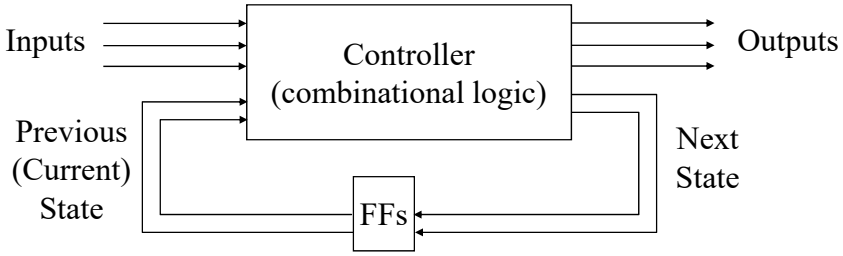
Finite State Machines

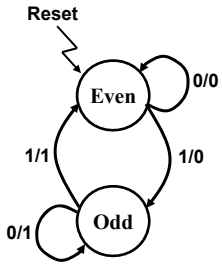
- Readings: 6-6.4.7
- Need to implement circuits that remember history
  - Traffic Light controller, Sequence Lock, ...
- History will be held in flip flops
- Sequential Logic needs more complex design steps
  - State Diagram to describe behavior
  - State Table to specify functions (like Truth Table)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



Finite State Machine 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)



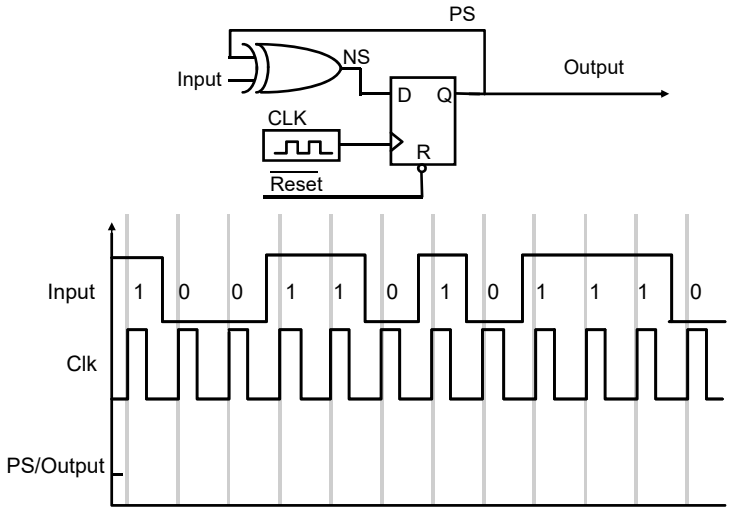
State Diagram

Present State	Input	Output	Next State
0	0		
0	1		
1	0		
1	1		

Even: State = 0, Odd: State = 1

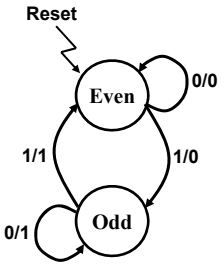
Finite State Machine Example (cont.)

NS = PS xor Input; 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 4th cycle.

State Table

- "Truth table" for sequential circuits

Present State	Input	Output	Next State
0	0	0	0
0	1	0	1
1	0	1	1
1	1	1	0

State Table Example

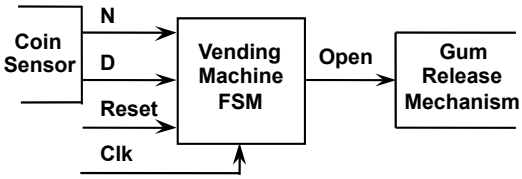
- State Table for 4^{th} cycle circuit

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

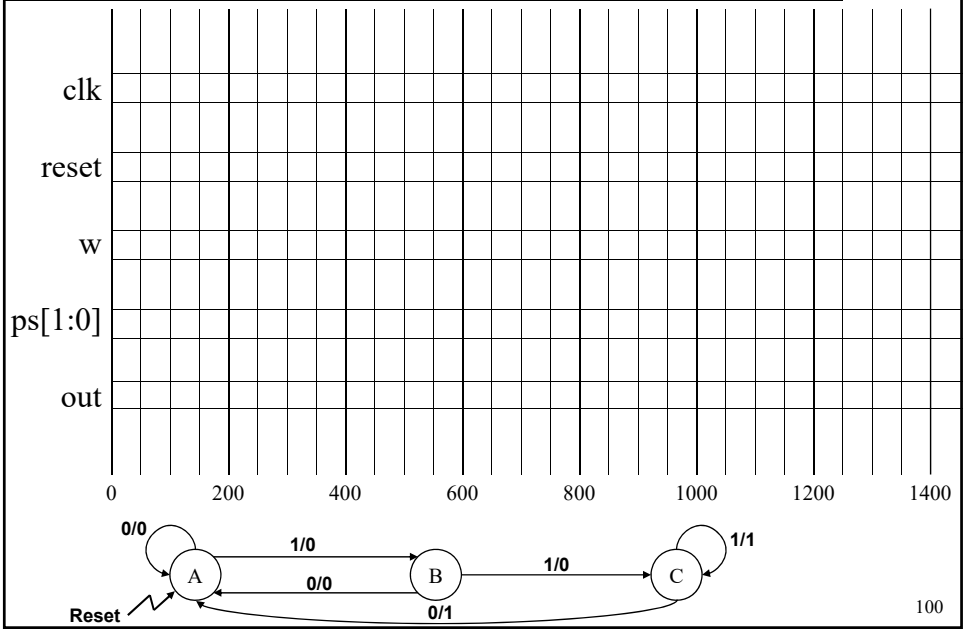
FSM Testbench

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


Testbench Waveforms



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:

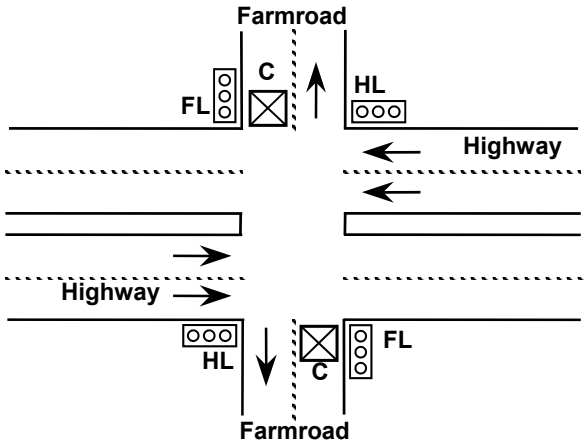
FSM Word Problem: Traffic Light Controller

A busy highway is intersected by a little used farmroad. Detectors C 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:**

Input Signal	*Description*
reset	place FSM in initial state
C	detect vehicle on farmroad
TS	short time interval expired
TL	long time interval expired

Output Signal	*Description*
HG, HY, HR	assert green/yellow/red highway lights
FG, FY, FR	assert green/yellow/red farmroad lights
ST	start timing a short or long interval

Traffic Light Controller (cont.)

- State Diagram

= vs. <=

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

```
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
```

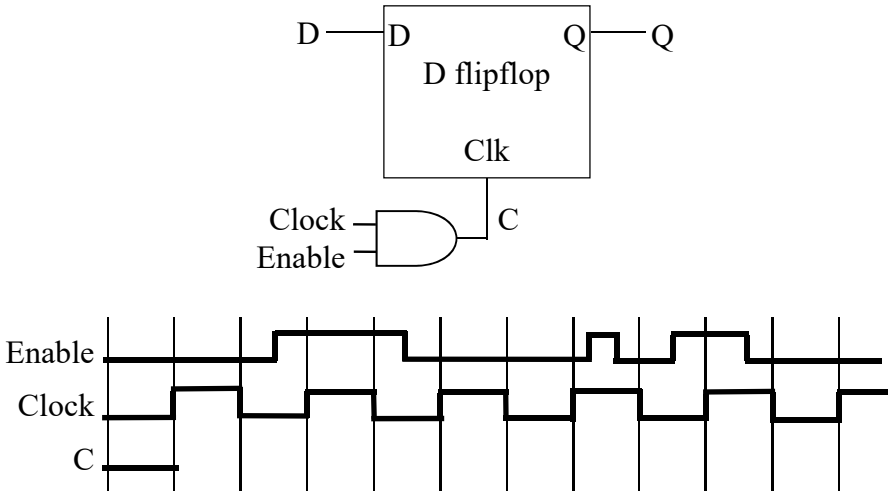

Subdividing FSMs

- 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.)

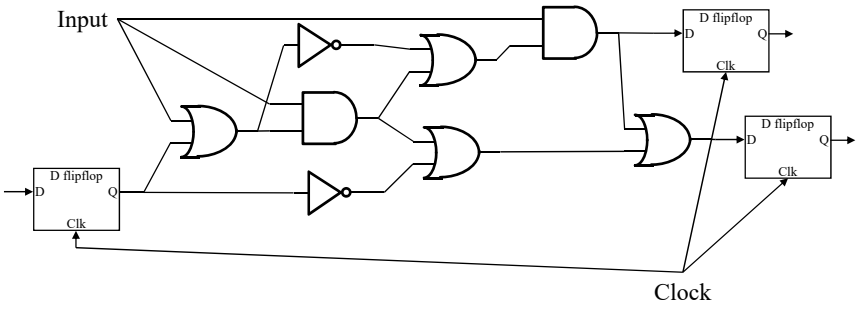
- Pieces?

Flipflop Realities 1: Gating the Clock

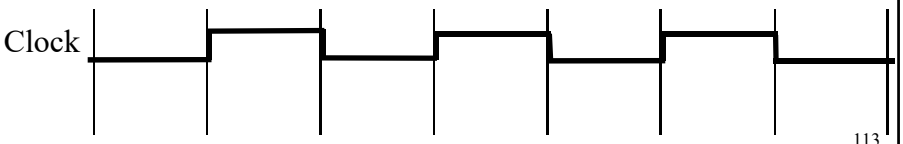


- **NEVER put a logic gate between the clock and DFF's CLK input.**

Flipflop Realities 2: Clock Period, Applying Stimulus

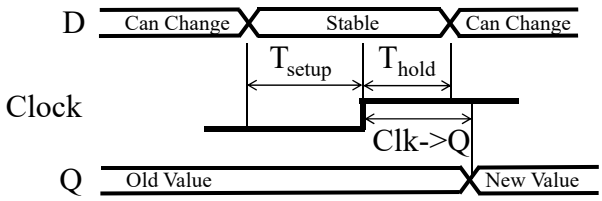


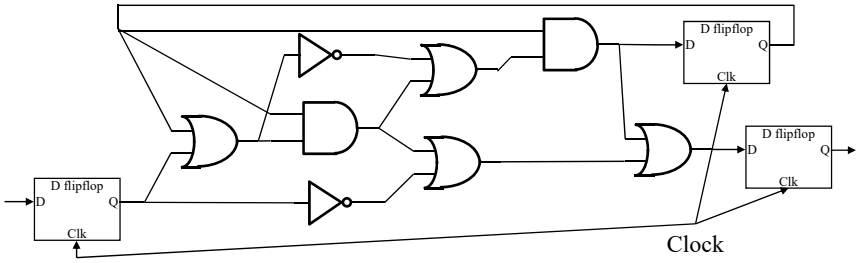
- Clock Period?
- Apply Inputs when?



T_{setup}, T_{hold}, Clk->Q

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



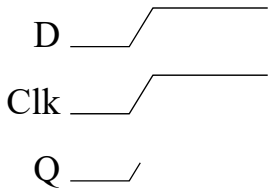


Timing Definitions

- T_{setup}: Time D must be stable BEFORE clock edge
  - Adds to critical path delay
- Clk->Q: Time from clock edge to Q changing
  - Adds to critical path delay
- T_{hold}: Time D must be stable AFTER clock edge
  - Sets minimum path from Q of one DFF to D of another

Flipflop Realities 3: External Inputs

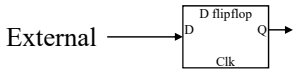
- External inputs aren't synchronized to the clock



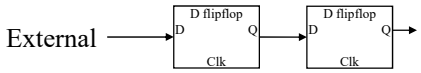
Metastability: input transition within T_{setup}..T_{hold} period causes DFF to strange middle value.
Behavior sketch:
after Clk->Q the Q output goes to ½
stays there for ~1-2ns
then randomly goes to 0 or 1

Dealing with Metastability

- Single DFF



- 2 DFFs in series



- 2 DFFs in parallel