Abstract: Multiple-valued input binary function is a mapping $F : X \times P \rightarrow B$, where $i = 1$ $P_i = \{0, 1, ..., p_i-1\}$ and $B = \{0, 1\}$. In this paper, tautology checking methods for sum-of-products expressions of multiple-valued input binary functions are discussed. Firstly, methods for decomposing a tautology problem into smaller ones are shown. Secondly, a fast hardware tautology checker is proposed. The computation time of the hardware tautology checker is proportional to the number of the terms in the expression. The hardware tautology checker for $n$-variable $p$-valued input binary functions requires $4pn$ copies of 2-input AND gates. Finally, applications of the tautology checker for generating prime implicants, for generating irredundant sum-of-products expressions, and for detection of the essential prime implicants are shown.

I. INTRODUCTION

In logic design, minimization or simplification of logical expression is important. A minimal sum-of-products expression for a two-valued input logic function corresponds to a minimal two-level AND-OR circuit or a minimal two-level PLA (Programmable Logic Array)\cite{11}. Similarly, a minimal sum-of-products expression for a four-valued input two-valued output function (also called a four-valued input binary function) corresponds to a minimal PLA with two-bit decoders\cite{22},\cite{33},\cite{44}. Suppose that we have to minimize $n$-variable two-valued logic function.

When $n \leq 10$, we can often obtain a minimum solution by using a Quine-McCluskey method. But this method is impractical for large problems because it needs all the prime implicants and their covering table. The average number of prime implicants is greater than $2^{n-1}$ when $n \geq 10$, and some class of functions have prime implicants proportional to $3^n/n$ \cite{44}. When $n \leq 16$, several algorithms which use the truth table of a given function are reported\cite{11}-\cite{13}. They obtain good solutions in a relatively short time.

Although these algorithms need not generate all the prime implicants at a time, they need memory space which is proportional to $2^n$. So these algorithms are also impractical for larger problems.

When $n \leq 40$, we have very good algorithms such as MINI and ESPRESSO which first obtains a complement of a given function\cite{22},\cite{55}. In these algorithms, the complement is effectively used to make the implicants prime or near prime. These algorithms often simplifies larger practical problems. However, recently, we found a class of functions whose size of the complement increases exponentially with the number of products in an original expression\cite{66}. For this class of functions both MINI and ESPRESSO failed to obtain the complement of the function even if sophisticated algorithms\cite{99},\cite{144} were used.

There are minimization methods without using the list of all the prime implicants, the truth table, nor the complement of the given function\cite{55},\cite{155}-\cite{188}. However, these methods require much computation time if we need a near minimum solution: we have to check whether an array $F$ covers a cube $c$ or not thousands of times. Most computation time is spend for the checking of this implication relation $c < F$. As will be shown in Section 5, $c < F$ holds if and only if the restriction of $F$ to $c$ is tautology. Therefore, a high speed tautology checker is useful for testing implication relations and thus, useful for logic minimization of many-variable problems. It is known that the implication relation $c < F$ can be examined by computing $c \oplus F$ \cite{166,177} or $c \odot F$ \cite{22}. But the tautology checking of $F(lc)$ is much faster.

In Section 2, the tautology problem of the binary functions is formally defined. The problem is Co-NP complete and there is virtually no hope to find a polynomial time algorithm to decide whether a given sum-of-products expression is tautology or not.

In Section 3, methods for decomposing a tautology problem into smaller ones are shown. Necessary conditions to be tautology are also given, which are useful for quick non-tautology detection.
In Section 4, a fast hardware tautology checker is introduced. The time to decide whether a given sum-of-products expression is tautology or not is proportional to the number of terms in the expression. The tautology checker for an n-variable p-valued input binary function requires 4\cdot p^n copies of 2-input AND gates. We can realize an efficient tautology checking system by combining the methods shown in Sections 3 and 4.

In Section 5, application of the tautology checking system for generating prime implicants, for generating irredundant sum-of-products expressions, and for detection of the essential prime implicants are shown. Experiments in [4] show that more than a half of the prime implicants in minimum sum-of-products expressions are essential for control circuits of microprocessors. Therefore, we often obtain good solutions quickly by first detecting all the essential prime implicants. Theorem 5.3 gives an efficient essential prime implicants detection method without generating all the prime implicants at once. Our experiments show that this method is much faster than local extraction algorithm[10].

II. BINARY FUNCTIONS AND TAUTOLOGY PROBLEM

Definition 2.1: A mapping \( \prod_{i=1}^{n} P_i \to B \) is called a multiple-valued input binary function (also called a multi-valued input two-valued output function), where \( P_i = \{0,1,\ldots,p_i-1\} \), and \( B = \{0,1\} \).

Definition 2.2: Let \( X_i \) be a variable on \( P_i \). \( X_i \) is a literal of \( X_i \), where \( S_i \subseteq P_i \). \( X_i \) represents a function such that \( X_i^{S_i} = 0 \) if \( X_i \not\in S_i \), and \( =1 \) if \( X_i \in S_i \).

Definition 2.3: A product of literals \( X_1 \cdot X_2 \cdot \ldots \cdot X_n \) is called a product. A sum of products is called a sum-of-products expression.

Theorem 2.1: An arbitrary binary function can be represented by a sum-of-products expression:
\[
\mathcal{F}(x_1, x_2, \ldots, x_n) = (s_1 \cdot s_2 \cdot \ldots \cdot s_n) x_1 \cdot x_2 \cdot \ldots \cdot x_n,
\] where \( S_i \subseteq P_i \).

From here, \( F \) (an upper case letter) represents a multiple-valued input binary function and \( \mathcal{F} \) (a script letter) represents its expression, and so on.

Example 2.1: A binary function \( F : \{0,1\} \times \{0,1\} \to \{0,1\} \) shown in Table 2.1 can be represented by the sum-of-products expression:
\[
\mathcal{F} = x_1^0 \cdot (x_2^0 \cdot x_1^1 \cdot x_2^1 \cdot x_1^0 \cdot x_2^0)\]
For simplicity, \( x_i^0 \) is represented by \( x_i^0 \) and so on.

Table 2.1 Truth table
<table>
<thead>
<tr>
<th>( X_1 )</th>
<th>( X_2 )</th>
<th>( F )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Example 2.2: An array for the expression in Example 2.1 is
\[
\begin{align*}
\mathcal{F} &= [10, 10] \\
               &=[01, 10] \\
               &=[11, 10]
\end{align*}
\]

Definition 2.4: (Positional cube notation). A product \( X_1 \cdot X_2 \cdot \ldots \cdot X_n \) can be represented as follows:
\[
x_1 \cdot x_2 \cdot x_n \cdot 1 \text{-} 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1
\] where \( c_i = 0 \) if \( i \in S_j \) and \( =1 \) if \( i \not\in S_j \).

The above notation is called a positional cube notation[7]. A sum-of-products expression represented by a set of positional cubes is called an array. Arrays are denoted by the same symbols as the corresponding expressions.

Example 2.2: An array for the expression in Example 2.1 is
\[
\begin{align*}
\mathcal{F} &= [10, 10] \\
               &=[01, 10] \\
               &=[11, 10]
\end{align*}
\]

Definition 2.5: Let \( \mathcal{F} \) be a sum-of-products expression. If \( \mathcal{F} \equiv 1 \), i.e., \( \mathcal{F} \) is equal to 1 for all the input combinations, then \( \mathcal{F} \) is said to be tautology. The problem to decide a given sum-of-products expression is tautology or not is said to be a tautology problem.

Example 2.3: A function in Example 2.1 is not tautology, which is obvious from Table 2.1.

Example 2.4: Consider a function \( F : \{0,1\} \times \{0,1,2,3\} \to B \), and its expression:
\[
\mathcal{F} = x_1^0 \cdot x_2^0 \cdot x_3^0 \cdot x_1^1 \cdot x_2^0 \cdot x_3^1 \cdot x_1^0 \cdot x_2^1 \cdot x_3^1 \]
By making a truth table for \( \mathcal{F} \), it is easy to verify that \( \mathcal{F} \) is tautology.

(End of Example).
Lemma 3.2: Let be an array, and c_i (i=1,2,...,m) be cubes, where m \( \bigvee_{i=1}^{m} c_i \equiv 1 \) and \( c_i \cdot c_j = 0 \) (i \( \not= \) j).

Then, \( \mathcal{F} \equiv 1 \Rightarrow \mathcal{F} (l_{c_i}) \equiv 1 \) (i=1,2,...,m).

(Proof) Obvious from Definition 2.5 and Lemma 3.1. (Q.E.D.)

Lemma 3.3: Let \( \mathcal{F} \) be an array, and \( c_i (i=1,2,...,m) \) be cubes, where \( m \bigvee_{i=1}^{m} c_i \equiv 1 \) and \( c_i \cdot c_j = 0 \) (i \( \not= \) j).

Then, \( \mathcal{F} \equiv 1 \Rightarrow \mathcal{F} (l_{c_i}) \equiv 1 \) (i=1,2,...,m).

(Proof) Obvious from Definition 2.5 and Lemma 3.1. (Q.E.D.)

Theorem 2.2: The tautology problem for binary function is Co-NP complete.

(Proof) Similar to the two-valued logic function. (Q.E.D.)

The above theorem implies that there is virtually no hope to find a polynomial time algorithm for the tautology problem.

III. DECOMPOSITION OF A TAUTOLOGY PROBLEM

Definition 3.1: Let c be a cube. The cube restriction of \( \mathcal{F} \) to c is obtained as follows and denoted by \( \mathcal{F} (l_c) \).

1) Make a logical product of \( \mathcal{F} \) and c.
2) Delete null products.
3) Change 0's of \( \mathcal{F} \) into 1's in the columns where c is 0.

Example 3.1: Consider an array \( \mathcal{F} \) and a cube c, where

\[
c = \begin{bmatrix}
000 & 111 & 1000 \\
010 & 111 & 0100 \\
000 & 111 & 0100 \\
010 & 111 & 1000 \\
011 & 011 & 0111 \\
010 & 111 & 0110 \\
011 & 111 & 0110 \\
011 & 111 & 0110
\end{bmatrix}
\]

We can make \( \mathcal{F} (l_{c}) \) as follows:
1) By making a logical product of \( \mathcal{F} \) and c, we have
\[
c \cdot \mathcal{F} = \begin{bmatrix}
000 & 111 & 0000 \\
010 & 111 & 0100 \\
000 & 111 & 0100 \\
010 & 111 & 1010 \\
011 & 011 & 0111 \\
010 & 111 & 0110 \\
011 & 111 & 0110 \\
011 & 111 & 0110
\end{bmatrix}
\]
2) By deleting the null cube (denoted by *), we have
\[
c \cdot \mathcal{F} = \begin{bmatrix}
000 & 111 & 0000 \\
010 & 111 & 0100 \\
000 & 111 & 0100 \\
010 & 111 & 1010 \\
011 & 011 & 0111 \\
010 & 111 & 0110 \\
011 & 111 & 0110 \\
011 & 111 & 0110
\end{bmatrix}
\]
3) By changing 0's in \( \mathcal{F} \) into 1's in the columns where c is 0 (denoted by *), we have,
\[
\mathcal{F} (l_{c}) = \begin{bmatrix}
110 & 010 & 1111 \\
110 & 010 & 1111 \\
110 & 010 & 1111 \\
110 & 010 & 1111 \\
110 & 010 & 1111 \\
110 & 010 & 1111 \\
110 & 010 & 1111 \\
110 & 010 & 1111
\end{bmatrix}
\]

(End of example).

Lemma 3.1: \( \mathcal{F} = c \cdot \mathcal{F} (l_{c}) \).

(Proof) Clear from Definition 3.1. (Q.E.D.)

Lemma 3.2: Let \( \mathcal{F} \) be an array, and \( c_i (i=1,2,...,m) \) be cubes, where

\[
\bigvee_{i=1}^{m} c_i \equiv 1 \quad \text{and} \quad c_i \cdot c_j = 0 \quad (i \not= j).
\]

Then, \( \mathcal{F} = \bigvee_{i=1}^{m} c_i \cdot \mathcal{F} (l_{c_i}) \).

(Proof)
\[
\mathcal{F} = (\bigvee_{i=1}^{m} c_i) \cdot \mathcal{F} = \bigvee_{i=1}^{m} (c_i \cdot \mathcal{F}) \quad \text{by Lemma 3.1},
\]
we have \( \mathcal{F} = \bigvee_{i=1}^{m} c_i \cdot \mathcal{F} (l_{c_i}) \). (Q.E.D.)
Example 3.2: Consider an array $\mathcal{F}$:

$$
\mathcal{F} = \begin{bmatrix}
111 & 1010 & 1101 \\
1000 & 1100 & 1111 \\
111 & 1010 & 1101 \\
1000 & 1100 & 1111
\end{bmatrix}
$$

By deleting columns with all 1's (denoted by *'s), we have

$$
\mathcal{G}_1 = \begin{bmatrix}
111 & 1010 & 1101 \\
000 & 1100 & 1111 \\
111 & 1010 & 1101 \\
000 & 1100 & 1111
\end{bmatrix}
$$

By deleting cubes with all-0 variables (denoted by under lines), we have

$$
\mathcal{G}_2 = \begin{bmatrix}
011 & 010 & 101 \\
011 & 010 & 101 \\
111 & 110 & 110
\end{bmatrix}
$$

By deleting cubes with all-0 variables (denoted by under lines), we have

$$
\mathcal{G}_3 = \begin{bmatrix}
011 & 010 & 101 \\
011 & 010 & 101 \\
111 & 110 & 110
\end{bmatrix}
$$

By deleting cubes with all-0 variables (denoted by under lines), we have

$$
\mathcal{G}_4 = \begin{bmatrix}
111 & 010 & 101 \\
111 & 010 & 101 \\
111 & 010 & 101
\end{bmatrix}
$$

By Lemma 3.5, $\mathcal{G}_4$ is non-tautology. Hence, $\mathcal{F}$ is non-tautology. (End of example)

Theorem 3.2: (Split-by-variable decomposition)

$\mathcal{F} \equiv 1 \rightarrow \mathcal{G}_1 \equiv 1$ (i=0,1,2,...,p-1),

where $\mathcal{F}_1 = \mathcal{F}(1|X_k)$.

(Proof) Let $c_i = x_i$ (i=0,1,2,...,p-1).

By Lemma 3.3, we have the theorem. (Q.E.D.)

Example 3.3: Consider an array $\mathcal{F}$:

$$
\mathcal{F} = \begin{bmatrix}
10 & 101 & 1011 \\
10 & 101 & 1011 \\
10 & 101 & 1011 \\
10 & 101 & 1011
\end{bmatrix}
$$

In Theorem 3.2, let k=1, then

$$
\mathcal{G}_0 = \begin{bmatrix}
11 & 101 & 1011 \\
11 & 101 & 1011 \\
11 & 101 & 1011 \\
11 & 101 & 1011
\end{bmatrix}
$$

and

$$
\mathcal{G}_1 = \begin{bmatrix}
11 & 101 & 1011 \\
11 & 101 & 1011 \\
11 & 101 & 1011 \\
11 & 101 & 1011
\end{bmatrix}
$$

Therefore,

$\mathcal{F} \equiv 1 \rightarrow (\mathcal{G}_0 \equiv 1$ and $\mathcal{F}_1 \equiv 1$).

(End of Example).

In Example 3.3, every cube of $\mathcal{F}$ has singleton 1 in $X_1$. It is easy to see that when every cube has singleton 1 in a same variable, Theorem 3.2 is especially effective. However, when many elements are 1's in all the variables, Theorem 3.2 is not so effective. This theorem has been discussed for the two-valued logic functions.[9].

Theorem 3.3: (Split-by-term decomposition). Suppose that the given expression can be written as $\mathcal{F} = \mathcal{G} \vee \mathcal{H}$, where $c=\prod_{i=1}^{m} X_i$. Then, $\mathcal{G} \equiv 1 \rightarrow \mathcal{G}(1|c_1) \equiv 1$ (i=1,2,...,m).

By Lemma 3.3, $\mathcal{G} \equiv 1 \rightarrow \mathcal{G}(1|c_1) \equiv 1$ (i=1,2,...,m). Hence, we have $\mathcal{F} \equiv 1 \rightarrow \mathcal{G}(1|c_1) \equiv 1 (i=1,2,...,m)$. (Q.E.D.)

Example 3.4: Consider an array $\mathcal{F}$:

$$
\mathcal{F} = \begin{bmatrix}
01 & 001 & 111 \\
01 & 101 & 1011 \\
11 & 011 & 1111 \\
11 & 110 & 1111
\end{bmatrix}
$$

By Theorem 3.3, $\mathcal{G} \equiv 1 \rightarrow (\mathcal{G}(1|c_1) \equiv 1$ and $\mathcal{G}(1|c_2) \equiv 1$), where $c_1 = X_1 \cdot x_2$, $c_2 = X_1 \cdot x_2$.

(Proof) Let $c\equiv c_{m+1}$. Then $c_{m+1} = 0$. By Lemma 3.3, $\mathcal{G} \equiv 1 \rightarrow \mathcal{G}(1|c_1) \equiv 1$ (i=1,2,...,m+1). Because $\mathcal{F} = \mathcal{G} \vee \mathcal{H}$, where $c=\prod_{i=1}^{m+1} X_i$. Hence, we have $\mathcal{F} \equiv 1 \rightarrow \mathcal{G}(1|c_1) \equiv 1 (i=1,2,...,m)$. (Q.E.D.)

As easily seen from Example 3.4, Theorem 3.3 is effective when the given expression has a cube with a small number of literals.

Definition 3.2: The volume of a cube $c=\prod_{i=1}^{n} X_i$ is defined as

$S_1 \cdot S_2 \cdot \ldots \cdot S_n$.
vol(c) = \sum_{i=1}^{n} |S_i| \text{, where } |S_i| \text{ denotes the number of elements in } S_i. \text{ In other words, } vol(c) \text{ is the number of minterms contained in } c.

Theorem 3.4: Let \( \mathcal{F} = \bigvee_{i=1}^{m} c_i \).

If \( \mathcal{F} \equiv 1 \) then \( \sum_{i=1}^{m} \text{vol}(c_i) \geq \prod_{i=1}^{n} p_i. \)

The above theorem says that, if the sum of the volume of each cube is smaller than the number of minterms in the universal cube, then \( \mathcal{F} \) is non-tautology. This theorem is useful for quick non-tautology detection.

Example 3.5: Consider an array \( \mathcal{F} \):

\[
\mathcal{F} = \begin{bmatrix}
10 & 01 & 101 \\
01 & 01 & 01 \\
10 & 01 & 001 \\
11 & 01 & 010
\end{bmatrix}
\]

It is easy to see that \( \text{vol}(c_1) = 4 \), \( \text{vol}(c_2) = 4 \), \( \text{vol}(c_3) = 3 \), and \( \text{vol}(c_4) = 6 \).

We have \( \sum_{i=1}^{n} \text{vol}(c_i) = 17 \). On the other hand

\[ \prod_{i=1}^{n} p_i = 2 \times 3 \times 3 = 18. \]

By Theorem 3.4, \( \mathcal{F} \) is non-tautology. (End of example)

IV. HARDWARE TAUTOLOGY CHECKER

As shown in the previous section, the tautology problem can be solved by an iterative applications of Theorems 3.1, 3.2, and 3.3. Another method to solve the tautology problem is to make a truth table of the array. The tautology checking by using a truth table is, in general, time consuming because we have to check \( \prod_{i=1}^{n} p_i \) combinations.

However, when the problem is small (say \( n \leq 7 \) and \( p_i = 2 \) (\( i = 1, 2, ..., n \))), tautology checking by the truth table is faster than by decomposition. Therefore, the tautology problem can be solved efficiently first by decomposing the problem into small ones, and then by making the truth tables of the small ones.

In this section, we will show a fast tautology checking method by using a special logic circuit. The computation time for this tautology checker is proportional to the number of the terms in the expression.

4.1 Tautology Checking Circuit

Schematic diagram of the hardware tautology checker is shown in Fig. 4.1.

INPUT
Minterm Generator
Latch
AND gate

\[\text{Fig. 4.1 Tautology Checking Circuit}\]

Example 4.1

Minterm Generator has \( H = \sum_{i=1}^{n} p_i \) inputs and \( W = \prod_{i=1}^{n} p_i \) outputs. Each output corresponds to a minterm. When a cube \( c \) is applied to the input, all the outputs which correspond to the minterms of \( c \) become one.

Latch Part consists of \( W \) latches. Every latch is reset to zero at the initial state. When a cube is applied to the minterm generator, all the latches which correspond to the minterms of \( c \) will be set to one.

AND Gate Part has \( W \) inputs and one output. The output becomes one if all the \( W \) inputs are one, which shows that the given array is tautology.
Example 4.1: Consider a binary function:
\[ F: C_0,1 \times C_0,1,2 \rightarrow \{0,1\}, \]
and its array
\[
\begin{array}{c|c|c|c|c|c|c|c|c}
X_1 & X_2 & X_1 & X_2 & X_1 & X_2 & c_1 & c_2 & c_3 & c_4 \\
0 & 0 & f_0 & X & X & 0 & 0 & 1 & 0 & 1 \\
1 & 0 & f_1 & X & X & 0 & 1 & 1 & 1 & 0 \\
0 & 1 & f_2 & X & X & 0 & 1 & 1 & 1 & 0 \\
1 & 0 & f_3 & X & X & 1 & 0 & 1 & 0 & 1 \\
0 & 1 & f_4 & X & X & 1 & 0 & 1 & 0 & 1 \\
1 & 1 & f_5 & X & X & 1 & 0 & 1 & 0 & 1 \\
\end{array}
\]

The tautology checking circuit for this function is shown in Fig.4.2.

At the initial state, all the latches are reset to zero.

When cube \( c_1 \) is applied, \( f_0 \) and \( f_2 \) are set to one.
When cube \( c_2 \) is applied, \( f_0 \) and \( f_1 \) are set to one.
When cube \( c_3 \) is applied, \( f_3 \) and \( f_4 \) are set to one.
When cube \( c_4 \) is applied, \( f_1, f_2, f_4, \) and \( f_5 \) are set to one.

In the following table, \( X \) marks show the latches which are set by the application of each cube.

<table>
<thead>
<tr>
<th>( X_1 \times X_2 )</th>
<th>( c_1 )</th>
<th>( c_2 )</th>
<th>( c_3 )</th>
<th>( c_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>( f_0 )</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1 0</td>
<td>( f_1 )</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0 1</td>
<td>( f_2 )</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1 0</td>
<td>( f_3 )</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0 1</td>
<td>( f_4 )</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1 1</td>
<td>( f_5 )</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

When all the cubes are applied to Fig.4.2, the output of the AND gate will be one, which shows the given array \( \mathcal{G} \) is tautology.

(End of example).

4.2 Estimation of the Gate Counts

In this part, we assume that \( p=p_i \ (i=1, 2, \ldots, n) \) and \( n=2^m \), where \( m \) is an integer.

Minterm Generator: Let \( g(n) \) be the number of the 2-input AND gates to realize an n-variable minterm generator. A 2-variable minterm generator is realized by using \( p^2 \) copies of 2-input AND gate. So we have \( g(2)=p^2 \). An n-variable minterm generator is realized by using two \( (n/2) \)-variable minterm generators and \( p^n \) copies of 2-input AND gates. From these, we have,
\[
g(n)=2^n g(n/2)+p^n+2^n p(n/2)+4\cdot p(n/4)+\ldots+(n/2)\cdot p^2
\]

Latch Part: Each latch is realized by a pair of 2-input AND gates. So the number of AND gates is \( 2\cdot p^n \).

AND Gate Part: \( p^n \) input AND gate is realized by \( (p^n-1) \) copies of 2-input AND gates.

Hence, the total number of 2-input AND gates is
\[
4\cdot p^{n+2} \cdot p(n/2)+4\cdot p(n/4)+\ldots+(n/2)\cdot p^2-1.
\]
Table 4.1 shows the number of 2-input AND gates necessary to realize a tautology checker when \( p=2 \).

<table>
<thead>
<tr>
<th>Number of Inputs ( n )</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minterm Generator</td>
<td>88</td>
<td>304</td>
<td>1120</td>
<td>4272</td>
<td>16712</td>
</tr>
<tr>
<td>Latch part</td>
<td>128</td>
<td>512</td>
<td>2048</td>
<td>8192</td>
<td>32768</td>
</tr>
<tr>
<td>AND gate part</td>
<td>63</td>
<td>255</td>
<td>1023</td>
<td>4095</td>
<td>16383</td>
</tr>
<tr>
<td>Total</td>
<td>279</td>
<td>1071</td>
<td>4191</td>
<td>16559</td>
<td>65863</td>
</tr>
</tbody>
</table>

4.3 Estimation of the Computation Time

Suppose that the bit pattern of each cube is sent to the output port of the computer by an output instruction, and that tautology is checked by the logic circuit shown in Fig.4.1. The circuit can be realized in at most \( [n\log_2 p+2+\log_2 n] \) levels. When \( n=10 \) and \( p=2 \), it is 16 levels, and the delay time of the circuit is in the order of nano-seconds if we use TTL technology. On the other hand, the time to send a bit pattern of a cube to the output port of a computer is in the order of micro-seconds. So, the total computation time depends only on the output instruction time, and is \( m\cdot t_1 \), where \( m \) is the number of the cubes and \( t_1 \) is the time to send the bit pattern of a cube to the output port.

V. APPLICATION FOR LOGIC MINIMIZATION

Minimization methods in this section may be slower than MINI or ESPRESSO if we use only a software for tautology checking. But if we make a system which first decomposes a large tautology problem into many small problems, and then use a high-speed hardware tautology checker to solve the small problems, we can make a fast minimization system for many input problems.

Definition 5.1: A product \( P=X_1 \cdot X_2 \cdot \ldots \cdot X_n \) is called an implicant of \( F \) if \( F \) is equal to one whenever \( P \) is equal to one, and denoted by \( P \prec F \). \( P \) is called a prime implicant of \( F \) if \( P \prec F \) and \( S_i \ (i=1,2,\ldots,n) \) is maximal. When \( P \prec F \), \( F \) is said to cover \( P \).
From here, we will consider a problem to represent a given function $f$ by using a minimum (or minimal) number of prime implicants.

Lemma 5.1: Let $c$ be a cube and $f$ be an array. Then $c \in f \rightarrow f(c) \equiv 1$.

(Proof) By Lemma 3.1, $c \in f \rightarrow f(c) \equiv 1$. Note that $c \in f \rightarrow c \in f$. If $c \in f$ then $c \in c \in f$. Therefore $f(c) \equiv 1$. If $f(c) \equiv 1$, then $c \in c \in f$. Therefore $c \in f$.

(Q.E.D.)

Example 5.1: Let $c$ and $f$ be a cube and an array as follows:

$c = \begin{bmatrix} 11-111-0110 \\ 01-011-0110 \\ 01-101-1110 \\ 10-101-0110 \\ 10-011-1110 \end{bmatrix}$

By Theorem 3.1, $s(c) \in f$. Therefore $c \in f$. By Lemma 5.1, we have $c \in f$.

(End of Example).

Lemma 5.1 shows that whether an array $f$ covers a cube $c$ or not is examined by the tautology checking.

Theorems 5.1 through 5.3 show methods for generating prime implicants, for deriving irredundant sum-of-products expressions, and for detecting the essential prime implicants. In these methods, we have to examine whether $c \in f$ or not thousands of times, which can be done efficiently by the tautology checking of $f(c)$.

Theorem 5.1: Let $f = c \lor f$, where

$c = \begin{bmatrix} S_1 \ S_2 \ \ldots \ S_k \ \ldots \ S_n \end{bmatrix}$

$\begin{bmatrix} X_1 \ X_2 \ \ldots \ X_k \ \ldots \ X_n \end{bmatrix}$

$c = \begin{bmatrix} c_1 \ c_2 \ \ldots \ c_k \ \ldots \ c_n \end{bmatrix}$

If $b \subset c$, then $f = c \lor f$, where

$b = \begin{bmatrix} S_1 \ S_2 \ \ldots \ S_k \ \ldots \ S_n \end{bmatrix}$

$\begin{bmatrix} a_1 \ a_2 \ \ldots \ a_k \ \ldots \ a_n \end{bmatrix}$

1) If $b \subset c$, then $f = c \lor f$, where

$c_1 = \begin{bmatrix} S_1 \ S_2 \ \ldots \ S_k \ \ldots \ S_n \end{bmatrix}$

$\begin{bmatrix} c_1 \ c_2 \ \ldots \ c_k \ \ldots \ c_n \end{bmatrix}$

If $b \subset c$ for all $k$, then $c_1$ is a prime implicant of $f$.

Example 5.2: Consider a cube $b_4 = (10-010-0110)$.

Because $b_4 \subset 9$, $c_1$ is expandable to the direction $X_1$.

So $9$ can be written as $9 = c_1 \lor 9_1$, where $c_1 = (11-010-1010)$.

$c_1$ is a prime implicant because it is unexpandable to any other direction. Similarly, both $c_2$ and $c_3$ are prime implicants.

Now, $9$ can be written as $9 = c_4 \lor 9_2$, where $c_4 = (10-100-0110)$.

Consider a cube $b_4 = (10-010-0110)$.

Because $b_4 \subset 9$, $c_1$ is expandable to the direction $X_2$.

So $9$ can be written as $9 = c_4 \lor 9_2$, where $c_4 = (10-100-0110)$.

$c_4$ is a prime implicant because it is unexpandable to any other direction. Hence, $9 = c_1 \lor c_2 \lor c_3 \lor c_4$ is a sum-of-products expression consisting of prime implicants.

(End of example).

Theorem 5.2: Let $f = c \lor f$, and $c$ be a cube.

1) If $c \subset f$, then cube $c$ can be deleted, i.e., $f = f$. If $f$ is irredundant (minimal).

2) Let $f$ be represented as $f = c_1 \lor c_2 \lor \ldots \lor c_m$, where $c_i$ is a prime implicant, Let $9$ be a sum of prime implicants other than $c_i$. If $c_i \subset 9$ for all $i (i=1,2,\ldots,m)$, then $9$ is irredundant (minimal).

Example 5.3: Consider an array consisting of prime implicants:

$9 = \begin{bmatrix} 01-100-1110 \\ 10-010-1110 \\ 10-100-1110 \end{bmatrix}$

$\begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix}$

Consider a cube $b_1 = (10-010-1010)$.

Because $b_1 \subset 9$, $c_1$ is expandable to the direction $X_1$.

So $9$ can be written as $9 = c_1 \lor 9_1$, where $c_1 = (11-010-1010)$.

$c_1$ is a prime implicant because it is unexpandable to any other direction.

Similarly, both $c_2$ and $c_3$ are prime implicants.

Therefore, $9 = c_1 \lor 9_1$. (End of Example).
Because \( c_3 \not\in \mathcal{G}_3 \), \( c_3 \) can be deleted.

Similarly, \( \mathcal{F} \) is written as \( \mathcal{F} = \mathcal{F}_4 \), where

\[
\mathcal{G}_4 = \{11-010-1010, 01-100-1110, 10-110-0110\}
\]

Because \( c_4 \not\in \mathcal{G}_4 \), \( c_4 \) cannot be deleted.

Hence, we have an irredundant sum-of-products expression:

\[
\mathcal{G}_3 = \{11-010-1010, 01-100-1110, 10-110-0110\}
\]

(End of example).

**Definition 5.2**: Let \( c \) be a cube of \( f \), and let \( v \) be a minterm of \( f \). If the prime implicant which covers \( v \) is unique, then \( c \) is an essential prime implicant, and \( v \) is a distinguished minterm.

**Definition 5.3**: A sum-of-products expression is said to be minimum if it consists of the minimum number of prime implicants.

**Lemma 5.2**: A minimum sum-of-products expression for \( f \) contains all the essential prime implicants of \( f \), if any.

**Definition 5.4**: Let \( c_1 \) and \( c_2 \) be cubes, where

\[
c_1 = x_{i_1} \cdot x_{i_2} \cdot \ldots \cdot x_{i_n} \quad \text{and} \quad c_2 = x_{j_1} \cdot x_{j_2} \cdot \ldots \cdot x_{j_n}.
\]

A consensus of \( c_1 \) and \( c_2 \) is defined as

\[
\text{cons}(c_1, c_2) = \bigcup_{i=1}^{n} \{ x_{i_1} \cdot x_{i_2} \cdot \ldots \cdot x_{i_n} \}.
\]

and denoted by \( \text{cons}(c_1, c_2) \).

**Definition 5.5**: Let \( c \) be a cube and \( \mathcal{G} \) be an array. A consensus of \( c \) and \( \mathcal{G} \) is defined as

\[
\text{cons}(c, \mathcal{G}) = \bigcup_{c_i \in \mathcal{G}} \text{cons}(c, c_i).
\]

**Theorem 5.3**: Suppose that \( \mathcal{F} \) can be written as \( \mathcal{F} = c \cup \mathcal{G} \), where \( c \) is a prime implicant. Let \( \mathcal{H} = \text{cons}(c, \mathcal{G}) \). If \( c \not\in \mathcal{H} \), then \( c \) is essential.

(Please refer to the proof for details.)

Suppose that \( c = x_{i_1} \cdot x_{i_2} \cdot \ldots \cdot x_{i_n} \) and \( c \not\in \mathcal{H} \).

There exists a minterm, where

\[
v = x_{a_1} \cdot x_{a_2} \cdot \ldots \cdot x_{a_n}
\]

such that \( v \not\in \mathcal{H} \), where \( a_i \in S_i (i = 1, 2, \ldots, n) \).

Suppose that a prime implicant \( c' \) which is different from \( c \) covers \( v \), where

\[
c' = x_{k_1} \cdot x_{k_2} \cdot \ldots \cdot x_{k_n}. \quad \text{Because} \quad c \cap c' = \emptyset \quad \text{and} \quad c \not\in c', \quad \text{we can assume that} \quad T_k - S_k \neq \emptyset,
\]

and that there is a minterm in \( c' \) such that

\[
v' = x_{k_1} \cdot x_{k_2} \cdot \ldots \cdot x_{k_{k-1}} \cdot x_{k_k} \cdot x_{k_{k+1}} \cdot \ldots \cdot x_{k_n},
\]

where \( b_k \in T_k - S_k \).

Because \( v' \in c \) and \( v' \in c' \), \( v' \) is a minterm of \( \mathcal{G} \). Therefore, there exists a cube \( d \) in \( \mathcal{G} \) which covers \( v' \).

Let \( d = x_{1} \cdot x_{2} \cdot \ldots \cdot x_{n} \).

Note that \( a_i \in D_i \) for \( i = 1, 2, \ldots, n \) and \( b_k \in D_k \). Consider a consensus of \( c \) and \( d \),

\[
h_k = \text{cons}(c, d).
\]

Similarly, consider a consensus of \( c' \) and \( d \),

\[
h'_k = \text{cons}(c', d).
\]

Because \( a_1 \in S_1 \cap D_1 \) and \( b_k \in S_k \cup D_k \), we have \( v \not\in h_k \).

However, this contradicts the hypothesis that \( v \not\in \mathcal{H} \) because \( h_k \not\in \mathcal{H} \).

Hence, the prime implicants which covers \( c \) is unique. In other words, \( v \) is distinguished minterm and \( c \) is an essential prime implicant.

**Example 5.4**: Consider an array consisting of prime implicants:

\[
\mathcal{G} = \begin{bmatrix}
01-01-0111 & c_1 \\
10-01-0111 & c_2 \\
10-11-0001 & c_3 \\
01-10-0111 & c_4 \\
10-01-0111 & c_5 \\
10-11-0001 & c_6
\end{bmatrix}
\]

Let's find the essential prime implicants of the array.

\( \mathcal{F} \) is written as \( \mathcal{F} = c_1 \cup \mathcal{G}_1 \), where

\[
c_1 = \{01-01-1110\}
\]

and

\[
\mathcal{G}_1 = \begin{bmatrix}
01-01-0111 & c_2 \\
10-01-0111 & c_3 \\
10-11-0001 & c_4
\end{bmatrix}
\]

First, make a consensus of \( c_1 \) and \( \mathcal{G}_1 \):

\[
\mathcal{H}_1 = \text{cons}(c_1, \mathcal{G}_1) = \{01-11-0110\}
\]

Because \( c_1 \not\in \mathcal{H}_1 \), \( c_1 \) is an essential prime implicant.

\( \mathcal{F} \) is written as \( \mathcal{F} = c_2 \cup \mathcal{G}_2 \), where

\[
c_2 = \{01-10-0111\}
\]

and

\[
\mathcal{G}_2 = \begin{bmatrix}
01-01-1110 & c_3 \\
10-01-0111 & c_4 \\
10-11-0001 & c_6
\end{bmatrix}
\]

Similarly, make a consensus of \( c_2 \) and \( \mathcal{G}_2 \):

\[
\mathcal{H}_2 = \text{cons}(c_2, \mathcal{G}_2) = \{01-11-0110\}
\]

Because \( c_2 \not\in \mathcal{H}_2 \), \( c_2 \) is not essential.

Similarly, we can see that neither \( c_3 \) nor \( c_4 \) are essential. (End of Example).

**VI. CONCLUSION**

1. Two methods for decomposing a tautology problem into smaller ones are shown.
2. A hardware tautology checker is proposed. The computation time of the checker is proportional to the number of products in a given sum-of-products expression.
3. Application of the tautology checker for simplifying logical expressions with many variables is shown.
REFERENCES