Verification of USB 3.0 Device IP using Universal Verification Methodology

Krunal Kapadiya
(Department of Electronics & Communication Engineering, Gujarat Technological University, Ahmedabad – 382424, India,)

ABSTRACT

Verification of integrated USB 2.0 – USB 3.0 Device IP by developing device reference module connected to XDC side and ready to use USB 3.0 Host Verification IP connected to physical layer. Device reference module has been supporting AHB-XDC in addition to Generic-XDC path.

Keywords – AHB (Advanced High-Performance Bus), AXI (Advanced Extensible Interface), IP (Intellectual Property), SV (System-Verilog), USB (Universal Serial Bus), UVM (Universal Verification Methodology), XDC (Extensible Device Controller).

I. INTRODUCTION

Universal Serial Bus (USB) is an industry standard developed in the mid-1990s that defines the cables, connectors and communications protocols used in a bus for connection, communication and power supply between computers and electronic devices. USB 3.0 utilized a dual-bus architecture that provides backward compatibility with USB 2.0. It provides for simultaneous operation of superspeed and non-superspeed (USB 2.0 speeds) information exchanges.

Verification of integrated USB 2.0 - USB 3.0 Device IP by developing device reference module supporting Generic-XDC and AHB-XDC paths connect to XDC side of USB 3.0 Device IP and ready to use USB 3.0 Host Verification IP at physical layer of USB 3.0 Device IP. Verification architecture includes uvm_test, uvm_env, uvm_agent, uvm_driver, uvm_scoreboard, etc. components of universal verification methodology with SystemVerilog.

II. OVERVIEW OF USB

The design architecture of USB is asymmetrical in its topology, consisting of a host, a multitude of downstream USB ports, and multiple peripheral devices connected in a tiered-star topology. Additional USB hubs may be included in the tiers, allowing branching into a tree structure with up to five tier levels. A USB host may implement multiple host controllers and each host controller may provide one or more USB ports. If hub device present, up to 127 devices may be connected to a single host controller [1]. USB devices are linked in series through hubs. One hub is known as the root hub which is built into the host controller.

A physical USB device may consist of several logical sub-devices that are referred to as *device functions*. A single device may provide several functions, for example, a webcam (video device function) with a built-in microphone (audio device function). This kind of device is called *composite device*. An alternative for this is *compound device* in which each logical device is assigned a distinctive address by the host and all logical devices are connected to a built-in hub to which the physical USB wire is connected.

USB device communication is based on *pipes* (logical channels). A pipe is a connection from the host controller to a logical entity, found on a device, and named an *endpoint*. Because pipes correspond 1-to-1 to endpoints, the terms are sometimes used interchangeably. A USB device can have up to 32 endpoints, though USB devices seldom have this many endpoints. An endpoint is built into the USB device by the manufacturer and therefore exists permanently, while a pipe may be opened and closed.

There are two types of pipes: stream and message pipes. A message pipe is bi-directional and is used for *control* transfers. Message pipes are typically used for short, simple commands to the device, and a status response, used, for example, by the bus control pipe number 0. A stream pipe is a uni-directional pipe connected to a uni-directional endpoint that transfers data using an *isochronous*, *interrupt*, or *bulk* transfer:

- *Isochronous transfers*: at some guaranteed data rate (often, but not necessarily, as fast as possible) but with possible data loss (e.g., real-time audio or video).
- *Interrupt transfers*: devices that need guaranteed quick responses (bounded latency) (e.g., pointing devices and keyboards).
- *Bulk transfers*: large sporadic transfers using all remaining available bandwidth, but with no guarantees on bandwidth or latency (e.g., file transfers).

An endpoint of a pipe is addressable with tuple *(device_address, endpoint_number)* as specified in a

TOKEN packet that the host sends when it wants to start a data transfer session. If the direction of the data transfer is from the host to the endpoint, an OUT packet (a specialization of a TOKEN packet) having the desired device address and endpoint number is sent by the host. If the direction of the data transfer is from the device to the host, the host sends an IN packet instead. If the destination endpoint is a uni-directional endpoint whose manufacturer's designated direction does not match the TOKEN packet (e.g., the manufacturer's designated direction is IN while the TOKEN packet is an OUT packet), the TOKEN packet will be ignored. Otherwise, it will be accepted and the data transaction can start. A bi-directional endpoint, on the other hand, accepts both IN and OUT packets.

Endpoints are grouped into *interfaces* and each interface is associated with a single device function. An exception to this is endpoint zero, which is used for device configuration and which is not associated with any interface. A single device function composed of independently controlled interfaces is called a *composite device*. A composite device only has a single device address because the host only assigns a device address to a function.

When a USB device is first connected to a USB host, the USB device enumeration process is started. The enumeration starts by sending a reset signal to the USB device. The data rate of the USB device is determined during the reset signaling. After reset, the USB device's information is read by the host and the device is assigned a unique 7-bit address. If the device is supported by the host, the device drivers needed for communicating with the device are loaded and the device is set to a configured state. If the USB host is restarted, the enumeration process is repeated for all connected devices.

The host controller directs traffic flow to devices, so no USB device can transfer any data on the bus without an explicit request from the host controller. In USB 2.0, the host controller polls the bus for traffic, usually in a round-robin fashion. The throughput of each USB port is determined by the slower speed of either the USB port or the USB device connected to the port.

High-speed USB 2.0 hubs contain devices called transaction translators that convert between high-speed USB 2.0 buses and full and low speed buses [2]. When a high-speed USB 2.0 hub is plugged into a high-speed USB host or hub, it will operate in high-speed mode. The USB hub will then either use one transaction translator per hub to create a full/low-speed bus that is routed to all full and low speed devices on the hub, or will use one transaction translator per port to create an isolated full/low-speed bus per port on the hub.

Because there are two separate controllers in each USB 3.0 host, USB 3.0 devices will transmit and receive at USB 3.0 data rates regardless of USB 2.0 or earlier devices connected to that host. Operating data rates for them will be set in the legacy manner.

III. Related Work

I proposed my solution on Universal Serial Bus. Currently USB 3.0 Device IP is not capable for certain functionality which needed at XDC side. i.e. above protocol layer. There are different ways to verify the USB 3.0 Device IP at XDC side.

The USB 3.0 Device IP connected to USB 3.0 Host VIP through PIPE (USB 3.0) [3] / ULPI (USB 2.0) [4] interface. The USB 3.0 Device IP core connected to application through back-end interfaces. Back-end interfaces are CFG, Master TX, Slave TX, Master RX and Slave RX.

Data transfer between USB 3.0 device core and application occurs through following paths:

- Generic-XDC
- AHB-XDC
- AXI-XDC

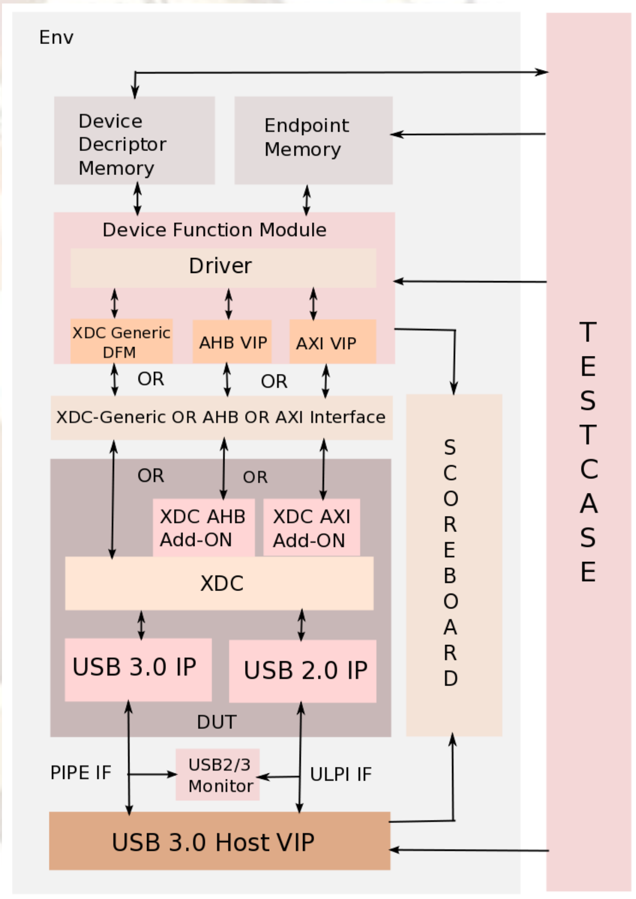

Fig. 1. USB 3.0 Device IP Verification

The verification environment should support any one type of path (Generic-XDC/ AHB-XDC or AXI-XDC) via define switch on XDC side and should support USB 3.0 PIPE and USB 2.0 ULPI interface on USB 3.0 physical layer. In current architecture, Generic-XDC and AHB-XDC paths are supported.

AXI-XDC path would be supported in future version.
USB 3.0 Verification IP can be configured to work as a Host model to verify Device DUT. And the USB 3.0 Device core connected to application through Generic-XDC/ AHB-XDC/ AXI-XDC path.
The USB 3.0 Host VIP generates the data/ transfer traffic to USB 3.0 Device DUT. Scoreboard is a part of usb3dev_env, ensures the data integrity between the USB Host VIP environment and USB Control environment through USB 3.0 Device DUT for both transmit and receive paths.
USB 3.0 VIP configured as active host and passive device and XDC environment instantiated in the usb3dev_env, which instantiated in uvm_test. In top module, USB 3.0 Device IP connected as DUT and run_test() method would be called [3].

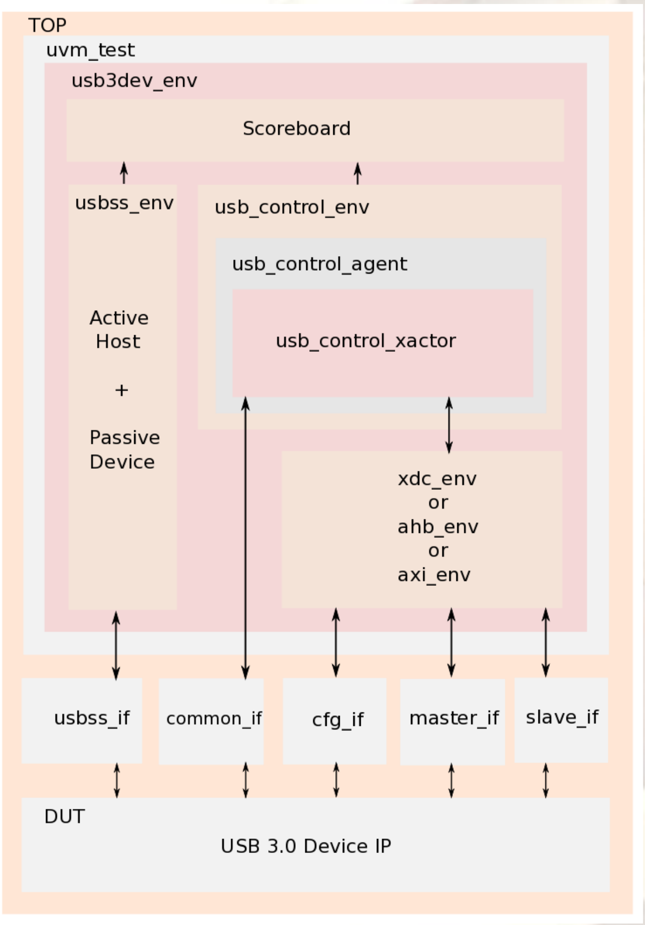

Fig. 2. Verification Architecture using UVM

The USB 3.0 VIP connected to USB 3.0 Device DUT through USB 3.0 PIPE/ USB 2.0 ULPI interface and USB 3.0 Device DUT connected to Generic-XDC environment or AHB-XDC environment or AXI-XDC environment through Generic-XDC or AHB-XDC interface or AXI-XDC environment respectively. The memory manager would be instantiated and used by USB Control environment.
In top module, following interfaces are instantiated:

- USB SS interfaces: Active Host and Passive Device
- Common interface
- Generic-XDC interfaces: XDC-CFG, XDC-Master, XDC-Slave
- AHB-XDC interfaces: AHB-CFG, AHB-Master-TX, AHB-Master-RX, AHB-Slave-TX and AHB-Slave-RX

In top module, the AHB Add-ON and AXI Add-ON module would connect XDC interface to AHB interface and AXI interface respectively. The top module is having the functionality to multiplex between the Generic-XDC, AHB-XDC and AXI-XDC paths. The XDC interface directly connected from USB 3.0 Device DUT to XDC environment. The AMBA-AHB interface connected from USB 3.0 Device DUT to AHB environment through AHB-XDC Add-ON. The AMBA-AXI interface connected from Device DUT to AXI environment through AXI-XDC Add-ON.
XDC Environment works as follows:

- Interrupt request (IRQ) signal sent from USB 3.0 Device DUT.
- Device initialization configuration registers would be written.
- Memory would be allocated by memory manager through backdoor access in case of bulk-IN transfer.
- Endpoint address mapping would be done through memory manager.
- Host would send the burst length for bulk-OUT transfer to device.
- Bulk-OUT from host would transfer to mapped endpoint of the device.

After completing transfers, the transfers on PIPE/ULPI or XDC interface would be sent to scoreboard to compare with expected transfers.
After the allocation of memory as per endpoint's address map, burst-length would be provided to transfer the data. For big data, the transfer would be done in segments for which memory pre-allocation would be done. e.g. 4 GB data transfer could be done in segments of 112KB data.
The XDC environment is the verification component, which interfaces from an XDC of USB 3.0 Device IP to the application. The XDC environment contains the functionality to read data from and write data to registers located at Device DUT through Generic-XDC path. When the USB 3.0 VIP connected as host, first of all it initiates the default enumeration process. The process of identifying and configuring a USB device is referred to as USB device enumeration. At the end of enumeration process, the device is ready to do any transfer with the device as per the test-cases written by user.
The usb3dev_env environment is having the instance of XDC environment, USB Control environment and USB Scoreboard.

The CFG, Master-TX, Master-RX, Slave-TX, Slave-RX and Common interfaces passed through assign_vi() method of XDC environment and USB Control environment in assign_vi() method of usb3dev_env.

The mailboxes regarding request and response packets of CFG, Slave-TX and Slave-RX inside USB Control Xactor and CFG driver, Slave-TX driver and Slave-RX driver connected in usb3dev_env environment.

3.1. Generic-XDC Environment

The XDC environment contains the functionality to read/write on XDC-CFG, XDC-Master and XDC-Slave interfaces.

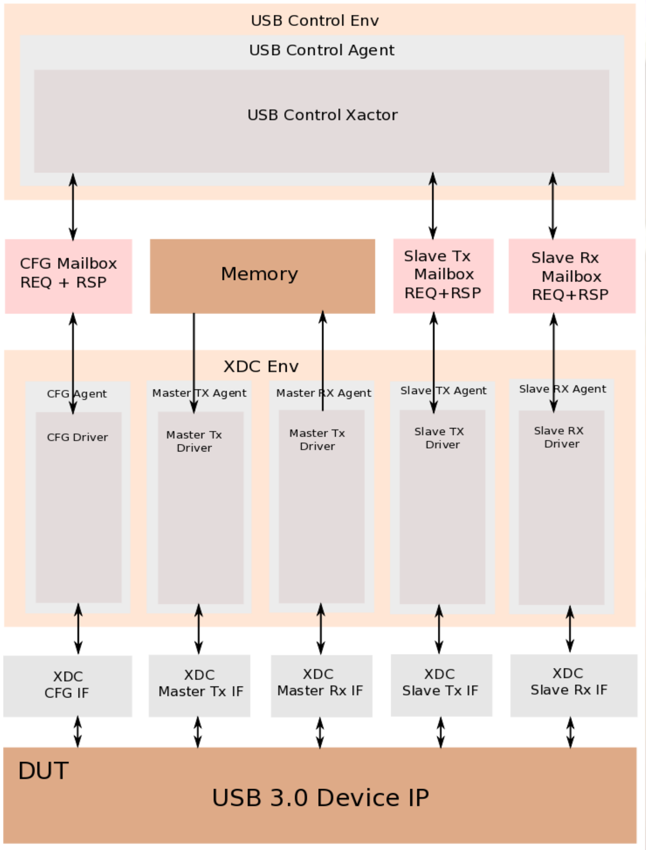

Fig. 3. Generic-XDC Environment

The XDC-agent and XDC-driver regarding XDC-CFG, XDC-Master-TX, XDC-Master-RX, XDC-Slave-TX and XDC-Slave-RX instantiated. The Generic-XDC interfaces passed hierarchically through assign_vi() method from uvm_test to usb3dev_env to environment to respective agent to respective driver.

The XDC-CFG driver gets the XDC-CFG packet from request mailbox and drive accordingly on XDC-CFG interface. In case of read, it also puts the XDC-CFG packet on response mailbox after reading data.

The XDC-Master-TX driver drives the signal on XDC-Master-TX interface with data read from memory in top module.

The XDC-Master-RX driver samples the signal on XDC-Master-RX interface and writes the data to memory in top module.

The XDC-Slave-TX driver gets the XDC-Slave-TX packet from request mailbox and drive accordingly on XDC-Slave-TX interface.

The XDC-Slave-RX driver gets the XDC-Slave-RX packet from request mailbox and drive accordingly on XDC-Slave-RX interface. It put the XDC-Slave-RX packet on response mailbox after reading data.

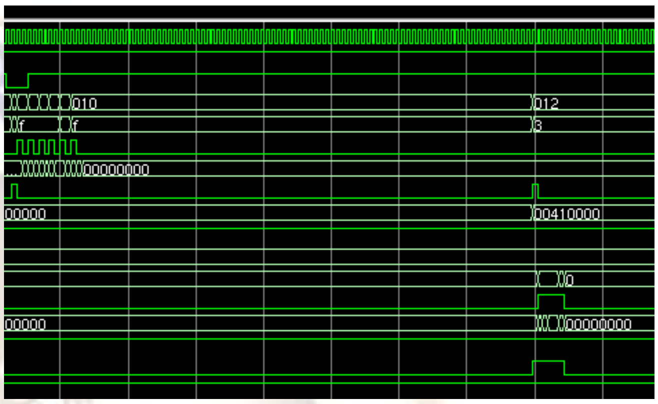

Fig. 4. Waveform of read/write on XDC-CFG interface and command on XDC-Slave-TX interface

3.2. AHB-XDC Environment

The AHB-XDC environment contains the functionality to read/write on AHB-CFG, AHB-Master-TX, AHB-Master-RX, AHB-Slave-TX and AHB-Slave-RX interfaces.

The AHB-Agent and AHB-driver regarding AHB-CFG, AHB-Master-TX, AHB-Master-RX, AHB-Slave-TX and AHB-Slave-RX instantiated. The respective AHB-XDC interfaces passed using uvm_config_db #(virtual ahb_if)::set() method to respective AHB-XDC driver.

The AHB-Master-BFM instantiated in driver regarding AHB-CFG, AHB-Slave-TX and AHB-Slave-RX. The AHB-Slave-BFM instantiated in driver regarding AHB-Master-TX and AHB-Master-Rx.

The AHB-CFG driver gets the XDC-CFG packet from request mailbox from USB Control Xactor. It calls AHB read/write method as per read/write packet, which will convert XDC-CFG packet to AHB-CFG packet. Those AHB read/write method puts converted AHB-CFG packet to mailbox of AHB-CFG BFM.

The AHB-Master-TX driver gets the address and size of data from request mailbox in AHB-Master-TX BFM. It reads the data from the memory as per address and puts data on request mailbox in AHB-Master-TX BFM, which drives the data on signals on AHB-Master-TX interface.

The AHB-Master-RX driver gets the address, data and size of data from request mailbox on AHB-Master-RX BFM, which samples the data on signals on AHB-Master-RX interface. It writes the data to the memory as per address.

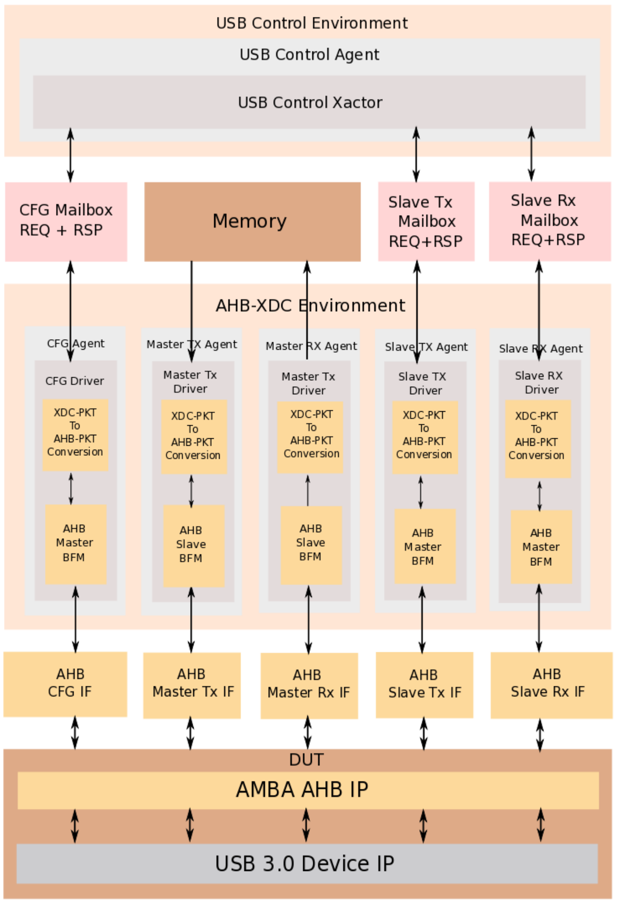

Fig. 5. AHB-XDC Environment

The AHB-Slave-TX driver gets the XDC-Slave-TX packet from request mailbox from USB Control Xactor. It calls AHB write method, which convert XDC-Slave-TX packet to AHB-Slave-TX packet. That AHB write method puts converted AHB-Slave-TX packet to mailbox of AHB-Slave-TX BFM.
The AHB-Slave-RX driver gets the XDC-Slave-RX packet from request mailbox from USB Control Xactor. It calls AHB read method, which convert XDC-Slave-RX packet to AHB-Slave-RX packet. That AHB read method puts converted AHB-Slave-RX packet to mailbox of AHB-Slave-RX BFM.

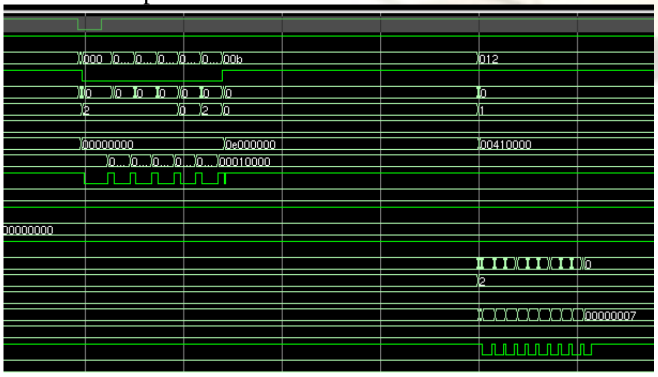

Fig. 6. Waveform of read/write on AHB-CFG interface and command on AHB-Slave-TX interface

3.3. USB Control Environment

The agent and driver regarding USB device control instantiated. The mailboxes and packets regarding XDC-CFG, XDC-Slave-TX and XDC-Slave-RX also instantiated.
The common interface passed hierarchically through assign_vi() method from uvm_test to usb3dev_env to USB control environment to USB control agent to USB control driver.
The device in/out analysis port regarding USB scoreboard packet connected to analysis implementation port of USB scoreboard. The instantiated memory manager used to allocate memory as per starting address and memory size.
The USB control Xactor read or initialize all common registers. Then wait for IRQ signal to be generated from DUT. If IRQ received, read registers and check special type and store it on queue. Then process further if no other task is pending.
The methods to send packet to read/write on XDC-CFG, XDC-Slave-TX and XDC-Slave-RX interfaces are putting or getting on respective mailboxes.
The interrupt handler takes care of IRQ handling and DMA request to master and event request to slave.

3.4. USB Scoreboard

The USB scoreboard used to compare the transfers made by USB control environment and USB Host Verification environment.
As analysis implementation ports for Host/Device In/Out instantiated, write() method for each analysis implementation port gets implemented.
While bulk-out transfer, USB protocol-data packet received from USB host verification environment written into write() method implemented for host-out, where extracted scoreboard packet packed to host-out queue [5]. And scoreboard packet received from USB control xactor written into device-in queue of write() method implemented for device-in.
While bulk-in transfer, USB protocol-data packet received at host verification environment written into write() method implemented for host-in, where extracted scoreboard packet packed to host-in queue. And scoreboard packet received from USB control xactor written into device-out queue of write() method implemented for device-out.
After getting all scoreboard packets in the form queue, run_phase() method [6] having scoreboard IN/OUT methods. Scoreboard IN method compares the data of scoreboard packet for each sequence number from host-in queue and device-out queue. Scoreboard OUT method compares the data of scoreboard packet for each sequence number from host-out queue and device-in queue.
In report_phase() method, errors will be fired and test-case fails in case of mismatch of payload data and/or payload size. Otherwise test-case passes in case of correct match of payload data and payload size.

3.5. Test-case Simulation Flow

- During build_phase() method, setting number of active/passive host/device, endpoints, interfaces, configuration, etc. Setting SCSI command as default_sequence of protocol sequencer of host_agent. Updating EP IN/OUT table.
- During connect_phase() method, passing the interface declared in top to usb3dev_env using assign_vi() method or uvm_config_db #()::set() method.
- During end_of_elaboration_phase() method, configuring interface and configuration descriptor, setting BULK/INT/ISO IN/OUT endpoint descriptors.
- During run_phase() method, requesting to 1) set device address 2) get device descriptor 3) get configuration descriptor 4) set configuration no. 1. Executing BULK/INT/ISO IN/OUT transfers.
- Set SCSI write command with packet size to SCSI sequence.
- Allocates endpoint OUT memory with 32 bytes, in which it allocates the memory using memory allocation method memory manager of control Xactor, which returns the memory start address and total memory size.
- Bulk-OUT transfer with 32 bytes, in which it set transfer and executes bulk transfer from host’s protocol BFM.
- Wait for transfer to complete from XDC, in which it inserts delay till memory size reduces to zero.
- Send scoreboard packet to scoreboard to compare, in which it call method of control Xactor, which forms the scoreboard packet and write in TLM analysis port connected to scoreboard.
- If command with proper packet size has been transferred, allocates endpoint OUT memory with packet size, in which it allocates the memory using memory allocation method memory manager of control Xactor, which returns the memory start address and total memory size.
- Bulk-OUT transfer with packet size, in which it set transfer and executes bulk transfer from host’s protocol BFM.
- Wait for transfer to complete from XDC, in which it inserts delay till memory size reduces to zero.
- Send scoreboard packet to scoreboard to compare, in which it call method of control Xactor, which forms the scoreboard packet and write in TLM analysis port connected to scoreboard.

IV. CONCLUSION

By verifying the USB 3.0 Device IP, one gets in-depth protocol knowledge of USB 3.0 specification. As DUT with integrated AHB-XDC add-on, one also gets the in-depth protocol knowledge of AMBA AHB specification. Verification using randomization technique supported by SystemVerilog and Universal Verification Methodology helps the USB 3.0 Device IP becomes more robust and stable as per USB 3.0 specification. By developing environment from scratch to verify the USB 3.0 Device IP, one gets the exposure to SystemVerilog and Universal Verification Methodology, which is de-facto standard of verification language and methodology respectively.

As Generic-XDC provides byte-enable combinations including single byte to eight bytes and AHB-XDC provides byte-enable combinations including single byte, half-word, word and double-word [7], AHB-XDC simulation has overhead over Generic-XDC simulations.

V. ACKNOWLEDGMENTS

I am grateful to Mr. Umesh Patel (Director – ASIC) and Mr. Manish Desai (Sr. Member Technical Staff) of Sibridge Technologies, Ahmedabad for providing guidelines and encouragement. I feel motivated and encouraged every time I attend their meeting.

REFERENCES

[1] Universal Serial Bus 3.0 Specification, June 2011.
[2] Universal Serial Bus Specification Revision 2.0, April 2000.
[3] PHY Interface for the PCI Express and USB 3.0 Architectures (PIPE), April 2009.
[4] UTMI Low Pin Interface (ULPI), October 2009.
[5] IEEE 1800-2012 Standard for SystemVerilog – LRM.
[6] Universal Verification Methodology (UVM) 1.1 User’s Guide, May 2011.
[7] AMBA Specification 2.0, May 1999.