2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Automotive RADAR Bitfields Verification to support Validation of Silicon bring-up

Amol Dhok and Paulraj M K

NXP Semiconductors, Bengaluru, India

***Abstract* - RAdar Detection And Ranging (RADAR) is becoming a common sensor part in Automotive industry for ADAS application.**

Faster silicon bring-up requires pre-verified register bitfield (functional register's fields) settings and lots of background preparation of SW APIs. The final bitfield settings are provided by functional block designers to the SW API team. We can see a gap in terms of correctness of block configuration settings given by design owners to SW API team for Silicon Validation. This gap is because of un-verified block settings for a functional scenario being used in the silicon validation, where multiple functional blocks can work together to form a usecase.

To bridge gap, the block configuration settings are verified first using UVM based verification environment. The block configuration settings written in a pre-defined format is converted to System Verilog compatible format using Python script. This generated System Verilog code is compatible with RADAR sensor UVM verification environment and it is run as RTL simulation. Once the scenarios related to Tx, Rx, RxBIST, Chirps etc. are done successfully then these block configuration settings can be considered as the golden and ready to use for Silicon first day bring-up. The flow is block bitfield settings -> Python script -> generated System Verilog code -> RTL Simulation with the UVM environment and signoff the block configuration settings.

Design, Verification, Validation and SW Development teams in our industry are using tools, flows and methodology which are not interoperable and leads to rewrite of code instead of reuse of code across the domains. The real benefit of this way of working is in reduced debug time and faster silicon bring up. The proposed methodology is to capture the configuration settings in a generic syntax which can be translated with the help of automation scripts to different teams customized flows and language/tool needs.

Keywords—RADAR, ADAS, Automotive functional Safety std: ISO26262, UVM, Python, System Verilog, bitfields.

I. INTRODUCTION

Radar sensors are a fundamental building block of highly autonomous driving (HAD) vehicles. They are typically more robust than cameras to adverse weather conditions, more cost effective than LiDAR, provide a better velocity and distance resolution and offer a view of the surrounding based on different sensing technology.

Automotive radars detects the speed and range of objects in the vicinity of the car. An automotive radar consists of a transmitter and a receiver. The transmitter sends out radio waves that reflects off an object back to the receiver, determining the objects' distance, speed and direction.

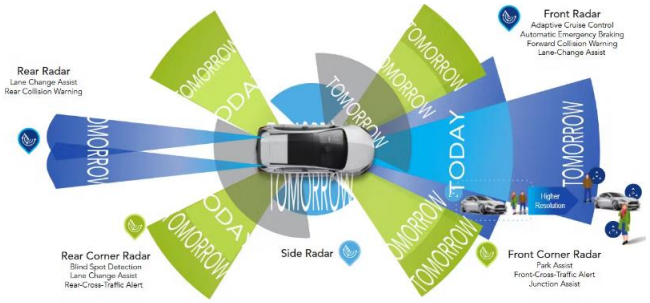

Figure 1: RADAR sensing the objects, Reference [3].

II. RADAR SENSOR VERIFICATION

Verification of RADAR sensor includes specifying and simulating multiple Analog and Digital blocks. Most of the blocks are specific to RADAR application, which doesn't support standard way of verification using VIPs for accurate and fast verification. It includes verification of many safety features as per ISO26262 compliance automotive standard. The complete RADAR sensor verification includes lot of randomization of chirp profile parameters. Analog part equally includes setting for LDOs, Clock generators, LO Interface, Rx, Tx, Chirp generator etc. The sensor usecase and full chain verification is done with constraint random verification using UVM methodology.

The RADAR transceiver operates at 76 to 81 GHz frequency, The transceiver includes multiple transmitters and receivers.

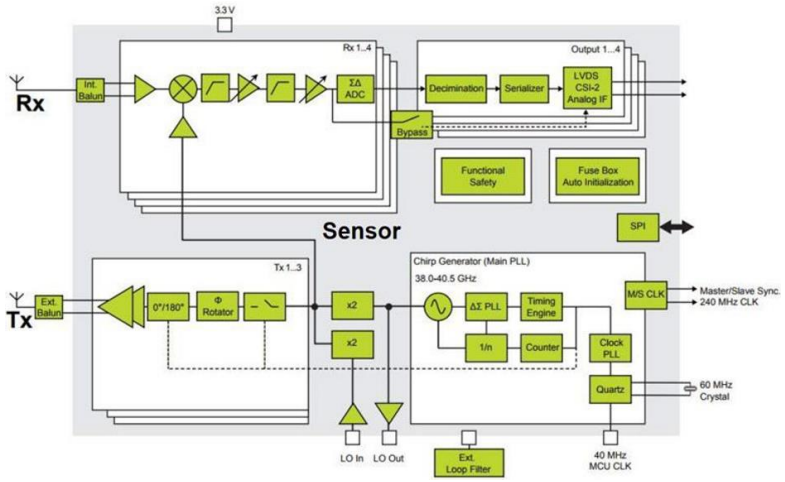

Figure 2: Automotive radar system architecture showing RF, analog, and digital subsystems, Reference [4]

In the figure2, each block configuration setting is the key part for RADAR sensor initialization and to bring-up the chip in Validation. The block configuration settings are given by designer towards the end of the project.

A. Verification and Validation Phases

The **Verification phase** includes;

i) Write the Test Specification for the given requirements, maintaining the horizontal linking.

ii) Implement the testcases in System Verilog and run the simulation for functional verification, which includes full-chain or scenario base verification

iii) Gate level simulation using SDF (Standard delay format) and netlist and VCD (Value change dump) for power analysis.

The **Validation phase** includes;

i) Initial bring-up for 3-4 weeks to make sure all the clk, reset, module access etc. are functional.

ii) Full chip validation with multiple scenarios.

iii) Silicon characterization

The block configuration settings for each individual blocks is verified with the limited block level testbench environment. This often leads to the wrong settings or missing of scenarios base setting, where multiple blocks are stitched together. There is no detail pre-verification of block settings before delivering to the SW API team.

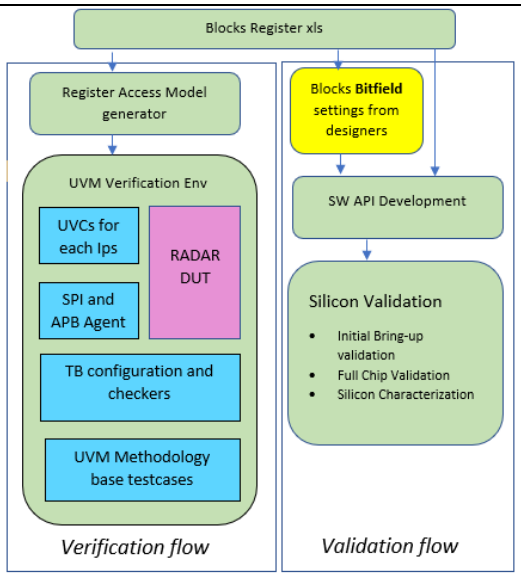

Figure3: RADAR sensor Verification &Validation flow

The block bitfields are the Register's field which are programmable to configure the design. These registers description is specified in xls, which is the common source for Verification and Validation flow. The example of a register is shown in table1.

Register Name	Field Name	Description	Offset	Width	Reset Value	Access
DYNAMIC_POWER_CONTROL_DELAY		**This is primarily to avoid any race conditions within the system**	**0x108**	**32**		**rw**
	DY_PD_DELAY_VAL_SEQ	This control delays the powerdown of the modules which are configured for chirp sequence based powerdown	0	16	0x28	rw
	DY_PD_DELAY_VAL_CHIRP	This control delays the powerdown of the modules which are configured for chirp based powerdown	16	16	0x28	rw

Table1: Example of Register fields i.e. bitfields

B. Verification and Validation Gap

The ISO26262 is an international functional safety standard of electrical and electronic systems that are installed in Car. It defines guidelines to ensure that automotive components performance and safety. The V&V development guidelines are as per figure3,

Sr No	Product development at the hardware level
1	General topics for the product development at the hardware level
2	Specification of hardware safety requirements
3	Hardware design
4	Evaluation of the hardware architectural metrics
5	Evaluation of safety goal violations and to random hardware failures
6	Hardware integration and verification

Sr No	Product development at the software level
1	General topics for the product development at the software level
2	Specification of software safety requirements
3	Software architectural design
4	Software unit design and implementation
5	Software integration and verification
6	Testing of the embedded software

Figure3: ISO26262 standard Verification & Validation development guidelines, Reference [2]

The missing link is a verification gap between Verification and Validation. The challenges posed by Analog blocks are need of proper hardware initialization routines for voltages, current values, calibration routines to program VCO start frequency, configuration of functional routines like Tx, Rx, Chirp etc., need for proper safety checks and so on. These analog challenges cause silicon bring up time to increase exponentially compare to digital logic bring up.

C. Bridging Verification and Validation

The motivation towards bridging the gap is to solve the V&V challenges towards hardware Verification and Validation of silicon bring up with software.

The source for Verification and Validation is same as block registers xls, which includes the configurable register map for each block. Though the source for Verification and Validation is same but the final delivery of block bitfields configuration by block designer to the SW API team is not pre-verified in simulation.

With block configuration settings written in a specific format, the Python script converts the provided bitfield settings into UVM compatible System Verilog code/test. This converted SV code is included in the UVM test and we can run the testcase as normal System Verilog test. This test or scenario is verified with the help of Verification components, Checkers and Assertions available in the verification testbench environment. The feedback of the block settings is shared with the block designers well in advance of Silicon availability.

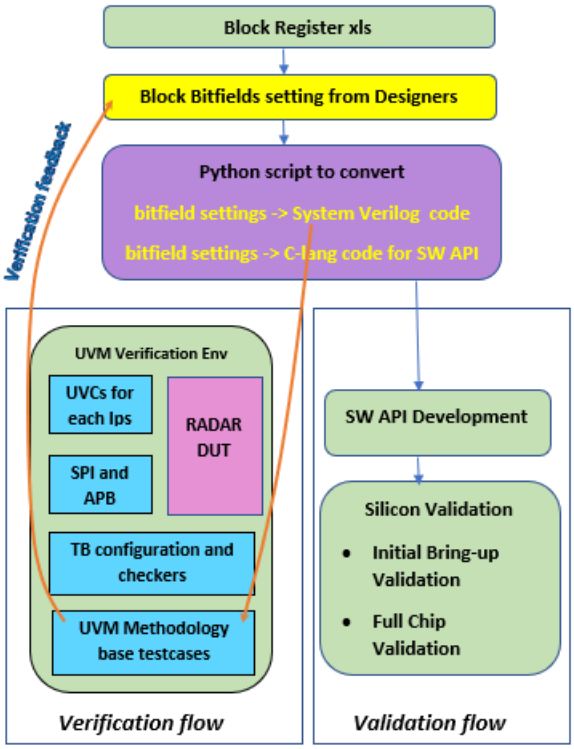

Figure4: Multiple blocks Bitfield Verification

III. BITFIELD VERIFICATION PROCESS

Bitfield Verification process consists of

- Bitfield coding in a specific format
- Conversion of .bf file to .sv file (Bitfield to System Verilog file) by Python script
- Include the newly generated .sv file as a test and run the simulation to verify the settings.

A. Block level Bitfield coding format

The bitfield (.bf) files are created by block designers as RADAR sensor configuration of different blocks. These files are delivery to SW API team to develop SW Applications. The block settings should be in the specific format for all the blocks. This is pre-requisite for Python script to convert the .bf file into compatible .sv file.

The format of .bf file is a simple assignments of values to the bitfields of the individual register fields. For example;

BLOCK_1.REGISTER_XYZ.BITFIELD_A=0x1;

BLOCK_2.REGISTER_PQR.BITFIELD_B=0x0;

The scenarios for RADAR sub-system is created using different bitfield files. The sub-system initialization .bf file is common in all the scenarios.

The example of frequency-modulated continuous wave radar i.e. FMCW RADAR transmission usecase is implemented by the combination of init.bf + chirp_pll.bf + loi.bf + tx.bf + rx.bf with loopback enabled.

init.bf -> Initialization of sub-system with power on, configure the clocks, enabling the blocks etc.

chirp_pll.bf -> configuration of chirp profiles, stdby2on, calibration, start frequency, bandwidth etc.

loi.bf -> configure the local oscillator.

tx.bf -> enable ldo and power on tx

rx.bf -> enable ldo and power on rx.

Similarly, different scenarios can be implemented using combinations of .bf files

B. Python Script: Conversion of .bf to .sv

The accuracy of .bf setting correctness is verified by running the RTL simulation before delivering the .bf files to SW API team or to Validation team. This conversion is done with the help of Python script.

Python script is used for converting .bf file format to .sv file compatible RAL model writes and read calls. The pre-verified .bf files are delivered to the SW API team for the firmware development.

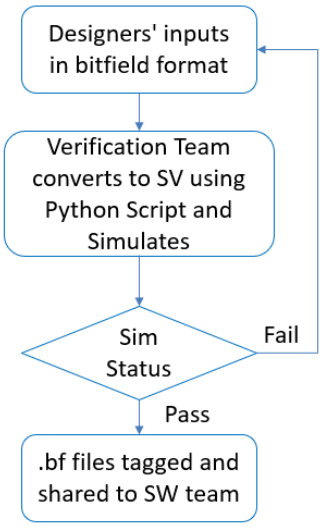

Figure6: Flow chart .bf simulation

BF (Bitfield File) code:

```
BLOCK1.REGISTER1.BITFIELD_BLOCK1=0xF;
sleep_bv(BLOCK1.REGISTER1.BITFIELD_FLAG_OK,1);          // Poll BITFIELD_FLAT_OK=1
sleep(100);                                             // Delay of 100us
```

Python script converts the above code as SV RAL model compatible write and read format.

SV (System Verilog) code:

```
reg_model.block1.REGISTER1.BITFIELD_BLOCK1.set('b1111);
reg_model.block1.REGISTER1.update(status);
sleep_bv("block1","REGISTER1"," BITFIELD_FLAG_OK",'d1);     // Poll BITFIELD_FLAT_OK=1
#100us;                                                    // Delay of 100us;
```

The pre-verified bitfield code is converted to C-lang for SW API development.

C-lang code:

```
writeSpi("BLOCK1.REGISTER1.BITFIELD_BLOCK1",0xF);
sleep_bv(BLOCK1.REGISTER1.BITFIELD_FLAG_OK,1);          // Poll BITFIELD_FLAG_OK=1
sleep(100);                                             // Delay of 100us;
```

C. RTL simulation of .SV testcase

The generated SV testcase/file is the sequence of different block configuration settings, which is called from the UVM tests and will be simulated same as a normal testcase simulation. On the proper review and expected functionality checks, the final .bf files are delivered to SW API team.

UVM SV testcase code:

```
class radar_chirppll_76G_test extends radar_base_test;
…
    `include "radar_chirppll_76G.sv";
…
endclass; //class radar_chirppll_76G_test
```

The generated .sv file is a simple `include in the System Verilog test. This file is the .sv generated from multiple .bf files, which described a scenario or usecase. It is a single flat .sv file generated from multiple .bf files, for example init.bf, mcgen.bf, tx.bf, rx.bf, chrippll.bf etc.

The issues found from the RTL simulation are like pon_ls, ldo enabling was not done, no proper configuration of chirp parameters, issues in atb (test bus) programming and correction in the configurations of the bitfields settings of different modules.

The same testbench verification environment is used for both UVM based test and generated .bf -> .sv base test. There is no change in the checker/assertions or any other verification mechanism.

D. Results

Correct-by-Construction is an approach to incrementally create correct programs for SW API development. The bitfield verification flow has been successful in addressing the challenges of an automotive RADAR design in an effective, efficient and robust manner etc.

- Silicon bring-up time for initialization of RADAR sensor is reduced to 1 week compare to 3 to 5 weeks of plan schedule.
- SW APIs ready with pre-verified configurations before the silicon bring-up makes the team to close the activities faster than the schedule plan. Up to 25% time reduction in validation test closure.

E. Re-usable Stimulus

The pre-verified bitfield configuration is re-usable in

- SoC level by calling the sub-system level sequences directly from SoC level.
- Emulation level by using the bitfield configuration for FPGA programming.
- Validation to bring-up the Silicon in short time.
- SW API development.

IV. CONCLUSION

A unified method is the need of the hour to enable portability of configuration settings across the teams and enable reuse across domains. The sensor configuration bitfields before delivering to SW API team, can be converted into .sv format to verify using the existing UVM verification environment. The re-usability of UVM verification environment is done to the fullest in this case. The important points to consider is the block bitfield setting's pre-required format and the Python script to convert it into System Verilog format. The link between Verification and Validation is successfully done using a Python script.

On the day one of Silicon bring-up, the pre-verified RADAR sub-system configuration setting helps in successful execution of RADAR initialization and scenarios check.

Also the .bf -> .sv simulation is useful for the failure debugging by performing the RTL simulation of the RADAR sub-system related Validation tests.

V. ACKNOWLEDGMENT

The authors would like to thank the RADAR team at NXP Semiconductors. We would like to express our gratitude to Salvatore Drago and Manikandan Panchapakesan for their initiative on bridging the gap between Verification and Validation.

VI. REFERENCES

[1] NXP Introduces Advanced Automotive Radar One-Chip Family for Next-Gen ADAS and Autonomous Driving Systems

[2] International Organization for Standardization, "ISO 262621:2011 -- Road vehicles -- Functional safety," [Online]. Available: https://www.iso.org/standard/43464.html.

[3] https://www.electronicdesign.com/markets/automotive/article/21257737/electronic-design-nxp-unveils-new-onechip-radar-for-autonomous-vehicles

[4] Environment-in-the-Loop Verification of Automotive Radar IC Designs - MATLAB & Simulink (mathworks.com)