Wiretap your SoC

Why scattering Verification IPs throughout your design is a smart thing to do

Avidan Efody
Mentor Graphics Corp.
Wilsonville, OR, USA
avidan_efody@mentor.com

Abstract—**Today's SoCs often contain tens or even hundreds of standard interfaces such as AXI, PCIe, USB, DDR and similar. From a verification point of view these standard interfaces can be an ideal point to extract analysis data from the Device Under Test (DUT), since users can connect to those interfaces using off-the-shelf or internal Verification IPs (VIPs), rather than their own custom code. Starting with a few common SoC verification challenges, such as debug and integration validation, we explain how connecting VIPs to the standard interfaces available can help users overcome those. We then describe the flip side of the coin and point out the problems associated with a large scale deployment of VIPs in a single verification environment, focusing mainly on creation effort, integration effort and performance penalty. Finally we suggest a SystemVerilog/UVM testbench structure that addresses these problems. We show how creation effort can be largely mitigated by automation, and how integration effort can be avoided using separate hierarchies for all VIP code. As for VIP performance penalty, we place it under full user control, by allowing any instantiated VIP to be turned on or off at run time.**

Keywords—Verification IP, standard interfaces, SoC, debug, integration, performance

I. Introduction

As different as SoCs are from one another, their verification process is often quite similar. Since SoCs are made of IPs acquired from different sources and various interconnects that link them together, the very first step is often **integration validation**, i.e. making sure that all of the SoC parts are connected correctly together, and that the various IPs can communicate as they should. As integration validation advances, performance and power validation tests are gradually phased in to make sure the SoC meets performance and power goals under various use cases. This stage is often followed by software tests that test various high level scenarios and how elements play together in synchronization.

With a typical SoC such as the one shown in Fig. 1, even simple integration validation tests can already present a non-trivial challenge. Consider a simple read-modify-write test designed to check that IPs A and B can communicate with one another. If the test fails for some reason, for example, because of a mistake in the configuration of a specific address map segment or an interconnect queue size, debugging it would require that the transaction is tracked down to the point where the failure occurred, across bridges and through interconnects using multiple protocols. If performed at the signal level, such debug could become very time consuming.

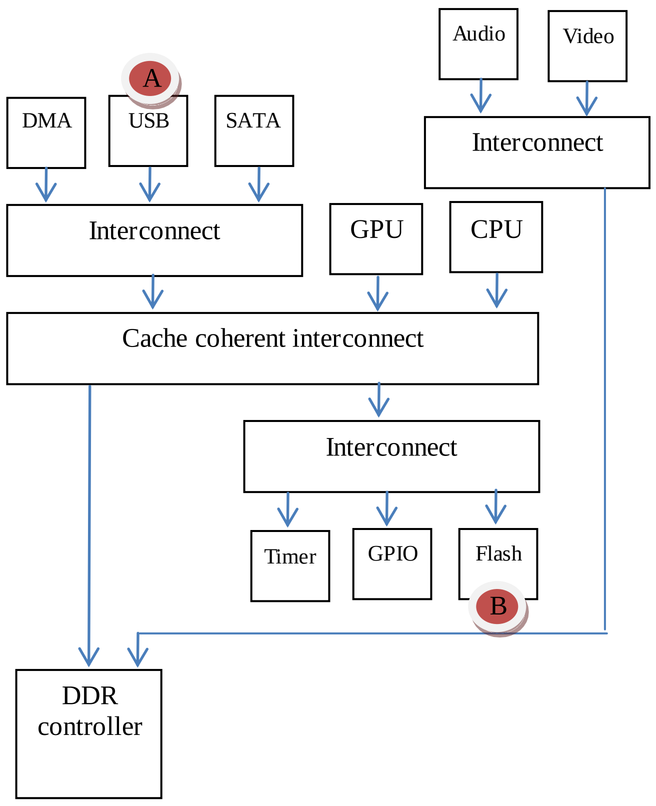


Fig. 1. A typical SoC with various standard intefaces

Verification IPs, or VIPs, could be used along the path to simplify debugging by moving it from signal level to transaction level. VIPs are verification components capable of turning signal level activity into transactions and vice versa. A typical VIP usually includes a monitor that observes signal activity and converts it into transactions, and a driver that takes transactions and sequences of transactions and converts those to signal level activity. Protocol checkers, protocol coverage and data integrity scoreboard are often included in VIPs as well.

Leaving performance aside for a moment, It is clear that the more VIPs are placed on the path from A to B, the easier it would be to debug failing tests, since it would be easier to follow transactions as they make their way through the SoC. Since integration tests normally iterate over each and every possible path, there is an obvious benefit to placing VIPs along every possible path, which practically means almost everywhere. This remains the case as SoC verification gradually moves into performance and power analysis. During performance analysis VIPs can provide latency and bandwidth measurements that can be used to find the root cause of bottlenecks along specific paths. During power analysis, the transactions reported by VIPs can be used for power estimates which are simpler to calculate (though less accurate) than power estimates that are based on signal level activity. The more data points are available for such calculations, the more accurate they become.

In an environment that contains tens to hundreds of interfaces, however, placing VIPs on every interface and along every possible path can be a time consuming achievement. Also, adding VIPs into an existing testbench is often an intrusive process that might result in various testbench errors and problems. Hence, enhancements such as adding a VIP to improve debug on a specific path are usually pushed down the priority list and end up never being executed. Finally, since VIPs often implement complex state machines, mass instantiating them has testbench performance implications that can't be ignored. In the sections that follow we will look at the advantages and the problems of having VIPs instantiated throughout a complex SoC, and at a real-life solution that tries to minimize the difficulties and maximize the benefits.

II. THE ADVANTAGES OF MAXIMIZING VIP USAGE

A. Transaction level debug

VIPs allow users to debug at transaction level rather than at signal level. In complex SoCs, where information could be passed over a few protocols on its way from source to destination, they reduce the need for users to understand the specifics of the protocols in question.

B. Transaction linking

Transaction linking is a debug infrastructure element that gives a good return on investment when working with complex SoCs and refers to anything that tries to tie transactions happening on one interface to transactions happening on another interface. As a simple example of transaction linking, one might think of the capability to match transactions with the same address on two different ports of an AXI interconnect. This simple transaction linking capability could be refined and enhanced as required. For example, transactions could be matched based on address **and** transaction ID rather than based on address alone. Or, the transaction linker could monitor **all the ports** of the interconnect instead of only a single pair. The increased scope and additional accuracy of the transaction linking element come, of course, at the price of further complexity.

Transaction linking is enabled by the above mentioned transaction level debug features of VIPs, as well as by the availability of transaction level information from multiple points in the design. The more VIPs are connected, the closer to one another they will be, and the simpler it becomes tomatch a transaction on one interface to any transactions it triggers on other interfaces. . As verification and design engineers try to track a transaction through the design they could benefit from a library of simple yet powerful queries of the transaction database, that would allow them to cross interconnects quickly, and create customizations where needed.

One growingly important area of application for transaction linking is interconnect performance analysis. As transactions cross interconnects and bridges, they might get into various bottlenecks that impact latency, bandwidth, or both. To detect these bottlenecks it is often useful to look at average transaction duration, and average transaction phase duration, before and after a bridge or an interconnect. The average values are usually obtained from thousands of transactions or even more. Therefore it is not imperative that the linking between incoming and outgoing transactions is always correct, and errors can be tolerated as long as their impact on the average values remains small. It is this error tolerance that allows performance analysis to be based on a simple transaction linking element rather than on a full blown scoreboard that often requires too much effort to code. Another area of application is cache coherent systems where linking transactions that target a single cache line and covering their timing relationship, can provide valuable information on verification quality and efficiency. For more information on the latter application refer to[i].

C. Protocol checking

Since VIPs have to monitor and drive a standard interface, they often come with some protocol checking features built-in. Verification techniques such as Assertion Based Verification (ABV) aim to cut down debug time by placing assertions on interfaces and other strategic points such as FIFOs and arbiters, so that abnormal behavior is flagged out immediately and doesn't have to propagate all the way to an external checker. Using VIP protocol checkers on the various standard interfaces in the design, has a similar effect, since the VIP protocol checkers can detect some errors closer to their source. This implies that the more VIPs are connected to the DUT, the better are the chances of catching various issues early on.

D. Coverage

Since VIPs are used to verify a standard interface, they often come with a coverage model that gives some measure of verification progress. In cases where the VIP does contain a coverage model, it can be useful to reuse it during SoC verification. Although when testing an SoC the coverage numbers on any given interface are not expected to be very high, they are still a valuable metric, both to sanity check the tests being run, and to provide some indication of whether more tests are needed.

E. Portability across platforms

Standard interfaces normally mark the boundary between IPs, and are therefore likely to be preserved on platforms such as emulation, FPGA prototypes, and even on real silicon. Any infrastructure built for debug, coverage, checking and analysis using data from the standard interfaces, is likely to be reusable across all of these, and can allow users some level of comparison between pre-silicon and post-silicon. For example, if **simulation** VIP instances are being used to measure latency and bandwidth on a specific set of interfaces, the signals for this set of interfaces could be logged into a Value Change Dump (VCD) log file during **emulation**, then partially or fully replayed in simulation to get the same latency and bandwidth measurements.

III. THE PROBLEMS OF MAXIMIZING VIP USAGE

Though instantiating VIPs on most or all standard interfaces of a large SoC has significant advantages, it can also be a time consuming process that results in a non-operational verification environment for a significant amount of time, and in a considerable degradation of testbench performance. In this section we take a closer look at the problems associated with VIP instantiation, in order to put together a list of requirements that a solution should fulfill.

A. Creation effort

Instantiating tens or hundreds of VIPs, connecting them right, and maintaining the code created aligned with RTL changes is a repetitive, error prone and time consuming task. Therefore any proposed solution should be required to generate all VIP instantiation and connection code automatically, preferably using the RTL itself as source, and with as little user intervention as possible. Using the RTL itself as source means that the solution has to be able to search the design for various standard interfaces, and pick up specific interface parameters such as bus widths and optional signals automatically.

B. Integration effort

When it comes to VIPs, integration effort can be divided into two major areas:

1) Making the VIP work with the specific standard interface in question – With some standards all that is required in order to understand the traffic on the bus are the bus signals themselves[1].. With other protocols, the user might have to provide some additional information. For example, with AMBA 4 ACE the user should provide the cache line size for a given master, a design time parameter that is not visible from the bus itself, but is still required in order to monitor transactions and perform protocol checking[2]. With other protocols, there might be some run-time configuration that both sides of the interface share and that has to be known before signals on the bus could be deciphered into transactions.

2) Making the VIP work with the testbench – this part includes such tasks as instantiating the VIP in an OVM/UVM/VMM testbench or a testbench that uses none of those, connecting it to the signals via the means available in the HDL or HVL in question, configuring the VIP's behavior, connecting it to analysis components such as coverage, scoreboards and similar, and creating stimuli to run on top of it.

Out of these two areas the more risky and time consuming part is usually #2, since it requires modifications to the user's testbench itself. As always, such modifications might come at the price of various regression failures due to compilation/elaboration/run-time errors introduced by the VIPs or the new integration code. That will be true whether the VIP in use is an off-the-shelf product or internally developed, since even internally developed VIPs need to be integrated into testbenches.

To reduce integration effort and risk it is better if the solution for instantiating VIPs is fully non-intrusive, requiring absolutely no changes to existing DUT and testbench code. It is also preferable that the user maintains the choice of running either with the instantiated VIPs or without them, to guarantee that any false alarms coming from the instantiated VIPs don't end up becoming a blocking point for regression.

C. Performance penalty

VIPs do have a performance cost, which, of course, grows in proportion to the number of VIPs used in the testbench. But, in some cases, users are happy to run slower in order to have the information and visibility they want. For example, while debugging a failing RTL test it is often helpful to preserve some level of visibility of the RTL signals in the design, and a user might choose to keep the ports or internal signals of a specific Verilog module instance visible, even though this might slow down a test[3]. In the same manner, a user might prefer to debug at a high message verbosity mode, even if the additional messages that are being printed out slow the simulation down.

In a similar way, it is likely that while users are debugging a failing test where a transaction needs to be followed over a few standard interfaces, transaction level view of these

[1] For example, AMBA APB, AHB or AXI

[2] For example, in order to know if certain types of operations such as WriteLineUnique are legal. See section C3.1.4 of the AMBA 4 specification, which is publicly available from the following location http://www.arm.com/products/system-ip/amba/amba-open-specifications.php

[3] Some commercial simulators allow users to run with either fully optimized code or partially optimized code that preserves signal visibility in certain areas of interest to enable debug.

interfaces will be required to avoid debug at signal level, and users will be ready to run a bit slower to get it. Note that while users are debugging a test, they are often already running with a few options that make the run slower yet easier to debug. For example, they would run with full visibility of RTL or with full message verbosity as mentioned above, or in GUI mode which also has a negative impact on performance.

While the VIPs along the path that is being debugged will be turned on, the remaining VIPs could be turned off to save their performance cost. For example, the VIPs connected on the ports of some inactive interconnect that the transaction doesn't cross could be disabled since their transaction level information is not needed.

The requirement is therefore to allow users to enable the VIPs that are required for the specific task and test at hand, while keeping the rest of the VIPs disabled. It is also required, that a simulation can run with all VIPs disabled, for example during regression. Once a problem is discovered during regression, a user might want to turn on any subset of VIPs to debug it, trading off performance for visibility.

IV. A SOLUTION OUTLINE

In this section we will first summarize the requirements for a solution, described in the sections above, then move on to give an overview of the solution itself, the assumptions it is based on, and how it is shaped by the requirements. To make the overview more concrete it will be accompanied by a simple example.

A. *Solution requirements*

Table I summarizes the requirements for a solution. As we move through the solution description in the remaining parts of this section, we will be referring to this table and explaining how each of these requirements is addressed.

TABLE I. VIP INSTANTIATION SOLUTION REQUIREMENTS

#	Requirement	Reason
1	It should be possible to generate almost all solution code automatically. Source for generation should be RTL itself.	Avoid creation and maintenance effort
2	It should be possible to have the solution code run with any testbench/DUT without modifications to testbench/DUT code. User could always choose to switch back to original version without and VIPs instantiated.	Avoid integration effort and risk
3	It should be possible to disable all or a subset of the VIPs, preferably at fast turn-around time	Control performance penalty introduced by VIPs
4	It should be possible to easily add support for analysis capabilities such as transaction linkers or scoreboards	

B. *Assumptions about underlying VIP*

As can be seen in table II, the solution described below assumes almost nothing about the VIP being used. The only assumption taken is that the VIP will have some API that allows it to be turned on or off. VIPs that don't have such an API can often be turned off by holding them in reset state.

TABLE II. ASSUMPTIONS TAKEN ABOUT THE UNDERLYING VIP

#	Assumption	Reason
4	VIP can be disabled at run-time, hence not monitoring the bus signals and consuming little or no CPU cycles.	Control performacne penalty introduced by VIPs

The example we will be showing along with our solution description assumes that the VIP has a standard UVM structure. This means that the VIP contains an agent that drives and monitors DUT signals via a virtual interface, and that the VIP can be configured via a configuration object passed through the UVM configuration database. To instantiate the VIP, the user needs to instantiate a SystemVerilog interface and an agent uvm_component, and pass the SystemVerilog interface to the component via the configuration database. If the VIP is structured differently, for example using the two-kingdoms approach[ii], the example code will need small adaptations. . Obviously, it is impossible to create code that will apply to any VIP's structure and API. Instead, the code examples aim to show code that will be easy to modify to fit any VIP's structure and API.

C. *Assumptions about DUT and existing testbench*

Table III lists the assumptions about the DUT RTL code that are required for the solution described below. Note that we're not making any assumptions about the language/methodology of the DUT or existing testbench code. DUT can be VHDL or Verilog, and testbench can be written in any HVL (SystemVerilog,e) or HDL (Verilog, VHDL) and use any methodology, either an off-the-shelf one (OVM/UVM/VMM/eRM), or a proprietary one.

TABLE III. ASSUMPTIONS TAKEN ABOUT DUT CODE

#	Assumption	Reason
1	Standard interface signal names are not modified beyond uppercase/lowercase and additional postfix/prefix	Required in order to detect standard interfaces inside the RTL automatically. Modifications beyond that are considered a bad coding practice and are hard to maintain.

D. *Detailed solution description*

1) *Example*

While going over the proposed solution we will refer to a simple SoC containing one subsystem with AMBA ACE to AXI3 bridge and another subsystem with AXI3 to AHB-lite bridge. This SoC is shown in Fig. 2 below.

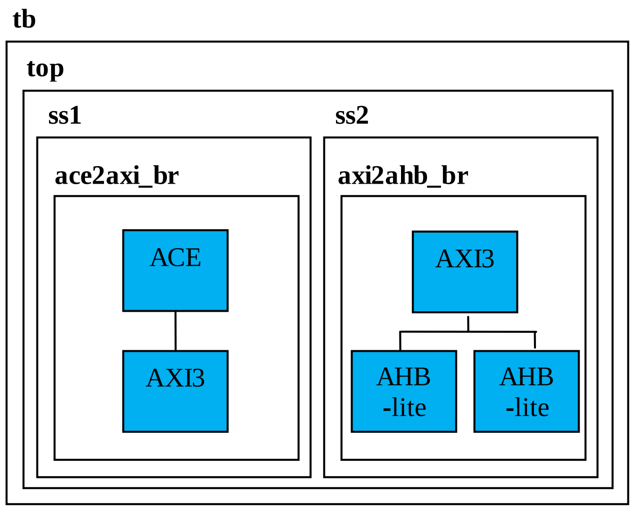


Fig. 2. Example SoC used to demonstrate the solution

2) *Automatic code generation*

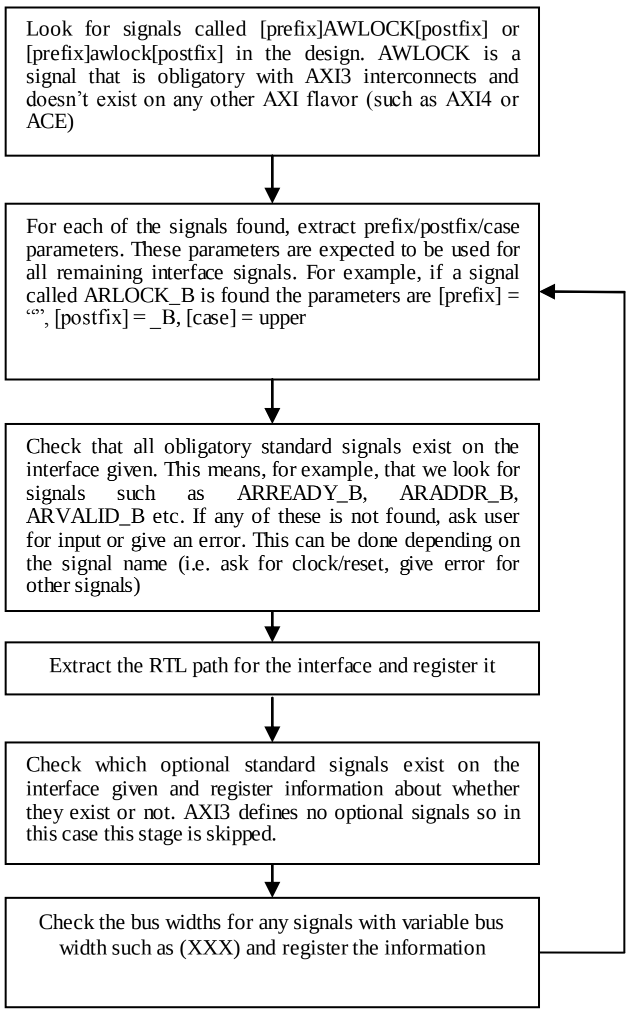
As specified by table II, automatic code generation should be driven by the DUT if possible. If the input to the generator is not automatically created from RTL, it means that the user will have to maintain an additional representation of the DUT in some format, and make sure it is always aligned with changes in DUT RTL. Table IV lists the information that needs to be automatically retrieved from the DUT's RTL, and explains why it is needed. The "name" column will be used to link back from auto-generated code samples to the information in the table.

TABLE IV. INFORMATION THAT NEEDS TO BE AUTOMATICALLY OBTAINED FROM DUT RTL CODE

Information	Name	Reason
Type of standard interface (i.e. AXI, PCIe, USB)	[protocol]	Which VIP should be instantiated?
Hierarchical RTL path of standard interface.	[rtl_path]	Required in order to connect signals
Bus widths where configurable (i.e. on an AXI bus ARADDR width)	[parameters]	Might be required in order to connect signals and configure VIP properly.
Optional signals (i.e. for example AWUSER on an AXI4 bus)	[optional]	If the standard defines some optional signals the generator will need to know whether those exist on the interface or not, in order to create the correct connection code.
Standard interface name	[prefix] [postfix] [case]	In case multiple interfaces of the same kind are defined in the same RTL block, this should allow us to qualify the signal names. We discuss naming in detail below.
Sources for required signals that are not part of the standard or that don't follow standard naming convention (clock/reset are usually good examples)	[additional_signals]	If there is some signal that is required by the standard but is not specified in it, or that isn't named according to the same naming convention as other signals the user needs to provide this information.

To detect standard interfaces within the design, it is possible to rely on the signal names defined by the standard. It is considered a good RTL coding practice that minimizes name collisions and connectivity bugs to avoid modifying standard interface signal names beyond making them lowercase/uppercase or adding prefixes/postfixes to them. Common commercial interconnect IPs such as ARM's NIC, Altera's Avalon, Arteris' FlexNoC and Synopsys' DesignWare, all follow this guideline, so the above mentioned standard interface detection mechanism will work for those. If the code uses custom names that don't match the above naming convention they will require additional manual intervention in the generation process. Additional manual intervention will also be required for the clock/reset signals which are expected to deviate from any names specified by the standard.

The code to detect interfaces and extract the information in table IV from a design can be written in a variety of ways. It could make use of proprietary simulator commands that operate on an elaborated DUT, or be packaged in a DPI or PLI application. Fig. 3 shows the flow for searching for AXI3 interfaces and extracting the information in table IV out of the RTL. User input will be required while running such code in order to specify the connections for signals that don't follow the naming convention, such as clock/reset.


Look for signals called [prefix]AWLOCK[postfix] or [prefix]awlock[postfix] in the design. AWLOCK is a signal that is obligatory with AXI3 interconnects and doesn't exist on any other AXI flavor (such as AXI4 or ACE)

For each of the signals found, extract prefix/postfix/case parameters. These parameters are expected to be used for all remaining interface signals. For example, if a signal called ARLOCK_B is found the parameters are [prefix] = "", [postfix] = _B, [case] = upper

Check that all obligatory standard signals exist on the interface given. This means, for example, that we look for signals such as ARREADY_B, ARADDR_B, ARVALID_B etc. If any of these is not found, ask user for input or give an error. This can be done depending on the signal name (i.e. ask for clock/reset, give error for other signals)

Extract the RTL path for the interface and register it

Check which optional standard signals exist on the interface given and register information about whether they exist or not. AXI3 defines no optional signals so in this case this stage is skipped.

Check the bus widths for any signals with variable bus width such as (XXX) and register the information

Fig. 3. Flow for extracting the information specified in table IV for any AXI3 interfaces that exist in a DUT RTL.

Proper naming is a major aspect of automatic code generation. The generator is required to have a proper naming scheme so that users can identify a given interface in messages, coverage reports, debug windows such as simulation hierarchy viewer, and in any code they add on top of the auto-generated one (for example, code for transaction linking). At a high level, we see the convention below as an intuitive way for unique identification of standard interface within a DUT.

[rtl_path].[prefix]_[protocol+case]_[postfix]

The [protocol] part is required in case there are two interfaces of different types (i.e. AXI and AHB), that don't have a prefix or a postfix attached to signal names. Prefix or postfix could be empty. In case of an uppercase/lowercase variation, it is reflected in the [protocol+case] field. Table V shows the signals found in our example and how a unique identification is created out of them.

TABLE V. UNIQUE IDENTIFICATION OF INTERFACES IN THE EXAMPLE SOC, DERIVED FROM SIGNALS FOUND IN RTL CODE

Signal found	Assigned standard interface unique identification
tb.top.ss1.ace2axi_br.ARSNOOP_A	tb.top.ss1.ace2axi_br.ACE_A
tb.top.ss1.ace2axi_br.ARLOCK_B	tb.top.ss1.ace2axi_br.AXI_B
tb.top.ss2.axi2ahb_br.ARLOCK	tb.top.ss1.ace2axi_br.axi
tb.top.ss2.axi2ahb_br.HADDR	tb.top.ss1.ace2axi_br.ahblite
tb.top.ss2.axi2ahb_br.cfg_HADDR	tb.top.ss1.ace2axi_br.cfg_ahblite

As we go through generated code samples in later sections, we will point out elements that are derived from the information specified in table IV, using square brackets, similar to the way we have specified the unique identification above. We will also use the names defined in table VI for common combinations.

TABLE VI. COMMON SHORTHANDS

Shorthand name	Assigned standard interface unique ID
[unique_id]	[rtl_path].[prefix]_[protocol+case]_[postfix]
[local_id]	[prefix]_[protocol+case]_[postfix]

Table VII gives an example of the information extracted from the RTL for the port tb.top.ss1.ace2axi_br.axi

TABLE VII. INFORMATION EXTRACTED OUT OF THE RTL OF THE EXAMPLE SOC FOR PORT TB.TOP.SS1.ACE2AXI_BR.AXI

Parameter name	Value for tb.top.ss1.ace2axi_br.axi
[protocol]	Axi
[rtl_path]	tb.top.ss1.ace2axi_br

[parameters]	ADDR_WIDTH=32
	RDATA_WIDTH=64
	WDATA_WIDTH=64
	ID_WIDTH=11

[optional]	
[prefix] [postfix] [case]	"" "_B" Upper
[additional_signals]	ACLK <= sys_clk ARESETn <= sys_reset

3) VIP instantiation

As mentioned in section IV.b we assume that VIPs are written in SystemVerilog/UVM and connect to the DUT via

SystemVerilog interfaces which are then passed to the UVM classes via a virtual interface handles[4], as shown in Fig. 4. This implies that two parts have to be instantiated – the SystemVerilog interface and the VIP agent uvm_component which in turn will instantiate all other parts of the VIP.

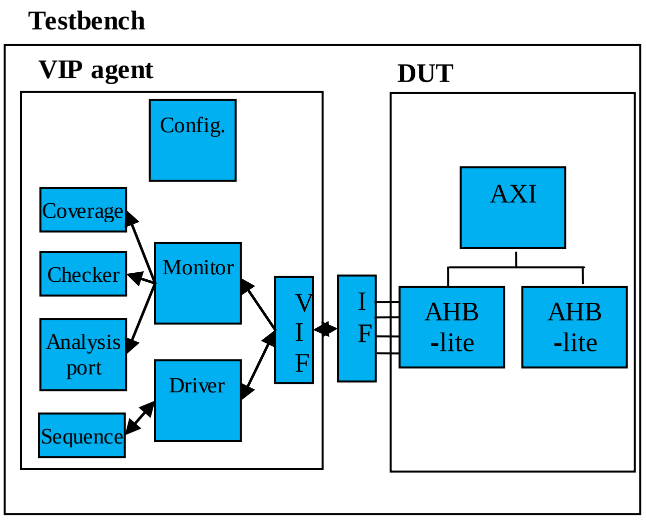


Fig. 4. Assumed SystemVerilog/UVM VIP structure

To create legal SystemVerilog instance names that are guaranteed not to collide with each other, they need to be derived from [unique_id]. There are a few different ways to do that, but the simplest and most intuitive way we found is to replicate the design hierarchy using Verilog modules for the static part and UVM components for the dynamic part.

Fig. 5 shows the code that is generated for the instantiation of the SystemVerilog interfaces required by our example. Note that the module types are unimportant and are therefore just serially numbered. To avoid having to instantiate a meaningless top level module such as "container_module_0", we can add another layer of hierarchy on top, and assign it some more meaningful name. This is shown at the bottom of the code, where "container_module_0" is instantiated from a module of type vip_instantiation_top. The module instance names reflect the various sub-elements of [rtl_path] and put together they reflect the full RTL path. The VIP instance names are simply [local_id]. The code for the UVM hierarchy is very similar, so it can be skipped here.

According to requirement #2 in table I no code modifications should be required in any existing DUT or testbench code to integrate the code that instantiates the VIPs. With the static hierarchy, it is possible to simply instantiate it under a separate top. If any existing testbench code doesn't run UVM then the UVM hierarchy will be instantiated as a separate top anyhow under uvm_package::uvm_test_top.top_levels[0][5]. If some of the existing testbench is UVM based it is possible to create another top level under uvm_package::uvm_test_top.top_levels, by simply creating the top component directly. The vip_instantiation_top module defined in Fig. 5 can be used to create this top component.

```
// module types are arbitrary.
// Since all modules share a namespace,
// it is hard to find a unique, meaningful,
// collision-free name for them.

module container_module_0();
    // module name "tb" is extracted from [rtl_path]
    container_module_1 tb();
endmodule

module container_module_1();
    // module name "top" is extracted from [rtl_path]
    container_module_2 top();
endmodule

module container_module_2();
    // module names "ss1", "ss2" are extracted from [rtl_path]
    container_module_3 ss1();
    container_module_5 ss2();
endmodule

module container_module_3();
    container_module_4 ace2axi_br();
endmodule

module container_module_5();
    container_module_6 axi2ahb_br();
endmodule

module container_module_4();
    // VIP interfaces instance names are [local_id]
    ace_vip_if ACE_A();
    axi_vip_if AXI_B();
endmodule

module container_module_6();
```

[4] This assumption was not taken because of any conceptual necessity, but rather because such VIP structure is relatively common.

[5] top_levels is part of the uvm_root class. See $UVM_HOME/base/uvm_root.svh line 95

```
        axi_vip_if axi();
        ahblite_vip_if ahblite();
        ahblite_vip_if cfg_ahblite();
endmodule


// top level module with a meaningful name
module vip_instantiation_top();
        import uvm_pkg::*;
        container_module_0 hierarchy();
        central_test vip_instantiation_test;
        initial
          vip_instantiation_test =
central_test::type_id::create("vip_instantiation_test", null);

endmodule
```

Fig. 5. Auto-generated code for the instantiaion of the SystemVerilog interfaces partaining to the VIP

4) VIP connection

To connect the VIP to the design, the VIP’s interface signals should be assigned by the corresponding RTL signals. Fig. 8 below shows an example of how this is done. Note that the hierarchical path to the VIP interface instance is identical to the RTL path, except that it is preceded by the separate top level name *vip_instantiation_top.hierarchy*. This corresponds to the code shown in Fig. 5. . The module that contains the connection code (i.e. signal_connections() in Fig. 8), can either be instantiated as another top level module, or be instantiated under the VIP instantiation top module (i.e. vip_instantiation_top() at the bottom of Fig. 5). Fig. 7 shows what our testbench looks like when the signal_connections() module is instantiated as another top.

```
     module signal_connections();
     // derived from [additional_signals] specified explicitly by the user
     assign
vip_instantiation_top.hierarchy.tb.top.ss2.axi2ahb_br.axi.ACLK          =
tb.top.ss2.axi2ahb_br.sys_clk;
     assign
vip_instantiation_top.hierarchy.tb.top.ss2.axi2ahb_br.axi.ARESETn       =
~tb.top.ss2.axi2ahb_br.sys_reset;
     // both LHS and RHS are derived from [rtl_path]
     assign
vip_instantiation_top.hierarchy.tb.top.ss2.axi2ahb_br.axi.AWVALID       =
tb.top.ss2.axi2ahb_br.awvalid;
     assign
vip_instantiation_top.hierarchy.tb.top.ss2.axi2ahb_br.axi.AWADDR        =
tb.top.ss2.axi2ahb_br.awaddr;
     assign
vip_instantiation_top.hierarchy.tb.top.ss2.axi2ahb_br.axi.AWLEN         =
tb.top.ss2.axi2ahb_br.awlen;
     assign
vip_instantiation_top.hierarchy.tb.top.ss2.axi2ahb_br.axi.AWSIZE        =
tb.top.ss2.axi2ahb_br.awsize;
     assign
vip_instantiation_top.hierarchy.tb.top.ss2.axi2ahb_br.axi.AWBURST       =
tb.top.ss2.axi2ahb_br.awburst;
     // more signal assignments...


endmodule
```

Fig. 6. Signal connection for one of the VIPs

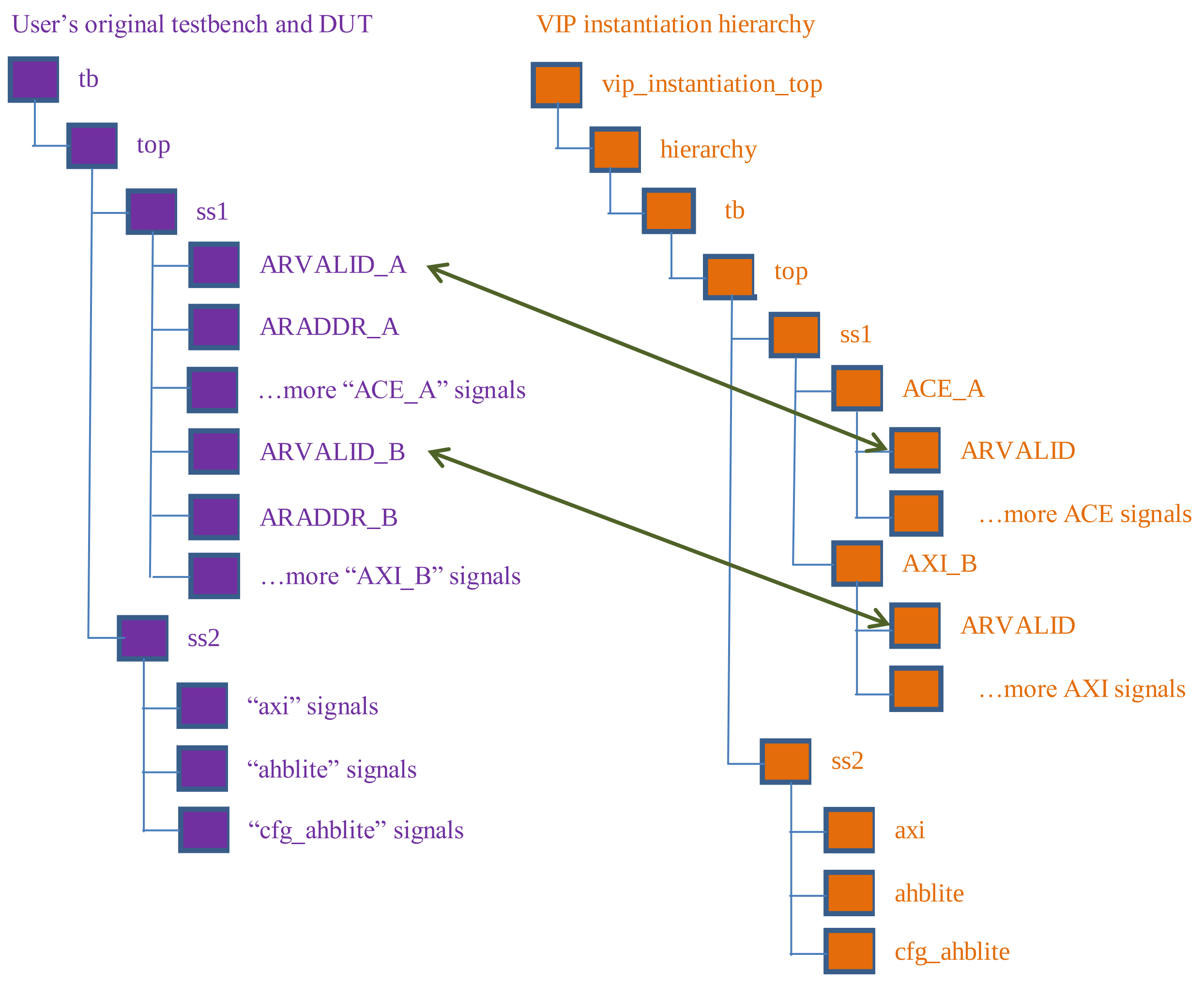


Fig. 7. An overview of the user's testbench hierarchy and the parts that are added for VIP instantiation. Purple hierarchy shows existing user environment, and orange hierarchy shows the parts added for VIP instantiations. UVM hierarchy is not shown here but is very similar in structure.

5) VIP configuration

According to requirement #3 in table II it should be possible to disable a subset of the instantiated VIPs or all of them. This allows users to enable only VIPs that are monitoring interfaces they're interested in. To support this requirement we can trigger a call to the API that turns the VIP on or off according to a UVM configuration parameter that is specified either from a test or from the command line. This would require the UVM components containing the VIP agents to contain the code shown in Fig. 9.

```
// Classes below are used to configure the VIPs
class vip_config extends uvm_object;
  `uvm_object_utils(vip_config)

  on_off_t on_off;

  function new (string name);
    super.new(name);
  endfunction
endclass

class ahblite_config extends vip_config;
  `uvm_object_utils(ahblite_config)

  // AHB-lite parameters for determining bus widths
  int ADDR_WIDTH;
  int RDATA_WIDTH;
  int WDATA_WIDTH;

  function new (string name);
    super.new(name);
  endfunction
endclass

//more configuration classes skipped…

//This is the component that contains VIP instantiations for all VIPs found at tb.top.ss2.axi2ahb_br
class container_component_5 extends uvm_component;
  `uvm_component_utils(container_component_5)

  function new(string name, uvm_component parent = null);
    super.new(name, parent);
  endfunction

  // A pointer to the test used to allow users do some additional configuration
  central_test test;

  // code for tb.top.ss2.axi2ahb_br.cfg_ahblite interface. same code exists for all other interfaces
  // field names are derived from [local_id]=cfg_ahblite to prevent collisions with other interfaces in the same block
  on_off_t cfg_ahblite_on_off = Off;
  ahblite_agent cfg_ahblite_agent;
  ahblite_config cfg_ahblite_config;
  virtual ahblite_if cfg_ahblite_vif;

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    // get a pointer to central test file to allow test to do some additional configuration
    uvm_config_db #(central_test)::get(this, "",  "central_test_ptr", test);

    begin
      reg signed [4095:0] tmp;

      // code for tb.top.ss2.axi2ahb_br.cfg_ahblite interface, same code exists for all other interfaces
      // all names are derived from [local_id]

      // get the virtual interface connected to the DUT signals
      if (!uvm_config_db #(virtual axi_if)::get(this, "",  "cfg_ahblite", cfg_ahblite_vif))
        `uvm_error("build_phase", "couldn't find interface pointer in config db for cfg_ahblite");


      uvm_config_db #(reg signed [4095:0])::get(this, "", "cfg_ahblite_on_off", tmp);
      cfg_ahblite_on_off = on_off_t'(tmp);

      cfg_ahblite_agent = axi_agent::type_id::create("cfg_ahblite_agent", this);
      cfg_ahblite_agent.vif = cfg_ahblite_vif;

      cfg_ahblite_config = axi_config::type_id::create("cfg_ahblite_config");
      cfg_ahblite_config.on_off = cfg_ahblite_on_off;

      test.configure_vips("tb.top.ss2.axi2ahb_br.cfg_ahblite",
cfg_ahblite_conifg);


       cfg_ahblite_agent.set_config(cfg_ahblite_config);
      end


     //code for more interfaces skipped here…
Endclass
```

Fig. 8. Adding support for fetching configuration from UVM's configuration database for the VIP container UVM components

Setting the on/off configuration and other configuration parameters could be done from a central test file. For example, if the VIP has configuration parameters for setting bus widths, they could be configured from the central test file in a way that is similar to the one shown in Fig. 10. As shown in Fig. 9 all VIPs ask the test to configure them during UVM's build_phase(), by calling the test's configure_vips() function and identifying themselves by [unique_id].

```
    class central_test extends uvm_test;
     `uvm_component_utils(central_test)


     container_component_0 hierarchy;


     function new(string name, uvm_component parent = null);
      super.new(name, parent);
     endfunction


     function void build_phase(uvm_phase phase);
      super.build_phase(phase);


      hierarchy = container_component_0::type_id::create("hierarchy",
this);


      // pointer sent to all children so they can call central configuration
function

      uvm_config_db #(central_test)::set(this, "*", "central_test_ptr",
this);
     endfunction


     // function is virtual so it can be overridden by user if more
configuration is needed
    // This function configures the VIPs and can be overridden by the
user to add/override some of the configuration parameters
    virtual function void configure_vips(string full_name, vip_config
cfg);
      case(full_name)


        "tb.top.ss2.axi2ahb_br.cfg_ahblite":
         begin
          ahblite_conifg local_cfg = ahblite_config'(cfg);


          // setting parameters according to the [parameters] value
derived from the RTL
          local_cfg.ADDR_WIDTH = 32;
          local_cfg.RDATA_WIDTH = 64;
          local_cfg.WDATA_WIDTH = 64;


          //more user configuration goes here
         end


        // More interface unique IDs
      endcase
     endfunction
    endclass
```

Fig. 9. Code for a central test file that allows each VIP to be configured according to its unique identifcation

The on/off configuration parameter can be set from the test in the same way as ADDR_WIDTH or via the command line, which is useful to cut turn-around time. To do so, the user is required to add a plusarg of the following format, derived from [unique_id] and [local_id].

+uvm_set_config_int=uvm_test_top.vip_instantiation_test.hierarchy.tb.top.ss2.axi2ahb_br,cfg_ahblite_on_off,1

A file containing such plusargs for all interfaces can also be automatically generated, thus enabling users to turn on or off any VIP they wish before rerunning a simulation.

1) *Transaction linking*

Transaction linking infrastructure can be created in a similar way to the central test file. At a high level all VIPs transaction analysis ports are connected to one central element. When the VIPs report a transaction they send it together with their [unique_id], so that the transaction linker can know which interface the transaction is coming from. The users extend a transaction linker base class and override a virtual method to create the links required. In the example in Fig. 11 we show how a user might extend the transaction linker to link transactions going from the AXI ports to the AHB-lite ports of axi2ahb_br. The user's transaction linker assumes that the AXI transactions end after any AHB-lite transactions triggered by them. Once an AXI transaction is finished the user's linker searches for AHB-lite transactions with the same address and within the same time window.

```
class vip_transaction extends uvm_transaction;
  time start_time;
  time end_time;
  longint unsigned addr;

  // more fields...
endclass

class specific_linker extends transaction_linker;
  `uvm_component_utils(transaction_linker)

  function new(string name, uvm_component parent = null);
    super.new(name, parent);
  endfunction

  // store transactions coming from each of the ports
  vip_transaction port_transactions[string][$];

  // the reporter_name parameter is the [unique_id] of the VIP, and
informs the transaction linker where a specific transaction is coming
from.
  virtual function void transaction_reported(string reporter_name,
vip_transaction txn);
    //The transaction linker decides what to do with a transaction
based on the [unique_id] of the VIP that sent it
    case(reporter_name)

      "tb.top.ss2.axi2ahb_br.axi":
        begin
          int parent_handle;
          string ahblite_uid[$] = '{"tb.top.ss2.axi2ahb_br.cfg_ahblite",
"tb.top.ss2.axi2ahb_br.ahblite"};
          axi_transaction axi_txn = axi_transaction'(txn);

          // create parent transaction using UVM transaction recording
API
          parent_handle = begin_tr(axi_txn,
"axi2ahb_br","","",txn.start_time);

          // log some AXI specific fiels using axi_txn (not shown here)

          // find child transactions
          foreach (ahblite_uid[i]) begin
            ahblite_transaction child_txn;
            int child_handle;

            // create child transactions using UVM transaction recording
API
            // transactions are matched according to address and start
time
            //child_txn =
ahblite_transaction'(port_transactions[ahblite_uid[i]].find_first() with
((item.addr = axi_txn.addr) && (item.start_time >=
axi_txn.start_time))[0]);
            child_handle = begin_child_tr(child_txn, parent_handle,
"axi2ahb_br", "", "", child_txn.start_time);

            // log some ahblite specific fields using child_txn (not shown
here)

            // release the handle
            end_tr(child_txn, child_txn.end_time, 1);
          end

          end_tr(axi_txn, $time, 1);
        end

      "tb.top.ss2.axi2ahb_br.cfg_ahblite":
        begin
          // AHB-lite transactions simply add themselves to the queue
since parent-child relationship are created when AXI transaction arrives
          port_transactions["tb.top.ss2.axi2ahb_br.cfg_ahblite"][$+1] =
txn;

        end

      "tb.top.ss2.axi2ahb_br.ahblite":
        begin
          port_transactions["tb.top.ss2.axi2ahb_br.ahblite"][$+1] = txn;
        end

      // More interface unique IDs
    endcase
  endfunction
endclass
```

Fig. 10. Example code for a transaction linker created by the user. The transaction linker links transactions on the AXI port of axi2ahb_br to transactions on the AHB-lite ports based on address. Using UVM's transaction recording API ensures that this code is simulator independent, and linked Abstract BFMs Outshine Virtual Interfaces for Advance SystemVerilog Testbenches – Rich, David and Bromley Jonathan [transactions will be recorded by any simulator.

V. SUMMARY

Although VIPs give significant added value in debug, protocol checking, coverage, and by enabling various sorts of end-to-end analysis, they're not without issues. The complex state machines they implement have a cost in performance, and deploying them in large numbers requires a good amount of investment in the creation and maintenance of testbench code. In addition, their integration into existing testbenches often results in making those testbenches non-operational for some time. These difficulties require very careful consideration prior to deploying VIPs.

In this paper we have tried to present a solution that would address the above mentioned problems and allow for a less restricted use of VIPs. We have shown that code for instantiating, connecting and configuring VIPs could be created automatically, directly from RTL, and be placed in isolated hierarchies so that no integration risk needs to be taken. And we have suggested a simple way to control the performance impact by turning VIPs on or off at run time according to the task at hand. Though the example provided was making some assumptions about VIP API, we believe it was provided in a general enough way to make porting it to other VIP APIs a feasible task.

---

[i] Meyer, A. & Hunter, A (Feb 2014) "So you think you have good stimulus: System-level distributed metrics analysis and results", Design & Verification Conference, San Jose , CA.

[ii] Bromley, J. & Rich, D (Feb 2008) "Abstract BFMs Outshine Virtual Interfaces for Advanced SystemVerilog Testbenches", Design & Verification Conference, San Jose, CA.