A Method for Test Pattern Generation of Combinational Circuits Using Ordinary Algebra

Zubair Ahmed

***Abstract*—This paper introduces ordinary algebra to express the truth value of a logic function. The algebraic expressions are based on switching variables that take the values 0, 1, or unspecified. The expressions contain addition and subtraction operators from ordinary algebra. It is shown that these algebraic expressions can be used in conjunction with the Boolean difference equation to generate test patterns for a combinational logic circuit. The test pattern generation method is complete because it will find a test set for a fault or otherwise prove the fault to be untestable.**

***Index Terms* — ATPG, 0-1 controllability, observability, stuck-at faults.**

I. Introduction

TEST pattern generation for combinational and sequential circuits is an intractable problem. Many algorithms have been proposed for test pattern generation in the literature. Some of these algorithms are algebraic while others are based on backtracking method. Boolean difference is a powerful algebraic method described in [1] and [2]. The Boolean difference method will find a test for a fault if one exists or otherwise prove the fault to be untestable (i.e., the fault cannot be tested). Other algebraic methods are also available in the literature. For example, in [3], line conditions are attached to every line in the circuit and then these conditions are used to determine the value of a line in normal and faulty circuits. Several other methods that are an enhancement of the method described in [3] are also available in the literature. These are the equivalent normal procedure [4], the cause-effect equation [5], and the SPOOF procedure [6]. The algebraic methods described in [4-6] are complete because these methods will find a complete test set for a fault or prove the fault to be untestable.

On the other hand, there are algorithms for test pattern generation that are based on a backtracking method. In these methods, a target fault is controlled and the fault effect is propagated to an observable point such as a primary output in the circuit. If a conflict occurs, in the process of controlling and observing a fault, then the program must backtrack to re-decide on a previous decision. These Automatic Test Pattern Generation (ATPG) programs work with an equivalent fault set in order to reduce the number of target faults [7-9]. The D-algorithm was proposed to tackle the test pattern generation problem for combinational circuits [10-11]. In this algorithm, a discrepancy signal D or $\overline{D}$ is propagated to an observable point in the circuit. The decision points in this algorithm could be the entire circuit. An alternative algorithm was proposed in [12] where the decision points are only on the primary inputs. This reduces the number of decision points from the number of gates to the number of the primary inputs in the circuit. Further enhancements were introduced in [13] where essential signal values of internal nodes are determined which reduces the number of backtracking an ATPG program makes. This concept of essential signal assignment (unique implication) was also used in [14-15] to reduce the number of backtracks.

In this paper we apply ordinary algebra to express the truth value of a logic function. The algebraic expressions described in this paper are based on switching variables that may take the values 0, 1, or unspecified. The expressions contain addition and subtraction operators from ordinary algebra. For any internal node or output of a circuit we determine two algebraic expressions. One expression enumerates all input combinations for which the internal node or the output assumes the logic value 1. The other expression enumerates all input combinations for which the internal node or the output assumes the logic value 0. Using these expressions in conjunction with the Boolean difference equation, we are able to determine the test set for any fault in the circuit. The proposed algebraic method is complete because it either finds a complete test set for a fault or otherwise proves the fault to be untestable.

Topological description of a logic network is frequently used in many types of VLSI CAD applications. Logic simulation, fault simulation, timing analysis, and test pattern generation are some examples of VLSI CAD applications where the topological description of the circuit is used. The algebraic method described in this paper operates on a topological description of the circuit.

II. BINARY EXPRESSION

A combinational switching circuit *C* realizing a switching function *y* of *n* variables assigns a value of 0 or 1 to a bit string of length *n*. Each bit string of length *n* is called an input combination. The response of the switching circuit *C* is a function only of the input combinations. Since there are *n* bit positions and there exists two choices for each bit position (i.e., 0 or 1), there are 2^n such input combinations. We will show that such input combinations that define a switching function can often be expressed implicitly.

Definition 1: For any input variable or identifier the notation $\hat{x}$ means the value of *x* is 1. The notation $\underline{x}$ means the value of *x* is 0. The notation *x* means that the value of the variable or identifier is not specified. Therefore, *x* can be either a 0 or a 1.

The notation described in definition 1 allows us to specify a set of input combinations of length *n* in an implicit way. For example, consider a set of input variables or identifiers $x_1, x_2, \ldots, x_n$. We can implicitly specify all possible input combinations of length *n* in which the first identifier is a 0 and the third identifier is a 1 by writing $\underline{x}_1, x_{2,} \hat{x}_3, x_4, \ldots, x_n$. Since x_1 is specified as $\underline{x}_1$ and x_3 is specified as $\hat{x}_3$ they must be 0 and 1 respectively in each input combination. The identifiers $x_2, x_4, \ldots, x_n$ have two choices. That is, these remaining identifiers can be either a 0 or a 1. Therefore, we can write a product term $\underline{x}_1 x_2 \hat{x}_3 \ldots x_n$ that implicitly enumerates 2^{n-2} input combinations in which the first identifier is a 0 and the third identifier is a 1.

For every node in a circuit, whether that is a primary input, an internal node, or a primary output, we define two sets: the first set contains input combinations that set the node to a 1. The other set contains input combinations that set the node to a 0. A definition of these two sets is given below.

Definition 2: Let *C* be a switching circuit realizing a function *y* of *n* variables $x_1, x_2, \ldots, x_n$. We define $\hat{y}$ as a set and as an expression that consists of all input combinations that set the output node *y* to 1. The complement of $\hat{y}$ is $\underline{y}$ such that $\underline{y}$ is a set and an expression that contains all input combinations that set the output node *y* to 0. Let $|\hat{y}|$ and $|\underline{y}|$ denote the number of input combinations that set *y* to 1 and 0 respectively.

Consider a 3-input primitive OR gate with inputs x_1, x_2 and x_3 and output y. The only input combination that sets the output to 0 is $\underline{x}_1 \underline{x}_2 \underline{x}_3$. Since this is the only input combination for which $y = 0$, it follows that

$$\underline{y} = \underline{x}_1 \underline{x}_2 \underline{x}_3 \text{ and } |\underline{y}| = (1)(1)(1) = 1.$$

The product term $x_1 x_2 x_3$ is an implicit enumeration of 8 input combinations which is the total input space of the 3-input OR gate. Therefore, if we subtract $\underline{y}$ from the total input space we get the 7 input combinations for which the output is a 1. Then it follows that

$$\hat{y} = x_1 x_2 x_3 - \underline{x}_1 \underline{x}_2 \underline{x}_3 \text{ and}$$

$$|\hat{y}| = (2)(2)(2) - (1)(1)(1) = 7$$

For a 3-input primitive AND gate with inputs x_1, x_2, and x_3 and output y, we can similarly write that

$$\hat{y} = \hat{x}_1 \hat{x}_2 \hat{x}_3, \; |\hat{y}| = (1)(1)(1) = 1$$

$$\underline{y} = x_1 x_2 x_3 - \hat{x}_1 \hat{x}_2 \hat{x}_3, \; |\underline{y}| = (2)(2)(2) - (1)(1)(1) = 7$$

A product term is the AND operation of input variables or identifiers. In a product term, zero or more identifiers are unspecified. A product term is often an implicit enumeration of a set of input combinations. An identifier that is not specified in a product term can have two values (0 or 1). An identifier that is specified in a product term can assume only the specified value. A binary expression consists of product term(s) where each product term appears in the expression with an addition or a subtraction sign. For example, for the 3-input primitive OR gate, $\underline{y} = \underline{x}_1 \underline{x}_2 \underline{x}_3$ is a binary expression. Similarly, for the same gate, $\hat{y} = x_1 x_2 x_3 - \underline{x}_1 \underline{x}_2 \underline{x}_3$ is a binary expression. The name binary expression is based on the observation that each variable or identifier in this expression is either specified or not specified. If a variable or identifier is specified then it is specified to be either 0 or 1. If a variable or identifier is not specified then it is don't care and its value for the evaluation of the function does not matter.

If a product term contains *n* identifiers and each identifier in the product term is unspecified, then the product term is an implicit enumeration of 2^n input combinations. Consider a function *y* of *n* variables. Any subset of input combinations of the function is called an input space of the function. The 2^n input combinations of the function is called the total input space of the function. We use *X* to denote the total input space of a function *y*.

The union (sum) and intersection (product) operations can be defined for binary expressions $\hat{y}$ and $\underline{y}$ for a function *y* of *n* variables. Let *X* denote the total input space of the function *y*. Since $\hat{y}$ is a set of all input combinations that set *y* = 1 and $\underline{y}$ is a set of all input combinations for which *y* = 0, it follows that the union of these two sets is the total input space. That is,

$$\hat{y} \cup \underline{y} = y = X$$

In algebraic terms, the sum of the two expressions is an expression that contains all input combinations (the total input space). Therefore,

$$\hat{y} + \underline{y} = y = X$$

The intersection of the two sets must be empty because an input combination that sets *y* to 1 cannot set *y* to 0 as well and therefore, that input combination cannot be in both sets. Therefore,

$$\hat{y} \cap \underline{y} = \phi$$

The product operation of the two expressions $\hat{y}$ and $\underline{y}$ is 0. This is because an input combination that sets $y = 1$ must differ by at least one bit from an input combination that sets $y = 0$. Therefore,

$$\hat{y} \bullet \underline{y} = 0$$

III. ALGEBRAIC THEOREMS

We defined X as the total input space of a function, i.e., the 2^n input combinations of a n-variable function. A single variable function has a total input space of size two, namely 0 and 1. Thus a single variable function x can be expanded to write as

$$x = \hat{x} + \underline{x} \tag{1}$$

Using (1) we can expand a product term into two product terms for an identifier that is not specified. Also (1) can be used to combine two product terms that differ only by one identifier. That is, in one product term the identifier is specified as $\hat{x}$ and in the other product term the identifier is specified as $\underline{x}$.

We can specify the property of complementation [16]. A variable or identifier cannot be set to opposite values simultaneously and, therefore, such a proposition must be false.

$$\hat{x}\,\underline{x} = 0 \tag{2}$$

The property of idempotency is specified below and it can be proved by perfect induction [16].

$$x\,x = x \tag{3}$$

If a variable or identifier is specified in one product term and unspecified in another product term then multiplying these two product terms gives a product term in which the variable or the identifier is specified.

$$\begin{aligned} \hat{x}\,x &= \hat{x} \\ \underline{x}\,x &= \underline{x} \end{aligned} \tag{4}$$

Proof:

$$\hat{x}\,x = \hat{x}\,(\hat{x} + \underline{x}) = \hat{x}\hat{x} + \hat{x}\underline{x} = \hat{x}$$

$$\underline{x}\,x = \underline{x}\,(\hat{x} + \underline{x}) = \underline{x}\hat{x} + \underline{x}\underline{x} = \underline{x}$$

The addition and subtraction rules are from ordinary algebra and are shown below.

$$x + x = x + x$$

$$x - x = 0$$

Note that the sum rule for adding two variables is different from the Boolean idempotency rule. If x is a Boolean variable, then the idempotency rule states that summing the variable x with itself yields the variable itself as shown below [16].

$$x + x = x$$

On the other hand, in manipulating binary expressions, the addition and subtraction of variables or product terms follow the arithmetic rules.

The expansion rule has been described for a single variable. Given a product term in a binary expression, the expansion rule shows how to expand the product term into two product terms. We can expand a product term such that in the expanded form each product term represents a single input combination. That will produce too many input combinations. The alternative way to expand a product term is to expand one identifier at a time to obtain a desired product term. Consider the first form of expansion.

$$x_1\,x_2 = \underline{x}_1\,\underline{x}_2 + \underline{x}_1\,\hat{x}_2 + \hat{x}_1\,\underline{x}_2 + \hat{x}_1\,\hat{x}_2$$

The above form of expansion expands each variable in the expression to obtain an expression in which each product term represents a single input combination. An alternative form of expansion is sometimes very helpful in the manipulation of binary expressions. This form of expansion is shown below.

$$\begin{aligned} x_1 x_2 x_3 &= \hat{x}_1 x_2 x_3 + \underline{x}_1 x_2 x_3 \\ &= \hat{x}_1 x_2 x_3 + \underline{x}_1 \hat{x}_2 x_3 + \underline{x}_1 \underline{x}_2 x_3 \\ &= \hat{x}_1 x_2 x_3 + \underline{x}_1 \hat{x}_2 x_3 + \underline{x}_1 \underline{x}_2 \hat{x}_3 + \underline{x}_1 \underline{x}_2 \underline{x}_3 \end{aligned}$$

The expansion procedure works as follows. In the first line, the variable x_1 is expanded using (1). In the second line, the variable x_2 is expanded using (1) for the second product term. In the third line, the variable x_3 is expanded using (1) for the third product term. If a product term is expanded for n variables then the above expansion procedure produces (n + 1) terms. For example, to reduce the binary expression ($x_1 x_2 x_3 - \underline{x}_1 \hat{x}_2 \underline{x}_3$) such that the negative term is eliminated we perform the following steps.

$$\begin{aligned} x_1 x_2 x_3 - \underline{x}_1 \hat{x}_2 \underline{x}_3 &= \hat{x}_1 x_2 x_3 + \underline{x}_1 \underline{x}_2 x_3 + \underline{x}_1 \hat{x}_2 \hat{x}_3 \\ &\quad + \underline{x}_1 \hat{x}_2 \underline{x}_3 - \underline{x}_1 \hat{x}_2 \underline{x}_3 \\ &= \hat{x}_1 x_2 x_3 + \underline{x}_1 \underline{x}_2 x_3 + \underline{x}_1 \hat{x}_2 \hat{x}_3 \end{aligned}$$

This form of expansion is far better because it yields fewer terms. If we had to do a full expansion to eliminate the negative term in the expression, we would end up with seven terms in the expression. When all the negative terms of a binary expression for a function y are eliminated, then the expression is a union of zero or more product terms that enumerate input combinations that set the function y to a 1 or a 0.

For any node y in a circuit whether that be an internal node or a primary output we determine both $\hat{y}$ and $\underline{y}$. These two expressions are enumeration of input combinations that set y to 1 and 0 respectively. Therefore, we determine input combinations both for a function and its complement.

IV. ALGEBRAIC EQUATIONS FOR PRIMITIVE GATES

All gates of a combinational circuit must be evaluated in order to determine the binary expressions associated with the output nodes of these gates. Therefore, algebraic equations must be established for the evaluation of all primitive gates. For any gate G in the circuit following are the rules for the evaluation of gate G.

(i) If *G* is a *NOT* gate with input y_1 and output y_2 then $\hat{y}_2 = \underline{y}_1$ and $\underline{y}_2 = \hat{y}_1$.

(ii) If *G* is an *AND* gate with inputs $y_1, y_2, \ldots, y_k$ and output y_{k+1} then k

$$\hat{y}_{k+1} = \prod_{j=1}^{k} \hat{y}_j \text{ and } \underline{y}_{k+1} = y_1 y_2 \ldots y_k - \hat{y}_{k+1}.$$

(iii) If *G* is an *OR* gate with inputs $y_1, y_2, \ldots, y_k$ and output y_{k+1} then

$$\underline{y}_{k+1} = \prod_{j=1}^{k} \underline{y}_j \text{ and } \hat{y}_{k+1} = y_1 y_2 \ldots y_k - \underline{y}_{k+1}.$$

(iv) If *G* is an *XOR* gate with inputs y_1, y_2 and output y_3 then $\underline{y}_3 = \underline{y}_1 \underline{y}_2 + \hat{y}_1 \hat{y}_2$ and $\hat{y}_3 = \underline{y}_1 \hat{y}_2 + \hat{y}_1 \underline{y}_2$.

For the AND(OR) gates in the above equations, $y_1 y_2 \ldots y_k$ denote the total input space of output node y_{k+1}. Therefore, this product term is the product of all input variables (unspecified) that feed the output node y_{k+1}.

We now discuss the algebraic equations for the primitive gates. For a NOT gate, the input combinations that set the input of the NOT gate to 0 also set the output to 1 and vice versa. For an AND gate, the binary expressions $\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_k$ set the inputs to a logic 1 by definition. In determining the output binary expression $\hat{y}_{k+1}$, we have to consider three cases.

Case 1: A product term in the expression for $\hat{y}_{k+1}$ is 0 if the product term contains an identifier x both as $\hat{x}$ and $\underline{x}$ (using (2)).

Case 2: If an unspecified identifier x appears multiple times in a product term of $\hat{y}_{k+1}$ then it is reduced to one identifier x (using (3)).

Case 3: If a product term of $\hat{y}_{k+1}$ contains an identifier x both in specified ($\hat{x}$ *or* $\underline{x}$) form and unspecified (x) form then the identifier becomes specified (using (4)).

The multiplication operation gives the input combinations that set node y_{k+1} to 1 for an AND gate. Therefore, if $\hat{y}_{k+1}$ is the set of input combinations that set y_{k+1} to a 1, then subtracting $\hat{y}_{k+1}$ from the total input space of node y_{k+1} gives the input combinations that set y_{k+1} to a 0. Similar argument can be made about an OR gate. Note that the above argument is for a gate of convergence where a reconvergent fanout node converges. The equation for the XOR gate follows from its Boolean equation.

V. USING THE BINARY EXPRESSION

The determination of binary expressions for circuit nodes is illustrated in this section with an example (Fig. 1). The circuit must be levelized first. Then binary expressions are determined for level 1 gates, level 2 gates and so on using the algebraic equations described in the previous section.

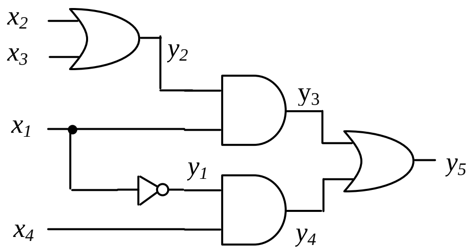

Figure 1: Example circuit for finding of binary expressions.

The level 1 gates y_1 and y_2 are evaluated first. The gate y_1 is an inverter. Therefore,

$$\hat{y}_1 = \underline{x}_1;$$

$$\underline{y}_1 = \hat{x}_1;$$

The gate y_2 is an OR gate and its binary expressions are given by

$$\underline{y}_2 = \underline{x}_2 \underline{x}_3;$$

$$\hat{y}_2 = x_2 x_3 - \underline{x}_2 \underline{x}_3;$$

Next we determine the binary expressions associated with the outputs of level 2 gates y_3 and y_4.

$$\hat{y}_3 = \hat{x}_1 \hat{y}_2 = \hat{x}_1 x_2 x_3 - \hat{x}_1 \underline{x}_2 \underline{x}_3;$$

$$\underline{y}_3 = x_1 y_2 - \hat{y}_3 = x_1 x_2 x_3 - \hat{x}_1 x_2 x_3 + \hat{x}_1 \underline{x}_2 \underline{x}_3$$

$$\hat{y}_4 = \hat{x}_4 \hat{y}_1 = \underline{x}_1 \hat{x}_4;$$

$$\underline{y}_4 = x_4 y_1 - \hat{y}_4 = x_1 x_4 - \underline{x}_1 \hat{x}_4$$

The last gate in the example of Fig. 1 is a level 3 gate y_5. The binary expressions corresponding to the output of this gate are determined as follows.

$$\begin{aligned}\underline{y}_5 = \underline{y}_3 \underline{y}_4 &= (x_1 x_2 x_3 - \hat{x}_1 x_2 x_3 + \hat{x}_1 \underline{x}_2 \underline{x}_3) \\ &\quad (x_1 x_4 - \underline{x}_1 \hat{x}_4) \\ &= x_1 x_2 x_3 x_4 - \underline{x}_1 x_2 x_3 \hat{x}_4 - \hat{x}_1 x_2 x_3 x_4 + \\ &\quad \hat{x}_1 \underline{x}_2 \underline{x}_3 x_4\end{aligned}$$

Next we proceed to determine the binary expression for node y_5 that sets y_5 to logic 1.

$$\begin{aligned}\hat{y}_5 &= y_3 y_4 - \underline{y}_5 \\ &= x_1x_2x_3x_4 - x_1x_2x_3x_4 + \underline{x}_1x_2x_3\hat{x}_4 + \\ &\quad \hat{x}_1x_2x_3x_4 - \hat{x}_1\underline{x}_2\underline{x}_3x_4 \\ &= \underline{x}_1x_2x_3\hat{x}_4 + \hat{x}_1x_2x_3x_4 - \hat{x}_1\underline{x}_2\underline{x}_3x_4\end{aligned}$$

The expression for $\underline{y}_5$ readily gives all input combinations that set node y_5 to 0. The number of input combinations that set y_5 to 0 can be determined from the expression as follows.

$$\begin{aligned}\left|\underline{y}_5\right| &= (2)(2)(2)(2) - (1)(2)(2)(1) - (1)(2)(2)(2) \\ &\quad + (1)(1)(1)(2) = 6\end{aligned}$$

The negative terms from the expression for $\underline{y}_5$ can be eliminated as follows.

$$\begin{aligned}\underline{y}_5 &= \underline{x}_1x_2x_3x_4 + \hat{x}_1x_2x_3x_4 - \underline{x}_1x_2x_3\hat{x}_4 \\ &\quad - \hat{x}_1x_2x_3x_4 + \hat{x}_1\underline{x}_2\underline{x}_3x_4 \\ &= \underline{x}_1x_2x_3\underline{x}_4 + \underline{x}_1x_2x_3\hat{x}_4 - \underline{x}_1x_2x_3\hat{x}_4 \\ &\quad + \hat{x}_1\underline{x}_2\underline{x}_3x_4 \\ &= \underline{x}_1x_2x_3\underline{x}_4 + \hat{x}_1\underline{x}_2\underline{x}_3x_4\end{aligned}$$

In the above expansion, the product term $x_1x_2x_3x_4$ was first expanded for the variable x_1 using (1). Then the variable x_4 was expanded using (1) again in the second line. Then it follows that the output node y_5 is 0 for the following input combinations {$x_1x_2x_3x_4$ = 0XX0, 100X}.

We now manipulate the binary expression that set node y_5 to a 1. First, the number of input combinations that set the output node to 1 is given by the following.

$$\begin{aligned}\left|\hat{y}_5\right| &= (1)(2)(2)(1) + (1)(2)(2)(2) - (1)(1)(1)(2) \\ &= 10\end{aligned}$$

We can eliminate the negative terms from the expression for y_5 that set y_5 to 1.

$$\begin{aligned}\hat{y}_5 &= \hat{x}_1x_2x_3x_4 + \underline{x}_1x_2x_3\hat{x}_4 - \hat{x}_1\underline{x}_2\underline{x}_3x_4 \\ &= \hat{x}_1\hat{x}_2x_3x_4 + \hat{x}_1\underline{x}_2x_3x_4 + \underline{x}_1x_2x_3\hat{x}_4 - \\ &\quad \hat{x}_1\underline{x}_2\underline{x}_3x_4 \\ &= \hat{x}_1\hat{x}_2x_3x_4 + \hat{x}_1\underline{x}_2\hat{x}_3x_4 + \underline{x}_1x_2x_3\hat{x}_4\end{aligned}$$

Therefore, the input combinations that set y_5 to 1 is {$x_1x_2x_3x_4$ =11XX, 101X, 0XX1}.

VI. TEST PATTERN GENERATION METHODOLOGY

The objective of a test pattern generation algorithm for combinational circuits is to determine a set of input patterns such that when these patterns are applied at the inputs of a circuit, the response of the circuit is incorrect in the presence of a fault. A fault is an abstract model of a physical defect. Thus a fault in an integrated circuit represents the manifestation of a physical defect in the functional behavior of the circuit. The most commonly used fault model is the stuck-at fault. A stuck-at fault denotes the functionality of a signal line in a circuit as either stuck-at 0 (s-a-0) or stuck-at 1 (s-a-1) in the presence of a physical defect. One important assumption made by most test pattern generation algorithms is that only a single line in the circuit is faulty whether it is a s-a-0 fault or a s-a-1 fault. This is often known as the single stuck fault (SSF) assumption. We assume a single stuck at fault model in this paper.

We show the use of the Boolean difference method for determining the test patterns for a combinational circuit. The Boolean difference is a powerful method for determining test patterns because it is guaranteed to find a test pattern for a fault if there exists a test pattern for the fault [1-2]. Consider a function y of input variables $x_1, x_2, \ldots x_n$. The set of test patterns that detect the fault x_j s-a-0 is given by the equation

$$T = x_j \frac{dy}{dx_j} \qquad (5)$$

and the set of test patterns that detect the fault x_j s-a-1 is given by the equation

$$T = \bar{x}_j \frac{dy}{dx_j} \qquad (6)$$

In the above equations, $\frac{dy}{dx_j}$ is called the Boolean difference of y with respect to x_j and it is given by the expression

$$\begin{aligned}\frac{dy}{dx_j} &= y(x_j = 1) \oplus y(x_j = 0) \\ &= y(x_j = 1)\bar{y}(x_j = 0) + \bar{y}(x_j = 1)y(x_j = 0)\end{aligned}$$

The binary expressions can be substituted in the Boolean difference equation as follows. Consider a binary expression $\hat{y}$ for a function y. Let x_j be an input variable of y. The product operation $\hat{x}_j\hat{y}$ gives all input combinations for which y is a 1 and in each of these input combinations the variable x_j has a value of 1. Following the product operation, if we remove the literal $\hat{x}_j$ from the expression $\hat{x}_j\hat{y}$ then this expression is equivalent to the switching expression $y(x_j = 1)$. This is because $y(x_j = 1)$ denotes all input combinations for which y is a 1 and in each of these input combinations the value of x_j is a 1. We denote this product operation by

$\hat{y}(\hat{x}_j)$.

That is, $\hat{y}(\hat{x}_j) = \hat{x}_j\hat{y}$. The same definition applies for the expressions $\hat{y}(\underline{x}_j)$, $\underline{y}(\hat{x}_j)$ and $\underline{y}(\underline{x}_j)$. Consider a 3-input OR gate with y = x_1 + x_2 + x_3. The binary expression $\underline{y} = \underline{x}_1\underline{x}_2\underline{x}_3$ represents the input combination for which *y* is 0. Similarly, $\hat{y} = x_1x_2x_3 - \underline{x}_1\underline{x}_2\underline{x}_3$ represents the input combinations for which y is 1. Then

$$\hat{y}(\hat{x}_1) = \hat{x}_1\,\hat{y} = x_2x_3$$

$$\hat{y}(\underline{x}_1) = \underline{x}_1\,\hat{y} = x_2x_3 - \underline{x}_2\underline{x}_3$$

$$\underline{y}(\hat{x}_1) = \hat{x}_1\,\underline{y} = 0$$

$$\underline{y}(\underline{x}_1) = \underline{x}_1\,\underline{y} = \underline{x}_2\underline{x}_3$$

The function $y(x_j = 1)$ denotes all input combinations for which the function *y* is 1 and in each of these input combinations the value of the variable x_j is 1. Therefore, $y(x_j = 1)$ is equivalent to the expression $\hat{y}(\hat{x}_j)$ because the binary expression denotes all input combinations for which the function y is 1 and in each of those input combinations the value of the variable x_j is 1. Similarly, the function $\bar{y}(x_j = 0)$ denotes all input combinations for which the function *y* is 0 and in each those input combinations the value of the variable x_j is 0. Therefore, this term is equivalent to the expression $\underline{y}(\underline{x}_j)$ because this binary expression denotes all input combinations for which the function y is 0 and in each of those input combinations the value of the variable x_j is 0. Similar argument can be made about the equivalence of the terms $\bar{y}(x_j = 1)$ and $\underline{y}(\hat{x}_j)$ and the terms $y(x_j = 0)$ and $\hat{y}(\underline{x}_j)$. Based on this discussion, the Boolean difference of a function *y* with respect to the variable x_j can be rewritten for the binary expressions as,

$$\frac{dy}{dx_j} = y(x_j = 1)\,\bar{y}(x_j = 0) + \bar{y}(x_j = 1)\,y(x_j = 0)$$
$$= \hat{y}(\hat{x}_j)\,\underline{y}(\underline{x}_j) + \underline{y}(\hat{x}_j)\,\hat{y}(\underline{x}_j)$$

With the Boolean difference equation defined for binary expressions, we can now determine the test sets for a fault x_j s-a-0 and s-a-1 by using equations (5) and (6) respectively.

The Boolean difference equation described above determines test patterns for faults on primary inputs of a circuit. The Boolean difference method can also be used to determine test patterns for faults on the internal nodes of a circuit. This is made possible using the chain rule [1]. Consider a function y_i with input variables $X = x_1, x_2, \ldots, x_n$. Let *C* be the circuit realization of this function and y_j be an internal node in *C*. Then the function realized by node y_j can be written in terms of the input variables as $y_j(X)$. Moreover, the function $y_i(X)$ can be expressed as a function of the internal node y_j and the input variables *X* as $g(y_j, X)$. Then we can determine the Boolean difference of the function $g(y_j, X)$ with respect to y_j. The Boolean difference is given by

$$\frac{dg(y_j, X)}{dy_j} = g(y_j = 0) \oplus g(y_j = 1)$$
$$= g(y_j = 0)\,\bar{g}(y_j = 1) + \bar{g}(y_j = 0)\,g(y_j = 1)$$

In order to use the binary expressions in determining test vectors for an internal node, the binary expression must also be derived based on the identifier of the internal node. Therefore, the binary expressions for the output node must be expressed in terms of y_j and *X*. With this modification of the binary expressions, we write the Boolean difference expression in terms of the binary expressions as

$$\frac{dy_i}{dy_j} = \hat{y}_i(\underline{y}_j)\,\underline{y}_i(\hat{y}_j) + \underline{y}_i(\underline{y}_j)\,\hat{y}_i(\hat{y}_j)$$

Then the set of test vectors that detect the fault y_j s-a-0 is given by

$$T = \hat{y}_j\,\frac{dy_i}{dy_j}$$

The set of test vectors that detect the fault y_j s-a-1 is given by

$$T = \underline{y}_j\,\frac{dy_i}{dy_j}$$

We illustrate the use of the chain rule with the circuit of Fig. 2. Consider deriving tests for stuck-at faults on node y_2 using the chain rule. In order to use the chain rule, the binary expression must be derived using the identifier for node y_2.

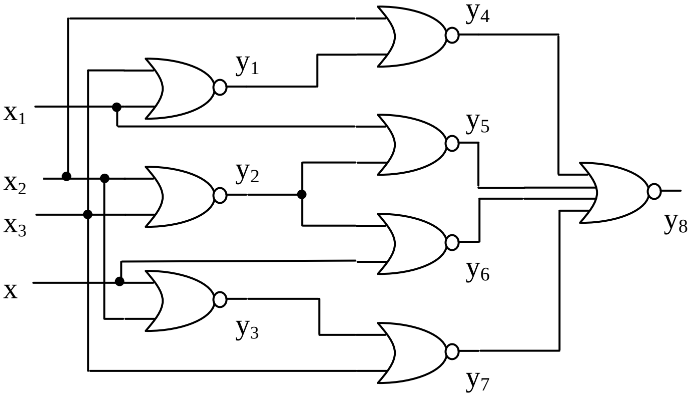

Figure 2: Example circuit for test pattern generation.

The binary expressions for node y_8 as a function of node y_2 are given below.

$$\hat{y}_8 = x_1\hat{x}_2\hat{x}_3\hat{x}_4 y_2 + x_1\hat{x}_2\hat{x}_3\underline{x}_4\hat{y}_2 - \underline{x}_1\hat{x}_2\hat{x}_3\hat{x}_4\underline{y}_2 + \underline{x}_1\underline{x}_2\underline{x}_3\underline{x}_4\hat{y}_2$$

$$\underline{y}_8 = x_1x_2x_3x_4y_2 - x_1\hat{x}_2\hat{x}_3\hat{x}_4y_2 - x_1\hat{x}_2\hat{x}_3\underline{x}_4\hat{y}_2 + \underline{x}_1\hat{x}_2\hat{x}_3\hat{x}_4\underline{y}_2 - \underline{x}_1\underline{x}_2\underline{x}_3\underline{x}_4\hat{y}_2$$

The Boolean difference of output node y_8 with respect to node y_2 is given by

$$\frac{dy_8}{dy_2} = \hat{y}_8(\underline{y}_2)\underline{y}_8(\hat{y}_2) + \underline{y}_8(\underline{y}_2)\hat{y}_8(\hat{y}_2)$$

Each of the terms in the Boolean expression is given below.

$$\hat{y}_8(\underline{y}_2) = \underline{y}_2\,\hat{y}_8 = \hat{x}_1\hat{x}_2\hat{x}_3\hat{x}_4$$

$$\underline{y}_8(\hat{y}_2) = \hat{y}_2\,\underline{y}_8 = x_1x_2x_3x_4 - x_1\hat{x}_2\hat{x}_3x_4 - \underline{x}_1\underline{x}_2\underline{x}_3\underline{x}_4$$

$$\underline{y}_8(\underline{y}_2) = \underline{y}_2\,\underline{y}_8 = x_1x_2x_3x_4 - \hat{x}_1\hat{x}_2\hat{x}_3\hat{x}_4$$

$$\hat{y}_8(\hat{y}_2) = \hat{y}_2\,\hat{y}_8 = x_1\hat{x}_2\hat{x}_3x_4 + \underline{x}_1\underline{x}_2\underline{x}_3\underline{x}_4$$

The difference equation evaluates to the following expression.

$$\frac{dy_8}{dy_2} = \underline{x}_1\hat{x}_2\hat{x}_3\hat{x}_4 + x_1\hat{x}_2\hat{x}_3\underline{x}_4 + \underline{x}_1\underline{x}_2\underline{x}_3\underline{x}_4$$

The set of test vectors to detect the fault y_2 s-a-0 is given by the following expression.

$$T = \hat{y}_2\frac{dy_8}{dy_2} = \underline{x}_1\underline{x}_2\underline{x}_3\underline{x}_4$$

The binary pattern to detect the fault y_2 s-a-0 is $\{x_1x_2x_3x_4 = 0000\}$. The set of test vectors that detect the fault y_2 s-a-1 is given by the following expression.

$$T = \underline{y}_2\frac{dy_8}{dy_2} = \underline{x}_1\hat{x}_2\hat{x}_3\hat{x}_4 + x_1\hat{x}_2\hat{x}_3\underline{x}_4$$

The binary patterns for this fault are $\{x_1x_2x_3x_4 = 0111, X110\}$.

The second product term in the expression $\hat{y}_8$ makes inconsistent assignment because y_2 is 0 when x_2 or x_3 is a 1. When we determine the binary expression for the output node y_8, y_2 is treated as an independent variable. Moreover, x_2 and x_3 are reconvergent fanout nodes that converge at node y_8. If we substitute $\hat{y}_2 = \underline{x}_2\underline{x}_3$ then this product term becomes 0 and therefore, does not cause any problem in the determination of the input combinations that set the output node to a 1.

VII. CONCLUSION

This paper presented an algebraic method for determining test set for a fault in a combinational circuit. A binary expression has been developed that incorporates rules from ordinary algebra. The binary expressions are based on logic variables.

The binary expression for a function is not unique. There is not a one-to-one correspondence or unique relationship between a function and its binary expression. A function may be represented by more than one binary expression that are different literal wise. However, the primary input combinations represented by the two different binary expressions of a function must of course be the same.

In a Boolean expression, two product terms may contain common primary input combinations. In other words, the input spaces denoted by two product terms in a Boolean expression may overlap. In a binary expression, however, two product terms denote input spaces that are disjoint; there is no overlap.

REFERENCES

[1] M. A. Breuer and A. D. Friedman, “Diagnosis & Reliable Design of Digital Systems”, Maryland: Computer Science Press, 1976, ch. 2.

[2] L-T Wang, C-W Wu, X. Wen, “VLSI Test Principles and Architectures: Design for Testability”, California: Morgan Kufmann Publishers, 2006, ch. 4.

[3] J. F. Poage, “Derivation of Optimum Tests to Detect Faults in Combinational Circuits”, Proc. Symposium on Mathematical Theory of Automata, Polytechnic Institute of Brooklyn, pp. 483-528, 1963.

[4] D. B. Armstrong, “On Finding a Nearly Minimal Set of Fault Detection Tests for Combinational Logic Nets”, IEEE Transactions on Electronic Computers, Vol. EC-15, pp. 66-73, February 1966.

[5] D. C. Bossen and S. J. Hong, “Cause-Effect Analysis for Multiple Fault Detection in Combinational Networks”, IEEE Transactions on Computers, vol. C-20, pp. 1252-1257, November 1971.

[6] F. W. Clegg, “Use of SPOOF’s in the Analysis of Faulty Logic Networks, IEEE Transactions on Computers, vol. C-22, pp. 229-234, March 1973.

[7] M. Abramovici, M. A. Breuer, and A. D. Friedman, “Digital Systems Testing and Testable Design”, IEEE Press, Piscataway, NJ, 1994.

[8] M. L. Bushnell and V. D. Agrawal, “Essentials of Electronic Testing for Digital, Memory, and Mixed-Signal VLSI Circuits”, Springer, New York, 2000.

[9] N. Jha and S. Gupta, “Testing of Digital Systems”, Cambridge University Press, Cambridge, U. K. , 2003.

[10] J. P. Roth, “Diagnosis of Automata Failures: A Calculus and A Method”, IBM J R&D, 10(4), pp. 278-291, 1966.

[11] J. P. Roth, W. G. Bouricius, and P. R. Schneider, “Programmed Algorithms to Compute Tests To Detect and Distinguish between Failures in Logic Circuits”, IEEE Trans. Electron. Computer, EC-16(10), pp. 567-579, 1967.

[12] P. Goel, “An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits”, IEEE Transactions on Computers, C-30(3), pp. 215-222, 1981.

[13] H. Fujiwara and T. Shimono, “On the Acceleration of Test Generation Algorithms”, IEEE Transactions on Computers, C-32(12), pp. 1137-1144, 1983.

[14] M. H. Schulz, E. Trischler, and T. M. Sarfert, "SOCRATES: A Highly Efficient Automatic Test Pattern Generation System", IEEE Transaction on Computer-Aided Design, 8(7), pp. 126-137, 1988.

[15] M. H. Schulz and E. Auth, "Improved Deterministic Test Pattern Generation with Applications to Redundancy Identification", IEEE Transactions on Computer-Aided Design, 8(7), pp. 811-816, 1989.

[16] Z. Kohavi, "Switching and Finite Automata Theory", New York: McGraw-Hill Book Company, 2nd Edition, 1978, ch. 3.

Z. Ahmed – Dr. Ahmed received his B.S.E in Electrical Engineering from The University of Iowa, USA in 1985, and M.E. and Ph.D. in Electrical Engineering from Rensselaer Polytechnic Institute, USA in 1989 and 1991 respectively. After completion of his doctoral studies, he served as an Advisory Engineer & Scientist at IBM Corporation for over five years. Subsequently, he served in a senior engineering capacity at several high-tech companies in the western part of USA including as a Senior Engineer with Fujitsu Microelectronics Corporation. He has accumulated over a decade of experience in the high-tech industry in the specialized fields of microprocessor design and test, electronic design automation (computer-aided design), and microelectronics circuit design. His research interest is in the area of VLSI testability. He can be contacted at his email address zahmed1212@yahoo.com.