2019 DESIGN AND VERIFICATION DVCON CONFERENCE AND EXHIBITION INDIA

# A pragmatic approach leveraging portable stimulus from subsystem to SoC level and SoC emulation

Karandeep Singh, (karandeep.singh@nxp.com) [1]

Aditya Chopra,(aditya.chopra@nxp.com) [1]

Joachim Geishauser, (joachim.geishauser@nxp.com) [2]

Nitin Verma (nitin.verma@nxp.com) [1]

**NXP Semiconductors, [1]Noida, India ,[2]Munich, Germany**

***Abstract*—The growing complexity of SoC's these days and early time to market poses great challenges for verification closure. This paper targets various levels of verification closure from module to SoC. With the motivation to adhere portable stimulus-based approach, emphasize developing C based stimulus from beginning and to reduce time to market, this paper demonstrates leveraging C tests and testbench infrastructure from subsystem to SoC simulation to SoC emulation to GLS closure. Considering the requirement of reusability, portability and interoperability the infrastructure is developed around TLM 2.0 Generic Payload for all communication interfaces. This paper demonstrates the advantages of fast simulation as well as SV powerful constraint randomization, UVM sequence layering and ease of use and all this encapsulated in C wrapper used via DPI function calls to reuse at SoC. Various modelling/infrastructural challenges like reset, interrupt etc. are addressed, demonstrating the huge advantage of light weight and high-speed architecture. Key of the testbench architecture ensures that all the infrastructure can be used the same way for native UVM verification (UVM sequence-based verification).**

***Keywords—Functional verification, TLM2.0, TLM2-GP, subsystem verification, SoC verification, early driver development, emulation, SV/UVM***

## I. INTRODUCTION

There has been significant work done which proposes subsystem level verification leveraging to SoC, posing enormous advantages like portable stimulus, early time to market, fast simulation, faster developments and debug cycles. Porting of testcases from Subsystem to SoC simulation to Emulation is always a challenge. This methodology helps in developing C based stimulus right from the beginning at subsystem with fast development cycles. This paper aims further to demonstrate a working solution for the same exploring TLM2.0- GP for all interface communications making Reusable-Interoperable-Protocol Independent test sequence also covering various problems faced during execution and the solutions/learnings. Emphasis is further laid on enabling the verification scalability to SoC emulation level as shown in Figure 1. below, such that, the testcases/infrastructure created for Sub-system level can be easily ported to SoC level and then leveraging it further at emulation expediting the number of packets/data.

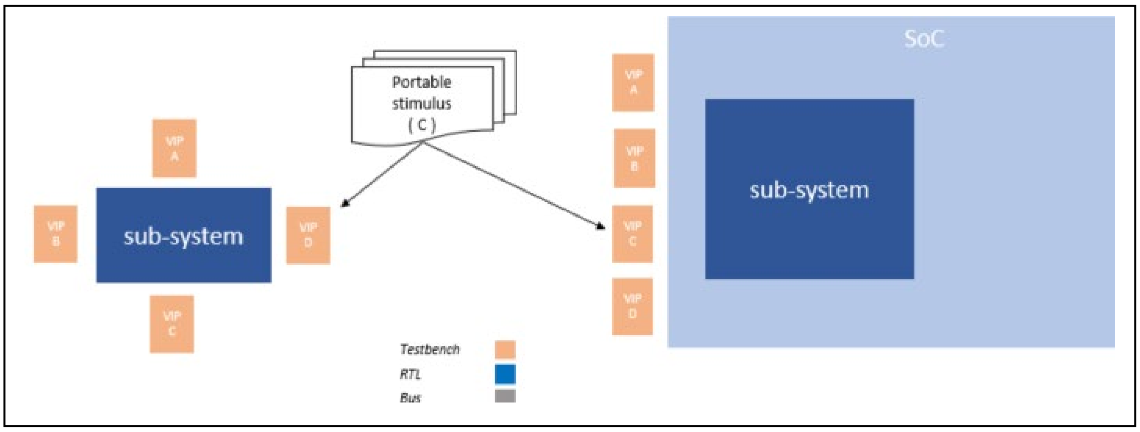


Figure 1 Execution at Subsystem Level

Various techniques like switching off undesired clocks, reducing activity, putting cores to sleep mode and back has been discussed well [2]. This paper further leverages creating the subsystem approach to achieve the same as well. Further, subsystem based env using UVM/SV based transactors which are not only faster (minimum 3x faster than core based on various experiments) but quite simpler to implement/update. There are various approaches based on creating the virtual platform which has a processor model encapsulated in as VAL layer [3] using SystemC model as core layer in Verilog wrapper. Considering the advantages of UVM based transactors, approach has been established with generic interrupt and reset layer which is simple to implement and faster as well. Developing the C test at much faster rate at Subsystem can be directly ported to SoC simulation as well as SoC emulation. Enhanced advantages of DPI's for C and SV dynamic communication allows users to simulate the changed C code without recompiling the bench.

This paper aims to demonstrate an easier and proven way to meet the goals for porting the stimulus from subsystems to SoC simulation to SoC emulation.

## II. INFRASTRUCTURE DETAILS

### *A. Subsystem Infrastructure*

The infrastructure used to model the verification testbench at subsystem level uses the SV-UVM based VIPs which are wrapped in C-BFM infrastructure using TLM2 GP ports and DPI's for exchanging cross-language function calls. Figure 2 shows the subsystem testbench architecture. While execution core is UVM driver at Subsystem level which have been replaced as it as with Real RTL core at SoC.

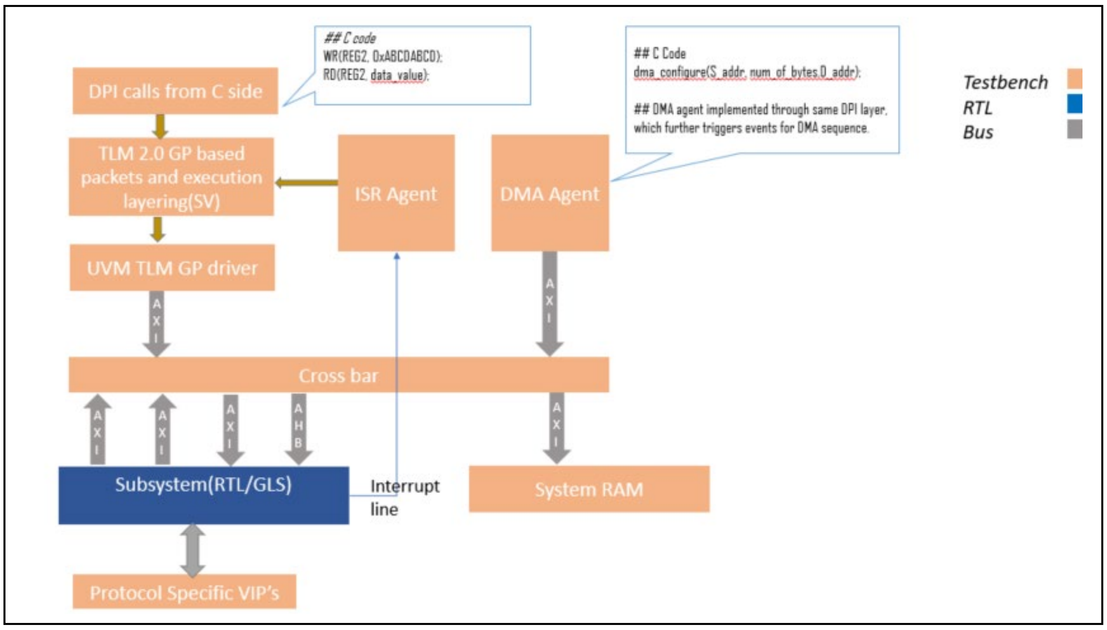


Figure 2 Subsystem Testbench architecture

Further Figure 3 shows the stimulus execution at subsystem while Figure 4 at SoC level.

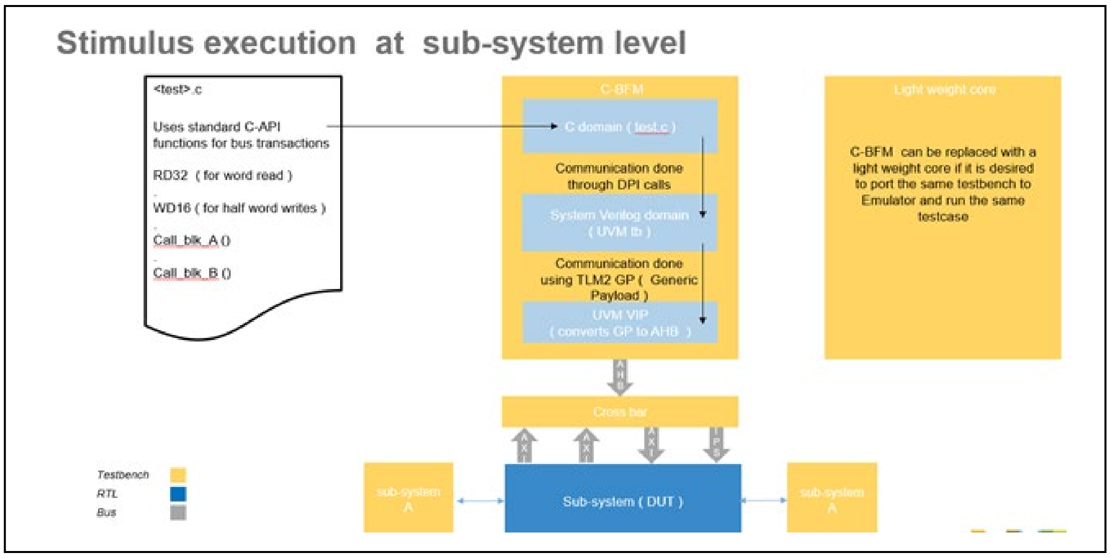

Stimulus execution at sub-system level
<test>.c
Uses standard C-API functions for bus transactions
RD32 ( for word read )
WD16 ( for half word writes )
Call_blk_A ()
Call_blk_B ()
C-BFM
C domain ( test.c )
Communication done through DPI calls
System Verilog domain ( UVM tb )
Communication done using TLM2 GP ( Generic Payload )
UVM VIP ( converts GP to AHB )
Light weight core
C-BFM can be replaced with a light weight core if it is desired to port the same testbench to Emulator and run the same testcase
Cross bar
AXI
APB
Testbench
RTL
Bus
sub-system A
Sub-system ( DUT )
sub-system A

Figure 3 Execution at Subsystem Level

The same test which has been developed at subsystem level will be directly reused at SoC. The set of overheads related to SoC like boot/clock configuration etc has been taken care with the help of context mapping, where dummy functions implemented at subsystem are mapped to SoC API's through context.

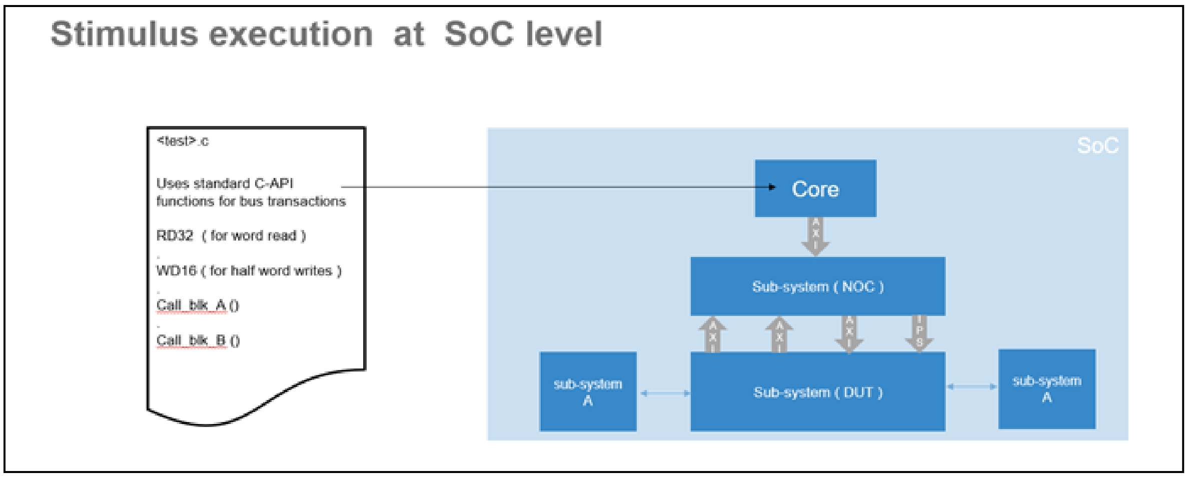


Figure 4 Execution at SoC Level

## B. *Hooks for Subsystem and SoC*

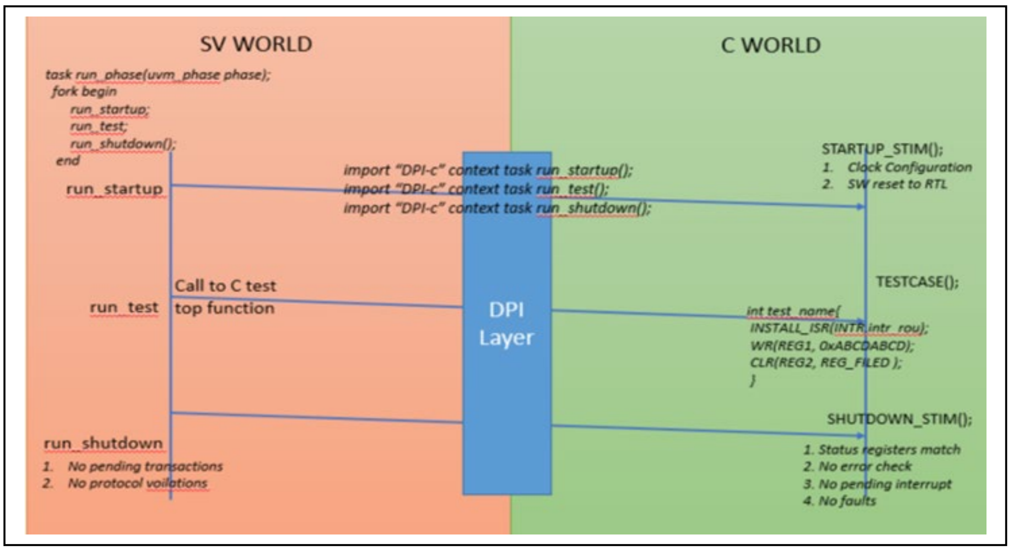
Various functions and call-backs are placed at C level for other peripheral communication interfaces such as DMA/Interrupts/etc. These will serve the purpose via UVM driver at Subsystem and RTL master at SoC.

## C. *Reusable-Interoperable-Protocol Independent test sequence*

- Portable C tests: Test developed at subsystems will be directly ported to SoC simulation and SoC emulation, with context-based pointers to SoC specific executions, which at SoC will be mapped to SoC specific functions, for e.g.: the case of boot, clock, reset, fault triggering and clearing.
- Interoperable Sequences: Sequences used for Protocol Driver(protocol) X using TLM2-GP extensions (for added-on non-generic transfers/sideband signals) are compliance with Protocol Driver Y without extensions.
- Protocol independent: Consider the case for register layer in UVM, if there is any change in Protocol Bus, only Driver/Adapter needs to be updated and the test/sequences would not change. Similar is the case using TLM2-GP.

### D. *DPI compliance*

- Synchronicity
- TLM 2.0 GP based non-blocking port communication.
- The C code updates need not to compile as it is dynamically linked via DPI calls.
- Infrastructure is direct compatible with SystemC models due to use TLM based interfaces, hence can be used to develop/test low level drivers.
- Figure 5 demonstrates the communication flow between C and SV through DPI, various processes created to implement synchronicity.



```
task run_phase(uvm_phase phase);
 fork begin
   run_startup;
   run_test;
   run_shutdown();
 end
```

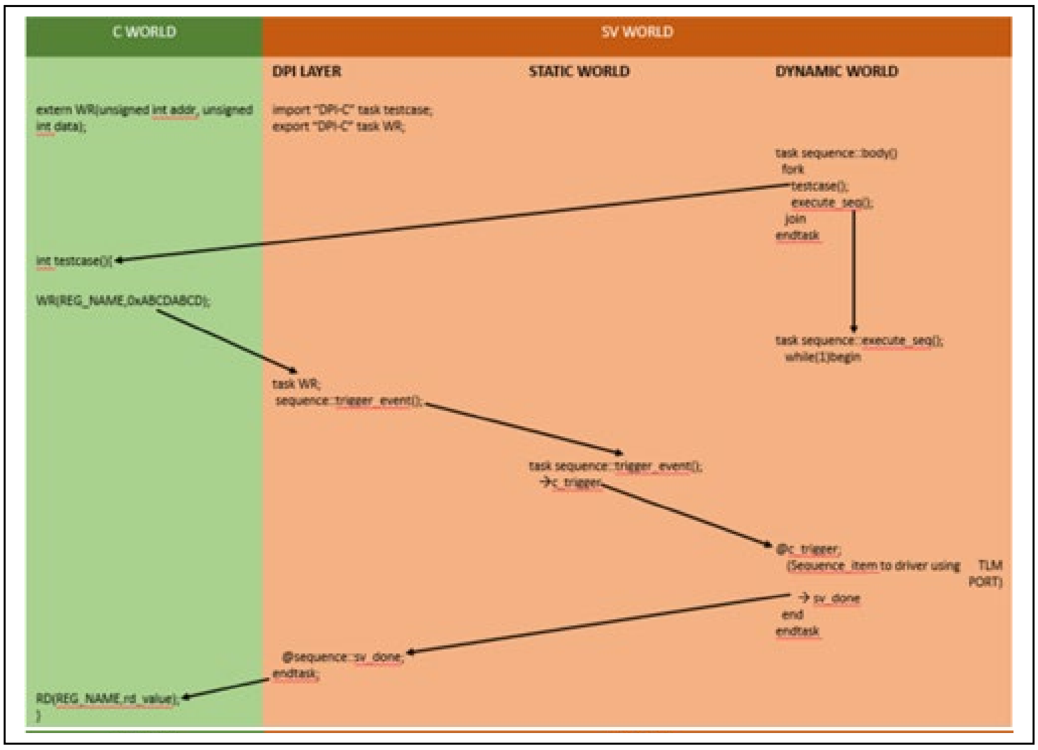
```
import "DPI-c" context task run_startup();
import "DPI-c" context task run_test();
import "DPI-c" context task run_shutdown();
```

```
int test_name{
 INSTALL_ISR(INTR,intr_rou);
 WR(REG1, 0xABCDABCD);
 CLR(REG2, REG_FILED );
 }
```

Figure 5 C and SV communication through DPI calls.

### E. *Test execution flow*

SV world will initiate and end test execution. Figure 6 depicts the test execution flow considering the implementation for register read/write API's. The run_test() api which invokes the simulation further transfers the controls to sequence body , which has the implementation tasks to invoke further the C based test through DPI calls as well execution sequence in parallel. C test further calls the instructions as in WR/RD API's in C which are calling the Systemverilog counterpart by generating the trigger_event, based on this trigger event the task waiting for c_trigger further gets unblocked and start executing the command received from C side. This communication is done via NON-BLOCKING TLM PORT which completes the loop implemented in DPI layer and hence one command processed and now C jumps to next command/instruction.



```
extern WR(unsigned int addr, unsigned
int data);
```

```
import "DPI-C" task testcase;
export "DPI-C" task WR;
```

```
task sequence::body()
 fork
  testcase();
  execute_seq();
 join
endtask
```

```
int testcase(){

WR(REG_NAME,0xABCDABCD);
```

```
task sequence::execute_seq();
  while(1)begin
```

```
task WR;
 sequence::trigger_event();
```

```
task sequence::trigger_event();
 →c_trigger
```

```
@c_trigger;
  (Sequence_item to driver using     TLM
                                     PORT)
  → sv_done
 end
endtask
```

```
 @sequence::sv_done;
endtask;
```

```
RD(REG_NAME,rd_value);
}
```

Figure 6 Test execution flow

## *F. Reset Handling*

Reset handling has been done considering that reset of various types (SoC/DUT specific) can be generated implicitly (where DUT itself generates the Reset) and explicitly (where C - test triggers the reset) can be handled by this approach. A reset env has been created which act as a base layer for all connections and abstract functions for execution. Depending upon the DUT/SoC requirements the extended reset env can be created and overridden. BFM which can be called as CBFM (C based calls from Test to UVM SV agent) has the event-based analysis port which is connected to reset env monitor. This implementation makes sure to get sampled both implicit as well as explicit reset on the reset interface. Further at emulation, there are various techniques available for handling this reset, which are vendor specific, but this paper demonstrates the basic implementation, where a TCL based driver has been created to insert the reset to synthesizable driver. This synthesizable driver is parametrized to make sure proper assert and deassert sequence to be followed as per the various types of reset (POR/functional reset/destructive reset).

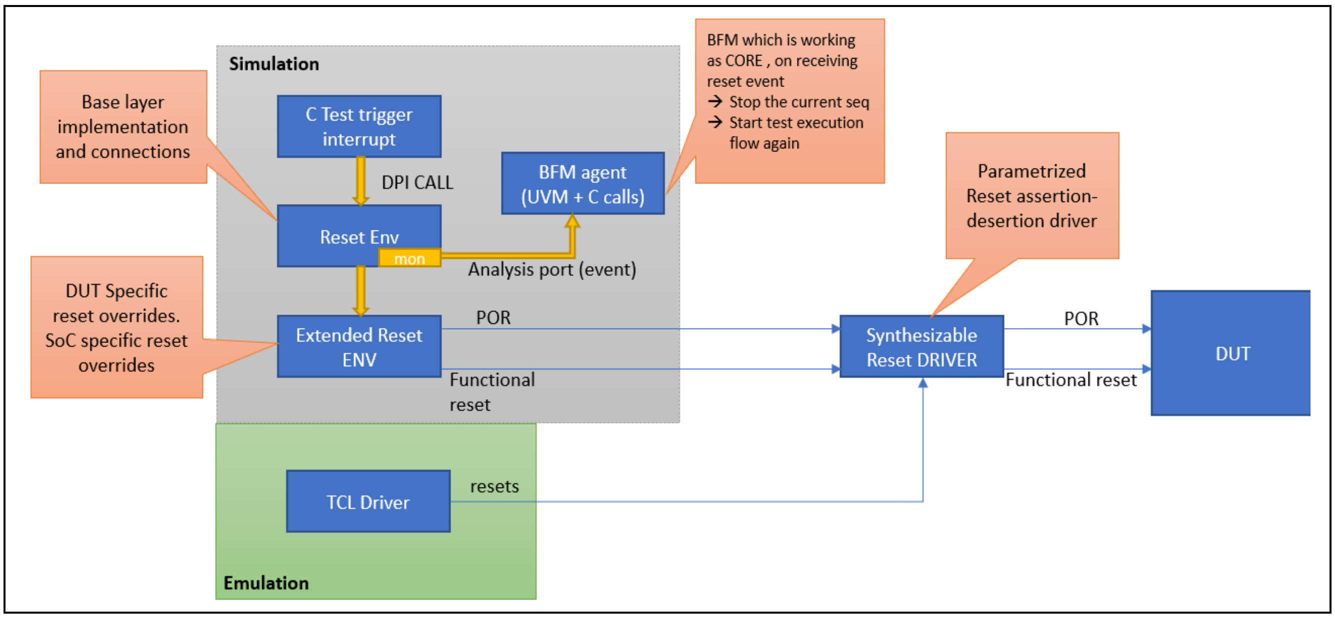

Simulation
C Test trigger interrupt
DPI CALL
BFM agent (UVM + C calls)
Reset Env
mon
Analysis port (event)
Extended Reset ENV
POR
Functional reset
Base layer implementation and connections
DUT Specific reset overrides. SoC specific reset overrides
BFM which is working as CORE , on receiving reset event
→ Stop the current seq
→ Start test execution flow again
Parametrized Reset assertion-desertion driver
Synthesizable Reset DRIVER
POR
Functional reset
DUT
TCL Driver
resets
Emulation

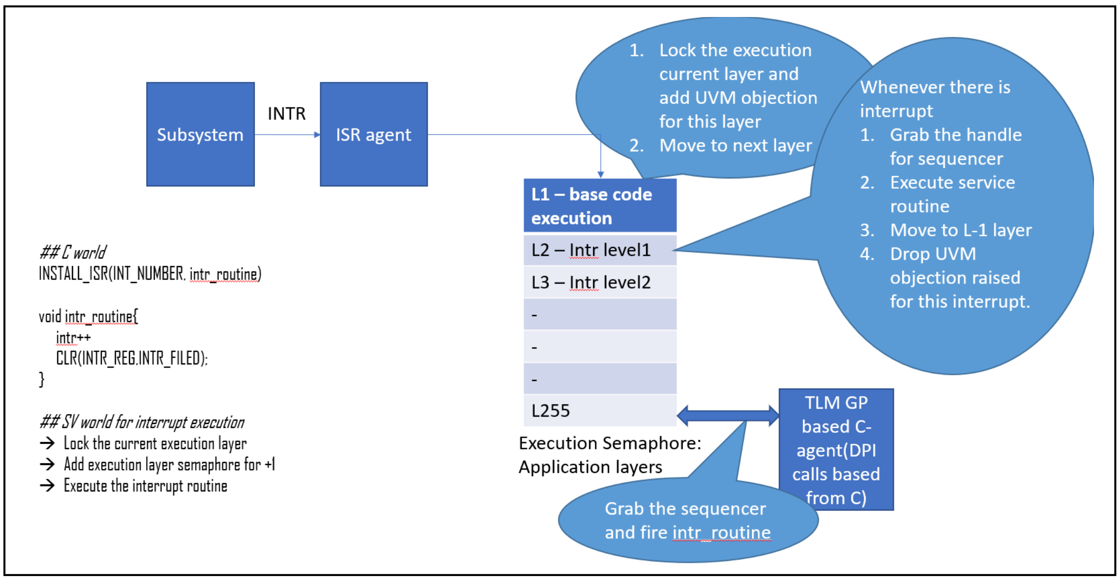
Figure 7 Reset Handling

*G. Vertical reuse*

AMBA UVM based BFM which are in active mode at Subsystem level are overridden via factory to be in passive mode to monitor protocol activity when used at SoC. This helps to monitor the protocol activity by leveraging the efforts made at subsystem.

*H. Interrupt Modelling*

Figure 7 demonstrates the interrupt handling and execution. A 256-depth semaphore is implemented for various level of interrupt execution. All the processes are executed at L1 level of semaphore, once there is an interrupt, base application layer L1 is locked making sure no new transaction will be processed and control moved to L1+1 next layer where interrupt service routine will be served.


```
## C world
INSTALL_ISR(INT_NUMBER, intr_routine)

void intr_routine{
   intr++
   CLR(INTR_REG.INTR_FILED);
}

## SV world for interrupt execution
→ Lock the current execution layer
→ Add execution layer semaphore for +1
→ Execute the interrupt routine
```
Subsystem
INTR
ISR agent
1. Lock the execution current layer and add UVM objection for this layer 2. Move to next layer
Whenever there is interrupt 1. Grab the handle for sequencer 2. Execute service routine 3. Move to L-1 layer 4. Drop UVM objection raised for this interrupt.
L1 – base code execution
L2 – Intr level1
L3 – Intr level2
-
-
-
L255
Execution Semaphore: Application layers
TLM GP based C-agent(DPI calls based from C)
Grab the sequencer and fire intr_routine

Figure 8 Interrupt execution diagram

### I. *CBFM vs CORE for execution*

This paper emphasis on using UVM based driver through C bases DPI call since with various experiments it has been observed that it is minimum 3x faster than executing it through core. Keeping the entire processing in SV so that SV powerful execution and simulator support can make the simulations faster.

### J. *Gate Level Synthesis (GLS)*

- Functional verification needs to sample one of the following:
  - virtual SDF – which doesn't need TB to drive input skews
  - Wrapper based - timing wrapper on TB to ensure input skews as captured by BE data.
- Random deposits on Non-resettable flops – may use Perl script
- All the Netlist code will be compiled in parallel to the TB and merged at the end using super block, which saves the time of GLS compilation.
- The results are enormous using this approach, a huge benefit of fast GLS closure has been achieved.

### K. *Direct re-use of testbench and testcases in emulator*

- Frontend/RTL Changes for emulation:
  - Update the memory view from Behavioral to Vendor specific
  - Update the library cell view from Behavioral to RTL if any.
- Testbench Changes for emulation:
  - Replace the simulation clock generation logic with vendor/tool clock generation method
  - Replace the reset generation logic from TCL based approach
- Execution Flow:

Test case C file is used to generate the hex file which is loaded into the system memory at CORE boot vector address, soon after the reset the core reads the first instruction from memory and starts executing via execution core as shown in figure 9.

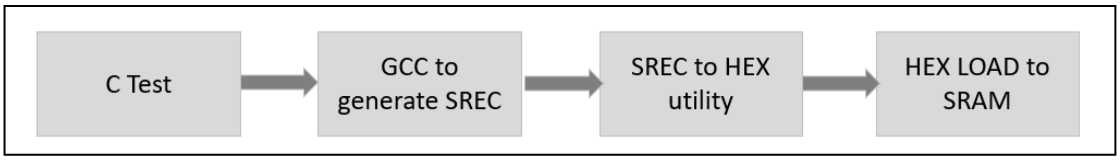


Figure 9 C to HEX loading for Emulator

## III. Application

- Stimulus creation that can be re-used in SoC level testbench.
- L-BIST and M-BIST verification.
- Gate level verification.
- Special purpose RTL simulations (Ex: To generate power analysis VCD).
- Direct re-use of testbench and testcases in emulator.
- DFT verification.

## IV. Summary

- The testbench architecture makes sure that it can be used seamlessly for native UVM verification. There is no need to have a separate UVM testbench for UVM sequence-based verification.
- RTL compilation/simulations much faster due to smaller sized database and fast UVM-SV based driver, hence fast testcase development leading early verification closure.
- Synthesizable Verilog based slave memories are used which eased portability to SoC emulation.
- Provides interoperable and protocol independent implementation since communication port is TLM 2.0 GP compliance.
- Powerful System Verilog constraint resolver, UVM sequence layering and ease of use assisting early verification closure.
- SoC Independent tests development (i.e. even when the sub-system has not been integrated into the SoC).
- TLM 2.0 compliance testbench can easily plugged with SYSTEMC hosts for early low-level driver development.
- GLS faster run time and can be run independent of the SoC.

ACKNOWLEDGEMENT

Authors would like to thank Ankit Khandelwal, Ravinder Dhanjjal, Ashish Goel and Rohit Malyan for their technical support. Further we would also extend thanks to our colleagues of NXP semiconductors.

## REFERENCES

[1] https://www.accellera.org/activities/working-groups/portable-stimulus

[2] Pranav Kumar, "A methodology for vertical resuse of functional verification from subsytem to SoC level with seamless SoC emulation", DVCON EUROPE 2014.

[3] Gordan Allan, "Tried and tested speedups for SW-driven SoC simulation", DVCON 2014

[4] S. Sutherlad, "Integrating SystemC models with verilog using system verilog direct programming interface." SNUG 2014

[5] www.systemc.org

[6] https://www.accellera.org/images//downloads/standards/uvm/uvm_users_guide_1.2.pdf