Agenda

- Example Two – Pattern Recognizer
  - Mealy Machine Implementation
  - Moore Machine Implementation
- Glitch in FSM
  - Glitch problem
Review FSM design procedure

(1) Determine inputs and outputs
(2) Determine possible states of machine
   state minimization
(3) Encode states and outputs into a binary code
   state assignment or state encoding
   output encoding
   possibly input encoding (if under our control)
(4) Realize logic to implement functions for states and outputs
   combinational logic implementation and optimization
   choices in steps 2 and 3 can have large effect on resulting logic

Finite String Pattern Recognizer (1)
Tutorial on a simple design

- The pattern recognizer for tutorial
  - Recognize 110 and 101

```
<table>
<thead>
<tr>
<th>t</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

S0: input pattern ‘0’
When x(t-2, t-1, t) = "110" or "101", z = 1;
Otherwise z = 0;
Finite String Pattern Recognizer (2)

Tutorial on a simple design
- The pattern recognizer for tutorial
  - Recognize 110 and 101

States
S0: input pattern '0'
X
Z

<table>
<thead>
<tr>
<th>t</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

S1: input pattern '1'

Finite String Pattern Recognizer (3)

Tutorial on a simple design
- The pattern recognizer for tutorial
  - Recognize 110 and 101

States
S0: input pattern '0'
S1: input pattern '1'
X
Z

<table>
<thead>
<tr>
<th>t</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

S2: input pattern '11'
Tutorial on a simple design

- The pattern recognizer for tutorial
  - Recognize 110 and 101

States
S0: input pattern ‘0’
S1: input pattern ‘1’
S2: input pattern ‘11’
S3: input pattern ‘110’
S4: input pattern ‘101’
Tutorial on a simple design

- The pattern recognizer for tutorial
  - Recognize 110 and 101

<table>
<thead>
<tr>
<th>t</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

States:
- S0: input pattern '0'
- S1: input pattern '1'
- S2: input pattern '11'
- S3: input pattern '110'
- S4: input pattern '1101'
- S5: input pattern '10'

S0: input pattern '0'
Tutorial on a simple design

- The pattern recognizer for tutorial
  - Recognize 110 and 101

### States

<table>
<thead>
<tr>
<th>States</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>input pattern ‘0’</td>
</tr>
<tr>
<td>S1</td>
<td>input pattern ‘1’</td>
</tr>
<tr>
<td>S2</td>
<td>input pattern ‘11’</td>
</tr>
<tr>
<td>S3</td>
<td>input pattern ‘110’</td>
</tr>
<tr>
<td>S4</td>
<td>input pattern ‘101’</td>
</tr>
<tr>
<td>S5</td>
<td>input pattern ‘10’</td>
</tr>
</tbody>
</table>

### Table

<table>
<thead>
<tr>
<th>t</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

S1: input pattern ‘1’

S5: input pattern ‘10’
Tutorial on a simple design

- The pattern recognizer for tutorial
  - Recognize 110 and 101

<table>
<thead>
<tr>
<th>States</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S0: input patt. '0'</td>
<td></td>
</tr>
<tr>
<td>S1: input patt. '1'</td>
<td></td>
</tr>
<tr>
<td>S2: input patt. '11'</td>
<td></td>
</tr>
<tr>
<td>S3: input patt. '110'</td>
<td></td>
</tr>
<tr>
<td>S4: input patt. '101'</td>
<td></td>
</tr>
<tr>
<td>S5: input patt. '10'</td>
<td></td>
</tr>
</tbody>
</table>

X → 110  101 → Z

<table>
<thead>
<tr>
<th>t</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Z</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

S4: input pattern '101'

S5: input pattern '10'
Memorize the state transition:
To detect a pattern of 110, memorize the sequence 11 in the input, that is the state S2. In order to find the input sequence 11, the machine needs to memorize the input 1, which gives the state S1. State S3 means the machine find a input sequence of 10, and is for detecting the other pattern 101. State S0 means the machine does not find any (partial) match with the two patterns. S0 is also the initial state.

Find the transitions between the states.
For each state, determine the next state base on the current input X. For example, if the current state is S3 and the current input is 1, the next state is S1. The transition from S3 to S1 also means the machine detects a pattern 101, which should give the output 1. Note that when input is 1, the next state for S2 is S2, not S1.

---

### Mealy Machine Implementation

- Memorize the state transition:
- To detect a pattern of 110, memorize the sequence 11 in the input, that is the state S2. In order to find the input sequence 11, the machine needs to memorize the input 1, which gives the state S1. State S3 means the machine find a input sequence of 10, and is for detecting the other pattern 101. State S0 means the machine does not find any (partial) match with the two patterns. S0 is also the initial state.
- Find the transitions between the states.
- For each state, determine the next state base on the current input X. For example, if the current state is S3 and the current input is 1, the next state is S1. The transition from S3 to S1 also means the machine detects a pattern 101, which should give the output 1. Note that when input is 1, the next state for S2 is S2, not S1.
Manual State Assignment

Modify the VHDL program with state assignments.

The VHDL codes look like the following:

```
ARCHITECTURE BEHAVIOR OF MEALY IS
TYPE type_sreg IS (s0,s1,s2,s3);
attribute enum_encoding : string;
attribute enum_encoding of type_sreg : type
is "00 01 10 11";
SIGNAL sreg, next_sreg : type_sreg;
BEGIN
...
```

VHDL of Mealy Machine

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
------------------------------------------------
ENTITY string_detector IS
PORT (d,clk,rst: IN BIT; q: OUT BIT);
END string_detector;
------------------------------------------------
ARCHITECTURE mealy_machine OF string_detector IS
TYPE state IS (zero, one, two, three);
SIGNAL pr_state, nx_state: state;
BEGIN
-------------- Lower Section : ---------------------------
PROCESS (rst, clk)
BEGIN
IF (rst='1') THEN
pr_state <= zero;
ELSEIF (clk'EVENT AND clk='1') THEN
pr_state < = nx_state;
END IF;
END PROCESS;
---------------- Upper Section :-----------------------------
PROCESS (d, pr_state)
BEGIN
CASE pr_state IS
  WHEN zero =>
    q <= '0';
    IF (d='1') THEN nx_state <= two;
    ELSE nx_state <= zero;
    END IF;
  WHEN one =>
    q <= '0';
    IF (d='1') THEN nx_state <= two;
    ELSE nx_state <= zero;
    END IF;
  WHEN two =>
    q <= '0';
    IF (d='1') THEN nx_state <= three;
    ELSE nx_state <= zero;
    END IF;
  WHEN three =>
    q <= '0';
    IF (d='1') THEN nx_state <= zero;
    ELSE nx_state <= zero;
    END IF;
END CASE;
END PROCESS;
END mealy_machine;

Notice: This is only an example. Code is not exactly identical to the previous state diagram
Finite string pattern recognizer (step 1)

- Finite string pattern recognizer
  - one input (X) and one output (Z)
  - output is asserted whenever the input sequence …010... has been observed, as long as the sequence …100... has never been seen

- Step 1: understanding the problem statement
  - sample input/output behavior:
    
    \[
    X: \quad 0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 0 \quad 1 \quad 0 \ldots
    \]
    
    \[
    Z: \quad 0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0 \ldots
    \]
    
    \[
    X: \quad 1 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 0 \quad 0 \quad 1 \quad 0 \ldots
    \]
    
    \[
    Z: \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0 \ldots
    \]

Finite string pattern recognizer (step 2)

- Step 2: draw state diagram
  - for the strings that must be recognized, i.e., 010 and 100
  - a Moore implementation

![State Diagram](chart.png)
Finite string pattern recognizer (step 3)

- Exit conditions from state S3: have recognized …010
  - if next input is 0 then have …0100 = …100 (state S6)
  - if next input is 1 then have …0101 = …01 (state S2)
- Exit conditions from S1: recognizes strings of form …0 (no 1 seen)
  - loop back to S1 if input is 0
- Exit conditions from S4: recognizes strings of form …1 (no 0 seen)
  - loop back to S4 if input is 1

Finite string pattern recognizer (step 4)

- S2 and S5 still have incomplete transitions
  - S2 = …01; If next input is 1, then string could be prefix of (01)1(00) S4 handles just this case
  - S5 = …10; If next input is 1, then string could be prefix of (10)1(0) S2 handles just this case
- Reuse states as much as possible
  - look for same meaning
  - state minimization leads to smaller number of bits to represent states
- Once all states have a complete set of transitions we have a final state diagram
Finite string pattern recognizer (step 5)

- Verilog description including state assignment (or state encoding)

```verilog
module string (clk, X, rst, Q0, Q1, Q2, Z);
    input clk, X, rst;
    output Q0, Q1, Q2, Z;
    parameter S0 = [0,0,0]; // reset state
    parameter S1 = [0,0,1]; // strings ending in ...0
    parameter S2 = [0,1,0]; // strings ending in ...01
    parameter S3 = [0,1,1]; // strings ending in ...010
    parameter S4 = [1,0,0]; // strings ending in ...1
    parameter S5 = [1,0,1]; // strings ending in ...10
    parameter S6 = [1,1,0]; // strings ending in ...100
    reg state[0:2];
    assign Q0 = state[0];
    assign Q1 = state[1];
    assign Q2 = state[2];
    assign Z = (state == S3);
    always @(posedge clk) begin
        if (rst) state = S0;
        else
            case (state)
                S0: if (X) state = S4 else state = S1;
                S1: if (X) state = S2 else state = S1;
                S2: if (X) state = S4 else state = S3;
                S3: if (X) state = S2 else state = S6;
                S4: if (X) state = S4 else state = S5;
                S5: if (X) state = S2 else state = S6;
                S6: state = S6;
            default: begin
                $display("invalid state reached");
                state = 3'bxxx;
            end
            endcase
        end
endmodule
```

Finite string pattern recognizer

- Review of design process
  - Understanding problem
  - write down sample inputs and outputs to understand specification
  - Derive a state diagram
  - write down sequences of states and transitions for sequences to be recognized
  - Minimize number of states
  - add missing transitions; reuse states as much as possible
  - State assignment or encoding
  - encode states with unique patterns
  - Generate VHDL/Verilog codes
  - Simulate realization in QuartusII (or ModelSim)
  - verify I/O behavior of your state diagram to ensure it matches specification
**What is Glitches?**

- **Glitches:**
  - The undesired transitions observed at the output
  - An example on XOR gate

- Two input transitions
- Transitions happen at different time
- Output is identical before and after the two input transitions, but it's different between the two transitions.

=> Glitch will happen between the two transitions

---

**Glitch in Pattern Recognizer**

\[ Z = f(\text{State}, X) \]
How to Avoid Glitch?

- Glitch is an undesired output transition. Glitch happens due to transit output changes between multiple input transitions.
- Mealy machine is more likely to have glitches, because it’s output depends on inputs.
- Moore machine is less likely to have glitches, since it’s outputs depends on FFs’ output, which may change close to the same time.

Example of Level-to-pulse FSM

- A level-to-pulse converter produces a single-cycle pulse each time its input goes high.
- In other words, it’s a synchronous rising-edge detector.
- Sample uses:
  - Buttons and switches pressed by humans for arbitrary periods of time
  - Single-cycle enable signals for counters

Whenever input L goes from low to high...

...output P produces a single pulse, one clock period wide.
Design of Moore Level-to-pulse converter

- Block diagram of desired system:

- State transition diagram is a useful FSM representation and design aid

Moore Finite State Machine

- Transition diagram is readily converted to a state transition table (just a truth table)

- Combinational logic may be derived by Karnaugh maps
Moore Machine Implementation

Moore FSM circuit implementation of level-to-pulse converter:

Design of Mealy Level-to-pulse Converter

- Since outputs are determined by state and inputs, Mealy FSMs may need fewer states than Moore FSM implementations.

1. When \( L=1 \) and \( S=0 \), this output is asserted immediately and until the state transition occurs (or \( L \) changes).

2. After the transition to \( S=1 \) and as long as \( L \) remains at 1, this output is asserted.
Mealy Level-to-pulse Converter

Mealy FSM circuit implementation of level-to-pulse converter:

- FSM’s state simply remembers the previous value of L
- Circuit benefits from the Mealy FSM’s implicit single-cycle assertion of outputs during state transitions

Moore and Mealy Machine Comparison

- Remember that the difference is in the output:
  - Moore outputs are based on state only
  - Mealy outputs are based on state and input
  - Therefore, Mealy outputs generally occur one cycle earlier than a Moore:

Moore: delayed assertion of P

Mealy: immediate assertion of P

- Compared to a Moore FSM, a Mealy FSM might...
  - Be more difficult to conceptualize and design
  - Have fewer states
FSM Timing Requirement

- Timing requirements for FSM are identical to any generic sequential system with feedback

![Diagram showing timing requirements for FSM](image)

\[ T > T_{cq} + T_{\text{logic}} + T_{su} \]

\[ T_{cq,cd} + T_{\text{logic,cd}} > T_{\text{hold}} \]