Expediting SoC Design Verification Closure by Accelerating Gate Level Simulations using Streamlined Smart Decentralized Testbench

Harshal Kothari, Staff Engineer, Samsung Semiconductor India Research, Bengaluru, India (harshal.k1@samsung.com)

Eldin Ben Jacob, Staff Engineer, Samsung Semiconductor India Research, Bengaluru, India (eldin.jacob@samsung.com)

Chandrachud Murali, Staff Engineer, Samsung Semiconductor India Research, Bengaluru, India (chandru.m@samsung.com)

Sriram Kazhiyur Sounderrajan, Associate Director, Samsung Semiconductor India Research, Bengaluru, India (sriram.k.s@samsung.com)

Somasunder Kattepura Sreenath, Director, Samsung Semiconductor India Research, Bengaluru, India (soma.ks@samsung.com)

Abstract*- Gate level simulations (GLS) are an integral part of the ASIC verification cycle to verify the impact of synthesis, insertion of power elements and placement & routing by running functional datapaths using dynamic simulations. It is imperative to run timing gate level and power aware simulations with Standard Delay Format (SDF) back-annotation to validate that the hardware designs function as intended in the best and the worst PVT corner delays on the post PnR netlists. This however, comes with a trade-off of huge simulation times, especially when run at a full chip netlist level. Streamlined Smart Decentralized Testbench (SSDT) addresses these vulnerabilities with its robust and flexible architecture. The testbench environment is self-aware and capable of auditing the logs, failures and test plan status, reporting all issues, timing violations and intelligently identifying erroneous environment which will need reruns at later stages. This paper highlights how SSDT, a holistic approach to expedite gate level simulation, resulted in achieving 100% pass rate closure well ahead of metal tapeout. Audits and automations result in early capture of environment issues even ahead of a simulation being run. Smart modular hybrid testbench setup can be reused across all flavours of simulations - unit delay GLS, timing GLS, unit delay power aware GLS and timing power aware GLS across projects. Furthermore, with Dynamic Save and Restore (DSNR), Capture and Replay (CNR) and LSF optimizations integrated across simulation flavours, SSDT which is deployed on AR/VR SoC and HPC 3DIC SoC delivered up to 65% savings on simulation time and 35% less resources without compromising on quality.

I. Introduction

SSDT employs a few techniques, which are a mix of traditional, innovative and novel processes to expedite and automate the GLS cycle (Figure 1). The traditional techniques are common testbench for different flavours of DUTs resulting in environment variable based version control for RTL/PRENET/POSTNET/PGNET/SDF, on the fly hybrid elaboration and simulation and timing violation parser. GLS audit of elaboration and simulation logs, test plan status audit, label on label timing violation mapper employ a mix of traditional and novel methods. The innovative and novel methods involve improving simulation time by optimizing environment on which the simulation is run and accelerating the tool and compute along with scaling up the methodologies and deploying them across simulation flavours. All these methodologies work in tandem under the SSDT hood.

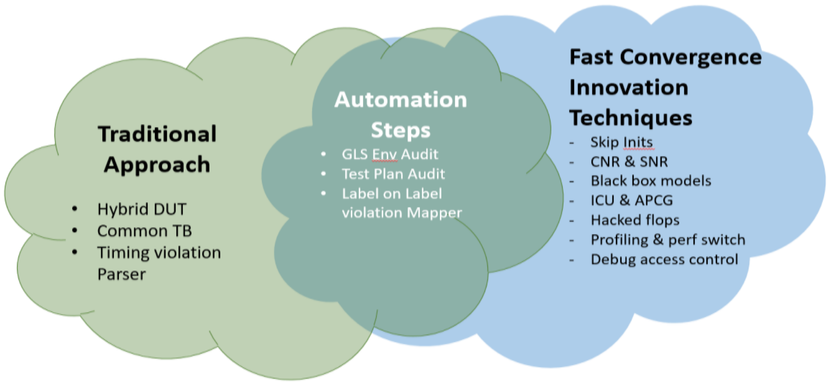
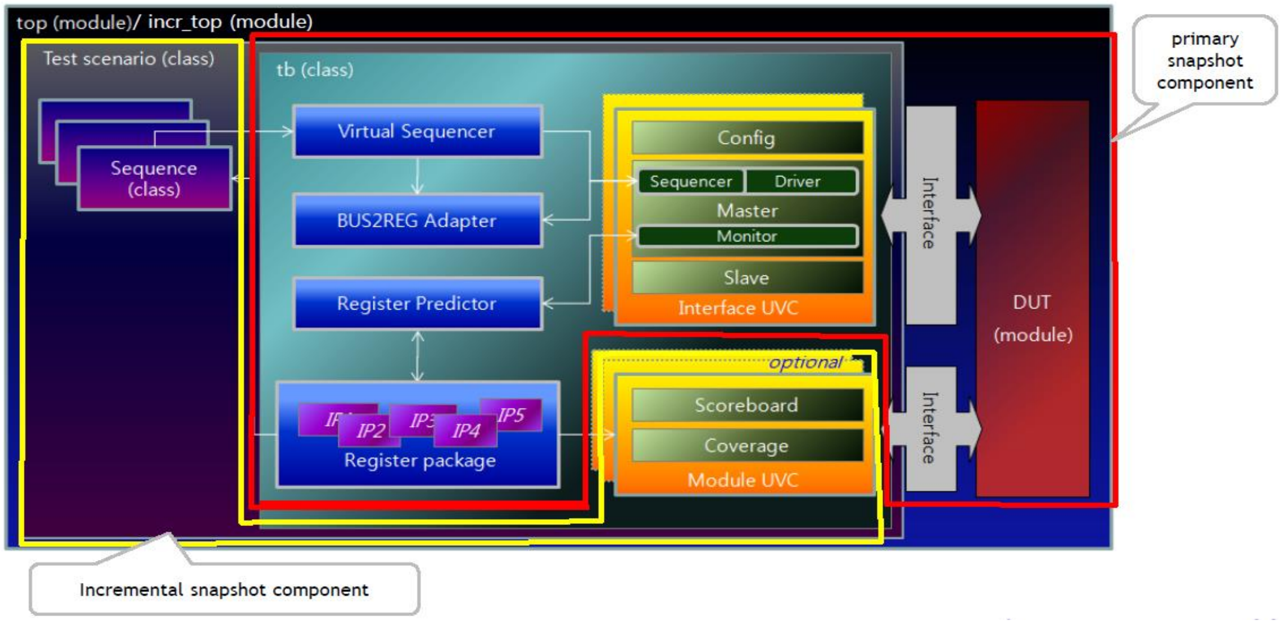


Figure 1. SSDT Implementation Strategy

II. SSDT Traditional Techniques

Common TB: To develop the common testbench for different flavoured DUTs, the testbench is architected on UVM that provides high standardization with respect to skeleton architecture, file/directory structure, naming rule, Verification IP (VIP) configuration, register model and general access sequences. The environment (Figure 2) is scalable to any IP/Sub-system/SoC that bolsters the reusability aspect or different flavours of DUTs (RTL/PRENET/POSTNET/PGNET). An internal tool utility dumps out this skeleton testbench based on excel input containing top level DUT spec. The power intent of the SOC is defined using UPF2.0/3.0 and liberty files for standard library cells and hard macros. The testbench was developed by compiling the power information for the design by loading UPF files containing design power information and the liberty files containing power information for library cells. The elaboration tool requires the appropriate IEEE power defines and tool dependent power defines. The testbench was architected to mimic the behavior of the PMIC. The SOC power up/down was achieved by simulating the power balls at the DUT top using a System Verilog model. The power ports are driven with a combination of $supply_on/off system tasks and CPU firmware masters which controls the power domains of the IPs based on the use case by the Power manager IP. To reinforce the through verification, additional checkers for isolation strategies, level shifter strategies, retention strategies, power switches of the power domains and power state table were developed. While simulating timing enabled power aware gate level netlists, block wise SDF files in min (ffpg) and max (sspg) corners are passed at elaboration stage via -sdf_cmd_file option.



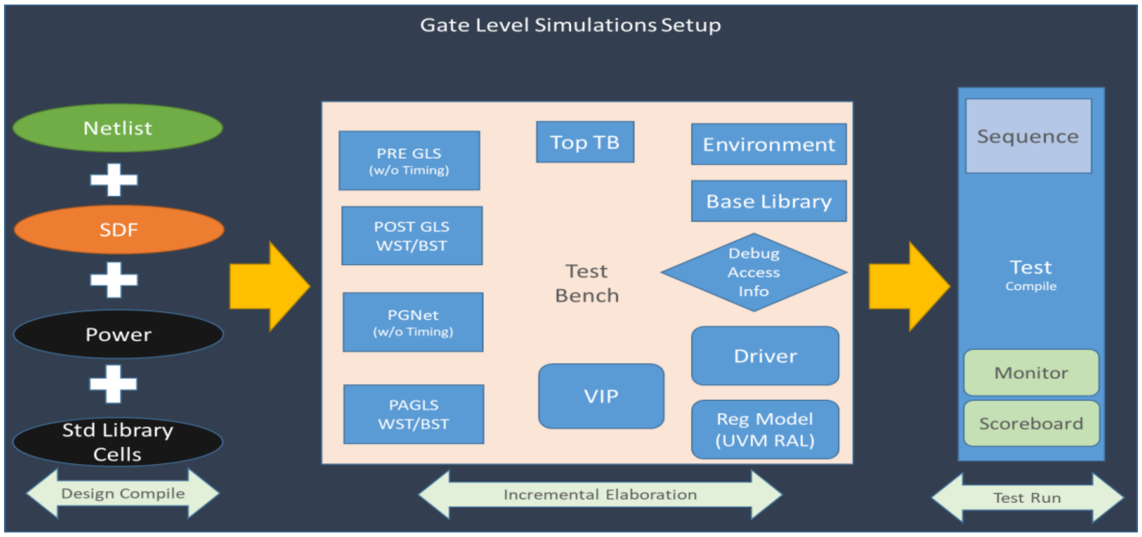


Figure 2. Traditional SSDT testbench architecture

Hybrid DUT: Traditionally, by replacing non-targeted block netlist with black-box or RTL and employing fast boot using forces, time saving of around 6-10% can be achieved by bringing down the gate count. Apart from the simulation time, the hybrid verification method (Figure 3) helps in saving precious engineer's time due to the reduced volume of errors originating from a block/IP which is not the point of interest. PERL macro based configuration offers the flexibility to pick simulator options [1], switches, the model type (RTL/Netlist/Fake) and the DUT version of each block on the fly for elaboration/simulation. The combinations of configuration results in creation of a unique Design Environment (DE) and Verification Environment (VE). The environment also supports the combinational usage of DE/VEs resulting in faster simulation of cross functional datapaths. Using a hybrid DUT for simulation helps save tremendous amount of run time when a functional datapath does not traverse all the blocks. For example, when running a camera stream datapath to DRAM, only the blocks containing CPU, DPHY, Camera Serial Interface, system network fabric and memory controller are needed as netlists. Remaining blocks can either be stubbed out by fake blackbox driving a known output or RTL. Similarly, for 3DIC, if top die is running DMA transfers to local SRAM, bottom die is stubbed out to save half the total gate count. The multi-step incremental elaboration and simulation flow eliminates recompile of DUT due to VE changes.

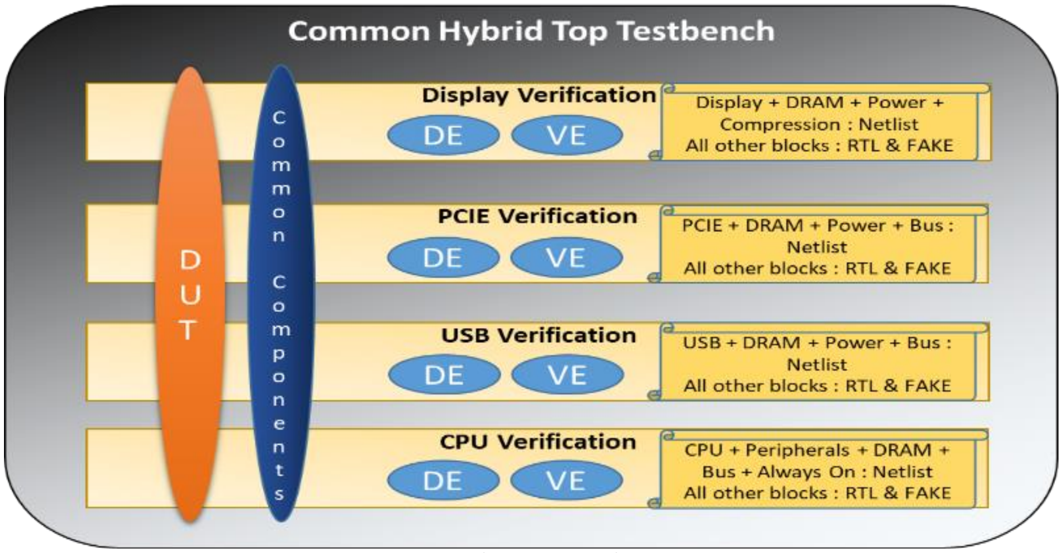
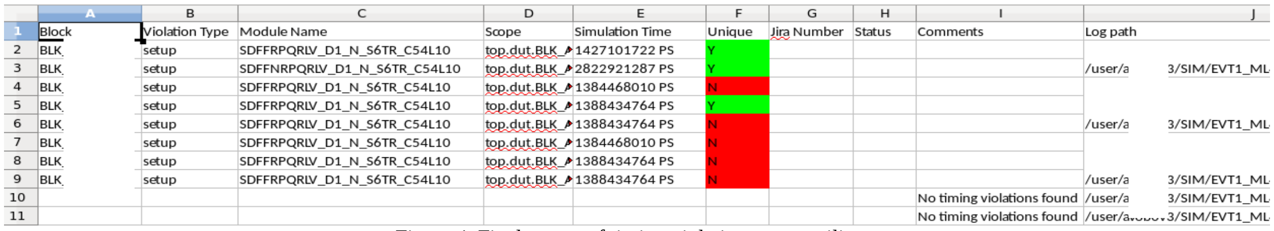

Common Hybrid Top Testbench
DUT
Common Components
Display Verification
DE
VE
Display + DRAM + Power + Compression : Netlist
All other blocks : RTL & FAKE
PCIE Verification
DE
VE
PCIE + DRAM + Power + Bus : Netlist
All other blocks : RTL & FAKE
USB Verification
DE
VE
USB + DRAM + Power + Bus : Netlist
All other blocks : RTL & FAKE
CPU Verification
DE
VE
CPU + Peripherals + DRAM + Bus + Always On : Netlist
All other blocks : RTL & FAKE

Figure 3. IP wise on the fly Hybrid Verification Environment

Violation Parser: Timing GLS simulation aids in identifying all hold/setup/recovery/removal violations of the DUT. The violation parser utility comprises of a set of Python scripts with artificial intelligence to map the violation type and reports the timing violation analysis using machine learning with in depth details about the block, violation type, scope, violation time in the run at the end of the simulation automatically (Figure 4). It can also be run on multiple simulations or regression in parallel.

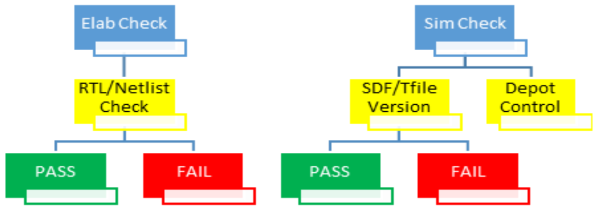


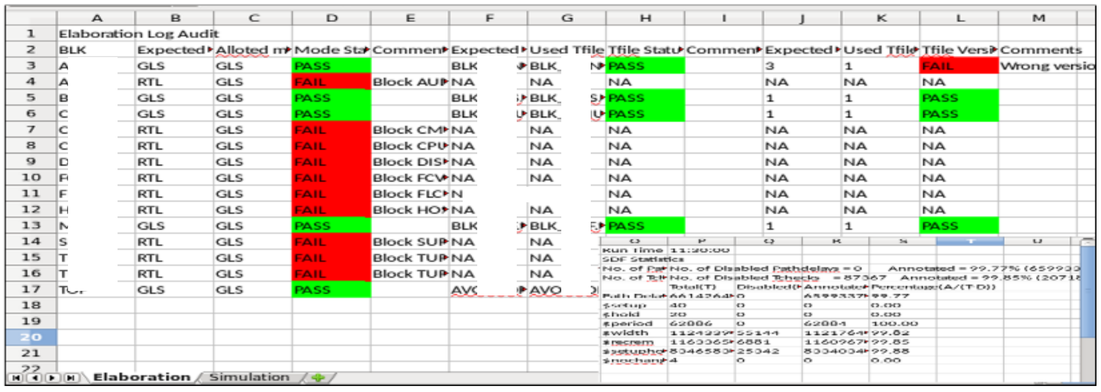
	A	B	C	D	E	F	G	H	I	J
1	Block	Violation Type	Module Name	Scope	Simulation Time	Unique	Jira Number	Status	Comments	Log path
2	BLK	setup	SDFFRPQRLV_D1_N_S6TR_C54L10	top.dut.BLK_A	1427101722 PS	🟩 Y				
3	BLK	setup	SDFFNRPQRLV_D1_N_S6TR_C54L10	top.dut.BLK_A	2822921287 PS	🟩 Y				/user/a 3/SIM/EVT1_ML
4	BLK	setup	SDFFRPQRLV_D1_N_S6TR_C54L10	top.dut.BLK_A	1384468010 PS	🟥 N				
5	BLK	setup	SDFFRPQRLV_D1_N_S6TR_C54L10	top.dut.BLK_A	1388434764 PS	🟩 Y				
6	BLK	setup	SDFFRPQRLV_D1_N_S6TR_C54L10	top.dut.BLK_A	1388434764 PS	🟥 N				/user/a 3/SIM/EVT1_ML
7	BLK	setup	SDFFRPQRLV_D1_N_S6TR_C54L10	top.dut.BLK_A	1384468010 PS	🟥 N				
8	BLK	setup	SDFFRPQRLV_D1_N_S6TR_C54L10	top.dut.BLK_A	1388434764 PS	🟥 N				
9	BLK	setup	SDFFRPQRLV_D1_N_S6TR_C54L10	top.dut.BLK_A	1388434764 PS	🟥 N				/user/a 3/SIM/EVT1_ML
10									No timing violations found	/user/a 3/SIM/EVT1_ML
11									No timing violations found	/user/a 3/SIM/EVT1_ML

Figure 4. Final output of timing violation parser utility

III. SSDT Enhanced Automation Techniques

ENV Audit: While running simulations with a highly hybrid yet complex environment to get faster turnaround, the probability of an oversight occurring increases. GLS environment configuration requires an in depth understanding of functional path spanning across blocks failing which may result in a fake pass, where a block of concern or importance might be run as non-netlist. SDF files, accurate timing checks off files for asynchronous flops/false path/multicycle path and incorrect file version control also impact the result of simulation and warranting reruns. With the Python based environment audit script (Figure 5), such mistakes which results in vast simulation rerun times can be caught even before starting the simulation, ensuring first time correctness of GLS environment and runs as per design requirement. This project agnostic utility checks for annotation percentage as well as ascertains the genuineness of passing test in the most optimum conditions. This audit utility brings down the number of iterations before stable environment convergence from 4 or 5 to 1 (Table 1).





	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Elaboration Log Audit												
2	BLK	Expected	Alloted m	Mode Sta	Comment	Expected	Used Tfile	Tfile Statu	Comment	Expected	Used Tfile	Tfile Versi	Comments
3	A	GLS	GLS	🟩 PASS		BLK	BLK_	🟩 PASS		3	1	🟥 FAIL	Wrong versio
4	A	RTL	GLS	🟥 FAIL	Block AU	NA	NA	NA		NA	NA	NA	
5	B	GLS	GLS	🟩 PASS		BLK	BLK_	🟩 PASS		1	1	🟩 PASS	
6	C	GLS	GLS	🟩 PASS		BLK	BLK_	🟩 PASS		1	1	🟩 PASS	
7	C	RTL	GLS	🟥 FAIL	Block CM	NA	NA	NA		NA	NA	NA	
8	C	RTL	GLS	🟥 FAIL	Block CPU	NA	NA	NA		NA	NA	NA	
9	D	RTL	GLS	🟥 FAIL	Block DIS	NA	NA	NA		NA	NA	NA	
10	F	RTL	GLS	🟥 FAIL	Block FCV	NA	NA	NA		NA	NA	NA	
11	F	RTL	GLS	🟥 FAIL	Block FLC	N		NA		NA	NA	NA	
12	H	RTL	GLS	🟥 FAIL	Block HO	NA	NA	NA		NA	NA	NA	
13	N	GLS	GLS	🟩 PASS		BLK	BLK_	🟩 PASS		1	1	🟩 PASS	
14	S	RTL	GLS	🟥 FAIL	Block SUP	NA	NA						
15	T	RTL	GLS	🟥 FAIL	Block TUP	NA	NA						
16	T	RTL	GLS	🟥 FAIL	Block TUP	NA	NA						
17	TO	GLS	GLS	🟩 PASS		AVO	AVO						

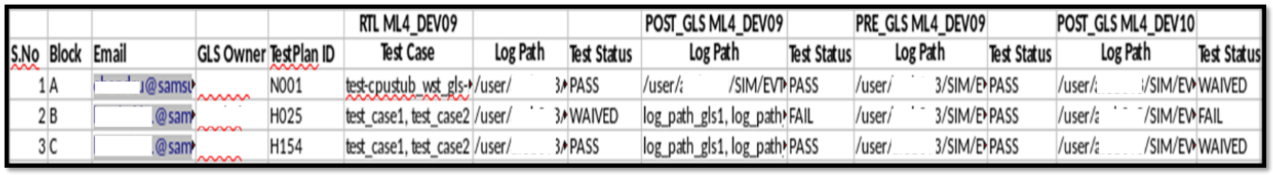
```
Run Time     11:30:00
SDF Statistics
No. of Pat No. of Disabled Pathdelays = 0       Annotated = 99.77% (659933
No. of Tch No. of Disabled Tchecks   = 87367   Annotated = 99.85% (20718
             Total(T)   Disabled(  Annotate  Percentage(A/(T-D))
Path Delay   66142646   0          65993376  99.77
$setup       40         0          0         0.00
$hold        20         0          0         0.00
$period      62086      0          62084     100.00
$width       1124229    55144      1121764   99.82
$recrem      1163365    6881       1160967   99.85
$setuphold   8346583    25342      8334034   99.88
$nochange    4          0          0         0.00
```

Elaboration / Simulation

Figure 5. GLS environment audit script output

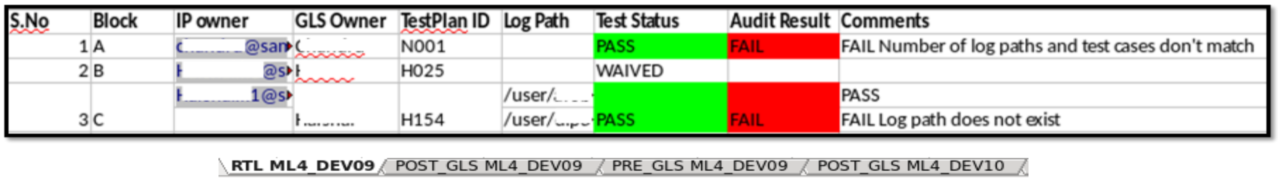
Plan Audit: In order to save time on GLS runs, it is prudent to have a historical data of the test progress and reference to narrow down on the failures. This utility audits the verification plan status based on the inputs from the engineers and checks for errors in the logs, existence of test database for a particular release version/flavour as required. When

SoC is nearing tapeout, it might get difficult to procure a regression manager license as thousands of RTL and netlist simulations would be running in parallel. In such scenario, it can mimic that functionality without needing an additional costly license. The utility is capable of saving at least 1.5 – 2 man months with audit ensuring correctness of the databases across different flavour of DUT over different releases (Figure 7).



					RTL ML4_DEV09			POST_GLS ML4_DEV09		PRE_GLS ML4_DEV09		POST_GLS ML4_DEV10	
S.No	Block	Email	GLS Owner	TestPlan ID	Test Case	Log Path	Test Status	Log Path	Test Status	Log Path	Test Status	Log Path	Test Status
1	A	[redacted]@samsu	[redacted]	N001	test-cpustub_wst_gls-	/user/[redacted]3/	PASS	/user/[redacted]/SIM/EVT	PASS	/user/[redacted]3/SIM/E	PASS	/user/[redacted]/SIM/EV	WAIVED
2	B	[redacted]@sam	[redacted]	H025	test_case1, test_case2	/user/[redacted]3/	WAIVED	log_path_gls1, log_path	FAIL	/user/[redacted]3/SIM/E	PASS	/user/[redacted]/SIM/EV	FAIL
3	C	[redacted]@sam	[redacted]	H154	test_case1, test_case2	/user/[redacted]3/	PASS	log_path_gls1, log_path	PASS	/user/[redacted]3/SIM/E	PASS	/user/[redacted]/SIM/EV	WAIVED

Figure 6. Sample input test plan status xls

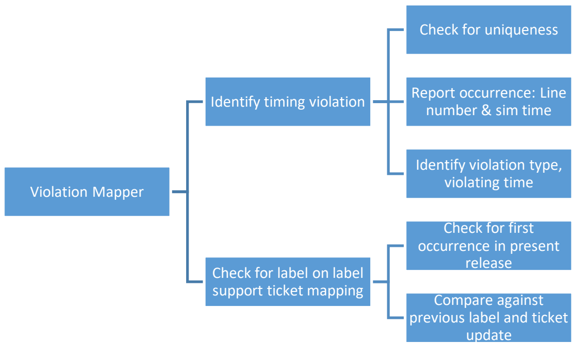


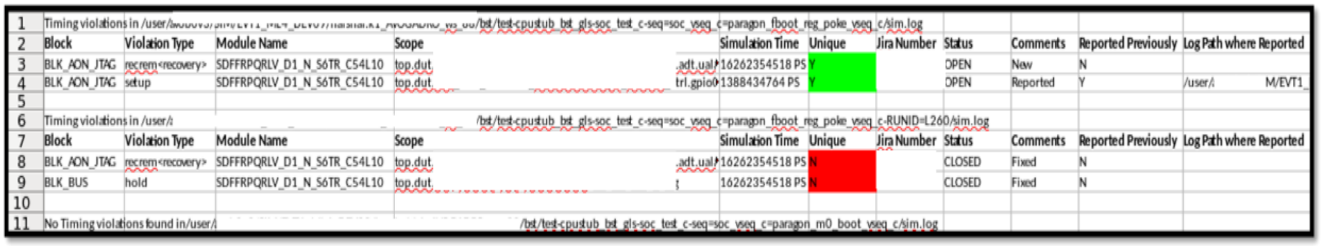
S.No	Block	IP owner	GLS Owner	TestPlan ID	Log Path	Test Status	Audit Result	Comments
1	A	[redacted]@san	[redacted]	N001		PASS	FAIL	FAIL Number of log paths and test cases don't match
2	B	[redacted]@s	[redacted]	H025		WAIVED		
		[redacted]1@s			/user/[redacted]			PASS
3	C		[redacted]	H154	/user/[redacted]	PASS	FAIL	FAIL Log path does not exist

RTL ML4_DEV09 / POST_GLS ML4_DEV09 / PRE_GLS ML4_DEV09 / POST_GLS ML4_DEV10

Figure 7. Audit output based on DUT type and release label

Violation Mapper: Simulation free of timing violations is key to successful silicon results. Debugging the violation on GLS is very time consuming and has a direct impact on valuable engineer’s time. This calls for an efficient and methodical approach to map the violations over different DUT versions, grouping similar violations across blocks to avoid duplicate efforts from engineers working on the SoC and to dynamically track the resolution of the violation in subsequent release and in the JIRA or bug tracker system. The Violation Mapper script (Figure 8) with artificial intelligence built-in, reduced not just the turn-around time for the resolution of the issues but also improved the efficiency of the team by avoiding duplicate/rework resulting in a saving of at least 10 – 15 months for bigger SoCs with efficient tracking of all the violations.


Violation Mapper
Identify timing violation
Check for uniqueness
Report occurrence: Line number & sim time
Identify violation type, violating time
Check for label on label support ticket mapping
Check for first occurrence in present release
Compare against previous label and ticket update



1	Timing violations in /user/[redacted]/bst/test-cpustub_bst_gls-soc_test_c-seq=soc_vseq_c=paragon_fboot_reg_poke_vseq_c/sim.log										
2	Block	Violation Type	Module Name	Scope	Simulation Time	Unique	Jira Number	Status	Comments	Reported Previously	Log Path where Reported
3	BLK_AON_JTAG	recrem<recovery>	SDFFRPQRLV_D1_N_S6TR_C54L10	top.dut.[redacted].adt.ual	16262354518 PS	Y		OPEN	New	N	
4	BLK_AON_JTAG	setup	SDFFRPQRLV_D1_N_S6TR_C54L10	top.dut.[redacted]trl.gpio0	1388434764 PS	Y		OPEN	Reported	Y	/user/[redacted]M/EVT1_
5											
6	Timing violations in /user/[redacted]/bst/test-cpustub_bst_gls-soc_test_c-seq=soc_vseq_c=paragon_fboot_reg_poke_vseq_c-RUNID=L260/sim.log										
7	Block	Violation Type	Module Name	Scope	Simulation Time	Unique	Jira Number	Status	Comments	Reported Previously	Log Path where Reported
8	BLK_AON_JTAG	recrem<recovery>	SDFFRPQRLV_D1_N_S6TR_C54L10	top.dut.[redacted].adt.ual	16262354518 PS	N		CLOSED	Fixed	N	
9	BLK_BUS	hold	SDFFRPQRLV_D1_N_S6TR_C54L10	top.dut.[redacted]	16262354518 PS	N		CLOSED	Fixed	N	
10											
11	No Timing violations found in/user/[redacted]/bst/test-cpustub_bst_gls-soc_test_c-seq=soc_vseq_c=paragon_m0_boot_vseq_c/sim.log										

Figure 8. Utility output

With a mix of traditional and enhanced audit automation techniques combined with erstwhile pure traditional techniques, around 4x-5x performance improvement is observed for a GLS suite of around 120 tests (Table 1).

Table 1. Initial v/s improved GLS metrics

Initial Metrics						Improved Metrics via Traditional+Audit techniques			
Block	Total Tests	No. of netlist labels	Avg Iterations for first pass	Avg run time (hrs)	Total runtime (hrs)	Avg Iterations for first pass	Avg run time (hrs)	Improved Total runtime (hrs)	Net Improvement (X times)
A	19	8	4	85	51680	1	78	11856	4.4
B	12	8	4	76	29184	1	68	6528	4.5
C	4	8	2	94	6016	1	84	2688	2.2
D	3	7	3	63	3969	1	57	1197	3.3
E	2	8	2	67	2144	1	55	880	2.4
F	14	8	4	78	34944	1	69	7728	4.5
G	8	8	3	163	31296	1	145	9280	3.4
H	22	8	4	102	71808	1	97	17072	4.2
I	13	6	3	73	17082	1	62	4836	3.5
J	17	8	4	68	36992	1	60	8160	4.5
K	6	8	5	152	36480	1	129	6192	5.9
L	10	8	3	79	18960	1	74	5920	3.2

IV. SSDT Fast Convergence Innovation Techniques

A. Custom Simulation Strategies (CSS)

The simulations for the current DUT takes approximately 60 hours to complete a basic SoC boot with full chip netlist which can go up to 200+ hours for timing PG net simulations. The internal methods via CSS to curtail these simulation times include the design and sequence based changes that regulates the load on the simulator. SSDT employs Skip Initialization Analysis to retain the intent of simulation by eliminating irrelevant clock/firewall/controller/DRAM initialisations. Upon further analysis, a matrix was developed for skipping power up based on block/IP function resulting in further saving of simulation time. X propagation originating from non-resettable flops which was optimized at zero time with library cell hack deposits instead of tcl initialisation. Additionally, with CSS deployed across the project, savings on disk space became notably visible with reduction in build/simulation sizes. The CSS techniques resulted in saving close to 8000 man-hours (Table 2) over the course of the project.

Table 2. SSDT improvement metrics via CSS

Scenario	Compile	Boot test	Mem test	CPU test	Power test	Disp test	PCIE test
Traditional	23hrs	38hrs	71hrs	60hrs	62hrs	66hrs	72hrs
Skip Init runs	23hrs	28hrs	45hrs	42hrs	38hrs	40hrs	55hrs
Skip Power up runs	23hrs	24hrs	38hrs	36hrs	34hrs	46hrs	53hrs
GLS + RTL runs	13hrs	24hrs	38hrs	36hrs	34hrs	46hrs	53hrs
GLS + FAKE runs	12hrs	20hrs	29hrs	30hrs	26hrs	32hrs	38hrs
Improvement	1.9x	1.9x	2.4x	2x	2.3x	2x	2.2x

B. Simulation Performance Optimization Wrapper (SPOW)

SPOW deploys combination of simulation tool methods and Load Sharing Facility (LSF) optimizations to deliver up to 5x simulation improvements:

Capture and Replay (CNR): CNR is a mechanism by which the simulation result is captured first and run on a different version of the same (Figure 9). In SSDT architecture, this flow is deployed to capture stimulus from RTL to replay at GLS for the first time. It additionally optimizes runs for the targeted block and aids VCD generation for power estimation and IR drop analysis.

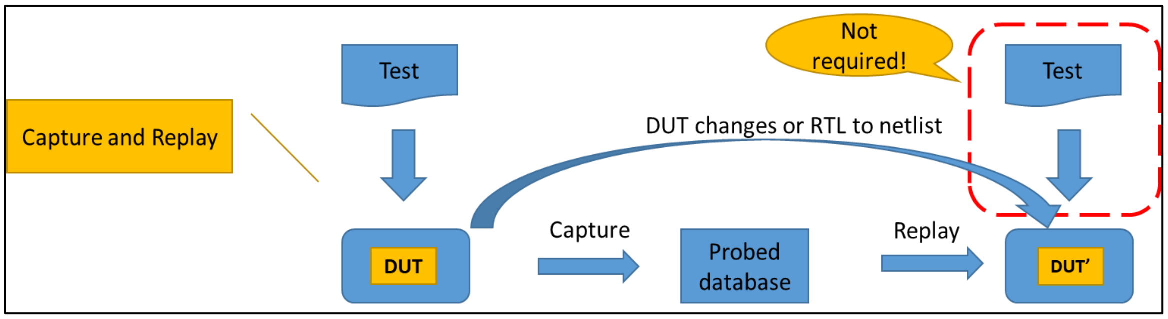


Figure 9. Capture and replay from RTL to GLS

Incremental Checkpoint Utility - Save and Restore (ICU - SNR): SNR is a method in which the common sequences of a SoC level test can be saved and any other test can be restored from the end of previous simulation snapshot. SSDT additionally enhances the simulation time savings with extending the save timeframe to IP specific sequences with ICU. Every IP has its own set of initialization sequences that are required to be executed after the common sequences. Adding it as a part of the SoC common sequences will be counterproductive. ICU enables the creation of checkpoints at the end of the IP specific initialization sequences by leveraging the SoC snapshot (Figure 10).

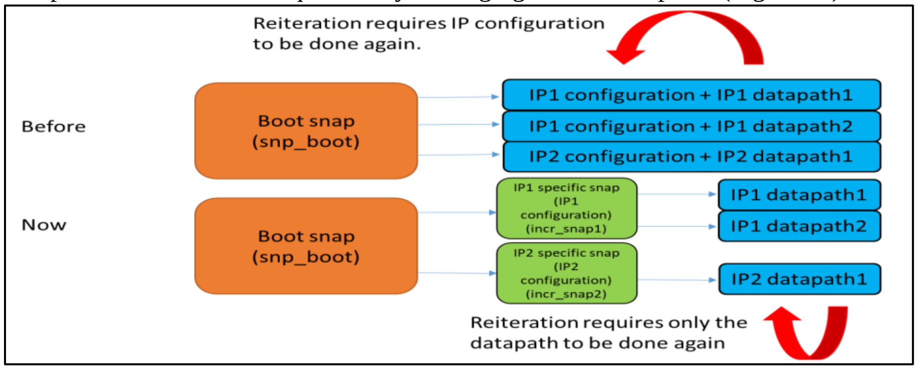

Reiteration requires IP configuration to be done again.
Before
Boot snap (snp_boot)
IP1 configuration + IP1 datapath1
IP1 configuration + IP1 datapath2
IP2 configuration + IP2 datapath1
Now
Boot snap (snp_boot)
IP1 specific snap (IP1 configuration) (incr_snap1)
IP1 datapath1
IP1 datapath2
IP2 specific snap (IP2 configuration) (incr_snap2)
IP2 datapath1
Reiteration requires only the datapath to be done again
Figure 10. Before and after incremental snapshot implementation

Table 3. Runs with and without ICU

Criteria	Memory overhead (For 1 IP)	Avg Run time (For 1 IP)
Without ICU	0	86.1hrs
With ICU	31GB	28.4hrs
Improvement	-	3.02x

Automatic Periodic Checkpoint Generation (APCG): APCG plugin periodically saves the simulation snapshot of the run based on simulation time or wall clock time. The plugin has flexibility to program the number of snapshots to be saved to get rid of redundant snapshots. The periodic checkpoint generation (Figure 11) setting is based on the analysis of simulation kill, user mistakes, failure stop & rerun scenarios and regression run times. In Figure 11, the periodic checkpoint generation is represented by picking the number of snapshots that are stored at any time to be 3. Here, the parallel running tcl script spawns 3 individual breakpoints and each breakpoint maintains its own checkpoint. Another add-on to APCG plugin ensures that restarted simulations and their status are accurately represented in the HTML regression dashboards. This custom script ensures that the correct test command is replaced by the older test command so that when the abruptly ended run is restarted, it will appropriately modify the regression dashboard to reflect the correct number of tests and their run status.

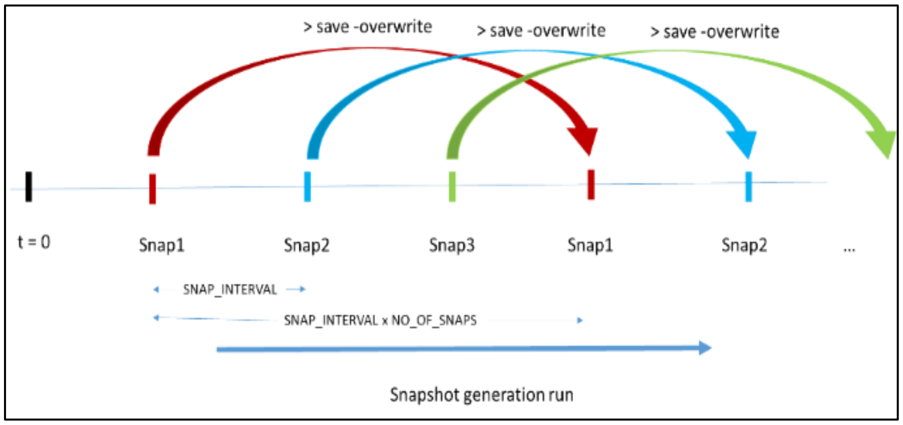


Figure 11. Automatic Periodic Checkpoint Generator flow

Table 4. Normal v/s runs with APCG

Criteria for 1 IP	Memory overhead	No. of abruptly killed runs	Time spent on rerun of abruptly killed runs
Without APCG	0	5	157h
With APCG	90GB	7	84h
Improvement (%)	-	-	46

Profiling & Access: Simulation profiling and further analysis of the weight each component had resulted in modifying the testbench and RTL coding styles for optimal performance. The profiling activity resulted in tweaking the simulator with appropriate switches that are custom suited for the database resulting in up to 20% simulation speed improvements. The simulation tool debug permission across the full compiled database was tweaked due to the confidence resulting from other SSDT checks. Optimization at signals level were carried with AI and ML scripts that restricts the debug access to signals that explicitly require permissions. Experiments, including dedicated machine runs, were carried out to understand, analyze and optimize the machine (LSF) on which the simulation is executed.

Table 5. SSDT improvement metrics for external methods via SPOW

Scenario	Boot test	Mem test	CPU test	Power test	Disp test	PCIE test
Access +RWC	50hrs	75hrs	69hrs	72hrs	73hrs	85hrs
Access +R	38hrs	71hrs	60hrs	62hrs	66hrs	72hrs
Afile	32hrs	65hrs	56hrs	57hrs	60hrs	65hrs
Dedicated	31hrs	63hrs	52hrs	54hrs	54hrs	63hrs
Improvement	1.4x	1.2x	1.3x	1.4x	1.4x	1.3x
With SNR	31hrs	29hrs	33hrs	25hrs	28hrs	40hrs
Improvement	1x	2.2x	1.7x	2.1x	2.1x	1.7x
w/o ICU & APCG Avg time for 100 tests	31hrs	156hrs	120hrs	98hrs	186hrs	264hrs
w/ ICU & APCG Avg time for 100 tests	31hrs	45hrs	43hrs	30hrs	49hrs	60hrs
Improvement	1x	3.4x	2.8x	3.2x	3.8x	4.4x

V. Result

SSDT's modularity and ease to deploy across any projects and different abstraction of the testbench resulted in huge savings in terms of manual effort the engineer spends, simulation run time and performance enhancement in the range of 6x-10x (Table 6). It has been implemented in 3 recent SoC DV projects for Mixed Reality (AR/VR/XR), HPC 3DIC and Generative AI 2.5DIC applications and it has provided consistent improvement in GLS execution.

Table 6. Improvement metrics with SSDT per IP test

Scenario	Boot test	Memory tests	CPU tests	Power tests	Display tests	PCIE tests
Initial avg	50hrs	156hrs	120hrs	98hrs	186hrs	264hrs
FCMAST Optimized avg	18hrs	26hrs	25hrs	24hrs	28hrs	29hrs
Improvement	2.7x	6x	4.8x	4.1x	7x	9.1x

VI. Conclusion

With increasing size and complexity of the chip, the total number of tests to be verified and net run time rise exponentially. With time to market and adherence to schedule being critical parameters for the success of a product, the need to strategize optimal selection of tests to be run without compromise on quality while simultaneously obtaining quicker results without doing multiple iterations is of paramount importance. SSDT has provided consistent improvement in GLS execution by guaranteeing first run correctness and fast convergence to meet stringent timelines with its automated plug-ins saving both precious man months and the costlier licensing, storage and infrastructure costs (Table 7). Future scope with integration of emulation into SSDT, n-die distributed simulation strategies for multi-chiplet based SoCs and leveraging AI and ML tools for wavemining and automated debugs have already been evaluated and are under development stage.

Table 7. Consolidated initial v/s final SSDT GLS metrics

Initial Metrics						FCMAST Improved Metrics			
Block	Total Tests	No. of netlist labels	Avg Iterations for first pass	Avg run time (hrs)	Total runtime (hrs)	Avg Iterations for first pass	Avg run time (hrs)	Improved Total runtime (hrs)	Net Improvement (X times)
A	19	8	4	85	51680	1	48	7296	7.1
B	12	8	4	76	29184	1	33	3168	9.2
C	4	8	2	94	6016	1	42	1344	4.5
D	3	7	3	63	3969	1	35	735	5.4
E	2	8	2	67	2144	1	31	496	4.3
F	14	8	4	78	34944	1	38	4256	8.2
G	8	8	3	163	31296	1	101	6464	4.8
H	22	8	4	102	71808	1	44	7744	9.3
I	13	6	3	73	17082	1	24	1872	9.1
J	17	8	4	68	36992	1	28	3808	9.7
K	6	8	5	152	36480	1	86	4128	8.8
L	10	8	3	79	18960	1	31	2480	7.6

References

[1] Harshal Kothari, Vinay Swargam, Sriram Kazhiyur Soundarrajan, Somasunder Kattepura Sreenath, "*A Novel Approach to Expedite Verification Cycle using an Adaptive and Performance Optimized Simulator Independent Verification Platform Development*", DVCON Europe 2022.

[2] Harshal Kothari, Eldin Ben Jacob, Sriram Kazhiyur Sounderrajan, Somasunder Kattepura Sreenath, "*Centralized Regression Optimization Toolkit (CROT) for expediting Regression Closure with vManager & Xcelium Performance Optimization*", CadenceLive India 2021.