Traversing the Interconnect: Automating Configurable Verification Environment Development

Prashanth Srinivasa
LSI India R&D Pvt. Ltd
GTP, Devarabeesanahalli
Outer Ring Rd, Bangalore, India
(+91)80-41979643
Prashanth.Srinivasa@lsi.com

Mathew Roy
LSI India R&D Pvt. Ltd
GTP, Devarabeesanahalli
Outer Ring Rd, Bangalore, India
(+91)80-41979550
Mathew.Roy@lsi.com

ABSTRACT

There are several challenges in verifying a complex SoC (System on Chip) on time, like frequent specification changes, which include architectural and protocol changes that impact both verification effort and the delivery schedules. As an example, a SoC of ~140 M gates currently we are working on consists of large sub-chip components having hierarchical interconnects, needs to be comprehensively verified. The number of verification environments that need to be created and maintained for all the sub-chip designs is another challenge. Even if reusable components like VIPs (Verification Intellectual Property) are used, there is a considerable amount of effort involved in their integration, handling communication with other components, verifying the same and so on. To tackle these challenges, the need of the hour is to have a generic infrastructure which can be reused for various environments corresponding to the sub-chip designs. This paper presents such an infrastructure which supports different bus protocols with an intuitive interface to the user for integrating multiple diverse components. The paper also talks about the additional facilities provided by this infrastructure, like the built in generic scoreboard for data integrity checks and the performance analyzer which can be used to measure the performance of the DUT (Design Under Test). To provide such infrastructure for variety of sub-chip components, a highly configurable environment is built using the powerful features of contemporary verification methodologies.

The configurable environment can be used to verify large sub-chip designs having interconnects, bridges with any number of master and slave interfaces of different protocols. The environment has built-in support for standard protocols and provides facilities for the users to add their own custom components, with minimum effort. It is built at a higher abstraction level but hooks are provided to access any component at any hierarchy. The environment has a list of VIPs of different standard protocols and each VIP is associated with a configuration descriptor. Each configuration descriptor contains generic information like its kind, protocol, address map, bus widths, supported targets, etc. as well as protocol specific configuration (like reordering depth, in case of AXI interface). Based on the configurations, appropriate VIPs get activated with the specified behavior in the configuration. The corresponding communication interface with the other components like scoreboard and a default random scenario to the valid slaves are also generated in an automated manner and the paper describes how this is done. The environment has a performance analyzer too, to measure the performance of DUT like traffic and arbitration when it is enabled.

The scoreboard used in the environment is a byte level generic scoreboard which verifies the data integrity across interfaces of different protocols/bus widths. Also, the scoreboard is made efficient in terms of runtime memory, considering the complexity of the environments. The paper talks about how these features are implemented. Additionally it talks about leveraging standard off-the shelf available methodology features to handle various diverse requirements such as configurable 'end of test' detection and traversing the environment hierarchy.

Categories and Subject Descriptors

[System on Chip Verification]: Generic Interconnect Verification Infrastructure – Architecture and flow, methodology constructs

General Terms

Verification.

Keywords

SoC, System Verilog, VMM, Reuse, VIP, Abstraction.
RTL: Register Transfer Language
SoC: System on Chip
VMM: Verification Methodology Manual, by Synopsys
OVM: Open Verification Methodology
UVM: Universal Verification Methodology, by Accellera
VIP: Verification Intellectual Property

1. INTRODUCTION

Though it is not desired, but inevitably the specification of a product keeps changing based on customer requirements, performance, timing, etc. This may lead to change in the protocols, change in the number of interfaces of the sub-systems, change in the number of sub-systems and so on to meet the requirements. A typical complex SoC as shown in Figure 1., consists of multiple sub-systems of same and different kinds and each of them needs to be verified efficiently. It is difficult to develop and maintain a large number of verification environments at different levels in a given time. With the usage of standard verification methodologies like VMM/OVM/UVM and reusable VIPs, verification environments can be developed faster, but still there is a considerable amount of effort involved like integration, building scoreboards, maintaining each environment separately, to name a few. Again whenever the specification changes, accommodating corresponding changes in the environments requires considerable effort. This paper describes how verification environments can be developed with a very little effort using the configurable generic interconnect infrastructure and how they can be made almost immune to the specification changes.

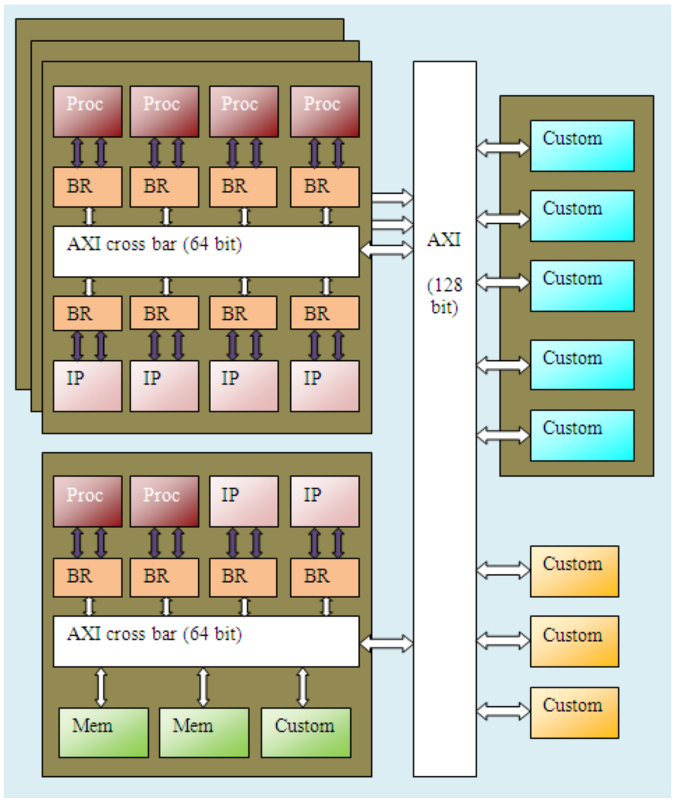


Figure 1. Typical SoC (System on Chip) example

2. GENERIC INTERCONNECT VERIFICATION ENVIRONMENT

To address the issues discussed above, a highly configurable generic verification environment is developed with built-in support for standard protocols. The environment is built using VMM methodology, but there is no restriction to develop this environment with OVM/UVM since almost all features used in this environment are available in these methodologies as well. The architecture of the environment is as shown in Figure 2.

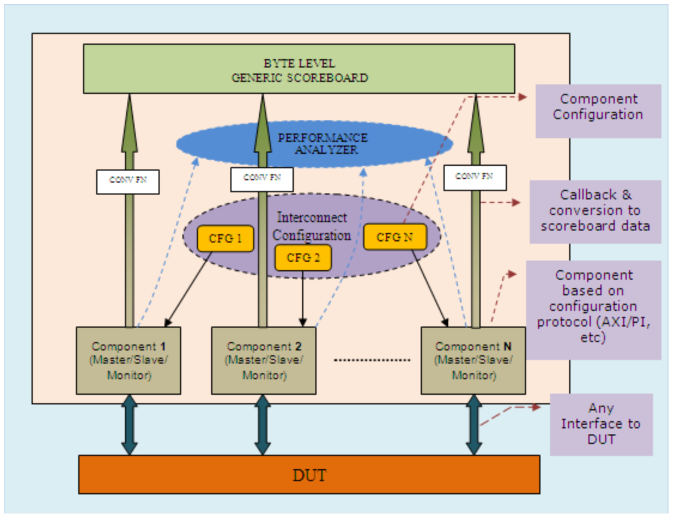

BYTE LEVEL GENERIC SCOREBOARD
PERFORMANCE ANALYZER
CONV FN
Interconnect Configuration
CFG 1
CFG 2
CFG N
Component 1 (Master/Slave/Monitor)
Component 2 (Master/Slave/Monitor)
Component N (Master/Slave/Monitor)
DUT
Component Configuration
Callback & conversion to scoreboard data
Component based on configuration protocol (AXI/PI, etc)
Any Interface to DUT

Figure 2. Generic Interconnect Verification Environment

The Generic Interconnect Verification Environment has a set of configuration descriptors to control the behavior of the environment and its components. It encapsulates a list of VIPs for each protocol and they get initialized based on the configuration. There is a protocol independent byte level scoreboard which performs data integrity checks. The environment has a performance analyzer also for monitoring bus and arbitration performances when enabled. Communication between generators and drivers are handled through channels and for the rest of the components, callbacks are used. All the components of the environment are built hierarchically with parent-child relationship so that any component can be accessed using *vmm_object* utility methods and *vmm_opts* from the test case.

The Generic environment execution flow is same as the vmm_env where configuration descriptors are built in the method *vmm_env::gen_cfg()*. Based on the configuration, appropriate VIPs get initialized in *vmm_env::build()* and get registered to the consensus mechanism. Then the test execution continues in subsequent phases and the end of test is determined by the consensus mechanism after all the registered VIPs are done with their execution. Finally scoreboard provides a summary of the transactions and the result of the test. Additionally, performance analyzer if enabled provides summary of the traffic details and an SQL report on arbitration and DUT performance.

2.1 Configuration

Configuration is the core of the Generic Environment through which all the components are initialized and controlled. There is an environment specific configuration descriptor which has parameters to control the environment execution. These parameters use *vmm_opts* and hence they can be set from the test case or command line. The environment configuration has a list of component specific configurations. Each component configuration controls the kind and behavior of the corresponding component/VIP. It contains information like the protocol (AXI, AHB, OCP, etc), behavior (master or slave, active or passive, etc) and other information such as address range. It also contains protocol specific parameters like reordering depth in case of AXI and any such parameters can be set through *vmm_opts*. There is a set of predefined methods in the environment configuration through which user can easily add configuration for all the components. In addition to standard protocols, provision is made to add configuration for custom components as well. Based on the configuration, appropriate VIPs get initialized with the specified behavior.

Another feature in the configuration is that the user can specify a set of valid slaves for a master. This information is taken by random scenario generator to generate traffic only to those slaves.

2.2 Scoreboard

The Generic Scoreboard in the environment is protocol independent and performs data integrity checks. It has a set of byte level scoreboard units, one for each slave as shown in Figure 3. Any protocol specific transaction appearing at a master/slave is broken down into a list of protocol independent scoreboard data items, each of which has byte data and its address. Since the data of any bus widths are converted to a set of bytes and used for comparison, scoreboard can process transactions of different bus widths. For the supported protocols, communication from monitor to scoreboard is done using VMM callbacks and converting transactions to scoreboard items is taken care automatically in the environment itself, i.e., just by providing configuration information, user can make use of scoreboard without any additional effort. For adding custom component information to the scoreboard, there are intuitive methods like *write_at_master()*, *write_at_slave()*, *read_at_master()*, *read_at_slave()* which help the user to call these methods without requiring any knowledge about internal details of the scoreboard. Also it has methods like *final_check()* for checking if there are any pending transactions and *report()* for providing a summary of the report.

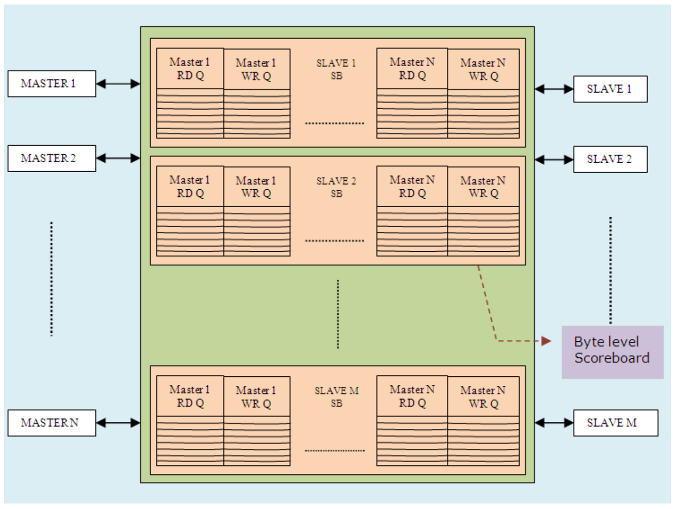
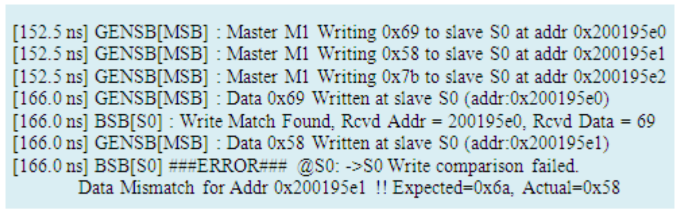


Figure 3. Generic Scoreboard structure

Every byte level scoreboard unit handles write data check and read data check separately. Whenever a write transaction occurs at the master, master transaction is broken into a list of byte level scoreboard data (*genIc_sb_data*) which has address and byte information. This list is saved in the expected slave scoreboard with the master id. When the transaction reaches the slave, the slave transaction is broken into a list of scoreboard data (*genIc_sb_data*) and this list is compared with that of the master. If there are multiple master ids present in the expected slave scoreboard, search happens in all the lists. In case of read transaction, read request transaction at the master is broken into a list of byte level scoreboard data (*genIc_sb_data*). Data in this case are don't cares since it is just a request. This list is saved in the expected slave scoreboard with the master id. When the read happens at the slave, this list is updated with the data available at the slave. In the end, when the response reaches master, transaction is again broken into a list of scoreboard data (*genIc_sb_data*) and compared with the ones in the expected list. Compared items in both write and read transactions are deleted immediately from the scoreboard.

For complex sub-system environments, there could be numerous scoreboard data items at a time during the simulation. Hence data item class for the scoreboard (*genIc_sb_data*) was made light weight by not extending it from *vmm_data* as *vmm_data* extends *vmm_object* and also has notification events which are not used by the scoreboard data item.

The scoreboard takes information of different components through the configuration and provides intuitive messages about the progress of simulation as shown in Figure 4. It gives information about which master has started transactions, where the transaction is at specific simulation time, etc.


```
[152.5 ns] GENSB[MSB] : Master M1 Writing 0x69 to slave S0 at addr 0x200195e0
[152.5 ns] GENSB[MSB] : Master M1 Writing 0x58 to slave S0 at addr 0x200195e1
[152.5 ns] GENSB[MSB] : Master M1 Writing 0x7b to slave S0 at addr 0x200195e2
[166.0 ns] GENSB[MSB] : Data 0x69 Written at slave S0 (addr:0x200195e0)
[166.0 ns] BSB[S0] : Write Match Found, Rcvd Addr = 200195e0, Rcvd Data = 69
[166.0 ns] GENSB[MSB] : Data 0x58 Written at slave S0 (addr:0x200195e1)
[166.0 ns] BSB[S0] ###ERROR### @S0: ->S0 Write comparison failed.
         Data Mismatch for Addr 0x200195e1 !! Expected=0x6a, Actual=0x58
```
Figure 4. Scoreboard output in the simulation log

2.3 Scenario Generation

When a component is configured as a master, a scenario generation mechanism gets enabled with a default random scenario. The default random scenario gets target information from the configuration and automatically generates random transactions to the valid slave ranges. Number and speed of transactions can be controlled from the test case or from the command line through vmm_opts. Thus multiple masters can concurrently generate random traffic to valid slaves including their range boundaries with different speeds. Default random scenarios are available for all supported protocols.

Scenario generation mechanism uses VMM multi-stream scenario generator and hence the user can replace default scenario with customized scenarios for covering corner cases like accessing the same location by multiple masters, etc. This also allows synchronization of multiple scenarios of different generators.

2.4 Performance Analyzer

The generic environment has a performance analyzer to analyze the arbitration and bus performance of the DUT. A user can take its benefits just by enabling a runtime switch. It uses two instances of *vmm_perf_analyzer* class, one for arbitration performance and another for the bus performance. The performance analyzer can be enabled during the runtime using *vmm_opts*. It provides details of bus utilization in SQL format. SQL processing tools can be used to view the performance analysis as shown in Figure 5.

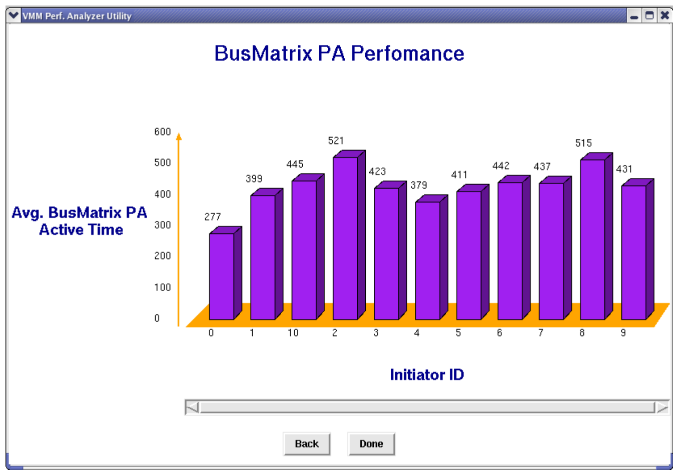


Figure 5. Bus performance: Average active time of each master

The generated report provides information such as arbitration time, bandwidth, average and peak latency for each initiator and so on. This will help to find any issues with the performance of the DUT.

For analyzing the bus performance, amount of bytes transferred across the bus is considered and for analyzing arbitration performance, request to grant time is considered. Connection between the monitors and the performance analyzer is taken care for supported standard protocols. If any new master of unsupported protocol is added to the environment, then performance analyzer methods have to be called explicitly by the user.

3. USING GENERIC INTERCONNECT VERIFICATION ENVIRONMENT

This section describes steps to build a verification environment using the Generic Interconnect Verification environment and explains how easy it is to incorporate specification changes, taking an example of one of the subsystems we are verifying.

The subsystem initially had some master and slave interfaces with AHB and APB protocols, some with proprietary standard interfaces and few custom interfaces. A simplified version of the DUT for the sake of our discussion is shown in Figure 6. Due to performance requirements in our project, specification got changed with AXI replacing the AHB interfaces besides addition of more interfaces. A simplified version of the revised DUT is as shown in Figure 7. Below sub-sections describe the steps followed to develop verification environment for the initial DUT and the effort taken to change the environment for the revised DUT.

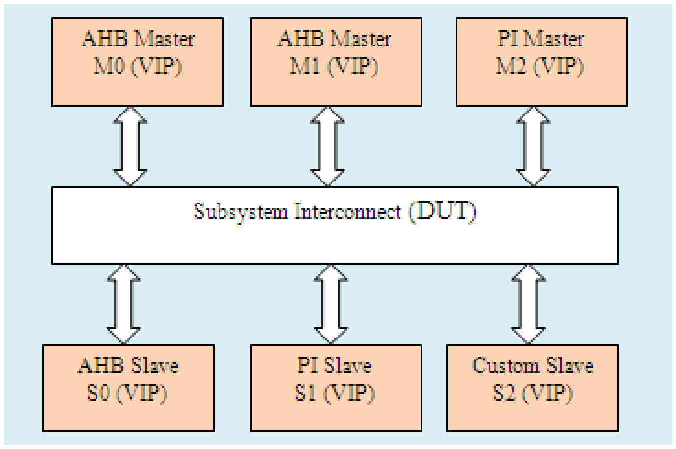


Figure 6. Initial Subsystem Interconnect with AHB interfaces

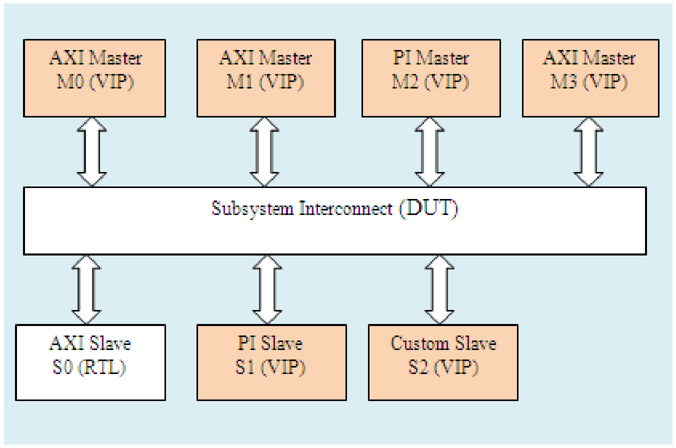


Figure 7. Revised Subsystem Interconnect with AXI interfaces

3.1 Verification Environment for Initial Subsystem

The verification environment development for the initial version on the DUT as shown in Figure 6. was done as follows.

3.1.1 Creating configuration

The generic interconnect configuration base class was extended and all the component specific configurations were added inside the method *generate_config ()*. This was done using the predefined method add_vip_cfg () as shown in Figure 8. Each component was added with unique instance name, its protocol, behavior and other required information. All the masters and slaves were configured as *genIc_vip_cfg::MST_ACTIVE* and *genIc_vip_cfg::SLV_ACTIVE* so that appropriate BFMs get initialized. Address range was specified for each slave. For adding configuration for any component of unsupported protocol, *genIc_vip_cfg::CUSTOM* was specified for protocol argument. For each master, valid slaves were specified using the predefined methods *add_target_to_master()* and *add_all_targets_to_master()*. For overriding any default protocol specific configuration parameter for any component, *vmm_opts* methods were used in the constructor.

```
class dut_env_config  extends genIc_config;

   function new(string name, bit [63:0] offset=0);
      super.new(name,  offset);
     //Overrides the address width of master M0 to 64
      vmm_opts::set_int("%*:M0:addr_width",  64);
   endfunction

    function void generate_config();
      add_vip_cfg("M0",  genIcVipCfg::AHB,  genIcVipCfg::MST_ACTIVE);
      add_vip_cfg("M1",  genIcVipCfg::AHB,  genIcVipCfg::MST_ACTIVE);
      add_vip_cfg("M2",  genIcVipCfg::PI,    genIcVipCfg::MST_ACTIVE);

      add_vip_cfg("S0",  genIcVipCfg::AHB,   genIcVipCfg::SLV_ACTIVE,
                                             0, 32'h0_0000,  32'h1_3FFF);
      add_vip_cfg("S1",  genIcVipCfg::PI,    genIcVipCfg::SLV_ACTIVE,
                                             0, 32'h1_4000, 32'h2_3FFF);
      add_vip_cfg("S2",  genIcVipCfg::CUSTOM,  genIcVipCfg::SLV_ACTIVE,
                                             0, 32'h3_4000, 32'h4_3FFF);
      add_all_targets_to_master("M1");
      add_all_targets_to_master("M2")
      add_target_to_master("M0",   "S1");
      add_target_to_master("M0",   "S2");

   endfunction

endclass
```

Figure 8. Configuration class

3.1.2 Adding configuration to the Generic Environment

```
class dut_env  extends  genIc_env;

   dut_env_config cfg;

   function new();
      super.new("dut_env");
      cfg = new("dut_env_cfg");
      this.set_config(cfg);
   endfunction

   virtual function void build();
      super.build();
     //Add custom components if any
   endfunction

endclass
```

Figure 9. Configuration addition to Environment Class

The generic interconnect configuration instance is added to the generic environment using a predefined *set_config ()* method as shown in Figure 9.

3.1.3 Providing Interface connectivity for the BFMs.

Connecting the interfaces to appropriate VIPs was done using the virtual ports arrays existing in the Generic environment as shown in Figure 10. This was simple since virtual ports array was indexed with the instance names of the VIPs.

```
program P;
dut_env env;
initial begin
    env = new;
    env.ahb_ports[“M0”] = top.ahb_M0_if;
    env.ahb_ports[“M1”] = top.ahb_M1_if;
    env.pi_ports[“M2”]    = top.pi_M2_if;
    env.ahb_ports[“S0”]  = top.ahb_S0_if;
    env.pi_ports[“S0”]     = top.pi_S0_if;
end
endprogram
```

Figure 10. Connecting VIP ports to DUT interface

As discussed before, all the scoreboard connections for the supported protocols were managed automatically without requiring any additional effort. However, for adding custom components to the environment and scoreboard, the next set of steps was followed.

3.1.4 Adding Custom components.

For reaping the benefits of the Generic environment, configuration for custom VIPs was added. VIP was instantiated in build () method.

```
class  custom_port extends genIc_sb_port;

       virtual function void convert_to_sb(vmm_data tr,
                                           output genIc_sb_data sbQ[$]);
            //Add conversion code
       endfunction
 endclass

class custom_callback extends  custom_callbacks;
     genIc_scoreboard sb;

     virtual function tr_wr_call(custom_mon mon_trans);
        sb.write_at_slave(inst, mon_trans);
     endfunction

      virtual function tr_rd_call(custom_mon mon_trans);
       sb.read_at_slave(inst, mon_trans);
     endfunction
endclass

class dut_env extends genIc_matrix_env;

    virtual function void build();
         ….
         custom_port prt = new();
         custom_callback cbk = new(“S2”, sb);
         sb.add_sb_port(“S2”, prt);
         custom_mon.append_callback(cbk);
    endfunction

endclass
```

Figure 11. Scoreboard connectivity for a custom component

For adding the VIP information to the generic scoreboard, following steps were followed as shown in Figure 11:

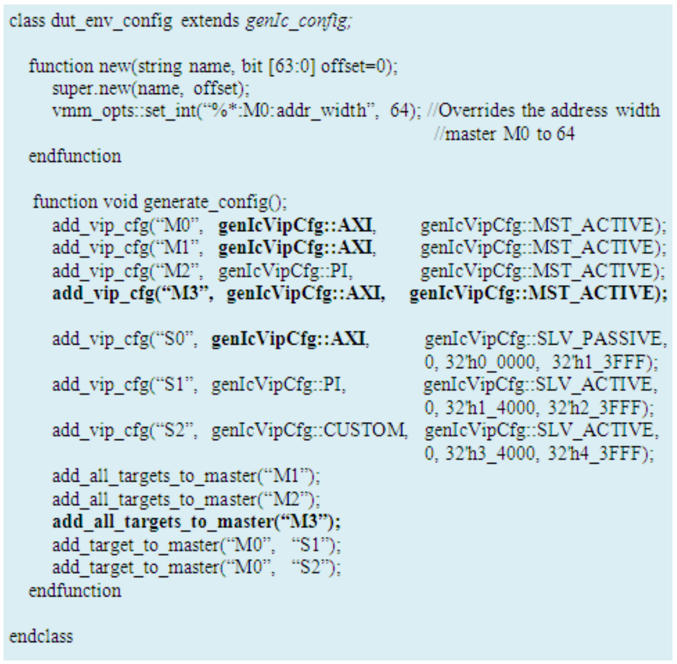
1. Conversion function which converts protocol specific transaction to a set of scoreboard transactions was added by extending the base class *genIc_sb_port*.
2. Callback class was created extending the callback class provided by the custom VIP monitor to call scoreboard methods, *write_at_slave()* and *read_at_slave()*.
3. Conversion port instance was added to the scoreboard with the instance name and callback instance was connected to the custom VIP monitor.

3.1.5 Creating test cases and starting verification.

For creating test cases, the environment is instantiated in a program block as per VMM methodology guidelines and *vmm_test* mechanism was used.

3.2 Changes in Verification Environment due to Revised Subsystem

When the sub-system specification changed as in Figure 7., addressing the changes in the verification environment was easy since only modifications required were in the configuration and the interface connection. The changed configuration for the new specification is as shown in Figure 12. Also, when actual RTL was ready at S0, the slave VIP at S0 had to be replaced with RTL; and a monitor VIP had to be connected to capture the traffic information. All these modifications were done in the configuration itself just by changing the type from AHB to AXI for M0 and M1, changing the behavior mode of S0 component from SLV_ACTIVE to SLV_PASSIVE and specifying configuration information for M3.



```
class dut_env_config  extends genIc_config;

   function new(string name, bit [63:0] offset=0);
      super.new(name,  offset);
      vmm_opts::set_int(“%*:M0:addr_width”,  64); //Overrides the address width
                                                   //master M0 to 64
   endfunction

    function void generate_config();
      add_vip_cfg(“M0”,  genIcVipCfg::AXI,     genIcVipCfg::MST_ACTIVE);
      add_vip_cfg(“M1”,  genIcVipCfg::AXI,     genIcVipCfg::MST_ACTIVE);
      add_vip_cfg(“M2”,  genIcVipCfg::PI,      genIcVipCfg::MST_ACTIVE);
      add_vip_cfg(“M3”,  genIcVipCfg::AXI,   genIcVipCfg::MST_ACTIVE);

      add_vip_cfg(“S0”,  genIcVipCfg::AXI,       genIcVipCfg::SLV_PASSIVE,
                                                  0, 32'h0_0000,  32'h1_3FFF);
      add_vip_cfg(“S1”,  genIcVipCfg::PI,        genIcVipCfg::SLV_ACTIVE,
                                                  0, 32'h1_4000, 32'h2_3FFF);
      add_vip_cfg(“S2”,  genIcVipCfg::CUSTOM,  genIcVipCfg::SLV_ACTIVE,
                                                  0, 32'h3_4000, 32'h4_3FFF);
      add_all_targets_to_master(“M1”);
      add_all_targets_to_master(“M2”);
      add_all_targets_to_master(“M3”);
      add_target_to_master(“M0”,   “S1”);
      add_target_to_master(“M0”,   “S2”);
   endfunction

endclass
```

Figure 12. Configuration Class for the revised subsystem.

4. RESULTS

The Generic Interconnect Verification environment enabled us to quickly build verification environments for different subsystems with lesser resources, thus increasing productivity. It took around 6 staff weeks to build the Generic environment. Typically it took around 2 staff days to build a verification environment using the Generic Interconnect environment which would have taken around 2-3 staff weeks otherwise. It has avoided duplication of efforts both in developing and maintaining the verification environments for each sub-chip. The chip we are currently working on has 5 sub-chips (clusters) which are being verified at various locations. By having one environment that can be scaled and enhanced to suit specific components, significant saving in terms of effort and schedule is achieved. The same generic environment can be upgraded to support custom protocols, by using the corresponding VIP which anyway will have to be developed. We are also planning for enhancements like a set of generic scenarios in addition to the currently available default scenario.

5. ACKNOWLEDGMENTS
Our thanks to Suresh Bandaru and Jayendra Dwaraka Bhamidipatti for their guidance and Amit Sharma (Synopsys) for providing us the support required for this environment.

6. REFERENCES
[1] Janick Bergeron, Eduard Cerny Alan Hunter and Andrew Nightingale. 2006. Verification Methodology Manual for System Verilog