EECS 150 - Components and Design Techniques for Digital Systems

FSMs
9/11/2007

Sarah Bird
Electrical Engineering and Computer Sciences
University of California, Berkeley

Slides borrowed from David Culler Fa04 and Randy Katz Sp07
Sequential Logic Implementation

- Models for representing sequential circuits
  - Finite-state machines (Moore and Mealy)
  - Representation of memory (states)
  - Changes in state (transitions)

- Design procedure
  - State diagrams
  - State transition table
  - Next state functions
Recall: What makes Digital Systems tick?
**Abstraction of State Elements**

- Divide circuit into combinational logic and state
- Localize 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 (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 - the clock)
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**
- Sequences through a series of states
- Based on sequence of values on input signals
- Clock period defines elements of sequence
Can Any Sequential System be Represented with a State Diagram?

- **Shift Register**
  - Input value shown on transition arcs
  - Output values shown within state node
Two Kinds of FSMs

- Moore Machine vs Mealy Machine

Moore Machine

- State (t+1) = F (state(t), input(t))
- Output (t) = G(state(t))

Input / Out

Mealy Machine

- State (t+1) = F (state(t), input)
- Output (t) = G(state(t), Input)

Input / Out
Counters are Simple Finite State Machines

- **Counters**
  - Proceed thru well-defined state sequence in response to enable

- **Many types of counters:** binary, BCD, Gray-code
  - 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, ...
Verilog Upcounter

module binary_cntr (q, clk)
    inputs    clk;
    outputs    [2:0] q;
    reg        [2:0] q;
    reg        [2:0] p;

always @(q)            //Calculate next state
    case (q)
        3'b000: p = 3'b001;
        3'b001: p = 3'b010;
        ...    
        3'b111: p = 3'b000;
    endcase

always @(posedge clk)  //next becomes current state
    q <= p;
endmodule
How Do We Turn a State Diagram into Logic?

- **Counter**
  - Three 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
    - Don't wait too long as that is low performance

```
    D  Q    D  Q    D  Q
  CLK   OUT1  OUT2  OUT3

      "1"
```
FSM Design Procedure

- **Start with counters**
  - Simple because output is just state
  - Simple because no choice of next state based on input

- **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

### 3-bit up-counter

<table>
<thead>
<tr>
<th>current state</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
</tr>
</tbody>
</table>
Implementation

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

<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</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

N1 := C1'
N2 := C1C2' + C1'C2
     := C1 \text{ xor } C2
N3 := C1C2C3' + C1'C3 + C2'C3
     := C1C2C3' + (C1' + C2')C3
     := (C1C2) \text{ xor } C3

notation to show function represent input to D-FF
Parity Checker FSM

- 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.

Example:

<table>
<thead>
<tr>
<th>bit stream</th>
<th>IN</th>
<th>Parity Checker</th>
<th>OUT</th>
<th>0 if even parity</th>
<th>1 if odd parity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CLK</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example:

```
0 0 1 1 1 0 1
even even odd even odd odd even
```

“State Transition Diagram”
Formal Design Process

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:

OUT = PS
NS = PS \text{ xor } \text{IN}

Circuit Diagram:

- XOR gate for ns calculation
- DFF to hold present state
- no logic needed for output

Review of Design Steps:

1. Circuit functional specification
2. State Transition Diagram
3. Symbolic State Transition Table
4. Encoded State Transition Table
5. Derive Logic Equations
6. Circuit Diagram

FFs for state
CL for NS and OUT
Another example

Door combination lock:
- punch in 3 values in sequence and the door opens; if there is an error the lock must be reset; once the door opens the lock must be reset
- inputs: sequence of input values, reset
- outputs: door open/close
- memory: must remember combination or always have it available as an input
Sequential example: abstract control

- Finite-state diagram
  - States: 5 states
    - represent point in execution of machine
    - each state has outputs
  - Transitions: 6 from state to state, 5 self transitions, 1 global
    - changes of state occur when clock says it’s ok
    - based on value of inputs
  - Inputs: reset, new, results of comparisons
  - Output: open/closed
Sequential example (cont’d): finite-state machine

Finite-state machine
- generate state table (much like a truth-table)

Symbolic states

<table>
<thead>
<tr>
<th>reset</th>
<th>new</th>
<th>equal</th>
<th>state</th>
<th>next state</th>
<th>mux</th>
<th>open/closed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>S1</td>
<td>C1</td>
<td>closed</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>S1</td>
<td>S1</td>
<td>C1</td>
<td>closed</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>S1</td>
<td>ERR</td>
<td>-</td>
<td>closed</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>S1</td>
<td>S2</td>
<td>C2</td>
<td>closed</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>S2</td>
<td>S2</td>
<td>C2</td>
<td>closed</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>S2</td>
<td>ERR</td>
<td>-</td>
<td>closed</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>S2</td>
<td>S3</td>
<td>C3</td>
<td>closed</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>S3</td>
<td>S3</td>
<td>C3</td>
<td>closed</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>S3</td>
<td>ERR</td>
<td>-</td>
<td>closed</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>S3</td>
<td>OPEN</td>
<td>-</td>
<td>closed</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>OPEN</td>
<td>OPEN</td>
<td>-</td>
<td>open</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>ERR</td>
<td>ERR</td>
<td>-</td>
<td>closed</td>
</tr>
</tbody>
</table>
Sequential example: encoding

- **Encode state table**
  - state can be: S1, S2, S3, OPEN, or ERR
    - needs at least 3 bits to encode: 000, 001, 010, 011, 100
    - and as many as 5: 00001, 00010, 00100, 01000, 10000
    - choose 4 bits: 0001, 0010, 0100, 1000, 0000

- **Encode outputs**
  - output mux can be: C1, C2, or C3
    - needs 2 to 3 bits to encode
    - choose 3 bits: 001, 010, 100
  - output open/closed can be: open or closed
    - needs 1 or 2 bits to encode
    - choose 1 bits: 1, 0
Sequential example (cont’d): encoding

- Encode state table
  - state can be: S1, S2, S3, OPEN, or ERR
    - choose 4 bits: 0001, 0010, 0100, 1000, 0000
  - output mux can be: C1, C2, or C3
    - choose 3 bits: 001, 010, 100
  - output open/closed can be: open or closed
    - choose 1 bits: 1, 0

<table>
<thead>
<tr>
<th>reset</th>
<th>new</th>
<th>equal</th>
<th>state</th>
<th>next state</th>
<th>mux</th>
<th>open/closed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0001</td>
<td>001</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>0001</td>
<td>0001</td>
<td>001</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0001</td>
<td>0000</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0001</td>
<td>0010</td>
<td>010</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>0010</td>
<td>0010</td>
<td>010</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0010</td>
<td>0000</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0010</td>
<td>0100</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>0100</td>
<td>0100</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0100</td>
<td>0000</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0100</td>
<td>1000</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>1000</td>
<td>1000</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>0000</td>
<td>0000</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>

- good choice of encoding!
- mux is identical to last 3 bits of next state
- open/closed is identical to first bit of state
State Minimization

- Fewer states may mean fewer state variables
- High-level synthesis may generate many redundant states
- Two state are equivalent if they are impossible to distinguish from the outputs of the FSM, i.e., for any input sequence the outputs are the same

- Two conditions for two states to be equivalent:
  1) Output must be the same in both states
  2) Must transition to equivalent states for all input combinations
Sequential Logic Implementation 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