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

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

Example Finite State Machine Diagram

- Combination lock from first lecture

Can Any Sequential System be Represented with a State Diagram?

- Shift Register
  - Input value shown on transition arcs
  - Output values shown within state node
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, ...

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

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 encoding**

Verilog Upcounter

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

FSM Design Procedure

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

FSM Design Procedure: State Diagram to Encoded State Transition Table

- Tabular form of state diagram
- Like a truth-table

State encoding

- Decide on representation of state
- For counters it is simple: just its value

Implementation

- Flip-flop for each state bit
- Combinational logic based on encoding
Implementation (cont’d)

- Programmable Logic Building Block for Sequential Logic
  - Macro-cell: FF + logic
  - D-FF
  - Two-level logic capability like PAL (e.g., 8 product terms)

More Complex Counter Example

- Complex Counter
  - Repeats five states in sequence
  - Not a binary number representation
- Step 1: Derive the state transition diagram
  - Count sequence: 000, 010, 011, 110
- Step 2: Derive the state transition table from the state transition diagram

More Complex Counter Example (cont’d)

- Step 3: K-maps for Next State Functions

Self-Starting Counters (cont’d)

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

Self-Starting Counters

- Start-up States
  - At power-up, counter may be in an unused or invalid state
  - Designer must guarantee it (eventually) enters a valid state
- Self-starting Solution
  - Design counter so that invalid states eventually transition to a valid state
  - May limit exploitation of don’t cares

Another Example

- Shift Register
  - Input determines next state

Implementation on previous slide
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 only (Moore machine).
- **Inputs Outputs Next State**

State Machine Model (cont’d)

- **States:** $S_1, S_2, \ldots, S_n$
- **Inputs:** $I_1, I_2, \ldots, I_m$
- **Outputs:** $O_1, O_2, \ldots, O_n$
- **Transition function:** $F_S(S_i, I_j)$
- **Output function:** $F_O(S_i, I_j)$

First Midterm Exam—15 February 2007

- **Topics to be covered:**
  - Combinational logic design
  - From spec to truth table to K-map to Boolean Expression
    - Canonical forms of Boolean Expressions
    - Conversions of AND-OR logic to NAND or NOR logic
  - Two level logic implementations using gates, PLA, MUX, DEC, ROM, Xilinx CLB FPGA structures
    - Comparing implementation complexities/figures of merit
  - Basic Sequential logic design
  - Flip flop behavior, analysis, and timing diagrams
  - Using flip flops to design registers, shifters, counters
  - From spec to state diagram to Sequential Verilog
  - Amount of FSM implementation through end of today

Example: Ant Brain (Ward, MIT)

- **Sensors:** L and R antenna, 1 if in 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 Brain
Ant Behavior

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

Designing an Ant Brain

- State Diagram

Synthesizing the Ant Brain Circuit

- Encode States Using a Set of State Variables
  - Arbitrary choice – may affect cost, speed
- Use Transition Truth Table
  - Define next state function for each state variable
  - Define output function for each output
- Implement next state and output functions using
  - 2-level logic (ROM/PLA/PAL)
  - Multi-level logic
  - Next state and output functions can be optimized together

Transition Truth Table

- Using symbolic states and outputs

Synthesis

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

Synthesis of Next State and Output Functions
Circuit Implementation

- Outputs are a function of the current state only - Moore machine

![Diagram of a Moore machine circuit](CS 150 - Spring 2007 – Lec #6: Moore and Mealy Machines - 31)

Don't Cares in FSM Synthesis

- What happens to the "unused" states (101, 110, 111)?
- Exploited as don't cares to minimize the logic
  - If states can't happen, then don't care what the functions do
  - If states do happen, we may be in trouble

![Diagram of state minimization](CS 150 - Spring 2007 – Lec #6: Moore and Mealy Machines - 33)

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

![Diagram of state minimization](CS 150 - Spring 2007 – Lec #6: Moore and Mealy Machines - 34)

Ant Brain Revisited

- Any equivalent states?

![Diagram of state transitions](CS 150 - Spring 2007 – Lec #6: Moore and Mealy Machines - 35)

New Improved Brain

- Merge equivalent B and C states
- Behavior is exactly the same as the 5-state brain
- We now need only 2 state variables rather than 3

![Diagram of state transitions](CS 150 - Spring 2007 – Lec #6: Moore and Mealy Machines - 36)
New Brain Implementation

<table>
<thead>
<tr>
<th>state inputs</th>
<th>next state outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>X Y L R</td>
<td>X' Y' F TR TL</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>0 0 1 0 0</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>0 1 0 0 0</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>1 1 0 1 1</td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>0 1 0 0 1</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>1 1 1 0 0</td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>1 1 0 1 1</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>1 1 1 0 0</td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>1 1 1 0 1</td>
</tr>
</tbody>
</table>

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