FAULT EMULATION TECHNIQUES FOR LOGIC LOCKING AND MULTI-CYCLE TEST GENERATION

by
Cemil Cem Gürsoy

Submitted to Graduate School of Natural and Applied Sciences
in Partial Fulfillment of the Requirements
for the Degree of Master of Science in
Computer Engineering

Yeditepe University
2017

FAULT EMULATION TECHNIQUES FOR LOGIC LOCKING AND MULTI-CYCLE TEST GENERATION

APPROVED BY:

Prof. Dr. Sezer Gören Uğurdağ ……
(Thesis Supervisor)

Assoc. Prof. Dr. Hasan Fatih Uğurdağ ……

Assist. Prof. Dr. Onur Demir ……

DATE OF APPROVAL: ..../..../2017

ACKNOWLEDGEMENTS

It is with immense gratitude that I acknowledge the support and help of Prof. Dr. Sezer Gören Uğurdağ. Pursuing my thesis under her supervision has been an experience which broadens the mind and presents an unlimited source of learning. I participated in a project titled "Sayısal Yonga Çoklu-Darbeli Üretim Testleri İçin Hata Emülasyonu Yönteminin Geliştirilmesi" supported by TÜBİTAK under the contract number 114E022 which enabled this thesis to be accomplished. I am grateful for being supported by TÜBİTAK thoroughout my master study.

I thank Abdullah Yıldız for testing and debugging many problems that were encountered during the project as well as implementing the multi-cycle test generation method proposed in this thesis on Zynq (all programmable SoC).

Finally, I would like to thank my family for their endless love and support, which makes everything more beautiful.

ABSTRACT

FAULT EMULATION TECHNIQUES FOR LOGIC LOCKING AND MULTI-CYCLE TEST GENERATION

The testing phase for testing the actual, post-manufactured chips is what is referred as test or testing in digital system design industry. Recently, multi-cycle tests that offer high test quality have been proposed. Multi-cycle tests are accomplished by feeding the input vector constantly and putting the circuit in functional mode for multiple cycles. Multi-cycle tests are often needed by partial-scan circuits and circuits with multiple clock-domains. In addition, the VLSI Test community is interested in multi-cycle tests because they can reduce the test time and cost by detecting more faults with the same test vector. However, the fault simulation of multi-cycle tests is computationally expensive. In literature, no fault emulation method for multi-cycle tests has been proposed yet. In this thesis, a new multi-cycle test generation algorithm is proposed and its fault emulation method is developed. With the help of our fault emulation method, we accelerate the process of multi-cycle test generation. In our multi-cycle test generation method, dynamic single fault activation technique has been used. Later, this method is modified to enable multiple fault activation, which is then applied to another computationally expensive problem. The process of determination of key gate locations in the logic locking problem in hardware security is accelerated by the second emulation method proposed in this thesis. The effectiveness of both emulation methods is evaluated on the ISCAS'89 benchmark circuits and results are presented.

ÖZET

LOJİK KİLİTLEME VE ÇOKLU-DARBELİ TEST ÜRETİMİ İÇİN HATA EMÜLASYONU TEKNİKLERİ

İmal sonrasındaki çiplerin test aşaması, dijital sistem tasarımı endüstrisinde kısaca test veya test yapmak olarak geçer. Son yıllarda, yüksek test kalitesi sunan çoklu-darbeli testler önerilmiştir. Çoklu-darbeli testler, devre fonksiyonel durumda iken giriş vektörünün devreye birden fazla saat darbesi süresince beslenmesiyle yapılır. Kısmi-tarama devreler ve çoklu saat alanı bulunan devreler için genellikle çoklu-darbeli testler gerekmektedir. Bunların yanı sıra, çoklu-darbeli testler, test süresini ve maliyetini düşürebileceklerinden ve aynı test vektörü ile daha fazla hata yakalamaya olanak sağladıklarından, sayısal entegre devre sektöründe ilgi uyandırmışlardır. Fakat, çoklu-darbeli testlerin oluşturulmasında kullanılan hata benzetimi yüksek hesaplama gerektirmektedir. Literatürde, çoklu-darbeli testler için henüz bir hata emülasyonu yöntemi önerilmemiştir. Bu tezde, yeni bir çoklu-darbeli test seti üretim prosedürü ve bu prosedürü kullanan hata emülasyonu yöntemi geliştirilmiştir. Geliştirilen hata emülasyonu yöntemi ile çoklu-darbeli test üretimini hızlandırmaktayız. Çoklu-darbeli test üretim yöntemimizde, dinamik tek hata aktivasyonu tekniği kullanılmıştır. Daha sonra, bu metod, modifiye edilerek çoklu hata aktivasyonuna olanak sağlayacak hale getirilmiş ve bir diğer yüksek hesaplama gerektiren problem için uygulanmıştır. Donanım güvenliğinde, lojik kilitleme için gereken anahtar kapılarının yerlerinin belirlenmesi, bu tezde geliştirilen ikinci emülasyon tekniği ile hızlandırılmıştır. Geliştirilen emülasyon tekniklerinin etkinliği ISCAS'89 karşılaştırma devreleri üzerinde denenmiş ve sonuçları sunulmuştur.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ATE	Automatic test equipment
BRAM	Block RAM
CI	Circuit instrumentation
CIDFI	Circuit instrumentation with dynamic fault injection
CIDMFI	Circuit instrumentation with static fault injection
CISFI	Circuit instrumentation with static fault injection
CSA	Carry save adder
CUI	Circuit under instrumentation
DFI	Dynamic fault injection
DMFI	Dynamic multiple fault injection
FIE	Fault injection element
FPGA	Field programmable gate array
HD	Hamming distance
IC	Integrated circuit
ICAP	Internal configuration access port
LUT	Lookup table
ncd	Native circuit description
P&R	Place and route
PI	Primary input
PL	Programmable logic
PO	Primary output
PR	Partial reconfiguration
PRSFI	Partial reconfiguration
PS	Processing system
PUF	Physically unclonable functions
SFI	Static fault injection
SoC	System on a chip
VLSI	Very-large-scale integration
xdl	Xilinx description language

1. INTRODUCTION

Building large circuit structures or integrating billions of transistors at very small dimensions raises its own set of challenges. The precise transfer of circuit layout on to a silicon wafer using lithography is difficult as dimensions of the transistors are becoming much smaller than the wavelength of the optical sources. Thus, faults manifest themselves as permanent defects in wires and transistors during the complex manufacturing steps. Therefore, the manufactured chips must be individually tested and verified against flaws before they are shipped to the customers. The testing phase for testing the actual, post-manufactured chips is what is referred as test or testing in digital system design industry. A test set is prepared ahead of time by the test-generation process. Test generation is done using a model of the circuit under test. After the manufacturing process, the test vectors are applied to the chips by means of a tester (automatic test equipment, ATE). A tester is basically a special computer on which a test program runs. Exhaustive testing requires exponential number of tests, which causes the test time to grow exponentially. A set of algorithms and methods have been proposed to help reduce of the number of test vectors by selecting them more wisely than just trying every combination. *Fault Simulation* is the most important test method where it is decided if a test vector is worth keeping and needed for test quality, however, it is computationally very complex. Many computationally expensive problems can be efficiently implemented on *Field Programmable Gate Array (FPGA)* platforms and run faster than it runs on a CPU. *Fault Emulation* can be done on such reconfigurable computing platforms so that fault simulation can be accelerated.

The tests that will be applied to the chip are prepared before fabrication, when the design at gate-level. Gate-level design, must be designed to be testable. To do that, the circuit must be considered as a *Finite State Machine (FSM)*. After the fabrication, state values of the chip in an unknown state. Therefore, when an input pattern applied to the chip, it is not possible to know beforehand what the outputs will be. For this reason, to test an FSM, state values must be controllable and observable. To do that, flip-flops (FF, i.e. state variables) of the circuit must be converted to scan flip-flops. In Figure 1.1, a simple FSM is shown with one register (FF). The flip-flop denoted by the block R is converted to a scan flip-flop. Combinational part of the circuit is represented by the block C. Scan flip-flops must also be connected as a chain to form a shift register. This shift register is called scan chain and when all state

variables are included in the scan chain(s) then the circuit can be called a full scan circuit. Beginning and end points of a scan chain (scan-in, scan-out) and the test-enable pin of the scan flip-flops, must be accessible from outside of the chip, so that the tester can change the state values. Test-enable input of the scan flip flops are all connected together. Thus, the chip can be switched to test mode from functional mode using a single input pin.

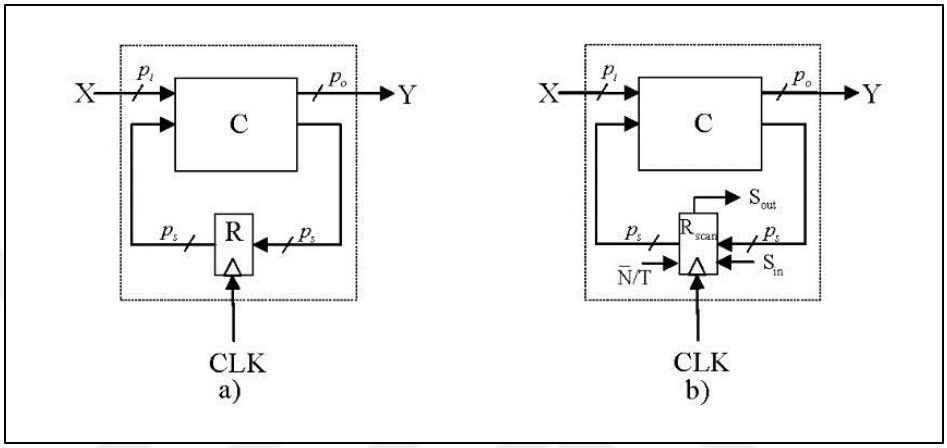

Figure 1.1. An overview of a FSM. a) With a regular flip-flop, b) The flip-flop converted to a scan flip-flop to make the circuit testable.

A test vector consists of primary inputs and state values of the FSM. Values of the state variables are set by using scan-in pin when the chip is in test mode. After initialization of the circuit (i.e. state values and primary inputs from the test vector is applied), primary outputs can be captured. In order to capture next-state values, FSM is set to functional mode from test mode using test-enable pin. After one clock cycle in functional mode, next-state values of the circuit can be captured through the scan-out pin of the chip. To obtain state values, the FSM is set back to test mode to shift out state values using scan-out pin.

In order to have better fault coverage, number of test vectors can be increased. However, this will cause longer test application time. Testing millions of chips time consuming and therefore an expensive task. For this reason, significant percentage of the chip cost is due to testing. Recently, multi-cycle tests that offer high test quality have been proposed [23, 24]. Multi-cycle tests are accomplished by keeping the circuit in functional mode for multiple cycles. This can cause effects of some of the faults propagate to the outputs that previously

did not. Multi-cycle tests are often needed by partial-scan circuits and circuits with multiple clock-domains. In addition, the VLSI Test community is interested in multi-cycle tests because of the fact that they can increase fault coverage or reduce the test application time and cost by compaction or compression of the test set. However, the fault simulation of multi-cycle tests is computationally expensive. This thesis presents an efficient algorithm that generates a multi-cycle test set by optimizing a single-cycle test set for both the fault coverage and the test application time. In this work, stuck-at, bridging and transition faults are considered and a fault emulation technique for the proposed algorithm is developed.

Additionally, *Integrated Circuit (IC)* industry is vulnerable to [9,17] serious threats. Hardware Trojans, *Intellectual Property (IP)* piracy and IC overbuilding, reverse engineering, side-channel analysis, and counterfeiting are the possible hardware-based threats that can be encountered in anywhere of the IC supply chain. Currently, there is an ongoing research effort [11] on hardware security how to systematize the hardware security knowledge in terms of the hardware-based attacks, countermeasures, metrics for evaluation, and as well as terminology [13]. In order to prevent hardware Trojans, IP piracy and IC overbuilding, reverse engineering, and counterfeiting, hardware security techniques are proposed in the past. Based on the terminology given in [13], the related work in this thesis is classified as *obfuscation* and *logic locking* based techniques. Obfuscation approach can be found in the prior works [4,3,7,8]. In these approaches, obfuscation is realized in the design phase, basically the RTL. *Physically Unclonable Functions (PUFs)* [18] are used in [1,7,8] for key generation to make the obfuscation hard to reveal. Obfuscation hides the functionality, but renders the structure unintelligible [2,13]. Logic locking [15,14,13] on the other hand, preserves the structure by rendering the circuit temporarily unusable [13], until it is unlocked. As opposed to obfuscation, logic locking is realized in the implementation phase by inserting key-gates into the original netlist. Upon applying a correct (wrong) key, a circuit is logic unlocked (locked).

In this thesis, fault emulation techniques are developed and used in two different applications. The first application area is hardware security. Logic locking is an IC protection method done by inserting locks in various locations of the circuit. Determining these locations is computationally expensive process. For this reason, the fault emulation techniques that developed in this thesis is applied to accelerate this process. The other application area, multi-cycle test generation is accelerated using fault emulation. Also an

algorithm that produces a multi-cycle test set by optimizing a single-cycle test set for fault coverage and test application time while considering stuck-at, bridging and transition faults at the same time is developed. In Figure 1.2, contributions that are made are shown in yellow.

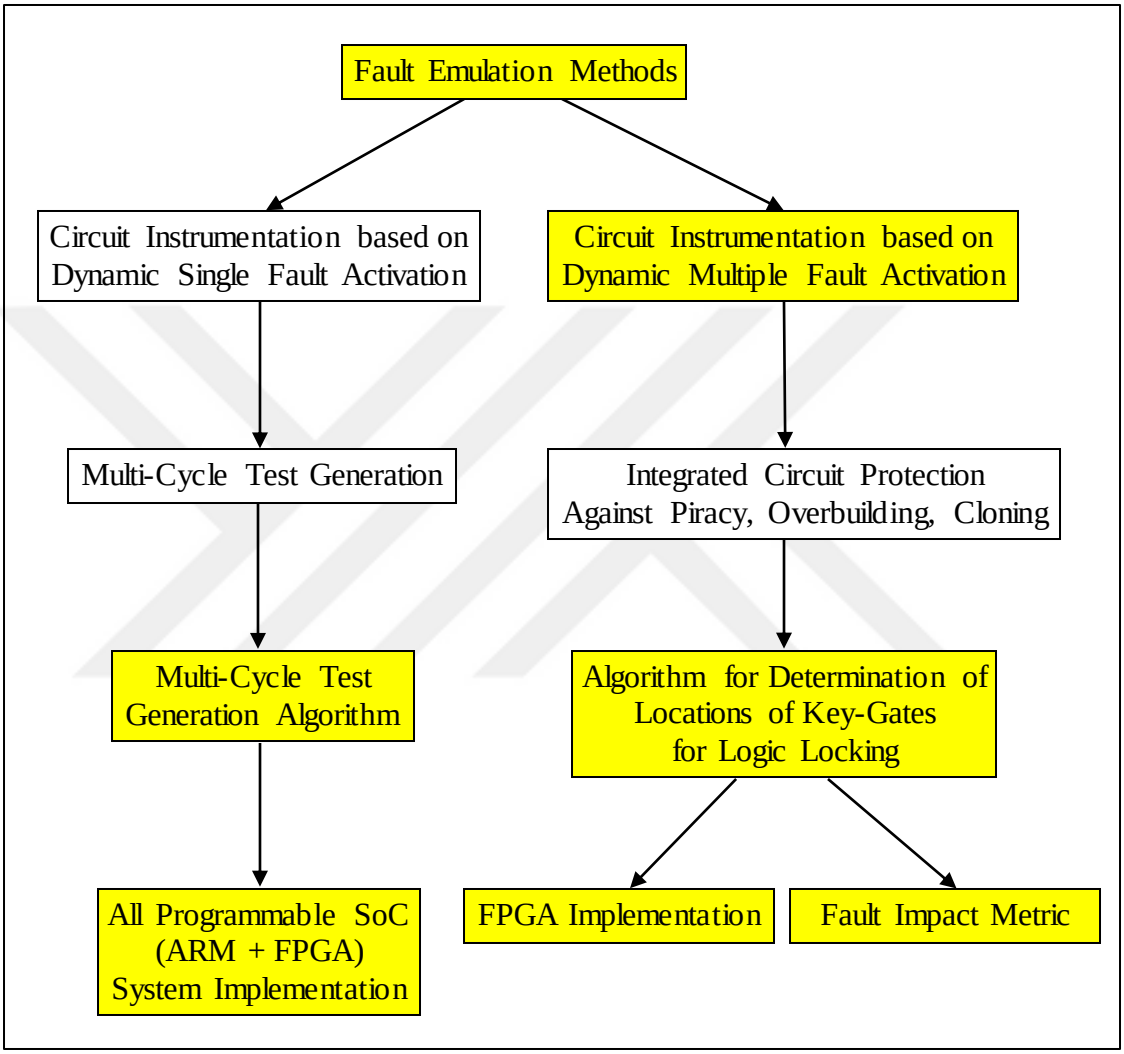

Figure 1.2. Contributions of the thesis.

2. PREVIOUS WORK

In this section, the state-of-the-art techniques developed for fault emulation, fault analysis based logic locking and multi-cycle test generation are explained. Also, the notations and definitions used in this thesis are given.

In [14], instead of inserting key-gates at random locations in the design, an IC testing approach including fault-analysis techniques such as fault activation, fault propagation, and fault masking is used for stronger logic locking. A metric for logic locking is also proposed in [14] in order to enable a designer to have control over the corruption effects of a logic locking. Both XOR/XNOR gates and multiplexers (MUXs) are used for insertion during logic locking. The major drawback of this technique is that it uses fault simulation to assess the fault impacts. However, fault simulation is a very time-consuming task. Several tries have to be done to find out the best combination of fault locations and assess the corruption effects.

Due to the time-consuming nature of fault simulation, fault emulation techniques were proposed in the past. Unlike a fault simulator, a hardware emulator performs gate evaluation in parallel, which can provide real-time logic operation. Usually, a typical hardware emulator includes several boards, each having several FPGAs. An FPGA offers reconfigurability through thousands of logic elements that are connected by programming the interconnects. Fault emulation requires usual FPGA design flow such as circuit synthesis, mapping, Place & Route (P&R), and bitstream generation for programming the emulator.

Table 2.1. Comparison of fault emulation approaches.

Technique	**Synthesis**	**Map**	**P&R**	**Bitstream Generation**	**Reconf.**	**Memory**
CISFI	for each fault	for each fault	for each fault	for each fault (full)	for each fault (full)	for each fault (non-volatile)
PRSFI	once	once	once	for each fault (partial)	for each fault (partial)	for each fault (volatile and non-volatile)
CIDFI	once	once	once	once	none	none

Circuit Instrumentation (CI) and *Partial Reconfiguration (PR)* based techniques are the two common approaches for fault emulation of digital circuits on FPGAs. In CI-based techniques, the original netlist is modified such that extra logic gates that correspond to a fault are added to the circuit to be emulated. The efficiency of circuit instrumentation techniques depends on the way how the fault injection is realized. Fault injection can be either static or dynamic. Wieler et al. [23] proposed CI with *Static Fault Injection (SFI) (CISFI)* where the netlist is modified statically such that every time a fault is injected, full FPGA flow (compilation plus full reconfiguration) is repeated. Because of the lengthy overheads of the FPGA flow, efficient fault injection techniques [5,10] are proposed. In CI with *Dynamic Fault Injection (DFI) (CIDFI)*, the circuit is instrumented such that the circuit is compiled and configuration bitstream is generated only once.

In CISFI, a full configuration bitstream has to be loaded on FPGA for each fault. Since the reconfiguration time is dependent on the size of the configuration bitstream, PR-based techniques are considered in order to decrease the reconfiguration time by decreasing the size of the configuration bitstream. Usually, the size of a partial bitstream is less than the size of a full configuration bitstream whereas the size of a full configuration bitstream is constant (independent of resource utilization), but vary from FPGA family to family. In PR-based techniques [16], instead of netlist modification, the configuration bitstream is modified to inject faults and instead of a full configuration bitstream, a partial bitstream is loaded on the FPGA for each fault. Although PR avoids lengthy recompilation times, however, this technique requires additional partial reconfiguration times as well as memory to store the partial bitstreams. Another problem with the PR-based techniques is that the fault injection can only be realized statically, because a partial bitstream corresponding to a fault should be prepared in advance. The realization of multiple fault injection with PR-based techniques is also almost impossible due to the huge memory requirement to store the partial bitstreams for several fault combinations.

The comparison of fault emulation techniques in terms of compilation, reconfiguration, and memory overhead is given in Table 2.1. In Table 2.1, three fault emulation techniques are compared in terms of time spent in different steps of FPGA flow as well as required memory. Due to the static fault injection nature in PR technique, this technique is denoted as PRSFI in Table 2.1. Both CISFI and PRSFI require configuration bitstream generation for each fault where full (partial) bitstream is generated for each fault in CISFI (PRSFI). Both CISFI and

PRSFI require reconfiguration where full (partial) reconfiguration is required for CISFI (PRSFI). In addition, there is memory overhead in both CISFI and PRSFI. In CISFI (PRSFI), full(partial) bitstreams generated for each fault require additional nonvolatile memory. In PRSFI, the partial bitstreams must be loaded on RAM in order to reconfigure the FPGA partially at runtime. Therefore, PRSFI requires additional volatile memory to store the partial bitstreams. Based on this comparison, time and memory overhead in CIDFI approach is remarkably less than the others. Unlike PRSFI, CIDFI also does not require additional memory.

In this thesis, a fault emulation based logic locking technique to speed up the determination of the fault locations and fault impacts in real-time is proposed. First contribution of this thesis is *Dynamic Multiple Fault Injection (DMFI)* and *Circuit Instrumentation with Dynamic Multiple Fault Injection (CIDMFI)*. DMFI extends dynamic single fault injection such that multiple faults can be dynamically injected at runtime. CIDMFI is the modification of a netlist such that multiple faults can be activated dynamically at runtime. Second contribution of this thesis is to apply CIDMFI to logic locking. A code generator is written such that for a given netlist, it automatically generates an emulation circuit which includes the instrumented netlist with a controller, fault impact calculator, and a serial interface to communicate a PC. An FPGA board is configured with the generated emulation circuit for each benchmark from ISCAS'89. During the emulation process, the emulation circuit checks different fault combinations using a set of test patterns, calculates fault impacts, determines the fault locations as well as the key to unlock the circuit, and outputs the key via serial interface. Based on the key obtained from the emulation, the final logic locked netlist is generated with the help of another code generator.

2.1. FAULT EMULATION

Fault emulation is emulating a circuit in the presence of a fault. Comparing the fault emulation results with those of the fault-free emulation of the same circuit emulated with the same applied test, the faults detected by that test can be determined. With the increasing performance of FPGAs and logic emulation technology, hardware fault emulation systems have become not only feasible but also very efficient as compared with the existing software-based methods.

In order to do fault emulation, faults must be modeled and inserted to netlist of the circuit. In this thesis, three type of faults are implemented for fault emulation.

- Stuck-at 0 and stuck-at 1 faults
- Bridging fault
- Transition fault

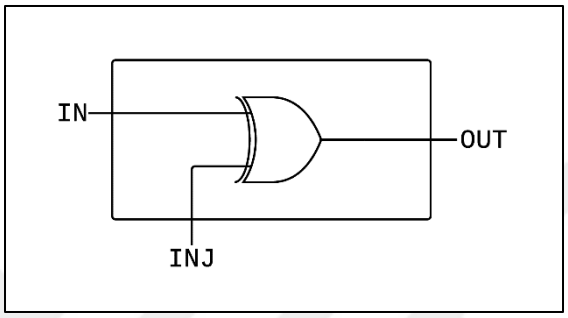

Figure 2.1. Stuck-at fault model [10].

A simple XOR gate is used to model stuck-at faults in [10] and shown in Figure 2.1. When INJ input is 1, fault is injected and input is inverted. When IN is 0 and INJ is 1, the model behaves as if there is a stuck-at 1 fault is present. On the other hand, when IN is 1 and INJ is 1, a stuck-at 0 fault is injected to that net. Bridging fault model has been proposed in [22] and shown in Figure 2.2. Table 2.2 shows the behaviour of this fault model. Complete model of this fault type is implemented in Section 3.2.1.

Bridging fault model has been proposed in [22] and shown in Figure 2.2. Table 2.2 shows the behaviour of this fault model. Complete model of this fault type is implemented in Section 3.2.1.

There are a number of methods to use a logic emulation system for fault grading. Wieler et al. [23] proposed a *serial fault emulation* algorithm that emulates one faulty circuit at a time sequentially. In serial fault emulation, the implementation of each faulty circuit is constructed from the fault-free circuit before the emulation process through SFI which requires reconfiguration of the emulator. The major drawback in SFI lies in the large amount of time spent in reconfiguration.

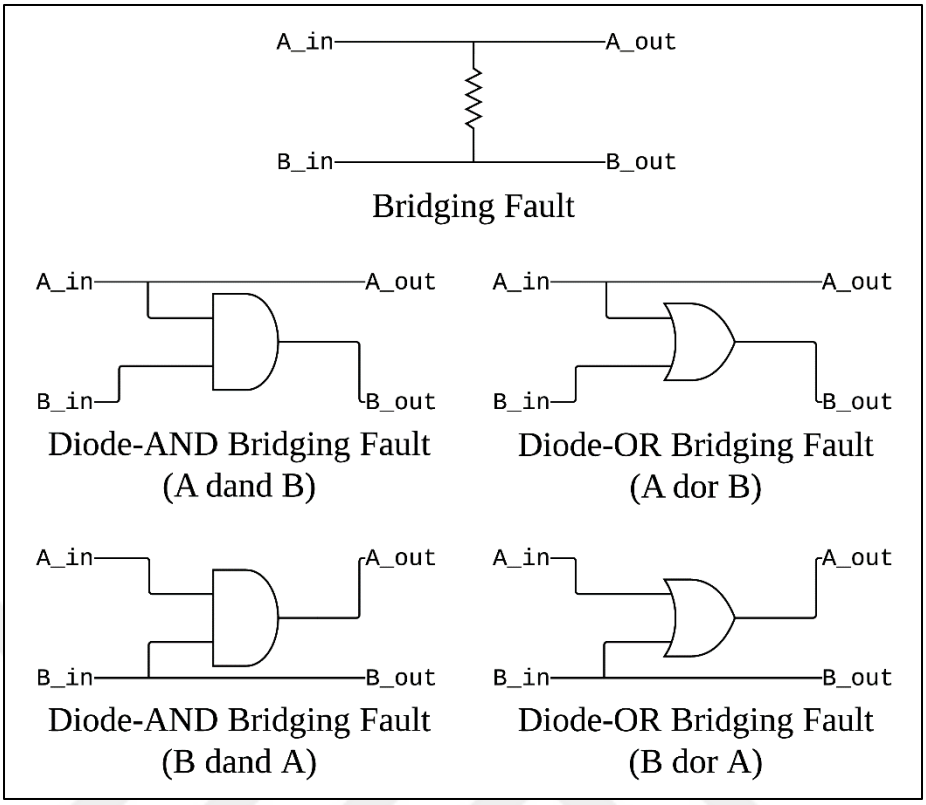

Figure 2.2. Bridging fault and Diode-AND/OR fault models.

Table 2.2. Truth table for Diode-AND/OR fault model.

In	**Out**				
	Fault-free	**A dand B**	**A dor B**	**B dand A**	**B dor A**
A_in, B_in	**A_out, B_out**	**A_out, B_out**	**A_out, B_out**	**A_out, B_out**	**A_out, B_out**
0 0	0 0	0 0	0 0	0 0	0 0
0 1	0 1	0 **0**	0 1	0 1	**1** 1
1 0	1 0	1 0	1 **1**	**0** 0	1 0
1 1	1 1	1 1	1 1	1 1	1 1

Let the number of faults, the number of patterns, the time required for compilation of the circuit (including the full bitstream generation), the time required for full configuration, and the critical path delay be denoted as N_f, N_p, T_{comp}, T_{config} and T_{pd} respectively. Then the total fault emulation time, which is denoted as T_{CISFI} considering CISFI is as Equation 2.1.

$$T_{CISFI} = N_f \times \left(T_{comp} + T_{config} + N_p \times T_{pd}\right) \quad (2.1)$$

Then the total emulation time required for fault location selection phase of logic locking algorithm for N_k keysize using CISFI is denoted as $T_{CISFI-LL}$ and calculated by Equation 2.2.

$$T_{CISFI-LL} = N_k \times \left(N_f \times \left(T_{comp} + T_{config} + N_p \times T_{pd}\right)\right) \quad (2.2)$$

In PRSFI, although lengthy recompilation times are avoided, there is still an overhead of partial bitstream generation and reconfiguration time which is denoted as $T_{reconfig}$. We conducted a simple experiment to compare $T_{reconfig}$ versus T_{config}. In this experiment, c17 from ISCAS'85 combinational benchmarks is used. C17 is first synthesized and then the *xdl* (Xilinx Description Language) representation from the *ncd* (Native Circuit Description) of c17 is extracted using *ncd2xdl* command from Xilinx ISE. We modified a LUT content by editing the *xdl* file. With *xdl2ncd* command from Xilinx ISE, we generated back the *ncd* of the faulty circuit. By using the difference-based PR approach and using the *bitgen* of Xilinx ISE, we generated a partial bitstream of 9KB. Assuming 400MBytes/s configuration speed through the *Internal Configuration Access Port (ICAP)* which is the fastest reconfiguration interface, $T_{reconfig}$ is about 0.023ms, whereas a full reconfiguration T_{config} is about 2s, considering Xilinx Zynq-7000 (XC7Z020-CLG484). Therefore, PRSFI offers better emulation times than CISFI. The total fault emulation time considering PRSFI is denoted as T_{PRSFI} in Equation 2.3.

$$T_{PRSFI} = T_{comp} + T_{config} + N_f \times \left(T_{reconfig} + N_p \times T_{pd}\right) \quad (2.3)$$

The total emulation time required for fault location selection phase of logic locking algorithm, which is denoted as $T_{PRSFI-LL}$, for N_k keysize using PISFI is given in Equation 2.4.

$$T_{PRSFI-LL} = N_k \times \left(T_{comp} + T_{config} + N_f \times \left(T_{reconfig} + N_p \times T_{pd}\right)\right) \quad (2.4)$$

Cheng et al. [5] proposed first DFI to reduce the reconfiguration time in fault emulation for fault grading whereas Lu et al. [10] later proposed DFI for fault diagnosis. Through the insertion of extra hardware to the circuit, DFI enables the emulation of multiple structural dependent faults within a single configuration. In this technique, a fault is activated every clock cycle. An example for dynamic single fault injection is shown in Figure 2.3b for ISCAS'89 sequential benchmark circuit s27 shown in Figure 2.3a. First step in CIDFI is to convert a sequential circuit into combinational circuit by removing the flip-flops (FFs), adding FF outputs as primary inputs (PIs), and FF inputs as primary outputs (POs). Second step in CIDFI is to add a single fault activation scan chain. In Figure 2.3b, a single fault activation scan chain is shown. The output of each FF in the single fault activation scan chain

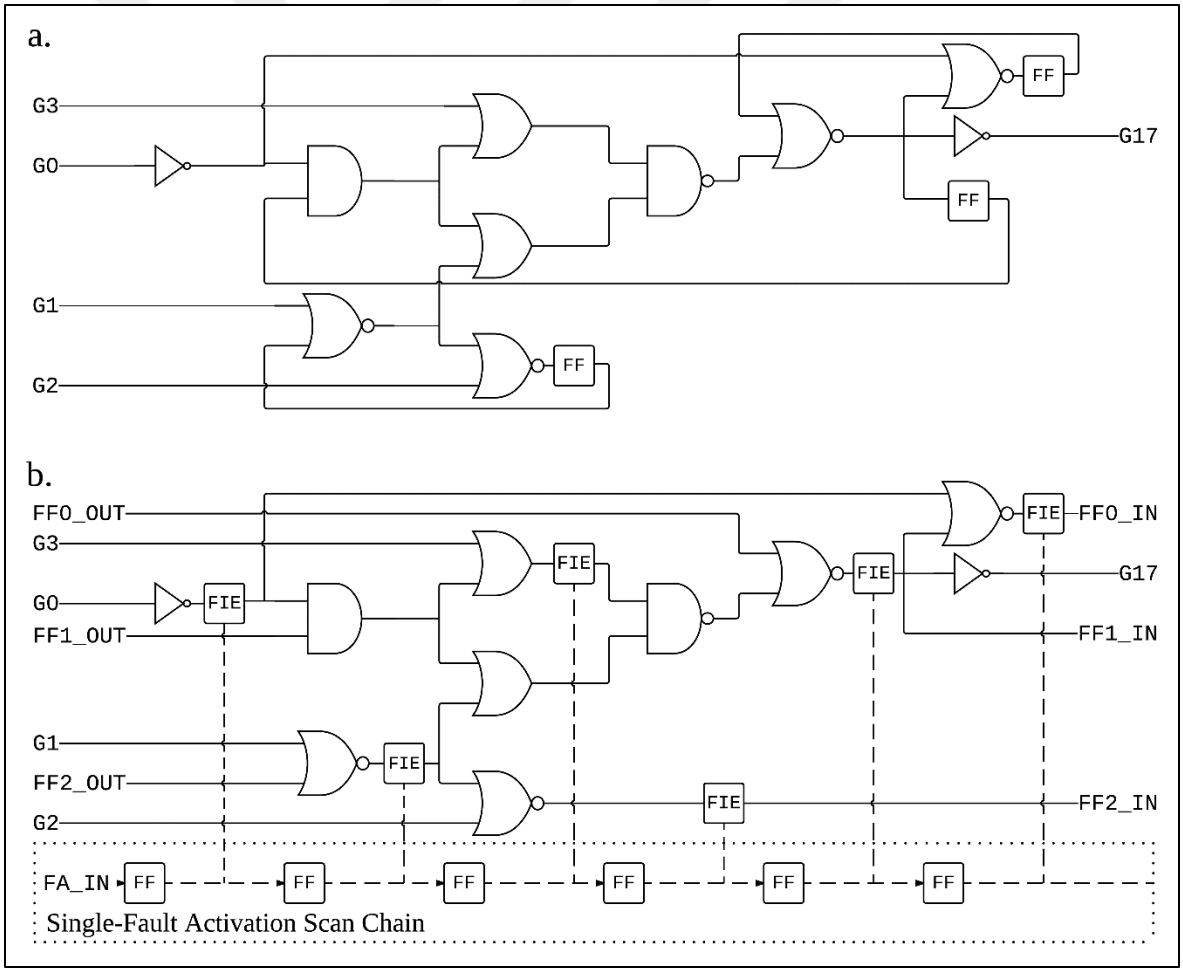

Figure 2.3. Dynamic fault injection. a. Original netlist of s27 b. Instrumented s27 with dynamic single fault injection.

is connected to the INJ terminal of its corresponding *Fault Injection Element (FIE)*. The number of FFs in the single fault activation scan chain is same as with the number of FIEs. An FIE that was formerly proposed in [10] is shown in Figure 2.1 where a single XOR2 gate implements s-a-0, s-a-1, and fault free cases. Note that the functionality of an FIE can be changed. In [14], both XOR2 and MUX based FIEs are used for the logic locking. A pulse through the scan chain activates FIEs one by one.

Once the instrumentation according to CIDFI is done, the instrumented netlist is compiled once and the FPGA is configured once. For every N_p clock cycles, an FIE is activated. For every cycle, a test pattern is fed to the circuit. Therefore, the total fault emulation time considering CIDFI is given as T_{CIDFI} in Equation 2.5.

$$T_{CIDFI} = T_{comp} + T_{config} + N_f \times N_p \times T_{pd} \tag{2.5}$$

In the case of fault location selection phase of the logic locking algorithm, when the netlist is instrumented with our CIDMFI, the netlist is compiled once and the emulator is configured only once. In Section 3.1, we will explain how our emulation technique based on CIDMFI works in order to speed up logic locking.

2.2. FAULT ANALYSIS BASED LOGIC LOCKING

In order to assess the impact of a fault, the same metric given in [14] is used.

2.2.1. Logic Locking Metric

Consider a digital circuit shown in Figure 2.4 with N_i-bit input, N_o-bit output locked with N_k-bit key bits. Let $B = \{0,1\}$ and $x \in B^{Ni}$ be a functional input. Let $y \in B^{No}$ be the correct output, $z \in B^{Nk}$ be a key, and $c \in B^{Nk}$ be the correct key.

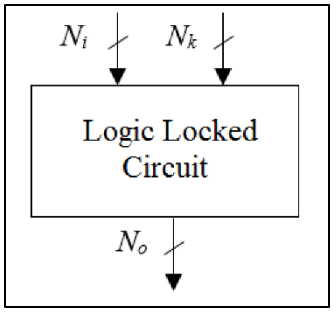

Figure 2.4. A logic locked circuit block

(i) A circuit f locked with a key c should produce correct outputs for all input patterns when the correct key c is applied.

$$f(x,z)|_{z=c} = y \quad \forall x \in B^{N_i}, y \in B^{N_o} \tag{2.6}$$

(ii) A circuit f locked with a key c should produce wrong outputs for all input patterns when a wrong key is applied.

$$f(x,z)|_{z\neq c} = y' \quad \forall x \in B^{N_i}, z \in B^{N_k}, y' \in B^{N_o} \quad where\ y' \neq y \tag{2.7}$$

(iii) Let the Hamming Distance between y and y' be denoted by HD(y, y'). Let P be the output bit combinations that an attacker is forced to consider corresponding to every input combination. If N_q-out-of-N_o output bits are wrong (i.e. HD(y, y') = N_q), then P can be computed as $\binom{N_o}{N_q}$. P is maximum when $N_q = N_o/2$ (i.e. when HD(y, y') = $N_o/2$).

Based on this metric [14], a logic locking technique should insert key-gates such that 50% of the output bits should be corrupted on applying a wrong key. An example key-gate is shown in Figure 2.1, originally proposed in [10]. In order to find out the best location to insert a key-gate, its *fault impact* must be calculated. Fault impact was defined in [14] as the total number of output bits that get affected by a fault for a given set of test patterns. In this thesis, we use the same definition in [14], but here we present its logical and arithmetic expression as follows.

2.2.2. Fault Impact

Let the N_o-bit output of the fault-free circuit f be denoted as $y^p_{N_o-1}y^p_{N_o-2}y^p_{N_o-3}\dots y^p_2 y^p_1 y^p_0$ and the N_o-bit output of key-gate g injected circuit be denoted as $y'^p_{N_o-1}y'^p_{N_o-2}y'^p_{N_o-3}\dots y'^p_2 y'^p_1 y'^p_0$ for pattern p. The sum of corrupted bits for pattern p is $\sum_{o=0}^{N_o-1} y'^p_o \wedge y^p_o$. Then the fault impact of key-gate g for circuit f and N_p patterns is given as the following:

$$Fault\ Impact(f,g) = \sum_{p=1}^{N_p}\sum_{o=0}^{N_o-1} y'^p_o \wedge y^p_o \tag{2.8}$$

2.2.3. Fault Simulation based Logic Locking

The determination of location of the key-gates of the algorithm proposed in [14] is given as follows:

Algorithm 2.1. Location Selection Phase of Logic Locking Algorithm [14]:

```
for i = 1 to keysize do
        foreach gate in netlist do
                Compute FaultImpact;
        end
        Select the gate with the highest FaultImpact;
        Insert key-gate and update the netlist;
        Apply test patterns;
end
```

There are two bottlenecks of this approach. The first one is the intensive usage of fault simulation for both fault impact computation as well as test pattern application. The second bottleneck is the modification of the netlist statically every time a key-gate is injected. We will present our solution in Section 3.1 and explain how our solution will speed up the location determination phase of logic locking algorithm given in [14].

2.3. MULTI-CYCLE TEST GENERATION

In [23], two algorithms proposed to generate a multi-cycle test set, out of a single or two cycle test set. Algorithms increased functional clock cycles of the tests, while keeping primary inputs constant. Generated multi-cycle test sets achieved higher fault coverage at the cost of slight increase in the test application time.

2.3.1. Pomeranz's Algorithm 1

First algorithm (Alg.#1) is given in Algorithm 2.2. The algorithm starts from compact one-detection single-cycle test set. The goal of this algorithm is to increase fault coverage for stuck-at, bridging and transition faults. However, since the test set was initially for stuck-at faults, stuck-at fault coverage were already at their maximum. For this reason, the algorithm

Algorithm 2.2. Defining an L-Cycle Test Set With a Target Fault Coverage [23]:

1) Let T_1 be a given test set. Let F be the set of target faults. Perform fault simulation with fault dropping of F under T_1. Let D_1 be the set of detected faults. For every $f \in D_1$, let $det1(f)$ be the index of the first test in T_1 that detects it.
2) Set $T_L = T_1$. For every test $t_i = < s_i, v_i > \in T_L$, duplicate v_i to form an L-cycle test.
3) Perform fault simulation with fault dropping of F under T_L. Let D_L be the set of detected faults. For every $f \in D_L$, let $detL(f)$ be the index of the first test in T_L that detects it.
4) For every $t_i \in T_L$, if D_1-D_L contains a fault f such that $det1(f) = i$.
    a) Remove half of the clock cycles of t_i by removing half of its primary input vectors.
    b) Move every fault f such that $detL(f) = i$ from D_L to F.
    c) Perform fault simulation with fault dropping of F under T_L . Update D_L and $detL(f)$ for every fault f whose detection information changes.
5) If D_1-$D_L \neq \emptyset$ go to Step 4).

tries to increase overall fault coverage (i.e. combined fault coverage of stuck-at, bridging and transition faults), while keeping stuck-at fault coverage the same. The reason for this, changing number of functional cycles of the test can cause loss of detection of some faults. Since we can already get maximum fault coverage for stuck-at faults in single-cycle, the algorithm starts from a high clock cycle amount, and it reduceses the cycles if the test looses detection of a stuck-at fault that already detected in single-cycle.

In this algorithm and the next one, T represents a test set and subscript of T indicates number of functional clock cycles. Initially all tests in the test set is single-cycle, hence it is denoted as T_1. The algorithm starts from a high functional clock cycle count L. In step 2, primary input vector (v) of each single-cycle test in T_1 held constant for L functional clock cycles to form the test set T_L. After that each test in T_L becomes L-cycle. However, later in step 4, the algorithm may remove some cycles from some of the tests in T_L. So, at the end of the algorithm, not all tests in the test set is L-cycle.

2.3.2. Pomeranz's Algorithm 2

Second algorithm from [23] (Alg.#2) starts from a two cycle test set for transition faults. The goal of this algorithm is to increase overall fault coverage. The algorithm can be summerized as follows:

For each test t in the test set,

(i) Find all faults that are detected by other tests while recording first test that detects each of them.

(ii) Find all faults that are detected by test t, and record that these faults are detected by test t.

(iii) Set all faults that are detected by test t and faults that are not detected by any test as target faults.

(iv) Find the best cycle for test t, such that number of detected faults is highest. Then record it as result.

Second algorithm from [23] is given as follows:

Algorithm 2.3. Defining an L-Cycle Test Set With Increased Target Fault Coverage [23]:

1) Let T be a given test set. Let F be the set of target faults. Set $T_L = T$. Perform fault simulation with fault dropping of F under T_L. Let D_L be the set of detected faults. For every $f \in D_L$, let detL(f) be the index of the first test in T_L that detects it.
2) For every $t_i = < s_i, v_i, ..., v_i > \in T_L$, apply the following steps.
   a) Duplicate v_i until t_i becomes an L-cycle test.
   b) Move every fault f such that $detL(f) = i$ from D_L to F.
   c) Perform fault simulation with fault dropping of F under $T_L - \{t_i\}$ followed by fault simulation with fault dropping of F under t_i. Update D_L and $detL(f)$ for every fault f whose detection information changes.
   d) Define $F_{targ} = \{f \in D_L : detL(f) = i\} \cup (F - D_L)$.
   e) Perform fault simulation of F_{targ} under t_i in order to find, for every fault $f \in F_{targ}$, the set of clock cycles $DET(f)$ such that f can be detected by t_i if it is turned into a $(u + 1)$-cycle test for every $u \in DET(f)$.
   f) For $u = 1, 2, ..., L - 1$, if $u \in DET(f)$ for every f such that $detL(f) = i$, compute $n_{det}(u)$ as the number of faults in F_{targ} such that $u \in DET(f)$.
   g) Of all the values of u considered in Step 2f), select the smallest one for which $n_{det}(u)$ is the highest.
   h) Remove clock cycles from t_i to change it into a $(u + 1)$-cycle test.
   i) Perform fault simulation of F_{targ} under t_i to update D_L and the values of $detL(f)$ for $f \in F_{targ}$.

3. PROPOSED METHODS

3.1. PROPOSED METHODS FOR LOGIC LOCKING

3.1.1. Dynamic Multiple Fault Activation

First step in CIDMFI is to convert a sequential circuit into a combinational circuit by removing FFs, adding FF outputs as PIs, and FF inputs as POs. Second step in CIDMFI is to insert an FIE at the output of a gate. Then, the third step in CIDMFI is to insert a *multiple fault activation scan chain*. A multiple fault activation scan chain consists of a regular shift register and MUXs. Both the number of FFs and MUXs in the chain are equal to the number of inserted FIEs. Each FIE is activated or deactivated by the output of the corresponding MUX. The select input to a MUX is connected to the dedicated *Chosen* bit. When *Chosen* bit is a "1", the corresponding FIE is activated. When *Chosen* bit is "0", the corresponding FIE is either activated or deactivated depending on the value at the output of the corresponding FF in the scan chain. *Chosen* input activates the FIEs that are already picked according to the logic locking metric and fault impact given in Equation 2.8. Together with the FIEs activated by *Chosen*, a pulse through the scan chain activates other candidate FIEs one by one in order to determine the next chosen FIE to be. In this paper, the next FIE with FaultImpact closest to $N_p \times N_o$ / 2 is selected as opposed to [14]. It was addressed in [14]

Algorithm 3.1. Proposed Location Selection Phase:

```
for i = 1 to keysize do
        foreach unchosen FIE do
          Activate;
          Apply test patterns;
                Compute FaultImpact;
                Deactivate;
        end
        Pick the FIE with FaultImpact closest to (Np × No) / 2;
        Set the FIE chosen;
end
```

that for larger benchmarks, fault masking affects the fault impact. Their solution was to pick a fault with the highest fault impact. Our proposed location selection phase of the logic locking algorithm is shown in Algorithm 3.1.

Next, we will explain the determination of fault locations based on the proposed fault injection mechanism on s27 example. A CIDMFI example is shown in Figure 3.1b for ISCAS'89 benchmark circuit s27 shown in Figure 3.1a. In Figure 3.1b, six FIEs are inserted. The benchmark s27 has originally 4 PIs, 1 PO, and 3 FFs. After converting s27 into a combinational circuit, the instrumented circuit has now 7 PIs and 4 POs. The exhaustive

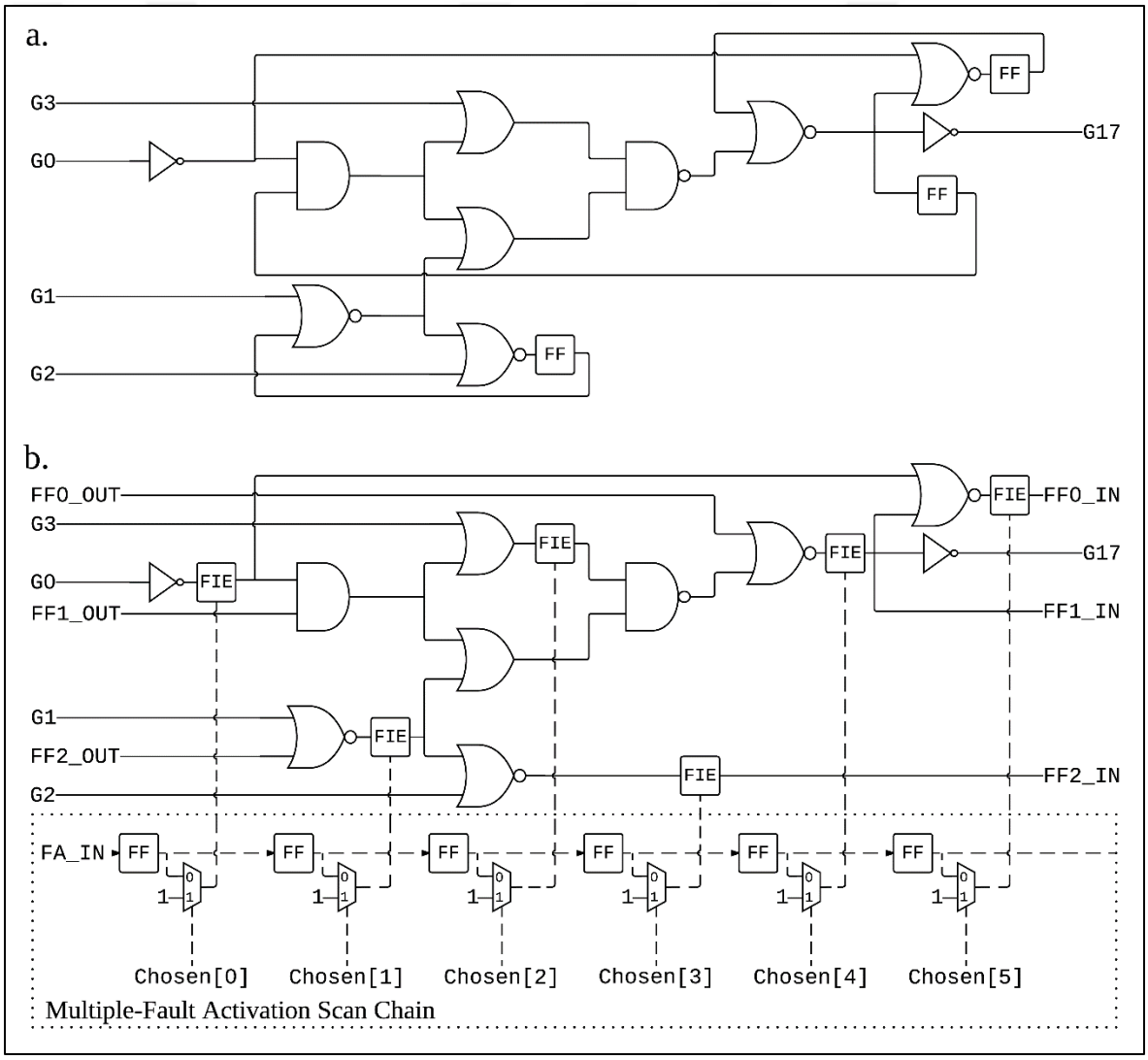

Figure 3.1. CIDMFI for s27 benchmark circuit. a. Original netlist of s27, b. Instrumented s27 with dynamic multiple fault injection.

number of test patterns is then $2^7 = 128$. The maximum total number of wrong output bits is $4 \times 128 = 512$. In order to achieve the maximum output-bit combinations, 50% HD is required, which is the half of the maximum total number of wrong output bits (512/2 = 256). Therefore, for each candidate fault, its fault impact should be compared to 256.

In Figure 3.2, three snapshots of the circuit operation are shown. In Figure 3.2a, *Chosen* input is set to "000000" which means that no FIE is chosen yet. The fault activation scan chain is in "100000" state that activates the 0^{th} FIE. The state of the circuit is kept at this state for 128 clock cycles and all 128 test patterns are applied. Four outputs are compared to the fault-free output values. The sum of the mismatch outputs for all the test patterns is obtained. This value is the fault impact of the 0^{th} FIE. Then, fault activation scan chain is shifted right in order to activate the next FIE (1^{st}) and so on. Similarly, each of the FIEs are activated one by one and fault impacts are obtained. Once all FIEs are tried, the one whose fault impact is close to 50% HD (256) is selected. The 4^{th} FIE is chosen since the fault impact is 63%. In the next snapshot, Figure 3.2b, *Chosen* input is set to "000010" in order to activate the 4^{th} FIE. In Figure 3.2 the other FIEs are activated one by one by shifting the scan chain. The fault impact of the 4^{th} and 2^{nd} FIEs together is 50% so then, the 2^{nd} FIE is chosen. *Chosen* input is set to "001010" shown as in Figure 3.2c. In the third round, the 0^{th} FIE together with the 4^{th} and 2^{nd} FIEs is found to be the next chosen FIE due to 53% HD. The final *Chosen* input is then set to "101010". These steps can be repeated to inject and emulate more faults together until the number of "1"s in *Chosen* is reached to the keysize. Figure 3.3 presents the logic locked s27 having 3-bit key input after determination of three fault locations.

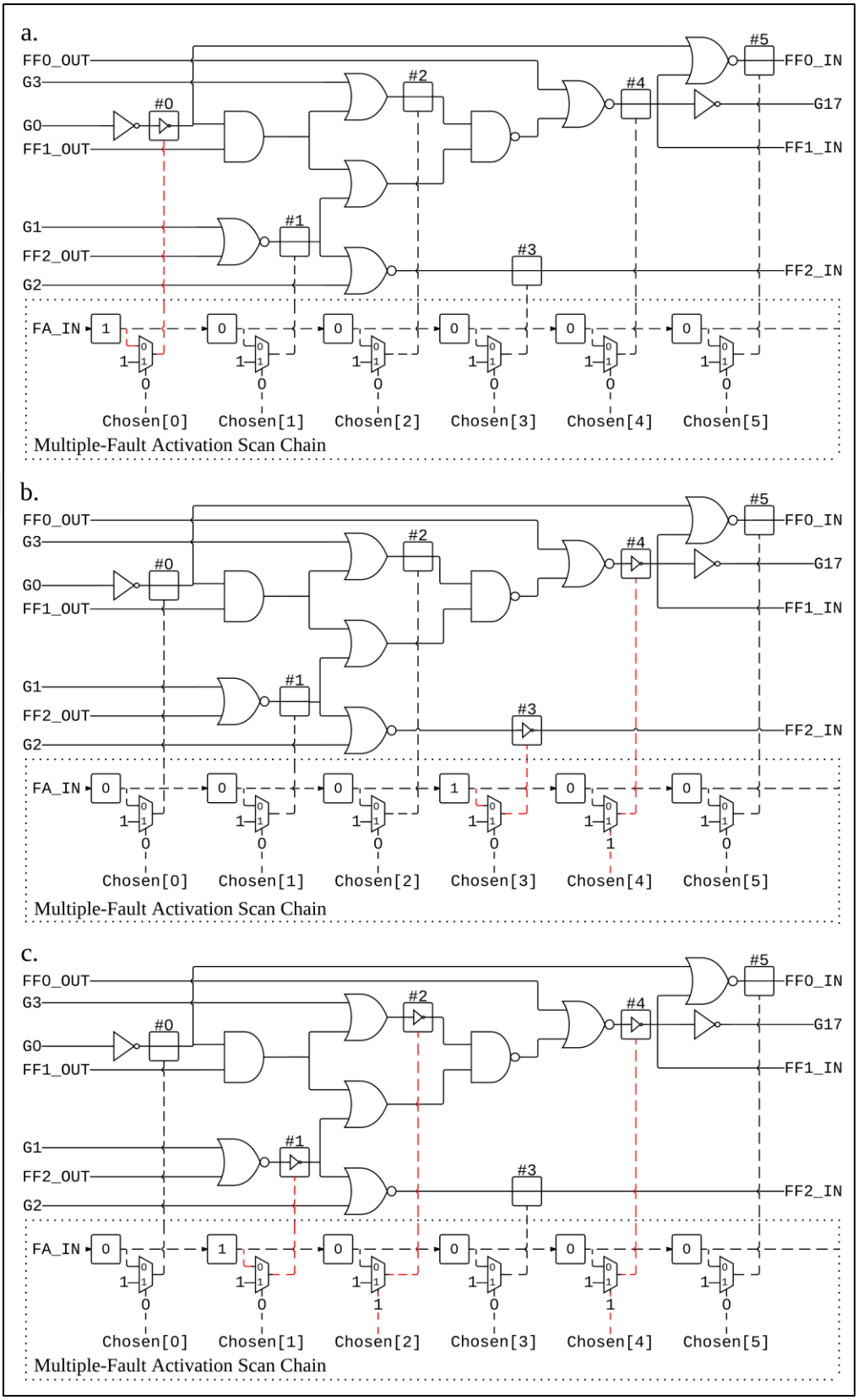

Figure 3.2. Determination of fault locations process (three snapshots are given for s27).

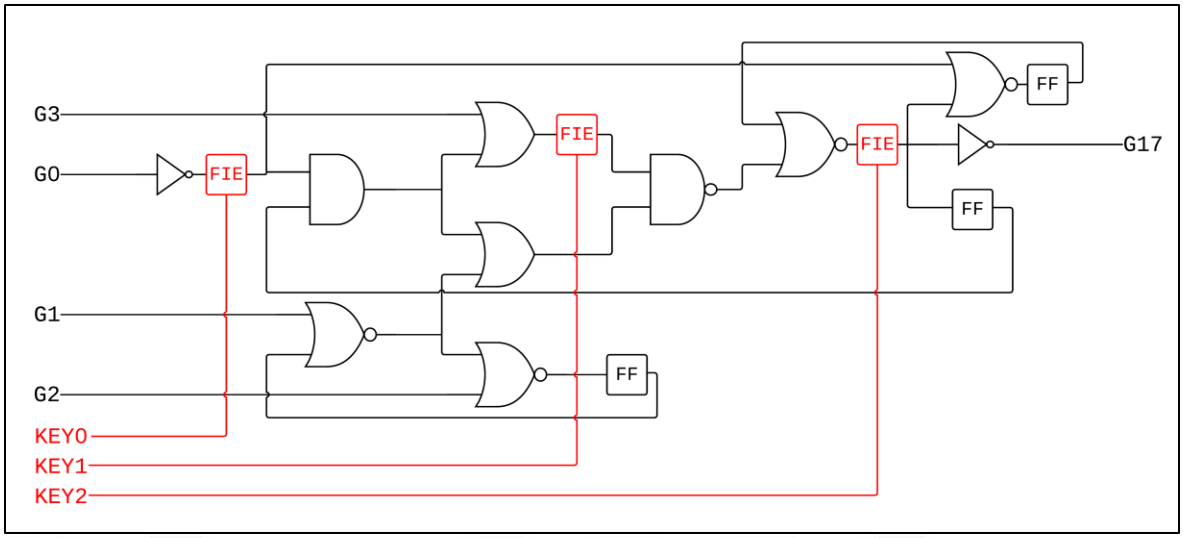

Figure 3.3. Logic locked s27.

Based on the above explanation, the total emulation time required for fault location selection phase of logic locking algorithm for N_k keysize using CIDMFI is calculated by Equation 3.1 and given as $T_{CIDMFI-LL}$. The total emulation time given in Equation 3.1 is remarkably less than the total emulation times of CISFI and PRSFI methods given in Equation 2.2 and Equation 2.4, respectively.

$$T_{CIDMFI-LL} = T_{comp} + T_{config} + N_k \times N_f \times N_p \times T_{pd} \quad (3.1)$$

3.1.2. Design & Implementation of Emulation Circuit

Up to here, we present how to instrument a netlist using CIDMFI and how to choose FIEs. Now, we will present the top level fault emulation circuit that consists of a *Circuit Under Instrumentation (CUI)*, ROM, BRAM, XOR, *Carry Save Adder (CSA)*, Controller, and UART units shown in Figure 3.4. CUI corresponds to the instrumented circuit based on CIDMFI. ROM is used to store the test patterns. BRAM is used to store expected output values. XOR compares the CUI output with the expected value from BRAM for a test pattern every cycle. Controller has a state machine that is responsible of:

(i) Initializing the BRAM unit with the golden expected values for the fault-free CUI which correspond to the test patterns that are stored in the ROM unit,

(ii) Generating the fault activation clock (*CK_FA*) and the fault activation pulse (*FA_IN*),

(iii) Keeping the current maximum fault impact value and comparing the maximum fault value with the current fault impact which is the sum of CSA output for all test patterns,

(iv) Deciding the next FIE to activate and setting the corresponding bit of the Chosen input based on the fault impact comparison, and

(v) Serializing and sending the final Chosen value through the UART unit.

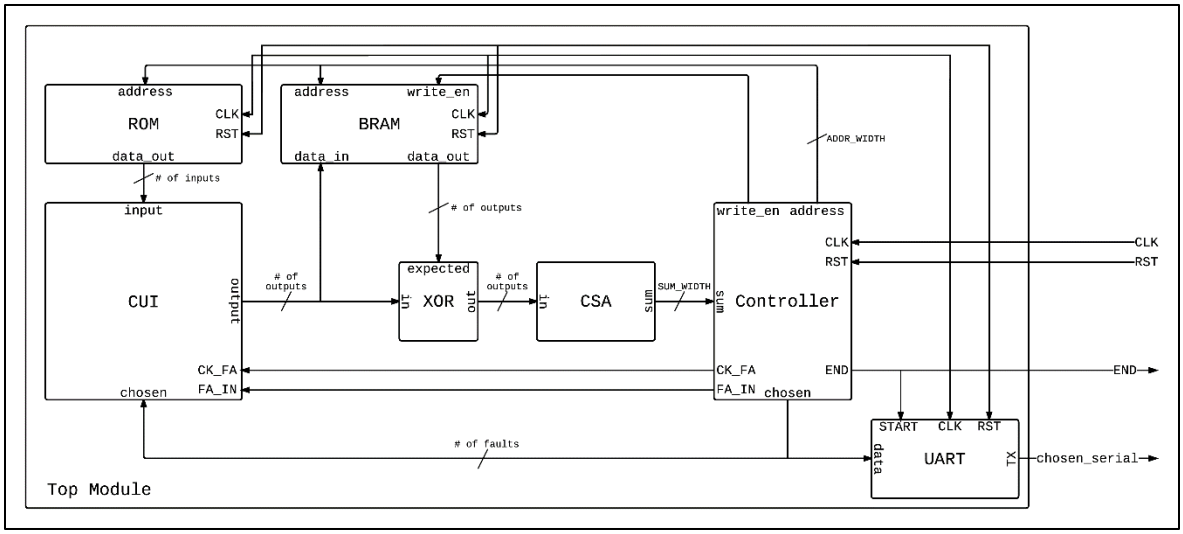

Figure 3.4. Proposed emulation circuit for logic locking.

The period of *CK_FA* is $N_p \times T_{pd}$ whereas T_{pd} is the period of the main clock. Every bit of *Chosen* and *FA_IN* are set to "0" for the fault-free CUI in order to obtain golden expected output values. In the fault activation process, a new test pattern and the corresponding expected output value are read from the ROM and BRAM units every clock cycle. For every $N_p \times T_{pd}$ period, the fault activation chain is shifted. Chosen is a N_f-bit signal and it should be constant for $N_f \times N_p \times T_{pd}$. Every $N_f \times N_p \times T_{pd}$ period, Controller picks the best FIE candidate based on the fault impact comparison with the maximum and repeats the process for N_k times. Final Chosen signal having N_k bits set to "1" out of N_f bits is then serialized and sent out.

One of the important units of the fault emulator circuit is CSA. CSA adds up the output of XOR to determine the fault impact of a key-gate for a given pattern based on Equation 2.8. In other words, CSA implements the inner summation of Equation 2.8. CSA in the emulation circuit adds up N_o bits. The idea of CSA is that saving carry bits instead of propagating [12]. Full adders and half adders can be used as CSAs, they receive three and two input bits respectively, then both output two bits as usual, but none of the inputs is carry-out bit of an other adder except the connections between different logic levels. Since carry propagation chains are eliminated, CSA trees provide a higher speed for multipliers and the other circuits that include multiple-operand summation. In CSA trees, partial products are compressed until two rows remain. Then the two rows are summed by a fast adder. The study in [19] proposes a new CSA tree called RoCoCo. It is the fastest unsigned multiplier 9 out of 22 cases compared to Wallace Tree [20], Dadda Tree [6], and native implementation of multiplication operator in Xilinx ISE. Here, we implemented the inner summation of Equation 2.8 with CSA based on RoCoCo. We used a fast parallel adder to implement the outer summation shown in Equation 2.8.

The final module in the top level emulation circuit is the UART module that serializes the N_f-bit Chosen signal and transmits the emulation result. The "1"s in Chosen indicates the selected FIEs. Based on this information, the netlist modification phase of the logic locking algorithm can be executed. In this phase, the original netlist is modified by adding chosen N_k FIEs and N_k-bit input in order to activate them.

3.1.3. Effect of Number of Test patterns to Hamming Distance

In this experiment, we devised five sets of randomly generated test sets with 10, 100, 1000, 10000, and 20000 test patterns. We applied all tests to s510 with 211 FIEs and s1423 with 657 FIEs and key-size of 128. After these simulations, we obtained Chosen register setting for each key-size from 1 to 128 for all tests and for each benchmark. In order to observe the effect of number of test patterns, we calculated HDs by applying 100000 test patterns for each *Chosen* register setting. In Figure 3.5 and Figure 3.6, we present the corresponding plots of five simulation runs. It can be clearly seen in Figure 3.5 and Figure 3.6 that more test patterns yield closer HD to 50%. However, after 1000 test patterns, we did not observed noticeable improvement for both benchmarks.

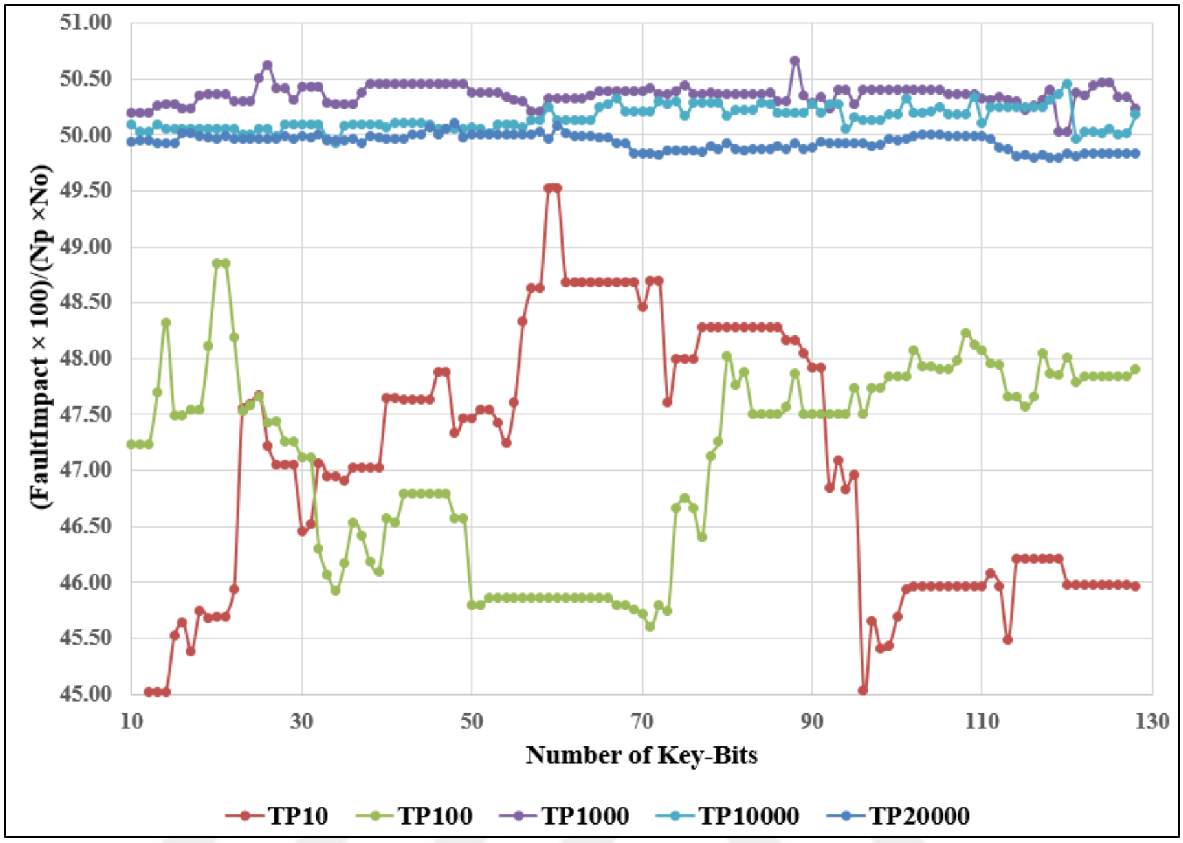

Figure 3.5. Effect of number of test patterns on HD for s510 (TP10, TP100, TP1000, TP10000, and TP20000 are sets of 10, 100, 1000, 10000, and 20000 test patterns).

3.1.4. Effect of Number of FIEs to Hamming Distance

In this experiment, we devised five of instrumented s1423 with 128, 256, 384, 512, and 640 FIEs. FIEs are inserted randomly in the circuit instrumentation. We simulated these five cases with 1000 patterns for a key-size of 128 with our DMFI testbench. We obtained *Chosen* register setting for each key-size from 1 to 128 for all cases. Then we calculated HDs by applying 100000 test patterns for each *Chosen* register setting, as we did in the previous experiment. In the case with only 128 FIEs, all FIEs were chosen at the end of the simulation. In Figure 3.7, five plots of HD versus the number of chosen key-bits are shown. From this experiment, we observed that increasing the number of FIEs used in the circuit instrumentation helps to reach overall 50% corruption of the output bits. However, if it reaches 50% at smaller key-sizes than the target key-size, instrumentation with more FIEs

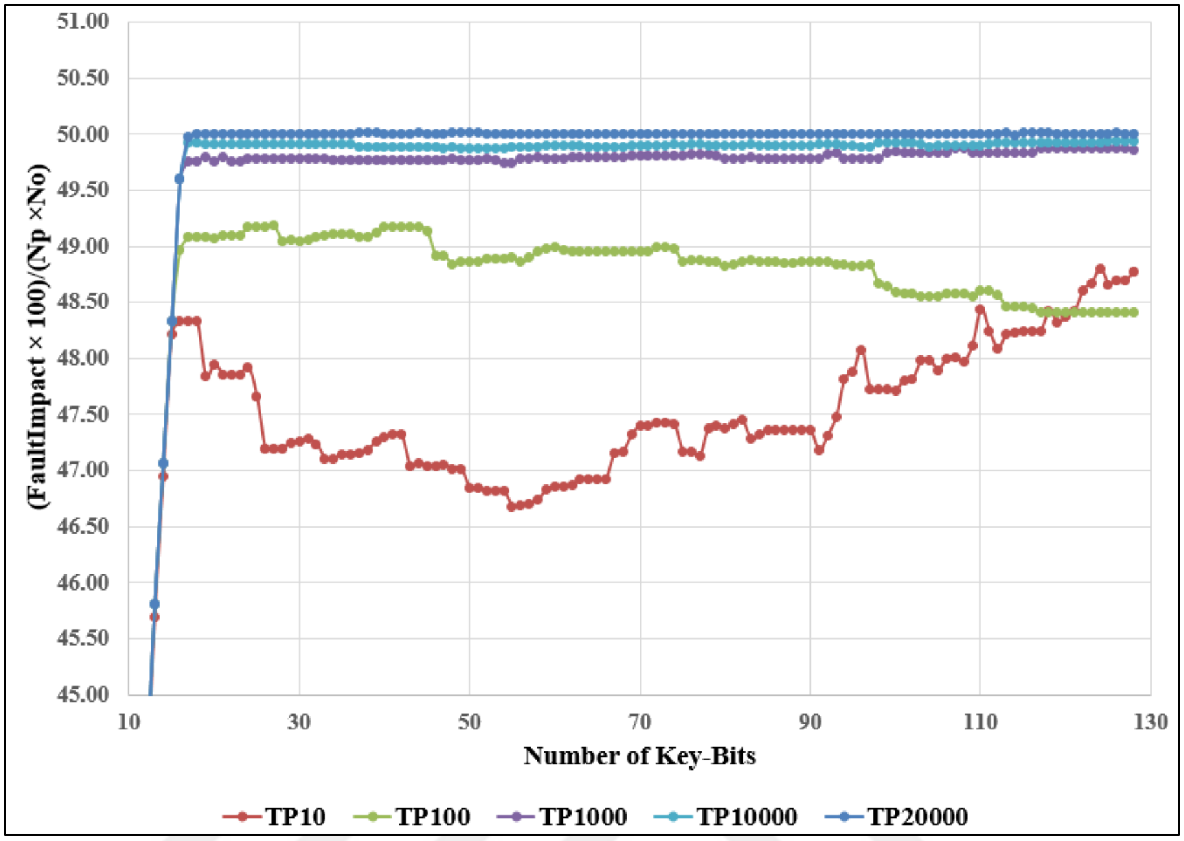

Figure 3.6. Effect of number of test patterns on HD for s1423 (TP10, TP100, TP1000, TP10000, and TP20000 are sets of 10, 100, 1000, 10000, and 20000 test patterns).

does not improve the result much, since 50% HD is already reached. For this reason, more FIEs do not always mean better quality results. Since emulation times are very small compared to compilation time for less number of FIEs, it is advisable that starting with less number of FIEs and checking whether or not it can reach to 50% HD, and instrumenting with more FIEs, only if it is necessary.

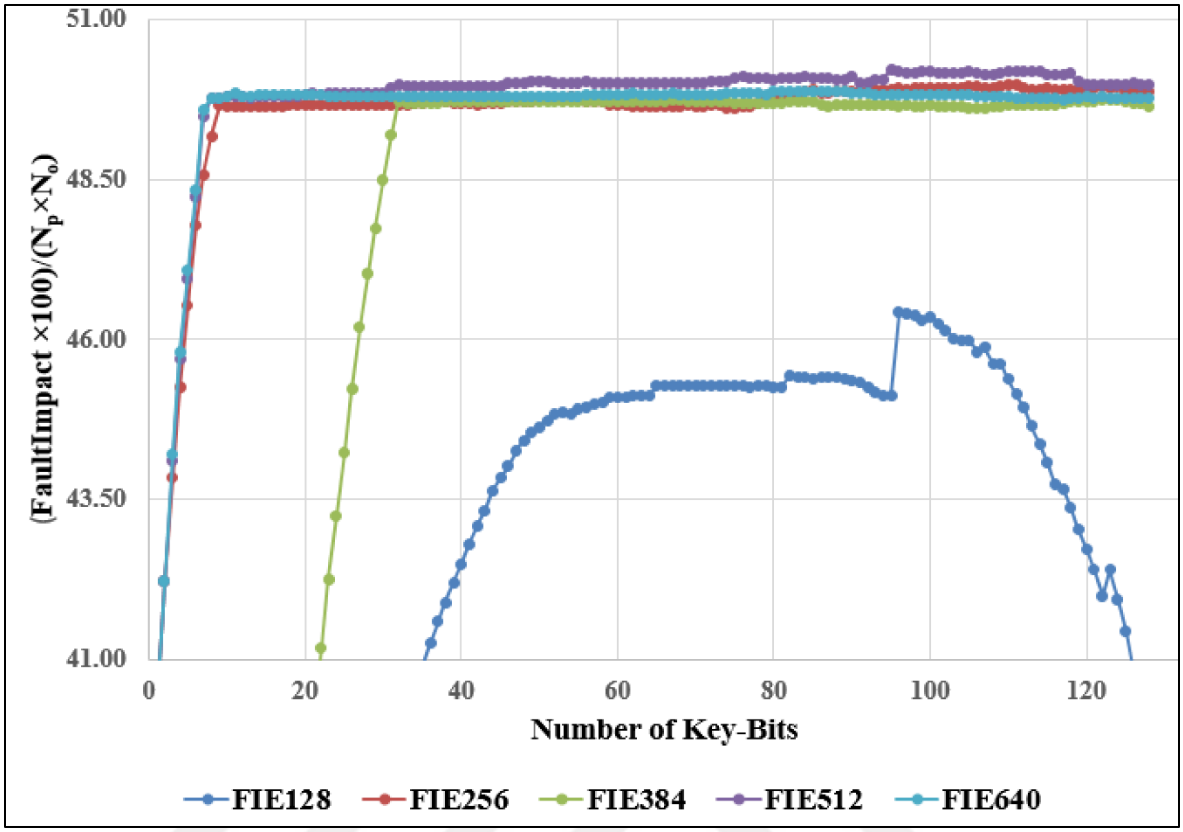

Figure 3.7. Effect of number of FIEs on HD for s1423.

3.2. PROPOSED METHODS FOR MULTI-CYCLE TEST GENERATION

3.2.1. Fault Models Used for Fault Emulation

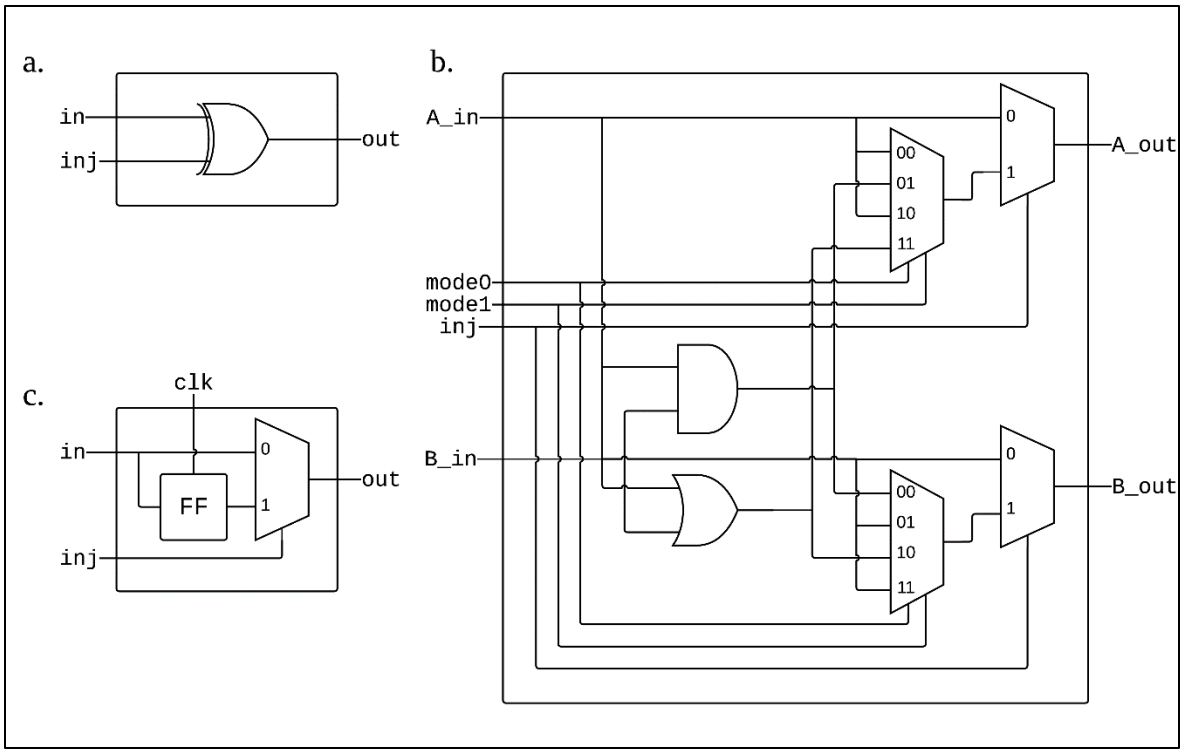

Figure 3.8. Fault models for three types of faults. a. Stuck-at, b. Bridging, c. Transition.

Three types of fault models used for fault emulation of proposed algorithm. First one shown in Figure 3.8a is stuck-at fault model from [10]. Four-way Diode-AND/OR bridging fault models that introduced in Section 2.1 is combined into a single module in Figure 3.8b. In Figure 3.8c, a simple transition fault model that contains a FF and MUX is shown. All three faults models can be activated/deactivated using *inj* input.

3.2.2. Proposed Multi-Cycle Test Generation algorithm

In the algorithms from [23], target faults are either stuck-at faults or transition faults. This approach in [23] yields optimized fault coverage for only one fault type. However, overall

fault coverage can be better, if more fault types are targeted. The proposed algorithm in this thesis, which is shown in Algorithm 3.2, considers stuck-at, bridging, and transition faults while selecting tests and their number of clock cycles for multi-cycle testing.

Algorithm 3.2. Proposed algorithm for multi-cycle test generation

1. Obtain expected values for each input vector and for each clock cycle from 1 to 32.
2. Let F be the set of target faults. For every $f \in F$, let $incF$ be the set of included faults. Initially, set $incF$ to F.
3. Let V be the set of input vectors. For every $v \in V$, let $cLim(v)$ be the set of clock cycles that the circuit must run in functional mode to detect most faults after (or while) v is applied.
4. For every $v \in V$, let $numDet(v)$ show the number of faults detected with v. Initially $numDet(v)$ is set to zeros.
5. For each fault type t,
   For each input vector v,
   For each fault f such that f is a t type fault and $f \in incF$,
   Activate fault f and deactivate other injected faults.
   Apply input vector v.
   For each clock cycle c from 1 to 32,
   (i) Run the circuit for one clock cycle in functional mode.
   (ii) If c is 1 and t is a transition fault type, skip this iteration.
   (iii) If fault f is detected at primary outputs, mark fault f as detected for clock cycles $\geq c$, then break the loop.
   (iv) If fault f is detected after scan-out operation, mark fault f as detected for clock cycle = c.
   Find the number of clock cycles c that most faults detected (if equal number of faults detected for more than one c's, select the lowest one), then set $cLim(v)$ to c and add number of detected faults when the circuit is run c clock cycles to $numDet(v)$.
6. For every $v \in V$, find the vector v that gives highest $numDet(v)$, then set $bestV$ to v. Record $bestV$ and $cLim(bestV)$ as result.
7. Remove detected faults from $incF$ when $bestV$ is applied for $cLim(bestV)$ clock cycles, as they are detected at step 5.
8. If $incF \neq \emptyset$ and the number of recorded result vectors is less than the number of input vectors in V, then reset $numDet(v)$ and $cLim(v)$ for every $v \in V$ and go to step 5.

In the algorithms of [23], test sets are fixed. Algorithms find a number of clock cycles information for each test, so that when the tests are applied for that number of clock cycles, in the hope of obtaining better fault coverage. On the other hand, our algorithm shown in Algorithm 3.2, may reduce number of tests or it can select same test more than once for different number of clock cycles.

In [25], in order to detect more faults in a multi-cycle test, present state values and primary outputs are observed on every functional clock cycle of the test and compared with expected values. Since this method requires scan-out and scan-in operation for each functional cycle

of the test, test time is dramatically increased. In the proposed algorithm only primary outputs are observed in each clock cycle of a multi-cycle test and the present state values and primary outputs are observed at every scan-out and scan-in operation, therefore the test application time is not affected.

3.2.3. Circuit Instrumentation and ATPG

In order to evaluate the performance of our algorithm, ISCAS'89 sequential benchmarks are used. Dynamic single-fault activation scan chain and faults models are added to the netlist as shown previously in Figure 2.3b. However, circuits did not converted to combinational, like we did for logic locking. Also, bridging faults added in such way that no feedback occurs when the fault is activated.

In Pomeranz's Algorithm #1, a compact one-detection single-cycle test set for single stuck-at faults was used. Stuck-at test patterns were generated via Synopsys Tetramax. In order to obtain a compact test set for single stuck-at faults, reducing the number of tests that obtained from Tetramax was needed. We ran our algorithm for single-cycle, only stuck-at faults as target faults. Then we kept the tests that are selected by our algorithm, and removed the others. In order to compare algorithms, this resulting test set was used for all three algorithms.

3.2.4. Comparison of Multi-Cycle Test Generation Algorithms

Note that we use the same notation given in [23]. Column Avg. shows average clock cycle of a test in resulting multi-cycle test set. Column $C(T_L)$ is a metric used in [23] to show total number of clock cycles required to apply resulting multi-cycle test set. Column $\hat{m}$, shows the number of single-cycle tests that takes same number of cycles to apply as resulting multi-cycle test set.

Then, the resulting test sets are used in the following algorithms shown in Table 3.1:

(i) In "All 1 cyc.", tests are applied for one cycle when the circuit is in functional mode.
(ii) In "All 2 cyc.", tests are applied for two cycles when the circuit is in functional mode.

(iii) In “Algorithm #1” we run the first algorithm [23]. This algorithm is for stuck-at faults, but after running the algorithm for stuck-at faults, resulting multi-cycle test set is applied to detect bridging and transition faults to obtain their fault coverage. The algorithm starts from L = 32.

(iv) In “Algorithm #2” we run the second algorithm [23]. This algorithm is for transition faults and it uses two cycle test set for transition faults as target faults. After running the algorithm for transition faults as target faults, the resulting multi-cycle test set is applied to detect stuck-at and bridging faults in order to obtain their fault coverage. The algorithm is applied until L = 32.

(v) In “Proposed”, we run our proposed algorithm shown in Algorithm 3.2. All three fault types were considered during our algorithm.

When all tests applied for one cycle, maximum stuck-at coverage was reached. Since there is no transition fault coverage for single cycle tests, overall coverage is the lowest. Test application time is usually the lowest.

When all tests applied for two cycles, tests detect more transition and bridging faults, but stuck-at fault coverage decreases.

Algorithm #1 cannot have less stuck-at fault coverage than single cycle tests. Therefore, stuck-at fault coverage is high. However overall coverage is low and test application time is the highest.

Algorithm #2 has the lowest stuck-at fault coverage, since target fault type is transition faults. Overall coverage and test time is better than Algorithm #1.

The proposed algorithm has the best overall fault coverage. Stuck-at fault coverage obtained from multi-cycle tests can be lower than the fault coverage obtained from the single-cycle tests. In some benchmarks, the algorithm reduced the number of tests. However only in s35932 test time is lower than single cycle tests. If allowed, the algorithm can be changed to add more tests in order to reach the maximum coverage for all fault types. However, this would cause higher test application time.

Table 3.1. Comparison of the algorithms

(The number of clock cycles required for applying test set T_L is $C(T_L)$ [23].)

Algorithm	Circuit	Cycles			Fault Coverage			
		Avg	$C(T_L)$	$\hat{m}$	s.a.	Bridg	Trans	Overall
All 1 cyc.	s298	1	104	6	100	58.75	0	52.2
All 2 cyc.	s298	2	110	6.4	99.16	61.25	46.22	69.81
Alg.#1 (32)	s298	17.33	202	12.53	100	75	63.03	79.87
Alg.#2	s298	14.33	184	11.33	99.16	73.75	75.63	83.96
Proposed	s298	15.33	190	11.73	100	88.75	81.51	90.25
All 1 cyc.	s344	1	95	5	100	64.58	0	51.9
All 2 cyc.	s344	2	100	5.31	99.38	70.83	58.13	77.72
Alg.#1 (32)	s344	21	195	11.25	100	77.08	63.13	80.98
Alg.#2	s344	4	110	5.94	99.38	75	78.13	86.96
Proposed	s344	3.8	109	5.88	100	79.17	80	88.59
All 1 cyc.	s382	1	131	5	100	51.04	0	50.24
All 2 cyc.	s382	2	136	5.23	98.73	56.25	53.16	71.36
Alg.#1 (32)	s382	15.2	202	8.23	100	59.38	50	71.36
Alg.#2	s382	11.8	185	7.45	98.73	64.58	74.05	81.31
Proposed	s382	16.4	208	8.5	100	73.96	80.38	86.41
All 1 cyc.	s510	1	76	10	100	38.79	0	47.58
All 2 cyc.	s510	2	86	11.43	100	46.55	55.45	71
Alg.#1 (32)	s510	29.2	358	50.29	100	47.41	59.24	72.68
Alg.#2	s510	3.6	102	13.71	100	52.59	70.14	78.07
Proposed	s510	4.13	87	11.57	100	56.9	74.41	80.67
All 1 cyc.	s1423	1	824	10	100	59.25	0	51.68
All 2 cyc.	s1423	2	834	10.13	97.72	64.38	53.42	73.54
Alg.#1 (32)	s1423	17.1	985	12.15	100	64.38	51.75	73.79
Alg.#2	s1423	9.1	905	11.08	100	65.75	61.64	78.08
Proposed	s1423	12.1	935	11.48	100	67.12	63.32	79.02
All 1 cyc.	s9234	1	5935	27	94.24	63.5	0	49.8
All 2 cyc.	s9234	2	5962	27.13	78.03	58	38.28	58.13
Alg.#1 (32)	s9234	4.78	6037	27.48	94.24	65.5	27.25	61.52
Alg.#2	s9234	7.56	6112	27.83	81.35	58.25	43.07	61.56
Proposed	s9234	9.15	6155	28.04	89.55	61.75	40.72	64.58
All 1 cyc.	s35932	1	13831	7	92.58	63.5	0	49.1
All 2 cyc.	s35932	2	13838	7	92.58	75.5	71.88	81.13
Alg.#1 (32)	s35932	29.71	14032	7.12	92.58	86.5	81.05	86.76
Alg.#2	s35932	9.71	13892	7.04	92.58	86	81.05	86.68
Proposed	s35932	11.83	12167	6.04	92.58	87.25	81.35	87.01
All 1 cyc.	s38417	1	26191	15	100	78.75	0	54.7
All 2 cyc.	s38417	2	26206	15.01	99.02	81.75	75.29	86.27
Alg.#1 (32)	s38417	13.73	26382	15.12	100	80.25	65.23	82.23
Alg.#2	s38417	7.6	26290	15.06	97.75	82.75	82.26	88.81
Proposed	s38417	9.27	26315	15.08	98.93	84.25	81.15	89.09
All 1 cyc.	s38584	1	39955	27	98.63	72.25	0	53.06
All 2 cyc.	s38584	2	39982	27.02	93.85	74.75	55.86	74.84
Alg.#1 (32)	s38584	8.44	40156	27.14	98.63	75.75	44.43	72.22
Alg.#2	s38584	10.96	40224	27.19	93.75	77	63.28	78.27
Proposed	s38584	14.37	40316	27.25	98.34	82	65.33	81.86

3.2.5. Hardware-Software Co-Design

In this section, implementation of proposed algorithm from Section 3.2.2. will be explained. ISCAS'89 benchmark circuits emulated using an Xilinx Zynq-7000 All Programmable SoC shown in Figure 3.9.

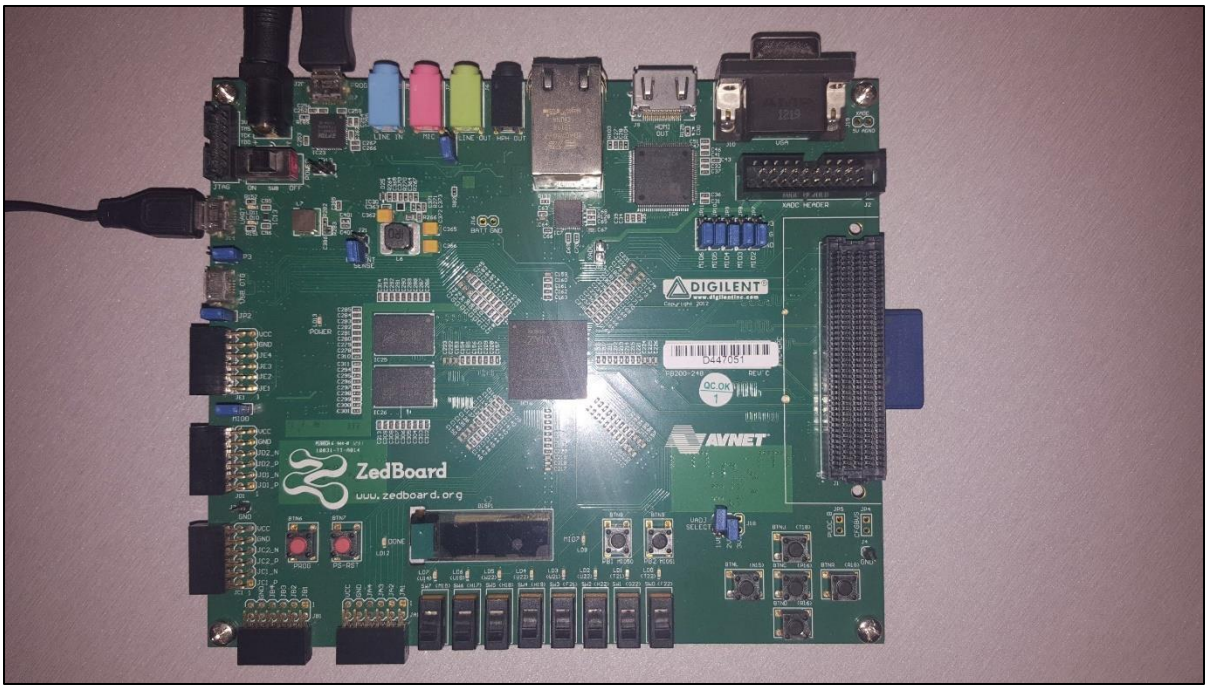

Figure 3.9. Digilent Zedboard with Xilinx Zynq-7000.

In order to realize fault analysis, circuits instrumented as explained in Section 3.2.3. For each benchmark circuit, three copies of the circuit is instantiated in *Programmable Logic (PL)* part of the Zynq. Each instance of the circuit injected with a different fault model. Proposed algorithm is implemented in *Processing System (PS)* part of the SoC. To reduce runtime of the algorithm, 5^{th} step of the algoritm is seperated for each fault type and executed in parallel for all three instances of the circuit.

Table 3.2. Utilization of three instrumented s510 benchmark circuits

Site Type	Used	Fixed	Available	Util%
Slice LUTs	1295	0	53200	2.43
LUT as Logic	1225	0	53200	2.30
LUT as Memory	70	0	17400	0.40
LUT as Distributed RAM	0	0		
LUT as Shift Register	70	0		
Slice Registers	1515	0	106400	1.42
Register as Flip Flop	1515	0	106400	1.42
Register as Latch	0	0	106400	0.00
F7 Muxes	5	0	26600	0.02
F8 Muxes	0	0	13300	0.00

Table 3.2 shows utilization information for three instrumented s510 benchmark circuits. In Table 3.3, 3.4 and 3.5, synthesis results are shown for circuits instrumented with stuck-at, bridging and transition faults respectively. Results of emulation times and their comparison with simulation times are given in Section 4.2.

Table 3.3. Synthesis results for circuits with stuck-at faults

Circuit	Slice Registers	Utilization of Slice Regs (%)	Slice LUTs	Utilization of Slice LUTs (%)	Maximum Clock Frequency (MHz)
s1423	734	0.69	600	1.13	171.29
s27	13	0.01	19	0.04	636.54
s298	133	0.13	93	0.18	336.02
s344	175	0.16	125	0.24	268.82
s35932	17793	16.72	10389	19.53	288.02
s382	179	0.17	155	0.29	364.96
s38417	23830	22.40	12362	23.24	150.06
s38584	20681	19.44	11073	20.81	160.00
s510	220	0.21	164	0.31	249.27
s9234	5809	5.46	2625	4.93	133.23

Table 3.4. Synthesis results for circuits with bridging faults

Circuit	Slice Registers	Utilization of Slice Regs (%)	Slice LUTs	Utilization of Slice LUTs (%)	Maximum Clock Frequency (MHz)
s1423	147	0.14	410	0.77	186.15
s27	3	0.00	12	0.02	751.32
s298	35	0.03	104	0.20	381.24
s344	27	0.03	93	0.18	346.62
s35932	1826	1.72	3021	5.68	382.55
s382	45	0.04	132	0.25	301.11
s38417	1750	1.64	3367	6.33	210.17
s38584	1530	1.44	3864	7.26	230.20
s510	41	0.04	181	0.34	318.00
s9234	313	0.29	973	1.83	233.32

Table 3.5. Synthesis results for circuits with transition faults

Circuit	Slice Registers	Utilization of Slice Regs (%)	Slice LUTs	Utilization of Slice LUTs (%)	Maximum Clock Frequency (MHz)
s1423	1392	1.31	1437	2.70	125.27
s27	23	0.02	29	0.06	421.23
s298	253	0.24	187	0.35	257.33
s344	335	0.32	347	0.65	223.86
s35932	33858	31.82	30063	56.51	149.08
s382	337	0.32	275	0.52	235.52
s38417	45996	43.23	32090	60.32	103.09
s38584	39932	37.53	20924	39.33	90.31
s510	435	0.41	444	0.84	265.68
s9234	11405	10.72	8891	16.71	92.69

4. RESULTS

4.1. LOGIC LOCKING

In order to evaluate our emulation technique, we wrote a Verilog code generator in Perl that can generate emulation circuit for a given netlist based on parameters such as the number of PIs, POs, FFs, FIEs, and patterns. ISCAS'89 sequential benchmark circuits are used for this purpose. In Table 4.1, we present the synthesis results of the automatically generated emulation circuits. These results are obtained on a PC with Intel Core i7-950 Processor and 6GB RAM running Xilinx ISE 14.7. For all the benchmarks, N_p is set to 1000 and the test patterns are generated randomly. In addition, N_k is set to 128. N_f values are determined based on the size of the benchmarks as well as limitations of Xilinx ISE. For example, in the case of s15850, s35932, s38417, and s38584, with FIEs inserted at every gate output, during the FPGA flow of the emulation circuit, although the synthesis runs were completed, P&R failed. Therefore, we had to drop the number of FIEs to 3072 for s15850 and 4096 for s35932, s38417, and s38584 as shown in Table 4.1, whereas the rest of the benchmarks are instrumented by inserting FIEs at every gate output. In Table 4.1, the number of registers, LUTs, and BRAMs as well as FPGA resource utilization percentages are given in the 5th, 6th, and 7th columns.

Table 4.1. Synthesis results for emulation circuits

Circuit	#PI/#PO/#FF	#FIE	Max. Freq. (MHz)	#Reg.	#LUT orig./instr.	#BRAM	T_{comp} (s)
s510	19/7/6	211	227	535 (1%)	31/751 (0/1%)	1 (1%)	159
s838	31/1/32	446	151	1011(1%)	60/1517(0/2%)	3 (2%)	180
s1423	17/5/74	657	89	1437 (1%)	138/2021(0/3%)	5 (3%)	240
s5378	35/49/179	2779	116	5687 (5%)	330/7972 (0/14%)	12 (8%)	540
s9234	36/39/211	5597	93	11325 (10%)	385/14857 (0/27%)	14 (10%)	1980
s13207	62/152/638	7951	80	16037 (15%)	657/21372 (1/40%)	41 (29%)	3660
s15850	77/150/534	3072	90	6277 (5%)	828/10046 (1/18%)	36 (25%)	1440
s35932	35/320/1728	4096	111	8331 (7%)	2272/16289 (4/30%)	106 (75%)	3000
s38417	28/106/1636	4096	105	8329 (7%)	2218/16995 (4/31%)	94 (67%)	5220
s38584	38/304/1426	4096	103	8301 (7%)	2248/17470 (4/32%)	89 (63%)	10200

The maximum clock frequencies of the emulation circuits are listed in the 4th column. The final column in Table 4.1 is the sum of both the recorded compilation and bitstream generation times, T_{comp}. Since the benchmarks are first converted into combinational circuits, the number of registers and the corresponding FPGA resource utilization percentages shown in the 5th column are obtained from the synthesis of the emulation circuits. As it can be seen from the 5th column, the number of FFs is about the twice of the number of FIEs because of the fault activation scan chain and *Chosen* register. In the 6th column, the required number of LUTs and LUT utilization percentages for both the original and emulation circuit are given. As the number of FIEs increases, the number of required LUTs increases. In the emulation circuit, we allocate BRAM to store the expected output values. The original circuits do not consume BRAM, hence the 7th columns refers to the number of BRAMs and FPGA utilization of the emulation circuits. Instead of storing the expected output values corresponding to the test patterns, we could have implemented a replica of the circuit itself to generate them on the fly. However, a replica also consumes LUTs so that BRAMs are utilized to save more LUTs.

Table 4.2. Performance comparison

Circuit	**Emu. Time**	**Sim. Time**
s510	0.38 s	1 min 57 s
s838	0.99 s	12 min 18 s
s1423	1.53 s	44 min 57 s
s5378	7.01 s	13 hrs 27 min
s9234	14.29 s	2.36 days
s13207	20.37 s	5.90 days
s15850	7.77 s	1.65 days
s35932	10.41 s	6.23 days
s38417	10.41 s	5.12 days
s38584	10.41 s	6.02 days
Total Time	**83.57 s**	**28 days**

Table 4.2 presents the simulation and emulation times for the fault location selection phase of the logic locking algorithm. In our emulation experiments, Zedboard with Xilinx Zynq-7000 (XC7Z020-CLG484) is used as the target FPGA platform. The clock frequency in all of the emulation experiments is 50MHz. Note that shorter emulation times can also be achieved with the maximum clock frequencies given in Table 4.1. The full bitstream size

and full Zynq configuration (T_{config}) are about 4MB and 2s, respectively. Note that T_{comp} and T_{config} are excluded from the emulation times, but the time of the serial transmission of *Chosen* bits is included. In addition, our emulation circuit activates only unchosen FIEs. This also decreases the measured emulation time compared to Equation 3.1. The total emulation time, $T_{CIDMFI-LL}$, is dependent on T_{comp}, T_{config}, N_k, N_f, N_p, and T_{pd} whereas N_k is 128, N_p is 1000, and T_{pd} is 20ns in our emulation experiments. Hence, the emulation times are found to be the same for the benchmarks (s35932, s38417, and s38584) with the same number of FIEs, N_f, in our emulation experiments. In the simulation experiments, automatically generated DMFI based emulation circuit and its testbench, and Synopsys VCS are used. Note that T_{comp} is also excluded from the simulation times shown in Table 4.2. The simulation results are obtained on an Ubuntu 10.04.4 LTS running PC with Intel Xeon 64-bit CPU and 8GB RAM. It takes about 28 days to simulate all the benchmarks shown in Table 4.2, whereas 84 seconds to emulate all of them. Therefore, the speed-up is 27806. The quality of a key (*Chosen* register setting) is related to the number of test patterns and FIEs. More test patterns and FIEs can yield a better *Chosen* register setting that causes closer HD to 50% overall. However, we have to limit their quantity, since we don't have unlimited area for them. In order to determine the effect of number of test patterns or FIEs on HD, we conducted two DMFI simulation experiments that mentioned in Section 3.1.3 and 3.1.4.

4.2. MULTI-CYCLE TEST GENERATION

In Table 4.3 the simulation and emulation times are shown for proposed algorithm for multi-cycle test generation. In the simulation experiments, automatically generated DFI based emulation circuit and its testbench, and Synopsys VCS are used. The simulation results are obtained on an Ubuntu 10.04.4 LTS running PC with Intel Xeon 64-bit CPU and 8GB RAM.

In small benchmarks, simulation times were better. However overal speed-up of 6 is reached with emulation. With same amount of tests and FIEs, higher speed-up is achieved for s38584 compared to s9234. This indicates that as circuit complexity increases, speed-up of the emulation technique also increases.

Table 4.3. Performance comparison

Circuit	**#Tests**	**#FIE (stuck-at or transition)**	**#FIE (bridging)**	**Simulation Time (s)**	**Emulation Time (s)**
s510	10	211	29	1	8
s1423	10	657	73	18	40
s9234	27	1024	100	720	570
s35932	7	1024	100	360	30
s38417	15	1024	100	1080	120
s38584	27	1024	100	3840	300
Total Time (s)				6019	1068

5. CONCLUSIONS AND FUTURE WORK

In this thesis, DMFI technique is proposed in order to emulate more than one fault within a single FPGA configuration. Secondly, we propose an emulation technique based on DMFI to speed up the fault location determination phase of the logic locking algorithm. As opposed to simulation-based techniques, our technique enables real-time assessment of fault impacts and determination of the locations of the faults to be inserted. We have fully automated the process by developing a code generator in Perl which dumps out the top-level Verilog netlist of the emulation circuit for a given Verilog netlist of a sequential circuit. The performance of fault location determination of the logic locking algorithm based on DMFI emulation is evaluated with ISCAS'89 sequential benchmark circuits and compared with DMFI simulation. Based on the experimental results, a remarkable speed-up is observed. Our proposed technique can be easily adapted to other types of FIEs.

For the multi-cycle test generation, an efficient algorithm that generates a multi-cycle test set from a single-cycle test set and improves both the fault coverage and the test application time by considering stuck-at, bridging, and transition faults is proposed. Emulation of this algorithm is implemented using an all programmable SoC. To take full advantage of multi-cycle tests, both test quality and test compaction should be considered at the same time.

Two publications [26, 27] have been made as a result of the work carried out in this thesis.

As a future work, in order to improve security of logic locking, other types of faults such as bridging and transition faults as well as FIE combinations and also sequential logic locking will be considered.

REFERENCES

1. Y.M. Alkabani, and F. Koushanfar. Active Hardware Metering for Intellectual Property Protection and Security. *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, 1-16, 2007.

2. B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (Im)Possibility of Obfuscating Programs. *Journal of the ACM*, 2:1-48, 2012.

3. R. Chakraborty, and S. Bhunia. Security Against Hardware Trojan Attacks Using Key-Based Design Obfuscation. *Journal of Electronic Testing*, 27:767-785, 2011.

4. R.S. Chakraborty, and S. Bhunia. Security Against Hardware Trojan Through A Novel Application of Design Obfuscation. *IEEE/ACM International Conference on Computer-Aided Design - Digest of Technical Papers (ICCAD)*, 113-116, 2009.

5. K.T. Cheng, S.Y. Huang, and W.J. Dai. Fault Emulation: A New Methodology for Fault Grading. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10:1487–1495, 2006.

6. L. Dadda. Some Schemes for Parallel Multipliers. *Alta Frequenza*, 34:349–356, 1965.

7. S. Gören, O. Ozkurt, A. Yildiz, H.F. Ugurdag, R. Chakraborty, and D. Mukhopadhyay. Partial Bitstream Protection for Low-Cost FPGAs with Physical Unclonable Function, Obfuscation, and Dynamic Partial Self Reconfiguration. *Computers and Electrical Engineering*, 2:386-397, 2013.

8. S. Gören, A. Yildiz, O. Ozkurt, and H.F. Ugurdag. FPGA Bitstream Protection with PUFs, Obfuscation and Multi-boot. *6th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, 1-2, 2011.

9. KPMG. Managing the Risks of Counterfeiting in the Information Technology, 2006. http://www.agmaglobal.org/press_events/press_docs/Counterfeit_WhitePaper_Final.pdf [retrieved 07 October 2015].

10. S.K. Lu, S.Y. Huang, C.W. Wu, and Y.M. Chen. Speeding Up Emulation-Based Diagnosis Techniques for Logic Cores. *IEEE Design and Test of Computers*, 4:88-97, 2011.

11. M. Rostami, F. Koushanfar, and R. Karri. A Primer on Hardware Security: Models, Methods, and Metrics. *Proceedings of the IEEE*, 8:1283-1295, 2014.

12. B. Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*, Oxford University Press, 2000.

13. S. Plaza, and I. Markov. Solving the Third-Shift Problem in IC Piracy With Test-Aware Logic Locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6:961–971, 2015.

14. J. Rajendran, H. Zhang, C. Zhang, G.S. Rose, Y. Pino, O. Sinanoglu, and R. Karri. Fault Analysis-Based Logic Encryption. *IEEE Transactions on Computers*, 2:410-424, 2015.

15. J. Roy, F. Koushanfar, and I. Markov. EPIC: Ending Piracy of Integrated Circuits. *Conference on Design, Automation and Test in Europe*, 1069-1074, 2008.

16. E. Sanchez, L. Sterpone, and A. Ullah. Effective Emulation of Permanent Faults in ASICs Through Dynamically Reconfigurable FPGAs. *Field Programmable Logic and Applications (FPL)*, 1-6, 2014.

17. SEMI. Innovation Is at Risk as Semiconductor Equipment and Materials Industry Loses up to 4 Billion Annually due to IP Infringement, http://www.semi.org/en/Press/P043775 [retrieved 07 October 2015].

18. G.E. Suh, and S. Devadas. Physical Unclonable Functions for Device Authentication and Secret Key Generation. *Design Automation Conference*, 9-14, 2007.

19. F. Ugurdag, O. Keskin, C. Tunc, F. Temizkan, G. Fici, and S. Dedeoglu. RoCoCo: Row and Column Compression for High-Performance Multiplication on FPGAs. *East-West Design and Test Symposium (EWDTS)*, 98-101, 2011.

20. C. Wallace. A Suggestion for a Fast Multiplier. *IEEE Transactions on Electronic Computers*, 1:14-17, 1964.

21. R. Wieler, Z. Zhang, and R.D. McLeod. Emulating Static Faults Using a Xilinx Based Emulator. *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, 110-115, 1995.

22. J.M. Emmert, C.E. Stroud, and J.R. Bailey. A New Bridging Fault Model for More Accurate Fault Behavior. *IEEE AUTOTESTCON Proceedings*, 481-485, 2000.

23. I. Pomeranz. Multicycle Tests with Constant Primary Input Vectors for Increased Fault Coverage. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9:1428-1438, 2012.

24. I. Pomeranz. Multi-Cycle Broadside Tests with Runs of Constant Primary Input Vectors, *IET Computers and Digital Techniques*, 2:90-96, 2014.

25. S. Kajihara, M. Matsuzono, H. Yamaguchi, Y. Sato, K. Miyase, and X. Wen. On Test Pattern Compaction with Multi-Cycle and Multi-Observation Scan Test. *International Symposium on Communications and Information Technologies (ISCIT)*, 723-726, 2010.

26. S. Gören, C.C. Gursoy, and A. Yildiz. Speeding Up Logic Locking via Fault Emulation and Dynamic Multiple Fault Injection, *Journal of Electronic Testing*, 5:525-536, 2015.

27. C.C. Gursoy, A. Yildiz, and S. Gören. On Optimization of Multi-Cycle Tests for Test Quality and Application Time, *Proceedings of IEEE East-West Design and Test Symposium (EWDTS)*, 1-4, 2016.