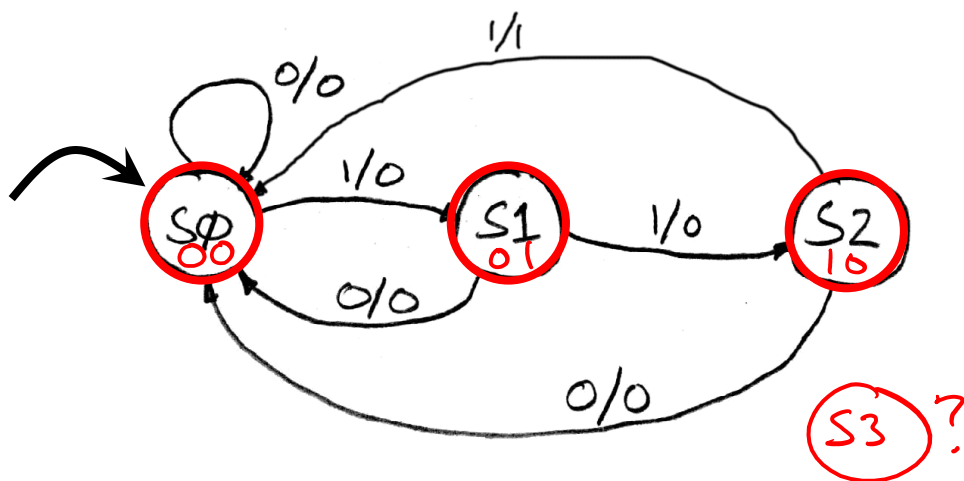
Hardware Implementation of FSM

- ❖ 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

state bits input bits ↑↑ ↑

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

FSM: Logic Simplification

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

NS₁ ↓ *NS₀* ✓

NS₁

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

NS₀

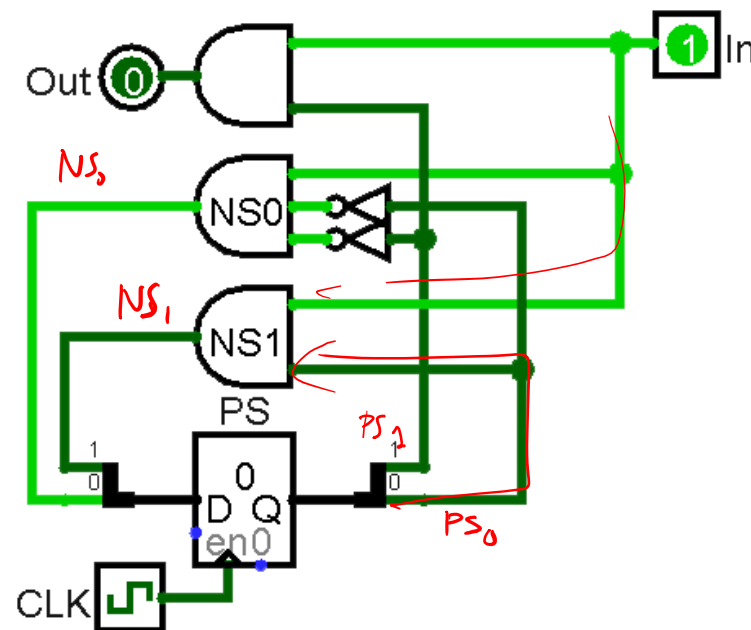
In \ PS	00	01	11	10
0	0	0	X	0
1	1	0	X	0

Out

In \ PS	00	01	11	10
0	0	0	X	0
1	0	0	X	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!

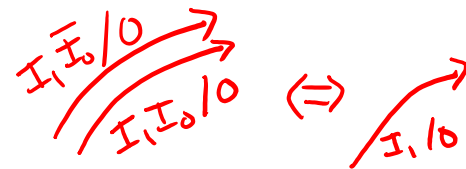
State Diagram Properties

- ❖ For S states, how many state bits^(s) do I use?

$$s = \lceil \log_2 S \rceil$$

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

$$S \times 2^I$$



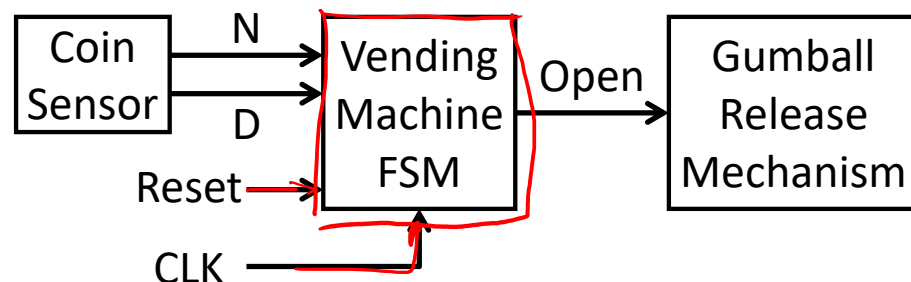
- Can sometimes combine transition arrows:
 - Can sometimes omit transitions (don't cares)
- ❖ For s state bits and I inputs, how big is the truth table?

$$2^{I+s}$$

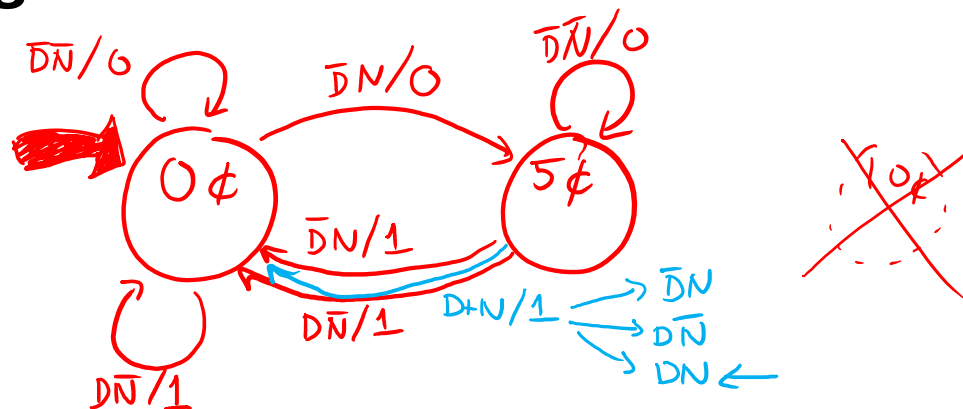
Vending Machine Example

❖ Vending machine description/behavior:

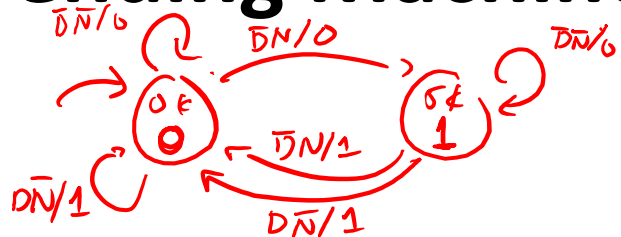
- Single coin slot for dimes and nickels \Rightarrow 2 inputs \Rightarrow 4 transitions
 $\begin{matrix} 00 \\ 01 \\ 10 \\ 11 \end{matrix} \rightarrow X$
- Releases gumball after ≥ 10 cents deposited
- Gives no change



❖ State Diagram:



Vending Machine State Table



PS	N	D	NS	Open
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	X	X
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	X	X

NS

D \ PS,N		PS,N			
		00	01	11	10
D	0	0	1	0	1
	1	0	X	X	0

$$NS = \bar{P}S \cdot N + P\bar{S} \cdot \bar{D}$$

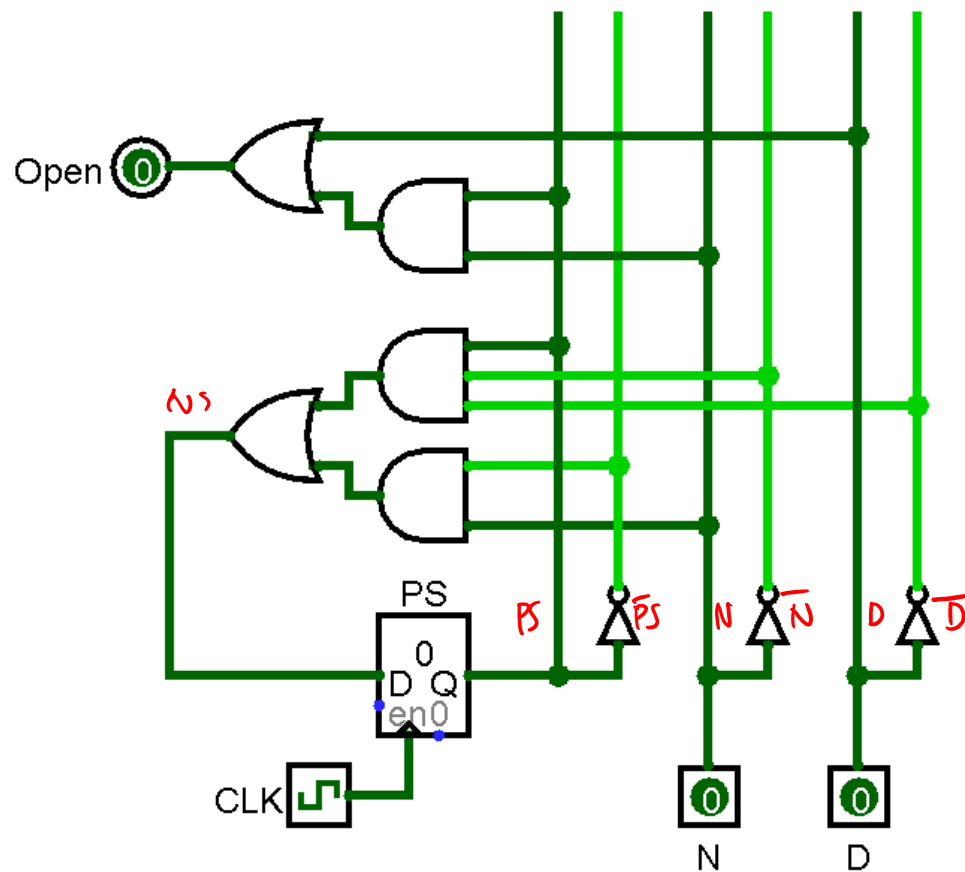
Open

D \ PS,N		PS,N			
		00	01	11	10
D	0	0	0	1	0
	1	1	X	X	1

$$Open = D + P\bar{S} \cdot N$$

Vending Machine Implementation

- ❖ $Open = D + PS \cdot N$
- ❖ $NS = \overline{PS} \cdot N + PS \cdot \overline{N} \cdot \overline{D}$



Outline

- ❖ Flip-Flop Realities
- ❖ Finite State Machines
- ❖ **FSMs in Verilog**

FSMs in Verilog: Declarations

- ❖ Let's examine the components of the Verilog FSM example module on the next few slides

```
module simpleFSM (clk, reset, w, out);  
  input logic clk, reset, w;  
  output logic out;  
  
  // State Encodings and variables  
  enum logic [1:0] {S0 = 2'b00, S1 = 2'b01, S11 = 2'b10}  
  ps, ns; // ps = Present State, ns = Next State  
  ...
```

input (pointing to `w`)
output (pointing to `out`)
important to "initialize" hardware (pointing to `reset`)
enumeration is restriction on existing type (pointing to `enum logic`)
defines new constants for these variables (pointing to `S0 = 2'b00, S1 = 2'b01, S11 = 2'b10`)
explicit definition of binary encoding is optional (pointing to `2'b10`)

FSMs in Verilog: Combinational Logic

...

```
// Next State Logic
```

```
always_comb
```

```
  case (ps)
```

```
    S0: if (w)  ns = S1;
```

```
        else   ns = S0;
```

```
    S1: if (w)  ns = S11;
```

```
        else   ns = S0;
```

```
    S11: if (w) ns = S11;
```

```
         else   ns = S0;
```

```
  endcase
```

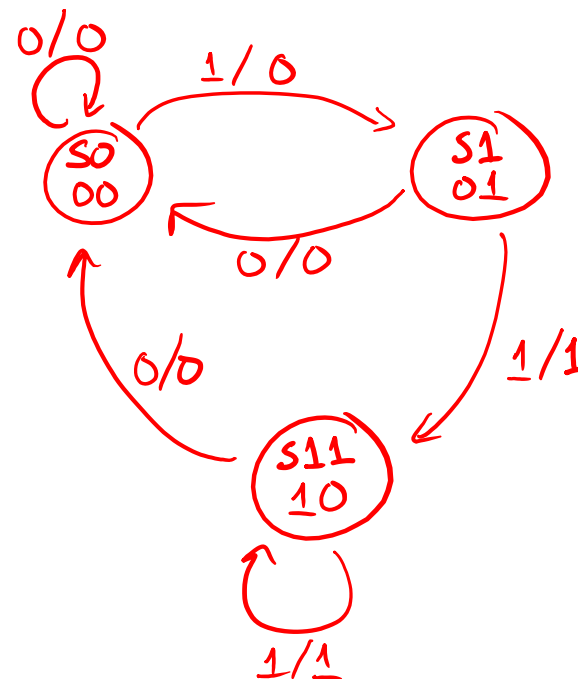
```
// Output Logic - could have been in "always" block
```

```
// or part of Next State Logic.
```

```
assign out = (ns == S11);
```

*output 1 from any transition going into S11
(checking ns, not ps)*

...



FSMs in Verilog: State

...

```
// Sequential Logic (DFFs)
```

```
always_ff @(posedge clk)
```

```
  if (reset)
```

```
    ps <= S0; // "initial" state
```

```
  else
```

```
    ps <= ns;
```

update state
(or reset)
on clock trigger

```
endmodule
```


Reminder: Blocking vs. Non-blocking

- ❖ NEVER mix in one always block!
- ❖ Each variable written in only one always block

- ❖ Blocking (=) in CL:

```
// Output logic
assign out = (ns == S11);

// Next State Logic
always_comb
  case (ps)
    S0: if (w) ns = S1;
        else ns = S0;
    S1: if (w) ns = S11;
        else ns = S0;
    S11: if (w) ns = S11;
        else ns = S0;
  endcase
```

- ❖ Non-blocking (<=) in SL:

```
// Sequential Logic
always_ff @(posedge clk)
  if (reset)
    ps <= S0;
  else
    ps <= ns;
```

One or Two Blocks?

- ❖ We showed the state update in two separate blocks:
 - `always_comb` block that calculates the next state (`ns`)
 - `always_ff` block that defines the register (`ps` updates to last `ns` on clock trigger)
- ❖ Can this be done with a single block?
 - If so, which one: `always_comb` or `always_ff`

↑ means we need to use non-blocking statements

One or Two Blocks? *your choice!*

```
always_comb  
  case (ps)  
    S0: if (w) ns = S1;  
        else ns = S0;  
    S1: if (w) ns = S11;  
        else ns = S0;  
    S11: if (w) ns = S11;  
         else ns = S0;  
  endcase
```

```
always_ff @(posedge clk)  
  if (reset)  
    ps <= S0;  
  else  
    ps <= ns;
```

```
always_ff @(posedge clk)  
  if (reset)  
    ps <= S0;  
  else  
    case (ps)  
      S0: if (w) ps <= S1;  
          else ps <= S0;  
      S1: if (w) ps <= S11;  
          else ps <= S0;  
      S11: if (w) ps <= S11;  
           else ps <= S0;  
    endcase
```

FSM Testbench (1/2)


```
module simpleFSM_tb();
  logic clk, reset, w;
  logic out;

  simpleFSM dut (.clk, .reset, .w, .out);

  // Set up the clock
  parameter CLOCK_PERIOD=100;

  initial begin
    clk <= 0;
    forever #(CLOCK_PERIOD/2) clk <= ~clk;
  end

  ...
endmodule
```



simulated
clock

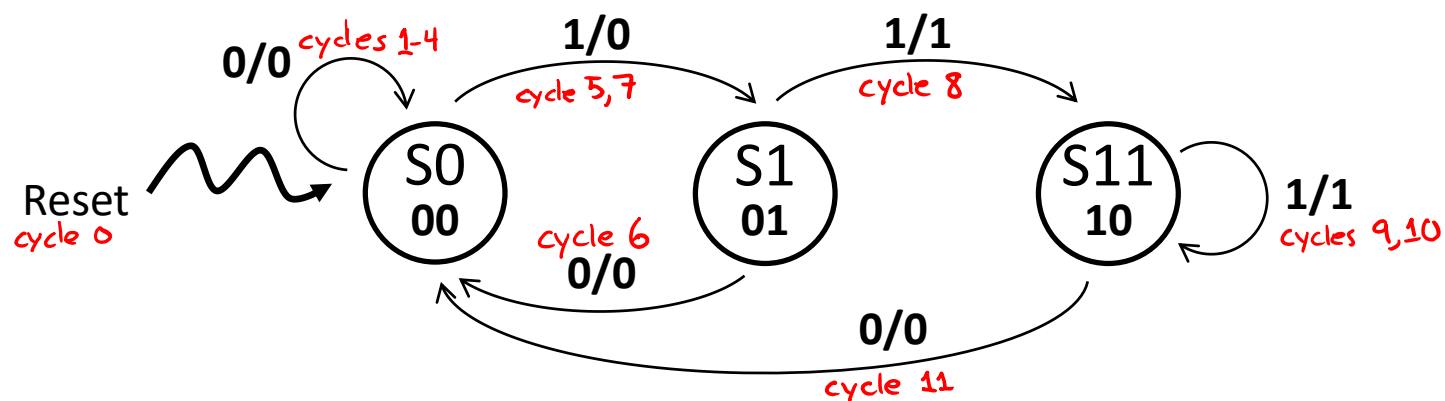
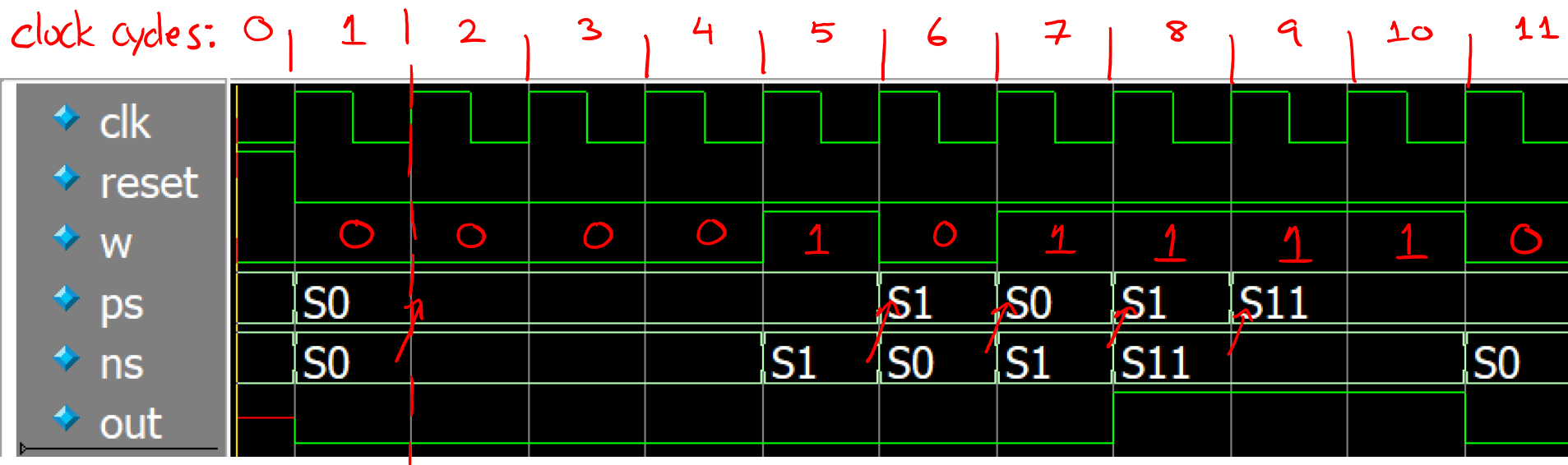
FSM Testbench (2/2)

```
// Set up the inputs to the design (each line is a clock cycle)
initial begin
  reset <= 1; w <= 0; @(posedge clk);
  reset <= 0; w <= 0; @(posedge clk);
  reset <= 0; w <= 0; @(posedge clk);
  reset <= 0; w <= 0; @(posedge clk);
  reset <= 0; w <= 0; @(posedge clk);
  w <= 1; @(posedge clk);
  w <= 0; @(posedge clk);
  w <= 1; @(posedge clk);
  w <= 1; @(posedge clk);
  w <= 1; @(posedge clk);
  w <= 1; @(posedge clk);
  w <= 0; @(posedge clk);
  w <= 0; @(posedge clk);
  $stop; // pause the simulation
end
endmodule
```

reset ↪

*no explicit change of input,
but FSM still transitioning!* ↓

Testbench Waveforms



- ❖ What is the min # of clock cycles to *completely* test this FSM?
8 cycles = 1 reset + 7 transitions (w=1 from state A taken twice)

Summary

- ❖ Gating the clock and external inputs can cause timing issues and metastability
- ❖ FSMs visualize state-based computations
 - Implementations use registers for the state (PS) and combinational logic to compute the next state and output(s)
 - Mealy machines have outputs based on *state transitions*
- ❖ FSMs in Verilog usually have separate blocks for state updates and CL
 - Blocking assignments in CL, non-blocking assignments in SL
 - Testbenches need to be carefully designed to test all state transitions