State Assignment

The problem:

- Assign a unique code to each state to produce a logic level description.
- Given $|S|$ states, needed at least $\lceil \log |S| \rceil$ state bits (minimum width encoding), at most $|S|$ state bits (one-hot encoding).
- Need to estimate impact of encoding on the size and delay of the optimized implementation.
- Most known techniques are directed towards reducing the size. Difficult to estimate delay before optimization and thus to see the impact of encoding on it.
- There are $\binom{2^n}{|S|} |S|!$ possible state assignments for 2^n codes ($n \geq \lceil \log |S| \rceil$), since there are $\binom{2^n}{|S|}$ ways to select $|S|$ distinct state codes and $|S|!$ ways to permute them.

State Assignment

Techniques are in two categories:

- Heuristic techniques: try to capture some aspect of the process of optimizing the resulting logic, e.g. common cube extraction. Usually, they are two-steps processes:
  - Construct a weighted graph of the states. Weights express the gain in keeping 'close' the codes of the states.
  - Assign codes that minimize a proximity function (graph embedding step).
- Exact techniques: model precisely the process of optimizing the resulting logic as an *encoding problem*. Usually, they are three-steps processes:
  - Perform an appropriate multi-valued logic minimization (optimization step).
  - Extract a set of encoding constraints, that set conditions on the codes.
  - Assign codes that satisfy the encoding constraints.

State Assignment as an Encoding Problem

State assignment is a difficult problem in a family of encoding problems: transform 'optimally' a cover of multi-valued (also called symbolic) logic functions into an equivalent cover of two-valued logic functions.

Reason of transformation: available circuits realize two-valued logic.

Encoding problems are hard because an optimal transformation is sought. Various applications dictate various definitions of optimality.

Encoding problems are classified as input, output and input-output encoding problems, according to whether the symbolic variables appear as input, output or both.

State assignment is a case of input-output encoding where the the state variable appears both as input and output.

Encoding Problems

ENCODING PROBLEMS
INPUT ENCODING

two-level	**multi-level**
Well-understood theory	Basic theory developed
Efficient algorithms	Algorithms not yet mature
Many applications	Few applications

OUTPUT ENCODING

two-level	**multi-level**
Well-understood theory	No theory
No efficient algorithm	Heuristic algorithms
Few applications	Few applications

INPUT-OUTPUT ENCODING

two-level	**multi-level**
Well-understood theory	No theory
No efficient algorithm	Heuristic algorithms
Few applications	Few applications

Input Encoding for Two-Level Implementations

Reference: [De Micheli,Brayton,Sangiovanni 1985]

To solve the input encoding problem for minimum product-term count:

- Represent the function as a multi-valued function.
- Apply multi-valued minimization to it.
- Extract from the minimized multi-valued cover input encoding constraints.
- Obtain codes (of minimum length) that satisfy the input encoding constraints.

Example of Input Encoding

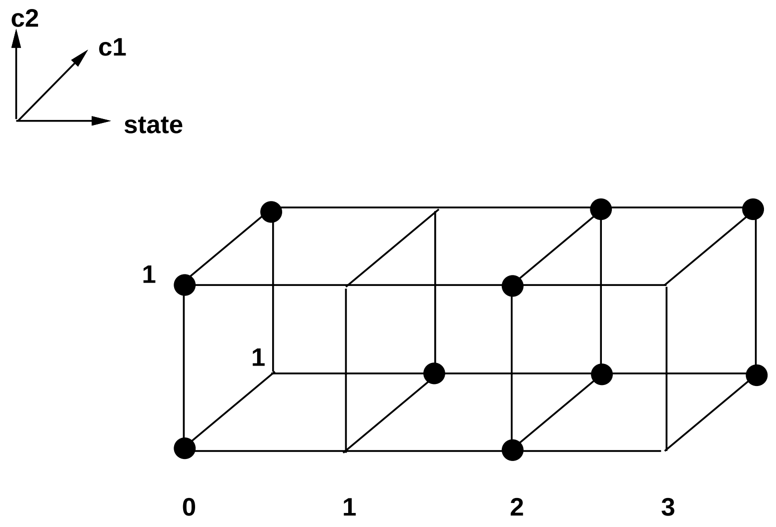
Example: Single output of an FSM

3 inputs: state and two conditions c_1 and c_2
4 states: s_0, s_1, s_2, s_3
1 output: y

y is 1 when under the following conditions:

$state = s_0$ and $c_2 = 1$
$((state = s_0)$ or $(state = s_2))$ and $c_1 = 0$
$state = s_1$ and $c_2 = 0$ and $c_1 = 1$
$((state = s_3)$ or $(state = s_2))$ and $c_1 = 1$

Pictorially:



Values along the state axis are not ordered.

Example of Input Encoding

Function Representation

- Symbolic variable represented by a multiple-valued (MV) variable X restricted to $P = \{0, 1, \ldots, n-1\}$
- Output is a binary valued function f of a single MV variable X and $m-1$ binary variables

$$f : P \times B^{m-1} \mapsto B$$

- Let $S \subseteq P$, then X^S (a literal of X) is defined as:

$$X^S = \begin{cases} 1 & \text{if } X \in S \\ 0 & \text{otherwise} \end{cases}$$

$$y = X^{\{0\}}\, c_2 \;+\; X^{\{0,2\}}\, \overline{c_1} \;+\; X^{\{1\}}\, c_1\, \overline{c_2} \;+\; X^{\{2,3\}}\, c_1$$

The cover of the function can also be written in one-hot encoded form:

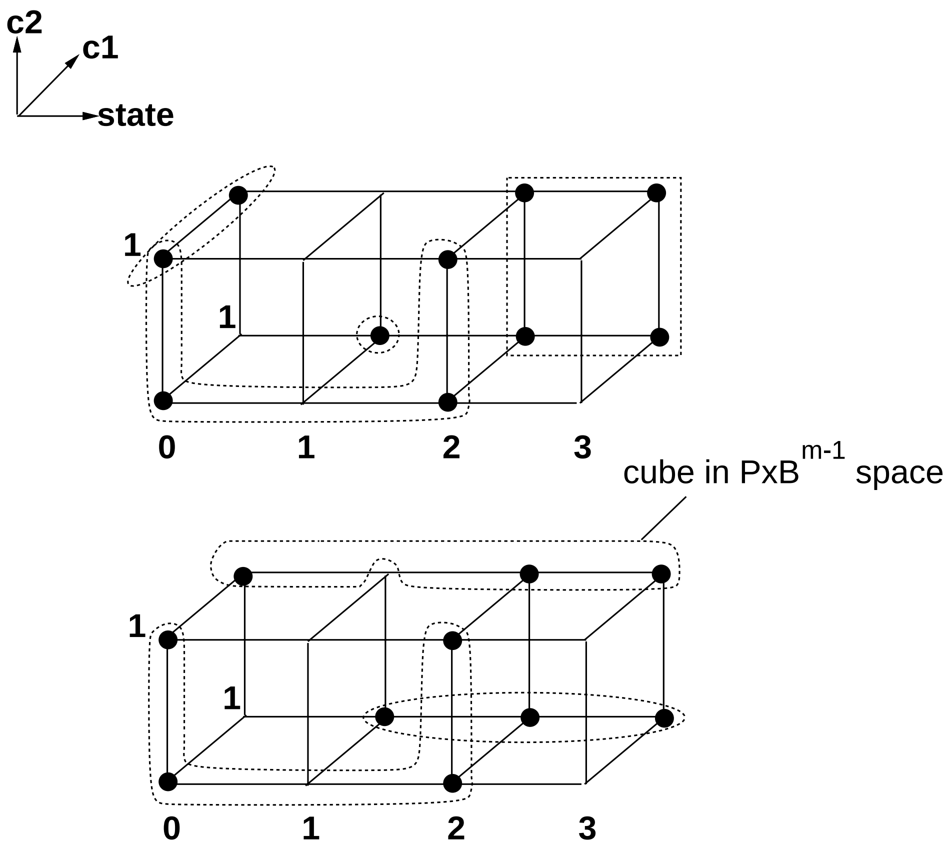
```
-1 1000 1
0- 1010 1
10 0100 1
1- 0011 1
```

Example of Input Encoding

Two-Level Multi-Valued Minimization

$$y = X^{\{0\}}\ c_2\ +\ X^{\{0,2\}}\ \overline{c_1}\ +\ X^{\{1\}}\ c_1\ \overline{c_2}\ +\ X^{\{2,3\}}\ c_1$$

Minimize using classical two-level minimization.



Minimized representation:

$$y = X^{\{0,2,3\}}\ c_1\ c_2\ +\ X^{\{0,2\}}\ \overline{c_1}\ +\ X^{\{1,2,3\}}\ c_1\ \overline{c_2}$$

Example of Input Encoding

Extraction of Input (or Face) Constraints

$$y = X^{\{0,2,3\}}\ c_1\ c_2\ +\ X^{\{0,2\}}\ \overline{c_1}\ +\ X^{\{1,2,3\}}\ c_1\ \overline{c_2}$$

The face constraints are:

$$\begin{gathered}(s_0, s_2, s_3),\\ (s_0, s_2),\\ (s_1, s_2, s_3).\end{gathered}$$

- two-level minimization results in the fewest product terms for any possible encoding
- would like to select an encoding that results in the same number of product terms
- each MV literal should be embedded in a face (subspace or cube) in the encoded space
- unused codes may be used as don't cares

Example of Input Encoding

Satisfaction of face constraints

- lower bound can always be achieved (one-hot code will do it), just need to do it with the fewest bits
- for a fixed code width need to find embedding that will result in the fewest cubes

A satisfying solution is:

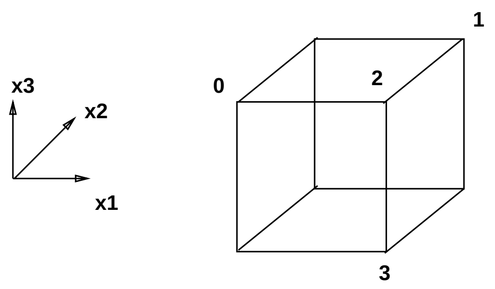
$$s_0 = 001,\ s_1 = 111,\ s_2 = 101,\ s_3 = 100.$$

The face constraints are assigned to the following faces:

$$\begin{array}{lcl} (s_0, s_2, s_3) & \Longleftrightarrow & 202 \\ (s_0, s_2) & \Longleftrightarrow & 201 \\ (s_1, s_2, s_3) & \Longleftrightarrow & 122 \end{array}$$

Encoded representation:

$$y = \overline{x_2}\ c_1\ c_2\ +\ x_3\ \overline{x_2}\ \overline{c_1} +\ x_1\ c_1\ \overline{c_2}$$



Procedure for Input Encoding

A summary of the steps is:

- Given an input constraint corresponding to a multi-valued literal (group of symbolic values), the face constraint is satisfied by an encoding if the super-cube of the codes of the symbolic values in the literal does not intersect the codes of symbolic values not in the literal.

- Satisfying all the face constraints of a multi-valued cover, guarantees that the encoded and minimized binary-valued cover will have a number of product-terms no greater than the multi-valued cover.

- Once an encoding satisfying all face constraints has been found, a binary-valued encoded cover can be constructed directly from the multi-valued cover, by replacing each multi-valued literal by the super-cube corresponding to that literal.

Satisfaction of Input Constraints

Any set of face constraints can be satisfied by one-hot encoding the symbols. But code-length too long ...

To find an encoding of minimum code-length that satisfies a given set of constraints is NP-hard (Saldanha,Villa,Brayton,Sangiovanni 1991).

Many approaches proposed to the problem. Previous reference describes the best algorithm currently known. It is based on the concept of encoding-dichotomies (Tracey 1965, Ciesielski 1989).

Applications of Input Encoding

Some of them are:

- PLA decomposition with multi-input decoders (Sasao 1984, K.C.Chen and Muroga 1988, Yang and Ciesielski 1989)
- Boolean decomposition in multi-level logic optimization (Devadas, Wang, Newton, Sangiovanni 1989)
- Communication-based logic partitioning (Beardslee 1992, Murgai 1993)
- Approximation to state assignment (De Micheli, Brayton, Sangiovanni 1985, Villa and Sangiovanni 1989)

Input Encoding: PLA Decomposition

Consider the PLA:

$$\begin{aligned}
y_1 &= x_1x_3x_4'x_6 + x_2x_5x_6'x_7 + x_1x_4x_7' + x_1'x_3'x_4x_7' \\
&\quad +x_3'x_5'x_6'x_7' \\
y_2 &= x_2'x_4'x_6 + x_3x_4'x_6 + x_2'x_5x_6'x_7 + x_1'x_3'x_4x_7' \\
&\quad +x_1'x_3'x_5'x_6'x_7' + x_2'x_4'x_5'x_7'
\end{aligned}$$

Decompose it in a driving PLA fed by inputs $\{X_4, X_5, X_6, X_7\}$ and a driven PLA fed by the outputs of the driving PLA and the remaining inputs $\{X_1, X_2, X_3\}$ so as to minimize the area.

Input Encoding: PLA Decomposition

Projecting on the selected inputs there are 5 product terms: $x'_4x_6, x_5x'_6x_7, x_4x'_7, x'_5x'_6x'_7, x'_4x'_5x'_7$.

Product terms $x_4x'_7$ and $x'_5x'_6x'_7$ are not disjoint, neither are $x'_5x'_6x'_7$ and $x'_4x'_5x'_7$.

To re-encode make disjoint all product-terms involving selected inputs:

$$\begin{aligned} y_1 &= x_1x_3x'_4x_6 + x_2x_5x'_6x_7 + x_1x_4x'_7 + x'_1x'_3x_4x'_7 \\ &\quad + x'_3x'_4x'_5x'_6x'_7 \\ y_2 &= x'_2x'_4x_6 + x_3x'_4x_6 + x'_2x_5x'_6x_7 + x'_1x'_3x_4x'_7 \\ &\quad + x'_1x'_3x'_4x'_5x'_6x'_7 + x'_2x'_4x'_5x'_6x'_7 \end{aligned}$$

Projecting on the selected inputs there are 4 disjoint product terms: $x'_4x_6, x_5x'_6x_7, x_4x'_7, x'_4x'_5x'_6x'_7$.

Input Encoding: PLA Decomposition

View $x'_4x_6, x_5x'_6x_7, x_4x'_7, x'_4x'_5x'_6x'_7$ as 4 values, s_1, s_2, s_3, s_4, of an MV variable S:

$$\begin{aligned} y_1 &= x_1x_3S^{\{1\}} + x_2S^{\{2\}} + x_1S^{\{3\}} + x'_1x'_3S^{\{3\}} + x'_3S^{\{4\}} \\ y_2 &= x'_2S^{\{1\}} + x_3S^{\{1\}} + x'_2S^{\{2\}} + x'_1x'_3S^{\{3\}} + x'_1x'_3S^{\{4\}} \\ &\quad +x'_2S^{\{4\}} \end{aligned}$$

Perform MV two-level minimization:

$$\begin{aligned} y_1 &= x_1x_3S^{\{1,3\}} + x_2S^{\{2\}} + x'_3S^{\{3,4\}} \\ y_2 &= x'_2S^{\{1,2,4\}} + x_3S^{\{1\}} + x'_1x'_3S^{\{3,4\}} \end{aligned}$$

Face constraints are: $(s_1, s_3), (s_3, s_4), (s_1, s_2, s_4)$.

Codes of minimum length are: $enc(s_1) = 001, enc(s_2) = 011, enc(s_3) = 100, enc(s_4) = 111$

Input Encoding: PLA Decomposition

The driven PLA becomes:

$$\begin{aligned} y_1 &= x_1x_3x_9' + x_2x_8'x_9x_{10} + x_3'x_8 \\ y_2 &= x_2'x_{10} + x_3x_8'x_9'x_{10} + x_1'x_3'x_8 \end{aligned}$$

The driving PLA implements the function:

$$f : \{X_4, X_5, X_6, X_7\} \rightarrow \{X_8, X_9, X_{10}\}.$$

$$\begin{aligned} f(x_4'x_6) &= enc(s_1) \equiv 001, \\ f(x_5x_6'x_7) &= enc(s_2) \equiv 011, \\ f(x_4x_7') &= enc(s_3) \equiv 100, \\ f(x_4'x_5'x_6'x_7') &= enc(s_4) \equiv 111. \end{aligned}$$

represented in SOP as:

$$\begin{aligned} x_8 &= x_4 + x_7' + x_4'x_5'x_6'x_7' \\ x_9 &= x_5 + x_6'x_7 + x_4'x_5'x_6'x_7' \\ x_{10} &= x_4'x_6 + x_5x_6'x_7 + x_4'x_5'x_6'x_7' \end{aligned}$$

Area before decomposition: 160. Area after decomposition: 128.

Output Encoding for Two-Level Implementations

The problem: Find binary codes for symbolic outputs in a logic function so as to minimize a two-level implementation of the function.

Terminology:

- Assume that we have a symbolic cover S with a symbolic output assuming n values. The different values are denoted $v_0, \ldots, v_{n-1}$.
- The encoding of a symbolic value v_i is denoted $enc(v_i)$.
- The on-set of v_i is denoted ON_i. Each ON_i is a set of D_i minterms $\{m_{i1}, \ldots, m_{iD_i}\}$.
- Each minterm m_{ij} has a tag as to what symbolic value's on-set it belongs to. A minterm can only belong to a single symbolic value's on-set. Minterms are also called 0-cubes.

Facts on Output Encoding

Consider a function f with symbolic outputs and two different encoded realizations of f:

```
0001 out1     0001 001     0001 10000
00-0 out2     00-0 010     00-0 01000
0011 out2     0011 010     0011 01000
0100 out3     0100 011     0100 00100
1000 out3     1000 011     1000 00100
1011 out4     1011 100     1011 00010
1111 out5     1111 101     1111 00001
```

An encoded cover is a multiple-output logic function. Two-level logic minimization exploits the sharing between the different outputs to produce a minimum cover.

In the second realization no sharing is possible. The first realization is reduced by two-level minimization to:

```
1111 001
1-11 100
0100 011
0001 101
1000 011
00-- 010
```

A good output encoding maximizes the sharing at the two-level minimization step.

Facts on Output Encoding: Dominance Constraints

Say that $enc(v_i) > enc(v_j)$ iff the code of v_i bit-wise dominates the code of v_j, i.e. for each bit position where v_j has 1, v_i also has 1.

If $enc(v_i) > enc(v_j)$, then ON_i can be used as a DC-set when minimizing ON_j.

Example of symbolic cover, encoded cover, minimized encoded cover:

```
0001 out1        0001 110        0001 110
00-0 out2        00-0 010        00-- 010
0011 out2        0011 010
```

Here $enc(out1) = 110 > enc(out2) = 010$. The input minterm 0001 of $out1$ has been merged into the single cube $00--$ that asserts the code of $out2$. Note that $00--$ contains the minterm 0001 that asserts $out1$.

Algorithms to exploit dominance constraints implemented in Cappuccino (De Micheli, 1986) and Nova (Villa and Sangiovanni, 1990).

Facts on Output Encoding: Disjunctive Constraints

If $enc(v_i) = enc(v_j) + enc(v_k)$ ($+$ here is the boolean disjunctive operator), ON_i can be reduced using ON_j and ON_k.

Example of symbolic cover, encoded cover, minimized encoded cover:

```
101 out1 1      101 11 1      10- 01 1
100 out2 1      100 01 1      1-1 10 1
111 out3 1      111 10 1
```

Here $enc(out1) = enc(out2) + enc(out3)$. The input minterm 101 of $out1$ has been merged both with the input minterm 100 of $out2$ (resulting in 10-) and with the input minterm 111 of $out3$ (resulting in 1-1). Input minterm 101 asserts 11 (i.e. the code of $out1$), by activating both cube $10-$ that asserts 01 and cube $1-1$ that asserts 10.

Algorithm to exploit dominance and disjunctive constraints implemented in esp_sa (Villa, Saldanha, Brayton and Sangiovanni, 1995).

Exact Output Encoding

Proposed by Devadas and Newton, 1991.

The algorithm consists of the following steps:

- Generate generalized prime implicants (GPIs) from the original symbolic cover.
- Solve a constrained covering problem, that requires the selection of a minimum number of GPIs that form an encodeable cover.
- Obtain codes (of minimum length) that satisfy the encoding constraints.
- Given the codes of the symbolic outputs and the selected GPIs, construct trivially a PLA with product-term cardinality equal to the number of GPIs.

Generation of GPIs

- Minterms in the original symbolic cover are called 0-cubes.
- Each 0-cube has a tag corresponding to the symbolic output it belongs to.
- 0-cubes can be merged to form 1-cubes, which in turn can be merged to form 2-cubes and so on.
- The rules for generating GPIs are:
  - When two k-cubes merge to form a $k+1$-cube, the tag of the $k+1$-cube is the union of the tags of the two k-cubes.
  - A $k+1$-cube can cancel a k-cube only if the $k+1$-cube covers the k-cube and they have identical tags.

Generation of GPIs

Example of function with symbolic outputs, list of 0-cubes, list of 1-cubes:

1101 out1	1101 (out1)	110- (out1, out2)
1100 out2	1100 (out2)	11-1 (out1, out3)
1111 out3	1111 (out3)	000- (out4)
0000 out4	0000 (out4)	
0001 out4	0001 (out4)	

Since the 1-cube $000-\ (out4)$ cancels the 0-cubes 0000 $(out4)$ and 0001 $(out4)$, the GPIs of the function are:

```
110- (out1, out2)
11-1 (out1, out3)
000- (out4)
1101 (out1)
1100 (out2)
1111 (out3)
```

Generation of GPIs by Reduction to PIs

One can transform a function with a symbolic output into a function with multiple binary-valued outputs such that the prime implicants (PIs) for this new multiple-output function are in 1-1 correspondence with the GPIs of the original function.

Example of transformation:

```
1101 out1       1101 0111
1100 out2       1100 1011
1111 out3       1111 1101
0000 out4       0000 1110
0001 out4       0001 1110
```

Generation of GPIs by Reduction to PIs

espresso -Dprimes

```
.i 4
.o 4
.p 6
000- 1110
1101 0111
1100 1011
110- 0011
1111 1101
11-1 0101
```

espresso -Dprimes -fr

```
.i 4
.o 4
.p 22
01-- 1111
10-- 1111
0-1- 1111
-01- 1111
--10 1111
--1- 1101
0--- 1110
-0-- 1110
1--0 1011
-1-0 1011
---0 1010
1-01 0111
-101 0111
```

```
1-0- 0011
-10- 0011
1--1 0101
-1-1 0101
1--- 0001
-1-- 0001
--01 0110
--0- 0010
---1 0100
```

Generation of GPIs by Reduction to PIs

Why is this a wrong transformation ?

```
1101 out1        1101 1000
1100 out2        1100 0100
1111 out3        1111 0010
0000 out4        0000 0001
0001 out4        0001 0001
```

espresso -Dprimes

```
.i 4
.o 4
.p 4
000- 0001
1111 0010
1100 0100
1101 1000
```

espresso -Dprimes -fr

```
.i 4
.o 4
.p 12
01-- 1111
10-- 1111
0-1- 1111
-01- 1111
--10 1111
0--- 0001
-0-- 0001
```

```
--1- 0010
1--0 0100
-1-0 0100
1-01 1000
-101 1000
```

Encodeability of a Set of GPIs

Given all the GPIs, one has to select a minimum subset of GPIs such that they cover all the minterms and form an *encodeable cover*.

- Say that minterm m belongs to symbolic output v_m.
- Obviously, in any encoded and optimized cover, m has to assert the code given to v_m, namely $e(v_m)$.
- Let the selected set of GPIs be p_1, .. p_G.
- Let the GPIs that cover m in this selected subset be $p_{m,1}$, .. $p_{m,M}$.
- For functionality to be maintained

$$\bigcup_{i=1}^{M} \bigcap_{j} e(\ v_{p_{m,i},\ j}\ ) \quad = \quad e(\ v_m\ ) \qquad \forall \quad m.$$

  where the $v_{p_{m,i},\ j}$ are the symbolic outputs in the tag of the GPI $p_{m,i}$. These equations define a set of encoding constraints on the selected GPIs.

Encodeability of a Set of GPIs

Example of a selection of GPIs:

```
110- (out1, out2)
11-1 (out1, out3)
000- (out4)
```

Encoding constraints for each minterm:
1101: $(enc(out1) \cap enc(out2)) \cup (enc(out1) \cap enc(out3)) = enc(out1)$
1100: $(enc(out1) \cap enc(out2)) = enc(out2)$
1111: $(enc(out1) \cap enc(out3)) = enc(out3)$
0000: $(enc(out4) = enc(out4)$
0001: $(enc(out4) = enc(out4)$

If an encoding can be found that satisfies all these constraints, then the selection of GPIs is encodeable.

The constraints associated with a selection of GPIs may be mutually conflicting.

Covering with Encodeability Constraints

Solve a constrained covering problem, that requires the selection of a minimum number of GPIs that form an encodeable cover.

Must adapt definitions of domination.

Once a selected set of GPIs covers all elements, perform an encodeability check. If the cover is not encodeable, branch-and-bound to find a minimum number of GPIs which render the selected set encodeable.

An 'encodeability' lower bound must be defined.

Computation of the Codes

If a selection of GPIs covers all minterms and is encodeable, then codes of minimum length must be obtained that satisfy the encoding constraints.

Best algorithm to check satisfiability and find codes of minimum length based on encoding-dichotomies (Saldanha, Villa, Sangiovanni 1991).

Example of an encodeable selection of GPIs:

```
110- (out1, out2)
11-1 (out1, out3)
000- (out4)
```

Codes of minimum length that satisfy the encoding constraints are:

```
out1: 11
out2: 01
out3: 10
out4: 00
```

Construction of the Optimized Cover

Once codes have been computed, it is easy to compute an encoded and optimized cover.

The cover will contain the selected GPIs. For each GPI, the codes corresponding to all the symbolic values in the tag of the GPI are bitwise ANDed to produce the output part.

Example of an encodeable selection of GPIs and of corresponding optimized cover, using the codes previously computed:

```
110- (out1, out2)       110- 01 1
11-1 (out1, out3)       11-1 10 1
000- (out4)             000- 00 1
```

Correctness of the Procedure

Theorem. A minimum cardinality encodeable cover can be made up entirely of GPIs.

Theorem. The selection of a minimum cardinality encodeable cover of GPIs represents an exact solution to the output encoding problem.

[Devadas and Newton, 1991]

Problems with the Procedure

Only very small problems could be attempted because:

- Number of GPIs exceeds soon memory limitations
- Covering table exceeds soon capabilities of existing covering solvers.

Some heuristics proposed [Devadas and Newton, 1991], but no conclusive experimental evidence provided.

Problems with the Procedure

Covering with encodeability constraints is a clumsy procedure.

- A binate covering formulation has been proposed that combines covering and encodeability check [Somenzi, 1991].
- It returns an encodeable selection of GPIs and codes that satisfy the encoding constraints.
- Limited to a fixed code-length.
- Current implementations not practical at all.

Extension to Symbolic Output Don't Cares

An extension to the case when some input minterms assert one of a set of output symbols has been treated in detail by Lin and Somenzi, 1990.

The problem is formulated as one of minimizing symbolic boolean relations and an algorithm based on binate covering has been proposed.

Input-Output Encoding for Two-Level Implementations

If symbolic variables appear both in the input and output part, the previous techniques for input encoding and output encoding can be unified.

In particular, we are interested to the case of *state assignment*, that is an input-output encoding problem where one symbolic variable (representing the states) appears both in the input and output part.

Facts on Input-Output Encoding: Dominance Constraints

The initial specification:

```
10 st1     st2 11
00 st2     st2 11
01 st2     st2 00
00 st3     st2 00

10 st2     st1 11
00 st1     st1 --
01 st3     st0 00
```

is equivalent to:

```
-0 st1,st2 st2 11
0- st2,st3 st2 00

10 st2     st1 11
00 st1     st1 --
01 st3     st0 00
```

provided that $enc(st1) > enc(st2)$, and $enc(st0) > enc(st2)$ (i.e. $st1$ asserted implies $st2$ asserted and $st0$ asserted implies $st2$ asserted) and face constraints $(st1, ts2), (st2, st3)$ are satisfied.

Facts on Input-Output Encoding: Disjunctive Constraints

The initial specification:

```
01 st2     st1 0
01 st1     st2 0
01 st4     st3 0
```

is equivalent to:

```
01 st2,st4 st1 0
01 st1,st4 st1 0
```

provided that $enc(st3) = enc(st1) \vee enc(st2)$ (i.e. $st1$ and $st2$ asserted are equivalent to $st3$ asserted and face constraints $(st2, st4), (st1, st4)$ are satisfied.

A solution is $enc(st1) = 01, enc(st2) = 10, enc(st3) = 11, enc(st4) = 00$

Exact State Assignment for Two-Level Implementations

A technique based on GPIs can be extended to state assignment.

- Each minterm has a tag corresponding to the symbolic next state whose ON-set it belongs to.
- Each minterm also has a tag that corresponds to all the outputs asserted by the minterm.
- Minterms in the original symbolic cover are called 0-cubes.

Generation of GPIs

- 0-cubes can be merged to form 1-cubes. Merging may occur between minterms with the same binary-valued part and different multiple-valued parts or uni-distant binary-valued parts and the same multiple-valued parts. The next state tag of the 1-cube is the union of the next state tags of the two minterms. The binary-valued output tag of the 1-cube contains only the outputs that both minterms assert.

- A 1-cube can cancel a 0-cube iff their next state and binary-valued output tags are identical and their multiple-valued parts are identical (except when the multiple-valued input part of the 1-cube contains all the symbolic states).

Generation of GPIs

Generalizing to k-cubes, the rules for generating GPIs are:

- A $k+1$-cube formed from two k-cubes has a next state tag that is the union of the two k-cubes' next state tags and an output tag that is the intersection of the outputs in the k-cubes' output tags.

- A $k+1$-cube can cancel a k-cube only if their multiple-valued input parts are identical or if the multiple-valued input part of the $k+1$-cube contains all the symbolic states. In addition, the next state and output tags have to be identical.

A cube with a next state tag containing all the symbolic states and with a null output tag can be discarded.

Generation of GPIs

Example of FSM and list of gpis (gpis are denoted by a *):

```
0 s1 s1 1                           * 0 100 (s1) (o1)
1 s1 s2 0                           * 1 100 (s2) ()
1 s2 s2 0                           * 1 010 (s2) ()
0 s2 s3 0                           * 0 010 (s3) ()
1 s3 s3 1                             1 001 (s3) (o1)
0 s3 s3 1                             0 001 (s3) (o1)
                                    * - 100 (s1,s2) ()
                                      0 110 (s1,s3) ()
                                    * 0 101 (s1,s3) (o1)
                                    * 1 110 (s2) ()
                                      1 101 (s2,s3) ()
                                    * - 010 (s2,s3) ()
                                      1 011 (s2,s3) ()
                                    * 0 011 (s3) ()
                                    * - 001 (s3) (o1)
                                    * 0 111 (s1,s3) ()
                                    * - 011 (s2,s3) ()
                                    * 1 111 (s2,s3) ()
                                      - 110 (s1,s2,s3) ()
```

Generation of GPIs by Reduction to PIs

One can transform a STT into a multiple-valued input, binary-valued output function such that the GPIs for the STT correspond to the PIs of the binary-valued output function.

Example of transformation:

```
0 s1 s1 1        0 001 1 110 110
1 s1 s2 0        1 001 0 101 110
1 s2 s2 0        1 010 0 101 101
0 s2 s3 0        0 010 0 011 101
1 s3 s3 1        1 100 1 011 011
0 s3 s3 1        0 100 1 011 011
```

An explanation of the form taken by the transformation involves some technicalities.

Encodeability of a set of GPIs and Covering

The rest of the procedure follows what seen for output encoding.

The encodeability check must take into account that GPIs may carry also face constraints. Although this makes the constraint satisfaction problem more complex, it can still be solved with the algorithms referenced previously.

Example of selection of GPIs for previous example:

```
- 010 (s2,s3) ()
- 001 (s3) (o1)
0 100 (s1) (o1)
1 110 (s2) ()
```

Encoding constraints for each minterm:
0 100:
1 100:
1 010: $((enc(s2) \cap enc(s3)) \cup enc(s2) = enc(s2)$
0 010: $(enc(s2) \cap enc(s3)) = enc(s3)$
1 001:
0 001:

face encoding constraint: (s1, s2) from GPI 1 110 ($s2$) ()

Codes and Construction of the Cover

Codes of minimum length that satisfy the encoding constraints are:

```
s1: 01
s2: 11
s3: 10
```

Corresponding optimized cover:

```
- 010 (s2,s3) ()        - 11 10 0
- 001 (s3) (o1)         - 10 10 1
0 100 (s1) (o1)         0 01 01 1
1 110 (s2) ()           1 -1 11 0
```

Problems with the procedure

Only very small problems could be attempted because:

- Number of GPIs exceeds soon memory limitations
- Covering table exceeds soon capabilities of existing covering solvers.
- Also binate covering formulation that combines covering and encodeability check not practical at all.

Encoding Algorithms for Multi-Level Implementations

- Given a function specified with symbolic variables, encoding algorithms for multi-level implementations try to minimize the number of literals of the encoded and multi-level minimized implementation.

- Current multi-level encoding algorithms can be classified as:

  1. Estimation-based algorithms, that define a distance measure between symbols. If ”close” symbols are assigned ”close” codes (in terms of Hamming distance), multi-level synthesis should give good results.
     mustang and *jedi* belong to this class.

  2. Synthesis-based algorithms, that use the result of a multi-level optimization on one-hot encoded or unencoded symbolic cover to drive the encoding process.
     muse and *mis-MV* belong to this class.

Mustang

- *Mustang* uses the state transition graph to assign a weight to each pair of symbols. This weight measures the desirability of giving to the two symbols codes that are ”as close as possible”.

- *Mustang* has two distinct algorithms to assign the weights, one of them (”fanout oriented”) takes into account the next state symbols, while the other one (”fanin oriented”) takes into account the present state symbols.

- Such a pair of algorithms is common to most multi-level encoding programs, namely *mustang*, *jedi* and *muse*.

Mustang

Adjacency Embedding

Reference: [DMNSV88]

1. Construct the *attraction graph*.

2. Use this graph for code assignment.

Attraction graph: The vertices correspond to states in the STG. The weight on an edge, $w(s1, s2)$ indicates the number of different places these two states will appear together in the logic description.

Two ways in which the attraction graph is constructed:

1. *Fanin-Oriented Algorithm*

2. *Fanout-Oriented Algorithm*

Mustang

- *Fanin-Oriented Algorithm*:
  - # of places where a state pair asserts the same output

    $$\begin{array}{cccc} *** & s1 & s2 & **1* \\ *** & s3 & s4 & **1* \end{array}$$

    Add 1 to $w(s1, s3)$ since they both result in $out3 = 1$.
  - # of places where a state pair have same next state

    $$\begin{array}{cccc} *** & s1 & s2 & **** \\ *** & s3 & s2 & **** \end{array}$$

    Add $n/2$ to $w(s1, s3)$ since they go to $s2$. (n is the number of bits in the code. On the average, half the bits in the code are 1.)

Mustang

- *Fanout-Oriented Algorithm*:
  - # of places where same input causes transition to next state pair

    $$\begin{array}{llll} *0* & s1 & s2 & **** \\ *0* & s3 & s4 & **** \end{array}$$

    Add 1 to $w(s2, s4)$ since they both use $in2 = 0$.
  - # of places where same present state causes transition to next-state pair

    $$\begin{array}{llll} *** & s1 & s2 & **** \\ *** & s1 & s4 & **** \end{array}$$

    Add n to $w(s2, s4)$ since they have a transition from $s1$.

Mustang

Code Assignment

Assign codes such that:

$$\sum_{(s_i,s_j)} w(s_i, s_j) * distance(enc(s_i), enc(s_j))$$

is minimized.

Annealing Solution: Use some heuristic to select an initial solution. Pairwise interchange to improve solution.

Approximate solution:

```
while(unassigned codes) {
    (s1, s2) =  edge with highest weight;
    if(enc(s1) != NIL) enc(s2) = closest(enc(s1));
    else if(enc(s2) != NIL) enc(s1) = closest(enc(s2));
    else (enc(s1),  enc(s2)) = closest unused codes;
    foreach((si, sj))
        delete (si,sj) if both si and sj
        have codes assigned;
}
```

Results sensitive to the initial form of the STT.

Jedi

- *Jedi* is aimed at generic symbol encoding rather than at state assignment, and it applies a set of heuristics that is similar to *mustang*'s to define a set of weights among pairs of symbols.

- It uses either a simulated annealing algorithm or a greedy assignment algorithm to perform the embedding.

- The proximity of two cubes in a symbolic cover is defined as the number of non-empty literals in the intersection of the cubes. It is the "opposite" of the Hamming distance between two cubes, defined as the number of empty literals in their intersection.

Muse

- *muse* uses a one-hot encoding for both input and output symbols, and then performs a multi-level optimization.

- Some of the actual potential optimizations can be evaluated, and their gain can be used to guide the embedding.

Steps of Muse

- Encode symbolic inputs and outputs with one-hot codes.
- Use *misII* to generate an optimized boolean network.
- Compute a weight for each symbol pair
- Use a greedy embedding algorithm trying to minimize the sum over all state pairs of the weighted distance among the codes.
- Encode the symbolic cover, and run *misII* again.

Mis-MV

- Performs input encoding in the multi-level case.
- As for the two-level case, perform a multi-level symbolic minimization, and derive constraints that, if satisfied, can guarantee some degree of minimality of the encoded network.
- *Mis-MV*, unlike the previous programs, performs a full *multi-level multiple-valued* minimization of a network with one symbolic input. Its algorithms are an extension to the multiple-valued case of those used by *misII*

Steps of Mis-MV

- Read the symbolic cover. The symbolic output is encoded one-hot, the symbolic input is left as a multiple-valued variable.
- Perform multi-level optimization (simplification, common subexpression extraction, decomposition) of the multiple-valued network.
- Encode the symbolic input so that the total number of literals in the encoded network is minimal (either simulated annealing or a dichotomoy-based algorithm are used for this purpose).