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

![Diagram of Finite State Machine]

- **Inputs**
- **Previous (Current) State**
- **Next State**
- **Controller (combinational logic)**
- **Outputs**
- **FFs**
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></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

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

\[ NS = PS \oplus \text{Input}; \quad \text{OUT} = PS \]
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
State Diagram Example

- Circuit that is true every 4\textsuperscript{th} cycle.
State Table

“Truth table” for sequential circuits

<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>1</td>
<td>0</td>
</tr>
</tbody>
</table>
State Table Example

- State Table for 4\textsuperscript{th} cycle circuit
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:
Vending Machine Example (cont.)

- Implementation:
FSMs in Verilog - Declarations

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

    enum { A, B, C} ps, ns; // Present state, next state
FSMs in Verilog – Combinational Logic

// Next State Logic
always_comb 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;
    endcase
end

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

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

endmodule
Circuit Diagram of FSM
module simple_testbench();
  logic  clk, reset, w, out;

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

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

  initial begin
    clk <= 0;
    forever #(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);
  @(posedge clk);
  @(posedge clk);
  w <= 1; @(posedge clk);
  w <= 0; @(posedge clk);
  w <= 1; @(posedge clk);
  $stop; // End the simulation.
end
endmodule
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:
String Recognizer Example (cont.)

- Implementation:
A busy highway is intersected by a little used farmroad. Detectors sense the presence of cars waiting on the farmroad. With no car on farmroad, lights remain green in highway direction. If vehicle on farmroad, highway lights go from Green to Yellow to Red, allowing the farmroad lights to become green. These stay green only as long as a farmroad car is detected but never longer than a set interval. When these are met, farm lights transition from Green to Yellow to Red, allowing highway to return to green. Even if farmroad vehicles are waiting, highway gets at least a set interval as green.

Assume you have an interval timer that generates a short time signal (TS) and a long time signal (TL) in response to a START signal. TS is to be used for timing yellow lights and TL for green lights. Once these signals go true, they stay true until the next START or RESET
Traffic Light Controller (cont.)

Picture of Highway/Farmroad Intersection:
Traffic Light Controller (cont.)

- Tabulation of Inputs and Outputs:

<table>
<thead>
<tr>
<th>Input Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>detect vehicle on farmroad</td>
</tr>
<tr>
<td>TS</td>
<td>short time interval expired</td>
</tr>
<tr>
<td>TL</td>
<td>long time interval expired</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HG, HY, HR</td>
<td>assert green/yellow/red highway lights</td>
</tr>
<tr>
<td>FG, FY, FR</td>
<td>assert green/yellow/red farmroad lights</td>
</tr>
<tr>
<td>START</td>
<td>start timing a short or long interval</td>
</tr>
</tbody>
</table>
Traffic Light Controller (cont.)

- State Diagram
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?
Subdividing FSMs (cont.)

- Pieces?
\[= \text{ vs. } \leq\]

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

```verilog
module swap1();
    ...
    logic [3:0] val0, val1;
    always_ff @(posedge clk) begin
        if (swap) begin
            val0=val1;
            val1=val0;
        end
        out=val1;
    end
endmodule
```

```verilog
module swap2();
    ...
    logic [3:0] val0, val1;
    always_ff @(posedge clk) begin
        if (swap) begin
            val0<=val1;
            val1<=val0;
        end
        out<=val1;
    end
endmodule
```
= vs. <= in practice

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

```vhdl
// Output logic
always_comb begin
    out = (ps == A);
end

// Next State Logic
always_comb 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;
    endcase
end
```

```vhdl
// Sequential Logic
always_ff @(posedge clk) begin
    if (reset)
        ps <= A;
    else
        ps <= ns;
end
```
Flipflop Realities 1: Gating the Clock

NEVER put a logic gate between the clock and DFF’s CLK input.
Flipflop Realities 2: Clock Period, Applying Stimulus

- Clock Period?
- Apply Inputs when?

Diagram of D flipflops with clock inputs and outputs.
$T_{\text{setup}}, T_{\text{hold}}, \text{Clk} \rightarrow Q$

- Flipflops require their inputs be stable for time period around clock edge

D (Can Change) \[\overset{T_{\text{setup}}}{\longrightarrow}\] Stable \[\overset{T_{\text{hold}}}{\longrightarrow}\] Can Change

Clock

Q (Old Value) \[\overset{\text{Clk} \rightarrow Q}{\longrightarrow}\] New Value

Diagram of D flipflops and clock connections.
Timing Definitions

- **T\textsubscript{setup}:** Time D must be stable BEFORE clock edge
  - Often adds to critical path delay

- **Clk->Q:** Time from clock edge to Q changing
  - Often adds to critical path delay

- **T\textsubscript{hold}:** Time D must be stable AFTER clock edge
  - Can get violated by paths that are too short
Flipflop Realities 3: External Inputs

- External inputs aren’t synchronized to the clock

\[ D \quad Clk \quad Q \]

Metastability: input transition within \( T_{\text{setup}}..T_{\text{hold}} \) period causes DFF to strange middle value.

Behavior sketch:
- after \( Clk \rightarrow Q \) the \( Q \) output goes to \( \frac{1}{2} \)
- stays there for \( \sim 1-2 \) ns
- then randomly goes to 0 or 1
Dealing with Metastability

- Single DFF

- 2 DFFs in series

- 2 DFFs in parallel