Automation for Early Detection of X-propagation in Power-Aware Simulation Verification using UPF IEEE 1801

Tony Gladvin George*, Ramesh Kumar*, Kyuho Shim*, Karan K*, Wooseong Cheong*, ByungChul Yoo*
*Memory Division, Samsung Electronics
email: (tony.gg, ramesh.srk, kyuho.shim, karan1.k, ws.cheong, byung.yoo)@samsung.com

***Abstract*- One of the goals of power-aware simulation verification is to detect X-propagation and Isolation violations in the design. The current methods devised to mitigate these issues largely depend on the scoreboard to detect data integrity failures caused by X-propagation. In this traditional approach, a significant amount of time is necessary to trace the origin of invalid X-values using waveforms. Hence, the turn-around time for bug detection and validating the corresponding bug fix is high for the power-aware simulation. The proposed automation approach automatically translates the power design into checkers with assertions. The generated checkers are capable of detecting X-propagation and Isolation violations and are instantiated in a hierarchy parallel to the SoC-design. These checkers will enable the verification engineer to pinpoint the origin of invalid X-values with low turn-around time. This paper covers detailed methods to generate assertions & checkers for power domains, signal isolation, state retention, and restoration, which essentially left-shift the power-aware verification. The unexpected challenges in generating such automated assertions are addressed in this paper, and practical solutions are provided.**

Keywords: Power-Aware Verification, UPF simulation verification, Automated Assertions.

I. INTRODUCTION

This paper facilitates verification engineers who write power-aware test benches at the SoC level. Despite its importance, power design is often added at later stages to the SoC because it is beyond regular functional operation [1]. The new release of Wilson Research Group and Mentor Functional Verification Study [2] finds that the third leading cause of chip re-spin is due to issues in Power Design. This observation suggests the need for completion of power-aware verification with a shorter verification cycle.

In an SoC design with multiple IPs, each IP can move between power modes (power-off/power-on/sleep) to reduce leakage power. Depending on the power mode in operation, the independently powered regions of the IP, namely power domains, can be turned on and off, as shown in Figure 1. During such power on/off situations, retention cells save & restore memories and register values before and after shut down, respectively. There are isolation cells that keep the turned-off IP outputs in a previously defined value, and this is how the shut-down IP does not corrupt other active IP functionality [3]. Successful development of low-power semiconductor designs includes checking UPF descriptions as well as verifying UPF against the design at multiple stages in the project [4]. The power-management verification approach described in this paper verifies the functionality of the power-management elements described above and addresses the challenges in left-shifting the verification.

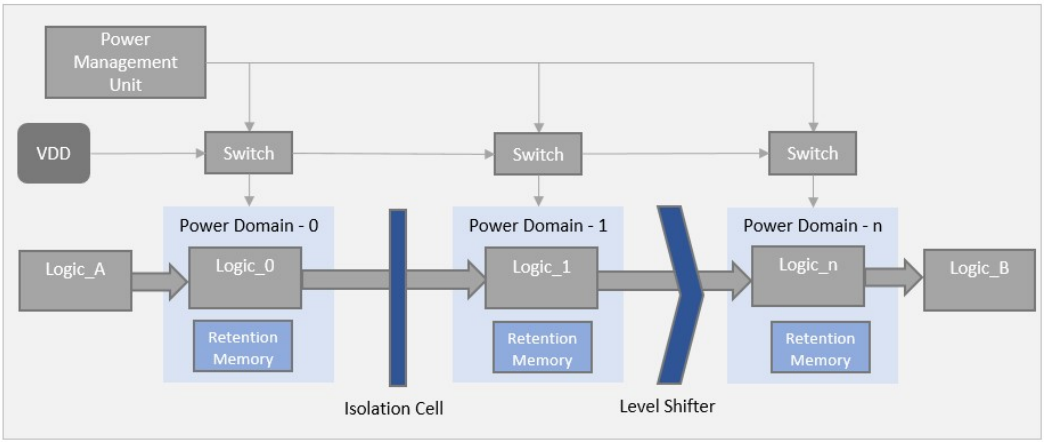

Figure 1: SoC with Power Design

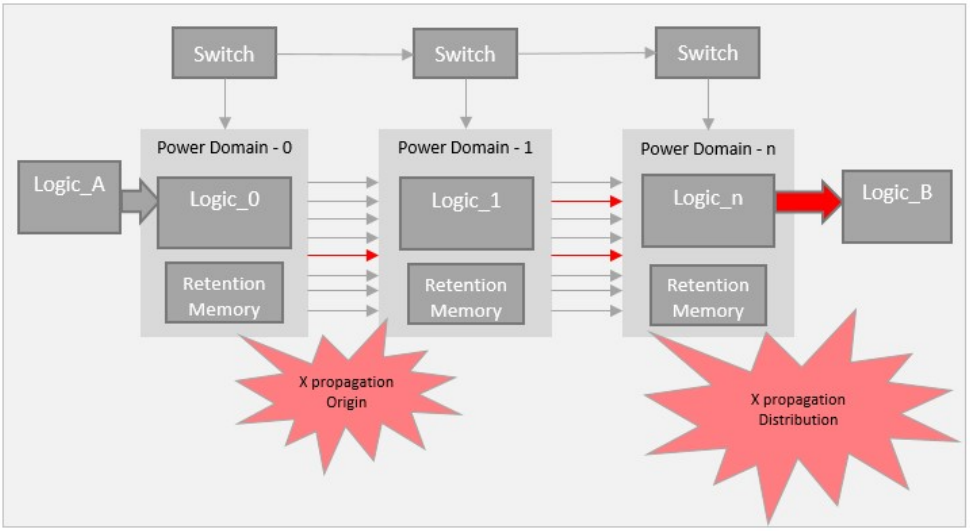

Figure 2: X-Propagation and distribution

A. Problem Statement

For low-power design and verification, the debug challenges are complicated, resource, and time-consuming due to complex power management architectures and instrumentation implications on the actual design [5]. Power-aware verification, which employs a UPF file describing the power intent, requires significant simulation time compared to the simulation without UPF. The power switches described in the UPF file control the power supply to the power domains based on the power modes. Since not all blocks are powered down during a UPF simulation, X-propagation can occur and poses challenges to root-cause-analysis (RCA) of the bugs for the origin of X-propagation. A single X-Value can propagate and gets distributed to thousands of signals within a few clock cycles, as shown in Figure 2. Among those signals, many can have valid X-value during the power-down phase, but the rest of the signals are supposed to retain the binary values. Waveform analysis for RCA is time-consuming, and it is necessary to leverage the automation approach.

Tool-generated assertions (or low-power checks) are used widely in low-power verification. However, they may not be exhaustive in all the designs for the following reasons [5]. a) Specific design requirements, b) New set of protocols appears now and then c) Not all EDA vendors support both UPF and CPF-based automated assertion generation [6]. The static checker provided by a major EDA company can cover the 'Missing isolation Cell', which checks the presence of the isolation cells at the inputs/outputs of a power domain. However, this strategy does not cover the operation of control signals of the isolation & retention cells that need functional verification to ensure seamless switching between power modes, as shown in Figure 3.

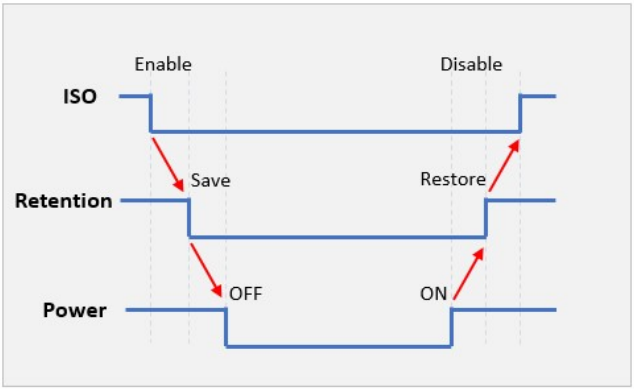

Figure 3: Power Sequence. ISO enable → Retention Enable → Power off → Power On → Retention Disable→ ISO disable

B. *Possible Root cause of X-propagation bugs*

X-propagation can occur for multiple reasons [5], and it can be categorized into four.

1) Missing isolation cell
2) Missing control signal
3) Missing connection in the data signal
4) Wrong power sequence control

These scenarios must not be present in power-aware simulation to eliminate issues like X-propagation and other bugs.

C. *Proposed Solution*

To address the issues mentioned above, we propose an approach by generating automated checkers before simulation to verify the power controls/sequencing in the SoC. The proposed approach will also enable the early detection of design bugs, independent of the simulator tool used.

At first, the UPF and RTL files are compiled, and from the elaboration log, the list of all UPF objects and the RTL signals connected to those UPF objects are extracted. A python based parser was developed to automate this process. Then assertions are generated from template and instantiated in the testbench via checker modules.

The added advantage of the automated assertions is the left shift of RCA. Unexpected X-value in the signals will fail the corresponding assertions. This failure of automated assertions enables the verification team to RCA the X-propagation instantly. Unlike simple assertions provided by static checking tools, these automated assertions are aware of the SoC design hierarchy along with the SoC's power control signals, which is the prime feature that enables easy detection of X-propagation.

This paper is organized as follows. In section II, background and related works about specification and verification of power-aware design are studied. Section III details the solution for generating automated checkers and assertions to verify the power controls/sequencing. The case study in section IV describes our observations in a real SoC design. Finally, section V summarizes our work and outlines future work.

II. Background and Related Work

A. *Unified Power Format*

HDL-independent power specification format is standardized by the Electronic Design and Automation (EDA) industry because of the benefits of early power-aware design. Two similar formats to specify the power design was developed by industrial consortiums, the Common Power Format (CPF) and the Unified Power Format (UPF) [7]. The new IEEE 1801 standard extends UPF [8]. In this paper, we discuss the approach using the UPF standard.

B *Power Domains, Retention and Isolation Cell*

The power-management module is the one that controls the power-up and power-down sequences. It uses power switches to turn off unused parts of the design that needs to be isolated by power domains. The power domains can also contain retention cells. A register's power is turned off when its power domain or IP is turned-off. Some registers also need to retain their values after being turned off. These are called retention registers, which restore their previous active value after being shut down. An Isolation Cell passes logic values during normal operation mode and clamps its output to some specified logic value when a control signal is asserted. It is required when the driving logic supply is switched off while the receiving logic supply is still on [5].

C *Automated Assertion Generation from use cases*

Traditionally, designers manually generate assertions according to their understanding of the design, which demands specialized knowledge. Even for experts, creating all required instances manually can be time consuming. These assertions can be grouped into two main categories, static and dynamic [9]. We utilize the static extraction technique, which analyzes the UPF description statically to derive the assertion. The UPF elaboration log is parsed, and RTL hierarchical signals in the UPF objects are obtained, eliminating the need to check the validity of the generated assertions by design engineers or other techniques [9].

D *Verification of the power design*

In general, verification of the power design falls into three steps. First, it is verified that the design correctly enters the power state. Second, the SoC has to provide the correct signal transitions at each power state. Third, it is ensured that the logic behaves correctly when the SoC enters a power state; for example, the unit has no activity when powered down [1].

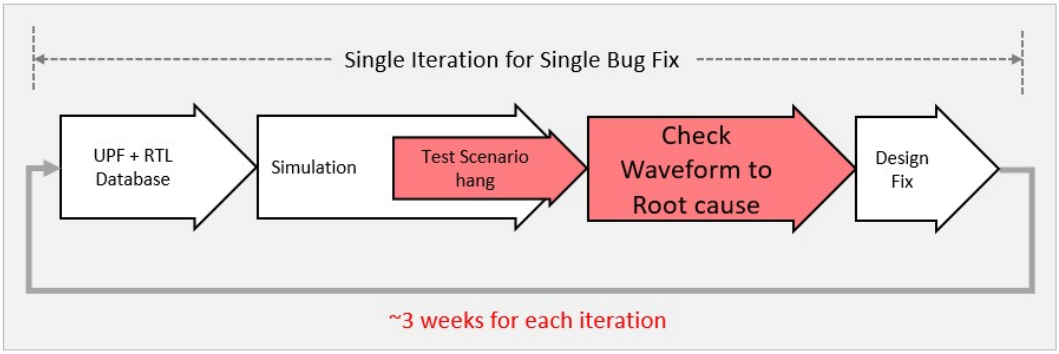

Figure 4: Traditional Approach for verifying Power-Design

Many functional defects can be exposed by simulating the power design. The most important among them are incorrect switching of power switches for power domains, falsely timed power-state transitions, wrong power states, and improper save/restore sequences for retention. Moreover, inadequate isolation, problems resetting the blocks to an operating state when power is restored, and incorrect level shifting between domains can also be detected [10], [3].

The traditional approach to verify the X-propagation is to simulate the functional verification testcases and check if any of the scoreboards for data integrity check is failing. This traditional approach adds delay in detecting bugs in power-aware simulation. It is because the maturity of the design to support data integrity will be available at a later point of time in the development cycle. As shown in Figure 4, it also adds additional overhead to the root cause of the bug. On average, Lint/Compilation/Elaboration with UPF takes a day, then sanity simulation for two days, and regression simulation takes five days. After the regression, the triage takes five days in case multiple failures are detected, and finally, fixing the design in the UPF file takes two days. Exact timelines per bug were not tracked during the original project; hence, we estimate an average of 5 days for triage after each regression.

E Related Work

In the paper "Confidently Sign-off any Low-Power Designs without Consequences" [5], multiple low-power design issues and debugging strategies are discussed. The paper states, "With verification tools dynamic checking capability combined with user's custom low-power assertions, one is ensured that all the low-power checks are built into the design or simulation run". The solutions proposed in this paper are a combination of the tool's dynamic checks and custom LP-Checks. Our paper extends this approach by automating the generation of custom LP-Checks, reducing the necessity of manual intervention, and enabling scaling at the SoC level.

III. SOLUTION

To address the issues mentioned above, we introduce a novel approach by generating automated checkers before simulation to verify the power controls/sequencing of the SoC. This approach comprises three stages. In the first stage, RTL and UPF files are compiled and elaborated. The resulting elaboration log contains the list of all UPF objects and RTL signals connected to those UPF objects. In the second stage, the elaboration log and UPF files are passed through a python-based text parser to extract relevant information related to isolation and retention strategies. The parser generates an intermediate file that contains the extracted data, which is necessary for the next stage. In the third and final stage, the python checker generator uses this intermediate file to generate checkers & assertions to meet the power sequence shown in Figure 3. Python is applied to automate the stages mentioned above. The generated assertions are instantiated in the testbench via checker modules, as shown in Figure 11. Cover-bins are included in the checkers to track Checker Coverage.

A. Extending UPF Verification Environment by adding Automated Assertions

The python-based text parser is invoked from Makefile after the compile and elaboration is succeeded. The generated assertions/checkers for verifying the power controls/sequencing are written in System Verilog. During simulation, the verification environment applies testcases to the testbench. It invokes a behaviour corresponding to the functionality described in power intent. Once the checker coverage meets the target coverage, the power controls/sequencing is verified.

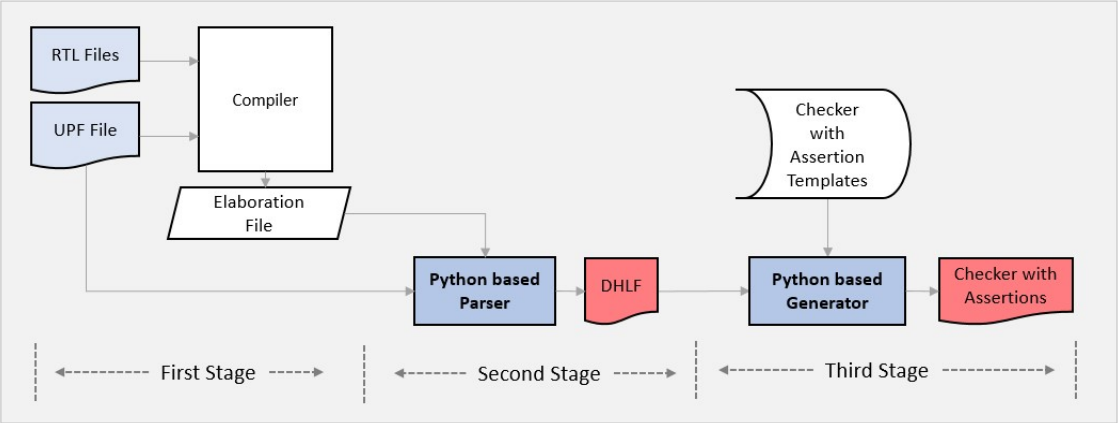

Figure 5. Automated generation checkers with assertions in Python

B. First Stage – Preparation for Automation

One of the challenges in creating assertions is, UPF file alone is insufficient to gather the RTL hierarchy information. An elaboration log is a readily available source to gather details of the RTL hierarchy because the RTL files are compiled and elaborated along with the UPF file. In our case, we use the Cadence tool flow, and the elaboration log is generated by Xcelium. We can also consider developing a script to parse RTL instead of parsing the elaboration log, which is beyond this paper's scope.

C. Second Stage - Text Parser and Intermediate file generation

In the second stage, the elaboration file along with the UPF file is parsed to fetch the signal hierarchy that matches the power controls/sequencing, as shown in Figure 5. UPF syntax structure is parsed/evaluated, and information on power domains is extracted as an intermediate file named Design-Hierarchy List File (DHLF). Information about supply states for the power domains, isolation for inactive domains, memory retention, and RTL hierarchy is gathered.

The following is an example of UPF elements that are parsed by the python script. As shown in Figure 6, the isolation cells can have many input elements such as domain, clamp value, isolation sense, etc. UPF file simplifies how to describe how the rules apply to a power domain as a whole. For example, *-applies_to_inputs* can apply to all inputs instead of explicitly mentioning all individual ports. This rule can be overridden by additional rules for specific ports. This syntax limits the parser to rely on the UPF file solely, so we have to use an elaboration log, as stated before.

Figure 6 defines the Isolation Cell for power domain PD0, with power supply VDD_ON and ground VSSI. The clamp value is defined by *–clamp_value* construct. These definitions are in line with UPF standards. *-applies_to_inputs* makes the rule apply to all inputs in the power domain PD0 so that the elaboration log will help the parser with the list of all the signals.

## Controller Input signal	## ISO Rule Declaration	## ISO Rule Exception cases
set_isolation_control PD_ISO \	set_isolation PD_ISO \	set_isolation PD_ISO \
-domain PD0 \	-domain PD0 \	-domain PD0 \
-isolation_signal SoC_design /power_manager /ISO_PD0_EN \	-isolation_power_net VDD_ON \	-isolation_power_net VDD_ON \
-isolation_sense low \	-isolation_ground_net VSSI \	-isolation_ground_net VSSI \
-location self	-clamp_value 0 \	-clamp_value 1 \
	-applies_to inputs	-elements { \
		SoC_design /power_manager/*axi* }

Figure 6: Identifiers for isolation

## Retention Cell Control	## Retention Policy	## Retention Cell
set_retention_control PD_RET -domain PD0 \ -save_signal {SoC_design /power_manager /PD0_RETENB high} \ -restore_signal { SoC_design /power_manager /PD0_RETENB low}	set_retention PD_RET -domain PD0 \ -retention_power_net VDD_ON -retention_ground_net VSSI \ -elements { \ SoC_design /Logic_0/RETEN_0 }	map_retention_cell PD_RET -domain PD0 \ -lib_cells {*SDRFF*}

Figure 7: Identifiers for Retention Memory

Figure 7 shows how the Retention cell is defined, and the retention policy is set for power domain PD0. PD0_RETENB is the save and restore signal with sense Low and high, respectively. In Figure 8, supply Net VDD_ON and supply port AVD_TS are created per the UPF constructs.

## Supply Net	## Supply Set	## Supply Port
create_supply_net VDD_ON	create_supply_set S_VDD_ON \ -function {power VDD_ON} \ -function {ground VSSI}	create_supply_port AVD_TS

Figure 8: Identifiers for Power Domains

D. Third Stage – Generation of Assertion and Checkers

In the third stage, the assertions and checker are generated from templates, and the verification environment is extended by adding assertions and checkers. From the DHLF file, RTL hierarchy information is furnished into the checker template, as shown in Figure 9. The generated checker will be compiled along with the necessary assertions, and that will be included in the testbench. Two types of assertions are generated and added to checkers. The first type of assertion is for basic power flow, all signals going outside the power domain, and the second type of assertion are for retention cells. We have two types of checkers to verify isolation cells:

1) Check if the control signals get toggled correctly in a power state. Sequences/properties are used to capture this logic, and assertions are coded, which helps us check the operation of the SoC during different power states.

2) Check if the signals connected to the isolation strategy are clamped to the correct value, as mentioned in the UPF object. The RTL hierarchical signals for these assertions are obtained automatically using the python parser as discussed above.

Figure 9 shows how the property *power_gating* is used to check the low-power state entry and if the ISO is enabled. This assertion fails if the ISO_EN is not set. This property is used with cover property to facilitate coverage. This cover property helps us to ensure that all the power domains are covered for entering a low-power state.

Sequences *seq_power_gating*, *seq_iso_check_*N,* and property *iso_check_*N* are used to check the clamp value. These assertions will help us detect the X prop if the clamp value is improper. This will also help us to find the root cause of the functional failures during low-power simulations.

Retention cells do the save and restore operation, as stated earlier. So, we need to check if the signals for save and restore are toggling during power down and power up, respectively. Similar to the isolation checkers, we have two categories of checkers:

1) Save and restore signals toggling at the appropriate power states, i.e., during power down and power up, respectively.

2) The signals connected to retention cells should have the same value before power down and after power up. This is achieved by using automated assertions.

In Figure 10, Sequences *seq_power_gating*, *seq_ret_check_*N,* and property *ret_check_*N* are used to check if the value is retained during sleep. As mentioned earlier, cover property is used to enable coverage for this retention check and will make sure all the retention cells are covered for low-power entry. This assertion helps easy debug and find the root cause of functional failures in low-power simulation.

```
module isolation_checker;
 property power_gating;
 @(posedge clk) (Power_state = POWER_DOWN) |=> (ISO_EN == *h1);
 Endproperty
 assert property (power_gating) $display ("power gating enabled for POWER_DOWN ");
 else `uvm_error("PMU_CHECKER power gating", $sformatf("power gating and iso en not matching in POWER_DOWN "));

 c_p_power_gating: cover property(power_gating);

 Sequence seq_power_gating;
    @ (posedge clk) (Power_state = power down) |=> (ISO_EN == *h1);
 end sequence

 Sequence seq_iso_check_*N
   @(posedge clk) $rose (ISO_EN) |=> <RTL_HIERARCHY.ISO_*N>.signal = Clamp_value;
 end sequence

 property iso_check_*N
   seq_power_gating |=> seq_iso_check_*N
 end property
 assert property iso_check_*N
   $display ("ISO _*N enabled in pg_all");
 else
   `uvm_error ("PMU_CHECKER ISO Check ", $sformatf ("ISO _*N  Not enabled "));

  cp_iso_check_*N cover property iso_check_*N;
endmodule
```

Figure 9: Template for checker with Assertions

```
Module retention_checker;
 Sequence seq_ret_check_*N
   Bit v;
   @(posedge clk) $rose (save_en) v = <RTL_HIERARCHY>.signal |->  [1:*] (restore_en) <RTL_HIERARCHY>.signal == v;
 end sequence

 property ret_check_*N
   seq_power_gating |=> seq_ret_check_*N
 end property

 assert property ret_check_*N
   $display ("RET _*N enabled in pg_all");
 else
   `uvm_error ("PMU_CHECKER RET Check ", $sformatf ("RET _*N Not enabled in power state "));

 cp_ret_check_*N cover property ret_check_*N;
endmodule
```

Figure 10: Template for Save Restore - for retention cells

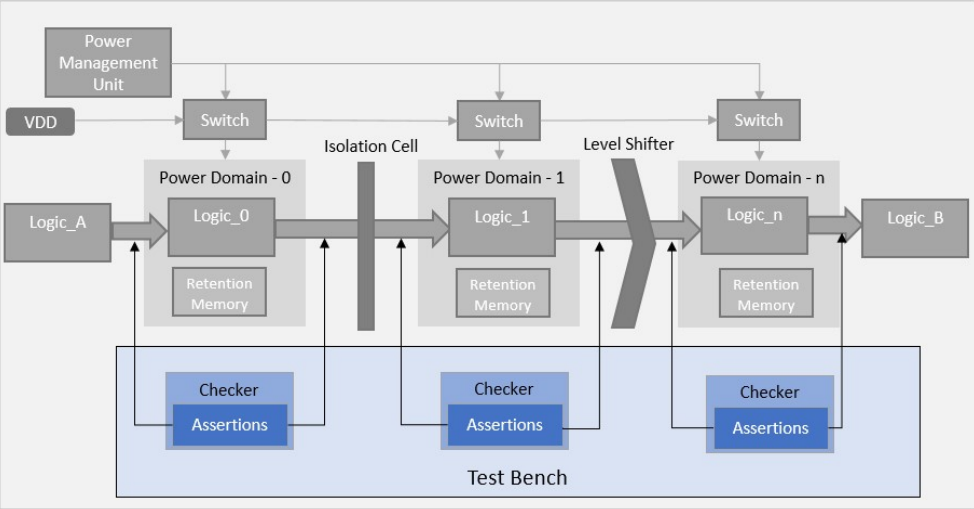

Figure 11: Insertion of generated checkers and assertions

Putting all together, the parser will look out for UPF syntax (ex: set_isolation), and the RTL hierarchy paths are inferred from the elaboration log file. The assertions are then generated based on template files. These assertions are grouped into power, retention, and isolation for generating the checker. Finally, the generated checkers are instantiated in the testbench, as shown in Figure 11.

The added advantage of the automated assertions is the left shift of RCA (Root Cause Analysis). Unexpected X-value in the signals will fail the corresponding assertions. This failure of automated assertions enables the verification team to RCA the X-propagation quickly. Unlike simple assertions provided by static checking tools, these automated assertions are aware of the SoC design hierarchy along with the SoC's power control signals, which is the prime feature that enables easy detection of X-propagation.

IV. Case Study and Results

In this case study, a chip tape out which took 40 days starting from sanity-clean-RTL to the beginning of synthesis phase is used for analysis. At the beginning of the project, it was not practical to predict the categories of issues we could encounter while the verification was progressing. It was also impossible to expect any of those issues could slow down the verification effort. Once the project commenced, a few bugs were detected at the early stage, which belonged to basic sanity. After the initial bug findings, the detection of the rest of the bugs was delayed despite continuous verification efforts by the verification team. This is due to the complexity of root cause analysis and associated long simulation cycles. This impact on schedule can be observed in Figure 12, which shows the number of bugs accumulated over time. Here most of the bugs were reported mid-way through the project timeline, and the bug trend was stabilized only towards the tape out.

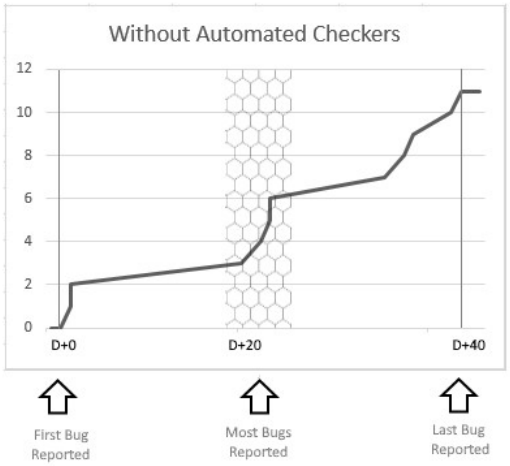

Figure 12: Project without automation module.

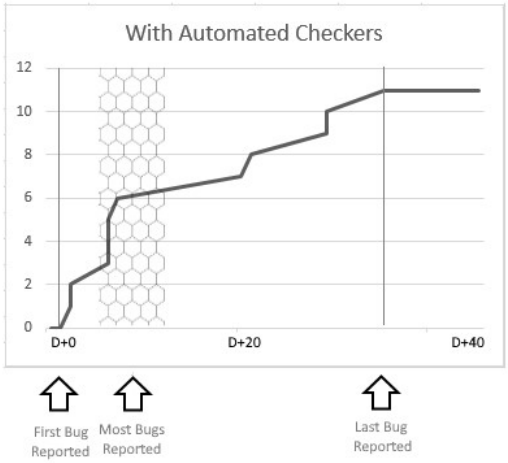

Figure 13: Project with automation module.

To left-shift the verification cycle and avoid the delays mentioned above for future projects, the team re-evaluated all the bugs reported and the time taken for RCA of each bug. Based on this analysis, the team concluded to introduce automated checkers that can accelerate the bug detection process and reduce the time necessary for RCA. The reusability of checkers was also important, which could further reduce the effort essential in upcoming projects. Hence the checkers were developed generically to ensure reusability. After the proposed automation approach was developed, the same project was executed to observe the expected advantages. For proof of concept, we evaluated the time taken to find the same amount of bugs in real project, i.e., 11 bugs, after introducing the proposed methodology into the verification pipeline.

The key observation in this exercise was that the effort for Root-Cause-Analysis had reduced significantly since the checker failures brought the attention of the RCA engineer closer to the root cause of the bug. As observed in Figure 13, the most number of the bugs are identified at the early stage of verification, and the bug trend is stabilized nine days before the synthesis phase, which is approximately 20% shift-left. The observed results met the expectation by introducing automation into the verification pipeline.

Figure 14 shows the timelines for the traditional vs. automation approach and the left shift achieved for reducing the verification cycle. It is also important to point out that the simulation time necessary to find the same bug using the traditional approach and automation approach is comparatively lesser. In the traditional approach, the simulation has to wait until the X-value propagates through the SoC and reaches the scoreboard for the assertion to fail. On the contrary, in the automation approach, the assertions in the checker will fail the moment an unexpected X-value is found or the power sequence is violated.

To implement the proposed approach, the effort necessary is approximated as follows. For the preparation of scripts and checkers, 10% of additional effort was required at the beginning of the project. However, 25% time-saving was observed in the triage phase since checkers helped to pinpoint the root cause. Since we prepared the proposed approach with reusability as a key factor, the subsequent projects had reduced effort at the deployment stage.

V. Conclusion

Our contribution comprises the automatic generation of checkers with assertions and cover points from the UPF design. The generated checkers compactly describe the functionality of the power controls/sequencing. Simulating the power-aware scenarios which include generated checkers, will automatically verify the SoC's power controls/sequencing. The key benefit of this approach is the high degree of scalability irrespective of the number of power domains present in the design and the reduction in the need for manual intervention for each of the assertions generated. Additionally, the assertion coverage enables metric-driven verification for verification sign-off. A shorter verification cycle for the power design lowers the risk of late and costly re-design.

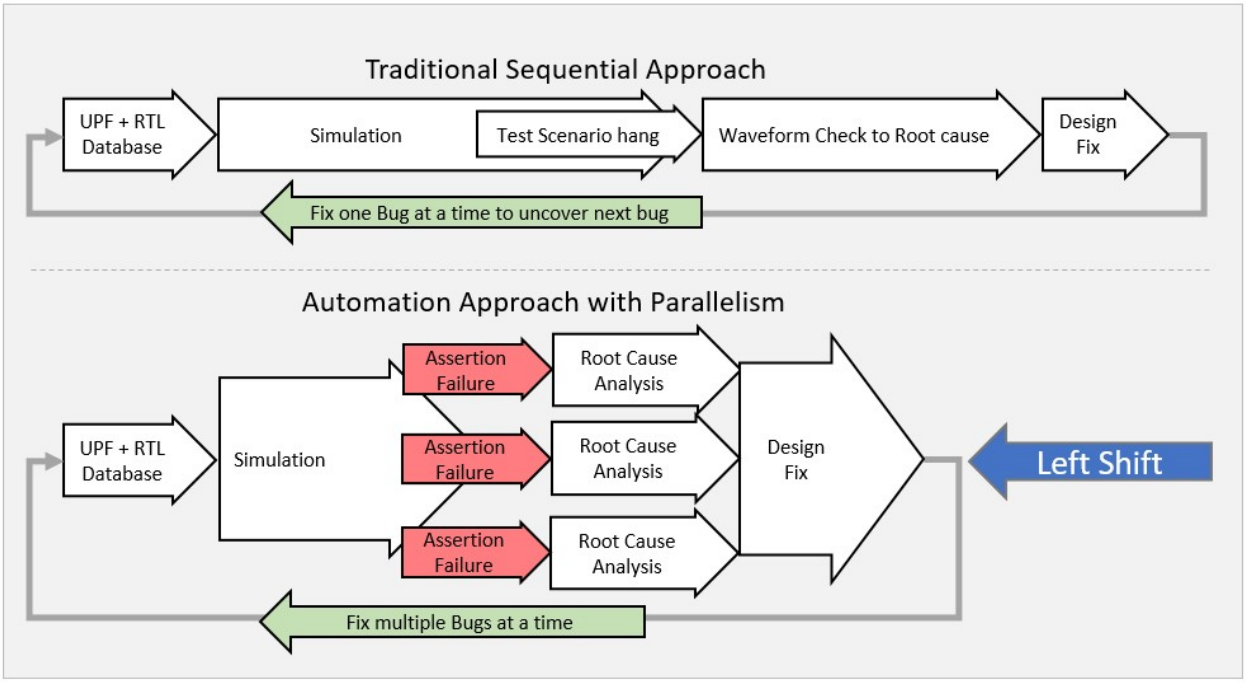

Figure 14: RCA – Left Shift

In this contribution, we are motivated by the need for a left shift of verification effort by leveraging the automation approach to avoid the detection of bugs while nearing the tape-out schedule. Without automated checkers, most bugs were reported mid-way through the project timeline, and the bug trend stabilized towards the tape-out. However, in the case of automated checkers, the most significant number of the bugs are identified at the early stage of verification, and the bug trend stabilized much before the tape out. The observed results met the expectation by introducing automation into the verification process. A small investment in automation will significantly reduce the verification cycle and unnecessary design iterations during later phases of the design.

REFERENCES

[1] B. Wile, J. Goss, and W. Roesner, Comprehensive Functional Verification the Complete Industry Cycle. Elsevier/Morgan Kaufmann, 2005.
[2] Foster H., 2020 Functional Verification Study, Wilson Research Group and Mentor, A Siemens Business, 2020
[3] F. Bembaron, S. Kakkar, R. Mukherjee, and A. Srivastava, 2009. "Low Power Verification Methodology Using UPF," in Conference on Electronic SoCs Design and Verification Solutions, DVCON, pp. 228–233.
[4] Himanshu Bhatt, Kiran Vittal. Four Steps for Static Verification of Low Power Designs Using UPF with VC LP, Synopsys white paper.
[5] Madhur Bhargava, Jitesh Bansal, and Progyna Khondkar, 2022. "Confidently Sign-off any Low-Power Designs without Consequences," DVCON2022.
[6] John Decker, Neyaz Khan, and Richard Goering, Power-Aware Verification Spans IC Design Cycle A Plan-To-Closure Approach Helps Ensure Silicon Success, Cadence Design Systems
[7] Christoph Trummer, Simulation-based Verification of Power Aware SoC-on-Chip Designs Using UPF IEEE 1801, 2010.
[8] IEEE Std 1801™-2015 for Design and Verification of Low Power Integrated Circuits. IEEE Computer Society, 05 Dec 2015.
[9] Tong Zhang, 2017. Automatic Assertion Generation for Simulation, Formal Verification and Emulation" IEEE Computer Society Annual Symposium on VLSI https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7987564
[10] A. Crone and G. Chidolue, 2007. "Functional Verification of Low Power Designs at RTL," Lecture Notes in Computer Science, vol. 4644, pp. 288–299.