Abstract Layer for Firmware Access: A Unique Approach for SOC Functional Verification

[1]Gupta Girish Kumar [2]Rugmini Navia Vishnu [3]Choudhary Praval [4]Syed Mobeenuddin

[1]Samsung Semiconductor India, Bangalore, India, g.girish@samsung.com; [2]Samsung Semiconductor India, Bangalore, India; navia.vr@samsung.com, [3]Samsung Semiconductor India, Bangalore, India, praval.ch@samsung.com, [4]Samsung Semiconductor India, Bangalore, India, mobeen.syed@samsung.com

Abstract- System-on-chip (SOCs) complexity is growing many folds in the recent past; SOCs are ranging from single to multi-core processors, more complex than ever interconnects and multiple stakeholders are involved in functional verification hence verifying such complex SOCs becomes challenging. Software teams are developing the firmware and hardware teams are developing the verification environment using different languages and methodologies. In industry, abstract languages like System Verilog/UVM is prominently used for developing verification environment, similarly, C/C++ is widely used for firmware development. It is extremely challenging to have a constructive interaction between the testbench and firmware due to different compilation sequence and the inability to control firmware at simulation run time. This paper proposes a unique approach to have an efficient interaction between the firmware and testbench at run time. Testbench can query the firmware at run time, instruct the firmware to alter its states, and change the hardware characteristics as per test. Hardware-Software co-debugging becomes easy as the testbench can alter the firmware and hardware behavior based on test scenarios. Overall verification efforts are reduced by approx. 50% using this abstract layer approach.

I. INTRODUCTION

Modern System-on-chips are complex interconnections of different IPs, single/multi-core processors; memories, DMAs, and bus interconnects. In an ASIC design cycle, verification/validation of such SOCs needs to be started at the early stage of design development to meet the time to market requirement. Hardware-Software co-simulation is a key that makes it possible to have functional verification of such complex SOCs at an early stage.

In Hardware-software co-simulation, controlling firmware at run time from the testbench is a challenging task as the firmware is compiled statically and binaries are loaded into processor instruction memory before the processor starts execution. However, in case, firmware configuration needs to be changed, then the only way is to modify the firmware, re-compile and reload binary to processor instruction memory. For each firmware configuration, this process has to be repeated and the developer has to manage multiple versions.

HVL language like system Verilog comes with inbuilt constraint randomization and functional coverage whereas software language like C/C++ do not support it. The proposed approach will bring in constraint randomization capability, functional coverage and timing delay insertion during simulation run time.

Ref [4] approach is to coordinates the S/W and HVL stimulus. This approach has the software stimulus as master, triggering HVL stimulus (e.g. UVM sequences) via a memory -mapped adapter. The adapter is, implemented within the testbench, connected to the DUT processor bus or bus-to-memory signals. This adapter implements the linkage needed to transform the specific bus access to the required testbench stimulus or action.

Ref [5], UVM SDV (Software Driven Verification) package enables embedded software running on a processor model, enabling the embedded software to act as another verification component in the UVM testbench environment. This coordinates with other activity occurring in the System Verilog portion of the UVM testbench and publishes information on the state of the software for use by UVM testbench.

II. PROPOSED ARCHITECTURE

In the proposed approach we have, a System Verilog UVM based abstract layer on the top of the firmware that can instruct the firmware to operate. The interaction between the abstract layer and the firmware is through common memory that is accessible by both. The abstract layer is capable of querying the firmware at run time and altering the firmware state machine and hardware characteristics. The following are the salient features supported by the proposed architecture.

1) Controllability: Abstract layer can control the firmware; it has direct access to the firmware variables and descriptors. At run time, firmware functionality is dynamically overridden, which is not possible in the traditional approach.

2) Randomization: DUT configurations and timing delays can be randomized and provided to firmware to control the Hardware functionality.

3) Scalability: A scalable architecture that accommodates new features or interfaces to interact between the abstract layer and firmware, currently communication between them is through common memory connected on AXI bus; In future, the communication interface could be enhanced to support standard interfaces like JTAG, UART, etc. This abstract layer approach is scalable to the multi-core SOC.

4) Reusability: Abstract layer is horizontally and vertically reusable across various sub-system and SOC projects.

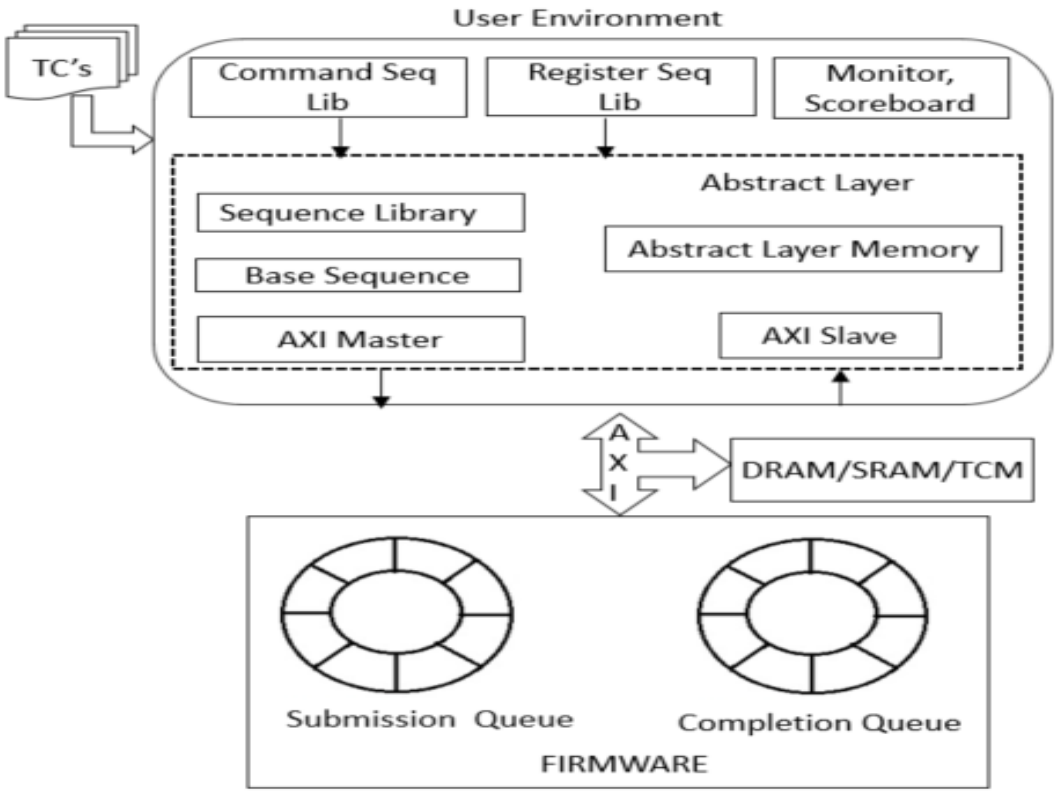

Figure 1. Top Level block Diagram

5) Observability/Debug-ability: Ability to track the data/control flow through the abstract layer using callbacks.

6) Zero C/C++ Test Cases: In the proposed approach SV –UVM test cases are written on top of the abstract layer and no C/C++ test cases are needed.

The proposed architecture in Fig. 1 uses the AXI bus snooping to interact with Firmware. Abstract layer perform AXI Write and Read operation onto a common accessible memory (DRAM, SRAM, or processors TCM) within system memory range. The abstract layer will register this memory range during the initialization time and firmware will create the circular submission and completion queues. All the communication through the abstract layer will happen using these queues in a non-blocking mode. Queue will contain multiple commands for the firmware to decode and execute. These commands can include the read-write on hardware special function registers to alter the hardware state, commands can instruct the firmware to go to a specific mode i.e. switch to polling mode from interrupt mode or vice versa.

As shown in Fig. 1, the Test cases coded by hardware teams create the required random register configuration and random command sequences using system Verilog constrained random methodology. The abstract layer (highlighted in dotted-box) typecasts these register configurations, command sequences into register descriptor, and command descriptor packets respectively. These are further processed by the firmware.

A. Proposed Vs Traditional Verification Approaches

In the traditional approach firmware and testbench are developed by different teams and there is absolutely no direct interaction/controllability between testbench and firmware. Firmware is compiled using ARM compilers and generated binaries are loaded into instruction memory. These binaries are generated for a particular configuration and run throughout the simulation. In case firmware configuration needs to be changed dynamically, it is not possible unless a new C/C++ test case coded, re-compiled and reloaded into the instruction memory. This adds an overhead of writing several C/C++ TCs for each configuration.

The proposed approach has an abstract layer on top of firmware which is capable of controlling it dynamically, existing SV-UVM TCs are modified for firmware controllability thus avoiding need of additional C/C++ TCs and firmware recompilation. This saves the overall simulation time, TCs development time, and debugging efforts.

III. ABSTRACT LAYER

In Fig. 2, The Abstract layer is built on top of AXI Verification IP and contains base sequence, sequence library, AXI master agent, AXI slave agent, Abstract layer memory and AXI monitor to sample transactions on the AXI interface with Firmware.

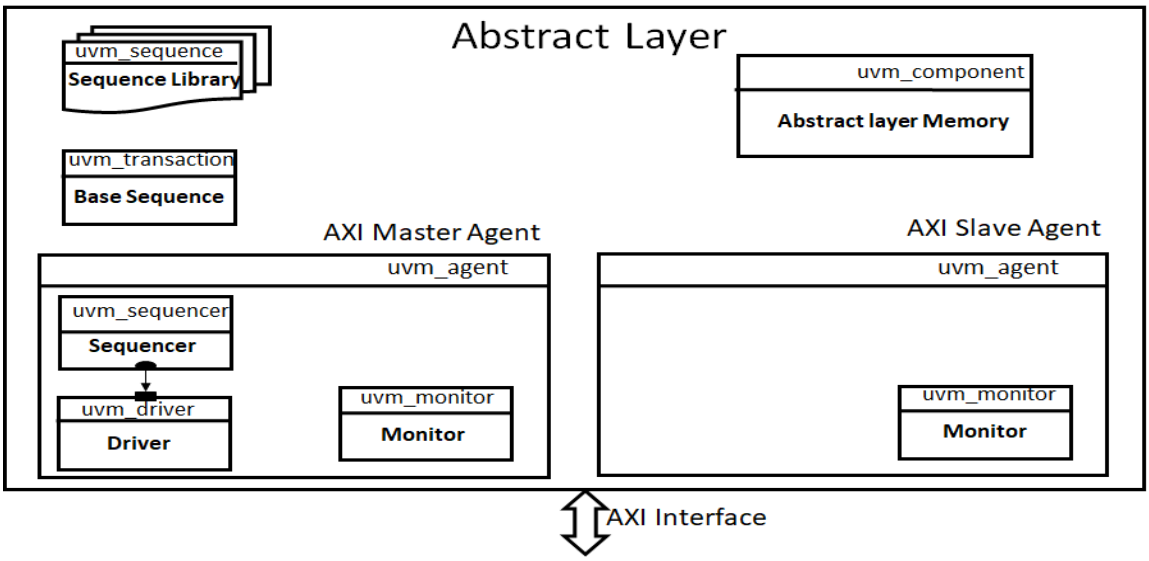

Figure 2. Abstract Layer: Detailed Abstract Layer Block Diagram

A. Base Sequence

```
// Base Packet Class (Pseudo Code)

class reg_axi_pkt extends axi_base_pkt;

        rand bit[31:0] reg_addr;
        rand bit[31:0] reg_data;
        rand bit [7:0]   opcode;

         `uvm_object_utils_begin (reg_axi_pkt)
        `uvm_field_int (reg_addr, UVM_ALL_ON)

         `uvm_object_utils_end

        function void post_randomize
           this.pack_data (wdata)
        endfunction
endclass
```

Figure 3. Abstract Layer Sequence Item code snippet

Base Sequence contains a collection of sequence items corresponding to the structure of various commands. The structure of the commands are defined using descriptor formats. In Fig. 3, pseudo code snippet for a sequence item is given. In Fig. 4, Register descriptor defined in the Base Sequence Library is shown. Similar formats are also devised for other command descriptors, which are firmware compatible.

Description															
D31	**D30**	**D29**	**D28**	**D27**	**D26**	**D25**	**D24**	**D23**	**D22**	**D21**	**D20**	**D19**	**D18**	**D17**	**D16**
D15	**D14**	**D13**	**D12**	**D11**	**D10**	**D09**	**D08**	**D07**	**D06**	**D05**	**D04**	**D03**	**D02**	**D01**	**D00**
Opcode[7:0] - Write 0x0, Read 0x1								Reserved							
Reserved															
LOWER_ADDRESS[31:0]															
DATA[31:0]															
Reserved															
Reserved												UPPER_ADDRESS[4:0]			
COUNT															

Figure 4. Abstract Layer Register Descriptor

In Table 1, the sample opcode of register descriptor is given. Different commands have unique opcodes.

TABLE I. COMMANDS OPCODE

Commands	Opcode
Register Write	8'h00
Register Read	8'h01
Memory Write	8'h02
GDMA	8'h03

B. Sequence Library

Sequence library provides APIs/tasks, which are used in test sequences to initiate various commands or register transactions. These methods will create the register/command packets based on the user inputs, typecast them to corresponding descriptor format, and sends to lower-level AXI sequences. Master agent translates these AXI sequences to AXI transactions.

In Fig. 5, pseudo code snippet of the Sequence available in Sequence Library is shown.

```
// Sequence Class (Pseudo Code)

  class abstract_layer_axi_sequence extends uvm_sequence;
    `uvm_declare_p_sequencer (axi_sequencer)

     reg_axi_pkt       reg_wr_axi_pkt
     reg_axi_pkt       reg_rd_axi_pkt
     virtual task reg_write (input bit [31:0] addr, input bit [31:0] data, input int count)
        reg_wr_axi_pkt = new ();
        if ( ! (reg_write_axi_pkt.randomize () with {reg_addr = addr;
                                                     reg_data = data;
                                                     opcode = 8'h00;
                                                     fw_collect_dword_count = count;
                                                    })
       // Task defined in AXI-Master-Agent to drive packets in AXI protocol
          axi_write (reg_wr_axi_pkt. Reg_addr, reg_wr_axi_pkt. Reg_data);
       endtask
 endclass
```

Figure 5. Abstract Layer Sequence Class Code snippet

C. AXI Master and Slave Agents

The AXI master agent comprises the sequencer, driver and monitor in accordance with UVM-methodology. The sequencer sends the packet to the driver through the TLM port. The driver transmits the packet onto the AXI interface to DUT.

The AXI slave agent is configured as a passive slave. The monitor present in the AXI slave continuously monitors the AXI interface for AXI transactions initiated from the firmware. The monitor snoops information such as the base address of the firmware queue in common accessible memory location and populates the register information in the Abstract layer memory.

D. Abstract Layer Memory

For efficient interaction between the firmware and abstract layer, the memory space within the Abstract layer is mapped for purposes such as storing read data from the DUT Registers, write pointer update, completion status update, etc. Fig. 6 shows sample memory map of the Abstract Layer memory space.

Address	System Memory
C0000	Reserved
A0000	DRAM
80000	Shared SRAM
60000	Abstract Layer Memory
40000	Data Memory
20000	Reserved
00000	Instruction Memory

Offset	Name
60300	READ_DATA_REG
60200	READ_POINTER_REG
60100	WRITE_POINTER_REG
60000	BASE_ADDR_REG

Figure 6. Abstract Layer Memory Space

Fig. 7 shows sample memory map in the Abstract layer testbench using Verilog macros.

```
// Memory Defines (Pseudo Code)

`define   ABSTRACT_LAYER_BASE_ADDR      `h0
`define   READ_POINTER_REG              ABSTRACT_LAYER_BASE_ADDR + `h10
`define   WRITE_POINTER_REG             ABSTRACT_LAYER_BASE_ADDR + `h100
`define   CQ_STATUS_REG                 ABSTRACT_LAYER_BASE_ADDR + `h200
```

Figure 7. Abstract Layer Sample Memory Map defines

IV. Abstract Layer and Firmware Interaction

The Test cases coded by hardware teams create the required random register/command configuration using system Verilog constrained random methodology. These commands are sent to the firmware by Abstract Layer's AXI master agent. The firmware and Abstract layer interacts based on the write pointer-read pointer update mechanism.

Fig. 8 shows the handshake between the firmware and the Abstract Layer for a register descriptor sent from the Abstract Layer.

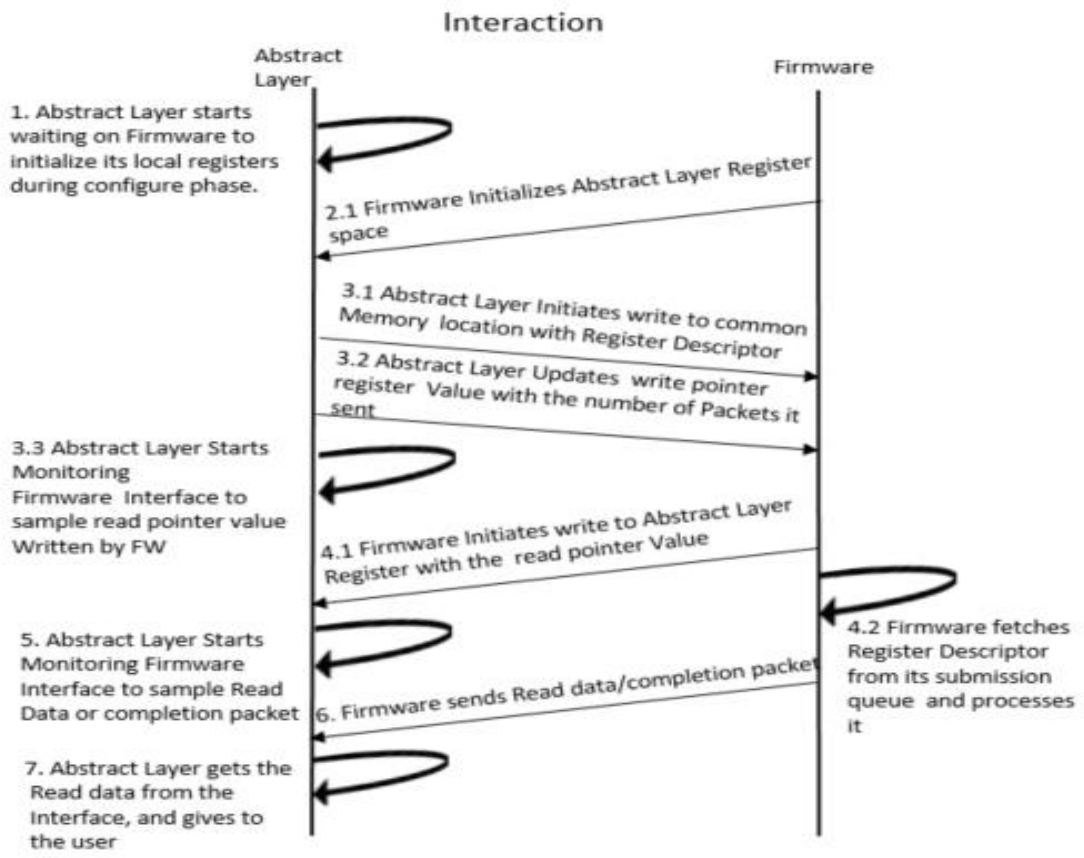

Figure 8. Abstract Layer and Firmware communication line diagram

V. FIRMWARE DESIGN

A dedicated Adaptor Layer is added in the firmware to communicate with Abstract Layer as shown in Fig. 9. At firmware compile time with the appropriate compiler directive, the decision is taken to run firmware in Abstract Layer mode or in conventional mode. Abstract Layer mode enables conventional operations as well as Adaptor layer. There are two main jobs of Adaptor layer:

1. Memory (Processor Memory, SRAM, DRAM and Register) Read-Write request handling
2. Abstract Layer command request and response handling

Memory read-write request handling is done by parsing of address and data embedded inside AXI transactions between Abstract Layer and Firmware.

In the conventional mode, Host posts the command into descriptor memory present in Core' Data Tightly Coupled Memory (DTCM) and updates the Submission Queue (SQ) write pointer. Firmware receives a notification about the SQ write and read pointer difference, it reads command descriptor, updates SQ read pointer, and start processing the command. When the command processing completes, firmware posts the completion in descriptor memory and updates the Completion Queue (CQ) write pointer. Finally, Host is notified about the command completion and acknowledge by updating the CQ read pointer.

In the Abstract Layer mode, Host does not send the command. Instead, Abstract Layer posts the command into Abstract Layer Reserved memory present in Core's DTCM and updates the Abstract Layer-SQ write pointer. On receiving the notification of Abstract Layer-SQ read and write pointer difference, the Adaptor layer of firmware performs the below operations:

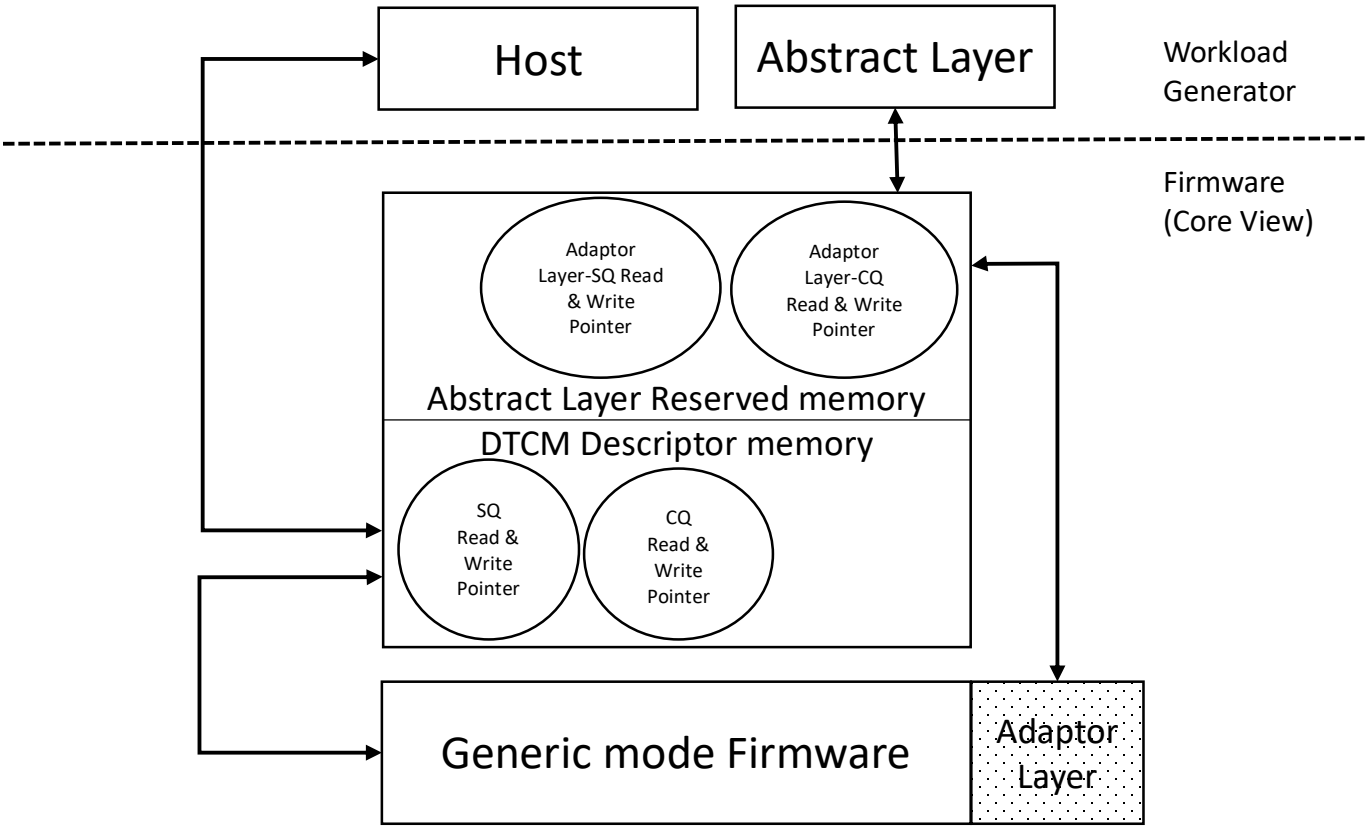

Figure 9. Firmware design diagram

1. Prepare the command descriptor based on command posted by Abstract Layer
2. Update the SQ write pointer inside descriptor memory
3. Update the Adaptor Layer-SQ read pointer to notify Abstract Layer about command receive

After this conventional firmware flow, takes place and command is completed. Firmware's Adaptor layer receives the notification of the CQ read pointer update and it updates the Adaptor Layer-CQ read pointer, which acts as a trigger to Abstract Layer for command completion. Abstract Layer updates the Adaptor Layer-CQ read pointer and acknowledge the command completion.

In this proposed approach, firmware's conventional flow is not changed. For Abstract Layer command handling, a separate layer is introduced. With this design, firmware can handle the command sent by Abstract Layer as well as by host.

VI. RESULTS

In Table-2, a comparison between the traditional SOC verification and proposed verification is given for an approx. 100 million gates design SOC. Test setup comprises System Verilog/UVM based Host verification environment along with multicore SOC design. Firmware binaries are loaded into each processor memory.

TABLE II. COMPARISON BETWEEN TRADITIONAL AND PROPOSED APPROACH

Sr. No.	Features	Traditional SOC verification Approach	Proposed SOC Verification Approach
1	Functional Coverage (Firmware dependent)	Not Available	Available
2	Efforts required for feature verifications, TC coding/modify	~ 2 Months	~ 3 Weeks
3	Firmware Tests (C/C++)	~400	0

In traditional approach Functional coverage, for firmware dependent features is not possible, user has to write multiple C/C++ TCs to cover firmware dependent features and there is no way to quantify coverage for all features. Whereas in the proposed approach, firmware is controlled by Abstract layer which has inbuilt functional coverage, thus use this matrix for verification sign-off.

In this proposed approach, verification engineer is also writing the tests for firmware features, hence there is less dependency on firmware developer and thereby reducing the overall development efforts required in Test Case coding and debugging. We have observed that for an approx. 100 million-gate design SOC, efforts are reduced by 4-5 weeks, which helps in gaining Time to Market. Another advantage we have observed that is we do not need any C/C++ firmware test cases. Test cases are written on top of Abstract Layer, which are managed by verification engineers, firmware developer has no role to play in writing firmware specific test cases. Also as the Abstract layer supports the randomization, constraint random or fully random tests are written which helps in un-earth corner case scenarios and reduces the overall TC development efforts.

VII. Conclusion

Abstract layer for firmware access is an approach, which can reduce the verification challenges, efforts and provide better controllability over the traditional SOC verification approach. We have currently deployed the proposed approach in a couple of memory controller SSD SOCs and succeeded in reducing verification efforts by approx. 50% in addition to functional coverage metric for signoff. Abstract layer approach is a way forward for complex SOC verification involving single/multi-core processors.

References

[1] *Accellera Organization, Inc.pp. 321-335* , System Verilog 3.1a Language Reference Manual

[2] *Mentor Graphics* UVM Cookbook, Verification academy

[3] *Wesley ISBN 978-0321563842. May 2013,* The C++ Programming Language Addison

[4] *Gordon Allen* , Tried/Tested speedups for SW-driven SoC Simulation

[5] *Matthew Ballance* , Boost Verification Results by Bridging the Hardware/Software Testbench Gap