Design and Security verification of a Finite State Machine

Katayoon Yahyaei
Electrical and Computer Engineering
University of Florida
Gainesville, Fl, U.S.
ka.yahyaei@ufl.edu

Ummay Sumaya Khan
Electrical and Computer Engineering*)*
University of Florida
Gainesville, Fl, U.S
khanummaysumaya@ufl.edu

Daniyal Tahsildar
Electrical and Computer Engineering
U*niversity of Florida*
Gainesville, Fl, U.S
daniyaltahsildar@ufl.edu

Abstract- *Controllers are an important module in most designs in terms of functionality and security. Since they control the flow of the whole system, it's crucial to be carefully tested, validated, and verified. In this work, a sample controller design is considered. To analyze if the provided design implements its specifications correctly, first functional testing is used. Then formal verification with property checking is also performed to provide a stronger guarantee about the design's behavior. The results of the performed validation, verification, and the found bugs are provided, as well as a comparison and discussion on the reliability of these results and the scalability of the approaches.*

I. Introduction

A Finite State Machine (FSM) implemented in hardware is a sequential circuit with a limited number of states. The outputs for an FSM can depend on either the present states (Moore) or a combination of current states and inputs (Mealy). An FSM can work independently or be a controller for a larger circuit. Either way, it becomes very important to test these models as systems are getting larger and less reliable with advancement in computer technology [1]. Testing of FSM models is done to ensure correct functionality and discover any bugs in the design. For this purpose, we test the design using functional and formal verification. Functional verification involves writing test cases for different input combinations while formal verification is used to prove the design's functionality and achieve maximum coverage [2]. The previous works that have been done on the functional verification of the design [3] and [4] can be mentioned that performed on a FIFO. The rest of this paper is organized as followed: section II encourages this work, section III states the problem that is intended to be solved, section IV describes the methodology used, followed by section V which discusses the results, and finally, section VI concludes the paper.

II. Motivation

Integrated circuits (IC) are vulnerable to a variety of existing and emerging trust and security issues that need to be identified as early as possible during the design process, as the famous rule of ten goes. Vulnerabilities can be inserted in the ICs in all stages of the supply chain starting from design a. A vulnerable system is susceptible to malicious alterations that could result in system failure or functional changes [5]. The controller circuit of a device controls the flow of the program, and the overall security of the system heavily depends on it. An FSM which is the realization of the controller is vulnerable to fault injection (FI) and Trojan insertion attacks [8]. Also, some vulnerabilities can be introduced during the synthesis process due to creating don't care states and additional transitions [7]. From a design perspective, security can be viewed as a subset of correctness for a given hardware IP. So, before one can address various security threats and countermeasures for a design, its correctness needs to be checked thoroughly. A designer should guarantee the correct functionality of the design for all possible cases before hardware security can be considered. Functional and formal verification is performed to give a measure of correct behavior and coverage of the hardware IP under test.

III. Problem Statement

This project focuses on the design as well as the verification of an FSM design. The state diagram of the desired FSM is shown in figure 1. It has 4 inputs and 4 states. The transitions are based on the conditions of the inputs. The RTL design of it should be done by using an HDL code. Then the functionality of the design should be verified using functional and formal verification approaches. A functional testbench is needed to rigorously check the implementation of all the possible FSM transitions. For formal verification, 10 functional properties of the design need to be extracted from its behavior and checked with their SystemVerilog assertions. The design should be modified after discovering each functional bug, and all versions of the design need to be kept for the purpose of process observation.

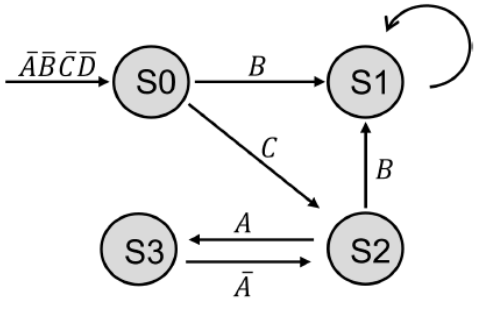

Fig 1: Given State Diagram

IV. METHODS

A. Design Implementation

In this section, the description of the implemented FSM design is provided. The design will have 4 inputs, 4 states and one output which gives the current state the design is in. The design also has a clock and an asynchronous reset signal. The FSM is created based on a two-process model which reduces the number of registers used when compared to a one process model. It consists of two blocks, namely 'sequential logic' and 'combinational logic'.

(i) The sequential logic block contains only sequential elements i.e., State flipflops.

(ii) The combinational logic block contains the logic for the 'next-state'. Next-state depends on current-state and current external inputs.

There is an "out" output signal which can be accessed from the testbench to check and show the states.

FSM state encoding is traditionally done based on design constraints such as area, power, and delay. There are several schemes such as binary, one-hot, and gray encoding to encode the FSM states. The Binary encoding scheme ensures the maximum utilization of State Flipflops, and it is better suited for FSM with a fewer number of states [6]. As the state machine has only 4 states, a binary encoding scheme for the states has been implemented.

Finally, the combinational logic for the next states is implemented. An assumption is made that only one input can change per clock cycle. And if an input changes that does not cause a state transition, the FSM remains in its current state. A case statement is used to define the behaviour of the design.

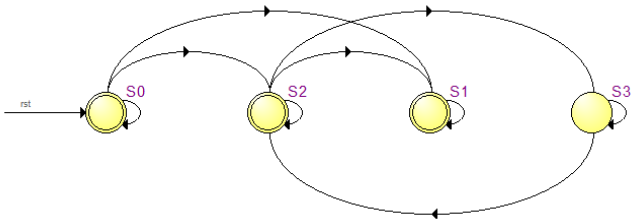

Fig 2: Generated State Diagram in Quartus

- While in S0 state, if input B becomes '1' and input C becomes '0', the next state will be S1. On the other hand, if input C becomes '1' and input B becomes '0', the next state will be S2.
- While in S1 state, there is no logic as there is no state to go from S1 state. The machine will be stuck in this state until it is reset.
- While in S2 state, if input B becomes '1' and input A becomes '0', the next state will be S1. On the other hand, if input A becomes '1' and input B becomes '0', the next state will be S3.
- While in state S3, if input A becomes '0', the next state will be S2.

This design is simulated in Quartus, and the following FSM is generated which is identical to the provided FSM.

Design Versions and Changes History:

- In the first design, the inputs were defined as register which resulted in latches and a one cycle additional delay. Later, the inputs were changed to wire and the latch and clocking issues were resolved in the following versions of the design.
- In the 2nd version of the design, a delay was used in the sequential logic block to improve setup and hold violations. As this design is simple, it's not necessary to use it. In the later versions, the delays were removed.
- In the 3rd as well as the final version of the design, the sensitivity list of the combination logic of the FSM was changed from "always @(*) begin" to "always @(state_r,A,B,C,D) begin" and, the definition of state variables was changed from "localparam" to "parameter". These changes were made due to the errors while doing formal verification in Jaspergold. Additionally, the default case was added in the case statements.

B. Functional Verification using a testbench

A System Verilog testbench was created that checks the current state the FSM is in by triggering different inputs. Every testbench involves creating test signals, connecting the testbench with the module being tested, triggering inputs, and checking the correctness of the output. For this design, we have an asynchronous reset signal which sets the design back to S0 and all signals are triggered on a positive clock edge. A signal is used to verify the correct output state for each state transition. If the state determined by the FSM does not match it, an error message is displayed on the console. The implemented design has 3 versions of the FSM, and a testbench is written for each of them. It is as follows:

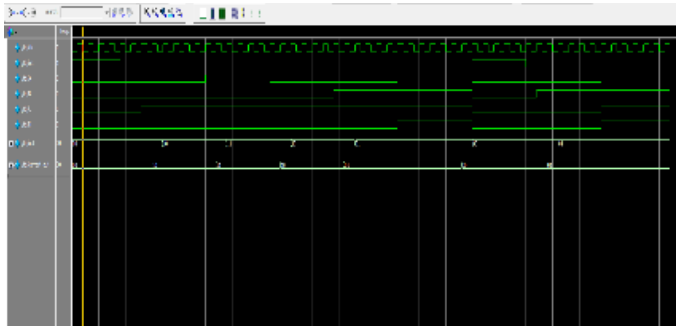

Fig 3: Simulation results for FSM Version_1

- The 1st version of the testbench was quite simple and limited, which checks each state transition just once by triggering every input one time and checking the state transition. In the case of S1 the design was reset to check both transitions from S0 and S2. The reset is

turned on at the beginning of the test and then turned off after a few cycles. This ensures that the design is in S0 at the start of the test. This is done for all versions of the testbench.

```
# Reading C:/altera/13.1/modelsim_ase/tcl/vsim/pref.tcl
# Loading project FSM_testing
# Compile of fsm_1.v was successful.
# Compile of fsm_1_tb.sv was successful.
# 2 compiles, 0 failed with no errors.
ModelSim> vsim -gui work.tb
# vsim -gui work.tb
# Loading sv_std.std
# Loading work.tb
# Loading work.fsm
add wave -position insertpoint sim:/tb/*
VSIM 3> run -all
# Tests completed
```

Fig 4: Console Output for FSM Version_1

- The 2nd version of the test bench for this version was limited to changing A, C, D at random while keeping B at 0. This was done to prevent the test from being stuck at S1. The testing can only resume once the design was reset from being in S1. This process was automated in the 3rd version along with adding tests for B.

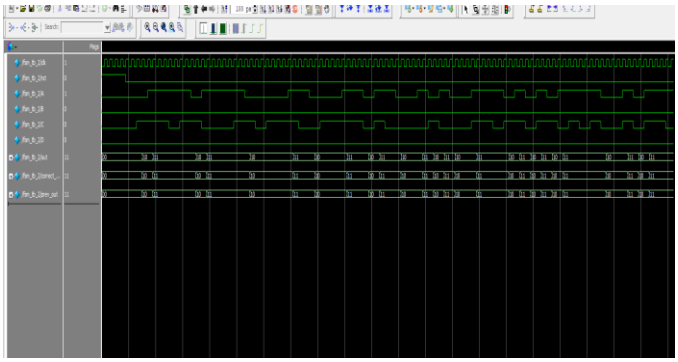

Fig 5: Simulation results for FSM Version_2

```
# Compile of fsm_2.v was successful.
# Compile of fsm_2_tb.sv was successful.
# 2 compiles, 0 failed with no errors.
VSIM 4> vsim -gui work.fsm_tb_2
# vsim -gui work.fsm_tb_2
# Loading sv_std.std
# Loading work.fsm_tb_2
# Loading work.fsm_2
add wave -position insertpoint sim:/fsm_tb_2/*
VSIM 6> run -all
# Tests completed
```

Fig 6: Console Output for FSM Version_2

- In the last version of the testbench, 1000 tests were run by randomizing A, C, and D while B was triggered only when A and C are 0. This was done to prevent having multiple inputs triggered at the same clock cycle. Since A and C are independent from each other for state transitions, it was considered feasible to have some cases when they are triggered together. This was not the case for B when the current state is S0 and both B and C are triggered. Similarly, for S2 where A and B are '1'. For a continuous random test, the reset is triggered every next cycle the state is S1. The testbench also includes the total number of passed and failed tests for easier debugging.

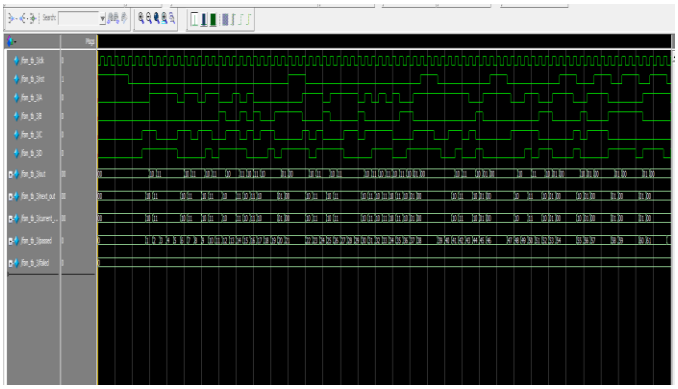

Fig 7: Simulation results for FSM Version_3

```
# Compile of fsm_3.v was successful.
# Compile of fsm_3_tb.sv was successful.
# 2 compiles, 0 failed with no errors.
VSIM 9> vsim -gui work.fsm_tb_3
# vsim -gui work.fsm_tb_3
# Loading sv_std.std
# Loading work.fsm_tb_3
# Loading work.fsm_3
add wave -position insertpoint sim:/fsm_tb_3/*
VSIM 11> run -all
# Tests completed : 1000 passed, 0 failed
```

Fig 8: Console Output for FSM Version_3

The 3rd version of the testbench achieves maximum coverage of the implemented design. A variable is declared for the total number of tests to be run which tests 1000 different combinations of randomized inputs. This variable can be changed to test as many combinations as required. Since the reset is automated to go to S0 whenever the design is in S1, extensive test cases are generated without being stuck in one state. The resetting is also done one cycle later after the system is in S1. This ensures and tests that there is no state transition when the machine is in S1, and any other input is triggered, without having to write additional code for it. Testing so many different combinations reassure the programmer that the design is functioning correctly both structurally and behaviorally.

C. Formal Verification using assertions

The properties of the FSM design formally checked are listed in table 1.

Property name	*Natural language description*	*SystemVerilog assertion*

P1_reset	Check reset function	rst_s==1'b1)|-> ((FSM_design.state_r==S0)&&(FSM_design.out_r==2'b00))
P2	check s0 to s1 transition	@(posedge clk_s) ((FSM_design.state_r==S0)&&(B_s==1'b1) &&(C_s==1'b0)) |-> (FSM_design.next_state==S1);
P3	check s0 to s2 transition	@(posedge clk_s) ((FSM_design.state_r==S0)&&(B_s==1'b0) &&(C_s==1'b1)) |-> (FSM_design.next_state==S2);
P4	considering c=0 and b=0 in the s0 to s2 transition	@(posedge clk_s) ((FSM_design.state_r==S0)&&(B_s==1'b0) &&(C_s==1'b0)) |-> (FSM_design.next_state==S0);
P5	check s1 to s1 transition	@(posedge clk_s) (FSM_design.state_r==S1) |-> (FSM_design.next_state==S1);
P6	check s2 to s1 transition	@(posedge clk_s) ((FSM_design.state_r==S2)&&(B_s==1'b1) &&(A_s==1'b0)) |-> (FSM_design.next_state==S1);
P7	check s2 to s3 transition	@(posedge clk_s) ((FSM_design.state_r==S2)&&(B_s==1'b0) &&(A_s==1'b1)) |-> (FSM_design.next_state==S3);
P8	considering a=0 and b=0 in checking s2 to s3 transition	@(posedge clk_s) ((FSM_design.state_r==S2)&&(B_s==1'b0) &&(A_s==1'b1)) |-> (FSM_design.next_state==S3);
P9	check s3 to s2	@(posedge clk_s) ((FSM_design.state_r==S3)&&(A_s==1'b0)) |-> (FSM_design.next_state==S2);
P10	check s3 considering a=1	@(posedge clk_s) ((FSM_design.state_r==S3)&&(A_s==1'b1)) |-> (FSM_design.next_state==S3);

Table 1: SystemVerilog assertions for the designed FSM

To extract the functional properties, the state diagram of the FSM should be considered. Also, the HDL code can be helpful in finding the properties. One can start from the beginning of the FSM, in other words, the reset and initial state. When the reset is high, it is expected to be in the S0 state and has 0 in the output. Property P1_reset checks if the reset is high. Then checks if the state variable of the FSM is S0 and if the output is equal to 0. Since the reset of this FSM is designed asynchronously, then there is no need to check the edge of the clock in the assertion.

Moving on to the FSM transitions, there is a transition from S0 to S1. To check if the transition is happening correctly, first, it should be confirmed that we are in state S0. Then the conditions for this transition to happen should be obtained from the state diagram or the Verilog code, which are B=1 and C=0. Once all the conditions mentioned above hold true, it should be checked if the state S1 is assigned to the state variable of the FSM. The same goes for the transition from S0 to S2, except that the conditions are B=0 and C=0. When the state variable is S0, there are 4 possibilities for the values of B and C. The values of A and D inputs do not matter at this point. Two of those possibilities are already addressed in P2 and P3. It is assumed that only one input can change per clock cycle. Therefore, B=1 and C=1 cannot take place and only one more possibility is left to be addressed, which is B=0 and C=0. The property P4 is checking if the state variable is S0, B=0, and C=0, then the state variable should keep its value and there should not be a transition to any other state.

The next transition to consider can be S1 to S1. It does not have any conditions and once the system is in S1 it should stay there. Property P5 checks if the state variable is S1, and the next state register is assigned to S1. Therefore, the state variable keeps the same value in the following clock cycles. Like P2 and P3, property P6 is checking the transition from S2 to S1, and property P7 is checking the transition from S2 to S3, which is based on the values of B and A. like P4, a property is needed here to check that if B=0 and A=0, no transitions happen. Property P8 is responsible for that. The same thing goes for the transition from S3 to S2 based on the condition on the value of A, which is checked in property P9. The other possible value of A while in S3 is checked in property P10.

```
================================================================
SUMMARY
================================================================
          Properties Considered              : 20
               assertions                    : 10
                - proven                     : 10 (100%)
                - bounded_proven (user)      : 0 (0%)
                - bounded_proven (auto)      : 0 (0%)
                - marked_proven              : 0 (0%)
                - cex                        : 0 (0%)
                - ar_cex                     : 0 (0%)
                - undetermined               : 0 (0%)
                - unknown                    : 0 (0%)
                - error                      : 0 (0%)
               covers                        : 10
                - unreachable                : 0 (0%)
                - bounded_unreachable (user): 0 (0%)
                - covered                    : 10 (100%)
                - ar_covered                 : 0 (0%)
                - undetermined               : 0 (0%)
                - unknown                    : 0 (0%)
                - error                      : 0 (0%)
determined
```

Fig 9: Assertion Summary generated in JasperGold

As the summary of the verification report shows in fig. 8, all the preconditions in the assertions are covered and the properties are asserted so there is no property violation and no counterexample. Therefore, there are no functional bugs in the design that require fixing. The functionality of the FSM is fully checked and verified with the above assertions. It is proven that all the desired state transitions happen in the FSM design and there is no transition between the states other than what was intended. To extend the functional property list of this design, the output and the intermediate registers can be checked to ensure that the correct output is being assigned according to the current state.

V. RESULTS

A. *Design of the Finite State Machine and testing:*

The FSM design is based on a two-process model with a clock and asynchronous reset. In the initial design, the inputs were registered which caused an additional delay in the clock cycle (See *Fig 3.*). Observing simulation results was vital for exposing this bug. The 2nd version of the design resolved this issue while also setting up a small delay in the register to help with preventing set-up and hold time violations. For the testing of the design, it was important to test a large set of different combinations. To do this the inputs were randomized in the testbench. This posed the problem of the machine being stuck in S1. So, the second version only tests S2 and S3 extensively. The final version improved on this by testing all states extensively and automating reset to come out of S1 for testing. Also, the delay in the registers was removed since it did not make much difference in this case. For all the testbenches the output state was verified for each state.

The extracted functional properties check for all the transitions in the FSM. It also checks if any unintended transition happens. Since the design is rather small and the previous functional testing and debugging were done carefully, there were no bugs left in the design to be detected with formal verification. There were only minor syntax changes needed to be made to the design to be analyzed in JasperGold; for example, changing "localparam" to "parameter" to define the state names and encodings.

B. *Ease of Debugging:*

Formal testbenches are easier to debug since they test cases that should be true when asserted, so any exception is easily captured and displayed on the console. Also, it automates the testing process for a lot of cases. In the case of a functional testbench, the simulation results must be observed and ensure that the inputs are driven correctly. Any warnings that could lead to a design failure also need to be looked at. Often, these warnings get overlayed by other process statements displayed on the console.

C. *Coverage:*

- **Code Coverage:**

The code coverage of the design is 85.42% due to the assumption of the project.

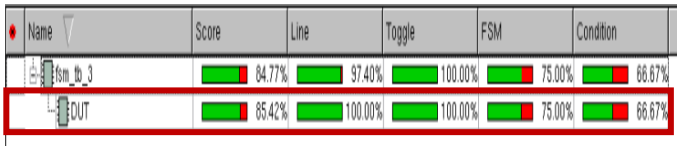

Fig 10: Code coverage of the design

The Line, Toggle, FSM as well as the Condition coverages of the design are 100%, 100%, 75%, and 66.67% respectively.

The reason for lower FSM and condition coverage is the following:

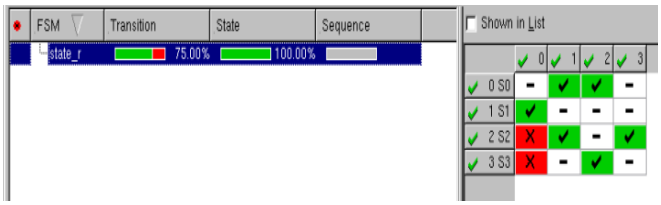

Fig 11: FSM coverage of the design

Provided FSM does not have any state transition from S2 to S0, and from S3 to S0. That's why FSM coverage is lower.

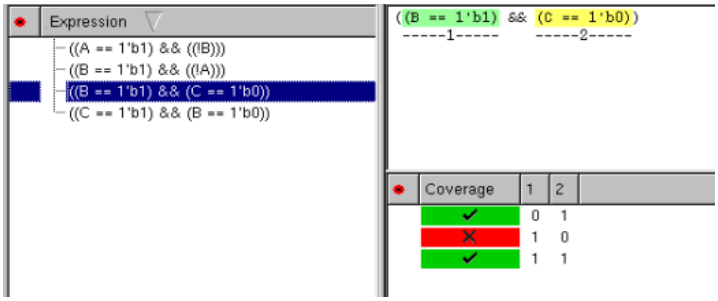

Fig 12: Condition coverage of the design

There was a condition for the FSM design that only one bit can be changed/flipped at a time in each clock cycle. That's why the condition shown here (B=1 & C=1) cannot be covered and so, the condition coverage is lower.

- **Assertion Coverage:**

The asserting coverage is 100%, all the assertions verified the implemented FSM. The coverage result was shown in Fig.9 in the Formal verification part.

D. *Scaling to larger systems:*

The State Machine implemented was simple enough to test all combinations of the inputs. Scaling to a larger system would involve more complex testing. For a larger system, having a functional test bench would require testing various combinations and it is possible that an important input configuration might be missed out. Also setting up a testbench for a larger design gets more time-consuming as the design scales. In the case of formal verification, complete testing can be achieved for cases that might be left out during functional testing of the same design. Assertions are particularly useful for detecting parts of a design that are incomplete or error-prone and can be quickly set up for a design. For larger designs, the complexity of these assertions can vary based on the signals and parameters being tested. Also, finding all possible functional properties of the design can be challenging if the design is complicated. Missing a functional property to verify in the formal verification can also result in not having confidence in the design. Using an automated property generation tool can help with addressing that.

E. Trade-offs between formal and functional testing:

Functional testing requires setting up test cases for every scenario and can be automated by randomizing the inputs. It also requires keen observation of simulated results for debugging. While functional verification gives greater control to test exact scenarios for test cases, formal verification gives more exhaustive testing of the same test case and is mostly automated. It tests the design for exceptions rather than checking the correct output generation for each case. Formal verification gives better coverage of test cases that could be missed out when testing the design functionally.

VI. CONCLUSIONS

A simple FSM was designed and verified using both Functional and Formal Verification methods. Also, a detailed analysis of both methods is done for the implemented design. In the case of functional verification, every test pattern needs to be generated while in the case of formal verification all combinations can be tested exhaustively based on the assertion property. For the FSM created, all assertions cover precondition and successfully assert the properties. Also, writing testbenches to test the functionality of the design is much simpler compared to writing assertions which requires a good understanding of the design being generated and the combinations that can cause errors. While writing testbenches and assertions is time-consuming, any design that passes all these tests is said to be functionally correct and bug-free. This design can now be tested for its security without worrying about its correctness.

VII. REFERENCES

[1] D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines-a survey," in Proceedings of the IEEE, vol. 84, no. 8, pp. 1090-1123, Aug. 1996, doi: 10.1109/5.533956.

[2] M. Girish, G. Gopakumar and D. S. Divya, "Formal and Simulation Verification: Comparing and Contrasting the two Verification Approaches," 2021 2nd International Conference on Advances in Computing, Communication, Embedded and Secure Systems (ACCESS), 2021, pp. 41-44, doi: 10.1109/ACCESS51619.2021.9563305.

[3] Cummings, Clifford E. “Simulation and synthesis techniques for asynchronous FIFO design.” SNUG 2002 (Synopsys Users Group Conference, San Jose, CA, 2002) User Papers. 2002.

[4] Rizvi, N. Z., Arora, R., & Agrawal, N. (2015). Implementation and Verification of Synchronous FIFO using System Verilog Verification Methodology. Journal of Communications Technology, Electronics and Computer Science, 2, 18-23.

[5] Xiaoxiao Wang, M. Tehranipoor and J. Plusquellic, "Detecting malicious inclusions in secure hardware: Challenges and solutions," 2008 IEEE International Workshop on Hardware-Oriented Security and Trust, 2008, pp. 15-19, doi: 10.1109/HST.2008.4559039.

[6] K. Kuusilinna, V. Lahtinen, T. Hämäläinen, and J. Saarinen, “Finite state machine encoding for VHDL synthesis,” IEE Proc. Comput. Digit. Tech., vol. 148, no. 1, pp. 23–30, Jan. 2001.

[7] F. Farahmandi and P. Mishra, "FSM Anomaly Detection Using Formal Analysis," 2017 IEEE International Conference on Computer Design (ICCD), 2017, pp. 313-320, doi: 10.1109/ICCD.2017.55.

[8] A. Nahiyan, K. Xiao, K. Yang, Y. Jin, D. Forte, and M. Tehranipoor, "AVFSM: A framework for identifying and mitigating vulnerabilities in FSMs," 2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC), 2016, pp. 1-6, doi: 10.1145/2897937.2897992.