設 計 自 動 化 60－15
（1991．12．12）

ファンイン数最小化による論理回路最適化

藤田昌宏 松永裕介

富士通研究所

川崎市中原区上子田中1015

テーブル参照タイプのFPGAに対し特に有効な多段論理回路最適化手法について述べる。従来の論理回路最適化手法は、各ゲートそれぞれをより簡単なものに変換する（リテラル数を減少させる）ことにより回路を簡単化しており、各ゲートの入力数はむしろ増加する傾向があった。しかし、テーブル参照タイプのFPGAは一定数（通常4か5）以下の入力であれば、任意の論理関数を実現できるため、各ゲートのファンイン数の最小化が望まれる。ここでは、ファンイン数最小化手法を示し、ベンチマーク回路への適用結果からその有効性を示す。

Multi-level Logic Minimization Based on Minimal Support

Masahiro Fujita and Yusuke Matsunaga

FUJITSU LABORATORIES LTD.

1015 Kamikodanaka, Nakahara_ku, Kawasaki 211, JAPAN

We present a method for multi-level logic minimization which is particularly suitable for the minimization of look-up table type FPGAs. Almost all multi-level logic minimizers are designed to minimize the complexity of each internal node and there is almost no consideration about reducing numbers of fan-ins of each internal node. Since look-up tables of FPGA can realize any functions of variables less than or equal to a predetermined number (usually 4 or 5), we should minimize the number of fan-ins for each node instead of reducing the complexity of the node, i.e. reducing the number of literals. We present a logic minimization method based on minimal support and show the effectiveness of our method by applying it to benchmark circuits.

1. INTRODUCTION

Field Programmable Gate Arrays (FPGAs) are getting more and more attention, because of their ease and speed of use and their growing integration density. However, the tools for FPGAs are not so well developed compared with tools for popular gate arrays and standard cells. For example, although there are many logic synthesis tools for (mask programmable) gate arrays and standard cells, there are only a limited number for FPGAs (Francis *et al* 1990, Murgai *et al* 1990). Currently logic synthesis tools for FPGAs are only limited to FPGA mapping, i.e. mapping technology independent circuits to cells for FPGAs. Moreover, they mainly use algebraic operations not Boolean operations, although Boolean operations have recently been successful in technology independent circuit optimization for gate arrays and standard cells. This is simply because FPGAs have different architectures and needs tools whose cost functions are very different, and the Boolean optimization method developed so far cannot be applied directly to these different cost functions.

There are two different architectures in FPGAs, i.e. look-up table and multiplexer types. In this paper, we present a Boolean minimization method for look-up table type FPGAs. Look-up table type FPGAs consist of function blocks which can realize any logic functions with fixed numbers of fan-ins. This contrasts with the cells of mask programmable gate arrays and standard cells, because they can only realize a limited number of different logic functions, such as AND, AND-OR-INVERT, etc.. For the minimization of mask programmable gate arrays and standard cells, numbers of literals have been used for the cost function of multi-level logic minimization. For look-up table type FPGAs, we have to minimize the numbers of nodes in a circuit after modifying each node so that it has only a limited number of fan-ins. This means we should modify circuit nodes to have fewer fan-ins. The method presented here does this by Boolean operations.

Our minimization algorithm works as follows. First, we select candidate nodes which may be used for fan-ins of the node currently being minimized. Then we calculate the characteristic function for the candidate nodes and the node being minimized. From it we can compute sets of minimal supports for each node being minimized (minimal number of fan-ins which can realize the function of the node) based on the Halatsis and Gaintans's method (1978) and its extension (for the case of simultaneous minimization of multiple nodes). From those sets we can get the minimal supports and the corresponding irredundant cover with those minimal supports. In this algorithm, we can use don't care sets derived from circuit structure by appropriately setting the characteristic function.

Note that almost all multi-level logic minimizers developed so far (Brayton *et al* 1987, Bostick *et al* 1987, Sato *et al* 1990) are designed to minimize the complexity of each internal node, i.e. reducing numbers of literals, and there is almost no consideration about reducing the numbers of fan-ins of each internal node. On the other hand, the method presented here minimizes the numbers of fan-ins and generates irredundant covers using those minimal supports. This gives us a possibility to apply the method as an alternative or as an aid to other multi-level logic minimizers even for popular gate arrays and standard cells.

This paper is organized as follows. First, we present Halatsis and Gaitanis's method by which we can compute minimal supports for positive irredundant sum-of-products forms for a node. Then, we apply it to node minimization for reducing numbers of fan-ins. Experimental results are also shown. We have implemented our algorithm and we have applied it to the look-up table type FPGA synthesis of ISCAS sequential benchmark circuits. Although our current implementation utilizes only very limited don't care sets, the results are very encouraging both in terms of quality and processing speed.

2. FUNCTIONAL REDUCTION AND MINIMAL SUPPORT

In this section, we show Halatsis and Gaintans's algorithm (1978) to get the sets of minimal supports of a given logic function. The correctness of the algorithm can be found in (Halatsis and Gaintans 1978). Similar discussions can be found in (Brown 1978). The example shown here is cited from (Halatsis and Gaintans 1978). We use "+" to represent OR and "*" to represent logical AND (but often omitted). Complement or negation of a formula is expressed with " ' " , i.e. p' represents the complement of p.

Suppose we are minimizing a node, u, and its candidate supports are a_1, a_2, ..., a_r (candidates which may be fan-ins of u). Halatsis and Gaintans's method or the method presented in (Brown 1978) has two

parts: the first part computes the set of minimal supports for u and the second part computes the irredundant sum-of-products forms. The following algorithm computes the set of minimal supports.

Algorithm 1: computing the set of minimal supports

(1) Express the relationship between the node to be minimized and its candidate supports as a characteristic function, $f(a_1,a_2,...,a_r u) = 1$.

(2) Compute the co-factors of f with respect to u and u', i.e. f_u and $f_{u'}$, and from them, compute the ON-set, R_1, and OFF-set, R_0, of u in sum-of-products form as follows:

$$R_1 = f_u(f_u f_{u'})' = pp_1+pp_2+...pp_m,$$

$$R_0 = f_{u'}(f_u f_{u'})' = qq_1+qq_2+...+qq_n$$

where pp_i and qq_i are product terms. Note that $f_u f_{u'}$ corresponds to the don't care set.

(3) For each pair (pp_i, qq_j), compute a complement free alterm (sum-of-literals) s_{ij},

ss_{ij} = Σ (literals that appear opposed in pp_i and qq_j)

for $1 \le i \le m$ and $1 \le j \le n$.

(4) Define a Boolean function F_u by the product-of-sums formula:

$$F_u = \prod_{i=1}^{m} \prod_{j=1}^{n} ss_{ij}$$

(5) Multiply out, to form a sum-of-products formula for F_u, and delete absorbed terms. With each term of the resulting formula, associate a set of arguments having the same letters; the resulting sets are the minimal supports for u among $a_1,a_2,...,a_r$

The second algorithm computes the positive irredundant sum-of-products form for u using the minimal supports which are obtained by the Algorithm 1.

Algorithm 2: computing the positive irredundant normal forms for u

(1) Same as (1) of Algorithm 1.

(2) Compute the cofactors of f with respect to u and u', i.e. f_u and $f_{u'}$, and from then, compute the ON-set, R_1,and OFF-set, R_0, of u in sum-of-products from as follows:

$$R_1 = f_u(f_u f_{u'})' = pp_1+pp_2+...pp_m,$$

$$R_0 = f_{u'}(f_u f_{u'})' = qq_1+qq_2+...+qq_n$$

where pp_i and qq_i are product terms.

Extract only uncomplemented literals for each product term of R_1 and call them F_1. Also extract only complemented literals for each product term of R_0 and call them F_0.

$$F_1 = p_1+p_2+...p_m,$$

$$F_0 = q_1+q_2+...+q_n$$

where p_i has only uncomplemented literals of pp_i and q_i has only complemented literals of qq_i.

(3) For each pair (p_i, q_j), compute a complement free alterm (sum-of-literals) s_{ij},

s_{ij} = Σ (literals that appear opposed in p_i and q_j)

for $1 \le i \le m$ and $1 \le j \le n$.

(4) For each p_i, compute H_{pi} and R_u as follows:

$$H_{pi} = \prod_{j=1}^{m} s_{ij} = t_1+t_2+...+t_{ki}$$

$$R_u = \prod_{H_{pi} = t_{i1}+t_{i2}+...+t_{ik}} (z_1+z_2+...+z_k)$$

where z^i are distinct literals corespondent to each distinct t_{ij} term.

(5) Multiply out, to form a sum-of-products formula for R_u, and delete absorbed terms. Each term of the resulting formula corresponds to a positive irredundant normal form for u. So, select one from them which has the minimum number of supports.

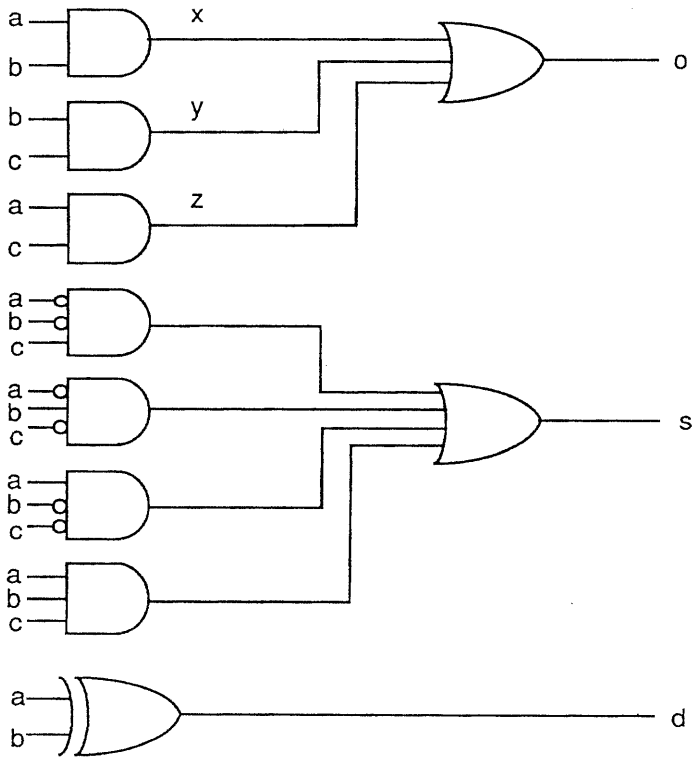
Note that the above algorithm computes the positive irredundant normal forms for u with minimal supports, although it can be easily extended to handle the (not necessarily positive) irredundant sum-of-products forms.

Example

Now let us show an example of the Algorithm 2 (the Algorithms 1 and 2 are very similar, so we here only show an example of the Algorithm 2 which is more complex). Suppose the following one-set, R_1, and off-set, R_0, are computed in step (2) above.

a	b	c	o	s	d
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	1
1	0	0	0	1	1
1	0	1	1	0	1
1	1	0	1	0	0
1	1	1	1	1	0

(a) Truth table for one-bit full adder



(b) A direct gate-level implementation of (a)

Figure 1 One bit full adder

R_1 = a'b'c'def+a'b'cde'f+abc'd'ef'

R_0 = a'bc'de'f'+a'bcd'e'f+a'bcd'ef'+ab'c'de'f'+ab'cd'e'f+ab'cd'ef'

Then we get F_1 and F_0 as follows.

F_1 = def+cdf+abe

F_0 = a'c'e'f'+a'd'e'+a'd'f'+b'c'e'f'+b'd'e'+b'd'f'

Therefore,

p_1 = def, p_2 = cdf, p_3 = abe,

q_1 = a'c'e'f', q_2 = a'd'e', q_3 = a'd'f',

q_4 = b'c'e'f', q_5 = b'd'e', q_6 = b'd'f'

The resulting s^{ij} (step (3)) are as follows:

s_{11} = e+f, s_{21} = c+f, s_{31} = a+e,

s_{12} = d+e, s_{22} = d, s_{32} = a+e,

s_{13} = d+f, s_{23} = d+f, s_{33} = a,

s_{14} = e+f, s_{24} = c+f, s_{34} = b+e,

s_{15} = d+e, s_{25} = d, s_{35} = b+e,

s_{16} = d+f, s_{26} = d+f, s_{36} = b

Carrying out step (4), and deleting repeated factors, we obtain:

H_{p1} = (e+f)(d+e)(d+f) = de+df+ef = z_1+z_2+z_3,

H_{p2} = (c+f)d(d+f) = cd+df= z_4+z_2,

H_{p3} = (a+e)a(b+e)b = ab = z_5,

R_u = (z_1+z_2+z_3)(z_4+z_2)z_5

= z_2z_5+$z_1z_4z_5$+$z_3z_4z_5$

So the positive irredundant sum-of-products forms for u are:

N_1 = df+ab (from z_2z_5),

N_2 = de+cd+ab (from $z_1z_4z_5$),

N_3 = ef+cd+ab (from $z_3z_4z_5$),

From the above, we can say the minimal support for u is a, b, d, and f which correspond to N_1.

Note that we simultaneously generate irredundant cover for each minimal support as shown above. So, we can use the numbers of product terms which are required for each minimal support as part of the cost function. Also note that although the above algorithm is for single node minimization, we can use it to minimize multiple nodes at the same time in the following way. First sets of minimal supports for each node to be minimized are computed separately by the above algorithm. Then those sets are gathered and form a minimum covering table. Solving that table corresponds to the simultaneous minimization of multiple nodes.

Also, since we formulate the problem by first expressing it as a characteristic function, we can easily incorporate don't cares (satisfiability don't cares and observability don't cares).

Another Example

Now we show another example which includes the generation of characteristic function from circuit descriptions. The example is a one-bit full adder. The truth table for the adder is shown in Figure 1 (a), where a and b are inputs, c is the carry input, o is overflow output (carry output), and s is the summation. Also, in this truth table, the logic function for another output d, which is the exclusive-or of the two inputs a and b is also shown. A direct implementation of the truth table is shown in Figure 1 (b). Here we compute the minimal support set for the output s, assuming that all other variables (including intermediate variables) shown in Figure 1 (b) are candidates for support.

In Boolean equations, the relations shown in Figure 1 (b) are expressed as:

o = x+y+z,

x = ab,

y = bc,

z = ac,

s = a'b'c'+a'bc'+ab'c'+abc,

d = ab'+a'b

These equations can be merged into one equation as the characteristic function, f(a,b,c,d,x,y,z,s,o):

f(a,b,c,d,x,y,z,s,o) =

(o=x+y+z)(x=ab)(y=bc)(z=ac)(s=a'b'c'+a'bc'+ab'c'+abc)(d=ab'+a'b)

Expanding the right hand side of the above formula, we can get the characteristic function in sum-of-products form, and we can get the set of minimal supports.

3. MINIMAL SUPPORT PROCEDURE

The algorithms presented in the previous section are combined into one procedure as the minimal support procedure shown in Figure 2. We first calculate the set of supports which consists of minimum numbers of candidates variables (line 5 ~ line 17 in Figure 2), and then among supports in the set, we compute the positive irredundant normal forms (line 18 ~ line 33 in Figure 2).

We can make several remarks about the minimal

```
1: minimal_support ()
2: {
3:     sort nodes in Boolean network according to their level from outputs
4:     foreach node u in Boolean network {
5:             R^0 = ON-set of u
6:             R^1 = OFF-set of u
7:             M1 = φ
8:             foreach cube p ∈ R^0 {
9:                     foreach cube q ∈ R^1 {
10:                            c = make_a_new_row (p,q)
11:                                    /* ci = 1 (if  pi ∩ qi == φ)        */
12:                                    /* ci = 0 (otherwise)               */
13:                            M1 = M1 ∪ {c}
14:                    }
15:            }
16:            calculate cost for each column of M1
17:            J1 = minucov(M1)
18:            (RR^0, RR^1) = eliminate literals included in J1 from R^0 and R^1
19:            Z = φ          /* pool of variable zi */
20:            M2 = φ         /* matrix represented by {zi} */
21:            foreach cube p ∈ RR^0 {
22:                    foreach cube q ∈ RR^1 {
23:                            x = variable such that p ⊂ x AND q ⊂ x'
24:                            y = variable such that p ⊂ y' AND q ⊂ y
                                      Hk = Σ_{i=1}^{n} xi + Σ_{j=1}^{m} yj'
25:
26:                    }
                               H = ∏_{k=1}^{|R'|} Hk
27:
28:                    Z = Z ∪ {terms in H represented sum-of-products }
29:                    substitute terms of H by {zi}
30:                    M2 = M2 ∪ {a cube representing H by {zi} }
31:            }
32:            J2 = minucov(M2)
33:            new expression of u = J2 represented by original variables
34:    }
35: }/*end*/
```

Figure 2. The minimal support procedure

support procedure.

- We use the minimum unate cover procedure (that is, only uncomplemented variables appear) to multiply out a product-of-sums formula into a sum-of-products formula with no duplication.

- The minimum unate cover procedure (*minucov* in line 17 and 32 of Figure 2) is the same as in (Rudel 1989). We implemented both exact and heuristic procedures in (Rudel 1989). Mostly we use the heuristic procedure because of CPU time (the heuristic procedure gives very good results).

- When executing the minimum unate cover procedure, we can attach a cost to each column. The minimum unate cover procedure reduces the total cost. If we attach the same cost to all columns, we are reducing the number of supports. On the other hand, one may want to eliminate nodes which have not yet been used as much as possible. In the case, we attach large costs to nodes which have not yet been used and attach small costs to nodes which have already been used. Using this cost function, we can avoid using nodes which have not yet been used.

- Although the procedure shown in Figure 2 minimizes each node one by one, we can modify it to minimize multiple nodes at the same time by changing the handling of **M1** in Figure 2 as follows. **M1** is made for all nodes to be minimized at the same time. The *minucov* procedure is applied to that **M1**.

4. IMPLEMENTATION AND EXPERIMENTAL RESULTS

We have implemented the method presented above. It is a preliminary implementation, and we use only limited satisfiability don't care sets just like MIS2.1 filters (Saldanha *et al* 1989). Only one node is minimized at one time (no implementation of concurrent minimization of multiple nodes, although it is not difficult). We did experiments using ISCAS sequential benchmark circuits, which are rather large circuits as logic synthesis benchmark circuits. The results are shown in Table 1. Table 1 shows the numbers of cells (nodes) after mapped to five-input look-up table type FPGAs. We used MIS's technology mapper for look-up table type FPGAs, i.e. the following commands.

```
xl_split -n 5
xl_partition -n 5
xl_cover
```

We call these as "xl_map". The second column in Table 1 shows the results when only "xl_map" is applied to the benchmark data.
The third column shows the results when
(1) "xl_map" is applied,
(2) the minimal support procedure presented in the previous section is applied,
(3) "xl_map" is applied again.

From the table, we can see an average 7% reduction of the numbers of cells from the synthesis results by the MIS commands (1), (2) and (3) above. For some circuits, we can get more than 10% reduction, which indicates that the minimal support procedure works very effectively even on the circuits generated by popular FPGA synthesis tools.

We are now implementing the procedures for the concurrent minimization of multiple nodes and to utilize more don't care sets.

5. CONCLUSIONS

We have presented a method for multi-level logic minimization which is particularly suitable for the minimization of look-up table type FPGAs. Our method guarantees that we can get the minimal support when minimizing a node. We have also presented a preliminary implementation and its synthesis results of ISCAS sequential benchmark circuits, and have shown the effectiveness of our method.

REFERENCES

Bostick, D., Hachtel, G.D., Jacoby, R.M., Lightner, M.R., Moceyunas, P., Morrison, C.R. and Ravenscrofit, D., "The boulder optimal logic design system," *Proc. ICCAD '87*, November 1987.

Brayton, R.K., Rudell, R., Sangiovanni-Vincentelli, A.L. and Wang, A.R., "MIS: A Multiple-Level Logic Optimization," *IEEE Trans. CAD*, pp.1062-1081, Nov. 1987.

Brown, F.M., "Boolean Reasoning," Kluwer Academic Publishers, 1990.

Francis, R.J., Rose, J. and Chung, K., "Chorle:A Technology Mapping Program for Lookup Table-

Based Field Programmable Gate Arrays," *IEEE/ACM 27th Design Automation Conference*, June 1990.

Fujita, M., Matsunaga, K. and Kakuda, T., "On Variable Ordering of Binary Decision Diagrams for the Application of Multi-Level Logic Synthesis," *Proc. 2nd EDAC*, Feb. 1990.

Halatsis, C. and Gaitanis, N., "Irredundant Normal Forms and Minimal Dependence Sets of a Boolean Function," *IEEE Trans. Computer*, C-27(11):1064-1068, November 1978.

Murgai, R., Nishizaki, Y., Shenoy, N., Brayton, R.K. and Sangiovanni-Vincentelli, A.L., "Logic Synthesis for Programmable Gate Arrays," *IEEE/ACM 27th Design Automation Conference*, June 1990.

Saldanha, A., Wang, A.R., Brayton, R.K. and Sangiovanni-Vincentelli, A.L., "Multi-Level Logic Simplification using Don't Cares and Filters," *Proc. 25th DAC*, June 1989.

Rudell, R.L., "Logic Synthesis for VLSI Design," Technical Report UCB/ERL M89/49, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, 1989.

Sato, H., Yasue, Y., Matsunaga, Y. and Fujita, M., "Boolean resubstitution with permissible functions and Binary Decision Diagrams," *Proc. 27th ACM/IEEE Design Automation Conference*, June 1990.

Circuit	xl_map	minsup+xl_map	CPU time for minsup (sec)
s1196	169	166	21.50
s1238	182	175	38.42
s1423	140	137	10.52
s1488	220	212	55.08
s1494	226	212	63.00
s510	90	87	8.37
s526	68	58	2.53
s526n	68	57	2.47
s641	73	71	2.15
s713	74	71	2.15
s820	126	112	9.47
s832	126	113	6.78
s838	95	95	4.40
s953	156	154	12.40
s5378	491	464	132.88
s9234	637	561	167.90
Total	2941	2745	-
Ratio	1.00	0.93	-

Table1. Minimization results of ISCAS89 benchmark circuits (Machine: Sparc2)