Overview

◆ Last lecture
  ● Introduction to finite-state machines
  ● Moore versus Mealy machines
  ● Synchronous Mealy machines
  ● Example: A parity checker

◆ Today
  ● Example: A sequence detector FSM
  ● Example: A vending machine FSM
  ● FSMs in Verilog

FSM design

◆ FSM-design procedure
  1. State diagram and state-transition table
  2. State minimization
  3. State assignment (or state encoding)
  4. Minimize next-state logic
  5. Implement the design

Example: Sequence detector

◆ Design a circuit to detect 3 or more 1’s in a bit string
  ● Assume Moore machine
  ● Assume D flip-flops
  ● Assume flip-flops have a reset

1. State diagram and state-transition table

2. State minimization & 3. State encoding

◆ State diagram is already minimized

◆ Try a binary encoding

4. Minimize next-state logic

Notation

- \( M \) := MSB
- \( L \) := LSB
- \( \text{In} \) := Input
5. Implement the design

Design example: A vending machine

- Release item after receiving 15 cents
- Single coin slot for dimes and nickels
- Sensor specifies coin type
- Machine does not give change

1a. State diagram

- Consider input sequences
  - 3 nickels
  - 2 nickels, dime
  - nickel, dime
  - dime, nickel
  - two dimes

- Draw state diagram
  - Assume Moore machine
  - Inputs: N, D, reset
  - Output: Open

1b. Symbolic state-transition table

<table>
<thead>
<tr>
<th>present state</th>
<th>inputs</th>
<th>next state</th>
<th>present output</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>1</td>
<td>0</td>
</tr>
<tr>
<td>5¢</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5¢</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10¢</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10¢</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>15¢</td>
<td>–</td>
<td>–</td>
<td>15¢</td>
</tr>
</tbody>
</table>

2. State minimization

- Reuse states where possible
  - Notice we can use the deposited coin values for states
  - State is the same if we input 2 nickels or 1 dime

3. State encoding

- Encode states uniquely
  - 4 states:
    - 2 bits minimum
    - 4 bits maximum
  - Look for optimal encoding
  - Assume D flip-flops
4. Minimize the logic

K-map for $P_1$

<table>
<thead>
<tr>
<th>$Q_1Q_0$</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_1$</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>$D_0$</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

K-map for $P_0$

<table>
<thead>
<tr>
<th>$Q_1Q_0$</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_1$</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>$D_0$</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

K-map for Open

<table>
<thead>
<tr>
<th>$Q_1Q_0$</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_1$</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>$D_0$</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

$P_1 = Q_1 + D + Q_0N$

$P_2 = Q_1'N + Q_2'N + Q_0 + Q_0D$

$OPEN = Q_1Q_0$

if FFs do not have a reset pin then

$P_1 = reset(Q_1 + D + Q_0N)$

$P_2 = reset(Q_1'N + Q_2'N + Q_0 + Q_0D)$

5. Implement the design

5. Implement the design

Retiming design

- OPEN is delayed by AND gate after $Q_1$ and $Q_0$
  - Can remove this delay by retiming
  - Move output logic (AND gate) to eliminate delay
  - $OPEN = Q_1Q_0 = (Q_1 + D + Q_0N)(Q_0N' + Q_0N' + Q_0N + Q_0D)$

Moore versus Mealy vending machine

Moore machine

Mealy machine

Hardware Description Languages and Sequential Logic

- Flip-flops
  - representation of clocks - timing of state changes
  - asynchronous vs. synchronous

- FSMs
  - structural view (FFs separate from combinational logic)
  - behavioral view (synthesis of sequencers – not in this course)

- Data-paths = data computation (e.g., ALUs, comparators) + registers
  - use of arithmetic/logical operators
  - control of storage elements

Example: reduce-1-string-by-1

- Remove one 1 from every string of 1s on the input
module reduce (clk, reset, in, out);
input clk, reset, in;
output out;
reg out;
reg [2:1] state; // state variables
reg [2:1] next_state;
always @(posedge clk)
if (reset) state = `zero;
else       state = next_state;
always @(in or state)
case (state)
  `zero: // last input was a zero
        begin
        if (in) next_state = `one1;
        else    next_state = `zero;
        end
  `one1: // we've seen one 1
        begin
        if (in) next_state = `two1s;
        else    next_state = `zero;
        end
  `two1s: // we've seen at least 2 ones
        begin
        if (in) next_state = `two1s;
        else    next_state = `zero;
        end
endcase
always @(state)
case (state)
  `zero: out = 0;
  `one1: out = 0;
  `two1s: out = 1;
endcase
endmodule

(* *)

module reduce (clk, reset, in, out);
input clk, reset, in;
output out;
reg out;
reg state; // state variables
reg next_state;
always @(posedge clk)
if (reset) state = `zero;
else
  case (state)
    `zero: // last input was a zero
      begin
      out = 0;
      if (in) state = `one;
      else    state = `zero;
      end
    `one: // we've seen one 1
      begin
      if (in) begin
      state = `one; out = 1;
      end else begin
      state = `zero; out = 0;
      end
    endcase
end)
endmodule

Synchronous Mealy Machine
module reduce (clk, reset, in, out);
input clk, reset, in;
output out;
reg out;
reg state; // state variables
always @(posedge clk)
if (reset) state = `zero;
else
  case (state)
    `zero: // last input was a zero
      begin
      out = 0;
      if (in) state = `one;
      else    state = `zero;
      end
    `one: // we've seen one 1
      begin
      if (in) begin
      state = `one; out = 1;
      end else begin
      state = `zero; out = 0;
      end
    endcase
end)