PLAtestGA: A CNF-Satisfiability Problem for the Generation of Test Vectors for Missing Faults in VLSI Circuits

Alfredo Cruz, PhD
Associate Professor
Department of Electrical Engineering
Polytechnic University of Puerto Rico
across@coqui.net

Sumitra Mukherjee, PhD.
School of Computer and Information Sciences
Nova Southeastern University
sumitra@scis.nova.edu

ABSTRACT

An evolutionary algorithm (EA) approach is used in the development of a test vector generation application for single and multiple fault detection of growth faults in Programmable Logic Arrays (PLA). Three basic steps are performed during the generation of the test vectors: crossover, mutation and selection. The genetic operators are applied to the CNF-satisfiability problem for the generation of test vectors for growth faults. Once crossover and mutation have occurred, the new candidate test vectors with higher fitness function scores replace the old ones. With this scheme, population members steadily improve their fitness level with each new generation. The resulting process yields improved solutions to the problem of the PLA test vector generation.

SINOPSIS

Por medio de un algoritmo que evoluciona (Evolutionary Algorithm; EA), se desarrolla una aplicación que genera vectores de prueba para la detección de fallas de crecimiento múltiples y sencillo en Arreglos Lógicos Programables (Programmable Logic Arrays; PLA). Los algoritmos de evolución son unos procedimientos de búsqueda y optimización que encuentran su origen e inspiración en el mundo biológico. En este escrito se aplican los operadores genéticos al problema de satisfiabilidad-CNF en la generación de vectores de prueba para las fallas de crecimiento. CNF tiene varias ventajas. No hay dependencias entre "bits". Cualquier cambio puede resultar en un vector legal (o que hace sentido) que puede ser un "minterm" o un "maxterm". Por lo tanto, podemos aplicar mutaciones y "crossover" sin la necesidad de decodificadores o algoritmos reparadores. La operación de "crossover" no utiliza "lookups" o "backtracking" como lo utilizan los operadores que han sido utilizados previamente en la generación de pruebas de PLA.

I- INTRODUCTION

PLA testing has attracted the attention of many researchers in recent years [1, 2]. Genetic and evolutionary algorithms based solutions have been proposed [3, 4, 5, 6] for sequential circuits. However, PLA testing based on genetic and evolutionary algorithms is in its earliest development [7].

Algorithms proposed in [8] formulate the PLA test generation by using the sharp (T) operation. The T operation is widely used for logic manipulation algorithms (ESPRESSO II). Several other algorithms have been also proposed for PLA testing using the T operation, but they tend to be computationally expensive. A major disadvantage of this operator is that backtracking is necessary when a test cannot be found. The computational overheads required by backtracking can be prohibitive. Bose [9] proposed an algorithm, using the T operation for extra devices that utilizes the Quine-McCluskey method for testing missing devices. The memory requirements for this algorithm rise exponentially as PLA size increases. An algorithm reported in [10] assigns a proper logic value in the specified inputs of a potential vector. The aim is to achieve path sensitization by deselecting product lines connected to output functions. The latter however, may fail to find a test even if one exists.

Smith [11] suggests the simplification of the algorithm by generating a test for every fault. This results in considerably larger test vectors. Hence, a minimal test set is not guaranteed. One of the disadvantages in this approach is the backtracking

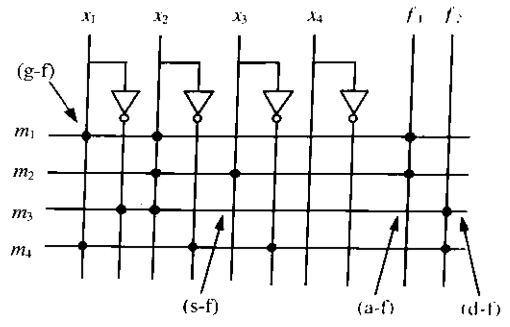

Figure 1: *A PLA Schematic*

that could occur when the test is chosen and fails to propagate.

Other approaches have been developed for generating a PLA test set. For example, the *Design-for-testability* (DFT) uses extra components to facilitate the PLA test, making test generation unnecessary in some instances. All these methods employ additional hardware, which means greater costs and potential degradation of PLA performance. Clearly, present DFT methods have not addressed the problem adequately [12].

A- Basic Notation and Definitions

The PLA consists of input lines (uncomplemented and complemented) and product term lines. The intersections between product lines and input bit lines or between output function lines and product term lines are called *crosspoints*. Each product line is used to realize an implicant (product term) of the given function by placing appropriate crosspoint devices into what is known as the AND plane. Figure 1 shows a simple schematic of a PLA, implementing the two switching functions:

$$f_1(x_1,x_2,x_3,x_4)=(x_1 \text{ AND } x_2) \text{ OR } (x_2 \text{ AND } x_3)$$
$$f_2(x_1,x_2,x_3,x_4)=(x_1' \text{ AND } x_2) \text{ OR } (x_1 \text{ AND } x_2' \text{ AND } x_3')$$

This PLA has four inputs (x_1, x_2, x_3, x_4), four product terms (m_1, m_2, m_3, m_4), and two output functions (f_1, f_2).

The following definitions apply to the discussions that follow.

Definition 1: *Hamming distance*: The number of bit positions in which two product terms hold non-don't care values that are different is called the Hamming distance, d_H.

For example, in Figure 1 the hamming distance between m3 and m4 is two, i.e. these terms differ in bit positions x_1 and x_2.

II- FAULT MODELING

In testing digital circuits, the most commonly considered fault model is the stuck-at fault (i.e., *s-a-0* or *s-a-1*). However, because of the PLA's array structure the stuck-at fault alone cannot adequately model all physical defects in a PLA [13]. Therefore, a new fault class model, known as the *crosspoint* model is used. The unintentional presence or absence of a device in the PLA causes a crosspoint fault.

Different types of faults are shown in Figure 1. The symbols (*g-f*), (*s-f*), (*a-f*), and (*d-f*) denote the growth, shrinkage, appearance and disappearance faults respectively.

The focus of this paper is on the use of genetic algorithms for the generation of test vectors growth faults.

III- THE GROWTH FAULT

Growth faults correspond to the removal of a literal, in the AND plane, from an implicant (product term) of the function which causes the growth of the implicant. A growth fault causes the ON-set (i.e., minterms) of a fault-free PLA to grow into the OFF-set (i.e., maxterms). To detect a fault in a PLA, it is important that the PLA output in the presence of the fault differ from the PLA output in the absence of the fault. The two requirements for fault detection are: *Fault Sensitization* and *Fault Propagation*.

A missing device fault in the AND plane will be sensitized if and only if the implicant under testing carries a 0 when fault-free and if the implicant carries a 1 in the presence of the fault. Once a fault has been sensitized then a propagation path must be established, otherwise the fault is masked[1]. The propagation is done by deselecting all other product

[1] The necessary condition under which masking occurs in PLA is given by [14].

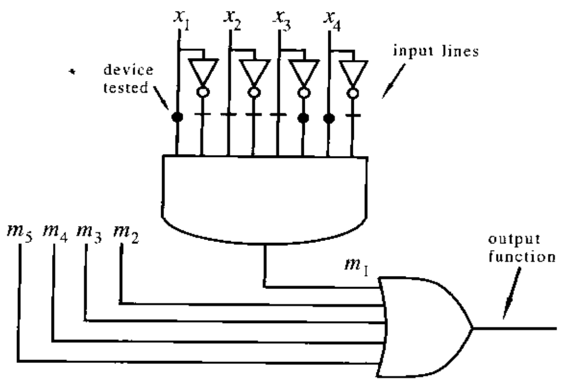

***Figure 2:** Product Term Under Test*

lines connected to the output except the product term under test.

The procedure for deriving the growth test vectors is explained with aid of Figure 2. The product term is represented by an AND gate of 4 inputs. A dash '-' in the input lines indicates the *absence* of a device, whereas a circle 'o' in the input lines indicates the *presence* of a device.

For example, to detect a missing device at x_1 of the implicant under consideration [1X01], a logic 0 must be applied to the input x_1, while the care values (at input x_3 x_4) remain unchanged. Since the value of the literal was changed from 1 to 0, then a term from this set could detect a fault in the uncomplemented bit-line. To detect a fault in the complemented bit-line the literal must be changed from 0 to 1.

Now we should be able to sensitize this fault (if one exists) at the output of the AND gate under consideration. A value 1 at the output of the AND gate denotes a fault while the implicant under test carries a 0 in the absence of a fault. To generate a 0 on the product line (required for sensitization of growth faults) the input value connected to the target growth fault bit-line is toggled to the value opposite the value representing the used bit-line. The simple PLA of Figure 3 will be used to illustrate the test pattern generation for growth faults.

The function on Table 1 of the sample PLA of Figure 3 can be expressed as

$$f(x_1,x_2,x_3,x_4)=\sum(0,1,2,4,5,6,9,13)$$

The complement function and the fitness for each element are also shown on Table 1.

The above discussion leads to the following rules that must be established for the generation of test vectors, for growth faults. A *growth term* stands

***Table 1:** Truth Table of the Sample PLA*

Decimal Code	x_1 x_2 x_3 x_4	f	f'	Fitness $f(x_i)$
0	0 0 0 0	1	0	4
1	0 0 0 1	1	0	3
2	0 0 1 0	1	0	4
3	0 0 1 1	0	1	5
4	0 1 0 0	1	0	4
5	0 1 0 1	1	0	4
6	0 1 1 0	1	0	4
7	0 1 1 1	0	1	5
8	1 0 0 0	0	1	5
9	1 0 0 1	1	0	4
10	1 0 1 0	0	1	5
11	1 0 1 1	0	1	5
12	1 1 0 0	0	1	5
13	1 1 0 1	1	0	4
14	1 1 1 0	0	1	5
15	1 1 1 1	0	1	5

m_1	1	X	0	1
m_2	0	X	1	0
m_3	0	1	0	X
m_4	0	0	0	X
m_5	0	X	0	1

Figure 3: *Sample PLA for Growth Faults*

for the set of extra terms contributed by a growth fault. The minterms {[$0\underline{0}01$], [$0\underline{1}01$]} are the components of the growth term for m_l of Figure 2. The input x_2 is a don't care and can be replaced either by 0 or 1. The bits of this unspecified input are shown underlined.

The growth term from a given set of product terms is derived as follows:

```
PROCEDURE 1:

For each product line m_j
 Do (n - log_2Ω)  times
      {
        Construct a growth term as follows:

        Scan the product term from left to
        right until an unmarked literal is
        found;

           Mark the literal and toggle its
           value from 0 to 1, or from 1 to
           0;

        Leave the other components of the
        product term intact (both literal
        and don't care values). These extra
        terms correspond to the growth
        term.
      }
```

(n - $\log_2\Omega$) is the number of literals on each product term.

A growth term may have terms in the ON-set function (i. e., minterms) and in the OFF-set function (i. e., maxterms). The terms in the ON-set function fail to select uniquely the product line on which the target is located, since it will also select the product terms that cover them. Therefore, the fault can not be propagated. Furthermore, since a fault can be sensitized by a term from the ON-set function, it is necessary to delete those terms from the growth term. This procedure can be carried out by computing the intersection (denoted by ($\cap$)) between the growth term generated for each product with the complement function (OFF-set) [8, 10]. One of the disadvantages in this approach is the backtracking that could occur when the test is chosen and fails to propagate.

Another approach is to apply the sharp operation (T) between the growth term generated for each product term and the ON-set function. Bose in [1, 9] uses this operation to find terms that are not covered by the ON-set function. However, terms partially covered by any input product can not be eliminated from the growth term without eliminating the growth term - an invalid test vector may be generated after the Quine-McCluskey method is applied.

PLATESTGA uses the conjunctive normal form (CNF) logical expression, equivalent to the complement's function, to derive the test set for growth faults. The use of the CNF is supported by the De Morgan's theorem [15]. The terms complement and OFF-set function are equivalent.

The generalized form of this theorem states that the complement of an expression is obtained by interchanging AND and OR operations and complementing each variable and constant. The complement of the function of Table 1 is derived by taking duals and complementing each literal.

The simple PLA of Figure 3 with the expression in its CNF will be used to illustrate the test pattern generation. This expression is equivalent to the complement function (see Table 1).

$$
\begin{aligned}
f'(x_1,x_2,x_3,x_4) = {} & (x_1' \text{ OR } x_3 \text{ OR } x_4') \\
& \text{AND}(x_1 \text{ OR } x_3' \text{ OR } x_4) \\
& \text{AND}(x_1 \text{ OR } x_2' \text{ OR } x_3) \\
& \text{AND}(x_1 \text{ OR } x_2 \text{ OR } x_3) \\
& \text{AND}(x_1 \text{ OR } x_3 \text{ OR } x_4')
\end{aligned}
$$

The use of CNF has several advantages over the two approaches mentioned before. It elimirates the possibility of intersecting a redundant growth term with a valid candidate test vector and consequently eliminates a good test vector. Also a minimal test set is guaranteed.

We apply a genetic algorithm using the CNF-satisfiability problem for the generation of test vectors for growth faults. The problem is to determine whether there exists a truth assignment for the variables in the expression, so that the whole expression evaluates to true. We must find an assignment of TRUE or FALSE (1 or 0) to each of the 4 literals of the sample PLA, so that the CNF

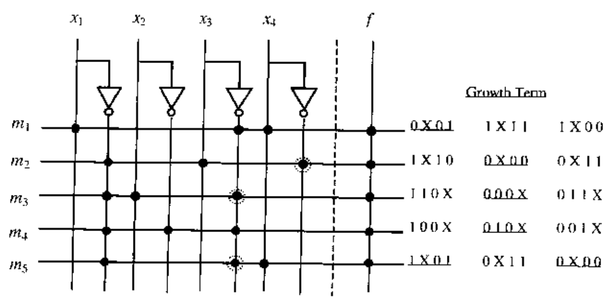

Figure 4: The Growth Term of the PLA

expression evaluates to TRUE. The CNF expression of the sample PLA is made up of five clauses. That will allow us to rank potential bit pattern solution in the range of 0 to 5, depending on the number of clauses that pattern satisfies. Table 1 shows the fitness of each element. When a pattern has a fitness of 5, a maxterm of the function is evaluated. A growth fault can be detected by this pattern if the intersection exists with a term(s) from the growth term set. The growth term set is the union ($\bigcup$) of the growth term generated by each product term (refer to Procedure 1). It is important to understand that an undetectable fault can not be detected by any pattern.

It is hard to imagine a problem with better suited representation: a binary vector of fixed length similar as the PLA physical layout should do the job. There are other several advantages, there are not dependencies between bits: any change would result in a legal (meaning) vector (either a minterm or a maxterm). Thus we can apply mutations and crossovers without any need for decoders or repair algorithms. Even other less frequently used genetic operators, such as the *inversion* (reversing the order of bits in the pattern) *or exchange* (interchanging two different bits in the pattern) leave the resulting bit pattern a legitimate possible solution [16, 17].

The growth term for the sample PLA is derived using Procedure 1 (see Figure 4). The fully redundant growth terms are underlined. A growth term is fully redundant when it is *fully covered* by one or more input product terms. A growth term may be partially redundant, i. e., *partially covered* by one or more input product terms.

PLATESTGA removes any possibility of intersecting a redundant term of the growth term with the solution found using the GA. That is, the offspring generated by the truth assignment using the genetic operator is always a valid candidate.

The following Lemma is necessary to the present discussion.

Lemma 1: A maxterm generated with the genetic operators with a d_H equal to 1 from any product term is qualified to detect a missing device fault in that product.

The proof of this Lemma is supported by Procedure 1 and the CNF used for the pattern generation.

IV- THE PLA GENETIC ALGORITHM

The basic genetic algorithm, where P(t) is the population of strings at generation t is given below:

```
procedure genetic algorithm
{
  set time t := 0
  select an initial population P(t)
  while the termination condition is not
  met, do:
  {
    evaluate fitness of each member of
    P(t);

    select the fittest members from P(t);
    generate offspring of the fittest pairs
    (using genetic operators);

    replace the weakest members of P(t) by
    these offspring;

    set time t := t+1
  }
}
```

Selection alone cannot introduce any new individuals into the population, i.e., it cannot find new candidate test vectors in the search space. Selection is done on the basis of relative fitness and it probabilistically eliminates from the population those candidate test vectors which have relatively low fitness. Recombination, which consists of mutation and crossover, imitates sexual reproduction.

Crossover is performed with crossover probability P_{cross} between two selected strings, called *parents*, by exchanging parts of their genomes (i.e., encoding) to form two new individuals, called *offspring*. It is implemented by choosing a random

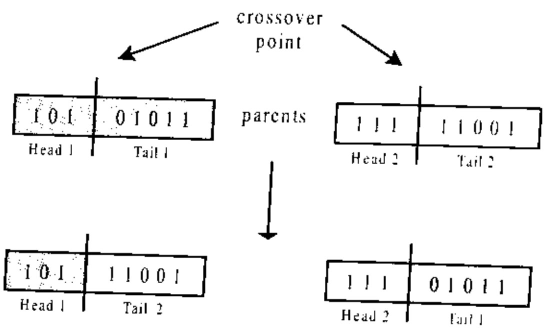

Figure 5. *The Effect of the Crossover Operation*

point between 1 and the string length (δ) minus one [1, δ − 1] in the selected pair of parents and exchanging the substring defined by that point (i.e., swap the tail portion of the string) to produce new offspring. That is, all the information from one parent is copied from the start up to the crossover point, and then all the information from the other parent is copied from the crossover point to the end of the offspring (chromosome). The new chromosome thus gets the head of one parent's chromosome combined with the tail of the other. Figure 5 illustrates crossover with parents '10101011' and '11111001', and crossover occurring after the third bit.

The crossover operation, unlike previous operators used in PLA test generation, does not use lookups or backtracking. Crossover is both simple and efficient. This operation enables the evolutionary process to move towards optimal solutions in the search space. The usefulness of crossover is due to the combination of better than average substrings coming from different individual [18].

Mutation probabilistically chooses a bit and flips it. Mutation is needed because if selection and crossover together search new solutions, they tend to cause rapid convergence and there is a danger of losing potentially useful genetic materials, such as 0s and 1s at particular location of the specified values of the candidate test vector under evolution.

The following GA parameters are used for testing growth faults of the sample PLA of Figure 3:

- Uniform Crossover Single Cut Point
- Number of generations : Until a minimal test set is found
- Size of Population : 8
- Crossover Probability : 1.0
- Mutation Probability : 0.1

PLATESTGA begins, at generation 0, with a population of 8 patterns. For each generation, each individual in the population is calculated as the number of clauses that pattern satisfies. A maximum value of 5 means that the pattern (candidate test vector) matches each clause of the CNF expression and consequently it is a valid candidate. For example, the fitness for the pattern [0001] of Table 1 is 3, while the fitness for the pattern [1100] is 5. The string [1100] in particular permit the detection of missing devices in product terms that are not activated, i. e., product terms that are compatible with the pattern under consideration.

The patterns {[1110], [0111]} generated on Table 2 are valid tests. Since genetic operators generate the pattern [1110], then it is a valid candidate valid test. This is necessary to assure propagation of the fault. The next step is to determine if there are product terms with a d_H equal to 1 from the pattern (candidate test). The *missing* fault that can be detected by this pattern is the complement bit-line of the first input line of the product term m_2.

The string [0111] uses Lemma 1 to find product terms, which are dissidents in one literal. These product terms are: m_2, m_3, and m_5. Therefore, this pattern detects a growth fault in the product terms where they have the dissident bits (one Hamming distance away). The *missing* faults detected by the pattern [0111] (with fitness 5) with the aid of Lemma 1 are shown in Figure 4. The faults detected are circled by a broken-line in their respective positions in the PLA.

The patterns {[0111], and [1110]} have a fitness of 5. The fittest members can be selected more than once. A bias roulette-wheel is used as the reproduction operator. In this way the selection of fittest members have proportionally more chances of being reproduced and patterns can be selected more than once. Table 2 shows how many times an individual is reproduced. Once the new population has been reproduced, strings are paired at random and recombined through crossover. The new individuals (patterns) will enter the new population

TABLE 2: GENERATION 0

String	Population X_1 X_2 X_3 X_4	Fitness $f(x_i)$	# of copies reproduced	Mate Pool (cross point site shown)	Mate #	Crossover $P_{cross}= 1.0$	Mutation $P_{mut}= 0.1$
(1)	0 0 0 1	3	0	0 1 | 0 1	3	0 0 1 ⓪	0 0 1 ①
(2)	0 1 0 1	4	1	1 | 1 1 0	6	1 1 1 0	1 1 1 0
(3)	1 1 1 0	⑤	1	0 0 | 1 0	1	0 0 0 1	0 0 0 1
(4)	0 0 1 0	4	1	0 1 1 | 0	5	0 1 1 0	0 1 1 0
(5)	0 1 1 0	4	1	1 1 1 | 0	4	1 1 1 0	1 1 1 0
(6)	1 1 1 0	⑤	2	1 | 1 1 0	2	1 ① 1 0	1 ⓪ 1 0
(7)	0 1 1 1	⑤	1	0 1 1 | 1	8	0 1 1 1	0 1 1 1
(8)	1 0 1 0	4	1	0 1 0 | 1	7	0 1 ⓪ 1	0 1 ① 0

Sum	35
Average	4.25
Max	5
Min	3

in place of their parents. Crossover is applied with a frequency $P_{cross} = 1.0$.

After crossover, mutation is applied to the population members with a frequency $P_{mut} = 0.1$. It is interesting to note that after these genetic operators are applied in each generation the population average fitness continues to improve until the population become little differentiated and the fitness levels-off.

The final growth test set for the PLA under consideration were found after the second generation. The test vectors are: {[0011], [0111], [1100], [1110], [1111]}.

V- CONCLUSIONS AND PARALLEL IMPLEMENTATION

This article describes the use of genetic algorithms to generate patterns for testing programmable logic arrays. Existing methods tend to be computationally expensive. Our proposed algorithm overcomes this problem to generate good solutions efficiently. While the preliminary results are encouraging, further testing with larger size PLA is necessary to validate these results.

The algorithm described is well suited for parallel processing. PLA fault simulation in parallel using GA should help to overcome the well-known bottleneck of serial simulation. Genetic algorithms are inherently parallel algorithms. Hence **PLA**TESTGA should be easily scaleable to multiple processor systems with shared memory.

VI- REFERENCES

[1] P. Bose, "A Novel Technique for Efficient Implementation of a Classical Logic/Fault Simulation Problem," *IEEE Trans. on Computers*, Vol. 37, pp.1569-1577, December 1984.

[2] A.Cruz and R. Reilova, "A Hardware Performance Analysis for a CAD Tool for PLA Testing, " *39th Midwest Symposium on Circuits and Systems*, 1997.

[3] M. S. Hsiao, E, M. Rudnik and Janak H. Patel, "Automatic Test Generation Using Genetically Engineering Distinguishing Sequences," *Proceedings of the VLSI Test Symposium*, pp. 216-223, April 1996.

[4] E. M. Rudnik, J. G. Holm, D. G. Saab and Janak Patel, "Application of Simple Genetic Algorithms to Sequential Circuit Test Generation," *Design Automation Conference*, pp. 717-721, June 1994.

[5] Janak Patel et al., "Parallel Genetic Algorithms for Simulation-based Sequential Circuit Test Generation," *Proceedings of the International Conference on VLSI Design*, pp. 475-481, January 1997.

[6] E. M. Rudnik and Janak Patel, "A Genetic Approach to Test Application Time Reduction for Full Scan and Partial Scan Circuits," *Proceedings of the Eight International Conference on VLSI Design*," pp. 288-293, January 1995.

[7] A. Cruz and S. Mukherjee, "PLAGA: A Highly Parallelizable Genetic Algorithm for

Programmable Logic Arrays Test Pattern Generation," *Congress on Evolutionary Computation*, Vol. 2, July 6-9, 1999.

[8] R. S. Wei and Sangiovanni-Vicentelli, "PLAtypus: A PLA Test Generation Tool." *Trans. on Computer Aided Design*, Vol. CAD-5, October 1986.

[9] P. Bose, "Generation of Minimal and Near-Minimal Test Set for Programmable Logic Arrays," *International Conference on Computers*, December 1984.

[10] M. Robinson and Rajski, "An Algorithm Branch and Bound Method For PLA Test Pattern Generation," *International Test Conference*, pp. 66-74, 1988.

[11] J. E. Smith, "Detection of Faults in Programmable Logic Arrays," *IEEE Transactions on Computers*, Vol. C-28, No. 11, November 1979.

[12] Hua and et al., "Built-in Tests for VLSI Finite State Machines," *Dig. of Papers 14th Int'l Conf. on Fault Tolerant Computing*, June 1984.

[13] M. Abramoci and et al., "Digital Systems Testing and Testable Design," 41 Madison Avenue, New York, NY 10010: *Computer Science Press*, 1990.

[14] V. K. Agarwal, "Multiple Fault Detection in Programmable Logic Arrays," *IEEE Trans. on Computers*, Vol. C-28, pp. 518-522, June 1980.

[15] M. Mano and R. Kime, "Logic and Computer Design," 2nd Ed., *Prentice Hall*, 2000.

[16] Z. Michalewicz, "Genetic Algorithms + Data Structures = Evolution Programs," Third, Revised Edition, *Springer Verlag*, 1996.

[17] G. Lugger and W. A. Stubblefield, Artificial Intelligence: Structures and Strategies for Complex Problem Solving," 3rd Ed., *Addison-Wesley*, 1998.

[18] G. Goldberg, "Genetic Algorithms in Search, Optimization and Machine Learning," *Addison Wesley, Reading, MA*, 1989.