cadence

Working with Interfaces EZ-Start Guide

October 2006

Working with Interfaces EZ-Start Guide

The procedures described in this document are deliberately broad and generic.

About EZ-Start Guides

EZ-Start guides are provided by Cadence Design Systems as entry point tutorials for various technologies. EZ-Start guides are meant to quickly review high-level concepts of a specific technology, and allow the user to independently experience and explore it. For a deeper understanding of how to architect a quality verification environment, consult a Cadence methodology specialist or refer to the *Plan-to-Closure Methodology User Guide*.

Introduction

Managing the connectivity between blocks can be difficult, because it entails keeping all of the ports current, synchronizing instantiations with the ports, passing data through the hierarchy, and so on. The connectivity between major blocks becomes increasingly difficult to manage when you want to start verifying protocols.

One of most powerful constructs in SystemVerilog is the *interface* construct, because it can encapsulate the communication between blocks in one single area. Placing all of this information in one place also facilitates reuse.

The *Interface EZ-start Guide* focuses on helping new SystemVerilog users become familiar with the interface construct. Although interfaces can be used anywhere in your design or verification environment, this guide limits the discussion to using interfaces within a verification environment.

Types of Interfaces

Using interfaces is one the methods recommended by the Cadence Univeral Reuse Methodology (uRM) for modeling the communication between verification components. Using interfaces, blocks can communicate through tasks (task-level interfaces) or through signals (signal-level interfaces). The communication within a verification environment is done primarily through task-level interfaces, and the communication to the DUT is done primarily through signal-level interfaces.

Signal-Level Interfaces

Signal-level interfaces are simply just a collection of nets.

Tip If you need to verify a protocol across the interface, you can also add assertions.

The following is an example of a module header for an arbiter design. In this example, all of the design's connectivity is described within the module header.

```
module arbiter (ReqA, GntA, ReqB, GntB, Busy, Done, ResetN, clk);
  input  ReqA, ReqB, Done;
  input  ResetN, clk;
  output GntA, GntB, Busy;
endmodule
```

The following example takes the same design, but captures all of the signals within an interface. Then, the interface is simply included in the port list:

```
interface arb_if ();
  logic  ReqA, ReqB, Done;
  logic  ResetN, clk;
  wire   GntA, GntB, Busy;
endinterface

module arbiter (arb_if ai);
endmodule
module top ();
  arb_if ai();
  arbiter dut(.ai(ai));
endmodule
```

Using a signal-level interface has the following advantages:

Provides a simple, clean way to connect blocks without having to worry about the individual signals.

Eliminates redundant declarations of the same signals in multiple modules, which can significantly reduce the size of a description.

Grouping signals together in one place also improves design maintainability. For example, if a change to the port specification is required, the change can be made in one place instead of in multiple modules. Another advantage is that assertions can be added to the interface for protocol checking.

Task-Level Interfaces

Task-level interfaces encapsulate functionality in addition to connectivity. An interface can contain data type declarations, tasks and functions, `initial` and `always` blocks, continuous assignments, and so on. This allows the definition of communication protocols, protocol checking routines, and other verification routines in one place. This provides an abstract communication channel to items such as bus functional models (BFMs). The following is an example of a standard task-level interface (as recommended in the uRM):

```
interface bfm_if();
  arbpkt_s bfm_pkt;
  event    item_ready, item_done;

  task automatic put(input arbpkt_s test_pkt);
    bfm_pkt = test_pkt;
    ->item_ready;
  endtask

  task automatic get(output arbpkt_s test_pkt);
    test_pkt = bfm_pkt;
  endfunction
endinterface
```

In task-level interfaces, communication is done through calls to tasks in the interface. For example, the block that is communicating with the interface calls the `put` task with a packet structure. The packet is then stored in the interface, and any blocks waiting on the `item_ready` event are notified. After notification, the block can then call the `get` task to retrieve the packet from the interface. Since all of the communication is done through the interface, neither side has to know who the end consumer or producer is—they just have to know to put or get the data from a standard interface.

Instantiating and Using the Interface

An interface is instantiated in the same way that modules and primitives are instantiated. Interfaces must be instantiated at the highest point in the hierarchy where they can be used.

```
module top ();
  bfm_if bi(); // instantiate task-level interface
  arb_if ai(); // instantiate signal-level interface
  arb_bfm bfm(.*); // .* ports with same name as local signal
  arbiter arb_dut (.*);
  test test(.*);
endmodule : top
```

Interface ports on modules are declared as module ports that are a specific type of interface. Use the following syntax:

```
module module_name (interface_name port_name, other_module_ports);
```

For example:

```
module arbiter (arb_if ai);
```

Once the interface is instantiated in the top level and declared on the port list, the objects in the interface are referenced from by using an interface reference (*`port_name.interface_object_name`*). An example of a signal-level interface:

```
always @(state or ai.ReqA or ai.ReqB or ai.Busy or ai.Done )
```

If you cannot append the interface reference to all necessary signals, you can use an `assign` to make the connection:

```
assign Done    = ai.Done;
assign ai.GntA = GntA;
```

You can reference a task-level interface in a similar fashion:

```
module test(bfm_if bi);
 initial begin
   bi.put(arb_pkt);
   @(bi.item_done);
 end
endmodule

module arb_bfm(bfm_if bi, arb_if ai);
 always @(bi.item_ready) begin
   arb_pkt = bi.get();
   drive_sigs();
   ->bi.item_done;
 end
endmodule
```

Conclusion

The SystemVerilog interface construct is key to achieving reuse and simplifying code. By using interfaces to facilitate communication between components, you also raise the verification environment's level of abstraction. You can pass data from the testbench, through the BFM, and to the DUT without having to know anything about the consumer.

Although creating interfaces is an extra step at the beginning of the verification cycle, the interfaces quickly return the time invested by making test writing and connectivity simpler and more efficient.

Interfaces Lab

In this lab, you will create a verification environment that has a testbench, BFM, and DUT, and that is connected using interfaces. The testbench, BFM, and DUT are provided, but you will need to create the interfaces and the top-level netlist that connect everything together.

Lab instructions:

1. Untar and unzip the `EZStartInterfaces.tgz` file included with this document:

```
gtar –xzvf EZStartInterfaces.tar.gz
```

This contains an `/rtl` directory, which contains the arbiter design, and an `/ius` directory, which is for running the simulation.

2. Create the signal-level interface for the arbiter design. It must include the following variables:

```
logic ResetN;
logic ReqA, ReqB;
logic Done;
wire GntA, GntB;
wire Busy;
```

3. Create the task-level interface for the BFM. It must contain the following events and tasks:

```
event item_ready, item_done;
task automatic put(...);
task automatic get(...);
```

The `/ius/tb/def_pkg.sv` package file contains the `arb_pkt_s` structure definition, which you must use for the data type of the structure being passed in the BFM.

4. Modify the `/ius/tb/test.sv` file such that it uses the BFM interface, instead of the individual ports.

The functionality in the original `test.sv` file will be split between the test and the BFM. The tasks, latency randomization, and signal assignments will take place in the BFM. The randomization of task selection will take place in the test file.

Note: The solution for this step is in the `test.sv` and `arb_bfm.sv` files of the `/.solutions` directory, which was provided with this document.

5. Modify the `/tb/top.sv` file to instantiate the arbiter interface, BFM interface, and BFM module. Also, change the test and arbiter instance port lists so that they use the interfaces.

Note: So that you do not have to modify the `arbiter` design to use the interface, an `arb_wrapper.sv` file is included in the `/rtl` directory. Instantiate the `arb_wrapper`, which instantiates the `arbiter`, in the top level, instead of directly instantiating the arbiter design.

6. Simulate the design from the `ius` directory.

```
./run_sim
```

or

```
./run_sim +gui
```

At the end of the simulation, you should see the following message:

```
Running XX transfers
Simulation complete via $finish(1)
```

You should not receive any error messages.

The solution to this exercise is available in the /`.solutions` directory, which was provided with this document.