Qualification of Tied-off Signals inSoC VerificationUsing Mutation Analysis

Prokash Ghosh

Freescale Semiconductor (Part of NXP semiconductor), India

prokash.ghosh@nxp.com

***Abstract*—In typical MP(multi-processor)-SoCs, there are several thousands of tie-off signals.These are growing as complexity and size of SoC is increasing day by day. Mutation testing is popular to find faults in the software program of software system design. There were many attempts to apply the same concept in SoC or embedded SoC testing or verification. In this paper, we are proposing a verification methodology which qualifies tied-off signal values based on mutation testing.The proposed methodology discusses extraction process of tied-off values for different instances of SoC design. Different mutants are generated using inverted value of each tied-off signal per IP instance basis. Later regression is run for each IP per inverted TF signal (bit) value. It helps to list tied-off signals, whose values areinsensitive in present testbench/testcases/verification environment for each IP. This testbench infrastructure was implemented for few IP modules of SoCdesign. We haveshared implementation details, results and impactsof the proposed methodology. This helpsto detect inadequacy of testcase(s)/testbench or verification environment related to tied-off signal faults.It improves the SoC verification quality to high standard.Finally,qualified TF signals can be used to write UVM sequence for relevant IP instances.**

Keywords –SoC Verification,Functional Verification, vcd, Tied off signal, Universal Verification Methodology, SoC integration, Logic Simulation, Constant assignment, fixed assignment, RTL, UVM, IP design, IP verification, Mutation Analysis.

I. INTRODUCTION

Nowadays SoCs are having several hundred thousand tied-off(also called hard tied[1]) signals,to be tied with correct value during the SoC integration.Thesetied-off(TF) signals/portsare either tied to 1 or 0 during the SoC integration as per device architecture requirement. The signals marked as "unreachable"[5] in the toggle coverage report generated by vcs/urg[5] in functional logic simulation, can be considered as tied off signals for respective IP. Example shown in figure 1.1 and figure 1.2 .The conventional method for the verification of TF signals in industry is mostly manual [1][4], which is error prone and time consuming.Mutation based testing is very effective in software programs [2].Developer can carefullychoose the location and type of mutant, and simulate any test adequacy criteria. Such faults aredeliberately seeded into the original program by simplesyntactic changes to create a set of faulty programs calledmutants, each containing a different syntactic change[2].*Mutation analysis* is used to design new software tests and evaluate the quality of existing software tests. Each mutated version is called a *mutant.* All tests detect and reject mutants by causing the behavior of the original version to differ from the mutant. This is called *killing* the mutant[2].New tests can be designed to kill additional mutants[2].The purpose is to help the developer to write effective tests or locate weaknesses in the test data.In [3], functional SoC verification's different aspects are tried to check adequacy in testcases(s), testbench or verification environment, checker, monitor etc. In this paper we are proposing mutation analysis based methodology to justify the correctness of TF signals of any IP instances. There could be few TF signals of an IP instance in SoC. There could be several testcases to check the integration or connectivity behavior of that IP in the SoC. There is no surety that corresponding testbench, verification environment, testcasesof that IP check each TF signal's value. But ideally if tied value of a signal (say signal1) is expected to be 1, there should be corresponding testcase which would fail if it finds the signal's value 0 (of signal1) in the SoC design. But at present there is no mechanism to determine the test adequacy for TF signals in SoC design[1][3][11-12]. Late detection of TF category bugs in design cycle, enforces re-synthesis of design. If it remains undetected in SoC design, it could lead to silicon bug which is intolerable in competitive time to market scenario. To address several such challenges, we are proposing a methodology which generates the list of TF signal of an IP instance (or set of IP instances) along with the values as per SoC design. In the next step, we flip (invert) the value of each signal of that instance and generate mutated version of design. We run regression of all tests forchosen instance(s) for each TF signal inversion. Here we are introducing mutant in SoC design by flipping each TF signal and trying to find if there is killer test to detect it. This checks the adequacy of testcases of IP in SoC design. If at least one test fails for each signal inversion, then we canconclude that adequate test exists for that IP instance. Otherwise, adequate tests to be re-written for the set of TF signals for which no

ISBN: 978-1-5090-1666-2/16/$31.00 ©2016 IEEE

test fail. This methodology also discusses how repeated regression can be run without re-compiling[14] the design again and again for flipping each TF signal bit. Simulation argument[8][14] based approach is adopted in this flow. Overall, this methodology proposes a framework for mutation based testing of TF signal for different IPinstances of SoC design. We have shown how it can be implemented on few instancesof our SoC design. Finally we generate UVM[14] sequences for each IP instance for all TF signals which are qualified using this methodology. The UVM sequences are re-usable in future SoCs for re-use IPs. This methodology helps to improve quality of TF signals or constant assignments or hard tie-off signals in SoC verification and it improves productivity of the SoC verification also.

The contents of the paper are organized as follows. Section II shows brief about the generation of TF signals for any IP instance. Section III discusses analytical model. Section IV discusses about the algorithm for detecting insensitive TF signal for any IP. Section V discusses the implementation and benefits, results. Section VIprovides features of this flow. Finally, section VII concludes the paper.

II. BRIEF ABOUT GENERATION OF TF SIGNAL VALUES

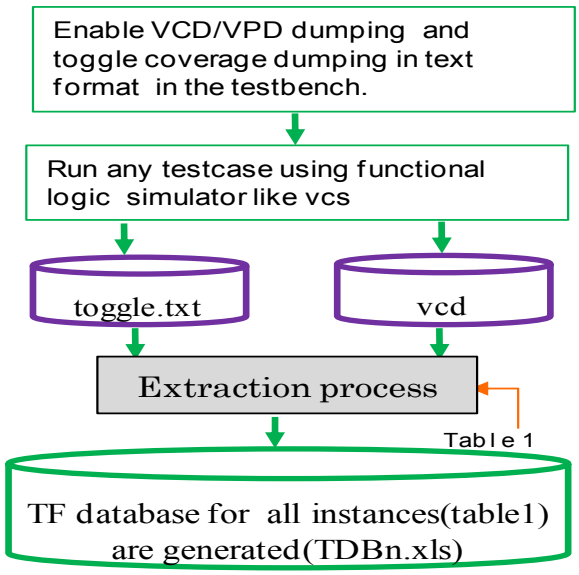

Figure 1: Shows top level flow for generation of TF signals from any instance or set of instances from any SoC design.

Table 1: Shows the sample list of hierarchies of few instances ofSoC1 [10]

SlNo	Depth	Instance Name in new SoC1
1	1	tb.top.sog_off_top.ddr_top_axi_256_wrapper
2	1	tb.top.wwof.smmu_pex
3	1	tb.top.wwof.smmu_edma

Fig1 depicts the high level flow for the generation of TF signal list of any IP instance ofSoC. Here the SoC design is simulated with any testcase using any logic simulator like VCS[5]. During the simulation the entire design is enabled for vcd[6] dumping and toggle database (toggle.txt) generation of each signal of design. At the end of simulation, these two databases(i.e vcd and toggle.txt) are generated. Sample toggle.txt for one instance is shown in figure 1.1. Here all the signals which are marked "unreachable" are called tie-off [1] signals. They are also called hard tie signals or constant assignment signals (or tied signals).We have developed an parsing/extraction script by extending standard script [13], which extract the TF signals from "toggle.txt" by searching the keyword "unreachable" [1][5] for the given instance(s). And the corresponding value of each TF signal is extracted from VCD database for the user provided instances. At the end, it dumps all the TF signals with full verilog hierarchy for the said instance(s) in xls format or text format. This database is called TDBn.xls.Figure 2 shows sample list of signal for an instance ddr_top_wrapper. If user provides a set of instances, it can extract TF signal with values for each instance. This can generate all TF signals in standard Microsoft excel (.xls) format also.

Here we have shown one method of generating list of tie-off signals. There could be other ways of generating the signals such as running user definedtcl program on ucli mode[5]or dve mode [5]. It can also be extracted using other simulators such as irun[17].

```
Toggle Coverage for Module : ddr_top_axi_256_wrapper
                Total Covered Percent
Totals          440   279     63.41
Total Bits      4326  3201    73.99
Total Bits 0->1 2163  1609    74.39
Total Bits 1->0 2163  1592    73.60

Ports          440  279  63.41
Port Bits      4326 3201 73.99
Port Bits 0->1 2163 1609 74.39
Port Bits 1->0 2163 1592 73.60

Port Details

                                                    Toggle      Toggle 1->0 Toggle 0->1 Direction
cfg_ema_access_r1_rdb[2:0]                          No          No          No          INPUT
cgu_apb_pwdata[31]                                  Yes         Yes         Yes         INPUT
ctrl_ddr_mode[1:3]                                  Yes         Yes         Yes         INPUT
ctrl_ddr_mode[0]                                    Unreachable Unreachable Unreachable INPUT
ipa_arid[9:0]                                       Yes         Yes         Yes         INPUT
ipa_arid[10]                                        No          No          No          INPUT
ipa_arid[11]                                        Unreachable Unreachable Unreachable INPUT
ipa_awaddr[38]                                      No          No          Yes         INPUT
ipa_awburst[1:0]                                    Yes         Yes         Yes         INPUT
ipa_awid[11]                                        Unreachable Unreachable Unreachable INPUT
ipa_awlen[1:0]                                      Yes         Yes         Yes         INPUT
cop_clk_wrapper_ipt_cgu_pll_protect                 Yes         Yes         Yes         INPUT
cop_top_cop_clk_wrapper_clk_ccu_ddr_mid_cyc_pd_en   Unreachable Unreachable Unreachable INPUT
ctrl_drain_queues_for_sleep                         Unreachable Unreachable Unreachable INPUT
ddrc_ccu_pll_tstclk_sel                             Yes         Yes         Yes         INPUT
ddrc_ctrl_config_wr_en                              Unreachable Unreachable Unreachable INPUT
ddrc_ipi_int_self_ref_d1                            Unreachable Unreachable Unreachable INPUT
ipg_sync_dickens                                    Unreachable Unreachable Unreachable INPUT
```

Figure 1.1: Shows sample toggle coverage data in text format for few signals of DDR controller.

Figure 1.2 shows sample tie-off signals for an IP instance. Here the signal apb_PAddr is 32 bit bus. But for this instance upper 20 bits are tie-offs with 20'h00000 values. Remaining 12 bits (apb_PAddr [11:0])are connected as normal signals. Similarly all 3 bits of apb_PProt are tie-offs with 3'h000.

```
apb2sb_wrapper  #(.BIG_ENDIAN(1),
      .APB_DATA_WIDTH(8))  apb2sb_duart1(
    .apb_PAddr({
        20'h00000,
        ccsr_noc_network_T_ccsr_duart1_PAddr
    }),
    .apb_PProt(3'b000),
    .apb_PRData(apb2sb_duart1_apb_PRData),
    .apb_PStrb({
        3'b111,
        ccsr_noc_network_T_ccsr_duart1_PWBe
    }),
    .apb_PWData(ccsr_noc_network_T_ccsr_duart1_PWData),
```

Figure 1.2: Shows sample tie-off signals of an IP instance.

```
testbench.top.ddr_top_wrapper.ctrl_ddr_mode[0]=1'b0
testbench.top.ddr_top_wrapper.ipg_sync_dickens=1'b1
```

Figure 2: Example of TF signals for the instances ddr_top_wrapper (DDR4 Controller[10])

III. Analytical Model

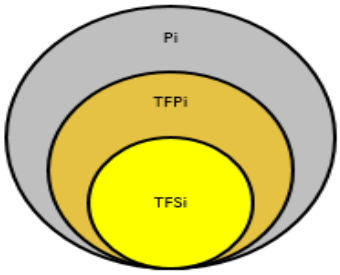

Figure 3: Shows the relation among different set of TF signals of any IP instance

Figure 3 shows venn diagram about relation of TF signals from fault detection sensitivity prospective. If flipping of any TF signal, enforces failing of testcases in logic or functional simulation, it is considered that corresponding TF signal is sensitive to testcase.Here Pi denotes total number of ports of a IPi.TFPi is total number of TFports of the IPi in SoC.TFPi can be generated by the flow as shown in figure 1 for IPi.TFSi is the total number of TF port sensitive to SoC testcase(s) or verification environment or testbench. While the value of any of theTF signal of TFSi category isinverted, SoC verification environmentreports error in logic simulation regression run.It meansthat injected TF fault(ormutant) detected by test. Total number of TF signals are not sensitive to verification environment,can be written asTFNSi= (TFPi∩ (TFSi)^c); TFPi⊆Pi; TFSi⊆TFPi; (TFSi)^c is complement of TFSi.The set of signals in TFNSi category could be problematic as their values donot enforce any testcase, testbench, VIP component and verification environment [14] to fail in functional logic simulation. More importantly, these set of signals' values are assigned during SoC integration. These values need to be qualified during SoC verification phase. If any IP instance contains any TF signal of TFNSi category that IP instance's test vector or testcase(s) needs to be re-designed for high quality SoC design verification.A SoC can have N IPs, we can list all categories of TF signals for each IPi(i=1 to N). The TFNSi category signals for IPs to be listed for any SoC design. We need to write proper testbench or testcases or monitors or checkers or cover points[11-12]for testing or verifying all TFNSi categories of signals for high quality SoC verification. In the next section, we have shown how to find TFNSi list for any IP (say IPi) in SoC design.

IV. Algorithm To Detect Insensitive TF Signal

Here the set of testcase(s) for an IP areTCi(for IPi).The set of testcases are developed for the verification of IPi in associated SoC design. Figure 4 shows the pseudo code for generating the list of TF insensitive (TFNSi) signal of IPi.Input to the proposed pseude code is TFPi for an IP instance IPi. The outputs are list of TFSi and TFNSi for that instance. RUN_ALL procedure is for running all tests (TCi) in functional simulation using logic simulator like VCS[5] or ies[17] concurrently.This can be considered as regression of all tests of IPi in SoC verification environment. CHECK_STATUS is procedure to check if at least one testcase has beenfailed due to flipping of corresponding TF signal. It returns true in that condition. Otherwise, it returns false.

Input: TFPi

Output :TFSi, TFNSi;

1.Assign : TFSi=Null, TFNSi=Null, FAIL=FALSE;

2.Foreachej∈TFPi

3.Begin

4.FLIP(ej) // Flip the bit value using sim_arg

5.RUN_ALL // Run regression for all testcase(s)

6.CHECK_STATUS(FAIL); // Check status

7.IF FAIL==TRUE

8.TFSi={TFSi∪ej)

9.ELSE

10.TFNSi={TFNSi∪ej}

11.End // end of foreach

Figure 4: Pseudo code of the extraction of TFNSi signal list of instance IPi.

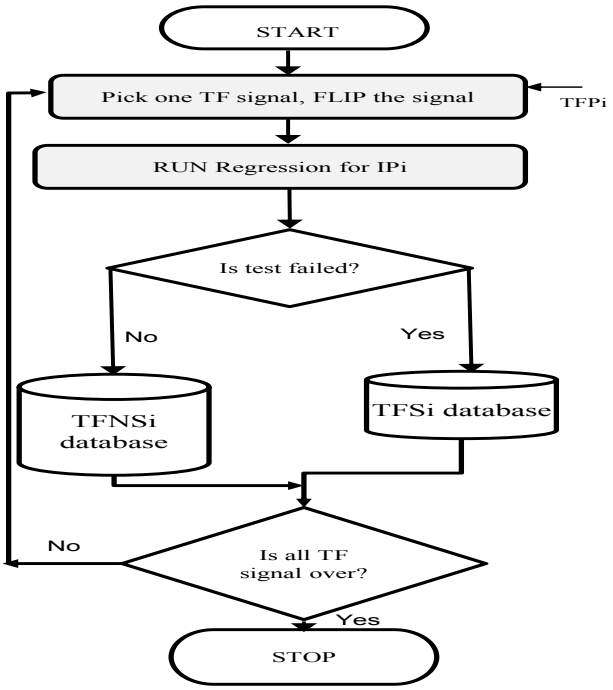

Figure 5: Flowchart for generating insensitive TF signals for any IP instance.

Figure 5 shows high level flowchart for generating set of TF signals for which testcases (TCi) do not fail, when TF value is inverted or flipped. At first, we start by picking up one TF signal at a time. Once, the TF signal is inverted, the SoC regression is run for corresponding IP (say IPi). In the next step, the status of regression run is checked. If it fails, the corresponding TF signal put in TFSi database. Otherwise, it puts into TFNSi database. If it passes, corresponding TF signal is moved to TFNSi database. Otherwise, it is put into TFSi database. In the next step, if all TF signal analysis is not done, it then moves to next TF signal and continue the process again. Otherwise, it stops and returns list of TF signals, those are insensitive (TFNSi). It also returns TFSi. In this repeated regression, simulation is re-run but compilation is done once only. It is achieved with UVM testbench infrastructure. It saves significant overall testcase run time. All regressions are run concurrently. If for any instance after executing the above qualification process, TFNSi becomes "null", the test set or test vector set of that instance to be considered as adequate. The set of test vectors are considered to be high standard as they checks/covers/verifies all TFPi signals.

Fig 5.1 explains the flowchart for generating UVM sequences for the qualified TF signals per instance basis. This can be useful for generating UVM sequences for TFPi signals of IPi also. It takes the list of TF signal with associated values and generates the UVM sequence code. This code is written manually. This can be put into body [14] task of the sequence of IPi or it can be made part of UVM package [14] also. The same can be run on each SoC regression. In future SoCs, these sequences can be directly used as UVM package and it will significantly reduce verification effort and improve quality of SoC verification for IPi. Similarly, UVM sequences can be done for all IPs of any SoC.These UVM sequences can be re-used in future SoCs for improving quality and productivity of SoC verification.

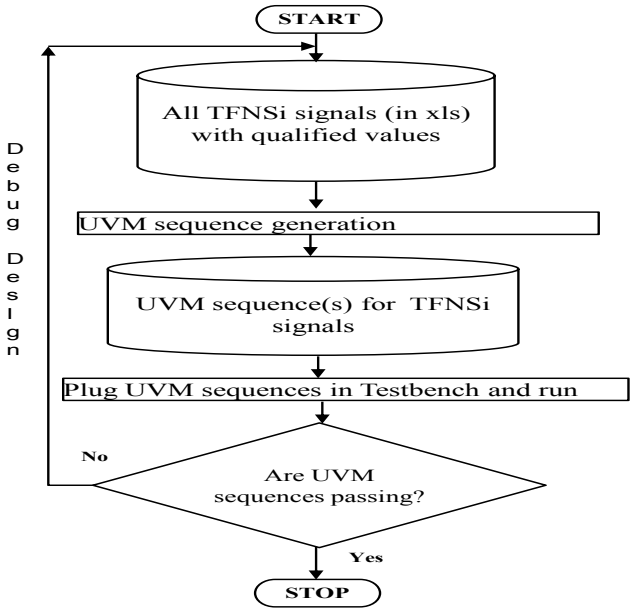

Figure 5.1: UVM sequence generation for all TF signals

V. IMPLEMENTATION AND BENEFITS

A. Implementation strategy

In SoC functional simulation, we have broadly two phases. In the first phase we do compilation of the design, testbench, VIP, verification environment, testbench

component[11-12][14]etc in one go. It is usually time consuming as design size of system on chip is very large. The second step is to run simulation of testcase. System verilog[8] or UVM[14] are most efficient ways to implement testbench or verification environment. The compilation argument as well as simulation argument can be implemented vey easily with these languages. In this flow, we implemented all the flip (inverting) bit mechanism in design using simulation argument[8][14]. For example, if an IP instance have a few TFPi signals, each signal's flipping bit can be implemented in testbench using simulation argument. For easier implementation, each signal name can be used as simulation argument. For example, assume that there are TF signals TFP1, TFP2, TFP3 and TFP4 for instance "ip_inst1" at design hierarchy testbench.top_design.ip_inst1 (say). The TF signal value of each signal extracted from design as given below:

testbench.top_design.ip_inst1.TFP1=1'b0;
testbench.top_design.ip_inst1.TFP2=1'b1;
testbench.top_design.ip_inst1.TFP3=1'b1;
testbench.top_design.ip_inst1.TFP4=1'b0;

The testbench can have some variables like tb_tfp1, tb_tfp2, tb_tfp3 and tb_tfp4. These variable can be initialized with design extracted values such as tb_tfp1=0,tb_tfp2=1,tb_tfp3=1 and tb_tfp4=0.In the testbench,following assignments are re-written as given below:
if(sim_args(TFP1), value_sim_args(TFP1,tb_tfp1) else tb_tfp1=0
assigntestbench.top_design.ip_inst1.TFP1=tb_tfp1;
if(sim_args(TFP2), value_sim_args(TFP2,tb_tfp2) else tb_tfp2=0
assigntestbench.top_design.ip_inst1.TFP2=tb_tfp2;
so on.

Thedefault values are extracted values from design. Here mutant is the flipped (or inverted) value. And mutated program is the design with only one flipped (inverted) TF bit at a time. For example, if there "P" number of TF signals for IPi, then "P" number of mutated design (mutated program) can be generated. Each version of the mutated design is tested with all tests (TCi) to check test adequacy. Fig 6 shows sample example of implementation of this flowon our design[10] for DDR4 controller instance. This is implemented in system verilog[8] /UVM[14] testbench setup.

Figure 6 shows the simulation argument based implementation of this methodology for DDR ip instance in one of our SoC. This is an example for testbench code or implementation. Here we can find the simulation argument for each TF signals for SoC. The entire testbench code can be generated manually or using some script. A simple script can reads the TFPi signal list and later can generate testbench specific code for the implementation of this methodology. We have shown here for system verilog or UVM testbench but it can be extended to other testbench or verification environment also. For example, ipa-awid_11 signal's default value in design is 1'b0. Usually normal testcases are run with the same setting. If we run simulation (regression) with +ipa_awaid_11=1 from command line, the design's value of the same signal is flipped to 1'b1. The entire simulation regression of IPi is run with this flipped value. At the end of the regression, we check status of each testcase. If any test fails due to above change, then this signal ipa_awid_11 can be considered as TFSi signal. Otherwise, ipa_awid_11 can be considered as TFNSi signal for IPi(DDR). Similarly, a set of TFNSi signals can be found out for this IP and also for other IPs.

```
if ($test$plusargs("ipa_awid_11"))        status = $value$plusargs("ipa_awid_11=%h",ipa_awid_11);
          else ipa_awid_11    = 0;
$display("Sample ipa_awid_11=%h",ipa_awid_11);
force testbench.top.ddr_top_wrapper.ipa_awid[11]  =ipa_awid_11;
if ($test$plusargs("cop_top_cop_clk_wrapper_clk_ccu_ddr_mid_cyc_pd_en"))        status = $value$plus
args("cop_top_cop_clk_wrapper_clk_ccu_ddr_mid_cyc_pd_en=%h",cop_top_cop_clk_wrapper_clk_ccu_ddr_mid_
cyc_pd_en);              else cop_top_cop_clk_wrapper_clk_ccu_ddr_mid_cyc_pd_en    = 0;
$display("Sample cop_top_cop_clk_wrapper_clk_ccu_ddr_mid_cyc_pd_en=%h",cop_top_cop_clk_wrapper_clk_c
cu_ddr_mid_cyc_pd_en);
force testbench.top.ddr_top_wrapper.cop_top_cop_clk_wrapper_clk_ccu_ddr_mid_cyc_pd_en= cop_top_cop_c
lk_wrapper_clk_ccu_ddr_mid_cyc_pd_en;
if ($test$plusargs("ctrl_drain_queues_for_sleep"))        status = $value$plusargs("ctrl_drain_queue
s_for_sleep=%h",ctrl_drain_queues_for_sleep); else ctrl_drain_queues_for_sleep = 1;
$display("Sample ctrl_drain_queues_for_sleep=%h",ctrl_drain_queues_for_sleep);
force testbench.top.ddr_top_wrapper.ctrl_drain_queues_for_sleep =ctrl_drain_queues_for_sleep;
if ($test$plusargs("ddrc_ctrl_config_wr_en"))        status = $value$plusargs("ddrc_ctrl_config_wr_e
n=%h",ddrc_ctrl_config_wr_en); else ddrc_ctrl_config_wr_en = 1;
$display("Sample ddrc_ctrl_config_wr_en=%h",ddrc_ctrl_config_wr_en);
force testbench.top.ddr_top_wrapper.ddrc_ctrl_config_wr_en = ddrc_ctrl_config_wr_en;

if ($test$plusargs("ddrc_ipi_int_self_ref_d1"))        status = $value$plusargs("ddrc_ipi_int_self_r
ef_d1=%h",ddrc_ipi_int_self_ref_d1);              else ddrc_ipi_int_self_ref_d1 = 0;
$display("Sample ddrc_ipi_int_self_ref_d1=%h",ddrc_ipi_int_self_ref_d1);
force testbench.top.ddr_top_wrapper.ddrc_ipi_int_self_ref_d1 =ddrc_ipi_int_self_ref_d1;

if ($test$plusargs("ipg_sync_dickens"))        status = $value$plusargs("ipg_sync_dickens=%h",ipg_sy
nc_dickens);              else ipg_sync_dickens = 0;
$display("Sample =ipg_sync_dickens%h",ipg_sync_dickens);
force testbench.top.ddr_top_wrapper.ipg_sync_dickens        = ipg_sync_dickens;
```

Figure 6 : Sample Implementation for DDR IP instance in SoC3[10]

B. Results and benefits

This flow is developed on DDRip instance in one of our SoC3.There are 8 TF signals are taken as an example. We had around 38 tests for this instance. We have re-run all the tests for each TF signal flipping. Here eight separate mutated version of designs were created by flipping every TF signal at a time.The regression were run on each mutated version. We have found that our tests suites are enough to detect each mutant. This qualifies that our ddrip instance is having enough check for each TF signal. At the end of the regression, we found that all TF signals in the category of TFSi(i.e TFNSi=null).

The flow was applied in another new IP instance of JESD of SoC1 and we have found two insensitiveTF signals (TFNSi)memo_pgen and mem0_ret1_b. There were no

checks in TB/verification environment for these TF signals. They were analyzed and appropriate tests were re-written.

The main benefit of this methodology is make sure that there is enough checkers/testcase(s) in testbench or verification environment for cross-checking correctness of each TF signal value for selected IP instances. The proposed methodology is very straight forward to implement and it improves quality of verification significantly.

C. Comparison with other existing technique(s)

SoC verification engineers fixes some of the TF signals' values while developing the testcases/verification environment for corresponding IP. Most of the time, the TF signal values are checked manually [1].Prior work [16] compares the TF signals value with parent SoC (working silicon) or IP database(s).But there is no exclusive check for every TF signal value in any testbench or verification environment. This is one attempt made for the qualification of verification environment or testbench/testcase for every TF signal to make sure that there are adequate tests/checkers/monitors in testbench for every TF signal of SoC.

D. Running Strategy

We run repeated regressionfor each TF signal without compiling the database. This is enabled in our simulation using VCS simulator. It saves significant overall run time. We have taken eight different TF signals of ddrip instance and run the regression. We had 38 tests for ddrip instance. We had run regression eight times but they were run concurrently. The total time = 2*8=16 hours. Here average simulation running time of each testcase is 2 hours.Here design/verification/testbench qualification is done once in the SoC design cycle. This time is ignorable. Becausewe need to do it only once for an IP instance in the entire SoC design cycle.The test suites of ddrip instance were run in approximately one day.

VI. KEY FEATURES

It processes industry standard data as input like toggle database and vcd dump. The toggle database generated by different tools like [5][17] can be used. Outputs are reported in text or Microsoft excel format(.xls). It may happen in the new SoC for few IPs, we might need delta changes in features. There will be some new IPs. We can prioritize to apply this flow for new IPs and majorly modified IPs. Because the reuse IPs are less error prone as they have been already qualified in previous SoC design or SoC silicon. As the run time is in the higher side for this qualification process, SoC verification engineers need to plan for critical IP instances to get more impactful results. Even though run time is in the higher but the qualification makes high qualityverification of SoC. Hence a trade-off planning can be done at the start of the project.

VII. CONCLUSION AND FUTURE WORK

This flow is enabled for a fewIP instances. The flow brining up effort is minimal and reusable across SoC design. We can see significant quality improvement if this can be enabled for critical IPs (or all IPs) in SoC design. The testbench or verification environmentchanges for implementing this flow, are generic in nature and can be attempt in any SoC design. It improvesquality ofTF signal verification significantly.In future, we are exploring for formal or equivalence checking based methodology. We are also planning to make up this flow for all other critical protocols interface of SoC.

REFERENCES

[1] P.Ghosh,S.Ghosh,P.Singh,S.Mishra,"Case Study:Revisiting SoC Verification Challenges and Best Practices" Procs of 19thVDAT, IEEE conf. , June-2015

[2] Y. Jia, M. Harman,"An Analysis and Survey of the Development of Mutation Testing", IEEE transactions on software engineering, Vol:37, No.5, Sept/Oct, 2011.

[3] K. Huang,P.Jhu,R.Yan,X.Yan, "Functional Testbench Qualification by Mutation Analysis", Hindawi Publishing Corporation, VLSI Design, Volume 2015, Article ID 256474, 9 pages, http://dx.doi.org/10.1155/2015/256474, 2015.

[4] D. Murray, S. Rance, "Leveraging IP-XACT standardized IP interfaces for rapid IP integration", White Paper, ARM Inc, 2015.

[5] VCS tool, URG tool(for toggle coverage report in text format), DVE, ucli (version:J-2014.12-SP3-4) details available at :http://www.synopsys.com/

[6] VCD description is available in IEEE Std 1364-2001 at www.ieee.org

[7] B3421 SoCdescription&reference manual (SoC1):http://www.nxp.com/

[8] System verilog standard: IEEE 1800-2013, available in www.ieee.org,

[9] LS1043 SoC description, documentation andRM(SoC2): http://www.nxp.com/products/microcontrollers-and-processors/arm-processors/qoriq-arm-processors/qoriq-ls1043a-and-ls1023a-multicore-communications-processors:LS1043A?

[10] LS1046SoC documentation, RM (SoC3):http://www.nxp.com/products/microcontrollers-and-processors/arm-processors/qoriq-arm-processors/qoriq-ls1046a-and-ls1026a-multicore-communications-processors:LS1046A?

[11] P. Rashinkar, P. Patterson, L. Singh, "System on a Chip : Verification Methodology and Technique", Springer publication, 2002

[12] J.Bergeron, "Writing Testbenches: Functional Verification of HDL Models", Springer publication, 2012.

[13] Parse vcd: www.perl.org , http://search.cpan.org/~gsullivan/Verilog-VCD-0.03/lib/Verilog/VCD.pm

[14] UVM documents and manual(IEEE 1800.2): accellera.org or ieee.org

[15] SoC Integration and Verification tools/methodologies: http://www.synopsys.com/Services/SoCDesign/Pages/default.aspx

[16] P.Ghosh,S.Ghosh,R.Srinivas "A Framework for Verification of Hard Tied Signals of SoC", 16thIEEE Workshop on RTL and High level Testing(WRTLT),(24thATS),http://www.ieee-wrtlt.org/, IIT Bombay, India,2015

[17] IUS/IES/IMC/irun/ncvlog tools : www.cadence.com.