Advances in RF Transceiver SoC Verification: A Walk-Through over a 2.4 GHz Multi-Modal Integrated Transceiver Verification Cycle

Charul Agrawal[1], Ashwin Vijayan[2], and Jakub Dudek[3]

[1]Senior Design Engineer, Analog Devices, Inc., Bangalore, India
[2]Design Engineer, Analog Devices, Inc., Bangalore, India
[3]Design Verification Engineer, Analog Devices, Inc., Wilmington, MA, USA

Abstract

To realize the vision of a shorter time-to-market and bug free silicon with exceedingly complex SoCs, novel verification strategies have to be adopted. In this regard, we showcase the advances in an RF transceiver SoC verification scheme by discussing a variety of novel simulation and modeling methodologies bridging the gap between the design, verification and firmware teams. We focus on a verification environment capable of randomizing all packet parameters across protocols, introducing blockers, verifying various mixed-signal time critical loops etc., thereby, subjecting the design to more realistic and aggressive scenarios. The conventional co-simulation based full-system simulation of RF transceiver SoCs is a challenging problem because high-frequency RF data result in excessive run time. The firmware oriented system verification, dynamic radio configuration, datapath specific DV solutions, MATLAB® independent analysis are some of the other techniques discussed. These were successfully employed in the verification of a 2.4 GHz, ultra-low power, multi-core, integrated radio transceiver SoC with best in class sensitivity that supports BLE and proprietary modes.

Keywords: *UVM, Multi-Protocol Wireless SoC, RF Agents, Blocker Modeling, Mixed-Signal, RNM, UVM-C Communication, Firmware development*

I. Introduction

As the complexity of systems on chip (SoC) grows, the effort spent on verification increases significantly. The conventional verification strategies are unable to ensure the functionality of the design fully and therefore the product quality cannot be guaranteed. Verification has turned into a bottleneck of an SoC product development cycle and takes about 40% to 60% of the overall effort. To mitigate this bottleneck, several advances have been made in the EDA tools, languages, methodologies etc. A successful verification endeavor involves a flexible yet powerful DV environment capable of handling multifaceted DV scenarios. Therefore, architecting the verification environment itself demands a huge and intelligent effort.

A typical RF SoC contains RF and analog blocks, processor or controller cores, memories and peripherals, a number of mixed signal control loops, and data processing blocks across analog and digital [1]. Hence, the verification of RF SoC is extremely challenging both at block level and system level. A comprehensive verification of a transceiver necessitates the generation of packets with varying power, payload length, frequency offset etc. The DV environment should be capable of generating signal in the frequency band of interest (wanted signal) along with blockers (unwanted signal at different frequency offsets) for realistic verification of the Rx path. Moreover, the black-box approach for the RF analog blocks in the verification environment does not verify the critical control loops such as automatic gain control (AGC), dc offset and frequency corrections, etc. exhaustively. Unlike other mixed signal systems, co-simulation of continuous time domain and event driven simulations is not an option for extensive verification of a multi-protocol, high frequency transceiver with time critical loops due to microscopic

time-resolutions [2]. Furthermore, use of system software packages, such as MATLAB®, for stimulus and post-processing analysis is inevitable in a transceiver verification cycle. Given the cost of license, limited set of stimuli and elevated execution time, the dependency on such softwares needs to be curtailed.

This paper walks through the advances in the various phases of an RF SoC verification. These advances in verification methodologies have eased many of the challenges traditionally associated with the RF integrated transceiver SoCs. The flexible constrained-random multi-modal simulation environment described in this paper has been successfully leveraged for firmware development early in the project cycle in parallel with analog and digital design tasks.

The paper is organized as follows. Section II describes about the tool developed for block level verification. Section III details about the system level environment with emphasis on the testbench architecture, packet generation, system level randomization, firmware (FW)-like DV setup etc. Section IV concludes the paper with coverage and silicon results and describes some of the ideas for future work.

II. Block Level Verification

The first phase of SoC verification effort is to verify the blocks individually. An integrated transceiver SoC's block level verification is hugely challenging due to numerous signal processing/datapath designs both in Tx and Rx paths. Fig. 1 shows the block diagram of a typical receiver PHY layer. There are a whole bunch of signal processing designs that have to be verified. Based on our prior experience with signal processing blocks, we identified that automating the testbench generation with in-built tests, stimuli, checkers that can be run out-of-the-box, can substantially reduce the overall verification time. It also aids the design engineers to perform sanity checks during the design development stage, which is usually not possible until the first-cut testbench is up and running.

Incorporating the stimuli such as impulse, step, tones, modulated signal, noise etc., checkers like threshold-amplitude, frequency monitoring and features such as SNR calculation, etc. is imperative for a signal processing DV environment. Typical approaches follow heavy use of file based I/O for stimulus and post-processing which demands system software packages such as MATLAB®. This can prove to be cumbersome, costly and acutely impede the feasibility of extensive UVM randomization capabilities. We need a solution that decouples the DV effort from MATLAB®, resulting in significant savings in license requirement and simulation time.

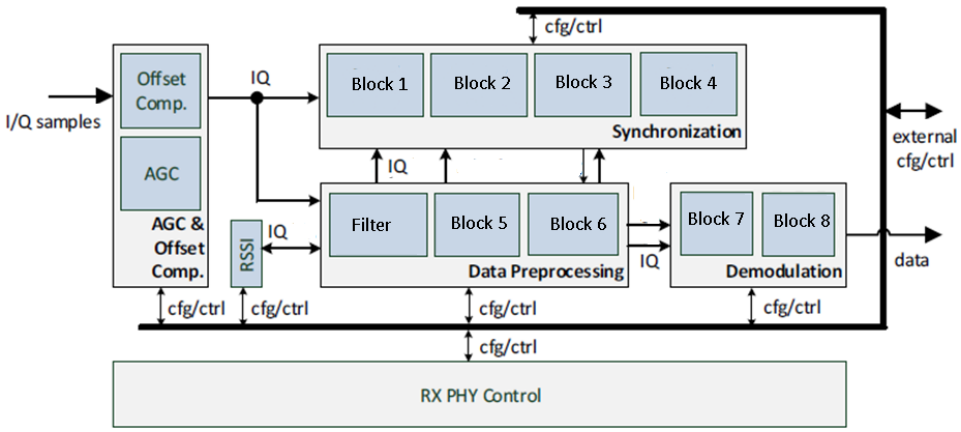

Fig. 1: Block diagram of a typical digital datapath in a receiver PHY layer

We developed a tool namely "Veri-DSP" to implement the aforementioned features. In general, most signal processing/datapath blocks come with a typical interface in which there is a clock, reset, and set of control and data signals. Veri-DSP is equipped to generate the aforementioned stimuli within the UVM sequences itself. Veri-DSP is integrated with a bunch of Python scripts that can perform typical post-processing tasks such as plotting frequency response, FFT computation, and SNR calculation. Thus, Veri-DSP cuts the requirement of MATLAB® licenses heavily. Veri-DSP also creates file based input sequence and signal dumping functions which enable seamless interaction with MATLAB® if needed. Furthermore, the tool also gives C-DPI feature which enables the user to generate stimulus or perform checks using C model effortlessly.

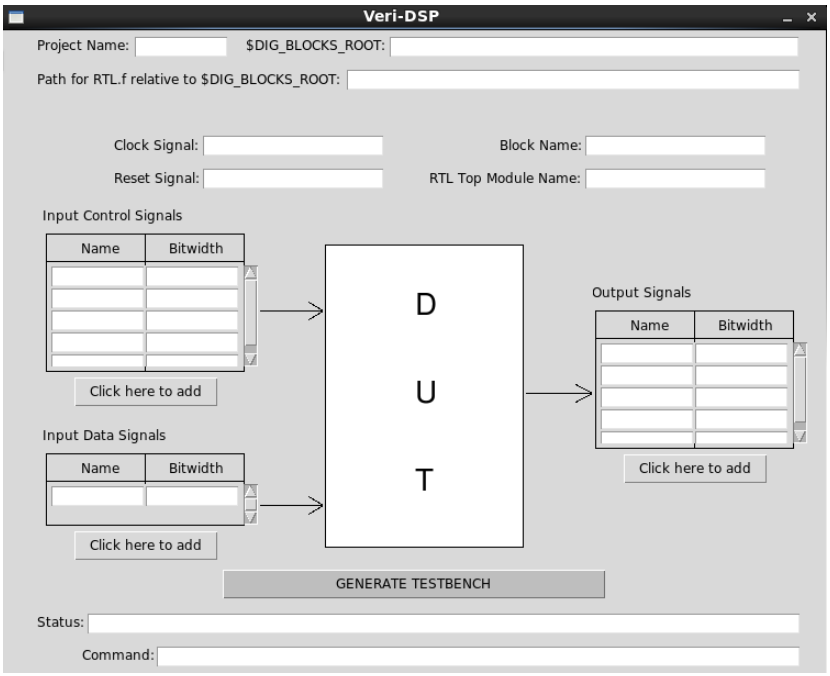

Fig. 2: The GUI environment associated with Veri-DSP

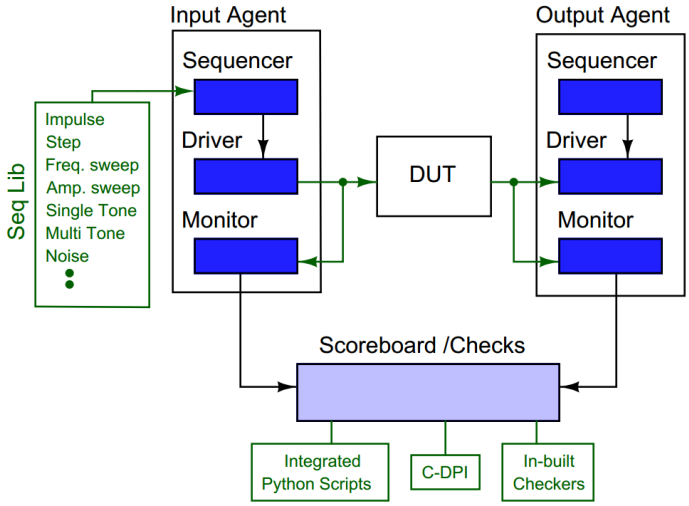

Fig. 3: The architecture of a TB generated using Veri-DSP

Fig. 2 shows the GUI environment associated with Veri-DSP wherein the user only needs to enter the details of design under test (DUT), input signals, output signals and RTL files to generate fully functional testbench at the *click of a button*. Fig. 3 shows the architecure of a testbench generated using Veri-DSP. All the essential connections such as monitor-DUT interface connection, driver-DUT interface, monitors to scoreboard etc. are automatically created. The design is also instantiated at the testbench top and will be connected to the interfaces automatically. Thus, Veri-DSP brings down the development time of a first-cut compiling TB to almost zero! DV engineer needs to only implement the scoreboard checks and manipulate the already existing sequences to create required scenarios. Furthermore, a beginner with minimum knowledge of UVM can kick-off the DV process using Veri-DSP. The system level integration of these test benches is easier as they all have a similar pattern.

Veri-DSP was used for the verification of signal processing blocks such as Digital Channel Filter, Interpolator, Received Signal Strength Indicator (RSSI), Timing Sync., Frame Sync., Demodulator etc. This encompasses a wide variety of signal processing blocks which require different kind of stimuli and checks. A considerable reduction in the overall verification time was achieved for all the blocks. As an example, the entire block-level DV exercise

for a 31-tap, fully programmable FIR filter was completed in 4 weeks by a beginner which includes verification planning, cycle accurate-modelling, test case development, time and frequency domain testing and coverage closure. A regular approach would have required additional two to three weeks per block which translates to around 20-25 weeks for verifying numerous blocks in digital base-band (DBB).

III. System Level Verification

Traditionally for Rx DBB verification, the baseband received signal is generated by a stimulus generator in C or MATLAB® and passed on to the simulation environment that adds interferers or blockers[3]. Whereas, for Tx DBB the output is stored and post-processed in MATLAB® to compare the results against the specifications, generate pass/fail and for generating spectral plots [3]. With complex analog-digital loop interactions, such an environment does not prove to be effective for comprehensive verification. In this section, we describe how we used the randomization capabilities of UVM to extensively verify an RF SoC in a faster, MATLAB® independent manner.

A. Testbench

In this section, we describe the environment used to verify the RF SoC at the system level. Fig 4. illustrates the environment. The system level verification was digital-centric where the entire SoC from pads down to RTL

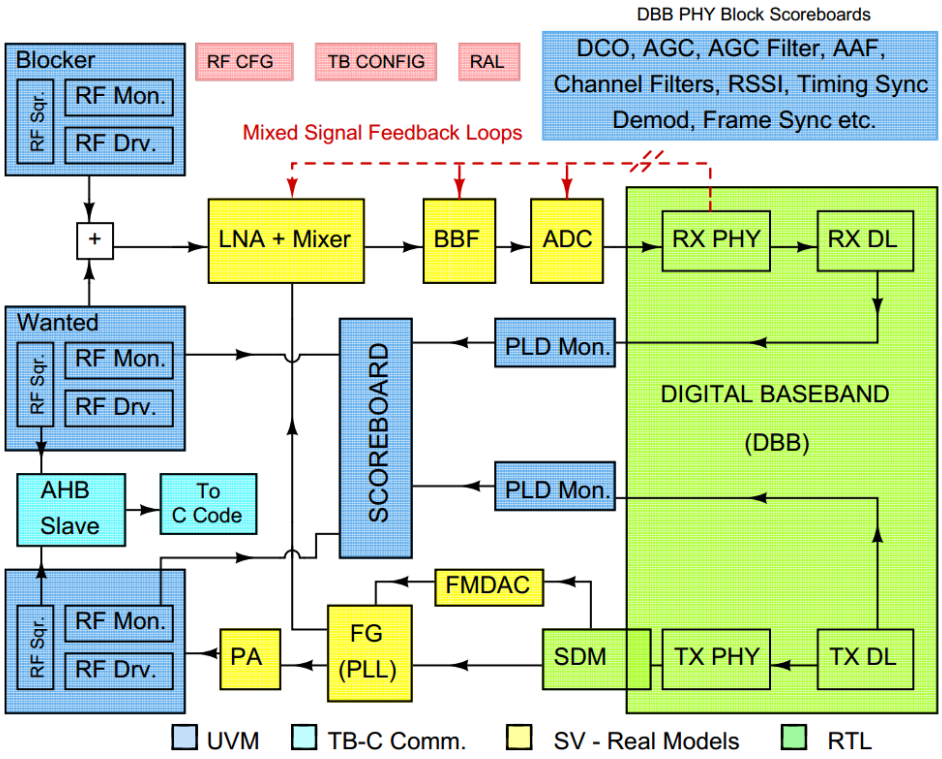

Fig. 4: A high level view of the radio testbench setup

was represented in some form of Hardware Descriptive Language (HDL) like SV-RNM, Verilog/VHDL and a testbench was built around it. The DV environment consists of a pair of identical RF agents, each connected to the power amplifier (PA) and Low Noise Amplifier (LNA) ports of the transceiver. The RF agent connected to the PA (referred to as PA agent henceforth) port is passive and serves as a monitor for the device's Tx path. The RF agent on the LNA port (referred to as LNA agent henceforth) is active and generates stimulus for the device's Rx path. The environment also has two payload monitors that are bound within the Tx and Rx DBB wrappers. Packets are collected at this level and sent to the scoreboard for comparison with the PA/LNA agents (Refer Code Snippet 1). The payload monitors observe signals such as start of frame, end of frame and data going in and out of the DBB to construct received packets or packets intended for transmission.

Code Snippet 1: Rx Packet Payload Check

```
foreach(lna_q[i]) begin
// ..............Some code ............
if (!lna_q[i].compare(rxpld_q[i])) begin
`uvm_error(get_type_name(), $sformatf("Miscompare at Payload monitor,
   packet %0d", i))
foreach(lna_q[i].payload[j]) begin
if(lna_q[i].payload[j]!=rxpld_q[i].payload[j])
`uvm_info(get_type_name(), $sformatf("[%d] %x %x", j, lna_q[i].payload[j],
   rxpld_q[i].payload[j]), UVM_LOW);
end
break;
end
end
```

Variable	Description
adc_offset_i	Input offset in mV for I channel in ADC
adc_offset_q	Input offset in mV for Q channel in ADC
bbf_offset_i	Input offset in mV for I channel in BBF
bbf_offset_q	Input offset in mV for Q channel in BBF

TABLE I: RF Configuration class

The environment contains an additional RF agent to generate blockers for the device's Rx path. This active agent is connected to the LNA port of the deivce which added one more dimension to the verification and was used to verify the interference (blocker rejection) performance. More blocker agents can be added to introduce multiple blockers simultaneously.

RAL (register access layer) is used in the environment to collect the state of any memory mapped register (MMR) in the system so as to create checkers that verify consistency with intended programming. For example, if the measured bitrate at the PA is 1 Mbps, this will be compared with the relevant MMR to check that the intended mode of operation is indeed 1 Mbps.

Finally, a simple RF configuration class exists to control certain static settings in the environment (Refer Table I). Currently, it is only used to set DC offsets in the analog model. It can also be used for process variation randomization of analog models. The class can be randomized, yielding different scenarios over multiple seed runs.

One of the standout features of our verification environment is the use of UVM agents for stimulus generation instead of MATLAB® generated file-based stimuli. We were able to randomize all the parameters listed in Table II by encapsulating them in one transaction called as transmission. A transmission is defined as the entire bit stream from the time the PA ramps up to the time the PA rams downs. A transmission comprises of one or more packets which is another transaction consisting of packet_type, address, header, payload, crc and header_crc. Packets are not instantiated by the user, rather they are part of the transmission class and are instantiated and randomized within the transmission.

The transmission class also contains serialize() and packetize() functions which translate between the high level information (preamble, packet header and payload, etc) in to a stream of bits and vice versa. Packets are randomized in the post_randomize() function of the transmission class. The length of the packet queue is randomized with other transmission class variables. Once this length is set, the post_randomize() function traverses the length of the queue and creates each packet according to the local randomized values of header_size, pld_min, pld_max and packet_type (Refer Code Snippet 2).

Variable	Description
pkt_q	Queue of packets
preamble	Dynamic array of bytes, contains the preamble.
postamble	Dynamic array of bytes, contains the postamble.
midamble	Dynamic array of bytes, contains the interpacket-amble
rampup	Use a ramp up of the signal strengths for Tx from testbench (DUT Rx)
rampdn	Use a ramp downbefore of the signal strengths for Tx from testbench (DUT Rx)
dBm	Input signal power in dBm
impedance	Impedance used for amplitude calculation, typically set to 50 ohms
channel	Channel to be used for transmit or receive
protocol	Type of protocol used for the test
MI	Selector for Modulation index
BITRATE	Data rate
direction	Direction of transfer
f_err	Frequency error in carrier
Header_size	Size of header used in the packets of this transmission
pld_min	Minimum size of payload, used in packet constrained
pld_max	Maximum size of payload, used in packets constrained.
pld_crc_size	Payload crc size
crc_init	Initial value of crc calculation for packet payload
crc_poly	Polynomial for payload crc calculation
hcrc_init	Initial value of crc calculation for packet header
crc_poly	Polynomial for header crc calculation
whitening	Enable whitening of data
spacing	Space after packet has completed, in symbols
Packet_type	Type of packets used for this transmission
Address	Packet address

TABLE II: Randomizable variables in the transmission class

Code Snippet 2: Creation of array of packets inside transmission

```
1  for(int i = 0; i<pkt_q.size(); i++) begin
2    pkt_q[i] = packet::type_id::create("", null);
3    void'(pkt_q[i].randomize() with {has_hcrc == |h_crc_poly; header.size ==
        header_size; payload.size inside {[pld_min:pld_max]}; packet_type ==
        local::packet_type;});
4    pkt_q[i].crc = calc_crc({crc_init,crc_poly}, pkt_q[i].payload);
5  end
```

B. System Level Randomization

Requirement: For any radio sub-system, the DBB is required to be configured according to the transmission parameters such as packet address, protocol etc. With our environment randomizing all transmission parameters, communication between the UVM test and code running on the processor was mandated to not lose out on our randomization capability. We have used C language for writing the codes running on the processor. Moreover, we also faced the issue of random number generation (rand function) in C that is not reproducible with the same seed and obstructs in the debug process.

It is important to note that the transmission generated in the UVM testbench are dynamic objects that are not guaranteed to exist at any given point in time (as opposed to a verilog module which exists for the entire simulation). As such it is possible to be in a situation where nothing is returned from the C code get() calls. It is therefore

important to setup proper handshaking via sync function between C code and system Verilog in order to obtain valid information from the testbench.

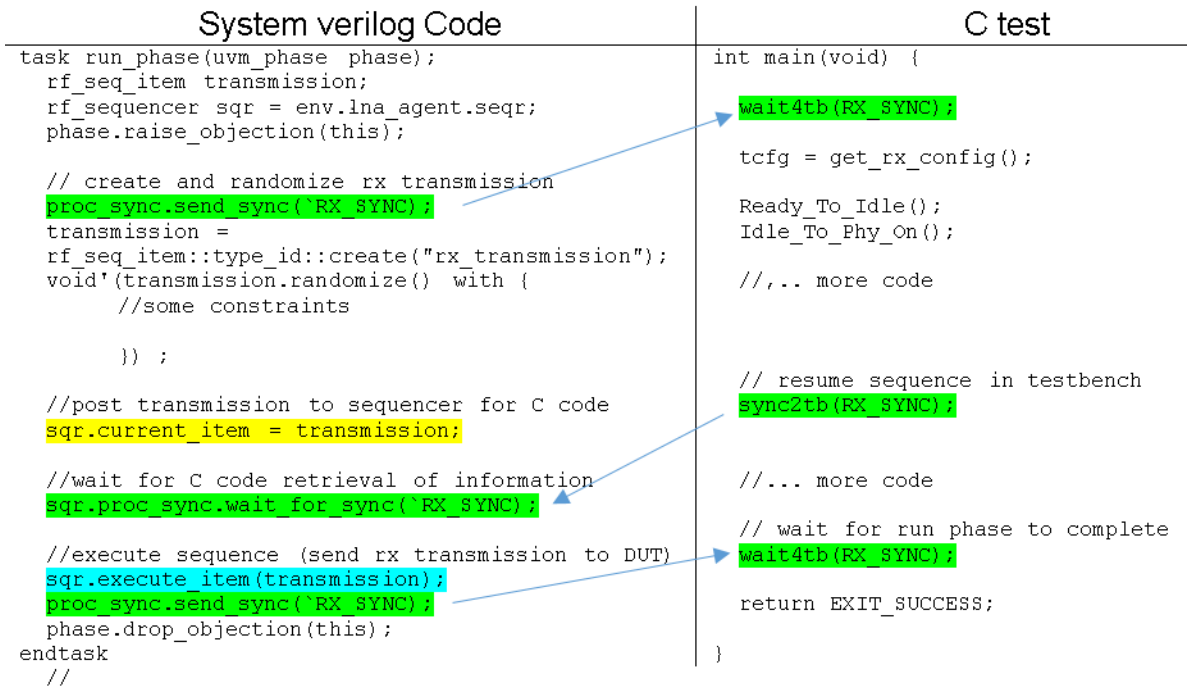

Fig. 5: Testbench - C handshaking

Method: A virtual address space is assigned in the memory map for the processor to communicate with the digital testbench. An AHB slave is present in the testbench to allow SV and C communication. A virtual address space, say, A000_0000 to A000_0400, is assigned. When writing to these virtual addresses, AHB transactions are sent to the testbench. An AHB monitor listens to the transactions and dispatches them to the components in the testbench, based on a channel encoded in the AHB transaction.

The slave has the ability to toggle the response, and read data lines by forcing HRESP and HRDATA at the appropriate moment in an AHB transaction. This way, the processor can see the testbench as a normal memory mapped slave and bidirectional communication can be established.

The AHB slave contains an associative array of TLM *put* ports and an associative array of TLM *get* ports. Each pair of port establishes bidirectional communication with an agent connected to the AHB slave. Each block agent can register a pair of TLM ports by passing in the wanted index of the associative array (which we will refer to as a channel). The registration function creates two new ports and puts them into the array at the requested index.

```
1  function void register(int id);
2  if(this.reqs.exists(id))
3  `uvm_warning("INIT", $sformatf("id %d already registered", id))
4  else
5  begin
6  uvm_blocking_put_port#(int) req = new($sformatf("req%0d", id), this);
7  uvm_blocking_get_port#(int) rsp = new($sformatf("rsp%0d", id), this);
8  this.reqs[id] = req;
9  this.rsps[id] = rsp;
10 end
11 endfunction
```

Fig. 6 shows two of the components that are attached to the slave for communication to and from the DUT. These are the message monitor and the processor synchronization agent. To register a channel, a component calls this

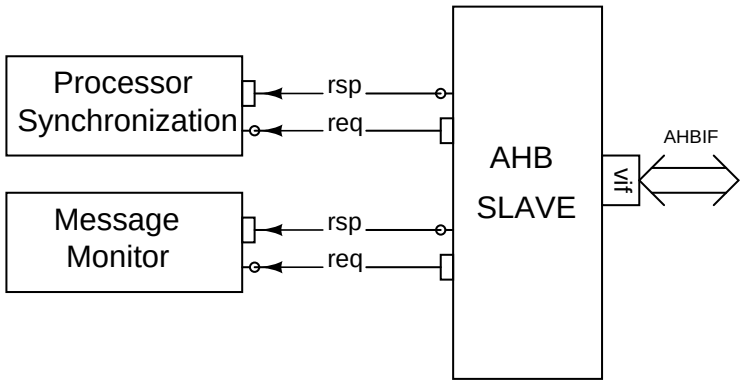

Fig. 6: Connectivity diagram of components involved in TB-C communication

function in the ENV and passes in the desired channel number. For example, the message monitor registers its TLM ports on channel 1 :

```
void'(ahb_slv.register(MSGMONITOR_CHANNEL));
```

A new TLM connection is thus created and can be used in the connect phase of the env to connect the message monitor with the AHB slave.

```
ahb_slv.reqs[1].connect(msg_mon.put_imp);
ahb_slv.rsps[1].connect(msg_mon.get_imp);
```

Each component that wants to connect to a channel must instantiate a *put* import and a *get* import (put_imp and get_imp in the code above). Each component defines its own put and get function. The *put* function defines what happens when the component is written. The *get* function defines what happens when the component is read.

Passing configurations from TB to C: The simulation environment allows for the transmission configuration to be requested by the code running on the processor, thus allowing for MMR programming in accordance to randomized values. A configuration (trxc_cfg) structure exists in C to match the system verilog transmission class.

```
typedef struct trx_cfg
{
uint32_t packets;
uint32_t dbm;
uint32_t protocol;
.
.
.
uint32_t address;
uint32_t pld;
 } trx_cfg_t;
```

A configuration structure can be declared in the C code and populated by calling the get_rx_config() and get_tx_config() routines. The routines communicate with Rx and Tx UVM sequencers respectively. Single configuration items can be obtained by calling get_tx_word(address) and get_rx_word(address). The addresses for the structure elements are defined as follows:

```
static const uint32_t PACKETS= 0xA0000008;
static const uint32_t DBM=0xA0000044;
static const uint32_t PROTOCOL = 0xA000004C;
.
```

```
  .
  .
  static const uint32_t ADDRESS = 0xA0000050;
  static const uint32_t PLD = 0xA0000054;
```

Handshaking: A synchronization mechanism exists for the DUT to pause/resume testbench activity and for the testbench to do the same to the DUT. Functions are provided in the C code library to send a sync or wait for a sync during execution by the processor.

```
#define sync2tb(channel) TB->CHANNEL = 2; TB->DATA = channel;
#define wait4tb(channel) TB->CHANNEL = 2; while((TB->DATA != channel));
```

Both these functions address the testbench on channel 2 (where the processor synchronization component is registered) and pass along a channel to either send or read a sync. When a wait4tb function is executed, the C code will loop reading the AHB bus until the returned data is the channel number (signifying that a synchronization has been sent back from the testbench).

In the testbench, there are two classes used by the synchronization mechanism. The sync_channel class contains only a single event. It is not extended from a uvm_object. The purpose of that class is to wrap an event in such a way that it can be used in an associative array.

```
class sync_channel;
event sync_from_dut;
endclass
```

C. Dealing with sluggish simulation

We all are aware of the advantages associated with the use of RNM models : fast turn-around time and exhaustive verification of analog-digital boundary [4]. Traditional mixed signal simulation approaches such as AMS models and Fast Spice simulators are specifically more cumbersome for RF SoCs given the associated high speeds. Also, a fast turn-around platform provides a base for firmware development early in the project cycle, much before a stable FPGA image is available. A further advantage is the ability to run a substantial number of simulations. This in turn allows for the introduction of a reasonable amount of system level randomization which gives the ability to test the radio and full DUT over many different packet parameters and radio setups. Attributes such as channel number, packet length, packet type, frequency deviation, gain error, offset error, input power etc. can be thrown into a the mix of a random constrained test case and generate appreciable coverage in system level simulation. [2]

The complexity and precision requirements of RF circuits such as Phase Locked Loops (PLLs) have traditionally kept real number modeling efforts at bay. The purpose of real number modeling of RF circuits is to create a fast and accurate-enough full chip simulation environment. RNM modeling of RF circuits has been discussed in detail in [2].

D. Pseudo-Firmware Environment

In general, the firmware (FW) development occurs in the following way:

- Design team specifies the required sequencing and processing to FW team (Fig. 7)
- FW team delivers FW functions & sequences
- A strategy for design team to sign-off on implemented FW.

It must be ensured that FW function hierarchy must be directly mappable to the requirements presented by the design team and should not deviate from the agreed state and function flow. For each sub-function sequence, the design team will provide sequencing information in some form (timing diagrams, state diagram (Refer Fig. 8)) and a pseudo-code representing required FW processing.

The FW development happens in a series of trial and error attempts. Therefore, a co-simulation based DV platform is an infeasible solution due to the excessive run time. An alternative option is an FPGA. However, a stable FPGA image is available only during the last phase of the project. This inherently delays the FW bring-up thereby affecting the time-to-market.

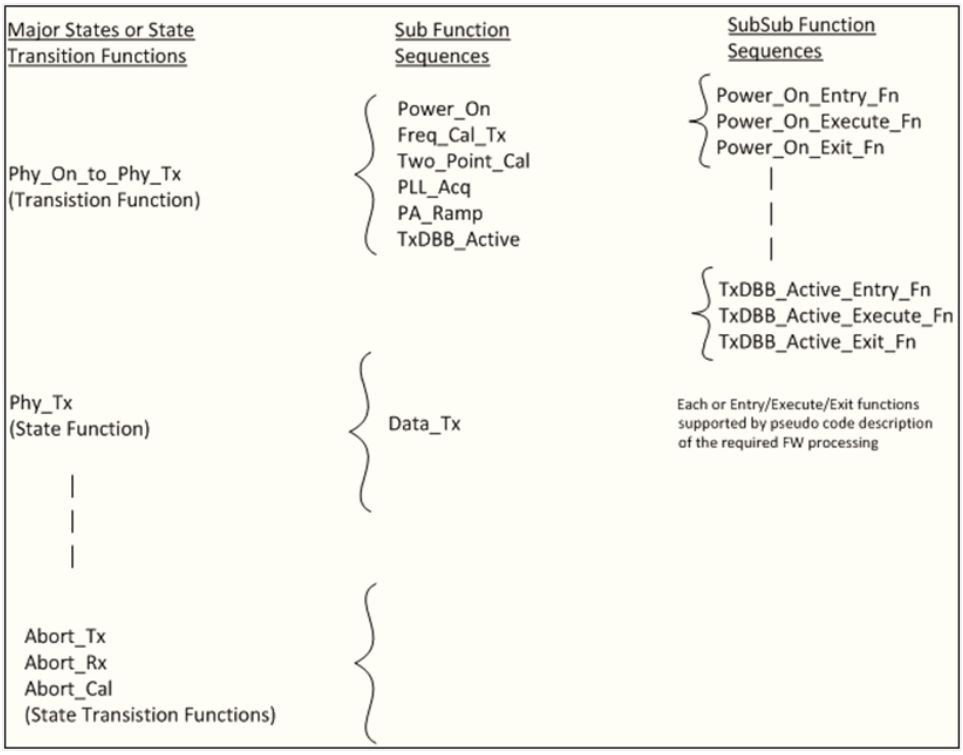

Fig. 7: An example of major states transition function, sub-function sequences and sub-sub function sequences list provided by design team to FW team.

The faster RNM based verification environment enabled FW development to begin much early in the project cycle. We maintained a close synchronization between the FW engineers and the DV engineers very early. This early engagement led us to have FW-alike code for the system level tests, ensuring a very realistic verification of the device.

The actual FW code was written and tested with the DV environment and on an FPGA. The software team and the DV team used same function names, enabling seamless FW based testing in the DV ecosystem.

E. Rx Testcase: An example

For the Rx DBB to function, we need to write a number of configuration registers which control various blocks in Rx PHY. These configurations are based on usecases and protocols. Appendix I shows an example code of a random protocol, random packet parameters, multi-transmission Rx system test. In each transmission, the packet parameters, protocol, blocker presence and its parameters are randomized while the static analog non-idealities such as DC offsets are constant as this is a one time calibration activity. Below is a pseudo-code of a general Rx testcase.

- Setup RF lineup, randomize analog process randomizations.
- Randomize the packet, blocker parameters
- Based on usecase determine settings for DBB in LUT.
- Enable Rx PHY.
- Release PMU reset
- Wait for start-of-frame detect interrupt (This indicates that current packet is valid)
- Wait for DMA DONE (Indicating end of packet).
- Disable Rx PHY
- Activate power management unit reset.

IV. Coverage

Interrupt Coverage: In our project, we have introduced a novel technique to map the coverage for all the interrupts by defining unique coverage channels for each interrupt. The coverage was collected for interrupts

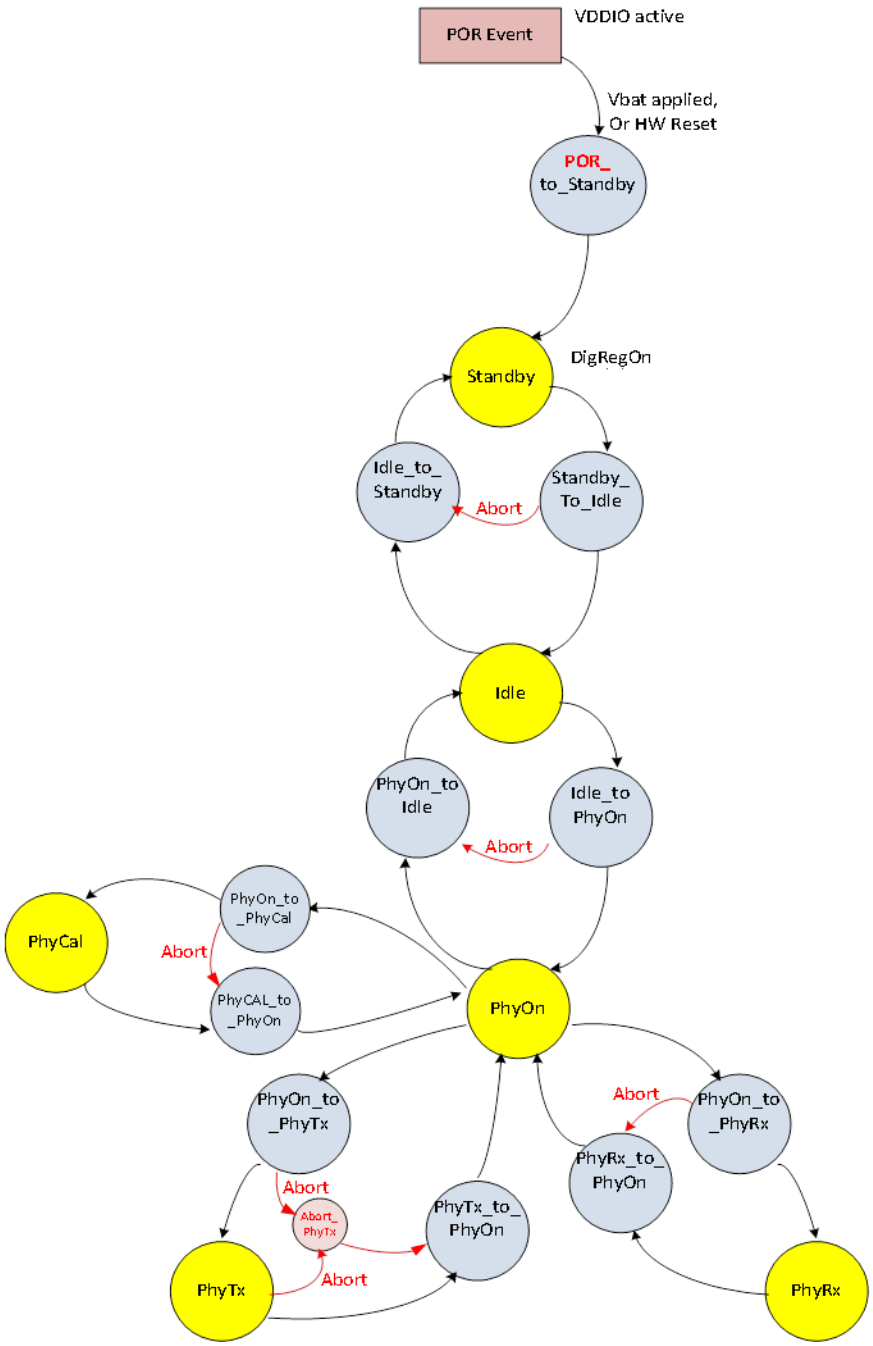

Fig. 8: Major state transition functions in a typical transceiver. PHY, Tx, Rx have their usual meaning.

triggered in a test by adding a sync2tb call (as described in Section II B) in the interrupt handler in the C code. The channel (argument to the sync2tb function) can be chosen between all possible interrupt sources, including sub-interrupts grouped under a single interrupt. A sample code has been given below to illustrate this.

```
1  void DBB_ERR_SOF_TO_IRQHandler(void) {
2  sync2tb(IRQ_DBB_ERR_SOF_TO);}
```

Stimuli Coverage: Usually at system level verification of a transceiver, we rely mostly on the code coverage to sign-off verification. This is due to the inability to decipher various transmission parameters just by observing the data stream in the monitor or scoreboard. In our project, we made sure that stimuli coverage was also captured. This was done by sampling the functional covergroups in the system level scoreboard and as well as in the test

code.

Average Grade	Covered Grade	Goal	Weight	Uncovered Bins	Excluded Bins	Total Bins	Item	Name
100.00%	100.00% (6/6)	100%	1	0	0	6	CoverPoint	rx_cg.rx_pkt_q_size
100.00%	100.00% (3/3)	100%	1	0	0	3	CoverPoint	rx_cg.rx_protocol
100.00%	100.00% (9/9)	100%	1	0	0	9	CoverPoint	rx_cg.rx_preamble_size
100.00%	100.00% (3/3)	100%	1	0	0	3	CoverPoint	rx_cg.rx_postamble_size
100.00%	100.00% (3/3)	100%	1	0	0	3	CoverPoint	rx_cg.rx_midamble_size
100.00%	100.00% (7/7)	100%	1	0	0	7	CoverPoint	rx_cg.rx_channel
100.00%	100.00% (2/2)	100%	1	0	0	2	CoverPoint	rx_cg.rx_header_size
100.00%	100.00% (2/2)	100%	1	0	0	2	CoverPoint	rx_cg.rx_pld_crc_size
100.00%	100.00% (2/2)	100%	1	0	0	2	CoverPoint	rx_cg.rx_whiten
100.00%	100.00% (97/97)	100%	1	0	0	97	CoverPoint	rx_cg.rx_dbm
100.00%	100.00% (10/10)	100%	1	0	0	10	CoverPoint	rx_cg.rx_f_err
100.00%	100.00% (23/23)	100%	1	0	0	23	Cross	rx_cg.__rx_protocol_X_rx_preamble_size
100.00%	100.00% (18/18)	100%	1	0	0	18	Cross	rx_cg.__rx_protocol_X_rx_channel
100.00%	100.00% (6/6)	100%	1	0	0	6	Cross	rx_cg.__rx_protocol_X_rx_whiten
100.00%	100.00% (70/70)	100%	1	0	0	70	Cross	rx_cg.__rx_f_err_X_rx_channel
100.00%	100.00% (6/6)	100%	1	0	0	6	Cross	rx_cg.__rx_protocol_X_rx_midamble_size
100.00%	100.00% (6/6)	100%	1	0	0	6	Cross	rx_cg.__rx_protocol_X_rx_postamble_size
100.00%	100.00% (3/3)	100%	1	0	0	3	Cross	rx_cg.__rx_protocol_X_rx_header_size
100.00%	100.00% (4/4)	100%	1	0	0	4	Cross	rx_cg.__rx_protocol_X_rx_pld_crc_size
100.00%	100.00% (237/237)	100%	1	0	0	237	Cross	rx_cg.__rx_protocol_X_rx_dbm
100.00%	100.00% (24/24)	100%	1	0	0	24	Cross	rx_cg.__rx_protocol_X_rx_f_err
100.00%	100.00% (13/13)	100%	1	0	0	13	Cross	rx_cg.__rx_protocol_X_rx_pkt_q_size

Fig. 9: A snippet of functional coverage report for the Rx coverpoints

IV. Conclusion and Future Work

The application of a UVM based RF environment, intelligent checkers etc. aided in finding a substantial number of design issues early in the course of the project, immensely cutting down the risk of a large tape-out slip. The instant testbenches with numerous in-built stimuli and checks helped even UVM beginners to close the block level verifications at a very accelerated pace. RNM models not only present an opportunity to build a fast and efficient simulation platform for test and firmware development but also are perfectly suited for leveraging UVM based random constrained verification techniques. Fig. 9 shows a snippet of functional coverage report for the Rx coverpoints after a full system regression. This clearly demonstrates the exhaustive randomization of various transmission parameters we were able to achieve. The coherent DV environment also bridged the gap between the verification and FW development, saving many weeks of efforts for firmware bring-up.

The analog models and RF agent are easily extensible to further projects. New modulation schemes can easily be added to the RF agent, as can new packet formats. Object oriented techniques such as inheritance makes this a simple exercise and this is indeed the model on follow up projects with different 2.4 GHz standards. The analog models require more tweaking to meet different analog specification, but they do represent a standard 2.4 GHz narrowband transceiver topology which is very likely to be used in other radio SoCs.

Silicon evaluation is in progress and no bug has been reported so far on the digital side. Within one week from the silicon arrival, the measurement team was able to bring-up the functional transceiver with Phy_Tx and Phy_Rx state, which is transmitting and receiving data, both working seamlessly with embedded firmware. Samples were tested and shipped to customer 1.5 weeks ahead of the planned date. These advances in verification methodologies have eased many of the challenges traditionally associated with the integrated RF transceiver SoCs and enabled the team to deliver first-silicon within one year of project launch.

However, there are some aspects of these strategies that can be improved to make it more organized and faster, as with any verification approach. An eyesore (literally) in our method was the implementation of the constraints on interferer (blocker) power which have to be specified in each test. A sample code for single blocker generation is given in code snippet 3. The specification of blocker performance for co-channel and adjacent channels varies with protocol. We plan to make it centralized rather than repeating in each test. Furthermore, we will enhance the environment for simulating antenna diversity. Also, the reporting of BER (bit error rate) needs to be added.

Code Snippet 3: A snippet of single blocker generation

```
// create a blocker
blocker = rf_seq_item::type_id::create("blocker");

channel_diff = 0;

void'(blocker.randomize() with {
 .
 .
 channel inside {[min_ch:max_ch]};
});

channel_diff = $abs(blocker.channel - transmission.channel);

 if(wanted_protocol==protocol_1)
  begin
  if(channel_diff == 0)
   blocker.dbm = prot_1_spec_0 ;
  else if(channel_diff == 1)
  blocker.dbm = prot_1_spec_1;
   else if(channel_diff == 2)
   .
   .
 end
else if(wanted_protocol==protocol_2)
 begin
  if(channel_diff == 0)
  blocker.dbm = prot_2_spec_0 ;
  else if(channel_diff == 1)
  blocker.dbm = prot_2_spec_1;
  else if(channel_diff == 2)
  .
  .
 end
.
.
.
```

REFERENCES

[1] Donnacha O'Riordan, and Cormac O'Sullivan, "Mixed Signal Design & Verification Methodology for Complex SoCs", http://www.design-reuse.com/articles/33499/mixed-signal-design-verification-methodology-for-complex-socs.html

[2] Jakub Dudek, Joshua Nekl, and Keith O'Donoghue, "Real Number Modeling of RF Circuits", DVCon US, 2017 (Accepted)

[3] Muhammad, K., T. Murphy, and R. B. Staszewski. "Verification of RF SoCs: RF, analog, baseband and software.", IEEE Radio Frequency Integrated Circuits (RFIC) Symposium, 2006. IEEE.

[4] Dushyant Juneja, Siddharth Prabhu, and Syam Veluri, "Practical Considerations for Real Valued Modeling of High Performance Analog Systems", DVCon US, 2016.

APPENDIX I

```
#include <....>

rf_cfg_t cfg;
trx_cfg_t tcfg;
rx_cfg_t rx_cfg;

int main(void) {
cfg = get_config(); // Getting the randomized static configurations
loop(number_of_transmissions) {

// Establishing the sync between TB and C code to enable packet
   randomizations
wait4tb(RX_SYNC);

// Obtaining the random packet parameters
tcfg = get_rx_config();
info("printing randomized packet parameters");
info("Carrier is %d", tcfg.fc);
...
// Updating the Rx PHY DBB register structure based on randomized packet.
rx_cfg.rx_phy_dbb = rx_phy_dbb_init(&tcfg);

// STATE Transitions
////////////////////////////////////////////////////////////////////////
Ready_To_Idle();
// Configuring the Rx PHY DBB registers
Idle_To_Phy_On(&rx_cfg);
Phy_On_To_Phy_Rx(...some parameters....);

sync2tb(RX_SYNC);

//DMA RX channel setup
...
// Waiting for DMA done status
...
} // Looping over number of transmission

//Processor only waits for tb to return
wait4tb(RX_SYNC);
return EXIT_SUCCESS;
}

void DBB_RX_IRQHandler(void) {
...
}
void DBB_ERR_IRQHandler(void) {
...
fatal("Unexpected SOF Timeout interrupt detected");
...
}
```