2014 DESIGN AND VERIFICATION DVCON CONFERENCE AND EXHIBITION EUROPE

Accelerated SOC verification Using UVM Methodology for a Mix-signal Low Power Design

Giuseppe Scata
Texas Instruments - MCU
Freising - Germany
g-scata@ti.com

Ashwini Padoor
Texas Instruments - MCU
Bangalore – India
ashwini.padoor@ti.com

Vladimir Milosevic
ELSYS Eastern Europe
Belgrade - Serbia
v-milosevic@ti.com

***Abstract*—This paper proposes a solution to verify a microcontroller based System on Chip (SoC) using a verification environment which has been architected following System Verilog Universal Verification Methodology (UVM) guidelines and still allows the coding of direct tests in Verilog style. It highlights the possibilities of randomizing the software parameters and its execution while deploying a unified simulation for along RTL, gate level and Analog Mixed Signal (AMS) simulations.**

***Keywords*— Verification, System Verilog, UVM, microcontroller, Random Constrained Verification, Direct Testing, AMS verification, hardware-software co-verification, software randomization.**

I. Introduction

Digital verification engineering emerged in the last 20 years as an indispensable part of chip design. As complexities grow and productivity pressures rise, the expansion of verification engineering into the mix-signal low power design space in the short term is inevitable. Realizing silicon in the face of these challenges requires new approaches and a very flexible workforce capable of adapting and changing on a regular basis. The verification environment for any new design had to follow these needs: fast setup, easy reusability options and strong verification possibilities.

This paper outlines a deterministic microcontroller based SoC design verification approach to manage a mixed-signal low power design complexity using assertions and Metric driven verification methodologies in a UVM (Universal Verification Methodology) based environment. One of the advantages is leveraging the IP verification reuse and keeping open the possibility for plain Verilog directed testcases to unify the flow between the need for a modern verification environment and in parallel to allow the designers to create basic sanity test cases. The constrained random capabilities of the UVM were extended even for the software part of the SoC in order to allow randomization of data and code execution.

The deployed simulation environment was used not only for the basic SoC level interconnect checks, but also for the IP verification as well as for the application software validation. UVM randomization capabilities were applied for IP functionality checks at SOC level and hence it was even possible to eliminate the IP verification gaps. It was also used for SoC level use-case randomization, which allowed more SoC level scenario generation including the corner cases, and has added to the quality of verification.

An important aspect of this methodology is to focus not only on the digital part of the SoC but also into the entire mixed signal design definition. The analog top centric design incorporates the analog netlist which contains behavioral models and for each analog block these models could be replaced through a configuration file to a full transistor level SPICE description.

II. Verification Environment Architecture

The shrinking development cycles defines the need of a comprehensive, efficient and flexible test bench platform and to achieve this intercepting the evolving verification methodologies becomes primary importance.

We focused to achieve below goals during the verification process of the low-power mixed-signal microcontroller design.

- Use pre-verified Bus Functional models instead of integrating the native Bus Functional Models (BFMs) in the testbench;

- unified simulation flow for RTL, gatelevel and AMS representation of the Device Under Verification (DUV);
- Support for constrained random testcases as well as plain Verilog based directed testcases;
- the SoC verification close to an IP verification approach;
- Most importantly the need to follow an Industry standard verification methodology;

The testbench framework defined in such a way that all above goals could be achieved using the following components:

- Verilog segment: plain Verilog part;
- System Verilog segment: UVM compliant components;
- Hardware software synchronization logic;

Any test-case can be either a UVM sequence or a pure Verilog directed test, both combined with a software part. The testcase pass-fail criterion is determined by evaluating System Verilog errors (from monitors and assertions), Verilog errors and C-code software errors along with the testbench timer status.

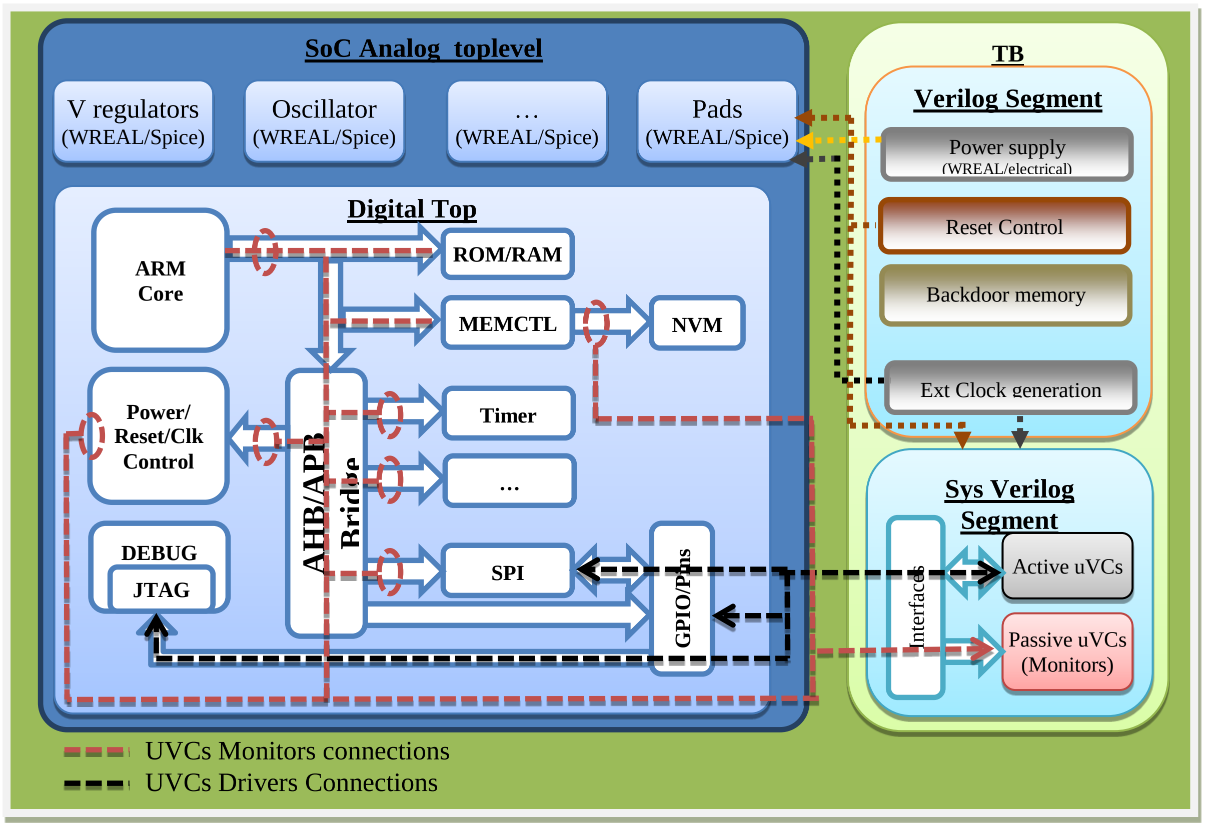


Figure 1

The testbench architecture used for the microcontroller based SOC has been outlined in Figure 1.

The verification environment is architected in such a way that the user can utilize the flexibility of using real number (WREAL) models, transistor level spice representation or a mix of the two. Each test-case could be configured to use any of the previous mentioned abstraction levels.

This provides the flexibility to use real number behavioral models for the faster debug and to reuse the environment including the test cases for the AMS simulation just by replacing all or some of the models with actual SPICE netlist (See Figure 2).

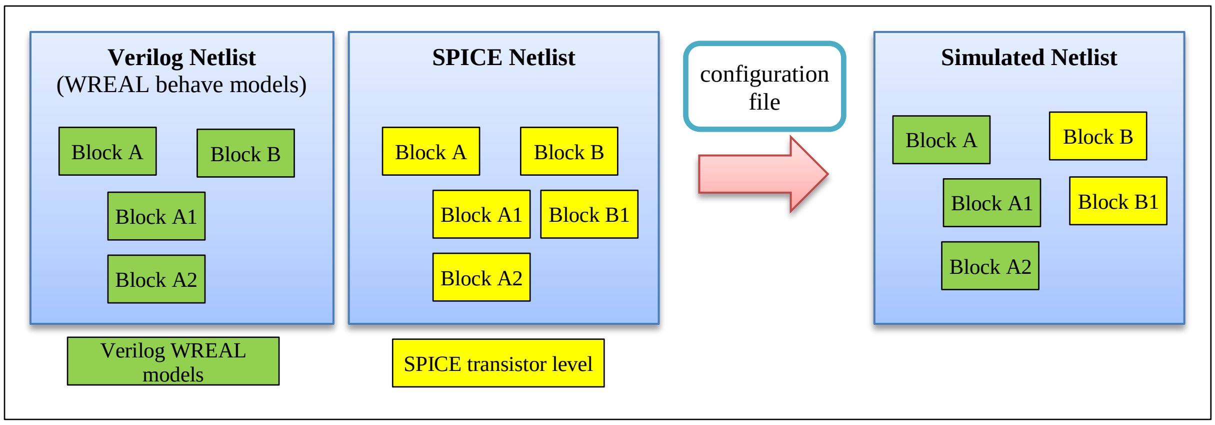


Figure 2

In order to consider the power domain information the digital part of the DUV is paired with a CPF (common power format) file that is created for each of the step of the design (RTL, gate-level pre-layout and gate level post-layout). It is not in the scope of this document to go deeper in the detail of this process.

A. *Verilog Segment*

The legacy part of the testbench is implemented in plain Verilog and this segment comprises four main blocks which are associated with analog power signal generation, memory backdoor access function definitions, reset generation and clock generation.

The analog power signals are generated according to the test-case configuration and are either real numbers or Verilog-AMS electrical quantities. This approach enables the possibility to simulate power ramps, power drops, glitches, etc.

The generation of these signals could be easily migrated into a UVM approach.

The memory backdoor access functions are used for:

- Memory initialization with the predefined data/instructions;
- UVM component and software synchronization and data exchange;
- User memory backdoor accesses;

The Verilog segment of the test bench also contains a clock generation and reset control blocks. The DUT external clock and uVC clock generation is handled by this block. Reset generated by the testbench is used as the main power-on reset to the DUT and also for UVM components.

B. *System Verilog Segment*

The System Verilog segment is associated with interface definitions, active and passive uVC components and assertions. The subsequent section describes the basic tenets of UVM testbench components.

1. *UVM sequences and direct testing*

The uVC sequences are used for defining the stimulus for various peripherals which includes IO ports, serial communication peripherals, debug subsystem etc. All the sequences are implemented in a way to enable transparent reuse between RTL, GLS (Gate Level Simulation) and AMS simulations.

The synchronization between various sequences is achieved by a virtual sequencer.

An important feature is to give the possibility to the non UVM trained engineers to code direct tests in plain Verilog.

To achieve this a default dummy uVC Sequence has been defined, it controls the simulation time and ends when the software execution is completed. The sequence raises/drops uvm_test_done objection to control start/end of the simulation according to UVM guidelines. It also implements a timeout mechanism.

DIRECTED TEST WITH DUMMY UVM

```
// DIRECTED TEST
initial begin
  // DIRECTED TEST BODY
end

// DUMMY UVC INSTANTIATION
class `tc_name extends uvm_test;
 soc_tb ve;
 `uvm_component_utils(`tc_name)
 function new(input string name,
              input uvm_component parent=null);
  super.new(name,parent);
 endfunction

 virtual function void build_phase(uvm_phase phase);
 super.build_phase(phase);
 uvm_config_db#(uvm_object_wrapper)::set(this, "ve.virtual_sequencer.run_phase",
          "default_sequence", sw_seq::type_id::get());

    set_type_override("vr_ahb_agent_monitor", "ahb_monitor");
    super.build_phase(phase);
    ve = soc_tb::type_id::create("ve", this);
  endfunction : build_phase

  function void end_of_elaboration_phase(uvm_phase phase);
    ve.top_env.set_report_verbosity_level (UVM_LOW);
  endfunction : end_of_elaboration_phase
 endclass : `tc_name
```

2. *Checkers*

Checkers are implemented at various design abstraction levels to ensure the data flow correctness, protocol compliance for various bus interfaces (e.g. AHB, APB, JTAG), reset assertion/de-assertion correctness and to ensure the precision of Analog & digital control timings. The checker implementation highlights are described below.

a. *standard bus protocol checkers*

The uVC based checkers are implemented to ensure the protocol compliance for various standard interfaces like AHB, APB, Serial communication interfaces, debug interfaces etc.

These checkers are always enabled and the protocol violations are flagged out by providing the appropriate errors details. All the bus protocol checks are enabled for all the interfaces irrespective of the number of instances connected.

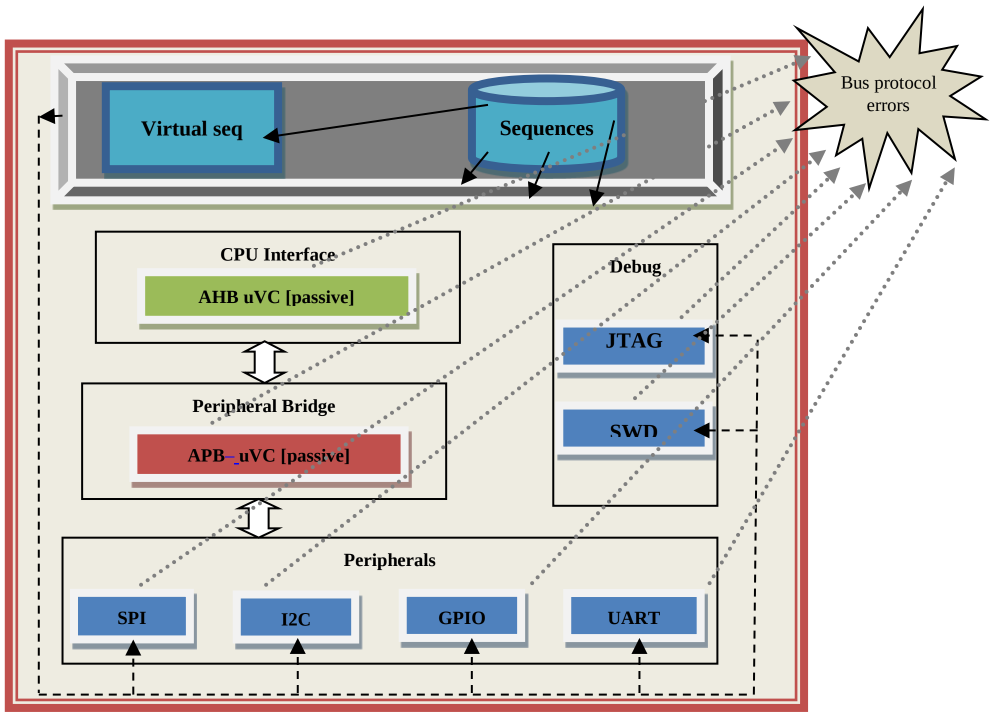


Figure 3

b. Assertions between digital & analog blocks

Always running assertions based checkers are implemented to ensure the clock and power management block functionality correctness at system level.

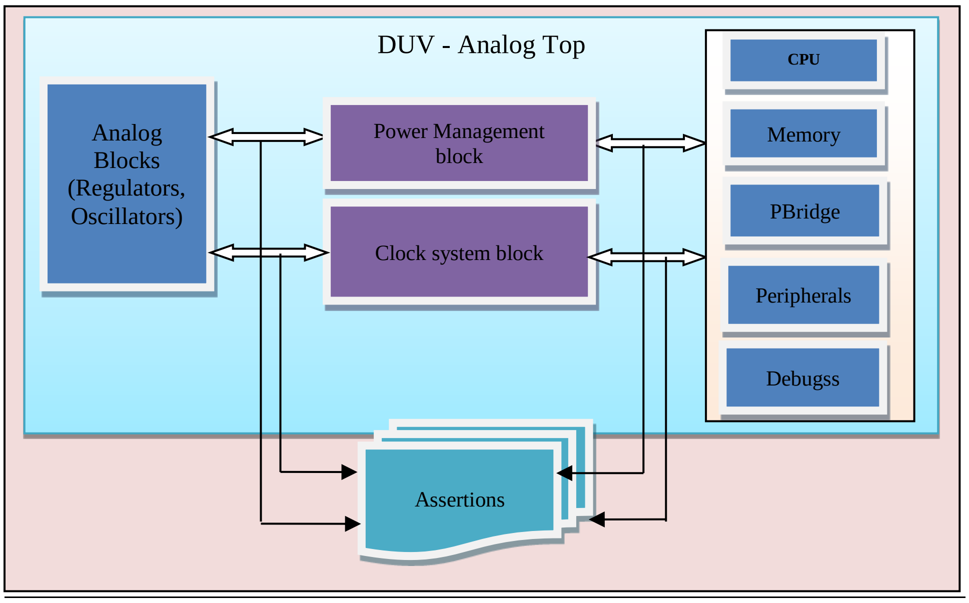


Figure 4

In the example below the checker ensures that when the LDO voltage level is dropped beyond the minimal threshold value, the assertion ensures that CORE RESET is getting asserted.

```
task ams_regulator_chk();
 forever begin
 @(posedge vif.tbclk iff (vif.resetall === 0 &&
          cfg.checkers_en == 1               &&
          (vif.vdd_lvl < `VDD_LVL_MIN)))
   nRST_AMS_CHK: assert (vif.pmucore_poresetn_i == 1'b0) else
   begin
     `uvm_error(get_type_name(),
        $psprintf("CORE_RESET: %1b,not expected (low Regulator Voltage LvL) @ %0t!\n",
                       vif.pmucore_poresetn_i, $time))
   end
 end
endtask
```

The digital domain checks are targeted for ensuring the clock frequency correctness, system LPM Mode definition versus clock source usage, isolation assertion/de-assertion timings, reset assertion/de-assertion, analog and digital handshake timings, glitch checks on the important control signals including the reset etc.

```
/*=============================================
  The following checker ensures
    During Emulated MOD1, CONFIGISO gets asserted
    During Emulated MOD1, ISO doesn't get asserted
    ============================================*/
task iso_dbg_state();
  forever begin
   @(posedge vif.fclk iff (cfg.checkers_en == 1))
   if (vif.cdbgpwrupack==1 && vif.sleeping==1 && pwr_dsm==4'hF)
    begin
    // Chk during MOD1 iso_.3P3V is not asserted
    ISO_AO_O_3P3V: assert(vif.iso_ao_o_3P3V == 0)
     `uvm_info(get_type_name(),$psprintf("%s", $psprintf("ISO CHECKING!")),UVM_FULL)
   else
    `uvm_error(get_type_name(),$psprintf("\n CDBGPWR: %1b,SLEEPING: %1b,
               PWR_DSM: %1b, ISO_AO3P3V signal asserted, and it should not! @ %0t!\n",
               vif.cdbgpwrupack, vif.sleeping, pwr_dsm, $time))
   // Chk during MOD1 CFG_ISO signal is asserted
   CFG_ISO: assert(vif.config_iso==1 || vif.sysresetn==1)
         `uvm_info(get_type_name(),
             $psprintf("%s", $psprintf("\n CFG CHK! ")),UVM_FULL)
    else
     `uvm_error(get_type_name(),
        $psprintf("\n CONFIG_ISO: %1b, SLEEP: %1b, PWR_DSM: %1b, CFG_ISO isn't
                    asserted! @ %0t!\n", vif.config_iso, vif.sleeping, pwr_dsm, $time))
      end
    end
  endtask
```

All the assertions related to the control signals have implemented as two way checks.

- The control signal is getting asserted/de-asserted when it is expected;
- No spurious assertion/de-assertion is observed;

Also the assertions are associated inbuilt switches which can be used to turn ON/OFF assertion checks. This was helpful to bring up the initial gate level simulations (GLS) runs quickly and the assertions were enabled one-by-one which helped to speed-up the GLS regression handling

c. Data checkers

APB and AHB monitor are normally used for defining the protocol checks and coverage bins, but can also be used for:

- IP behavior prediction based on the register configuration;
- Implementing the scoreboard based checks for the SOC;

In Figure 5 an example of scoreboard implementation for a memory controller is shown.

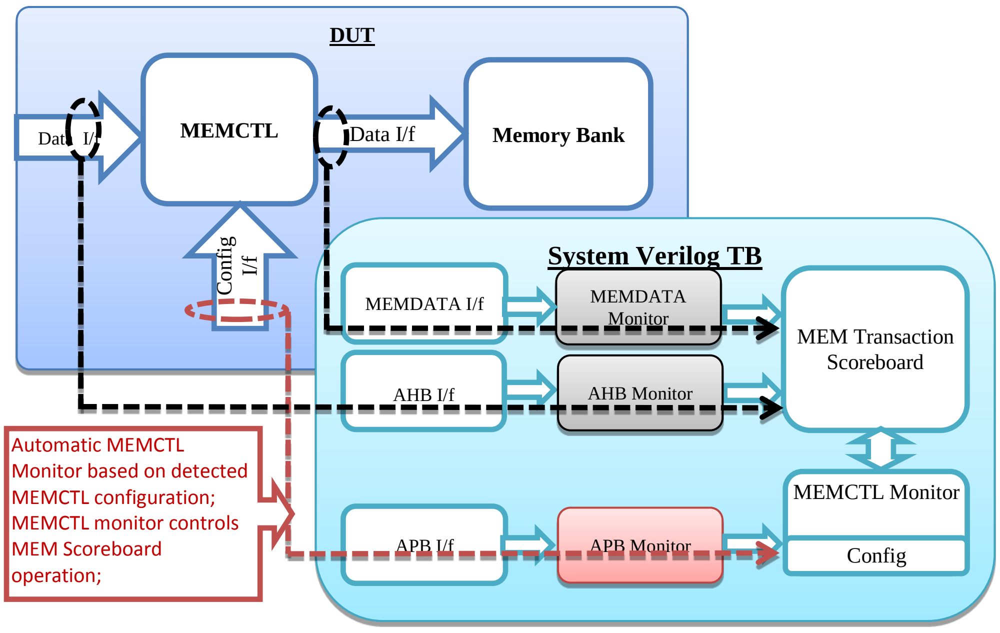


Figure 5

The scoreboard implementation depends on the register configuration by the CPU through the APB Interface, the data transfers initiated by the CPU through the AHB Interface as well as the data path between the memory controller and the memory banks. This information is captured into the scoreboard consistently by following components of the testbench:

- The “MEMCONTROL Monitor” snoops the APB transactions on the “MEMCTL” register address space and automatically gets configured based on the CPU initiated configuration.
- The “AHB Monitor” monitors the data transfers between CPU and memory controller and the “MEMDATA Monitor” monitors the data transfers between Memory controller and memory bank.
- Using the MEMCONTROL Monitor and above two bus snoopers the scoreboard is filled. The “MEM Transaction scoreboard” ensures the correctness of data transfers between CPU and memory banks.

This approach allowed fully automatic uVC/Monitor configuration and the test case development focus was only on the stimulus related details not on the checking part.

d. Built-in Self checking mechanisms

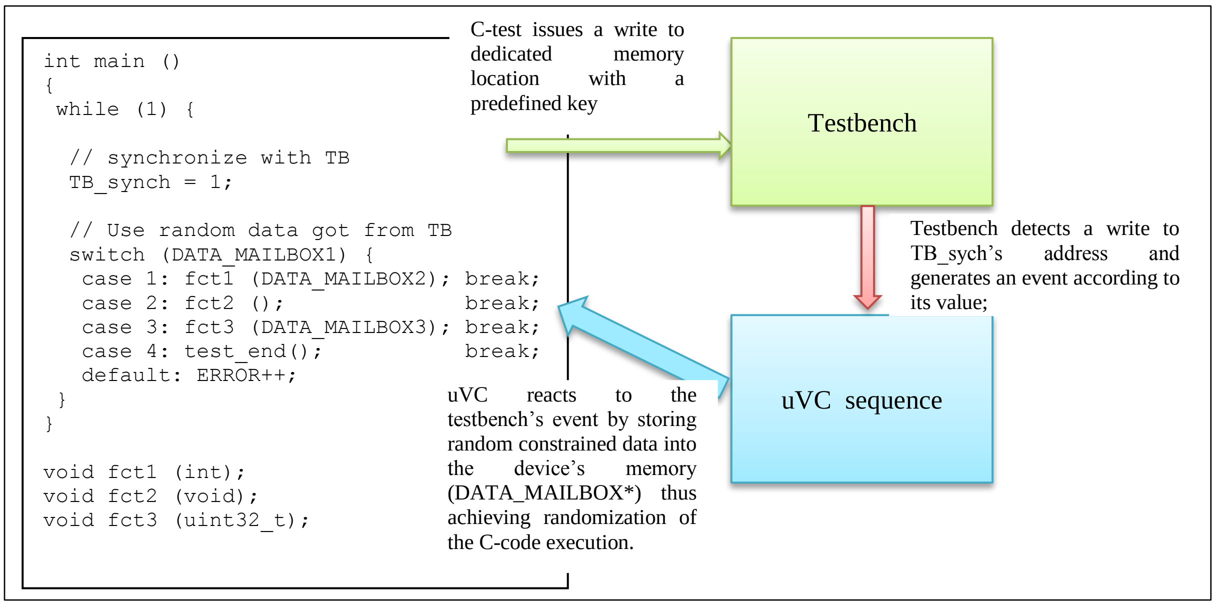
The software and direct sequences are implemented in such a way that a subset of checks can be proven via dedicated directed testcases. This set mainly includes Status register set/clear functionality, reset influence after various power modes, Isolation defaults for various registers, automatic hardware influences on various registers and so on.

C. Hardware software synchronization

In a microcontroller based product the software is the main driver of the internal busses so we can consider it as stimuli generator. Hence we can consider it as part of the testbench/testcase and we need a mechanism to synchronize it with the SystemVerilog testbench and we need the possibility to randomize it.

In order to synchronize the C-code with the testbench functions and UVM sequences, events are triggered in testbench by monitoring write accesses to the predetermined memory addresses locations. Thus the software part of the code can fire an event by writing predefined patterns into these memory locations. Similarly some memory locations can be used to enable data exchange between the testbench and the software and vice-versa.

The possibility to synchronize and to exchange information between the testbench/testcase and the software allows achieving the software randomization (for both execution and input data), the flow is depicted in the Figure 6.



```
int main ()
{
 while (1) {

   // synchronize with TB
   TB_synch = 1;

   // Use random data got from TB
   switch (DATA_MAILBOX1) {
    case 1: fct1 (DATA_MAILBOX2); break;
    case 2: fct2 ();              break;
    case 3: fct3 (DATA_MAILBOX3); break;
    case 4: test_end();           break;
    default: ERROR++;
  }
}

void fct1 (int);
void fct2 (void);
void fct3 (uint32_t);
```

C-test issues a write to dedicated memory location with a predefined key

Testbench detects a write to TB_sych's address and generates an event according to its value;

uVC reacts to the testbench's event by storing random constrained data into the device's memory (DATA_MAILBOX*) thus achieving randomization of the C-code execution.

Figure 6

III. Functional Coverage Models

The functional coverage is collected into two different ways: via coverage groups in the uVC monitors and via assertions.

In order to make sure the IP functionality is covered in detail, coverage groups are defined within the uVC monitors.

Having the possibility to code non UVM compliant direct tests necessitated an automatic tracking of the coverage for these scenarios. This was achieved by a native defined method and is known as "Verification Item Tags" (VITag). The verification engineer can define a VITag for each scenario defined in the verification plan and should be used for any items that cannot be covered automatically. VITags are implemented in the Testbench as System Verilog assertions and are mapped to the verification plan. The VITags are triggered manually either from the C code (by using a library function) or from the testbench (by using a testbench auxiliary task) and can yield to a PASS/FAIL status.

VITag PASS trigger from software `setVITAG(VITAG_BASIC_SW,RES_PASS);`
VITag FAIL trigger from the testbench ``tb.setVitag(`VITAG_BASIC_TB,`RES_FAIL);``

IV. RESULTS AND CONCLUSION

This paper presented practical experience of an uVM based verification approach followed for a mix-signal Low Power design. This flow has been implemented successfully and the verification was completed on-time. The UVM based verification infrastructure also helped to have a single verification environment for digital, AMS and software validation. This for sure helped us to maximize verification quality versus effort and especially to overcome IP verification gaps. The silicon is out from the fab and the silicon validation is showing positive results.

V:Acknowledgment

The authors would like to thank the entire TI Security design and management team without whose support this flow would not have been a success.

References

[1] Accellera, "UVM User Guide", www.uvmworld.org