Finite State Machines

- Finite State Machines (FSMs)
  - general models for representing sequential circuits
  - two principal types based on output behavior (Moore and Mealy)
- Basic sequential circuits revisited and cast as FSMs
  - shift registers
  - counters
- Design procedure for FSMs
  - state diagrams
  - state transition table
  - next state functions
  - potential optimizations
- Hardware description languages

Abstraction of state elements

- Divide circuit into combinational logic and state
- Localize the feedback loops and make it easy to break cycles
- Implementation of storage elements leads to various forms of sequential logic
Forms of sequential logic

- Asynchronous sequential logic – state changes occur whenever state inputs change (seq. elements may be simple wires or delay elements)
- Synchronous sequential logic – state changes occur in lock step across all storage elements (using a periodic waveform to trigger FFs)

Finite state machine representations

- States: determined by possible values in sequential storage elements
- Transitions: change of state
- Clock: controls when state can change by controlling storage elements
- Sequential logic
  - transitions through a series of states
  - which transitions are taken depends on values of input signals
  - clock period defines elements of input sequence

In = 0
In = 1
In = 0
In = 1
100
010
110
111
001
In = 1
In = 0
In = X
In = X
010
001
1
010
001
1
010

Example finite state machine diagram

- 5 states
- 8 other transitions between states
  - 6 conditioned by input
  - 1 self-transition (on 0 from 001 to 001)
  - 2 independent of input (to/from 111)
- 1 reset transition (from all states) to state 100
  - represents 5 transitions (from each state to 100), one a self-arc
  - simplifies condition on other transitions – all would include AND reset'
  - short-hand – rather than drawing a transition arc from each state

Counters are simple finite state machines

- Counters
  - proceed through well-defined sequence of states (if enabled)
- Many types of counters: binary, BCD, Gray-code, etc.…
  - 3-bit up-counter: 000, 001, 010, 011, 100, 101, 110, 111, 000, ...
  - 3-bit down-counter: 111, 110, 101, 100, 011, 010, 001, 000, 111, ...

3-bit up-counter

reset
Can any sequential system be represented with a state diagram?

- **Shift register**
  - input value shown on transition arcs
  - output values shown within state node

![State Diagram](image)

How do we turn a state diagram into logic?

- **Counter**
  - 3 flip-flops to hold state
  - logic to compute next state
  - clock signal controls when flip-flop memory can change
    - wait long enough for combinational logic to compute new value
    - though waiting too long is a waste of time

![Logic Diagram](image)
FSM design procedure

- We started with counters
  - simple because the output is just its state
  - simple because there is no input used to choose next state

- State diagram to state transition table
  - tabular form of state diagram
  - like a truth-table

- State encoding
  - decide on representation of states
  - for counters it is simple: just its value

- Implementation
  - flip-flop for each state bit
  - combinational logic based on encoding

FSM design procedure: state diagram to encoded state transition table

- Tabular form of state diagram
- Like a truth-table (specify output for all input combinations)
- Encoding of states: easy for counters – just use value
Implementation

- D flip-flop for each state bit
- Combinational logic based on state encoding

Verilog notation to show function represents an input to D-FF

\[
\begin{align*}
N1 &= C1' \\
N2 &= C1C2' + C1'C2 \\
N3 &= C1C2C3' + C1'C3 + C2'C3 \\
\end{align*}
\]

In

<table>
<thead>
<tr>
<th>C3</th>
<th>C2</th>
<th>C1</th>
<th>N3</th>
<th>N2</th>
<th>N1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1 0 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 0 0 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 1 1 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 0 1 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 1 1 1 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 0 1 1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1 0 1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1 1 1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Back to the shift register

- Input determines next state

<table>
<thead>
<tr>
<th>In</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>N1</th>
<th>N2</th>
<th>N3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0 1 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1 0 0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 0 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 1 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 0 0 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 1 0 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 0 0 1 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 1 0 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 0 1 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1 0 1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1 1 1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Autumn 2010  CSE370 - XVI - Finite State Machines  11

Autumn 2010  CSE370 - XVI - Finite State Machines  12
More complex counter example

- Complex counter
  - repeats 5 states in sequence
  - not a binary number representation
- Step 1: derive the state transition diagram
  - count sequence: 000, 010, 011, 101, 110
- Step 2: derive the state transition table from the state transition diagram

```
<table>
<thead>
<tr>
<th>Present State</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

note the don’t care conditions that arise from the unused state codes

More complex counter example (cont’d)

- Step 3: K-maps for next state functions

```
A
C+                B+                A+
| C | 0  | 0  | 0  | X     | 0  | 0  | 1  | X     | 0  | 1  | 0  | X   |
| A | X | 1  | X  | 1     | X | 0  | X  | 1     | X | 1  | 1  | 0   |
```

\[
C+ \leq A
\]
\[
B+ \leq B' + A'C'
\]
\[
A+ \leq BC'
\]
Self-starting counters (cont’d)

- Re-deriving state transition table from don’t care assignment

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C+</th>
<th>C</th>
<th>B+</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C+</th>
<th>C</th>
<th>A+</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Present State | Next State
---|---
000 | 110
100 | 010
010 | 101
110 | 000
001

Self-starting counters

- Start-up states
  - at power-up, counter may be in an unused or invalid state
  - designer must guarantee that it (eventually) enters a valid state

- Self-starting solution
  - design counter so that invalid states eventually transition to a valid state
    - this may or may not be acceptable
    - may limit exploitation of don’t cares
Activity

- 2-bit up-down counter (2 inputs)
  - direction: D = 0 for up, D = 1 for down
  - count: C = 0 for hold, C = 1 for count

<table>
<thead>
<tr>
<th>S1</th>
<th>S0</th>
<th>C</th>
<th>D</th>
<th>N1</th>
<th>N0</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>11</td>
<td>C=0</td>
<td>D=X</td>
<td>C=0</td>
<td>D=X</td>
</tr>
<tr>
<td>00</td>
<td>11</td>
<td>C=1</td>
<td>D=1</td>
<td>C=1</td>
<td>D=1</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>C=0</td>
<td>D=0</td>
<td>C=0</td>
<td>D=0</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>C=1</td>
<td>D=0</td>
<td>C=0</td>
<td>D=X</td>
</tr>
</tbody>
</table>

Activity

- 2-bit up-down counter (2 inputs)
  - direction: D = 0 for up, D = 1 for down
  - count: C = 0 for hold, C = 1 for count

<table>
<thead>
<tr>
<th>S1</th>
<th>S0</th>
<th>C</th>
<th>D</th>
<th>N1</th>
<th>N0</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Activity (cont’d)

Autumn 2010  CSE370 - XVI - Finite State Machines

Counter/shift-register model

- Values stored in registers represent the state of the circuit
- Combinational logic computes:
  - next state
    - function of current state and inputs
  - outputs
    - values of flip-flops
General state machine model

- Values stored in registers represent the state of the circuit
- Combinational logic computes:
  - next state
    - function of current state and inputs
  - outputs
    - function of current state and inputs (**Mealy machine**)
    - function of current state only (**Moore machine**)

State machine model (cont’d)

- States: \( S_1, S_2, ..., S_k \)
- Inputs: \( I_1, I_2, ..., I_m \)
- Outputs: \( O_1, O_2, ..., O_n \)
- Transition function: \( F_s(S_i, I_j) \)
- Output function: \( F_o(S_i) \) or \( F_o(S_i, I_j) \)
Comparison of Mealy and Moore machines (cont’d)

- **Moore**

- **Mealy**

- **Synchronous Mealy**

Comparison of Mealy and Moore machines

- Mealy machines tend to have less states
  - outputs depend on arc taken from a state to another state ($n^2$) rather than just the state of the FSM ($n$)

- Moore machines are safer to use
  - outputs change at next clock edge
  - in Mealy machines, input change can cause async output change (after prop delay of logic) – a BIG problem when two machines are interconnected – asynchronous feedback may occur if one isn’t careful (input to fsm1, changes output of fsm1, which is an input to fsm2, whose output changes, and turns out to be input to fsm1)

- Mealy machines advantage? – they react faster to inputs
  - react in same cycle – don’t need to wait for clock
  - in Moore machines, more logic may be necessary to decode state into outputs that are needed – more gate delays after clock edge
Specifying outputs for a Moore machine

- Output is only function of state
  - specify in state bubble in state diagram
  - example: sequence detector for 01 or 10

<table>
<thead>
<tr>
<th>reset</th>
<th>input</th>
<th>current state</th>
<th>next state</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>A</td>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>A</td>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>A</td>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>B</td>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>B</td>
<td>D</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>C</td>
<td>E</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>C</td>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>D</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>D</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>E</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>E</td>
<td>D</td>
<td>1</td>
</tr>
</tbody>
</table>

Specifying outputs for a Mealy machine

- Output is function of state and inputs
  - specify output on transition arc between states
  - example: sequence detector for 01 or 10

<table>
<thead>
<tr>
<th>reset</th>
<th>input</th>
<th>current state</th>
<th>next state</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>A</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>A</td>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>A</td>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>B</td>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>B</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>C</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>C</td>
<td>C</td>
<td>0</td>
</tr>
</tbody>
</table>
Synchronous Mealy machine (really Moore)

- Synchronous (or registered) Mealy machine
  - state AND output FFs
  - avoids ‘glitchy’ outputs (no hazards)
  - easy to implement in programmable logic (function blocks + FF)
- Same as a Moore machine with no output decoding
  - outputs computed on transition to next state rather than after entering state
  - view outputs as expanded state vector – “output-encoded state”

![Diagram of Synchronous Mealy machine]

Tug-of-War Game FSM

- Tug of War game
  - 7 LEDS, 2 push buttons (LPB, RPB)

![Diagram of Tug-of-War Game FSM]
Light Game FSM Verilog

module Light_Game (LEDS, LPB, RPB, CLK, RESET);

input LPB;
input RPB;
input CLK;
input RESET;
output [6:0] LEDS;
reg [6:0] position;
reg left;
reg right;

always @(posedge CLK)
begin
    left <= LPB;
    right <= RPB;
    if (RESET) position <= 7'b0001000;
    else if ((position == 7'b0000001) || (position == 7'b1000000))
    else if (L) position <= position << 1;
    else if (R) position <= position >> 1;
end

download

Do you see a problem with this game?

Activity

- Where is the problem? What is the fix?

always @(posedge CLK)
begin
    left <= LPB;
    right <= RPB;
    if (RESET) position <= 7'b0001000;
    else if ((position == 7'b0000001) || (position == 7'b1000000))
        position <= position;
    else if (L) position <= position << 1;
    else if (R) position <= position >> 1;
end

always @(posedge CLK) begin // no longer biased in favor of L player
    left <= LPB;
    right <= RPB;
    if (RESET) position <= 7'b0001000;
    else if ((position == 7'b0000001) || (position == 7'b1000000))
        position <= position;
    else if (L & ~R) position <= position << 1; // correct error in state diag.
    else if (R & ~L) position <= position >> 1; // favoring L player
    else
        position <= position; // otherwise, just hold
Example: vending machine

- Release item after 15 cents are deposited
- Single coin slot for dimes, nickels
- No change

Example: vending machine (cont’d)

- Suitable abstract representation
  - tabulate typical input sequences:
    - 3 nickels
    - nickel, dime
    - dime, nickel
    - two dimes
  - draw state diagram:
    - inputs: N, D, reset
    - output: open chute
  - assumptions:
    - assume N and D asserted for one cycle
    - each state has a self loop for N = D = 0 (no coin)
Example: vending machine (cont’d)

- Suitable abstract representation
  - tabulate typical input sequences:
    - 3 nickels
    - nickel, dime
    - dime, nickel
    - two dimes
  - draw state diagram:
    - inputs: N, D, reset
    - output: open chute
  - assumptions:
    - assume N and D asserted for one cycle
    - each state has a self loop for N = D = 0 (no coin)

Activity: reuse states

- Redraw the state diagram using as few states as possible
Example: vending machine (cont’d)

- Minimize number of states - reuse states whenever possible

![Symbolic state table](image)

Example: vending machine (cont’d)

- Uniquely encode states

![Uniquely encode states](image)
Example: Moore implementation

- Mapping to logic

\[
D_1 = Q_1 + D + Q_0 N
\]
\[
D_0 = Q_0' N + Q_0 N' + Q_1 N + Q_1 D
\]
\[
OPEN = Q_1 Q_0
\]

Equivalent Mealy and Moore state diagrams

- Moore machine
  - outputs associated with state

- Mealy machine
  - outputs associated with transitions
Example: Mealy implementation

<table>
<thead>
<tr>
<th>present state</th>
<th>inputs</th>
<th>next state</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>Q0</td>
<td>D</td>
<td>N</td>
</tr>
<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>1</td>
</tr>
<tr>
<td>1</td>
<td>0</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>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ D_0 = Q_0'N + Q_0N' + Q_1N + Q_1D \]
\[ D_1 = Q_1 + D + Q_0N \]
\[ OPEN = Q_1Q_0 + Q_1N + Q_1D + Q_0D \]
Vending machine: Moore to synch. Mealy

- OPEN = Q1Q0 creates a combinational delay after Q1 and Q0 change in Moore implementation
- This can be corrected by retiming, i.e., move flip-flops and logic through each other to improve delay – pre-compute OPEN then store it in FF
- OPEN.d = (Q1 + D + Q0N)(Q0'N + Q0N' + Q1N + Q1D) = Q1Q0N' + Q1N + Q1D + Q0'ND + Q0N'D
- Implementation now looks like a synchronous Mealy machine
  - another reason programmable devices have FF at end of logic

Vending machine: Mealy to synch. Mealy

- OPEN.d = Q1Q0 + Q1N + Q1D + Q0D
- OPEN.d = (Q1 + D + Q0N)(Q0'N + Q0N' + Q1N + Q1D) = Q1Q0N' + Q1N + Q1D + Q0'ND + Q0N'D
Vending machine example (Moore PLD mapping)

\[
\begin{align*}
D_0 &= \text{reset}'(Q_0'N + Q_0N' + Q_1N + Q_1D) \\
D_1 &= \text{reset}'(Q_1 + D + Q_0N) \\
\text{OPEN} &= Q_1Q_0
\end{align*}
\]

Vending machine (synch. Mealy PLD mapping)

\[
\text{OPEN} = \text{reset}'(Q_1Q_0N' + Q_1N + Q_1D + Q_0'ND + Q_0N'D)
\]
One-hot encoded transition table

<table>
<thead>
<tr>
<th>present state inputs</th>
<th>next state</th>
<th>output</th>
<th>open</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q_0 Q_1 Q_2 D N</td>
<td>D_0 D_1 D_2 D_3</td>
<td>Q_0 D' N'</td>
<td></td>
</tr>
<tr>
<td>0 0 0 1 0 0</td>
<td>0 0 0 1 0 0</td>
<td>0 0 0 1 0 0</td>
<td>0 0 0 1 0 0</td>
</tr>
<tr>
<td>0 1 0 0 1 0</td>
<td>0 1 0 0 1 0</td>
<td>0 0 0 1 0 0</td>
<td>0 0 0 1 0 0</td>
</tr>
<tr>
<td>1 0 0 1 0 0</td>
<td>1 0 0 1 0 0</td>
<td>0 0 0 1 0 0</td>
<td>0 0 0 1 0 0</td>
</tr>
<tr>
<td>1 1 0 1 0 0</td>
<td>1 1 0 1 0 0</td>
<td>0 0 0 1 0 0</td>
<td>0 0 0 1 0 0</td>
</tr>
<tr>
<td>0 0 1 0 0 0</td>
<td>0 0 1 0 0 0</td>
<td>0 0 1 0 0 0</td>
<td>0 0 1 0 0 0</td>
</tr>
<tr>
<td>0 1 0 0 0 0</td>
<td>0 1 0 0 0 0</td>
<td>0 0 1 0 0 0</td>
<td>0 0 1 0 0 0</td>
</tr>
<tr>
<td>1 0 0 0 0 0</td>
<td>1 0 0 0 0 0</td>
<td>0 0 1 0 0 0</td>
<td>0 0 1 0 0 0</td>
</tr>
<tr>
<td>1 1 0 0 0 0</td>
<td>1 1 0 0 0 0</td>
<td>0 0 1 0 0 0</td>
<td>0 0 1 0 0 0</td>
</tr>
</tbody>
</table>

D_0 = Q_0 D' N'
D_1 = Q_0 N + Q_1 D' N'
D_2 = Q_0 D + Q_1 N + Q_2 D' N'
D_3 = Q_1 D + Q_2 D + Q_2 N + Q_3
OPEN = Q_3

Designing from the state diagram

D_0 = Q_0 D' N'
D_1 = Q_0 N + Q_1 D' N'
D_2 = Q_0 D + Q_1 N + Q_2 D' N'
D_3 = Q_1 D + Q_2 D + Q_2 N + Q_3
OPEN = Q_3
Output encoding

- Reuse outputs as state bits
  - Why create new functions when you can use outputs?
  - Bits from state assignments are the outputs for that state
    - Take outputs directly from the flip-flops
- Ad hoc - no tools
  - Yields small circuits for most FSMs
  - Fits nicely with synchronous Mealy machines

Mealy and Moore examples

- Recognize $A, B = 0,1$
  - Mealy or Moore?
Mealy and Moore examples (cont’d)

- Recognize A,B = 1,0 then 0,1
  - Mealy or Moore?

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 (a filter)
  - E.g., 00011100 -> 00001100; 00100110 -> 00000010

Moore

Mealy

Verilog FSM - Reduce 1s example

- Moore machine

```verilog
module reduce (clk, reset, in, out);
input clk, reset, in;
output out;

parameter zero  = 2'b00;
parameter one1  = 2'b01;
parameter two1s = 2'b10;

reg out;
reg [2:1] state; // state variables
reg [2:1] next_state;

always @(posedge clk)
  if (reset) state = zero;
  else    state = next_state;
endmodule
```
Moore Verilog FSM (cont’d)

always @(in or state)
  case (state)
    zero: // last input was a zero
      begin
        if (in) next_state = one;
        else    next_state = zero;
      end
    one: // we've seen one 1
      begin
        if (in) next_state = twols;
        else    next_state = zero;
      end
    twols: // we've seen at least 2 ones
      begin
        if (in) next_state = twols;
        else    next_state = zero;
      end
  endcase

Mealy Verilog FSM

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       state = next_state;
  always @(in or state)
    case (state)
      zero: // last input was a zero
        begin
          out = 0;
          if (in) next_state = one;
          else    next_state = zero;
        end
      one: // we've seen one 1
        begin
          if (in) next_state = one; out = 1;
          else begin
            next_state = zero; out = 0;
          end
        end
    endcase
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
        if (in) begin
          state = one; out = 1;
        end else begin
          state = zero; out = 0;
        end
    endcase
endmodule

Finite state machines summary

- Models for representing sequential circuits
  - abstraction of sequential elements
  - finite state machines and their state diagrams
  - inputs/outputs
  - Mealy, Moore, and synchronous Mealy machines
- Finite state machine design procedure
  - deriving state diagram
  - deriving state transition table
  - determining next state and output functions
  - implementing combinational logic
- Hardware description languages