The Missing Link: The Testbench to DUT Connection

David Rich
Design and Verification Technologies
Mentor Graphics
Fremont, CA
dave_rich@mentor.com

***Abstract*— In recent years, there has been a lot of attention given to Object Oriented Programming, Constrained Random and Coverage Driven Verification with SystemVerilog. The various openly available verification methodologies have put a lot of effort into explaining how to use these technologies within the testbench. Of course, RTL synthesis for design has been relatively stable for the last 20 years. The connection between the verification environment (the Testbench) and the design under test (the DUT) has received relatively little attention.**

This paper focuses on several methodologies used in practice to connect the Testbench to the DUT. The most common approach is the use of SystemVerilog's virtual interface. This is so common that people fail to investigate other methodologies that have merit in certain situations. The abstract class methodology has been presented before, but still seems to have barriers to adoption. There are also some obvious direct connection methodologies that are often overlooked. This paper will compare and contrast each one so that users may choose the methodology that meets their requirements.

Keywords-SystemVerilog; testbench; DUT

I. INTRODUCTION

One of the key notions of SystemVerilog was the merging of hardware verification language (HVL) concepts used in a testbench with hardware description language (HDL) concepts used in a design. Even though the language has merged, the experiences of the users have not – they still have different ideas about which constructs can and cannot be used in their environment.

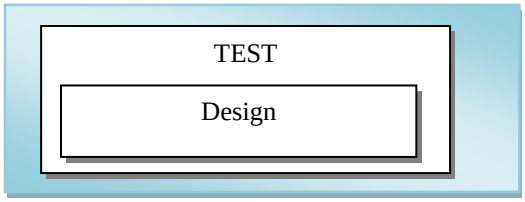
As has been true since the beginning of logic design, a design under test (DUT) is a boundary between what will be implemented in hardware and everything else needed to verify that implementation. In SystemVerilog as in Verilog, that boundary is represented by a module[1]. The job of the testbench is to provide the necessary pin wiggles to stimulate the DUT and analyze the pin wiggles coming out. Although this may seem like an over simplification, no matter how complex the environment becomes this point remains the same.

The difference that SystemVerilog introduces is that most of the testbench will written in dynamically constructed classes after the beginning of simulation. That means connections between the DUT and testbench normally need to be dynamic as well.

Let us start with a progression of testbench environments starting with an original Verilog testbench and gradually introduce additional levels of complexity along with the features in SystemVerilog that address this added complexity.

II. STATIC PIN TO PIN CONNECTIONS

In Verilog, a Design Under Test (DUT) can be modeled exactly like that – a testbench module above with the design instantiated in a module underneath. The DUT port connections are made with variables and wires directly connected to the DUT instance. Procedural code at the top level stimulates and observes the port signals.

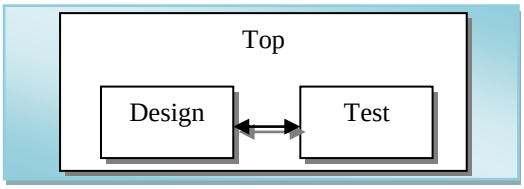



```
module testTop;
  reg clock,reset;
  wire [15:0] data;
  reg [15:0] address;

  DUT d1(.clk(clock),.rst(reset),.bus(data),
         .address(address));

  initial begin // the test
        reset = 1;
    #100 reset = 0;
       …
   end
endmodule
module DUT(input wire clk,
           input wire reset,
           input wire [15:0] address,
           inout wire [15:0] bus);
endmodule
```

The structure above rapidly breaks down as the design becomes more complex. The test usually needs to become modularized just as the DUT is, so the testbench is broken into a separate module or several modules and is instantiated alongside the DUT. Wires at the top level connect the ports of the test and DUT together.




```
module testTop;
  wire c,r;
  wire [15:0] d;
  wire [15:0] a;

  DUT d1(.clk(c),.rst(r),.bus(data),address(a));
  TEST t1(.clk(c),.rst(r),.bus(d),.address(a));
endmodule
endmodule
module DUT(input wire clk,
           input wire reset,
           input wire [15:0] address,
           inout wire [15:0] bus);
endmodule
module TEST(ouput reg clk,
            output reg reset,
            output reg [15:0] address,
            inout wire [15:0] bus);
  initial begin // the test
         reset = 1;
    #100 reset = 0;
        …
  end
endmodule
```

Observing the redundancy of repeatedly specifying the names of signals involved in connections, SystemVerilog added the concept of an **interface** to represent a collection of signals. Those signals are defined once in an interface and used in the port connections to the DUT and testbench.

```
interface dut_itf;
  logic clock,reset;
  wire  [15:0] data;
  logic [15:0] address;
endinterface
module testTop;
  dut_itf i1();
  DUT d1(.itf(i1));
  TEST t1(.itf(i1));
endmodule
module DUT(dut_itf itf);
endmodule
module TEST(dut_itf itf);
  initial begin // the test
         itf.reset = 1;
    #100 itf.reset = 0;
        …
  end
endmodule
```

This can significantly reduce the total number of code lines, especially when there are a large number of signals that can be put into the interface. Sometimes modules are brought in from legacy designs, or from environments that do not support SystemVerilog interfaces. In that case you can simply replace the port list of the DUT instantiation line with hierarchical references to the interface signals.

```
DUT d1(itf.clock,itf.reset,itf.data,itf.address);
```

The connections shown up to this point have all been structurally static. The testbench and DUT modules as well as the connection to those modules are declared at compile time. Any change to the structure requires recompilation and elaboration of that structure.

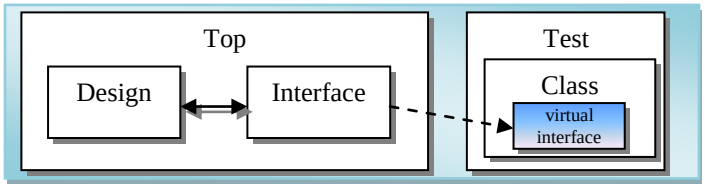
III. Dynamic Connections

In a class-based testbench environment, classes are used instead of modules to represent different components of a testbench, like drivers and monitors. Because SystemVerilog classes are always constructed dynamically, we can take advantage of that to randomize the testbench, as well as override the behavior of those classes by extending them.

Because classes do not have ports that can be connected to other module ports, some other mechanisms must be used to communicate with the DUT. We could simply use hierarchical references to signals in a module, but as shown previously in [2], this leads to un-reusable and unmanageable code. Using the recommended practice of putting class declarations in packages enforces this restriction because hierarchical references are not allowed from inside packages.

A. Virtual Interfaces

A **virtual interface** variable is the simplest mechanism to dynamically refer to an interface instance. This type of variable can be procedurally assigned to reference an interface of the same type.




```
package my_pkg;
class driver;
  virtual dut_itf vitf;
  task run;
         forever @(posedge vitf.clock)
           begin … end
  endtask
endclass
endpackage
// other modules & interface same as previous
module TEST(dut_itf itf);
import my_pkg::*;
driver d;
  initial begin
       d = (new);
       d.vitf = itf;
       d.run;
  end
endmodule
```

The driver class is free of hierarchical references and its run method can synchronize to the clock inside the interface. In this way a virtual interface variable is similar to a class handle variable where the interface is used as a type and you are referencing members of the class. Because the interface is

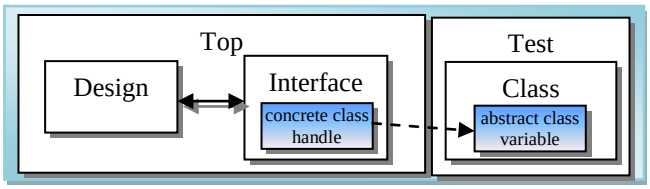
treated like a type, any parameterization of the interface instance needs to be repeated in the virtual interface declaration.

```
interface itf #(int width, size);
  wire [width-1:0] bus;
  logic [7:0] mem[size];
endinterface
module top;
  itf #(.width(8),.size(16)) i1();
  itf #(.width(16),.size(32)) i2();
endmodule
class monitor;
  virtual itf #(#(.width(8),.size(16)) vitf1;
  virtual itf #(#(.width(16),.size(32)) vitf2;
endclass
```

In the example above, vitf1 can only be assigned to top.i1 and vitf2 only to top.i2. As interfaces get more complicated in larger designs, keeping all the virtual interface parameters in sync with the interface instance parameters becomes a challenge.

B. Abstract Classes

The abstract class construct of SystemVerilog is an object-oriented programming concept used to define software interfaces. It has functionality similar to that of a virtual interface, with the benefit of a class based approach that may include inheritance and polymorphism. Another benefit is that an abstract class can completely decouple a testbench class component from any dependencies on the SystemVerilog interface, such as parameters overrides. A downside is that all members of the interface need to be accessed via methods, never bi direct reference. However, this is the normal programming style for object-oriented software.




```
package abstract_pkg;
 virtual class abstract_intf #(int awidth);
  pure virtual function void set_address(
          input logic [awidth-1:0] a);
  pure virtual task posedge_clock;
endclass
endpackage
interface dut_itf#(int dwidth, awidth);
  logic clock,reset;
  wire  [dwidth-1:0] data;
  logic [awidth-1:0] address;
 import abstract_pkg::*;
 class concrete_intf#(int width) extends
                        abstract_intf#(width);
  function void set_address(
               input logic [width-1:0] a);
   address = a;
  endfunction
  task posedge_clock;
   @(posedge clock);
  endtask
 endclass
 concrete_intf#(awidth) c = new();
endinterface
```

Now our testbench classes can be written to use the concrete class handle referenced via an abstract class variable instead of the virtual interface variable.

```
package my_pkg;
  import abstract_pkg::*;
class driver;
  abstract_intf#(16) c_h;
  task run;
   forever begin
            c_h.posedge_clock;
            c_h.set_address('h1234);
           end
   endtask
 endclass
endpackage
module testTop;
  dut_itf #(8,16) i1();
  DUT d1(.itf(i1));
  TEST t1(.itf(i1));
endmodule
module TEST();
  import my_pkg::*;
  driver d;
  initial begin
       d = (new);
       d.c_h = itf.c;
       d.run;
  end
endmodule
```

IV. WHITEBOX VERIFICATION

It is not always possible to treat the DUT as a black box; that is to monitor and drive signals for only the top-level ports of the DUT. This is true especially as one moves from block-level testing to larger system level testing. Sometimes we need implementation knowledge to access signals internal to the DUT. This is known as *whitebox* verification.

A. *Hierarchical references*

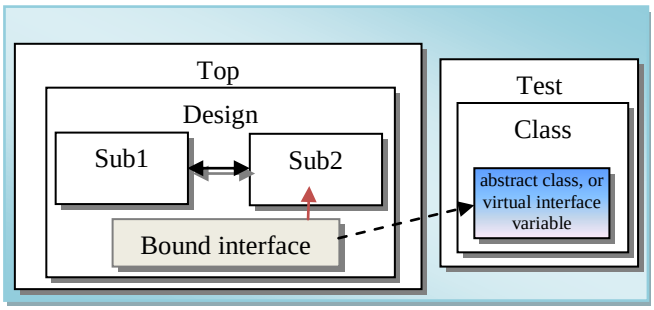
Verilog has always provided the ability to reach inside almost any hierarchical scope from another scope. Although this is a very convenient feature, it has several drawbacks:

1. It makes the code less reusable because the references in the testbench are dependent on the structure of the DUT.
2. It requires full or partial recompilation of the DUT to provide access to internal signals
3. It creates a poorly optimized DUT because internal signals may need to be preserved to provide access.

It may be impossible to avoid all hierarchical references. As a general rule, it is best to keep them at the top level of the testbench, or isolated to as few modules as practical.

B. *Bind*

SystemVerilog provides a *bind* construct that allows you to instantiate one module or interface into another target module or interface without modifying the source code of the target. The ports of the instance are usually connected to the internal signals of the target. If you bind an interface, you can use either the virtual interface or abstract class mechanisms to reference the interface.



An interface used in a bind construct typically has ports used to connect to the internal signals of the target module.

```
package probe_pkg;
 virtual class abstract_probe;
  pure virtual function get_signal;
 endclass
endpackage
interface probe(inout signal);
 import probe_pkg::*;
 class concrete_probe extends
        abstract_probe;
  function get_signal();
    return signal;
  endfunction
 endclass
 concrete_intf c = new();
endinterface
```

The top-level TEST can declare the bind statement, or some other designated module suited for that purpose can declare all the bind statements for a particular testbench.

```
module DUT (…);
 wire InternalSignal;
endmodule
module TOP;
  DUT d1();
endmodule
package another_pkg;
  import probe_pkg::*;
class monitor;
  bit s;
  abstract_intf c_h;
  task run;
         forever begin
           …
          s = c_h.get_signal;
          …
       end
  endtask
endclass
endpackage

module TEST;
bind DUT : TOP.d1 probe p1(InternalSignal);
import another_pkg::*;
monitor m;
  initial begin
       m = (new);
       m.c_h = TOP.d1.p1.c;
       m.run;
  end
endmodule
```

A complete UVM based example to probe internal signals using bind is shown in Appendix A. This example also shows the recommend way to reach into the design hierarchy using a configuration database [5].

V. Special Design Consideration

Some aspects of the DUT to testbench connection require more detailed knowledge of basic Verilog modeling issues, especially when dealing with signal strengths and race conditions.

A. *Bidirectional or Tri-State Busses*

Any signal with multiple drivers (continuous assignments, in this context) needs to be modeled using a net. A net is the only construct that resolves the effect of different states and strengths simultaneously driving the same signal. The behavior of a net is defined by a built-in resolution function using the values and strengths of all the drivers on a net. Every time there is a change on one of the drivers, the function is called to produce a resolved value. The function is created at elaboration (before simulation starts) and is based on the kind of net type, wand, wor, tri1, etc.

Procedural assignments to variables use the simple rule: last write wins. You are not allowed to make procedural assignments to nets because there is no way to represent how the value you are assigning should be resolved with the other drivers. There is also no way to represent how long the procedural assignment should be in effect before another continuous assignment takes over.

Class based testbenches cannot have continuous assignments because classes are dynamically created objects

and are not allowed to have structural constructs like continuous assignments. Although a class can read the resolved value of nets, it can only make procedural assignments to variables. Therefore, the testbench needs to create a variable that is continuously assigned to a wire.

In this example, procedural assignments are made to **bus_reg** for the class-based testbench**,** while **bus** has the value of the resolved signals.

```
interface my_if;
wire [31:0] bus;
//assign to z when not driving
logic [31:0] bus_reg='z;
assign bus = bus_reg;
modport DUT(inout bus);
modport TB(input bus, output bus_reg);
class concrete_intf extends abstract_intf;
  function logic [31:0] get_bus;
    return bus; //resolved value
  endfunction
  function void set_bus(input [31:0] value);
    bus_reg <= value; // driving value
  endfunction
endclass
endinterface
```

B. Race Conditions and Clocking blocks

If not modeled correctly, a testbench is susceptible to the same race conditions as the DUT. Any signal that is written by one process and read in another process when the two processes are synchronized by the same clock or event must be assigned using a non-blocking assignment (NBA).

A **clocking** block can address these race conditions even further by sampling or driving signals some number of time units away from the clock edge. It also takes care of the procedural assignment to a net problem by implicitly creating a continuous assignment from the clocking block variable to the net.

```
interface my_if;
  wire [31:0] bus;
  logic [31:0] address;
  bit clk;
  clocking cb @(posedge clk);
    output #1 address;
    inout bus;
  endclocking
  class concrete_intf extends abstract_intf;
    task posedge_clock;
      @cb;
    endtask
    function logic [31:0] get_bus;
      return cb.bus; //resolved value
    endfunction
    function void set_bus(input [31:0] value);
      cb.bus <= value; // driving value
    endfunction
  endclass
  modport DUT(inout bus, input clk, address);
  modport TB(clocking cb);
endinterface
```

One caution about using clocking blocks: use only the @cb event to synchronize the process that is using the clocking block variables. Using @(posedge clk) or any other event will introduce race conditions.

VI. SUMMARY

A number of different mechanisms have been shown to connect the DUT to the testbench. They are not meant to be exclusive. The complexity of your verification environment will dictate the most efficient mechanism for you to use. Above all, it is important to be as consistent as possible with your coding decisions and document those decisions.

REFERENCES

[1] IEEE (2009) "Standard for SystemVerilog- Unified Hardware Design, Specification, and Verification Language", *IEEE Std 1800-2009*.

[2] Bromley, J. & Rich, D (Feb 2008) "Abstract BFMs Outshine Virtual Interfaces for Advanced SystemVerilog Testbenches", *Design & Verification Conference*, San Jose, CA.

[3] Baird, M (Feb 2010) "Coverage Driven Verification of an Unmodified DUT within an OVM Testbench", *Design & Verification Conference*, San Jose, CA.

[4] Bromley, J. (Feb 2012) "First Reports from the UVM Trenches: User-friendly, Versatile and Malleable, or Just the Emperor's New Methodology?", *Design & Verification Conference*, San Jose, CA.

[5] Horn, M, Peryer, M. et. al. (retrieved on February 7, 2012) "Connect/Dut Interface", *Verification Academy UVM/OVM Cookbook*, <http://verificationacademy.com/uvm-ovm/Connect/Dut_Interface>

Appendix A. – EXAMPLE OF USING VIRTUAL INTERFACE AND ABSTRACT CLASS TOGETHER

```
// $Id: probe.sv,v 1.4 2010/04/01 14:34:38 drich Exp $
//------------------------------------------------------------------------
//   Dave Rich dave_rich@mentor.com
//   Copyright 2007-2012 Mentor Graphics Corporation
//   All Rights Reserved Worldwide
//
//   Licensed under the Apache License, Version 2.0 (the
//   "License"); you may not use this file except in
//   compliance with the License.  You may obtain a copy of
//   the License at
//
//       http://www.apache.org/licenses/LICENSE-2.0
//
//   Unless required by applicable law or agreed to in
//   writing, software distributed under the License is
//   distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
//   CONDITIONS OF ANY KIND, either express or implied.  See
//   the License for the specific language governing
//   permissions and limitations under the License.
//------------------------------------------------------------------------

// ----------------------------------------------------------------------------------
// file RTL.sv

// Package of parameters to be shared by DUT and Testbench
package common_pkg;
   parameter WordSize1 = 32;
   parameter WordSize2 = 16;
endpackage

// simple DUT containing two sub models.
module DUT (input    wire        CLK,CS,WE);

   import common_pkg::*;

   wire CS_L = !CS;

   model   #(WordSize1) sub1 (.CLK, .CS, .WE);
   model   #(WordSize2) sub2 (.CLK, .CS(CS_L), .WE);
endmodule

// simple lower level modules internal to the DUT
module model (input wire CLK, CS, WE);

parameter WordSize = 1;
reg [WordSize-1:0] Mem;
wire  [WordSize-1:0] InternalBus;

   always @(posedge CLK)
     if (CS && WE)
       begin
          Mem = InternalBus;
          $display("%m Wrote %h at %t",InternalBus,$time);
       end

   assign InternalBus = (CS && !WE) ? Mem : 'z;

endmodule

// ----------------------------------------------------------------------------------
// file probe_pkg.sv
//

// abstract class interface
package probe_pkg;
   import uvm_pkg::*;
   virtual class  probe_abstract #(type T=int) extends uvm_object;
      function new(string name="");
         super.new(name);
```

```
      endfunction
      // the API for the internal probe
      pure virtual function T get_probe();
      pure virtual function void set_probe(T Data );
      pure virtual task edge_probe(bit Edge=1);

   endclass : probe_abstract

endpackage : probe_pkg

// This interface will be bound inside the DUT and provides the concrete class defintion.
interface probe_itf #(int WIDTH) (inout wire [WIDTH-1:0] WData);
   import uvm_pkg::*;

   typedef logic [WIDTH-1:0] T;

   T Data_reg = 'z;

   assign WData = Data_reg;

   import probe_pkg::*;
   // String used for factory by_name registration
   localparam string PATH = $psprintf("%m");

   // concrete class
   class probe  extends probe_abstract #(T);
      function new(string name="");
        super.new(name);
      endfunction // new
      typedef uvm_object_registry #(probe,{"probe_",PATH}) type_id;

      static function type_id get_type();
        return type_id::get();
      endfunction

      // provide the implementations for the  pure methods

      function T get_probe();
        return WData;
      endfunction

      function void set_probe(T Data );
        Data_reg = Data;
      endfunction

      task edge_probe(bit Edge=1);
        @(WData iff (WData === Edge));
        endtask
   endclass : probe

endinterface : probe_itf

// ------------------------------------------------------------------------------
// file test_pkg.sv
//
// This package defines the UVM test environment
`include "uvm_macros.svh"
package test_pkg;
   import uvm_pkg::*;
   import common_pkg::*;
   import probe_pkg::*;

   //My top level UVM test class

   class my_driver extends uvm_component;
      function new(string name="",uvm_component parent=null);
        super.new(name,parent);
      endfunction
      typedef uvm_component_registry #(my_driver,"my_driver") type_id;

      // Virtual interface for accessing top-level DUT signals
      typedef virtual DUT_itf vi_itf_t;
      vi_itf_t                vi_itf_h;
```

```
      // abstract class variables that will hold handles to concrete classes built by the factory
      // These handle names shouldn't be tied to actual bind instance location - just doing it to help
      // follow the example. You could use config strings to set the factory names.
      probe_abstract #(logic [WordSize1-1:0]) sub1_InternalBus_h;
      probe_abstract #(logic [WordSize2-1:0]) sub2_InternalBus_h;
      probe_abstract #(logic) sub1_ChipSelect_h;

      function void build_phase(uvm_phase phase);
        if (!uvm_config_db#(vi_itf_t)::get(this,"","DUT_itf",vi_itf_h))
          uvm_report_fatal("NOVITF","No DUT_itf instance set",,`__FILE__,`__LINE__);

        $cast(sub1_InternalBus_h,
factory.create_object_by_name("probe_testbench.dut.sub1.m1_1",,"sub1_InternalBus_h"));
        $cast(sub2_InternalBus_h,
factory.create_object_by_name("probe_testbench.dut.sub2.m1_2",,"sub2_InternalBus_h"));
        $cast(sub1_ChipSelect_h,
factory.create_object_by_name("probe_testbench.dut.sub1.m1_3",,"sub1_ChipSelect_h"));
      endfunction : build_phase

      // simple driver routine just for testing  probe class
      task run_phase(uvm_phase phase);
        phase.raise_objection( this );
        vi_itf_h.WriteEnable <= 1;
        vi_itf_h.ChipSelect <= 0;
        fork
          process1: forever begin
            @(posedge vi_itf_h.Clock);
            `uvm_info("GET1",$psprintf("%h",sub1_InternalBus_h.get_probe()));
            `uvm_info("GET2",$psprintf("%h",sub2_InternalBus_h.get_probe()));
          end
          process2: begin
            sub1_ChipSelect_h.edge_probe();
            `uvm_info("EDGE3","CS had a posedge");
            sub1_ChipSelect_h.edge_probe(0);
            `uvm_info("EDGE3","CS had a negedge");
          end
          process3: begin
            @(posedge vi_itf_h.Clock);
            vi_itf_h.ChipSelect <= 0;
            sub2_InternalBus_h.set_probe('1);
            @(posedge vi_itf_h.Clock);
            vi_itf_h.ChipSelect <= 1;
            sub1_InternalBus_h.set_probe('1);
            @(posedge vi_itf_h.Clock);
            vi_itf_h.ChipSelect <= 0;
            sub2_InternalBus_h.set_probe('0);
            @(posedge vi_itf_h.Clock);
            @(posedge vi_itf_h.Clock);
          end
        join_any

        phase.drop_objection( this );
      endtask : run_phase
   endclass : my_driver

   class my_test extends uvm_test;
      function new(string name="",uvm_component parent=null);
        super.new(name,parent);
      endfunction
      typedef uvm_component_registry #(my_test,"my_test") type_id;

      my_driver my_drv_h;

      function void build_phase(uvm_phase phase);
        my_drv_h = my_driver::type_id::create("my_drv_h",this);
      endfunction : build_phase
   endclass : my_test

endpackage : test_pkg
```

```
// ------------------------------------------------------------------------------
// file testbench.sv
//
interface DUT_itf(input bit Clock);
   logic             ChipSelect;
   logic             WriteEnable;
endinterface : DUT_itf

module testbench;
   import common_pkg::*;
   import uvm_pkg::*;
   import test_pkg::*;

   bit SystemCLK=1;
   always #5 SystemCLK++;

   // The DUT interface;
   DUT_itf itf(.Clock(SystemCLK));
   typedef virtual DUT_itf vi_itf_t;

   // The DUT
   DUT dut(.CLK(itf.Clock), .CS(itf.ChipSelect), .WE(itf.WriteEnable));

   // instantiate interfaces internal to DUT
   bind model : dut.sub1 probe_itf #(.WIDTH(common_pkg::WordSize1)) m1_1(InternalBus);
   bind model : dut.sub2 probe_itf #(.WIDTH(common_pkg::WordSize2)) m1_2(InternalBus);
   bind model : dut.sub1 probe_itf #(.WIDTH(1)) m1_3(CS);

   initial begin
      uvm_config_db#(vi_itf_t)::set(null,"","DUT_itf",itf);
      run_test("my_test");
   end

endmodule : testbench
```