Developing a Gigabit Ethernet VIP Using the Plan to Closure Methodology Featuring SystemVerilog

By TATA ELXSI Limited

In this era of reusable IP, multi-million gate ASICs and SoC designs, verification has become the major bottleneck because it consumes a significant portion of overall development efforts. Thus, it has become even more challenging for us to know when the verification of a design is complete and has met specified implementation criteria. These increasing design complexities have driven us to look at higher abstraction levels and modular approaches to help us develop an effective verification plan for our verification IP development.

Verification reuse is one area we are focused on because it enhances reliability and saves a tremendous amount of time, especially when developing a complex protocol such as a Gigabit Ethernet. In our case we used the Cadence Incisive Plan-to-Closure Methodology that features the Universal Reuse component of Plan-to-Closure with SystemVerilog. Lets take a closer look at the protocol we were developing and many features of this methodology we incorporated into our flow that helped us successfully complete the task.

Getting started on our Gigabit Ethernet project

Gigabit Ethernet is a transmission technology based on the Ethernet frame format and protocol. Its wide applications include Local Area Networks (LANs) for transmission at higher data-rates. It is defined in IEEE standard-802.3z. GbE employs the CSMA/CD & MAC protocol and frame format similar to its predecessors (10Mb/s and 100Mb/s). Because of these attributes as well as support for Full-duplex operation, Gigabit Ethernet is an ideal backbone interconnect technology.

For our overall development we chose the SystemVerilog language. Some key features we liked included randomization and the object-oriented capabilities. The module-based view of the Universal Reuse Methodology (URM) within Plan-to-Closure uses these features in a way that is appealing to design team and users new to SystemVerilog. This capability provides reusability at multiple levels including constraint-driven random stimulus generation, independent environment checking and overall reduction in the code density.

These features of SystemVerilog and URM together are what helped us develop our automated test environment for our Gigabit Ethernet VIP. But, before we get too far ahead of ourselves lets introduce the various aspects in our VIP development – then take a deeper look.

1) **VIP Design and Development**
   This included development of different modules of Verification IP (i.e. transactions, sequence driver, bfm, monitor, functional coverage etc.)

2) **Testbench Development**
   This stage included development of PHY module (Collision detection and collision sensing module) and stitching the VIP with reference module.

3) **Random Test Generation**
   This stage included development of different test scenarios for GbE testing. Generated scenarios are properly constrained so that they won't generate any invalid scenarios. Users can configure to generate either random packets or directed packets.

4) **Assertions Development**
   Development of assertions for GMII (Gigabit Media Independent Interface) Using SystemVerilog Assertions (SVA) were used to check this interface.

5) **Scripts Development**
   To test the verification environment, scripts were developed in the Perl scripting language. Standalone and regression scripts were also developed.

Universal Re-use Methodology (URM)
URM within the Plan-to-Closure methodology provided our team the framework to easily develop reusable, high quality universal verification components (UVCs) for a mixed-language environment. UVCs can be developed using SystemVerilog, SystemC (IEEE 1666), *e* (IEEE 1647), or a mixed code in any of these languages. URM draws on hundreds of successful projects and provides a blueprint methodology so that the testbench code is effectively organized for constrained-random testing. Our UVCs are developed in SystemVerilog.

Building Blocks of the URM-based Verification Environment

In our case the URM methodology help us build the basic building blocks of our verification environment. These building blocks of the URM based verification environment are shown in figure 1. The architecture we followed is point to point architecture having a single agent. A brief description about these building blocks of our VIP Environment is also given below.

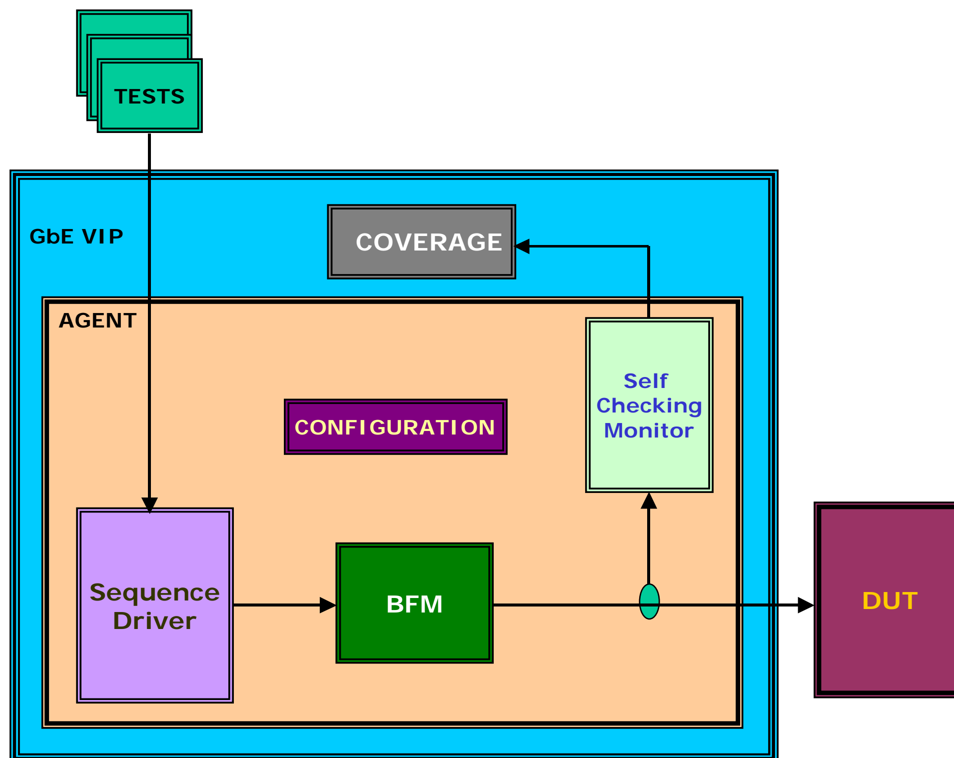

Fig. 1: GbE Verification Environment (uRM)

Configurations of Our GbE VIP

In our case we had two classes that had been used in GbE verification environment. The configuration points are distributed in transaction class and configuration class. The configuration points that are related to normal frame generation are present in transaction class. On the other hand, configuration points for error injection in different frame fields (for error-frame generation) and other VIP configuration points are present in configuration class.

Randomization of these classes are dynamic in nature, some of the configuration points of these classes were randomized during simulation to generate different frames and different scenarios.

Transaction Class

This class defines different packet fields of GbE Frame, which are used by the sequence driver for randomization. It also defines a certain set of configuration points for normal frame generation.

In URM, transactions are defined inside a **class**. This class, generally known as base class, contains basic constraints, properties (structures and variables) and methods (basic functions/tasks) e.g. the CRC calculation function, post-randomize function, display

function, padding function etc. This base class can be **extended** anywhere to add new constraints, properties and methods. Users have the ability to extend this class and constrain the properties to generate different scenarios.

The class is declared inside **package** and can be imported across the modules to access the contents of base class. Some pseudo code is provided below illustrating transaction class usage:

```
Package tel_trans_P;
 typedef struct packed {
     bit [56:0] preamble;
     bit [47:0] sfd;
     bit [47:0] sa;
     bit [47:0] da;
   } tel_struct_packet_S;

class tel_data_C;
 rand  tel_struct_packet_S packet_obj;   //object of struct defined as rand
 constraint  basic_const {
      packet_obj.preamble == {28{2'b10}};
      packet_obj.sfd == 8'b10101011;
     }            //Adding basic constraints

          //add properties to the class
          //add various methods to the class (functions and tasks)
  endclass
endpackage
```

Configuration class

Configurations are defined inside a class in a package in uRM. Our block contains various configuration points for controlling the verification IP and this package is used in most of the modules. Below are we dive deeper into a few of the configuration points such as the number of packets, pause timer value, burst timer value, operation mode, frame type, and error type.

Sequence Driver

Sequences are basically a series of transactions for commonly used scenarios. Sequences are implemented in SystemVerilog using tasks. In module-based URM, the sequence driver (Generator) is defined as a **module** and also uses interface implementation. The sequence driver generates constrained random stimuli**,** defines a set of test scenarios, controls the stimuli, and provides functionality to inject errors in randomized transactions. The randomly generated packet is passed to a BFM module and uses a BFM interface as a channel to pass the random transaction to the BFM (driver).

The generator will **"put"** the random packet to BFM Interface. At the time of our work, mailboxes were not supported, so to pass the transaction across modules, we developed a

workaround. (Mailboxes are now supported in the Incisive simulator.) We developed an interface called BFM interface, with **"put"** and **"get"** functions which take and return the transaction type. Thus, the BFM interface gave an exact solution for the mailbox functionality. The pseudo code for sequence driver functionality is shown below:

```
module tel_sequence_driver (tel_bfm_if bfm_if);
  tel_data_C class_obj;  //object of classes
 tel_struct_packet_S struct_packet;
task main();
  int success;
  success = class_obj.randomize();  //randomization of class
  bfm_if.put(struct_packet);       //passing the transaction to bfm interface
                                   //put task will block until transaction is executed by bfm
endtask
endmodule
```

Bus Functional Module (BFM)

Our BFM (driver) is defined as a **module** in the module-based URM. It is a protocol specific module that implements a low level protocol. It works at pin-level and directly connects to the DUT Interface. It is designed to work in pull mode and it pulls the next transactions (GbE Packet) from the BFM Interface using the **"get"** functionality and will block if there is no pending transactions. Then it converts the transactions to bit-stream and drives this bit-stream (transactions) to DUT Interface pins following the GbE Protocol. The pseudo code for BFM functionality is listed below:

```
module tel_bfm  (tel_bfm_if bfm_if, tel_dut_if dut_if);
  tel_struct_packet_S struct_packet;   //object of structure
initial
 begin
 forever
 begin
  bfm_if.get(struct_packet);      //getting the packet from bfm interface
  bfm_if.done(struct_packet);  //indicates execution of transaction
 end

  end
endmodule
```

Monitor

The monitor is a passive element of the VIP Environment and defined as a **module** in module-based URM. It is responsible for monitoring bus-level (pin-level) activities. It is connected to DUT Interface to receive the transactions after DUT has performed its operation on the transaction. The monitor collects the bit stream from DUT Interface and reassembles it to form GbE packet. Protocol checkers are implemented in monitor, and they check the received packet. Bug reporting and error-checking mechanisms are the

main parts of monitor. The monitor will also pass the data to coverage module for coverage calculation and analysis.

Assertions Module

In our case a set of SystemVerilog assertions (SVA) were developed to check the GMII interface, these assertions sit closely to the GMII interface and report any interface violations. The pseudo code for assertions for GMMI is given below:

```
module tel_gmii_assert  (tel_dut_if dut_if);
typedef enum {half_duplex, full_duplex} mode_operartion_t;
mode_operartion_t mode_oper;

 property p1;
  @(posedge dut_if.clk)
   disable iff(mode_oper != half_duplex)
    ($rose(dut_if.COL) ##[1:$] $fell(dut_if.COL)) within  ($rose(dut_if.CRS) ##[1:$]
$fell(dut_if.CRS));
endproperty

assert property(p1)
  $display($time,"COL Signal is deasserted before CRS ");
else
  $error($time,"BUG : CHKR ID -> CHGBE_COL_CRS_ERR2: ASSERTION: CRS
Signal is deasserted before COL");

endmodule
```

Functional Coverage

When applying random test generation, functional coverage is an essential part of the verification flow. In our case to calculate and analyze the functional coverage of our GbE protocol, we define a coverage **module**. This block used a coverage interface to receive transactions from the BFM (driver). Our coverage interface has the mailbox functionality implemented using the “**put**” and “**get**” functions. It also has **covergroups**, which are sampled at appropriate events. Various **coverpoints** (variables) were taken inside these covergroups and have **bins** defined for capturing specific coverage values.

Cross-coverage of different coverpoints can also be defined. The coverage details with individual bins can be observed in the GUI window of ICCR tool within the Incisive Design Team simulator. The tool generates functional coverage (fcov) files for each testcase after they are run. These fcov files can later be merged to check overall coverage and generate complete coverage analysis report.
The pseudo code for coverage module is given below:

```
module tel_coverage  (tel_bfm_if bfm_if, tel_dut_if dut_if);
   tel_struct_packet_S struct_packe_covt;   //object of structure
```

```
covergroup fcs_error_frame_reception_cg @(mon_cov_I.FCS_error_frame_received);
   PAYLOAD_TYPE_E: coverpoint mon_cov_I.payload_type_cov
    {
      bins ALL_ONES_PAYLOAD_E  = {ALL_ONES};
      bins ALL_ZEROS_PAYLOAD_E = {ALL_ZEROES};
      bins RANDOM_PAYLOAD_E    = {RANDOM};
    }
  LENGTH_TYPE_E: coverpoint type_len_mon
    {
      bins SMALL_LENGTH_E = {[46:64]};
      bins MID_LENGTH_E   = {[64:800]};
      bins LARGE_LENGTH_E = {[801:1500]};
      bins JUMBO_LENGTH_E = {[10200:10245]};
    }

  FCS_ERR_X1: cross PAYLOAD_TYPE_E, LENGTH_TYPE_E;
 Endgroup

//MAKING INSTANCE OF FCS-ERROR FRAME RECEPTION COVERGROUP
fcs_error_frame_reception_cg fcs_error_frame_instance = new();
endmodule
```

In our case the above components were instantiated in an **"agent"** which is defined as a module in uRM and the agent is then instantiated in **"environment"** module. Environment module and testcases module are instantiated in **"top"** module.

Our GbE VIP Testbench Architecture

The testbench architecture for our GbE VIP includes the VIP with a golden reference mode (DUT) and PHY module. The PHY module provides CRS and COL control functionality. It also has 8B/10B encoding decoding modules for data transfer between the VIP and DUT and a scoreboard that was developed for data integrity checking.
A user interface was also provided for coverage analysis and scoreboard. The coverage interface was developed to transfer transactions from the monitor to the coverage module. Also, for transferring transactions to scoreboard, a user interface was developed using "**put**" and "**get**" functionalities.

To verify the GbE VIP, we have used the following architecture: -

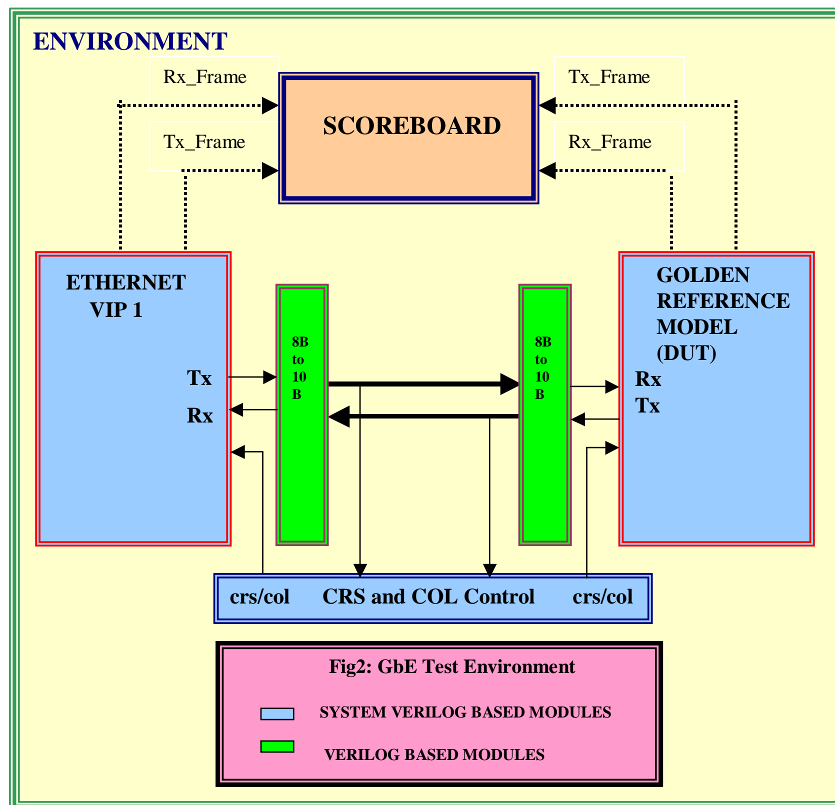

Fig.2: GbE VIP Testbench Architecture.

Benefits Realized

Benefits of using URM methodology and the Incisive Design Team simulator are as follows:

- URM implementation gave us maximum reusability at different modules e.g. BFM, agents, environments.
- Various user configuration points had been used. So, it provided a higher degree of flexibility to the user for testing the environment in different modes by simply altering the configuration points.
- GbE VIP was our first VIP developed using URM. The Cadence support team provided valuable support while we developing our VIP with the URM.
- The Incisive Design Team simulator version 05.83-v128 (check http://www.cadence.com for the latest release) was successfully tested and used for its effectiveness in URM and SystemVerilog.

Limitations

While the current version of the Incisive Design Team simulator supports the SystemVerilog features in the Incisive Design Team simulator our version of the simulator did not and we needed to work around those limitations. We are looking

forward to working with the newer version and eliminating these work arounds. A list of our workarounds is listed below.

- **Queues (Now supported in the Incisive Simulator 6.1 and later)**

Queues, one of the features of SystemVerilog was unsupported. To resolve this, we had to declare an array of maximum size resulting in memory consumption. This was the only workaround for the queues. So, we have to declare full range of variable in an array.

For example:

```
bit [0:10245][0:7] payload_class_var;
```

- **Enumeration Type ranges inside Constraint Block (Now supported in the Incisive Simulator 6.1 and later)**

Inside constraint blocks, we could not specify the ranges for enumeration types. For example, If we have enum type as enum {red, green, blue, white} color; and if we wanted to use ranges in constraint block like color inside {[red : white] then it is unsupported.

To resolve this, the only workaround was to declare full range in constraint like

```
color inside {[red, green, blue, white]};
```

- **Mailboxes (Now supported in the Incisive Simulator 6.1 and later)**

To communicate across different blocks or to pass data between classes, SystemVerilog has defined a feature called as mailbox. Mailbox is a kind of FIFO, which passes the data between the classes. Cadence provides a workaround technique for mailbox named as “BFM interface”. This interface has two methods named as “put” and “get” same as mailbox has, and the functionality is somewhat similar to put and get of mailbox. The “put” task of BFM interface is called to put the transaction on the interface and the “get” task is called to take the transaction from the interface. The arguments to be passed must be of same type as mentioned in the put and get task of BFM interface.

For example:

```
//put task of bfm interface
    task automatic put (tel_gbe_struct_packet_S obj);
      cur_trans_prvt = obj;
     -> bfm_if_ready;
     //Signal defining that a transaction is ready
     trans_ready_prvt = 1;
     @trans_done_prvt;
     //Release the semaphore. Calling task to release semaphore
     release_sem_prvt();
    endtask // put


//get task of bfm interface
```

```
task automatic get (output tel_gbe_struct_packet_S obj);
  // wait for a transaction to be ready
 while (trans_ready_prvt == 0)
  begin
   @(trans_ready_prvt);
  end
  // consume the transaction
 trans_ready_prvt = 0;
 $display($time," BFM I/F:: GET TASK");
 obj = cur_trans_prvt;
endtask // get
```

- **Randomization of 2-D array (Now supported in the Incisive Simulator 6.1 and later)**

If we have to randomize a 2-d array like bit [0:10245][0:7] payload_class_var, then declaring it as "rand" would not randomize this array. The workaround for this limitation is to use "$urandom" function to randomize the full array bit by bit. $urandom is an in-line randomization function, which returns 32-bit random value. By the help of inline constraints, we could constraint the randomization.

For example:

```
payload_class_var[1] = $urandom;  // "u" is used with random to
                                  // Specify the unsigned data.
```

- **"ignore" and "binsof" constructs were unsupported in cross coverage (Now supported in the Incisive Simulator 6.1 and later)**

"ignore" keyword is used to reject some of the bins in cross coverage, which have no significance. But ignore was not supported in our older version of the simulator. Also "binsof" construct, which is used to declare specific bins, was unsupported. The only workaround was to declare only those bins, which are significant. This will resulted in an increase in code, but it helped us to achieve true & good coverage.

For example:

```
covergroup yy;
    cross a, b
    {
     ignore_bins foo = binsof(a) intersect {5, [1:3]};
    }
endgroup
```

is unsupported.

Conclusion:

For developing highly reusable verification components, URM is a highly effective methodology. We also found the module-based approach gave us a fast independent development environment for modules, especially for logic designers unfamiliar with

advanced object-oriented programming. Our logic designers were able to build an environment because we could use many reusable components.

The environment was highly user-configurable which provided greater flexibility to the user. SystemVerilog, being an object-oriented language with features such as randomization and functional coverage proved to be a very effective language for creating our test environment. While the latest Incisive Design Team simulator version supports the object-oriented SystemVerilog features, our version 05.83-v128 had some limitations but had very good support for the SVA (System verilog assertions), so overall it was a great solution to use for SystemVerilog with the URM methodology.

References:

- IEEE Std 802.3™-2002 - 802.3-2002.pdf
- IUS User Guide (User Reference manual by Cadence).
- SystemVerilog IEEE 1800 LRM.
- www.demosondemand.com

Authors and Bios

Author: Sarvana Kumar Y N. Senior specialist at TATA ELXSI LTD., Bangalore.
Qualification: BE (Electronics and Communication), NIE College, Mysore University.
Total Experience: 9+ years, Having 6+ years in Functional Verification domain, 2 years in embedded domain, 1 year in C modeling Domain, presently working at TEL since 7 years and handling the verification activity.
Protocol knowledge: USB2.0, GbE, FiberChannel, CAN, LIN, I2C, SPI, FlexRay, OCP and AHB.
Language expertise: System Verilog, e, Verilog, VHDL,
Methodology expertise: eRM, URM, VMM, AVM
Processor knowledge: ARM, Microchip dsPIC, 8051.
Having good knowledge on the communication protocols, done functional verification of some of the protocols USB2.0, GbE, CAN, LIN, I2C, SPI used third party verification IP to verify the USB2.0, Involved in development of verification IP's, eVC's and C Models and SOC Verification based on Xtensa and ARM Processors.

Author: Jagvinder Yadav
Qualification: PG Diploma in VLSI Design from CDAC.
BE (Electronics and Instrumentation), Apeejay College of Engg, Haryana.
Total Experience: 1.5 Years in Functional Verification domain at TATA ELXSI Ltd.
Protocol Knowledge: GbE, I2C, ESCON, OPB.
Language expertise: System Verilog, Verilog, VHDL.
Methodology expertise: URM, VMM.
Presently working on development of VIPs for different protocols.

Author: Gaurav Singh
Qualification: PG Diploma in VLSI Design from CDAC.
BE (Electronics and Communication), J.E.C.R.C, Jaipur.

Total Experience: 1.8 years, 1.5 Years in Functional Verification domain at TATA ELXSI Ltd. 3 months in Quality testing at ELYMER Electronics.
Protocol Knowledge: GbE, HDLC, PHY, OPB
Language expertise: System Verilog, Verilog, VHDL.
Methodology expertise: URM, VMM.
Presently working on development of VIPs for different protocols