FSM Optimization

Thanks to Randy Katz (VC Research) for slides
State Reduction: Motivation

- Implement FSM with fewest possible states
  - Least number of flip flops
    - Boundaries are power of two number of states
  - Fewest states usually leads to more opportunities for don't cares
  - Reduce the combinational logic needed for implementation

- Odd Parity Checker: two alternative state diagrams
  - Identical output behavior on all input strings
  - FSMs are equivalent, but require different implementations

Material taken from:
Katz, Randy H., Contemporary Logic Design
Benjamin/Cummings Publishing, Co. Inc. © 1994
**State Reduction**

**Goal**

Identify and combine states that have equivalent behavior

*Equivalent States:* for all input combinations, states transition to the same or equivalent states

**Algorithmic Approach**

- Start with state transition table
- Identify states with same output behavior
- If such states transition to the same next state, they are equivalent
- Combine into a single new renamed state
- Repeat until no new states are combined
State Reduction

Row Matching Method

Example FSM Specification:

Single input X, output Z
Taking inputs grouped four at a time, output 1 if last four inputs were the string 1010 or 0110

Example I/O Behavior:

\[
X = 0010 \ 0110 \ 1100 \ 1010 \ 0011 \ \ldots
\]
\[
Z = 0000 \ 0001 \ 0000 \ 0001 \ 0000 \ \ldots
\]

Upper bound on FSM complexity:

Fifteen states \((1 + 2 + 4 + 8)\)

Thirty transitions \((2 + 4 + 8 + 16)\)

sufficient to recognize any binary string of length four!
State Reduction

Row Matching Method

State Diagram for Example FSM:
**State Reduction**

*Row Matching Method*

Initial State Transition Table:

<table>
<thead>
<tr>
<th>Input Sequence</th>
<th>Present State</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>S₀</td>
<td>S₁ S₂</td>
<td>X=0 X=1</td>
</tr>
<tr>
<td>0</td>
<td>S₁</td>
<td>S₃ S₄</td>
<td>0 0</td>
</tr>
<tr>
<td>1</td>
<td>S₂</td>
<td>S₅ S₆</td>
<td>0 0</td>
</tr>
<tr>
<td>00</td>
<td>S₃</td>
<td>S₇ S₈</td>
<td>0 0</td>
</tr>
<tr>
<td>01</td>
<td>S₄</td>
<td>S₉ S₁₀</td>
<td>0 0</td>
</tr>
<tr>
<td>10</td>
<td>S₅</td>
<td>S₁₁ S₁₂</td>
<td>0 0</td>
</tr>
<tr>
<td>11</td>
<td>S₆</td>
<td>S₁₃ S₁₄</td>
<td>0 0</td>
</tr>
<tr>
<td>000</td>
<td>S₇</td>
<td>S₀ S₀</td>
<td>0 0</td>
</tr>
<tr>
<td>001</td>
<td>S₈</td>
<td>S₀ S₀</td>
<td>0 0</td>
</tr>
<tr>
<td>010</td>
<td>S₉</td>
<td>S₀ S₀</td>
<td>0 0</td>
</tr>
<tr>
<td>011</td>
<td>S₁₀</td>
<td>S₀ S₀</td>
<td>1 0</td>
</tr>
<tr>
<td>100</td>
<td>S₁₁</td>
<td>S₀ S₀</td>
<td>0 0</td>
</tr>
<tr>
<td>101</td>
<td>S₁₂</td>
<td>S₀ S₀</td>
<td>1 0</td>
</tr>
<tr>
<td>110</td>
<td>S₁₃</td>
<td>S₀ S₀</td>
<td>0 0</td>
</tr>
<tr>
<td>111</td>
<td>S₁₄</td>
<td>S₀ S₀</td>
<td>0 0</td>
</tr>
</tbody>
</table>
### Row Matching Method

**Initial State Transition Table:**

<table>
<thead>
<tr>
<th>Input Sequence</th>
<th>Present State</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>$S_0$</td>
<td>$S_1$, $S_2$</td>
<td>0, 0</td>
</tr>
<tr>
<td>0</td>
<td>$S_1$</td>
<td>$S_3$, $S_4$</td>
<td>0, 0</td>
</tr>
<tr>
<td>1</td>
<td>$S_2$</td>
<td>$S_5$, $S_6$</td>
<td>0, 0</td>
</tr>
<tr>
<td>00</td>
<td>$S_3$</td>
<td>$S_7$, $S_8$</td>
<td>0, 0</td>
</tr>
<tr>
<td>01</td>
<td>$S_4$</td>
<td>$S_9$, $S_{10}$</td>
<td>0, 0</td>
</tr>
<tr>
<td>10</td>
<td>$S_5$</td>
<td>$S_{11}$, $S_{12}$</td>
<td>0, 0</td>
</tr>
<tr>
<td>11</td>
<td>$S_6$</td>
<td>$S_{13}$, $S_{14}$</td>
<td>0, 0</td>
</tr>
<tr>
<td>000</td>
<td>$S_7$</td>
<td>$S_0$, $S_0$</td>
<td>0, 0</td>
</tr>
<tr>
<td>001</td>
<td>$S_8$</td>
<td>$S_0$, $S_0$</td>
<td>0, 0</td>
</tr>
<tr>
<td>010</td>
<td>$S_9$</td>
<td>$S_0$, $S_0$</td>
<td>0, 0</td>
</tr>
<tr>
<td>011</td>
<td>$S_{10}$</td>
<td>$S_0$, $S_0$</td>
<td>1, 0</td>
</tr>
<tr>
<td>100</td>
<td>$S_{11}$</td>
<td>$S_0$, $S_0$</td>
<td>0, 0</td>
</tr>
<tr>
<td>101</td>
<td>$S_{12}$</td>
<td>$S_0$, $S_0$</td>
<td>1, 0</td>
</tr>
<tr>
<td>110</td>
<td>$S_{13}$</td>
<td>$S_0$, $S_0$</td>
<td>0, 0</td>
</tr>
<tr>
<td>111</td>
<td>$S_{14}$</td>
<td>$S_0$, $S_0$</td>
<td>0, 0</td>
</tr>
</tbody>
</table>
## State Reduction

### Row Matching Method

<table>
<thead>
<tr>
<th>Input Sequence</th>
<th>Present State</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>S₀</td>
<td>S₁, S₂</td>
<td>0, 0</td>
</tr>
<tr>
<td>0</td>
<td>S₁</td>
<td>S₃, S₄</td>
<td>0, 0</td>
</tr>
<tr>
<td>1</td>
<td>S₂</td>
<td>S₅, S₆</td>
<td>0, 0</td>
</tr>
<tr>
<td>00</td>
<td>S₃</td>
<td>S₇, S₈</td>
<td>0, 0</td>
</tr>
<tr>
<td>01</td>
<td>S₄</td>
<td>S₉, S₁₀</td>
<td>0, 0</td>
</tr>
<tr>
<td>10</td>
<td>S₅</td>
<td>S₁₁, S₁₀</td>
<td>0, 0</td>
</tr>
<tr>
<td>11</td>
<td>S₆</td>
<td>S₁₃, S₁₄</td>
<td>0, 0</td>
</tr>
<tr>
<td>000</td>
<td>S₇</td>
<td>S₀, S₀</td>
<td>0, 0</td>
</tr>
<tr>
<td>001</td>
<td>S₈</td>
<td>S₀, S₀</td>
<td>0, 0</td>
</tr>
<tr>
<td>010</td>
<td>S₉</td>
<td>S₀, S₀</td>
<td>0, 0</td>
</tr>
<tr>
<td>011 or 101</td>
<td>S₁₀</td>
<td>S₀, S₀</td>
<td>1, 0</td>
</tr>
<tr>
<td>100</td>
<td>S₁₁</td>
<td>S₀, S₀</td>
<td>0, 0</td>
</tr>
<tr>
<td>110</td>
<td>S₁₃</td>
<td>S₀, S₀</td>
<td>0, 0</td>
</tr>
<tr>
<td>111</td>
<td>S₁₄</td>
<td>S₀, S₀</td>
<td>0, 0</td>
</tr>
</tbody>
</table>
### State Reduction

#### Row Matching Method

<table>
<thead>
<tr>
<th>Input Sequence</th>
<th>Present State</th>
<th>Next State X=0</th>
<th>Next State X=1</th>
<th>Output X=0</th>
<th>Output X=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>S₀</td>
<td>S₁, S₂</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>S₁</td>
<td>S₃, S₄</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>S₂</td>
<td>S₅, S₆</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>S₃</td>
<td>S₇, S₈</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>S₄</td>
<td>S₉, S'₁₀</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>S₅</td>
<td>S₁₁, S'₁₀</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>S₆</td>
<td>S₁₃, S₁₄</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>000</td>
<td>S₇</td>
<td>S₀, S₀</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>S₈</td>
<td>S₀, S₀</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>010</td>
<td>S₉</td>
<td>S₀, S₀</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>011 or 101</td>
<td>S₁₀</td>
<td>S₀, S₀</td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>S₁₁</td>
<td>S₀, S₀</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>110</td>
<td>S₁₃</td>
<td>S₀, S₀</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>111</td>
<td>S₁₄</td>
<td>S₀, S₀</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
## State Reduction

### Row Matching Method

<table>
<thead>
<tr>
<th>Input Sequence</th>
<th>Present State</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$X=0$</td>
<td>$X=1$</td>
</tr>
<tr>
<td>Reset</td>
<td>$S_0$</td>
<td>$S_1$</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$S_2$</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>$S_1$</td>
<td>$S_3$</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>$S_2$</td>
<td>$S_4$</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>$S_3$</td>
<td>$S_7$</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>$S_4$</td>
<td>$S_7$</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>$S_5$</td>
<td>$S'_7$</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>$S_6$</td>
<td>$S'_7$</td>
<td>0</td>
</tr>
<tr>
<td>not (011 or 101)</td>
<td>$S'_7$</td>
<td>$S_0$</td>
<td>0</td>
</tr>
<tr>
<td>011 or 101</td>
<td>$S'_7$</td>
<td>$S'_0$</td>
<td>1</td>
</tr>
</tbody>
</table>
## State Reduction

### Row Matching Method

<table>
<thead>
<tr>
<th>Input Sequence</th>
<th>Present State</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$X=0$</td>
<td>$X=1$</td>
</tr>
<tr>
<td>Reset</td>
<td>$S_0$</td>
<td>$S_1$</td>
<td>$S_2$</td>
</tr>
<tr>
<td>0</td>
<td>$S_1$</td>
<td>$S_3$</td>
<td>$S_4$</td>
</tr>
<tr>
<td>1</td>
<td>$S_2$</td>
<td>$S_5$</td>
<td>$S_6$</td>
</tr>
<tr>
<td>00</td>
<td>$S_3$</td>
<td>$S_7'$</td>
<td>$S_7'$</td>
</tr>
<tr>
<td>01</td>
<td>$S_4$</td>
<td>$S_7'$</td>
<td>$S_{10}'$</td>
</tr>
<tr>
<td>10</td>
<td>$S_5$</td>
<td>$S_7'$</td>
<td>$S_{10}'$</td>
</tr>
<tr>
<td>11</td>
<td>$S_6$</td>
<td>$S_7'$</td>
<td>$S_7$</td>
</tr>
<tr>
<td>not (011 or 101)</td>
<td>$S_7'$</td>
<td>$S_0$</td>
<td>$S_0$</td>
</tr>
<tr>
<td>011 or 101</td>
<td>$S_{10}'$</td>
<td>$S_0$</td>
<td>$S_0$</td>
</tr>
</tbody>
</table>

The output column indicates the state transition and output for each input sequence.
# State Reduction

## Row Matching Method

<table>
<thead>
<tr>
<th>Input Sequence</th>
<th>Present State</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>S0</td>
<td>S1 S2</td>
<td>X=0 X=1</td>
</tr>
<tr>
<td>0</td>
<td>S1</td>
<td>S3' S4'</td>
<td>X=0 X=1</td>
</tr>
<tr>
<td>1</td>
<td>S2</td>
<td>S4' S3'</td>
<td>X=0 X=1</td>
</tr>
<tr>
<td>00 or 11</td>
<td>S3'</td>
<td>S7' S7'</td>
<td>X=0 X=1</td>
</tr>
<tr>
<td>01 or 10</td>
<td>S4'</td>
<td>S7' S10'</td>
<td>X=0 X=1</td>
</tr>
<tr>
<td>not (011 or 101)</td>
<td>S7'</td>
<td>S0 S0</td>
<td>X=0 X=1</td>
</tr>
<tr>
<td>011 or 101</td>
<td>S10'</td>
<td>S0 S0</td>
<td>X=0 X=1</td>
</tr>
</tbody>
</table>

### Final Reduced State Transition Table

### Corresponding State Diagram
State Reduction

- Row Matching Method
  - Useful heuristic and easy to apply
  - Does not always yield the most reduced state table!

Odd Parity Checker:
- No Way to combine state S0 and S2 based next state criterion!
State Reduction

Implication Chart Method

Enumerate all possible combinations of states taken two at a time

Next States Under all Input Combinations

Naive Data Structure:
X_{ij} will be the same as X_{ji}
Also, can eliminate the diagonal
State Reduction

Implication Chart Method

The detailed algorithm:

1. Construct implication chart, one square for each combination of states taken two at a time

2. Square labeled $S_i, S_j$, if outputs differ than square gets "X". Otherwise write down implied state pairs for all input combinations

3. Advance through chart top-to-bottom and left-to-right. If square $S_i, S_j$ contains next state pair $S_m, S_n$ and that pair labels a square already labeled "X", then $S_i, S_j$ is labeled "X".

4. Continue executing Step 3 until no new squares are marked with "X".

5. For each remaining unmarked square $S_i, S_j$, then $S_i$ and $S_j$ are equivalent.
State Reduction

Implication Chart Method

New example FSM:

Single input X, Single output Z

Output a 1 whenever the serial sequence 010 or 110 has been observed at the inputs

State transition table:

<table>
<thead>
<tr>
<th>Input Sequence</th>
<th>Present State</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>X=0</td>
<td>X=1</td>
</tr>
<tr>
<td>Reset</td>
<td>S₀</td>
<td>S₁</td>
<td>S₂</td>
</tr>
<tr>
<td>0</td>
<td>S₁</td>
<td>S₃</td>
<td>S₄</td>
</tr>
<tr>
<td>1</td>
<td>S₂</td>
<td>S₅</td>
<td>S₆</td>
</tr>
<tr>
<td>00</td>
<td>S₃</td>
<td>S₀</td>
<td>S₀</td>
</tr>
<tr>
<td>01</td>
<td>S₄</td>
<td>S₀</td>
<td>S₀</td>
</tr>
<tr>
<td>10</td>
<td>S₅</td>
<td>S₀</td>
<td>S₀</td>
</tr>
<tr>
<td>11</td>
<td>S₆</td>
<td>S₀</td>
<td>S₀</td>
</tr>
</tbody>
</table>
**State Reduction**

**Implication Chart Method**

Starting Implication Chart

- S1-S3
- S2-S4
- S2-S6
- S4-S6
- S1-S0
- S3-S0
- S5-S0
- S2-S0
- S4-S0
- S6-S0
- S1-S0
- S3-S0
- S5-S0
- S0-S0
- S2-S0
- S4-S0
- S6-S0
- S0-S0

S2 and S4 have different I/O behavior.

This implies that S1 and S0 cannot be combined.
State Reduction

Implication Chart Method

Results of First Marking Pass
Second Pass Adds No New Information
S3 and S5 are equivalent
S4 and S6 are equivalent
This implies that S1 and S2 are too!

Reduced State Transition Table
State Reduction

Multiple Input State Diagram Example

State Diagram

Symbolic State Diagram

<table>
<thead>
<tr>
<th>Present State</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>S₀</td>
<td>S₀ S₁ S₂ S₃</td>
<td>1</td>
</tr>
<tr>
<td>S₁</td>
<td>S₀ S₃ S₁ S₅</td>
<td>0</td>
</tr>
<tr>
<td>S₂</td>
<td>S₁ S₃ S₂ S₄</td>
<td>1</td>
</tr>
<tr>
<td>S₃</td>
<td>S₁ S₀ S₄ S₅</td>
<td>0</td>
</tr>
<tr>
<td>S₄</td>
<td>S₀ S₁ S₂ S₅</td>
<td>1</td>
</tr>
<tr>
<td>S₅</td>
<td>S₁ S₄ S₀ S₅</td>
<td>0</td>
</tr>
</tbody>
</table>
State Reduction

Multiple Input Example

Implication Chart

Minimized State Table

<table>
<thead>
<tr>
<th>Present State</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_0'</td>
<td>S_0' S_1</td>
<td>1</td>
</tr>
<tr>
<td>S_1</td>
<td>S_0' S_3'</td>
<td>0</td>
</tr>
<tr>
<td>S_2</td>
<td>S_1 S_3'</td>
<td>1</td>
</tr>
<tr>
<td>S_3'</td>
<td>S_1 S_0'</td>
<td>0</td>
</tr>
</tbody>
</table>

Implication Chart
FSM State Assignment
State Assignment

- When FSM implemented with gate logic, number of gates will depend on mapping between symbolic state names and binary encodings.
- 5 states = 8 choices for first state, 7 for second, 6 for third, 5 for forth, 4 for last = 6720 different encodings.
- Ex: combination lock FSM
  - 5 states
  - my assignment START=000, OK1=001, OK2=011, BAD1=100, BAD2=101.
  - only one of 120 possibilities.
## State Assignment

### Effect of Adjacencies on Next State Map

<table>
<thead>
<tr>
<th>Current State</th>
<th>Next State X = 0</th>
<th>X = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(S₀) 000</td>
<td>001</td>
<td>101</td>
</tr>
<tr>
<td>(S₁) 001</td>
<td>011</td>
<td>111</td>
</tr>
<tr>
<td>(S₂) 101</td>
<td>111</td>
<td>011</td>
</tr>
<tr>
<td>(S₃) 011</td>
<td>010</td>
<td>010</td>
</tr>
<tr>
<td>(S₄) 111</td>
<td>001</td>
<td>101</td>
</tr>
<tr>
<td>(S₅) 010</td>
<td>000</td>
<td>000</td>
</tr>
<tr>
<td>(S'₁₀) 110</td>
<td>000</td>
<td>000</td>
</tr>
</tbody>
</table>

**P₂**

<table>
<thead>
<tr>
<th>Q₂</th>
<th>Q₁</th>
<th>Q₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>10</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>00</td>
</tr>
</tbody>
</table>

**P₁**

<table>
<thead>
<tr>
<th>Q₂</th>
<th>Q₁</th>
<th>Q₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>X</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>X</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

**P₀**

<table>
<thead>
<tr>
<th>Q₂</th>
<th>Q₁</th>
<th>Q₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>X</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>X</td>
</tr>
</tbody>
</table>

**First encoding exhibits a better clustering of 1's in the next state map**
State Assignment

Pencil & Paper Heuristic Methods

State Maps: similar in concept to K-maps
If state X transitions to state Y, then assign "close" assignments to X and Y
State Assignment

Paper and Pencil Methods

Minimum Bit Distance Criterion

<table>
<thead>
<tr>
<th>Transition</th>
<th>First Assignment Bit Changes</th>
<th>Second Assignment Bit Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0 to S1:</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>S0 to S2:</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>S1 to S3:</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>S2 to S3:</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>S3 to S4:</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>S4 to S1:</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

13                               7
State Assignment

Paper & Pencil Methods

Alternative heuristics based on input and output behavior as well as transitions:

Adjacent assignments to:
- states that share a common next state (group 1's in next state map)
- states that share a common ancestor state (group 1's in next state map)
- states that have common output behavior (group 1's in output map)
State Assignment

Pencil and Paper Methods
Example: 3-bit Sequence Detector

- **Highest Priority:** (S3', S4')
- **Medium Priority:** (S3', S4')
- **Lowest Priority:**
  - 0/0: (S0, S1', S3')
  - 1/0: (S0, S1', S3', S4')
State Assignment

Paper and Pencil Methods

Reset State = 00
Highest Priority Adjacency

Not much difference in these two assignments
State Assignment

Paper & Pencil Methods

Another Example: 4 bit String Recognizer

Highest Priority: \((S3', S4'), (S7', S10')\)

Medium Priority:
\((S1, S2), 2x(S3', S4'), (S7', S10')\)

Lowest Priority:
\(0/0: (S0, S1, S2, S3', S4', S7')\)
\(1/0: (S0, S1, S2, S3', S4', S7')\)
State Assignment

Paper & Pencil Methods

00 = Reset = S0

(S1, S2), (S3', S4'), (S7', S10') placed adjacently
In general:

# of possible FSM states = $2^{\text{# of Flip-flops}}$

Example:

state1 = 01, state2 = 11, state3 = 10, state4 = 00

However, often more than $\log_2(\text{# of states})$ FFs are used, to simplify logic at the cost of more FFs.

Extreme example is one-hot state encoding.
State Encoding

- One-hot encoding of states.
- One FF per state.

Why one-hot encoding?
- Simple design procedure.
  - Circuit matches state transition diagram (example next page).
  - Often can lead to simpler and faster “next state” and output logic.

Why not do this?
- Can be costly in terms of Flip-flops for FSMs with large number of states.

FPGAs are “Flip-flop rich”, therefore one-hot state machine encoding is often a good approach.
One-hot encoded FSM

- Even Parity Checker Circuit:

Circuit generated through direct inspection of the STD.

- In General:
  - FFs must be initialized for correct operation (only one 1)
One-hot encoded combination lock
FSMs in Verilog
General FSM Design Process with Verilog Implementation

Design Steps:
1. Specify **circuit function** (English)
2. Draw **state transition diagram**
3. Write down **symbolic state transition table**
4. Assign encodings (bit patterns) to symbolic states
5. Code as Verilog behavioral description

✓ Use parameters to represent encoded states.
✓ Use register instances for present-state plus CL logic for next-state and outputs.
✓ Use case for CL block. Within each case section (state) assign all outputs and next state value based on inputs. Note: For Moore style machine make outputs dependent only on state not dependent on inputs.
Finite State Machine in Verilog

State Transition Diagram

Holds a symbol to keep track of which bubble the FSM is in.

CL functions to determine output and next state based on input and current state.

\[ \text{out} = f(\text{in}, \text{current state}) \]

\[ \text{next state} = f(\text{in}, \text{current state}) \]
module FSM1(clk, rst, in, out);
input clk, rst;
input in;
output out;

// Defined state encoding:
localparam IDLE = 2'b00;
localparam S0 = 2'b01;
localparam S1 = 2'b10;

reg out;
reg [1:0] next_state;
wire [1:0] present_state;

// state register
REGISTER_R #( .N(2), .INIT(IDLE) ) state
(.q(present_state), .d(next_state), .rst(rst));

An always block should be used for combination logic part of FSM. Next state and output generation.
// always block for combinational logic portion
always @(present_state or in)
case (present_state)
// For each state define output and next
   IDLE : begin
      out = 1'b0;
      if (in == 1'b1) next_state = S0;
      else next_state = IDLE;
   end
   S0 : begin
      out = 1'b0;
      if (in == 1'b1) next_state = S1;
      else next_state = IDLE;
   end
   S1 : begin
      out = 1'b1;
      if (in == 1'b1) next_state = S1;
      else next_state = IDLE;
      default: begin
         next_state = IDLE;
         out = 1'b0;
      end
   endcase
endmodule

Each state becomes a case clause.

For each state define:
Output value(s)
State transition

Use “default” to cover unassigned state. Usually unconditionally transition to reset state.

Mealy or Moore?
**Edge Detector Example**

**Mealy Machine**

```verilog
REGISTER_R #( .INIT(ZERO) )
state (.q(ps), .d(ns), .rst(rst));

always @(ps in)
  case (ps)
    ZERO: if (in) begin
      out = 1'b1;
      ns = ONE;
    end
    else begin
      out = 1'b0;
      ns = ZERO;
    end
  ONE: if (in) begin
    out = 1'b0;
    ns = ONE;
  end
  default: begin
    out = 1'b0;
    ns = default;
  end
```

**Moore Machine**

```verilog
REGISTER_R #( .N(2), .INIT(ZERO) )
state (.q(ps), .d(ns), .rst(rst));

always @(ps in)
  case (ps)
    ZERO: begin
      out = 1'b0;
      if (in) ns = CHANGE;
      else ns = ZERO;
    end
  CHANGE: begin
    out = 1'b1;
    if (in) ns = ONE;
    else ns = ZERO;
  end
  ONE: begin
    out = 1'b0;
    if (in) ns = ONE;
    else ns = ZERO;
  end
  default: begin
    out = 1'b0;
    if (in) ns = ONE;
    else ns = ZERO;
  end
```
always @(present_state or in)
  case (present_state)
    IDLE : begin
      out = 1'b0;
      if (in == 1'b1) next_state = S0;
      else next_state = IDLE;
    end
    S0  : begin
      out = 1'b0;
      if (in == 1'b1) next_state = S1;
      else next_state = IDLE;
    end
    S1  : begin
      out = 1'b1;
      if (in == 1'b1) next_state = S1;
      else next_state = IDLE;
    end
    default: begin
      next_state = IDLE;
      out = 1'b0;
    end
  endcase
endmodule

The sequential semantics of the blocking assignment allows variables to be multiply assigned within a single always block.
always @*

begin
  next_state = IDLE;
  out = 1'b0;
  case (state)
    IDLE : if (in == 1'b1) next_state = S0;
    S0  : if (in == 1'b1) next_state = S1;
    S1  : begin
      out = 1'b1;
      if (in == 1'b1) next_state = S1;
    end
    default: ;
  endcase
end
Endmodule

* for sensitivity list

Normal values: used unless specified below.

Within case only need to specify exceptions to the normal values.

Note: The use of “blocking assignments” allow signal values to be “rewritten”, simplifying the specification.
Some final Verilog warnings
Combinational logic always blocks

Make sure all signals assigned in a combinational always block are explicitly assigned values every time that the always block executes. Otherwise latches will be generated to hold the last value for the signals not assigned values.

Sel case value 2’d2 omitted.

Out is not updated when select line has 2’d2.

Latch is added by tool to hold the last value of out under this condition.

Similar problem with if-else statements.
To avoid synthesizing a latch in this case, add the missing select line:

\[ 2\text{'d2}: \text{out} = \text{c}; \]

Or, in general, use the “default” case:

\[ \text{default: out = foo;} \]

If you don’t care about the assignment in a case (for instance you know that it will never come up) then you can assign the value “x” to the variable. Example:

\[ \text{default: out = 1'bx;} \]

The x is treated as a “don’t care” for synthesis and will simplify the logic.

Be careful when assigning x (don’t care). If this case were to come up, then the synthesized circuit and simulation may differ.