2021 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE OCTOBER 26-27, 2021

One Testbench to Rule them all!

Salman Tanvir, Senior Staff Engineer Digital Verification, EPOS GmbH & Co. KG, An Infineon Technologies Company, Duisburg, Germany (*salman.tanvir@infineon.com*)

Markus Brosch, Senior Staff Engineer Digital Verification, Infineon Technologies Austria AG, Villach, Austria (*markus.brosch@infineon.com*)

Amer Siddiqi, Manager Digital Design, EPOS GmbH & Co. KG, An Infineon Technologies Company, Duisburg, Germany (*amer.siddiqi@infineon.com*)

***Abstract*—With increasing SoC design complexity and time-to-market pressure, functional verification is often at the critical path to tape-out signoff. Testbench development is a time intensive and repetitive process as a new testbench has to be developed for every Design under Verification (DUV). This includes setting up the infrastructure e.g. handling of clocks, sideband signals and/or interface UVCs. Our goal in this paper is to outline a SystemVerilog testbench architecture which enables efficient testbench bring-up and supports vertical and horizontal reuse. This is achieved by implementing a base layer which includes the common infrastructure for all deriving testbenches. Furthermore, different testbenches can be easily integrated together in a bottom up flow. This architecture was developed in an actual project involving the verification of a multi subsystem CMOS radar SoC which integrated over 50 testbenches. Finally, we will present the results and the experience of using this architecture in this project.**

Keywords—UVM; SystemVerilog; Testbench; Vertical/Horizontal Reuse; Bottom-Up Development;

I. INTRODUCTION

Verification engineers spend a significant part of their time developing and maintaining testbenches. With every new project, a testbench has to be built up. This also includes the development of the base UVM testbench skeleton, the infrastructure required to drive and monitor clocks, resets, sideband signals [1] and the integration of standard interface UVCs. In our experience, we frequently observed custom redevelopment for the aforementioned components for every new project rather than relying on a generic reusable verification IP (VIP). This has numerous disadvantages. Firstly, repetitive redevelopment wastes precious project resources. Quality of testbench components cannot be guaranteed and is difficult to maintain when not relying on pre-developed reusable VIP. The look and feel of each environment is also not consistent and hence makes it difficult to understand. Finally, the reusability and portability of components as well as testbench environments is compromised. Therefore, these testbench environments are not well suited for vertical reuse.

This paper outlines a testbench architecture to address the aforementioned problems. The first aspect of the architecture is to have one base testbench that provides the common infrastructure for all derived testbenches. This includes UVCs to handle clocks, sideband signals and the desired standard interfaces. The second aspect is to enable vertical integration of derived testbenches in a bottom up flow. For this to work, testbenches need to consider a certain level of configurability. The testbenches not only have to consider both active and passive contexts, but also need to provide the possibility to reconfigure the common infrastructure. This is due to the fact that when moving to a higher integration level, more than one testbench could be vying for the same common resource.

This work was developed in the context of an actual project consisting of a multi subsystem radar System-on-Chip. The goal was to be able to setup/develop testbench environments efficiently and to achieve maximum reuse by integrating them in a bottom up flow.

We begin by first defining the feature set that we consider vital for such a testbench architecture. The following chapter then describes the architecture and how it implements the defined features. At the end, we present a conclusion of this work.

II. REQUIREMENTS

This chapter describes the feature set that we consider vital for the testbench architecture.

A. *Reusable base testbench*

A base testbench is required that incorporates the common infrastructure needed for verification e.g. handling of clocks, sideband signals and/or interface UVCs. This includes a reusable class based testbench environment as well as a base module that instantiates the required BFM's and Interfaces.

B. *Hierarchical Reuse*

The testbench environments developed from this base environment should be able to be connected together in a bottom up flow. The following aspects should be supported:

- Passive Reuse: Reuse of lower level testbench environments that encapsulate the verification of an IP/sub-system. This not only includes checks but also the behavioral modelling of the components
- Active Reuse: Support to be able to run lower level sequences on a higher level
- Register models should be automatically integrated when moving to a higher level
- Integrating multiple instances of a lower level testbench including the register model should be supported

C. *Testbench Generation*

A code generator should be available to automate the following testbench aspects:

- Building the basic testbench skeleton inheriting from the base environment
- Integrating a pre-generated register model with the capability of reusing register models from lower levels
- Automated integration of previously generated testbenches from a lower level

III. ARCHITECTURE

This chapter will describe the architecture of the concept in detail. Initially, the base testbench consisting of the HVL class based parts and the top HDL module are explained. The integration of the common infrastructure as well as the configuration mechanism are also described here. This is followed by a description of how the testbench HVL classes and HDL module are extended from the base and how the hierarchical reuse concept is realized. The configurability necessary for enabling hierarchical reuse is then explained. Next, the automated setup of the register model at different integration levels is described. Finally, this section ends with the description of the stimulus reuse concept.

A. *Base HVL Classes & HDL module*

To fulfill the first requirement of having a base testbench encapsulating the common infrastructure, an extension mechanism is required in order to reuse or build on top of this. For the HVL class based part, the class inheritance mechanism of the object oriented programming (OOP) paradigm is used. The base classes are described individually in Table 1.

HVL Base Class	Description
Environment	Encapsulates the desired common infrastructure environments e.g. clocks, sidebands etc.
Configuration	Configurations for common infrastructure environments & general control knobs.
Base Test	Builds the base environment.
Base Sequence	Contains the base configuration handle and reusable properties/methods.
Base Virtual Sequencer	Consists of handles to all common infrastructure sequencers as well as an API to enable vertical stimulus reuse which will be described later.

Table 1 HVL Base Layer

The HDL part of the base testbench is encapsulated in a single top level SystemVerilog module. This module includes all desired BFMs and interfaces for the common infrastructure. In our testbench, the base HDL module only includes BFMs for AHB & SPI. The sideband interfaces are testbench specific and are generated in derived testbenches. Similarly, clock BFM's are also integrated in derived testbenches for more flexibility.

Similar to the HVL part, an extension mechanism is needed for derived testbenches. As there is no way to extend modules in SystemVerilog similar to classes another method is required. We consider the SystemVerilog binding and hierarchical upward referencing features ideally suited for this purpose. With this approach a derived testbench can simply be bound into the common HDL base module. This bound testbench can directly reference interfaces, BFM's and parameters encapsulated in the base HDL module.

To be able to configure the common infrastructure and to control general aspects of the testbench, the base HVL & HDL layer provide various configuration knobs and parameters. Figure 1 illustrates the configuration knobs for the HVL environment.

```
class common_base_tb_config extends uvm_object;

  // environment configuration objects
  rand clocks_env_config    clocks_env_cfg;
  rand sideband_env_config  sideband_env_cfg;
  rand cdn_ahb_env_cfg      ahb_env_cfg[];
  rand spi_env_cfg          spi_env_cfg;

  //configuration fields
  rand uvm_active_passive_enum  is_active;
  rand bit                      has_cov;
  rand bit                      has_sva;
  rand int unsigned             num_ahb_buses;
  rand bit                      has_ahb;
  rand bit                      has_spi;

  //regmodel
  rand bit                      build_regmodel;                 // switch to control register model building
  rand bit                      add_reg_hdl_paths;              // switch to add backdoor paths
  rand bit                      regmodel_supports_byte_enable;  // used to set corresponding reg_adapter field

  rand bit                      global_tr_rec_enable;           // used to globally enable transaction recording


  protected rand bit enabled_checks[string] = '{"checks_enable":1,"ip_checks_enable":1};

....

endclass
```

Figure 1 HVL Configuration

In our testbench, the clocks, sideband, AHB and SPI UVCs form the common infrastructure. As these must be configurable in derived testbenches, the base HVL environment encapsulates the configurations of these components. As some testbenches may not contain all supported standard interfaces, knobs are provided to be able to exclude them e.g. has_ahb and has_spi. Additionally, building of coverage and register model is configurable. When testbenches are reused on a higher level, they need to be switched from an active to a passive context as the interfaces of the reused testbench will be driven by the higher level environment. All drivers in the reused environment need to be disabled. This is achieved with the is_active configuration knob. A global checks enable knob is provided to disable checks if required. An additional knob allows selective enabling of checks that are only considered in an IP and not system context. This was done as we anticipated potential performance degradation on top level. For the HDL part, parameters are available to configure various aspects of the AHB and SPI BFMs. This includes e.g. the number of slaves for AHB and SPI, but also the width of AHB data and address bus.

B. Inheritance/Hierarchical reuse

Today's complex SoC's are employing the "Divide & Conquer" strategy for design and verification. The SoC is typically broken down into IP's/Subsystems to manage complexity. These blocks can be developed by various teams working in parallel and are integrated in a bottom up flow. Verification at every integration layer may or may not reuse lower level testbench components/environments. In our experience, reuse of components rather than environments is more common. This is a more straightforward approach, but can easily lead to scaling issues either

when many components have to be reused, or when required to be repeated for higher integration layers. Additionally the details regarding interconnection of components has to be taken care of. Complete environment reuse solves the aforementioned problems but also introduces its own set of challenges. Primarily the testbench architecture/setup becomes more complicated as testbenches at various integration levels have to coexist in a single environment. Additionally as mentioned in the introduction, the common infrastructure has to be shared amongst all testbenches. However the advantages of reusing complete environments outweighs the challenges. All passive aspects including checks, coverage and transaction modelling can be reused. For complex environments that e.g. encapsulate bus systems, the interconnect setup can be inherited rather than redoing. This can be seen in Figure 2 where the bus infrastructure at each layer is extended. In some cases stimulus can also be vertically reused. This was a particularly beneficial in our project and will be explained in section III.D.

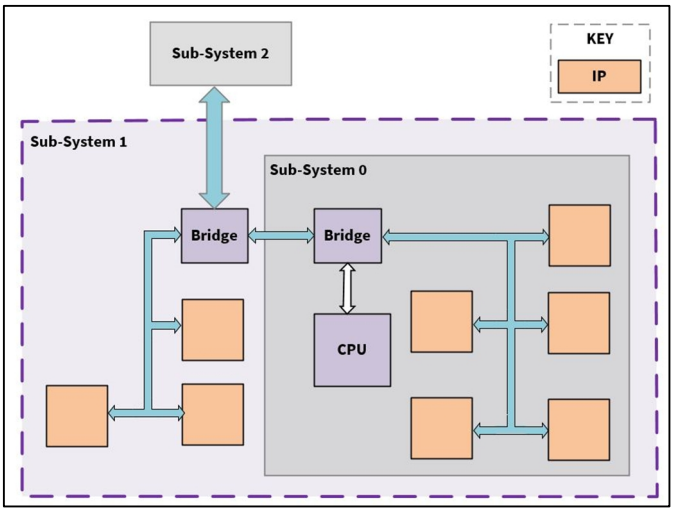

Figure 2 Interconnect Infrastructure Reuse

```
function void common_tb_base_test::build_phase(uvm_phase phase);
  pre_build();
  super.build_phase(phase);

  if (tb_cfg == null)
    `uvm_fatal(get_type_name(), "TB Config item must be passed from IP/System testbench base test!!");

  tb_env = common_tb_env::type_id::create("tb_env", this);
  tb_env.tb_cfg = tb_cfg;

  if (tb_cfg.global_tr_rec_enable)
    uvm_config_int::set( this, "*", "recording_detail", 1);  //TRANSACTION RECORDING

endfunction: build_phase


function void ip_a_tb_base_test::build_phase(uvm_phase phase);
  pre_build();
  create_tb_cfg();  // create and configure tb configuration object

  ip_a_tb_env             = ip_a_tb_env::type_id::create("ip_a_tb_env", this); // create tb env
  ip_a_tb_env.ip_a_tb_cfg = ip_a_tb_cfg; // pass tb cfg to tb env
  tb_cfg                  = ip_a_tb_cfg; // pass tb cfg to parent tb layer (common_tb_base_test)

  super.build_phase(phase);// build parent testbench environment (common_tb_base_test builds
                           // common_tb_env which contains common infrastructure).

endfunction: build_phase
```

Figure 3 Configuration and Building of Derived and Common Base Testbench Environments

In UVM, the test is the top-level component in a testbench and responsible for building and configuring the testbench environment. Typically all tests in a testbench are extending from a base test. In the proposed architecture the base test of each testbench is deriving from the common infrastructure base test class. This way every testbench can build its own environment on top of the common infrastructure. The environment configuration of each

testbench is also inheriting from the common testbench configuration object. This allows the extension of testbench specific knobs and also to configure the common infrastructure environment. Figure 3 illustrates the configuration and building of the testbench specific and common infrastructure environments. In the build phase of the derived testbench, the create_tb_cfg() method creates the configuration object and sets the knobs as desired. This object is then passed to the environments of the derived as well as the common base testbenches. The super.build_phase(phase) call will then build the common infrastructure environment.

As described above and illustrated in Figure 4, every testbench builds its own environment on top of the common infrastructure. This environment encapsulates all components required for this testbench as well as sub environments for vertical reuse. This self-contained environment includes the hooking up of all component TLM ports and allows the environment to be reused as is on the next higher level [2]. The reused environment(s) can be reconfigured as desired. An additional aspect for the proposed architecture is the reconfiguration/remapping of the common testbench infrastructure for every level. This is due to the fact that the single common infrastructure environment has to be shared between the top and every vertically reused testbench environment. All configuration is handled in the base test of every testbench. Environments of derived testbenches do not need to extend from the common base environment, as this is already built in the inherited common base test.

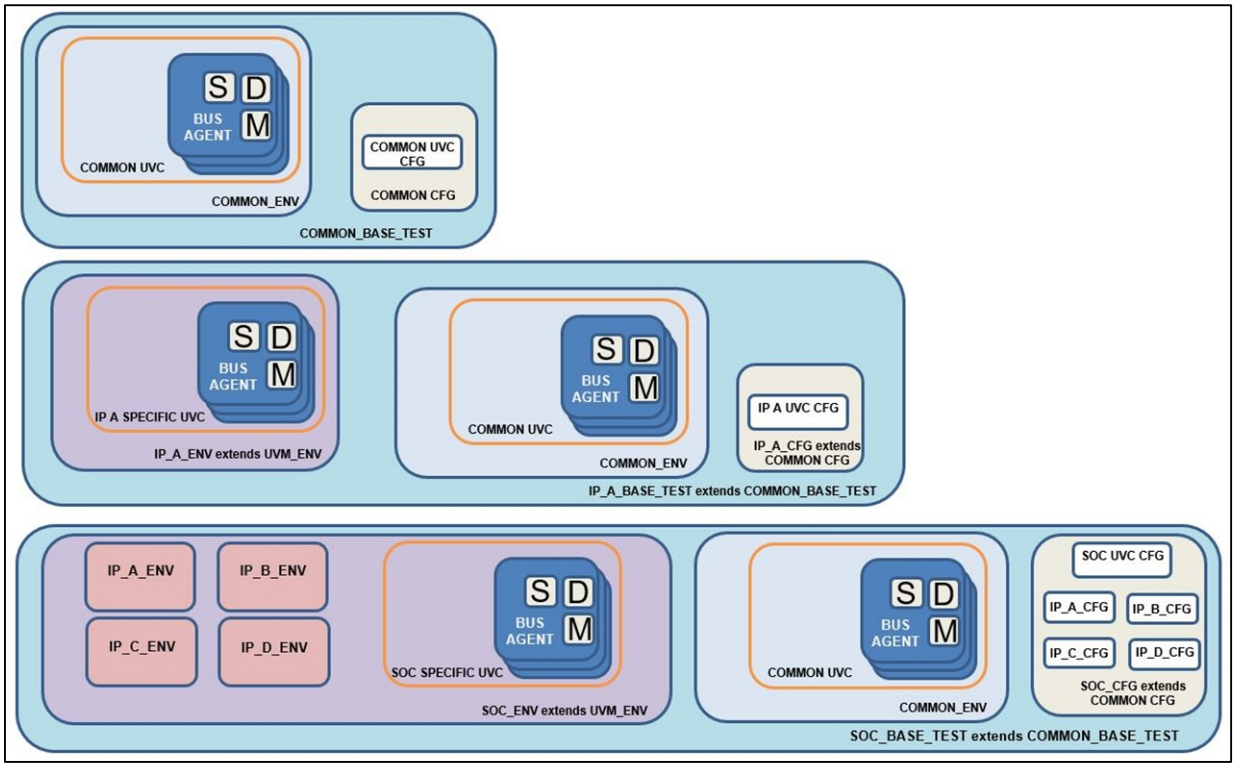

Figure 4 IP/System Testbench Class Hierarchy

As illustrated in Figure 5, the HDL part of the common testbench is encapsulated in a single top level SystemVerilog module, which in our case includes BFMs for AHB & SPI. In order for the derived testbenches to connect with the common infrastructure, they are bound into the top common module. Hierarchical upward referencing enables all common components to be resolved.

For vertical hierarchical reuse, the top testbench modules require a parameter to allow switching between IP or SoC contexts. The active components and DUV are only instantiated in an IP context. Additional parameters can be added as required. An example in our testbench are the AHB index parameters that are used to connect AHB slave agents to the desired AHB environment. These slave interfaces in the top module are registering with the UVM config database using these index parameters. This allows a flexible setup of the AHB bus interconnect at higher integration levels.

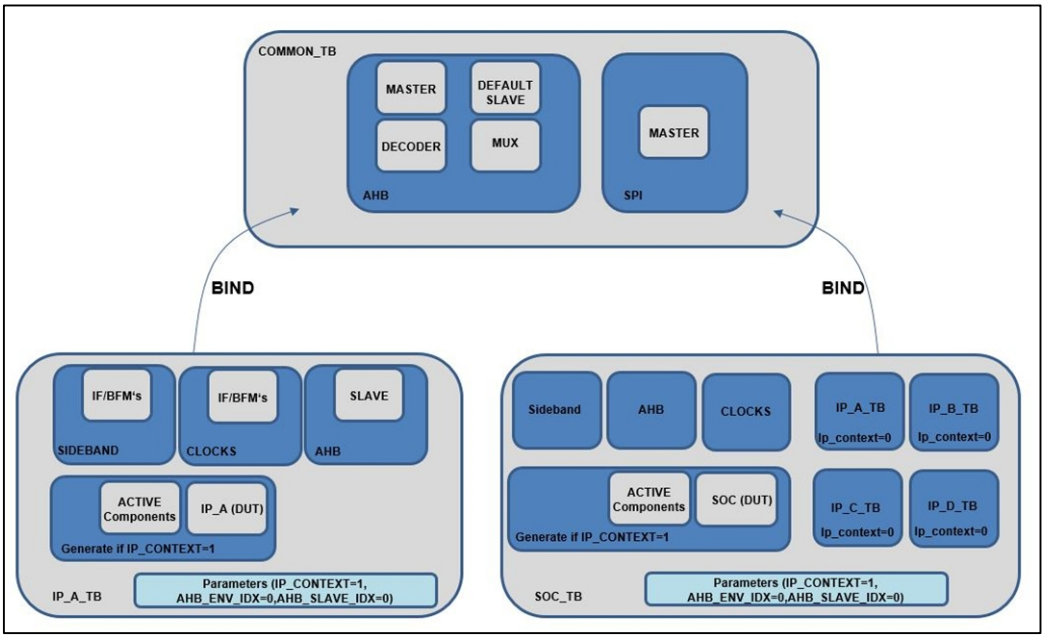

Figure 5 HDL Common Module and Binding

C. *Register Model Integration Concept*

UVM Register models are usually generated from a register file specification e.g. IP-XACT which is typically done on an IP basis. The generated model needs to be integrated into the testbench which requires the following steps: definition of a bus adapter, connecting the bus sequencer, implementing the bus specific extension of UVM frontdoor and finally to hook up bus monitor(s) to the UVM predictor. When moving to a higher integration layer, not only do these steps have to be repeated, but additionally, all register blocks first need to be combined into a system context.

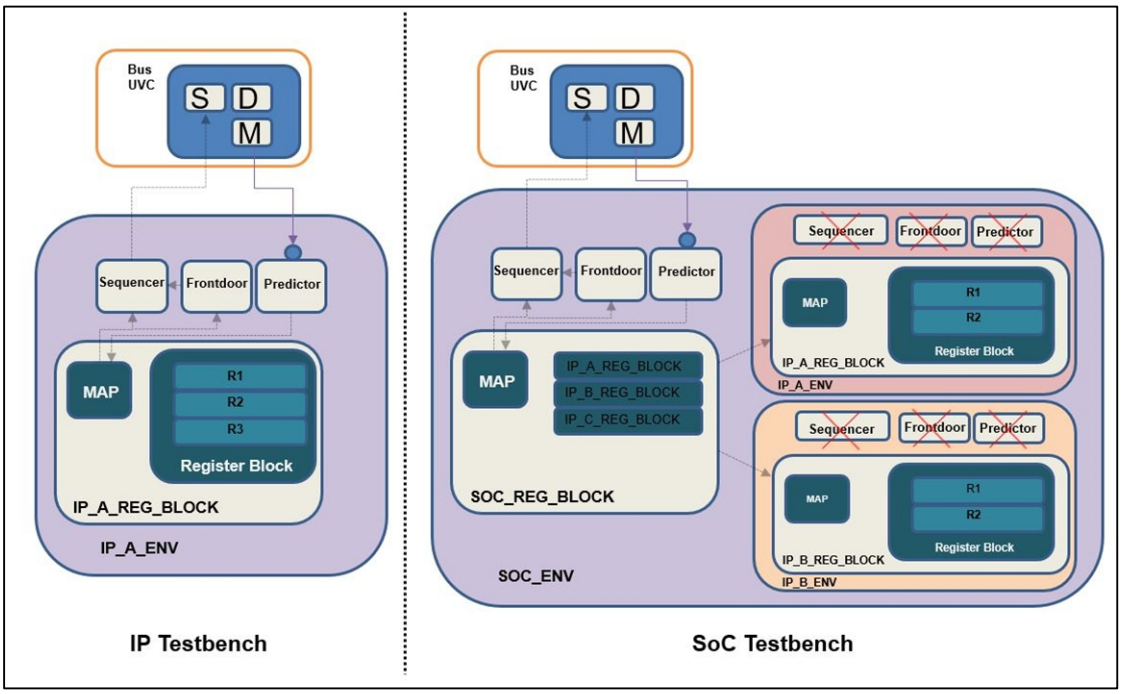

Figure 6 Register Model Integration

Our goal was to be able to automate the integration of pre-generated IP register models and also reuse them in a sub-system or SoC context. As illustrated in Figure 6, the register model blocks are encapsulated and built in the respective testbench environments. Register blocks on SoC/sub-system level encapsulate handles to lower level blocks, which are set in the UVM connect phase. Hence the responsibility for every testbench environment to build its register block is maintained when integrated at a higher level. All the aforementioned setup aspects e.g. predictor, frontdoor and adapter, need to be done for each testbench environment. However, these parts are switched off/reconfigured when a testbench gets reused on a higher level.

```
class ip_a_tb_env extends uvm_env;
  ip_a_block                  regmodel;
  bus_predictor               bus_predictor;
  bus_adapter#(bus_transaction) bus_adapter;

function void build_regmodel();
  bus_predictor =  create();
  bus_adapter   =  create();
  regmodel      =  create();
  regmodel.build();

  regmodel.lock_model();
  if (ip_a_tb_cfg.is_active == UVM_ACTIVE)
    add_regmodel_frontdoor();
endfunction: build_regmodel


function void build_phase(uvm_phase phase);
.....
  if(ip_a_tb_cfg.build_regmodel)
    build_regmodel();
.....
endfunction: build_phase

.....

endclass
```

Figure 7 Building of Register Model

Figure 7 shows simplified code snippets of an IP testbench environment with focus on the building of the register model. The code for a sub-system/SoC environment would be similar with the exception that the register model is not locked and the frontdoor is not setup in the build phase. This is because the lower level register models are integrated into the top register model in the connect phase and the build phase would be too early for these steps.

After building the register model(s), the bus sequencer needs to be configured for active stimulus and the required bus monitors need to be connected with the predictor. For these active and passive interfaces, the adapter must also be configured. For sub-system/SoC environments, the integration of all lower testbenches needs to be done additionally. All these aspects are handled inside a dedicated function for each environment. The function prototype is shown in Figure 8.

```
function void connect_regmodel(bus_monitor bus_monitor[],bus_predictor predictor_arg);
```

Figure 8 Register Model Integration Function Prototype

The connect_regmodel() function for the top testbench environment is called from the base test. In sub-system/SoC environments, the corresponding functions for lower level testbenches are called recursively downwards from the top environment.

```
function void connect_regmodel(bus_monitor bus_monitor[],bus_predictor predictor_arg);
  if(predictor_arg == null) begin                          // Top Level Context
    foreach (bus_monitor[i])
      bus_monitor[i].port.connect(bus_predictor.bus_in);
    bus_predictor.map     = regmodel.default_map;
    bus_predictor.adapter = bus_adapter;
  end else begin                                           // Integration Level Context
    bus_predictor = predictor_arg;

    if (! $cast(bus_adapter, bus_predictor.adapter))
      `uvm_fatal(get_type_name(),$sformatf("bus_predictor.adapter cast failed!"))
  end

  if (ip_a_tb_cfg.is_active == UVM_ACTIVE) begin
    regmodel.default_map.set_sequencer(virtual_sequencer.bus_sequencer, bus_adapter);
    virtual_sequencer.regmodel = regmodel;
  end
endfunction: connect_regmodel
```

Figure 9 Connect_regmodel () Method for IP Testbenches

Figures 9 and 10 show the connect_regmodel() function of an IP and SoC testbench environment respectively. The code shown in the IP level example is identical for system level testbenches and hence Figure 10 only shows the additional code. As mentioned earlier, the function of the top testbench environment is called from the base test. Handle(s) to the bus monitor(s) are passed as an argument. However, a null pointer is passed for the predictor argument. This argument is used to determine whether a testbench environment is the top most environment or a lower one. As the top testbench environment builds the predictor, no handle is needed to be passed from the base test. However the connect_regmodel() function of the top most environment will propagate the top predictor and bus monitor handles recursively down the register model hierarchy. Although the bus monitors are passed down, they will not be used by lower level register blocks as these are already connected to the predictor in the top environment.

```
function void connect_regmodel(bus_monitor bus_monitor[],bus_predictor predictor_arg);
...

  //Connect lower level register model blocks with top register model
  regmodel.ip_a_regmodel          = ip_a_tb_env.regmodel;
  regmodel.sub_system_b_regmodel  = sub_system_b_tb_env.regmodel;

  // Configure lower level register model blocks
  ip_a_tb_env.regmodel.configure(regmodel);
  sub_system_b_tb_env.regmodel.configure(regmodel);

  // Add lower level register model maps as sub maps of top level register model map
  regmodel.default_map.add_submap(ip_a_tb_env.regmodel.default_map,soc_tb_cfg.ip_a_base_addr);
  regmodel.default_map.add_submap(sub_system_b_tb_env.regmodel.default_map,soc_tb_cfg.sub_system_b_base_addr);

  // Call connect_regmodel methods of lower level testbenches. The toplevel predictor and bus monitors are passed down.
  sub_system_b_tb_env.connect_regmodel(bus_monitor,bus_predictor);
  ip_a_tb_env.connect_regmodel(bus_monitor,bus_predictor);

  regmodel.lock_model();
  if (soc_tb_cfg.is_active == UVM_ACTIVE)
    add_regmodel_frontdoor();

endfunction: connect_regmodel
```

Figure 10 Connect_regmodel () Method for System Testbenches

For the sub-system/SoC context a few additional steps are required to integrate lower level register models into the top register model block. The first step is to set the lower level register model handles in the top register block. These lower register blocks are then added as a child of the top block by calling the uvm_reg_block::configure() method. The maps of lower level blocks are then added as a sub-map of the top register block map with a configurable offset. This allows reusing this sub-system/SoC environment in another system with a different address map. Then all lower level connect_regmodel() functions are called. As seen in Figure 10 and already mentioned before, the locking of the register block is done at the very end for sub-system/SoC environments. Additionally the frontdoor is set only in the top environment.

Although our proposal is limited to a single bus interface, we believe that this can be easily extended to a multi bus system by scaling the number of maps for the register model.

D. Stimulus/Sequence Reuse

This architecture development started with the primary focus on reusing testbenches passively. It was anticipated that the Portable Test and Stimulus Standard [3] could be integrated at a later point to additionally support stimulus reuse. However during development we realized some limited although advantageous reuse scenarios could be directly supported with some enhancements. These scenarios are illustrated in Figure 11. A significant number of sequences in our environment are consisting of transactions on standardized interfaces e.g. AHB, SPI and JTAG. AHB stimulus can be remapped on a higher level to a different agent by providing a configurable base address offset. As the register model frontdoor is also integrated with AHB, this means that register model based stimulus can also be reused. Similarly sequences developed in block level testbenches for the SPI/JTAG protocol can be remapped to the higher level as these interfaces are connected directly to the lower level block. We are also able to support the limited case of remapping sideband stimulus on block level pins that are connected directly to the higher level entity.

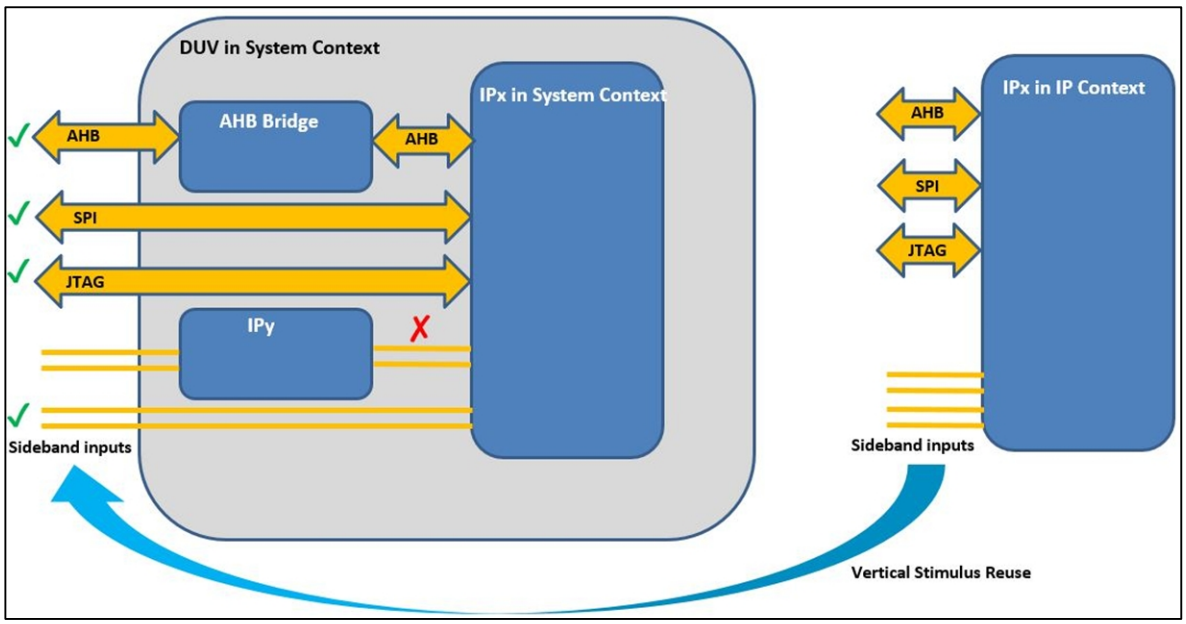

Figure 11 Stimulus Reuse Scenarios

The first step to support vertical reuse is to be able to run sequences on different virtual sequencers. As described in section III-A, all virtual sequencers are derived from the common base virtual sequencer which also implements an API to support vertical reuse. The base sequence of every testbench uses the UVM utility macro `uvm_declare_p_sequencer with a type parameter to specify the p_sequencer for the sequence [4]. The default value for this parameter is the type of the virtual sequencer of the testbench as illustrated in Figure 12.

```
class ip_a_tb_base_seq #(type T = ip_a_tb_virtual_sequencer) extends common_base_seq;
  ip_a_tb_config tb_cfg;    // tb configuration handle
  ip_a_block     regmodel;  // register model handle

  `uvm_declare_p_sequencer(T)

.....

endclass
```

Figure 12 Type Parameterized p_sequencer

Before a sequence starts, the encapsulated configuration object and register model handles must be set. These handles are pulled from the p_sequencer in the pre_start() method of the base sequence as shown in Figure 13. The common base virtual sequencer holds an associative array for string based lookup of handles. For each of the configuration and register model handles, an API allows setting and getting of these handles. They are set in the base test.

```
task ip_a_tb_base_seq::pre_start();
  super.pre_start();

...
  if (p_sequencer != null) begin
    if(!$cast(tb_cfg,p_sequencer.get_sequencer_cfg("ip_a"))) begin
      `uvm_fatal(get_type_name(),"$cast(tb_cfg,p_sequencer.get_sequencer_cfg(\"ip_a\"))) call failed!");
    end

    if (p_sequencer.regmodel == null)
      `uvm_fatal(get_type_name(), "regmodel handle is null!")
    else begin
      if(!$cast(regmodel,p_sequencer.get_sequencer_regmodel("ip_a"))) begin
        `uvm_fatal(get_type_name(),"$cast(regmodel,p_sequencer.get_sequencer_regmodel(\"ip_a\"))) call failed!");
      end
    end

  end else begin
    `uvm_fatal(get_type_name(), "p_sequencer handle is null!")
  end

endtask: pre_start
```

Figure 13 Setting of Register Model and Configuration Object Handles in Sequence

If lower level sequences are called from the higher level, it must be ensured that the required physical sequencers are encapsulated in the top virtual sequencer. In our case, the common infrastructure sequencers are always inherited from the base virtual sequencer. Any additional sequencers required can be integrated into the top virtual sequencer.

IV. CODE GENERATION

Testbenches must be built in a certain way to extend from the common base layer and support all the features listed in the requirements. The first important aspect is our proposed mechanism to reuse the common infrastructure via base test inheritance. Additionally the hierarchical bottom up reuse of testbenches, integration of register models and support for stimulus reuse must be setup. All these aspects are automated via a generator which has various configurations. The generator supports the building of IP testbenches as well as integrating pre-generated IP/system testbenches into a system context. The HVL part of the common infrastructure e.g. sideband, clocks and AHB is inherited from the base layer. The HDL part of the base testbench includes some BFMs for the AHB and SPI infrastructure. However the AHB/SPI slaves, sidebands and clock interfaces are optionally integrated into the derived testbench via the generator. The configuration of the common infrastructure HVL part is automated by generated code in the base test depending on the integrated BFMs. As described earlier, in a system context, the common infrastructure has to be reconfigured to accommodate all lower level testbenches. However, this reconfiguration effort is greatly reduced via the generated code in the base test.

V. RESULTS

This testbench architecture was developed in a pilot project for the verification of a CMOS Radar SoC. As illustrated in Figure 14, this resulted in a SoC testbench integrating 5 sub-systems containing over 50 IP testbenches.

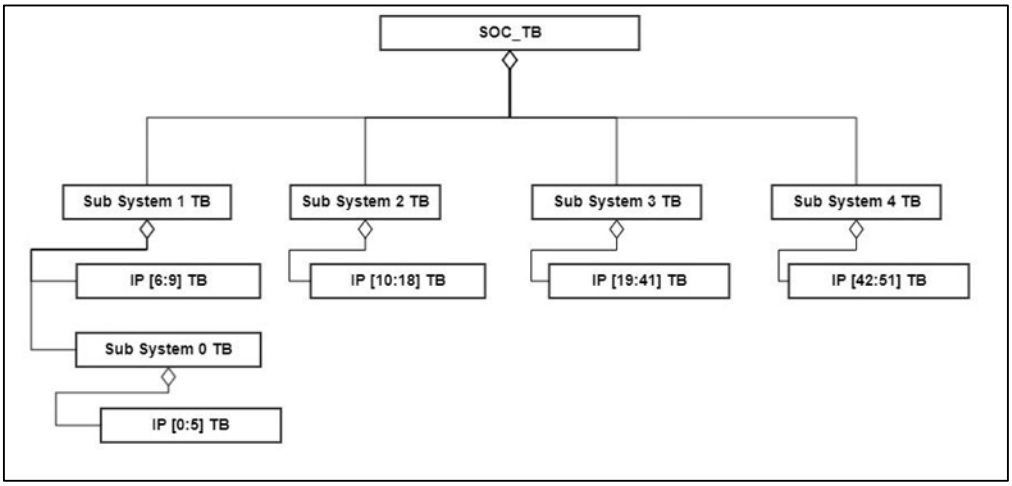

Figure 14 Pilot Project Testbench Hierarchy

The testbenches were reused without modifying or copy pasting any existing code. The sub-systems and encapsulated IPs were developed over multiple sites. Based on our experience and feedback received from all teams, this architecture vastly reduced the verification effort. This was especially effective at sub-system and Soc level as lower level testbenches were automatically integrated and with minor efforts the testbench was up and running. Stimulus reuse was particularly helpful as entire sub-subsystems and IPs could be configured and stimulated with existing sequences. Without such a mechanism, the top level team needs to put in significant repetitive effort to understand how to configure a sub-system/IP. The same applies to checks and assertions. Minimal checks need to be added on SoC level as all lower level checks are reused. The same look and feel in all testbenches not only reduces debug effort but also makes it easier/faster to understand other testbenches. Although over 50 testbenches were integrated, no significant performance impact was observed/measured. Overall the proposed architecture greatly accelerated verification progress on such a complex SoC. Compared to previous projects, the effort reduction is estimated to be at least 30%.

VI. CONCLUSION

This paper outlines a testbench architecture that enables efficient testbench development. The common infrastructure required for verification is encapsulated in a base testbench. This allows deriving testbenches to reuse the common infrastructure and avoids repetitive redevelopment. Furthermore, testbenches can not only be vertically reused at different integration levels but also horizontally in other projects. This architecture also proposes how the register model can be integrated in a bottom up development flow. Testbench generation and integration are supported by a code generator which greatly reduces the manual effort and avoids the risk of potential bugs being introduced. The resulting testbench architecture meets all the requirements defined in section II. The positive experience of using the proposed architecture in a complex SoC project has been discussed. Due to the successful utilization of this approach in the pilot project, this architecture is intended to be used in future projects.

REFERENCES

[1] A Generic Approach for Handling Sideband Signals, S. Tanvir, M. Brosch, DVCON Europe 2019.

[2] Pragmatic Reuse in a Vertical World, M. Litterick, DVCON 2013.

[3] Portable Stimulus Standard . https://www.accellera.org/activities/working-groups/portable-stimulus

[4] Using UVM Virtual Sequencers & Virtual Sequences, Clifford E. Cummings, Janick Bergeron.