Finite State Machines (FSMs)

- FSM circuits are a type of sequential circuit:
  - output depends on present and past inputs
    - effect of past inputs is 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.

So far we have learned how to implement in Verilog. Now we learn how to design “by hand” to the gate level.

Parity Checker Example

A string of bits has “even parity” if the number of 1’s in the string is even.
- Design a circuit that accepts a bit-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?
Formal Design Process

"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.

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

<table>
<thead>
<tr>
<th>present state (ps)</th>
<th>OUT</th>
<th>IN</th>
<th>next state (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Derive logic equations from table (how?):

OUT = PS
NS = PS xor IN
Formal Design Process

Logic equations from table:

\[
\text{OUT} = \text{PS} \\
\text{NS} = \text{PS} \oplus \text{IN}
\]

• Circuit Diagram:

- XOR gate for NS calculation
- DFF to hold present state
- no logic needed for output in this example.

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
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.

Combinational Lock STD

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>*</td>
<td>START</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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

![Decoder logic diagram]

---

### Encoded ST Table

- **Assign states:**
  - START=000, OK1=001, OK2=011
  - BAD1=100, 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?**

---

Spring 2013  
EECS150 - Lec19-FSM  
Page 11
State Encoding

In general:

\[
\text{# of possible FSM state} = 2^{\text{# of Flip-flops}}
\]

Example:

state1 = 01, state2 = 11, state3 = 10, state4 = 00

However, often 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.

Ex: 3 States

\[
\begin{align*}
\text{STATE1: } & \quad 001 \\
\text{STATE2: } & \quad 010 \\
\text{STATE3: } & \quad 100 \\
\end{align*}
\]

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:

- In General:

  FFs must be initialized for correct operation (only one 1)

Circuit generated through direct inspection of the STD.

One-hot encoded combination lock
**FSM Implementation Notes**

- General FSM form:

```
| inputs        |
|-------|--------|
| CL | CL |
```

- All examples so far generate output based only on the present state:

```
<table>
<thead>
<tr>
<th>inputs</th>
<th>outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL</td>
<td>CL</td>
</tr>
</tbody>
</table>
```

- Commonly name **Moore Machine**
  (If output functions include both present state and input then called a **Mealy Machine**)

---

**Finite State Machines**

- **Example: Edge Detector**

  Bit are received one at a time (one per cycle), such as: 000111010 → \textit{time}

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

  Try two different solutions.
State Transition Diagram Solution A

<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

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

\[ NS_1 = IN \ PS_0 \]

\[ NS_0 = IN \]

\[ OUT = \overline{PS_1} \ PS_0 \]
Solution B

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

Let ZERO = 0, ONE = 1

IN | PS | NS | OUT
---|----|----|----
0  | 0  | 0  | 0  
0  | 1  | 0  | 0  
1  | 0  | 1  | 1  
1  | 1  | 1  | 0  

NS = IN, OUT = IN PS'

What's the intuition about this solution?

Edge detector timing diagrams

- Solution A: output follows the clock
- Solution B: output changes with input rising edge and is asynchronous wrt the clock.
**FSM Comparison**

**Solution A**

**Moore Machine**
- output function only of PS
- maybe **more** states (why?)
- synchronous outputs
  - no glitches
  - one cycle “delay”
  - full cycle of stable output

**Solution B**

**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.

---

**FSM Recap**

**Moore Machine**

- input value
- inputs
- present state
- next state
- outputs

**Mealy Machine**

- input value/output values
- inputs
- present state
- next state
- outputs

*Both machine types allow one-hot implementations.*
Final Notes on Moore versus Mealy

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:

![Mealy Machine Diagram]

### General FSM Design Process with Verilog

**Design Steps:** **Implementation**

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 separate always blocks for register assignment and CL logic block.
   - Use case for CL block. Within each case section 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.*
FSMs in Verilog

### Mealy Machine

```verilog
always @(posedge clk)
    if (rst) ps <= ZERO;
    else ps <= ns;
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
    endcase
```

### Moore Machine

```verilog
always @(posedge clk)
    if (rst) ps <= ZERO;
    else ps <= ns;
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
    endcase
```