Summary

- Sequential logic minimization
  - State minimization
    - Completely specified FSMs
      - Identify and merge equivalent states
      - Efficient algorithm ($O(n \log n)$)
    - Incompletely specified FSMs
      - Identify minimum set of compatibles that is closed and complete
      - Problem is NP-hard [Pfleeger 1973]
  - State encoding
  - Combinational logic synthesis
State Encoding (a.k.a. State Assignment)

- Assign binary representation to “symbolic” states.
  - Defines the next state and output functions

Symbolic State Table

<table>
<thead>
<tr>
<th>PI</th>
<th>PS</th>
<th>NS</th>
<th>PO</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>S0</td>
<td>S0</td>
<td>01</td>
</tr>
<tr>
<td>100</td>
<td>S0</td>
<td>S1</td>
<td>01</td>
</tr>
<tr>
<td>010</td>
<td>S0</td>
<td>S3</td>
<td>01</td>
</tr>
<tr>
<td>--1</td>
<td>S0</td>
<td>S0</td>
<td>10</td>
</tr>
<tr>
<td>100</td>
<td>S1</td>
<td>S1</td>
<td>01</td>
</tr>
<tr>
<td>0-0</td>
<td>S1</td>
<td>S0</td>
<td>01</td>
</tr>
<tr>
<td>110</td>
<td>S1</td>
<td>S2</td>
<td>01</td>
</tr>
<tr>
<td>--1</td>
<td>S1</td>
<td>S0</td>
<td>10</td>
</tr>
<tr>
<td>110</td>
<td>S2</td>
<td>S2</td>
<td>01</td>
</tr>
<tr>
<td>100</td>
<td>S2</td>
<td>S1</td>
<td>01</td>
</tr>
<tr>
<td>---</td>
<td>S3</td>
<td>S3</td>
<td>--</td>
</tr>
</tbody>
</table>
State Encoding: Example

- State encoding has a strong impact on the combinational logic complexity (and hence, area, timing, and power)

```
<table>
<thead>
<tr>
<th>PS</th>
<th>NS x=0</th>
<th>NS x=1</th>
<th>PO x=0</th>
<th>PO x=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>01</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>PS x=0</th>
<th>x=1</th>
<th>NS x=0</th>
<th>x=1</th>
<th>PO x=0</th>
<th>x=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>S0</td>
<td>S1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td>S2</td>
<td>S1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td>S0</td>
<td>S1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
```

- Optimized Logic Expressions:
Complexity of State Encoding

• How many possible ways to encode an FSM that has \( s \) states using \( n \) bits?

\[
\begin{array}{c|c|c}
\text{PS} & \text{NS} & \text{PO} \\
00 & 00 & 0 \\
01 & 10 & 0 \\
10 & 00 & 1 \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{PS} & \text{NS} & \text{PO} \\
00 & 00 & 10 \\
10 & 01 & 10 \\
01 & 00 & 10 \\
\end{array}
\]

\[
2^n \cdot \frac{P_s}{n!}
\]

• What if permutations of state bits are considered equivalent?

\[
\begin{array}{c|c|c|c|c|c|c}
\text{NS} & \text{PO} & \text{NS} & \text{PO} \\
00 & 00 & 00 & 00 \\
01 & 01 & 01 & 01 \\
10 & 00 & 01 & 01 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c}
\text{NS} & \text{PO} & \text{NS} & \text{PO} \\
00 & 00 & 10 & 00 \\
10 & 01 & 10 & 00 \\
01 & 00 & 10 & 01 \\
\end{array}
\]

\[
2^n \cdot \frac{P_s}{n!}
\]
State Encoding to Minimize Combinational Logic Complexity

- **Key idea**: Perform encoding so as to create opportunities for logic minimization in the next state and output functions
  - Techniques differ depending on whether target implementation of next-state & output logic is two-level or multi-level

<table>
<thead>
<tr>
<th>PI</th>
<th>PS</th>
<th>NS</th>
<th>PO</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0-0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>--1</td>
<td>1</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>110</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>---</td>
<td>3</td>
<td>3</td>
<td>--</td>
</tr>
</tbody>
</table>
Guidelines for State Encoding

- States that have the same next state for the same input value should be given adjacent assignments

<table>
<thead>
<tr>
<th>PS</th>
<th>NS</th>
<th>PO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x=0</td>
<td>x=1</td>
</tr>
<tr>
<td></td>
<td>x=0</td>
<td>x=1</td>
</tr>
</tbody>
</table>

...  

s_1  

s_2  

...  

s_3  

...  

Encodings:

- s_1: 000
- s_2: 001

Transition condition (s_3):

\[ (s_3) = x'a'b'c' + x'a'b'c = x'a'b' (c'+c) = x'a'b' \]

Benefit: Potential for combining cubes in the next-state logic

\[ NS(i) = \sum_{\text{all states } s_j \text{ with bit } i = 1} Transition\ Condition(s_j) \]

- Same applies for states that have the same output for the same input value
Guidelines for State Encoding

- States that have the same next state (for any input value) should be given adjacent assignments.

<table>
<thead>
<tr>
<th>PS</th>
<th>NS</th>
<th>PO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x=0</td>
<td>x=1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>s₁</td>
<td>s₃</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>s₂</td>
<td>...</td>
<td>s₃</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Transition condition ($s₃$)

- $x'a'b'c' + xab'c$
- $a'b' (x'c' + xc)$

Benefits: Potential for common factors in the next-state logic.

What if we choose a different encoding?

- $s₁: 000$
- $s₂: 001$

Transition condition ($s₃$)

- $x'a'b'c' + xabc$
- $a'(x'b'c' +xbc)$

- $s₁: 000$
- $s₂: 111$

Transition condition ($s₃$)

- $x'a'b'c' + xa'bc$
- $a'(x'b'c' + xbc)$
Guidelines for State Encoding

- Next states that result from the same previous state should be given adjacent assignments.

<table>
<thead>
<tr>
<th>PS</th>
<th>NS (x=0)</th>
<th>PO (x=1)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s1</td>
<td>s2</td>
<td>s3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Transition condition (s2) = x'a'b'c' + …
Transition condition (s3) = x'a'b'c' + …

Encoding

s1: 000
s2: 001
s3: 011

Benefit: Potential for combining cubes or common factors in the next-state logic.

Transition condition (s2) = x_1'x_2'a'b'c' + …
Transition condition (s3) = x_1x_2a'b'c' + …

c+ = x_1x_2a'b'c' + … + a'b'c'(x_1'x_2' + x_1x_2)
State Encoding Algorithm

• **General approach**: Construct a complete graph with nodes representing states, and weighted edges representing “affinity”
  - Affinity\( (s_i, s_j) \) should reflect the potential benefit of assigning adjacent codes to states \( s_i \) and \( s_j \)
  - Label the vertices of the graph based on the edge weights

State Encoding Algorithm: Computing Edge Weights (Fanout-Oriented)

- **Fanout-oriented heuristic**: Present states that result in similar outputs and produce similar sets of next states are given high affinity
  - Intuition: Maximize the size of the most commonly occurring cube factors in the next-state and output logic

**Next state set**: Matrix that captures how often a (PS, NS) pair occurs

**Output set**: How often an output bit is asserted in each PS

\[
\begin{bmatrix}
S_1 & S_1 & S_2 & 0 & 1 \\
S_2 & S_1 & S_3 & 0 & 0 \\
S_3 & S_3 & S_1 & 0 & 1 \\
\end{bmatrix}
\]

\[
NS\_SET = \begin{pmatrix}
S_1^n & S_2^n & S_3^n \\
S_1^p & 1 & 1 & 0 \\
S_2^p & 1 & 0 & 1 \\
S_3^p & 1 & 0 & 1 \\
\end{pmatrix}
\]

\[
OUT\_SET = \begin{pmatrix}
z \\
S_1^p & 1 \\
S_2^p & 0 \\
S_3^p & 1 \\
\end{pmatrix}
\]
State Encoding Algorithms: Computing Edge Weights (Fanout-Oriented)

- Formula to compute edge weights

\[ W_{i,j} = \frac{N_b}{2} \cdot NS\_SET(i) \cdot NS\_SET(j)^T + OUT\_SET(i) \cdot OUT\_SET(j)^T \]

- \( N_b \): # of encoding bits
- \( NS\_SET(i) \): \( i \)th row of \( NS\_SET \) matrix
- \( OUT\_SET(i) \): \( i \)th row of \( OUT\_SET \) matrix

\[
NS\_SET = \begin{pmatrix}
S_1^n & S_2^n & S_3^n \\
S_1^p & 1 & 1 & 0 \\
S_2^p & 1 & 0 & 1 \\
S_3^p & 1 & 0 & 1
\end{pmatrix}
\]

\[
OUT\_SET = \begin{pmatrix}
z \\
S_1^p & 1 \\
S_2^p & 0 \\
S_3^p & 1
\end{pmatrix}
\]

\[
W_{1,2} = 1 \\
W_{1,3} = 2 \\
W_{2,3} = 2 \\
W_{2,3} = 1 \cdot [1 \ 0 \ 1][1 \ 0 \ 1]^T + [0][1]^T = 2
\]
State Encoding Algorithm: Computing Edge Weights (Fanin-Oriented)

- **Fanin-oriented heuristic**: Next states that are produced by similar inputs and similar sets of present states are given high affinity.

**Present state set**: Matrix that captures how often a (NS,PS) pair occurs.

**Input set**: How often a next state is caused for each input value.

\[
\begin{bmatrix}
S_1^n & 1 & 1 & 1 \\
S_1^p & x & x & 1 \\
S_2^n & 1 & 0 & 0 \\
S_3^n & 0 & 1 & 1 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
S_1^n & 1 & 2 \\
S_2^n & 1 & 0 \\
S_3^n & 1 & 1 \\
\end{bmatrix}
\]
State Encoding Algorithm: Computing Edge Weights (Fanin-Oriented)

- Formula to compute edge weights

\[ W_{i,j} = N_b \cdot PS_{\_SET}(i) \cdot PS_{\_SET}(j)^T + IN_{\_SET}(i) \cdot IN_{\_SET}(j)^T \]

- \( N_b \): # of encoding bits
- \( NS_{\_SET}(i) \): \( i \)th row of \( NS_{\_SET} \) matrix
- \( OUT_{\_SET}(i) \): \( i \)th row of \( OUT_{\_SET} \) matrix

\[
PS_{\_SET} = \begin{pmatrix}
S_1^n & S_2^n & S_3^n \\
S_1^p & 1 & 1 & 1 \\
S_2^n & 1 & 0 & 0 \\
S_3^n & 0 & 1 & 1 \\
\end{pmatrix}
\]

\[
IN_{\_SET} = \begin{pmatrix}
x & x' \\nS_1^n & 1 & 2 \\
S_2^n & 1 & 0 \\
S_3^n & 1 & 1 \\
\end{pmatrix}
\]

\[ W_{1,2} = 3 \]
\[ W_{1,3} = 7 \]
\[ W_{2,3} = 1 \]

\[ W_{1,3} = 2 \cdot [1 \ 1 \ 1][0 \ 1 \ 1]^T + [1 \ 2][1 \ 1]^T = 7 \]
State Encoding Algorithm

- **Algorithm for computing state encoding**
  1. Select the state for which sum of weights of $N_b$ heaviest incident edges is maximum
  2. Arbitrarily assign a code to it and assign adjacent codes to $N_b$ adjacent states
     - If some adjacent states have already been assigned codes, consider them when assigning a code to the selected state
  3. Remove the state and edges selected in step 1 from the graph
  4. Go to 1 and repeat, until graph is empty

- **How well does this work in practice?**
  - 30-40% lower literal count in the combinational logic (after multi-level optimization) compared to random state encoding

Summary: FSM synthesis

- State minimization
  - Completely specified FSMs: equivalent states
  - Incompletely specified: compatible states

- State encoding
  - Create opportunities for two-level and multi-level minimization algorithms to optimize the next state and output logic

- FSM-based synthesis is usually used only for control logic
Optimizing Structural Representations of Sequential Networks
Limitations of FSM synthesis

• FSM representation is too large for most circuits
  – Only parts of the design (e.g., control logic) with small state spaces can be represented as an FSM
  – Data-paths have HUGE state spaces

• Two key advances have extended the scale of FSMs that can be handled
  • Implicit representations (BDDs)
  • Network of interacting FSMs

• Even with these advances, FSM synthesis is not applicable to large circuits (> 1000s of FFs)
Structural Approaches to Sequential Circuit Optimization

- Optimize combinational logic using sequential Don’t Cares
- Retiming
- Retiming & Re-synthesis
Retiming

• Recall De Morgan’s law?
  – Moving “bubbles” across gates

• It turns out you can do the same thing with flip-flops!
  – Does not change I/O behavior

Retiming: Why?

• Re-position the flip-flops in the circuit to more “optimal” points
  – Increase the clock frequency
  – Reduce the number of registers
  – ...
Retiming: Example

Longest combinational path = 3

Gate delays

Longest combinational path = 2