Development of UVM based Reusabe Verification Environment for SHA-3 Cryptographic Core

M. N. Kubavat
Dept. of VLSI & Embedded Systems Design, GTU PG School
Gujarat Technological University
Ahmedabad, India
kubavat.mayur@gtuinstitutes.ac.in

Mr. Manjunath Gowda
Dept. of VLSI & Embedded Systems Design, GTU PG School
Gujarat Technological University
Ahmedabad, India
manjunathkarpur@gmail.com

Abstract— In this work, an industry standard methodology for ASIC verification domain, SystemVerilog (SV) with Universal Verification Methodology (UVM) is introduced with its features and application to Keccak SHA-3 Cryptographic Core. The ASIC verification flow for SHA-3 core is followed with creation of UVM based verification environment. By application of UVM on the core, horizontal and vertical re-use can be achieved in standard projects. Proposed verification environment uses OOPs concepts from SV UVM to develop layered testbench. In this approach initial learning curve is slow, considering overhead to learn new verification methodology. But, once full fledge working environment is created, re-usability feature from SV UVM can be achieved with less amount of time. Also coverage results give effectiveness of the proposed verification environment.

Keywords*-UVM; SystemVerilog; object-oriented programming; ASIC Verification; Keccak SHA-3*

*****

I. INTRODUCTION

Complex ASIC/SoC design requires creation of versatile verification environment for functional verification of RTL cores. This requires creation of modular/layered testbench with concepts of object-oriented programing. Application of object-oriented verification methodology comes into consideration to verify complex designs, these include eRM (e Reuse Methodology), OVM (Open Verification Methodology), and UVM (Universal Verification Methodology). From which UVM is supported by major EDA tool vendors and widely used in industry. Through application of UVM, vertical reuse across different hierarchical level as well as horizontal reuse across different project can be achieved. By creation of UVCs (UVM Verification Components), development of VIP is possible.

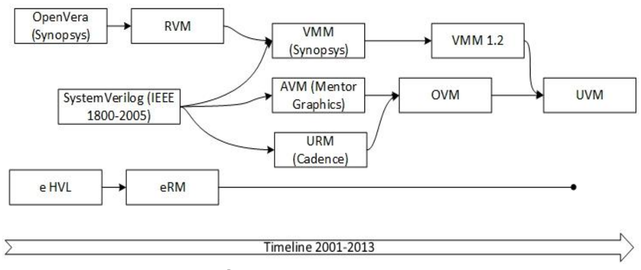


Figure 1 Verification Methodology Timeline

This paper introduces industry standard approach to create verification environment for Keccak SHA-3 [1], [2] core by application of UVM methodology. The approach discusses creation of reusable verification component inside verification environment by application of Object Oriented Programing (OOP) concept. Configurable Agent, Score boarding, reusable Sequences are implemented in proposed verification environment.

Organization of remaining paper is in following sections. Section II introduces with introduction of Keccak SHA-3, section III gives UVM methodology construct details, section IV describes component details of proposed verification environment. Section V consists of proposed environment block and section VI evaluates coverage information and efficiency of proposed environment. Conclusion is made in section VII.

II. SECURE HASH ALGORITHM-3 SPECIFICATIONS

Keccak SHA-3 is recently introduced hash function based on Sponge construction [2]. Keccak is family of sponge function with function as *Keccak-p = [b, 2l+12]* permutation as underlying function and padding rule - *pad1*01*. Any specific function from family is defined by choices of parameters rate r and capacity c such that *r+c* is 25, 50, 100, 200, 400, 800, and 1600. Proposed verification environment make use of crypto processor core with 512-bit hash value for which value of b is 1600-bits [3]. So, if we consider case with b=1600, Keccak family is denoted by Keccak[c]; in this case r

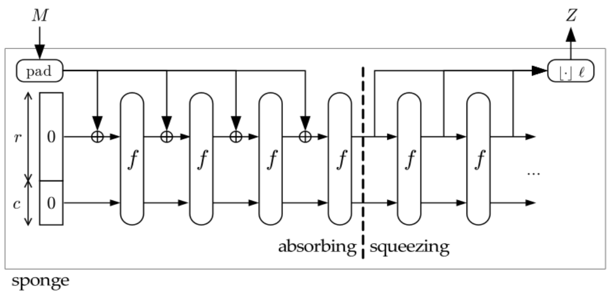


Figure 2 Sponge Construction

is determined by choice of c.

*Keccak[c] = SPONGE[Keccak-p[1600, 24], pad10*1, 1600 –c]*

Thus, given M and output d we have,

*Keccak[c] (M, d) = SPONGE[Keccak-p[1600, 24], pad10*1, 1600 –c] (M, d).*

III. UNIVERSAL VERIFICATION METHODOLOGY

UVM guidelines provide uses of SystemVerilog language to create reusable efficient testbench. UVM class library provides automation to SystemVerilog language. Methodology is used to develop recommended architecture for creating testbench. UVM provides framework for Coverage Driven Verification (CDV). CDV has combination of automatic test generation (pseudo-random pattern generation), self-checking architecture and coverage metrics to reduce time for developing testbench significantly [4]. UVM class library consist of different components for consistent testbench architecture. These classes consist of Environment, Agent, Driver, Sequencer etc. Descried classes are also known as verification components. UVM Verification Component (UVC) can be extended to make DUT specific components. UVC are ready to use component for Bus Protocol, Design Module or Systems. UVC have consistent architecture and composed of methods for simulating, verifying and collecting coverage data.

The UVM class library provides all building blocks you need to quickly develop verification environment. It consist of UVM base classes, pre-defined methods like *print(), copy()* etc. It also consists of utilities, macros and provides Transaction Level Modeling (TLM 1.0) set for communication between UVCs.

A. *UVM Class Library*

UVM class library has three types of constructs inside. One is *uvm_component* base class, it provides hierarchical components like Driver, Monitor etc. Other two types are *uvm_object* and *uvm_transaction* provides configuration objects and stimulus carrying packets in that order. A partial list of UVM class hierarchy is given below,

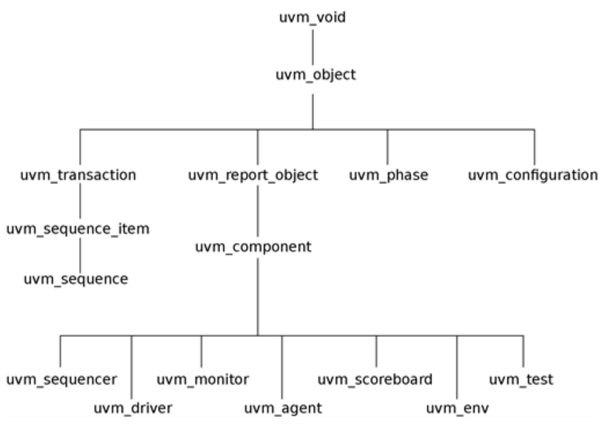


Figure 3 UVM Class Library Hierarchy

B. *UVM Phases*

UVM has long list of simulation phases, which based on complexity of verification environment, added to construct, configure and connect components in UVM based environment [5]. Some of the phases used in proposed TB architecture are;

- *function void build()* phase constructs components of testbench hierarchy. For e.g. build phase of agent constructs Driver, Monitor and Sequencer.
- *function void connect()* connects components created in previous phase via TLM 1.0
- *function void end_of_elaboration()* this uvm phase is used to print name of components created. *Print()* method called here prints components hierarchically.
- *run_test()* phase will run the test provided by +UVM_TESTNAME=<test_name> and in turn will call *run_phase()* of every component top-down in test environment hierarchy.

C. *TLM 1.0*

Transaction Level Modelling provides communication infrastructure between UVM component objects. A component can communicate with other components that implements specific interface. TLM specifies behavior but does not implement the method, it is provided by classes inheriting TLM interface. TLM 1.0 is message passing system, while TLM 2.0 is mainly designed for high-performance

IV. VERIFICATION PLAN FOR KECCAK SHA-3 RTL CORE

The plan includes architecture of the DUT, specification extraction, creating test scenarios and testcases based on test scenarios. Verification environment has been created based on the DUT architecture. This verification environment includes VCs such as test module, environment module, agent and scoreboard etc. Agent class creates abstraction by instantiating Driver and Monitor block inside it. Env class provides Scoreboard, active and passive Agents and connection between them through TLM 1.0 ports [6].

Sequence Item: It extends *uvm_sequence_item* and represent packet with no initial values. Sequence items are called in Driver component's run phase and driven to dut through virtual interface.

Driver: This component contains definition of virtual interface, sha3_vif in our design. It extends *uvm_driver* class and parameterized with type Sequence Item. Driver also searches for sha_vif from *uvm_config* database.

Monitor: Monitor collects pin-wiggles from virtual interface and creates package of type Sequence Item. These sequence items are sent to any component which is connected with monitor class through *uvm_analysis_port*, TLM 1.0

Configurable Agent: Layered approach in the design requires to create abstraction level containing lower level modules in hierarchy. Proposed verification environment has active and passive agents which creates necessary abstraction to connect and configure whole Env block with ease. Active Agent block creates Driver and Monitor both, whereas passive Agent only has monitor and is connected to dut output in our environment. Note, active Agent also defines sequencer which is connected to Driver through ports and exports. Here, in Agent abstraction level is not necessary in some cases, and also sequencer can be called from another abstraction level too. Use of active/passive configuration of agents improves level of abstraction. And creation of two different Monitor, coding time, can be avoided.

Scoreboard: Module contain *tlm_fifo* to collect sequence item packets. Scoreboard is connected to input Monitor and output Monitor and defines *uvm_analysis_export* method. This analysis export can be connected to *uvm_tlm_analysis_fifo* defined in TLM 1.0. Output from FIFO are collected and documented by application of file I/O methods [7].

Env: Environment module encapsulates all modules of lower hierarchy. In our environment active/passive Agent,

agent configuration and Scoreboard are created and connected. This environment class is called off and created in test modules.

A. *Requirements for Funcitonal Coverage and Code Coverage*

To successfully verify DUT and RTL sign-off time to do that is very important question every verification engineer faces. Certain parameters defines this problem with one thing in consideration, it's how much to test. These test coverage criteria are provided by code coverage and function coverage requirements of DUT. Typical code coverage conventions are tool dependent and includes statement coverage, First Expression Coverage (FEC), Toggle coverage etc. Whereas functional coverage is defined by person who verifies design. Functional coverage help keep record of how much functionality is covered. Effectiveness of developed verification environment is measured by these two parameters.

V. Development of Verification Environment

Proposed verification environment make use of all sub-blocks discussed in section IV. Figure 4 shows connection between modules developed using UVM methodology along with SystemVerilog verification environment. SHA-3 DUT is connected to our test environment via virtual interface which is dynamic in nature.

In comparison of static testbench modules using Verilog for verification. By utilizing concepts of object oriented programming provided by SystemVerilog necessary objects and test patterns can be created at run-time and objects no longer necessary will be garbage collected with in-built feature.

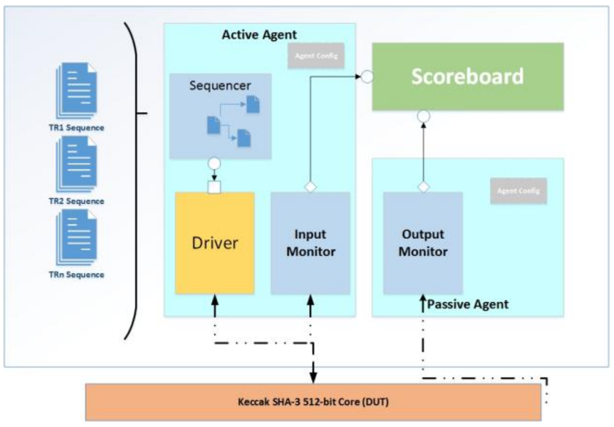


Figure 4 Proposed Verification Environment

So, advantage of proposed SV UVM verification environment over static testbench using Verilog is, less system memory will be occupied when running the test. Here, input monitor collects data provided to the SHA-3 core and sends the data to Scoreboard for checking purposes. Communication between the scoreboard and monitor has been made using TLM 1.0 discussed in section III. Due to hierarchical approach used for proposed environment scoreboard to monitor connection is indirect in nature. The connection is made in Env block between Active Agent and Scoreboard with TLM port-export. Advantage here is, if we want to utilize Sequencer-Driver-Monitor pair in different hierarchical level which uses Hashing Algorithms, there we can apply stimulus to the inputs by reusing configurable agent block. Same concept will be application to passive agent as well.

Driver component in proposed architecture is parameterized block which takes transaction item created by sequencer and initialized with values and drives it to DUT at signal levels in form of pin wiggles. Parameterized driver module will look like this;

```
class sha3_driver extends
uvm_driver#(sequence_item);
  `uvm_component_utils(sha3_driver)

          virtual sha3_intf sha3_vif;

          \\Constructor and build phase

    //run phase to drive packet to DUT
endclass: sha3_driver
```

To generate test cases for the selected DUT, port list should be available with legal combination of signals to be applied to generate desired outputs.

Typical inputs inside Cryptographic core are vector input, input enable, clock, etc. Our concern here is to develop reusable UVM environment where port widths and/signals can differ, so we will not go to that part. Instead for typical list of ports in design, our *transaction* packet module will look like this,

```
class sequence_item extends
uvm_sequence_item;
  `uvm_object_utils(sequence_item)

  //Constructor

  logic rand [WIDTH-1:0] in;
  logic   in_ready;
  logic   is_last;
  ………
endclass: sequence_item
```

Here, applying randomization coverage can be improved with less manual effort to create directed testbench

VI. Simulation Results

Application of testcases generated based on Keccak core specification results in different coverage scenarios. To implement UVM verification environment system configurations which we used are, QuestaSim 10.0b running on Linux Ubuntu 15.04 64-bit. Different testcases, for e.g. Input golden message sequence to the Design under Verification results in required value of 512-bit hash:
Figure 5, is again shown after reference list with appropriate display of waveforms as Figure 8. As there's 12 clock cycles of difference between applications of inputs and valid output results. To capture whole simulation window will need more space to be displayed properly.

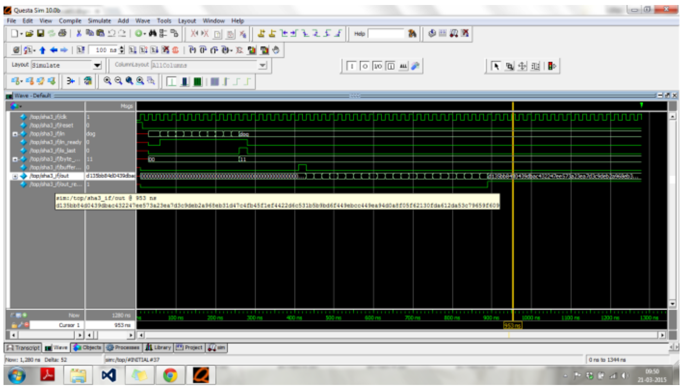


Figure 5 Input golden message sequence

Input of golden message to Keccak SHA-3 takes 1280ns for simulation runtime which is devised in 52 deltas. Correct value for hash can be verified by look-up tables stored in scoreboard against golden message string.

To check correctness of DUT, standard testcases are needed. But, to check reliability of RTL designs boundary cases are necessary to confirm correctness of output on extreme usages. Waveforms generated by application of empty string on core input is in figure 6 and figure 9. Other boundary cases could be application of very long message which can cause buffer to overflow. And interrupting input text sequence time and again by asserting input not ready signal in design under verification, which will affect throughput of the test module.

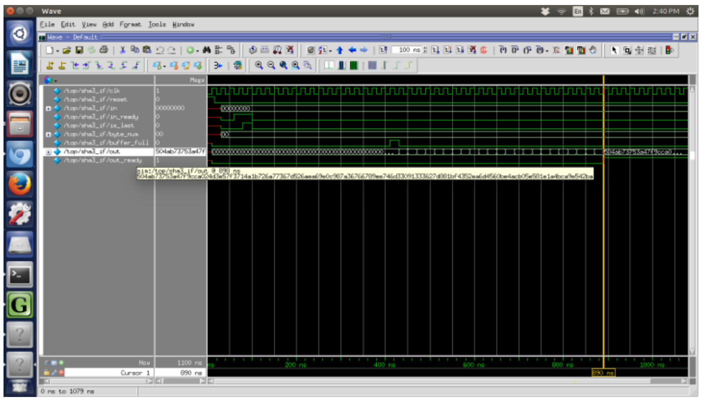


Figure 6 Input empty string

Sometimes to measure reliability of design, verification person introduces error scenario and applies error cases which causes design to perform under unexpected circumstances. In those cases verification intent is not to store wrong results but to observe critical signals of design under test and to observe effect of corrupt data in normal input sequence to the design.

By application of proposed verification environment to the SHA-3 RTL core, code coverage results achieved are given in figure 7 below and also figure 10 after reference list,

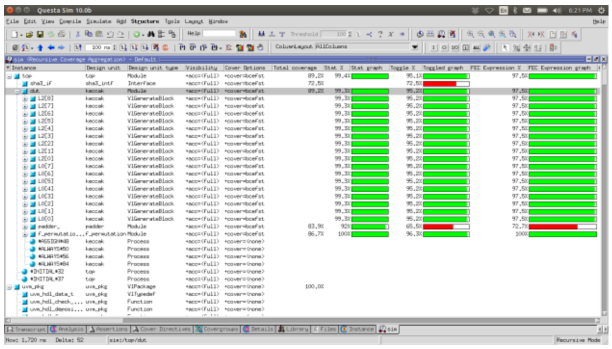


Figure 7 Coverage Results

Coverage statistics collected for SHA-3 Keccak Core in text file format is given in table 1,

File: low_throughput_core/rtl/dut.v	
Enabled Coverage	**% Covered**
Stmts	100
Branches	100
Toggle Bins	94.6

Table 1 Coverage Statistics

Conclusion

To develop verification environment using industry standard methodology, UVM is very effective for creating dynamic layered testbench. UVM is Accellera standard methodology to be used in RTL verification projects and it is openly available and supported by major EDA tool vendors.. Only initial learning curse for UVM is low, but once full fledge working environment is created reusability comes into picture which will make components flexible and portable for reuse. In this work, UVM environment is developed and applied to verify SHA-3 Core DUT. Results we achieved implies efficiency of proposed environment with dynamic nature of testbench and flexibility to port environment to other verification projects.

Copy of proposed SV UVM verification environment is made openly available on Git Repository and can be accessed from this link - https://github.com/mayur13/UVM-Verification-Environment

Acknowledgment

This work is guided by Gujarat Technological University and various institute personnel. I would like to acknowledge Mr. Ashish Prabhu for his advices on industry standard ASIC verification flow and ways to approach verification plan with SV/UVM Methodology.

References

[1] NIST Computer Security Division (CSD). "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", Available online at www.csrc.nist.gov/publications/drafts/fips-202/fips_202_draft.pdf

[2] Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche, "The Keccak reference", Available online at www.keccak.noekeon.org/Keccak-reference-3.0.pdf

[3] Sklavos, N., "Towards to SHA-3 Hashing Standard for Secure Communications: On the Hardware Evaluation Development", in Latin America Transactions, IEEE, pp. 1433-1434, 2012.

[4] Mentor Graphics, “UVM Cookbook”, from verification academy.

[5] “UVM User’s Guide” from accellera.org

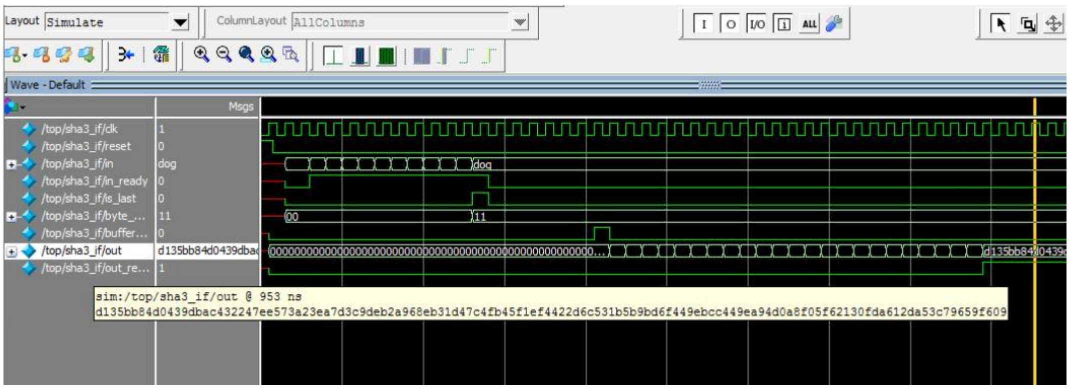
[6] Bromley, Jonathan, “If SystemVerilog is so good, why we need the UVM? Sharing responsibilities between libraries and the core language”, in Specification & Design Languages (FDL), pp. 1-7, 2013.

[7] Francesconi J.; Agustin Rodriguez J.; Julian P.M., “UVM Based Testbench Architecture for Unit Verification”, Argentine Conference on Micro-Nanoelectronics, Technology and Applications (EAMTA), pp. 89-94, 2014.

[8] Geng Zhong; Jian Zhou ; Bei Xia, “Parameter and UVM, Making a Layered Testbench Powerful”, IEEE 10th International Conference on ASIC (ASICON), pp. 1-4, 2013.

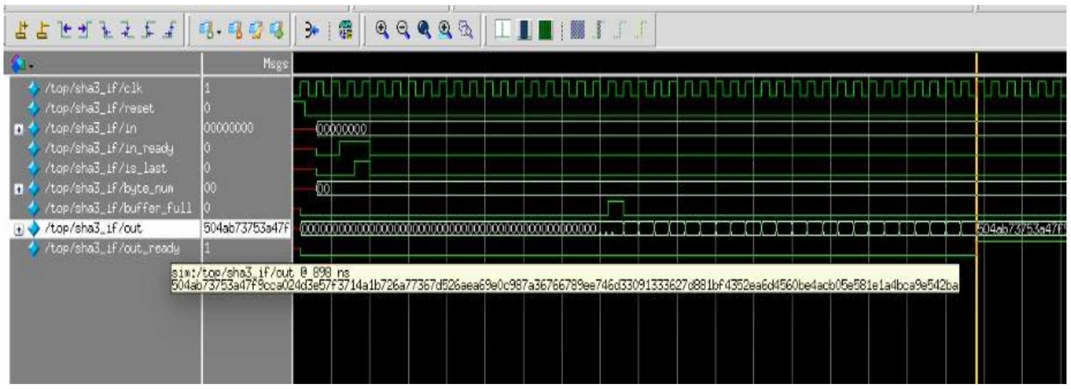
[9] www.OpenCores.org.

[10] Mentor Graphics, “Coverage Cookbook”, from verification academy.



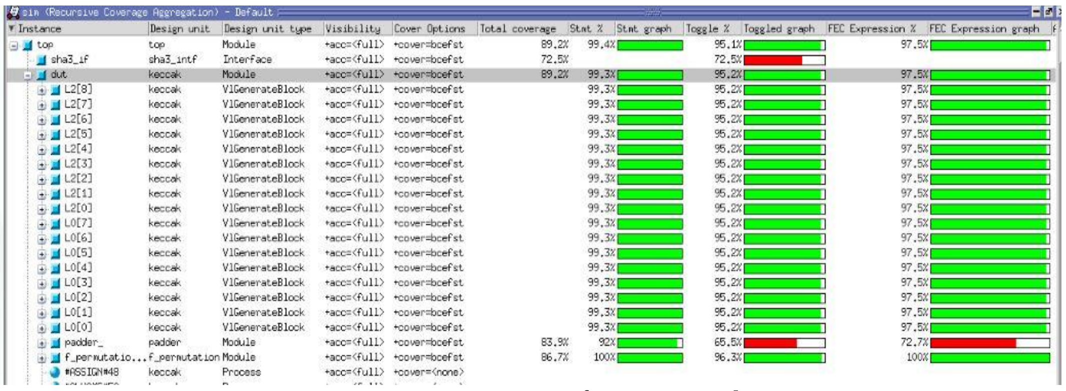
```
sim:/top/sha3_if/out @ 953 ns
d135bb84d0439dbac432247ee573a23ea7d3c9deb2a968eb31d47c4fb45f1ef4422d6c531b5b9bd6f449ebcc449ea94d0a8f05f62130fda612da53c79659f609
```

Figure 8 Golden Inputs and Hash Result



```
sim:/top/sha3_if/out @ 898 ns
504ab73753a47f9cca024d3e57f3714a1b726a77367d526aea69e0c987a36766789ee746d33091333627d881bf4352ea6d4560be4acb05e581e1a4bca9e542ba
```

Figure 9 Empty Input and Hash Result


sim (Recursive Coverage Aggregation) - Default

Instance	Design unit	Design unit type	Visibility	Cover Options	Total coverage	Stmt %	Stmt graph	Toggle %	Toggled graph	FEC Expression %	FEC Expression graph
top	top	Module	+acc=<full>	+cover=bcefst	89.2%	99.4%	🟩	95.1%	🟩	97.5%	🟩
sha3_if	sha3_intf	Interface	+acc=<full>	+cover=bcefst	72.5%			72.5%	🟥		
dut	keccak	Module	+acc=<full>	+cover=bcefst	89.2%	99.3%	🟩	95.2%	🟩	97.5%	🟩
L2[8]	keccak	VlGenerateBlock	+acc=<full>	+cover=bcefst		99.3%	🟩	95.2%	🟩	97.5%	🟩
L2[7]	keccak	VlGenerateBlock	+acc=<full>	+cover=bcefst		99.3%	🟩	95.2%	🟩	97.5%	🟩
L2[6]	keccak	VlGenerateBlock	+acc=<full>	+cover=bcefst		99.3%	🟩	95.2%	🟩	97.5%	🟩
L2[5]	keccak	VlGenerateBlock	+acc=<full>	+cover=bcefst		99.3%	🟩	95.2%	🟩	97.5%	🟩
L2[4]	keccak	VlGenerateBlock	+acc=<full>	+cover=bcefst		99.3%	🟩	95.2%	🟩	97.5%	🟩
L2[3]	keccak	VlGenerateBlock	+acc=<full>	+cover=bcefst		99.3%	🟩	95.2%	🟩	97.5%	🟩
L2[2]	keccak	VlGenerateBlock	+acc=<full>	+cover=bcefst		99.3%	🟩	95.2%	🟩	97.5%	🟩
L2[1]	keccak	VlGenerateBlock	+acc=<full>	+cover=bcefst		99.3%	🟩	95.2%	🟩	97.5%	🟩
L2[0]	keccak	VlGenerateBlock	+acc=<full>	+cover=bcefst		99.3%	🟩	95.2%	🟩	97.5%	🟩
L0[7]	keccak	VlGenerateBlock	+acc=<full>	+cover=bcefst		99.3%	🟩	95.2%	🟩	97.5%	🟩
L0[6]	keccak	VlGenerateBlock	+acc=<full>	+cover=bcefst		99.3%	🟩	95.2%	🟩	97.5%	🟩
L0[5]	keccak	VlGenerateBlock	+acc=<full>	+cover=bcefst		99.3%	🟩	95.2%	🟩	97.5%	🟩
L0[4]	keccak	VlGenerateBlock	+acc=<full>	+cover=bcefst		99.3%	🟩	95.2%	🟩	97.5%	🟩
L0[3]	keccak	VlGenerateBlock	+acc=<full>	+cover=bcefst		99.3%	🟩	95.2%	🟩	97.5%	🟩
L0[2]	keccak	VlGenerateBlock	+acc=<full>	+cover=bcefst		99.3%	🟩	95.2%	🟩	97.5%	🟩
L0[1]	keccak	VlGenerateBlock	+acc=<full>	+cover=bcefst		99.3%	🟩	95.2%	🟩	97.5%	🟩
L0[0]	keccak	VlGenerateBlock	+acc=<full>	+cover=bcefst		99.3%	🟩	95.2%	🟩	97.5%	🟩
padder_	padder	Module	+acc=<full>	+cover=bcefst	83.9%	92%	🟩	65.5%	🟥	72.7%	🟥
f_permutatio...	f_permutation	Module	+acc=<full>	+cover=bcefst	86.7%	100%	🟩	96.3%	🟩	100%	🟩
#ASSIGN#48	keccak	Process	+acc=<full>	+cover=<none>							

Figure 10 Coverage Properties for Design under Test