Automatic SOC Test Bench Creation

David Crutchfield
CAD Engr Sr Principal
Cypress Semiconductor
Lexington, KY 40507
daac@cypress.com

Mark Glasser
Verification Architect
NVIDIA Corporation
Santa Clara, CA 95051
mglasser@nvidia.com

Stephen Roe
Elect Design Engr Sr Principal
Cypress Semiconductor
Lynwood, WA 98087
srr@cypress.com

***Abstract* - Reuse is critical to significant productivity increases in test bench development and deployment. Typically, discussions of reuse revolve around the reuse of Verification IP (VIP). Reuse of test bench structure is also important. In a platform environment where many designs are derived from a base platform, the structure of the test benches for these derived test benches will be very similar, if not identical. The best way to ensure consistency of test bench structure across a family of related designs is to use a generator to generate the structural test bench code. There is a symbiotic relationship between the overall test bench structure and the VIP employed in it. The test bench relies on the VIP it contains and the VIP relies on the structure of the test bench. This paper describes TBGen, a test bench generator, built and utilized at Cypress Semiconductor to efficiently produce test benches for its family of semiconductor products. The paper discusses the essential architectural elements of TBGen and how the generator is used.**

I. Introduction

Building a test bench by hand is a time consuming and expensive process. Furthermore, it is difficult to enforce consistency between hand-built test benches. To mitigate these issues Cypress built a test bench generator called TBGen to automate the process of building test benches. TBGen utilizes a library of VIP and structural knowledge of a family of designs the generated test benches are intended to verify. Most of the VIP is built internally, though some is purchased on the commercial market.

Designing the generator itself is a balance between generality and specific design knowledge. The generator must be general enough to generate any test bench for designs in the family. Yet, it must understand enough about the structure of the design family so that it can generate test benches specifically for those designs. One criteria of the architecture of TBGen is to make it operate as a *data driven* program to the extent possible. This enables separation between the generator structure and the generated test benches. Most of the specifics of any design are captured in various kinds of data that are input to the generator. That way the generator itself does not have to have specific knowledge of any design. The generator itself is a highly reusable piece of code, able to generate test benches for any design or potential design within the design family. The kinds of data that are consumed by the generator include system parameters and bind templates. System parameters are pieces of information that are used to drive the design itself. These include things like bus sizes, number of instances of certain units, and so forth. Typically these define structural aspects of the design. Bind templates are little pieces of code that identify how to bind VIP to RTL code. The generator ultimately resolves these to SystemVerilog bind statements. Bind templates will be described in detail later on.

III. Definition of Terms

SAS	System Architecture Specification. Spreadsheet document that details a product configuration, such as, collection of IPs, their connections, memory size, etc.
TBGen	Test Bench Generator.
VIP	Verification Intellectual Property. The test bench for a given IP.

VMS Verification Management System. Cypress created tool that manages execution of verification tasks, such as, design and test bench compilation and simulation launch.

IV. TBGEN

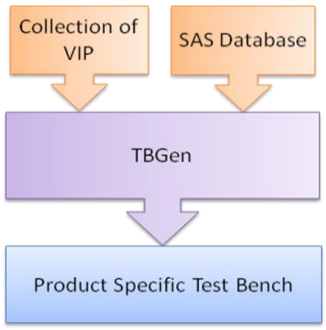

Figure 1: Overview of TBGen Flow

TBGen is a platform test bench generator script, meaning that it can generate any test bench for designs based on a hardware platform. As shown in Figure 1, TBGen relies on two main inputs. First, it utilizes a System Architecture Spec (SAS) database to obtain data about the platform design to be verified. The SAS database contains design specific information such as which subsystems are used to create the platform, instance count of each subsystem, parameters passed to each subsystem instance for configuration of busses and instances, and the system register map. Second, TBGen relies on predefined information within each VIP designed to verify each subsystem design selected. Interface or connectivity information is passed through bind templates. Bind templates contain interface specific information and a simple list of connections to be made. Additionally, integration tests are provided to automatically check proper instantiation of targeted IP. By building test structures recognized by the Verification Management System (VMS) developed at Cypress, test lists can be generated and launched automatically. Finally, TBGen requires that each VIP be built in a uniform way. If a VIP is not constructed based on TBGen guidelines then it is unusable. Once guidelines are followed and all information is provided then platform level test benches can easily be created drastically reducing verification development time.

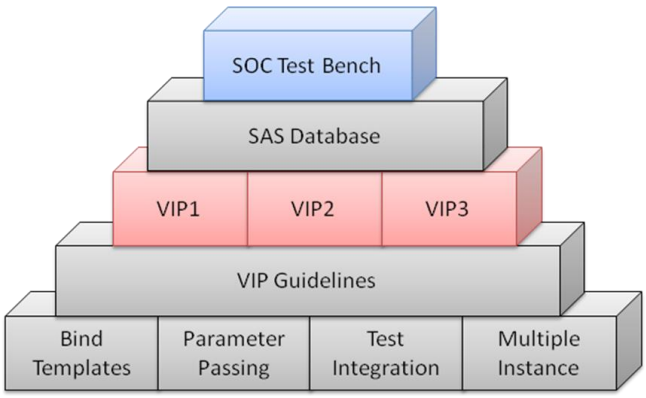

Figure 2: Building Blocks for Test Bench Generation

Figure 2 highlights the fundamental building blocks required in successfully building a platform. At the foundation are uniform constructs that any VIP must conform to or provide. These are necessary to communicate intent for interface binding, parameter handling, automatic test integration, and multiple instantiation. Each of these topics is discussed in later sections. Building on the foundation are guidelines to communicate these requirements to each verification engineer. Once engineers follow these guidelines in creating subsystem test benches a collection

of VIP for platform integration will be created. TBGen will use this collection along with data extracted from the SAS database to build a platform level test bench by properly stitching together subsystem test benches.

A. Design and Test Bench Parameter Classes

Subsystem test benches can be executed either separately or as part of a platform level test bench. They are configured for operation in a platform by design parameters extracted from the SAS. When running as a standalone test bench, the top levels of the subsystem test bench supply the same set of design parameters as the platform test bench. These design parameters are used in the subsystem test bench core to create a class containing test bench parameters. The core test bench parameter class is used by the top levels of the test bench, either SOC or subsystem, to communicate with the subsystem test bench core. This also provides a means for inheriting test bench parameters and overriding them at the platform level as necessary. Design parameters for each IP are located within a unique class named for the given IP. Test bench parameters are placed within a unique class named after the IP but prefixed with "v".

To contain all parameter classes a package named ***sys*** is provided. Both the platform test bench and VIP test bench contain this package. Platform level tests will utilize package ***sys*** generated from SAS parameter information. VIP ***sys*** packages are used for subsystem level tests only and will be replaced by the platform ***sys*** package when the VIP is integrated. Through this flow design parameters are generated for a platform and passed down to a VIP.

Test bench parameters are also included within package ***sys***. However, their definition is provided from each test bench through a class named ***v<ip>_params*** were ***<ip>*** is the name of the IP in question. At the platform level ***vsys.svh*** is generated to define classes for each VIP named after the IP but prefixed by "v". Each VIP class extends a corresponding ***v<ip>_params*** class. The ***vsys.svh*** file is included within package ***sys***, thereby, passing subsystem test bench parameters up to the platform. In some cases test bench parameters are calculated based on design parameters. In the method described here design parameters can be leveraged through IP parameter classes to augment test bench parameters as necessary.

Parameters are accessed by importing "sys::*" into the namespace where access to design or test bench parameters is required. Namespaces must be dereferenced to access each parameter.

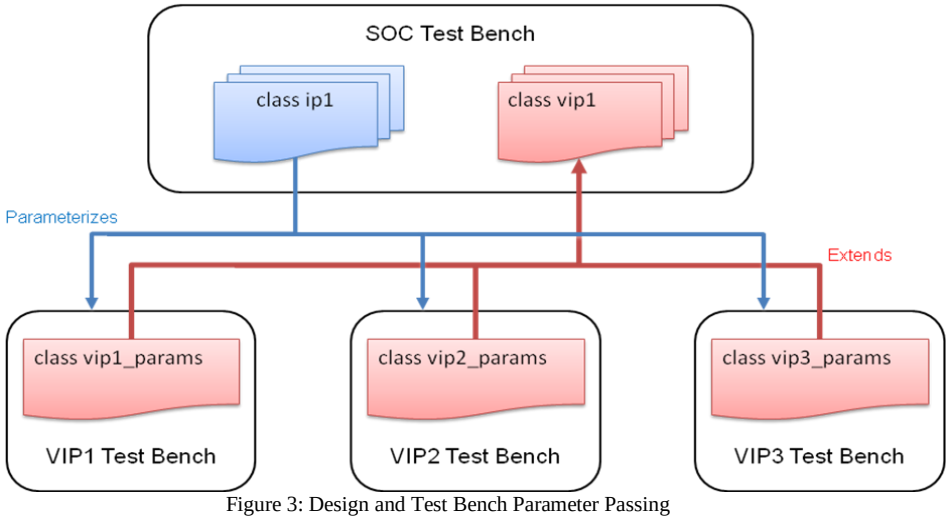

Figure 3: Design and Test Bench Parameter Passing

Consider Figure 3, which depicts design parameters passing from the SOC to each subsystem VIP. Also shown is subsystem parameter passing from each VIP to the platform test bench. In this example, subsystem IP's ***ip1***, ***ip2*** and ***ip3*** are instantiated. Therefore, in package sys there will be classes ***ip1***, ***ip2*** and ***ip3***. Included as well in package sys through ***vsys.svh*** are classes ***vip1***, ***vip2*** and ***vip3***. Below are code examples for VIP test bench parameter files and platform package ***sys***.

vip1_params.svh

```
virtual class vip1_params;
   parameter TB_PARAM1 = 100;
endclass
```

vip2_params.svh

```
virtual class vip2_params;
   parameter TB_PARAM1 = 40;
   parameter TB_PARAM2 = 10;
endclass
```

vip3_params.svh

```
virtual class vip3_params;
   parameter TB_PARAM1 = 5;
endclass
```

Figure 4: Example VIP Parameter Classes

In Figure 4, three subsystem test bench parameter classes are defined containing parameters with values specific to the VIP. Class ***vip1_params*** declares **TB_PARAM1** to be 100, class ***vip2_params*** declares **TB_PARAM1** to be 40 and **TB_PARAM2** to be 10, and class ***vip3_params*** declares **TB_PARAM1** to be 5. In Figure 5, package ***sys*** is shown to include three IP parameter classes. The three classes define **PARAM1** and **PARAM2** as shown. Additionally, package ***sys*** includes ***vsys.svh***. This file simply extends each subsystem test bench parameter class to inherit needed parameters accordingly. A simple example of parameter usage through dereference is shown in sample.sv.

sys.sv

```
package sys;
   virtual class ip1;
      parameter PARAM1 = 0;
      parameter PARAM2 = 5;
   endclass

   virtual class ip2;
      parameter PARAM1 = 15;
   endclass

   virtual class ip3;
      parameter PARAM1 = 2;
      parameter PARAM2 = 1;
   endclass

   `include "vsys.svh"
endpackage
```

vsys.svh

```
`include "vip1_params.svh"
class vip1 extends vip1_params;
endclass

`include "vip2_params.svh"
class vip2 extends vip2_params;
endclass

`include "vip3_params.svh"
class vip3 extends vip3_params;
endclass
```

sample.sv

```
import sys::*;

module sample;
   logic [ip1::PARAM2-1:0]          somebus;
   logic [vip1::TB_PARAM1-1:0]      tb_bus;
   ....
endmodule
```

Figure 5: Example ***sys.sv***, ***vsys.svh*** and Parameter Usage

B. Connections

One of the primary functions of the generator is to forge connections between the test bench and the DUT. While the set of DUTs that a generator has to deal with is constrained to a family, there can be many variations within that family. The number of instances of interface units may vary, as well as, the exact wiring of any interface. To understand how the generator deals with this we first look at a test bench/DUT connection model.

Figure 6, Test bench/ DUT connection model, illustrates a tripartite connection model that includes protocol agents, SystemVerilog interfaces, and DUT interfaces. A DUT interface is a collection of signals that operate together to move data or control in or out of the device. Each DUT interface is connected to a SystemVerilog interface. A SystemVerilog interface is a construct similar to a module that provides a means for representing a set of wires as a single object. Protocol agents in the test bench are connected to the SystemVerilog interfaces, which in turn are connected to the DUT interfaces.

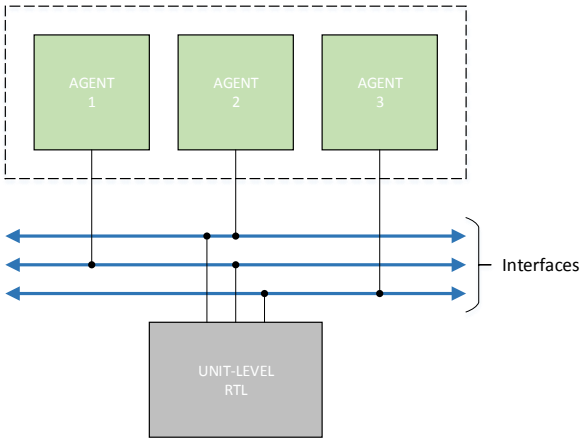

Figure 6: Test bench/DUT connection model

C. *Bind Template*

A bind methodology has been defined to allow subsystem VIP a mechanism for building bind instantiations within a top-level test bench based on the SystemVerilog bind-statement. In this methodology it is important for bind naming and instantiation to match between what the generator creates and what a subsystem VIP expects. This is required because TBGen will build field names for each bind and place them in the configuration database using the static function set_value_in_global_config. The set_value_in_global_config function essentially associates an object handle with a configuration name and places the handle in the configuration database. A subsystem VIP will need a uniform naming approach to know what configuration name to retrieve from the configuration database through static function get_value_from_config when trying to access stored bind interfaces.

To make bind instantiation simple at the platform level all bind interfaces and wrapper modules are bound in the subsystem level design scope. Interfaces and wrapper modules are self-contained meaning that there are no external code requirements outside of a simple bind instance.

Associated with each VIP is a set of *bind templates* which are used to convey bind information to TBGen. The bind template structure is defined to give each VIP owner exactly what is needed for TBGen to bind a test bench to an RTL interface. Bind templates will be used by TBGen to generate bind-statements for interfaces and/or modules at the platform level. They are also used to store virtual interfaces or non-interface modules in configuration items at the top level.

The bind template file name contains information about the interface to be instantiated. The file name will contain the bind name, which represents the interface or wrapper module to be instantiated, and an instance label, which is used in naming the bind instance. Each item is separated with a dash, "-". The bind instance will use the interface name as the module or interface to instantiate. Additionally the IP name, interface or module name, and instance label will be used to create a unique instance name. Therefore, the naming follows the convention where ***interface_name-template_label*** generates an instantiation of ***interface_name_template_label*** with the instance name ***ip_name_interface_name_template_label***.

Predetermined forms of information are stored in the bind template file. This information is contained in a section of predefined parameters, used to further specify pertinent information about the bind interface, followed by a section of port to DUT module signal connections. Table 1 shows the available bind template parameters. Each of these will be discussed in detail below.

Table 1: Bind Template Parameters

Parameter	Options	Description
bind_req	Design parameter expression(s).	Expression using design parameters for controlling instantiation.
generate / not_generate	Define label.	Define label to be wrapped around a bind instance.
interface	No	Bind wrapper module into design, but don't store a virtual interface in the config database.
	if_name, if_inst_name	Interface names and instances in wrapper

		modules
	if_name, if_inst_array_name, if_limit	Interface arrays within wrapper modules.
sas_param	Design parameter.	Design parameter to be passed during instantiation of bind.
tb_param	Test bench parameter.	Test bench parameter to be passed during instantiation of bind.

Each bind template parameter provides control over what will be created. Parameter ***bind_iterate*** indicates if a bind should be repeated with a different instance name. For instance, if ***bind_iterate*** is 2 then TBGen will instantiate two unique instances of the same interface by adding "_0" and "_1" to the instance name.

Parameter ***bind_req*** is used to qualify a bind template based on design parameters or complex arithmetic or logical expressions involving design parameters and constants. Complex expressions consist of positive integers and identifiers separated by the operator characters "?:<>=!-+*/|&" with optional whitespace. Combinations of operator characters, such as "<=" and "&&" are allowed. If the specified expression result is not true then the bind instantiation will be ignored. Multiple ***bind_req*** lines can be given. If so, then all ***bind_req*** lines must be met. This is a logical AND of ***bind_req*** expressions. Additionally, a ***bind_req*** line can have multiple comma separated expressions. If any expression result is true within a ***bind_req*** line then the bind will be instantiated assuming other ***bind_req*** lines are satisfied as well. This is a logical OR of ***bind_req*** expressions.

Using parameters ***generate*** or ***not_generate*** allow for bind instances to be controlled through preprocessor directives. The parameters will indicate `ifdef <***Define label***> and `ifndef <***Define label***> respectfully. At simulation launch time +define+<***Define label***> can be given as needed.

Parameter ***interface*** allows for control of several things. First, in some cases a non-interface wrapper or module needs to be bound into another module but should not be placed in configuration items. An example would be an assertion module. This is controlled by setting ***interface*** = no. Second, wrappers being bound will likely have interfaces within them that need to be accessed through the configuration. In this case ***interface*** can be set with two comma separated strings where the first string is the interface name and the second is the expected interface instance within the wrapper. Given ***interface*** = ***if_name***, ***if_inst_name*** the configuration item name would be ***ip_name_interface_name_template_label_if_name_if_inst_name*** where the module instance it refers to would be ***ip_name.interface_name.template_label.if_name.if_inst_name***. Third, a wrapper may contain an array of interfaces. If so, ***interface*** can be set with three comma separated strings. The third string is the interface array limit. Given ***interface*** = ***if_name***, ***if_inst_name***, ***if_limit*** the configuration item name would be ***ip_name_interface_name_template_label_if_name_if_inst_name[if_limit_index]*** where the corresponding module instance would be ***ip_name.interface_name.template_label.if_name.if_inst_name[if_limit_index].***

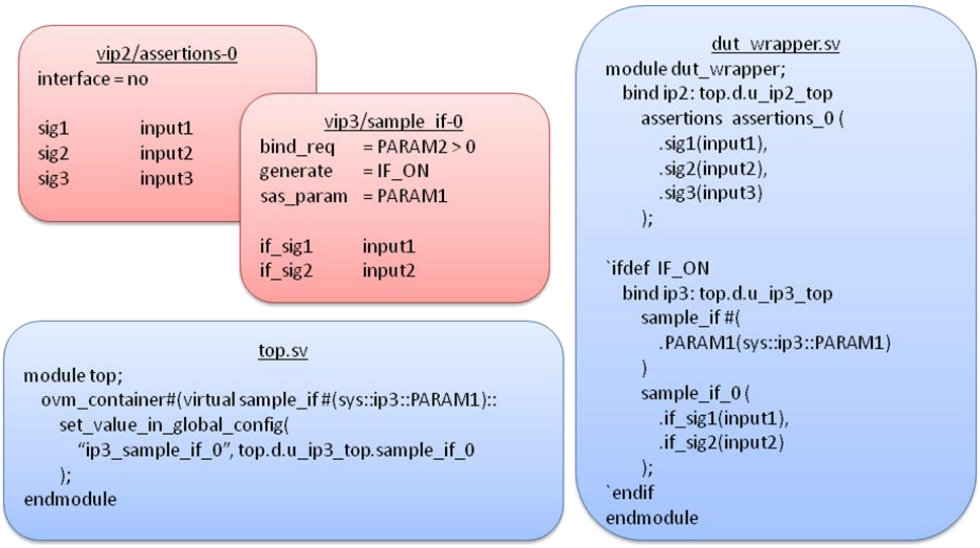

Figure 7: Bind Template Example

Figure 7 includes two simple bind template examples. First, ***vip2*** declares a bind template for an assertions module. The module is to be bound into ***ip2*** but is not an interface and should not have a configuration handle and

name defined. This is indicated with ***interface = no***. There are three signal connections provided. The assertion module signals ***sig1***, ***sig2***, and ***sig3*** are to be connected to ***input1***, ***input2***, and ***input3*** respectfully. Second, ***vip3*** declares a bind template for an interface named ***sample_if***. This interface is to be bound into ***ip3*** if ***ip3*** class parameter **PARAM2** is greater than zero. This is required based on ***bind_req = PARAM2 > 0***. Referencing Figure 5, **PARAM2** is 1 for ***ip3***. Interface ***sample_if*** should be wrapped with an ***`ifdef*** statement using **IF_ON** as indicated with ***generate = IF_ON***. Furthermore, ***ip3*** design parameter **PARAM1** is to be passed to the interface during instantiation based on ***sas_parmater = PARAM1***. There are two connections provided. The interface signals ***if_sig1*** and ***if_sig2*** should be connected to ***input1*** and ***input2*** respectfully.

At the platform level TBGen will take these bind templates and create the correct bind instances. This can be shown in ***dut_wrapper.sv*** where ***assetions_0*** is bound into module ***if2*** instance ***u_ip2_top*** and ***sample_if_0*** is bound into ***ip3*** instance ***u_ip3_top***. TBGen will also create a set_value_in_global_config function call for ***sample_if_0*** but not ***assertions_0***.

D. Design Multi-Instance Support

Some subsystems may have multiple instances in an SOC, and each subsystem instance may instantiate a different module hierarchy. For example, one instance of a communications subsystem might support I2C while a second instance might not. The subsystem testbenches for these two communications subsystem instances would need to build and connect themselves differently. The first testbench would need objects and interfaces to monitor I2C traffic, while the second would not.

Cypress uses design parameters to control the differences in the internal structure of multi-instance subsystems and multi-instance subsystem testbenches. Design parameters are passed to the subsystem module during instantiation, thereby, configuring the sub-module hierarchy. In the subsystem testbench, design and testbench parameter classes are used to control instantiation of interfaces and the build and connect phases.

Single instance subsystems can do this with hard-coded references to design and testbench parameters in the sys package, but multiple instance subsystems cannot because each instance has its own parameters. Instead, multiple instance subsystems have unique design parameter classes that are instance specific. These design parameter classes are used to parameterize the ***v<ip>_params*** class from which the testbench parameter class is derived. Each multiple instance subsystem testbench environment class instance is then parameterized with its own design and testbench parameter class. Once the subsystem environment class has access to its own design and testbench parameter classes, it can use those parameters to control its own build and connect phases and also parameterize any of its children than need access to design or test bench parameters.

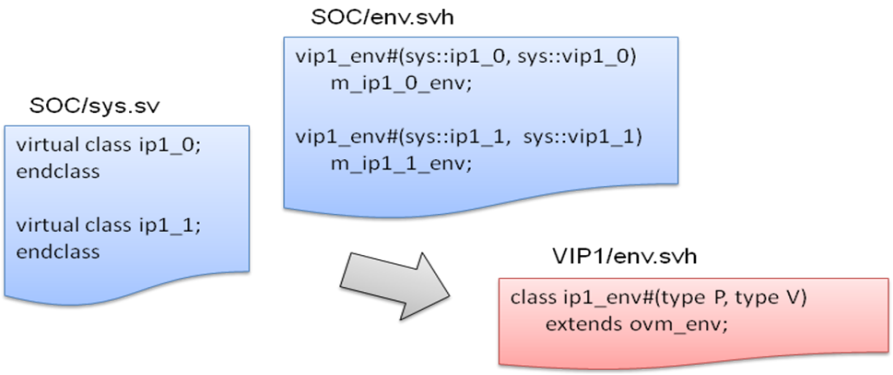

Figure 8: Multi-Instance Example

Figure 8 presents a simple example where the top-level environment, env.svh, is shown for both the SOC and subsystem test bench. The subsystem environment ***vip1_env*** is instantiated twice in the SOC environment. The first instantiation passes in design and test bench parameters as ***sys::ip1_0*** and ***sys::vip1_0*** for instance 0. The second instantiation passes parameters in as ***sys::ip1_1*** and ***sys::vip1_0*** for instance 1. With this methodology class ***ip1_env*** receives the proper classes and can be configured properly for each instantiation.

E. Subsystem Level Integration Tests

To enable full automation, TBGen must go beyond creating a working platform level test bench. It must also create or enable automatic regression generation. This is accomplished by dividing responsibilities between TBGen and another Cypress developed tool named VMS (Verification Management System). There are two main

responsibilities. First, TBGen looks for integration sequences and tests within a subsystem test bench to copy over into the platform test bench. Second, VMS must automatically build a regression list and launch tests accordingly.

Subsystem VIP developers can create folders named ***integration_tests*** and ***integration_seqs*** in predefined locations within a VIP's directory structure. TBGen will locate these folders and automatically copy them into the platform test bench directory structure. All sequences and tests are placed under a unique directory for each type (***seqs*** or ***tests***). Integration sequence or test folders from a subsystem VIP will be placed in a folder named after the VIP. For instance, if ***integration_seqs*** is found in ***vip1*** then ***seqs/vip1/integration_seqs*** will be created within the platform test bench. In the same way, if ***integration_tests*** is found in ***vip1*** then ***tests/vip1/integration_tests*** will be created. Once all integration sequences and tests have been copied over, TBGen will build package ***seqs*** and package ***tests***. These packages will include all .svh files found in each case. Figure 9 presents an overview of this process.

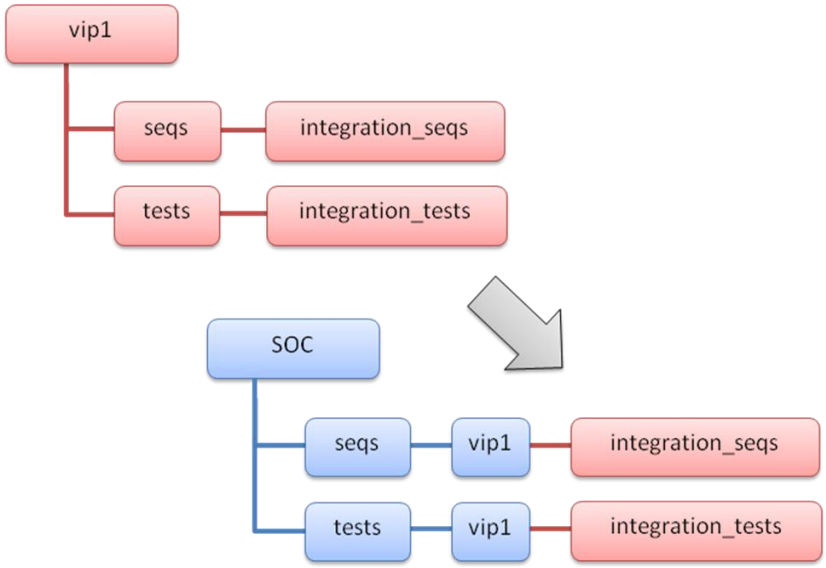

Figure 9: Integration Sequences and Tests in Platform Test Bench

Cypress' VMS tool is leveraged to automatically create a regression list. VMS was developed to handle file compilation, test list creation, and test launch. Details on how this is done are outside the scope of this discussion. For test list creation, the basic concept is that VMS can traverse a ***tests*** folder, referred to as a test tree, and identify test names based on configuration information and folder structure. The test names and configuration information are combined to build a test list with appropriate arguments. Once created VMS will use the test list to launch each found test.

F. Built-in Register Package Generation

Cypress' test benches use a centralized register package methodology based on Mentor Graphics' Register Assistant tool. This tool generates register packages based on flat files. The SAS is used to create register descriptions through flat files that can be feed into Register Assistant. At the subsystem level, register package generation is driven by the SAS register descriptions for the subsystem. At the SOC level, register package generation is driven by the SAS register descriptions for the subsystems contained in the SOC. Since the same descriptions are used to generate a subsystem's register block at both the subsystem and SOC levels, register accesses can be modeled consistently at both levels.

Different design parameters can cause variations between a subsystem's register block at the subsystem level and in that of different SOCs. The register generation scripts handle this by operating in two phases. The first phase reads the SAS register descriptions and the design parameters for a given subsystem. The design parameters control which modules are instantiated within the subsystem and also which sections of subsystem registers are written to intermediate register flat files. Once the specific flat files have been created for the design at either the subsystem or SOC level, the second phase converts the information in the flat files into the actual field, register, register block,

and memory map classes in the register package using Mentor Graphic's Register Assistant. Figure10 provides a high-level flow of register package generation.

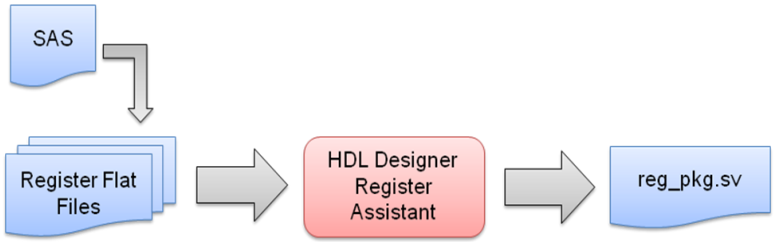

Figure 10: Register Package Generation Flow

VI. CONCLUSION

TBGen provides an automated environment to build an SOC level test bench using a library of prerequisite verification IP. It does this by handling subsystem test bench instantiation and configuration in a uniform way. Using this tool at Cypress has reduced the time to create new SOC level test benches, whether they are derivatives or not, from weeks to minutes. Additionally, with automatic test integration and VMS, SOC level test bench creation and regression launch can be accomplished in two simple steps, launch TBGen and launch VMS. While extra effort is needed to prepare verification IP for TBGen uniformity the benefits are substantial.