2014 DESIGN AND VERIFICATION DVCON CONFERENCE AND EXHIBITION INDIA
Sept 25-26, 2014
Hotel Park Plaza
Bangalore, India

Bring IP verification closer to SoC

Scalable Methods to Bridge the Gap between IP and SoC Verification

Gaurav Gupta, Tejbal Prasad, Rohit Goyal, Sachin Jain, Vipin verma
Automative Industrial Solutions Group
Freescale Semiconductor
Noida, India

B42437@freescale.com, B18358@freescale.com, B37481@freescale.com, B31084@freescale.com, B32849@freescale.com

***Abstract*—** With size of designs ever so increasing, Verification is becoming a bottleneck in modern day designs. Advent of '*Constrained Random, Coverage driven verification'* has greatly influenced the dynamics of verification and provided much needed controllability and observability. Introduction of re-usable methodologies, like '*UVM'*, has greatly helped the standardization among verification community and has provided much needed scalability and *distributability*.

Block level verification is the major beneficiary of the new methodologies, unfortunately not much of the benefits have propagated to SoC. Majority of SoC verification is still accomplished in more directed way barring the re-use of monitors and checkers from the IP verification environment.

To cater to exponentially growing complexity and ever so shirking '*Time to Market*', we need to find better ways to achieve our verification goals faster. Distinction between IP and SoC verification is getting obscured as dynamics of verification is shifting to a '*Sub-System*' containing multiple IPs working together. There is a clear need to make SoC verification and IP verification more '*inter-reusable*' in-order to mitigate not just the issues in modeling of environments around Standalone IPs versus actual SoC/Sub-System but also to empower system level verification environment with scalable and re-usable methodology which defines guidelines about how to handle and manage verification problems efficiently in structured manner. It would be desirable to not just be able to port stimulus from IP verification environment to an SoC verification environment but to have SoC environment an extension of the IP verification environment.

Keywords— IP, SoC, UVM, DUT, BFM, CORE, PORTHOLE, RAL

I. INTRODUCTION

This paper reflects our work to bridge the gap between IP and SoC verification. Through this paper, we share our experience to build a reusable methodology to control *Processor Cores*, which form the back-bone of any SoC environment, through '*sequences'* in a standalone '*UVM*' based environment. This methodology eliminates the need to develop 'C' patterns on the processor core as is done in traditional SoC verification environment to mimic the embedded software and instead an IP transaction is dynamically converted into an instruction to be run on processor core. There is no need to replace the processor core with a bus functional model (BFM) as is sometimes done in traditional SoC verification environments. The proposed methodology enables the use of advanced verification concepts offered by languages like '*System verilog'* and methodologies like '*UVM'* while doing SoC verification, resulting in a greater control and exhaustive verification.

The proposed methodology does not require existing IP or SoC flows to be altered and running embedded software is still seamless.

We share a case study to show how seamless it becomes to scale up IP verification environment to SoC by using this methodology. We also share the methods and apparatus to enable this methodology to any SoC without hindering existing IP and SoC flows. We have taken *UVM* as our reference verification methodology in rest of this paper.

II. IP AND SOC VERIFICATION – TWO DIFFERENT WORLDS

Figure 1 shows a basic execution flow in an IP verification environment versus that in a SoC verification environment.

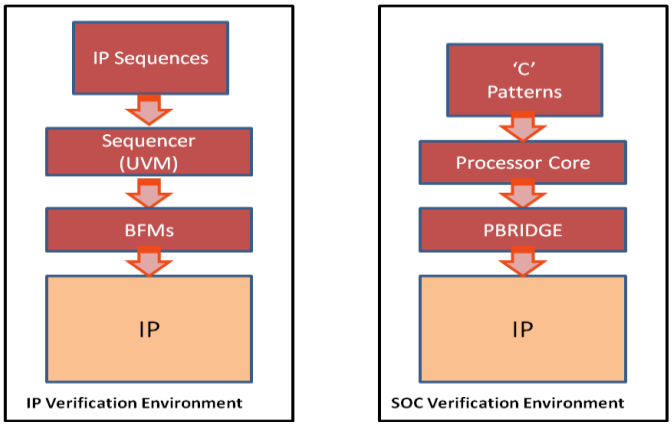

Figure 1 : Execution in IP and SoC environment

Typically at IP level, sequences run on a 'sequencer' which then drives the stimulus to the *DUT* (IP) via *BFMs*. At SoC, 'C' patterns, mimicking embedded software, are executed on processor core and the *DUT* (IP) is driven through a peripheral bridge. Due to this difference in execution, there is hardly any pattern re-use from IP level to SoC level which leads to effort duplication when IP is moved to a SoC environment. Growing complexity and shrinking '*Time to market*' has forced users to evolve and adopt specialized languages like System Verilog and standard methodologies such as UVM. These languages are rich in features to address the verification problems at higher abstraction level. Methodology defines standard guidelines about how to solve the verification problems effectively. We are now able to build an eco system where our verification environments are standardized and are using best practices to solve the problems. We now have tools to effectively apply '*Divide and Conquer*' principle to distribute the complexity by building a layered environment. Sequence partitioning helps us to get greater control especially with multiple IPs working together in a sub-system. Proliferation of these features seems to be significantly more at IP level than the sub-system and SoC, especially if they have processor cores. Presence of a processor core makes sub-system or SoC environment to look very different from an IP environment and require all together a different execution flow where 'C' patterns are executed on processor cores to mimic the embedded software restricting re-use from IP environment. Bridging this gap is need of the hour.

III. BRIDGING THE GAP

To bridge this gap, we considered two major aspects. One is portability of stimulus from IP to SoC environment and the other one is to maximize the benefits of UVM while working on SoC environment. We deployed our scheme to verify a system containing multiple complex IPs working together concurrently with high degree of interdependent interactions. We have shared requirements and results in a case study later in this paper.

Figure 2 shows the proposed approach we have implemented to address the issues. Idea is to be able to control the processor core through UVM sequences and eliminate any need to execute 'C' patterns to do the same. The proposed approach provides us two major benefits:

1. IP level sequences are re-usable at SoC
2. SoC verification environment can be built with UVM with all its capabilities just like an IP verification environment

In our proposed approach, we have developed an adapter to facilitate Hardware-Software communication. we execute SV/UVM sequences on UVM sequencer but unlike IP verification, sequencer drives transaction to an adapter instead of an IP via a BFM. The adapter dynamically translates transaction from sequencer to an atomic instruction and communicates with the processor core adhering to a protocol which we call *porthole protocol.* The instruction then is executed by the processor core. The *porthole protocol is defined later in the paper.* Adapter has a one to one mapping with processor core and multiple instances of the adapter can be created if there are multiple processor cores in the system.

We have also added capability in the adapter to dumps out 'C' files containing these atomic instructions that can be re-used at system/SoC level for verification and validation using legacy environment. This enables automatic conversion of IP sequences to a re-usable pre-verified 'C' code.

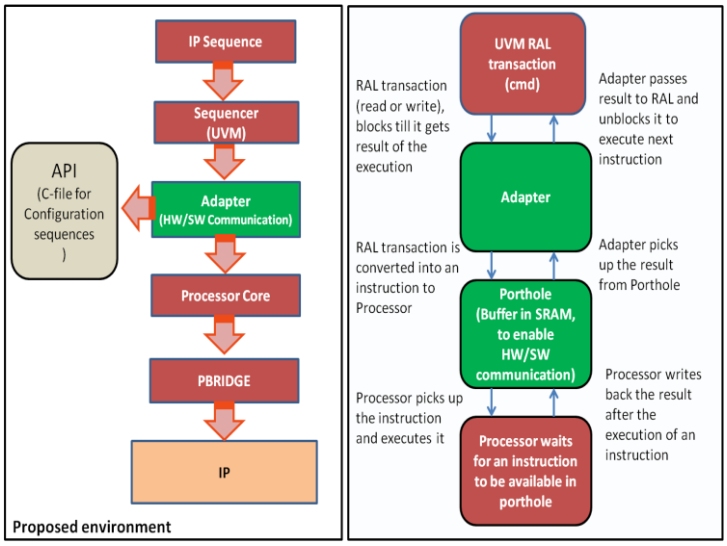

Figure 2 : Proposed Approach

Figure 3 shows execution of a '*write*' and Figure 4 shows execution of a '*read*' transaction by processor core which have been initiated by a UVM sequence. UVM has added RAL framework to mimic DUT's register interface. AT IP level, RAL executes the transactions that are executed by processor core at SoC level. We identified this as a good candidate to implement a scheme which provides seamless handshake between processor core and IP level sequences. In our proposed approach, a transaction initiated by an IP level sequence is dynamically converted to an instruction to the processor core. Simple '*porthole protocol*' has been defined and implemented to facilitate communication between adapter and the processor core to complete the execution of generated instruction.

The porthole protocol provides a seamless handshake between UVM RAL adapter and the processor core. Figure 5 shows the flow chart for the implemented protocol.

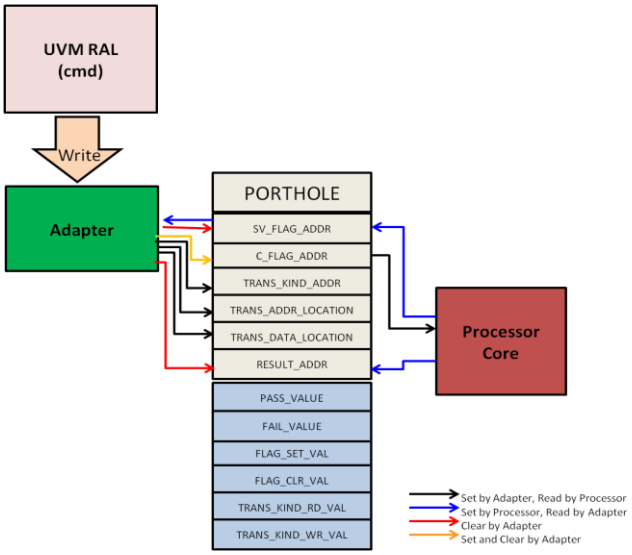

Figure 3 : Execution of a 'Write'

Whenever there is a transaction initiated using UVM RAL, it is converted into an instruction for the processor core inside '*reg2bus*'or '*bus2reg*' methods of UVM RAL and is driven by adapter to a porthole (small buffer in SRAM). The adapter is developed like a VIP that communicates with processor core adhering to '*porthole protocol*'. Developing adapter as a VIP makes it re-usable and portable across the projects.

The porthole protocol is defined in following steps:

1. Processor core initially waits on a porthole location called 'C_FLAG_ADDR' to be set to the value specified as 'FLAG_SET_VAL'.
2. When adapter receives any UVM RAL transaction, *It* writes value defined as 'TRANS_KIND_RD_VAL or TRANS_KIND_WR_VAL' respectively depending upon a 'read or write' transaction on a porthole location defined as 'TRANS_KIND_ADDR'.
3. In case of a write transaction, adapter also writes address where write is to be executed at the porthole location 'TRANS_ADDR_LOCATION' and the data to be written to 'TRANS_DATA_LOCATION'.
4. Adapter then writes a value defined as 'FLAG_SET_VAL' to the porthole location 'C_FLAG_ADDR' to indicate to processor core that a valid instruction is available in the porthole buffer and needs to be executed.
5. Adapter then waits on a porthole location called 'SV_FLAG_ADDR' to be set to 'FLAG_SET_VAL' that indicates that processor core has completed the execution of the instruction.
6. Processor core on the other hand picks up required instruction from the porthole locations and executes it.
7. Processor core then writes 'FLAG_SET_VAL' to porthole location 'SV_FLAG_ADDR' to indicate that it has executed the instruction.
8. Adapter finally clears both the flags ('C_FLAG' and 'SV_FLAG') by writing a value defined as 'FLAG_CLR_VAL'.

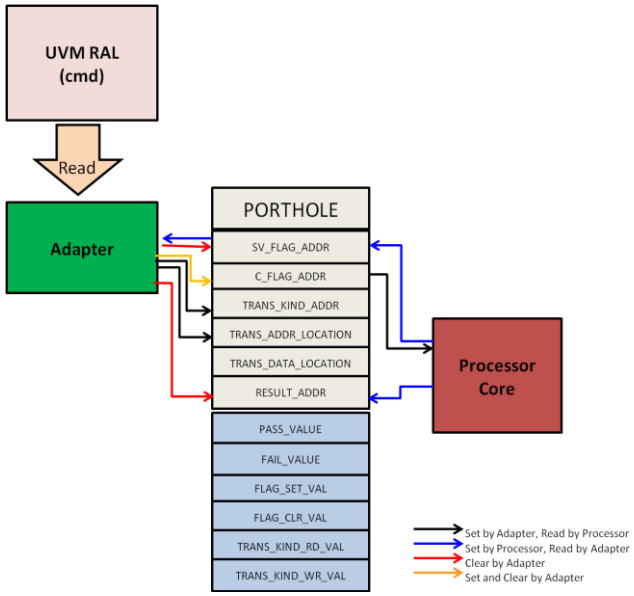

Figure 4 : Execution of a 'Read'

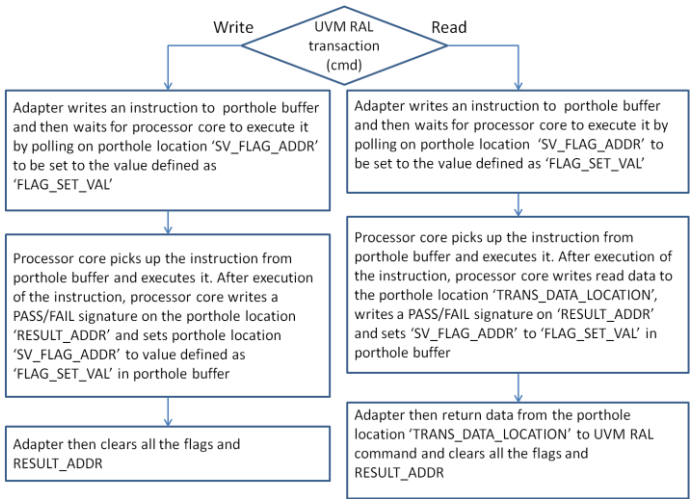

Figure 5 : Flow Chart for '*Porthole Protocol*'

We have added 'C' code that needs to be executed on processor core to facilitate Hardware/Software (HW/SW) communication, as specified by the *porthole protocol*, as part of Appendix II.

We are able to standardize the verification environment at SoC using this approach. IP sequences are seamlessly ported to SoC. The environment is build based upon layered architecture providing fine grain control on different IPs in the system. Users from both IP and SoC domain can co-relate with the environment easily now because of the standard execution flow across IP and SoC.

IV. A CASE STUDY

Figure 6 shows how this methodology has been implemented on a real system. The transaction initiated by IP sequence is now converted to an instruction by adapter instead of being driven to IPs (peripherals P0, P1 … Pn) via BFMs. Adapter communicates with processor core via a *porthole buffer*. *Processor core* execute the instruction to drive the transaction which is routed by crossbar (XBAR) and peripheral bridge (slave port S1 on XBAR) to the target IP peripheral (P0 or P1 or … Pn) on a peripheral bus.

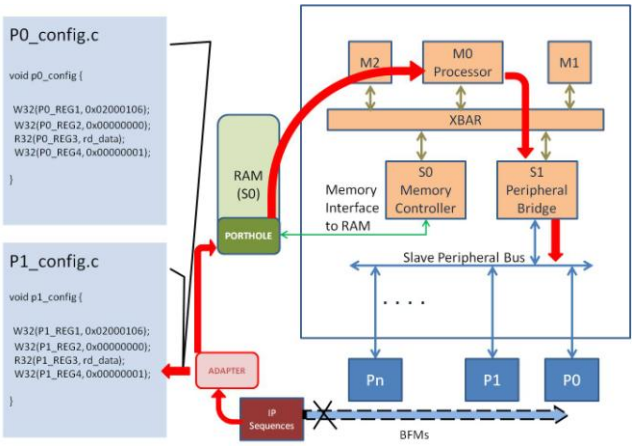

Figure 6 : Case Study

Adapter also dumps out multiple 'C' files, one per peripheral, which contain a function with generated instructions. These files can be used in legacy SoC environment to run in a traditional way. Interestingly, we can run the above environment with these 'C' files as well because the above environment is not much different from the traditional one but we have better control on it now due to better structuring and control flow provided by UVM. The next section explains about the execution flow in detail.

V. TOO DIFFERENT BUT TOO ALIKE

Figure 7 depict the actual system more closely. In the system, a camera image is processed by multiple components (shown as component 1 and component 2) and a RAM is used as the place holder for the data.

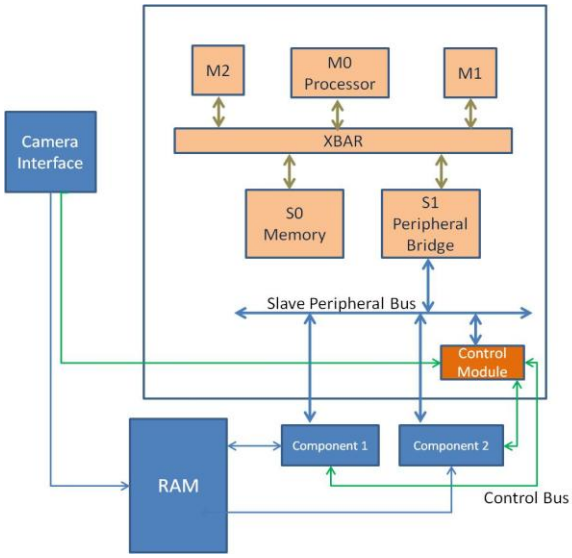

Figure 7 : Platform & Peripherals

Individual components have been verified at block (IP) level using UVM based verification environments. To verify the subsystem, block level testbenches required to be ported to the subsystem. Presence of a processor core (connected to the port M0) makes it look like a SoC. We applied the new methodology to enable maximum re-use of IP verification environments at sub-system and at the same time to have fine grain control at the top level to take care of the various interdependent interactions among different peripherals. The verification environment developed at sub-system is controlled via UVM sequences. IP level sequences are ported to sub-system seamlessly. There is no need to develop any 'C' pattern at sub-system now.

We have also added flexibility in the verification environment at sub-system to run in it in a traditional manner by executing the 'C' patterns on processor core. This helps us to simulate our environment along with the '*embedded software*'. Embedded Software is typically provided by the software teams and would be written in a language like 'C'.

We are able to test 'C' code generated by adapter in our environment using this approach. Figure 8 shows how we are able to make use of this alternate approach to test the 'C' code that adapter generate to provide backward compatibility with legacy environment.

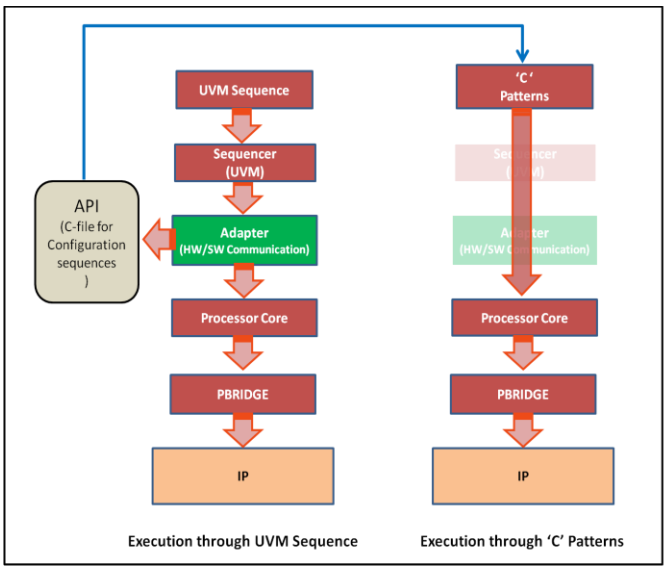

Figure 8 : Execution Flows

Figure 9 shows required processing flow and how the verification environment is structured to get the fine grain control on low level components.

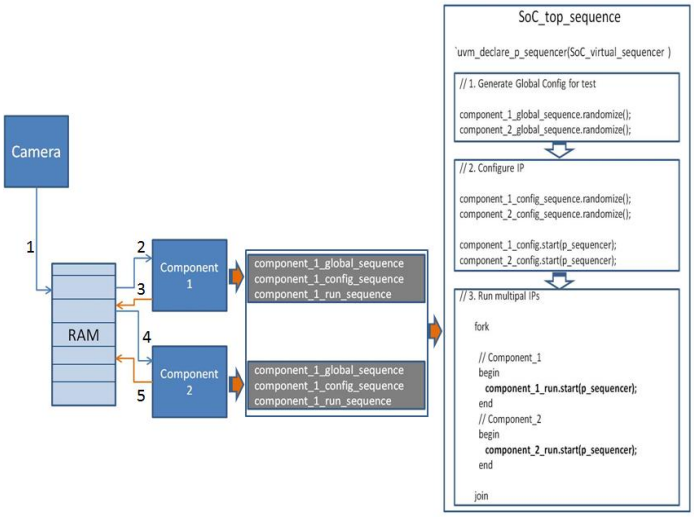

Figure 9 : Processing Flow

The input data from camera needs to be processed by component 1 first. Once component 1 completes its processing, the processed data again needs to be processed by component 2. This sequencing can differ in a case to case basis.

We have built a UVM based environment to achieve this control flow. IP verification sequences, pertaining to each component, have been ported and a top level sequence has been written encapsulating all the required sub-sequences. This layering of sequences helps to divide whole processing flow into multiple low level tasks. This layered testbench architecture embedded with the adapter helps to build a re-usable testbench at SoC level and provide a much needed control among different peripherals right at the top level. To execute different cases, only the top level sequence needs to change now which can directly be controlled from the testcase.

Additionally, we can also provide an image for processing in terms of a file and then processed data is also dumped in a file. This processed data then can be used as a reference data while doing verification using traditional approach by executing 'C' patterns on processor core.

The system under verification is very much similar to a SoC but the testbench to verify it is similar to a block level testbench.

VI. EXECUTING GENERATED 'C' PATTERNS

Making a 'C' program that can be executed by processor core, with the help of generated 'C' configuration functions, is very easy now. Figure 10 shows an example:

```
// Generated 'C' function for component 1
void component_1_config()
{
W32 (COMPONENT_1_REG1, 0x03200000);
W32 (COMPONENT_1_REG2, 0x00010101);
}

// Generated 'C' function for component 2
void component_2config()
{
W32 (COMPONENT_2_REG1, 0x03200000);
W32 (COMPONENT_2_REG2, 0x00010101);
}

// Function to trigger all the IPs
void Run_all_IPs()
{
int rdata1, rdata2;
W32 (COMPONENT_1_MCR, 0x01);
W32 (COMPONENT_2_MCR, 0x01);
R32 (COMPONENT_1_MSR, rdata1);
R32 (COMPONENT_2_MSR, rdata2);
```

```
while ((rdata1 != 0x0) && (rdata1 != 0x0))
{
R32 (COMPONENT_1_MSR, rdata1);
R32 (COMPONENT_2_MSR, rdata2);
}
}

// Main function – execute the processing flow
void main()
{
component_1_config();
component_2_config();
Run_all_IPs();
}
```

Figure 10 : 'C' Program to execute on processor core

We also need to disable the drivers of different peripherals now as processor core has become the master in this flow. We can very easily do this now in our test by making it as a passive component by setting it as 'UVM_PASSIVE' without changing the existing environment.

We can also use multiple phases provided by UVM to make our testbench execution layered to get better control. As shown in Figure 11, we have made three sequences, one to boot up the core (core_startup_seqs) executed in 'configure' phase of UVM, second to execute a 'C' program (core_seqs) in 'main' phase of UVM and third and last to check the results (core_check_processed_op_data_seqs) in 'post_main' phase of UVM.

```
// Disable drivers of 'component 1 and 2' PASSIVE as they don't drive the IPs now
uvm_config_db#(uvm_active_passive_enum)::set(this,"env","
component_1_is_active",UVM_PASSIVE);
uvm_config_db#(uvm_active_passive_enum)::set(this,"env","
component_2_is_active",UVM_PASSIVE);

// Execute a 'startup sequence' to boot the CORE in 'configure phase'
uvm_config_db#(uvm_object_wrapper)::set(this, "*.vsequencer.configure_phase",
"default_sequence", core_startup_sequence::type_id::get());

// Execute a 'C' program made after combining 'generated C functions' in 'main phase'
uvm_config_db#(uvm_object_wrapper)::set(this, "*.vsequencer.main_phase",
"default_sequence", core_seqs::type_id::get());

// Execute a sequence to check final processed data with the reference data in
// post_main phase when all the processing has completed
uvm_config_db#(uvm_object_wrapper)::set(this, "*.vsequencer.post_main_phase",
"default_sequence", core_check_processed_op_data_seqs::type_id::get());
```

Figure 11 : Top level Control

VII. CONCLUSION

To cater to growing complexity and shrinking execution time, methodologies like UVM have been forthcoming but had very limited usage at SoC level so far. Architecturally, distributed system implemented by today's SoCs also requires greater control on its individual components and it is always desired to re-use the exhaustive work that has happened during IP level verification. Sub-System approach has bridged the gap in understanding of varying verification requirements between an IP verification engineer and SoC verification engineer so it has become even more compelling to provide a verification methodology that bridges the gap between these two worlds. Our work described in this paper has attempted to do the same.

VIII. APPENDIX I

Figure 12 shows 'C' Code that processor core needs to execute in order to communicate to the '*porthole buffer*'.

```
int execute_ral_sequence()
{
  int    *cFlagAddress;
  int    *svFlagAddress;
  int    *transactionAddress;
  int    *memAddressLocation;
  int    *dataAddressLocation;
  int    *memAddress;
  int    *dataAddress;
```

```
  int     j=0;
  uint32_t   flag_val;
  uint32_t   read_val;
  uint32_t   wr_val;

  flag_val = FLAG_SET_VAL;

  cFlagAddress = C_FLAG_ADDR;
  svFlagAddress = SV_FLAG_ADDR;

  memAddressLocation = TRANS_ADDR_LOCATION;

  while(1)
  {
    // Waiting For new instruction to be available in porthole buffer
    while(*cFlagAddress != flag_val){};

    // Read instruction attributes
    transactionAddress = TRANS_KIND_ADDR;

    memAddress = *memAddressLocation;
    dataAddress = TRANS_DATA_LOCATION;

    // Executing Read instruction
    if(*transactionAddress == TRANS_KIND_RD_VAL)
    {
      read_val = *memAddress;
      *dataAddress = read_val;
      *svFlagAddress = flag_val;
    }
    // Executing Write instruction
    else if(*transactionAddress == TRANS_KIND_WR_VAL)
    {
      wr_val = *dataAddress;
      *memAddress = wr_val;
      *svFlagAddress = flag_val;
    }
  }

  return 1;
}  // END execute_ral_sequence
```

Figure 12 : ‘C’ Code

IX. REFERENCES

[1] Matthew Ballance, **“**Boost verification results by bridging the hardware/software testbench gap”. Available: https://s3.amazonaws.com/verificationacademy-news/DVCon2013/Papers/MGC_DVCon_13_Boost_Verification_Results_by_Bridging_the_Hardware_Software_Testbench_Gap.pdf