2021 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE OCTOBER 26-27, 2021

Chiplevel Analog Regressions in Production

Analog-Mixed-Signal Simulation with Single Programmable Testbench, Code-Driven Stimulus and Self-Checking with SVA

Yi Wang, Dialog Semiconductor B.V., Den Bosch, the Netherlands (yi.wang@diasemi.com)

***Abstract*—This paper presents an AMS regression methodology at chip level using a single testbench with Verilog-AMS stimulus. All the simulation results are processed in the discrete domain using dedicated SystemVerilog module with SV assertions. An Asynchronous Finite-State-Machine (A-FSM) is used to align the Device-Under-Test (DUT), stimulus and assertions, it is fully configurable to cover 30+ test scenarios. The total regression run-time is less than 10 hours, which enables regular sign-off check within a day.**

Keywords---Analog Regression; Analog-Mixed-Signal Verification; Single Programmable testbench; Verilog-AMS Stimulus; System-Verilog-Assertion (SVA)

I. INTRODUCTION

The complexity of the power-management-unit (PMU) inside a System-on-Chip (SoC) is constantly increasing with market demands of performance and increased safety. Although using System Verilog User-Defined Nettype, it is possible to develop real-number models (RNM) accounting for loading effects and execute exhaustive Digital-Mixed-Signal (DMS) simulations [1], the leakage path, non-linear behavior, process/temperature dependence and capacitive loading from top-level connections of analog circuit are usually not covered in RNM. These effects have been exclusively simulated in SPICE simulators. Using advanced analog simulators and effective partitioning of signals interacting with discrete and continuous domains, it is possible to simulate the PMU blocks in spice netlist at chip level within 1 hour, which makes AMS Simulation - a feasible complementary verification method in addition to DMS using RNM.

Unlike digital verification with UVM methodology, analog verification mainly relies on manual inspection, usually comparing waveforms against requirements. Although there are self-checking methods in Verilog-A for analog signals with assertions[2]; the language itself is limited when compared to dedicated assertion languages like Property Specification Language (PSL) and System-Verilog Assertion (SVA). Additionally, implementing assertions in the continuous time domain using cross() function is computation intensive because the cross() function controls the time step of the analog solvers and can introduce additional simulation points especially when time and expression tolerance are absent[3]. The self-checking method presented in this paper is implemented using SVA modules which are tuned for analog behaviors. This method has the benefit of adopting mature assertion development from digital verification for inspecting the analog signal. It is faster since assertions are no longer executed by the analog simulator and provide opportunity for further data processing, therefore the overall simulation time of AMS is reduced.

Traditionally analog testbench setups have the stimulus and loads implemented using discrete components such as voltage sources, current sources, resistors, capacitors, etc. These setups have the following disadvantages:

- Hard to synchronize different sources and loads
- Stimulus does not self-adapt to process/temperature variations
- Hard to create complicated test patterns with DUT inside a closed loop
- One testbench can only serve limited number of test scenarios, in other words, multiple testbenches are required to cover the verification plan.

To mitigate these drawbacks, a single code-driven stimulus module implemented by Verilog-AMS is proposed in this paper. The stimulus module drives and loads the DUT. Inside the stimulus module, a built-in asynchronous finite-state-machine (A-FSM) controls stimulus based on the read-out from the DUT. Thus, a closed-loop is formed

which enables complicated test patterns and self-adaptation. Within Verilog-AMS stimulus code, the testbench configuration can be altered by compiler directives; and the selection of test patterns can be done via if-else or case-statement coupled with the stimulus module's parameters. Thus, using the proposed testbench it is possible to cover multiple test scenarios by varying variables from DUT/Stimlus/Simulator etc. The analog signals fed to and received from DUT are converted from/to discrete domain with customized drivers/probes, which are optimized for speed and accuracy. The converted analog signals are processed in the discrete domain based on the state of FSM from both DUT and Stimulus, thus the assertions are self-adaptive to the Process-Voltage-Temperature (PVT) variations. In the end, the pass/fail of the test scenario is determined by SVAs which monitor the complete waveform over the simulation. This makes it possible to put normal digital regression management tool on top of the presented flow.

The example in the paper is illustrated with a GUI based EDA tool which is widely used by the analog design engineer, but setup can be also be run on the command-line.

In section II, the testbench setup is explained starting from schematic till regression setup; an example of self-checking mechanism is explained in section III; followed by the job policy and statistics of regression run time.

II. Regressive Testbench Setup

A layered approach to testbench is used in this paper as demonstrated in Figure 1. At the bottom is the schematic which contains the DUT and stimulus module ($I_{STIMULUS}$). In the middle is the EDA GUI where the variables are defined and attached to the DUT and $I_{STIMULUS}$ for selecting the test scenarios, and the outcomes of SVA are also readout here. Finally, at the top is the regression control, where all the test scenarios are defined, and variables are given with the corresponding value.

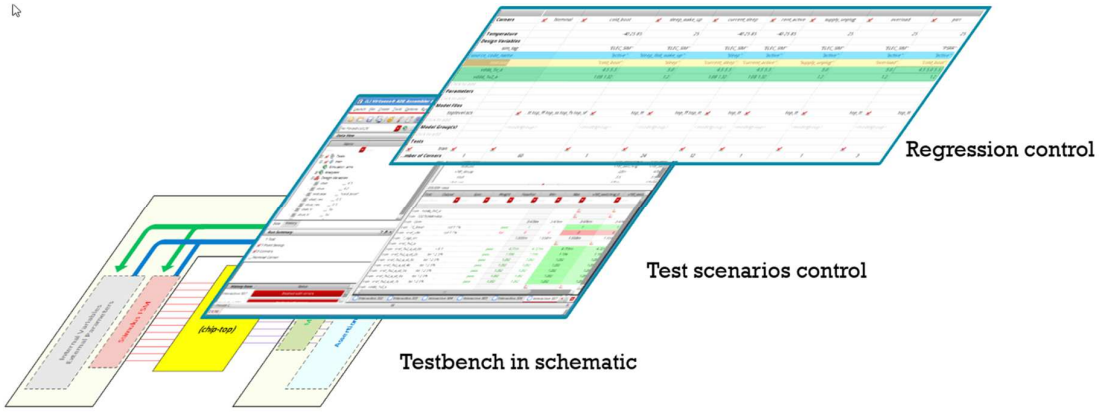

Figure 1. Layered testbench

A. *Testbench Schematic*

The testbench schematic in Figure 2 (a) contains two modules: DUT and $I_{STIMULUS}$. The DUT in this paper is a SoC composed of a digital core with a CPU; an analog core with multiple LDOs, DC-DC convertors and ADCs; and a radio core for Bluetooth Low Energy (BLE). The AMS verification focuses on the analog core in spice netlist using multiple test scenarios namely cold-boot, power supply switching, working mode transition, current consumption, etc.

The structure of $I_{STIMULUS}$ Verilog-AMS code is shown in Figure 2 (b). The electrical voltage/current sources for driving the DUT are provided by dedicated Verilog-AMS modules which reads the discrete real value as input. The resistive and capacitive loads can be altered by calling the task from the corresponding Verilog-AMS modules. With this setup, all the behavior of stimulus is described in the discrete domain which enables verification IP to be reused between AMS and DMS simulation. The built-in A-FSM (STI_FSM) controls the stimulus, when the specified test pattern is successfully executed, the stimulus terminates the simulation raising a pass flag. Otherwise, the simulation can be terminated either by time-out or unexpected reset setting a fail flag. Each test pattern is

wrapped in if-statements. The if-statement compares a string parameter against the specified test scenario name. Thus, the test pattern is selected by assigning different value to such parameter. On top of this, the numerical parameters can always be introduced to the $I_{STIMULUS}$ module for test environment, such as supply value, bias current value, load current, decoupling cap, etc... Figure 2 (c) is an example of $I_{STIMULUS}$'s parameters.

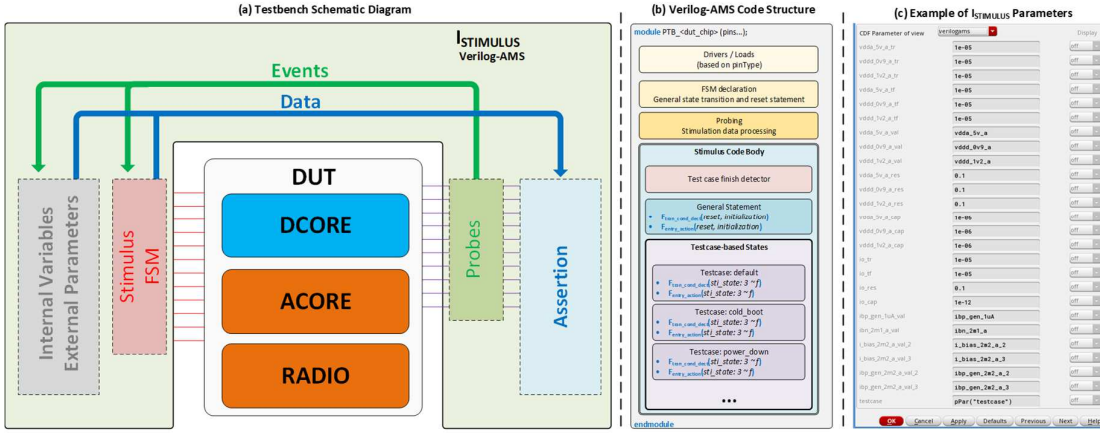

Figure 2. (a) Testbench Schematic Diagram; (b) $I_{STIMULUS}$ Verilog-AMS Code Structure; (c) $I_{STIMULUS}$'s parameters

B. *Test Setup*

For configurating the test bench and running the simulation, there is a single test with transient analysis named as "tran" implemented in the EDA tool. The Figure 3 demonstrates 4 types of variables defined in the tran test, which are explained in Table I. This setup associates the testbench configuration, test pattern, simulation environment and software running inside CPU together, which forms the framework of a programmable testbench. By assigning different value for each individual design variable, the single testbench can be tuned for all the test scenarios.

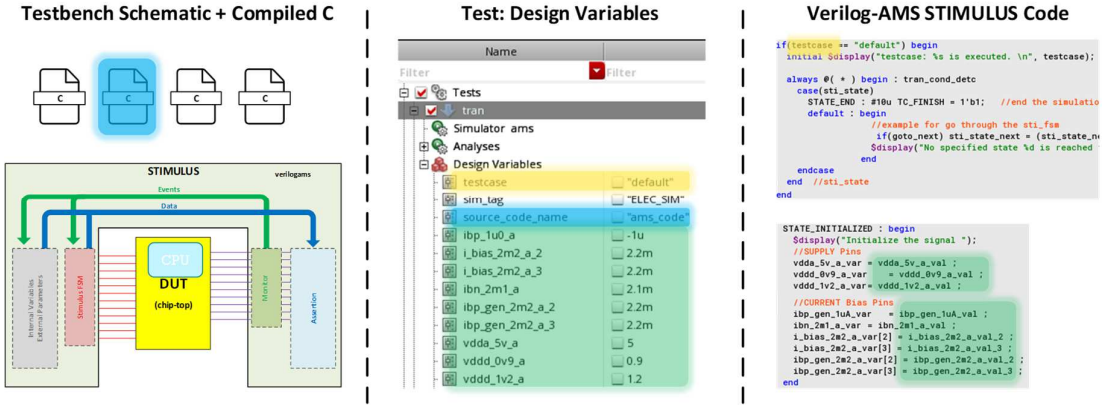

Figure 3. Design Variables in Associated among test pattern, DUT and Software

Table I. Design Variables in the Test

Variable Name	Type	Target	Purpose
testcase	string	$I_{STIMULUS}$ Verilog-AMS	Select the test pattern via a if-statement
source_code_name	string	TCL file	Loads the corresponding compiled C code into CPU. HW+SW co-verification
sim_tag	string	xrun argument	Control the compiler directives, which alter the net connection, components selection in the netlist.
vdda_5v_a vddd_0v9_a ibp_gen_2m2_a_2	real	$I_{STIMULUS}$ Verilog-AMS	Set the internal variables which control the drivers for generating the bias voltage and current at the specific value

C. *Regression Setup*

In the presented example, the regression setup is done via the corner sweep from the EDA tool. Each corner represents a test scenario as shown in Figure 4. Besides the variable described in Table I, process and temperate variables are also introduced in the corner setup, which increase the regression coverage of the analog core. With execution of all the corner simulations, one regression run for the chip is done.

Corners	Nominal	cold_boot	sleep_wake_up	current_sleep	...rent_active	supply_unplug	overload	psrr
Temperature		-40 25 85	25	-40 25 85	-40 25 85	25	25	25
Design Variables								
sim_tag		"ELEC_SIM"	"ELEC_SIM"	"ELEC_SIM"	"ELEC_SIM"	"ELEC_SIM"	"ELEC_SIM"	"PSRR"
source_code_name		"active"	"sleep_fast_wake_up"	"sleep"	"active"	"active"	"active"	"active"
testcase		"cold_boot"	"sleep"	"current_sleep"	"current_active"	"supply_unplug"	"overload"	"cold_boot"
vdda_5v_a		4.5 5.5	5.0	4.5 5.5	4.5 5.5	5.0	5.0	4.5 5.0 5.5
vddd_1v2_a		1.08 1.32	1.2	1.08 1.32	1.08 1.32	1.2	1.2	1.2
Click to add								
Parameters								
Click to add								
Model Files								
toplevel.scs		...tt top_ff top_ss top_fs top_sf	top_tt	top_ff top_tt	top_tt	top_tt	top_tt	top_tt
Click to add								
Model Group(s)		<modelgroup>	<modelgroup>	<modelgroup>	<modelgroup>	<modelgroup>	<modelgroup>	<modelgroup>
Click to add								
Tests								
tran								
...mber of Corners	1	60	1	24	12	1	1	3

Figure 4. Corner Setup inside for Regression

Figure 5 demonstrates two test cases: a) Cold-Boot and b) Sleep & Wake up, where the state of chip PMU_FSM and STI_FSM are shown together with one of the main output rails. The states of PMU_FSM and STI_FSM are described in Table II, the transition condition of the STI_FSM key states are mentioned in the Figure 5. The MEASURE state (E) of STI_FSM is used for signal processing, the expressions are automatically adjusted depending on the test cases, when in RUNNING state for Cold-Boot and Sleep Cycle for the Sleep & Wake up scenarios. This self-adaptation feature maximizes the re-usability of testbench setup, thus increasing the overall efficiency of the verification.

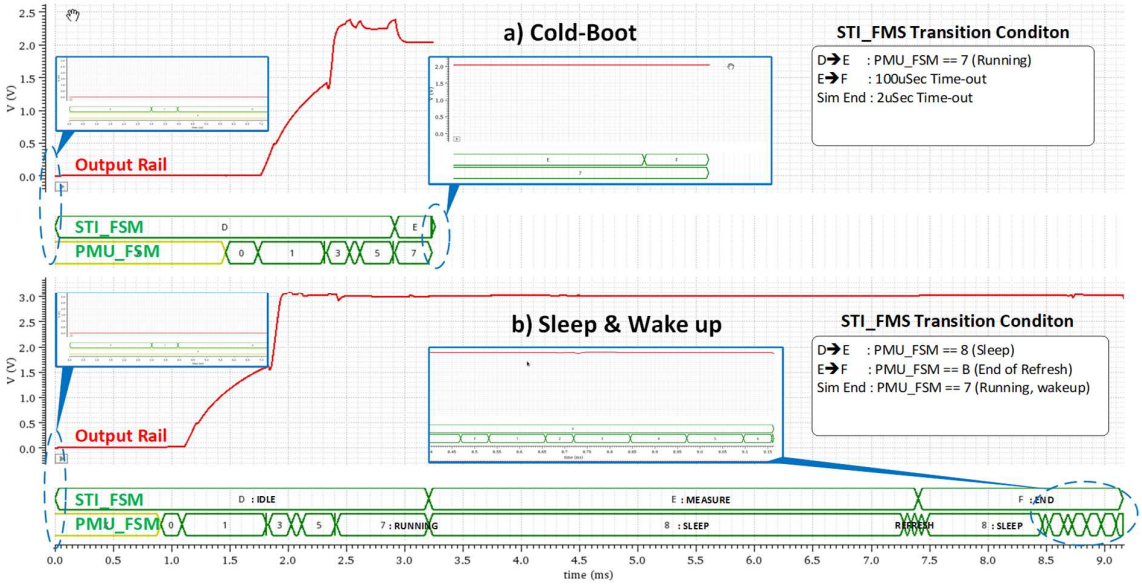

Figure 5. SVA results filtering per test scenario

Table II. PMU_FSM and STI_FSM States

STI_FSM State	Description		PMU_FSM State	Description
0	RESET		0 ~ 6	BOOTUP sequence
1	INITIALIZATION: bring up supply		7	RUNNING
D	IDLE		8	SLEEP
E	MEASURE		9 ~ B	REFRESH Sequence
F	END			

III. SELF-CHECKING

To enable the regression, self-checking without manual inspection is essential. This is achieved by using SVA. For SVA, the analog signals are converted to discrete real signals by several methods from EDA tool support and product manual[4][5][6]. An example of the analog SVA modules is presented in this section.

A. *Analog-aimed SVA module: assert_signal_within_limit*

The presented SVA module is used to monitor the complete waveform against spec envelop with an auto-pause feature. Part of the source code of assert_signal_within_limit module in SystemVerilog is shown in Figure 6, which has 5 input signals (4 real + 1 logic) and a concurrent assertion. The description for ports and internal variables is shown in Table III.

```
`timescale 1s/1ps

module assert_signal_within_limit (signal, minVal, maxVal, delay ,enable_check);

input signal, minVal, maxVal, delay,enable_check;

logic enable_check;
real signal, minVal, maxVal, delay;
• • •
property signal_within_limit;
  @(event_trigger) disable iff(~enable_check)
    enable_delayed |-> ((low_limit_ok && up_limit_ok));
endproperty
signal_within_limit_assertion : assert property (signal_within_limit)
    aFlag = (aFlag == -1)? -1 : 1;
  else begin
    aFlag = -1;
    $error("Signal value is %4.2e, and it must stay within %4.2e<x<%4.2e", signal, minVal, maxVal);
  end

endmodule
```

Figure 6. assert_signal_within_limit SVA module code snippet

Table III. Ports and Internal Variable for assert_signal_within_limit SVA module

Name	Type	Value Type	Description
signal	Port	Real	Input signal which needs to be inspected against the specific range when assertion module is enabled
minVal	Port	Real	Define the lower boundary for the spec range
maxVal	Port	Real	Define the upper boundary for the spec range
delay	Port	Real	Enable the assertion after the specified time when enable_check is set Temporarily disable the assertion for the specified time when either minVal or maxVal changes
enable_check	Port	Logic	Control signal to enable / disable the assertion module
aFlag	Variable	Integer	Internal variable to hold the assertion results, which is read out in the stimulus code expressions. 1: Pass; 0: Not Triggered; -1: Fail[a]

a. Once the assertion fails, the -1 value cannot be changed anymore.

B. *Application of assert_signal_within_limit SVA module*

An application of the assert_signal_within_limit module for inspecting the charging current of the charger block is presented in this paper. Figure 7 presents the code of the current probing, SVA module instantiation and spec calculation. Figure 8 presents a transient simulation result, where the charging current setting value (charge_current in code) is swept from min to max over time, this is reflected by the stair-case shape of the minVal (yellow waveform) and maxVal value (red waveform).

A typical analog behavior of current spike during setting transition is clearly visible for the input signal (green waveform) in Figure 8. Within the zoom-in window, it's clear that the current spike is larger than the spec maxVal. But as the assertion is automatically paused for a given time when minVal or maxVal changes, which is indicated by the gray area, the assertion does not fail. Shortly after the charging current is settled within the specified delay

time, the assertion is resumed automatically. When assertion is triggered and passed, the internal variable aFlag is expected to be 1.

```
real vbat_a_flow;
current_probe_00 #(.ppath(`hierPin(`hierDutPath,vbat_a))) Iprobe_vbat_cur (`hierDutPath.vbat_a, vbat_a_flow);

//Adjust the spec margin based on the current magnitude. add the 50% extra margin for undefined spec.
always @(charge_current) begin
  if(Icharge(charge_current) < -10e-3) Icharge_margin = 0.1; else Icharge_margin = 0.15;
end

assert_signal_within_limit Icheck_vbat_flow_cc (vbat_a_flow, (1+Icharge_margin)*Icharge(charge_current), (1-Icharge_margin)*Icharge(charge_current), 10e-6,  (stage_4 && charger_pwr_en_delay && cur_chk_en));

//function to calculate the Icharge, used for comparing the charging current during cc-mode
//It's negative as charge currnt running out of block.
function real Icharge(input real trim_code);
  if(trim_code <= 15)
    Icharge = - 0.005 * ( trim_code + 1);
  if(15 < trim_code && trim_code <= 31)
    Icharge = -(0.08 + 0.01 * (trim_code - 15));
  if(31 < trim_code && trim_code <= 47)
    Icharge = -(0.24 + 0.02 * (trim_code - 31));
  if(47 < trim_code)
    Icharge = -0.56;
endfunction
```

Figure 7. Application of SVA module to check charging current against all trim settings (code)

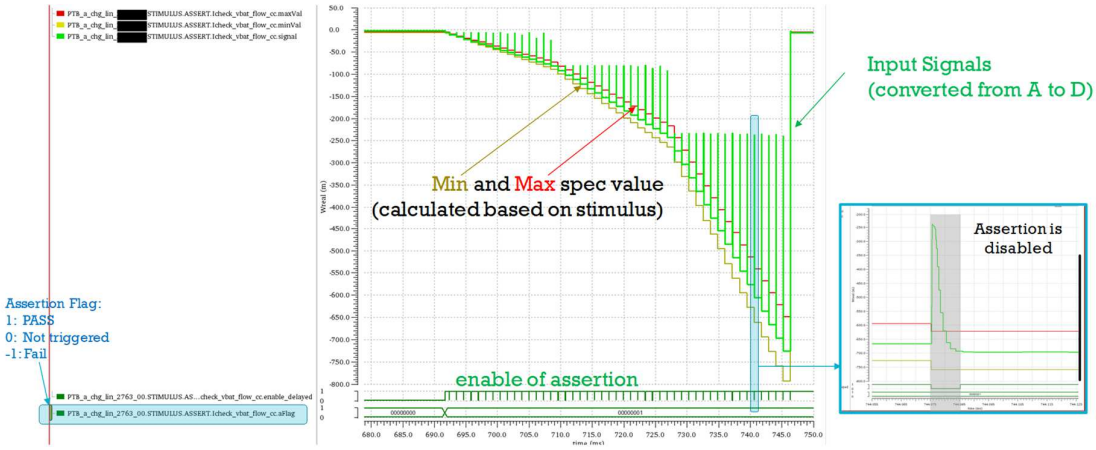

Figure 8. Application of SVA module to check charging current against all trim settings (sim results)

C. SVA module selection

Each key performance parameter of the chip, e.g. output rails, clock, reference voltage and current consumption etc, have dedicated SVA modules for monitoring. But not all the assertions are valid for specific test scenario. Such as the assertion for output rails monotonically ramping up during code-boot does not fit with power-down scenario. Thus, it's necessary to build up a selection mechanism to pick up only relevant SVA based on the testcase. In the presented example, it is done via case-statement indexed by testcase variable inside the stimulus code as shown in Figure 9. Thus, the total_pass is the only outcome which needs to be checked for the regression results. As total_pass is calculated inside the discrete domain, it can be easily picked up by any digital regression tool.

```
wire clk_wup_32k_ok ;
assign clk_wup_32k_ok = ASSERT.Ichk_f_clk_wup_32k.aFlag  == 1 &
                        ASSERT.Ichk_dc_clk_wup_32k.aFlag == 1;

always @(*) begin
  case(testcase)
    "supply_bounce"                    : total_pass = sup_cur_ok & testbus_cur_ok & rails_lvl_ok & vref_ok & clk_wup_32k_ok ;
    "supply_unplug_vbat"               : total_pass =              testbus_cur_ok & rails_lvl_ok & vref_ok & clk_wup_32k_ok ;
    "supply_unplug_vbus"               : total_pass =              testbus_cur_ok & rails_lvl_ok & vref_ok & clk_wup_32k_ok ;
    "sleep_cycle"                      : total_pass = sup_cur_ok & testbus_cur_ok & rails_lvl_ok & vref_ok & clk_wup_32k_ok ;
    "rail_discharge"                   : total_pass = sup_cur_ok & testbus_cur_ok & rails_lvl_ok & vref_ok & clk_wup_32k_ok ;
    "long_sleep"                       : total_pass = sup_cur_ok & testbus_cur_ok & rails_lvl_ok & vref_ok & clk_wup_32k_ok ;
    "battery_drain_during_sleep_no_reset" : total_pass = testbus_cur_ok & vref_ok & clk_wup_32k_ok ;
    "battery_drain_during_sleep_reset"    : total_pass = testbus_cur_ok & vref_ok & clk_wup_32k_ok ;
    "hibernation"                      : total_pass = sup_cur_ok & testbus_cur_ok & rails_lvl_ok & vref_ok & clk_wup_32k_ok ;
    default                            : total_pass = sup_cur_ok & testbus_cur_ok & rails_lvl_ok & vref_ok & clk_wup_32k_ok & rails_mono_up_ok;
  endcase
end
```

Figure 9. Example for fetching SVA results and Assertion Selection

Table IV. Brief of Regression Run for Job Policy and Run-Time

Job Policy	
Distribution Method	LBS
Parallel Num. Processors	4
Max. Jobs	16
Run-Time	
Total Run Time	9 hours 30 minutes
Num. Corner Runs	156
Avg Corner Run Time	51 minutes

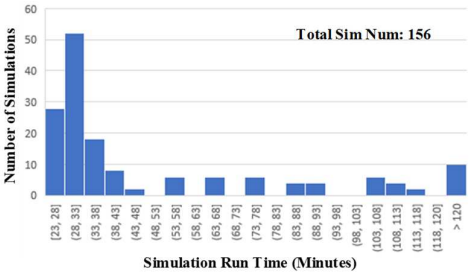

Figure 10. Run-Time Statistics

IV. JOB POLICY AND RUN-TIME

Table IV provides a brief of job policy and run time for the regression, and Figure 10 shows the run time statistics. A regression suite with 156 runs can be finished in less than 10 hours (9.5 hours), and about 73% of runs are finished within 1 hour using 64 CPUs from the compute farm.

V. CONCLUSION

This paper presented a methodology to introduce digital verification methods into the analog domain to obtain regression capability for analog scenarios. Verilog-AMS is used as stimulus instead of discrete components to drive and load the DUT. It encapsulates a fully programmable testbench controlled by EDA tool for both testbench configuration and test pattern execution. The self-checking is implemented using analog-centric SVA module significantly reduces the workload for the verification engineer and enable more iterations for verifying the analog circuits at system-level. The test scenarios are defined in a corner sweep, and regression has been optimized to the point of "push of a button", the regression summary report is updated automatically, which gives a clear picture of current status of the analog regression run. This flow can also be applied downstream to the AMS sub-systems such as an ADC or digitally-assisted power management unit comprising of digital-analog interaction.

Reference

[1] A. Caicedo, and S. Fritz, "Enabling Digital Mixed Signal Verification of Loading Effects in Power Regulation using System Verilog User-Defined Nettype," DVCON EUROPE, Munich, October 2019

[2] L. Balasubramanian, P. Sundar, and T.W. Fischer, "Assertion Based Self-checking of Analog Circuit for Circuit Verification and Model Validation in SPICE and Co-simulation Environments," DVCON

[3] "Verilog-AMS Language Reference Manual v2.4", Accellera System Initiative, USA, 2014.

[4] "Unified VerilogAMS monitor to probe electrical/real/wreal/logic signals in a test bench", Cadence Support, https://support.cadence.com/apex/ArticleAttachmentPortal?id=a1O0V000009ESqBUAW&pageName=ArticleContent

[5] "Smart VerilogAMS current monitor to handle change in design configuration during Design and Verification cycles", Cadence Support, https://support.cadence.com/apex/ArticleAttachmentPortal?id=a1O0V000009ESqQUAW&pageName=ArticleContent

[6] "Spectre AMS Designer and Xcelium Simulator Mixed-Signal User Guide 19.09", Cadence, September 2019