Novel Method To Speed-Up UVM Testbench Development

Prashantkumar Ravindra (prashantkumar.ravindra@analog.com), Analog Devices (India)
Barry Briscoe (Barry.Briscoe@analog.com), Analog Devices (Ireland)
Miguel Castillo (Miguel.Castillo@analog.com), Analog Devices (Philippines)
Nimay Shah (Nimay.Shah@analog.com), Analog Devices (US)

***Abstract*-Verification IPs are the building blocks of UVM testbenches, needed for Metric Driven Verification of complex designs. UVM Testbench generators can instantly create a basic UVM testbench template from scratch, but the VIP integration must be done manually. This makes the testbench development activity difficult and time-consuming. Automating VIP integration is the solution, but this is not straightforward due to the lack of an industry-wide standard to exchange VIP metadata. In this paper, the authors present a non-proprietary VIP metadata template that can enable this automation via a Testbench Generator. This paper will further highlight how, without restricting the creativity of VIP developers, multiple vendor VIP titles have been successfully integrated into the ADI's UVM Testbench generator, with the help of this metadata. This method has enabled the DV engineers to instantly create a ready-to-simulate sophisticated UVM TB from scratch, reducing the efforts from weeks to minutes.**

I. INTRODUCTION

It has been over a decade since the release of the first version of the Universal Verification Methodology (UVM). Over the years there have been multiple stable releases of UVM, and the adoption has been steadily growing with every passing day. The advent of Battery Electric Vehicles (EV), Artificial Intelligence (AI), 5G, Industry 4.0, Smart Gadgets and Wearables, Advanced Driver Assistance System (ADAS), Drones, etc. have resulted in a multifold increase in the verification complexity of the Application Specific Integrated Circuits (ASIC) and System on Chips (SoC) used in these applications.

On top of this, Time to Revenue (TTR) is shrinking and to meet this timeline, first-pass silicon is crucial. Hence, Design Verification (DV) teams leave no stone unturned when it comes to functional verification of the design. For thorough functional verification, a sophisticated testbench (TB) is necessary and hence developing a complex, yet flexible, UVM testbench is a critical requirement.

Figure 1 shows a typical UVM testbench development flow. Verification Intellectual Property (VIP) and Testbench Generator are two of the important pieces needed for any UVM testbench development.

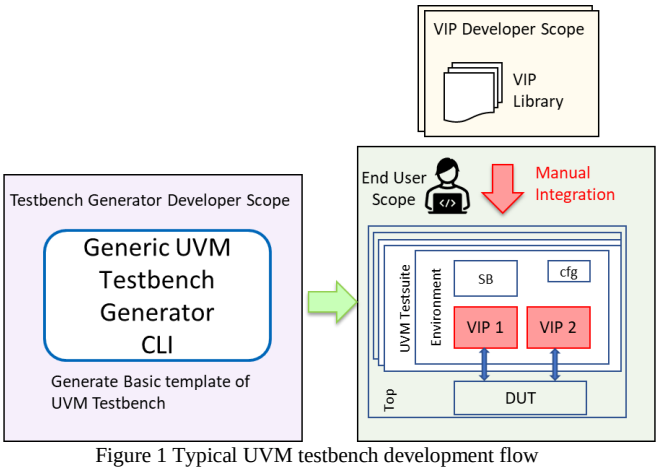

Figure 1 Typical UVM testbench development flow

The concept of a Graphical User Interface (GUI) or a Command Line Interface (CLI) based UVM Testbench Generator has been around for a while. Semiconductor design companies develop generators on their own or use ones from Electronic Design Automation (EDA) vendors. Homegrown generators are fine-tuned to be compatible with the

company's overall DV ecosystem. On the other hand, vendor solutions are fine-tuned to work best with their respective DV ecosystem offerings such as VIPs, Simulators, etc. Irrespective of the source, the Testbench Generator is undoubtedly the fastest way to bring up the UVM testbench in comparison to manually writing every piece of UVM testbench code from scratch.

VIPs are typically the building blocks of any standard UVM testbench. It is a widespread practice to either develop UVM based VIPs in-house, by central engineering teams, or to procure from vendors. Especially for protocols/interfaces that are based on industry standards defined by Institute of Electrical & Electronics Engineers (IEEE), Joint Electron Device Engineering Council (JEDEC), Mobile Industry Processor Interface (MIPI), Video Electronics Standards Association (VESA) and others, vendor VIPs offer benefits such as zero development time and cost, high quality, dedicated support. However, vendor VIPs are typically encrypted and thus it takes considerable time to integrate these into user testbench and to achieve first test-pass.

Parking the idea of listing the advantages of using Testbench Generator and VIPs, which are well known to the DV community, the authors would like to highlight the missing feature of "Automating Vendor VIP integration into a UVM testbench via UVM Testbench Generator" and how to address it.

II. Challenges

It is interesting (and surprising) that there is no defined standard to exchange UVM VIP integration data to ease/automate the integration. Vendors have ended up with their proprietary ways of enabling integration into end-users testbench. VIP vendors ship example testbenches, user guides and custom tools to ease integration. However, all these ways have challenges of their own.

For example

(1) Vendor TB generation tools do not support VIPs from other sources (in-house or third party).

(2) Porting VIP from the shipped example to the user testbench is easier said than done.

III. Existing Flow

At Analog Devices Inc (ADI), with the in-house developed advanced UVM Testbench Generator, DV teams can develop new UVM testbenches (and test cases) with the required in-house VIPs, project-specific UVCs, and reusable block-level environments instantly. The UVM Testbench Generator is a critical piece of the ADI's Unified Design Verification solution and hence testbench, so generated, is out-of-the-box compliant with the ADI's DV ecosystem (tools, flows and methods). These are compile-clean and ready-to-simulate testbenches, minus the vendor VIPs which are to be integrated manually, post the testbench generation.

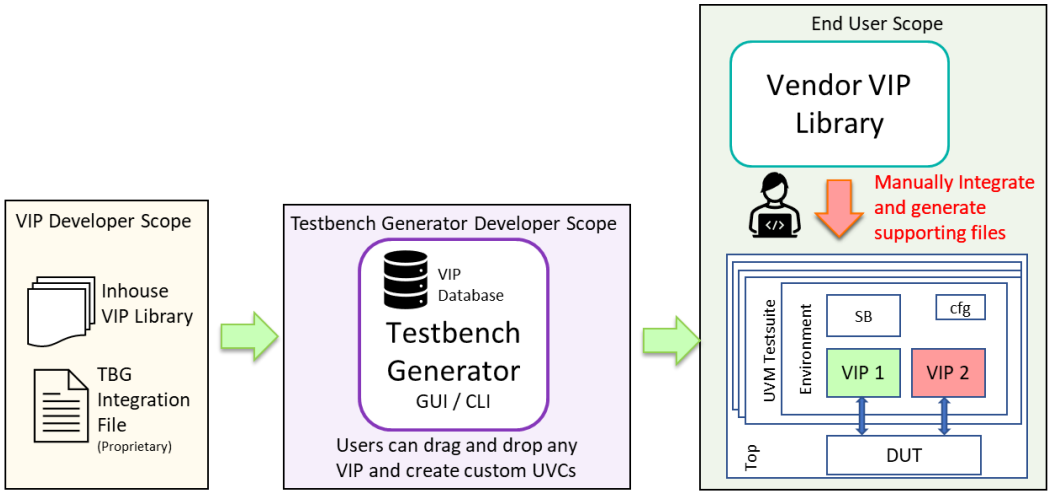

Figure 2 UVM testbench development flow using Testbench Generator and manual integration of vendor VIPs

As shown in Figure 2, the biggest challenge in the existing flow is the integration of complex vendor VIPs, as it is a manual and cumbersome process. VIP-1 (green), representing in-house developed VIP is added via Testbench Generator flow and VIP-2 (red), representing vendor VIP, is added manually.

Key challenges in the existing flow

1) *Stringent timelines – Testbench should be ready ASAP to make way for the verification.*
2) *Learning curve - VIP structure, configs, sequences, coverage, checkers, interfaces, etc.*
3) *For complex protocols, it is not straightforward to port example cases to a user's testbench.*
4) *VIP integration challenges- Find packages, class names, generate libraries, extract models, etc.*
5) *Manual VIP integration leads to issues which are difficult and time-consuming to root cause.*
6) *Testbench architecture updates lead to re-do of manual work.*

This flow is common across the industry and hence these challenges resonate with the entire DV community. Even the well-known Wilson Research Group study shows a similar trend. As shown in Figure 3, even with all the advanced solutions developed by the EDA vendors, 15% of the ASIC verification engineers time is spent in testbench development [1], which the authors believe can be significantly minimized with "automation".

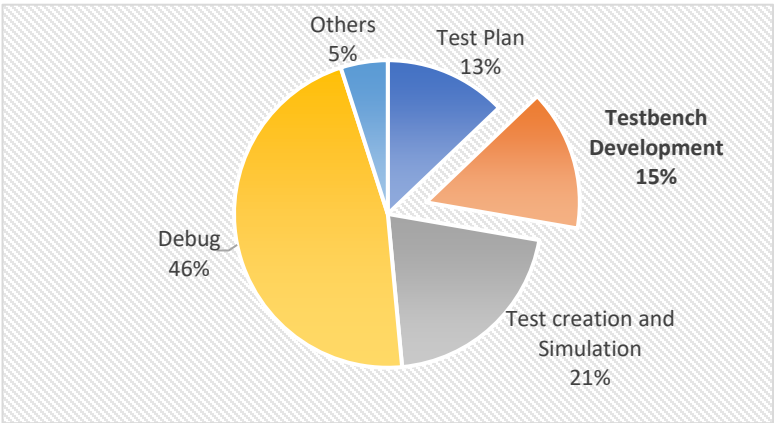

Figure 3 Time spent by ASIC Verification Engineers

IV. Proposed Solution

The integration of the vendor VIP portfolio into the Testbench Generator can address the challenges listed in the earlier section to a significant extent. But the question is *How*? *How to enable this automation?*

Over the past couple of years, authors collaborated with multiple VIP vendors and established that integration of (any) vendor VIP into a UVM testbench can be automated via generic Testbench Generator (developed by another vendor), if the necessary VIP integration details (VIP Metadata) is available in the pre-defined standard format (VIP Metadata template).

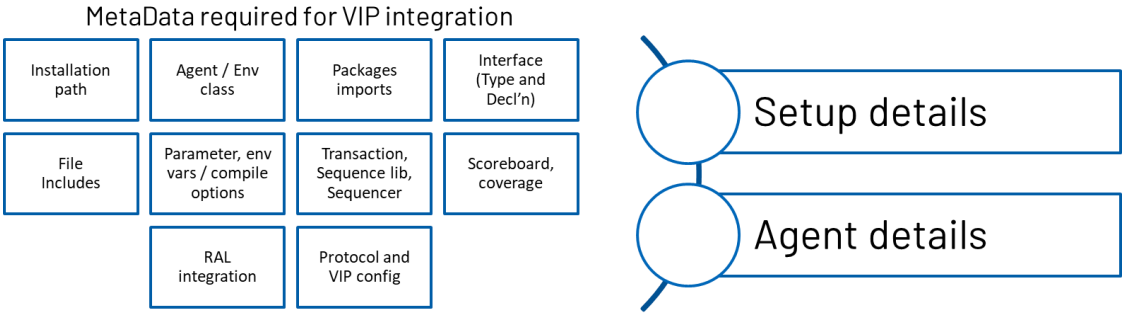

Figure 4 VIP metadata required for VIP integration

Figure 4 shows the metadata needed to integrate a standard UVM VIP into a UVM testbench. Close inspection reveals, this is the same information DV engineer uses during manual integration of the VIP into a testbench.

The basic idea here is to find all the necessary data needed for VIP integration and represent it in a standard format, which can be interpreted by any Testbench Generator. The Testbench Generator can then extract the metadata and inject the processed VIP code in the specific locations of the testbench based on element-attribute combination in the VIP Metadata File.

Figure 5 shows the representation of the VIP metadata template. The metadata template even has qualifier-attributes (Simulator, Sim_Arch and UVM_Ver to support multiple simulators, architectures and UVM versions, respectively. More elements and attributes can be added to support other vendors and testbench types (Ex: emulation, PSS).

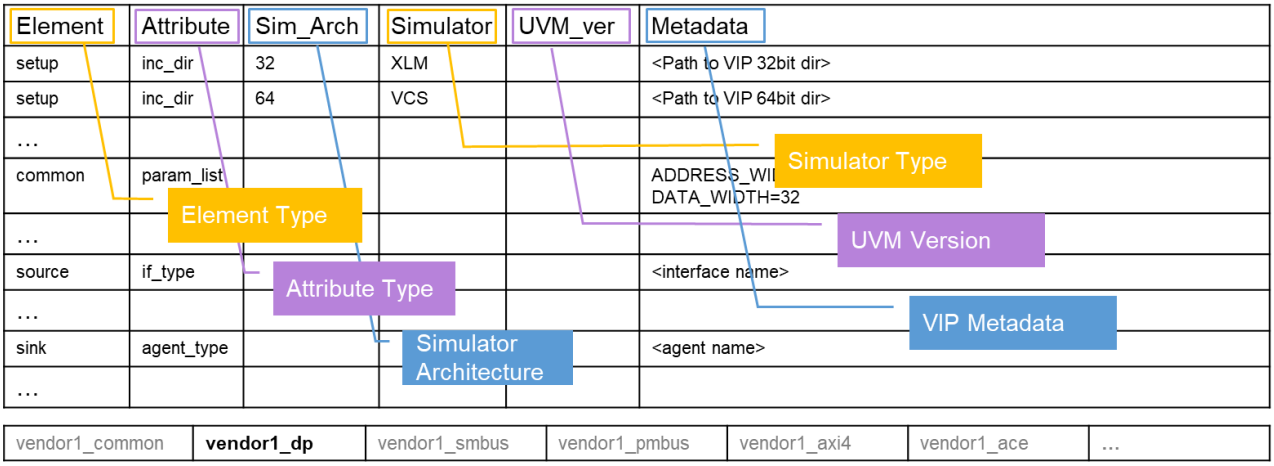

Figure 5. Sample representation of VIP metadata template

Through this paper, the authors are not only advocating the need of a well-defined standard to aid the automation of VIP integration into UVM testbench, but also have defined one (which is non-proprietary) and enabled the automation which can benefit the greater DV community.

Let us understand the key elements-attributes supported in the current version of VIP metadata template.

As shown in Figure 5, “Element” supports the following types:
1. Setup
2. Common
3. <Agent>

Note: <Agent> is of variable-type and shall be replaced with the name of agent (Ex: source, sink, tx, rx, main, sub). There could be one or more <Agent>.

1. Setup

This type of element corresponds to the requirement of “Setup Details” mentioned in Figure 4. The attributes associated with this element captures data needed to setup and prepare the user testbench environment, including all VIP supporting models and library files. Key attributes of this element are listed in Table 1.

Table 1. Key attributes of “Setup Details”

Attribute	Description	Reference
vendor_name	Name to identify the vendor in TB generator	<vendor name>
vip_name	Name to identify VIP in TB generator	<protocol>:<version>
env_var	Set required environment variables	setenv VIP_LIB_PATH ${TB_ROOT}/agents/<vendor>/<vip>
src_path	Path to the location of the user-editable VIP source code that needs to be copied to the generated TB	${VIP_ROOT}/…./
pre_comp	Perform required operation/ execute script before compilation	source $VIP_LIB_PATH/vip_comp.csh

pre_sim	Perform required operation/ execute script before simulation	source $VIP_LIB_PATH/vip_sim.csh
pre_comp_sim	Perform required operation/ execute script before compilation and simulation	source $VIP_LIB_PATH/vip_all.csh
comp_opt	Compilation options	-define VENDOR_PROTOCOL
comp_file	Files to be compiled	${TB_ROOT}/agents/protocol/x/y/z
inc_dir	Directories required for compilation	${TB_ROOT}/agents/protocol/x/y/z
sim_opt	Run time options for simulator	-pli ${VIP_ROOT}/somefile.so

2. Common & <Agent>

This type of elements corresponds to the requirement of "Agent Details" mentioned in Figure 4. If "Common" type is used then the attribute is considered as common for all the agents supported by the VIP, else if <Agent> element is used then its applicable only to that <Agent>. This section lists attributes that capture the data needed to integrate the VIP into the user testbench environment. Key attributes of these elements are listed in Table 2.

Table 2. Key attributes of "Agent Details"

Attribute	Description	Reference
pkg_import	VIP env/agent package to be imported in TB env scope Scope: TB env package	vendor_protocol_pkg
pkg_import_global	VIP env/agent package to be imported in global scope Scope: Global	vip_common_pkg
inc_file	Files to be included in TB environment package Scope: TB env package	vip_protocol_file.sv
inc_file_global	Files to be included at the global scope Scope: Global Useful to manage non-UVM file compilation, which can be done outside TB pkg declaration or in simulator command file	vip_protocol_interface.sv
add_tbpkg_code	Custom code required for VIP compilation This code will be added in the TB environment package scope prior to importing other packages Typically used for type or vendor specific forward-declaration/parameters Note: If used in <vendor>_common sheet, its vendor-specific else it is type-specific	typedef class vip_example_class;
add_tbpkg_code_global	Custom code required for VIP compilation This code will be added in the global scope Typically used for type or vendor specific forward-declaration/parameters Note: If used in <vendor>_common sheet, its vendor-specific else it is type-specific	typedef class vip_example_class;
param_list	List of parameters used in the TB env scope Comma separated list should have the parameter name and its value Suggestion: Recommended to use only one parameter per line Scope: TB Env package	ADDRESS_WIDTH=32, DATA_WIDTH=32
param_list_global	List of parameters used in the global scope Comma separated list should have the parameter name and its value Suggestion: Recommended to use only one parameter per line Scope: Global	ADDRESS_WIDTH=32, DATA_WIDTH=32
macro_list_global	List of compile macros used in the global scope Suggestion: Recommended to use only one macro per line Scope: Global	VIP_SAMPLE_EN_MACRO

build_phase_tbenv_code	Custom code can be added in the build_phase of TB env (Ex: overrides)	set_type_override_by_type(vip_driver::get_type(),vip_usr_driver::get_type());
eoe_phase_tbenv_code	Custom code to be added in the end of elaboration phase of the TB environment class	<code>
sos_phase_tbenv_code	Custom code to be added in the start of simulation phase of the TB environment class	<code>
agent_type	Agent type to instantiate and create the VIP instance in the TB environment	vip_protocol_agent
if_type	Interface type	vip_protocol_interface
sig_list	List of VIP interface signals available for connection with DUT ports	input sigA, output [1:0] sigB, inout sigC
cfg_type	VIP configuration class type	cfg_type
cfg_set_id	ID to get/set the cfg class handle via config_db	cfg_set_id
cfg_set_hier	Hierarchy path in which the VIP config handle must be accessible via config_db set for the “cfg” class Default: Typical config_db set statement with id as "cfg"	${inst}.*
cfg_vars	VIP configuration class variables to be available in generator GUI Syntax: <field> = <#value1,value2,value3#> <field> = <[value1:value10]> <field> = <value> Note: Relative to top VIP config instance	vip_protocol_kind = <#xkind,ykind#>;
sqr_type	VIP agent sequencer type Default: <vip_agent_type>_sqr	vip_protocol_sqr
sqr_conn	Setting the handle of the VIP sequencer in the environment virtual sequencer. (connect_phase) Note: TBG adds the typical connection statement if this attribute is not used. This attribute is mandatory if custom cfg_decl is used	sub_agent.sequencer
tr_type	Transaction class type	vip_xtn
sb_port	Used to Connect monitor analysis ports to scoreboard implementation ports	vip_protocol_monitor.x_dir_prt

Apart from the above listed attributes, there are others related to sequences, RAL (adapt, pred), virt_sqr, etc. that enable VIP integration. In addition, during the work, it was realized that few of the VIPs require added provision for custom code (non-generic) code insertion into the user testbench and hence attributes that would allow the VIP developer to inject such custom code were added ex: *_decl, *_init. These are optional attributes and if not defined, in the VIP Metadata file, Testbench Generator can follow the default implementation. Refer to Table 3 for the complete list.

V. New Flow (Deployed in Production)

Now that it is established that with the VIP metadata available, in a non-proprietary standard format, it is possible to automate the VIP integration. Let us understand the ownership, creation, and delivery of the VIP metadata file. Given that the VIP developers have the best understanding of their VIP’s capabilities, it is ideal for the VIP developers to create and deliver the VIP metadata file as part of the VIP deliverables for every release. On the other hand, Testbench Generator developers should develop a format converter script to convert the standard metadata format to a format best suited for their solution.

ADI has been using the in-house developed Testbench Generator for a decade. This in-house developed generator uses a proprietary format to capture VIP details to enable integration of in-house developed VIPs. This has enabled the users to drag and drop the required VIPs on the generator GUI canvas and create the required testbench architecture, followed by connection with DUT port, followed by VIP sequence-based test creation and finally the testbench generation. To support the VIP metadata delivered by VIP vendors, format converter script that converts the VIP metadata file into an ADI Testbench Generator specific integration file was developed. This enabled authors to retrofit multiple VIP titles from multiple vendors into the Testbench Generator, and that too without any modification to the tool. The format converter script is a python-based script that translates the standard non-proprietary VIP Metadata file into Testbench Generator understandable format. Authors are working to natively support VIP Metadata in Testbench Generator to get-away this script.

Representation of this new flow is shown in Figure 6. Since this flow is independent of source of Testbench Generator and VIPs, authors believe that, if EDA partners also develop Testbench Generators that support the proposed Metadata format, then their customers could also reap the benefit of faster TB development.

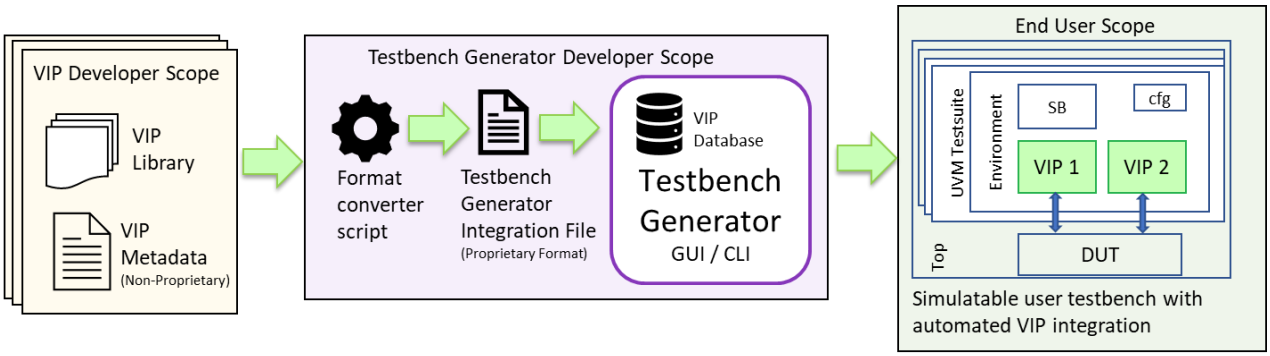

Figure 6. New UVM testbench development flow using Testbench Generator and automated integration of vendor VIP.

Figure 6 shows the proposed flow involving the use of a non-proprietary metadata template to automate the integration of VIP into the testbench.

Any "VIP Vendor" and any "Testbench Generator vendor" can use this metadata template to help their customers (specifically DV Engineers) to build a UVM testbench with the required VIPs in minutes. It is important to note that the metadata file supplied by each VIP vendor contains their confidential material, but this file wouldn't include anything more than what the vendors have already shared with their customer via user guide, example codes, training materials, app notes, etc. which are typically covered under the Non-Disclosure-Agreements (NDA). Given this, it can be seen as legally safe, even from vendor's perspective.

Here are few of the possible use cases:

Vendors VIP-Vend-A and VIP-Vend-B supply VIPs and its associated Metadata to their Customer Cust-A. Cust-A uses a Testbench Generator from Vendor TBG-Vend-A. As the Testbench Generator from TBG-Vend-A support Metadata, DV engineers from Cust-A can quickly generate testbench with required VIPs supplied by VIP-Vend-A and VIP-Vend-B. This ensures the Metadata file is made available only to Cust-A and is not shared with any 3rd Party. -> This use case highlights cross-compatibility.

Similarly, vendors VIP-Vend-A and VIP-Vend-B can also ship their respective VIPs and its associated Metadata to a different Customer Cust-B, who might use an in-house Testbench Generator or from a different Testbench Generator vendor TBG-Vend-B, which supports the Metadata -> This use case highlights reusability.

Key advantages of the proposed flow:

1) *Quick development of testbench of any level of complexity*
2) *VIP integration complexities are consumed by the Testbench Generator*
3) *Improved verification quality and debug efficiency*
4) *No restrictions on VIP and Testbench Generator developers - Scalable solution*

5) *Minimal manual effort for end-user - **The target is to make it zero.***

VI. RESULTS

So far, over 20 VIP titles including the IEEE Ethernet, USB, VESA DP, MIPI CSI-2, MIPI I3C, AMBA, etc. from three major VIP vendors have been integrated into the Testbench Generator. This is an ongoing collaboration with VIP vendors and many more VIP titles are in the pipeline.

Usually, it takes about a week to manually integrate complex-protocol VIPs. But users can now generate the same testbench with Testbench Generator, that supports Metadata, in just a few minutes. This has not only helped the seasoned Digital DV experts but also has significantly lowered the UVM entry bar for designer & Analog/MS DV experts who are not UVM savvy and want to adopt UVM.

Four projects signed-up and have reaped the benefit, while more projects are in pipeline.

In a recent engagement, full-fledged UVM Testbench for USB 3.2 DUT was created in under 30 mins. The architecture of the testbench is shown in Figure 6.

Key features of this testbench

1. Combination of both external VIP (USB) and in-house VIPs
2. GUI aided DUT and VIP port connections
3. Provision to create basic testcases with USB VIP sequences
4. Compile clean and ready to simulate

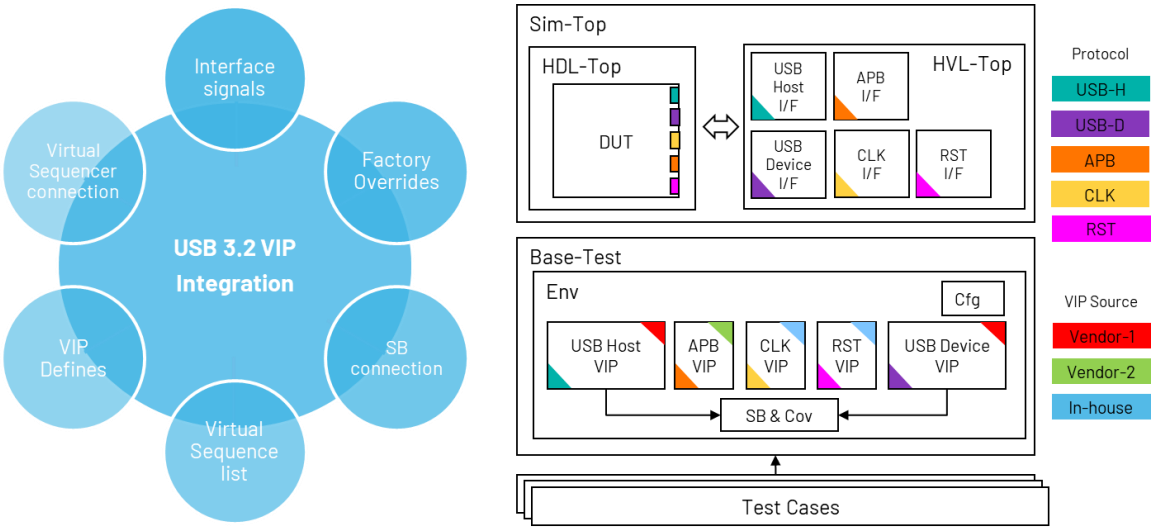

Figure 7. UVM testbench generation for USB 3.2 DUT

IV. CONCLUSION

Leveraging automation is one of the best ways to bring down the development time of (complex) testbenches. A well-architected and configurable testbench can further improve the verification quality and debug efficiency. VIPs and Testbench generators play a key role in development of such a sophisticated testbench. Through this paper, the authors have highlighted the lack of globally accepted standards for VIP developers to enable the automation of VIP integration into UVM testbenches via a Testbench Generator. The authors have also proposed a solution in the form of a non-proprietary metadata format to exchange data between VIP and Testbench generator development teams to enable the required automation and have further explained the workflow to be adhered to by VIP and Testbench Generator development teams.

Key Takeaways:

1) *Non-proprietary format to integrate VIP from one vendor into Testbench Generator from another*
2) *Rapid integration of vendor VIPs into UVM testbenches via Testbench Generator*
3) *Scalable and non-invasive solution ensuring liberty to VIP developers and benefitting entire DV community*

VII. REFERENCES

[1] "The 2022 Wilson Research Group Functional Verification Study", https://blogs.sw.siemens.com/

VIII. APPENDIX

All the supported elements and attributes are listed in Table 3.

Table 3. VIP Metadata element and attribute list

Element	Attribute	Sim_Arch	Simulator	UVM_Ver	Description	Metadata
setup; common; <agent>;		32; 64;	XLM; VCS; QSIM;	UVM11d; UVM12; UVMIEEE;		
setup	vendor_name				Name to identify the vendor in TB generator	vendor
setup	vip_name				Name to identify VIP in TB generator	<protocol> <version>
setup	vip_title				Name used by vendor to identify the VIP in their respective portfolio	protocol
setup	excl_agents				Are the agent exclusive (Yes/No). This applies to VIPs with multiple agents. Yes→ Only one agent can be enabled per instance of VIP Env. No → Multiple/All agents can be enabled per instance of VIP Env.	Yes/No
setup	env_var				Set required environment variables	setenv VIP_LIB_PATH ${TB_ROOT}/agents/<vendor>/<vip>
setup	src_path				Path to the location of the user-editable VIP source code that needs to be copied to the generated TB	${VIP_ROOT}/…./
setup	pre_comp				Perform required operation/ execute script before compilation	source $VIP_LIB_PATH/vip_comp.csh
setup	pre_sim				Perform required operation/ execute script before simulation	source $VIP_LIB_PATH/vip_sim.csh
setup	pre_comp_sim				Perform required operation/ execute script before compilation and simulation	source $VIP_LIB_PATH/vip_all.csh
setup	comp_opt				Compilation options	-define VENDOR_PROTOCOL
setup	comp_file				Files to be compiled	${TB_ROOT}/agents/x/y/z
setup	inc_dir				Directories required for compilation	${TB_ROOT}/agents/x/y/z
setup	sim_opt	32	XLM		Run time options for simulator in 32 bit mode	-pli ${VIP_ROOT}/somefile.so
setup	sim_opt	64	VCS		Run time options for simulator in 64 bit mode	-P ${VIP_ROOT}/somefile_64.so
common/ <agent>	pkg_import				VIP env/agent package to be imported in TB env scope Scope: TB env package	vendor_protocol_pkg
common/ <agent>	pkg_import_global				VIP env/agent package to be imported in global scope Scope: Global	vip_common_pkg
common/ <agent>	inc_file				Files to be included in TB environment package Scope: TB env package	vip_protocol_file.sv
common/ <agent>	inc_file_global				Files to be included at the global scope Scope: Global Useful to handle non-uvm file compilation, which can be done outside TB pkg declaration or in simulator command file	vip_protocol_interface.sv
common/ <agent>	add_tbpkg_code				Custom code needed for VIP compilation This code will be added in the TB environment package scope prior to importing other packages Typically used for type or vendor specific forward-declaration/parameters Note: If used in <vendor>_common sheet, its vendor-specific else it is type-specific	typedef class vip_example_class;
common/ <agent>	add_tbpkg_code_global				Custom code needed for VIP compilation This code will be added in the global scope Typically used for type or vendor specific forward-declaration/parameters Note: If used in <vendor>_common sheet, its vendor-specific else it is type-specific	typedef class vip_example_class;
common/ <agent>	param_list				List of parameters used in the TB env scope Comma separated list should have the parameter name and its value Suggestion: Recommended to use only one parameter per line Scope: TB Env package	ADDRESS_WIDTH=32, DATA_WIDTH=32
common/ <agent>	param_list_global				List of parameters used in the global scope Comma separated list should have the parameter name and its value Suggestion: Recommended to use only one parameter per line Scope: Global	ADDRESS_WIDTH=32, DATA_WIDTH=32
common/ <agent>	macro_list_global				List of compile macros used in the global scope Suggestion: Recommended to use only one macro per line Scope: Global	VIP_SAMPLE_EN_MACRO

common/ <agent>	build_phase_tb env_code				Custom code can be added in the build_phase of TB env (Ex: overrides)	set_type_override_by_type(vip_driver::get_type(),vip_usr_driver::get_type());
common/ <agent>	eoe_phase_tbe nv_code				Custom code to be added in the end of elaboration phase of the TB environment class	<code>
common/ <agent>	sos_phase_tbe nv_code				Custom code to be added in the start of simulation phase of the TB environment class	<code>
<agent>	agent_type				Agent type to instantiate and create the VIP instance in the TB environment	vip_protocol_agent
<agent>	agt_decl				Custom code for declaration of VIP agent in TB env Default: Typical UVM declaration code based on agt_type	vip_protocol_agent ${inst};
<agent>	agt_init				Custom code for creating the VIP agent in TB env Default: Typical UVM creation code based on agt_type	${inst} = vip_protocol_agent :: type_id :: create("${inst}");
common/ <agent>	if_type				Interface type	vip_protocol_interface
common/ <agent>	if_decl				Declaration of interface in top file	vip_protocol_interface ${inst}_if( );
common/ <agent>	if_set_id				ID to get/set the interface handle via config_db Note: TBG adds the typical i/f config_db set statement with id as "vif" if this attribute is not used	vif
common/ <agent>	if_set_hier				VIP hierarchy that needs access to the interface Note: TBG adds the typical i/f config_db set statement with id as "vif" if this attribute is not used	${inst}*
common/ <agent>	sig_list				List of VIP interface signals available for connection with DUT ports	input sigA, output [1:0] sigB, inout sigC
common/ <agent>	cfg_type				VIP configuration class type Default: <vip_agent_type>_cfg	vip_protocol_config
common/ <agent>	cfg_decl				Custom code for declaration of VIP agent configuration in TB configuration Default: Typical UVM declaration code based on cfg_type	vip_protocol_config ${inst}_cfg;
common/ <agent>	cfg_init				Custom code for creating the VIP agent configuration in TB configuration Default: Typical UVM creation code based on cfg_type	${inst}_cfg = vip_protocol_config :: type_id :: create("${inst}_cfg");
common/ <agent>	cfg_set_id				ID to get/set the cfg class handle via config_db Default: Typical config_db set statement with id as "cfg"	cfg
common/ <agent>	cfg_set_hier				Hierarchy path in which the VIP config handle must be accessible via config_db set for the “cfg” class Default: Typical config_db set statement with id as "cfg"	${inst}.*
common/ <agent>	cfg_vars				VIP configuration class variables to be available in generator GUI Syntax: <field> = <#value1,value2,value3#> <field> = <[value1:value10]> <field> = <value> Note: Relative to top VIP config instance	vip_protocol_kind = <#xkind,ykind#>;
common/ <agent>	cfg_util				Registration of configuration instance to UVM factory	`uvm_field_object(${inst}_cfg, UVM_DEFAULT)
common/ <agent>	cfg_ap				Configuration Variable to determine if agent is set as active or passive Note: TBG adds typical statement as per the selection list set while adding an agent (By default set as active)	${agt_inst}_agent_cfg.is_active = UVM_${ap};
<agent>	sqr_type				VIP agent sequencer type Default: <vip_agent_type>_sqr	vip_protocol_sqr
<agent>	sqr_decl				Custom code for declaration of VIP agent sequencer in TB virtual sequencer Default: Typical UVM declaration code based on sqr_type	vip_protocol_seqr ${inst}_sqr;
<agent>	sqr_conn				Setting the handle of the VIP sequencer in the environment virtual sequencer. (connect_phase) Note: TBG adds the typical connection statement if this attribute is not used. This attribute is mandatory if custom cfg_decl is used	sub_agent.sequencer

<agent>	seq_type				Declare the sequence type to be run on the sequencer in the virtual sequence	vip_seq `vip_VIP_PARAM
<agent>	seq_decl				Declare the VIP sequence type to be run on the sequencer in the virtual sequence	vip_seq2 ${inst}_seq;
<agent>	seq_init				Creating the sequence as per the sequence handle declared in the virtual sequence	${inst}_seq = vip_seq2::type_id::create(\"${inst}_seq\");
<agent>	seq_start_code				Custom code to Start VIP Seq on Sqr.	if (p_sequencer.cfg.has_${inst} && p_sequencer.cfg.${inst}_cfg.has_tx_agt) begin ${inst}_seq.start(p_sequencer.${inst}_sqr, this); end
<agent>	seq_lib				VIP sequence library If typical UVM sequence syntax is applicable, then this attribute can be used instead of listing each sequence. TBG manages the instantiation, creation, and execution of sequence.	vip_seq1, vip_seq2
common/ <agent>	vsqr_type				VIP Virtual Sequencer type name	vip_virtual_sequencer vip_virtual_sequencer `VIP_PARAM (if parameterized)
common/ <agent>	vsqr_decl				Custom code for declaration of VIP virtual sequencer in TB virtual sequencer Default: Typical UVM declaration code based on vsqr_type	vip_protocol_vsqr ${inst}_vsqr;
common/ <agent>	vsqr_conn_code				Setting the handle of the VIP Virtual Sequencer in the environment virtual sequencer (connect_phase) Note: TBG adds the typical connection statement if this attribute is not used. This attribute is mandatory if custom cfg_decl is used	$cast(vsqr.${inst}_<VIP_NAME>_vsqr.Seqr, ${inst}.sequencer);
common/ <agent>	vseq_list				List of Virtual Sequences available for use by the agent	vip_virtual_sequenceA, vip_virtual_sequenceB
common/ <agent>	reg_adapt_type				Used to declare register adapter. For RAL Usage.	vip_agent_adapter `VIP_PARAM
common/ <agent>	reg_act				Used to Connect the RAL model to agent register adapter	cfg.${agt_inst}_env_cfg.has_agent_vip
common/ <agent>	tr_type				Transaction class type	vip_xtn
common/ <agent>	sb_port				Used to Connect monitor analysis ports to scoreboard implementation ports	vip_protocol_monitor.x_dir_prt
common/ <agent>	sb_pkt_type				Use this if the SB packet type is different from tr_type	vip_checker_packet