Steering Through Verification Challenges of USB 3.0-Based SoC Using Cadence VIP

By Sunil Golasangi, Staff ASIC Verification Engineer, SanDisk India Device Design Centre
&
Akhilesh Mahajan, Manager – HW Design, Synapse Design India

Introduction

As the technology innovation marches forward, new kinds of devices, medium (internet of things), media formats and large inexpensive storage are converging. They required significantly higher bandwidth than the tradition technologies and protocols. USB3 addresses these needs by achieving higher data transfer speeds that are up to 10 times faster than the previous version of the standard, enabling rapid and efficient transfers of data to and from external storage and multimedia devices. USB3.0 utilizes a dual-bus architecture that provides backward compatibility with USB2.0.

The transition from High Speed USB to SuperSpeed USB presents a new set of challenges for both design and verification teams. For the design team, there is a little reason to develop highly complex interface IP blocks like USB3.0 because it requires very precise domain expertise, gives little opportunity to differentiate their end-products and increases time to market. Consequently, the value of IP providers is defined around delivering a high-quality product that designers can drop into their design with a high level of trust. Even if 3rd Party IPs solutions are silicon proven, still the whole verification cycle will be required to ensure that the 3rd parties IPs are integrated within the SoC ecosystem properly. Extensive verification is still needed, but the focus is now on integration testing, rather than compliance testing. This is why it's important to pick a tried and trusted IP provider in the first place.

This paper demonstrates the advantages of Cadence® Verification IP (VIP) for USB on enhancing stability and quality of a USB Solution (PHY + MAC) when used in traditional verification flow i.e. Verilog based verification environment. Also, we have demonstrated how Cadence VIP can be used to emulate a BOT (Bulk Only Transfer) host in the simulation. This emulation helps in covering all the features of the device using a single VIP model.

USB 2.0 to 3.0 Verification Challenges

The USB 3.0 is similar to USB 2.0 but with many improvements and an alternative implementation. Basic USB concepts like endpoints and four transfer types (bulk, control, isochronous and interrupt) are preserved but the physical and electrical interface are different. This creates a new set of challenges: the verification platform which was used for previous products needs to be reused for newer product where the top layer remains the same, however the physical/link layer changes drastically (see figure 2). USB 3.0 requires a number of verification steps, each with its own unique

requirements. The newer test cases should focus on additional features/layers provided in USB 3.0 like

- LTSSM
- Parallel (Rx/Tx) & Asynchronous Transfer
- Error Injection
- Link switching USB3.0 to USB2.0 and vice versa
- Compliance etc.

Characteristic	SuperSpeed USB	USB 2.0
Data Rate	SuperSpeed (5.0 Gbps)	low-speed (1.5 Mbps), full-speed (12 Mbps), and high-speed (480 Mbps)
Data Interface	Dual-simplex, four-wire differential signaling separate from USB 2.0 signaling Simultaneous bi-directional data flows	Half-duplex two-wire differential signaling Unidirectional data flow with negotiated directional bus transitions
Cable signal count	Six: Four for SuperSpeed data path Two for non-SuperSpeed data path	Two: Two for low-speed/full-speed/high-speed data path
Bus transaction protocol	Host directed, asynchronous traffic flow Packet traffic is explicitly routed	Host directed, polled traffic flow Packet traffic is broadcast to all devices.
Power management	Multi-level link power management supporting idle, sleep, and suspend states. Link-, Device-, and Function-level power management.	Port-level suspend with two levels of entry/exit latency Device-level power management
Bus power	Same as for USB 2.0 with a 50% increase for unconfigured power and an 80% increase for configured power	Support for low/high bus-powered devices with lower power limits for un-configured and suspended devices
Port State	Port hardware detects connect events and brings the port into operational state ready for SuperSpeed data communication.	Port hardware detects connect events. System software uses port commands to transition the port into an enabled state (i.e., can do USB data communication flows).
Data transfer types	USB 2.0 types with SuperSpeed constraints. Bulk has streams capability (refer to Section 3.2.8)	Four data transfer types: control, bulk, interrupt, and Isochronous

FIGURE 1: *Difference between USB 2.0 and USB 3.0 comparison table*

Source: Universal Serial Bus 3.0 Specification

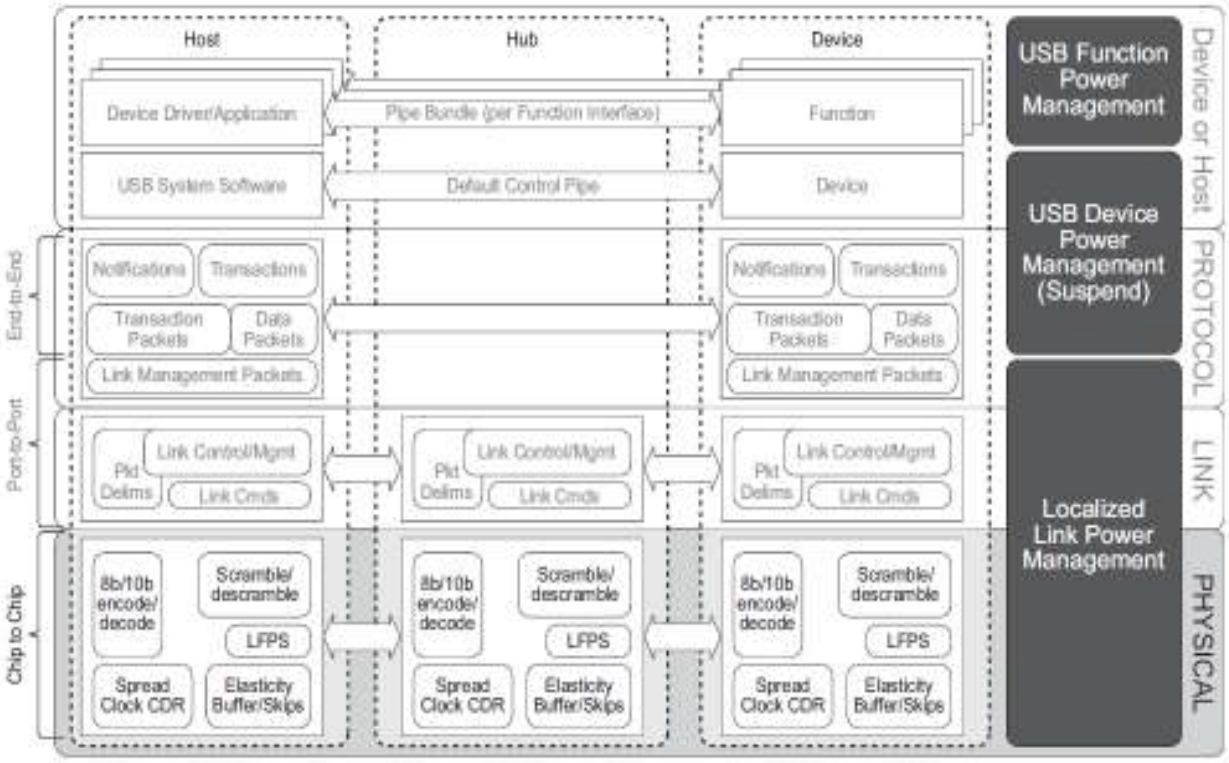

FIGURE 2: *USB 3.0 Different Layers Source: Universal Serial Bus 3.0 Specification*

Apart from verifying the feature set, bigger challenge the design & verification team was facing, is the verification of USB protocol timing parameters for various events like reset, connection detection, set_address , resume/suspend, LTSSM etc. Moreover, the time limits for these events are quite high. For example, the maximum time allow for SetAddress() Completion Time (TDSETADDR) is 50 ms, which translates to long hours of simulation time . The Solution lies in opting for Verification IP which could respond to these challenges.

Cadence Verification IP

The Cadence VIP for USB provides a mature, highly capable compliance verification solution for the USB 3.0 Protocol. The Superspeed USB VIP enables engineers to dramatically reduce the time and risk associated with functional verification, a task which regularly consumes huge percent of the entire chip development cycle. The VIP comes with host of features like reconfigurable timing parameter files (SOMA), error injection, packet retry and multiples queues as per protocol which plays a significant role in verifying all the features of device.

The VIP models a complete USB3 stack as depicted in the USB3 Specification. The USB3 VIP is built around *layers* and *queues*. Layers represent some well-defined functionality in the USB3 flow. *Queues* transport data from one layer to the other, or

represent a change in the contents of the packet. *Layers* are where the processing of input data happens to convert it into a form suitable for output. *Queues* transport data from one layer to the other, and ultimately to the pins that form the interface with another host or device.

In using Cadence USB3 VIP, you can generate tests quickly and efficiently to ensure that your design under test (DUT) is compatible with the other components that may be used in the end system. The extensive timing and protocol checks built into the VIP monitor help you verify your design by generating warnings and errors when protocol or timing violations occur.

Verification Environment & Planning

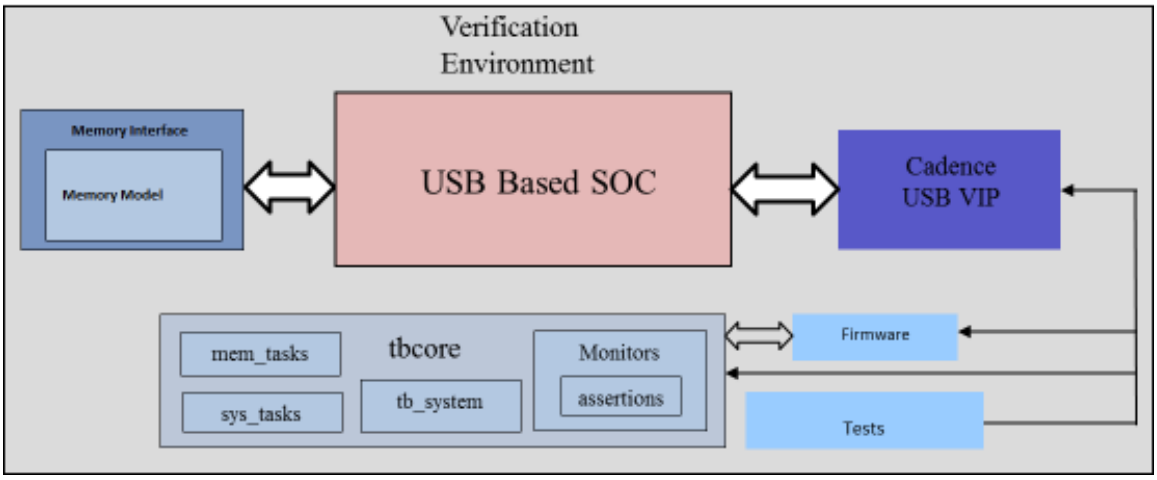

FIGURE 3: *Verification Environment Block Diagram*

Our verification environment is adopted from the earlier USB based projects verification environment, the verification environment is composed of three levels: The top level is the stimulus generator which generates directed test cases and required firmware. The second level is the functional level containing system tasks, memory tasks, assertion and simulation control required for verification environment. The lowest level is signal level which contains the host USB model (VIP), Memory Model and DUT.

The objective of verification activity was not only to verify the SoC functionality but also to provide a platform where the early stage firmware could be tested. Hence the verification plans were built around device as well as system level. The SoC under consideration is intended for USB based mass storage device where only USB pins drive data in the SoC and no other mechanism is available to push data inside SoC. For this purpose we have divided our verification efforts around 3 areas covering the entire USB interface. As shown in figure 5, we have divided our verification efforts in creating Super speed, High Speed, Full Speed and Common USB test cases. The common USB

functionality test (figure 7) contains test related to the protocol and application layer (SCSI & BOT) which remained unchanged in different version of USB. This division helped us in creating modular tasks as well hitting the corner cases easily. Overall we have planned more than 100 tests/corners in this way.

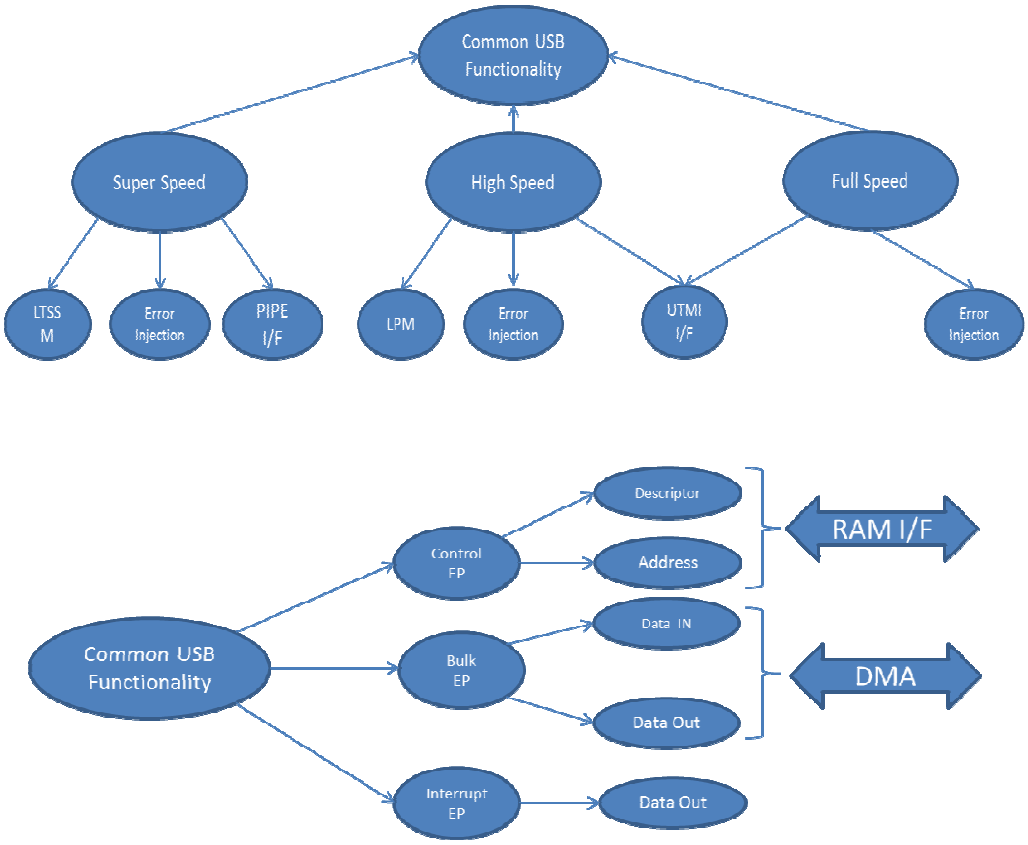

FIGURE 4: *Verification Partitioning*

Verification Execution

The verification execution was divided into various phases.

This helped in creating an effective and efficient platform for SoC. Below mentioned table shows our approach towards the same. It can be noticed that for few phases, the entire team was focused and for others, the resources were involved in different activities.

Stage	Task
Phase 1	Understanding VIP and Scaling Down timing parameters
Phase 2	Hooking up VIP with USB 3.0/2.0/1.1
Phase 3	a) Strength Modeling & Signaling b) LTSSM
Phase 4	PIPE Interface UTMI Interface
Phase 5	Error Injections
Phase 6	Common USB Task
Phase 7	BOT Host Modeling

TABLE 1: *Verification Planning*

Phase 1- Exploring VIP features - SOMA Files

Fortunately, our USB MAC+PHY Design supports scale down timing parameters. So at the first place we have decided to scale down VIP parameters to match scaled down design parameters. USB's functionality and timings are configurable in the Cadence SOMA file. The SOMA describes the VIP modeling rules and properties of each model. The SOMA file describes each timing parameter as per USB Specification and it quite easy to modify. We have modified SOMA file as per our design requirement. Beginners can use PureView GUI version also to modify the SOMA Files. Even otherwise manually modifying is not difficult task. Here is the list of timing parameters we have modified in the SOMA file.

```
<timing.parm name="Tdsetaddr_max" origin="default">
  <value>X ms</value>
</timing.parm>

<timing.parm name="Tdsetaddr_max" origin="default">
  <value>Y ms</value>
</timing.parm>
```

In the above mentioned figure we have changed the Tdsetaddress parameters value from X ms (millisecond) to Y us (microsecond). This will help to run faster simulation and keeping the functionality & protocol same.

Note 1: Cadence VIP also offers standard UVM System Verilog config class that can be used instead of the SOMA file

Note 2: Users must take precaution while modifying the SOMA File parameter as some parameter are dependent on other parameters and modifying these parameters should be done carefully. The authors would like to suggest contacting Cadence Support Engineer before making any changes.

Phase 2- Hooking Up VIP

There are two VIP modules instantiated at top level. First one is active host and the second one is passive USB monitor. The Host model acts as a bus functional model (BFM) with protocol checking. The Host model characteristics are described in the SOMA specification. The VIP monitor records the inputs to the DUT, and verifies correctness by tracking the DUT's state based upon what has been received, and checking that the DUT transmits appropriately for a given state. The DUT connection with VIP is shown below code snippets. Notice the connectivity between TX/RX pin in host and DUT model.

```
8    // **USB Host** model Instantiation
9    usb3host host ( .TX(TX), .TX_(TX_), .RX(RX),  .RX_(RX_)       );
10   // **USB Monitor** model Instantiation
11   usb3mon  mon  ( .TX(TX), .TX_(TX_), .RX(RX),  .RX_(RX_)       );
12   // **USB Device** model Instantiation
13   usb3dev device ( .TX(RX), .TX_(RX_), .RX(TX),  .RX_(TX_)      );
```

FIGURE 5: *Testbench Snapshot*

Cadence VIP also utilizes one of hidden files .denalirc which contains Simulation Specific Settings for The VIP that includes details about dump files, log files, trace files and other USB Specific Settings in VIP. Users are expected to set this file before running the VIP.

Phase 3a – Strength Modeling –USB 2.0/1.1

Since the USB2.0/1.1 has differential lines which are not possible to model in simulation, hence an intermediate module is used between DUT and HOST which could translate the drive strength based upon DUT specification. Users are advice to correctly model the drive strength of the DUT otherwise it will there will be unnecessary and untimely state transition can occur.

Phase 3b – LTSSM

Link Training and Status State Machine (LTSSM) is a state machine defied for link connectivity and the link power management. LTSSM consists of 12 different link states that can be characterized based on their functionalities.

- 4 Operational & Power State -> U0,U1,U2,U3
- 4 Link State Rx.Detect ,Polling, Recovery and Hot Reset
- Loopback and Compilance
- SS.Inactive and SS.Disable

Although the 3rd party USB PHY and MAC were silicon proven core, still we have perform a basic level of LTSSM state check to ensure that the SoC is able to enter and exit all the state properly.

VIP Supports entry and exit from various LTSSM state by configuring

DENALI_USB_REG_MOD_CTRL_LINK register. Also, the timing parameter for each LTSSM state can be configured in SOMA File.

Below is a code snippet to enter into different LTSSM state using VIP register.

```
71          #10;
72          result = $mmwriteword4(hostPortId, DENALI_USB_REG_MOD_CTRL_LINK, 32'h0100);
73          @device_ltssm_state_U1;
```

FIGURE 6: *Code Snippet to enter into U1 State*

Phase 4a – UTMI Interface

The USB 2.0 Transceiver Macrocell Interface (UTMI) handles the low level USB protocol and signaling. This includes features such as; data serialization and deserialization, bit stuffing and clock recovery and synchronization. The primary focus of this block is to shift the clock domain of the data from the USB 2.0 rate to one that is compatible with interface in the SoC.

The major portion of UTMI interface can be tested during the device detection and connection phase itself. Also, a Link Switch between USB 2.0 and USB 1.1 covers the most of control signal of UTMI interface. Hence we have not created any exclusive test case to verify control signal of UTMI interface. However to verify the data path we have switched data transfer rate in our test cases from 8 to 16 bit and vice versa. This switching will result in UTMI Clock switching.

Phase 4b – PIPE Interface

PIPE (PHY Interface for the PCI Express Architecture) interface is a standard interface defined between a PHY sub-layer which handles the lower levels of serial signaling and the Media Access Layer (MAC) which handles addressing/access control mechanisms.

In our Design most of PIPE signal are mapped to USB MAC registers, where values of this registers can be toggled to notice the corresponding changes in the signal interface.

One of the most important control signal in PIPE interface was PowerState which define the power state of SoC. The PowerState signal is directly mapped to LTSSM states and this signal gets covered in the LTSSM state test case itself.

Phase 5 – Error Injection

Error injection is one of the most critical parts of SoC Verification to identify whether the DUT is able to report various errors like protocol error, timing errors and link errors. It was not possible to introduce all kinds of error in the system; hence we have opted for the possible error scenario which could occur in "real world" situations. Following Error's packet and transfer were force into the DUT

- CRC 5/16/32
- Packet Retry – PID Error
- Soft Disconnect
- Timing Errors

Cadence USB VIP has a number of pre-defined errors. When these error injections are directed by the test bench, the VIP injects them according to the USB Specifications. The error injection type (*DENALI_USB_EI_**) is an integer value that can be set to the packet field *DENALI_USB_FLD_ErrorInjection*. Below is code snippet to introduce various error conditions

```
93          pkt.ErrorInjection[31:0] = DENALI_USB_EI_Crc16;
94          pkt.ErrorInjectionNumBits = 32;
```

Packet retry: While introducing CRC errors the DUT itself request for packet retry , but using VIP we can even inject CRC/PID error generated by the device itself. This help in verifying the retry data path in SoC. In order to do a packet retry, we either corrupt the CRC of the received packet or we can set Retry Bit of the ACK Packet. Below is the code snippet for BULK IN packet retry using retry bit.

```
if((cb.reason == DENALI_USB_CB_TX_ProtocolQueueEnter) && (pkt.TransferType == DENALI_USB_TRANSFER_TYPE_Bulk) &&
   (pkt.HeaderType == DENALI_USB_TYPE_TP) && (pkt.TpSubType == DENALI_USB_TP_SUBTYPE_Ack) && (pkt.TransferDir == DENALI_USB_TRANSFER_DIR_In))begin
    pkt.TpRetryDataPacket = 1;
    pkt.TpDpSeqNum = 0;
    pkt.transSet;
    $display("%0t -- DEBUG :: retry bit is set...", $time);
end
```

FIGURE 7: *Test case Snapshot*

Below is the code snippet for BULK IN packet retry by injecting CRC error

```
if(cb.reason == DENALI_USB_CB_RX_ProtocolQueueEnter) begin
 if ((pkt.HeaderType == DENALI_USB_TYPE_DP) && (pkt.DpSubType == DENALI_USB_DP_SUBTYPE_DPP) &&
     (pkt.TransferDir == DENALI_USB_TRANSFER_DIR_In) && (pkt.TransferType == DENALI_USB_TRANSFER_TYPE_Bulk)) begin
   pkt.ErrorInjection[31:0] = DENALI_USB_EI_Crc32;
   pkt.ErrorInjectionNumBits = 32;
   pkt.transSet;
 end
end
```

FIGURE 8: *Test case Snapshot*

Soft Disconnect: When the device is attached to host there is good chance that the host got rebooted for some reason and it is quite important to test how the SoC will behave under such circumstance. Hence we have created Soft Disconnect test case for all USB versions. The VIP can perform soft disconnect operation by just writing to DENALI_USB_REG_MOD_PROP_CONF register. Below is snapshot to perform a device soft disconnect.

```
wdata= 'h00;
result = $mmwriteword4(hostModelId, DENALI_USB_REG_MOD_PROP_CONF, wdata);
```

FIGURE 9: *Test case Snapshot*

Timing Errors: Timing errors can be easily created by modifying the SOMA file. The DUT Behavior under timing errors may not be certain. Hence users should decide what all timing parameter he/she can modify before writing the test case.

Phase 6 – Common USB Function

In the USB evolution the protocol layer remained more or less the same. Some of common functions we have developed are

- Device Enumeration – Set_Address( )
- Descriptor - Device, Configuration, Interface & Endpoint
- Endpoints – Control IN/OUT , Bulk (1) IN/OUT , Bulk (2) IN/OUT , Interrupts
- Bulk Data Transfer

The USB VIP has inbuilt feature which can generate the device enumeration packet, device descriptor request and decode the descriptor information sent by the device. To enable the auto enumeration by default, user need to enable auto enumeration parameter in .denalirc file as mentioned below

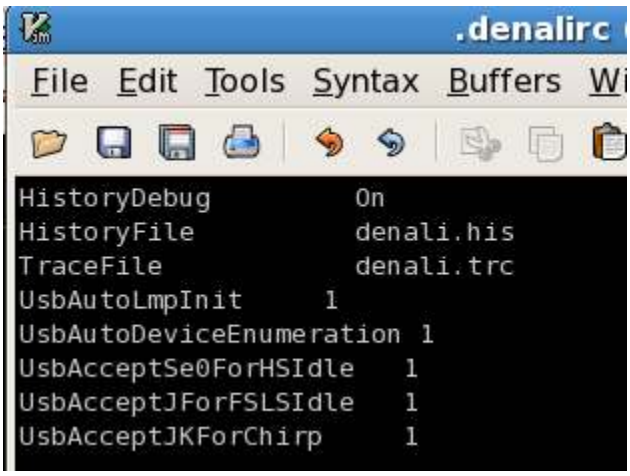

FIGURE 9: *.Denalrx file Snapshot*

When the auto enumeration parameter is set, VIP will generate the request and packets as per actual USB Host. The address of the device is going to be generated randomly by the VIP, in case user want to fix the device address, he can configure DENALI_USB_REG_MOD_PROP_ADDR register to new device address as shown in figure below.

```
result = $mmreadword2(devId, DENALI_USB_REG_MOD_PROP_ADDR, devAddress);
$display("%0t -- DEBUG :: Device Address %h ", devAddress , $time);
```

FIGURE 10: *Test case Snapshot*

After the Set_Address stage is done the VIP Monitor enters into the Address State. To verify whether the set_address step is done properly or not , user can grep the trace file. In the trace file, the model will switch the state from default to addressed as shown in figure below.

```
*Denali* Setting the device address of the connected device to be 1
*Denali* <top_tb.host>@1245846664030 fs :: top_tb.host(p_0.model): Model state changing from default to addressed
*Denali* Setting the device address to be 1
*Denali* <top_tb.dev_mon>@1246031204230 fs :: top_tb.dev_mon(model): Model state changing from default to addressed
```

FIGURE 11:*Log File Snapshot*

Once the set address state is done ,then VIP will start sending device descriptor request to the DUT. The VIP has capability to decode the descriptor data received from the device. Upon successful reception of descriptor data by VIP , it will display the endpoints information received as shown below.

```
*Denali* <top_tb.host>@1493241047630 fs :: Details of endpoints for device
*Denali* <top_tb.host>@1493241047630 fs :: Endp Number:  0 Direction: Inout TransferType: Control
*Denali* <top_tb.host>@1493241047630 fs :: Endp Number:  1 Direction:   Out TransferType: Bulk
*Denali* <top_tb.host>@1493241047630 fs :: Endp Number:  1 Direction:    In TransferType: Bulk
*Denali* <top_tb.host>@1493241047630 fs :: Endp Number:  2 Direction:   Out TransferType: Bulk
*Denali* <top_tb.host>@1493241047630 fs :: Endp Number:  2 Direction:    In TransferType: Bulk
*Denali* <top_tb.host>@1493241047630 fs :: Endp Number:  3 Direction:    In TransferType: Interrupt
*Denali* Class: internal  Instance: "top_tb.dev_mon(conf_0)" Size: 382x32

*Denali* Class: internal  Instance: "top_tb.dev_mon(conf_0.intf_0)" Size: 382x32
*Denali* Class: internal  Instance: "top_tb.dev_mon(conf_0.intf_0.endp_0)" Size: 382x32
*Denali* Class: internal  Instance: "top_tb.dev_mon(conf_0.intf_0.endp_1)" Size: 382x32
*Denali* Class: internal  Instance: "top_tb.dev_mon(conf_0.intf_0.endp_2)" Size: 382x32
*Denali* Class: internal  Instance: "top_tb.dev_mon(conf_0.intf_0.endp_3)" Size: 382x32
*Denali* Class: internal  Instance: "top_tb.dev_mon(conf_0.intf_0.endp_4)" Size: 382x32

*Denali* <top_tb.dev_mon>@1493244567600 fs :: Details of endpoints for device
*Denali* <top_tb.dev_mon>@1493244567600 fs :: Endp Number:  0 Direction: Inout TransferType: Control
*Denali* <top_tb.dev_mon>@1493244567600 fs :: Endp Number:  1 Direction:   Out TransferType: Bulk
*Denali* <top_tb.dev_mon>@1493244567600 fs :: Endp Number:  1 Direction:    In TransferType: Bulk
*Denali* <top_tb.dev_mon>@1493244567600 fs :: Endp Number:  2 Direction:   Out TransferType: Bulk
*Denali* <top_tb.dev_mon>@1493244567600 fs :: Endp Number:  2 Direction:    In TransferType: Bulk
*Denali* <top_tb.dev_mon>@1493244567600 fs :: Endp Number:  3 Direction:    In TransferType: Interrupt
```

FIGURE 12: *log File Snapshot*

The next task after device enumeration is to communicate with the actual Bulk endpoint. VIP has common template for USB 3.0/2.0/1.1 to send a Bulk IN/OUT packet.

```
pkt.transNewQueue(hostId, 0);
pkt.TagId = tagid;
pkt.Addr = devAddress;
pkt.Endp = ep_num
pkt.Type = DENALI_USB_TYPE_Transfer;
pkt.TransferType = DENALI_USB_TRANSFER_TYPE_Bulk;
pkt.TransferDir = DENALI_USB_TRANSFER_DIR_Out; //   DENALI_USB_TRANSFER_DIR_In for IN EP
pkt.Length = 1024;                             //   512 for USB2.0 and 64 for USB 1.1
pkt.Version = DENALI_USB_PKT_VERSION_3;        //   DENALI_USB_PKT_VERSION_2 for USB2.0
$display("%0t -- DEBUG :: Adding BULK OUT pkt to denali_model user queue", $time);
pkt.transAdd(DENALI_ARG_trans_append);
@bulkOutTarnsCompleted
```

FIGURE 13: *Test Case Snapshot*

Hundreds of Bulk IN/OUT packets were initiated from the VIP creating a huge data transfer in the SoC via DMA operation. This has helped in evaluating the bandwidth of the device also what are performance bottleneck in the design.

Phase 7 – System Level Verification: - SoC Simulations or FPGA Dilemma

The last phase of verification activity was aimed towards system level verification. After completing the basic RTL level check the dilemma system's team was

facing was how to verify early production level firmware. Although FPGA platform are most ideal and obvious choice for this however it has its own limitations. FPGA hardware in true sense can meet ASIC clock speed however it is not possible to perform timing closure as close as ASIC. Hence, FPGA Evaluation ends up in running system at lower clock frequency leaving wide functionality and performance gap.

But ASIC Simulation comes with own limitations. First and foremost is unavailability of Host model. It is imperative to have a proper host model to generate a software/application level transaction. Other challenge is long simulation time and complex firmware requirements.

To solve these challenges, we have emulated a Bulk Only Transfer (BOT) model using USB VIP. With Bulk-Only Transport, data is transferred between the host PC and a function using bulk data transport only. Bulk transport can be divided into two types, depending on the direction in which the data is sent. In the bulk only protocol, a successful data transfer has 3 stages

1) Command Transport 2) Data transport 3) Status transport

In the command transport stage, the host sends a command in structure called a command block wrapper (CBW).In the data transport stage, the host or device sends the requested data. In the status transport stage, the device sends status information in a structure called a command status wrapper.

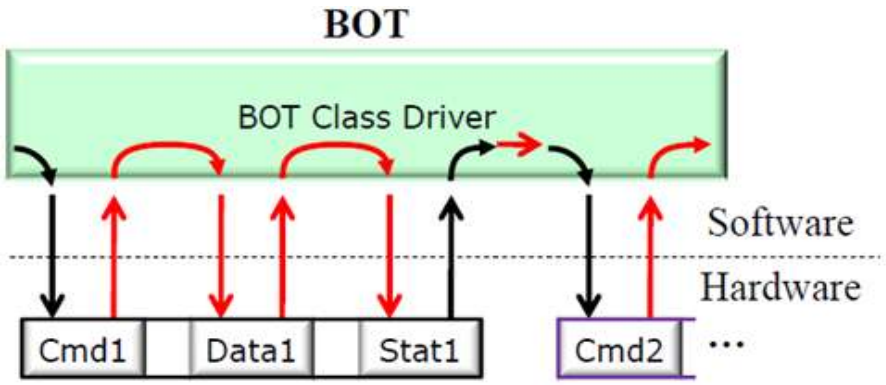

FIGURE 14: *Bulk Only Transfer Flow*

The structure for CBW and CSW is shown below.

Command Status Wrapper

bit / Byte	7	6	5	4	3	2	1	0
0-3	*dCSWSignature*							
4-7	*dCSWTag*							
8-11 (8-Bh)	*dCSWDataResidue*							
12 (Ch)	*bCSWStatus*							

FIGURE 15: *CSW Structure*

Command Block Wrapper

bit / Byte	7	6	5	4	3	2	1	0
0-3	*dCBWSignature*							
4-7	*dCBWTag*							
8-11 (08h-0Bh)	*dCBWDataTransferLength*							
12 (0Ch)	*bmCBWFlags*							
13 (0Dh)	Reserved (0)				*bCBWLUN*			
14 (0Eh)	Reserved (0)				*bCBWCBLength*			
15-30 (0Fh-1Eh)	*CBWCB*							

FIGURE 16: *CBW Structure*

Reference diagram from:

http://www.usb.org/developers/devclass_docs/usb_msc_overview_1.2.pdf

As shown BOT model lot of command and transfer types and it is not possible to support all the transfer type in emulated model, hence we decided to support command and transfer sequence which are mostly “Hardware Centric” rather than software centric. To make BOT Command more “Hardware Centric” we have modified basic

Command and Status structure itself. This has helped in reducing verification firmware complexity and overall system brings up time.

- Struct __command_t

```
{
  uint32_t num_of_sec
  uint8_t    read_write
  uint32_t addr_low
  uint32_t addr_high
  uint32_t ecc_config
  uint32_t tag_id
} __command
```

- Struct __status_t

```
{
  uint32_t num_of_sec
  uint8_t    read_write
  uint32_t addr_low
  uint32_t addr_high
  uint32_t ecc_config
  uint32_t tag_id
  uint8_t    success_fail
  uint32_t   error_code
} __status
```

Tricks for Cadence VIP

Cadence VIP provides lots of features and configurations, and sometimes it can get difficult to manage and play around with them. However, there are workarounds to the issues we encountered and we were able to extract the optimum use of the VIP.

1) Trace Files

Cadence VIP captures sub-nano second of activity over the UBS bus and thus the trace/history files generated are gigabyte size. It is bit difficult to manage with such a huge files. Here are some of tricks users can deploy to save the memory save and debug time.

- Compression: By default, the dump is generated without any compression. Users can generate dump file in zipped format which saves lot disk space. Following commands need to be added to .denalirc file.

```
HistoryFile -gzip  denali.his.gz
TraceFile   -gzip  denali.trc.gz
```

FIGURE 17: *.denalirc snapshot*

- Selective Dump: It is not necessary to dump the trace and history file for the complete test case run. Users can selectively dump these files and speed up the simulation run time. The dump control should be part of testbench code as shown below.

```
initial
    begin
   #10000  success = $mmdebugon ;
  end
```

FIGURE 18: *Test bench Snapshot*

2) Trace file Debug: As mentioned above, the dump is created for sub-nanosecond activity and it is quite tricky to go through these trace/dump files. You can create a list of shortcut commands which can collect the required information into a separate file. This will save lot of time spent on going through dump/history file.

```
LTSSM FSM transitions
    egrep "0 H.*LTSSM FSM state" denali.trc > ltssm.log

LFPS Signaling
    egrep "0 H.*LFPS-TX|LFPS-RX" denali.trc > lfps.log
```

FIGURE 19: *Example Command*

3) Multiple USB Version: While working on multiple USB versions, we noticed that VIP behavior is different for different versions. This is primarily because different USB versions have different conditions. For example, the maximum bulk packet size can vary from 64 byte in USB 1.1 to 1024 bytes in USB 3. To create a separation between different versions it is a good practice to differentiate each category of test cases with different defines. This will help in identifying the USB version and reduce the debug time.

```
`ifdef USB_FS
        parameter interface_soma
"$PROJ_DIR/project_top/testbench/verilog/models/denali_usb_host/host_fs.so
ma";
  `else
        parameter interface_soma
$PROJ_DIR/project_top/testbench/verilog/models/denali_usb_host/host.soma";
  `endif
```

FIGURE 20: *denali XML File Snapshot*

As a good practice in our project, we had dedicated engineers for each version of USB protocol. This way engineers don't get confused between different protocols versions and can focus only on particular bus protocol only.

Conclusion

It was a complex task to verify the USB 3.0 based mass storage system. However, Cadence VIP has really helped us in steering through the challenges of the verification.

Disclaimer: The intent of paper is ONLY to demonstrate the verification methodology using a VIP to verify our SOC controller. This should not be taken as endorsement of Cadence VIP since we have NOT compared pros and cons of difference USB 3.0 VIPs. We would recommend designers to evaluate VIPs with their verification requirements before choosing a VIP vendor.