Generic SCSI-Based Host Controller Verification Framework Using SystemVerilog

Mohamed Rayan
Mentor a Siemens Business
Cairo, Egypt
Mohamed_Rayan@mentor.com

Mohamed Samy
Mentor a Siemens Business
Cairo, Egypt
Mohamed_Samy@mentor.com

Haytham Ashour
Mentor a Siemens Business
Cairo, Egypt
Haytham_Ashour@mentor.com

Ashraf Salem
Mentor a Siemens Business
Cairo, Egypt
Ashraf_Salem@mentor.com

***Abstract*-Modern System on Chip (SoC) is becoming increasingly complex due to the advances in technology. Die size is increasing which allows packing more modules into the same SoC and hence the verification of such SoC is becoming a challenging task [1]. Host controller is an important component in an SoC. To complete the testing of a host controller, a real environment must be made where the host controller is integrated with a host machine running an operating system along with the host driver. Reaching this stage of testing is lengthy and debugging on the system level is time consuming. In this paper we provide a proposal for "Generic testing environment for Host controller" using SystemVerilog to mimic stimulus similar to the system level, allowing a thorough testing in an easy to debug environment. A case study is made demonstrating the methodology on two different Small Computer System Interface (SCSI) based host controllers compatible with Peripheral Component Interconnect Express (PCIe) interface; one supporting Universal Flash Storage (UFS) and the other supporting Serial-Attached SCSI (SAS).**

Keywords — SoC, HCI, Driver, HBA testing, SoC testing, Verification.

I. INTRODUCTION

A host controller or host bus adapter (HBA), sometimes called an HBA card, is a hardware device that represents the means of physical connectivity between a host system and a storage devices, servers or network peripheral, also provides input/output processing for data/command transfers between the host system and the peripheral [2]. It allows the host system to exchange information with a peripheral device attached to it and relieves the host microprocessor of the tasks such as storing and retrieving data which results in performance improvement in the host system [3]. The term HBA applies to a different types of interconnects, however it is commonly used with storage protocols such as Fibre Channel (FC) and Serial-Attached SCSI (SAS) [2]. HBAs are identified by their interconnect technology, speed and system interface. In x86 architecture, HBA is often connected to a PCI Express (PCIe) slot of a PC/Server as shown in Fig. 1, in other architectures (e.g. ARM, PowerPC) it may be connected to a standard interface like Advanced Extensible Interface (AXI). Fig. 1, shows the generic architecture of HBA where it is connected to host machine through a PCIe link from one side and to the peripheral device though a protocol specific (e.g. SAS, UFS) link on the other side. An operating system with low level driver is running over the host machine which sends the SCSI commands to host controller through a PCIe link. The PCIe endpoint takes the role of HBA PCIe interface with the host machine, it uses an AXI interface to drive the commands to HBA core. HBA initiates the communication with peripheral device through the protocol specific link to handle host machine requests.

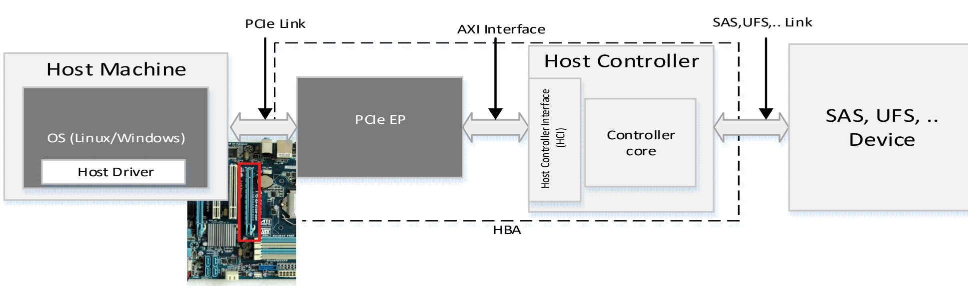

Fig. 1 HBA Generic Architecture

II. MOTIVATION

Functional verification in general is one of the major bottlenecks in the hardware IP and SoCs design, about 70% to 80% of the total effort in a hardware and IP designs is spent in the verification flow, and it is the most expensive job in-terms of cost and time. When it comes to complex SoC design (large gate count and large cost), functional verification turns to a critical phase where any behavioral or functional bug escaping this phase will not be detected in the subsequent implementation phases and will appear only after the first silicon is integrated into the target system, resulting in costly design and silicon iterations [4].

Host Bus Adapters are one of the important components in SoCs, so having a strong environment to verify and validate such complex component is valuable for the whole SoC quality and while the system complexity is growing, challenges in testing are getting bigger. Host controller verification and validation would never be complete until it is integrated and tested inside a real computer system having an operating system with a software driver running on it. When host controllers are ready for integration within the system, system testing and bring-up is needed where the HBA is connected to the host system running an O.S (Windows, Linux, Android …) with its software's low level Driver. This testing stage is mandatory but difficult and time consuming due to system complexity and bring-up time, also the need to identify a given issue either in the hardware part or the software part and finally fix it, so having a test environment for HBA that mimics real stimulus of the software driver behavior is very valuable and strongly needed, making it easy to test and validate any change of functionality or behavior in the HBA components, which in turn enables short design time, enhances debugging capabilities and decreases the time consumed in reproducing and fixing issues.

This paper is proposing "Generic testing environment for Host controllers using SystemVerilog". The proposed test environment mimics the stimulus generated from the host software driver in real life setup which helps spotting issues and fixing them easily and quickly compared to such complex system. The test environment has layers for generating SCSI commands [5] [6], Protocol specific commands, monitoring and score-boarding. It is a generic testing environment architecture for verifying SCSI based SoCs' HBA, and can be applied to any host controller that is based on SCSI protocol e.g. SAS, UFS [7] and USB Mass Storage. It is organized as follows: In section III, the micro-architecture of the verification environment is explained in details, in section IV, a case study is explained showing the suggested test environment used in testing UFS HBA and then re-using this environment to test SAS HBA, finally, conclusion is discussed in section V.

III. TEST ENVIRONMENT ARCHITECTURE

As shown in Fig. 1, HBA typically has an AXI interface that is connected to PCIe endpoint which is part of HBA card, so that the HBA card can be plugged into the PCIe slot of a server or host machine motherboard which constructs a PCIe link with the root complex in the host system. Testing the HBA along with the PCIe link embedded in it complicates the testing and debugging of issues, so the main idea is to generate stimulus starting from the AXI interface and bypass the PCIe endpoint. The same idea applies to any interconnect protocol made between the host controller and the host machine.

The HBA testbench is a pure SystemVerilog environment which is designed to run either using simulation or emulation (co-emulation environment), it consists of three main components as shown in Fig. 2, the test environment, the host controller (DUT), and the peripheral. The host controller (DUT) is connected from one side to the peripheral through the targeted protocol link (eg. SAS physical link), and from the other side to the proposed test environment through AXI interface. The test environment takes the role of host machine driver but without the presence of the PCIe link.

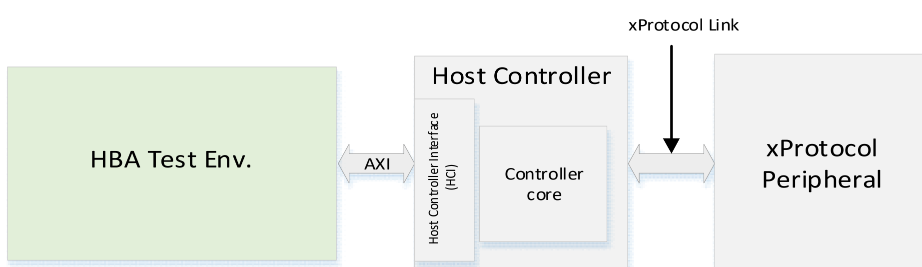

Fig. 2 HBA DUT Integration with Test Env.

The detailed architecture for the proposed generic HBA test environment is shown in Fig. 3.

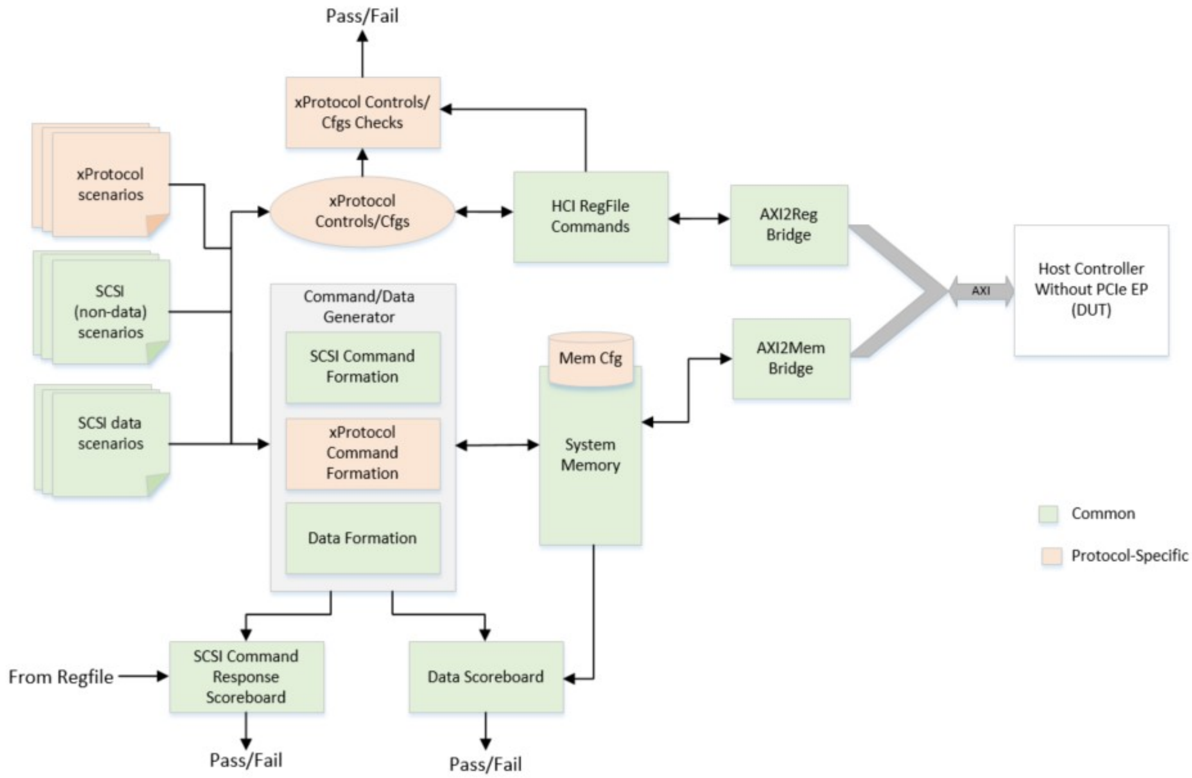

Fig. 3 Host Controller Test Environment Architecture

Test environment connects to DUT (host controller) through AXI interface, which is used by the environment to drive SCSI commands along with the protocol specific commands to the HBA HCI layer. For the sake of simplicity and to reduce AXI overhead throughout the environment two hardware modules (Bridges) have been implemented and connected to AXI interface of the DUT, (1) AXI2Reg which converts from/to AXI signaling interface from/to register interface e.g. register read and write, (2) AXI2MEM which converts from/to AXI signaling interface from/to memory interface, the main purpose of bridges is to detach the test environment from the host controller HCI interface make transaction processing and interface within the environment more simple.

HCI RegFile block is directly connected to the AXI2Reg Bridge which accepts register commands, e.g. read/write to certain register addresses, from upper layer (control/configurations) and coverts them to pin activity on the register signaling interface. It is also responsible for handling the software interrupts coming from host controller side to the driver, where it reads the responses from the host controller based on that interrupt and coverts it back to indication of transaction to the next block/layer in the environment.

A Multi-port memory (2 write and 2 read ports) is directly connected to the AXI2Mem Bridge which acts as the system memory of the host machine. It is used for storing commands from the software driver (SCSI/Protocol Specific) to the host controller and Responses from host controller, and Data for “read/write” operations. This memory is divided into three address space as shown in Fig. 4, (1) command/response space which is defined by static configuration “CMD_BASE_ADDR” and it is used for storing commands/response and scatter gather lists/tables, (2) data space which is defined by static configuration “DATA_BASE_ADDR” and it is used for storing read or write data, (3) user space which is defined by static configuration “USER_BASE_ADDR” and this space can be used by the environment for storing any protocol specific data or commands.

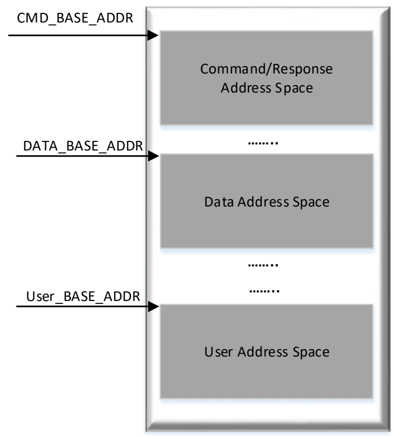

Fig. 4 System Memory Address Space

Command/Data Generator's main function is to compose either the commands or the data as an array of bytes and load them into their corresponding segments in the system memory so that they can be read through AXI2Mem by the host controller. As shown in Fig. 5, it consists of three blocks: (1) SCSI command formation which composes standard SCSI commands, e.g. "Inquiry", "Test Unit Ready", (2) protocol specific commands formation which composes specific commands (if exist) to the protocol used by the host controller e.g. "UFS Queries" or "SMP commands in SAS protocol", (3) Data formation which generates and prepares data in the system memory. The data is written or read in scatter gather manner as shown in Fig. 5. As an example for write commands, the Data formation block takes the total transfer length in bytes as an input and divides (randomly - if required) the whole data into portions, each with certain data length, then each data portion is written into certain location in the system memory. Data portions can be written consecutive or with free space "data spacing" in between i.e. adding fragmentation which mimic the real behavior at operating system memory. Each data portion has its own system address and length information that is written in the system memory i.e. scatter gather list (SGL) which is loaded along with its command. The "SCSI Command" and "Data Formation" blocks are generic/common blocks that can be re-used with different types of SCSI-based host controllers.

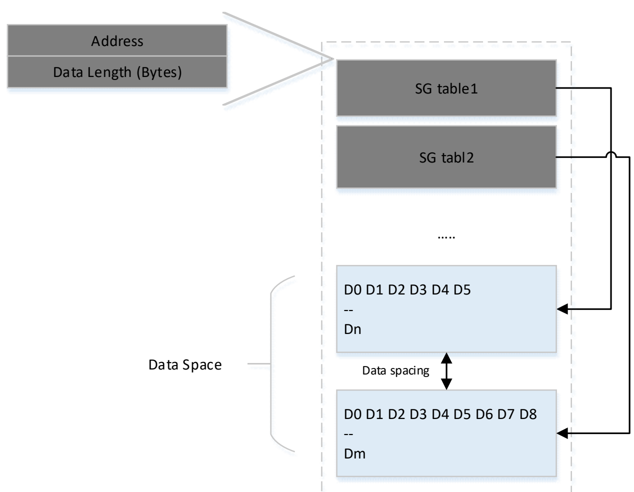

Fig. 5 Data generation in SGL manner

Each protocol has its own control and configuration requests e.g. link initialization, power modes, link speeds and attributes read/write. HCI registers for this part differs from protocol to another, hence "Control/Cfg" block is a protocol specific block. It holds the responsibility to handle these requests and translates the high level transactions coming from test scenario to its corresponding protocol specific operation.

The test environment has three scoreboards that do the necessary checks on the DUT behavior, (1) "Data Scoreboard" which utilities a peripheral memory model to track the Data written into the DUT different logical blocks.

```
typedef struct
{
  scsi_cmd_t       cmd;
  int unsigned     transfer_length;
  int unsigned     lba;
  logic [31:0]     data[];
} cmd_struct_t;
```

The "cmd_struct_t" struct is used to represent read/write transfer. It consists of command (cmd), data transfer length for this command (transfer_length), associated logical block addresses (lba) and data array organized in 32-bit (dword) manner.

```
typedef bit [31:0] blk_data_t[1023:0];

// one block of memory
class memory_block;
  int        blk_address;
  blk_data_t blk_data;

  function new (int _blk_address);
    this.blk_address = _blk_address;
  endfunction

  …

  function string convert2string();
    …
  endfunction

  function void compare(memory_block _that);
    if (this.blk_address != _that.blk_address)
      `uvm_error("memory_block", "Mismatch in Block address")

    foreach(this.blk_data[i]) begin
      if (this.blk_data[i] != _that.blk_data[i]) begin
        `uvm_error("memory_block", $sformatf("Data Mismatch. block address : %0h\n\t Exp: %8h,
                     Act: %8h", this.blk_address, this.blk_data[i], _that.blk_data[i]))
         $finish;
      end
    end
  endfunction: compare
endclass
```

The peripheral memory model is built up of a collection of memory blocks. A single memory block is implemented in the form of a class "memory_block". The class has two fields, "blk_address" and "blk_data" where each address points to certain data content, it constructs a block object using its block address, there are other member functions to aid processing block data content, e.g. a function to compare two blocks together in terms of their addresses and data content.

```
class device_mem;
  memory_block dev_blks [int];

   …

  function void set_blk(int _blk_address, blk_data_t _blk_data);
    if (!dev_blks.exists(_blk_address))
      dev_blks[_blk_address] = new(_blk_address);

    dev_blks[_blk_address].set_data(_blk_data);
  endfunction

  function blk_data_t get_blk(int _blk_address);
    blk_data_t blk_data;
    if (!dev_blks.exists(_blk_address))
      `uvm_error("device_mem", "Trying to read block that wasn't written before");

    blk_data = dev_blks[_blk_address].get_data();
  endfunction

  function void compare(device_mem _that);
    foreach(that.dev_blks[i])
      this.dev_blks[i].compare(_that.dev_blks[i]);
  endfunction: compare
  …
endclass;
```

The peripheral memory model itself is a class called "device_mem" that has an array of blocks "memory_block" objects. "set_blk" method is used to fill a block by giving it the block address and the data to fill in, it will check whether an object was made having the same address or not then fill the block. To get the content of a memory block use the method "get_blk", it will check for the block presence if not it will issue an error.

Using this approach allows the test scenario to have "read/write" with any possible configuration (e.g. logical block address, transfer length) and allows checking Data Written/Read back at any time during the test in an easy manner. (2) "Control/cfg Scoreboard" which checks for any protocol specific behaviors e.g. outputs coming from register file, and (3) "Response Scoreboard" which does the needed checking for the SCSI command response. The "Data Scoreboard" and "Response Scoreboard" are generic blocks that can be re-used with different types of SCSI-based host controllers.

A very important aspect of the testbench core is the automatic selection of a tag/slot and tracking of used tags/slots. The testcase should only ask the testbench core for an empty tag/slot to use and upon actual submission the Testbench locks this tag/slot. Doing this ensures no overlapped commands will overflow within the test scenario unless intended by the testcase itself. A tip that eases debugging and usually used by the Linux Driver itself is to use the slot number as the command tag. Doing so eases the debugging of failing scenarios.

```
task get_empty_slot(output int _slot_no,
                    input bit _seq_slot = 1
                   );
  ...

  wait (&doorbell_reg_local == 0);
  for (int i=0; i<MAX_SLOT_NUM ; i++) begin
    if (doorbell_reg_local[i] == 0) begin
      // 1: return first available slot
      if (_seq_slot) begin
        doorbell_reg_local[i] = 1;
        _slot_no = i;
        return;
      end
      else if (!_seq_slot) begin
        // randomize from available slots
        …
      end
    end
  end // for loop
  …
endtask
```

The method “get_empty_slot” is used to lock a free tag and return it back the testcase to use in its SCSI requests. It blocks till an empty slot is free then return it back to the caller.

Test scenarios is high level SystemVerilog transaction-based classes which are implemented to execute real sequences to test the host controllers. They are divided into three categories: (1) SCSI non-data command scenarios are used to request and test non-data SCSI commands e.g. “Inquiry”, “Test Unit Ready” (2) SCSI data command scenarios are used to request “read” and “write” commands for data transfer testing, (3) Protocol specific scenarios are used to request commands related to the protocol itself e.g. queries, power modes and link configurations.

“Environment Configuration“ block holds the run-time parameter configurations that affect the host and/or the peripheral e.g. QUEUE Depth (TCQ), Out of order Data enable/disable, Scatter gather enable/disable, when disabled, data associated with “read/write” commands will be allocated in a contiguous memory area.

IV. CASE STUDY

The proposed test environment is built to test Universal Flash Storage (UFS) host controller. UFS is a simple, high performance, serial interface. It is primarily for use in mobile systems, between host processing and NVM mass storage devices, UFS Controller architecture including the interface with the application layer (host machine i.e. OS/Driver) is shown in Fig. 6, UFS Host controller Interface (HCI) [8] defines the register interface between the host controller and the host software OS/Driver.

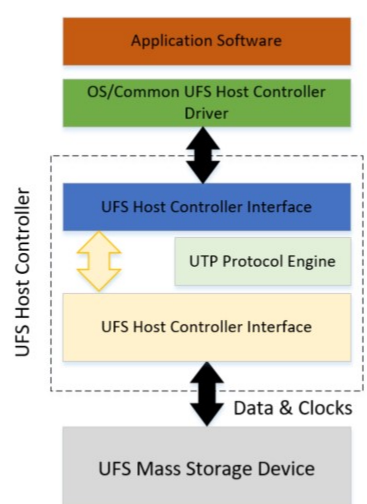

Fig. 6 UFS Host Controller Architecture

The software application sends commands such as UPIU packets to UFS host controller which is processed by the UTP layer. These commands may belong to the UFS SCSI command set or the UFS Native command set, however the latest UFS standard (UFS 3.0) does not define UFS native commands and they may be defined in the future to support specific flash storage or UFS native basic needs, hence UFS host controller only accepts the UFS SCSI command set (USC) which consists of a selection of commands from SCSI Primary Commands (SPC) [5], and SCSI Block Commands (SBC) [6]. Both command types share similar command descriptor block (CDB) format which defines the type of the SCSI command whether Write6, Read6 … or Inquiry [5]. Fig. 7, shows the structure of command UPIU and the location of CDB within the packet. UTP layer accepts other types of command rather than normal SCSI commands which are task management commands and UFS queries. For the sake of simplicity these commands will be excluded from the case study.

Command UPIU			
UPIU opcode	Flags	LUN	Task Tag
Set Type	Reserved		
Reserved		Data Segment Length	
Expected Data Transfer Length			
CDB Area (4 DWs)			

Fig. 7 SCSI Command in UFS UPIU

Fig. 8, shows the system memory architecture used for testing UFS host controller, it is built and configured to comply with the interface specifications between the host software and the UFS controller which is described in UFS HCI standard specification document [8] where the memory space is divided into two main parts: transfer request descriptors and task management requests. The transfer request descriptor is a data structure that describes a command to be executed and the data associated with it, the host software can issue up to 32 commands to the host controller, each descriptor points to the location of memory that contains the command UPIU, response UPIU and SG tables which holds the data details that are stored in a scatter gather list. Task management requests area is used to store any task management command to UFS controller. The host software can issue up to 8 commands which are located in the system memory as well.

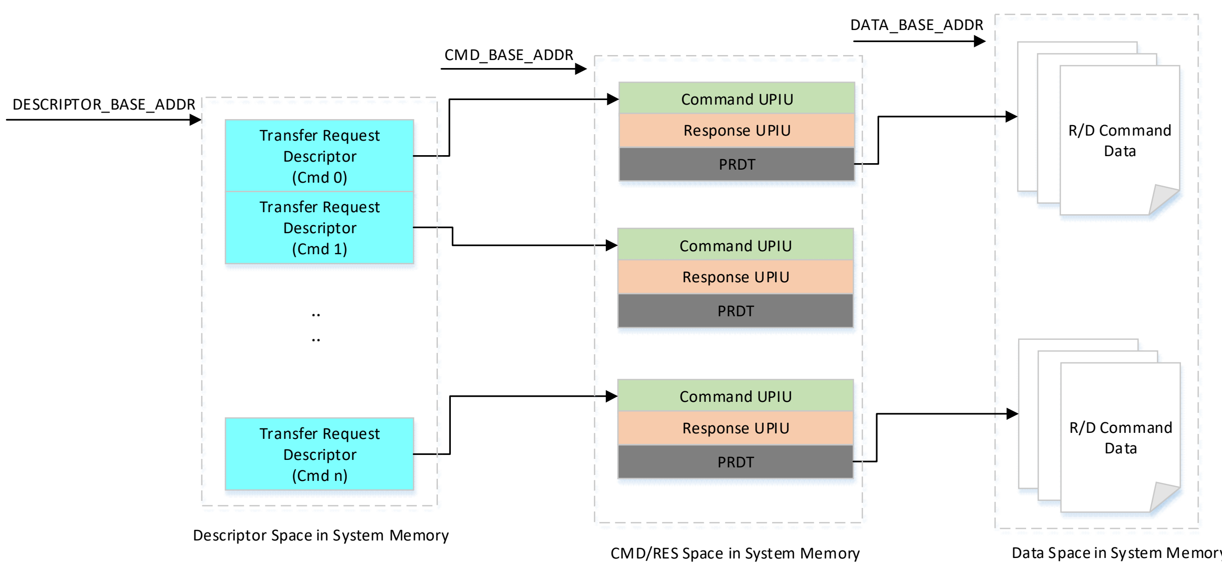

Fig. 8 UFS System Memory Space

Initially and during test configuration, three segments are reserved in the system memory to satisfy the above shown structure: “DESCRIPTOR_BASE_ADDR” address is reserved for UFS command descriptor, the length of any command list descriptor is 8 dwords. The transfer descriptor is not something common and defined only for UFS protocol so this memory region is mapped to the user address space as described in Fig. 4. “CMD_BASE_ADDR” is the area used to write command and receive the response, also it is used for writing SG tables which are used to address scatter gather data and “DATA_BASE_ADDR” is used for addressing any read/write command data in the system memory. The test mechanism uses these addresses to write commands/data for the host controller and to read responses/data retrieved from the peripheral/target. The test mechanism checks the commands status and compares written/read data with the expected values.

The same test environment was reused to test a SAS host controller (SAS Initiator). SAS is a serial protocol where data can be transferred between computer and storage devices such hard drives and tape drives [9], It is a multi-layer protocol, the standard defines five layers that integrate together to perform the desired functionality. Fig. 9, shows the SAS protocol layers.

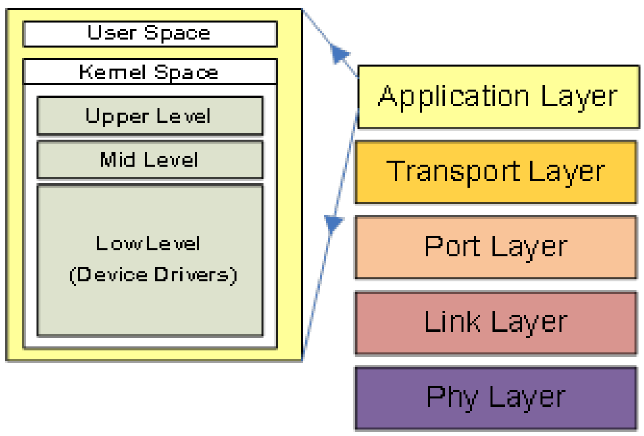

Fig. 9 SAS Protocol layers

The SAS protocol supports three internal protocols: SSP, STP and SMP. SSP is the serial SCSI protocol that is used to transfer the SCSI commands with data between computer and hard disks, STP is the Serial ATA tunneled protocol and is used when SATA devices are connected to SAS initiator through an expanders, SMP is the serial management protocol and is mainly in the complex topology when expanders are present. For simplicity, the main focus in the testing of SAS initiator is for SSP and SCSI commands.

SAS commands which is transferred from host driver to the controller and the register file interface was simpler than UFS where there was no descriptor defined for each command, and commands can be stored directly into the memory along with its SG list, i.e. there was no user space defined in the system memory. Command and response structure for SAS was different than UFS, however the CDB i.e. SCSI command generation is re-used from UFS environment, data generation is re-used as it is from UFS environment as well. Test scenarios which includes only SCSI commands that were developed for UFS host controller has been re-used within SAS environment.

As a result of that the SAS environment was ready in a short time because the basic blocks and architecture was ready and got re-used from UFS environment. It is worth mentioning that this environment has been used by the developers in the early stage of host controller development which eased the software and hardware integration and accelerated the bring-up of the whole design.

V. Conclusion

Our objective was to create SCSI based host controller test environment that can be easily used to test different SCSI based host controllers, the proposed test environment was implemented using SystemVerilog and was used to test UFS host controller and SAS host controller (SAS Initiator). It proved great value creating driver-like test scenarios, basic and complex ones and helped in spotting issues and fixing them easily compared to real life setup.

References

[1] N. Bamford et al., "Challenges in System on Chip Verification," Seventh International Workshop on Microprocessor Test and Verification (MTV'06), Austin, TX, 2006, pp. 52-60, doi: 10.1109/MTV.2006.5.

[2] M. Rouse, Host Bus Adapter (HBA), SearchStorage, Apr., 2015. Accessed on: June 3, 2019 [Online]. Available: https://searchstorage.techtarget.com/definition/host-bus-adapter

[3] Computer Hope, Host Adapter, Computer Hope, Apr. 26, 2017. Accessed on: June 3, 2019 [Online]. Available: https://www.computerhope.com/jargon/h/host_adapter.htm

[4] S. K. Roy and S. Ramesh, "Functional verification of system on chips - practices, issues and challenges," Proceedings of ASP-DAC/VLSI Design 2002. 7th Asia and South Pacific Design Automation Conference and 15h International Conference on VLSI Design, Bangalore, India, 2002, pp. 11-13.

[5] INCITS T10 draft standard: SCSI Primary Commands – 4 (SPC-4), Revision 27, 11 October 2010.

[6] INCITS T10 draft standard: SCSI Block Commands – 3 (SBC–3), Revision 24, 05 August 2010

[7] UFS Standard (V.1.0), JESD220-1, Sep 2013

[8] UNIVERSAL FLASH STORAGE HOST CONTROLLER INTERFACE (UFSHCI) Version 3.0, JESD223D, 2018

[9] T10/BSR INCITS 538 – SAS Protocol Layer – 4 (SPL-4)