Sequential logic implementation

- Sequential circuits
  - primitive sequential elements
  - combinational logic
- Models for representing sequential circuits
  - finite-state machines (Moore and Mealy)
  - representation of memory (states)
  - changes in state (transitions)
- Basic sequential circuits
  - shift registers
  - counters
- Design procedure
  - state diagrams
  - state transition table
  - next state functions

Any sequential system can be represented with a state diagram

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

State machine model

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

State Machine Model

- States: S1, S2, ..., Sk
- Inputs: I1, I2, ..., In
- Outputs: O1, O2, ..., On
- Transition function: F(s, I)
- Output function: F(s) or F(s, I)

How do we turn a state diagram into logic?

- e.g. counter
  - flip-flops to hold state
  - logic to compute next state
  - clock signal controls when flip-flop memory can change
    - wait just long enough for combinational logic to compute new value

FSM design procedure

- Describe FSM behavior, e.g. state diagram
  - Inputs and Outputs
  - States (symbolic)
  - State transitions
- State diagram to state transition table, i.e. truth table
  - Inputs: inputs and current state
  - Outputs: outputs and next state
- State encoding
  - decide on representation of states
  - lots of choices
- Implementation
  - flip-flop for each state bit
  - synthesize combinational logic from encoded state table
Design Problem – Run-Length Encoder

- 7-bit input stream
- 8-bit output stream
  - high-order bit == 0: data value
  - high-order bit == 1: repeat count of previous data value
- Valid bit set when 8-bit output is data or count

![Design Problem – Run-Length Encoder Diagram]

RLE Module

- Use parameter to define symbolic states

![RLE Module Diagram]

Start with Datapath

- We need to know what to control

![Start with Datapath Diagram]

FSM Controller

- FSM inputs
  - eq
- FSM outputs
  - clr, inc, valid, cnt,

![FSM Controller Diagram]

Reg For State Machines

- State machine has two parts
  - State register
  - Combinational Logic
  - Next state function
  - Output function
- Each in a different always block

![Reg For State Machines Diagram]

Verilog For State Machines

- rleFSM is one module
  - The datapath is specified as a schematic
  - rleFSM is part of schematic
  - Possible to describe datapath in the Verilog
  - Text description of inherently graphical design

```verilog
module rleFSM (clk, reset, eq, clr, inc, valid, cnttag);
  // input clk, reset;
  // current data value or previous data value
  // clear count value (1 means 2
  // clear output valid;
  // output value is valid
  // select the count for the output value
  // Use parameter to define symbolic states
  parameter START = 0, START2 = 1, COUNTING = 2, COUNTING = 3;
  reg [1:0] state; // current state
  assign state; // next state
endmodule
```

![Verilog For State Machines Diagram]
Verilog for Registers

- Triggering on "posedge CLK" generates a positive, edge-triggered register
- The only kind of register we will use

```verilog
reg [7:0] state; // current state
nextState; // next state

always @(posedge CLK) begin
  if (reset)
    state = START;
  else
    state = nextState;
end
```

Implementing an FSM

1. Perform state assignment
   - different assignments may give very different results
   - no really good heuristics
   - using an extra bit or two for state works well
   - FPGAs often use a 1-hot encoding
2. Convert state diagram to state table
   - equivalent representation
   - mechanical
3. State table gives truth table for next state and output functions
   - synthesize into logic circuit
   - e.g. 2-level logic implementation

RLE State Table

- reset is implicit - resets state register

<table>
<thead>
<tr>
<th>eq</th>
<th>state</th>
<th>next state</th>
<th>clr</th>
<th>inc</th>
<th>valid</th>
<th>comtag</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>10</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>10</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>11</td>
<td>x</td>
<td>x</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>10</td>
<td>x</td>
<td>x</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>x</td>
</tr>
</tbody>
</table>

Logic Synthesis

```
Q1 = Q1 + Q0
Q0 = Q0 + Q1
inc = Q1
valid = eq (Q1 + Q0) or (inc + Q1)
comtag = Q0
```
RLE State Table

- 1-hot encoding
- there are 8 output functions

<table>
<thead>
<tr>
<th>eq state</th>
<th>next state</th>
<th>cir inc valid cnttag</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0010</td>
<td>0 0100</td>
<td>x x 0 x</td>
</tr>
<tr>
<td>0 0100</td>
<td>0 1000</td>
<td>x x 1 0</td>
</tr>
<tr>
<td>1 0100</td>
<td>1 0000</td>
<td>1 x 1 0</td>
</tr>
<tr>
<td>0 1000</td>
<td>0 1000</td>
<td>x x 1 1</td>
</tr>
</tbody>
</table>

1-Hot Logic Functions

- Synthesize directly from table
- $Q_0$: reset
- $Q_1 = Q_0$
- $Q_2 = Q_1 \cdot \text{eq} (Q_2 \cdot Q_3)$
- $Q_3 = \text{eq} (Q_2 \cdot Q_3)$
- $\text{inc} = 1$
- $\text{cir} = Q_2$
- $\text{valid} = Q_2 \cdot \text{eq} Q_3$
- $\text{cnttag} = Q_3$
- Some cost as minimal encoding (6 gates)

Another Example: Ant Brain

- Sensors: L and R antennae, 1 if touching wall
- Actuators: F: forward step, TL/TR: turn left/right slightly
- Goal: find way out of maze
- Strategy: keep the wall on the right

Ant Behavior - Case Analysis

- A: Following wall, touching
  - Go forward, turning left slightly
- B: Following wall, not touching
  - Go forward, turning right slightly
- C: Break in wall
  - Go forward, turning right slightly
- D: Hit wall again
  - Back to state A
- E: Wall in front
  - Turn left until...
- F: ...we are here, same as state B
- G: Turn left until...
- LOST: Forward until we touch something

Designing an Ant Brain

- State Diagram
- Once we have state diagram, we just "turn the crank"

State Transition Truth Table

- Use symbolic states and outputs
- Take advantage of extra freedom

<table>
<thead>
<tr>
<th>state L</th>
<th>R</th>
<th>next state</th>
<th>outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0</td>
<td>L + R</td>
<td>F</td>
</tr>
<tr>
<td>0 1</td>
<td>0</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>A 0 0</td>
<td>B</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>A 0 1</td>
<td>A</td>
<td>TL F</td>
<td></td>
</tr>
<tr>
<td>A 1 0</td>
<td>B</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td>A 1 1</td>
<td>B</td>
<td>TL F</td>
<td></td>
</tr>
<tr>
<td>B/C 0 0</td>
<td>C</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>B/C 0 1</td>
<td>A</td>
<td>TR F</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
FSMs

Synthesis
- 5 states: at least 3 state variables required (X, Y, Z)
- State assignment (in this case, arbitrarily chosen)

<table>
<thead>
<tr>
<th>State</th>
<th>L</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>010</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>011</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>101</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>111</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

It now remains to synthesize these 6 functions

Don’t cares in FSM synthesis
- What happens to the “unused” states (101, 110, 111)?
- They were exploited as don’t cares to minimize the logic
  - If the states can’t happen, then we don’t care what the functions do
  - If states do happen, we may be in trouble

Don’t cares in FSMs (cont’d)
- Synthesize logic for next state functions derive input equations for flip-flops

<table>
<thead>
<tr>
<th>C'</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>X 1 1 0</td>
<td></td>
</tr>
<tr>
<td>X X 0 1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B'</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>X 1 0 0</td>
<td></td>
</tr>
<tr>
<td>X 1 0 1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A'</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>X 1 0 0</td>
<td></td>
</tr>
<tr>
<td>X 1 0 0</td>
<td></td>
</tr>
</tbody>
</table>

Self-starting FSMs
- Deriving state transition table from don’t care assignment

<table>
<thead>
<tr>
<th>Present State</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
</tr>
<tr>
<td>010</td>
<td>011</td>
</tr>
<tr>
<td>100</td>
<td>110</td>
</tr>
<tr>
<td>111</td>
<td>111</td>
</tr>
</tbody>
</table>

Don’t Care Example
- Implement simple count sequence: 000, 010, 011, 101, 110
- 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>000</td>
<td>001</td>
</tr>
<tr>
<td>010</td>
<td>011</td>
</tr>
<tr>
<td>100</td>
<td>110</td>
</tr>
<tr>
<td>111</td>
<td>111</td>
</tr>
</tbody>
</table>

note the don’t care conditions that arise from the unused state codes
Self-starting FSMs
- Start-up states
  - at power-up, FSM may be in an used or invalid state
  - design must guarantee that it (eventually) enters a valid state
- Self-starting solution
  - design FSM so that all the invalid states eventually
  - transition to a valid state may limit exploitation of don’t cares

Mealy vs. Moore Machines
- Moore: outputs depend on current state only
- Mealy: outputs may depend on current state and current inputs
- Our art brain is a Moore machine
  - output does not react immediately to input change
  - We could have specified a Mealy FSM
    - outputs have immediate reactions to inputs (do glitches matter?)
    - as inputs change, so does next_state, but doesn’t commit until clock tick

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

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

Comparison of Mealy and Moore machines
- Mealy machines tend to have fewer states
  - different outputs on arcs (\(\ell(n)\)) rather than states (n)
- Mealy machines react faster to inputs
  - react in same cycle – don’t need to wait for clock
  - delay to output depends on arrival of input
- Moore machines are generally safer to use
  - outputs change at clock edge (always one clock later)
  - in Mealy machines, input change can cause output change as soon as logic is done – a big problem when two machines are interconnected – asynchronous feedback

Isn’t Mealy Always Better?
- Example: serial adder - add bits arriving serially on two input wires
  - Adding 5 serial numbers on 5 separate lines
  - What is the clock period of this circuit?
Moore Implementation - Pipelined

- Example: adding 5 serial numbers

- What is the clock period of this circuit?