Lecture 8: Finite State Machines
Finite State Machines
Finite State Machines (FSMs)

- FSMs:
  - Can model behavior of *any* sequential circuit
  - Useful representation for designing sequential circuits
  - As with all sequential circuits: output depends on present *and* past inputs
    - effect of past inputs represented by the current *state*
  - Behavior is represented by *State Transition Diagram:*
    - traverse one edge per clock cycle.
FSM Implementation

- Flip-flops form *state register*
- Number of states $\leq 2^{\text{number of flip-flops}}$
- CL (combinational logic) calculates next state and output
- Remember: The FSM follows exactly one edge per cycle.

Later we will learn how to implement in Verilog. Now we learn how to design “by hand” to the gate level.
FSM Example: Parity Checker

A string of bits has “even parity” if the number of 1’s in the string is even.

- Design a circuit that accepts an infinite serial stream of bits, and outputs a 0 if the parity thus far is even and outputs a 1 if odd:

Next we take this example through the “formal design process”. But first, can you guess a circuit that performs this function?
“State Transition Diagram”

- circuit is in one of two “states”.
- transition on each cycle with each new input, over exactly one arc (edge).
- Output depends on which state the circuit is in.
By-hand Design Process (b)

State Transition Table:

<table>
<thead>
<tr>
<th>present state</th>
<th>OUT</th>
<th>IN</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEN</td>
<td>0</td>
<td>0</td>
<td>EVEN</td>
</tr>
<tr>
<td>EVEN</td>
<td>0</td>
<td>1</td>
<td>ODD</td>
</tr>
<tr>
<td>ODD</td>
<td>1</td>
<td>0</td>
<td>ODD</td>
</tr>
<tr>
<td>ODD</td>
<td>1</td>
<td>1</td>
<td>EVEN</td>
</tr>
</tbody>
</table>

Invent a code to represent states:
Let 0 = EVEN state, 1 = ODD state

Derive logic equations from table (how?):

- OUT = PS
- NS = PS xor IN
Logic equations from table:
\[ \text{OUT} = PS \]
\[ \text{NS} = PS \text{ xor } IN \]

- XOR gate for NS calculation
- Flip-Flop to hold present state
- no logic needed for output in this example.
“Formal” By-hand Design Process

Review of Design Steps:

1. Specify **circuit function** *(English)*
2. Draw **state transition diagram**
3. Write down **symbolic state transition table**
4. Write down **encoded state transition table**
5. Derive **logic equations**
6. Derive **circuit diagram**
   - Register to hold state
   - Combinational Logic for Next State and Outputs
Another FSM Design Example
Combination Lock Example

- Used to allow entry to a locked room:
  - 2-bit serial combination. Example 01,11:
    1. Set switches to 01, press ENTER
    2. Set switches to 11, press ENTER
    3. OPEN is asserted (OPEN=1).
      
      If wrong code, ERROR is asserted (after second combo word entry).
      
      Press Reset at anytime to try again.
Assume the ENTER button when pressed generates a pulse for only one clock cycle.
### Symbolic State Transition Table

<table>
<thead>
<tr>
<th>RESET</th>
<th>ENTER</th>
<th>COM1</th>
<th>COM2</th>
<th>Preset State</th>
<th>Next State</th>
<th>OPEN</th>
<th>ERROR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>*</td>
<td>*</td>
<td>START</td>
<td>START</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>*</td>
<td>START</td>
<td>BAD1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>*</td>
<td>START</td>
<td>OK1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>*</td>
<td>*</td>
<td>OK1</td>
<td>OK1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>*</td>
<td>0</td>
<td>OK1</td>
<td>BAD2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>*</td>
<td>1</td>
<td>OK1</td>
<td>OK2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>OK2</td>
<td>OK2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>*</td>
<td>*</td>
<td>BAD1</td>
<td>BAD1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>*</td>
<td>*</td>
<td>BAD1</td>
<td>BAD2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>BAD2</td>
<td>BAD2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>START</td>
<td>START</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

* represents “wild card” - expands to all combinations

### Decoder logic for checking combination (01,11):

```
left switch
```

```
right switch
```

```
COM1
```

```
COM2
```
**Encoded ST Table**

- **Assign states:**
  \[
  \text{START}=000, \text{OK1}=001, \text{OK2}=011 \\
  \text{BAD1}=100, \text{BAD2}=101
  \]

- **Omit reset. Assume that primitive flip-flops has reset input.**

- **Rows not shown have don't cares in output. Correspond to invalid PS values.**

- **What are the output functions for OPEN and ERROR?**

---

<table>
<thead>
<tr>
<th>ENTER</th>
<th>COM1</th>
<th>COM0</th>
<th>PS1</th>
<th>PS0</th>
<th>NS2</th>
<th>NS1</th>
<th>NS0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>0 1 0</td>
<td>0 1 0</td>
<td>0 1 0</td>
<td>0 1 0</td>
<td>0 1 0</td>
<td>0 1 0</td>
<td>0 1 0</td>
<td>0 1 0</td>
</tr>
<tr>
<td>1 0 0</td>
<td>1 0 0</td>
<td>1 0 0</td>
<td>1 0 0</td>
<td>1 0 0</td>
<td>1 0 0</td>
<td>1 0 0</td>
<td>1 0 0</td>
</tr>
<tr>
<td>1 1 0</td>
<td>1 1 0</td>
<td>1 1 0</td>
<td>1 1 0</td>
<td>1 1 0</td>
<td>1 1 0</td>
<td>1 1 0</td>
<td>1 1 0</td>
</tr>
<tr>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>1 1 1</td>
<td>1 1 1</td>
<td>1 1 1</td>
<td>1 1 1</td>
<td>1 1 1</td>
<td>1 1 1</td>
<td>1 1 1</td>
<td>1 1 1</td>
</tr>
</tbody>
</table>

**diagram**

- **NS2**
- **NS1**
- **NS0**

14
Moore Versus Mealy Machines
All examples so far generate output based only on the present state, commonly called a “Moore Machine”:

If output functions include both present state and input then called a “Mealy Machine”:
Example: Edge Detector

Bit are received one at a time (one per cycle), such as: 000111010

Design a circuit that asserts its output for one cycle when the input bit stream changes from 0 to 1.

We'll try two different solutions: Moore then Mealy.
<table>
<thead>
<tr>
<th>IN</th>
<th>PS</th>
<th>NS</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>11</td>
<td>0</td>
</tr>
</tbody>
</table>

State Transition Diagram Solution A - Moore
Solution A, circuit derivation

<table>
<thead>
<tr>
<th>IN</th>
<th>PS</th>
<th>NS</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>11</td>
<td>0</td>
</tr>
</tbody>
</table>

- **ZERO**
- **CHANGE**
- **ONE**

**Figure:**

- **PS**
  - $00, 01, 11, 10$
  - $IN = 0$
  - $NS_1 = IN \cdot PS_0$
  - $IN = 1$
  - $PS_0 = IN$
  - $OUT = PS_1 \cdot PS_0$

**Diagram:**

- FF
- $NS_0, NS_1$
- $PS_0, PS_1$
- $OUT$
Solution B - Mealy

Output depends not only on PS but also on input, IN

Let ZERO=0, ONE=1

<table>
<thead>
<tr>
<th>IN</th>
<th>PS</th>
<th>NS</th>
<th>OUT</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>0</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>

NS = IN, OUT = IN PS'

What's the intuition about this solution?
Edge detector timing diagrams

- **Solution A**: both edges of output follow the clock
- **Solution B**: output rises with input rising edge and is asynchronous wrt the clock, output fails synchronous with next clock edge
**FSM Comparison**

**Moore Machine**
- output function only of PS
- maybe more states
- synchronous outputs
  - Input glitches not send at output
  - one cycle “delay”
  - full cycle of stable output

**Mealy Machine**
- output function of both PS & input
- maybe fewer states
- asynchronous outputs
  - if input glitches, so does output
  - output immediately available
  - output may not be stable long enough to be useful (below):

If output of Mealy FSM goes through combinational logic before being registered, the CL might delay the signal and it could be missed by the clock edge (or violate set-up time requirement)
FSM Moore and Mealy Implementation Review

Moore Machine

- Input value
- State (output values)
- Inputs
- Present state
- Next state
- Outputs

Mealy Machine

- Input value/output values
- State
- Inputs
- Present state
- Next state
- Outputs
1. A given state machine could have both Moore and Mealy style outputs. Nothing wrong with this, but you need to be aware of the timing differences between the two types.

2. The output timing behavior of the Moore machine can be achieved in a Mealy machine by “registering” the Mealy output values:
State Assignment

- When FSM implemented with gate logic, number of gates will depend on mapping between symbolic state names and binary encodings.

- Ex: combination lock FSM
  - 5 states, 3 bits
  - my assignment \( \text{START}=000, \text{OK1}=001, \text{OK2}=011, \text{BAD1}=100, \text{BAD2}=101 \)
  - only one of 6720 3-bit assignments
    - (5 states = 8 choices for first state, 7 for second, 6 for third, 5 for forth, 4 for last = 6720 different encodings)
State Assignment

Pencil & Paper Heuristic Methods

State Maps: similar in concept to K-maps
If state X transitions to state Y, then assign "close" assignments to X and Y

<table>
<thead>
<tr>
<th>State Name</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>S₀</td>
<td>0 0 0</td>
</tr>
<tr>
<td>S₁</td>
<td>1 0 1</td>
</tr>
<tr>
<td>S₂</td>
<td>1 1 1</td>
</tr>
<tr>
<td>S₃</td>
<td>0 1 0</td>
</tr>
<tr>
<td>S₄</td>
<td>0 1 1</td>
</tr>
</tbody>
</table>

Assignment

<table>
<thead>
<tr>
<th>State Name</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>S₀</td>
<td>0 0 0</td>
</tr>
<tr>
<td>S₁</td>
<td>0 0 1</td>
</tr>
<tr>
<td>S₂</td>
<td>0 1 0</td>
</tr>
<tr>
<td>S₃</td>
<td>0 1 1</td>
</tr>
<tr>
<td>S₄</td>
<td>1 1 1</td>
</tr>
</tbody>
</table>

State Map

<table>
<thead>
<tr>
<th>Q₂</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S₀</td>
<td>S₄</td>
<td>S₃</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>S₁</td>
<td>S₂</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
State Encoding

- In general:
  
  \[ \text{# of possible FSM states} = 2^{\text{# of Flip-flops}} \]

  Example:

  \[
  \text{state1 = 01, state2 = 11, state3 = 10, state4 = 00}
  \]

- However, sometimes more than \( \log_2(\text{# of states}) \) FFs are used, to simplify logic at the cost of more FFs.

- Extreme example is **one-hot state encoding**.
State Encoding

- One-hot encoding of states.
- One FF per state.
- On a transition from state $i$ to $j$, a 1 moves from FF$_i$ to FF$_j$.

**Why one-hot encoding?**
- Simple design procedure.
  - Circuit matches state transition diagram (example next page).
  - Often can lead to simpler and faster “next state” and output logic.

**Why not do this?**
- Can be costly in terms of Flip-flops for FSMs with large number of states.
- FPGAs are “Flip-flop rich”, therefore one-hot state machine encoding is often a good approach.
One-hot encoded FSM

- Even Parity Checker Circuit:
  - Think about moving a single token from state to state.
  - Circuit generated through direct inspection of the STD.
  - FFs must be initialized for correct operation (only one 1)

- In General:
  - state FF
  - Input
  - to other state FF logic and/or output
One-hot encoded combination lock
FSMs in Verilog
General FSM Design Process with Verilog Implementation

Design Steps:

1. Specify **circuit function** (English)
2. Draw **state transition diagram**
3. Write down **symbolic state transition table**
4. Assign encodings (bit patterns) to symbolic states
5. Code as Verilog behavioral description

✓ Use parameters to represent encoded states.
✓ Use register instances for present-state plus CL logic for next-state and outputs.
✓ Use case for CL block. Within each case section (state) assign all outputs and next state value based on inputs. Note: For Moore style machine make outputs dependent only on state not dependent on inputs.
module FSM1(clk, rst, in, out);
input clk, rst;
input in;
output out;

// Defined state encoding:
localparam IDLE = 2'b00;
localparam S0 = 2'b01;
localparam S1 = 2'b10;

reg out;
reg [1:0] next_state;
wire [1:0] present_state;

// state register
REGISTER_R #( .N(2), .INIT(IDLE)) state
(.q(present_state), .d(next_state), .rst(rst));

An always block should be used for combination logic part of FSM. Next state and output generation.

Must use reset to force to initial state.
reset not always shown in STD

Constants local to this module.

out not a register, but assigned in always block

Combinational logic signals for transition.
// Always block for combination logic portion
always @(present_state or in)
case (present_state)
  IDLE   : begin
    out = 1'b0;
    if (in == 1'b1) next_state = S0;
    else next_state = IDLE;
  end
  S0     : begin
    out = 1'b0;
    if (in == 1'b1) next_state = S1;
    else next_state = IDLE;
  end
  S1     : begin
    out = 1'b1;
    if (in == 1'b1) next_state = S1;
    else next_state = IDLE;
  end
  default: begin
    next_state = IDLE;
    out = 1'b0;
  end
endcase
endmodule

For each state define:
Output value(s)
State transition

Each state becomes a case clause.

Use “default” to cover unassigned state. Usually unconditionally transition to reset state.

Mealy or Moore?
**Edge Detector Example**

### Mealy Machine

```
REGISTER_R #(.INIT(ZERO)) state (.q(ps), .d(ns), .rst(rst));
always @(ps in)
  case (ps)
    ZERO: if (in) begin
      out = 1'b1;
      ns = ONE;
    end
    else begin
      out = 1'b0;
      ns = ZERO;
    end
    ONE: if (in) begin
      out = 1'b0;
      ns = ONE;
    end
    else begin
      out = 1'b0;
      ns = ZERO;
    end
    default: begin
      out = 1'bx;
      ns = default;
    end
  end
```

### Moore Machine

```
REGISTER_R #(N(2), .INIT(ZERO)) state (.q(ps), .d(ns), .rst(rst));
always @(ps in)
  case (ps)
    ZERO: begin
      out = 1'b0;
      if (in) ns = CHANGE;
      else ns = ZERO;
    end
    CHANGE: begin
      out = 1'b1;
      if (in) ns = ONE;
      else ns = ZERO;
    end
    ONE: begin
      out = 1'b0;
      if (in) ns = ONE;
      else ns = ZERO;
    end
    default: begin
      out = 1'bx;
      ns = default;
    end
```
always @(present_state or in)
  case (present_state)
    IDLE   : begin
      out = 1'b0;
      if (in == 1'b1) next_state = S0;
      else next_state = IDLE;
    end
    S0     : begin
      out = 1'b0;
      if (in == 1'b1) next_state = S1;
      else next_state = IDLE;
    end
    S1     : begin
      out = 1'b1;
      if (in == 1'b1) next_state = S1;
      else next_state = IDLE;
    end
  default: begin
    next_state = IDLE;
    out = 1'b0;
  end
endcase
endmodule

The sequential semantics of the blocking assignment allows variables to be multiply assigned within a single always always block.
always @* begin
    next_state = IDLE;
    out = 1'b0;
    case (state)
        IDLE : if (in == 1'b1) next_state = S0;
        S0  : if (in == 1'b1) next_state = S1;
        S1  : begin
            out = 1'b1;
            if (in == 1'b1) next_state = S1;
        end
        default: ;
    endcase
end
Endmodule

Normal values: used unless specified below.

Within case only need to specify exceptions to the normal values.

Note: The use of “blocking assignments” allow signal values to be “rewritten”, simplifying the specification.