Two-Level Boolean Minimization

Two-level Boolean minimization is used to find a sum-of-products representation for a multiple-output Boolean function that is optimum according to a given cost function. The typical cost functions used are the number of product terms in a two-level realization, the number of literals, or a combination of both. The two steps in two-level Boolean minimization are:

1. Generation of the set of **prime product-terms** for a given function.

2. **Selection** of a minimum set of prime terms to implement the function.

We will briefly describe the Quine-McCluskey method which was the first algorithmic method proposed for two-level minimization and which follows the two steps outlined above. State-of-the-art logic minimization algorithms are all based on the Quine McCluskey method and also follow the two steps above.

Prime Term Generation

The Quine-McCluskey method is best illustrated with an example. Consider the completely specified Boolean function shown in (1A). It has been represented as a list of **0-terms**, which are fully specified product terms with no don't-care entries. Next, each 0-term has an associated decimal value obtained by converting the binary number represented by the 0-term into a decimal number—for instance the value of `0000` is `0` and that of `1110` is `14`. Each pair of 0-terms is checked to see if they can be merged into a single 1-term. Two 0-terms can be merged if they differ in exactly one position. The terms generated in our example by merging the pairs of 0-terms are shown in (1B). These terms are called 1-terms because they have exactly one "–" entry. Next, these l-terms are examined in pairs to see if they can be merged into 2-terms. This can only be done if the 1-terms differ in exactly one position and they have their "–" literal in the same position. Two 2-terms can be formed as shown in (1C). If a k-term is formed by merging two (k-l)-terms, then the two (k-1) l-terms are not primes and are marked so that they can be discarded later. The process ends when no merging is possible in the final set of L-terms. The unmarked k-terms, where $O \leq k \leq L$, are the complete set of prime terms of the function. The five prime terms for the function in this example are in **bold** type and labelled [A] through [E]. In the case of incompletely specified functions the initial list of 0-terms includes those 0-terms in the ON-set and the DC-set (don't-care set).

0	0000	**0, 8**	**-000 [E]**	**8,9,10,11**	**10-- [B]**
5	0101	**5, 7**	**01-1 [D]**	**10,11,14,15**	**1-1- [A]**
7	0111	**7,15**	**-111 [C]**		
8	1000	8, 9	100-		
9	1001	8,10	10-0		
10	1010	9,11	10-1		
11	1011	10,11	101-		
14	1110	10,14	1-10		
15	1111	11,15	1-11		
		14,15	111-		

(1A) 0-terms (1B) 1-terms (1C) 2-terms

Prime Term Table

We now construct the prime term table as shown below. The rows of the prime term table are the 0-terms of the ON-set of the function (we do not need to choose 0-terms in the DC-set) and the columns are the prime terms. An "X" in the prime term table in row i and column j signifies that

the 0-term corresponding to row i is contained by the prime corresponding to column j. For instance, the 0-term `0000` is contained only by prime E, `-000`.

	A	B	C	D	E
0000	.	.	.	.	X
0101	.	.	.	X	.
0111	.	.	X	X	.
1000	.	X	.	.	X
1001	.	X	.	.	.
1010	X	X	.	.	.
1011	X	X	.	.	.
1110	X	.	.	.	.
1111	X	.	X	.	.

Given the prime term table, we have to select a minimum set of primes (columns) such that there is at least one X in every row. This is the classical minimum covering problem.

Essential Prime Terms

A row with a single X signifies an essential prime term. Any prime implementation for the function will have to include the prime that contains the 0-term corresponding to this row because this 0-term is not contained by any other prime (column). In the prime term table above A, B, D, and E are essential prime terms.

We select the essential prime terms since they have to be included in any prime implementation. This results in an implementation for the function, since selecting A, B, D, and E results in an X in each row.

Dominated Columns

Some functions may not have essential prime terms. Consider the hypothetical prime term table shown in (2A). There is no row with a single X. It is necessary to make an arbitrary selection of a prime to begin with. Assume that prime A is selected. We obtain the reduced table of (2B) after deleting column A and the two rows contained by A, namely `0000` and `0001`. A column U of a prime term table is said to dominate another column V if U contains every row contained by V. In the reduced table of (2B) column B is dominated by column C and column H is dominated by column G. We can delete the dominated columns, since selecting the dominating column will result in covering more uncontained 0-terms than the dominated column. Note that the dominating column might not exist in a minimum solution.

	A	B	C	D	E	F	G	H
0000	X	.	.	.	.	.	.	X
0001	X	X	.	.	.	.	.	.
0101	.	X	X	.	.	.	.	.
0111	.	.	X	X	.	.	.	.
1000	.	.	.	.	.	.	X	X
1010	.	.	.	.	.	X	X	.
1110	.	.	.	.	X	X	.	.
1111	.	.	.	X	X	.	.	.

(2A) Prime table

	B	C	D	E	F	G	H
0101	X	X	.	.	.	.	.
0111	.	X	X	.	.	.	.
1000	.	.	.	.	.	X	X
1010	.	.	.	.	X	X	.
1110	.	.	.	X	X	.	.
1111	.	.	X	X	.	.	.

(2B) Table with A selected

	C	D	E	F	G
0101	X	.	.	.	.
0111	X	X	.	.	.
1000	.	.	.	.	X
1010	.	.	.	X	X
1110	.	.	X	X	.
1111	.	X	X	.	.

(2C) Table with B & H removed

Reducing the table (2B) gives the table of (2C). In this table C and G are relatively essential prime terms. (They are not essential to the original table but become essential given the selection of prime A.) Choosing C and G results in the selection of E, which completes the implementation f = {A, C, E, G}. We are not guaranteed that this implementation is minimum; we have to backtrack to our arbitrary choice of selecting prime A and delete prime A from the table, *i.e.*,

explore the possibility of constructing an implementation that does not have A in it. This results in f = {B, D, F, H}.

A Branch Covering Strategy

The covering procedure of the Quine-McCluskey method is summarized below. The input to the procedure is the prime term table T.

1. Delete the dominated primes (columns) in T. Detect essential primes in T by checking to see if any 0-term is contained by a single prime. Add these essential primes to the selected set. Repeat until no new essential primes are detected.

2. If the size of the selected set of primes equals or exceeds the best solution thus far return from this level of recursion. If there are no elements left to be contained, declare the selected set as the best solution recorded thus far.

3. Heuristically select a prime.

4. Add the chosen prime to the selected set and create a new table by deleting the prime and all 0-terms that are contained by this prime in the original table. Set T to this new table and go to Step 1.

   Then, create a new table by deleting the chosen prime from the original table without adding it to the selected set. No 0-terms are deleted from the original table. Set T to this new table and go to Step 1.

Multiple-Output Functions

An advantage of the Quine-McCluskey method is that it is generalized to functions with multiple outputs. Suppose we have a multiple-output function $f = \{f_1, ..., f_m\}$. Each term in f has an input part and an output part. The output part indicates which of the ON-sets of the f_j's that the 0-term belongs to. If the 0-term belongs to the j^{th} output, then the j^{th} position of the output part has a 1. For example, the two-output function $f = \{f_1, f_2\}$ where $f_1 = \text{and}(x_1,x_2)$ and $f_2 = \text{and}(x_2,x_3)$ can be represented as:

```
11-  10
-11  01
```

We can represent the 0-terms of each output *separately* as:

```
110  10
111  10
011  01
111  01
```

The 0-terms for the multiple-output function are:

```
011  01
110  10
111  11
```

The primes are generated given the 0-terms for the multiple-output function immediately above in a manner similar to single-output case. Two 0-terms are merged into a l-term if their input parts differ in exactly one bit and their output parts *intersect*, i.e., have at least one output in common. The l-term formed will have a "-" entry in the input bit which was different across the 0-terms

and will have an output part equaling the intersection (bitwise AND) of the two output parts of the 0-terms, i.e., will have a 1 in all positions where both the 0-terms have a 1, and O's in all other positions. A similar procedure is followed for generating k-terms from (k- l)-terms.

A (k-1)-term is a prime term if it is not **completely** contained in any k-term. In the case of multiple-output functions, two (k-1)-terms may generate a k-term but neither (k-1)-term may be completely contained in the k-term. This happens when the output parts of the two (k-1)-terms intersect, but are not equal.

Once all the prime terms have been generated, the prime term table is constructed. The columns of the prime term table are primes and the rows correspond to the 0-terms of the different outputs listed *separately*. A minimum selection of primes such that there is an X in every row corresponds to a minimum implementation of the multiple-output function.

An example of using the Quine-McCluskey method on a two-output function is shown below. In (3A) the 0-terms are listed separately for each output. The numbers to the left of the 0-terms correspond to the decimal value of the 0-term, including both the input and output part of the term. For example term `001  10` has the decimal value 6.

```
 6  001 10        5,6    001 11       5,6,13,14  0-1 11 [A]
14  011 10        13,14  011 11       6,22       -01 10 [B]
18  100 10        18     100 10       13,29      -11 01 [C]
22  101 10        22     101 10       18,22      10- 10 [D]
                  25     110 01       25,29      11- 01 [E]
 5  001 01        29     111 01
13  011 01
25  110 01
29  111 01
```

(3A) 0-terms | (3B) 0-terms for multiple output function | (3C) 1-terms

In (3B), the 0-terms for the multiple-output function are listed. Some of these 0-terms have more than one 1 in their output parts, for example, the 0-term `001 11`. These 0-terms are merged in a pairwise manner to generate the 1-terms of (3C). For example, the 0-terms `001 11` and `101 10` differ in one bit in their input parts and share a 1 in their first position of their output parts. Therefore, they can be merged into the 1-term `-01 10`.

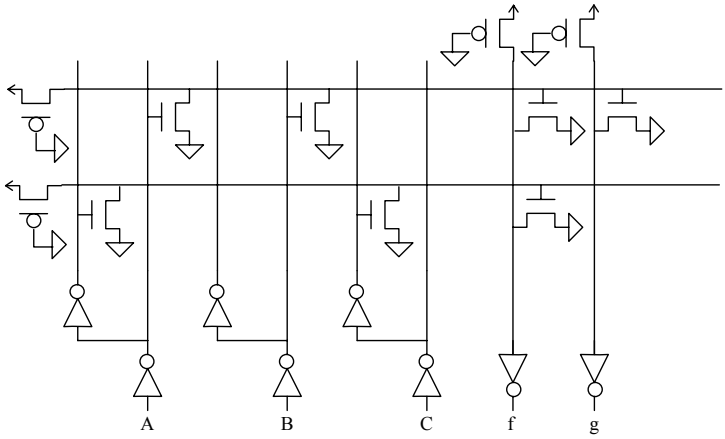
The 1-terms cannot be merged further and we obtain five prime terms for the two-output function A, B, C, D and E as shown in (3C). The 0-terms in (3B) are all contained in at least one of the 1-terms and hence no 0-term is a prime term. (This may not always be the case. Consider two 0-terms `111 101` and `110 110` corresponding to a three-input, three-output function. They can be merged into the 1-term `11- 100`. The 1 term does not *completely* contain either of the 0-terms. It contains the parts of the 0-terms corresponding to the first output, but not the second or third outputs.)

The prime term table for this set of primes is shown below. Here the columns correspond to the five prime terms and the rows are the 0-terms for each output considered separately. In order to obtain an implementation for the function, we need to select a set of prime terms such that there is an X in every row. We can identify essential prime terms from this table. `011 10` is only covered by prime A hence A is essential. `100 10` is covered only by D hence D is essential. Finally, `110 01` is covered only by E hence E is essential. Selecting A, D and E results in a X in every row, and we have a minimum implementation consisting of three prime terms.

```
          A B C D E
001 10    X X . . .
011 10    X . . . .
001 01    X . . . .
011 01    X . X . .
100 10    . . . X .
101 10    . X . X .
110 01    . . . . X
111 01    . . X . X
```

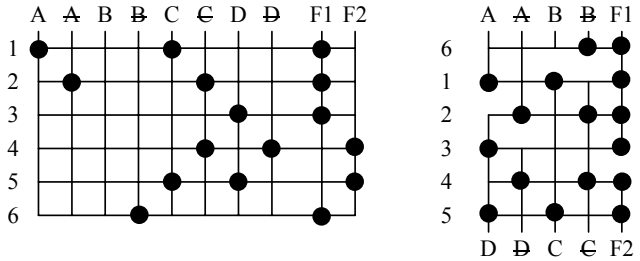
Programmable Logic Array Folding

Once a minimal logic implementation for a multiple-output function has been determined, a layout for the PLA can be automatically generated. The layout for the PLA will reflect the transistor-level topology show below.



The PLA obtained may be very *sparse,* i.e., may have a large area but the number of transistors in the PLA may be very small. Layout optimization is required to produce a more efficient structure.

One method of PLA layout optimization is called *folding*. In a folded PLA, more than one input/output signal can occupy the same column, and/or more than one product term can occupy the same row. There are thus two flavors of folding, column folding and row folding corresponding to folding inputs/outputs and folding product term lines. An example of column folding is shown below. Folding can reduce the area of the PLA by as much as 50%. The layout for folded PLAs is more complicated than for unfolded PLAs, however, the process can still be efficiently automated.



Sparser PLAs that contain fewer transistors are easier to fold since the constraints associated with folding are fewer. It is important that the logic minimization step produce maximally sparse PLAs, i.e., truth-tables with a maximal number of "-" entries in the AND plane and a minimal number of 1's in the OR plane.