Volume

1.3

UNIVERSITY OF TEHRAN AND CINI CYBERSECURITY NATIONAL LABORATORY

Test and Testability of SAYAC Embedded System

UNIVERSITY OF TEHRAN AND CINI CYBERSECURITY NATIONAL LABORATORY

SAYAC Post-manufacturing Test Analysis

Version 1.1
Test and Testability of Embedded Core & Memory

Under the supervision of Prof. Zain Navabi[1], Prof. Paolo Prinetto[2]
[1]Worcester Polytechnic Institute, [2] CINI Cybersecurity National Laboratory
navabi@wpi.edu, paolo.prinetto@polito.it

November 2022

Proprietary Notice

Authors

Nooshin NOSRATI (PhD candidate, University of Tehran) nosrati.nooshin@ut.ac.ir
Ebrahim NOURI (Master Student, University of Tehran) nouri.ebrahim@ut.ac.ir
Fatemeh MOHAMMADZADEH (Master Student, University of Tehran)
f.mohammadzadeh8@ut.ac.ir
Vesal BAKHTAZAD (Student, University of Tehran) bakhtazad.v@ut.ac.ir
Shahab KARBASIAN (PhD candidate, University of Tehran) sh.karbasian@ut.ac.ir
Zainalabedin NAVABI (Full Professor, University of Tehran) navabi@ut.ac.ir
Paolo PRINETTO (Director, CINI Cybersecurity National Lab) paolo.prinetto@polito.it

Acknowledgments

This work is the result of a fruitful collaboration and friendship between the CINI Cybersecurity National Laboratory and the University of Tehran, whose inspirers are certainly Prof. Paolo Prinetto and Prof. Zainalabedin Navabi.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 Introduction

The focus of this part is on post-manufacturing testing and testability of the SAYAC architecture. Testing is a systematic process used to make sure that an IC has been correctly manufactured and is free of defects. This correctness is verified with the application of appropriate inputs (called test patterns or test vectors) and the observation of the circuit's response which should be equal to the expected one. Various test architectures and methods have been designed to make an IC testable and measure its testability. Some of these test architectures like Full-scan, Multiple-scan, Logic BIST, Memory BIST, and boundary scan are well-established and widely used in academia [4, 6-9] and industry [5, 11, 12]. In the following sections, we will briefly discuss these popular test techniques and explain how to incorporate them into the SAYAC architecture and perform fault simulation.

Before diving into the SAYAC test architectures and testability evaluation, we need to determine fault models and the level of abstraction in which faults are injected and simulated. Then, we need to develop some mechanisms for test data generation, fault list generation, fault injection, and fault simulation.

The traditional fault model used in ATPG flows is the single stuck-at fault model that is considered in this work as well. The structural stuck-at faults are injected into the gate-level description of the SAYAC processor to perform fault simulation. In this work, we are provided with the RTL implementation of the SAYAC architecture. Therefore, first of all, an RTL-to-Gate level synthesis process is required to generate the gate-level netlist of SAYAC. The line-oriented fault collapsing procedure discussed in [3] is used to generate a fault list for the SAYAC processor. On the other hand, test patterns are generated using an open-source test pattern generator. The synthesis process, test pattern generation, and fault list generation are automated by a homemade toolchain that is discussed in Section 2.

After generating the gate-level netlist of SAYAC, test data, and the fault list, fault injection and fault simulation are implemented in an HDL testbench and a TCL script. Section 3 explains three different methods for fault injection and fault simulation.

Sections 4 to 7 explain Scan DFT testing, Logic BIST, Memory BIST, and Boundary-scan IEEE 1149.1 standard test architectures for the SAYAC processor, respectively. All these test architectures are implemented in a VHDL simulation environment in which test is spoken in the language of design.

The last section of this part discusses fault injection and fault simulation in the SystemC environment. An intrusive fault injection mechanism has been developed in the SystemC environment to provide fault simulation capability in an open-source environment. In addition, it provides the capability of using the same test program for running on the ATE and the virtual tester that mimics the ATE behavior in our models.

2 UT-DATE Toolchain Development

The toolchain developed in this work is to generate all files required for testing a design and make the process of testing automate. Figure 2-1 shows an overview of our toolchain, called UT-DATE (University of Tehran-Design and Test Environment). As shown in the figure, it takes an RTL description of the circuit under test (CUT) in an HDL format, i.e., Verilog or VHDL, and automatically generates the test files and finally test report. Some stages are dependent on the others, meaning that one phase might use outputs of the other. Hence, it is recommended to run all steps in specified order one after the other.

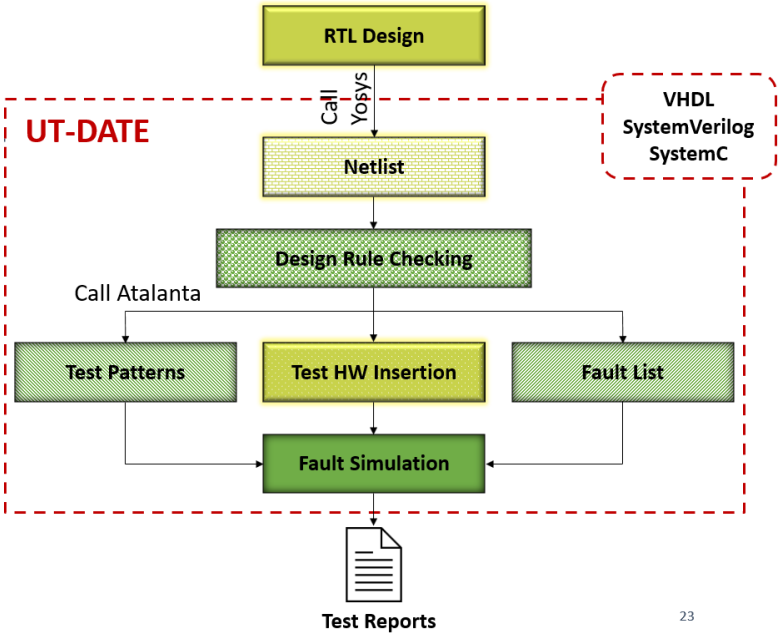

Figure 2-1 Overview of UT-DATE toolchain

Figure 2-2 shows the overall flow of UT-DATE regarding the generated files in each step. After running the toolchain, two main categories of files will be generated, synthesis and test-related files. The synthesis directory contains pre- and post-mapped synthesis outputs. The former is netlists of the default library of *Yosys* and the latter is netlists mapped to our custom library. The test directory gathers all files needed for test purposes including the ".bench" format of the CUT, fault list, and test set. The following subsections explain the main outputs of the toolchain that are used in this work for testing the SAYAC processor.

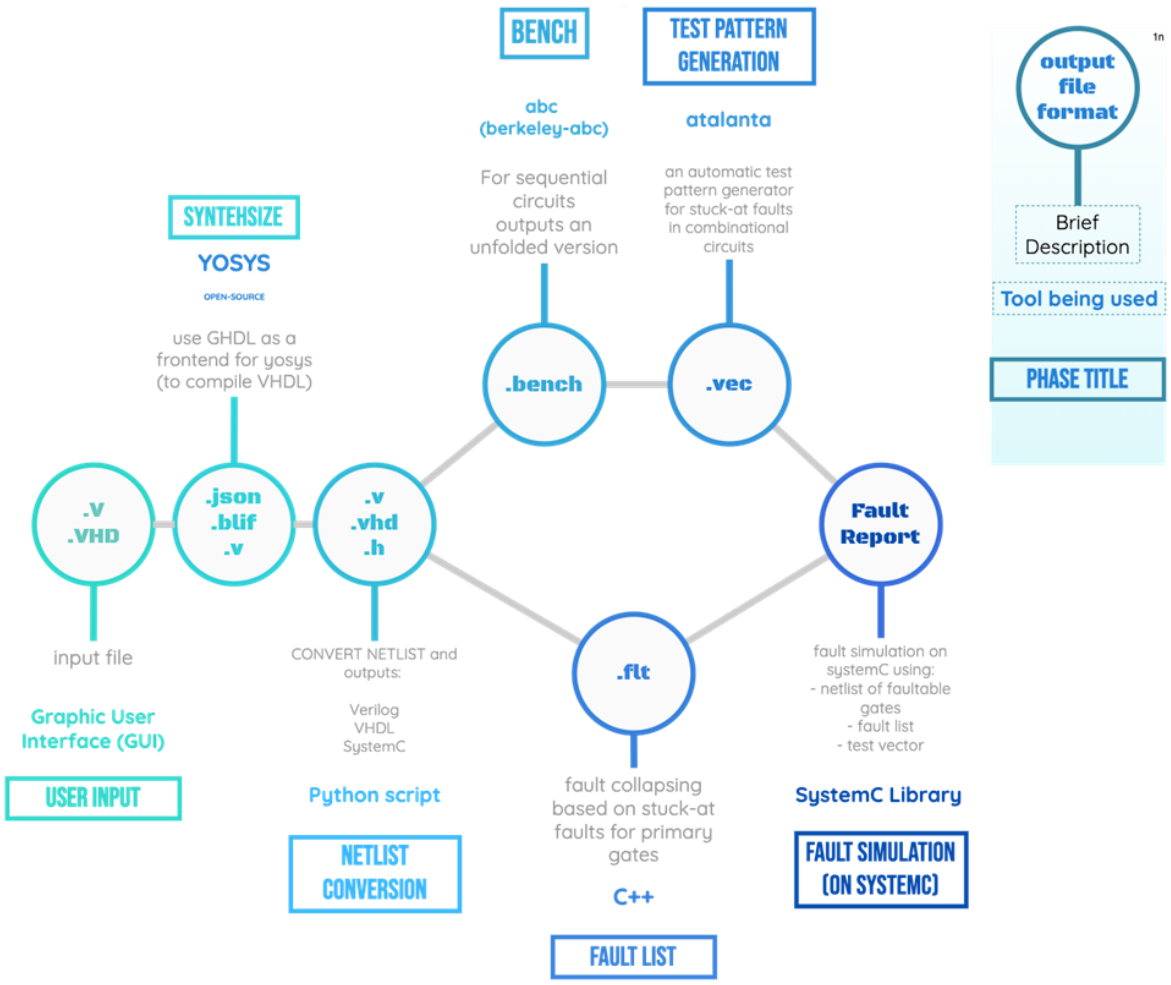

Figure 2-2 Overall flow of UT-DATE backend

2.1 Synthesis

The toolchain's flow starts from an HDL description of the design in form of Verilog/VHDL. It uses *Yosys* open synthesis suites [1] to synthesize the behavioral or/and structural description of the design. Since *Yosys* does not support the VHDL input, we employed the GHDL-Yosys plugin as an extension to provide VHDL synthesis as well. GHDL acts as a frontend for *Yosys*, compiling VHDL code to a format digestible for *Yosys*.

Yosys synthesizes the HDL description to its default library. In addition, it can map it to any given technology. In this work, we used our Gate-level component library that provides additional features for scan insertion and fault simulation. Like the input format, *Yosys* is not able to generate the synthesized netlist in the VHDL format. However, it supports several non-HDL formats like JSON (JavaScript Object Notation), BLIF (Berkeley Logic Interchange Format), and AIGER (And-Inverter Graphs). Using JSON format, we convert the synthesized netlist to three conventional HDL formats Verilog, VHDL, and SystemC. It is obvious that these conversions require having our component

library in three relevant versions. Figure 2-3 shows a snapshot of the typical input and output of the *Yosys* synthesizer.

The Verilog, VHDL, and SystemC netlists generated for SAYAC in this step execute the same functionality as the RTL behavioral model of SAYAC. The functionality verification has been done through a Testbench which runs the matrix multiplication benchmark on SAYAC. The testbench instantiates both RTL and gate-level netlist models of SAYAC and feeds them with the same stimulus.

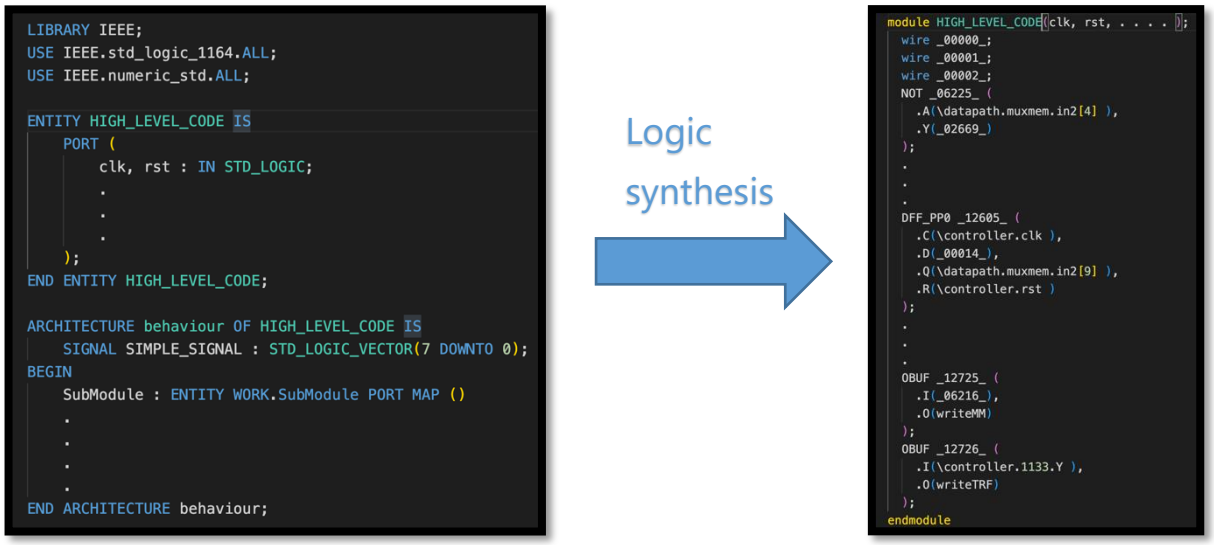

Figure 2-3 A typical input and output of the Yosys synthesizer

2.2 Test Pattern Generation

The post-manufacturing test process requires a fault list and a test set to perform fault simulation and evaluate the testability of a CUT. To generate the test set, our toolchain calls *Atalanta* [2] which is an automatic test pattern generator for stuck-at faults in combinational circuits. The input netlist format for *Atalanta* is ISCAS89 netlist format namely the "bench" format. Therefore, we need to have the netlist of the previous step in the bench format.

Toward this purpose, we employ the *ABC* (aka, Berkeley-ABC) tool for technology mapping of Yosys's internal gate library to the target format. In this mapping process, we just consider the combinational part of the netlist and limit the *ABC* library to non-sequential logic gates. The reason for this is the *Atalanta* tool just works for combinational circuits and requires the unfolding of sequential circuits to separate the combinational part. By unfolding a circuit, its registers are separated and ignored. In addition, the register outputs become pseudo primary inputs (PPIs) for the unfolded circuit, and register inputs become pseudo primary outputs (PPOs) of this circuit. We incorporate the unfolding process into the ABC technology mapping. Along this line, *ABC* treats sequential logic gates (like D-Flip-Flops) as black boxes and just leaves their input/output ports as pseudo primary output/input ports of the design which implies the unfolding process.

As another requirement for *Atalanta*, the bench file must not have any floating net. Therefore, we check the unfolded bench file before feeding it to *Atalanta* to remove floating nets. Finally, pruned bench file is fed to *Atalanta* to generate a set of test patterns for the combinational part of the CUT. *Atalanta* can compute the correct CUT responses for the generated test patterns as well. Figure 2-4 shows a snapshot of the output of Atalanta for a given circuit.

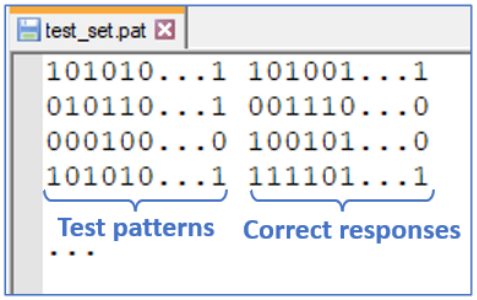

Figure 2-4 A typical output of Atalanta

2.3 Fault List Generation

To generate a fault list for fault injection and fault simulation, the line-oriented fault-collapsing technique discussed in [3] is employed for the primary gates of our component library. In this technique, instead of initially arranging all possible faults on the ports of gates, we only place those faults on circuit lines that cannot collapse any further. The rules shown in Table 2-1 summarize the line-oriented fault-collapsing for the primary gates. Based on this table, our toolchain generates a comprehensive list of faults in which each fault has a wire name and a fault value. The wire name uses a hierarchical naming to show where the faulty value is to be injected. Therefore, the naming convention must obey simulation engine rules. Assuming two levels of hierarchy for the netlist, both the testbench name and the instance name of the netlist (CUT) are set by the user. The typical format of a fault is shown below:

$$testbench\_name/fut\_name/gate\_name/port\_name \quad [0\ or\ 1]$$

To check the correctness of our fault list, we verified that with the method presented in [3] for Verilog netlists. In [3], the fault list is generated by a PLI function ($faultCollapsing) that employs the same fault collapsing technique for fault list generation.

Table 2-1 Line-oriented fault collapsing [3]

Type of target gate	Put this (these) fault(s) on the gate's input line(s)
AND, NAND	SA1
OR, NOR	SA0
INV, BUF	None

FANOUT	SA0, SA1
XOR	SA0, SA1
Primary Output	SA0, SA1
Primary Input	None

2.4 Fault simulation in SystemC

To perform fault simulation in the SystemC environment, we gather the generated files in the above steps into one directory (netlist, test set, fault list) and then generate a testbench code to handle fault injection and test pattern application. The generated testbench varies depending on whether the circuit under test is sequential or pure combinational and also the number of ports and their corresponding size.

Along this line, a "fault injector" module was developed to read from the fault list, injects a fault to the corresponding gate/signal, applies test vectors one by one, and then compares the output with the fault-free circuit (golden model). As discussed above, we need to provide two versions of the fault injector, one for combinational and the other for sequential circuits. Based on the existence of DFFs in the netlist of the CUT, the toolchain decides to use one of them. At the end of file generation, we call a "make" file to compile and output the testability results into the "faultReport.txt" file. This report file contains the detail of fault simulation, fault injection process, and output comparison for both faulty and fault-free models along with overall fault coverage. The details of this part will be discussed in Section 9.

2.5 Scan Insertion

To automate the process of making DUT testable and inserting scan registers, three steps have been considered as shown in Figure 2-5.

Design Rule Check: The first step to implement a scan design is to identify and repair all scan design rule violations. After that, the design is ready to be tested. The design must comply with a set of scan design rules and a set of design styles must be avoided. Table 2-2 shows a set of scan design rules, the first four of which is implemented in the UT-DATE toolchain.

Scan Synthesis: Once all scan design rule violations are identified and repaired, scan synthesis is performed to convert the ready-to-test design into a scan design without affecting the functionality of the original design. The scan synthesis includes four phases:

- Scan Configuration: the general structure of the scan design is determined. The main decisions that are made at this stage include: 1. The number of scan chains, 2. The type of

scan cells, 3. Storage elements to be excluded from the scan chain, 4. The arrangement of the scan cells within the scan chain.

- Scan Replacement: scan replacement replaces all original storage elements in the testable design with their functionally equivalent scan cells.
- Scan Stitching: Scan stitching refers to the process of connecting the output of each scan cell to the scan input of the next scan cell in each scan chain.
- Scan Reordering: Scan reordering is the process of reordering scan cells in scan chains, based on the physical scan cell locations. Some reordering algorithms such as Traveling Salesman, Genetic algorithm, and Improved traveling Salesman are used to minimize the power consumption, or the amount of interconnect wires used. If physical location of each scan cell instance is not available, a random scan order is considered. DATE-UT implements a random scan order based on the order of the storage elements in the design.
- **Scan Verification:** The scan design now includes one or more scan chains for scan testing. This step involves verification of the circuit's functionality has not changed by simulating in the normal functional mode, as well as verifying the scan shift operation and scan capture operation.

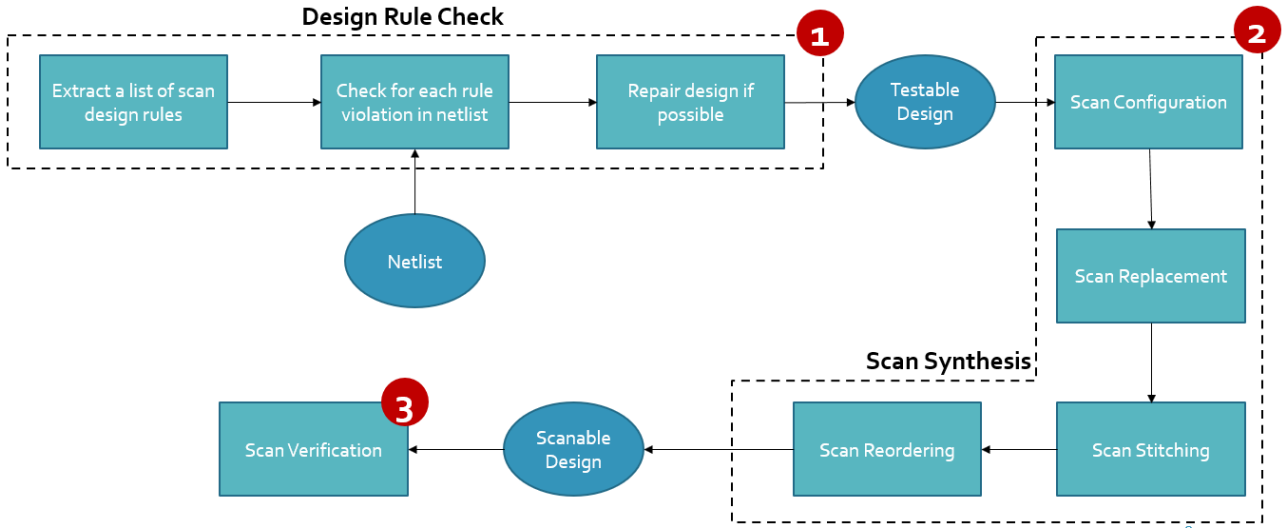

Figure 2-5 Scan DFT Flow

Table 2-2 Scan design rules

Scan Design Rule
Gated clocks
Derived clocks

Sequentially controlled asynchronous set/reset signals
Combinational feedback loops
Tristate buses
Bidirectional I/O ports
Floating Buses
Separate memory blocks

2.6 LBIST Insertion

To automate the process of adding Built-In-Self-Test (BIST) hardware around scan-inserted version of the design, the LBIST option is added to UT-DATE. The LBIST tab is only available and valid in continuation of the scan tab.

In LBIST, the toolchain places all the necessary components of a specific type of BIST around the design and provides a BIST-inserted module with all the primary inputs/outputs of the main module. Inserting the LBIST hardware involve three categories of properties: user-defined, DUT-dependent that are inferred based on design specifications, and BIST- dependent that are fixed for a specific type of BIST.

At first, the user selects the type of the desired BIST. He/she then fills out all the user-defined fields that are enabled for that particular BIST type. Under the hood, the toolchain grabs the scan-inserted design (provided by running the Scan tab) and extracts necessary information to build the BIST. Based on this information and those provided by the user the BIST module is constructed. The next step is the integration of BIST module around the DUT and binding the required ports. The output module retains all the primary ports of the DUT module with original naming. In this version of UT-DATE, we support only RTS BIST. Figure 2-6 shows a typical formation of BIST hardware around a DUT.

The user-defined parameters include polynomial and seed values for test pattern generator (TPG) and output response analysis (ORA) modules. Such parameters can be selected randomly or can be extracted from the BIST evaluation process. The BIST tab also facilitates the process of BIST evaluation through fault simulation.

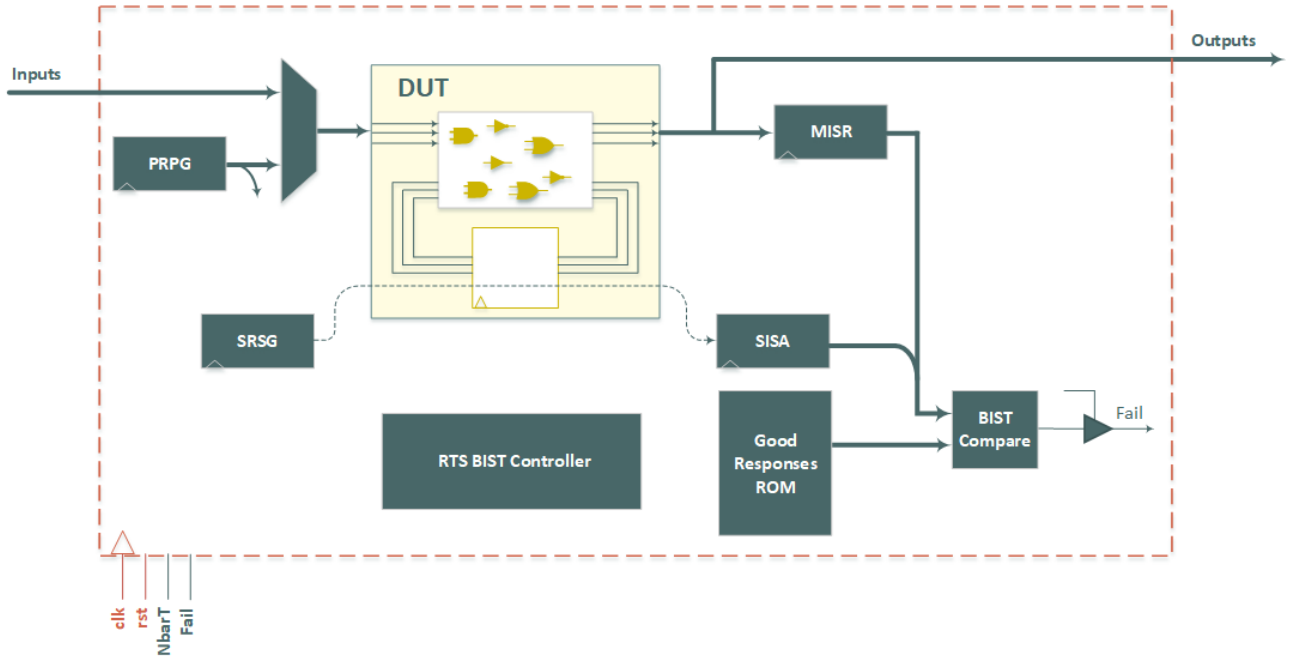

Figure 2-6 A typical formation of BIST hardware around DUT

3 TCL-based Fault Injection and Simulation

Fault simulation is a process in which faultable and golden versions of the circuit under test (CUT) are instantiated and fed by test data. A fault from the fault list is read and injected into the faultable model using a fault injection process to make a faulty model. A test pattern from the test set is read and applied to both golden and faulty models. Then, the outputs of the faulty model are compared with the golden one to check whether the injected fault is detected or not. If it is not detected, the next test pattern is applied as long as the injected fault is detected, or the test set is finished. Otherwise, the injected fault is removed, the next fault is injected, and the same procedure for test pattern application is repeated for the new injected fault. In a nutshell, the procedure consists of two nested loops. The outer loop considers every fault in the fault list. The inner loop applies test patterns of the test set to the faultable and golden circuits and compares their outputs until one test pattern detects the injected fault. Figure 3-1 shows the pseudo-code of the fault simulation process discussed above. Figure 3-2 also shows a block diagram of the components involved in fault simulation [3]. As discussed in the introduction section, fault injection is performed at the gate level. Hence, the faultable model is in the gate-level netlist form. However, the golden model can be at RT-level for the sake of simulation speed.

Fault injection and fault simulation can be implemented by a combination of an HDL testbench and a TCL script in ModelSim-Altera. Based on the participation of each part (HDL testbench & TCL script), we have implemented fault injection and fault simulation in three different ways: All-in-TCL, Dynamic TCL, and Static TCL.

```
Instantiate Golden Model
Instantiate Faultable Model

Given Test set T, n test vectors
Given Fault List F, m faults

For j in 1 to m loop -- every f in F
      Inject fj;
      For i in 1 to n loop -- every t in T
            While fj is not detected begin
                  Apply Ti;
                  Simulate faulty & golden circuits
                  Compare faulty & golden outputs
            End while
      End for
      Remove fj
End for
```

Figure 3-1 Pseudo code of fault simulation [3]

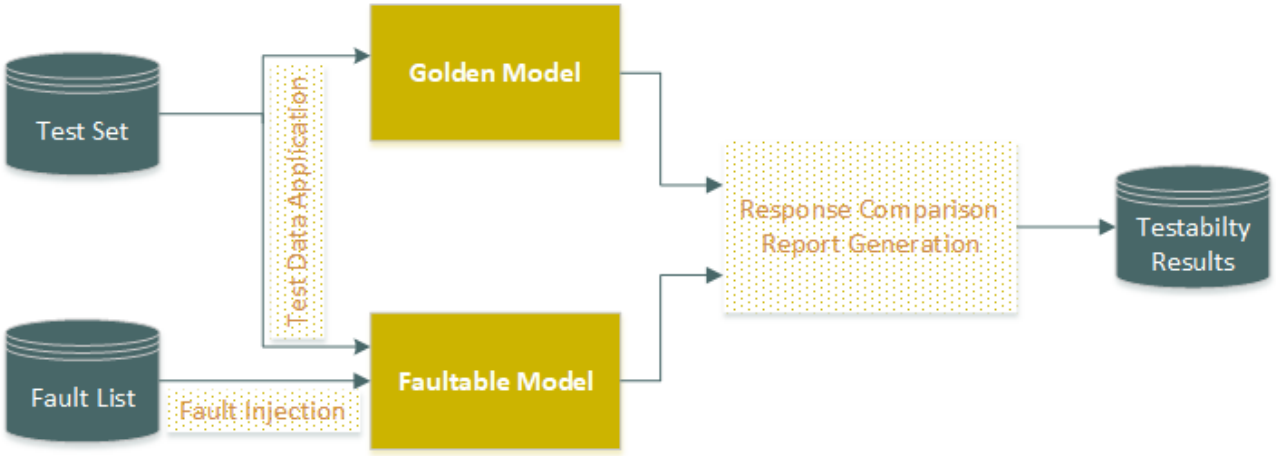

Figure 3-2 Block diagram of fault simulation process [3]

All-in-TCL: In this method, both fault injection and the main part of fault simulation are implemented in the TCL script. The HDL testbench is just responsible for the instantiation of the faulty and golden circuits. Figure 3-3 shows the block diagram of the fault simulation process for this scheme. In the figure, the red color components have been implemented in the TCL script. The TCL script is written based on the TCL and built-in simulator commands to perform fault injection and fault simulation shown in Figure 3-1. The grey components, here the instantiation of the faulty and golden circuits, are handled by the HDL testbench.

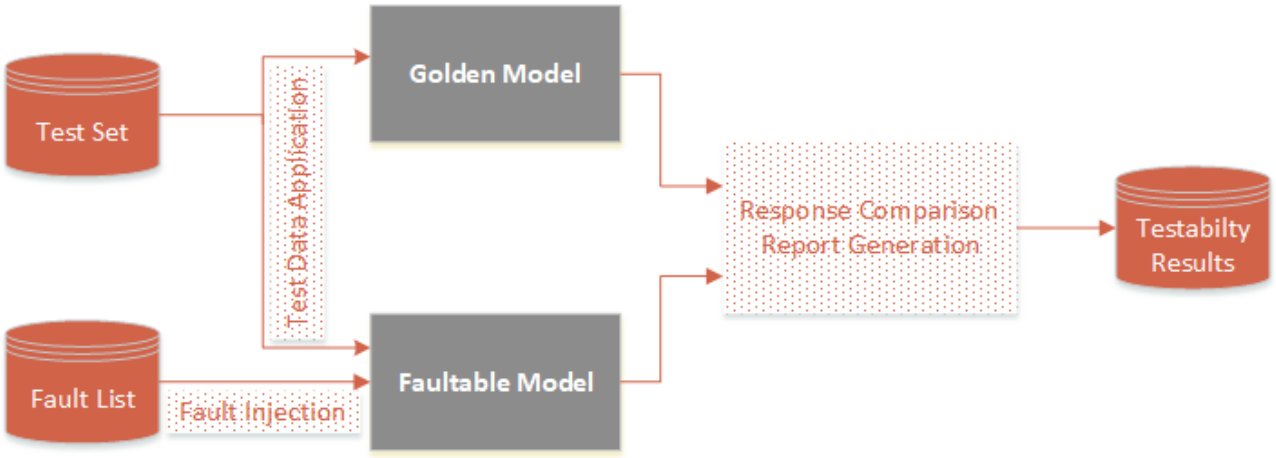

Figure 3-3 All-in-TCL method for fault simulation

Dynamic TCL: In this technique, the HDL testbench instantiates the faultable and golden models, reads the test data file, applies the test patterns, collects test responses and compares them, and finally reports testability results. The TCL script is responsible for reading the fault list, injecting faults, and finally removing faults. In this method, the HDL testbench and the TCL script are working in an interactive manner. Since two nested loops shown in Figure 3-1 have been handled from two different places. Therefore, this method requires more accurate timing. We are not allowed to use relative timesteps (i.e., wait on an arbitrary time slice) in the HDL testbench. This is due to the fact

that the TCL script works based on absolute timesteps to run the simulator. Figure 3-4 shows the block diagram of the fault simulation process for this scheme.

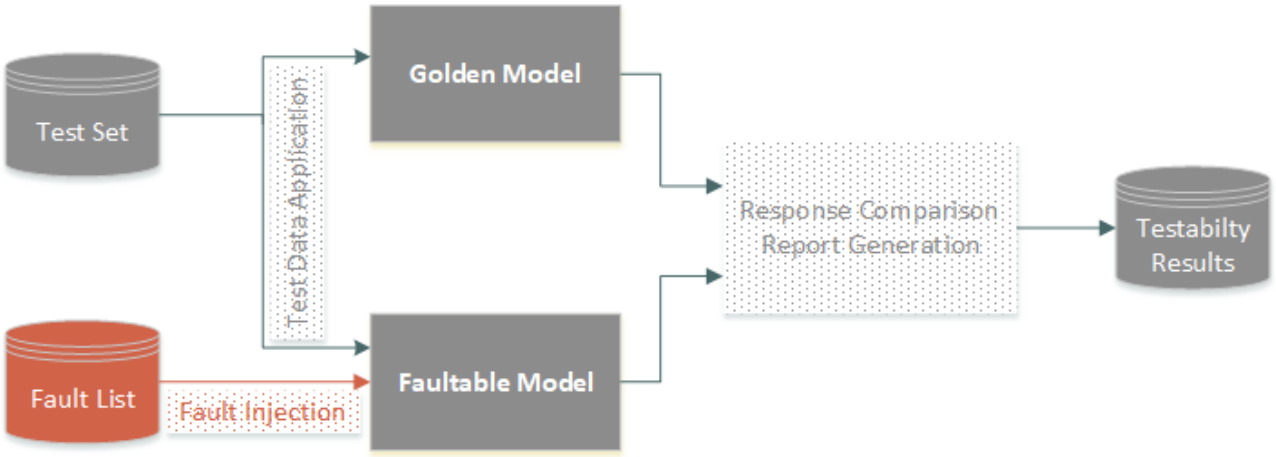

Figure 3-4 Dynamic TCL method for fault simulation

Static TCL: In this method, all things except the fault injection mechanism are implemented in the HDL testbench as shown in Figure 3-5. The TCL script does not have any timing aspect and just statically injects a fault that is determined by the HDL testbench.

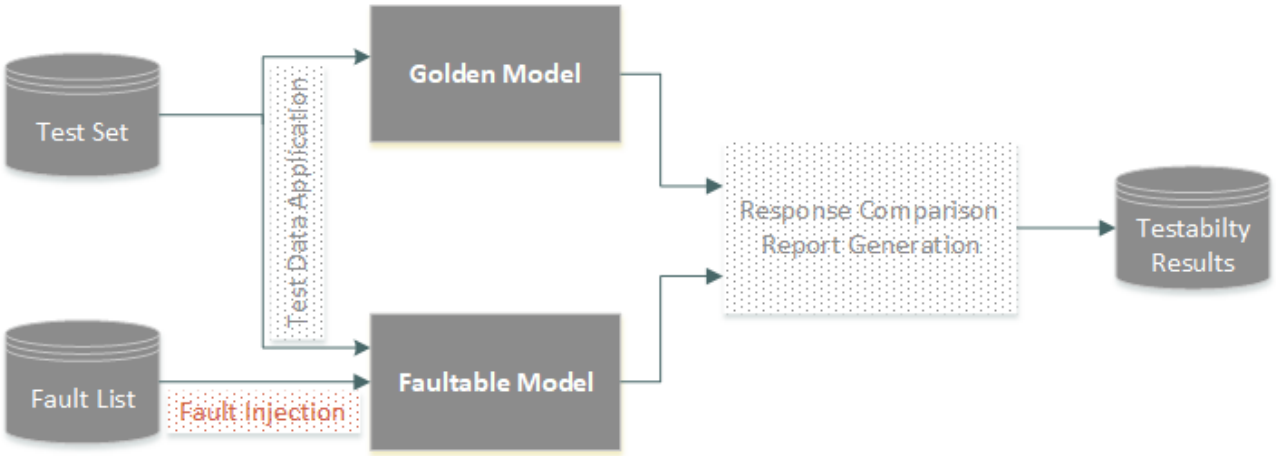

Figure 3-5 Static TCL method for fault simulation

Figure 3-6 shows the flow of the simulation execution for the three methods discussed above. In the all-in-TCL method, all timing and the simulation flow are on the TCL script side as shown in the figure. In the dynamic TCL scheme, both HDL testbench and TCL script have participated in the timing and control of the simulation flow. While the HDL testbench handles all timing and the execution flow in the static TCL method.

For all three methods, fault injection and fault removal are performed in the TCL script using the *force* and *noforce* commands defined by the Modelsim simulator. Using hierarchical naming makes the TCL script capable to access a given signal for forcing and unforcing. To eliminate learning barriers

of TCL coding for the HDL designer and test engineers, we use the static TCL method to test the SAYAC processor in the sections that follow.

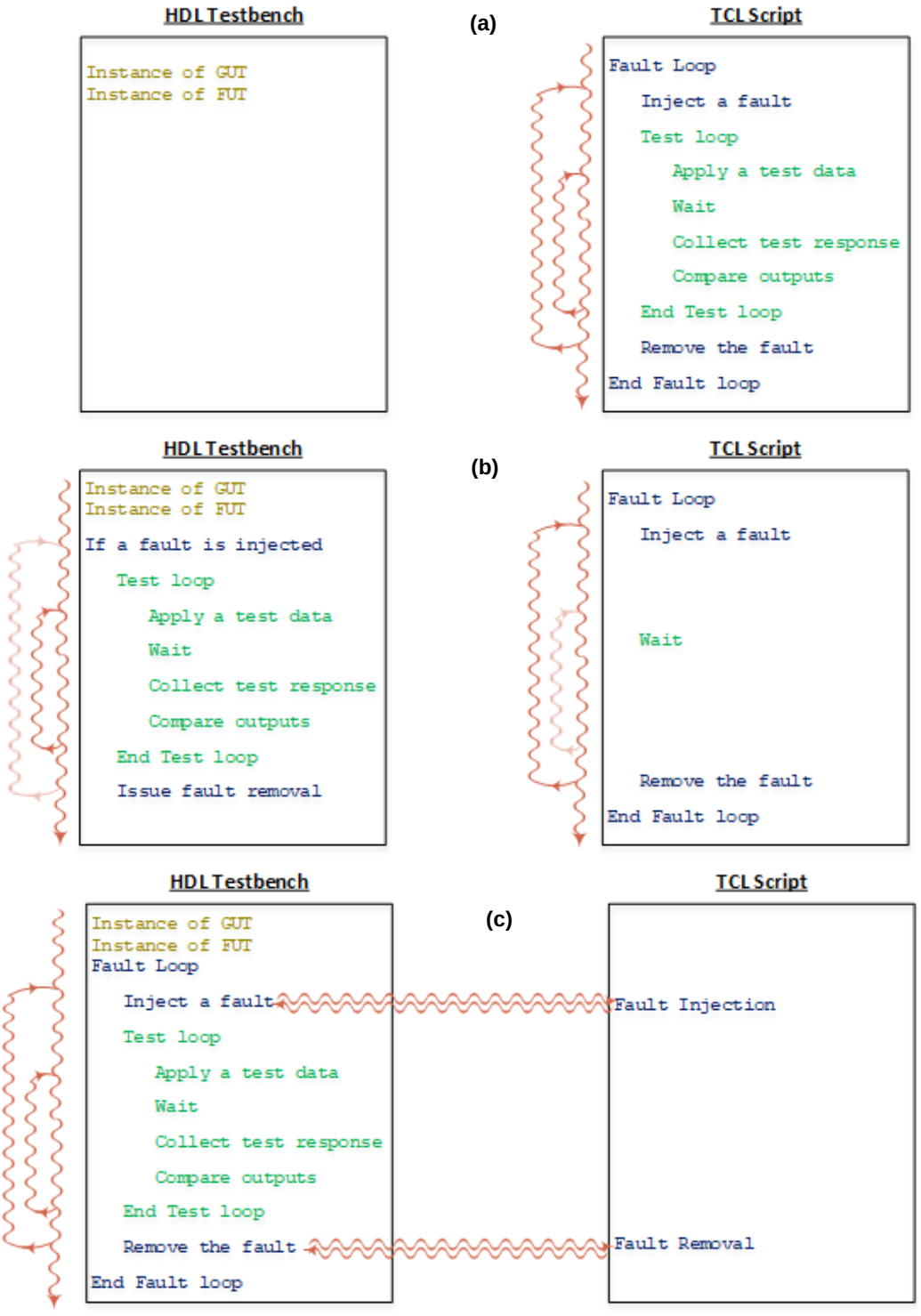

Figure 3-6 Simulation execution flow for (a) All-in-TCL (b) Dynamic TCL and (c) Static TCL

4 Scan DFT Testing

To make a circuit more testable, Design for Test (DFT) techniques are incorporated into the design to make the internal structure of a design more controllable and observable. The work presented in [4] uses the Cadence test tools to assess the impact of DFT in terms of timing, area, and power on a RISC-V processor called preDRAC. The synthesis tool for synthesizing the processor from RTL to Gate-level is Cadence Genus. [5] uses a RISCV processor as a case study to demonstrate the application of the Tessent hierarchical DFT and ATPG methodology. Tessent is a Siemens EDA tool for design augmentation and linked applications that detect, mitigate and eliminate risks throughout the IC lifecycle. Several Tessent DFT products including Tessent ScanPro, Tessent LogicBIST, Tessent MemoryBIST, and Tessent BoundaryScan have been used in this work. The test methods used in this work are very close to what we have done on SAYAC as a RISCV-like processor.

The SAYAC embedded processor is a synchronous design and therefore conventional scan DFT techniques can be used to achieve high test coverage with a low area overhead. However, scan testing requires several design modifications to the SAYAC description which will be explained in Section 4.1. Depending on test time, test pins, area overhead, and other such factors, several variations of scan DFT like full scan, shadow scan, or multiple scan [3] can be used for the SAYAC processor testing. The full scan and multiple scan test architectures, the way they can be applied to the SAYAC processor, and the corresponding simulation results have been discussed in Sections 4.2 and 4.3.

4.1 Make SAYAC Scan-ready

Figure 4-1 shows the original version of the SAYAC system without considering test and incorporation of design for test (DFT) techniques. The DFT schemes discussed in the following sections are built on top of this version of SAYAC.

The application of scan methods requires several design modifications on the SAYAC circuit to be test-ready. Along this line, the following steps have been done.

- ✓ **Bidirectional Pins:** The SAYAC circuit under test must be free of bidirectional input-output (I/O) pins. As shown in Figure 4-2, we split the bidirectional *dataBus* pin into separate I/O pins.
- ✓ **Register File Handling:** The register file has the physical structure of a small SRAM. Therefore, it cannot be tested as a logic considering the logic fault models. To handle the register file, we broke it up from the logic part of SAYAC and defined it as an individual module that is bounded to the SAYAC's logic through its ports. Figure 4-2 shows the SAYAC design modifications for handling the register file (TRF).
- ✓ **Synthesis:** After the two above modifications, the RTL description of SAYAC must be synthesized into a gate-level model with scannable registers. These registers include serial

shift facilities to serially load test data in a scan chain and shift out the response. The synthesis process is performed by our toolchain as discussed in Section 2.

- ✓ **Test Pins:** The post-synthesis netlist of SAYAC is manually modified to include the scan control pins PbarS (P for Parallel mode, S for Serial mode), *n* serial inputs (Si), and *n* serial outputs (So), where *n* is the number of scan chains. When the SAYAC processor is in the normal mode, the PbarS input is 0 to put the scannable registers in the parallel normal mode. In the test mode, the PbarS input is used to shift or capture test data.
- ✓ **Scan Chain:** The PbarS input pin connects to PbarS inputs (shift control) of all scannable flip-flops of SAYAC. Considering *n* groups of the flip-flops of the post-synthesis netlist of SAYAC, the Si inputs connect to the first flip-flop of each group. The output of the first flip-flop goes to the input of the next flip-flop to eventually form a chain of scan flip-flops in each group. The outputs of the last flip-flop of each group drive the So serial outputs. Figure 4-3 shows a scan chain in which all the SAYAC registers are chained together. The additional test pins for this chain can also be seen in the figure. At the end of this step, the logic of SAYAC is ready to be tested by means of the scan. However, the input and output ports of the logic part of SAYAC are connected to the embedded memories and are not accessible for test purposes. To address this issue, several design changes have been done that are discussed in the next step.
- ✓ **Bypassing Embedded Memories:** To put the SAYAC system in the test mode and provide accessibility to the input and output ports of the logic part of SAYAC, several multiplexers and scan flip-flops are added to the top level of the SAYAC logic. In addition, an NbarT (N for Normal mode, T for Test mode) pin is considered to provide switching between normal (when 0) and test (when 1) modes. Figure 4-3 illustrates the scan-ready version of SAYAC considering the additional HW for bypassing the embedded memories.

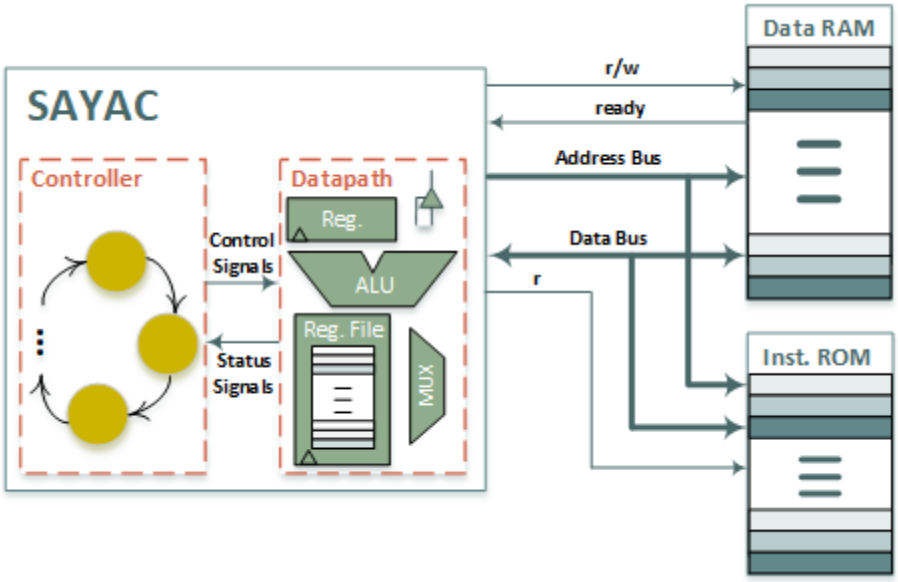

Figure 4-1 Original SAYAC design

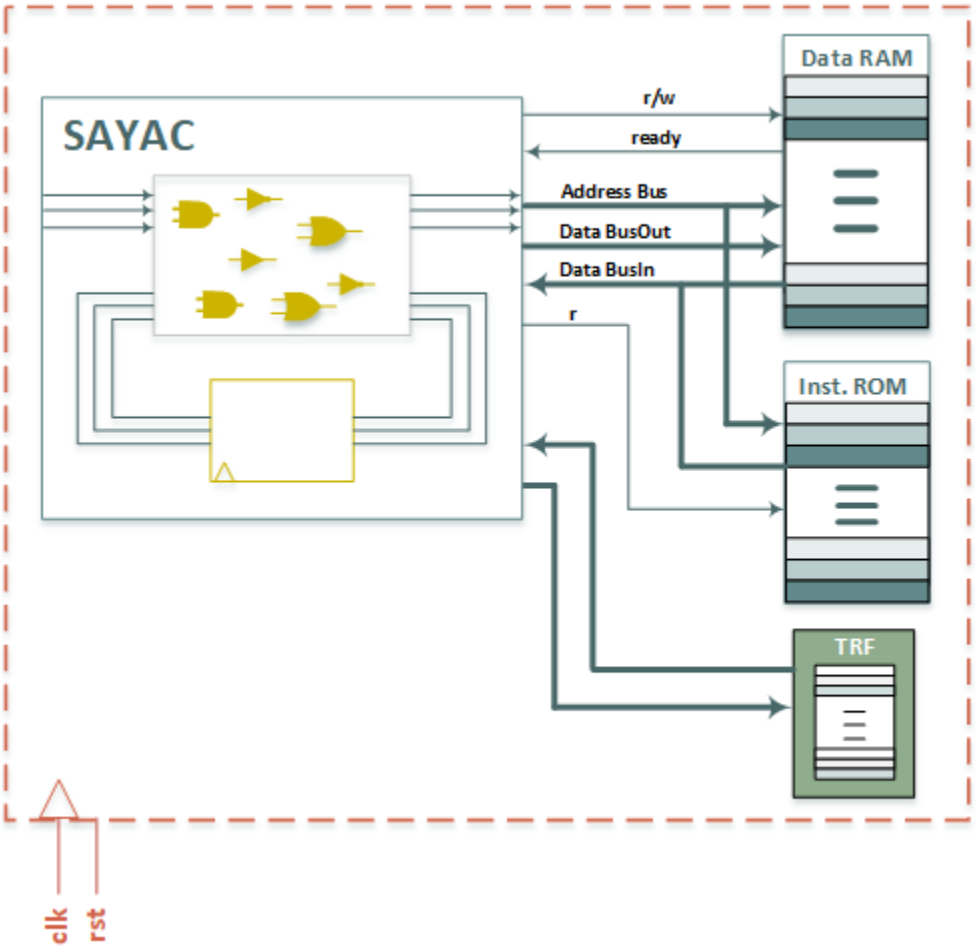

Figure 4-2 Testable SAYAC after synthesis

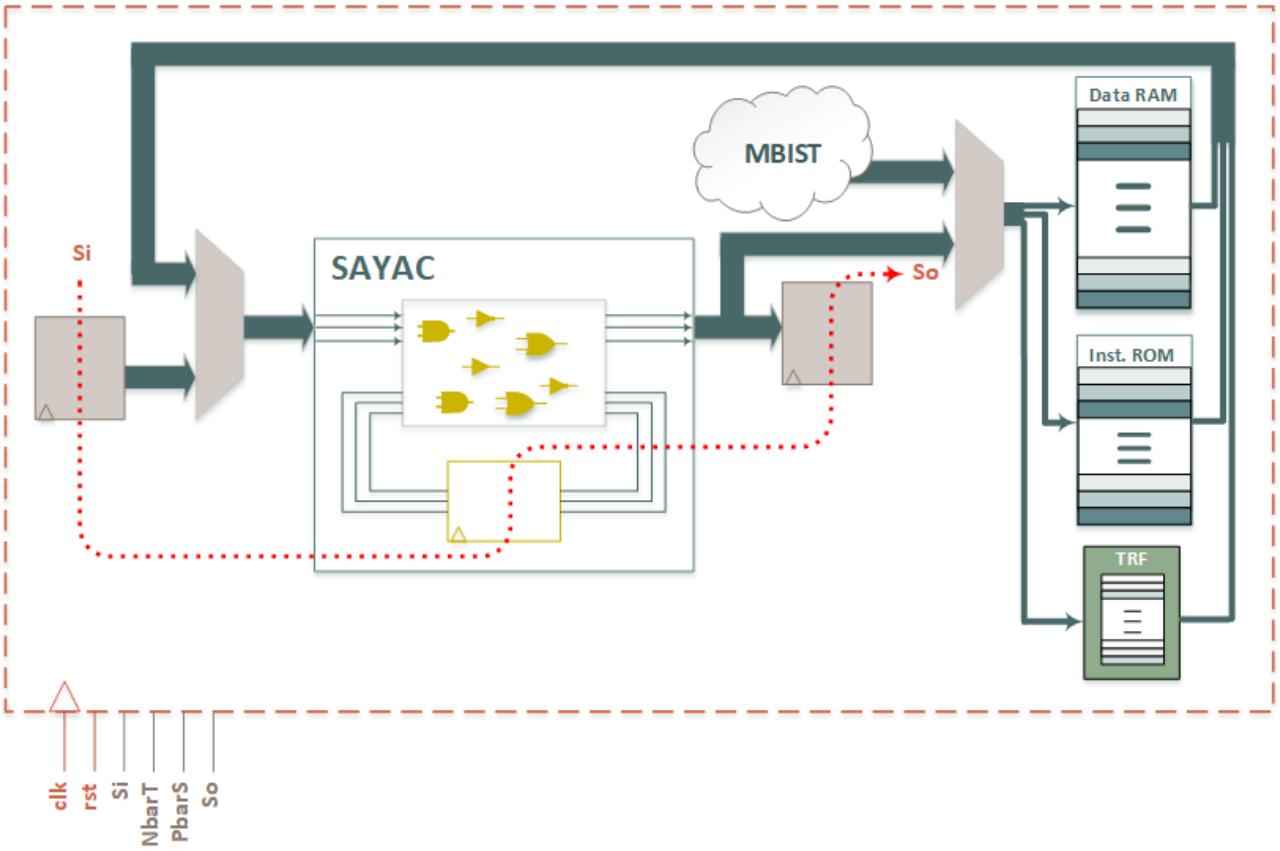

Figure 4-3 Scannable SAYAC

4.2 Full Scan Testing

Full scan scheme chains all the SAYAC registers including internal feedback registers and I/O registers for bypassing embedded memories. The full scan technique uses one serial-in port and one serial-out port. The red dotted line in Figure 4-4 shows the chain that has been formed for full scan testing. As shown, the length of the scan chain is 203 which consists of 90 feedback flip-flops, 49 and 64 scan flip-flops on the inputs and outputs of the SAYAC logic. The 90 internal scan flip-flops are made of a 16-bit instruction register (IR), 16-bit program counter (PC), 16-bit address register (ADR), 32-bit multiplier and divider unit (MDU) registers, and 2-bit controller registers. As shown in Figure 4-3, four pins NbarT, PbarS, Si, and So have been added for full scan testing.

Scan testing requires an external test equipment to apply test patterns to SAYAC and read back their response through test interfaces. To address this requirement, we developed an HDL testbench that is referred to as a virtual tester to imitate the behavior of a test equipment. The virtual tester acts the same as an actual test equipment from the CUT's point of view. As discussed in Section 3, the virtual tester statically interacts with a TCL script that implements our fault injection mechanism.

To develop a scan-based virtual tester, the testable SAYAC module is instantiated in the VHDL testbench. The SAYAC clock that is generated by an assignment statement in the testbench is used

for test purposes. The main task of the virtual tester is implemented by a process statement. This process statement reads predetermined test data from an external file generated by ATALANTA, applies it to SAYAC, collects the output of the CUT, and compares the response with the expected response from the external file generated by ATALANTA. On the other hand, SAYAC is not a faulty circuit, and the virtual tester also has the responsibility of injecting faults obtained by fault collapsing in SAYAC to see if the test set that is provided detects them. Figure 4-5 shows the pseudo-code implemented for full scan testing.

Table 4-1 shows the simulation results of the full scan testing on SAYAC. The rows contain the statistical information on the original SAYAC circuit and the results of full scan testing on the modified version of SAYAC. The columns contain the following information: the areas of the circuits in terms of the number of gates, IO pins, and DFFs, the number of collapsed faults that are injected, the number of the injected faults that are detected, the number of test patterns applied during each test session, the final fault coverage as the percentage of faults that have been detected, and the test time in minutes.

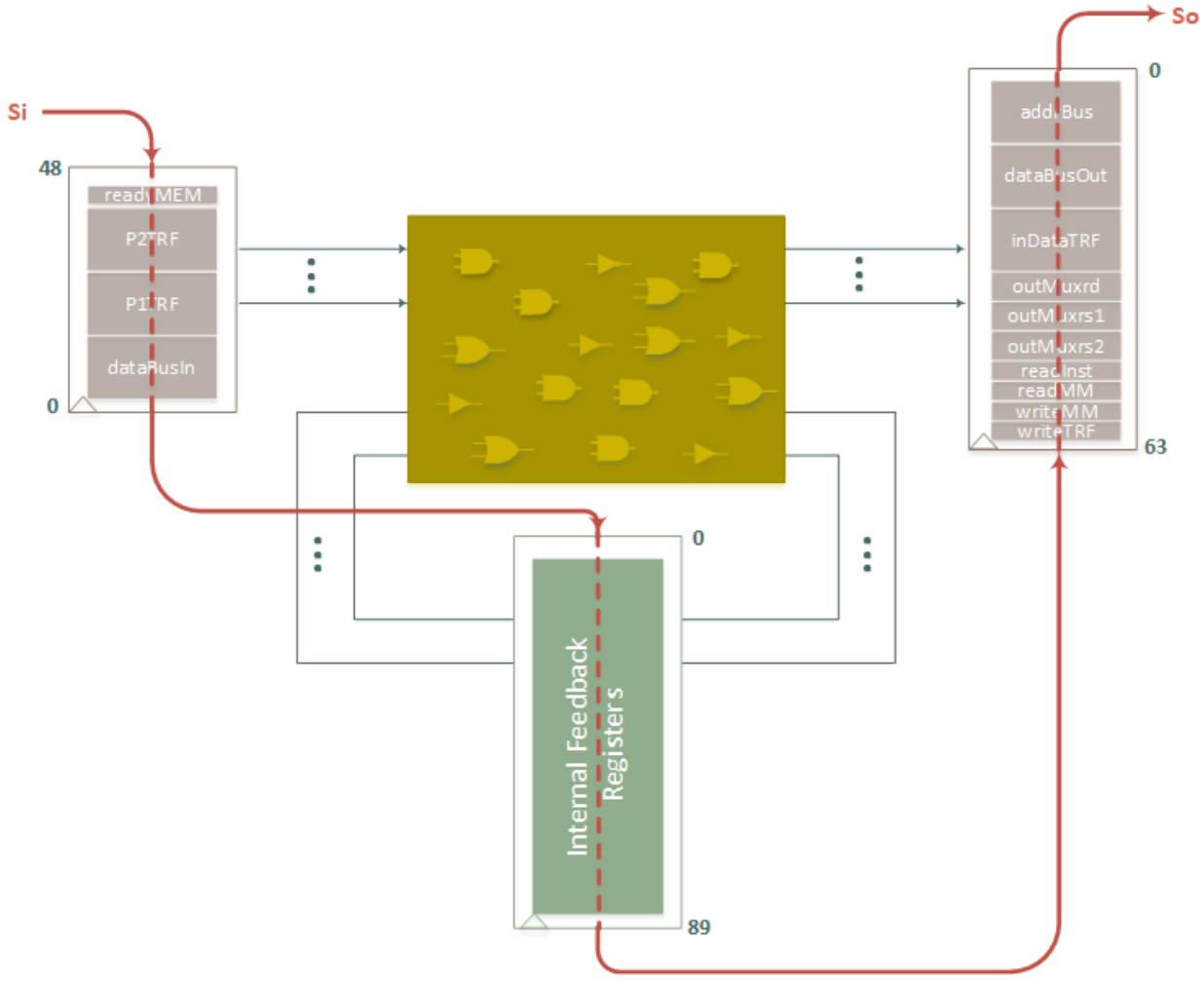

Figure 4-4 Full scan on SAYAC

```
NbarT = 1
Fault loop
       Inject a fault
       Reset CUT
       PbarS = 1
       Shift the first test pattern in
       Test loop
              PbarS = 0 for 1-clock cycle
              PbarS = 1
              Shift out the previous test response while shift in the next test pattern
              Compare the test response with the expected one
       End Test loop
       If the injected fault is not detected
              PbarS = 0 for 1-clock cycle
              PbarS = 1
              Shift out the last test response
              Compare the last test response with the expected one
       End If
       Remove the fault
End Fault loop
```

Figure 4-5 Pseudo-code of virtual tester for full scan method

Table 4-1 Simulation results of full scan testing

SAYAC	Area			#Injected faults	#Detected faults	#Test patterns	#Scan chains	#DFFs per chain	Fault coverage	Test time (min.)
	#Gates	#IO Pins	#D-FFs							
Original	6,305	2	90	-	-	-	-	-	-	-
Full Scan	8,868	6	406	11,760	11,715	398	1	203	99.61%	5,256

4.3 Multiple Scan Testing

The test objective for the SAYAC design is to have high test coverage and a small test time. Along this line to moderate the long scan chain used in full scan testing, multiple parallel scan chains have been formed. In multiple parallel scan chains, all scan registers are controlled by the same set of control signals if the scan registers are put into groups of an equal number of cells. If the number of flip-flops in the scan chains is not the same, then they need independent shift and clock enable control signals. Considering the former scenario, to put the SAYAC registers into groups of an equal number of cells with the same set of control signals, several fictitious scan flip-flops have been added to the input, internal (feedback), and output SAYAC flip-flops. The registers to be scanned are put into 9 groups of 25 registers as shown in Figure 4-6. This test architecture requires fifteen test pins NbarT, PbarS, Si 1 to 6, and So 1 to 7.

Test generation and fault simulation process for a multiple scan design is no different than test generation and fault simulation for the full scan design. The difference is in the application of test

patterns through serial test inputs and the collection of test responses through serial test outputs. Therefore, the pseudo-code of multiple scan testing is the same as what is shown in Figure 4-5.

The simulation results of multiple scan testing on the SAYAC processor including test HW overhead, fault coverage, and test time are reported in Table 4-2. As can be observed from the table, the multiple scan method significantly reduces test time with a small area overhead on extra test pins and fake scan flip-flops.

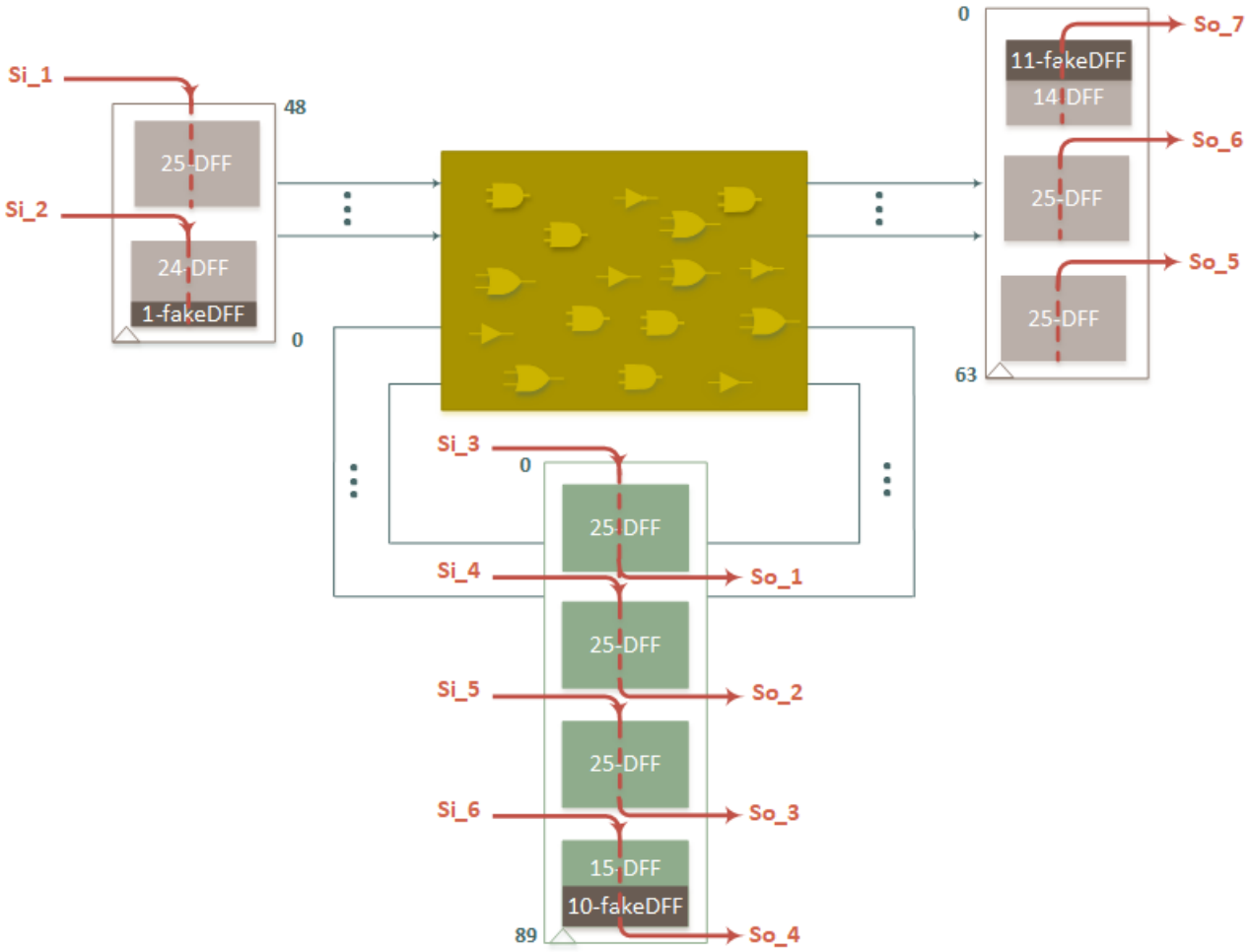

Figure 4-6 Multiple scan on SAYAC

Table 4-2 Simulation results of multiple scan testing

SAYAC	Area			#Injected faults	#Detected faults	#Test patterns	#Scan chains	#DFFs per chain	Fault coverage	Test time (min.)
	#Gates	#IO Pins	#D-FFs							
Original	6,305	2	90	-	-	-	-	-	-	-
Full Scan	8,868	6	406	11,760	11,715	398	1	203	99.61%	5,256
Multiple Scan	9,198	17	450	11,760	11,715	398	9	25	99.61%	846

5 Logic BIST (LBIST)

Self-testing, the ability of a circuit to test itself is a widely adopted Design for Test (DFT) methodology. It does not only contribute to the test cost reduction but also improves the quality of test because it allows a test to be performed at the actual speed of the device, to detect defects that manifest themselves as delay malfunctions. Furthermore, self-testing is a reusable test solution. It can be activated several times throughout the device's life cycle.

The self-testing when the circuit under test is a processor can be categorized into three main schemes: structural logic BIST, functional logic BIST, and Software-based self-testing (SBST). SBST is a nonintrusive method that involves the testing of microprocessors using processor instructions without requiring any design changes or the insertion of any additional hardware structures. In this method, the processor is reused as an existing testing infrastructure to execute embedded software routines that implement test generation, test application, and test response capturing tasks [6, 7]. Unlike software-based self-testing, hardware-based self-testing or built-in self-testing (BIST) techniques employ hardware structures to produce test data, apply it to the circuit under test, collect the output response, and finally verify the output correctness. The operation of a BIST inside a CUT is controlled by a BIST controller.

In the functional self-testing schemes, test opcodes are transferred from scan-in pin to instruction decoder logic while LFSR (linear feedback shift register) generates test pseudo-random operands. BIST controller has the responsibility of generating control signals. Works presented in [8, 9] employ the functional LBIST method for microprocessors. While structural LBIST is performed well on industrial ASICs, it faces many challenges when the circuit under test is a processor [7, 10]. This is because BIST relies on the generation and application of pseudo-random test patterns while microprocessors are random-pattern resistant. However, the feasibility of logic BIST on industrial circuits has been demonstrated in [11-13] in which different approaches like deterministic BIST or weighted random patterns were used to overcome the random-pattern resistance of processor circuits.

In this work, we focus on structural LBIST based on pseudorandom test patterns. In this realm, a BIST in a CUT consists of test pattern generators (TPGs), output response analyzers (ORAs), comparators, and a BIST controller that controls the operation and timing of these units [3]. BIST architectures define various arrangements of such units within a CUT. We will discuss two conventional BIST architectures for self-testing of the SAYAC processor in the following subsection. Memory arrays are not part of the logic BIST and are covered by separate self-test architectures that are discussed in Section 7.

5.1 Make SAYAC BIST-ready

Logic BIST typically builds upon scannable designs. Therefore, a large portion of the design modifications required for LBIST includes the design changes discussed in Section 4.1 to make SAYAC scan-ready. The design changes for BIST include: 1) splitting all bidirectional pins into separate I/O

pins, 2) handling the register file, 3) synthesizing the logic part of SAYAC and replacing the registers with the scannable versions, 4) adding the test input and output pins, 5) chaining the registers together to form the scan chain(s), 6) bypassing embedded memories with BIST structures in the test mode.

Steps 1 to 5 are not different than the steps discussed in Section 4.1. The difference is in Step 6 which replaces the TPGs and ORAs BIST structures with input and output scan flip-flops.

5.2 RTS-based BIST Architecture

The random test socket (RTS) BIST architecture uses a PRPG (Pseudo Random Pattern Generator) and a MISR (Multiple Input Signature Register) for the primary inputs and primary outputs of a CUT. In addition to this, it has an SRSG (Pseudo Random Sequence Generator) that generates pseudo-random scan inputs, and a SISA (Serial Input Signature Analyzer) that generates a signature from scanned outputs from the internal CUT's feedback registers [3].

Figure 5-1 shows the RTS hardware attached to the scan-inserted version of SAYAC. As shown, a 60-bit PRPG is instantiated and connected to the 49-bit SAYAC inputs. The MISR at the output of SAYAC is an 80-bit register, only 64-bit of which are driven by the SAYAC outputs. The SRSG connects to the scan input of SAYAC, and SISA's input is driven by scan output from the CUT. A parameterized BIST controller has been designed to take the number of shifts, the number of test cycles, and the number of test rounds as input. The state diagram of the BIST controller is shown in Figure 5-2. As shown in the figure, the BIST controller includes a sequencer and three counters to keep tracking the number of shifts, the number of test cycles, and the number of test rounds. The sequencer performs a complete test session after issuing the *runLBIST* signal. The BIST controller is responsible for enabling PRPGA and SRSG to apply test data to the CUT and capturing test response by activating MISR and SISA at the proper time. The test process of RTS-based BIST is the same as full scan testing and the controller has the responsibility of putting the feedback registers in the serial mode for shifting test data in and shifting the test response out, and the parallel mode for capturing the test response.

After RTS BIST insertion, the top-level module is instantiated in the VHDL testbench to perform the tasks of BIST evaluation and configuration. To achieve good fault coverage for LBIST of SAYAC, we configure the RTS BIST. In Logic BIST, the HDL testbench has the responsibility of configuring the seed and polynomial of TPGs and ORAs, generating the good response of CUT, and eventually performing fault simulation. Figure 5-3 shows the pseudo-code of the HDL testbench for LBIST.

Table 5-1 shows the RTS configuration and simulation results. As can be observed from the table, fault coverage for the RTS LBIST technique is 68.72% which is lower than the fault coverage of the full scan testing. This is due to the random-pattern-resistant nature of the processor. However, exploring different configurations of seed and polynomial values for TPGs and ORAs can result in a higher fault coverage. On the other hand, the LBIST requires a much higher area and delay overhead than full scan testing due to the need for numerous BIST circuitry, such as the LFSR, the MISR, and the BIST controller.

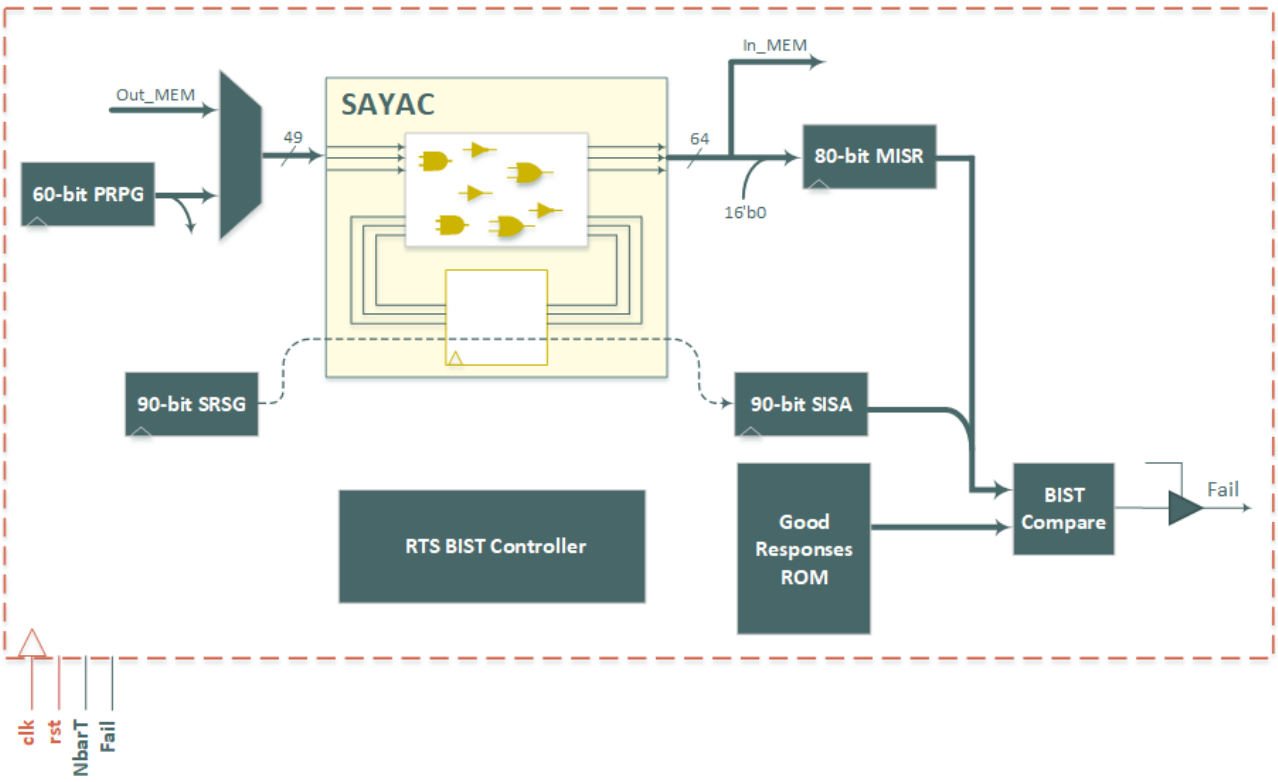

Figure 5-1 SAYAC with RTS BIST

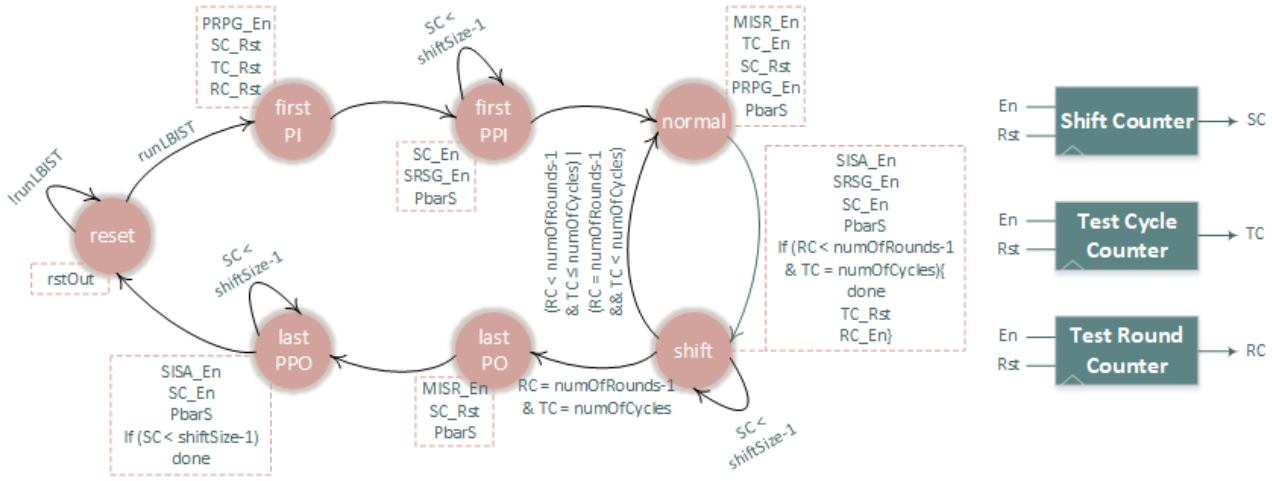

Figure 5-2 RTS BIST Controller

```
NbarT = 1

Configuration loop
       Generate random seed and polynomial values
       Write the results in Configuration File
End Configuration loop

GoodResGen loop
       For each configuration
              Reset CUT
              For each Test Round
                     Wait for the done signal
                     Write good response in Signature File
End GoodResGen loop

For each configuration
       Fault loop
              Inject a fault
              Reset CUT
              For each Test Round
                     Wait for the done signal
                     Compare the test response with the good response
       Remove the fault
End Fault loop
```

Figure 5-3 Pseudo-code of virtual tester for RTS BIST

Table 5-1 RTS BIST configuration and simulation results

BIST Configuration	PI LFSR	PPI LFSR	PO MISR	PPO MISR	#Shift	#Test Cycle	#Test Round
RTS	60	1	80	1	90	100	5

SAYAC	Area			#Injected faults	#Detected faults	#Test patterns	#Scan chains	#DFFs per chain	Fault coverage	Test time (min.)
	#Gates	#IO Pins	#D-FFs							
Original	6,305	2	90	-	-	-	-	-	-	-
Full Scan	8,868	6	406	11,760	11,715	398	1	203	99.61%	5,256
Multiple Scan	9,198	17	450	11,760	11,715	398	9	25	99.61%	846
RTS LBIST	11,484	4	520	11,760	8,082	500	1	203	68.72%	

5.3 STUMPS-based BIST Architecture

The STUMPS (Self-Test Using MISR and Parallel Shift Register Sequence Generator) test architecture was proposed in [14] and is the most widespread logic BIST architecture in the industry. STUMPS moderates the long scan chain used in the RTS BIST by forming multiple parallel scan chains. Test data of scan chains are generated by a PRPG and loaded in parallel into them. The scan chain responses are evaluated by a MISR in a parallel fashion. The STUMPS LBIST just focuses on testing feedback registers and leaves making decisions on how primary input test data are applied and how

primary output test response values are read to the test designer. In this work, we use a PRPG and a MISR for applying test patterns to the primary inputs and collecting test responses from primary outputs of the CUT. This is the way RTS deals with the primary inputs and outputs.

Figure 5-4 shows the STUMPS hardware attached to SAYAC. Like the RTS architecture, a 60-bit PRPG is instantiated and connected to the 49-bit SAYAC primary inputs. The MISR is an 80-bit register, only 64-bit of which are driven by the SAYAC primary outputs. 4 bits out of a 10-bit PRPG connect to the scan chains of SAYAC and a 10-bit MISR collects the test responses of scan chains. A BIST controller similar to RTS (Figure 5-2) is employed to control the self-testing procedure.

The STUMPS test process is similar to RTS and its pseudo-code is shown in Figure 5-3.

The STUMPS test configuration and simulation results are reported in Table 5-2. As can be seen, forming multiple scan chains significantly reduces test time in comparison with RTS. The test time reduction comes at the cost of a small area overhead.

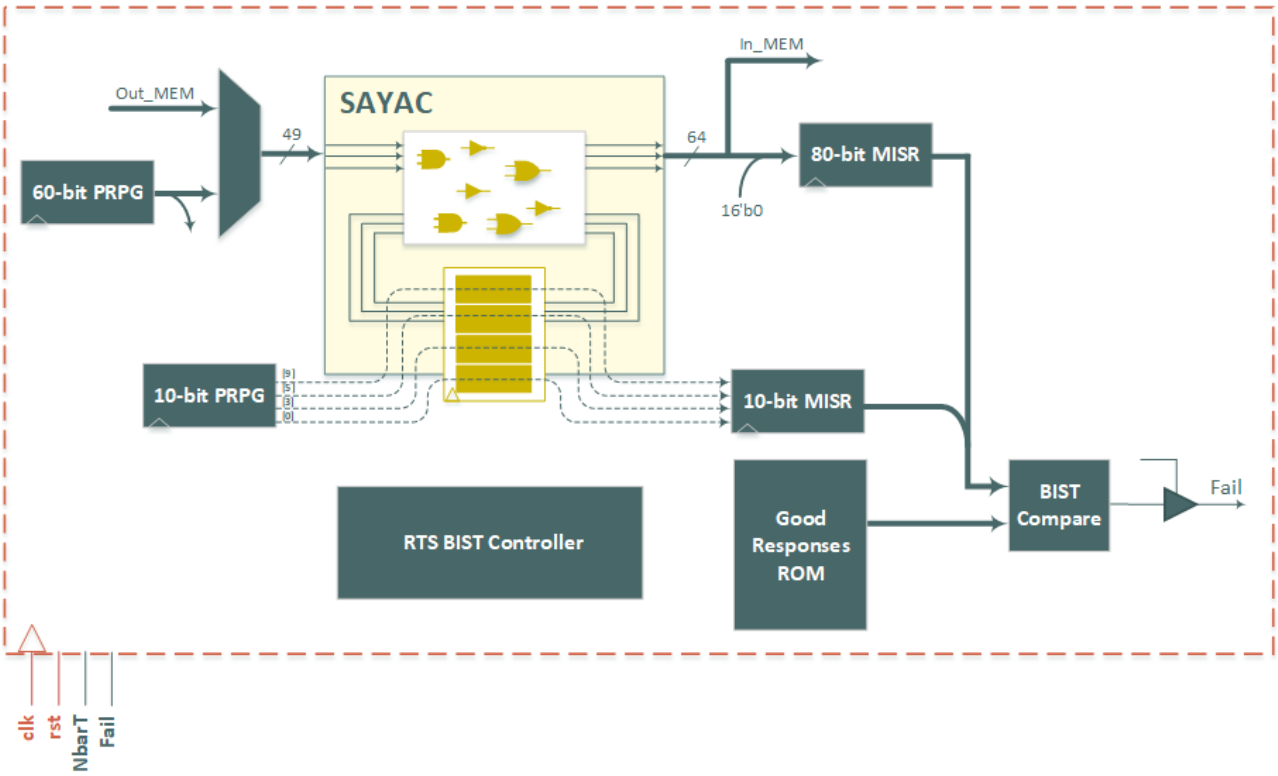

Figure 5-4 SAYAC with STUMPS BIST

Table 5-2 STUMPS BIST configuration and simulation results

BIST Configuration	PI LFSR	PPI LFSR	PO MISR	PPO MISR	#Shift	#Test Cycle	#Test Round
STUMPS	60	10	80	10	25	100	5

SAYAC	Area			#Injected faults	#Detected faults	#Test patterns	#Scan chains	#DFFs per chain	Fault coverage	Test time (min.)
	#Gates	#IO Pins	#D-FFs							
Original	6,305	2	90	-	-	-	-	-	-	-
Full Scan	8,868	6	406	11,760	11,715	398	1	203	99.61%	5,256
Multiple Scan	9,198	17	450	11,760	11,715	398	9	25	99.61%	846
RTS LBIST	11,484	4	520	11,760	8,082	500	1	203	68.72%	
STUMPS LBIST	9,572	4	358	11,760	8,082	500	4	25	68.72%	1,462

6 Boundary-scan IEEE 1149.1 Standard

IEEE 1149.1 standard defines a structural test circuitry that can be included in an integrated circuit to facilitate board-level testing. The test circuitry is incorporated into the integrated circuit to allow test instructions and associated test data to be applied to the circuit under test and then the test response to be read out [15]. Although the main purpose of the boundary scan technique is to provide interconnection testing between integrated circuits assembled onto a printed circuit board, it is also used for many other testing and non-testing purposes. Figure 6-1 shows the boundary scan test architecture based on the IEEE Std. 1149.1^{TM} 2013. As shown in the figure, the IEEE 1149.1 test circuitry requires the following components: Test Access Port (TAP), instruction register, a group of test data registers, TAP controller, decoder, and other units.

Four (or optionally five) pins are added to the normal inputs and outputs of the circuit under test, which form the general-purpose TAP port. The input Test Mode Select (TMS), Test Clock (TCLK) pins control test operations. Test Data Input (TDI) and Test Data Output (TDO) pins are for serially shifting data into and out of the CUT. The optional Test Reset (TRST) pin is used to reset the test circuitry to a known initial state. The instruction register and decoder select the mode of the test operation to be performed and the data register to be placed between TDI and TDO. Several instruction modes are mandated by the standard. It also defines some optional instructions and allows the addition of user-defined instructions. Two mandatory test data registers that are defined by the IEEE 1149.1 standard are the Bypass and Boundary-scan registers. Other registers are defined but are optional like the Device identification register. The test architecture is extensible beyond the mandatory and optional registers through design-specific registers. Boundary-scan testing is controlled by a TAP controller that is a synchronous finite state machine with sixteen states. It manages the operation of the instruction and the test data registers, decoder, and other units by issuing its control signals [16, 17].

To make the SAYAC processor compliant with the IEEE 1149.1 test standard, at first, the boundary scan architecture has been implemented and then specialized and incorporated into SAYAC. In this work, the IEEE 1149.1 test circuitry incorporated into SAYAC is used for both internal logic testing (INTEST) and interconnect testing (EXTEST). The following subsections discuss the test scenarios and test procedures corresponding to internal and external testing for the JTAG-compliant SAYAC processor.

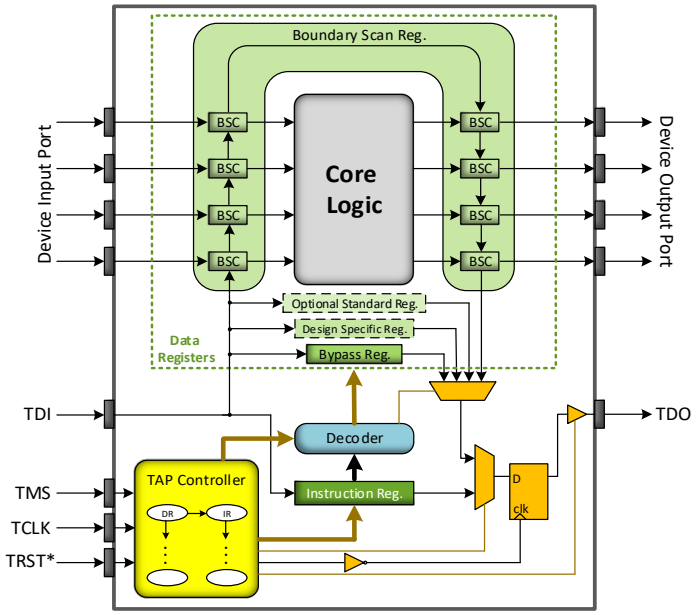

Figure 6-1 Boundary scan IEEE 1149.1 test architecture

6.1 Internal Testing (Logic)

To incorporate the JTAG circuitry into the SAYAC processor, the inputs and outputs of the SAYAC processor are connected to a 113-bit boundary scan register which consists of 49 input boundary scan cells and 64 output boundary scan cells. A 3-bit instruction register is implemented to support the following instructions for the JTAG-compliant SAYAC processor: BYPASS, INTEST, SAMPLE, PRELOAD, EXTEST, and IDCODE. The other hardware components shown in Figure 6-1 are added to our 1149.1 implementation of SAYAC. Figure 6-2 shows the test architecture implemented to test the internal logic of the JTAG-compliant SAYAC. As shown in the figure, the IEEE 1149.1 test circuitry builds upon the scannable version of SAYAC. The IEEE 1149.1 boundary scan registers are connected to the internal scan registers of SAYAC to form a scan chain from TDI to TDO.

After forming the test system shown in Figure 6-2, we develop an HDL testbench as a JTAG-compatible virtual tester. Similar to an actual JTAG tester, the JTAG virtual tester consists of two hardware and software parts. The hardware part is analogous to a JTAG adapter that connects the test system to the test program. This part puts the system in the test situation by instantiating it in an HDL testbench. The software part mimics the test program by providing appropriate signals for the test access port and, therefore, executing the test procedure. Figure 6-3 shows the pseudo-code of the JTAG compatible virtual tester for internal testing. As shown, the CUT is instantiated and connected to the testbench signals to put in test mode. Then, test signals are issued based on a given test plan to test the internal logic of SAYAC. The test clock generation can be performed by a concurrent signal assignment or a process statement. Another process statement shown in this figure is used to implement the TDI and TMS generation, and TDO collection. The TCLK and TMS pins are provided by the virtual tester to drive the TAP controller for controlling the test procedure.

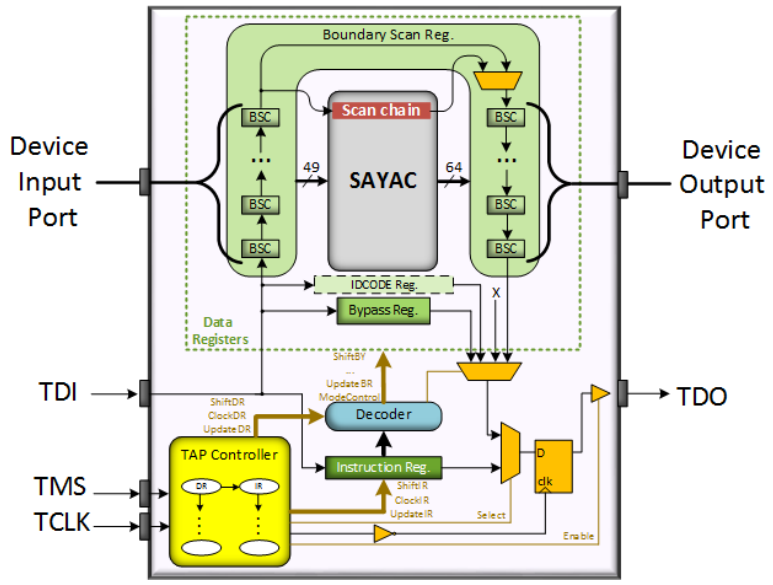

Figure 6-2 Logic testing of SAYAC

Virtual JTAG Compatible Tester:

Given: a JTAG compliant core/chip

// Connect the device-under-test to SW, and put it under SW control
Instantiate the device under test and bind its ports to testbench signals

// Provide a program for JTAG SW to drive the JTAG interfaces
TCLK generation:
Signal Assignment/PROCESS Statement
TDI & TMS generation and TDO collection:
PROCESS Statement
Insert INTEST Instruction in IR
Apply the first test vector to input BSCs
for (all test vectors)
Put DUT in the normal mode of operation
Wait to complete calculations
Shift in the next test pattern while shift out the current test response
Circuit control signals generation:
Signal Assignment/PROCESS Statement

Figure 6-3 Pseudo-code of virtual tester for internal testing

6.2 External Testing (Interconnect)

To perform external testing for the JTAG-compliant SAYAC processor and test its data bus and address bus interconnects, we have considered a fictitious accelerator inside an embedded bus that is referred to as the accelerator bus. The accelerator bus is equipped with 1149.1 hardware and is connected to the JTAG-compliant SAYAC through data and address interconnects. The connection between the SAYAC processor and the accelerator bus forms a two-way path. The path from SAYAC to the accelerator bus consists of 36 interconnect wires that include the 16-bit address bus, the 16-

bit output data bus, and 4-bit control signals. The other path includes 16 interconnect wires for the SAYAC input data bus and one control signal. The scenario for external interconnects testing of the SAYAC processor is shown in Figure 6-4. As shown, as an instance for fault simulation, a stuck-at-1 fault is considered on the most significant bit of the SAYAC input data bus and a stuck-at-1 fault is injected on bit 0 of the SAYAC outputs.

The test architecture shown in Figure 6-4 is instantiated in an HDL testbench as a JTAG-compatible virtual tester. Figure 6-5 shows the pseudo-code of the JTAG-compatible virtual tester for external testing. As shown in the pseudo-code, the JTAG interconnect test requires two phases. For each phase of external testing, after putting the TAP controller in an appropriate instruction state, a specific pattern (opcode) as instruction is shifted into the instruction register through the TDI pin. Subsequently, the virtual tester takes the TAP controller in a proper data state to serially apply a test pattern to the boundary-scan register through the TDI pin. When a complete test vector is applied, the corresponding test response is shifted out through TDO. Bringing test patterns and collecting test responses are handled by reading from and writing to Text IO files.

In the first phase, the PRELOAD and BYPASS instructions are shifted into the JTAG instruction registers of SAYAC and the accelerator bus, respectively. Then, the first test pattern is applied to the output boundary scan cells of the SAYAC processor. This phase is performed to overlap the two shifting operations, shifting the next test pattern in and shifting the current test response out. In the second phase, the EXTEST instruction is shifted into both instruction registers, and the other test patterns are applied for interconnect testing. In this phase, the next test pattern will be shifted in while the results from the current test are shifted out. At the end of the shifting process when the final test response is shifted out, a determinate test data must be shifted in that will leave the circuits in a consistent state.

Table 6-1 shows the simulation results for the interconnects from SAYAC to the accelerator bus. As expected, the test patterns with the least significant bit 0 (test patterns 2, 4, and 5) can detect the injected stack-at 1 fault on bit 0 of SAYAC output interconnects. Simulation results for the interconnect from the accelerator bus to SAYAC are shown in Table 6-2. As shown, test patterns 2 and 4 have detected the stuck-at fault injected on the most significant bit of the interconnect wires.

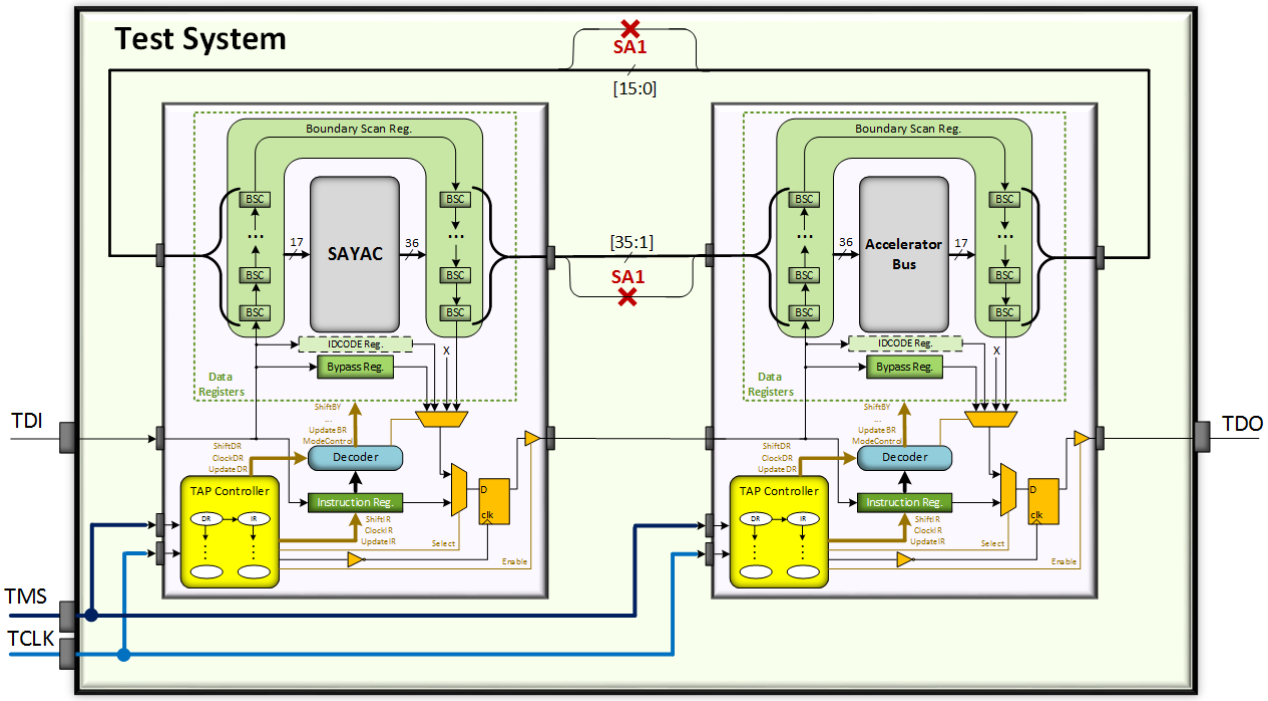

Figure 6-4 Interconnect testing of SAYAC

```
Given: Two JTAG compliant cores/chips

//  HW part analogous to JTAG adaptor
Instantiate the test system

//  SW part analogous to test program
TCLK generation:
   Signal Assignment/PROCESS Statement
TDI & TMS generation and TDO collection:
   PROCESS Statement
        // First phase of external testing
        Insert PRELOAD & BYPASS Instructions in IRs
        Apply the first test vector to interconnect BSCs
        // Second phase of external testing
        Insert EXTEST Instruction in IRs
        for (all test vectors)
             Shift in the next test pattern while shift out the current test response
```

Figure 6-5 Pseudo-code of virtual tester for external testing

Table 6-1 Simulation results for the interconnects from SAYAC to Accelerator

Data from faulty interconnects	Expected Data
11101001101100010011010101110101101 1	11101001101100010011010101110101101 1
00101101110010101010101010101010101 1	00101101110010101010101010101010101 0
10010011010001011110010011010011101 1	10010011010001011110010011010011101 1
11100111101010001011011001110010101 1	11100111101010001011011001110010101 0
00000001111111000011101001110111010 1	00000001111111000011101001110111010 0
11111000111000111011010101010111010 1	11111000111000111011010101010111010 1

Injected SA1

Table 6-2 Simulation results for the interconnects from Accelerator to SAYAC

Data from faulty interconnects	Expected Data
11101001101100010	11101001101100010
10101101110010101	00101101110010101
10010011010001010	10010011010001010
11100111101010000	11100111101010000
10000001111111001	00000001111111001
11111000111000111	11111000111000111

Injected SA1

6.3 External Testing (Memory)

To test the memory unit of SAYAC using the JTAG interface, a bidirectional data bus is utilized. This involves considering three BS cells for input, output, and enable signals. Figure 6-6 shows a block diagram of the test scenario.

To test the memory unit, MATS+ memory test algorithm is employed. This algorithm is represented by the pseudo-code displayed in Figure 6-7. The process involves writing 0s to all cells, reading from the lowest address (expected read value is 0), writing a 1 at this address, and repeating this process until the highest address is reached. Subsequently, it reads from the highest address (expected read value is 1), writes a 0 at this address, and continues until the lowest address is reached.

To implement the MATS+ memory test using JTAG, the EXTEST instruction is used to read and write the memory. The test data, address, and the control signals are shifted into the output BS cells to read/write the memory unit. Then, the output BS cells are updated to transfer the data, address, and the control signals to the memory ports. Afterwards, the control signal *readyMEM* and the output memory are captured through the input BS cells. The captured data is then shifted out while the next test data is shifted in.

The test data, address, and control signals are retained by shifting them into the output BS cells again, as long as the *readyMEM* signal indicates that the memory access is not yet completed.

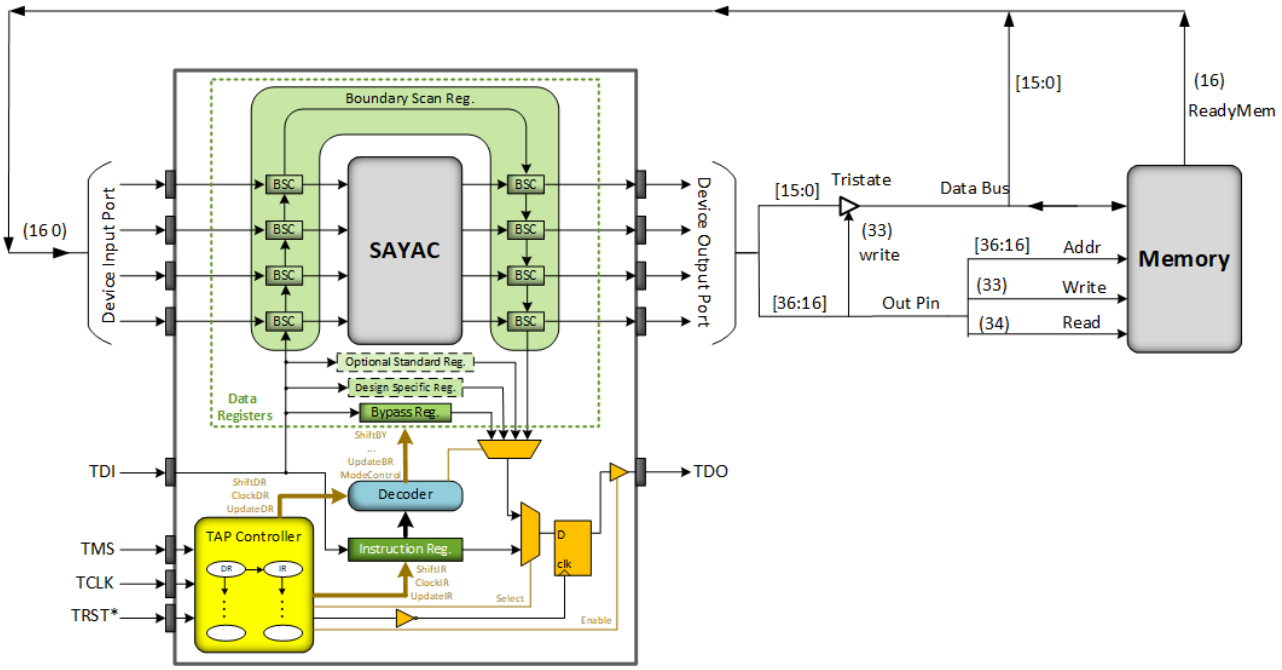

Figure 6-6 Memory testing of SAYAC

```
Initialize memory to 0
for (i=0;i<=(n-1);i++){
       if (m(i)=0) m(i)=1;
       else return fail;}
for (i=(n-1);i>=0;i--){
       if (m(i)=1) m(i)=0;
       else return fail;}
return pass;
```

Figure 6-7 Pseudo-code of MATS+ test algorithm

6.4 Security Extension for IEEE Std 1149.1

This section outlines a security extension scheme for the JTAG interface implemented for SAYAC building upon the concepts introduced in [23]. Along this line, two new instructions, LOCK and UNLOCK, are incorporated into the instruction set. The LOCK instruction secures the device operation by reassigning all instructions (except UNLOCK) to a benign BYPASS instruction until the UNLOCK instruction with valid key code is executed.

The security instructions necessitate the modification of the decoder logic and the incorporation of three registers (Key/Lock shift register, Lock register, Key register), along with a comparator as depicted in Figure 6-8.

The process of locking the TAP controller consists of the following steps: 1. LOCK instruction is entered into Instruction Register via TDI and decoded. 2. Lock Register and Key/Lock Shift Register are enabled. 3. The contents of the Key/Lock Shift Register is cleared at active Capture DR. 4. Lock code is entered into the Key/Lock Shift Register via TDI. 5. Lock code is transferred from the Key/Lock Shift Register to the Lock Register and Key Register is cleared at active Update DR. Comparator compares the contents of Lock Register and Key Register. If the contents are different, the Locked signal fed to the Instruction Decoder is activated. Consequently, the instruction decode logic maps all instructions except UNLOCK to the BYPASS instruction.

To lock the TAP controller, the LOCK instruction is entered into the instruction register via TDI. Subsequently, the *Lock* register and *Key/Lock* shift register are enabled. The contents of the *Key/Lock* shift register are cleared at active Capture DR, and the lock code is entered into the *Key/Lock* shift register. The lock code is then transferred from the *Key/Lock* shift register to the *Lock* register, and the *Key* register is cleared at active Update DR. The comparator then compares the contents of the *Lock* register and *Key* register. If the contents are different, the *Locked* signal fed to the instruction decoder is activated, causing the instruction decode logic to map all instructions except UNLOCK to the BYPASS instruction.

To unlock the TAP controller, the UNLOCK instruction is entered into the instruction register. The *Key* register and *Key/Lock* shift register are enabled, and the contents of the *Key/Lock* shift register are cleared at active Capture DR. The key code is shifted into the *Key/Lock* register and then transferred to the *Key* register at active Update DR. The comparator compares the contents of the *Lock* register and *Key* register. If the contents are equal, it deactivates the *Locked* signal, allowing the next instruction to be executed.

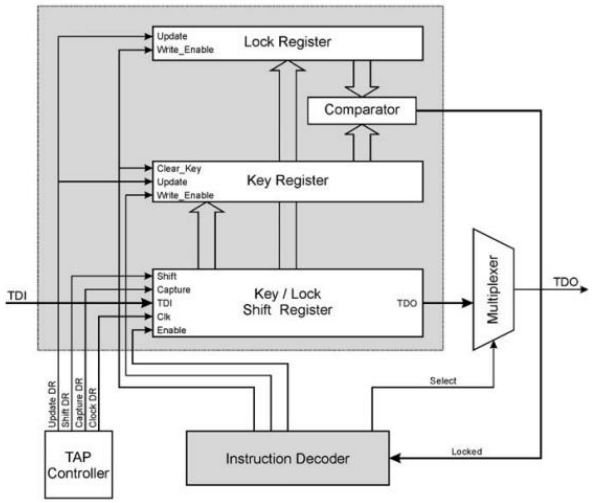

Figure 6-8 The structure of the security extension scheme

7 Memory BIST (MBIST)

SAYAC processor contains three memory blocks, RAM, instruction ROM, and register File. The ROM block holds the instructions of the specific program that is about to run on the SAYAC core. At each clock cycle CPU accesses this memory to fetch the next instruction. The RAM block or data memory contains the necessary data for program execution and also the result of every computation done by the CPU. The other memory block is the register file which is located inside the CPU and contains data and addresses required by the CPU during the execution of an instruction.

All these memory blocks are essential to the proper execution of programs on a core and consequently, even a single defect in any of these memories might lead to CPU malfunction. In order to prevent such outcomes, these memory blocks must be tested frequently. Memory testing can be done using MBIST modules that connect to the memory and verify its correctness. After developing such modules, fault simulation must be done which requires fault models for common defects of the memories. Common fault types for memory are as follows: stuck at '1' or '0', bit flip, coupling, and neighborhood pattern sensitive faults [18-20]. All of these fault types model misbehavior of faulty memory.

The basic mechanism of memory testing is to write the locations with known values and then read all those locations to check if every location was able to behave as expected. This mechanism was employed to test the RAM and TRF block and its algorithm is explained in the next subsection. In the case of a ROM block, writing the memory locations for testing is not possible. The ROM block is only initialized once and is never written into. To test the ROM block, first, good signatures of the ROM program are generated in an offline phase. Then the ROM block is initialized offline with the faulty instructions to handle fault injection. After the offline phase, the ROM is put in test mode. During the testing, signatures of the faulty memory under test are generated using the same signature method and checked against the golden ones to detect the injected faults.

The test mechanisms used for SAYAC memory blocks exhaustively check all the memory locations and therefore any fault type can be detected. To simplify the testing procedure only the stuck at and bit flip fault types are used in this work. In the following subsections, test architectures and testing procedures for each memory block will be discussed.

7.1 RAM Testing

March algorithms are widely used for memory testing. Basically, all of these algorithms write the memory locations and then read from them but they differ in the order of writing known values. The common algorithms and their descriptions are shown in Table 7-1 [3].

To test the RAM block, a simple march algorithm is used in which first a known value is written in every location of the memory then all of the memory locations are read and checked against that known value. In case of a mismatch, a memory fault is detected. In the March-1 algorithm, data written in each location is all zeros except for one bit, therefore the test data can be generated using

a decoder. For addressing the memory locations, a counter is used that will generate address values to test the memory exhaustively. To share the resources, a single counter can be used to generate the addresses and also feed the decoder. Therefore in the writing phase, the value generated by the decoder for example "0000000000000001" is written in all the memory locations and then all those addresses are read from the memory. In the next iteration, another value ("0000000000000010") is written and again read from all the locations. Since the values of the decoder at each writing phase have one bit with the value of '1', it looks like the '1' bit is marching, hence the March-1 algorithm naming. If a memory location is faulty for example stuck to '1' or '0', by writing and reading those locations the fault can be detected.

Table 7-1 March algorithms [3]

Algorithm	Description
MARCH X	{↕(w0);↑(r0,w1);↓(r1,w0);↕(r0)}
MARCH C-	{↕(w0);↑(r0,w1);↑(r1,w0);↓(r0,w1);↓(r1,w0);↕(r0)}
MARCH A	{↕(w0);↑(r0,w1,w0,w1);↑(r1,w0,w1);↓(r1,w0,w1,w0);↓(r0,w1,w0)}
MARCH Y	{↕(w0);↑(r0,w1,r1);↓(r1,w0,r0);↕(r0)}
MARCH B	{↕(w0);↑(r0,w1,r1,w0,r0,w1);↑(r1,w0,w1);↓(r1,w0,w1,w0);↓(r0,w1,w0)}

The test procedure includes hardware and software parts. The hardware part is the MBIST module that generates test data and addresses the memory to write or read from it. The software part is a test program that is implemented by a VHDL testbench. It runs the MBIST and injects faults during the test session and calculates the coverage of the performed test.

The designed datapath of the MBIST module is illustrated in Figure 7-1. As it is shown, the least significant 16 bits of the counter will address the memory locations and the 4 most significant bits would be the input of the decoder module. The decoder module generates 16-bit data which will be the test data written into the memory. The 16th bit of the counter is used as the readWriteBar signal which determines whether data has to be written into the memory or read from it. Note that at first, this bit is 0 hence memory locations are about to be written with test data. After writing in all the locations, the counter output reaches "00000111111111111111". In the next clock cycle, the bit 16 would be set to '1', and memory locations are read. Obviously, after reading all the memory locations this bit will toggle back to '0' and the next writing phase begins.

The MBIST controller is illustrated in Figure 7-2. After a positive edge on the start signal, test mode initiates and a new fault is injected. The fault injection method will be explained in the test program. After injecting a fault, the writing phase begins and if the fault is detected or readMemT is issued then the reading phase begins. Note that in the reading mode after putting the address on the address bus, the corresponding data would be available immediately. If the fault is detected or the

counter has reached its highest value, the counter must be reset and the next fault must be injected. As long as the start signal is held at '1' this cycle repeats.

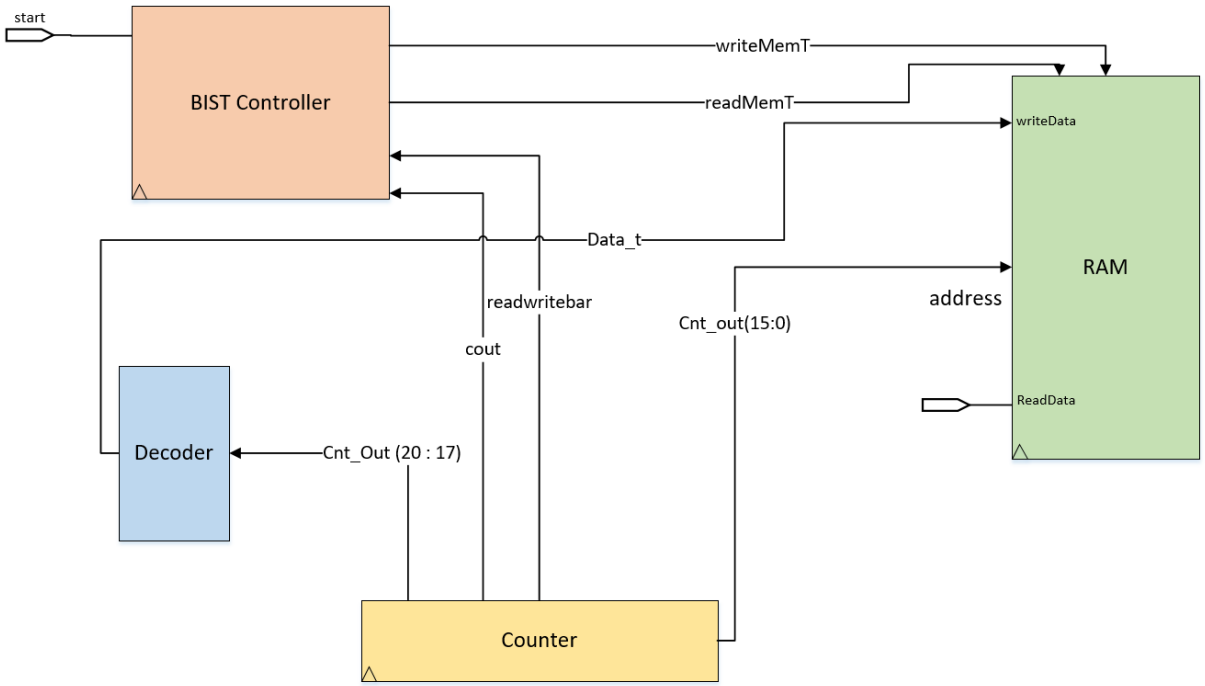

Figure 7-1 RAM MBIST datapath

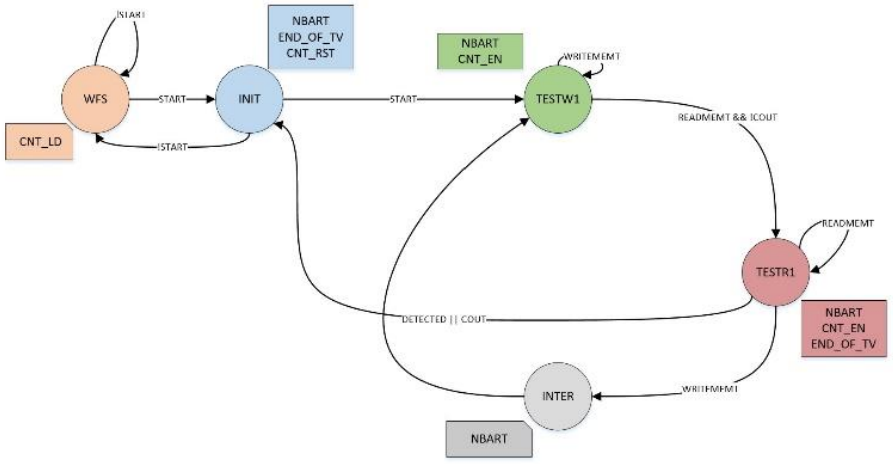

Figure 7-2 RAM MBIST controller

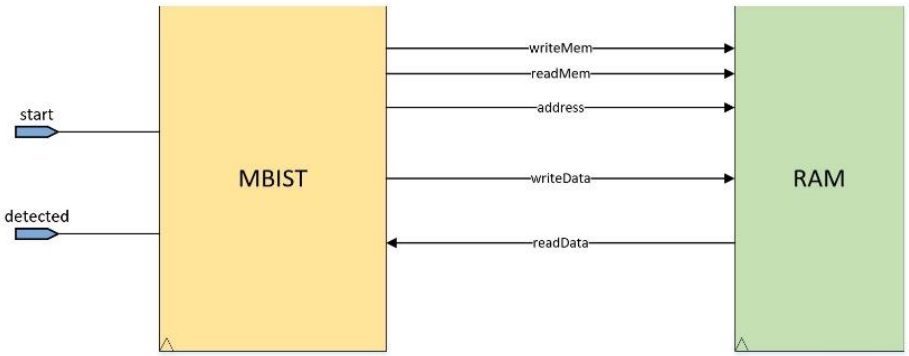

Figure 7-3 RAM MBIST top module

In the testbench the MBIST module connects to the RAM as illustrated in Figure 7-3. The test program is a single process in the TB module. After entering the test mode the start signal of the MBIST module is set. Whenever a new fault must be read from the fault list, the test program will read the fault list and parse the read data into address, bit position, and fault type. These values are stored and used in fault injection.

To inject faults during the fault simulation, at each clock cycle, the test program checks if the MBIST is writing the test data into the memory or not, if so, using a simple function it checks if the address placed on the address bus is supposed to be faulted. In case it is to be faulted, a faulty value would be created to be written into memory instead of the test data, then wireName and stuchAtVal variables are set according to the values that the previous process has specified and the *faultinjection_RAM* signal is set one. This signal triggers a TCL script which forces the writeData signal (that has the test data) to the generated faulty value and instead of the test data a faulty value is written into the memory which means the fault is injected.

The test program keeps counting the detected and injected faults and after injecting all the faults available in the fault list it will calculate the coverage and produces the report file. For a fault list containing 1000 fault vectors, the fault simulation report and synthesis reports are shown in Table 7-2. Note that the testing method employed here is exhaustive and checks every memory location with test vectors that would detect the fault. Therefore, 100 percent coverage is expected. To further analyze the consumed time of fault detection, the fault simulation results for a fault list containing 10 faults are shown in Figure 4. As expected, the simulation results show the March-1 algorithm detects the faulty bits stuck to '1' faster than those stuck to '0'.

Table 7-2 RAM testing results

SAYAC	Area			#Injected faults	#Detected faults	Fault coverage	Test time (sec.)
	#Gates	#IO Pins	#D-FFs				
RAM BIST	312	4	25	1000	1000	100%	2149

```
==================================> @ 0 ms   MBIST MODE is starting ...
faultNum 0 = SA@'0' on Address = 0000001100101000, Bit Position = 15 injected @ 0.000002 ms
Detected by TestVector = 1000000000000000 @ 12.324074 ms
faultNum 1 = SA@'0' on Address = 0100100100100001, Bit Position = 3 injected @ 12.324078 ms
Detected by TestVector = 0000000000001000 @ 15.28257 ms
faultNum 2 = SA@'0' on Address = 0000000011011000, Bit Position = 5 injected @ 15.282574 ms
Detected by TestVector = 0000000000100000 @ 19.739918 ms
faultNum 3 = SA@'0' on Address = 0000010000101111, Bit Position = 12 injected @ 19.739922 ms
Detected by TestVector = 0001000000000000 @ 29.705738 ms
faultNum 4 = SA@'1' on Address = 1100100000110000, Bit Position = 6 injected @ 29.705742 ms
Detected by TestVector = 0000000000000001 @ 30.435034 ms
faultNum 5 = SA@'1' on Address = 1101011001001010, Bit Position = 0 injected @ 30.435038 ms
Detected by TestVector = 0000000000000010 @ 31.965206 ms
faultNum 6 = SA@'0' on Address = 0111011001010101, Bit Position = 10 injected @ 31.96521 ms
Detected by TestVector = 0000010000000000 @ 40.475042 ms
faultNum 7 = SA@'0' on Address = 1001101111100001, Bit Position = 5 injected @ 40.475046 ms
Detected by TestVector = 0000000000100000 @ 45.091146 ms
faultNum 8 = SA@'1' on Address = 0011110001110100, Bit Position = 10 injected @ 45.09115 ms
Detected by TestVector = 0000000000000001 @ 45.677354 ms
faultNum 9 = SA@'0' on Address = 1010000011010111, Bit Position = 13 injected @ 45.677358 ms
Detected by TestVector = 0010000000000000 @ 56.590026 ms
**********************************************************************
**********************************************************************
**********************************************************************
numOfDetected: 10
numOfFaults: 10
Coverage = 1.000000e+02 %
```

Figure 7-4 Report file for RAM testing

7.2 Register File Testing

The register file is a memory block with 16 addressable locations. The testing method used for TRF is the same as the one covered in the RAM MBIST section. Due to the register file size, the counter used for TRF testing is smaller. The decoder, MBIST controller, and other elements used in the TRF MBIST are the same as RAM. The connection between the MBIST module and TRF is illustrated in Figure 7-5. Note that the writeTRF signal works as the writeReadBar signal and is set by the 5th bit of the counter output. Read and write addresses (rs1 and rd) are both connected to the 4 least significant bits of the counter and test data is generated by a decoder and available on the TRFData_in signal.

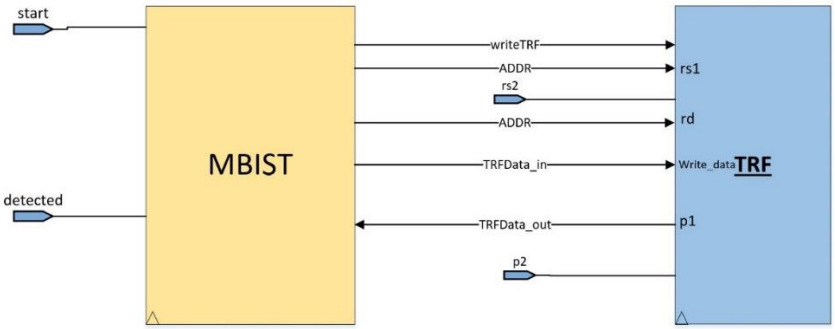

Figure 7-5 TRF MBIST top view

Like RAM testing, the TRF and MBIST modules are instantiated in the testbench. The HDL testbench alongside a TCL script handles the fault simulation. For 100 faults, the fault simulation and synthesis reports are shown in Table 7-3. It is important to note that the addresses 0x00 and 0xFF of the register file have not been tested by the discussed method. This is because we cannot write access them due to their application.

Table 7-3 TRF testing results

SAYAC	Area			#Injected faults	#Detected faults	Fault coverage	Test time (sec.)
	#Gates	#IO Pins	#D-FFs				
TRF BIST	238	4	13	100	100	100%	19

7.3 Instruction ROM testing

The instruction ROM memory is programmed once. Therefore, it cannot be tested by the MBIST architecture discussed above. The testing method employed for ROM testing is as follows.

At an offline phase, the ROM instructions are compressed into 16-bit signatures using the signature generator module. Signature generation could be done per program or by memory segmentation, the latter is employed here. In this method, the whole memory is devised into fixed-length windows and for each segment, one signature is generated. In this work, the golden signatures are stored in a text file. The signature generator has a simple structure, its datapath consists of a counter for addressing and a MISR to generate the signatures. The datapath and controller of the signature generator module are illustrated in Figures 7-6 and 7-7, respectively. Note that the signature generation is not part of the MBIST hardware and is done offline. To test a ROM, the golden signature file must be already provided to be used in the testing procedure.

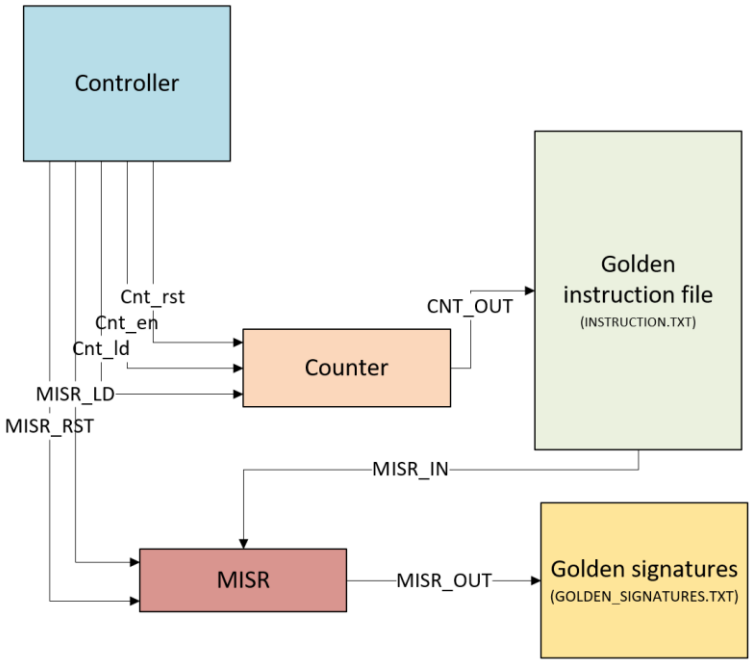

Figure 7-6 Signature generator datapath

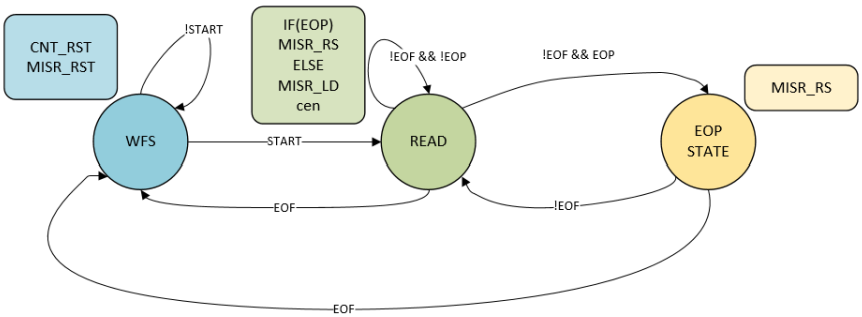

Figure 7-7 Signature generator controller

In test mode, the actual ROM under test would be initialized with a faulty text file. Generation of the faulty instruction file could be easily done by a python program. The MBIST module that tests the faulty ROM has the same datapath as the signature generator module except for the instruction file in which the actual ROM would place. The datapath and controller of ROM MBIST are illustrated in Figures 7-8 and 7-9, respectively.

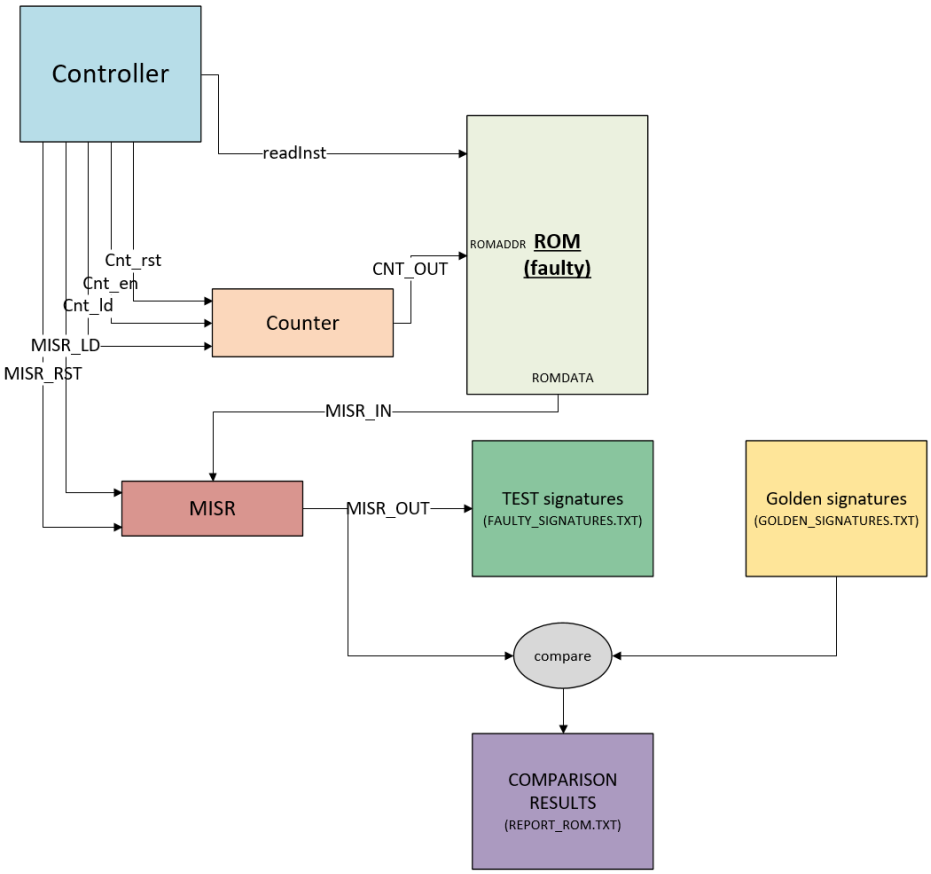

Figure 7-8 ROM MBIST datapath

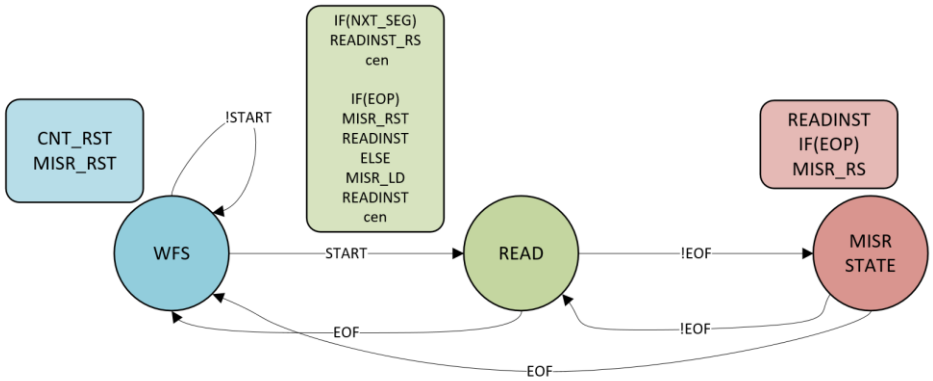

Figure 7-9 ROM MBIST controller

In the HDL test bench, the MBIST and ROM modules are instantiated. In the test program whenever the test procedure reaches the end of a segment, the generated signature is checked against the golden one. In case of a mismatch, the faulty segment is reported. The test program reads the golden signature file which has been generated by the signature generator module, before fault simulation.

Note that a fault in a memory segment results in the MISR output corruption. Therefore, the faults after that cannot be detected. By using a MISR for consecutive instructions, the number of signatures is reduced. However, just the number of faulty segments can be reported and not the number of faults in each segment. Testing results for the instruction ROM are shown in Table 7-4.

Table 7-4 ROM testing results

SAYAC	Area			#Faulty segments	#Detected segments	Fault coverage	Test time (sec.)
	#Gates	#IO Pins	#D-FFs				
ROM BIST	688	4	35	2	2	100%	3

8 Test Compression

Test data compression and decompression for SAYAC have been performed using two code-based schemes: Run_length and Golomb.

8.1 Golomb Method

Golomb Compression Code: Golomb codes map variable-length runs of 0s in difference vectors to variable-length codewords. Using Golomb code, each codeword has two parts: a group prefix and a tail. To encode data, each set of data should be partitioned into groups of size *m*. The runs of 0s in T_{diff} are mapped to a group of size *m*. The number of such groups is determined by the length of the longest runs of 0s in T_{diff}. An example of Golomb coding of four test vectors with *m* equal to 4 is shown in Figure 8-1.

A C/C++ code is developed for compressing the SAYAC test set using the Golomb method. This program follows the exact compression and arrangement steps shown in Figure 8-1. The compression flow is illustrated in Figure 8-2.

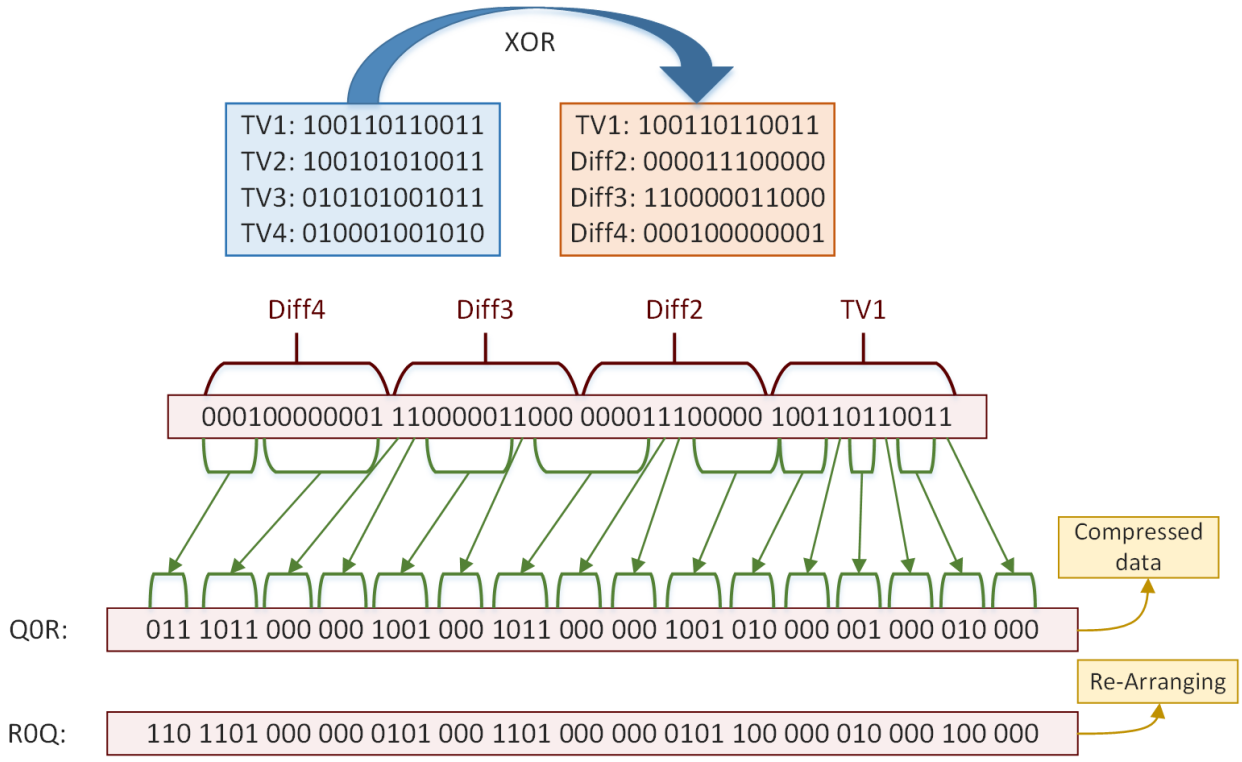

Figure 8-1 Golomb compression code

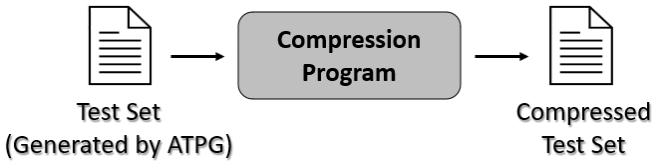

Figure 8-2 Compression flow

Golomb Decompression Hardware: The re-arranged compressed data is fed into the decompressor hardware. Most decompressor hardware consists of three major blocks, Synchronizer1, Decoder, and Syncronizer2. A simple representation of these blocks is illustrated in Figure 8-3. Synchronizer1 is responsible for providing the compressed data for the decoder on demand, then the decoder decompresses the data and produces the XORed differences between consequent vectors, and finally, Synchronizer2 produces the test vector using a CSR (Cyclic Scan Register). This decompressed test vector is used for testing the SAYAC processor through the scan method.

As mentioned above, Synchronizer1 is responsible for providing the compressed data for the Golomb decoder. The handshaking between these two modules is through a *ready* signal. Synchronizer1 reads the compressed test set from a text file and then whenever *ready* is '1', one bit is shifted out on *d_in*. On the other end, the decoder is connected to Synchronizer2, in which a CSR is used to generate the test vector from the difference vector. The handshaking between the decoder and Synchronizer2 is through a *valid* signal which indicates when to read the *diff_out* signal. The RTL design of the decoder to decompress Golomb code is presented in the following paragraph.

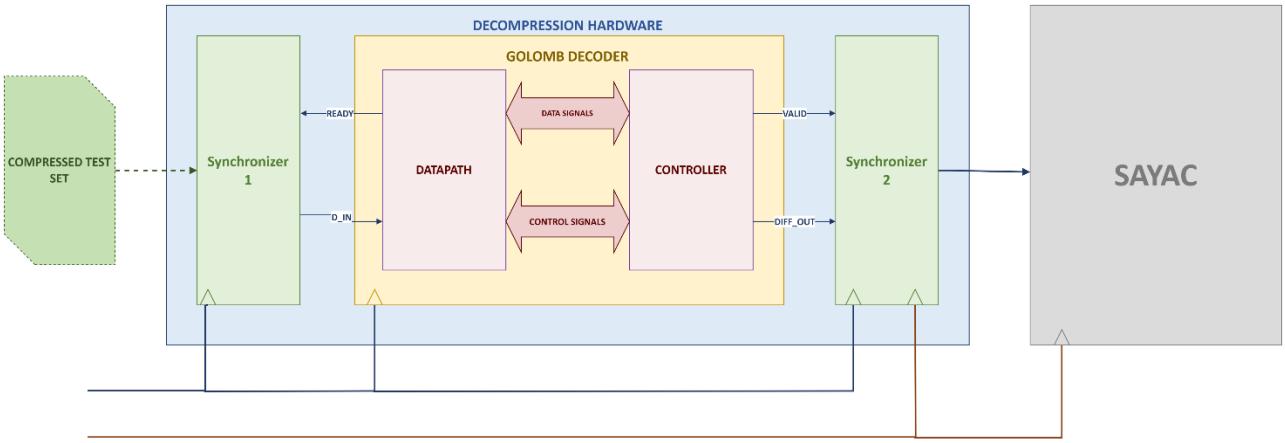

Figure 8-3 Overview of decompression hardware

The Golomb decoder receives bits of compressed data. The value of *m* must be set for the decoder as an input. Here, it is assumed that *m* is set to 4, and therefore its logarithm in 2 basis is equal to 2. This means that there are always 2 bits of *R* for any coded data. To decompress the data, the decoder must distinguish between the values of *Q* and *R* and store them in separate registers. To do so, the *Q_REG* register and *R_SHR* shift register are placed in the datapath. The first incoming bits are *Q*

values which are all ones. Therefore, the Golomb counter would count the number of incoming consecutive ones and then as soon as receiving a '0' on the *d_in* input, the number of counted ones is loaded in the *Q_REG*. After receiving a zero, the following two bits are the value of *R*. These two bits are shifted into the *R_SHR* while the Golomb counter is counting up to *log(m)*. While shifting in the *R* value, *cnt_out* must be compared to *log(m*), so *R_CNT_SEL* must be set to '1'.

Now, having all three values of *m, Q,* and *R*, the number of consecutive zeroes in the original data can be calculated (number of zeroes = *m*q* + *r*). By using a multiplier and an adder in the datapath, the *NUM_OF_ZEROES* is attained. Having the number of consecutive zeroes, the decoder must produce the original data by putting '0' and '1' on the *diff_out* output. To do so, in one clock cycle '1' is put on the *diff_*out then the Golomb counter will count up to the number of zeroes, and in each clock cycle a '0' is set on the *diff_out* while the *valid* signal is set. The datapath design of the Golomb decoder is illustrated in Figure 8-4.

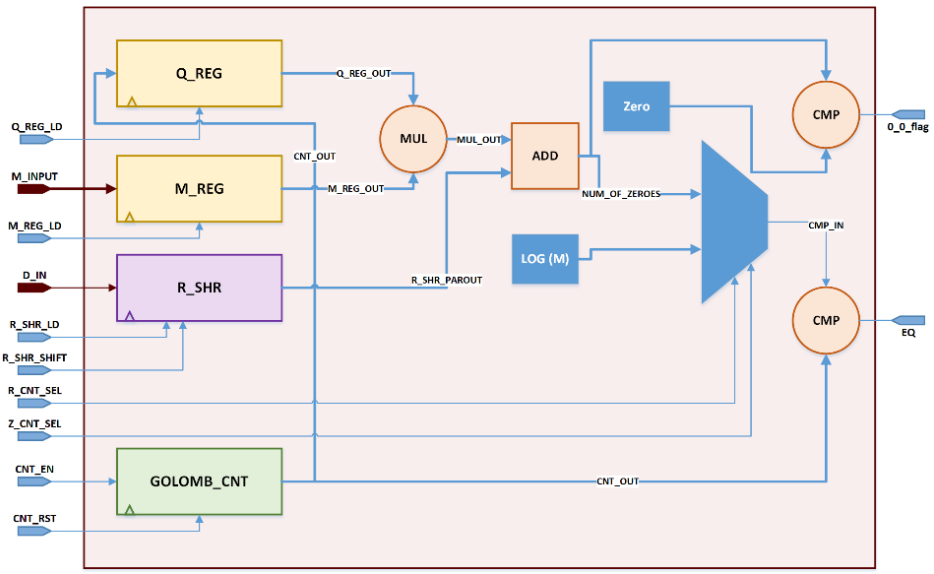

Figure 8-4 Datapath of Golomb decoder

The controller of the Golomb decoder is shown in Fig 8-5. In the controller, state_2 and state_3 are responsible for counting up to the value of *Q*. States 5 and 6 would shift in the *R* value. When the *R* value is shifted in (indicated by *EQ* signal), states 7 and 8 would generate the *diff_out* signal, in which state 7 would set a '1' on the *diff_out*. State_8 would put '0' on *diff_out* while the Golomb counter counts up to the NUM_OF_ZEROES value. As shown in Figure 8-5, the *valid* output is set to '1' in both states 7 and 8.

The difference vector produced by the decoder is fed to a CSR in the Syncronizer2 module. The architecture of a CSR is shown in Figure 8-6. The serial output of the CSR contains the original test vector that is used for testing the SAYAC processor.

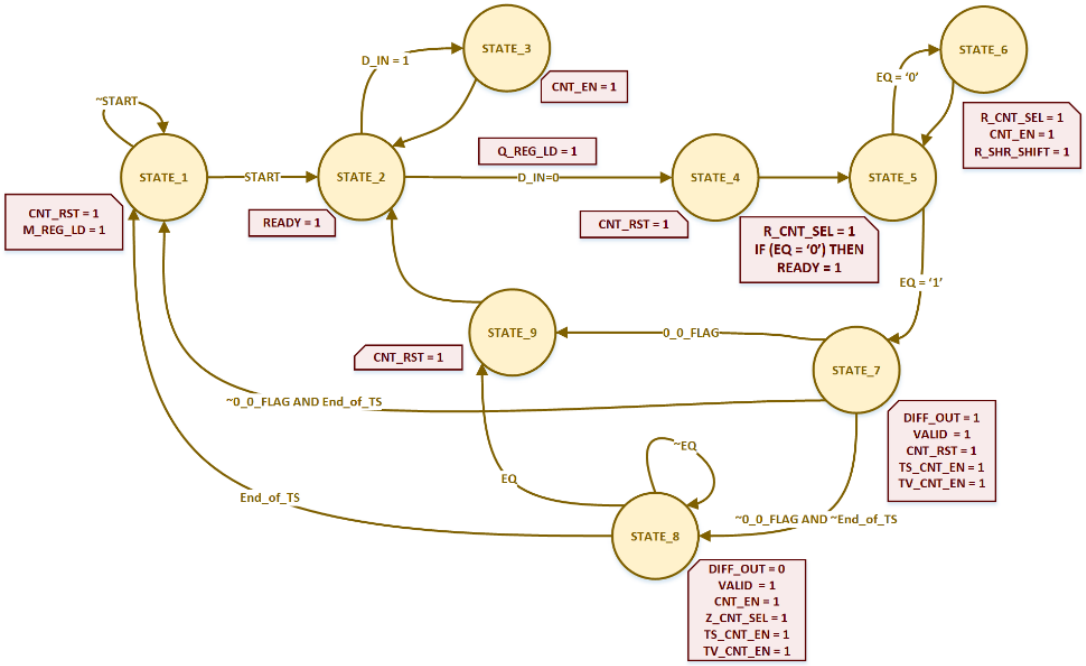

Figure 8-5 Controller of Golomb decoder

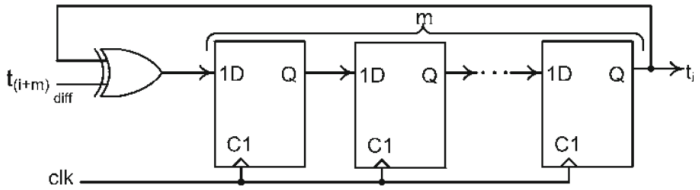

Figure 8-6 Cyclic Scan Register (CSR) architecture

Simulation Results: In order to test the Golomb decompressor, it is connected to the SAYAC processor to provide the test vectors during the fault simulation as shown in Figure 8-3. A snippet of the fault simulation log is shown in Figure 8-7, which indicates the correct detection of injected fault by the decompressed test vectors.

```
==================================> @ 12.5 ns    SCAN TEST MODE is starting ...
faultNum 1 = SA@0 on VirtualTester/FUT/SAYAC_Logic/nor_n_92/in1(0)     , injected @ 12.5 ns, detected by testVector 1 = 10001011101000111101000111010110000111111000101011101
faultNum 2 = SA@0 on VirtualTester/FUT/SAYAC_Logic/nor_n_92/in1(1)     , injected @ 14747 ns, detected by testVector 1 = 10001011101000111101000111010110000111111000101011110
faultNum 3 = SA@0 on VirtualTester/FUT/SAYAC_Logic/nor_n_94/in1(0)     , injected @ 29477 ns, detected by testVector 6 = 10001010000000111001100010110011011110111111110110010
faultNum 4 = SA@0 on VirtualTester/FUT/SAYAC_Logic/nor_n_94/in1(1)     , injected @ 79122 ns, detected by testVector 1 = 10001011101000111101000111010110000111111000101011110
faultNum 5 = SA@1 on VirtualTester/FUT/SAYAC_Logic/nand_n_95/in1(0)    , injected @ 93852 ns, detected by testVector 1 = 10001011101000111101000111010110000111111000101011110
```

Figure 8-7 Golomb simulation results

The C/C++ program developed for Golomb compression is used to compress the SAYAC test set and the result is used in fault simulation. The difference in test set size after the compression is shown in Table 8-1. It seems that compressing the test set using the Golomb method has increased the size of the test set, thus defying the purpose of compression. However, it must be considered that such methods of compression have proven effective in much larger test sets in which test vectors have more overlapping sections. The more similarity between test vectors results in having consecutive

runs of zeroes after XORing the vectors. In this case, the test set for the SAYAC processor is produced by the ATALANTA program and its test vectors do not have much similarity between themselves. This lack of similarity resulted in a larger test set after the compression.

Table 8-1 Golomb compression results

Compression method	Test set original size	Compressed test set size
Golomb	114 KB	159 KB

8.2 Run-Length Method

Run-Length Compression Code: The conventional Run-length code is a variable to fixed coding scheme. Runs of data are sequences in which the same data value is repeated. In this method, a test pattern is partitioned into variable length symbols that consist of runs of consecutive 0s or 1s. Each symbol is encoded as the length of runs of consecutive 0s or 1s. In this work, the runs of consecutive 0s are considered. This method of compression is illustrated in Figure 8-8 through a simple example. In this example, the length of code word (*lcw*) is assumed to be 3 which means the maximum run of zeroes can be 7.

A C/C++ code is developed for compressing the SAYAC test set using the Run-length method. This program follows the exact compression and arrangement steps shown in Figure 8-8.

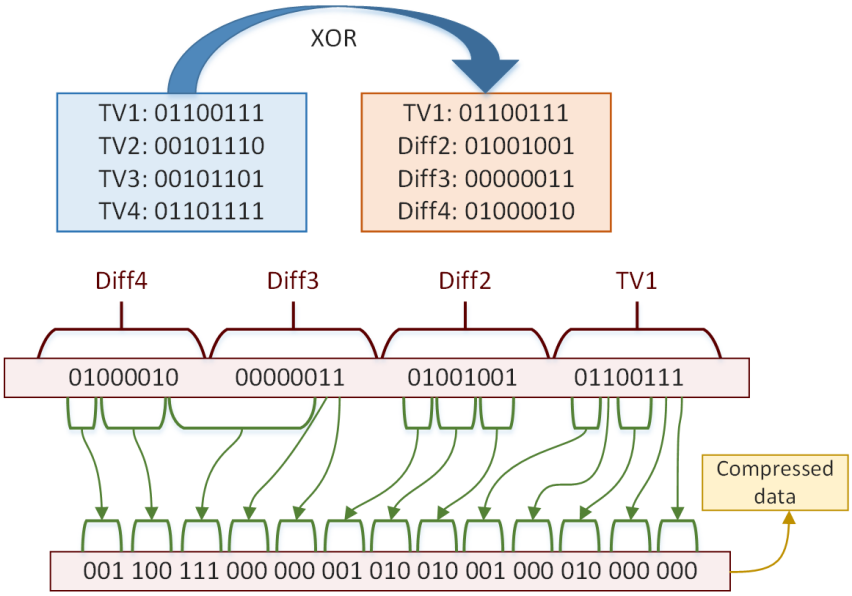

Figure 8-8 Run-Length compression code

Run-Length Decompression Hardware: Both modules Synchronizer1 and Synchronizer2 have the same roles as before. Synchronizer1 provides the compressed data for the decoder and

Synchronizer2 receives the difference vectors to generate the original test vector using a CSR. The design of the Run-length decoder is discussed below.

Due to the fixed length of the code word in this method of compression, the decoder must always shift in 3 (*lcw*, declared generic in the code) bits of data and decompress them. To do so, *LCW_CNT* counts up to *LCW* while shifting the *d_in* input into the *NUMZ_SHR* shift register. If this value is all zeroes, it means that there was a single '1' in the original code which did not have any runs of zeroes, and if this value is all ones, it means that there were 7 consecutive zeroes in the original code followed by another zero. These two situations are conveyed to the controller via *MAX_VAL_FLAG* and *MIN_VAL_FLAG*. The RTL design of the Run-length datapath is shown in Figure 8-9.

The controller of the Run-length decoder is shown in Figure 8-10. In the state machine, state_2 shifts in the 3 bits of compressed data into the *NUMZ_SHR* shift register. If none of the conditions of *MIN_VAL* and *MAX_VAL* apply, in state_4 a single '1' is put on the *diff_out* output. Then, in state_5, in each clock cycle *diff_out* has the value of '0' while the *valid* signal is set to one. The controller stays in state_5 while the *LCW_CNT* counts up to the value of *NUMZ_SHR*.

If the condition of *MAX_VAL* is met, the controller must put 7 consecutive zeroes on the *diff_out* output, which is done through the path of state_3, state_7, and state_5. If the condition of *MIN_VAL* is met, then the controller must put a single '1' on the *diff_out* output, which is done through the path of state_2 and state_3.

Same as Golomb decompression, the difference vector provided by the Run-length decoder is fed to Synchronizer2 in which by using a CSR the original test vector is attained.

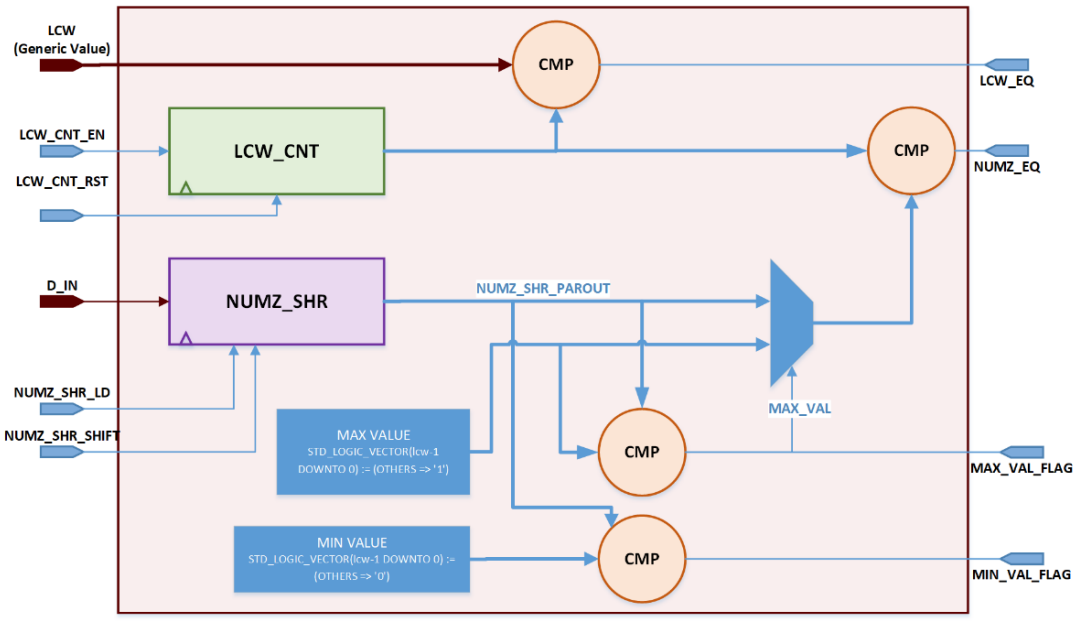

Figure 8-9 Datapath of Run-Length decoder

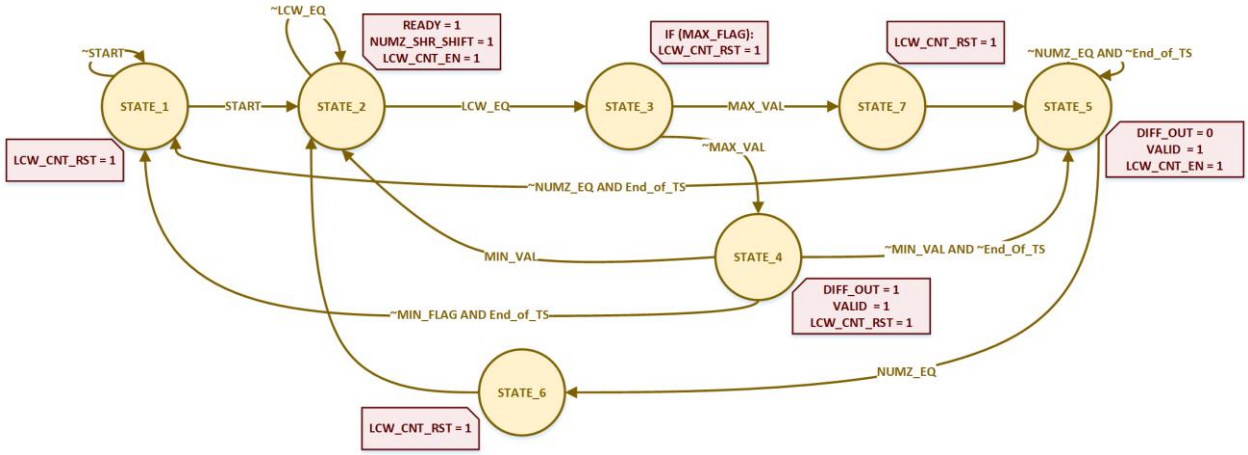

Figure 8-10 Controller of Run-Length decoder

Simulation Results: In order to test the Run-length decompressor, it is connected to the SAYAC processor to provide the test vectors during the fault simulation similar to Golomb. A snippet of the fault simulation log is shown in Figure 8-7, which indicates the correct detection of injected fault by the decompressed test vectors.

```
==================================> @ 12.5 ns    SCAN TEST MODE is starting ...
faultNum 1 = SA@0 on VirtualTester/FUT/SAYAC_Logic/nor_n_92/in1(0)     , injected @ 12.5 ns, detected by testVector 1 = 10001011101000111101000111010110000111111000101011101
faultNum 2 = SA@0 on VirtualTester/FUT/SAYAC_Logic/nor_n_92/in1(1)     , injected @ 9747 ns, detected by testVector 1 = 10001011101000111101000111010110000111111000101011101
faultNum 3 = SA@0 on VirtualTester/FUT/SAYAC_Logic/nor_n_94/in1(0)     , injected @ 19477 ns, detected by testVector 6 = 10001010000000111001100010110011011101111111110110010
faultNum 4 = SA@0 on VirtualTester/FUT/SAYAC_Logic/nor_n_94/in1(1)     , injected @ 52207 ns, detected by testVector 1 = 10001011101000111101000111010110000111111000101011110
faultNum 5 = SA@1 on VirtualTester/FUT/SAYAC_Logic/nand_n_95/in1(0)    , injected @ 61937 ns, detected by testVector 1 = 10001011101000111101000111010110000111111000101011110
faultNum 6 = SA@1 on VirtualTester/FUT/SAYAC_Logic/nand_n_95/in1(1)    , injected @ 71667 ns, detected by testVector 2 = 10000011000111100011110011101111000001111101100000100
faultNum 7 = SA@0 on VirtualTester/FUT/SAYAC_Logic/nor_n_96/in1(0)     , injected @ 85852 ns, detected by testVector 2 = 10000011000111100011110011101111000001111101100000100
```

Figure 8-11 Run-Length simulation results

The difference in test set size after the Run-length compression is shown in Table 8-2.

Table 8-2 Run-Length compression results

Compression method	Test set original size	Compressed test set size
Run-Length	114 KB	158 KB

9 Complete Testing of SAYAC System

Test time is an important factor in the test procedure which reflects the test cost. To reduce the test time, the logic and memory parts of SAYAC are tested simultaneously in a single test session. Along this line, the multiple scan testing and MBIST are executed for testing the logic and memory parts of SAYAC, respectively. It is worth mentioning that the logic part of SAYAC can be tested using the other test architectures discussed in Sections 4 to 6 like logic BIST or JTAG. To make a demo version of the complete testing of the SAYAC system, the multiple DFT scan is chosen here to test the logic part of SAYAC. In the complete system testing, the SCAN and MBIST methods are the same as what was discussed before. Figure 9-1 illustrates the architecture of complete testing of the SAYAC system.

To separate the normal and test modes, two major multiplexers that are controlled by the NbarT signal are added around the SAYAC processor and the memory blocks. After entering the test mode, the multiple scan method begins logic testing and simultaneously memory testing initiates the MBIST modules. RAM and ROM use a shared bus hence they cannot be tested simultaneously. Therefore, memory testing is programmed to be executed serially. At first, the TRF test program begins testing the register file. After that, the instruction ROM and the data RAM are respectively tested. The pseudo-code of the HDL testbench for complete testing of the SAYAC system is shown in Figure 9-2. Figure 9-3 also shows the simulation execution flow for the complete testing scenario.

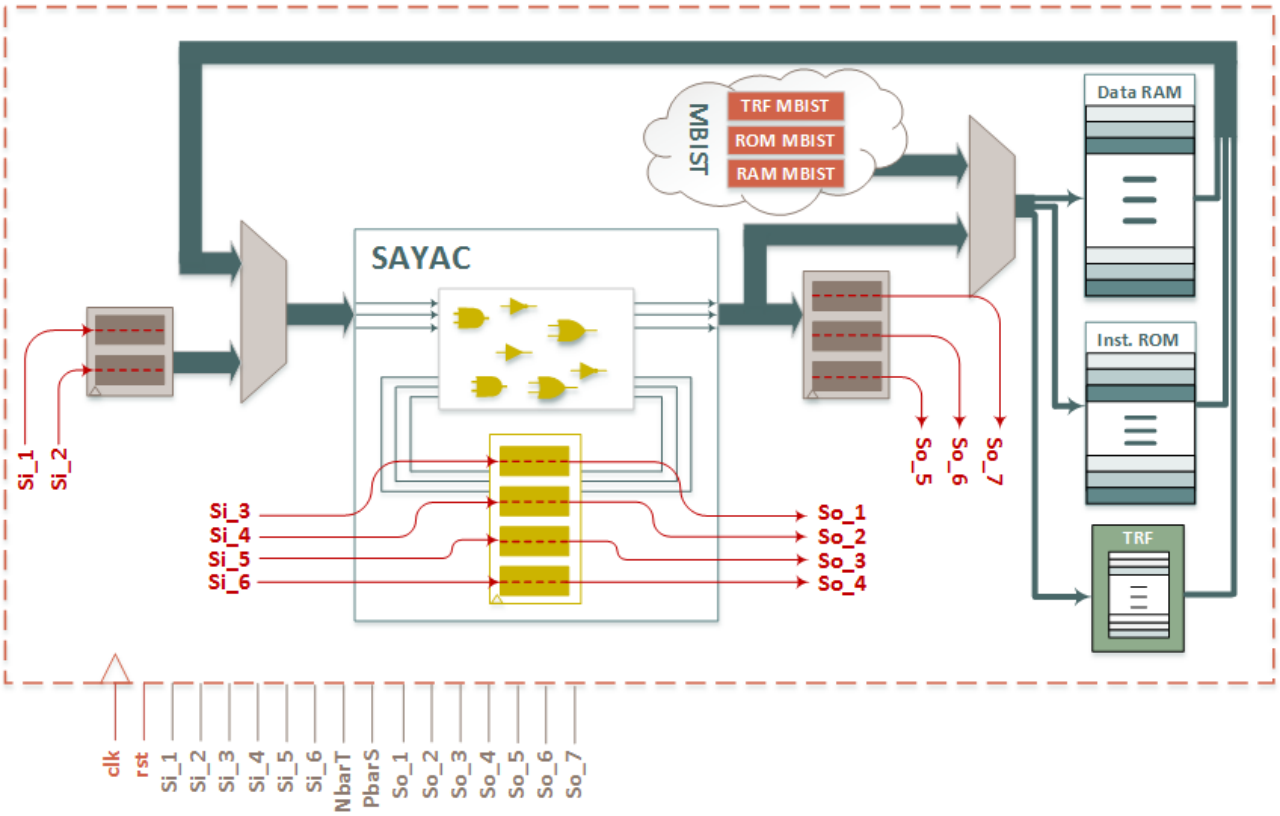

Figure 9-1 Architecture of complete testing of SAYAC system

```
Instance of testable SAYAC system

Normal/Test Process
       NbarT = 0
       Wait for functional results
       NbarT = 1
       Issue start_TRF_testing
       Wait for TRF testing results
       Issue start_ROM_testing
       Wait for ROM testing results
       Issue start_RAM_testing
       Wait for RAM testing results
End Normal/Test Process

//Test program for Logic of SAYAC
TP_LGC Process
       Wait for NbarT
       Perform multiple scan testing
End TP_LGC Process

//Test program for The Register File of SAYAC
TP_TRF Process
       Wait for start_TRF_testing
       Perform TRF MBIST
End TP_TRF Process

//Test program for ROM of SAYAC
TP_ROM Process
       Wait for start_ROM_testing
       Perform ROM MBIST
End TP_ROM Process

//Test program for RAM of SAYAC
TP_RAM Process
       Wait for start_RAM_testing
       Perform ROM MBIST
End TP_RAM Process
```

Figure 9-2 Pseudo-code of virtual tester for SAYAC system

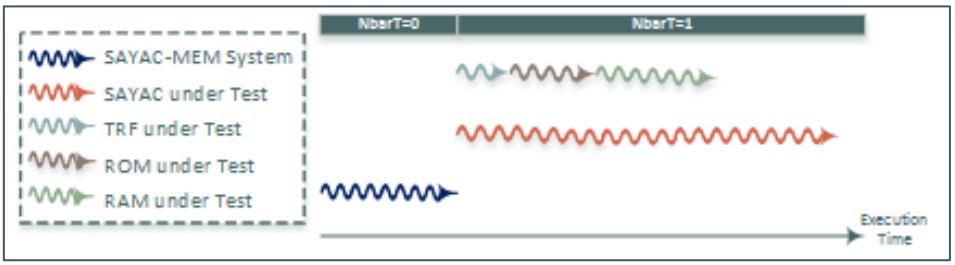

Figure 9-3 Simulation execution flow for complete testing of SAYAC system

10 SystemC Test Environment

The de facto hardware description language for ESL design and modeling is the SystemC [21] hardware description language. A library of C++ classes that handle timing and concurrency form the center part of SystemC. It provides an event-driven simulation interface for multi-domain, multi-level modeling and verification. It helps the designer to work in different levels of abstraction from the system level down to the transistor level. This language has an AMS (analog and mixed-signal) extension; SystemC-AMS; that gives it a very convenient interface for modeling analog electronics and non-electric systems and components. In addition, the C++ base of SystemC gives it an easy two-way interface with software programs.

The work presented in [22] shows that simulation speed in SystemC is much higher than an equivalent simulation in VHDL. They compared the simulation time for VHDL-only, SystemC-only, and VHDL+SystemC co-simulation. The experimental results illustrate that the SystemC-only scenario outperforms the other scenarios.

Fault simulation in the SystemC environment provides not only the C/C++ facilities for the development of fault injection mechanism but also the capability of using the same test program for running on the ATE and the virtual tester that mimics the ATE behavior in our models.

Table 10-1 Simulation versus co-simulation, normalized time [22]

	Real Time	*User Time*	*System Time*
VHLD-only	1.5	1.2	2.1
VHDL+SystemC	10.3	7.5	5.3
SystemC-only	1.0	1.0	1.0

In this part, we perform full-scan testing for the SAYAC processor in the SystemC environment. As discussed in Section 2, our toolchain provides the SystemC netlist of the design along with the corresponding fault list and test set. Therefore, fault simulation is done at the gate level (the same level of abstraction as what we had performed in VHDL). The same fault list and test set as Section 4.2 are also used here.

To handle fault injection, an intrusive mechanism has been developed in SystemC. In this method, the component library is replaced with its faultable version, and the stuck-at faults are injected into the faultable components. A fault injection manager (FIM) mimics a virtual tester and is responsible for performing the fault simulation process discussed in Section 3. The FIM module handles reading fault list, reading test patterns, applying them, collecting test responses, and comparing results. The FIM module is also auto-generated by our toolchain considering whether the CUT is sequential or pure combinational.

The overall view of fault simulation in the SystemC environment is shown in Figure 10-1. As shown, our toolchain (UT_DATE) receives the RTL description of CUT and test configuration (here full-scan) as inputs. It generates the SystemC netlist of CUT, the corresponding fault list and test set. In the

next step, it inserts the full scan architecture into the CUT. After that, it generates the FIM module as a virtual tester to handle the fault simulation process for the scannable SAYAC. In the last step, it can call SystemC to run the simulation and report the test metrics.

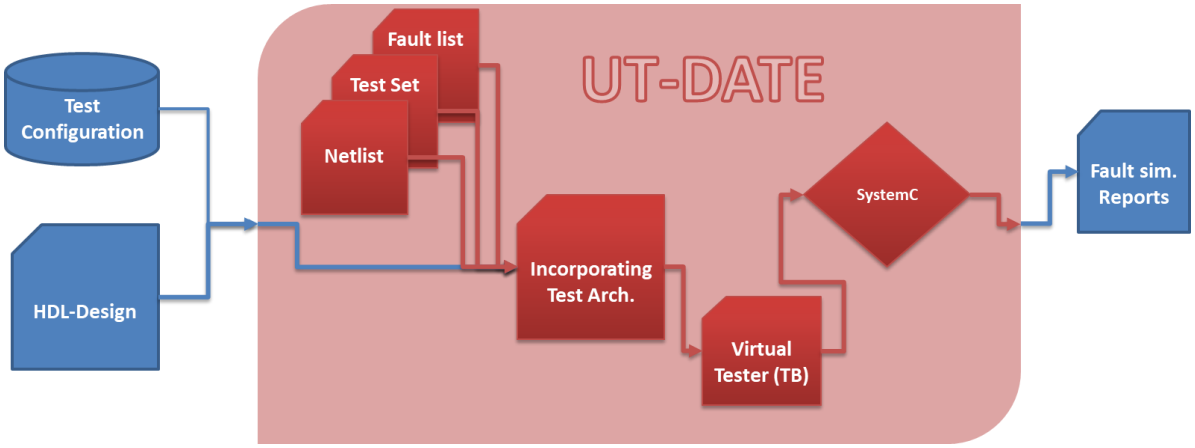

Figure 10-1 Fault simulation in SystemC

11 Verification

11.1 Element Base Verification

Design verification is a process of ensuring that a design exhibits intended behavior. There are two broad approaches to hardware design verification: formal methods, simulation-based methods, and semi-formal methods as shown in Figure 11-1. Among these three methods of verification, formal methods and open-source tools that are used are the main concern of this work.

Formal verification methods attempt to mathematically prove the correctness or incorrectness of the designed systems. Ideally, a formal method, consists of a formal language, tools, and a proof system, which can be used to specify and verify systems. The typical hardware verification scenarios where formal proof techniques are applied are Equivalence Checking (EC) and Property Checking (PC), also called Model Checking (MC).

The goal of EC is to ensure the equivalence of two given circuit descriptions. These circuits might be given on different levels of abstraction, i.e., register transfer level or gate level. In contrast to EC, where two circuits are considered, for PC a single circuit is given, and properties are formulated in a dedicated "verification language". It is then formally proven whether these properties hold under all circumstances.

Model checking process requires to determine *property specification* and *system modeling*. Typically, we can specify the properties of the system using temporal logic and automata. Temporal logic is a variant of modal logic for expressing temporal modalities and representing propositions qualified in terms of time. Depending upon the system types, we have to select the distinct types of temporal logic. LTL is more suitable for specifying sequential systems. Whenever we have to verify branching cases in some of the states, then CTL is more suitable. Once we know the system specifications from the requirement document, next step is to model the system. We can model the system text-based or graphics-based methods.

Depending upon the choices of the system model (FSM, Process Algebra, GTS) and property specification methods (temporal Logic, automata), different Model checking approaches like Explicit MC, Symbolic MC, Bounded MC and On-line MC (OMC).

In BMC, Model checking complexity is reduced to a propositional satisfiability problem that can be solved with SAT solvers. So, the size of state space will decrease and increases the speed. An extension of propositional satisfiability (SAT) is SMT (Satisfiability Modulo Theories) that is the most well know constraint-satisfaction problem. It generalizes Boolean satisfiability by adding equality reasoning, arithmetic, fixed-size bit vector, arrays, quantifiers and other useful first-order theories.

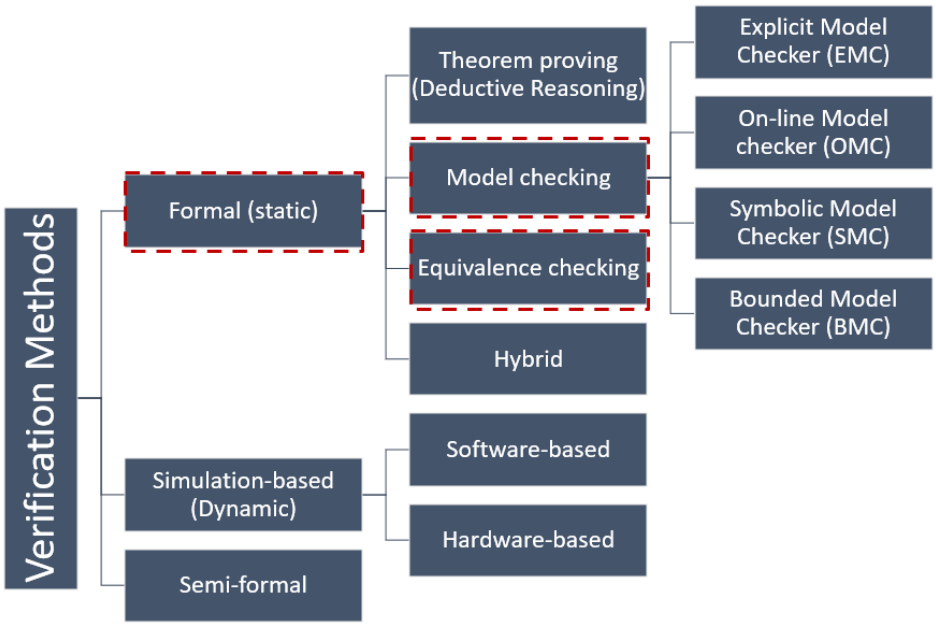

Figure 11-1 Taxonomy of verification methods

11.2 Open-source Verification Tools

Building upon the verification concepts discussed above, we use the following tools to formally prove the functional correctness of some designs.

UPPAAL: UPPAAL is an integrated tool environment for modeling, simulation, and verification of real-time systems. It performs formal verification based on model checking. Considering a Booth multiplier as DUT, we have specified the properties of the system with CTL. Figure 11-2 shows an overview of the inputs and outputs of UPPAAL for the test case that has been performed.

Yosys: Yosys is a well-known open-source framework for design synthesis and verification. It can perform formal verification based on equivalence checking. Figure 11-3 shows an overview of using Yosys for a full adder.

Yosys-SMTBMC: Yosys-SMTBMC is a tool based around Yosys and various SAT solvers to let you do formal verification on your code. Yosys-SMTBMC can use any SMT solver with support for the SMT-LIB2 QF_AUFBV logic. It has been used with the Z3 solver in our case study. SymbiYosys is a wrapper for several programs, including yosys-SMTBMC. Figure 11-4 shows an overview of using Yosys-SMTBMC to perform bounded model checking on an incrementor.

Netgen: Netgen is a program with two purposes: one is to convert netlists between different formats, and the other is to compare two netlists to determine if they are equivalent, and if not, what makes them different. Netlist comparison is often called LVS (Layout vs. Schematic), because its primary purpose is to check whether a VLSI layout is equivalent to the schematic from which it is derived.

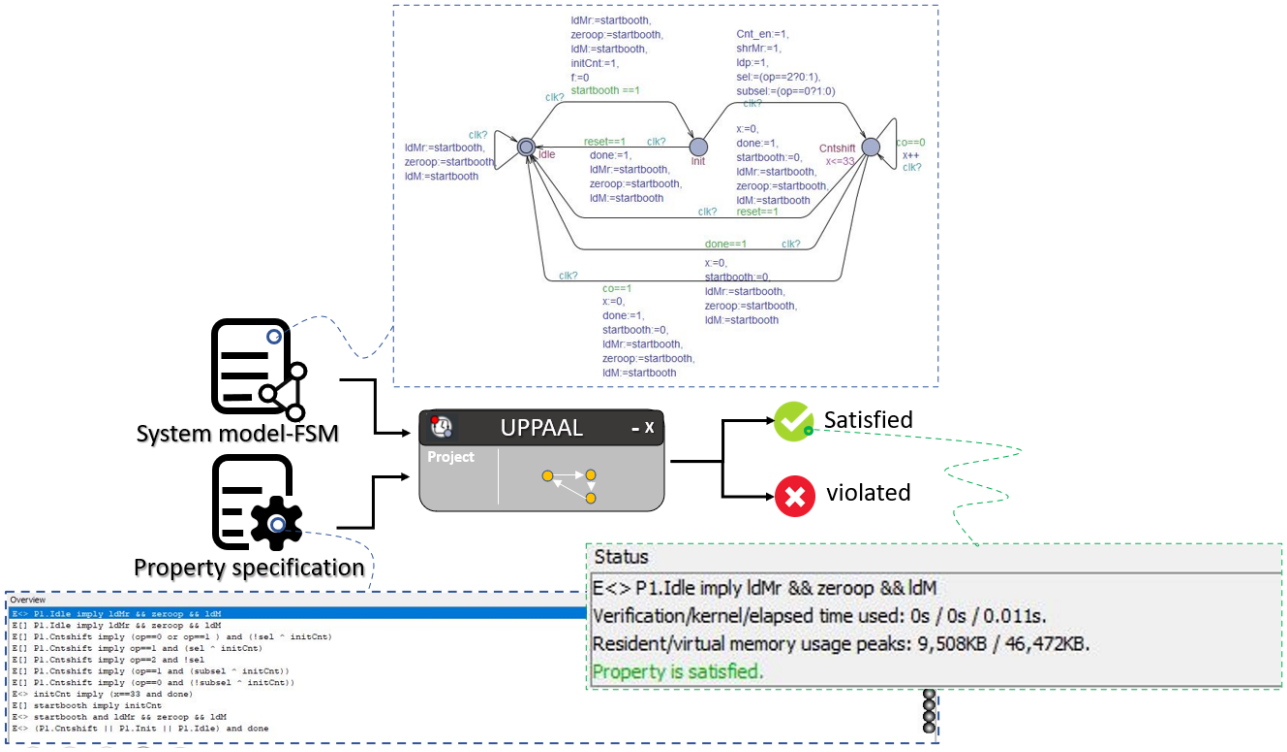

Figure 11-2 Overview of using UPPAAL for model checking

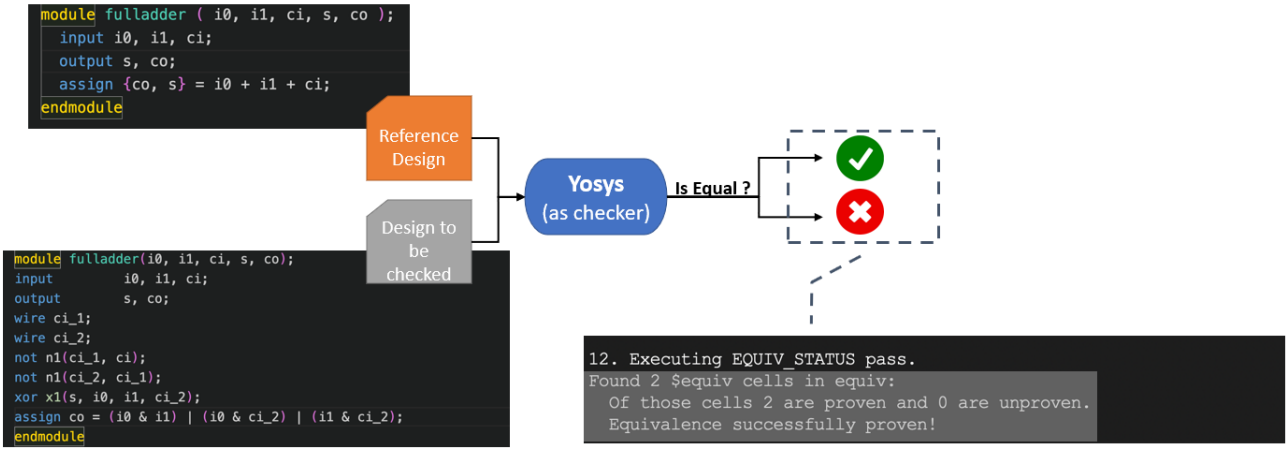

Figure 11-3 Overview of using Yosys for equivalence checking

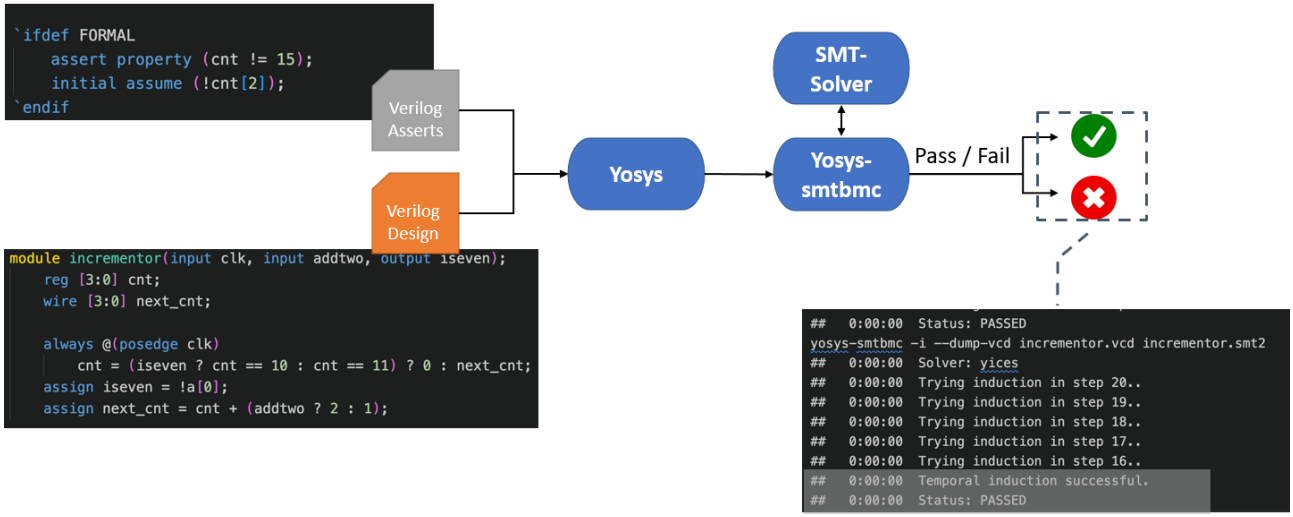

Figure 11-4 Overview of using Yosys-SMTBMC for model checking

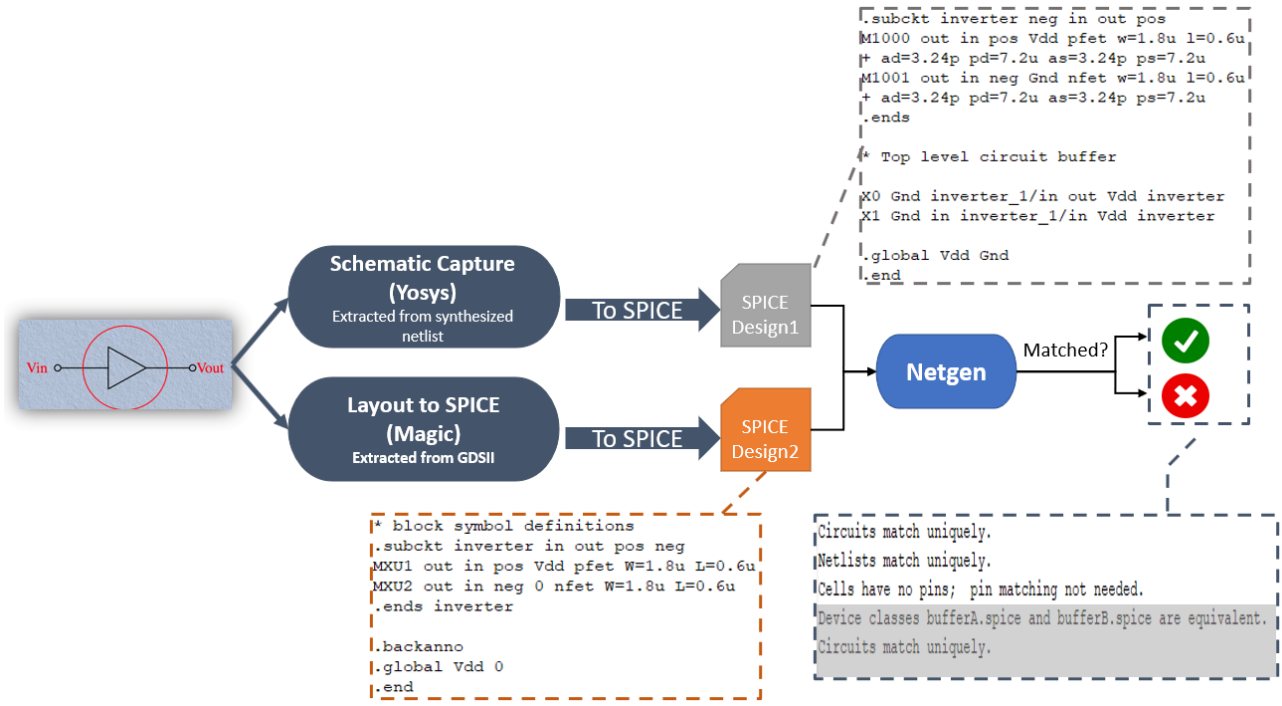

Figure 11-5 Overview of using Netgen for equivalence checking

12 Conclusions and Future Works

- A complete embedded system as a security benchmark
- What is done
  - Test methodology and Environment
  - Implementation of conventional test methods to the SAYAC-MEM system
- Some suggestions:
  - Test methods for cache, SAYAC peripherals, and other components in SAYAC L2
  - Test and testability of the AFTAB-MEM system
  - Implementation of other test architectures for the SAYAC-MEM system
    - Software-based self-test (SBST)
    - Memory testing using JATG
    - . . .
  - Other fault models for logic and memory
  - Fault injection and test evaluation in higher levels of abstraction
  - Test and testability in SystemC Environment
  - Implementation of conventional security methods to the testable SAYAC-MEM system
  - Moving the fault injection mechanism toward attack injection
  - Developing a configuration test environment
  - FPGA-based custom tester by synthesizing the virtual tester

13 References

[1] http://yosyshq.net/yosys

[2] https://github.com/hsluoyz/Atalanta

[3] Z. Navabi, Digital system test and testable design. E-ISBN, 97814419-97875485, 2011.

[4] Pau Fontova Must´e, Design for Testability methodologies applied to a RISC-V processor, Master Thesis, 2021.

[5] Semiconductors, Siemens, "Hierarchical DFT in a RISC-V processor," white paper, 2019.

[6] D. Gizopoulos, A. Paschalis, and Y. Zorian, "Embedded processor-based self-test," vol. 28, Springer Science & Business Media, 2013.

[7] L. Chen, et al. "Embedded hardware and software self-testing methodologies for processor cores," DAC 2000.

[8] R. Rajsuman, "Testing a system-on-a-chip with embedded microprocessor," ITC 1999.

[9] M. Tehranipour, et al. "A low-cost at-speed BIST architecture for embedded processor and SRAM cores," Journal of Electronic Testing, vol 20, no. 2, 2004.

[10] G. Hetherington, et al. "Logic BIST for large industrial designs: Real issues and case studies," ITC 1999.

[11] M. Riley, et al. "Testability features of the first-generation Cell processor," ICT, 2005.

[12] Semiconductors, N. X. P. "Using the Built-in Self-Test (BIST) on the MPC5744P," Application Note, 2017.

[13] R. Dorsch and H.-J. Wunderlich, "Accumulator based deterministic BIST," ITC, 1998.

[14] P. H. Bardell and W. H. McAnney, "Self-testing of multichip logic modules," ITC, 1982.

[15] EEE Standards Association, "IEEE Standard for Test Access Port and Boundary-Scan Architecture," IEEE Std 1149, 2013.

[16] N. Nosrati, et al. "Testing a RISCV-Like Architecture With an HDL-Based Virtual Tester." DTIS, 2021.

[17] L. Whetsel, "An ieee 1149.1-based test access architecture for ics with embeddded cores." ITC, 1997.

[18] M. Tuna, and M. Benabdenbi "Software Based Self-Test of Register Files in RISC Processor Cores using March Algorithms," IEEE Latin America Test Workshop, 2006.

[19] T. J. Bergfeld, D. Niggemeyer and E. M. Rudnick, "Diagnostic testing of embedded memories using BIST," DATE 2000.

[20] P. K. John and P. R. Antony, "Optimized BIST architecture for memory cores and logic circuits using CLFSR," ICICICT 2017.

[21] https://www.accellera.org/downloads/standards/systemc

[22] W. Müller, W. Rosenstiel, J. Ruf, "SystemC - Methodologies and Applications." Kluwer Academic Publishers, 2003.

[23] F. Novak and A. Biasizzo, "Security extension for IEEE Std 1149.1." *Journal of electronic testing*, 2006.