SYNAPSE design

A proven methodology to hierarchically reuse interface connections from the block to the chip level

UVM Harness Whitepaper

The missing link in interface connectivity.

By David Larson, Director of Corporate Verification
verification@synapse-da.com

Revision 1.1
August 4, 2011

Table of Contents

Introduction

In every UVM test bench, System Verilog interfaces must be used to connect signals from the RTL to the test environment. For a single block-level test bench, the interface connections are typically made by connecting each signal in an interface to the ports of the DUT, like this:

```
module top;
   wire clk, reset_n;

   my_signals ifc(clk, reset_n); // instantiate the interface

   switch DUT (
      .clk(clk),
      .reset(reset_n),
      .data(ifc.data), // connect each signal of the interface to ports on the DUT
      .status(ifc.status),
      .port0(ifc.port[0]),
      .port1(ifc.port[1]),
      ...
   );

   initial begin
      uvm_config_db#(virtual my_signals)::set(null, "tb.switch_env0.*", "vif", ifc);
   end
endmodule
```

This approach is fine for simple, single-use block level test benches... but it falls apart for multi-block test benches for several reasons:

1. The connections are not reusable.
   You must reconnect each signal on every block to your interfaces in a system test bench. This task becomes more ominous when there are many modules that need to be reconnected and each of those modules has several interfaces and each of those interfaces has many signals. Connecting the interfaces can be a major task.
2. The ports may not be available.
   In synthesizable netlists, interfaces often aren't used to connect the modules together because of tool (or corporate) limitations. Instead, wires are typically used to stitch the modules together. You could try to assign your interface signals to the wires, but it is very difficult to get the directions right and very easy to make a mistake (believe me, I've tried). You could easily get lost floating on an ocean of tedious signal connections.

A better approach is to use a reusable harness. *With a harness, the connections made in a block-level test bench can be reused in multi-block test benches. Also, a collection of harnesses can be grouped together into virtual (or system) harnesses that connect system-level environments to yet larger system environments.*

What is a Harness?

Figure 1 - A car stereo harness. A connector on one end snaps into your specific stereo; the other connectors snap into the wiring for your specific car.

Like a car stereo harness, a UVM harness is a collection of wires (grouped together in interfaces) with at least two "ends" or "connectors" to it. One end connects to the module(s) and the other end connects to the UVM environment.

The connectors in the harness are written for your module and environment. Once the harness is created, you can then use it in all of your block-level and system-level test benches. *All you have to do is snap in the connectors.*

Fundamentally, a connector is an *interface* that is *bound to your module*. Once the interfaces are bound, you never have to connect that end of the harness again.

Interface Binding

System Verilog allows you to *bind* (or add) some of your own items to modules from a separate file – allowing you to amend the definition of the module. This mechanism is sort of like aspect-oriented programming. It is intended to be used in the test bench to add things like coverage, assertions, monitors, and... *interfaces*[1].

When you bind an interface to a module, the compiler acts as if that the interface is instantiated *inside* the module. This also means that you can access signals and ports as if the context of your code is inside the module as well. So, referring to internal signals and ports from an external file like this is perfectly valid:

```
bind math_coprocessor_module reg_ifc regIfc(.clk(clk), .reset_n(reset_n),
.address(addr), .data(data), .wr(write), .req(request), .ack(ack));
```

Where the signals `clk`, `reset_n`, `addr`, `data`, `write`, `request` and `ack` are *ports* of a module (as seen from the inside of the module) called the `math_coprocessor_module`[2]. Notice that all of the interface signals are declared as ports of the interface, which allows you to connect all of the signals in the interface instantiation and *bidirectionally* when necessary. Also notice that since we are connecting the interface to the port names of the module, the connections are always valid for every instance of this module.

Binding the signal interfaces creates one end of the UVM harness. Now we need to create the other end of the harness that connects to the environment...

[1] See the Verilog LRM IEEE 1800-2009, section 23.11 "Binding auxiliary code to scopes or instances"

[2] System Verilog also allows you to bind to a particular *instance* of a module, though it is generally best to bind to the module itself, so that the interface instantiations are automatically added to every instance of the module in the hierarchy.

The Other End

The other end of the harness is a *function* that will connect the interfaces to the UVM environment. We do this by adding a set_vifs() function to the module. The problem is that System Verilog does not allow you to directly bind functions to a module. You can, however, put that function in a dummy interface and then bind that interface to the module (we'll go over other uses of this dummy interface later):

```
interface math_coprocesser_harness(); // the dummy interface
   function void set_vifs(string path);
      uvm_config_db#(virtual reg_ifc)::set(null, {path,".reg_agent.*"}[3], "vif",
math_coprocessor_module.regIfc);[4]
      uvm_config_db#(virtual cmd_ifc)::set(null, {path,".cmd_agent.*"}, "vif",
math_coprocessor_module.cmdIfc);
      uvm_config_db#(virtual usb_ifc)::set(null, {path,".usb_agent.*"}, "vif",
math_coprocessor_module.usbIfc);
      ...
   endfunction
endinterface

bind math_coprocessor_module math_coprocessor_harness harness(); // bind the interface
to the module
```

Notice that each of the calls to the uvm_config_db pass in one of the bound interfaces, and each of the interfaces refer to the module (this is called upwards name referencing[5]). Upwards name referencing is necessary because the compiler knows nothing about the other interfaces *within the context* of the math_coprocesor_harness interface. Adding the math_coprocessor_module prefix tells the compiler where to look for these interfaces.

A Complete Harness

A complete harness file defines the dummy interface and binds all of the interfaces. The following is what a complete harness file could look like:

```
interface math_coprocessor_harness();
   function void set_vifs (string path);
      uvm_config_db#(virtual reg_ifc)::set(null, {path,".reg_agent.*"}, "vif",
math_coprocessor_module.regIfc);
      uvm_config_db#(virtual cmd_ifc)::set(null, {path,".cmd_agent.*"}, "vif",
math_coprocessor_module.cmdIfc);
      uvm_config_db#(virtual usb_ifc)::set(null, {path,".usb_agent.*"}, "vif",
math_coprocessor_module.usbIfc);
      uvm_config_db#(virtual mem_ifc)::set(null, {path,".mem_agent[0].*"}, "vif",
math_coprocessor_module.cache.mem_ifc0);
      uvm_config_db#(virtual mem_ifc)::set(null, {path,".mem_agent[1].*"}, "vif",
math_coprocessor_module.txx.cmd.memory.mem_ifc0);
      uvm_config_db#(virtual mem_ifc)::set(null, {path,".mem_agent[2].*"}, "vif",
math_coprocessor_module.rxx.cmd.memory.mem_ifc0);
```

[3] It is easier to pass in the reference to the environment as the first argument and let UVM figure out the full path:
uvm_config_db#(virtual reg_ifc)::set(env, "reg_agent.*", "vif", math_coprocessor_module.regIfc);
But because set_vifs() has to be called in the build phase, the sub-env's will not be instantiated by this point – preventing hierarchical assignments.

[4] This document follows Accellera's recommendation of assigning the interfaces using the configuration DB. An alternative approach is to use the convention used in OVM, where interfaces are assigned directly using assign_vi(). It is worth comparing the two approaches – see the OVM Harness Whitepaper.

[5] See the Verilog LRM IEEE 1800-2009, section 23.8 "Upwards name referencing"

```
   endfunction
endinterface

bind math_coprocessor_module reg_ifc reg_ifc0(.clk(clk), .reset_n(reset_n),
.address(addr), .data(data), .wr(wr), .req(request), .ack(ack));
bind math_coprocessor_module cmd_ifc cmd_ifc0(.clk(clk), .reset_n(reset_n),
.cmd(command), .valid(valid));
bind math_coprocessor_module usb_ifc usb_ifc0(.clk(clk), .reset_n(reset_n), .tx(tx),
.rx(rx) );
bind memory_module mem_ifc mem_ifc0(.clk(clk), .reset_n(reset_n), .wr(write),
.data(data), .addr(address), .req(request) );

bind math_coprocessor_module math_coprocessor_harness harness(); // adds the set_vifs
function
```

This harness binds to *two* different modules, the math_coprocessor_module and the memory_module. There are three instantiations of the memory_module in the DUT so we grab each of the interfaces that were automatically instantiated for us and assign them to our mem_agents in the environment.

Note that you could optionally place the interfaces inside the dummy interface and only bind the harness. The advantage of placing them inside the harness interface is that the interfaces can automatically scale to module parameters. See the Scalable interfaces section below for more information.

```
interface math_coprocessor_harness();
   // have to use upwards name referencing for every signal
   reg_ifc regIfc(.clk(math_coprocessor_module.clk),
                  .reset_n(math_coprocessor_module.reset_n),
                  .address(math_coprocessor_module.addr),
                  .data(math_coprocessor_module.data),
                  .wr(math_coprocessor_module.wr),
                  .req(math_coprocessor_module.request),
                  .ack(math_coprocessor_module.ack));
   cmd_ifc cmdIfc(.clk(math_coprocessor_module.clk),
                  .reset_n(math_coprocessor_module.reset_n),
                  .cmd(math_coprocessor_module.command),
                  .valid(math_coprocessor_module.valid));
   usb_ifc usbIfc(.clk(math_coprocessor_module.clk),
                  .reset_n(math_coprocessor_module.reset_n),
                  .tx(math_coprocessor_module.tx),
                  .rx(math_coprocessor_module.rx) );

   function void set_vifs (string path);
      uvm_config_db#(virtual reg_ifc)::set(null, {path,".reg_agent.*"}, "vif",
regIfc); // no need for upwards name referencing here
      uvm_config_db#(virtual cmd_ifc)::set(null, {path,".cmd_agent.*"}, "vif",
cmdIfc);
      uvm_config_db#(virtual usb_ifc)::set(null, {path,".usb_agent.*"}, "vif",
usbIfc);
      uvm_config_db#(virtual mem_ifc)::set(null, {path,".mem_agent[0].*"}, "vif",
math_coprocessor_module.cache.mem_ifc0);
      uvm_config_db#(virtual mem_ifc)::set(null, {path,".mem_agent[1].*"}, "vif",
math_coprocessor_module.txx.cmd.memory.mem_ifc0);
      uvm_config_db#(virtual mem_ifc)::set(null, {path,".mem_agent[2].*"}, "vif",
math_coprocessor_module.rxx.cmd.memory.mem_ifc0);
   endfunction
endinterface
```

```
bind memory_module mem_ifc mem_ifc0(.clk(clk), .reset_n(reset_n), .wr(write),
.data(data), .addr(address), .req(request) );

bind math_coprocessor_module math_coprocessor_harness harness(); // only one bind
```

There are two minor disadvantages to this alternate approach though:

1. Upwards name referencing has to be used for the signals connections to internal interfaces.
2. Only interfaces that are to be bound to the DUT can be placed in the dummy interface. All of the other ones (if any) have to remain outside (like the memory_module in our example).

Connecting the Harness

Okay, great... we have a harness. Now all you have to do is one sweet little step to connect the entire block to the environment. In your test case (or preferably in your intermediate-base test class or test bench class), connect the harness to the environment like this,

```
function void build_phase(uvm_phase phase);
   super.build_phase(phase);
   ...
   env = math_coprocessor_env::type_id::create("env", this);
   top_module.dut_inst.harness.set_vifs(env.get_full_name()); // this connects the
whole environment!
   ...
endfunction
```

Do NOT connect the harness in your block env! If you do, then your env will not be reusable as a sub-env in larger system simulations because the path to the harness will be invalid.

Connecting Harnesses in System Simulations

Now we are ready to look at how to connect system-level environments that use the sub-environments created earlier. This is where it gets exciting. If you have created a harness for each of the sub-env's, then connecting them all up is *trivial*. Each of the sub-env's are connected with one line:

```
function void build_phase(uvm_phase phase);
   ...
   env = math_coprocessor_env::type_id::create("env", this);
   top_module.chip_inst.math_coprocessor.harness.set_vifs({env.get_full_name(),
".math_env0"});
   top_module.chip_inst.memory_cache.harness.set_vifs({env.get_full_name(),
".cache_env0"});
   top_module.chip_inst.arbiter.harness.set_vifs({env.get_full_name(), ".arb_env0"});
   ...
endfunction
```

Or, better yet, create a virtual harness so these harness connections can be reused as well...

Virtual Harnesses

A virtual harness is a collection of harnesses or other virtual harnesses and may contain additional interface connections. It is used to connect sub-environments in a system-level environment to sub-modules. Furthermore, virtual harness can be *reused hierarchically* in still larger system simulations.

```
interface chip_virtual_harness ();
   function void set_vifs(string path);
      // connect the sub-harnesses using upwards name referencing
      chip_module.math.harness.set_vifs({path, ".math_env"});
      chip_module.regs.harness.set_vifs({path, ".regs_env"});
      chip_module.sys.harness.set_vifs({path, ".sys_env"});
      // ... others ...

      // connect agents used in this environment (if any)
      uvm_config_db#(virtual jtag_ifc)::set(null, {path,".jtag_agent.*"}, "vif",
chip_module.jtagIfc);
      // ... others ...
   endfunction
endinterface

bind chip_module jtag_ifc jtagIfc(.clk(tck), .reset(trst), .data_in(tdi),
.data_out(tdo), .test_mode_select(tms));
bind chip_module chip_virtual_harness harness();
```

Notice that we use upwards name referencing when connecting the sub-harnesses to the sub-environments. That allows the virtual harness to be reused in larger test benches since the path to the sub-harnesses are still correct with respect to the definition of the module.

Now in the test, you only have to connect the virtual harness to connect all the signals for this block and all sub-blocks:

```
top.dxx.chip.harness.set_vifs(env.get_full_name()); // BAM! Hierarchically connect
everything!
```

Advanced Interfaces

Arrays of interfaces

Some modules need to have an array of interfaces (like a switch or top-level register programming modules). These pose an interesting challenge, since System Verilog does not allow you to loop over static constructs:

```
   reg_ifc regIfc[`RegDevs-1:0] (
      .clk(RegTopModule.SysClk),
      .reset_n(RegTopModule.SysReset_n),
      .rdAddr(RegTopModule.RegRdAddr),
      .rdData(RegTopModule.RegRdData),
      .rd(RegTopModule.RegRd),
      .rdDataValid(RegTopModule.RegRdDataValid),
      .wrAddr(RegTopModule.RegWrAddr),
      .wrData(RegTopModule.RegWrData),
      .wr(RegTopModule.RegWr)
   );

   function void set_vifs(string path);
      foreach (regIfc[i]) // NO! This is illegal!
         uvm_config_db#(virtual reg_ifc)::set(null, {path,".reg_a[",i,"]"}}, "vif",
regIfc[i]);
   endfunction
```

The good news is that SV *does* allow you to loop over *virtual interfaces*. We have to take an extra step and assign the interfaces to virtual interfaces before looping over them. Fortunately, this extra step is fairly painless:

```
   reg_ifc regIfc[`RegDevs-1:0][6] (
      .clk(RegTopModule.SysClk),
      .reset_n(RegTopModule.SysReset_n),
      .rdAddr(RegTopModule.RegRdAddr),
      .rdData(RegTopModule.RegRdData),
      .rd(RegTopModule.RegRd),
      .rdDataValid(RegTopModule.RegRdDataValid),
      .wrAddr(RegTopModule.RegWrAddr),
      .wrData(RegTopModule.RegWrData),
      .wr(RegTopModule.RegWr)
   );

   virtual interface reg_ifc vifc[`RegDevs]; // extra vif for looping
   initial begin
      vifc = regIfc[0:NUM_CLIENTS-1]; // assign the entire array in one step![7]
   end

   function void set_vifs(reg_top_env env);
      foreach (vifc[i]) // loop over the virtual interface
         uvm_config_db#(virtual reg_ifc)::set(null, {path,".reg_a[",i,"]"}}, "vif",
vifc[i]);
   endfunction
```

Scalable interfaces

It is not uncommon for modules to be parameterized… but what if one or more of those parameters affect your interfaces? A parameter can affect how many interfaces to instantiate, signal bus widths or other behavior. So, how can a harness scale to parameter settings?

The trick is to *parameterize your harness* and then when you instantiate and bind your harness you pass in the module parameters to your harness. This is what the above register top harness looks like parameterized (notice that we moved the instantiation of the reg_ifc inside the reg_harness):

```
interface reg_harness #(NUM_CLIENTS = 5);
   reg_ifc regIfc[NUM_CLIENTS-1:0] (
      .clk(RegTopModule.SysClk),
      .reset_n(RegTopModule.SysReset_n),
      .rdAddr(RegTopModule.RegRdAddr),
      .rdData(RegTopModule.RegRdData),
      .rd(RegTopModule.RegRd),
      .rdDataValid(RegTopModule.RegRdDataValid),
      .wrAddr(RegTopModule.RegWrAddr),
      .wrData(RegTopModule.RegWrData),
      .wr(RegTopModule.RegWr)
   );

   virtual interface reg_ifc vifc[NUM_CLIENTS];
```

[6] See the Verilog LRM IEEE 1800-2009, section 23.3.3.5 "Unpacked array ports and arrays of instances" for the port connections rules used here. Special care is needed to make sure that signals are spread across the ports correctly, using the rules spelled out in the LRM. In your environment, you may need to change the order to LSB:MSB.

[7] For your environment, you may need to assign the interfaces in MSB:LSB order – experiment with it. Also see footnote 6.

```
   initial begin
      vifc = regIfc[0:NUM_CLIENTS-1];
   end

   function void set_vifs(string path);
      foreach (vifc[i])
         uvm_config_db#(virtual reg_ifc)::set(null, {path,".reg_a[",i,"]"}}, "vif",
vifc[i]);
   endfunction

bind RegTopModule reg_top_harness #(.NUM_CLIENTS(NUM_CLIENTS)) harness();
```

Now, when a RegTopModule is instantiated, *a harness is also automatically instantiated with the correct settings*. The harness self-adjusts to conform to each instantiation of the module.

Summary

With harnesses, the connections made in block test benches can be leveraged in larger test benches. Furthermore, harnesses can be used hierarchically, where harnesses created for system test benches can be reused in larger test benches. In every case, all of the interfaces are connected to the RTL in just one statement.

Without harnesses, stitching all of the signals to interfaces in large system simulations can be very tedious and error prone (and almost impossible in some cases). The connections made in small, block level test benches have to be thrown out and cannot be reused.

Harnesses have been tested and proven to be highly beneficial in large-scale projects, resulting in large productivity gains and virtually eliminates all test bench connectivity problems.

Thanks

Special thanks goes to Senior RTL Developer Andy Iverson at Tektronix for his own ideas that refined the harness concept further.