DesignCon 2006

Leveraging Assertions in SystemVerilog Testbench to get to Closure

Leena Singh, Cadence Design Systems, Inc.

Tim Pylant, Cadence Design Systems, Inc.

Abstract

This paper discusses some best practices in functional verification by making use of SystemVerilog for testbench, assertions for static or dynamic checking and functional coverage. It also discusses how to construct different components of verification environment (for example BFMs, monitors, stimulus generation etc.) for ease of reuse in multiple projects and platforms. It illustrates through a complete verification example - how designers can compose their block level environment with assertions, coverage and a testbench that finds more bugs and is also reusable.

Author(s) Biography

Leena Singh is a Senior Member of Consulting Staff in the Verification Division Methodology group of Cadence Design Systems, Inc and is author of following books on Verification:
System-on-a-Chip Verification - Methodology and Techniques, Kluwer Academic Publishers (KAP), 1999.
Advanced Verification Techniques: A SystemC Based Approach for Successful Tapeout, Kluwer Academic Publishers (KAP), 2004.

1. Overview

This paper focuses on techniques for performing functional verification using SystemVerilog. It covers the essential testbench development topics and guidelines for developing BFMs, monitors, protocol checkers (SystemVerilog Assertions), interfaces and overall verification environment architecture. It also discusses how designers can make use of assertions for static and dynamic checking. The scope of the paper is the verification of block level or chip level verification. It does not address the advanced verification that requires the design teams to learn the entire systemVerilog language or advanced programming language techniques.

The paper illustrates through a complete verification example - how designers can compose their block level environment with assertions, coverage and a testbench which finds more bugs and is also reusable. It shows how to construct different components of verification environment (for example BFMs, monitors, stimulus generation etc.) for ease of reuse in multiple projects and platforms. The main topics covered in the paper are

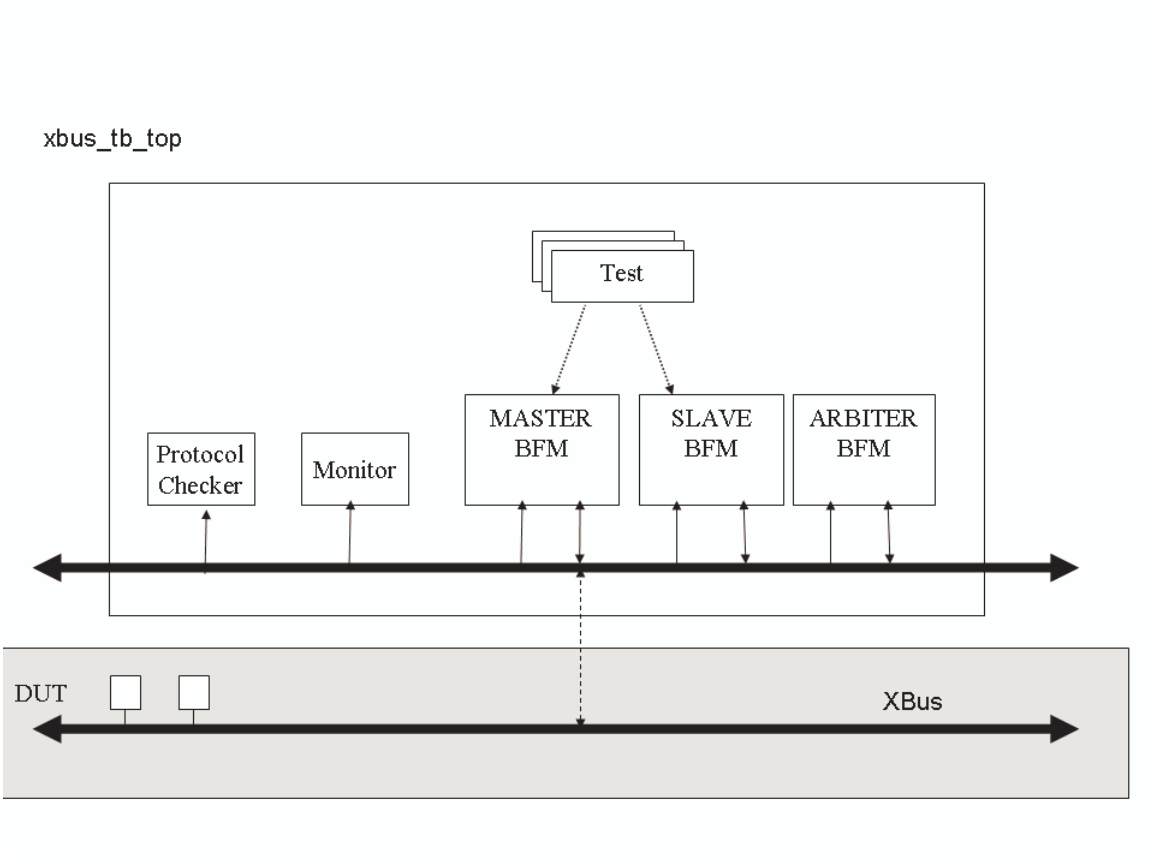
- Overview
- Using Assertions for Checking
- Writing Bus Functional Models
- Bus Monitors
- Protocol Checkers using SVA
- Top Level Connectivity
- Generating Test Stimulus

1.1. Example Used

The example used in this paper is to build a verification environment for a bus based protocol named XBus. XBus is designed to demonstrate all the important features of a typical modern bus standard while keeping the complexity to a minimum. The detailed spec for XBus protocol is out of scope for this paper. The bus protocol is deliberately simple so that attention is focused on the verification rather than the difficulties in coding the complex BFMs, and so on. The protocol has masters, slaves, and an arbiter. The XBus is a simple non-multiplexed, synchronous bus with no pipelining (to ensure simple bus functional models). The address bus is 16 bits wide and the data bus is byte-wide (to avoid alignment issues). Simple burst transfers are allowed and slaves are able to throttle data rates by inserting wait states. The bus can have any number of masters and slaves(the number of masters is only limited by the arbitration implementation). Masters and slaves are collectively known as ‘bus agents’.

The transfer of data is split into three phases: Arbitration Phase, Address Phase and Data Phase. Because no pipelining is allowed, these phases happen sequentially for each burst of data. The Arbitration and Address Phase, each take exactly one clock cycle. The Data Phase may take one or more clock cycles.

The example of overall architecture for verification environment is shown in following figure.



In the following sections we will use above example to learn how to create a complete verification testbench in systemVerilog and how to define assertions for static and dynamic checking. Following testbench components will be covered:

- Master BFM: The master BFM in above diagram is used to drive transfers over the bus to the DUT slave. It is capable of generating all types of XBus transfers. The verification environment should be configurable to emulate an unlimited number of XBus masters.
- Slave BFM: The XBus slave BFM responds to the traffic on the bus. Verification environments should be configured to have any number of slaves which can be active or passive. An active slave is used to emulate the behavior of an XBus slave device. A passive slave is used to monitor the behavior of an XBus slave device that is part of the DUT.
- Arbiter: The arbiter emulates behavior of the XBus arbiter device.
- Bus Monitor: The Bus monitor is responsible for monitoring all the bus traffic. It collects each transfer that is executed over the xbus and also collects coverage information on the bus traffic.
- Protocol Checker: This checks for adherence to the bus protocol. SystemVerilog assertions are written to check for correct behavior. An assertion is fired if any protocol violation is detected.

- Test scenarios: Tests are written to drive stimulus to the DUT. Examples of test scenarios that are concise and that can specify complex set of constraints for generation will be provided.

Relevant excerpt of the code for all the above components will be shared in the paper. Following sections will go in detail of each of the testbench components and how they can be developed for reusability.

2. Using Assertions for Checking

Assertions are basically active verification elements that

- Watch for forbidden behavior within a design block or on its interfaces
- Track expected behavior documented in the assertions.
- Are statements that specify required behavior.
- Allow the designer to capture the design intent and assumptions in a manner that can be verified.
- Provide benefits relative to bug detection, improving reuse, and capturing coverage information.
- Can be used to create transactions that aid in observing how the testbench is stimulating the design.

Assertion based verification (ABV) can be used with multiple tools such as formal analysis tools, simulators and accelerators.

- Formal tools attempt to prove that a given design behaves as required when used correctly. Assertions specify the required behavior. Assumptions specify how the design must be used. A formal analysis tool does not require any input stimulus, but does require that the behavior of the inputs be specified to avoid false failures.
- Simulators, including emulators and simulation accelerators, stimulate design inputs and observe the resulting behavior. They report errors when an asserted or assumed behavior is violated, and provide counters for cover points. ABV in a simulator is also known as dynamic ABV. ABV in an emulator or simulation accelerator is referred to as Assertion-Based Acceleration (ABA).

ABV provides a comprehensive verification approach, targeting common assertions with complementary technologies. It is recommended that designers write assertions while writing RTL. Formal analysis can then be used to start verification before the testbench is ready at the designers' desktop. The same assertions can then be used with simulation and acceleration at cluster and full-chip level.

To add the assertions, once you have identified the behavior you are interested in verifying, capture the behavior in simple English. Next, determine the type of assertion in context of the module to be either an `assert` or `assume`. When you begin adding assertions, you might benefit by starting with assertion libraries. These will enable you to get more comfortable with assertion usage. Then you can migrate to writing your own SystemVerilog Assertions (SVA) once you start writing more assertions that are at higher levels of abstraction, where languages are more efficient than simple library cells.

The assertions can be either embedded in the RTL code or put in an external file. Depending on the number and nature of the assertions, it may be desirable to write the

assertions so they can be optionally included, either via ifdef statements or by placing them in an external file.
Verification engineers should always place assertions in an external file or in the testbench to avoid file conflicts with the designer. Also you must remember, assertions associated with design modules get elaborated once for each instance of the module, whereas assertions associated with the testbench (or with the top module of the design) get elaborated only once.

2.1. Writing Assertions

For performance reasons, you may want to turn on /off any one category of assertions at different stages of the development process. For example, there may be little value in looking at functional coverage until the major bugs are worked out. Typically the initial focus is on bug catching, and later, coverage. Transaction viewing only is needed when waveform analysis is planned. For mature designs, it may be desirable to focus only on interface assertions that ensure that the device is being driven correctly. Here are some of examples of assertions embedded in the RTL for the DUT.

```
//read must have zero wait response

   assert_read_zero_wait : assert property (@(posedge xbus_clock) (xbus_start_c
      == 1'b1) && (xbus_selected == 1'b1) && (xbus_read == 1'b1) |=>( xbus_wait
      == 1'b0));
```

- The name **assert_read_zero_wait** before the colon ":" specifies the name of the assertion.
- The clock to be used for evaluation of the assertion is specified using **@ (posedge xbus_clock).**
- A property just defines a behavior pattern; it is not until a property is asserted using keword **assert** that an obligation is imposed on the design to behave as the property specifies.
- The keyword **property** specifies that the statement that follows is the check to be performed
- The output of the SVA assertion is a failure message, its associated waveform and coverage information.

Other examples of embedded assertions are as follows:

```
//The xbus_addr[15:8] must match base_addr [15:8]
   assert_xbus_selected : assert property (@(posedge xbus_clock)
(xbus_addr[15:8] == base_addr[15:8]) |-> (xbus_selected == 1'b1));

// sv assertions to check decoding of registers
   assert_xbus_reg0_select : assert property (@(posedge xbus_clock)
(xbus_addr[7:0] == 8'b00000000) |-> (xbus_reg0 == 1'b1));
   assert_xbus_reg1_select : assert property (@(posedge xbus_clock)
(xbus_addr[7:0] == 8'b00000001) |-> (xbus_reg1 == 1'b1));
   assert_xbus_reg2_select : assert property (@(posedge xbus_clock)
(xbus_addr[7:0] == 8'b00000010) |-> (xbus_reg2 == 1'b1));
   assert_xbus_reg3_select : assert property (@(posedge xbus_clock)
(xbus_addr[7:0] == 8'b00000011) |-> (xbus_reg3 == 1'b1));
```

Note that above specified assertions run with both formal analysis(static) and simulation(dynamic) tools.

The example of external assertions is in the protocol checker topic.

3. Writing Bus Functional Models

Bus Functional Model(BFM) for a device interacts with the DUT by both driving and sampling the DUT signals. One of the BFM's requirements for reusability is the ability to connect and interface to higher layers of testbench supporting multiple languages as well as support of different levels of RTL abstraction. For example support of Transaction Level Models or acceleration.

To develop a BFM first you need to create the data struct for transfer and then write the protocol implementation to drive the signals. The following examples walk you through all the significant details involved in developing a bus functional model. Before we dive into example details, let us discuss some guidelines for modeling a BFM:

- Declare a data item for transfer as a struct that has all the relevant fields for passing the data transaction to the BFM.
- Data item is preferably declared using packed struct at BFM level for synthesizability support and communication to other languages.
- BFM is written as a module that implements the protocol.
- The communication to BFM is via tasks defined in bfm interface while the actual implementation of protocol is in the BFM module.
- On one side BFM is connected to the DUT through DUT interface to connect to DUT signals and on the other side it is connected to the testbench through the task interface.

In the example, to facilitate the cross language communication and support the requirement from acceleration synthesis style, a packed struct is used to represent the transfers on the bus. This struct contains information on the required physical fields of the transaction, like address, data, size, type, and so forth but also the meta fields related to transaction, such as the delay for transmission, idle gap, and so on.

Example – Struct for the Master BFM transfer

```
typedef enum {NOP, READ, WRITE} vr_xbus_read_write_t;
typedef bit [VR_XBUS_ADDR_WIDTH - 1:0] vr_xbus_addr_t ;

//This struct represents base transaction data item
typedef struct packed{
     vr_xbus_addr_t  addr;  //address field
     vr_xbus_read_write_t read_write;  //read write field
     int size;  //size field
     bit [VR_XBUS_SIZE - 1:0][7:0] data; //data contents
     int error_pos_m;  //position of error
     int transmit_delay; //transmission delay before driving
     int idle_gap; //idle gap, no signal driven
```

```
} vr_xbus_trans_s;
```

The following example shows the tasks that implements the Master BFM protocol, making use of above struct for communication. BFM is written as a module that contains following tasks:

Example – Task implementing Master transfer

```
module vr_xbus_master_bfm(input…….);
……
//Drives all phases of transfer.
task drive_transfer (input vr_xbus_trans_s t);
        messagef(MEDIUM,"Transfer accepted: ");
        print_trans(t);
        messagef(MEDIUM,"Transfer started: ");
        repeat(bfm_trans.transfer.transmit_delay) @(posedge sig_clock);
        arbitrate_for_bus(); //arbitration phase
        messagef(MEDIUM,"Arbitration succeeded: ");
        drive_address_phase(t); //address phase
        messagef(MEDIUM,"Address Phase completed: ");
        drive_data_phase(t); //data phase
        messagef(MEDIUM,"Data Phase completed: ");
        print_trans(t);
endtask: drive_transfer
//other tasks
……
endmodule
```

It is the main task of the master to drive all phases of transfers. Note the usage of "messagef" task for consistent format of printing that takes in the verbosity of message as argument.

Following code shows the example of data phase driving task:

```
//Drives data phase of the transfer
task drive_data_phase(input vr_xbus_trans_s t);
        bit err;
        for(int i = 0; i <= t.size-1; i ++)
         begin
            if (i == (t.size-1))
                xbus.sig_bip <= 0; //Drive the xbus signal sig_bip
            else
                xbus.sig_bip <= 1;
            case (t.read_write)
                READ    :      read_byte(t, i, err);//call read_byte task

                WRITE   :      write_byte(t, i, err);//call write_byte task
            endcase
        end //for loop
        xbus.sig_data_out <= 8'bz;
        xbus.sig_bip <= 1'bz;
endtask: drive_data_phase
```

For reusability, the implementation of the BFM functionality should be kept as independent of the communication to the BFM as it can be. The SystemVerilog interface language construct and support of the tasks in the interface provides a simpler way to implement this.

The BFM itself is written as a module as shown. To communicate to the BFM, put_transaction and get_transaction tasks are defined in the BFM interface. To send a data item to the BFM, you can populate the fields of the struct and send it through the put_transcaction provided in the BFM interface. For example:

```
//Declare the xbus transaction data item - my_data
//populate the struct fields or randomize
//Send to the BFM where bfm_if is BFM interface that has task
//put_transaction
bfm_if.put_transaction(my_data);
```

BFM module also has the always block that looks for the data item to be ready and gets the transaction as soon as it sees it. This data item is transferred to the main drive_transfer task of the master BFM.

```
//This always block pulls the transaction from the higher layers when
// it sees the data item ready.
vr_xbus_seq_s bfm_trans;  //defined below
always @(bfm_if.item_ready)
begin
        bfm_if.get_transaction(bfm_trans);
        drive_transfer(bfm_trans.transfer);
        ->bfm_if.item_done;
end
```

Note that the transaction that represents the xbus transfer is obtained by the bfm_if.get_transaction task as “bfm_trans” struct. The transaction that represents the master BFM data item struct is passed onto the drive_transfer task as “bfm_trans.transfer” struct. The xbus “bfm_trans” struct(declared as vr_xbus_seq_s as shown below) has extra fields associated with it that are not required by the master but by the bus for example error position and wait states.

Below is the data item struct that represents the complete xbus transfer.

```
//This struct represents the transfer struct for Xbus
typedef struct packed{
     vr_xbus_trans_s transfer;
     bit[7:0][31:0] wait_states;
     int error_pos;
}vr_xbus_seq_s;
```

Notice that it has the transfer struct and extra fields for wait states per byte of data transfer and also an extra field for the error position.

The interface access tasks for the BFM can be implemented as shown below. These tasks are implemented in a generic way and can be used by any of the BFMs. Following tasks reside in an interface.

```
interface bfm_if();
      ……….
   task automatic put_transaction(input vr_xbus_seq_s T);
       //Get the bus lock
        bus_request();
       //Copy the transaction to local sequence
```

```
            seq = T;
           //Trigger event item_ready
            ->item_ready;
           //Wait for event item_done
            @item_done;
           //Release the bus lock
            bus_release();
    endtask: put_transaction

    task automatic get_transaction(output vr_xbus_seq_s T);
           //Copy the transaction to BFM.
           T = seq;
    endtask: get_transaction
    ……….
endinterface
```

Both of these tasks are automatic tasks so that multiple sources can call these at the same time. The put_transaction task asks for the bus request and triggers an event item_ready, notifying the BFM that the data item is ready for the transfer. The BFM gets this transaction and triggers the item_done event, notifying the initiating task that the transfer is done. At this point the control to the bus is relinquished. The bus_request and bus_release tasks are implemented as shown below:

```
    task get(input int n = 1);
           wait(n <= keys);
           keys = keys - n;
    endtask: get

    task put(input int n = 1);
           keys = keys + n;
    endtask: put

    task bus_request();
           get();
    endtask: bus_request

    task bus_release();
           put();
    endtask: bus_release
```

3.1. Slave BFM

The Slave BFM in the Xbus example is implemented the same way as recommended. The actual task implementing the protocol that responds to the transfers on the bus is called respond_to_transfers(). The slave BFM is connected to the slave ports of the DUT interface. The interface for communication to slave BFM is via the same bfm interface tasks.

3.2. Connectivity to the DUT

We discussed the connectivity of BFMs to the testbench via interface tasks. Here, let us discuss connectivity of the BFMs to the DUT.

It is recommended to use the interface to connect to DUT for connection to DUT signals. In most cases, because of legacy the DUT is written without providing any actual

systemVerilog interface for the ports in which case a wrapper can be created so that the BFMs are connected to the DUT through an interface. Here is an example of the DUT interface for Xbus:

Example – DUT Interface

```
interface xbus_if ( input logic xbus_clock, input logic xbus_reset);
   logic xbus_request;
   logic xbus_grant;
   logic [VR_XBUS_ADDR_WIDTH-1:0] xbus_addr;
   logic [VR_XBUS_SIZE:0] xbus_size;
   logic xbus_read;
   logic xbus_write;
   logic xbus_start;
   logic xbus_bip;
   wire  [VR_XBUS_DATA-1:0] xbus_data;
   logic xbus_wait;
   logic xbus_error;
   modport master(output  xbus_request,
                  ..........
                  input  xbus_error);
   modport slave (input xbus_clock,
                  .........
                  output xbus_error);
endinterface
```

Note the use of modports in the interface. Modport is required for the interface port to behave as the output for one module and the input for other. Modports configure the directions of various variables inside the interface based on which module is passed to the interface in its port list. For example sig_wait and sig_error are inputs to the master but outputs to the slave. So, in this case, there are two modports defined: one for master and one for slave. When the master and slave are instantiated, the master connection uses the "xbus_if.master' and the slave uses "xbus_if.slave".

Inside the master and slave BFMs, the access to the interface is directly through the interface name. For example, inside master or slave, the xbus_error can be seen as "dut_if.xbus_error" not "dut_if.master.xbus_error" or "dut_if.slave.xbus_error".

4. Bus Monitor

The bus monitor is responsible for extracting signal information from the DUT and translating it into the meaningful events and status information. This information is available to other components and to the test writer. The monitor should never rely on the information collected by other components such as the BFM. The bus monitor functionality has to be limited to the basic monitoring that is always required. The additional high-level functionality that might be required can be implemented separately on top of the monitor. This includes protocol checkers, scoreboards, coverage, and such.

The bus monitor in the XBus example is provided as a global monitor. The main features of Xbus monitors are:

- Generates events based on bus activity
- BFMs and monitors depend on these events for the main activities happening on the protocol.
- Collects the data for transfers

All the bus monitor events to be shared are in a package that is imported by master, slave and other modules.
As stated, the events that the XBus bus monitors emit are global to the XBus environment and are shared by different components through global declarations in the XBus package. Below are few examples of such events that are triggered by the monitor.

```
//address phase triggered one cycle after the sig_start
always @(posedge sig_start)
  @(posedge sig_clock)
  ->address_phase;

//Nop cycle
always @(address_phase)
  if((sig_read == 0) && (sig_write == 0))
  ->nop_cycle;

//Following always blocks triggers event at normal address phase
always @(posedge sig_clock)
 if((sig_read ==1) || (sig_write ==1)) ->normal_address_phase;
```

Here is the example monitor code that collects the transfer information from the bus:

```
task monitor_address_phase();
    monitor_resp.transfer.addr = sig_addr;
    monitor_resp.transfer.size = get_size();
    monitor_resp.transfer.read_write = get_read_write();
    monitor_resp.wait_states[g_i] = 0;
    monitor_resp.slave_id = 0;
endtask: monitor_address_phase

task monitor_wait_states();
    monitor_resp.wait_states[g_i] = monitor_resp.wait_states[g_i] + 1;
endtask: monitor_wait_states

always @(data_start)
g_i = 0;

task monitor_data_byte();
    monitor_resp.transfer.data[g_i] = sig_data;
    g_i++;
    if (sig_error == 1)
       monitor_resp.transfer.error_pos_m = g_i + 1;
    else monitor_resp.transfer.error_pos_m = 0;
    if ((g_i == (monitor_resp.transfer.size)) |
(monitor_resp.transfer.error_pos_m > 0))
          handle_transfer_end();
    else monitor_resp.wait_states[g_i] = 0;
endtask: monitor_data_byte

task handle_transfer_end();
    num_transfers++;
    -> transfer_end;
    print_monitor_resp(monitor_resp);
endtask: handle_transfer_end

task show_status();
    messagef(MEDIUM,"Bus monitor detected %d transfers",num_transfers);
endtask: show_status
```

Monitor can also collect the coverage information. In this example the coverage is collected based on each of the transfers. The following code is an example of coverage group defined in the master agent. It collects the coverage information at the end of each transfer:

```
covergroup transfer_end_cg @(transfer_end);
option.auto_bin_max=4;
trans_data : coverpoint data_cg;
trans_type : coverpoint rw_cg;
trans_addr : coverpoint addr_cg;
endgroup
transfer_end_cg transfer_end_inst = new();
```

The coverpoints are updated at end of transfer in monitor task as follows:

```
rw_cg = m_t.read_write; //m_t is the current transaction
        data_cg = m_t.data;
        addr_cg = m_t.addr;
```

5. Protocol Checker

Typically, the protocol checker is used to verify that the protocol specification is not violated. The checker can be implemented as a separate module, or can be part of the monitor. The checker operates based on the events and data collected by the monitor. It validates the basic data item that is transferred and the related timing requirements according to the protocol. The implemention is validated by assertions that get fired when the checks are violated.

The protocol checker provided in the XBus environment is a separate module that is instantiated at the top-level environment. It can be disabled by setting up the configuration for HAS_CHECKS at the environment level. The checks are written using SVA assertions. Here are some examples:

```
// Only one gnt line may be asserted at a time
// (Spec section 3)
always @(arbitration_phase)
begin
         if(!sig_reset)
           assertOneGrant:assert ($onehot0(gnt))
           else
           dut_error("ERR_READ_AND_WRITE\n Both read and write signals were
asserted together");
end

// Read must not be X or Z during Address Phase
// (Spec section 4)
always @(normal_address_phase)
       begin
          assertReadNotXorZ:assert (!($isunknown(mon_clk.sig_read)))
          else
            dut_error("ERR_READ\n READ went to X or Z during Address Phase");
end

// Start must not be X or Z except during reset
// (Spec section 3)
assertStartNotXorZ:assert property (
```

```
@posedge(sig_clock) disable iff (sig_reset) (sig_start == 1) || (sig_start ==
0))
else
 dut_error("ERR_START_XZ\n Start went to X or Z");
```

The dut_error is another utility task used here that allows the user to have control of the action taken on protocol violations such as stopping the simulation or continuing and updating the simulation with total number of errors or warnings that have been found.
The clocking block can be used at the protocol checker to make sure that the signals are sampled for checking with regard to the positive edge of clock when they are stable.

```
default clocking mon_clk @(posedge sig_clock);
   default input #VR_XBUS_CLOCK/10; //Input delay of 1 ns.
   input sig_size;
   input sig_read;
   input sig_write;
   input sig_bip;
   input sig_wait;
   input sig_error;
endclocking
```

6. Top Level connectivity

Following code shows the top-level testbench connectivity using a DUT interface:

```
   module xbus_tb_top();
   //Declarations
   dut_dummy_if xbus(xbus_clock,xbus_reset);  //instantiate dut interface
   dut_wrap dut( xbus); //instantiate dut
   vr_xbus_bus_monitor_wrap Monitor( xbus); //instantiate monitor
   vr_xbus_bfm_if mbfm_if(); //instantiate master bfm interface
   vr_xbus_bfm_if sbfm_if(); //instantiate slave bfm interface
   vr_xbus_slave_bfm Slave( xbus.slave, sbfm_if); //instantiare slave
   vr_xbus_protocol_checker Protocol_checker( xbus); //protocol checker
   vr_xbus_master_bfm Master( xbus.master, mbfm_if); //master BFM
   test tests(mbfm_if);  //Test
   always
      #5 xbus_clock = ~xbus_clock;
   endmodule
```

For making the above connectivity more configurable based on parameters, you can use the “generate” language construct as in the example below. Above definition of module will change for protocol checker and slave instantiations as follows:

```
   //For protocol checker instantiate based on the VR_XBUS_HAS_CHECKER
    //parameter.
   generate
   if(VR_XBUS_HAS_CHECKER)
   begin:pc
     vr_xbus_protocol_checker checker(xbus);
   end
   endgenerate

   //Instantiate the slave instances based on how many SLAVE_INSTANCES defined
   // in XBus environemnt setup.
   generate
   genvar si;
   for (si=0; si < VR_XBUS_SLAVE_INSTANCES; si++)
```

```
begin: sl
  vr_xbus_slave_bfm slave(xbus.slave, sbfm_if);
end
endgenerate
```

The hierarchical access to these components will be xbus_tb_top.pc.checker and xbus_tb_top.sl[0].slave (for first instance of slave) respectively.

6.1. Package Declarations

The package construct is provided in SystemVerilog for the declarations that can be shared among different modules, macromodules, interfaces, programs, or other packages. It also groups commonly used type declarations. Here is example of usage of package in XBus env:

```
package vr_xbus_pkg;

//All these parameters define the configuration related to Xbus environment
parameter VR_XBUS_MASTER_INSTANCES = 2;   //Number of Master instances
parameter VR_XBUS_SLAVE_INSTANCES  = 1;   //Number of Slave instances
parameter VR_XBUS_SIZE = 8;               //Size of data array of bytes
 ……..
typedef enum {NOP, READ, WRITE} vr_xbus_read_write_t;
typedef bit [VR_XBUS_ADDR_WIDTH - 1:0] vr_xbus_addr_t ;

//This struct represents base transaction data item
typedef struct packed{
     vr_xbus_addr_t  addr;
     vr_xbus_read_write_t read_write;
     int size;
     bit [VR_XBUS_SIZE - 1:0][7:0] data;
        bit check_error;
     int error_pos_m;
     int transmit_delay;
     int idle_gap;
     } vr_xbus_trans_s;


//Some utiltites for printing the structs

// Global declarations for Bus Monitor
vr_xbus_monitor_response_s monitor_resp;
  event normal_address_phase;
  event reset_end;
  event data_valid;
  event arbitration_phase;
  event wait_state;
  event transfer_end;
  int num_transfers;
……..

endpackage: vr_xbus_pkg
```

The package file provides the following three types of declarations:

- The typedef declarations are used for for commonly used structs ,which are the xbus transaction data item for master, slave and monitor. The typedef declarations are also used for the configuration setup structs for master and slave BFMs.
- The utility tasks are used for printing above structs.
- The global declarations are used for the bus monitor - the global events to be shared by other components of xbus environment.

To use the package, it is imported using the following statement:

```
import vr_xbus_pkg::*
```

While integrating different environments with different packages, name collisions must be avoided.

7. Generating Test Stimulus

Tests drive stimulus to the DUT via BFMs. A typical test creates a data item to be sent to the BFM and calls the BFM task to drive this data. Tests connect to the BFM through the interfaces and remain independent and reusable or can have direct calls to the BFM access tasks.

Following is an example of a test using interfaces. It is defined as a program block that sends the stimulus to the DUT through the BFM. The test defines a new transaction, randomizes its fields, and passes the transaction on to the BFM. The test is instantiated in top level as shown in example for top level connectivity.

```
program test(vr_xbus_bfm_if bfm_if);
xbus_trans_c trans1; //xbus_trans_c is a class data item
initial
begin
  start_test(); //Call utility task to start the test.
  trans1 = new();
  for (int i = 0; i <= 10; i ++)//Send 10 transactions
    begin
      trans1.set_size(1);//set the size for transaction to 1
      success = trans1.randomize();
      assert(success);
       bfm_if.put_transaction(trans1.data);
    end
     end_test(); //Call utility task to end the test
end
endprogram
```

The class object used in example below is defined as follows:

```
class xbus_trans_c;
   rand vr_xbus_seq_s data;
   int size_val;
   bit read_write;
   …..
//define the constraint block
constraint c1 {
seq_data.transfer.addr >= min_addrs;
seq_data.transfer.addr <= max_addrs;
if(size_val == 0)  seq_data.transfer.size inside {1,2,4,8};
else  seq_data.transfer.size == size_val;
seq_data.transfer.idle_gap == 0;
if(read_write == 0) seq_data.transfer.read_write == 2;
else if(read_write == 1) seq_data.transfer.read_write == 1;
```

```
seq_data.wait_states == 0;
}
//Constructor for the class.
function new();
min_addrs = 13;
max_addrs = 100;
read_write = 0;
size_val = 0;
endfunction
//Functions to set the constraints
task set_size(input int sz);
size_val = sz;
endtask
......
task print();
.........
endtask
endclass
```

Tests can share common functionality from a library. This library defines a common set of tasks that are required by different tests in the form of generalized scenarios. For example, each test requires some initialization sequence that can be generalized and put in that shares common tasks defined in the library is as follows:

```
program test();
 int num_of_writes;
 initial
 begin
   set_seed();
   for(int i = 0; i < 10; i ++)
   begin
     randcase
       10: seq_lib.BASIC();
       20: seq_lib.WRITE_READ_RANDOM(5);
       30: seq_lib.RAND_SIZE(10);
    endcase
  end
end
```

`endprogram`the library and made available as a task to be accessed by each test. An example of a test

8. References

1. Incisive Plan to Closure Methodology, Cadence Design Systems, Inc.
2. SystemVerilog 3.1a Language Reference Manual
3. SystemVerilog For Design : A Guide to Using SystemVerilog for Hardware Design and Modeling -- by Stuart Sutherland, et al;