International Journal of Innovation Studies

A Novel Approach to Generate Combinational ATPG using Genetic Algorithm

Dr Shahsidhara H R[1] Mr. Shuhaas K P[2]

1 Department of Electronics and Communication
2Department of Information Science and Engineering
The National Institute of Engineering, Mysuru-570008, INDIA
[1] E-mail: shashidharahr@nie.ac.in,[2] E-mail: suhaaskp@nie.ac.in

***Abstract*—**Testing is an important part of any digital IC design process. It is important to determine the faults and to diagnose the fault in an IC. A test vector is a set of inputs provided to the IC in order to test it. Generally test vector generation is a program used to automatically generate test data for use in automated testing circuits. This can generate many individual test vectors. Manual testing can be done for smaller designs, in which inputs can be given to the system by forcing the values and observing the output. As the complexity of the design increases the testing becomes tedious and hence automation is required. In VLSI testing we need Automated Test Pattern Generator (ATPG) to get input test vectors for the Device Under Test (DUT). In this paper we present a new way generate ATPG vectors using probability weights and find the optimal set of vector for the weights using genetic algorithm(GA) on excess three encoder as our DUT. The excess three encoder was simulated for stuck at fault modelling. We find that our approach has yielded promising results compared to other ATPG algorithms.

***Index Terms*—**ATPG, Genetic algorithm, combinational, IC testing.

I. INTRODUCTION

N the IC design flow, testing of designs plays a vital role in determining weather a chip is market ready and can be fabricated in large volumes. The improvements in wafer fabrication technology and industry acceptance of hierarchical design methodologies have enabled high levels of circuit integration while lowering overall IC product development costs. Unfortunately, the trend toward higher levels of integration has resulted in limiting the access to IP blocks for testing. In addition, each embedded block and interface, in the SoC design, requires a different test method or test period. These issues have increased the cost of testing. As a result, test cost-per-transistor, unlike manufacturing cost-per-transistor, has not tracked Moore's Law and is now a challenge to come up with procedures and techniques to solve this problem [1]. In testing, the most challenging part is the post-silicon validation. Where we have to test each IC for logical verification. Say an IC has N primary inputs(PIs), then we have to provide all 2^N inputs and compare the output to a logically correct device. As we see this is an NP-complete problem and is difficult to solve.

A defect is an error caused in a device during the man- ufacturing process. A fault model is a logical description of how a defect alters design behavior. The logic values observed at the device's primary outputs, while applying a test pattern to some device under test (DUT), are called the output of that test pattern. The output of a test pattern, when testing a fault-free device that works exactly as designed, is called the expected output of that test pattern. A fault is said to be detected by a test pattern if the output of that test pattern, when testing a device that has only that one fault, is different than the expected output. The generated patterns are used to test semiconductor devices after manufacture, and in some cases to assist with determining the cause of failure (failure analysis) the effectiveness of ATPG is measured by the number of modeled defects, or fault models, that are detected and the number of generated patterns. It is influenced by the fault model under consideration, the type of circuit under test, the level of abstraction used to represent the circuit under test and the required test quality. Fault models abstract the behavior of manufacturing defects so that test vectors can be generated to detect them.

1. Functional Defects: Stuck-at Fault Model,
2. Current defects: Pseudo Stuck-at Fault Model (IDDQ),
3. Speed defects: At-speed Fault Model, Path Delay Fault Model.

Stuck-at Faults is the most common fault model used in industry. The single stuck-at-fault model has been widely accepted as a standard target model to generate a set of test patterns to detect all the stuck faults in the circuit. A single stuck-at fault represents a 1 ; line in the circuit that is fixed to logic value 0 or 1. The single-stuck fault model is also referred to as the classical or standard fault model because it has been the first and the most widely studied and used. The ATPG process for a targeted fault consists of two phases, (1) Fault Activation and (2) Fault Propagation.

Fault activation establishes a signal value at the fault model site that is opposite of the value produced by the fault model. Fault propagation moves the resulting signal value, or fault effect, forward by sensitizing a path from the fault site to a primary output. It models manufacturing defects which occurs when a circuit node is shorted to VDD (stuck-at-1 fault) or GND (stuck-at-0 fault) permanently. The fault can be at the input or output of a gate. Thus, a simple 2-input AND gate has six possible stuck-at faults. Suppose we have a stuck-at-0, symbolically written as s@0 fault at the output of an AND gate. Note one important thing, there are two input ports in the circuit, thus we can have a combination of four different inputs or patterns 00, 01, 10, 11. Out of the four patterns, only one pattern 11 will be able to detect this fault. As with rest of the patterns the expected output will be same as the actual circuit output in the presence of s@0 fault. This is a simple circuit with one AND gate. So it wasn't difficult to find the pattern that can detect this fault, but for complex designs we have to rely on ATPG tools. The ATPG tools will try to generate patterns required to test all the possible faults locations using complex algorithms, but if it is unable to find patterns for few faults, then it will classify those faults as untestable.

In testing, the most challenging part is the post-silicon testing i.e. post-silicon validation. Where we have to test each IC for logical verification. Say an IC has N primary inputs(PIs), we can't provide and test for all 2^N inputs. The solution to this is to use random vectors, pseudo random vectors or Automated test pattern generator(ATPG). Random vectors and pseudo random vectors methods are inefficient and doesn't provide a stable fault coverage, so ATPG turns out be a better choice. In ATPG we use algorithms to generate test vectors for a model of our target device that simulate probable faults that can occur in our device during fabrication, So the flow of ATPG is the following

1. Model design,
2. Fault modelling,
3. Fault simulation,
4. Pattern Generation and
5. Pattern testing.

For ATPG generation we will use excess three encoder as our model for testing. We will first model single stuck at faults which is standard in the industry and our training model and use the results obtained from the training model to multi stuck at faults model of encoder. The training model is used to generate patterns using genetic algorithm(GA) frame work. Genetic Algorithm is based on the biological evolution process, it involves creating new fitter generations using the old one. This pattern are then tested on a randomly generated models of excess three encoder that is correct most of the time and some times faulty after a fixed interval of time we will stop the run and calculate the metrics like fault coverage.

II. Proposed Model

For testing our algorithm we will use excess three encoder as our test model, the reason to use excess three is that is a very well known digital circuit that has been thoroughly studied in academia. Also excess three encoder is a simple combinational circuit. In an excess three encoder, as the name suggests, we will add three to the given binary coded decimal(BCD) value to generate an excess three code(ETC). We can expand this to larger values and define that the excess three code of a value is the value of it when three is added to it. An N-bit excess three encoder by the above definition can be synthesised by using an N-bit adder with carry in set to zero and one of the operand set to three. An N-bit adder can be realised using N full adder blocks in cascade. A full adder can be implementedbe cascaded with other full adders to get an N-bit adder. For our test design we will consider an 8-bit excess three encoder, So we need an 8-bit adder as shown in figure 2. Such an adder system is also called an ripple carry adder(RCA).

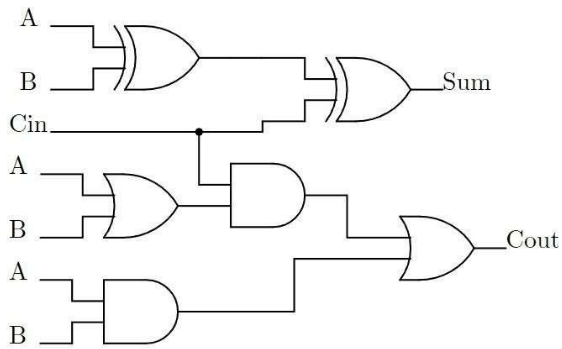


Fig. 1. Gate level implementation of full adder

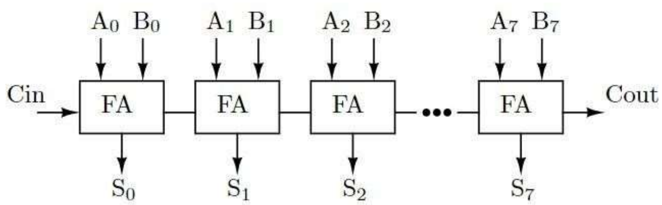


Fig. 2. Gate level implementation of full adder

III. Fault modelling

Once our proposed system is designed the next stage is to model faults in the design that can give a real world fault that can occur during fabrication. The real word faults that occur in ICs are very difficult simulate logically, so we abstract the process of system design to give a illusion of a fault in real world case. The abstraction can be gate level, transistor level or layout level. There are other sophisticated abstractions for larger designs to save the CPUs run time and memory requirement to simulate it. In the next sections we will look at some industry standard fault models.

1) *Fault models:* A fault model is a simplified representation of a potential defect or fault that can occur within the circuit. It serves as a model for analyzing and testing the behavior and resilience of the circuit in the presence of faults. Fault models help identify the types of faults that can occur, study their effects on circuit functionality, and develop strategies to detect and mitigate these faults.

Here are some commonly used fault models in digital circuits:

using the $sum = a \oplus b \oplus cin$ and $cout = a .b | cin .(a | b)$.

where " $\oplus$ " represents the XOR logic, ". " represents the AND logic and " | " represents the OR logic. *sum* represents the sum bit, *cout* represents the carry out bit, *a* and *b* are

operands bits and *cin* is the carry in bit. Figure 1 show the gate level implementation of full adder. This full adder has to *Stuck-At-Fault Model [2]*: This is the most widely used fault model. It assumes that a signal or wire within the circuit is stuck at a particular logic value (0 or 1), regardless of the input or operation of the circuit. Stuck-at faults represent permanent defects, such as open circuits or short circuits.

Transition Fault Model [3]: Transition faults occur when the circuit fails to make a proper transition from one logic state to another. These faults can be caused by issues such as delays, glitches, or excessive noise in the circuit.

Bridging Fault Model [4]: Bridging faults occur when two or more wires or nodes in the circuit are unintentionally connected, resulting in short circuits. This can lead to incorrect signal propagation and affect circuit operation.

Delay Fault Model [5]: Delay faults arise due to variations in signal propagation delays within the circuit. These faults can occur due to manufacturing defects, process variations, or temperature effects. Delay faults can cause timing violations and affect the correct operation of the circuit.

Single Stuck-Open and Single Stuck-Short Fault Models

[6]: These fault models consider specific types of stuck-at faults. A single stuck-open fault assumes that a connection or switch within the circuit is permanently open, while a single stuck-short fault assumes that a connection or switch is permanently shorted.

Fault models are essential for identifying, analyzing, and addressing potential faults in digital circuits. They provide a structured framework for fault detection, diagnosis, and testing, to ensure circuit reliability, improve design quality, and enhance system-level performance. By using fault models, engineers can effectively analyze the impact of faults, develop robust testing strategies, and implement fault-tolerant design techniques.

2) *Stuck at fault modelling:* The stuck-at fault model is a gate level abstraction for a logical circuit that assumes, a particular signal or wire within the circuit can be stuck at a specific logic value, either stuck at 0 or stuck at 1, regardless of the inputs or circuit operations. It is one of the most widely used fault models in digital circuit testing and an industry standard. The stuck-at fault model helps in identifying and detecting permanent defects that can occur in a digital circuit. There are two essential requirement for a stuck-at- fault model, one is *fault activation* and the other is *fault propagation*. here on a line/ wire that is stuck-at a particular logic level will be represented as *S@*0 or *S@*1 for stuck- at logic level 0 fault or stuck-at logic level 1 fault respectively.

There are two types of stuck-at-fault models (1) Single stuck-at-fault model and (2) multiple stuck-at-fault model. In single stuck-at-fault model we assume that only a single line/ wire in the circuit is stuck at a particular logic level. A simple two input logic gate can have 6 different possible single stuck- at-faults. Let us now consider an example for single stuck at faults shown in figure 3, where we assume that the output of the first xor gate have a stuck-at-fault. Notice that the two conditions are satisfied here.

Fault activation, where the outp

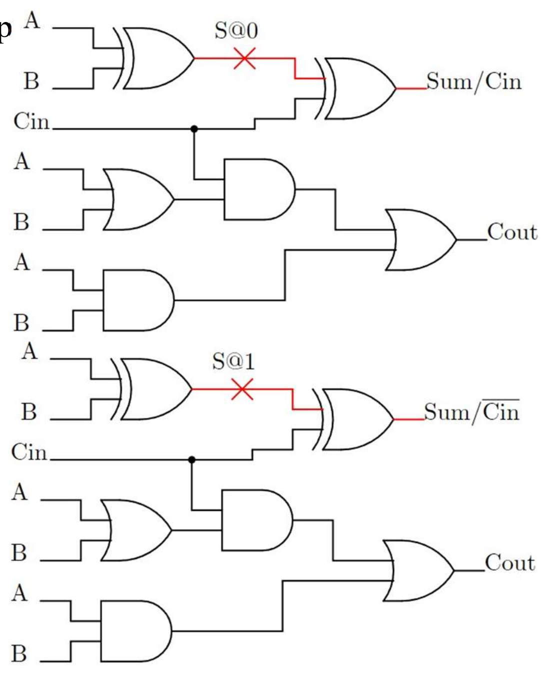


Fig. 3. Example for single stuck-at-fault in a full adder

is faulty irrespective of the inputs given and the fault is propagated to the *Sum* bit of the full adder. The *Sum* bit will produce the input logic at *Cin* line and complementary of the logic at *Cin* line for the *S@*0 and *S@*1 faults respectively, irrespective of the inputs *A* and *B*. Also note that no other lines, expect the path of the fault line, has got faults. In other words there operation is normal.

In multiple stuck-at-faults more than one lines/ wires are stuck at a particular logic. We can also say that a multiple stuck-at-fault is two or more single stuck-at-faults occurring together. Figure 4 shows a full adder that has multiple stuck- at-faults.

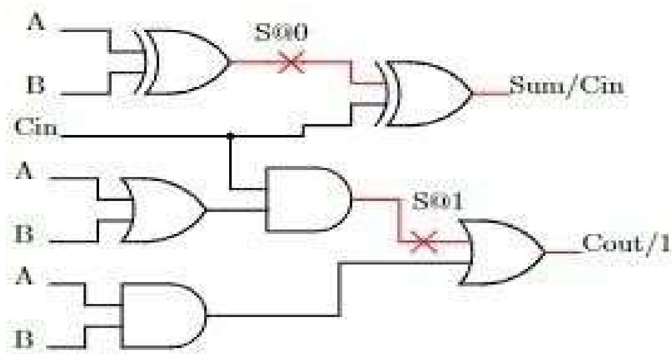


Fig. 4. Example of multiple stuck-at-faults

There are certain nuances in multiple stuck-at-faults that we need to keep in mind before we jump into the design of the fault model. The nuance is that, *the fan-outs of a fault is not a single stuck-at-fault if the fault propagates through more than one fan-outs*. Consider an example where that carry

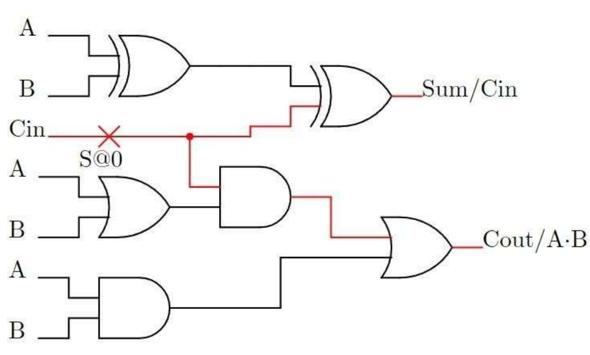


Fig. 5. An interesting case of stuck-at-faults

in bit, *Cin*, is S@0. Shown in Figure 5 is an example of a multiple stuck-at-fault not a single stuck-at-fault. The reason being our definition for single stuck-at-fault doesn't hold true for this case. Because only a single line has to have a fault. To model single stuck-at-fault in circuits that have fan-outs, we will assume that the origin of the error is at the fan-out point and there is no such single stuck-at-fault model where the *Cin* bit is stuck-at some logic. Also here we have considered an error in a primary input, if an gate output has fan-outs a similar procedure has to be followed.the end we will have $8 \times 28 = 224$ single stuck-at-fault models of ETC and one correct model of ETC.

Full adder codes:

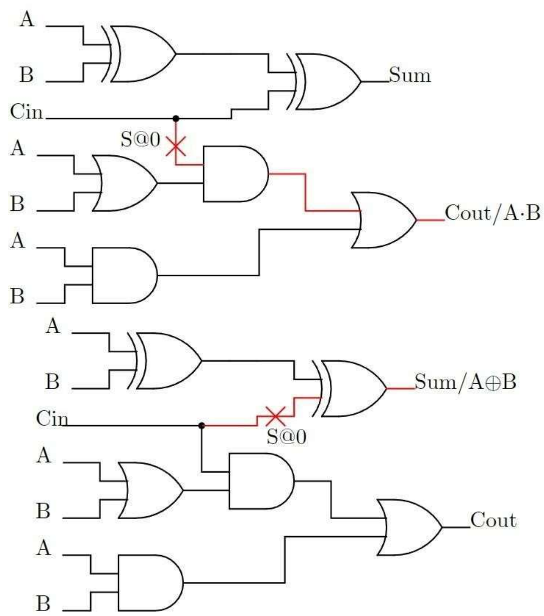
```
input a,b,cin; output sum,cout; assign sum = a^b^cin;
assign cout = a&b|cin&(a|b);
// initial begin
// $display("The correct adder");

input a,b,cin; output sum,cout; assign sum = a^b^cin;
assign cout = 1'b1&b|cin&(a|b);
// initial begin
// $display("The incorrect adder with and0 having in1/1");
```

IV. GENETIC ALGORITHM

Genetic Algorithm(GA) is one of the evolution algorithms and take inspiration from nature and evolution of species [7]. In context of ATPG, GA in generally used to find the test cases that can produce a high fault coverage of the fault model [5], [8]–[10]. Here we would like to define certain terms in context of GA and ATPG. Genetic Algorithm as three mains parts to it, (1) Population generation/selection and crossover, (2) calculate the fitness for the population and (3) add mutation to the population. In this project we would make a distinction between population generation and selection as they have different meaning. In the next section we will discuss each part of the algorithm.

A. Population generation

Most algorithm population generation means to generate the fist generation of members, so, only once a population is generated. The generation of the initial population can be random [8]–[10] or can be based on the given model of choice [5], where we calculate the certain parameters and then generate our initial population. In this project we would like to present a method to generate population by assigning weights to bits of the 8-bit binary value member. The weights represent the probability of occurrence of binary 1 in a particular bit. For a 8-bit number let the weights be $W_0, W_1, ..., W_7$. The value of each weight is between 0 to 255, i.e. $0 \leq W_n \leq 255$



or all $0 \leq n \leq$

MSB.
7 when 0 represents the LSB and 7 represents the

There is no need to chose a particular limits for the weights. The logic still holds for other values but we chose these as they represent the lowest and highest positive values of a 8-bit number. This is easy to implement in verilog, which we need to for testing our model, than other values. The weight represents the probability of occurrence of binary 1 for that bit. For example $W0$ represents
the probability of occurrence of binary 1 and $255 - W0$ represents
Fig. 6. The fault in Figure 5 is split into two single stuck-at-faults

In this project we will use the single stuck-at-fault models of the 8-bit ETC as our training model and multiple stuck- at- faults models of the 8-bit ETC as our testing model. The implementation is done using verilog HDL. We will first design an error free full adder module named full_adder and a full adder module named incorr_full_adder that has a single stuck-at-fault model of the full adder. We will have 28 such faulty full adder models. Then in the test bench we will instantiate one of the 8 full adder as incorr_full_adder and the rest as full_adder. At the probability of occurrence of binary 0 in bit 0 or LSB. Using these weights we can generate a population of a particular size. For the initial population we will chose the weights at random and later we determine the weights of each bit calculating there probabilities after the members for the next generation are determined.

There is also a draw back for this method. Lets say that we need to generate a population of size 10 where each member is unique using the weights $W1 \neq 0$ and $Wn = 0$ for $n = 1$ is impossible as there are only 2 possibilities. In GA not all members have to be unique but in the context of ATPG the uniqueness of members are important. To avoid this we may to consider the previous generation and perform the calculations again till we generate the new population. If it is the first generation then we have to generate a different set of weights.

B. Population selection

In a genetic algorithm, population selection refers to the process of selecting individuals from a population to undergo genetic operations such as crossover and mutation. The goal of population selection is to choose individuals that have higher fitness values or better solutions to the problem in hand, in order to guide the evolution of the population towards improved solutions over time. there are varies methods that are used in GA which enable us to select the best set of members that have to be crossed over.

Selection can be done in many ways. We may select the top best or most fit individual members of the population for crossover. This is called *Elitism*. This is most simplest form of selection and works for most of the time. In *Binary tournament selection* we select two individuals, at random or it can be weighted selection, and a better individual is selected of the two. The better individual can be better in terms of there fitness or some other metric. This is like a tournament between two individuals and we keep performing these tournaments till we meet the required population.

Fig. 7. A baised roulette wheel for Roulette wheel selection

Another method, known as *Roulette wheel selection*, a proportionate selection method in which the slots of a roulette wheel(Figure 7) are sized with the fitness of a individual. A ball is rolled over the wheel and the individual on which the ball lands is selected. To put it differently we are talking about a biased/ unfair roulette wheel, where each individual has the probability of being selected for next generation crossover is equal to its respective fitness value. Higher the fitness value higher the chances of selection. This brings in diversity in the population unlike in *Elitism* and over generations we see a much better population. As the low fit individuals may be the solutions to a few cases that can't be detected by high fit individuals. In this project we use the *Roulette wheel selection* method.

Stochastic universal selection method is similar to a *Roulette wheel selection* and will happen in a single turn of the wheel all required individuals are selected. We use markers(Figure 8), equal to size of the population, that are equally spaced around the roulette wheel. We turn the wheel and where the marker points to that individual is selected. If two maker points to the same individual then that many copies of that individual is taken. This can't be used for this reason of copies of same individual are being selected. Our individuals in the population have to be unique.

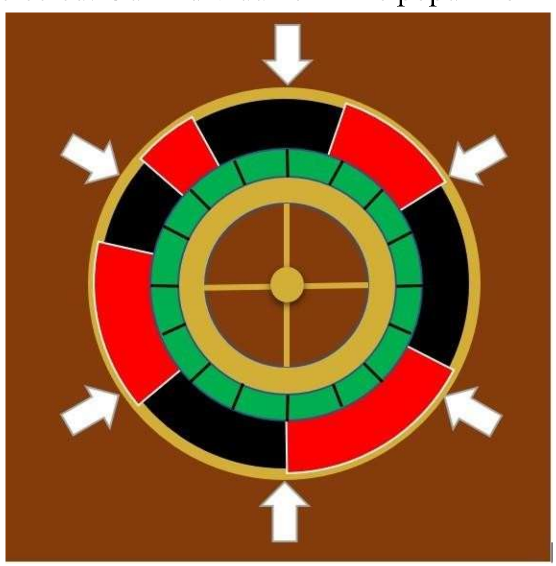


Fig. 8. Roulette wheel with pointers for Stochastic universal selection

C. **Crossover**

Crossover is inspired from crossing over of chromosomes in the process of cell division called Meiosis, that occur in organisms to produces gametes. This crossover is what brings about variations in the offsprings. In GA we need to perform crossover between the two selected individuals. In *One or two point crossover* we select one or two positions in the individual binary code and flip the bits from that position onwards. Lets say we have a binay code of length *L*. We need to perform a *one point crossover* to that code, say at position *m* for 1 ≤*m L*.≤We keep the code of the two respective individuals sames till the position *m* and fill all the bits from position *m*+1 to *L*.

In *uniform crossover* each position of the code has the same probability of being flipped. typically the probability is 50%. So, there is a 50% chance that a bit in a position may be flipped. Don't confuse this with mutation, where a bit is replace with it's complement there, here we just interchange the bits between two individuals. In this project we won't go for the crossover function of the GA. The reason is that the generation of individuals in our case is very different. If we perform crossover it makes no difference in the probability of occurrence of 1s and 0s. So the weights determined are still the same before and after crossover is done. So we might as well not perform this step.

D. **Fitness function**

A fitness function is a function that assigns a fitness value to each individual in a population in a genetic algorithm. It quantifies how well an individual solution performs or fits the problem being solved. The fitness value is typically a numerical representation of the quality or suitability of the individual. A fitness function is the most important function in GA it determines the run time, memory usage and other performances metrics to a large extent. We will have a discussion on the varies methods used to evaluate the fitness of a population.

In [5] the goal is to detect delay faults. Two vectors are required to evaluate the fitness of a vector-pair, so a pair of individuals is consider of fitness from the population. The first vector is evaluated and stored in the Global Record Table(GRT) and then the second vector is applied and the slow-to-rise and slow-to-fall faults conditions are evaluated by comparing the transitions of the signal to the previous vector. The vectors are also detecting thestuck- at-faults so if a pair has same fitness they are also subject to

the number of stuck-at-faults they detect.

In [8], the fitness function is evaluated in 4 phases. In phase 1 we generate a test vector and see weather it set all the flip-flops. If it does we move to phase 2 else we generate a new test vector. In phase 2, the number of fault detected is calculated for each individual. To differentiate vectors that have the same fitness, in addition to number of faults detected the number of faults that propagates to flip-flops are also calculated. If the vectors detect no additional faults, phase 3 is initialized. In phase 3, the noncontributing vectors are calculated. Vectors that activate more faults and propagate more fault effects will have a higher fitness. If we don't detect any additional faults we will check if the faults coverage is to a satisfying level, we end the program.

In [9], the fitness is calculated by evaluating two factors, 'Fault- excitability' and 'Fault-drivability'. For a vector, Fault-excitability means the fitness for setting the logic value opposite to the faulty value on the selected target fault point. The Fault-drivability mean the fitness for propagating the fault value D and D to any primary output from the fault point. The use of real-value simulation is also a unique. This simulation allows us to evaluate the logical circuit in terms of the probability of there respective gates.

In [10], to evaluate the fitness of an individual a linear combination with three components is used. The first component evaluates the ability of an individual to excite necessary value on victim line. The second component evaluates an individual ability to propagate the cross talk fault to the primary output. The third component evaluates the individual's ability to take into account the effect of aggressor lines effect. We take a weighted linear combination of the three components of the fitness function to evaluate the fitness.

In this project we will use a simple fitness function that evaluates the total fault coverage of the population in the single-stuck-at-fault model.

E. Mutation

Mutation occurs in organisms is a very common thing. A mutation in a genetic makeup could be harmful if that could lead to cause of some disease in the organism. But over generations mutations that a useful remain in the population and increase the fitness of a population through generations. Mutation is a random change in the DNA sequence of an organism. Mutation is a genetic operator in a genetic algorithm that introduces random changes to the individuals within a population. It helps introduce new genetic material into the population and promotes exploration of the search space. A harmful change reduces the fitness of a well fit individual. To avoid this the probability of occurrence of mutation is kept low. If it is too low the necessary variations in the population is not seen.

In this project we will use double mutation causing probability [9]. In this scheme we have two probabilities, probability Pa, that is the probability of occurrence of mutation in an individual of a population, and probability Pb, that is the probability of occurrence of mutation in that individual. To put it differently, probability Pa is used to see whether an individual in a population should undergo mutation and probability Pb is used to see whether a bit in that individual has to be inverted.

F. GA Implementation

The flow for the GA implementation is show in Figure 9. The final results are noted and the progress of the GA is also noted and is discussed in section VI of this report.

V. ATPG TESTING

For testing our ATPG vectors we will induce faults to ETC using the following approach. Each of the eight full adders will have a

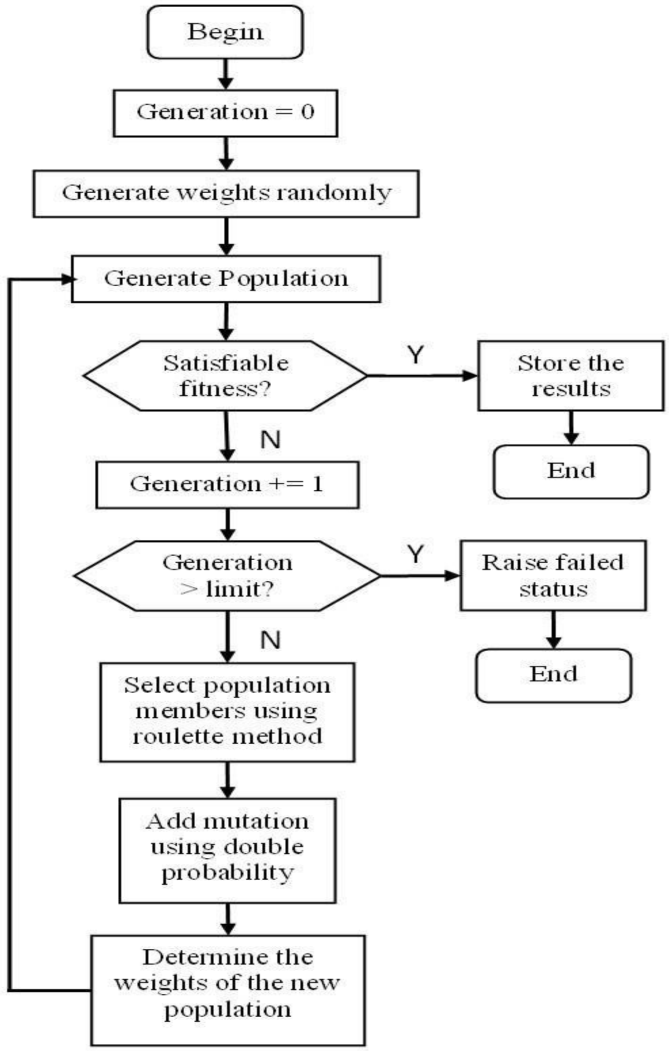


Fig. 9. Flow of proposed genetic algorithm implementation

50% chance of being faulty and 50% chance of being fault free. This allows us to test both fault free and faulty at a same time in a more realistic sense. The Figure 10 show the flow that is used for testing the ATPG. We initialize our variable for number of models that we need to test, number of faults models detected and those that were escaped. We then generate a model for ETC which may be fault free or faulty. If the faulty model is generated is already tested, we will generated a new model. If the model is not already tested we will increment models variable. We will initialize a variable K to keep track of number of vectors generated. Now we generate vector using the weights for our GA implementation. We will generate only unique vectors per model and test. If the vector doesn't detect a fault then increment K and then we check if the number of vector generated is greater than the population size as we used in GA implementation. If its greater then we increment the fault escaped variable as the fault has escaped. If the vector detects the fault then increment the fault detected variable. If the models to test have reached the required limit we end the simulation. We calculate the metrics like Fault coverage Fc given by

$$Fc = \frac{\text{Faults detected}}{\text{Total faulty models generated}}$$

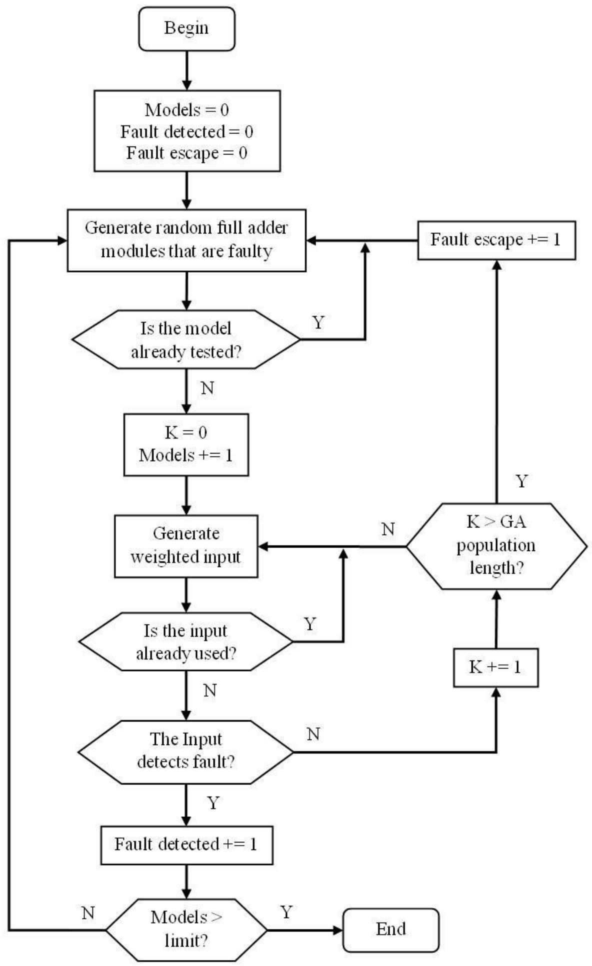


Fig. 10. Proposed ATPG based on GA

VI. RESULTS

In this section we will discuss the results obtained from both the GA implementation and testing. For first we go for the GA implementation. The size of the population is set to 15. The mutation rate and the individual selection rate is set at 5%. The fitness limit that we are setting to 98%. The generation limit is set at 1000 generations. The results obtained and metric of simulations are mentioned in Table I.

TABLE I

RESULTS OBTAINED FOR GA IMPLEMENTATION

'ime of execution	71.99 ms
itness achieved	00%
;eneration	48

The Table II show the percent mean and standard deviation of the fitness which is the fault coverage of the implementation over a sample space of 148 generations. For the ATPG testing we get a very promising result that the average fault coverage is about 97%. The Table III show the mean and standard deviation for ATPG testing.

The progression of the genetic algorithm is show in Figure 11.

As we can see from Figure 11 that there are cases where the population gave a fault coverage that is greater than 95%. There were about 32 of such instances. For testing our model we plot the fault coverage obtained for the number of test faults the test model

TABLE II

MEAN AND STANDARD DEVIATION FOR THE GA PROGRESS

Metric	Mean	Standard Deviation
'itness	5.1174	.0180

TABLE III

MEAN AND STANDARD DEVIATION FOR THE ATPG TESTING

Metric	Mean	Standard Deviation
'ault overage	7.6726	.4024
'ault scape	.3274	.4024

TABLE IV

MEAN AND SD FOR EACH BIT WHICH HAVE 95% OR MORE FAULT COVERAGE IN GA IMPLEMENTATION

Bit index	Mean	Standard Deviation
7	255	0
	58.50	2.0782
	41.38	9.9606
	41.24	7.0104
	81.59	2.6827
	78.15	9.2551
	5.56	4.3490
	50.82	4.1935

has detected. This can be done by taking the fault coverage for the testing scenarios (see Figure 12).

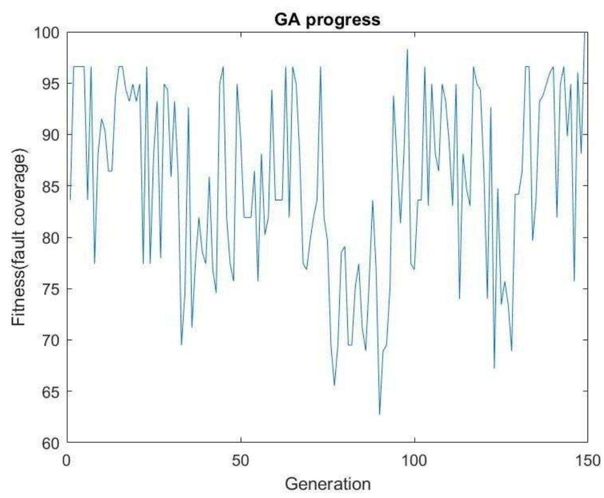


Fig. 11. Example for single stuck-at-fault in a full adder using GA

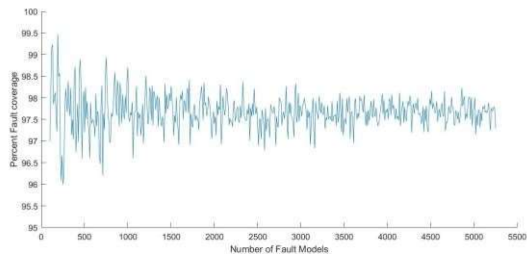


Fig. 12. The Fault coverage versus the number of test models used for testing

I. CONCLUSIONS

The Genetic proves to be an effective method to find test patterns for the testing of any combinational system. Our proposed fault modeling process is optimal to serve our purpose. The GA implementation was mostly manual in the sense that the selection of population size and mutation rates were chosen by intuition. We have achieved an execution time of 471.99 ms with 100% over 148 generations. The project has obtained a promising average fault coverage of 97% with a standard deviation of 0.4024 which is a substantial improvement in the filed of ATPG testing.

REFERENCES

[1] Zhen, Hui-Ling, et al. "Conflict-driven Structural Learning Towards Higher Coverage Rate in ATPG." arXiv preprint arXiv:2303.02290 (2023).

[2] Manjunath, T. D., and Biswajit Bhowmik. "Qusaf: A fast atpg for safs in vlsi circuits using a quantum computing algorithm." 2022 IEEE 19th India Council International Conference (INDICON). IEEE, 2022.

[3] Gaber, Lamya, Aziza I. Hussein, and Mohammed Moness. "Fault Detection based on Deep Learning for Digital VLSI Circuits." Procedia Computer Science 194 (2021): 122-131.

[4] Jain, Ayush, M. Tanjidur Rahman, and Ujjwal Guin. "Atpg-guided fault injection attacks on logic locking." 2020 IEEE Physical Assurance and Inspection of Electronics (PAINE). IEEE, 2020.

[5] Y. A. Skobtsov, "Genetic test generation for single and multiple crosstalk faults," in 2020 Wave Electronics and its Application in Information and Telecommunication Systems (WECONF). IEEE, 2020, pp. 1–5.

[6] A. Chitra, S. Vijay Murugan, and B. Sathiyabhama, "Detection of stuck open and short fault in sram based system," in Proceedings of the International Conference on Intelligent Computing Systems (ICICS 2017–Dec 15th-16th 2017) organized by Sona College of Technology, Salem, Tamilnadu, India, 2017.

[7] S. Sindia and V. D. Agrawal, "Defect level constrained optimization of analog and radio frequency specification tests," Journal of Electronic Testing, vol. 31, pp. 479–489, 2015.

[8] T. Arslan and M. O'Dare, "A genetic algorithm for multiple fault model test generation for combinational vlsi circuits," in Second International Conference On Genetic Algorithms In Engineering Systems: Innovations And Applications. IET, 1997, pp. 462–466.

[9] E. M. Rudnick, J. H. Patel, G. S. Greenstein, and T. M. Niermann, "Sequential circuit test generation in a genetic algorithm framework," in Proceedings of the 31st annual Design Automation Conference, 1994, pp. 698–704.

[10] T. Hayashi, H. Kita, and K. Hatayama, "A genetic approach to test generation for logic circuits," in Proceedings of IEEE 3rd Asian Test Symposium (ATS). IEEE, 1994, pp. 101–106.

[11] Shashidhara H. R., "Test vector Generation using different logic regression method," 2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT), Bangalore, India, 2016, pp. 736-742, doi: 10.1109/ICATCCT.2016.7912097.

[12] J. H. Holland, Adaptation in natural and artificial systems: an intro- ductory analysis with applications to biology, control, and artificial intelligence. MIT press, 1992.

[13] E. S. Park and M. R. Mercer, “An efficient delay test generation system for combinational logic circuits,” in Proceedings of the 27th ACM/IEEE Design Automation Conference, 1991, pp. 522–528.

[14] T.V. Naveen, M.V. Latte, Sathish Shet and H.R.. Shashidhara, "Design and Implementation of Test Vector Generation Using Random Forest Technique International", Journal for Research in Applied Science & Engineering Technology (IJRASET), vol. 4, no. 08, Aug 2017.

[15] T. M. Storey and W. Maly, “Cmos bridging fault detection,” in Proceed- ings. International Test Conference 1990. IEEE, 1990, pp. 842–851.

[16] J. A. Waicukauski, E. Lindbloom, B. K. Rosen, and V. S. Iyengar, “Transition fault simulation,” IEEE Design & Test of Computers, vol. 4, no. 2, pp. 32–38, 1987.