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:

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

$$NS = PS \oplus \text{Input}; \quad OUT = PS$$

![Diagram of a finite state machine with inputs, outputs, and logic gates.](image)

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

PS/Output: 1 0 0 1 1 0 1 0 1 1 1 0
State Diagrams

- Graphical diagram of FSM behavior
- States represented by circles
- Transitions (actions) represented by arrows connecting states
- Labels 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 4th 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 $\geq 10$ cents deposited
  - Single coin slot for dimes, nickels
  - No change returned

- **State Diagram:**
Vending Machine Example (cont.)

- State Table:
Vending Machine Example (cont.)

- Implementation:
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
FSM Testbench

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);
  w <= 1; @(posedge clk);
  w <= 0; @(posedge clk);
  w <= 1; @(posedge clk);
  w <= 1; @(posedge clk);
  w <= 1; @(posedge clk);
  w <= 0; @(posedge clk);
  $stop; // End the simulation.
end
endmodule
Testbench Waveforms

clk
reset
w
ps[1:0]
out

0/0 1/0 0/0 0/1 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:**
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 pulse (TS) and a long time pulse (TL) in response to a set (ST) signal. TS is to be used for timing yellow lights and TL for green lights.
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>reset</td>
<td>place FSM in initial state</td>
</tr>
<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>ST</td>
<td>start timing a short or long interval</td>
</tr>
</tbody>
</table>
Traffic Light Controller (cont.)

- State Diagram
= vs. <=

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

```vhdl
module swap1();

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

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

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

```
// 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

// Sequential Logic
always_ff @(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?
Subdividing FSMs (cont.)

❖ Pieces?
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 and clock inputs](image-url)
$T_{\text{setup}}, T_{\text{hold}}, \text{Clk} \rightarrow \text{Q}$

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

![Flipflop Diagram with $T_{\text{setup}}$ and $T_{\text{hold}}$](image)

Clock

Q: Old Value

New Value

Clk $\rightarrow$ Q

$D$ flipflop

$D$ $\rightarrow$ $Q$

$Clk$ $\rightarrow$ $Q$
Timing Definitions

- $T_{\text{setup}}$: Time D must be stable BEFORE clock edge
  - Often adds to critical path delay

- $\text{Clk-} \rightarrow \text{Q}$: Time from clock edge to Q changing
  - Often adds to critical path delay

- $T_{\text{hold}}$: 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 ___
Clk ___
Q ___
```

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

Behavior sketch:
- after Clk->Q the Q output goes to ½
- stays there for ~1-2ns
- then randomly goes to 0 or 1
Dealing with Metastability

- Single DFF

  ![Single DFF Diagram]

- 2 DFFs in series

  ![2 DFFs in series Diagram]

- 2 DFFs in parallel

  ![2 DFFs in parallel Diagram]