Sequential Logic Implementation

- Sequential Circuits
  - Primitive sequential elements
  - Combinational logic
- Models for representing sequential circuits
  - Finite-state machines (Moor and Moehly)
  - Representation of memory (states)
  - Changes in state (transitions)
- Basic sequential circuits
  - Shift registers
  - Counters
- 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 values 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, 100, 101, 110, 111, ...
    - 3-bit down-counter: 111, 110, 111, 100, 011, 001, 000, 111, ...

![Diagram of a 3-bit up-counter and down-counter](image)

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 so that this is a performance penalty

![Diagram of a counter circuit](image)

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

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

![FSM state transition table](image)

Implementation

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

![Truth table for state transitions](image)

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)
Another Example

- Shift Register
  - Input determines next state

---

More Complex Counter Example

- Complex Counter
  - Repeats five states in sequence
  - Not a binary number representation

  1. Derive the state transition diagram
  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

<table>
<thead>
<tr>
<th>C4</th>
<th>C3</th>
<th>C2</th>
<th>C1</th>
<th>C0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<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>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</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>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

---

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

---

State Machine Model

- Values stored in registers represent the state of the circuit
- Combinational logic computes:
  - Next state
- Outputs
  - Function of current state and inputs
  - Function of current state and inputs (Mealy machine)
  - Function of current state only (Moore machine)
State Machine Model (cont'd)

- States: S1, S2, ..., Sn
- Inputs: I1, I2, ..., Im
- Outputs: O1, O2, ..., On
- Transition function: \( f(s, i) \)
- Output function: \( f(s, i, j) \)

<table>
<thead>
<tr>
<th>State</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example: Ant Brain (Ward, MIT)

- Sensors: LandR antenna, 1 if in touching wall
- Actuators: L = forward step, T/L/T - turn left/right slightly
- Goal: Find way out of maze
- Strategy: Keep the wall on the right

Ant Behavior

- A: Following wall, touching
  - Go forward, turning left slightly
- B: Following wall, not touching
  - Go forward, turning right slightly
- C: Break in wall, turning right slightly
- D: Hit wall again, turn back to state A
- E: Wall in front
  - Turn left until...
- F: ...are there, come as state B
- G: Turn left until...
- H: 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
  - Combinational logic (ROM/LUT/MUX)
  - Multi-level logic
- Next state and output functions can be optimized together

Transition Truth Table
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>
<th>next state inputs</th>
<th>outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 0</td>
<td>0</td>
<td>0</td>
<td>0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>000 1</td>
<td>0</td>
<td>1</td>
<td>0 0</td>
<td>1 0</td>
</tr>
<tr>
<td>......</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>....</td>
</tr>
<tr>
<td>100 0</td>
<td>1</td>
<td>1</td>
<td>0 0</td>
<td>1 0</td>
</tr>
<tr>
<td>100 1</td>
<td>0</td>
<td>1</td>
<td>1 0</td>
<td>1 1</td>
</tr>
</tbody>
</table>

It may remain to synthesize these 6 functions.

Synthesis of Next State and Output Functions

<table>
<thead>
<tr>
<th>state inputs</th>
<th>next state outputs</th>
<th>L</th>
<th>R</th>
<th>F</th>
<th>W</th>
<th>T</th>
<th>RL</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 0 0</td>
<td>000 1 0 0</td>
<td>1 0</td>
<td>0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000 1 0</td>
<td>001 0 1 0</td>
<td>1 0</td>
<td>0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010 0 1</td>
<td>011 0 1 0</td>
<td>1 0</td>
<td>1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100 0 1</td>
<td>101 0 1 0</td>
<td>1 0</td>
<td>1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010 1 1</td>
<td>011 1 1 1</td>
<td>1 0</td>
<td>1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100 1 1</td>
<td>101 1 1 1</td>
<td>1 0</td>
<td>1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100 0 0</td>
<td>101 0 1 0</td>
<td>1 0</td>
<td>1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100 1 0</td>
<td>011 1 1 1</td>
<td>1 0</td>
<td>1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100 0 1</td>
<td>101 1 1 1</td>
<td>1 0</td>
<td>1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100 1 1</td>
<td>011 1 1 1</td>
<td>1 0</td>
<td>1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

Circuit Implementation

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

<table>
<thead>
<tr>
<th>output logic</th>
<th>next state logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>R</td>
</tr>
</tbody>
</table>

State Minimization

- Fewer states may mean fewer state variables
- High-level synthesis may generate many redundant states
- Two states 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

Ant Brain Revisited

- Any equivalent states?
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

Mealy vs. Moore Machines

- Moore: outputs depend on current state only
- Mealy: outputs depend on current state and inputs
- Any 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
    - As inputs change, so does next state, doesn’t commit until clicking event

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 less states
  - Different outputs on arcs (“2”) rather than states (ﬁ)
- Moore Machines are safer to use
  - Outputs change at clock edge (always one cycle later)
  - In Mealy machines, input change can cause output change as soon as logic is done - a bug problem when two machines are interconnected - asynchronous feedback
- Mealy Machines 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 - more gate delays after
Mealy and Moore Examples
- Recognize \( A, B = 0,1 \)
  - Mealy or Moore?

Registered Mealy Machine (Really Moore)
- Synchronous (or registered) Mealy Machine
- Registered state AND outputs
- Avoid glitchy outputs
- Easy to implement in PLDs
- Moore Machine with no output decoding
- Outputs computed on transition to next state rather than after entering
- View outputs as expanded state vector

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
    - 1 nickel, dime
    - dime, nickel
    - two dimes
  - Draw state diagram:
  - Output transition table
  - 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)

- **Uniquely Encode States**

<table>
<thead>
<tr>
<th>Present State</th>
<th>Input</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>0</td>
<td>1 0 1 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>1</td>
<td>0 1 0 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>0</td>
<td>1 0 1 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>1</td>
<td>0 1 0 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>0</td>
<td>1 0 1 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>1</td>
<td>0 1 0 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>0</td>
<td>1 0 1 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>1</td>
<td>0 1 0 0</td>
<td>0 0 0 0</td>
</tr>
</tbody>
</table>

Example: Vending Machine (cont’d)

- **Mapping To Logic**

Example: Vending Machine (cont’d)

- **One-hot Encoding**

<table>
<thead>
<tr>
<th>Present State</th>
<th>Input</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>0</td>
<td>1 0 1 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>1</td>
<td>0 1 0 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>0</td>
<td>1 0 1 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>1</td>
<td>0 1 0 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>0</td>
<td>1 0 1 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>1</td>
<td>0 1 0 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>0</td>
<td>1 0 1 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>1</td>
<td>0 1 0 0</td>
<td>0 0 0 0</td>
</tr>
</tbody>
</table>

Example: Traffic Light Controller

- A busy highway is intersected by a little used farm road
- Detectors sense the presence of cars waiting on the farm road
  - With no cars on farm road, light remains green in highway direction
  - If a vehicle on farm road, highway light goes from green to yellow to red, allowing the farm road to become green
  - Traffic on green as long as farm road car is detected but never longer than a short interval
  - When farm motor on farm road detects a car, green light becomes red and traffic flows to green
  - Even if farm road vehicle is not detected, highway light remains green at least a short interval
- Assume the system has an internal timer that generates:
  - A short time pulse (ST)
  - A long time pulse (LT)
  - A set of short (ST) signal
  - A TS light to be used for timing yellow lights and TL for green lights

Example: Traffic Light Controller (cont’d)

- **Highway/Farm Road Intersection**

```
+---+    +---+
|    |    |    |
|    |    |    |
|    | car sensors |
+---+    +---+
    |    |
    |    |
    +---+    +---+  Highway
```
Example: Traffic Light Controller (cont')

Tabulation of Inputs and Outputs

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Present State</th>
<th>Next State</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>HX</td>
<td>HY</td>
<td>D Green Red</td>
</tr>
<tr>
<td>1</td>
<td>HY</td>
<td>HY</td>
<td>D Yellow Red</td>
</tr>
<tr>
<td>D</td>
<td>PS</td>
<td>FY</td>
<td>H Green Red</td>
</tr>
<tr>
<td>E</td>
<td>FY</td>
<td>FY</td>
<td>H Yellow Red</td>
</tr>
<tr>
<td>S1</td>
<td>HY</td>
<td>FY</td>
<td>L Green Red</td>
</tr>
<tr>
<td>S2</td>
<td>FY</td>
<td>FY</td>
<td>L Yellow Red</td>
</tr>
</tbody>
</table>

Tabulation of unique states - some light configurations imply others

- S1: Highway green (form road red)
- S2: Highway yellow (form road red)
- S3: Form road yellow (highway red)

Example: Traffic Light Controller (cont')

State Diagram

Logic for Different State Assignments

<table>
<thead>
<tr>
<th>State Assignment</th>
<th>Inputs</th>
<th>Present State</th>
<th>Next State</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>HX</td>
<td>HY</td>
<td>D Green Red</td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td>HY</td>
<td>HY</td>
<td>D Yellow Red</td>
<td></td>
</tr>
<tr>
<td>S3</td>
<td>PS</td>
<td>FY</td>
<td>H Green Red</td>
<td></td>
</tr>
<tr>
<td>S4</td>
<td>FY</td>
<td>FY</td>
<td>H Yellow Red</td>
<td></td>
</tr>
</tbody>
</table>

Vending Machine Example (PLD mapping)

Vending Machine (cont'd)
Vending Machine (Retimed PLD Mapping)

Finite State Machine Optimization

- State Minimization
  - Fewer states require fewer state bits
  - Fewer bits require fewer logic equations
- Encodings: State, Inputs, Outputs
  - State encoding with fewer bits has fewer equations to implement
  - However, each may be more complex
  - State encoding with more bits (e.g., one-hot) has simpler equations
  - Complexity directly related to complexity of state diagram
  - Input/output encoding may or may not be under designer control

Algorithmic Approach to State Minimization

- Goal: Identify and combine states that have equivalent behavior
- Equivalent States:
  - Some output
  - For all input combinations, states transition to same or equivalent states
- Algorithm Sketch
  1. Place all states in one set
  2. Serially partition set based on output behavior
  3. Successive partition resulting subsets based on next state transitions
  4. Repeat (3) until no further partitioning is required
  5. States left in the same set are equivalent
  6. Polynomial time procedure

State Minimization Example

- Sequence Detector for 010 or 110

<table>
<thead>
<tr>
<th>Input Sequence</th>
<th>Present State</th>
<th>Next State</th>
<th>X=0</th>
<th>X=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S1</td>
<td>S1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>S2</td>
<td>S3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>S1</td>
<td>S1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>S2</td>
<td>S3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>S3</td>
<td>S3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>S4</td>
<td>S5</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Minimized FSM

- State minimized sequence detector for 010 or 110

<table>
<thead>
<tr>
<th>Input Sequence</th>
<th>Present State</th>
<th>Next State</th>
<th>Present State</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S1</td>
<td>S1</td>
<td>X=0</td>
<td>X=1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>S2</td>
<td>S3</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>S1</td>
<td>S1</td>
<td>X=0</td>
<td>X=1</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>S2</td>
<td>S3</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>S3</td>
<td>S3</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>S4</td>
<td>S5</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
More Complex State Minimization

- Multiple input example

Minimized FSM

- Implication Chart Method
  - Cross out incompatible states based on outputs
  - Then cross out more candidates if indexed chart entries are already crossed out

Minimizing Incompletely Specified FSMs

- Equivalence of states is transitive when machine is fully specified
- But its not transitive when don't cares are present
  - e.g., state output
    - s0 - 0
    - s1 - 1
    - s2 - 1
- No polynomial-time algorithm exists for determining best grouping of states into equivalent sets that will yield the smallest number of final states

Minimizing States May Not Yield Best Circuit

- Example: edge detector - outputs 1 when last two input changes from 0 to 1

Another Implementation of Edge Detector

- "Ad hoc" solution - not minimal but cheap and fast

State Assignment

- Choose bit vectors to assign to each "symbolic" state
  - With n state bits from states there are 2^n / (2^n - m) [m = m = 2^n]
  - 2^n codes possible for 1st state, 2^n-1 for 2nd, 2^n-2 for 3rd, ...
  - Huge number even for small values of n and m
- Introduction of state machines of any size
- Heuristics are not necessary for practical solutions
- Optimize some metric for the combinational logic
  - Size (amount of logic and number of FFs)
  - Speed (depth of logic and fanout)
  - Dependence list (decomposition)
State Assignment Strategies

- Possible Strategies
  - Sequential - just number states as they appear in the state table
  - Random - pick random codes
  - One-hot - use as many state bits as there are states (bits → states)
  - Output - use outputs to help encode states
  - Heuristic - rules of thumb that seem to work in most cases
  - No guarantee of optimality - another intractable problem

One-hot State Assignment

- Simple
  - Easy to encode, debug
- Small Logic Functions
  - Each state function requires only predecessor state bits as input
- Good for Programmable Devices
  - Lists of flip-flops readily available
  - Simple functions with small support (signals its dependent upon)
- Impractical for Large Machines
  - Too many states require too many flip-flops
  - Decompose FSMs into smaller pieces that can be one-hot encoded
- Many slight variations to one-hot
  - One-hot + allo

Heuristics for State Assignment

- Adjacent codes to states that share a common next state
  - Group 1's in next state map
    - \( \begin{bmatrix} a & b & c \end{bmatrix} \)
    - \( \begin{bmatrix} a & b \end{bmatrix} \)
  - Adjacent codes to states that share a common ancestor state
    - Group 1's in next state map
    - \( \begin{bmatrix} a \end{bmatrix} \)
    - \( \begin{bmatrix} a \end{bmatrix} \)
  - Adjacent codes to states that have a common output behavior
    - Group 1's in output map

General Approach to Heuristic State Assignment

- All current methods are variants of this
  - 1) Determine which states "attract" each other (weighted pairs)
  - 2) Generate constraints on codes (which should be in some code)
  - 3) Place codes or Boolean cube as to maximize constraints satisfied (weighted sum)
- Different weights in state depending on whether we are optimizing for two-level or multi-level forms
- Can't consider all possible embeddings of state clusters in Boolean cube
- Heuristics for ordering embedding
- To prune search for best embedding
- Expand cube (more state bits) to satisfy more constraints

Output-Based Encoding

- Reuse outputs as state bits - use outputs to help distinguish states
- Why create new functions for state bits when output can serve as well
- Fits nicely with synchronous Mealy implementations

Current State Assignment Approaches

- For tight encodings using close to the minimum number of state bits
  - Best of 10 random seems to be adequate (averages as well as heuristics)
  - Heuristic approaches are not even close to optimality
  - Used in custom chip design
- One-hot encoding
  - Easy for small state machines
  - Generates small equations with easy to estimate complexity
    - Common in ASICs and other programmable logic
- Output-based encoding
  - Advanced methods
    - Most common approach taken by human designers
  - Yields very small circuits for most FSMs
Sequential Logic Implementation Summary

- Models for representing sequential circuits
  - Abstracting sequential elements
  - Finite state machines and their state diagrams
  - Inputs/outputs
  - Moore, Mealy, 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

- Implementation of sequential logic
  - State minimization
  - State assignment
  - Support in programmable logic devices