An Efficient Verification Methodology to Achieve DV Sign-off with Emphasis on Quality and Quicker DV Cycle

Piyush Agnihotri (piyush.a@samsung.com), Nirmal Kumar (nirmal.k@samsung.com), Arnab Ghosh (arnab.ghosh@samsung.com), Parag S Lonkar (parag.lonkar@samsung.com)
Samsung Semiconductor India Research (SSIR), Bengaluru, India.

Abstract – **The semiconductor industry has seen tremendous growth in past few decades but in recent times requirements like higher computation, high bandwidth and quick time to market have put each domain under tremendous pressure to be more efficient while maintaining strict focus on quality. In this aspect the task of both design and verification engineers becomes very critical as onus of time to market as well as quality lies very much with them. This paper deals with this situation from the perspective of a verification engineer and suggests the process to optimize the entire DV signoff using some methodology enhancements and scripting.**

I. Background & Introduction

The design verification flow starts with Test-bench (TB) bring up along with RTL integration, followed by various phases of verification like introducing checkers, test sequences, debug, regression, reporting and finally signing off with aligned metrics which is coverage. Each phase has its own significance and challenges but is equally very critical when it comes to confidence on IP quality. One important phase which is usually missed or lesser valued is Quality Checks post DV completion. In the subsequent sections we have tried to touch upon most of the above phases along with QA (Quality Audit) metrics while adding focus on automation and scripting.

II. Related Work & Application

In this paper, a quick and efficient verification flow is proposed for attaining DV closure in smaller time as compared to standard flow. The flow starts with the automated generation of required TB files starting from RTL-TB Top integration using interface signals, creating generic checkers, coverage, external VIP connections and basic sequence files using a 'Python' script.

The next challenging phase is debugging the failures, where it becomes even more difficult for long running simulations. While debugging failures where RTL or TB updates needed, the updated results can be achieved faster using S-R technique as it can save the snapshot till desired timestamp. To reduce the efforts in coverage closure, we propose a combination of ranked VSIF (Verification Session Input File) and a Perl script which is capable of producing directed test-cases based on the information of coverage holes.

At the last phase, we perform Test-bench Quality Analysis using a Perl script having various checks, which ensures whether the test-bench is created with proper coding guidelines and standards, disk space optimisation and required DV metrics are met without any compromise.

A. DV Phase 1: Integration Phase

Automated TB file generation:

The automated TB file generation is achieved using a python script, which takes the top RTL module name and RTL file as inputs and it provides RTL and TB top integration, configuration parameters based on input signals of DUT, coverage bins, X/Z checkers, clock monitor checks, clock management unit, interface connections with external VIP interfaces.

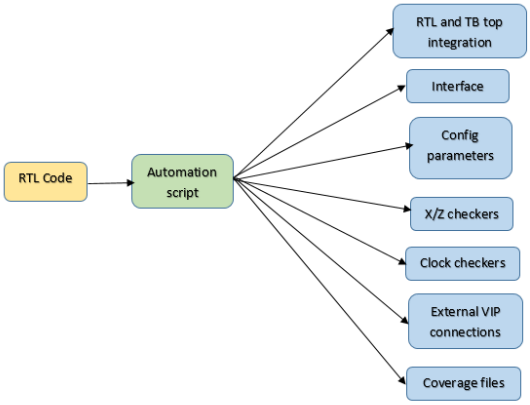

Figure 1. Files generated from automation script

The script takes a xls sheet (shown in Fig-2) as input which has the following details like RTL path, top module name, options to choose whether the user wants to generate files such as DUT instantiation with interface signal connections, interface file, config file, coverage file, xz checkers, clock monitors, clock management unit files. The other input to the script is the RTL top file (shown in Fig-3). In Figure .2, the user has selected option "Yes" for interface, x_z_checkers, config file, clock monitor files. So the script will generate the necessary files as per user request. During the script run, a few intermediate xls will appear to set the initial values of the interface signals, creating bins for coverpoints in a covergroup. The user can fill the xls as optional which will be reflected in the final output.

Type	Value
Dut_path	AHB.v
Dut_module	AHB
Sub_name	
Interface	Yes
Clock_gen	No
Config	Yes
Coverage	No
Basic_Sequence	No
X_Z_checker	Yes
APB_Connection	No
APB_Count	1
APB_VIP	cdns
AHB_Connection	No
AHB_Count	1
Vbuilder_top_file	top_temp.sv
CMU_Enable	No
Verilog_tb_enable	No
Verilog_files_path	input_verilog_modules
Vbuilder_tb_file	tb_temp.sv
Vbuilder_vseqr_file	vseqr_temp.sv
Clock_Checker Enable	Yes

Figure 2. Input XLS to script

```
module AHB    (
  input  wire                    HCLK,
  input  wire                    HRESETn,
  input  wire                    i_PCLKEN,

  // AHB Master Interface
  input  wire                    i_HSEL,
  input  wire [ADDRWIDTH-1:0]    i_HADDR,
  input  wire           [1:0]    i_HTRANS,
  input  wire           [2:0]    i_HSIZE,
  input  wire           [3:0]    i_HPROT,
  input  wire                    i_HWRITE,
  input  wire                    i_HREADY,
  input  wire          [31:0]    i_HWDATA,
  input  wire           [2:0]    i_HBURST,
  input  wire                    i_HMASTLOCK,
```

Figure 3. Input RTL file

a.) RTL and TB integration:

The script automatically integrates RTL and TB top using interface signals. It creates configuration parameters on DUT inputs and coverage bins on DV engineer inputs. It creates interface connections between RTL and external VIP interfaces like APB.

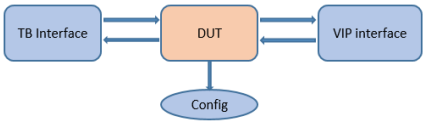

Figure 4. Block diagram of RTL and TB integration

The snippet of interface output file and DUT instantiation file is shown in Fig 5 and Fig 6 respectively. The script generated files can be placed in respective folders in user's workspace as per the need. The integration is done and the DV engineer can start to focus on sequence and checkers.

```
interface dut_top_if();
logic           i_hclk;
logic           i_hresetn;
logic           i_pclken;
logic           io_hsel;
logic   [`ADDRWIDTH-1:0] i_haddr;
logic           o_hreadyout;
logic   [31:0]  o_hrdata;
```

Figure 5. Output Interface File

```
AHB u_dut(
                .HCLK           (tb_dut_if.i_hclk),
                .HRESETn        (tb_dut_if.i_hresetn),
                .i_PCLKEN       (tb_dut_if.i_pclken),
                .i_HSEL         (tb_dut_if.io_hsel),
                .i_HADDR        (tb_dut_if.i_haddr),
                .i_HTRANS       (tb_dut_if.i_htrans),
                .i_HSIZE        (tb_dut_if.i_hsize),
                .i_HPROT        (tb_dut_if.i_hprot),
```

Figure 6. Dut instantiation file

b) X/Z checkers: X/Z checkers are automatically generated on output and input signals of major interfaces. The XZ checker is very helpful in finding the occurrence of any x or z on any of the input or output ports of a module which are undesirable. The XZ checker module snippet and the error report generated by xz checker module is shown in Fig-7 and Fig-8 respectively.

```
`define X_Z_CHKR(SIG , EN , GLOB_EN , STR)\
always@(SIG or EN or GLOB_EN) begin\
  if(EN && GLOB_EN) begin\
    if(($isunknown(SIG))) begin \
      `uvm_error("X_Z_CHKR" , $sformatf("FAIL SIG is X or Z"))\
      `uvm_error("X_Z_CHKR" , $sformatf(" FAIL : SIG_VAL :%0d . SIG_NAME : %s" , SIG , STR))\
    end \
    end\
end
```

Figure 7. XZ checker snippet

```
[0] UVM_ERROR (X_Z_CHKR) FAIL SIG is X or Z
[0] UVM_ERROR (X_Z_CHKR)  FAIL : SIG_VAL :z . SIG_NAME : top.u_dut.i_HSEL
```

Figure 8. Error report by XZ checker

c) Clock monitors: The script integrates DUT clock signals to different clock checkers like phase checkers, duty cycle checkers. The clock checker helps in checking the period time (high time and low time) of the clock period. It also provides checks for clock division, as it checks whether the output clock is exactly a divided version of the input clock by a factor of input value n. (ie : output_clock_frequency = input_clock_frequency/n) . The code snippet of clock monitor and error report generated by clock monitor is shown in Fig-9 and Fig – 10 respectively.

```
if(clk_cfg.tolerance_en[i])
  begin
    if( (t_period < (clk_cfg.clk_period_value_arr[i] - clk_cfg.clk_tolerance_value_arr[i])) || (t_period > (clk_cfg.clk_period_value_arr[i] + clk_cfg.clk_tolerance_value_arr[i]) ))
    begin
        `uvm_error("Clock Period with tolerance  Mismatch", $sformatf("Clock clk_in[%0d] : Expected Clock Period Value = %0f , tolerance = %0f, Actual Clock Period Value  = %0f", i, clk_cf
ce_value_arr[i], t_period ))
    end
  end
else if(rndoff_cfg_period != rndoff_actual_period)
  `uvm_error("Clock Period Mismatch", $sformatf("Clock clk_in[%0d] : Expected Clock Period Value = %0.6f, Actual Clock Period Value  = %0.6f", i, rndoff_cfg_period, rndoff_actual_period ))
 end //}
```

Figure 9. Clock checker code snippet

```
[84936] UVM_ERROR (Clock Period Mismatch) Clock clk_in[0] : Expected Clock Period Value = 45.043692, Actual Clock Period Value  = 0.000000  , tolerance = 0
[84951] UVM_ERROR (Clock Period Mismatch) Clock clk_in[0] : Expected Clock Period Value = 45.043692 , Actual Clock Period Value  = 15.004530 , tolerance = 0
```

Figure 10. Error report by clock monitor

B. DV Phase 2: Connection Checks & Failures Debug Phase

1) Input/output Connectivity script:

In High Speed PHY IPs, there are many interface boundaries like Analog – Digital boundaries in PMA. The manual effort to code large number of signals for connectivity check becomes cumbersome. To avoid this, we make use of a Perl script, which extracts the source and destination signals based on design input and creates a CSV. The csv file is given as input to sequence to perform I/O connectivity check.

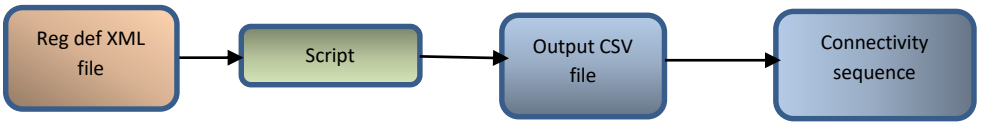

Figure 11. I/O Connectivity check flow

2) Save-Restore (S-R) Technique [EDA Feature]:

We make use of Save-Restore technique, where the common simulation task with identical configuration and loaded at zero timestamp reducing the simulation time to a great extent when we re-run with updates in RTL or TB. Regressions time also gets reduced significantly as initialization sequence which takes major time is not needed to run for all sims once already verified.

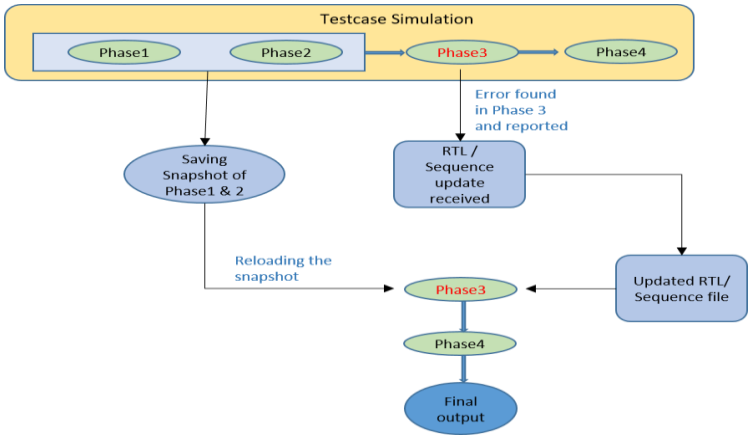

Figure 12. Debugging Failures and Rerun Using S-R Technique

Saving the snapshot means saving memory, file pointers etc. When the “process_save” option is added at elaboration time then “$save” system task is available for use in Verilog code through which the snapshot can

be saved. User can call $save after the completion of the init phase or wherever required. Below are the snapshot samples for reference:

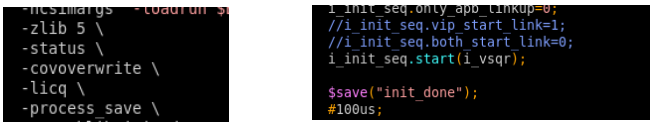

Figure 13. Saving the snapshot of init sequence using $save

Once the snapshot is saved after sim finished, the user can restore the snapshot for next simulations using uvm factory override. Below are the snapshots for reference where the init sequence which was saved as shown above restored to save the initialization sequence and how the log file for next simulation looks like:

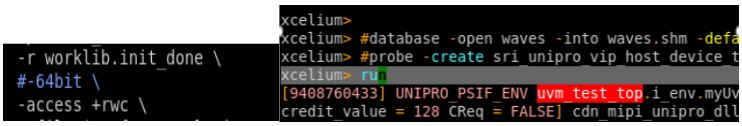

Figure 14. Reloading the snapshot of init sequence (Linkup of UniPro IP)

As you can see the log file snap, the sim got started from the previous saved snapshot (time_stamp = 9408760ns). This utility is found very useful while debugging and where the user needs to see the results fast after changing something tb other than saved snapshot phase.

C. DV Phase 3: Coverage Closure Phase

Smart and Fast Coverage Closure using Ranked Vsif & Scripting:

The Automated TB generation script (discussed above in phase1) provides us coverage file as per data provided by in XLS sheet filled by DV engineer. The Script created by us helps in generating the directed test vectors to cover the un-hit bins after fetching information from coverage database about the coverage holes. The Combination of scripting and ranked vsif helps in faster coverage closure.

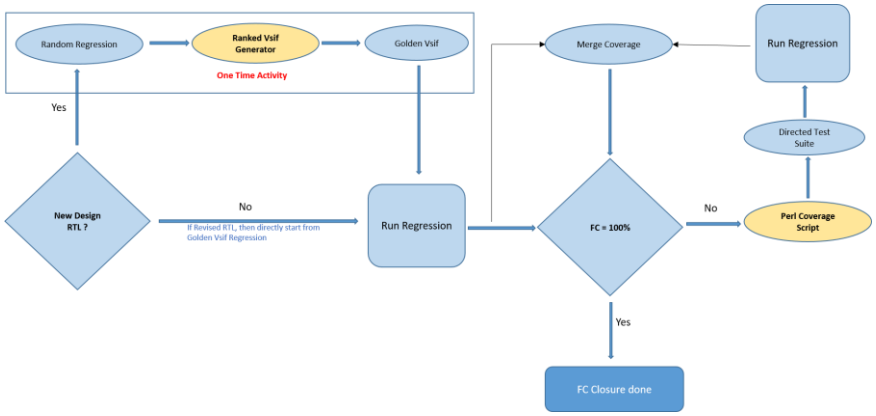

Figure 15. Smart Coverage Closure Flow

The main function of the coverage hole script agent is to obtain the latest coverage database, extract the information of un-hit bins and create directed test suite to hit the remaining bins with single run. The coverage ranking script ranks the regression runs based on the coverage contribution and creates a test suite to target the obtained coverage. This comes very handy during not so uncommon cases of multiple RTL releases, where verification team has to iteratively rush for quick coverage closures

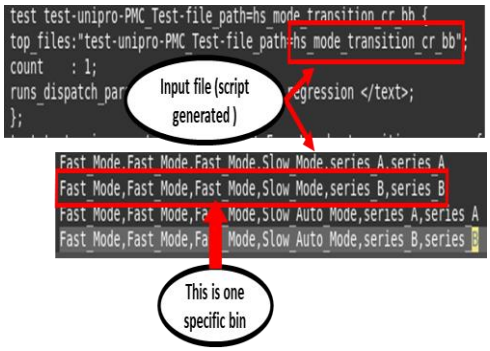

Figure 16. Input file generated by Coverage Holes Script (Ref: UniPro)

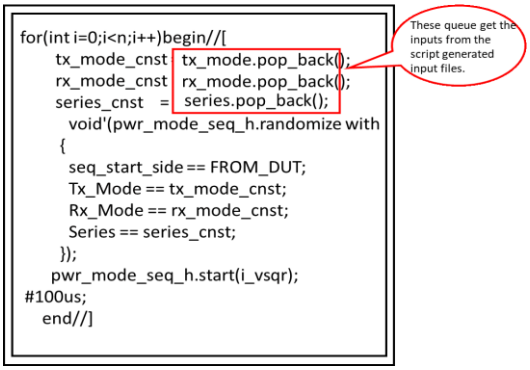

Figure 17. Example code of Power Mode Change (PMC)

D. DV Phase 4: Final Signoff & QA Phase

Test-bench Quality Analysis:

Once all the above mentioned DV phase are done and required signoff criterion is achieved, here we have introduced a next phase which does Quality Assurance on all DV deliverables and signoff metrics. This is achieved using a set of scripts which analyse and ensures the metrics listed below are thoroughly checked: RTL and Test-bench are latest ones based on label, register definitions files are latest, sanity test environment is clean, all reported issues are attended/closed, regressions are clean, functional and code coverage are 100%, tool versions are as per expectations, any force statements are used in TB, unconnected ports at different RTL interfaces, optimization of disk space by listing the workspace details using space more than threshold for all users.

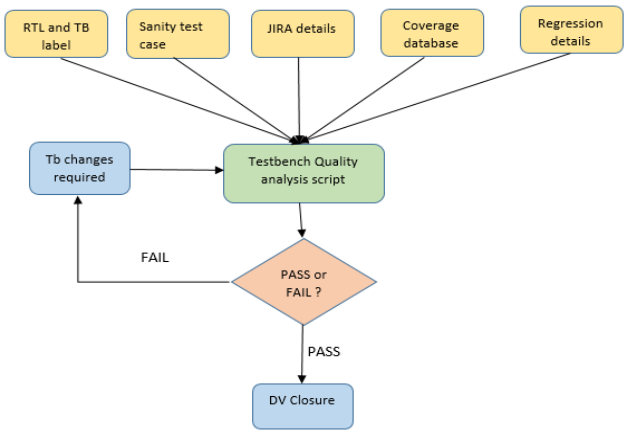

Figure 18. Test-bench Quality Analysis Flow

The script takes a xls sheet (shown in Fig. 19) as input which has the following details like IP name, RTL and Test bench latest label, Register definition files (if any), JIRA details, sanity test case command, Functional and Code coverage numbers, regression status. The script will initially sync the test bench with the latest label mentioned in input xls to ensure that all RTL and Tb files are updated. The script performs parallel tasks like coverage number check (which has to be 100%), sanity test run, regression status check (all tests should pass), JIRA details check (all JIRA has to be closed), usage of force statements check. Apart from this it performs Disk space optimisation where the script calculates the space occupied by different user in that particular project. Based on the threshold size, the script will decide whether a particular workspace has to be deleted or not in order to pass the Quality Analysis test. All the above checks are printed in separate text files and the overall analysis report is represented in the form of a HTML report (shown in Fig .20). The HTML report will have the hyperlink to all the result text files. The user can click on each link to view the result of each metric. The Disk space optimisation result is shown in Fig-21.

Label Information	Feature Availability	Yes
	Path to config file	<<config_file>>
	Depot Path – RTL	<<RTL depot_path>>
	Corresponding Label/Changelist	12345
	Path to XML/XLS file in RTL depot	sfr.xls
	Path to reg_def file (Testbench Version)	reg_Def.sv
	Vbuilder Version	342
	Depot Path - Test Bench	<<TB depot path>>
	Corresponding Label/Changelist	12345
Label Sanity	Feature Availability	Yes
	Build Name	<<build>>
	RTL Compile Command	<<rtl_compile>>
	VIP Compile Command	<<tb_compile>>
	Test Bench Elaboration Command	<<run_test()>>
	Sanity Test Simulation Command	
JIRA	Feature Availability	Yes
	JIRA Component Name of the project	<ABC>
	Path to JIRA Excel Sheet	ABC_JIRA.xls
Functional Coverage	Feature Availability	Yes
	Path to coverage report generated by CR without refines	functional_coverage_without_refine.txt
	Path to coverage report generated by CR with refines	functional_coverage_with_refine.txt

Figure 19. Input xls sheet to Quality Analysis script

LABEL SANITY	STATUS	PASS	
	Build Name	build	PASS
	RTL Compile Command	rtl_compile	PASS
	VIP Compile Command		
	Test Bench Elaboration Command	TB_compile	PASS
	Sanity Test Simulation Command	run_test()	PASS
	PROFILING		YES
JIRA	Unresolved JIRA/s	No	
	JIRA Component Name of the project	ABC	
	Path to JIRA Excel Sheet	ABC_JIRA.xls	
FUNCTIONAL COVERAGE	FC(%)	100.00%	
	Refine File used for FC	No	
	Path to Coverage without refines	functional_coverage_without_refine.txt	
	Path to Coverage with refines	functional_coverage_with_refine.txt	

Figure 20. Quality Analysis HTML report

```
<1S WS_NAME>          <1s WS SIZE COMBINING ALL LABELS>       <1S WS TO BE DELETED>
user_1                   202.076M                                NO
user_2                   28K                                     NO
user_3                   310M                                    NO

Total Number of Workspaces : 3
Workspace with red flags (to be cleaned up) : 0
Total size usage of 1S disk : 0.512104 G
```

Figure 21. Disk space optimization result

III. Results & Conclusion

With the help of Automated TB generation script, we were able to reduce the TB files creation time to a large extent. Here are some of the results based on number of DUT ports.

TABLE I
RESULTS OF BRING UP PHASE

DUT Ports Count	Time Taken (Manual Coding)	Time Taken (Automation script + DV engineer inputs)
50	1 day to 2 days	Less than 30 mins
200	2 days to 4 days	Less than an hour

Below results are shown for different test vector's suites debug-fix time for a high speed PHY IP.

TABLE II
RESULTS OF DEBUGGING PHASE

Test Vectors Scenarios	Without Save-Restore	With Save - Restore
Bring-up Related Failure	60 – 80 mins	60 – 80 mins
Basic Data-path Failure	6 hrs – 12 hrs	3 hrs – 6 hrs
Hibern8/Line Reset/POR failures	1 day – 3 days	12 hrs – 1.5 days

Below results are shown for time taken to achieve coverage closure from initial bring up phase for UniPro 2.0 LINK IP.

TABLE III
RESULTS OF COVERAGE CLOSURE PHASE

Milestone	Without Ranked vsif and scripting approach	With Ranked vsif and scripting approach
Final Coverage Closure (100% FC)	7 weeks – 10 weeks	4 weeks – 6 weeks

Conclusion:

The issues like usage of unnecessary force statements in TB files, registers definition mismatch, tool versions mismatch, RTL or TB database mismatch, unconnected ports at different RTL interfaces, high disk space usage etc. were detected in QA phase by using QA script.

This paper summarized the techniques and benefits of using scripting and EDA features during different phases of DV flow. The repetitive tasks of DV engineers can be reduced to a great extent and they can focus on other critical analysis and testing. The results also suggest the amount of time saved by the engineer while ensuring a quality DV closure for any complex IP design.

References:

[1] https://perldoc.perl.org/ , Perl 5.38.0 Documentation
[2] https://docs.python.org/3/library , The Python Standard Library