Today

- Encoding
  - Input
  - Output
- State Encoding
  - “exact” two-level

Input Encoding

- Pick codes for input cases to simplify logic
- E.g. Instruction Decoding
  - ADD, SUB, MUL, OR
- Have freedom in code assigned
- Pick code to minimize logic
  - E.g. number of product terms

Output Encoding

- Opposite problem
- Pick codes for output symbols
- E.g. allocation selection
  - Prefer N, Prefer S, Prefer E, Prefer W, No Preference
- Again, freedom in coding
- Use to maximize sharing
  - Common product terms, CSE

Finite-State Machine

- Logical behavior depends on state
- In response to inputs, may change state

State Encoding

- State encoding is a logical entity
- No a priori reason any particular state has any particular encoding
- Use freedom to simply logic
Finite State Machine

0 0/1 0/0 0 0/1 0/0
0 S1 S1 1 1 S1 S2 0
1 S2 S2 0 0 S2 S3 0
1 S3 S3 1 0 S3 S3 1

Example: Encoding Difference

<table>
<thead>
<tr>
<th>S1</th>
<th>S2</th>
<th>S3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1</td>
<td>0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>1 0</td>
<td>1 0</td>
<td>0 1</td>
</tr>
<tr>
<td>1 1</td>
<td>1 0</td>
<td>1 0</td>
</tr>
<tr>
<td>0 0</td>
<td>0 1</td>
<td>0 0</td>
</tr>
</tbody>
</table>

Similar outputs, code so S1+S2 is simple cube

S1=00
S2=11
S3=10

S1+S2 = -1

Problem:

- **Real**: pick state encodings (si’s) so as to minimize the implementation area
  - two-level
  - multi-level
- Simplified variants
  - minimize product terms
  - achieving minimum product terms, minimize state size
  - minimize literals

Two-Level Optimization

1. **Idea**: do symbolic minimization of two-level form
   - This represents effects of sharing
2. Generate encoding constraints from this
3. **Cover**
   - Properties code must have to maximize sharing
4. **Select Codes**
Kinds of Sharing

Input sharing:
encode inputs so cover set to reduce product terms

Output sharing:
share input cubes to produce individual output bits

<table>
<thead>
<tr>
<th>Input 1</th>
<th>Input 2</th>
<th>Input 3</th>
<th>Input 4</th>
<th>Output 1</th>
<th>Output 2</th>
<th>Output 3</th>
<th>Output 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>01</td>
<td>11</td>
<td>00</td>
<td>1101</td>
<td>1100</td>
<td>0000</td>
<td>0001</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>11</td>
<td>01</td>
<td>1100</td>
<td>1111</td>
<td>0000</td>
<td>0001</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>11</td>
<td>01</td>
<td>0000</td>
<td>0000</td>
<td>1111</td>
<td>1111</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>01</td>
<td>10</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>1100</td>
</tr>
</tbody>
</table>

Out1 = 11
Out2 = 01
Out3 = 10
Out4 = 00

Input Encoding

Two-Level Input Oriented

- Minimize product rows
  - by exploiting common-cube
  - next-state expressions
  - Does not account for possible sharing of terms to cover outputs

Outline Two-Level Input

- Represent states as one-hot codes
- Minimize using two-level optimization
  - Include: combine compatible next states
    - 1 S1 S2 0
    - 1 S2 S2 0 \rightarrow 1 (S1, S2) S2 0
  - Get disjunct on states deriving next state
  - Assuming no sharing due to outputs
    - gives minimum number of product terms
  - Cover to achieve
    - Try to do so with minimum number of state bits

Multiple Valued Input Set

- Treat input states as a multi-valued (not just 0, 1) input variable
- Effectively encode in one-hot form
  - One-hot: each state gets a bit, only one on
- Use to merge together input state sets

<table>
<thead>
<tr>
<th>State</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

One-hot Minimum

- One-hot gives minimum number of product terms
  - i.e. Can always maximally combine input sets into single product term
One-hot example

| 10 inp1 01 | One-hot: 10 100 01 |
| 01 inp1 10 | inp1=100 01 100 01 |
| 1- inp2 01 | inp2=010 10 010 10 |
| 01 inp2 10 | inp3=001 01 001 01 |
| 11 inp3 01 | 11 001 01 |
| 01 inp3 10 | 01 001 01 |

Key: can define a cube to cover any subset of states

State Combining

- Follows from standard 2-level optimization with don’t-care minimization
- Effectively groups together common predecessor states as shown
- (can define to combine directly)

Example

| 0 S | s6 00 | 0 1000000 0000010 00 |
| 0 s2 | s5 00 | 0 0100000 0000100 00 |
| 0 s3 | s5 00 | 0 0010000 0000100 00 |
| 0 s4 | s6 00 | 0 0000100 0000100 00 |
| 0 s5 | S 10 | 0 1000000 0000010 00 |
| 0 s6 | S 10 | 0 0010000 0000010 00 |
| 0 s7 | s5 00 | 0 0000010 0000100 00 |
| 1 S | s4 01 | 1 0000000 0000100 00 |
| 1 s2 | s3 10 | 1 0010000 0000100 00 |
| 1 s3 | s7 10 | 1 0000000 0000100 00 |
| 1 s4 | s6 10 | 1 0000000 0000100 00 |
| 1 s5 | s2 00 | 1 1000000 0000010 00 |
| 1 s6 | s2 00 | 1 0100000 0000010 00 |
| 1 s7 | s6 00 | 1 0010000 0000010 00 |

Two-Level Input

- One-hot identifies multivalue minimum number of product terms
- May be less product terms if get sharing (don’t cares) in generating the next state expressions
  - (was not part of optimization)
- Encoding places each disjunct on a unique cube face
  - Can distinguish with a single cube
- Can use fewer bits than one-hot
  - this part typically heuristic
  - Remember one-hot already minimized prod terms

Encoding Example

| 0 S | s6 00 | 0 0110001 0000100 00 |
| 0 s2 | s5 00 | 0 1001000 0000010 00 |
| 0 s3 | s5 00 | 0 0100001 0000010 00 |
| 0 s4 | s6 00 | 0 0010001 0000010 00 |
| 0 s5 | S 10 | 0 0000100 0000010 00 |
| 0 s6 | S 10 | 0 0000010 0000010 00 |
| 0 s7 | s5 00 | 0 0000010 0000010 00 |
| 1 S | s4 01 | 1 0000000 0000010 00 |
| 1 s2 | s3 10 | 1 0000000 0000010 00 |
| 1 s3 | s7 10 | 1 0000000 0000010 00 |
| 1 s4 | s6 10 | 1 0000000 0000010 00 |
| 1 s5 | s2 00 | 1 0000000 0000010 00 |
| 1 s6 | s2 00 | 1 0000000 0000010 00 |
| 1 s7 | s6 00 | 1 0000000 0000010 00 |

s1+s7=0-0
s2+s3+s7=1--
No 111 code

Encoding Example

| 0 S | s6 00 | 0 0110001 0000100 00 |
| 0 s2 | s5 00 | 0 1001000 0000010 00 |
| 0 s3 | s5 00 | 0 0100001 0000010 00 |
| 0 s4 | s6 00 | 0 0010001 0000010 00 |
| 0 s5 | S 10 | 0 0000100 0000010 00 |
| 0 s6 | S 10 | 0 0000010 0000010 00 |
| 0 s7 | s5 00 | 0 0000010 0000010 00 |
| 1 S | s4 01 | 1 0000000 0000010 00 |
| 1 s2 | s3 10 | 1 0000000 0000010 00 |
| 1 s3 | s7 10 | 1 0000000 0000010 00 |
| 1 s4 | s6 10 | 1 0000000 0000010 00 |
| 1 s5 | s2 00 | 1 0000000 0000010 00 |
| 1 s6 | s2 00 | 1 0000000 0000010 00 |
| 1 s7 | s6 00 | 1 0000000 0000010 00 |

s1+s4=0-0
s2+s3+s7=1--
(no 111 code)
Input and Output

Skip?

General Problem

- Track both input and output encoding constraints

General Two-Level Strategy

1. Generate "Generalized" Prime Implicants
2. Extract/identify encoding constraints
3. Cover with minimum number of GPIs that makes encodeable
4. Encode symbolic values

Output Symbolic Sets

- Maintain output state, PIs as a set
- Represent inputs one-hot as before

<table>
<thead>
<tr>
<th>S1</th>
<th>S1</th>
<th>S1</th>
<th>0 100</th>
<th>1 100</th>
<th>1 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>S1</td>
<td>S1</td>
<td>(S1) (o1)</td>
<td>(S2) (o2)</td>
<td>(S3) (o3)</td>
</tr>
</tbody>
</table>

Generate GPIs

- Same basic idea as PI generation
  - Quine-McKlusky
- ...but different

Merging

- Cubes merge if  
  - distance one in input
    - 000 100
    - 001 100 ➔ 00- 100
  - inputs same, differ in multi-valued input (state)
    - 000 100
    - 000 010 ➔ 000 110
Merging

- When merge
  - binary valued output contain outputs asserted in both (and)
    - 000 100 (foo) (o1,o2)
    - 001 100 (bar) (o1,o3) \(\rightarrow\) 00- 100 ? (o1)
  - next state tag is union of states in merged cubes
    - 000 100 (foo) (o1,o2)
    - 001 100 (bar) (o1,o3) \(\rightarrow\) 00- 100 (foo,bar) (o1)

Merged Outputs

- Merged outputs
  - Set of things asserted by this input
  - States would like to turn on together
    - 000 100 (foo) (o1,o2)
    - 001 100 (bar) (o1,o3) \(\rightarrow\) 00- 100 (foo,bar) (o1)

Cancellation

- K+1 cube cancels k-cube only if
  - multivalued input is identical
  - AND next state and output identical
    - 000 100 (foo) (o1)
    - 001 100 (foo) (o1)
  - Also cancel if multivalued input contains all inputs
    - 000 111 (foo) (o1)
  - Discard cube with next state containing all symbolic states and null output
    - 111 100 (foo,bar,baz…) () \(\rightarrow\) does nothing

Example (work on board)

<table>
<thead>
<tr>
<th>States</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 100  (S1) (o1)</td>
<td>1 100  (S2) ()</td>
</tr>
<tr>
<td>1 010  (S2) ()</td>
<td>0 010  (S3) ()</td>
</tr>
<tr>
<td>0 001  (S3) (o1)</td>
<td>1 001  (S3) (o1)</td>
</tr>
</tbody>
</table>

Cancellation

- K+1 cube cancels k-cube only if
  - multivalued input is identical
  - AND next state and output identical
    - 000 100 (foo) (o1) 00- 100 (foo) (o1)
    - 001 100 (foo) (o1)
  - Also cancel if multivalued input contains all inputs
    - 000 111 (foo) (o1)
  - Discard cube with next state containing all symbolic states and null output
    - 111 100 (foo,bar,baz…) () \(\rightarrow\) does nothing

Example

<table>
<thead>
<tr>
<th>States</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 100  (S1) (o1)</td>
<td>1 100  (S2) ()</td>
</tr>
<tr>
<td>1 010  (S2) ()</td>
<td>0 010  (S3) ()</td>
</tr>
<tr>
<td>0 001  (S3) (o1)</td>
<td>1 001  (S3) (o1)</td>
</tr>
<tr>
<td>0 111  (S1,S3) ()</td>
<td>0 111  (S1,S3) ()</td>
</tr>
<tr>
<td>1 100  (S2) ()</td>
<td>0 010  (S2,S3) ()</td>
</tr>
<tr>
<td>0 011  (S2,S3) ()</td>
<td>1 001  (S3) (o1)</td>
</tr>
<tr>
<td>1 001  (S3) (o1)</td>
<td>0 011  (S3) (o1)</td>
</tr>
<tr>
<td>1 100  (S2) ()</td>
<td>0 010  (S2,S3) ()</td>
</tr>
<tr>
<td>0 011  (S2,S3) ()</td>
<td>1 001  (S3) (o1)</td>
</tr>
<tr>
<td>0 011  (S2,S3) ()</td>
<td>1 001  (S3) (o1)</td>
</tr>
</tbody>
</table>
Covering

• Cover with branch-and-bound similar to two-level
  – row dominance only if
    • tags of two GPs are identical
    • OR tag of first is subset of second
  • Once cover, check encodeability
    – [talk about next]
  • If fail, branch-and-bound again on additional GPs to add to satisfy encodeability

Encoding Constraints

• Minterm to symbolic state v
  – should assert v

• For all minterms m
  – all GPs [(∩ all symbolic tags) e(tag state)] = e(v)

Example

\[ \bigcup_{\text{all GPs}} [\bigcap \text{all symbolic tags}] e(\text{tag state}) = e(v) \]

Consider 1101 (out1) covered by
110- (out1,out2)
1100 out2
1111 out3
x 0000 out4
x 0001 out4

\[ 0 \begin{array}{c|c|c|c}
    \text{out1} & \text{out2} & \text{out3} & \text{out4} \\
    \hline
    110 & 110 & 111 & 0 \\
    1100 & 11 & 11 & 0 \\
    1100 & 11 & 1 & 0 \\
    1100 & 0 & 0 & 0 \\
    0001 & 0 & 0 & 0 \\
  \end{array} \]

Or-plane gives me OR of these two

Want output to be e(out1)

1101 (out1) ∪ e(out2) ∪ e(out1) ∩ e(out3) = e(out1)

To Satisfy

• Dominance and disjunctive relationships from encoding constraints
  – e(out1) ∩ e(out2) ∪ e(out1) ∩ e(out3) = e(out1)
  – one of:
    • e(out2) > e(out1) [i.e. e(out1) ∩ e(out2) = e(out1)]
    • e(out3) > e(out1) [i.e. e(out1) ∩ e(out3) = e(out1)]
    • e(out2) ∩ e(out3) > e(out1)

Encodeability Graph

\[ 1100 \begin{array}{c|c|c|c|c}
  \text{out1} & \text{out2} & \text{out3} & \text{out4} & \text{One of:} \\
  \hline
  1101 & e(out1) & e(out2) & e(out1) & e(out1) \\
  1100 & e(out1) & e(out2) & e(out1) & e(out1) \\
  1111 & e(out1) & e(out2) & e(out1) & e(out1) \\
  0000 & e(out4) = e(out4) & e(out3) & e(out3) & e(out3) \\
  0001 & e(out4) = e(out4) & e(out3) & e(out3) & e(out3) \\
  \end{array} \]
Encoding Constraints

- No directed cycles (proper dominance)
- Siblings in disjunctive have no directed paths between
  - (one cannot dominate other)
- No two disjunctive equality can have exactly the same siblings for different parents
- Parent of disjunctive should not dominate all sibling arcs

Determining Encoding

- Can turn into boolean satisfiability problem for a target code length
- All selected encoding constraints become boolean expressions
- Also uniqueness constraints

What we’ve done

- Define another problem
  - Constrained coding
- This identifies the necessary coding constraints
  - Solve optimally with SAT solver
  - Or attack heuristically

Summary

- Encoding can have a big effect on area
- Freedom in encoding allows us to maximize opportunities for sharing
- Can do minimization around unencoded to understand structure in problem outside of encoding
- Can adapt two-level covering to include and generate constraints
- Multilevel limited by our understanding of structure we can find in expressions
  - heuristics try to maximize expected structure

Admin

- Assign 3 due Wednesday
- Wednesday Reading on Web
Today's Big Ideas

• Exploit freedom
• Bounding solutions
• Dominators
• Formulation and Reduction
• Technique:
  – branch and bound
  – SAT
  – Understanding structure of problem
• Creating structure in the problem