Scientia Iranica, Vol. 11, No. 3, pp 203–217
© Sharif University of Technology, July 2004

SCIENTIA IRANICA

Simplification of Boolean Functions Using Boolean Differences

M.B. Ghaznavi-Ghoushchi* **and A.R. Nabavi[1]**

This paper presents a new method for simplification of Boolean functions based on Boolean differences. The proposed method is applicable to various forms of Boolean functions, including truth tables and Binary Decision Diagrams (BDDs). The Boolean differences are extended to cover the truth tables with don't-care components and cutset graphs in BDDs. The results of simplification agree with Quine-McCluskey and ESPRESSO methods. Experimental tests on MCNC and Berkeley PLA benchmarks show that the proposed method gains a performance of 1.5-10 times faster than ESPRESSO. The algorithms of the proposed method are implemented in Java/Perl/C++, and a toolset for logic function simplification is developed.

INTRODUCTION

Logic synthesis is one of the most important steps in the design process of digital Integrated Circuits (ICs). There are two basic approaches for logic minimization, namely: exact and heuristic approaches. Many minimization approaches for circuit design have been proposed in [1-4]. Also, for two-level realizations, powerful tools for exact minimization have been developed [5-8]. The exact approach computes minimum cover based on the Quine-McCluskey (QM) method [2-4,9-11], which uses minterms as a starting point for minimization and requires computing all prime implicants [12,13]. Therefore, the number of minterms grows, exponentially, with the number of input variables [12]. In fact, prime implicants are, at most, $\mathcal{O}(3^n/\sqrt{n})$ and can be, at least, $\mathcal{O}(3^n/n)$, while minterms vary from 0 to 2^n [13]. Some more modern methods, including the well-known ESPRESSO [5-8], with its later improvements ESPRESSO-EXACT [5] and ESPRESSPO-SIGNATURE [4], combine the use of two basic phases known as PI generation and Covering Problem (CP) solution, reducing the number of implicants to be processed. The usual problems in conventional approaches are memory size and time (CPU Time) [4]. There has been some effort made to solve these problems [14,15]. In this paper, a new application scope of Boolean differences for simplification of Boolean functions is introduced. The proposed method can handle various formats of Boolean functions, including the truth table (complete, incomplete without don't-care, incomplete with don't-cares), various types of SOP and BDD-derived cutset graphs. The method has been tested on several different design benchmark circuits. These experiments proved that the new method has, up to an order of magnitude, speed improvement over ESPRESSO.

The remainder of the paper is organized as follows. First, the basic definitions and theorems are presented. Then, the implementation of the proposed algorithms and the theorems needed in simplification are described. After that, the experimental results are given, and, finally, the paper is concluded.

BASIC DEFINITIONS AND THEOREMS

In this section, basic definitions and theorems related to Boolean difference and the proposed simplification approach are given.

The Boolean difference of an n-variable Boolean function, $f(.) = f(\nu_1, \nu_1, \cdots, \nu_n)$, with respect to a variable v_i, is denoted as $\partial f/\partial \nu_i$ and defined as [16,17]:

$$\frac{\partial f}{\partial \nu_i} = f(\nu_i) \oplus f(\nu_i \oplus 1), \tag{1}$$

*. *Corresponding Author, Department of Electrical Engineering, Tarbiat Modarres University, P.O. Box 14115-143, Tehran, I.R. Iran.*

1. *Department of Electrical Engineering, Tarbiat Modarres University, P.O. Box 14115-143, Tehran, I.R. Iran.*

where:

$$f(\nu_i) = f(\nu_1, \nu_2, \cdots, \nu_{i-1}, 1, \nu_{i+1}, \cdots, \nu_n), \quad (2)$$

$$f(\nu_i \oplus 1) = f(\nu_1, \nu_2, \cdots, \nu_{i-1}, 0, \nu_{i+1}, \cdots, \nu_n), \quad (3)$$

and $\oplus$ denotes the exclusive-or operator. The Boolean difference represents the minterms, for which v_i is observable at f. When $\frac{\partial f}{\partial \nu_i} = 0$, then the function does not depend on v_i [17]. In this paper, this property is employed to extend the definition of Boolean difference to truth tables and binary decision diagrams.

Definitions and Theorems

Boolean Vector: For a given Boolean function:

$$f(\nu_i) = f(\nu_1, \nu_2, \cdots, \nu_{i-1}, \nu_i, \nu_{i+1}, \cdots, \nu_n). \quad (4)$$

The corresponding truth table contains $2^n \times (n+1)$ elements, where $2^n \times n$ elements are variables and $2^n \times 1$ elements are function values. Each combination of the input variables, ν_i, is called a Boolean vector and denoted here by L_i:

$$L_1 = (\nu_1, \nu_2, \cdots, \nu_{i-1}, \nu_i, \nu_{i+1}, \cdots, \nu_n)_1,$$

$$L_2 = (\nu_1, \nu_2, \cdots, \nu_{i-1}, \nu_i, \nu_{i+1}, \cdots, \nu_n)_2,$$

$$L_j = (\nu_1, \nu_2, \cdots, \nu_{i-1}, \nu_i, \nu_{i+1}, \cdots, \nu_n)_j,$$

$$L_{2^n} = (\nu_1, \nu_2, \cdots, \nu_{i-1}, \nu_i, \nu_{i+1}, \cdots, \nu_n)_{2^n}. \quad (5)$$

These sets are represented by a triplet $T_2 = \langle L, F, n \rangle$, where L is the set of Boolean vectors $L_i, 1 \leq i \leq 2^n$, F is the vector of Boolean values of f and n is the number of input variables. Note that $f(L_i) \in \{0, 1\}$.

Definition 1

Boolean Vector Complement

Given a Boolean vector $L_u = (\nu_1, \nu_2, \cdots, \nu_{i-1}, \nu_i, \nu_{i+1}, \cdots, \nu_n)_u$, the vector:

$$L_u^{C_i} = (\nu_1, \nu_2, \cdots, \nu_{i-1}, \nu_i', \nu_{i+1}, \cdots, \nu_n)_u, \quad (6)$$

obtained from L_u by negating the element i, is defined as the Boolean vector complement of L_u, with respect to ν_i. For instance, if $L_u = (1,1,0,1)$, then $L_u^{C_1} = (0,1,0,1), L_u^{C_2} = (1,0,0,1), L_u^{C_3} = (1,1,1,1)$ and $L_u^{C_3} = (1,1,0,0)$.

Definition 2

Boolean Vector Difference

The Boolean difference of a vector L_u, with respect to element i, is defined as:

$$\frac{\partial L_u}{\partial \nu_i} = f(L_u) \oplus f(L_u^{C_i}). \quad (7)$$

Theorem 1

Let $f = f(\nu_1, \nu_2, \cdots, \nu_{i-1}, \nu_i, \nu_{i+1}, \cdots, \nu_n)$ be a Boolean function with truth table $T_2 = \langle L, F, n \rangle$. When the Boolean vector difference of $L_u = f(\nu_1, \nu_2, \cdots, \nu_{i-1}, \nu_i, \nu_{i+1}, \cdots, \nu_n)_u$, with respect to ν_i, is zero, then, the truth table is independent of ν_i:

$$\frac{\partial L_u}{\partial \nu_i} = 0 \quad \Rightarrow \quad T_2 \text{ is independent of } \nu_i \text{ in row } u. \quad (8)$$

The proof for Theorem 1 is given in Appendix A.

Example 1

For a Boolean function $f = f(a,b,c) = a + b.c$, the truth table is shown in Table 1. For $u = 2$ and $i = 3$: $L_2 = (0,0,1)_2, L_2^{C_3} = (0,0,0)_2, f(L_2) = f(0,0,1)_2 = 0$ and $f(L_2^{C_3}) = f(0,0,0)_2 = 0 \Rightarrow (\frac{\partial L}{\partial \nu_i})_2 = f(L_2) \oplus f(L_2^{C_3}) = 0$. Thus, T_2 is independent of bit $c = 1$ in row 2. On the other hand, when $a = b = 0, f = 0$.

Definition 3

Incomplete Truth Table

A complete truth table with n variables has $2^n \times n$ elements. A truth table with less than $2^n \times n$ elements is called an incomplete truth table. An incomplete truth table is a truncated truth table, denoted here by $T_3 = \langle L, F_1, m \rangle; m \leq n$, where L is the set of Boolean Vectors, F_1 is the vector of Boolean values of f and m is the maximum number of input variables. The truth table is not complete, therefore, F_1 in T_3 is a subset of F in T_2.

Theorem 2

Let $T_3 = \langle L, F_1, m \rangle$; $m \leq n$ be an incomplete truth table. If the Boolean vector difference of L_u, with respect to ν_i, is zero, then, T_3 is independent of ν_i.

$$\frac{\partial L_u}{\partial \nu_i} = 0 \quad \Rightarrow \quad T_3 \text{ is independent of } \nu_i \text{ in unit } u. \quad (9)$$

Note that in T_3, despite $L_u, L_u^{C_i}$ may not exist. Therefore, the Boolean vector difference in T_3 is defined only if $L_u^{C_i}$ exists. The proof for Theorem 2 is given in Appendix A.

Table 1. Truth table of $f = a + b.c$.

Row	a	b	c	f
1	0	0	0	0
2	0	0	1	0
3	0	1	0	0
4	0	1	1	1
5	1	0	0	1
6	1	0	1	1
7	1	1	0	1
8	1	1	1	1

Definition 4

Truth Tables with Don't-Cares

Truth tables with don't-cares are denoted here by $T_X = \langle L_X, F_1, m \rangle$; $m \leq n$, where L_X is the set of Boolean vectors with don't-cares, F_1 is the vector of Boolean values of f and m is the maximum number of input variables. Since the truth table contains don't-care components, F_1 in T_X is a subset of F in T_2. A don't-care component (denoted by X) can be replaced by 1 or 0.

Definition 5

Boolean Vector Complement in T_X

A Boolean vector with don't-care is denoted by $LX_u = (s_1, s_2, \cdots, s_{i-1}, s_i, s_{i+1}, \cdots, s_n)$, where each element, s_i, is 0,1 or X (i.e. $s_i \in 0, 1, X$). The term Boolean vector complement is defined when the element s_i in row u is either 1 or 0, but not X. The Boolean vector complement (with don't-care) of LX_u for element i is $LX_u^{C_i} = (s_1^*, s_2^*, \cdots, s_{i-1}^*, s_i', s_{i+1}^*, \cdots, s_n^*)$, where $s_j^*, j = 1, 2, \cdots, m$ is defined below:

$$s_j^* = \begin{cases} s_j' & \text{if } (j = i), \\ 1 \text{ or } X & \text{if } (s_j = 1 \ \& \ j \neq i), \\ 0 \text{ or } X & \text{if } (s_j = 0 \ \& \ j \neq i), \\ 0 \text{ or } 1 \text{ or } X & \text{if } (s_j = X \ \& \ j \neq i). \end{cases} \tag{10}$$

Definition 6

Boolean Vector Complement Set

Through this definition, it is seen that $LX_u^{C_i}$ is not unique. The set of Boolean vectors, which meet the conditions of Equation 10, is called a Boolean vector complement set in this paper.

Definition 7

Boolean Vector Complement Set as a Regular Expression

For the sake of simplicity, the Boolean vector complement is denoted in a regular expression format [18]. The regular expression representation of $LX_u^{C_i}$ is:

$$LX_u^{C_i} = </re/(LX_u, i)> = \{\text{Set of all matches for } s_j^*\}, \tag{11}$$

where $/re/(LX_u, i)$ is the regular expression for the ith bit (element) in LX_u.

Example 2

For $LX_1 = (0, 0, X), LX_1^{C_1} = \langle /re/(LX_1, 1) \rangle$ and $\langle /re/(LX_1, 1) \rangle = \{$ set of all matches, such that their first element is 1, the second element is either 0 or X and the third element is $0, 1$, or $X\}$.

Definition 8

Extended-XOR

An Extended-XOR operator, denoted here by $]\widehat{\oplus}[$, is a two argument XOR with the following property:

$$[LX_{u_1}]\widehat{\oplus}[LX_{u_2}] = f(LX_{u_1}) \oplus f(LX_{u_2}). \tag{12}$$

This operator accepts two Boolean vectors as its arguments.

Definition 9

Overload-XOR

An overload-XOR operator, denoted here with $] \oplus \langle$, is a two argument XOR, which accepts a Boolean vector as its first argument and a set of Boolean vectors as its second argument. The final result of overload-XOR is zero when the XOR of individual extended-XOR is 1 and 0 otherwise.

Definition 10

Boolean Difference for LX_u

LX_u is a single Boolean vector, while $LX_u^{C_i}$ is a set of Boolean vectors. The Boolean difference for LX_u is defined as:

$$\frac{\partial LX_u}{\partial \nu_i} = [LX_u] \oplus \langle /re/(LX_u, i) \rangle. \tag{13}$$

By using Definitions 8 and 9, the Boolean difference for T_X is given by:

$$\frac{\partial LX_u}{\partial \nu_i} = [LX_u] \oplus \langle /re/(LX_u, i) \rangle,$$

$$\frac{\partial LX_4}{\partial \nu_3} = [LX_u] \oplus \{[LX_1], [LX_2], \cdots [LX_k]\}$$

$$= \bigoplus^k \{[LX_u]\widehat{\oplus}[LX_1], [LX_u]\widehat{\oplus}[LX_2], \cdots, [LX_u]\widehat{\oplus}[LX_k]\}$$

$$= \bigoplus^k \{f(LX_u) \oplus f(LX_1), f(LX_u) \oplus f(LX_2), \cdots, f(LX_u) \oplus f(LX_k)\}.$$

Note that in T_X, despite $LX_u, LX_u^{C_i}$ may not exist. Therefore, the Boolean difference in T_X can be defined only if $LX_u^{C_i}$ exists. Also, $\bigoplus^k$ denotes a multi-argument XOR operator.

Example 3

An incomplete truth table, T_X, is shown in Table 2. For variable c in $u = 4$ and $i = 3$:

Table 2. Incomplete truth table T_X.

Row	a	b	c	d	f
1	0	X	0	X	0
2	1	0	X	X	1
3	0	X	1	0	0
4	1	1	0	X	0
5	0	X	1	1	1
6	1	1	1	0	0
7	1	1	1	1	1

$$\frac{\partial LX_4}{\partial \nu_3} = [LX_4] \oplus \langle /re/(LX_4, 3)\rangle$$

$$= [110X] \oplus \langle /[1X][1X][1][01X]/\rangle$$

$$= [110X] \oplus \{[1110], [1111]\}$$

$$= \bigoplus^{2}\{[110X]\widehat{\oplus}[1110], [110X]\widehat{\oplus}[1111]\},$$

$$\frac{\partial L_4}{\partial \nu_3} = \bigoplus^{2}\{f(110X) \oplus f(1110), f(110X) \oplus f(1111)\}$$

$$= \bigoplus^{2}\{0 \oplus 0, 0 \oplus 1\} = \bigoplus^{2}\{0, 1\} = 1.$$

This shows that c for $i = 3$ and $u = 4$ cannot be removed. For variable c, $i = 3$ and $u = 3$:

$$\frac{\partial LX_3}{\partial \nu_3} = [LX_3] \oplus \langle /re/(LX_3, i3)\rangle$$

$$= [0X10] \oplus \langle /[0X][01X][0][0X]/\rangle$$

$$= [0X10] \oplus \{[0X0X], [110X]\},$$

$$\frac{\partial LX_3}{\partial \nu_3} = \bigoplus^{2}\{[0X10]\widehat{\oplus}[0X0X], [0X10]\widehat{\oplus}[110X]\}$$

$$= \bigoplus^{2}\{f(0X10) \oplus f(0X0X), f(0X10) \oplus f(110X)\}$$

$$= \bigoplus^{2}\{0 \oplus 0, 0 \oplus 0\} = \bigoplus^{2}\{0, 0\} = 0.$$

Therefore, c can be removed for $i = 3$ and $u = 3$.

Theorem 3

Suppose $T_X = \langle L_1, F_1, n\rangle$ with don't-care elements is an incomplete truth table. If the Boolean differences of LX_u and $LX_u^{C_i}$, with respect to ν_i, are zero, then, T_X is independent of ν_i.

$$\frac{\partial LX_u}{\partial \nu_i} = 0 \quad \Rightarrow \quad T_X \text{ is independent of } \nu_i \text{ in unit } u. \tag{14}$$

Proof for Theorem 3 is given in Appendix A.

Binary Decision Diagrams (BDD)

Binary Decision Diagrams (BDD) [19] are rooted directed acyclic graphs [18], which represent a canonical form of Boolean functions [20,21]. The BDD graph is first decomposed to a cutset graph, which is, then, simplified by the Boolean difference simplification algorithm.

Graph-Oriented Realization

Graph Oriented Realization (GOR) of Boolean functions is a systematic approach for the synthesis of Boolean functions [22]. In GOR, the BDD of the given Boolean function is decomposed to a new graph called a cutset graph, which is a set of all possible paths from root nodes to terminal nodes in the BDD. This cutset graph is then simplified by Logical Path Cardinal Number-Compilation (LPCN-C) rules. This step is called the LPCN-C Phase (LPCN-CP). The basic core of LPCN-CP is implemented with Boolean difference. In the simplification step, redundant nodes and cutsets are removed and a new cutset graph is obtained. The resulted graph is now ready to be employed for implementing further rules and theorems such as Mutual Merging, Minimum Span Rule, Balanced Span Rule and the Cutset Equivalency Rule. Technology mapping on the simplified graph is the last step and results in the final circuit [22].

Definition 11

Cutsets in Binary Decision Diagrams (BDD)

In a BDD, there is always one path from the root node to a terminal node. This path is called a cutset [22]. There is no duplicate node in a cutset. As mentioned above, the set of all possible paths from the root node to the terminal nodes is called a cutset graph. More detail on cutsets is presented in [22].

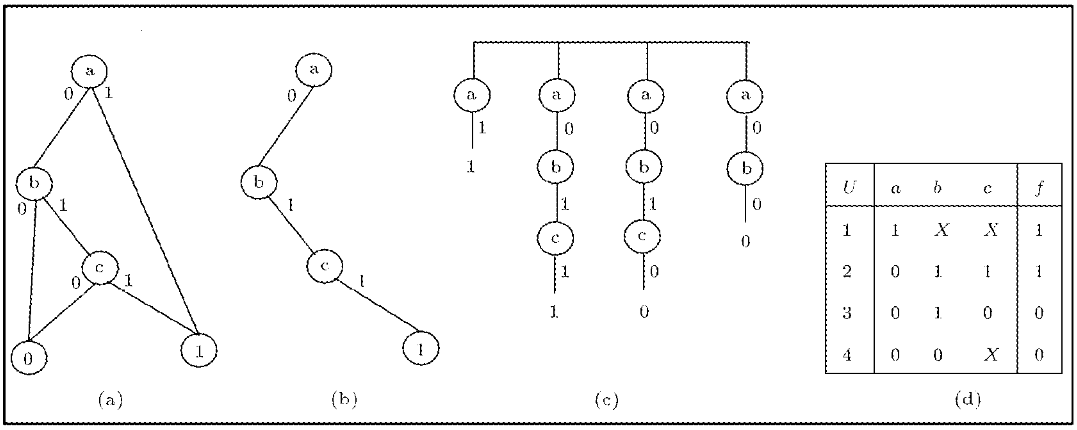
Example 4

For $f = a + b.c$, the BDD graph and cutset graph are illustrated in Figure 1.

Definition 12

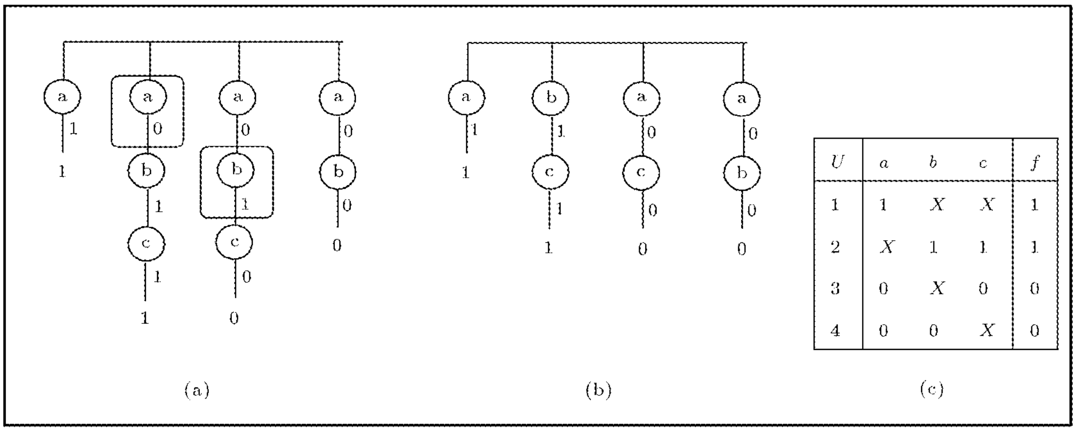
Cutset graph as a T_X

Each cutset graph has some variables and terminators that end in either logic 1 or 0. Converting the cutset graph into a table results in a T_X table, in which



U	a	b	c	f
1	1	X	X	1
2	0	1	1	1
3	0	1	0	0
4	0	0	X	0

Figure 1. BDD graph and cutsets; (a): BDD, (b): Cutset, (c): Cutset graph and (d): T_X.



U	a	b	c	f
1	1	X	X	1
2	X	1	1	1
3	0	X	0	0
4	0	0	X	0

Figure 2. Cutset reduction; (a) Removable nodes (b): Simplified cutset graph and (c): T_X.

the non-existent variables in each cutset are replaced by don't-care. The corresponding T_X in the above example is shown in part (d) of Figure 1.

Definition 13

Boolean Difference for a Cutset

Since a cutset graph corresponds to a T_X table, each cutset also corresponds to a Boolean vector, LX_u, in the T_X table. The Boolean difference for a cutset is defined as the Boolean difference for the corresponding Boolean vector, LX_u.

Theorem 4

In the cutset graph of a BDD, if the Boolean difference for a cutset, with respect to a variable ν_i, is zero, then, the corresponding cutset graph is independent of the variable ν_i. Proof for Theorem 4 is given in Appendix A.

Example 5

For the function given in Example 4, the step-by-step procedure of applying Theorem 4 for b in: $u = 4$, and $i = 2$ is given below:

$$\frac{\partial LX4}{\partial b} = [LX_4] \oplus \langle /re/(LX_4, 2)\rangle$$

$$= [00X] \oplus \langle /[0X][1][01X]/\rangle$$

$$= [00X] \oplus \{[011], [010]\}$$

$$= \bigoplus^{2}\{[00X]\widehat{\oplus}[011], [00X]\widehat{\oplus}[010]\}$$

$$= \bigoplus^{2}\{f(00X) \oplus f(011), f(00X) \oplus f(010)\}$$

$$= \bigoplus^{2}\{0 \oplus 1, 0 \oplus 0\} = \bigoplus^{2}\{1, 0\} = 1.$$

In Appendix C, Theorem 4 is applied to all inputs of the above function. The simplified cutset graph and T_X table are illustrated in Figure 2.

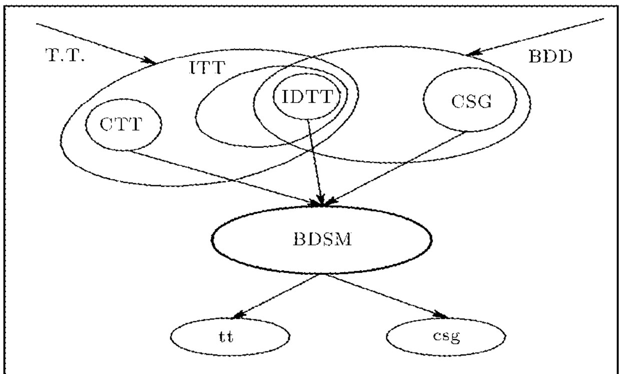
It is seen that the logical depth (maximum number of serially-connected variables) is reduced by the simplification procedure, which decreases the propagation delay in the corresponding circuit.

PROPOSED ALGORITHMS AND IMPLEMENTATIONS

The proposed algorithm on simplification of Boolean functions via Boolean differences is implemented with a combination of Perl/Java programming languages [23-25] and, finally, implemented in C++ for comparing the results on benchmarks. The core module is called the Boolean difference simplification module (BDSM). BDSM supports complete truth tables, incomplete truth tables, incomplete truth tables with don't-cares and cutset graphs. The overall block diagram for BDSM is depicted in Figure 3.

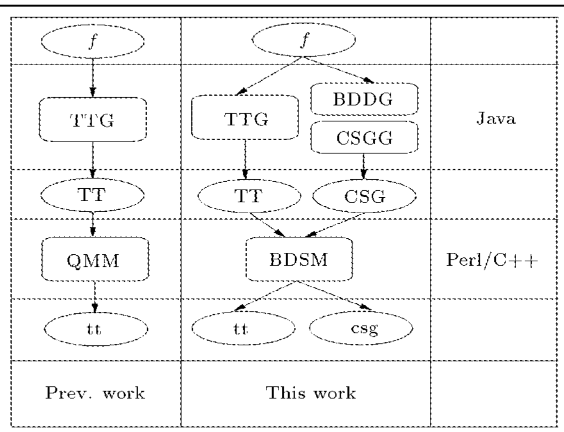
In Figure 4, the flow of the simplification procedures of Boolean functions is illustrated.

The BDD generation procedure is implemented with Java (as Java application and stand-alone executable for Win32). The BDSM modules are implemented with Perl (as Perl scripts and stand-alone executables for Win32 and Linux environments). The regular expression of Perl is directly used in the proposed algorithms. This means that each seeking string of bit patterns is first fed into a regex (regular expression) engine [17,23] and results in a regular expression string. This string is then matched with other strings by regular expression matching methods. This is the core of BDSM. Practical Extraction Reporting Language (PERL) has a comprehensive set of regular expression manipulation facilities [23]. BDSM cores are implemented in Perl using its built-in "/re/" utilities. In C++ there is a problem on implementation of /re/. First, GRETA [26] was used. GRETA is about 7 times faster than the regex library in boost and about 10 times faster than the regular expression classes in ATL7 [26]. Most regex engines are based on NFA/NDF (non-deterministic finite state automaton) with iterative execution. This execution is often done with a big, slow switch statement [26]. Also, matching regular expressions with backreferences is an NP-complete problem [26]. Therefore, two reduced methods are employed for generation of this particular regular expression and for matching the specified regular expressions (XgenRegex() and Xmatch()). The experimental results show that this makes the CPU time of the final program about 2-3 times faster.



TT: Truth table; ITT: Incomplete TT; CTT: Complete TT; IDTT: Incomplete TT with don't-cares; CSG: Cutsets graph; tt: Simplified TT; csg: Simplified CSG.

Figure 3. Block diagram of BDSM inputs.



TTG: Truth table generator; QMM: Quine-McCluskey module; BDDG: BDD generator; CSGG: Cutset graph generator.

Figure 4. Flow of operations on Boolean function simplification.

Example 6

In string "1010XX10", the regex generated string for the 4th bit ("0") is: "[1X][0X][1X][1][01X][01X][1X][0X]". This string is a regular expression and is used to match with other strings. In the simplification process, for each individual Boolean vector, the simplified bit is removed. Then, the simplification procedure is continued with the updated (simplified) vectors. In the final stage, redundant elements are removed. The particular implementations of BSM for each individual case are named by adding a suffix to BDSM. The general syntax for the resulted modules is BDSM4uuu, where uuu stands for TTT, CSG, CTT, ITT and IDTT. The sub-modules of BDSM are: BDSM4CTT, BDSM4ITT, BDSM4IDTT and BDSM4CSG. Another algorithm is also implemented for the case of SOP, where only 1-valued terms are considered. This module is also called the Boolean Difference Simplification Module for High-Term SOP (BDSM4HSOP). In the BDSM4HSOP algorithm, during the processing of a single input line, for each matched target vector, the corresponding bit is also marked as don't-care. Then, the current line is modified and the process is carried out for the remaining input vectors. the pseudo code for BDSM4CTT is presented in Table 3.

Table 3. Pseudo code for BDSM4CTT.

Procedure: BDSM4CTT

```
//Boolean Difference Simplification Method for Complete Truth Table
```

Input: $\nu = \{\nu_1, \nu_2, \cdots, \nu_n\}$, $U = \{u_1, u_2, \cdots, u_m\}$, $L = \{L_1, L_2, \cdots, L_m\}$, $m = 2^n$

Output: $\nu\alpha = \{\nu_1, \nu_2, \cdots, \nu_{n\alpha}\}$, $U\alpha = \{u_1, u_2, \cdots, u_{m\alpha}\}$, $L\alpha = \{L_1, L_2, \cdots, L_{n\alpha}\}$, $n\alpha \leq n$, $m\alpha \leq 2^n$

Processing:

```
foreach (u_i ∈ U)
   foreach (bit ∈ u_i and bit is not dont care){
      // Generate Regular expression
      re=regex(bit, u_i);
      foreach (u_j ∈ U){
         increment case;
         if (u_j =~ /re/){
            if(Lj==Li)
               increment normal;
               else
               increment inverse;
         }
      }//End of foreach
      if (case equal normal or case equal inverse)
         mask node as removable;
         replace node with "X";
   } //End of foreach
} //End of foreach, End of Processing
//End of Procedure
```

CASE STUDIES

The proposed algorithm for Boolean function simplification is examined on a set of Boolean functions. The results of the tests are summarized in the following tables. In these tables, the following notations are used:

Var.:	number of variables in the desired Boolean function,
BDD nodes:	number of nodes in the BDD graph,
H:	number of input nodes with HIGH value,
L:	number of input nodes with LOW value,
$H + L$:	total number of input nodes,
LD:	logical depth for the given Boolean function (see Appendix B),
ELD:	effective (equivalent) logical depth for the given Boolean function (see Appendix B).

Logical Depth (LD) is equal to the maximum number of serially-connected variables (nodes) in the cutset graph. Logical depth represents the signal propagation delay of the synthesized circuit. Therefore, it is used as a figure of merit.

Effective Logical Depth (ELD) is defined here as a factor to show incomplete truth tables, T_X, by an equivalent complete truth table. In this case, the effective logical depth is equal to the maximum number of serially-connected variables (nodes) of the equivalent complete truth table.

Each test is verified using the Quine-McCluskey method, based on the results obtained for $H, L, H + L$ terms and the final results of the simplification. Therefore, the "$(.)_A$" columns of Tables are the same for both the proposed algorithm and the Quine-McCluskey method. In these tables, $A(B)$ denotes after (before) simplification. The Quine-McCluskey method is an NP-complete problem [27]. However, the proposed algorithm has a fixed order of $\mathcal{O}([n_1 n_2]^2)$, where $n_1 \leq 2^n$ and $n_2 \leq n$ in T_X. Since n_1 is smaller than 2^n, the proposed algorithm has less computation complexity than the Quine-McCluskey method. The results of each

Table 4. Test results for AND gates.

f	Var/$N\#$	H_B/L_B	$(H+L)_B$	CS_B	LD_B/ELD_B	H_A/L_A	$(H+L)_A$	CS_A	LD_A/ELD_A
and_2	2/4	3/2	5	3	2/1.6667	2/2	4	3	2/1.3333
and_3	3/5	6/3	9	4	3/2.25	3/3	6	4	3/1.5
and_4	4/6	10/4	14	5	4/2.8	4/4	8	5	4/1.6
and_5	5/7	15/5	20	6	5/3.3333	5/5	10	6	5/1.6667
and_6	6/8	21/6	27	7	6/3.8571	6/6	12	7	6/1.7143
and_7	7/9	28/7	35	8	7/4.375	7/7	14	8	7/1.75
and_8	8/10	6/8	44	9	8/4.8889	8/8	16	9	8/1.7778

Table 5. Percents of reduction after simplification in AND.

#	f	$\%H$	$\%L$	$\%H+L$	$\%CS$	$\%LD$	$\%ELD$
1	and_2	33.3333	0	20.0000	0	0	20.0036
2	and_3	50	0	33.3333	0	0	33.3333
3	and_4	60	0	42.8571	0	0	42.8571
4	and_5	66.6667	0	50.0000	0	0	49.9985
5	and_6	71.4286	0	55.5555	0	0	55.5547
6	and_7	75	0	60.0000	0	0	60
7	and_8	77.7778	0	63.6363	0	0	63.636

Table 6. Test results for OR gates.

f	Var/$N\#$	H_B/L_B	$(H+L)_B$	CS_B	LD_B/ELD_B	H_A/L_A	$(H+L)_A$	CS_A	LD_A/ELD_A
or_2	2/4	2/3	5	3	2/1.6667	2/2	4	3	2/1.3333
or_3	3/5	3/6	9	4	3/2.25	3/3	6	4	3/1.5
or_4	4/6	4/10	14	5	4/2.8	4/4	8	5	4/1.6
or_5	5/7	5/15	20	6	5/3.3333	5/5	10	6	5/1.6667
or_6	6/8	6/21	27	7	6/3.8571	6/6	12	7	6/1.7143
or_7	7/9	7/28	35	8	7/4.375	7/7	14	8	7/1.75
or_8	8/10	8/36	44	9	8/4.8889	8/8	16	9	8/1.7778

test are summarized in two separate tables. The first (second) table shows the absolute values (the reduction in percent).

It is seen in Tables 4 to 7 that there is a simplification performance, but in Table 8, there is no simplification performance for XOR examples. This is mainly due to the symmetric property of these functions.

In Tables 9 to 12, there is simplification in L or H or both. In these cases, ELD is also decreased. The list of functions used for miscellaneous test cases are listed in Table 13. The experimental results of running benchmarks from MCNC and Berkeley PLA sets are presented in Table 14. All the listed benchmarks are tested with both ESPRESSO [28] and the proposed method. Both programs are compiled under Microsoft Visual C++ 6.0 and tested on a PC Windows 2000 Advanced Server with SP3 and performance tuned for application services with Intel P4 1.82GHZ, 256M-DDR and a reduced set of background services. It is seen that with the proposed algorithm, the CPU time has a performance between $1.5-10$ times better than ESPRESSO. The performance of the implemented program slowly degrades when the number of prime implicants increases. This is mainly due to the selected approach of using 2D and 3D matrices in the program. It is hoped to overcome this problem in the next versions with bitwise operations, string-based

Table 7. Simplification in OR.

#	f	$\%H$	$\%L$	$\%H+L$	$\%CS$	$\%LD$	$\%ELD$
1	or_2	0	33.3333	20.0000	0	0	20.0036
2	or_3	0	50	33.3333	0	0	33.3333
3	or_4	0	60	42.8571	0	0	42.8571
4	or_5	0	66.6667	50.0000	0	0	49.9985
5	or_6	0	71.4286	55.5555	0	0	55.5547
6	or_7	0	75	60.0000	0	0	60
7	or_8	0	77.7778	63.6363	0	0	63.636

Table 8. Test results for XOR gates.

f	Var/$N\#$	H_B/L_B	$(H+L)_B$	CS_B	LD_B/ELD_B	H_A/L_A	$(H+L)_A$	CS_A	LD_A/ELD_A
xor_2	2/5	4/4	8	4	2/2	4/4	8	4	2/2
xor_3	3/7	12/12	24	8	3/3	12/12	24	8	3/3
xor_4	4/9	32/32	64	16	4/4	32/32	64	16	4/4
xor_5	5/11	80/80	160	32	5/5	80/80	160	32	5/5
xor_6	6/13	192/192	384	64	6/6	192/192	384	64	6/6
xor_7	7/15	448/448	896	128	7/7	448/448	896	128	7/7
xor_8	8/17	1024/1024	2048	256	8/8	1024/1024	2048	256	8/8

algorithms and by converting all 3D matrices into 2D matrices.

CONCLUSIONS

In this paper, a new method for simplification of Boolean functions is presented using the well-known concept of Boolean differences and an original extension of the concept covering the Boolean vectors. This method is applicable to truth tables, BDDs, and cutset graphs. The results of the method are verified by the Quine-McCluskey method and compared to that of ESPRESSO on the MCNC and Berkeley PLA benchmarks. The experimental results show a reasonable gain in CPU time (1.5-10 times). The process of simplification with the proposed algorithms has a fixed order of $\mathcal{O}([n_1 n_2]^2)$, where $n_1 \leq 2^n$ and $n_2 \leq n$, while in the Quine-McCluskey method, the order of calculation highly depends on prime implicates. Use of the regular expression core engine of Perl for implementing the proposed method results in a compact code size, but it has some speed disadvantages. Also, using the regular expression core engine of C++ degrades the speed performance. Therefore, a simple regex generator and match finder (XgenRegex() and Xmatch()) are developed to speed up the implementation. The algo-

Table 9. Test results for AND-OR-INV gates.

f	Var/$N\#$	H_B/L_B	$(H+L)_B$	CS_B	LD_B/ELD_B	H_A/L_A	$(H+L)_A$	CS_A	LD_A/ELD_A
aoi_2x2	4/6	11/10	21	7	4/3	7/8	15	7	3/2.1429
aoi_2x3	6/8	33/21	54	13	6/4.1538	15/18	33	13	5/2.5385
aoi_2x4	8/10	74/36	110	21	8/5.2381	26/32	58	21	7/2.7619
aoi_2x5	10/12	140/55	195	31	10/6.2903	40/50	90	31	9/2.9032
aoi_2x6	12/14	237/78	315	43	12/7.3256	57/72	129	43	11/3
aoi_2x7	14/16	371/105	476	57	14/8.3509	77/98	175	57	13/3.0702
aoi_2x8	16/18	548/136	684	73	16/9.3699	100/128	228	73	15/3.1233

Table 10. Simplification in AND-OR-INV.

#	f	$\%H$	$\%L$	$\%H+L$	$\%CS$	$\%LD$	$\%ELD$
1	aoi_2x2	36.3636	20	28.5714	0	25	28.57
2	aoi_2x3	54.5455	14.2857	38.8888	0	16.6667	38.8873
3	aoi_2x4	64.8649	11.1111	47.2727	0	12.5	47.2729
4	aoi_2x5	71.4286	9.0909	53.8461	0	10	53.8464
5	aoi_2x6	75.9494	7.6923	59.0476	0	8.3333	59.0477
6	aoi_2x7	79.2453	6.6667	63.2352	0	7.1429	63.2351
7	aoi_2x8	81.7518	5.8824	66.6666	0	6.25	66.6667

Table 11. Test results for MISC functions.

f	Var/$N\#$	H_B/L_B	$(H+L)_B$	CS_B	LD_B/ELD_B	H_A/L_A	$(H+L)_A$	CS_A	LD_A/ELD_A
gf_1x8a	8/14	91/129	220	34	8/6.4706	79/95	174	34	7/5.1176
gf_1x8b	8/16	246/230	476	68	8/7	134/124	258	44	7/5.8636
gf_1x10	10/16	227/289	516	66	10/7.8182	144/185	329	57	9/5.7719
gf_1x12	12/18	531/673	1204	130	12/9.2615	264/390	654	100	10/6.54
gf_1x14	14/20	1203/1569	2772	258	14/10.7442	485/847	1332	178	11/7.4831
gf_1x16a	16/22	2675/3617	6292	514	16/12.2412	930/1875	2805	327	12/8.578
gf_1x16b	16/26	2999/4305	7304	562	16/12.9964	1990/2574	4564	461	15/9.9002
gf_1x24a	38/14	72759/104913	177672	9010	24/19.7194	37410/52762	90172	5406	23/16.68

Table 12. Simplification in MISC functions.

#	f	$\%H$	$\%L$	$\%H+L$	$\%CS$	$\%LD$	$\%ELD$
1	gf_1x8a	13.1868	26.3566	20.9090	0	12.5	20.91
2	gf_1x8b	45.5285	46.087	73.5655	35.2941	12.5	16.2343
3	gf_1x10	36.5639	35.9862	36.2403	13.6364	10	26.1735
4	gf_1x12	50.2825	42.0505	45.6810	23.0769	16.6667	29.3851
5	gf_1x14	59.6841	46.0166	51.9480	31.0078	21.4286	30.3522
6	gf_1x16a	65.2336	48.1615	55.4195	36.3813	25	29.9252
7	gf_1x16b	33.6445	40.2091	37.5136	17.9715	6.25	23.8235
8	gf_1x24a	48.5837	49.7088	49.2480	40	4.1667	15.4132

Table 13. List of functions used as MISC test functions.

Name	f	# Vars.
gf_1x8a	$a+b.c+(d\oplus e\oplus f.(g+h))'$	8
gf_1x8b	$a\oplus b\oplus c+(d\oplus e\oplus f.(g+h))'$	8
gf_1x10	$a+b.c+(d\oplus e\oplus f.(g+h))'+i.j$	10
gf_1x12	$a+b.c+(d\oplus e\oplus f.(g+h))'+i.j+k.l$	12
gf_1x14	$a+b.c+(d\oplus e\oplus f.(g+h))'+i.j+k.l+m.n$	14
gf_1x16b	$a+b.c+(d\oplus e\oplus f.(g+h))'+i+j.k+(l\oplus m\oplus n.(o+p))'$	16
gf_1x24a	$a+b.c+(d\oplus e\oplus f.(g+h))'+i+j.k+(l\oplus m\oplus n.(o+p))'+q+r.s+(t\oplus u\oplus v.(w+x))'$	24

Table 14. CPU time of ESPRESSO versus the proposed algorithm.

#	Name	$\#i$	$\#o$	$\#p$	ESPRESSO	Bdiff
1	alu1	12	8	19	<1ms	<1ms
2	alu2	10	8	91	10ms	<1ms
3	alu3	10	8	72	10ms	<1ms
4	alu4	14	8	1028	771ms	220ms
5	apex2	39	3	1035	1371ms	1131ms
6	al2	16	47	103	10ms	<1ms
7	alcom	15	38	47	10ms	1ms
8	b9	16	5	23	20ms	<1ms
9	misex2	25	18	29	<1ms	<1ms
10	misex3c	14	14	196	100ms	10ms
11	gary	15	11	214	30ms	10ms
12	dk48	15	17	148	10ms	<1ms
13	max64	9	1	46	10ms	<1ms
14	signet	39	8	124	150ms	10ms
15	shift	19	16	100	<1ms	<1ms
16	m181	15	9	430	20ms	<1ms
17	c8	28	18	79	10ms	<1ms
18	b12	15	9	43	10ms	<1ms
19	duke2	22	29	87	20ms	<1ms
20	vg2	25	8	110	10ms	<1ms
21	alupla	25	5	2144	2223ms	2103ms
22	5xp1	7	10	143	<1ms	<1ms

rithms of the proposed method are implemented with Java/Perl/C++, which can be employed for a two-level sum of product realization (PLA). Further research will be directed towards multi-level realization, multi-valued logic synthesis and code optimization for more speed improvement.

ACKNOWLEDGMENT

The authors wish to thank Mr. Mohammad Reza Tayefeh and Mr. Hassan Gholipour Khatir for their great help. The authors also wish to thank anonymous reviewers for their comments and suggestions.

REFERENCES

1. Muroga, S. "Simplification of logic expressions", chapter 25, "Expressions of logic functions", chapter 23, *The VLSI Handbook*, C. Wai-Kai, Ed., CRC Press (2000).
2. McCluskey Jr., E.L. "Minimization of Boolean functions", *Bell System Technical Journal*, **35**, pp 1417-1444 (1959).
3. Quine, W.V.O. "On cores and prime implicants of truth functions", *American Mathematics Monthly*, **66**, pp 755-760 (1959).
4. De Micheli, G., *Synthesis and Optimization of Digital Circuits*, McGraw-Hill Book Company, pp 269-342 (1994).
5. Rudell, R. and Sangiovanni-Vincentelli, A. "Espresso-MV: Algorithms for multiple-valued logic minimization", *Proc. Cust. Int. Circ. Conf.*, Portland (May 1985).
6. Rudell, R. "Multiple-valued minimization for PLA synthesis", Master's Report, University of California, Berkeley, USA (June 1986).
7. Rudell, R. and Sangiovanni-Vincentelli, A. "Exact minimization of multiple-valued functions for PLA optimization", *Int. Conf. Comp. Aid. Des.*, Santa Clara, USA (Nov. 1986).
8. Sanghavi, J. "Espresso-signature: A new exact minimizer for logic functions", Master's Report, University of California, Berkeley, USA (May 1993).
9. Pollak, V.T. "Das Vereinfachungsverfahren nach Quine-McClusky, (QMC)", URL: http://www.pollak-net.at/qmc/index.php (May 2002).
10. Chisholm, J. "Quine McClusky Boolean equation minimizer", URL: http://www.graflex.org/klotz/software/boolean-equation-minimizer.txt (2000).
11. Änderung, I. "Minimierungsmethode nach Quine-McCluskey", Institut für Technische Informatik, http://www.inf.tu-dresden.de/TU/Informatik/TeI/VLSI/Lehre/ Material/ gti-beispiele/netze/anleitung_qm.html (Feb. 1999).
12. Jacobi, R.P. "A Study of the application of binary decision diagrams in multilevel logic synthesis", MS Thesis, Chapter 1 (Introduction), http://www.cic.unb.br/docentes/jacobi/pesquisa/tese/Introduction.pdf (2002).
13. Cordone, R., Ferrandi, F., Sciuto, D. and Wolfler Calvo, R. "An efficient heuristic approach to solve the unate covering problem [logic minimisation]", *A Proceedings Design, Automation and Test in Europe Conference and Exhibition*, pp 364-371 (2000).
14. Coudert, O. "Doing two-level logic minimization 100 times faster", *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp 112-121 (1995).
15. Coudert, O. "Two-level logic minimization: An overview", *Integration*, **17**(2), pp 97-140 (Oct. 1994).
16. Thayse, A. "Boolean calculus of differences", In series of *Lecture Notes in Computer Science*, G., Hartmanian, Ed., Springer-Verlag (1981).
17. Kohavi, Z., *Switching and Finite Automata Theory*, McGraw-Hill Book Company (Reprint 14), pp 228–232, 269–270 (1991).
18. Friedl, J.E.F., *Mastering Regular Expressions, Powerful Techniques for Perl and Other Tools*, OŔeilly & Associates, Inc. (1997).

19. Minato, S. and Muroga, S. "Binary decision diagrams", *The VLSI Handbook*, Chapter 26, C. Wai-Kai, Ed., CRC Press (2000).

20. Bryant, R.E. "Graph-based algorithms for Boolean function manipulation", *IEEE Trans. Comput.*, **35**, pp 677–691 (1986).

21. Narayan, A. "BDD partitioning for formal verification and synthesis of digital systems", PhD. Thesis, University of California at Berkeley, USA (1999).

22. Ghaznavi-Ghoushchi, M.B. "GOR-graph oriented realization of Boolean functions and its applications", in *Proc. 6th Iranian Conf. on Elect. Eng.*, Tehran, Iran, pp 53-59 (May 1998).

23. Wall, L., Christiansen, T. and Orwant, J., *Programming Perl*, Ofeilly, 3rd Ed., ISBN 0-596-00027-8, pp 189-246 (2000).

24. Lewis, J. and Loftus, W., *Java Software Solutions Foundations of Program Design*, Addison-wesley, pp 20-110 (1998).

25. *The Java [Enterprise] CD Bookshelf, Set of 14 Java Books*, OŔeilly & Associates (2001).

26. Shackleford, B. et al., *Synthesis of Minimum-Cost Multilevel Logic Networks via Genetic Algorithms*, SASIMI, Kyoto (Apr. 2000).

27. Niebler, E. "GRETA: The GRETA regular expression template archive", *Microsoft Corporation*, 2003, URL:http://research.microsoft.com/projects/greta/.

28. Mishchenko, A. "ESPRESSO stand-alone program (VC++ 6.0 project)", (Windows version), URL: http://www.ee.pdx.edu/ alanmi/research/soft/softPorts.htm (2003).

APPENDIX A

Proof of Theorem 1

Rewriting L_u and $L_u^{C_j}$:

$$L_u = (\nu_1, \nu_2, \cdots, \nu_{j-1}, \nu_j, \nu_{j+1}, \cdots, \nu_n)_u,$$

$$L_u^{C_j} = (\nu_1, \nu_2, \cdots, \nu_{j-1}, \nu_j', \nu_{j+1}, \cdots, \nu_n)_u.$$

Since $T_2 = \langle L, F, n\rangle$ is a complete truth table (with 2^n elements), both L_u and $L_u^{C_j}$ belong to T_2.

Rearranging L_u and $L_u^{C_j}$: $L_u = (L_{u_1}, \nu_i), L_u^{C_j} = (L_{u_1}, \nu_i')$, where $L_{u_1} = (\nu_1, \nu_2, \cdots, \nu_{i-1}, \nu_{i+1}, \cdots, \nu_n)_u$. Since $\frac{\partial L_u}{\partial \nu_i} = 0, f(L_u) = f(L_u^{C_i}), f(L_{u1}, \nu_i)_u = f(L_{u1}, \nu_i')_u$. By rewriting the Shannon expansion theorem for ν_i in unit u, the following is obtained:

$$f = \nu_i.f(L_u, 1)_u + \nu_i'.f(L_{u_1}, 0)_u = (\nu_i + \nu_i').f(L_{u_1}, 1)_u$$

$$= f(L_{u_1}, 1)_u.$$

This shows that T_2 in row u is independent of ν_i and the proof is complete.

Proof of Theorem 2

Since both L_u and $L_u^{C_j}$ are in T_3, the proof is similar to that of Theorem 1.

Proof of Theorem 3

A truth table, T_X, with don't-cares componants is given. LX_u is a Boolean vector of binary values 1, 0 and X. The value of jth bit in LX_u is denoted by $\ell_{\nu_j} \in \{0, 1\}$.

Don't-care components can be replaced by either 1 or 0, which are used in the process of simplification. The set of $\langle /re/(LX_u, j)\rangle$ has k Boolean vectors $\{(1), (2), \cdots, (k)\}$.

The combination of L_u with $L_u^{C_j}$ gives a set of k pairs $\{(0, 1), (0, 2), \cdots, (0, k)\}$. There are two possible cases:

1. When at least one of the Boolean differences of L_u and $L_u^{C_j}$ is not zero;
2. When all Boolean vector differences of L_u and $L_u^{C_j}$ are zero.

In case 1, the Boolean vector difference for pair $(0, p)$ is assumed to be 1 and all other Boolean vector differences $(0, j), j = 1, 2, \cdots, k;\ j \neq p$ are zero. The corresponding situation is illustrated in Figure A1. In Figure A2 the sub-vectors are equal.

$$L_{0_a} = Lp_a \quad \text{and} \quad L_{0_b} = Lp_b.$$

The Shannon expansion for ν_i is:

$$f = \nu_j.f(\nu_j) + \nu_j'.f(\nu_j'),$$

$f(\nu_j)$ and $f(\nu_j')$ have different values, e.g. either (0,1) or (1,0).

$$f = \nu_j.f(\nu_j) + \nu_j'.f(\nu_j'). \tag{A1}$$

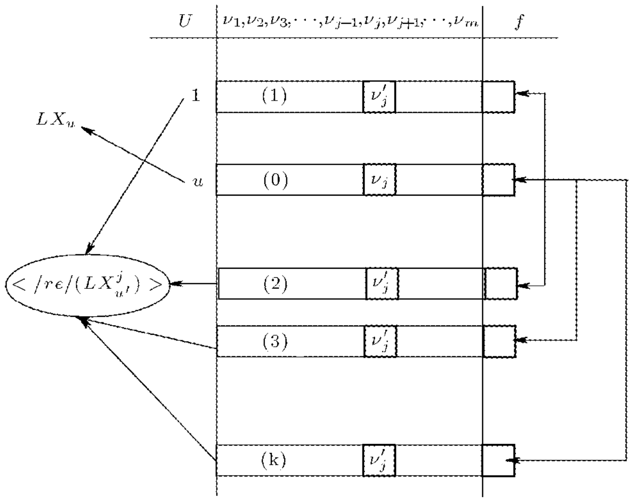


Figure A1. Set of matches for regular expression bit patterns.

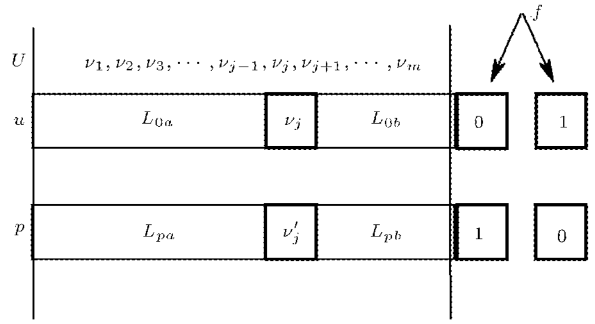


Figure A2. Boolean vector for pair (u, p).

This shows that f is not independent of ν_i and, therefore, ν_i is not removable. In case 1, all Boolean vector differences are zero. This situation is illustrated in Figure A3, where all sub-vectors are equal:

$$L_{0_a} = Lw_a \quad \text{and} \quad L_{0_b} = Lw_b,\ w = 1, 2, \cdots, k.$$

$f(\nu_j)$ and $f(\nu'_j)$ have the same values (both 1 or 0), for all possible combinations of $(0, j), j = 1, 2, \cdots, k; j \neq p$. The Shannon theorem implies that the functions f and T_X are independent of ν_j, and the proof is complete.

Proof of Theorem 4

Each cutset graph corresponds to a T_X table and each cutset corresponds to a Boolean vector in the T_X table. Therefore, the proof of Theorem 3 is valid for Theorem 4.

APPENDIX B

Logical Depth and Effective Logical Depth

For a given Boolean function $f = f(\nu_1, \nu_2, \cdots \nu_{i-1}, \nu_i, \nu_{i+1}, \cdots \nu_n)$, the corresponding complete truth table,

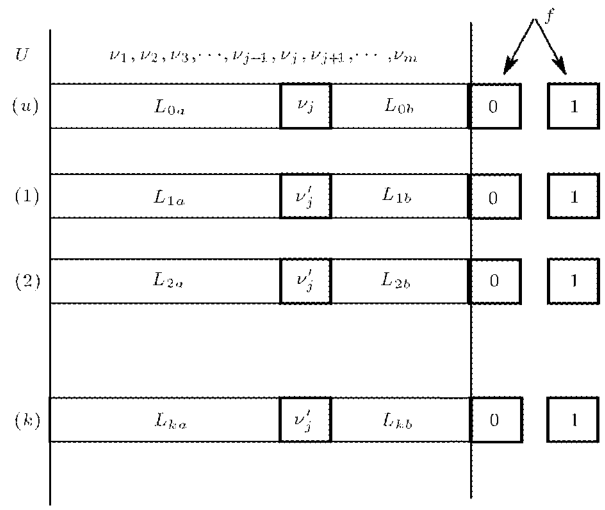


Figure A3. Set of Boolean vector when Boolean vector differences are zero.

has $2^n \times (n+1)$ elements, where $2^n \times n$ elements are variables and $2^n \times 1$ elements are function values. In the complete truth table each row has n variables. Therefore, the logical depth (i.e. the path of serially-connected variables) is n. If the truth table is assumed as a rectangle with $2^n \times 1$ as its height, then its width is n.

Incomplete truth table without don't-cares has $m^* \times (n^* + 1)$, where $m^* \leq 2^n$ and $n^* \leq n$. An incomplete truth table without don't-cares is a complete truth table with at least one row deleted.

In the incomplete truth tables without don't-cares, each row has n variables. Therefore, the logical depth (i.e., the path of serially-connected variables) is n.

An incomplete truth table with don't-cares has m^* rows, where $m^* \leq 2^n$. It is a truth table with at least one variable, which is dropped in one or more rows.

Therefore, the number of serially-connected variables in rows is not constant and varies from one row to the next. The sum of all elements in this case (truth table with don't-cares) is denoted by P1.

Effective Logical Depth (ELD) is defined here as: $ELD = \frac{P_1}{m^*}$. Figure A4 illustrates the relation of LD and ELD in truth tables. As seen in Figure A4, ELD times the number of rows makes a rectangle with a height of m* and a width of ELD.

Generally, in the process of simplification for a truth table, the simplified truth table has don't-care components. In this case, ELD is also defined. In the case of an incomplete truth table with don't-care components, ELD is used instead of LD.

LD is equivalent to the number of serially-connected variables. Therefore, in the process of synthesis, this is equivalent to the number of serially-connected transistors [22]. The less LD, the less is the propagation delay time.

APPENDIX C

Complete Steps of Applying Boolean Difference

For b in: $U = 3, i = 2$:

$$\frac{\partial LX_3}{\partial b} = [LX_3] \oplus \langle /re/(LX_3, 2) \rangle$$

$$= [010] \oplus \langle /[0X][0][0X]/ \rangle$$

$$= [010] \oplus \{[00X]\}$$

$$= \bigoplus^{1} \{[010] \widehat{\oplus} [00X]\},$$

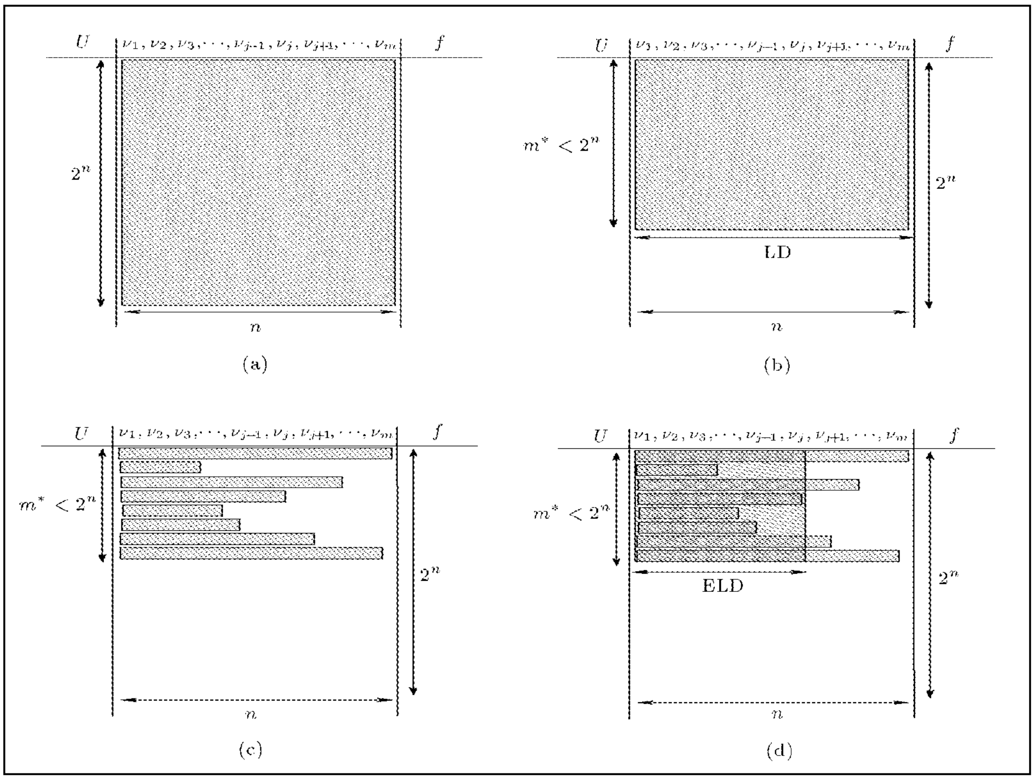


Figure A4. *ELD* interpretation; (a) Complete truth table; (b) Incomplete truth table; (c) Incomplete truth table with don't-care elements; (d) Complete truth table equivalent of part(c).

$$\frac{\partial LX_3}{\partial b} = \bigoplus^1 \{f(010) \oplus f(00X)\}$$

$$= \bigoplus^1 \{0 \oplus 0\} = \bigoplus^1 \{0\} = 0.$$

For b in: $U = 2, i = 1$:

$$\frac{\partial LX_2}{\partial a} = [LX_2] \oplus \langle /re/(LX_2, 1)\rangle$$

$$= [010] \oplus \langle /[1][1X][1X]/\rangle$$

$$= [011] \oplus \{[11X]\}$$

$$= \bigoplus^1 \{[011] \widehat{\oplus} [11X]\}$$

$$= \bigoplus^1 \{f(011) \oplus f(11X)\}$$

$$= \bigoplus^1 \{1 \oplus 1\} = \bigoplus^1 \{0\} = 0.$$

For b in: $U = 2, i = 2$:

$$\frac{\partial LX_2}{\partial b} = [LX_2] \oplus \langle /re/(LX_2, 2)\rangle$$

$$= [011] \oplus \langle /[0X][0][1X]/\rangle$$

$$= [011] \oplus \{[00X]\}$$

$$= \bigoplus^1 \{[011] \widehat{\oplus} [00X]\}$$

$$= \bigoplus^1 \{f(011) \oplus f(00X)\}$$

$$= \bigoplus^1 \{1 \oplus 0\} = \bigoplus^1 \{1\} = 1.$$

For b in: $U = 2, i = 3$:

$$\frac{\partial LX_2}{\partial c} = [LX_2] \oplus \langle /re/(LX_2, 3)\rangle$$

$$= [011] \oplus \langle /[0X][1X][0]/\rangle$$

$$= [011] \oplus \{[010]\}$$

$$= \bigoplus^1 \{[011] \widehat{\oplus} [010]\}$$

$$= \bigoplus^1 \{f(011) \oplus f(010)\}$$

$$= \bigoplus^{1}\{1 \oplus 0\} = \bigoplus^{1}\{1\} = 1.$$

For b in: $U = 3, i = 1$:

$$\frac{\partial LX_3}{\partial a} = [LX_3] \oplus \langle /re/(LX_3, 1)\rangle$$

$$= [010] \oplus \langle /[1][1X][0X]/\rangle$$

$$= [011] \oplus \{[1XX]\},$$

$$\frac{\partial LX_3}{\partial a} = \bigoplus^{1}\{[011]\widehat{\oplus}[1XX]\}$$

$$= \bigoplus^{1}\{f(011) \oplus f(1XX)\}$$

$$= \bigoplus^{1}\{0 \oplus 1\} = \bigoplus^{1}\{1\} = 1.$$

For b in: $U = 3, i = 2$:

$$\frac{\partial LX_3}{\partial b} = [LX_3] \oplus \langle /re/(LX_3, 2)\rangle$$

$$= [010] \oplus \langle /[0X][0][0X]/\rangle$$

$$= [010] \oplus \{[00X]\}$$

$$= \bigoplus^{1}\{[010]\widehat{\oplus}[00X]\}$$

$$= \bigoplus^{1}\{f(010) \oplus f(00X)\}$$

$$= \bigoplus^{1}\{0 \oplus 0\} = \bigoplus^{1}\{0\} = 0.$$

For b in: $U = 3, i = 3$:

$$\frac{\partial LX_3}{\partial c} = [LX_3] \oplus \langle /re/(LX_3, 3)\rangle$$

$$= [010] \oplus \langle /[0X][1X][1]/\rangle$$

$$= [010] \oplus \{[011]\}$$

$$= \bigoplus^{1}\{[010]\widehat{\oplus}[011]\}$$

$$= \bigoplus^{1}\{f(010) \oplus f(011)\}$$

$$= \bigoplus^{1}\{0 \oplus 1\} = \bigoplus^{1}\{1\} = 1.$$

For b in: $U = 4, i = 1$:

$$\frac{\partial LX_4}{\partial b} = [LX_4] \oplus \langle /re/(LX_4, 1)\rangle$$

$$= [00X] \oplus \langle /[1][0X][01X]/\rangle$$

$$= [00X] \oplus \{[1XX]\}$$

$$= \bigoplus^{1}\{[00X]\widehat{\oplus}[1XX]\}$$

$$= \bigoplus^{1}\{f(00X) \oplus f(1XX)\}$$

$$= \bigoplus^{1}\{0 \oplus 1\} = \bigoplus^{1}\{1\} = 1.$$

For b in: $U = 4, i = 2$:

$$\frac{\partial LX_4}{\partial b} = [LX_4] \oplus \langle /re/(LX_4, 2)\rangle$$

$$= [00X] \oplus \langle /[0X][1][01X]/\rangle,$$

$$\frac{\partial LX_4}{\partial b} = [00X] \oplus \{[011], [010]\}$$

$$= \bigoplus^{2}\{[00X]\widehat{\oplus}[011], [00X]\widehat{\oplus}[010]\}$$

$$= \bigoplus^{2}\{f(00X) \oplus f(011), f(00X) \oplus f(010)\}$$

$$= \bigoplus^{2}\{0 \oplus 1\, 0 \oplus 0\} = \bigoplus^{2}\{1, 0\} = 1.$$