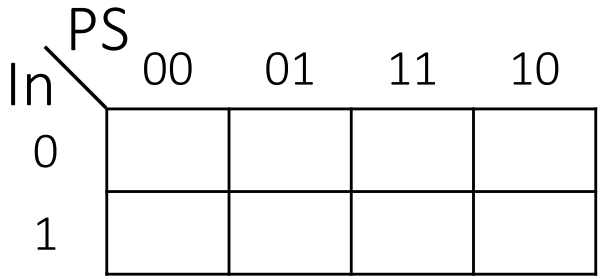
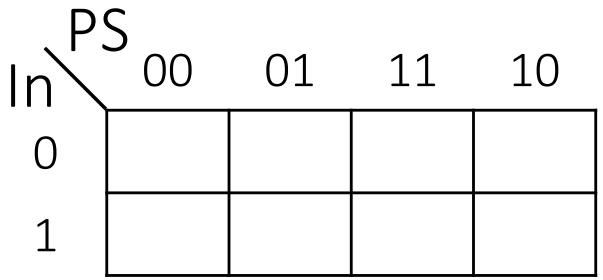
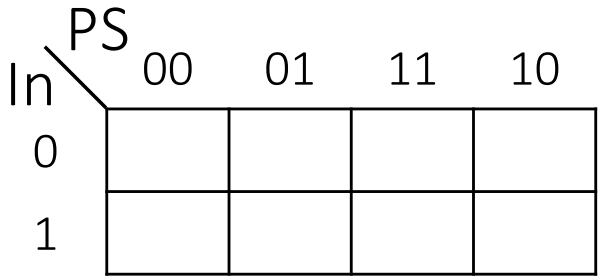
FSM: Logic Simplification

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







FSM: Implementation

- $\mathrm{NS}_1 = \mathrm{PS}_0 \cdot \mathrm{In}$
- $\mathrm{NS}_0 = \overline{\mathrm{PS}_1} \cdot \overline{\mathrm{PS}_0} \cdot \mathrm{In}$
- $\mathrm{Out} = \mathrm{PS}_1 \cdot \mathrm{In}$

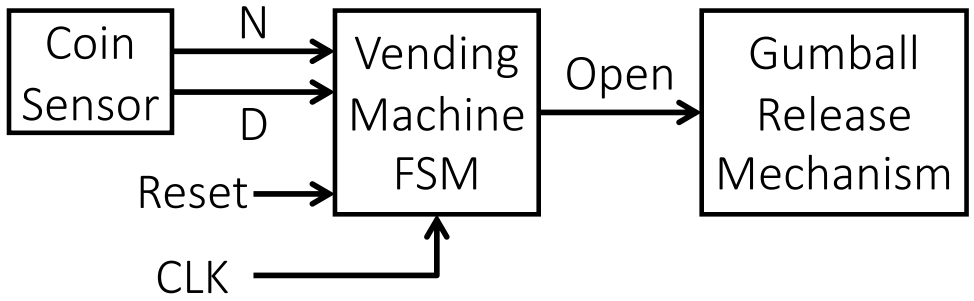
- How do we test the FSM?
  - "Take" every *transition* that we care about!

State Diagram Properties

- For S states, how many state bits do I use?
- For I inputs, what is the *maximum* number of transition arrows on the state diagram?
  - 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?

Vending Machine Example

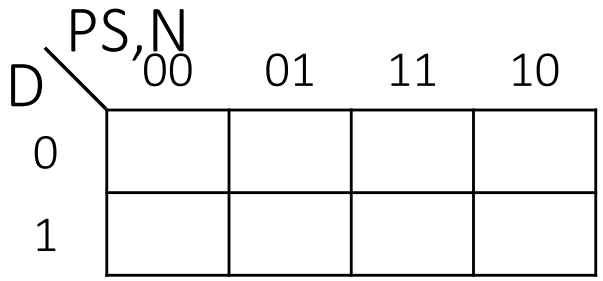
- Vending machine description/behavior:
  - Single coin slot for dimes and nickels
  - Releases gumball after ≥ 10 cents deposited
  - Gives no change

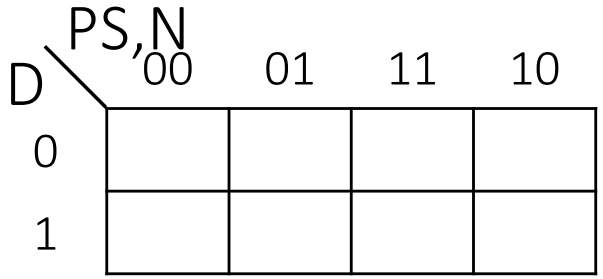


- State Diagram:

Vending Machine State Table

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





Vending Machine Implementation

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;

  logic [1:0] ps;  // Present State
  logic [1:0] ns;  // Next State

  ...
```

FSMs in Verilog: Combinational Logic

```
// State Encodings - so you don't have to remember!
parameter [1:0] A = 2'b00, B = 2'b01, C = 2'b10;

// Next State Logic
always_comb
  case (ps)
    A: if (w)   ns = B;
       else     ns = A;
    B: if (w)   ns = C;
       else     ns = A;
    C: if (w)   ns = C;
       else     ns = A;
    default:    ns = 2'bxx;
  endcase


// Output Logic - could have been in "always" block
// or part of Next State Logic.
assign out = (ps == C);
```

FSMs in Verilog: State

```
  ...

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

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 = (ps == C);

// Next State Logic
always_comb
  case (ps)
    A: if (w)  ns = B;
       else    ns = A;
    B: if (w)  ns = C;
       else    ns = A;
    C: if (w)  ns = C;
       else    ns = A;
    default:   ns = 2'bxx;
  endcase
```

- Non-blocking (<=) in SL:

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

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`

One or Two Blocks?

```
always_comb
  case (ps)
    A: if (w)   ns = B;
       else     ns = A;
    B: if (w)   ns = C;
       else     ns = A;
    C: if (w)   ns = C;
       else     ns = A;
    default:    ns = 2'bxx;
  endcase

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

```
always_ff @(posedge clk)
  if (reset)
    ps <= A;
  else
    case (ps)
      A: if (w)   ps <= B;
         else     ps <= A;
      B: if (w)   ps <= C;
         else     ps <= A;
      C: if (w)   ps <= C;
         else     ps <= A;
      default:    ps <= 2'bxx;
    endcase
```

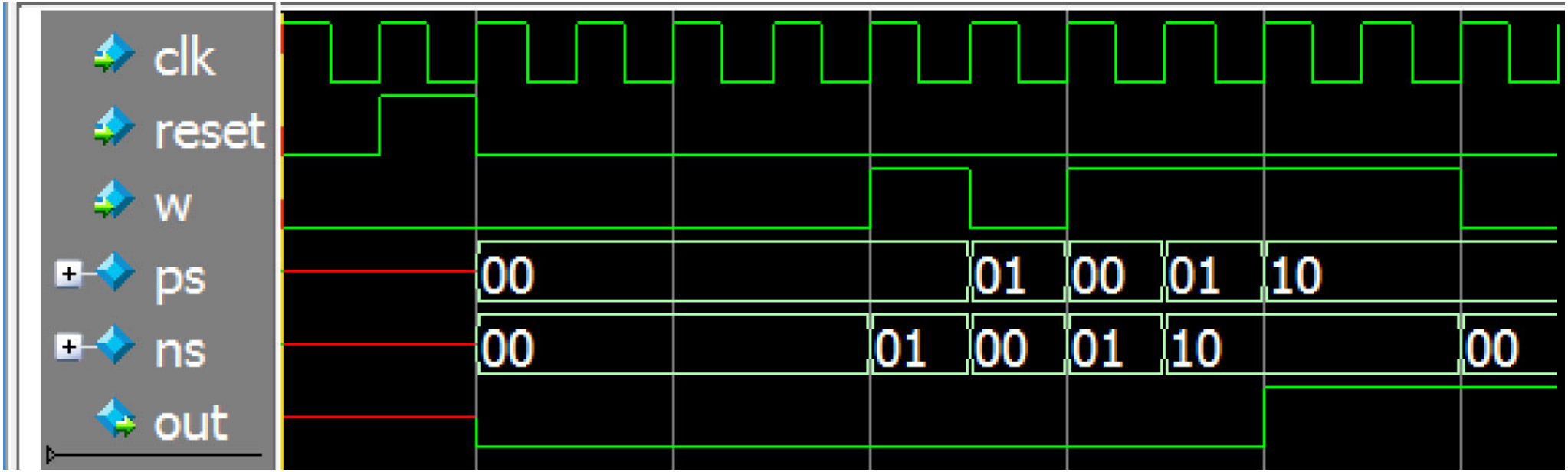
FSM Testbench (1/2)

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

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

  // Set up the clock
  parameter CLOCK_PERIOD=100;

  initial clk<=1;
  always #(CLOCK_PERIOD/2) clk <= ~clk;


  ...
```

FSM Testbench (2/2)

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

Testbench Waveforms



- What is the min # of clock cycles to *completely* test this FSM?

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