VIP Shielding

Jeremy Ridgeway
LSI Corporation
Ph: +1 408-433-4257
Email: Jeremy.Ridgeway@lsi.com

Karishma Dhruv
LSI Corporation
Ph: +1 408-433-8292
Email: Karishma.Dhruv@lsi.com

Abstract—**Third party verification intellectual property (VIP) is often leveraged to achieve testing and coverage goals more quickly. However, also in the interest of time, the verification architecture is not sufficiently shielded from a particular VIP vendor. VIP shielding delineates hard boundaries between the in-house verification environment and the third party VIP. While the acceptance of Universal Verification Methodology (UVM) provides a springboard to VIP shielding, simply relying on the common methodology is not enough. In this paper we show how we incorporated a third party PCI-Express (PCIe) VIP in our verification environment. We sufficiently shielded the environment to enable swap-out should management deem it necessary.**

I. Introduction

Verification environment architecture should consider the possibility of a vendor change when including third-party VIP. A change could be required if, for example, budgets or schedules change, the support relationship changes, or even if the vendor is acquired or goes defunct. In fact, acquisition must be considered whenever an independent vendor is chosen. [7], [12], [11].[1] Precisely where to instantiate third-party VIP and how it should be utilized in the verification environment is fraught with complication.

Starting with the device under test (DUT), a basic test bench architecture connects to all DUT interfaces. Our experience with a layered communication protocol guides the location of VIP. In communication protocols, for example PCI-Express, there is usually a clear delineation between data path and control path interfaces. In figure 1, the vertical protocol stack defines the data path while control tends to be sideband signalling.

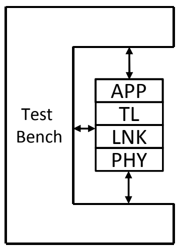

Fig. 1. General test bench for layered DUT: physical (PHY), link (LNK), transaction (TL), and application (APP) layers. The DUT may be a sub- or super-set of the above.

[1]In other words, a vendor that is not Mentor Graphics, Cadence, or Synopsys.

The test bench itself is also layered providing successively coarse levels of abstraction, as in figure 2 [10], [4], [3], [8]. **Signalling** occupies the lowest layer of the test bench. This layer contains the SystemVerilog interface and other constructs to connect the test bench to the DUT. Next, the **command** or **harness** layer manipulates the signals connected with the DUT [10], [3]. In the universal verification methodology (UVM), this layer contains the driver and monitor components *within* an agent [2]. Tasks in the command layer and encapsulated in components define the *how* and *when* to interface with the DUT: stimulating the primary inputs and responding to the primary outputs. The **functional** layer provides the means

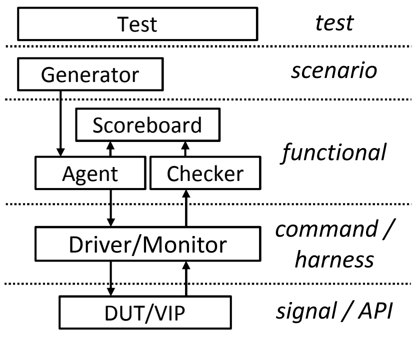

Fig. 2. General layered verification environment.

for generation, scoreboards, and checkers. UVM sequencers within UVM agents provide the means and connections for data and control path stimulus generation. UVM scoreboards and checker components implement verification environment self-checking [4], [2]. The **scenario** layer contains a library of sequences for random stimulus generation as well as coordinates when and where the sequences operate. UVM virtual sequencers and environments conduct the random verification by initiating UVM sequences in the appropriate functional layer sequencer [2]. The expansion of the random testing space occurs in the scenario layer as virtual sequencers randomly choose the order of stimulus generation. Finally, at the top of the verification stack, the **test** layer constricts the random testing space by (further) constraining random order and random value generation throughout the environment [10]. Directed testing in the constrained random verification environment is achieved by constraining everything in a specific test.

In this paper we focus on the first three layers of the verification stack for testing a layered communication protocol, see figure 3.

The missing point in the layered test bench is where and how to implement third-party verification intellectual property (VIP). In figure 3, scoreboards and checkers are used for both *ingress* and *egress* data flow directions through the DUT. Ingress path tests DUT correctness receiving data from some external device while egress tests correctness transmitting. Thus, scoreboards and checkers usually require detail about both the highest and lowest level protocol layers.

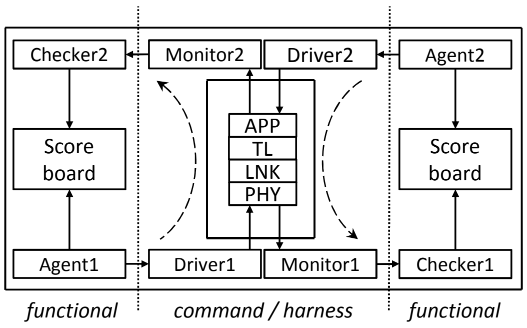

Fig. 3. A general layered verification environment connecting to a layered communication protocol DUT; left arc represents ingress flow testing while right is egress.

The physical layer, the PHY component in figure 3, usually connects the device to some external device, while the application layer, APP, usually interfaces internally within the device. While the PHY often implements a standard communication protocol (to ensure interoperability), the application usually does not. For example, PCI-Express devices must strictly adhere to the physical requirements to ensure communication with other PCI-Express devices. However, the interface on an application layer tends towards a proprietary interface depending on DUT features. Therefore, verification components, as in figure 4, often are built in-house while VIP connects to the standard interface. Figure 5 shows the connected VIP encompassing signalling, command/harness, and function verification layers. The VIP usually includes scenario and test layers, as well. Basically, the VIP can act as an autonomous external device transmitting stimulus to the DUT and responding to received data.

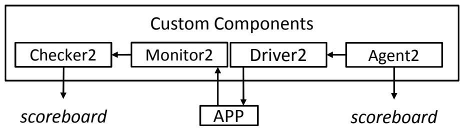

Fig. 4. Custom verification IP connecting to APP layer.

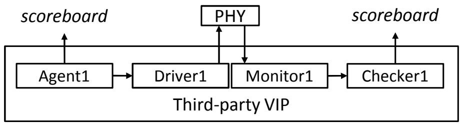

Fig. 5. Third-party VIP connecting to PHY layer.

There is some disconnect in the verification environment between third-party VIP and in-house custom components when VIP is connected to one interface only. The functional verification layer is impacted the most. Notice the connection to the common scoreboard in figures 4 and 5. Precisely how data enters a scoreboard is the root of the verification architecture problem when combining in-house and VIP components.

In this paper we consider verification of a single data path protocol with sideband configuration and control paths. In section II we present considerations for VIP inclusion in the verification environment architecture. We present VIP shielding in section III followed by a case study of a recent PCI-Express project in IV. Finally, we present results and discuss conclusions in V and VI, respectively.

II. VIP Considerations

When implementing third-party VIP into the verification architecture the following questions should be considered.

A. Full Data Path Testing

$$\text{Can a VIP test the full DUT data path interface?} \qquad \text{(Q1)}$$

If a VIP *can* fully test a DUT data path, and the project budget allows, then this is the most expedient and efficient course. The VIP will encompass both the highest and lowest layer interfaces, as shown in the general test bench in figure 3, and test the complete data path. Note that runtime-simulation configuration may still require special handling to coordinate, but the bulk of the testing may be directed by the VIP.

B. Partial Data Path Testing

When one extreme layer of the protocol stack conforms to an industry standard interface while the other does not, then the VIP is connected to one side only in the data path. This is indicated in figure 6.

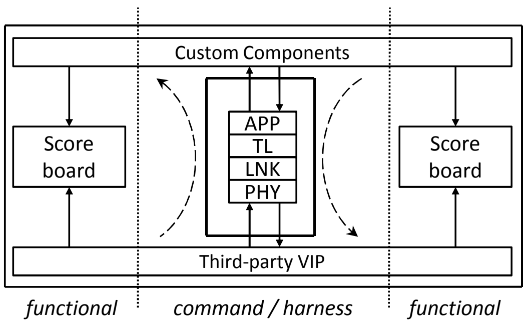

Fig. 6. Third-party VIP connects to the standard interface of a data path while in-house components connect to the proprietary.

The VIP usually provides the full verification stack, from command/harness layer to testing layer. With partial data path testing, should the simulation utilize the full or partial VIP verification stack? In the name of reuse, we often tend towards the full stack, but each ought to be considered separately.

1) Full VIP verification stack: When the full VIP verification stack is used in partial data path testing, VIP scenario and possibly test layers are used to drive simulation.

Two questions are critical to implementation:

$$\text{How to test DUT egress path?} \qquad \text{(Q2)}$$

$$\text{Who owns the functional layer scoreboard?} \qquad \text{(Q3)}$$

These questions are interrelated and directly imply a verification architecture.

Referring to (Q3), ownership of the scoreboard is relative. If the VIP owns the scoreboard then either the VIP vendor provides the scoreboard or the scoreboard is implemented in-house *with VIP-specific SystemVerilog classes*. If the in-house test bench owns the scoreboard, then the scoreboard is implemented *with in-house-specific SystemVerilog classes*.

First, consider when VIP owns the scoreboard. In this case, the custom components should also be implemented with VIP-specific classes. This is the most natural configuration for DUT ingress testing. Then, for DUT egress testing, third-party VIP re-use should be maximized. In figure 7, the VIP verification functional layer is separated from the command layer. In its place, custom driver and monitor components are implemented operating on VIP-specific SystemVerilog classes. The scoreboard now is straightforward using the same classes as the VIP-specific functional layers.

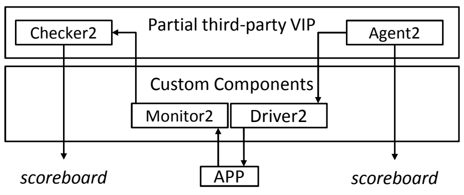

Fig. 7. Re-using third-party VIP functional layer to support DUT egress testing.

Now, consider when the test bench owns the scoreboard. In this case, the custom components should *not* use VIP-specific SystemVerilog classes. Instead of re-using the VIP components, a translation sub-layer provides conversion between in-house and VIP-specific SystemVerilog classes. Referring back to figure 6, the translation sub-layer would exist between the third-party VIP and the scoreboard.

This approach seems counter-intuitive. However, when the test bench owns the scoreboard, then a change in third-party VIP is possible within the confines of the existing verification environment. Otherwise a new verification environment built around a new VIP is required.

2) Partial VIP verification stack: When the full VIP verification stack is used in partial data path testing, VIP scenario and possibly test functional layers are used to drive simulation.

When the partial VIP verification stack is used in partial data path testing, VIP scenario and test layers are *not* used in simulation. Instead, the in-house test bench employs UVM sequences and virtual sequencers to direct the verification environment. As such, both (Q2) and (Q3) is owned by the in-house test bench. This is the most natural configuration for DUT egress testing. To maximize reuse, stimulus generation agents from the application layer side should also be used for the physical side, as shown in figure 8 (and is essentially the reverse of figure 7). Instead of using third-party VIP for scenario test layers, the in-house custom components perform the job to support DUT ingress testing.

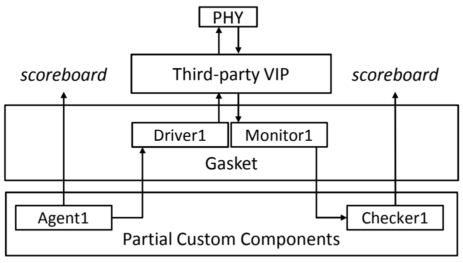

Fig. 8. Re-using custom component functional layer to support DUT ingress testing.

Now, the scoreboard, application layer interface, and physical layer interface components all use in-house SystemVerilog classes. The gasket shown in figure 8 implements necessary translations between the VIP and in-house test bench. Furthermore, the gasket is fully in the command/harness verification layer. As such, the full of the VIP instantiation is regulated to the command/harness and signalling verification layers.

The verification environment at large, in figure 9, is *shielded* from any specific VIP via the gasket.

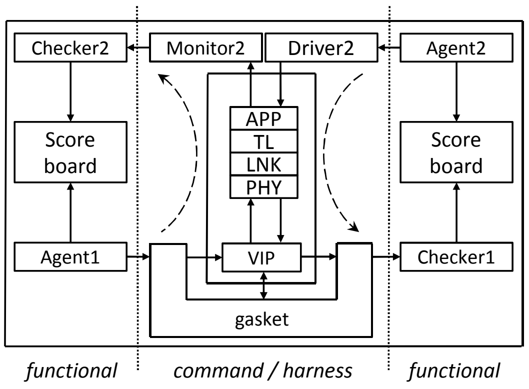

Fig. 9. Gasket shields the environment from the VIP.

III. VIP Shielding

VIP shielding insulates the verification environment from changes in the third-party VIP. This is accomplished in dual roles. First, shielding means separating the in-house test bench from the third-party code. Hard boundaries must exist in the test bench, ensuring no in-house code direct access to third-party VIP. This is especially important for data path components as they encompass the bulk of the functional testing and coverage. Second, shielding defines a common interface in which to interact with any specific third-party VIP. The essential common interface is one or more abstract classes that define a set of methods the in-house test bench interacts with. These methods provide to the test bench the means, in simulation, to:

A. configure the VIP, and
B. handle data.

If the vendor can meet the requirements of the abstract interface then the VIP is *guaranteed* compatible with the test bench. In figure 10, the **gasket** represents the abstract interface. The solid arrows indicate represent method calls between test bench components and the gasket. For example, driving data from the test bench to the gasket could be achieved via a function call to the gasket. In UVM, this could also be a transaction level model (TLM) port such as an analysis port. The dotted arrows represent the abstract methods that must be implemented in the gasket to integrate with the VIP, for example driving data to the VIP from the gasket. These methods are the minimum test bench requirements for any VIP. If the vendor can integrate with the environment only defining these methods, thereby defining a VIP-specific gasket, then the VIP is compatible.

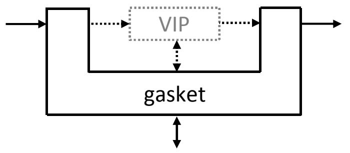

Fig. 10. Abstract gasket interface. Test bench components use methods depicted as solid arrows; VIP-specific gasket implements abstract methods depicted as dotted arrows.

Each of the configuration and data paths is described in more detail in this section. Following, a reference gasket class hierarchy is presented.

A. Configuration Path

A simple example of VIP shielding on the configuration or control path is the register abstraction layer (RAL) model in UVM, refer to figure 11. The verification environment does not need to know whether the actual CPU bus model exercising register reads and writes is from Synopsys, Cadence, or any other vendor. As long as the VIP conforms to the requirements of the RAL model, it will work in the verification environment.

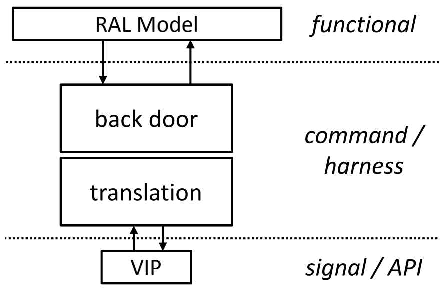

Fig. 11. Gasket layers in the configuration path.

The RAL model front door or back door access to registers is the gasket shielding the test bench in the configuration and/or control path. In figure 11, the RAL model exists in the functional layer because it encapsulates the agents and checkers required to drive and respond to command/harness layer operations. When test code requests a register read, the RAL model instructs the appropriate door to execute a register transaction and return results.

The generic RAL model gasket to VIP are the abstract UVM classes that define the back door interface, uvm_reg_backdoor [1]. With VIP shielding, the RAL model should contain all configuration parameters that the test bench can modify. In the vendor-specific back door gasket code, a translation from in-house parameter to vendor parameter occurs. This could be an address translation (one register to another) or a value translation. VIP shielding delineates a hard boundary between the in-house test bench configuration and VIP configuration that operates transparent to test code.

B. Data Path

Data path VIP shielding tends to be more complicated than RAL because it is more tightly integrated to the verification environment. In figure 9, we have isolated the third-party VIP within the boundaries of a gasket. The gasket is a collection of abstract classes that define an interface which the in-house test bench must use. Test bench components or sequences that require VIP access shall use this interface. The gasket contains both an application programming interface (API) as well as necessary UVM transaction level modeling (TLM) ports and implementation ports.

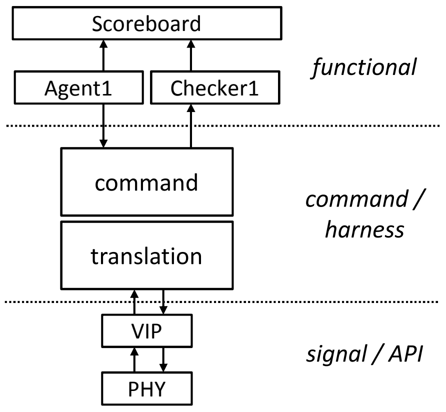

Fig. 12. Gasket layers in the data path.

In figure 12, the agents re-used from custom components handle scenario and random stimulus generation on the transmit path as well as checking on the receive path. The gasket defines methods for the functional layer to transmit and receive data packets. The agent shown drives data transmission. The gasket reacts by translating the packet to VIP-specific classes and forwarding to the VIP via API. Simultaneously, the VIP drives data reception through callbacks to the gasket. The gasket reacts again by translating the packet to in-house-specific classes and forwarding to the checkers.

C. Gasket

Conceptually, the gasket looks complicated, but implementation is generally straightforward. The internal facing side of the gasket is a lightweight set of APIs the test bench requires to interact with the VIP. These functions are fully defined. For example, in figure 13, gasket_base has three ports:

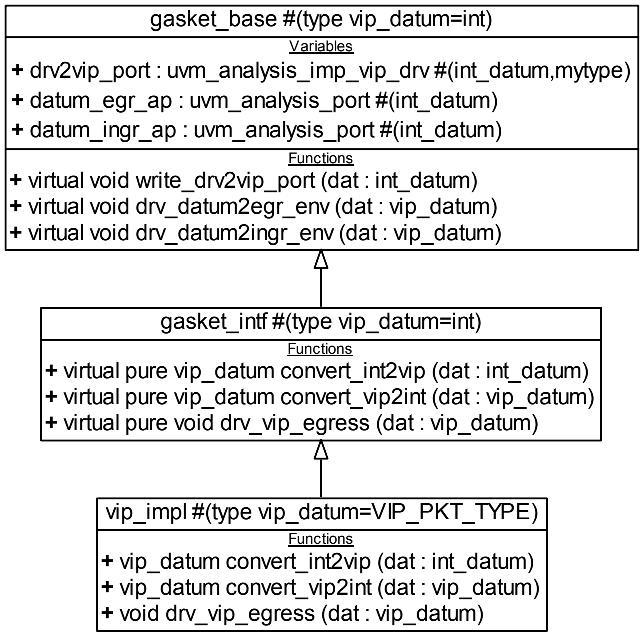
gasket_base #(type vip_datum=int)
Variables
+ drv2vip_port : uvm_analysis_imp_vip_drv #(int_datum,mytype)
+ datum_egr_ap : uvm_analysis_port #(int_datum)
+ datum_ingr_ap : uvm_analysis_port #(int_datum)
Functions
+ virtual void write_drv2vip_port (dat : int_datum)
+ virtual void drv_datum2egr_env (dat : vip_datum)
+ virtual void drv_datum2ingr_env (dat : vip_datum)
gasket_intf #(type vip_datum=int)
Functions
+ virtual pure vip_datum convert_int2vip (dat : int_datum)
+ virtual pure vip_datum convert_vip2int (dat : vip_datum)
+ virtual pure void drv_vip_egress (dat : vip_datum)
vip_impl #(type vip_datum=VIP_PKT_TYPE)
Functions
+ vip_datum convert_int2vip (dat : int_datum)
+ vip_datum convert_vip2int (dat : vip_datum)
+ void drv_vip_egress (dat : vip_datum)

Fig. 13. General data type gasket architecture.

- drv2vip_port is the data transmit path from test bench to the VIP,
- datum_egr_ap is an analysis port passing transmit data to the test bench scoreboard (expected data),
- datum_ingr_ap is an analysis port passing receive data to the test bench scoreboard (observed data).

Each port connects to a test bench component. The internal facing side of the gasket remains unchanged between VIP vendors.

The gasket external facing side interacts with the VIP. These functions are abstract (pure and virtual in SystemVerilog), gasket_intf from figure 13, and must be defined for a specific vendor. Essentially, they determine what needs to be implemented for test bench communication with the VIP. Then, they are defined for a specific VIP in some vip_impl extension class.

Once the initial gasket architecture is completed, the task to change between vendors involves implementing a new VIP-specific gasket, new_vip_impl. The test bench at large remains separate and unchanged.

IV. Case Study: PCI-Express Project

In a recent PCI-Express project, we were tasked with developing an approach for VIP shielding in the data path. This was because we anticipated a transition after starting the project from an older version to a newer version of the same Vendor's VIP. The versions were not strictly compatible because the vendor itself was transitioning to native UVM classes.

Our DUT consisted of PCI-Express transaction and link layers. The connection from link layer to physical layer adhered to standard PIPE interface [9]. The application layer interface was proprietary. Any VIP chosen could partially test the data path only, as in section II-B. We chose the Cadence PCI-Express VIP to fill the role in a partial VIP verification stack and focused our work on application layer custom components [6]. A gasket layer was placed between the VIP and the re-used custom components to handle data transmission and reception with the DUT, as in figure 9.

A. Gasket Architecture

The gasket is an abstract class that defines an interface for the test bench at large. The diagram in figure 14 shows the necessary connections between the gasket and other custom components. Finally, the class hierarchy for our PCI-Express project implementing Cadence VIP is in figure 15. The base class, pcie_gasket has two parts, UVM TLM port connections and translation services.

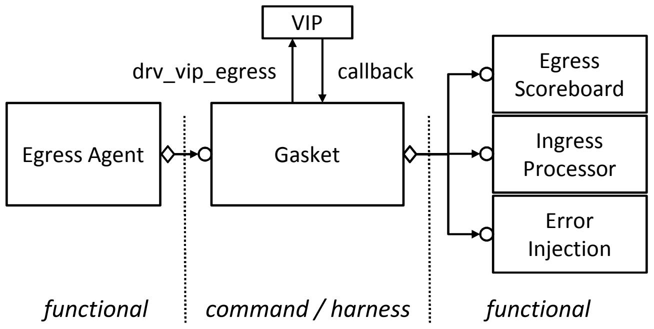

Fig. 14. Gasket connections to verification environment custom components: xmt_ap analysis port to egress scoreboard, rcv_ap to ingress processor, and errinj_ap for error injection .

First, the connections are all analysis ports. After a data packet is randomly generated, it passes to the gasket through its egress implementation port, vip_drv, and defined in write_vip_drv. Upon entering the gasket, the in-house-specific packet class is translated to VIP-specific class via the convert_pcie2vip function. Then, it's passed to the VIP via the drv_write_egress abstract function. This function is the main transmit data path, as shown in 14. Next, upon a VIP callback, the packet passes from the VIP into the gasket and out to the requisite functional layer component. The callback type indicates the destination component.

To support the data path connections, the VIP must support an API to drive *all* data traffic for transmission to the DUT. Also, the VIP must support distinguishing types of callbacks with data traffic that the gasket passes on to upper layer components. Note that both transmit and receive path callbacks must be supported. Finally, the VIP must have an open class definition to allow appropriate translation between in-house test bench and VIP-specific data packet classes.

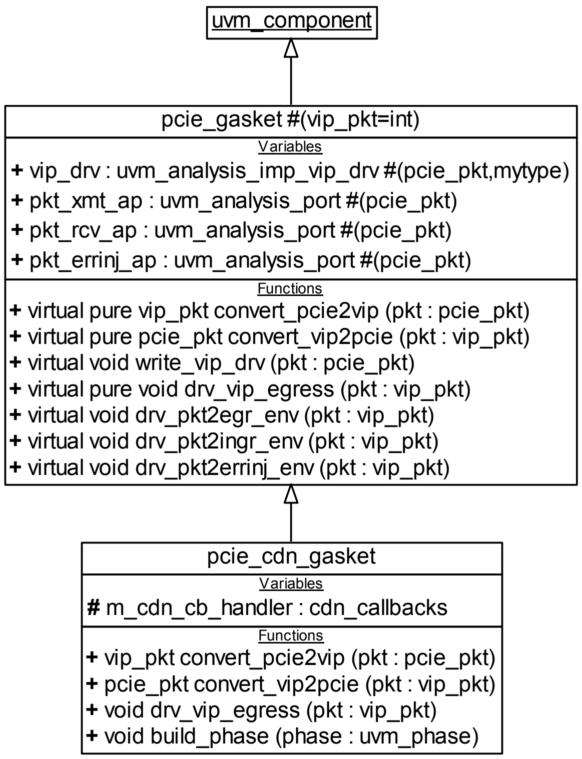
uvm_component
pcie_gasket #(vip_pkt=int)
Variables
+ vip_drv : uvm_analysis_imp_vip_drv #(pcie_pkt,mytype)
+ pkt_xmt_ap : uvm_analysis_port #(pcie_pkt)
+ pkt_rcv_ap : uvm_analysis_port #(pcie_pkt)
+ pkt_errinj_ap : uvm_analysis_port #(pcie_pkt)
Functions
+ virtual pure vip_pkt convert_pcie2vip (pkt : pcie_pkt)
+ virtual pure pcie_pkt convert_vip2pcie (pkt : vip_pkt)
+ virtual void write_vip_drv (pkt : pcie_pkt)
+ virtual pure void drv_vip_egress (pkt : vip_pkt)
+ virtual void drv_pkt2egr_env (pkt : vip_pkt)
+ virtual void drv_pkt2ingr_env (pkt : vip_pkt)
+ virtual void drv_pkt2errinj_env (pkt : vip_pkt)
pcie_cdn_gasket
Variables
# m_cdn_cb_handler : cdn_callbacks
Functions
+ vip_pkt convert_pcie2vip (pkt : pcie_pkt)
+ pcie_pkt convert_vip2pcie (pkt : vip_pkt)
+ void drv_vip_egress (pkt : vip_pkt)
+ void build_phase (phase : uvm_phase)

Fig. 15. Gasket interface class.

The Cadence PCI-Express VIP did support all connections. A single entry point into the Cadence VIP was a user transmit queue access via API. In figure 15, all callbacks were encapsulated in the handler class noted. The callbacks used were:

1) User queue exit to transaction layer,
2) Start transmitting packet,
3) Packet received.

When a test bench generated packet is ready for transmission by the VIP, it exits the user queue. This is the callback used for passing ingress traffic to the scoreboard. The received packet callback connects to the egress scoreboard. Finally, the callback just prior to transmission enables verification environment error injection.

B. Connecting the Custom Components

We re-used the egress path data generators on the ingress path. Figure 16 shows how the egress agent connects to the DUT application layer. In the figure, the driver maintains a local reference to the virtual interface connected to the DUT. Thus, this driver in the command/harness verification layer manipulates DUT primary IOs.

Our test bench handled random traffic generation on both the egress and ingress data paths. Therefore, we re-used the egress agent in the ingress data path. This was accomplished by replacing only the driver class instantiation in the egress agent with one that connects to the gasket implementation port, a protocol layering technique described in [5].

Note that while it is not shown in the figures, the monitor class was also replaced for the gasket connection. However, it was not required because VIP callbacks were used instead. The monitor class override provided no functionality; it was a stub.

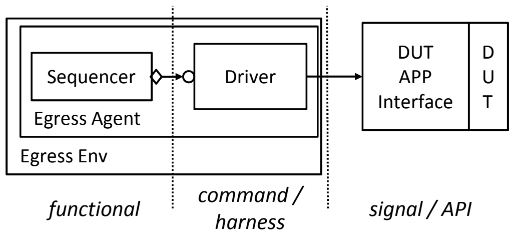

Fig. 16. Egress agent connecting to DUT application layer interface.

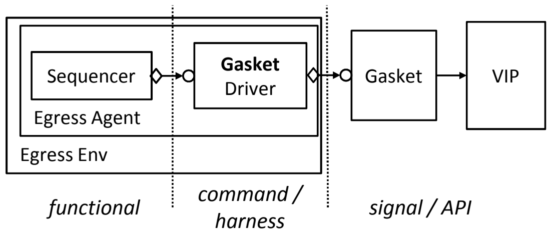

Fig. 17. Egress agent modified to connect to VIP through gasket interface.

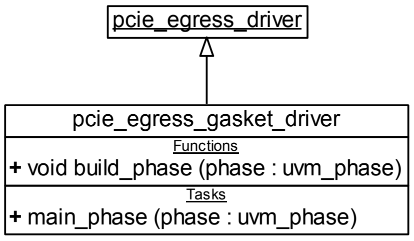
pcie_egress_driver
pcie_egress_gasket_driver
Functions
+ void build_phase (phase : uvm_phase)
Tasks
+ main_phase (phase : uvm_phase)

Fig. 18. Gasket driver extension class.

Referring to figure 18, the extended egress driver only contains functionality in the main_phase.

(It also uses the UVM automation macros for the factory to be aware of the class). In the parent UVM environment class that instantiates the egress agent on the gasket interface, we override the driver instance through the UVM factory.

```
// Override driver to connect to gasket
factory.set_inst_override_by_type(
    pcie_egress_driver::get_type(),
    pcie_egress_gasket_driver::get_type(),
    {get_full_name(), ".*"});

// Create the egress env through factory
egress_env = pcie_egress_env::type_id::
            create("egress_env", this);
```

At runtime, the physical layer egress agent connected to the gasket, see figure 17, uses the same scenario sequences as in the application layer interface. Instead of passing the generated data packets to the DUT, however, it passed them to the VIP for transmission to the DUT.

C. Connecting the RAL model

We instantiated a RAL model for VIP configuration. The VIP parameters that were pertinent to our simulation runtime setup were listed as registers in the RAL model. Then, as shown in figure 19, the back door access acted as gasket code for our environment.

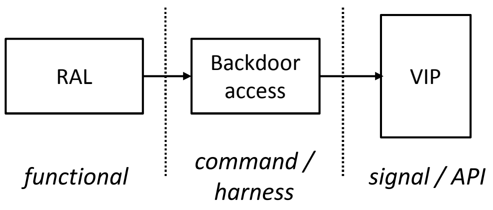

Fig. 19. RAL connects to VIP configuration; translation, if necessary, occurs in the back door class.

The RAL back door gasket fulfilled two roles. First, it connected the UVM RAL model to the API provided by the VIP for configuration. Second, it provided address translation to align the RAL model registers with what the VIP expected. In figure 20, the base back door class, pcie_reg_backdoor, implements the write function only, as required by UVM. The VIP specific back door gasket implements the address translation as well as connecting the read and write functions to Cadence provided APIs to configure the VIP. Note that a reference to the register being operated on is available in the provided uvm_reg_item. This allowed our back door access to include non-traditional parameters like a time unit string.

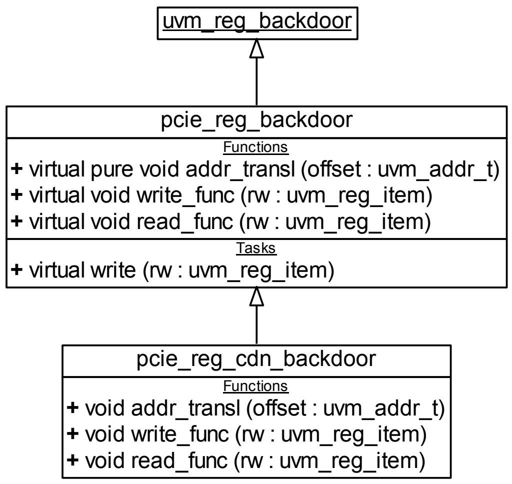
uvm_reg_backdoor
pcie_reg_backdoor
Functions
+ virtual pure void addr_transl (offset : uvm_addr_t)
+ virtual void write_func (rw : uvm_reg_item)
+ virtual void read_func (rw : uvm_reg_item)
Tasks
+ virtual write (rw : uvm_reg_item)
pcie_reg_cdn_backdoor
Functions
+ void addr_transl (offset : uvm_addr_t)
+ void write_func (rw : uvm_reg_item)
+ void read_func (rw : uvm_reg_item)

Fig. 20. RAL back door gasket for VIP configuration.

The RAL model is general enough to handle most zero-time configuration methods. However, because configuration is non-standard between vendors, we do anticipate this step taking some time in the case of VIP vendor change.

V. Results

Our verification environment supports VIP vendor change when they conform to the requirements of both the PCI-Express standard as well as the gaskets. First, the VIP is directly connected to the DUT via the standard PCI-Express physical layer interface. Any PCI-Express vendor should support this connection. In the test bench, only the VIP command/harness and signalling layers are utilized. We have regulated connections to the VIP through two gasket points, one for data path and one for configuration, as in figure 21. Any considered VIP vendor must conform to gasket requirements in order to integrate well in our test bench.

1) The VIP must support callbacks when:
   a) data is queued for transmission,
   b) data is ready for transmission, and
   c) data is received.
2) The VIP must accept user generated data for transmission, and:
   a) must allow user data to transmit as-is (no checking and no changes),
   b) must allow user to modify transmit data,
   c) modified data must be transmitted as-is (no checking and no changes).
3) The VIP must support dynamic configuration via function call(s).

The verification architecture was designed to support swap-out. However, as the version of the VIP was released prior to full environment implementation, a true swap-out was not required. We are not concerned with acquisition and do have requisite assurances on the existing VIP support model. The variable, as verification engineers, exists with upper-management business decisions. To that end, the test bench is able to withstand a vendor change within the confines of the current verification environment.

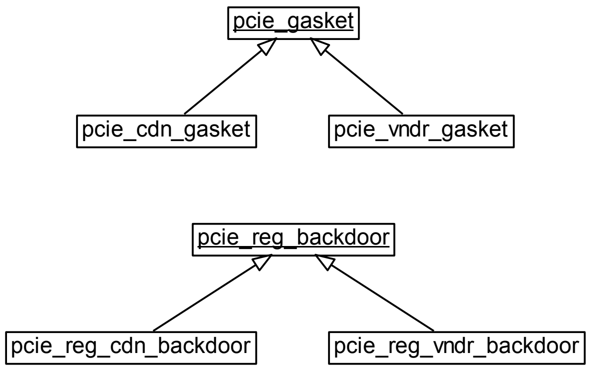

Fig. 21. Supporting multiple vendors simultaneously.

Referring to figure 21, integration into the full verification environment of a new VIP requires only gasket implementation. Once the VIP is instantiated and statically configured properly, dynamic configuration would be debugged at simulation start. Finally, full constrained random simulation occurs.

We found the effort required to implement the gasket itself to be minimal as only a few functions were required. For the VIP vendor chosen, most of our time was spent scouring documentation and contacting their support to understand how to define the gasket data path functions. Once determined, implementation was straightforward. The configuration path, however, was more complicated in that it was not immediately obvious how to compose the RAL model and connect to the VIP. For expediency, we settled on using the vendor's register set as our model. However, moving to another vendor will require a register mapping from one address set to another. We feel that VIP configuration will remain the greatest time requirement in swap-out.

VI. Conclusion

We presented an approach to VIP inclusion in verification architecture that allows for vendor or major revision change through VIP shielding. We found that a gasket is sufficient to shield the verification environment from the specificities of the VIP. Furthermore, implementation of the gasket is relatively straightforward utilizing features of UVM and SystemVerilog.

However, a trade-off in effort and responsibility does exist. With partial data path testing (VIP connected to one side of the DUT) and partial VIP stack (using some but not all the verification layers), the onus is on the in-house environment for all verification activities. The VIP can assist in protocol checking, but as its scenario layer is disabled it cannot produce random stimulus. However, if the VIP can be utilized in both extremities of the DUT protocol layer, then the verification onus returns to the VIP. Considering the unknown nature of the VIP market and management decisions, having a plan for VIP change without disrupting verification (too much) is a wise idea.

References

[1] Universal verification methodology reference implementation. http://www.accellera.org/downloads/standards/uvm, October 2010.

[2] Acellera. *Universal Verification Methodology (UVM) 1.1 User's Guide*, 2011.

[3] J. Bergeron. *Verification Methodology Manual for SystemVerilog*. Springer, 2006.

[4] J. Bergeron. *Writing testbenches using System Verilog*. Springer Science+Business Media, 2006.

[5] J. Bergeron, F. Delguste, S. Knoeck, S. McMaster, A. Pratt, and A. Sharma. Beyond uvm: Creating truly reusable protocol layering. In *Design & Verification Conference*, 2013.

[6] Cadence Design Systems, Inc. *Cadence PCI Express PureSpec VIP User Guide*, 11.3 edition, July 2011.

[7] Denali Software, Inc. Cadence to acquire Denali. http://www.denali.com/wordpress/index.php/news/2010/05/13/cadence-to-acquire-denali, May 2010.

[8] S. Iman. *Step-by-Step Functional Verification with SystemVerilog and OVM*. Hansen Brown Publishing Company, 2008.

[9] Intel Corporation, Inc. *PHY Interface for the PCI Express Architecture, PCI Express 3.0*, revision .9 edition, 2010.

[10] C. Spear. *SystemVerilog for Verification, a Guide to Learning the Testbench Language Features*. Springer Science+Business Media, 2010.

[11] Synopsys, Inc. Synopsys acquires nSys Design Systems. http://news.synopsys.com/index.php?s=20295&item=123308, Sep 2011.

[12] Synopsys, Inc. Synopsys acquires ExpertIO. http://news.synopsys.com/index.php?s=20295&item=123347, Jan 2012.