of each circuit in the traditional factored form. Note that, here, the factored literal counts have been reduces by a very extensive optimization script "script.rugged" from SIS [10]. And Column *3* presents the factored literal count using our proposed invert-factored form. The last column gives the ratio of Columns *3* and *2*. For example, consider circuit frg1, the literal count of the traditional factored form is *130*. Applying the algorithm presented in Section 3, we can express the circuit in the invert-factored form with the literal count *52* which is *40%* of the traditional factored form. Moreover, for all the circuits listed in this table, we can find the literal counts of the invert-factored form are less than the ones of the traditional factored form.

Table 1: Literal counts of traditional and invert-factored forms.

Circuits	Factored Form	Invert-Factored Form	Ratio
frg1	130	52	0.4
misex3	1223	1021	0.835
b12	95	80	0.842
9sym	281	239	0.851
Z9sym	266	239	0.898
pdc	1492	1362	0.913
duke2	739	680	0.920
spla	1394	1291	0.926
f51m	130	122	0.938
squar5	62	60	0.968
b9	131	127	0.969
s526n	251	244	0.972
s526	254	247	0.972
rot	733	713	0.973
inc	111	108	0.973
seq	3507	3413	0.973
rd73	158	154	0.975
cps	2718	2652	0.976

Table 1: Literal counts of traditional and invert-factored forms.

Circuits	Factored Form	Invert-Factored Form	Ratio
bw	178	174	0.978
sao	152	149	0.980

6 Conclusions

In this paper, we have presented a novel representation of Boolean functions, called the invert-factored form representation. This representation mainly takes advantage of the inversion of whole or part of a Boolean function which may lead to fewer literals. Based on this novel presentation, the algorithm for finding the minimal expression is described. Experimental results also shows the literal counts based on the novel representation are smaller than on the traditional factored form presentation.

References

[1] K. Bartlett, W. Cohen, A. J. De Geus, and G. D. Hachtel, "Synthesis of Multilevel Logic under Timing Constraints", *IEEE Trans. on Computer-Aided Design of Integrated Circuits*, pp.582-595, Oct. 1987.

[2] D. Bostick, G. D. Hachtel, R. Jacoby, M. R. Lightner, P. Moceyunas, C. R. Morrison, and D. Ravenscroft, "The Boulder Optimal Logic Design System", *Proc. of ICCAD*, pp. 62-65, Nov. 1987.

[3] R. K. Brayton and C. McMullen, "The Decomposition and Factorization of Boolean Expression", *Proc. of the International Symposium on Circuits and Systems*, pp. 49-54, May. 1982.

[4] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli, "Mutilevel Logic Synthesis", *Proc. of the IEEE*, pp.264-300, Feb. 1990.

[5] R. K. Brayton, R. Rudell, A. L. Sangiovanni-Vincentelli, and A. Wang, "MIS: A Multiple-Level Logic Optimization System", *IEEE Trans. on Computer-Aided Design of Integrated Circuits*, pp. 1062-1081, Nov. 1987.

[6] S. Devadas, A. Ghosh and K. Keutzer, *Logic Synthesis*, McGraw-Hill, Inc., 1994.

[7] G. D. Hachtel and Fabio Somenzi, *Logic Synthesis and Verification Algorithm.*, Kluwer Academic Publishers, 1996.

[8] G. D. Micheli, *Synthesis and Optimization of digital circuits*, McGraw-Hill, Inc., 1994.

[9] S. Minato, "Fast Generation of Prime-Irredundant Coverts from Binary Decision Diagrams, *IEICE Trans. on Fundamentals*, E76-A(6), 967-973, June 1993.

[10] SIS: A System for Sequential Curcuit Synthesis, University of California, Berkeley, CA, Rep. M92/41, 1992.

0-7803-5482-6/99/$10.00 ©2000 IEEE

$$F = \overline{\overline{F_1} + \overline{F_2}} \ . \qquad \text{(EQ 4)}$$

The factored literals in the above four Boolean expressions are different. After performing partial inversion, we can obtain all four factored forms and can then choose the best one among those four expressions (EQ 1 to EQ 4) above. As an example, consider the expression

$$F_B = a(b + \bar{c}) + c(b + \bar{a}) + de$$

which has the factored literal count 8. By EQ 2, we invert the partial expression $a(b + \bar{c}) + c(b + \bar{a})$,and we have

$$F_B = \overline{\overline{a(b + \bar{c}) + c(b + \bar{a})} + de} = \overline{\overline{a\bar{c}} + a\bar{b}c} + de$$

which has the factored literal count of *7* while without inversion the factored literal count is *8*. To generate an invert-factored form in this case, one need to decide the partition of a Boolean function first. A simple way to do so is to partition a Boolean expression so that two sub-functions F_1 and F_2 have different variable supports. In the above example, F_1 has the variable support of {*a, b, c*} and F_2 has the variable support of {*d, e*}. Their support sets are disjoint.

Case 3: Consider the don's care (DC) condition. Don't cares can be used to minimize the size of an inverted Boolean function. Properly applying the don's care (DC) condition, the invert-factored literal count of *F* can be further decreased. For example, consider Boolean function F_C (*a, b, c*) = $\sum m(1, 3, 6, 7)$ with $DC_C = \sum m(4)$. The traditional factored form is

$$F_C = a(b + \bar{c}) + c(b + \bar{a})$$

which has the factored literal count *6*. However, applying DC_C on $\overline{F}_C = \sum m(0, 2, 5)$, we can have the invert-factored form

$$F_C = \overline{\overline{ac} + a\bar{b}}$$

with the invert-factored literal count *4* less than *6*.

To apply the compatible observability don't cares (CODC), our proposed algorithm is based on the order of CODC such that we guarantee that a node is simplified after all its fanin nodes have been done. Beginning with PIs, for each node *N* with logic function *F*, we evaluated the minimal factored forms of *F* and $\overline{F}$. Next we decide to use *F* or $\overline{F}$ to implement *N*. If both of $\overline{F}$ literal count and factored literal count are less than $\overline{F}$'s, we choose $\overline{F}$; otherwise, *F* is chosen. During the selection, CODC can be applied. Once $\overline{F}$ is selected, one inverter must be added into the fanout-stem of node *N* and all functions of all *N*'s output nodes must be modified to reflect such change. The algorithm for simplifying node *N* with Boolean function *F* is given in Fig. 2.

```
optimization(N, F)
{
  F̄ = invert(F);
  if(F̄_factored_literal_count < F_factored_literal_count &&
     F̄_literal_count < F_literal_count){
       replace(F, F̄);
       modify the functions of all the fanout nodes
       of N;
  }
}
```

Fig. 2 Algorithm for minimizing factored literal count.

4 An Example of Benchmark Circuit

In this section, we use the benchmark circuit **frg1** to illustrate the advantage of the invert-factored form. In **frg1,** there is a node *d0* with boolean function F_{d0}. Using the traditional factored form representation, function F_{d0} can be expressed as

$F_{d0} = \bar{c}(\overline{gq}(\overline{m}(\bar{o} + \bar{j}) + \bar{y}(\overline{ow} + \bar{j}))(e + a)(\bar{u} + \bar{k}) + \bar{i}\,\bar{p}(\overline{st} + \bar{i})(e + a)(\overline{m}(\bar{o} + \bar{h}) + \bar{x}(\overline{ow} + \bar{h})) + \overline{ae}\,\bar{c}_0 + \bar{r}(\overline{hp}(\overline{xz} + \overline{m})(e + a)(\overline{stv} + \bar{k}) + \overline{jq}(\overline{yz} + \overline{m})(e + a)(\overline{uv} + \bar{k})) + (e + a)(\bar{j}(\overline{hr}(\bar{z} + \overline{m})(\bar{v} + \bar{k}) + \bar{i}) + \bar{g}(\overline{io}(\overline{w} + \overline{m}) + \bar{h}) + \bar{l}(\bar{i}(\overline{st} + \bar{g}) + \bar{v}(\overline{hst} + \bar{j}(\bar{u} + \bar{h})) + \overline{gu}) + (\overline{stuv} + \bar{k})(\overline{opqr}(\overline{wxyz} + \overline{m}) + \bar{l}) + \bar{n}(\overline{hz}(\bar{x} + \bar{j}) + \bar{i}(\bar{x}(\overline{w} + \bar{h}) + \overline{gw}) + \bar{y}(\bar{g}(\overline{w} + \bar{j}) + \bar{z}(\overline{w}\,\bar{x} + \bar{j})) + \overline{m} + \bar{l}))) + \bar{b}c$

with the factored literal count of ***121***. Using the invert-factored form representation, we invert the whole of function F_{d0} and we have

$F_{d0} = \overline{\overline{F}}_{d0}$ and $\overline{F}_{d0} = \bar{c}(\overline{aec}_0 + (lm(i(gz + h\,y) + j(g\,x + h\,w)) + n\,(h(iku + l(iq + jo)) + g(i(kv + lr) + j(k(t + s) + l\,p))))(e + a)) + bc$

with the invert-factored literal count of ***42*** less than ***121*** from the non-inverted one. In fact, for this circuit, even applying a very extensive optimization script, "script.rugged" from SIS [10], we get a even more factored literals of 126.

5 Experimental Results

We have implemented the algorithm in Fig. 2 and the results are given in Table 1. Column *1* gives the names of circuits. Column *2* presents the factored literal count

er inverter in the circuit. As a result, both in the factored form or in our method, we do not consider the cost of inverters.

One of the reasons which inhibit previous optimization to consider the inversion operations that the size of inversion of a Boolean equation can be exponential to the size of the non-inverted one. The exponential problem can cause memory exploration and huge CPU time. However, recently, there is research [9] which can quickly estimate the number of literals after inversion. With such estimation, one can find in advance that whether an inversion will cause memory exploration or not. If the size of an inversion of a Boolean function is within some bound, the inversion can then be carried out for better optimization results.

The remainders of this paper are organized as follows. The background for logic optimization is described in Section 2. In Section 3, we present the invert-factored form. Based on this novel form, the algorithm for finding a representation with the minimal literal counts will also be presented. Section 5 gives the literal counts of the invert-factored form expression for the benchmark circuit, *frg1*. The results are based on the algorithm presented in Section 3. Our conclusions is given in Section 6.

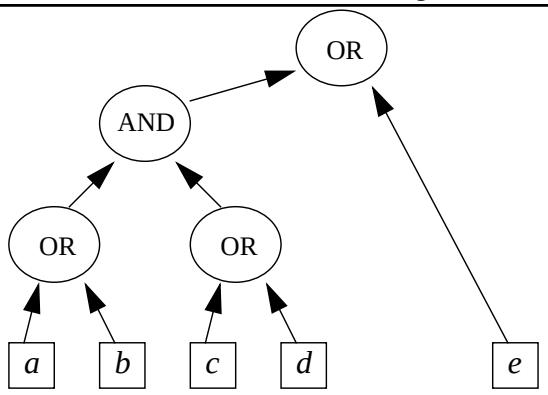

Fig. 1 The tree representation of $F = (a + b)(c + d) + e$.

2 Background

A *literal* is a Boolean variable or its complement. Literal count is the number of literals in a Boolean expression. In CMOS, the size of implementing a Boolean function can be estimated by the literal count of the Boolean function. Since multi-level circuits often result in a faster and smaller implementation of a Boolean function than two-level circuits, synthesis of multi-level circuits has become an attractive topic [1][2][3][4][5].

Similar to SOP form, a factored form is a way of expressing Boolean functions and is a more natural way for multi-level circuits than two level representation SOP. A factored form is a parenthesize representation of a tree network for a Boolean function, where each internal node is an AND or OR gate and each leaf is a literal [6][7]. And the literal count of a factored form representation is called the factored literal count. For example, a possible factored form of $F = ac + ad + bc + bd + e$ is $F = (a + b)(c + d) + e$ whose corresponding tree representation is given in Fig. 1. The factored literal count of F is *5*.

3 The invert-factored Form

In this section, we discuss a representation of a Boolean function called *the invert-factored form* which takes advantage of an inversion operation. The invert-factored form of a Boolean function has more freedom and can result in fewer factored literals than the traditional factored form. Some ways of generating an invert-factored form of a Boolean function are discussed as follows.

Case 1: Invert the whole expression of a Boolean function. Traditionally, two level minimization tries to reduce the literal count of an SOP and a "factoring" operation tries to reduce the factored literals of a given SOP. However, the inversion of a Boolean function is not explored before. The inversion of a function may have more literals but may have fewer factored literals.

For example, the traditional factored form of $F_A = a(b + \bar{c}) + c(b + \bar{a})$ has factored literal count of *6*. If F_A is inverted, we have

$$\bar{F}_A = \overline{a(b + \bar{c}) + c(b + \bar{a})}.$$

Since $F_A = \bar{\bar{F}}_A$, we have

$$F_A = \bar{\bar{F}}_A = \overline{\overline{a(b + \bar{c}) + c(b + \bar{a})}} = \overline{\overline{ac} + a\bar{b}c}$$

which has the factored literal count of *5*.

Case 2: Invert partial Boolean expression of a Boolean function. Consider a Boolean function F which are sum of F_1 and F_2 as in EQ 1

$$F = F_1 + F_2 , \qquad \text{(EQ 1)}$$

where *F1* and *F2* are in SOP forms. The Boolean function F can be written as one of the following:

$$F = \bar{\bar{F}}_1 + F_2 ; \qquad \text{(EQ 2)}$$

$$F = F_1 + \bar{\bar{F}}_2 ; \qquad \text{(EQ 3)}$$

A Compact Factored Form for a Boolean Function

J. C. Rau, Y. M. Chen, and S. C. Chang

Department of Computer Science and Information Engineering
National Chung-Cheng University
Chiayi, Taiwan, R. O. C.

Abstract

A factored form of a Boolean function is a common representation to express the complexity of a Boolean function in multi-level logic. However, a factored form which inhibits the appearance of the inversion operation is still a restricted way in representing a multi-level circuit. In this paper, we present a novel representation of a Boolean function, called *the invert-factored form representation*. This representation mainly takes advantage of the inversion of whole or part of a Boolean function so that fewer literals and better multi-level circuit implementation can be obtained. Based on this novel presentation, our algorithm attempts to find a minimal expression. Experimental results also show the literal counts based on the novel representation are smaller than those on the traditional factored form representation.

1 Introduction

There are many different representations for a Boolean function such as the truth table form, the sum of products (SOP) form and the factored form. Among these, the factored literal count of a Boolean function is a common way to measure the complexity of a Boolean function in multi-level logic. A factored form is a parenthesized representation of a Boolean function which allows only AND and OR operations [6][7][8]. Traditionally, a factored form is computed by recursively extracting common sub-functions in a Boolean expression. For example, suppose $F_1 = ab + a\bar{c} + bc + \bar{a}c$, then $F_1 = a(b + \bar{c}) + c(b + \bar{a})$ is a factored form for Boolean function F_1. In this example, the common sub-function a is factored out of sub-expression $ab + a\bar{c}$ and c is factored out of $bc + \bar{a}c$. The literal count of F_1 is 8 while the factored literal count of F_1 is 6. For another example, consider the Boolean function $F_2 = ab + cd + ef + g$ whose SOP form and factored form are the same. Although the literal count of F_1 is larger than the literal count of F_2, the factored literal count of F_1 is less than F_2. Ideally, the complexity of implementing F_1 is assumed to be cheaper than F_2 in multi-level logic. Quite frequently, a logic optimization algorithm may derive several new transformations and factored literal count can serve as a cost function to determine whether a new transformation is accepted or discarded [3]. If a new transformation by a logic optimization algorithm results in more factored literals, the new transformation will be rejected; otherwise it will be accepted.

The factored form of a Boolean function can be obtained recursively by algebraic or Boolean operations [7]. As an example, for Boolean function $F_3 = \bar{a}b + ac + bc + d$, the factored form $c(a+b) + \bar{a}b + d$ is algebraic, and the factored form $(a + b)(\bar{a} + c) + d$ is Boolean. Most previous work attempts to reduce the factored literal count using more complex Boolean operations from a fixed Boolean expression. On the other hand, a factored form itself is a restricted way of expressing a Boolean function in multi-level logic. The inversion operation is not allowed in a factored form. As an example, representation $\overline{d(e + ab) + ac(e + b)} + f$ is not a traditional factored form.

In this paper, we discuss a new form, called the invert-factored form, in which we take advantage of the inversion of whole or part of a Boolean function. We have found that some Boolean functions can be represented in a much compact way by allowing the inversion operation. For example, the Boolean function

$$F_D = \bar{\bar{F}}_D = \overline{\overline{d(e + ab) + ac(e + b)} + f}$$
$$= \overline{\bar{d}(\bar{c} + \bar{a}) + \bar{e}(\bar{a} + \bar{b})} + f$$

has the factored literal count of 7. Note that without inversion, the previous expression

$$F_D = d(e + ab) + ac(e + b) + f$$

has factored count of 9. When estimating the cost, we do not take into account the cost of an inverter because after logic optimization, usually, it is followed by a global inverter optimization algorithm to reduce the number of inverters. It is possible that an additional inverter can be removed with anoth-