Review Problem

If gate delays are 1.0ns each, $T_{\text{setup}}$ is 0.5, and $T_{\text{hold}}$ is 1.5, what is the best clock period for this computation?

- Clock period $\geq 3.5\,\text{ns} = \text{critical path time} + T_{\text{setup}}$
- Shortest path $\geq T_{\text{hold}}$

Better design: $2\,\text{ns} + 0.5\,\text{ns}$
Finite State Machines

- **Readings:** 6-6.4.7
- Need to implement circuits that remember history
  - Traffic Light controller, Sequence Lock, ...
- History will be held in flip flops
- Sequential Logic needs more complex design steps
  - State Diagram to describe behavior
  - State Table to specify functions (like Truth Table)
  - Implementation of combinational logic as controller

![Finite State Machine Diagram]

- **Inputs**
  - Controller (combinational logic)
  - FFs
- **Previous (Current) State**
  - Next State
- **Outputs**
Finite State Machine Example

**Example: Odd Parity Checker**

Assert output whenever have previously seen an odd # of 1's (i.e. how many have you seen NOT INCLUDING the current one)

<table>
<thead>
<tr>
<th>Present State</th>
<th>Input</th>
<th>Output</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Even: State = 0, Odd: State = 1
Finite State Machine Example (cont.)

NS = PS xor Input;  OUT = PS

![Finite State Machine Diagram]

Input

Output

PS

CLK

Reset

D
Q
R

PS/Output

Input

Clk

1 0 0 1 1 0 1 0 1 1 1 1 0
State Diagrams

- Graphical diagram of FSM behavior
- States represented by circles
- Transitions (actions) represented by arrows connecting states
- Lables on Transitions give `<triggering input pattern> / <outputs>`
  - Note: We cover Mealy machines here; Moore machines put outputs on states, not transitions
- Finite State Machine: State Diagram with finite number of states
FSM Design Process

■ 1. Understand the problem

■ 2. Draw the state diagram

■ 3. Use state diagram to produce state table

■ 4. Implement the combinational control logic
Vending Machine Example

- Vending Machine:
  - Deliver package of gum after >= 10 cents deposited
  - Single coin slot for dimes, nickels
  - No change returned

- State Diagram:
Vending Machine Example (cont.)

- State Table:

<table>
<thead>
<tr>
<th></th>
<th>PS</th>
<th>N</th>
<th>D</th>
<th>OPEN</th>
<th>NS</th>
<th>PSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

=> \( \text{OPEN} = D + PS \cdot N \)

\( \text{NS} = \overline{NP}S + PS \overline{N} \overline{D} \)

FYI, NOT COVERED IN THIS CLASS
Vending Machine Example (cont.)

Implementation:

\[ \text{OPEN} = \overline{D} + \overline{PSN} \quad \text{NS} = \overline{N} \overline{PS} + PS \overline{ND} \]
FSMs in Verilog - Declarations

module simple (clk, reset, w, out);
    input        clk, reset, w;
    output       out;

reg    [1:0] ps; // Present State
reg    [1:0] ns; // Next State
FSMs in Verilog – Combinational Logic

```verilog
parameter [1:0] A = 2'b00, B = 2'b01, C = 2'b10;

// Next State Logic
always @(*) begin
  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
end

// Output Logic – could also be “always”,
// or part of next-state logic.
assign out = (ps == C);
```
State Diagram of FSM
FSMs in Verilog – DFFs

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

endmodule
module simple_testbench();
    reg      clk, reset, w;
    wire     out;
    simple dut (.clk, .reset, .w, .out);

    // Set up the clock.
    parameter CLOCK_PERIOD=100;

    initial clk=1;

    always begin
        #(CLOCK_PERIOD/2);
        clk = ~clk;
    end
 FSM Testbench (cont.)

// Design inputs. Each line is a clock cycle.

// ONLY USE THIS FORM for testbenches!!!
initial begin
    @(posedge clk);
    reset <= 1;
    @(posedge clk);
    reset <= 0;
    w <= 0;
    @(posedge clk);
    @(posedge clk);
   @(posedge clk);
    w <= 1;
    @(posedge clk);
    @(posedge clk);
    @(posedge clk);
    @(posedge clk);
    w <= 0;
    @(posedge clk);
    w <= 1;
    @(posedge clk);
    @(posedge clk);
    @(posedge clk);
    w <= 0;
    @(posedge clk);
    @(posedge clk);
    $stop; // End the simulation.
end
endmodule
Testbench Waveforms

clk
reset
w
ps[1:0]
out

0/0 1/0 0/0 1/0 1/1

Reset
String Recognizer Example

- Recognize the string: 101

- Input: 1 0 0 1 0 1 0 1 1 0 0 1 0

- Output:

- State Machine:
String Recognizer Example (cont.)

■ State Table:
= vs. <=

- = (“Blocking”) assign immediately
- <= (“Non-Blocking”) first eval all righthand sides, then do all assignments simultaneously.

module swap1();

... 
reg [3:0] val0, val1;

always @(posedge clk) begin
  if (swap) begin
    val0=val1;
    val1=val0;
  end
  out=val1;
end
endmodule

module swap2();

... 
reg [3:0] val0, val1;

always @(posedge clk) begin
  if (swap) begin
    val0<=val1;
    val1<=val0;
  end
  out<=val1;
end
endmodule
= vs. <= in practice

- = in combinational logic: always @*
- <= in sequential, ps<=ns: always @(posedge clk)
- NEVER mix in one always block!
- Each variable written in only one always block

```verbatim
// Output logic
always @(*) begin
  out = (ps == A);
end

// Next State Logic
always @(*) begin
  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
end

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

■ Some problems best solved with multiple pieces

■ Psychic Tester:
  ■ Machine generates pattern of 4 values (on or off)
  ■ If user guesses 8 patterns in a row, they’re psychic

■ States?

\[ 2^4 = 16 \text{ patterns} \]

\[ 0..7 = 8 \text{ guesses} \]

\[ 16 \times 8 = 128 \text{ states} \]
Subdividing FSMs (cont.)

■ Pieces?

- Pick a pattern (16 state FSM)
  - Pattern
  - Count correct guesses (8 state FSM)
    - Right?
  - Check Guess (combinational)
    - Guess
  - User Input (4 switches, 1 “go” button)

- "go"