The UVM Tutorial

by Brian Hunter

Version 1.0

Table of Contents

Acknowledgments

The tutorial and the coding guidelines presented here could not have been made possible without significant contributions from the following individuals. I greatly appreciate all of their efforts and encouragement:

Janick Bergeron, Synopsys
Michael Carns, Cavium
Stephen Chinatti, Cavium
Ben Chen, Cavium
Leo Chen, Cavium
Joeseph D'Errico, Cavium
James Ellis, Cavium
Srihari Gosike, Cavium
Rebecca Lipon, Synopsys
Ethan Robbins, Cavium
Jeffrey Schroeder, Cavium
Ahmed Shahid, Cavium
Chris Spears, Synopsys
Joe Stadolnik, Cavium
Claudine Wong, Cavium

Revision History

Version	Author	Date	Notes
0.1	Brian Hunter	Oct. 5, 2011	Initial release to SPOC working group.
0.2	Brian Hunter	Oct. 27, 2011	Second release to SPOC working group.
0.3	Brian Hunter	Dec. 1, 2011	All major content complete.
0.4	Brian Hunter	Dec. 30, 2011	Release for testing.
0.5	Brian Hunter	Jan. 13, 2012	Updates after review.
0.6	Brian Hunter	Feb. 7, 2012	Added appendices for vertical reuse, reset testing, and self-testing.
1.0	Brian Hunter	Jul. 17, 2013	Major overhaul after many reviews.

Introduction

Welcome to Cavium's UVM Tutorial. In these lessons, you will learn the key concepts of UVM and how to use it with SystemVerilog for verification within Cavium's environment.

To get the source code associated with this tutorial, use the `utut` script described below.

About This Tutorial

This tutorial walks you through the process of adding to an existing testbench. It takes a dive-right-in approach. Each chapter introduces one or more concepts and presents problems that you are expected to attempt to solve. *Some of these problems are intentionally difficult*. The objective of the problems is to challenge you to find the correct answers through trial-and-error, in order to better familiarize yourself with what works, what doesn't, and where to find the answers.

If you need help solving the problems, your best courses of action are to:

- Read through the existing UVM code that comprises the testbench;
- Refer to the UVM reference manual and user's guide, the SystemVerilog Language Reference Manual, or;
- Visit http://www.uvmworld.org.

If you are unable to solve a problem, a solution is presented on the following page.

Design Decisions

The tutorial and the coding guidelines that it is based on do not always approach each problem with the most efficient answer in terms of simulation performance. Where priority of resources is concerned the methodology attempts to espouse the following order of precedence:

Debug Time < Design Time < Simulation Run Time < Simulation Compile Time

In other words, you may find examples of the methodology that may be harmful to simulation run-time performance or may cause longer compile-times, depending on your choice and version of simulator. These decisions were made intentionally because they help to minimize the time it takes to initially design a verification environment and/or simplifies the process of reading and debugging it.

Tool Scripts

utut

Use this script to checkout the source code, obtain code snippets that exist throughout this tutorial, or to synchronize your working area with the chapters.

To create a new work environment tree in the directory '`mydir`':

```
> utut create mydir
```

Many code snippets presented in this tutorial are preceded by a small number. Using the `utut` script as follows will print out the code that you can then cut-and-paste into your own files. This example prints out code snippet #17:

```
mydir/verif/alutb> utut code 17
```

Each chapter throughout this tutorial has a checkpoint file available. You can run the following command to update your working copy to be identical to the solutions.

```
mydir/verif> utut chapter 3
```

Using this command allows you to start at (or skip) any chapter you wish and still be able to continue with the rest of the tutorial. It will bring your current working directory up to the *end* of the chapter that you specify. However **this will delete any modifications that currently exist in your working directory.**

As with all scripts, help can be viewed with the `-h` switch on the command-line.

utg and untb

These scripts operate in accordance with the coding guidelines that are strictly followed in this tutorial. `utg` generates files, classes, and vkits. `untb` calls `utg` to generate a new testbench.

The templates that they generate can be customized to your company's needs as you see fit. To do so, you would need to edit the files located in the `tools/uvmtmp` directory.

Prerequisites

Before beginning the UVM Tutorial, you should already be quite familiar with SystemVerilog's verification components, including but not limited to these chapters of specification:

- Classes and Inheritance
- Constrained Random Value Generation
- Processes
- Functional Coverage Collection
- Data Types and Aggregate Data Types
- Interfaces, Modports, and Clocking Blocks
- Interprocess Synchronization and Communication

Also, you should have the **SystemVerilog 1800-2009** specification handy for your reference.

While this is a tutorial on Cavium's usage of UVM, it assumes at least a nascent knowledge of UVM. Therefore, it is helpful for you to read through the **UVM User's Guide 1.1** and familiarize yourself with the concepts it lays out.

It is also helpful to keep the **UVM 1.1 Class Reference** handy.

You should definitely familiarize yourself with our company's **SV Guidelines** on our internal wiki. This document describes coding guidelines, naming conventions, testbench organization, and generally preferred best practices. It also points you to the **NaturalDocs website**, which describes how code should be documented to work with this documentation generator.

Updated links to these documents will be kept on our *SV & UVM Resources* page.

Chapter 1: The ALUTB Testbench

The ALU testbench (`verif/alutb`) is a pure UVM testbench created for the purposes of this tutorial. The Device Under Test (DUT) is a block called `alu` (`rtl/alu`). This block was designed to be used for this tutorial. It has CSRs on a CTX interface, and has a simple interface to perform math computation requests. In this tutorial, you will be responsible for creating UVM testbench code to test this device (and find bugs!).

The top-level hierarchy of the current ALUTB testbench approximates the following diagram. In this document, solid-line objects represent instantiations, and dashed lines represent handles or references. Also, colorful objects represent the environment, while black and white is for the RTL.

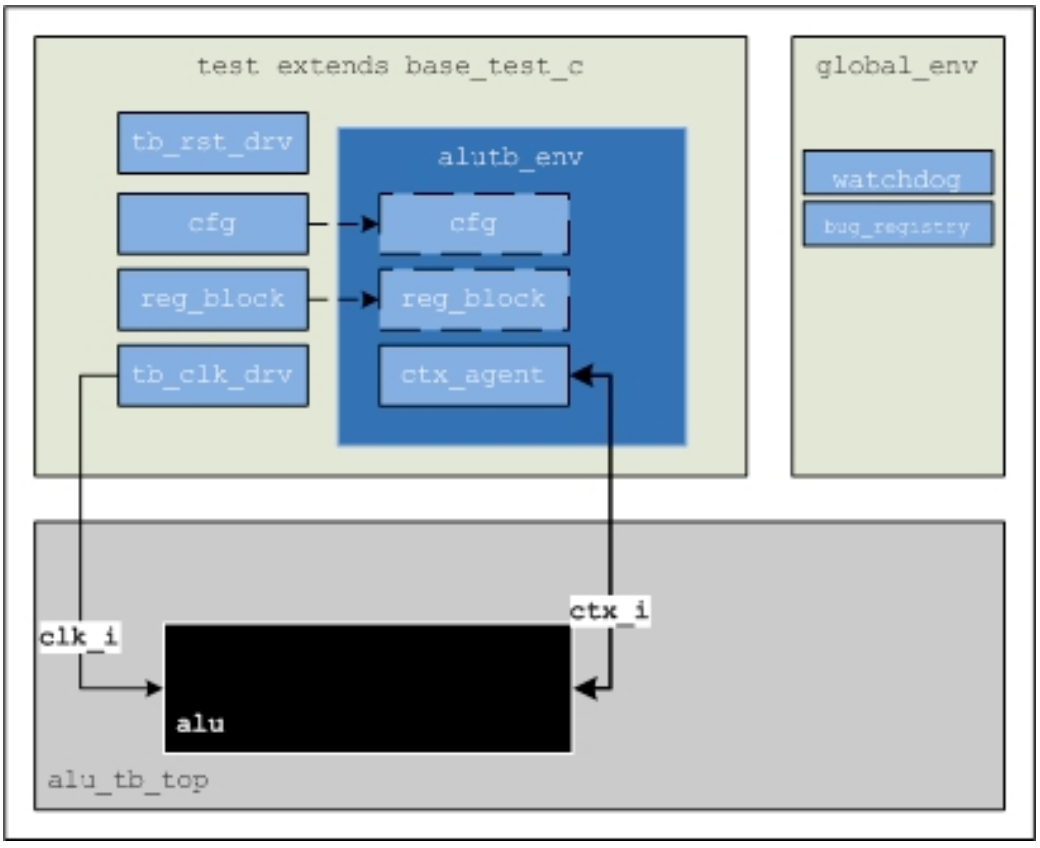


Figure 1: *The ALUTB Testbench*

We'll discuss these particulars in more detail later. But first note that the top-level of the environment hierarchy is the test itself. Also notice that each testbench instantiates a `global_pkg::env`, which is a container for functionality that is needed by every UVM testbench.

Feel free to read some of the SystemVerilog code that is already present in this testbench, and familiarize yourself with its layout and style. The code is located in these directories:

- `verif/alutb` - Contains this testbench and all of its tests.
- `verif/vkits/alutb` - Contains the `alutb` vkit (verification code that can be reused in a higher level testbench).
- `verif/vkits/ctx` - Contains the CTX vkit, which will be used to program the design's configuration and status registers.
- `verif/vkits/global` - Contains the global vkit and things that live in the global environment.
- `verif/vkits/cn` - A vkit containing miscellaneous utility functions, macros, message customizations, and components.
- `verif/vkits/uvm/1_1b` - Contains the UVM 1.1b library itself.

If you enter the `verif/alutb` directory and run the following, you should get a test to run and PASS.

```
verif/alutb> cnmake sim
```

If your simulation fails, proceed no further and seek some help.

Chapter 2: Your First Test

In this chapter, you will learn to use the UVM Template Generator to write a test in the `alutb` testbench.

Using the UVM Template Generator

The first objective is to write a test that speeds up the clock frequency.

The template generator automates the creation of UVM files and classes. You should always use it because it will save you a considerable amount of typing. It also creates files and classes with a consistent look and feel, which encourages readability.

The first thing to do is go into the `alutb/tests` directory and create a new test. The script is called `utg` and it takes as its argument the name of a template. You also give it the name of the file/class to create:

```
verif/alutb/tests> utg test -n fast_clk
++ Creating fast_clk.sv
fast_clk test: Enter substitution for <description>: A test that makes the clock go super fast!
//-*- mode: Verilog; verilog-indent-level: 3; indent-tabs-mode: nil; tab-width: 1 -*-

// ***************************************************************************
// * CAVIUM CONFIDENTIAL
// *
// *                          PROPRIETARY NOTE
// *
// * This software contains information confidential and proprietary to
// * Cavium, Inc. It shall not be reproduced in whole or in part, or
// * transferred to other documents, or disclosed to third parties, or
// * used for any purpose other than that for which it was obtained,
// * without the prior written consent of Cavium, Inc.
// * (c) 2013, Cavium, Inc.  All rights reserved.
// * (utg v0.8.2)
// ***************************************************************************
// File:   fast_clk.sv
// Author: username
/* About:  A test that makes the clock go super fast!
 ***************************************************************************/

`ifndef __FAST_CLK_SV__
   `define __FAST_CLK_SV__

   `include "base_test.sv"
// (`includes go here)

// class: fast_clk_test_c
// (Describe me)
class fast_clk_test_c extends base_test_c;
   `uvm_component_utils_begin(fast_clk_test_c)
   `uvm_component_utils_end
```

Because we did not add `-f` to the command-line, `utg` just printed its output to the screen.

As you can see, `utg` chose to name the class `fast_clk_test_c`, even though we asked it to be named `fast_clk`. This is because according to the coding guidelines, tests must

have the word `test` as a suffix **[Rule 2.6-1]**, and all classes must end in `_c` **[Rule 2.5-1]**. Some of our internal tools expect these conventions to be followed, and it is much easier to tell what kind of class it is, isn't it? It also created lots of (nearly) empty tasks and functions for you, to save you some typing. If you don't need these, you can leave them there for future expansion, or you can delete them if you like smaller files **[Rec 4.2-1]**.

`utg` also asked you to fill in a description. The templates have tagged identifiers in them, and if `utg` doesn't know what to put in there, it will ask you. In order to skip these requests and just have `utg` leave the tags in the document, add `-q` to the command-line. Note that some tags are not within comments, and will break the compile if they remain.

You may cut-and-paste this into a file, or you may have `utg` pipe this directly into a file called `fast_clk.sv`, with the `-f` argument:

```
verif/alutb/tests> utg test -n fast_clk -f -q
++ Creating fast_clk.sv
```

Now you have a file called `fast_clk.sv` in the `tests` directory and it is time to actually do something with it.

flist Files

But before we go much further, there's one bit of housecleaning we need to do. The `cnmake` script relies upon files called `flists` to help it know what SystemVerilog files to compile and in what order to compile them. Edit the file `verif/alutb/alutb.flist` and add your new test file to it, so that it looks like this:

1. **verif/alutb/alutb.flist**

```
+incdir+../../verif/alutb
+incdir+../../verif/common
+incdir+../../verif/uvm_common
+incdir+../../verif/hdl
+incdir+../../verif/alutb/tests
+incdir+../../rtl/include
../../verif/alutb/alutb_tb_top.sv
../../verif/alutb/alu_wrapper.sv
../../verif/alutb/tests/basic.sv
../../verif/alutb/tests/base_test.sv
../../verif/alutb/tests/fast_clk.sv
-f ../../verif/alutb/rtl.flist
```

The change you need to make is highlighted in **bold**. Each directory has an flist file in it, but as you'll see later you won't need to do this all the time.

Extended from the Base Test

Now take a look at your new test, `fast_clk.sv`. Pretty, isn't it? You'll notice that it extends from the class called `base_test_c`. In fact, all tests descend from this test, but the base test is not a test at all. Well, it is, but it's not one that does any testing.

Actually, `base_test_c` will be the tippy-top of the component hierarchy in every simulation you run. It's the one that instantiates all of the same components that (most) tests will share, so they don't have to do the same thing over and over.

Go ahead and take a gander at `base_test.sv`. It looks a lot like `fast_clk_test_c`, except it has *stuff* in it. As the coding guidelines say, it has all of its fields at the top, and all of its methods at the bottom **[Rec 4.2-1]**. The phase methods are declared in the order that they will run. Each section has a `Group:` comment, each field has a `field:` comment, and each method has a `func:` comment. These are parsed by the documentation generator to make all that pretty documentation **[Rule 2.8-1]**.

Changing the Clock Frequency

You're probably eager to actually write some code, so let's get to it. In `base_test_c`, look for the instantiation of the clock driver, `tb_clk_drv`, in the `build_phase` function. This driver is a component, and so can be configured directly through the `uvm_config_db::set` function. As you can see in the line just before it is instantiated, the base test has configured the period of the clock driver to be 2,000ps. Let's get it going faster in our new test.

Back in our `fast_clk_test_c` class, modify its `build_phase()` function to look like this:

2. verif/alutb/tests/fast_clk.sv

```
virtual function void build_phase(uvm_phase phase);
   super.build_phase(phase);
   uvm_config_db#(int)::set(this, "tb_clk_drv", "period_ps", 1800);
endfunction : build_phase
```

This will re-configure the clock driver's period to be 1800ps. A detailed explanation of how this works will be given in Chapter 7. For now know that it is important that these configuration calls happen during a component's build phase, and that they happen after you call `super.build_phase()`.

Now go ahead and simulate and you should see a period of 1.8ns. Here's the command-line, which you'll execute from the `verif/alutb` directory:

```
verif/alutb> cnmake sim UTEST=fast_clk
```

The `UTEST` command-line argument selects the UVM test to run. See cnmake's help function for more information on how to use that tool.

Overriding Constraints

That's great, you say, but what if we want more randomness? What if I want a clock period between 1.8ns and 2ns?

Well, that's what constraints are great for. A component is a SystemVerilog class, and the `period_ps` field is a `rand int`, so it can be randomized anytime after it is new'ed. So, let's change our `fast_clk_test_c::build_phase` to this instead:

3. verif/alutb/tests/fast_clk.sv

```
virtual function void build_phase(uvm_phase phase);
   super.build_phase(phase);
   tb_clk_drv.randomize(period_ps) with {
      period_ps inside {[1800:1999]};
   };
   `cn_info(("Selected a period of %0dps", tb_clk_drv.period_ps))
endfunction : build_phase
```

Note our first use of the messaging macro `` `cn_info ``.[1] These work similarly to the UVM reporting macros that they replace, but you need to use two sets of parentheses because the argument will be sent directly to `$sformatf`. Also, you are advised **not** to put a semi-colon at the end of any macro calls **[Rec. 3.6-8]** because depending on its location within the code and the macro definition (both of which may change over time), it may not behave as you expect.

Go ahead and run that, but add `NOBLD=lib` to your command-line. **This prevents cnmake from re-compiling the C code**, which hasn't changed at all:

```
verif/alutb> cnmake sim UTEST=fast_clk NOBLD=lib
```

Also you can tell `cnmake` to create waves for DVE by adding `WAVE=1` on the command-line, or Verdi with `FSDB=1`.

Question

If you take a look at the resultant wave file, you'll find that it didn't work. Instead, you got a period of 2,000ps, which is what the base test specified.

Do you know why? Do you know how to fix it?

[1] All of the messaging macro definitions can be found in `verif/vkits/cn_msgs.sv`.

Answer

Here's what happens:

1. The base test's build phase stuffs a period of 2,000ps into the resource database.
2. It then creates the `tb_clk_drv` component.
3. Then, we randomize the `tb_clk_drv` component and get a nice random value.
4. Later, the `tb_clk_drv` component's build phase runs, fetches the value of 2,000ps, and assigns it to the period.

So how do we fix it? Simple: we randomize it in a later phase. Either the `connect`, `end_of_elaboration`, or `start_of_simulation` phases will do, since they run after the `build_phase`.

You can move your randomize call from `build_phase` to `end_of_elaboration_phase` instead. Re-simulate and check your results.

Conclusion

This was a dive-right-in lesson that familiarized you with the UVM template generator, flist files, writing a little bit of SystemVerilog, and running using `cnmake`.

Chapter 3: Class Factories

In this chapter, you will learn to create a derived driver class and use the factory to instantiate it instead of the base class.

Extending the Driver

The goal of this chapter is to create a clock driver that has a 75/25 duty cycle. The generic clock driver, located in `verif/vkits/cn/cn_clk_drv.sv`, has lots of optional but complex functionality such as jitter and drift that we will not need. Instead, we want to create a simple clock driver that has a '1' for 75% of its period and a '0' for the remaining 25%.

To start, head over to the `verif/alutb` directory and create a generic component using `utg`:

```
verif/alutb> utg component -n clk_duty_cycle -f -q
++ Creating clk_duty_cycle.sv
```

This doesn't precisely do what you want. When you open the file, you'll see that there are still template variables all around because we specified `utg` to run quietly (`-q`). It also didn't know that we wanted to extend from an existing driver. But, using `utg` is still much faster than writing all of this boilerplate code. And, since we're creating this driver in the testbench itself, there is no package to reference. So you also need to get rid of all `<pkg_name>` things.

You'll want to extend this not from `uvm_component` but from the generic clock driver instead, which being in a separate package will need the scope operator (`cn_pkg::`).

Problem 3-1

Do your best to modify the `clk_duty_cycle_c` to drive a clock with a 75/25 duty cycle. Consider the variables `period_ps`, `init_delay`, `init_value`, and `init_x` as defined in the base class in your algorithm.

You won't be able to simulate it just yet.

Solution

Here is one possible solution:

4. verif/alutb/clk_duty_cycle.sv

```
// class: clk_duty_cycle_c
// A clock with a duty cycle of 75/25.
class clk_duty_cycle_c extends cn_pkg::clk_drv_c;
   `uvm_component_utils(clk_duty_cycle_c)

   //------------------------------------------------------------------------------
   // Group: Methods
   function new(string name="clk_duty_cycle",
                uvm_component parent=null);
      super.new(name, parent);
   endfunction : new

   ////////////////////////////////////////
   // func: run_phase
   virtual task run_phase(uvm_phase phase);
      int uptime = 3*period_ps / 4;
      int downtime = period_ps - uptime;

      // set to initial value
      clk_vi.clk = (init_x)? 'bx : init_value;
      #(init_delay_ps * 1ps);

      forever begin
         clk_vi.clk = 1;
         #(uptime * 1ps);
         clk_vi.clk = 0;
         #(downtime * 1ps);
      end
   endtask : run_phase

endclass : clk_duty_cycle_c
```

Note that all of the empty methods have been deleted. You could keep them there in case you need them later, with no effect on performance. However, one method that you can never remove is the constructor function, `new`. This function must always be present in derived UVM classes.

Using The Factory

The base test creates the testbench clock driver (`tb_clk_drv`) using the factory overridable create method:

```
      // Create the clock driver
      uvm_config_db#(string)::set(this, "tb_clk_drv", "intf_name", "tb_clk_vi");
      uvm_config_db#(int)::set(this, "tb_clk_drv", "period_ps",    2000);
      tb_clk_drv = cn_pkg::clk_drv_c::type_id::create("tb_clk_drv", this);
```

This is essentially the same as calling `new()`, but if there were earlier factory overrides, then they would be used. It is usually best to never call `new()` on a UVM object, but to call `create()` instead, as it provides this flexibility for free[2] **[Rec 10.5-3]**.

Problem 3-2

Create a test called `duty_cycle_test_c` that overrides the generic clock driver with your new clock driver. Run this test and see your new duty cycle.

Hint:

See `set_type_override_by_type()` in the UVM Reference.

[2] Well, almost free. The environment will perform a database lookup before newing the object. While this is hardly free in terms of run-time, the ability to dynamically alter the types of objects being created without having to re-write numerous lines of code can be a tremendous savings.

Solution

If you were successful, the signal `alutb_tb_top.tb_clk` should now have the new duty cycle. This is one possible solution:

5. verif/alutb/tests/duty_cycle.sv

```
`ifndef __DUTY_CYCLE_SV__
   `define __DUTY_CYCLE_SV__

   `include "base_test.sv"
   `include "clk_duty_cycle.sv"

// class: duty_cycle_test_c
// Run using the clk_duty_cycle_c instead of cn_pkg::clk_drv_c
class duty_cycle_test_c extends base_test_c;
   `uvm_component_utils(duty_cycle_test_c)

   //------------------------------------------------------------------------------
   // Group: Methods
   function new(string name="duty_cycle_test",
                uvm_component parent=null);
      super.new(name, parent);
   endfunction : new

   ////////////////////////////////////////////
   // func: build_phase
   virtual function void build_phase(uvm_phase phase);
      set_type_override_by_type(cn_pkg::clk_drv_c::get_type(),
                                clk_duty_cycle_c::get_type(), .replace(1));
      super.build_phase(phase);
   endfunction : build_phase

endclass : duty_cycle_test_c

`endif // __DUTY_CYCLE_SV__
```

There are two keys here: the first is that the `clk_duty_cycle.sv` file must be included by this file, because it is a dependency **[Rule 3.4.2.1-1]**. The second is that the `set_type_override_by_type` function call must precede the call to `super.build_phase()`. Do you know why? (if not, you will find out in Chapter 7).

You could also use `set_type_override_by_name` instead when you want to override a specifically named clock.

Problem 3-3

How could you run the earlier `fast_clk` test that you wrote with your new clock driver but *without* needing to modify any code at all?

Hint:

You can add runtime options to your `cnmake` command-line with:

```
verif/alutb> cnmake sim SIMOPTS+=+MY_PLUSARG=1
```

Solution

UVM offers a set of options that allow you to perform a `set_type_override` from the command-line. Running your `fast_clk` test with this command-line will do it:

```
6.
verif/alutb> cnmake sim UTEST=fast_clk \
                 SIMOPTS+="+uvm_set_type_override=cn_pkg::clk_drv_c,clk_duty_cycle_c"
```

Conclusion

This lesson introduced what class factories are capable of and had you write a smattering of SystemVerilog. Up next, you will start doing some real work on the ALU testbench.

Chapter 4: The ALU Protocol

The ALU module operates as a simple arithmetic logic unit. It has a 1-bit control signal, an 8-bit data input interface, and a 32-bit data output interface with a 1-bit output ready signal:

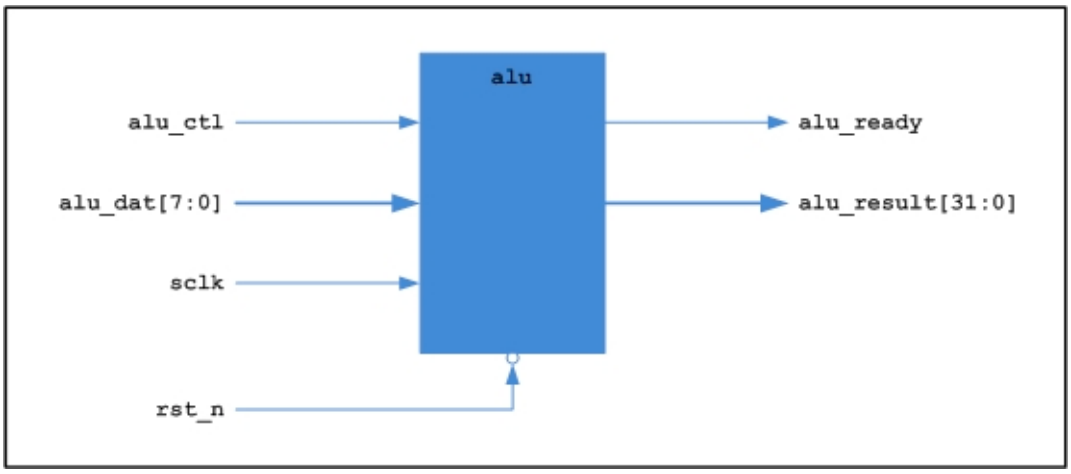


Figure 2: *The ALU Block*

Input data and control are sampled on the rising edge of `sclk`. The first cycle in an input transaction contains the 8-bit operation on `alu_dat[7:0]` and the `alu_ctl` signal is high. `alu_ctl` is held low for the remainder of the transaction and may not go high again to signal a new transaction until after the `alu_ready` output signal has been sampled high. The `alu_dat[7:0]` lines contain the remaining operands based on the ten operation types as per this chart:

Operation	Cycle 0	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Formula	Notes
ADD_A_B	8'h0	a[15:8]	a[7:0]	b[15:8]	b[7:0]	k * (a+b) + c	
SUB_A_B	8'h1	a[15:8]	a[7:0]	b[15:8]	b[7:0]	k * (a-b) + c	a > b
SUB_B_A	8'h2	a[15:8]	a[7:0]	b[15:8]	b[7:0]	k * (b-a) + c	b > a
MUL_A_B	8'h3	a[15:8]	a[7:0]	b[15:8]	b[7:0]	k * (a*b) + c	
DIV_A_B	8'h4	a[15:8]	a[7:0]	b[15:8]	b[7:0]	k * (a/b) + c	b != 0
DIV_B_A	8'h5	a[15:8]	a[7:0]	b[15:8]	b[7:0]	k * (b/a) + c	a != 0
INC_A	8'h6	a[15:8]	a[7:0]	N/A	N/A	k * (a+1) + c	
INC_B	8'h7	b[15:8]	b[7:0]	N/A	N/A	k * (b+1) + c	
CLR_RES	8'h8	N/A	N/A	N/A	N/A	0	Sets previous result to zero
ACCUM	8'h9	a[15:8]	a[7:0]	N/A	N/A	a+result	Uses previous result value

Table 1: *ALU Protocol*

As you can see, some transactions take fewer cycles than others. The result is valid on the assertion of the `ready` signal, and the number of cycles that takes also depends on the operation chosen.

The `k` and `c` values are the 8-bit values programmed into the CSR named `CONST`. The sum of all results since it was last read should be in the CSR `RESULT[SOR]`, which is a read-clear CSR.

Your job in future chapters will be to verify this simple sub-module using an agent that you create, leverage your work in a sub-system environment, and maybe even find a bug or two.

ALU_CONST = 0x7E8				
Bit	**Field Name**	**Type**	**Reset**	**Description**
<63:16>	NS	RAZ	NS	Reserved
<15:8>	K_VAL	R/W	1	The constant value that is first multiplied to the results of all ALU computations before the C_VAL is added in.
<7:0>	C_VAL	R/W	0	The constant value that is added to the results of all ALU computations.

Table 2: *ALU CONST Register Details*

ALU_RESULT = 0x7F0				
Bit	**Field Name**	**Type**	**Reset**	**Description**
<63:32>	NS	RAZ	NS	Reserved
<31:0>	SOR	RC	0	The sum of all results so far.

Table 3: *ALU RESULT Register Details*

Chapter 5: Creating a Transaction

It is often considered good Object-Oriented Programming (OOP) practice to begin with figuring out what the data should look like. We'll create a transaction that can be generated randomly by a sequence, sent to the driver via a sequencer, and monitored by a monitor. The predictor will also be able to receive these transaction classes. It would also be nice to be able to pack and unpack these transactions off the wire.

We will put all of our ALU-related code in a separate vkit, because it is possible that the ALU interface is common to multiple blocks. A *vkit* is composed of any reusable code, if any other testbenches wanted to use just the ALU agent, this might be appropriate. If not, then putting this code in the ALUTB vkit would be correct.

With these goals in mind, let's get started.

Creating a vkit

A vkit is meant to be a single reusable package that can be compiled with the help of other vkits. For a detailed discussion on what a vkit is and what it isn't, see the SV Coding Guidelines on the wiki.

To create a vkit, you can again use `utg` in the `verif/vkits` directory:

```
verif/vkits> utg vkit -n alu
++ Creating alu_pkg.sv
++ Created vkit alu
++ Exiting.
```

This creates the `alu` directory, the `alu_pkg.sv` package file, and an `alu.flist` file for you. Other vkits that this package relies upon will be placed ahead of this one in the build order. Then, the files needed to create the ALU package will be `` `included `` inside the package file. You will not need to touch the `alu.flist` file at this time.

To incorporate this new vkit into the ALUTB testbench, you need to modify the `alutb/Makefile` and add the path to this vkit's flist file to the `FLISTS` variable. **The order in which it is added is important based on this vkit's dependencies on other vkits**. This vkit will depend upon `cn` and `global` flists, so add it after those. The `alutb` testbench depends upon this vkit, so it needs to be added before that flist as well.

The FLISTS variable in the file alutb/Makefile should now be set as follows:

7. verif/alutb/Makefile

```
FLISTS= verif/vkits/cn/cn.flist \
            verif/vkits/global/global.flist \
            verif/vkits/sps/sps.flist \
            verif/vkits/cn_csr/cn_csr.flist \
            verif/vkits/reg/reg.flist \
            verif/vkits/ctx/ctx.flist \
            verif/vkits/alu/alu.flist \
            verif/vkits/alutb/alutb.flist \
            verif/alutb/alutb.flist \
            verif/alutb/rtl.flist
```

A Sequence Item is a Transaction

The class uvm_sequence_item derives directly from uvm_transaction, but has the additional property of being able to be driven by a sequence through a sequencer and on to a driver, which are components that we'll discuss in the near future. It is a rule that we must derive our new class from uvm_sequence_item **[Rule 10.6-1]**.

Create the sequence item file by specifying the item template. Call it 'item', and have it saved to the file named alu_item.sv:

8.

```
verif/vkits/alu> utg item -n item --filename alu_item.sv -q
++ Creating alu_item.sv
```

Do not forget to `include it into the ALU package located in verif/vkits/alu/alu_pkg.sv.

The first field in our transaction is the operation, which is a good place for an enumerated type. Enumerated types need to be specified before the call to `uvm_field_enum, so put the following at the very top of the class **[Rec 4.2-1]**:

9. verif/vkits/alu/alu_item.sv

```
//------------------------------------------------------------------------------
// Group: Types
typedef enum bit [7:0] {ADD_A_B = 0,
                        SUB_A_B = 1,
                        SUB_B_A = 2,
                        MUL_A_B = 3,
                        DIV_A_B = 4,
                        DIV_B_A = 5,
                        INC_A   = 6,
                        INC_B   = 7,
                        CLR_RES = 8,
                        ACCUM   = 9
                        } operation_e;
```

Notice that we declared it with the bit [7:0] syntax to tell SystemVerilog how wide it should be. This will be useful when we pack and unpack this transaction. Also notice that we provided the suffix of _e to help identify it as an enumerated type, as required in the SV Coding Guidelines **[Rule 2.5-1]**.

Now we add it to the list of fields with the `uvm_field_enum macro:

10. verif/vkits/alu/alu_item.sv

```
`uvm_object_utils_begin(alu_pkg::item_c)
   `uvm_field_enum(operation_e, operation, UVM_ALL_ON)
`uvm_object_utils_end
```

We chose to give it the flag UVM_ALL_ON because we want UVM to pack it, print it, and all the other fancy stuff that UVM provides for free.

Now we need to declare the operation as a member field of the class. Put this in the Fields group:

11. verif/vkits/alu/alu_item.sv

```
//------------------------------------------------------------------------------
// Group: Fields
// field: operation
rand operation_e operation;
```

We declared it as a random field because we're going to want to produce lots of random transactions later on.

Now let's do what we need to do for variables a and b:

12. verif/vkits/alu/alu_item.sv

```
`uvm_object_utils_begin(alu_pkg::item_c)
   `uvm_field_enum(operation_e, operation, UVM_ALL_ON)
   `uvm_field_int (alpha,                  UVM_ALL_ON | UVM_NOPACK)
   `uvm_field_int (beta,                   UVM_ALL_ON | UVM_NOPACK)
`uvm_object_utils_end
```

We chose to call them alpha and beta because a and b are not very good variable names **[Rec 4.3-1]**. We also specified that they are not to be packed. This, as you'll see later, is important when we go to pack and unpack these transactions, because A and B do not appear in all operation types.

Declare the alpha and beta fields as rand bit [15:0].

Typedefs Rule

At some point in the life of this ALU, somebody's going to want to make a bus wider. Or narrower. Or they're going to add a square root operation. Or they're going to turn it into a router that sorts packets by size. Whatever happens, it will almost certainly change somehow at some point, and when it does you're going to have to change it in a *lot* of places. Using constants would require you to only make the change in one place. And defining constants is much safer because it prevents typos from creating a nightmare debugging session.

However, defining constants for bus widths using `defines is so passé. Instead, we're going to create a typedef for the result type. All typedefs end with _t **[Rule 2.5-1]**:

13. verif/vkits/alu/alu_pkg.sv
```
typedef logic [31:0] result_t;
```

That will also be much easier to type all over the place than this would:

```
logic [`RESULT_WIDTH-1:0] result;
```

You can place your typedefs in the vkit's package (above all of the included files that depend on it), or you can create an alu_types.sv file and `include it in any file that needs it. The choice is yours. Either way, the type will be local to the vkit's package, so there's no concern for namespace pollution.

Create typedefs for any type that will be commonly used within a package, including enumerated types. The operation_e enumerated type was placed in the item_c class, but it could just as easily have been placed in the package. You may also, for example, create types for the operands alpha and beta.

Sequence Item U-Turns

We're also going to put the result field in the transaction, like so:

14. verif/vkit/alu/alu_item.sv
```
    // field: result
    // This is the result of the operation, filled in by the driver and sent back with the response
    result_t result;
```

When the transaction is sent to the driver by the sequencer, the driver will (eventually) send a response back. Sometimes, a block gives no response (for one-way communications, for example). In cases such as those, it is common for the driver to send the same request item back to the sequencer after the transaction has been completely driven.

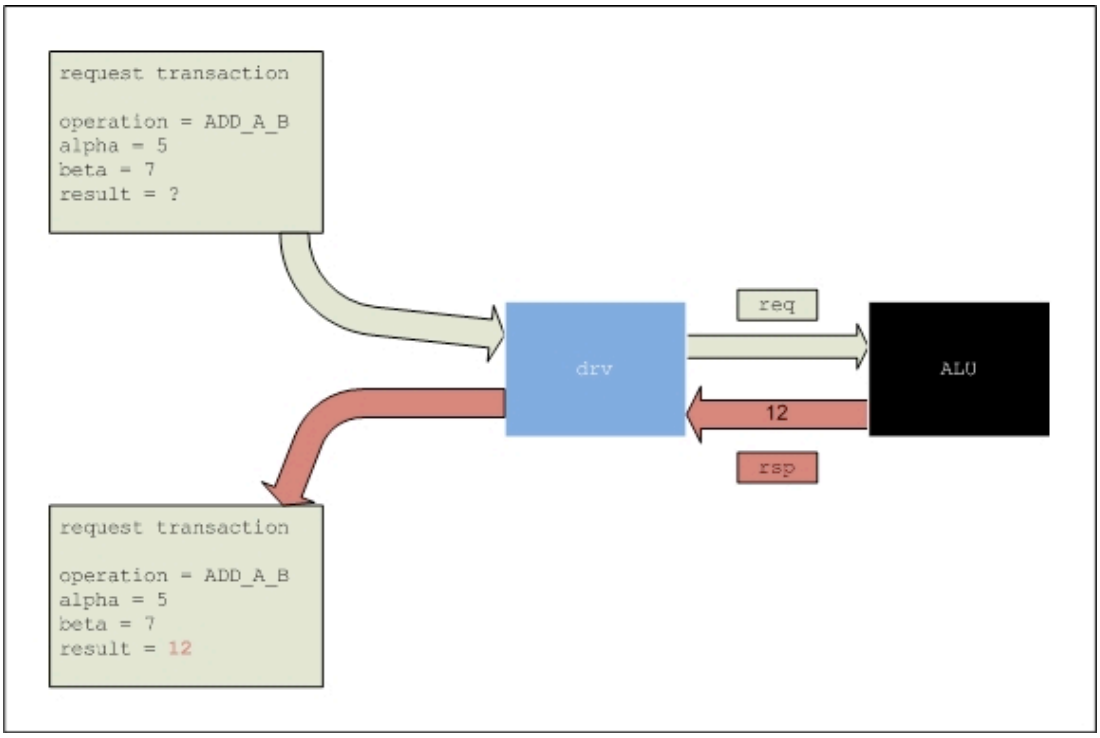


Figure 3: *Sequence Item U-Turns*

In our case, there is a response. But it is just a 32-bit integer. Both request and response types have to be derived from `uvm_sequence_item`, to ensure that they have all the functionality that the sequencer needs to track them. So, our design will simply have the driver put the result back into the request transaction and send it back once it has been received.

Adding Constraints

You can't divide by zero. And the transaction rules don't allow for negative results during subtractions, either. Constraints are the way to do this. Fortunately, SystemVerilog really shines in this department.

Problem 5-1

Exercise your SystemVerilog muscles to add constraints such that all randomized transactions follow the protocol.

Solution

There are several ways you can approach this, but this is probably the clearest and most concise way:

15. verif/vkits/alu/alu_item.sv

```
// ensure that all operands have legal values
constraint protocol_cnstr {
   (operation == DIV_A_B) -> beta != 0;
   (operation == DIV_B_A) -> alpha != 0;
   (operation == SUB_A_B) -> alpha >= beta;
   (operation == SUB_B_A) -> beta >= alpha;
}
```

Note that all constraint blocks need a name, and they should end in `_cnstr` **[Rule 2.5-1]**. Also, you should put your constraints near the field declarations themselves, in the Fields group.

Printing Your Classes

Printing your UVM classes to a logfile is something you'll want to do all over the place. All UVM classes have a method that returns the class as a string—`sprint`. This function takes an optional argument of type `uvm_printer` which you can use to define what you want your string to look like. This class is called a policy class, and they are sprinkled all over UVM. You then pass this string to one of the `` `cn `` messaging macros, `` `cn_info ``, `` `cn_dbg ``, `` `cn_warn ``, `` `cn_err ``, or `` `cn_fatal ``.

Table Format

Table format is the default, so you do not need to provide a printer class to the `sprint` function. The following code snippet will produce the table format of a class:

```
`cn_info(("Monitored:\n%s", item.sprint()))
```

And the results are shown below:

```
%I-(   alu_mon.sv:   140) [alutb_env.alu_agent.mon              ] { 38597ns} Monitored:
----------------------------------------------------------------------------------------------------
Name                                Type                                Size  Value
----------------------------------------------------------------------------------------------------
mon_item                            alu_pkg::item_c                     -     @6203
  operation                         operation_e                         8     DIV_A_B
  alpha                             integral                            16    'hc6eb
  beta                              integral                            16    'h8a74
----------------------------------------------------------------------------------------------------
```

By default, your printed class uses the table printer, which prints your class in a nice, easy-to-read format. However, it is expensive in simulation time. For the thrifty, consider using the tree format instead.

Tree Format

To print in the tree format, use the following instead.

```
`cn_info(("Monitored:\n%s", item.sprint(uvm_default_tree_printer)))
```

And it appears like this in the logfile:

```
%I-(   alu_mon.sv:   142) [alutb_env.alu_agent.mon              ] { 38573ns} Monitored:
mon_item: (alu_pkg::item_c@6195) {
  operation: DIV_A_B
  alpha: 'hcc3d
  beta: 'h38de
}
```

The `uvm_default_*_printer` objects are globally scoped instances of the `uvm_printer` policy classes that can be used when you don't want to type too much. Alternatively, you can create your own tree printer instance and use it over and over again:

```
uvm_tree_printer tp = new();
`cn_info(("Monitored:\n%s", item.sprint(tp)))
```

The benefit of this approach is that you can tweak a variety of printer knobs to get your class just as pretty as you like. The variety of knobs available is impressive. See the UVM reference manual's description of the `uvm_printer_knobs` class for details.

The same approach can be used for the table and line printer classes, too.

Line Format

The cheapest formatting is the line printer:

```
`cn_info(("Monitored:\n%s", item.sprint(uvm_default_line_printer)))
```

And its appearance is as shown here:

```
%I-( alu_mon.sv:   142) [alutb_env.alu_agent.mon ] { 38597ns} Monitored:
mon_item: (alu_pkg::item_c@6203) { operation: DIV_A_B  alpha: 'hc6eb  beta: 'h8a74  }
```

Convert2String

It is very handy to single-line print things—especially small transactions like these. But UVM's standard `uvm_line_printer` format is not terribly readable. The `convert2string()` function is defined by UVM as an empty function and is our place to do this **[Rec 10.4-1]**, and `$sformatf` is the right tool for the job:

16. verif/vkits/alu/alu_item.sv

```
////////////////////////////////////////////
// func: convert2string
// Single-line printing
virtual function string convert2string();
   convert2string = $sformatf("%s A:%04X B:%04X", operation, alpha, beta);
endfunction : convert2string
```

You can optionally make your function fancier by not printing out the A and B operands for operations that don't use them, but this will suffice for now. Now your transaction will print out nice and neat and just the way you like it:

```
%I-(   alu_drv.sv:   107) [alutb_env.alu_agent.drv  ] {   890ns} Monitored: DIV_A_B A:C6EB B:8A74
```

Packing and Unpacking

There are many routes that will get us to our end goal, but this is a great opportunity to learn about how to pack and unpack in UVM. Later, the driver will be able to pack this transaction down to the bytes that it will put on the wire, and the monitor will be able to collect the bytes and unpack them.

When you call the pack function on a transaction, it will automatically pack all of the fields that are marked by the flags in the macros as packable. Because the `alpha` and `beta` fields will not always be used, though, we'll need to *conditionally* pack them. To add this functionality to a UVM object, you override the function `do_pack`:

17. verif/vkits/alu/alu_item.sv

```
////////////////////////////////////////////
// func: do_pack
virtual function void do_pack(uvm_packer packer);
   super.do_pack(packer);

   if(operation inside {[ADD_A_B : INC_A], ACCUM})
      packer.pack_field_int(alpha, 16);
   if(operation inside {[ADD_A_B : DIV_B_A], INC_B})
      packer.pack_field_int(beta, 16);
endfunction : do_pack
```

We've chosen to use SystemVerilog's handy `inside` operator to conditionally pack these ints as 16-bit values.

Like the `sprint` function, the `pack` and `unpack` functions take an optional policy class called `uvm_packer`. Here, we let the policy class do the work for us, in case the caller of pack or unpack chose to modify the default policy to handle things differently.

The following example shows how the user can pack into a list of bytes in little-endian format instead of the default big-endian format:

```
uvm_packer little_endian = new();
byte unsigned stream[];

little_endian.big_endian = 0;
item.pack_bytes(stream, little_endian);
```

Problem 5-2

With the `do_pack` method as a boilerplate, write the corresponding `do_unpack` method.

Solution

The solution should look very familiar:

18. verif/vkits/alu/alu_item.sv

```
////////////////////////////////////////////
// func: do_unpack
virtual function void do_unpack(uvm_packer packer);
   super.do_unpack(packer);

   if(operation inside {[ADD_A_B : INC_A], ACCUM})
      alpha = packer.unpack_field_int(16)
   if(operation inside {[ADD_A_B : DIV_B_A], INC_B})
      beta = packer.unpack_field_int(16);
endfunction : do_unpack
```

Problem 5-3

Let's now fill in the test `basic_test_c` such that during the `main_phase` it creates a stream of 50 random ALU transactions, prints them out, and packs them into a list of unsigned bytes. Then, unpack this list of bytes into another transaction, and print it out. Finally, compare the two transactions and print an error if they do not compare.

Use `` `cn_info `` and `` `cn_err `` to print informational messages and error messages, respectively. You can also use the function `cn_pkg::print_ubyte_array()` to print out your byte array (or `print_byte_array` if you used signed bytes), to aid in debugging your work.

What happens?

Hint:

See `do_compare` in the UVM reference.

Solution

Here is one possible solution:

19. verif/alutb/tests/basic.sv

```
class basic_test_c extends base_test_c;
   `uvm_component_utils(basic_test_c)

   //------------------------------------------------------------------------------
   // Group: Methods
   function new(string name="test",
                uvm_component parent=null);
      super.new(name, parent);
   endfunction : new

   ////////////////////////////////////////
   // func: main_phase
   virtual task main_phase(uvm_phase phase);
      byte unsigned stream[];
      alu_pkg::item_c item;
      alu_pkg::item_c unp_item = alu_pkg::item_c::type_id::create("unp_item");;

      phase.raise_objection(this);

      repeat(50) begin
         item = alu_pkg::item_c::type_id::create("item");
         item.randomize();
         `cn_info(("Created ALU transaction: %s", item.convert2string()))
         item.pack_bytes(stream);
         `cn_info(("Bytes: %s", cn_pkg::print_ubyte_array(stream)))
         unp_item.unpack_bytes(stream);
         `cn_info(("Unpacked: %s", unp_item.convert2string()))
         if(item.compare(unp_item) == 0)
            `cn_err(("Miscompare!"))
      end

      phase.drop_objection(this);
   endtask : main_phase

endclass : basic_test_c
```

Your test will almost certainly fail for the increment operations. But why?

Well, we're both randomizing and comparing the `alpha` and the `beta` fields unconditionally, yet these fields will be neither packed nor unpacked, so will be different for the increment operations which do not use them all.

Problem 5-4

This problem can be solved by overriding the `do_compare()` method, in the same manner as the `do_pack()` and `do_unpack()` methods. What else do you need to do?

Solution

You'll need to add the flag `UVM_NOCOMPARE` to the `alpha` and `beta` field macro calls. If everything worked correctly, your test should PASS. Notice that the `do_compare` function must first `$cast` the object passed to it to the type `item_c` before you can look at its `alpha` or `beta` member fields.

The complete class looks like this:

20. **verif/vkits/alu/alu_item.sv**

```
// class: item_c
// An ALU Transaction as a sequence item
class item_c extends uvm_sequence_item;
   //------------------------------------------------------------------------------
   // Group: Types
   typedef enum bit [7:0] {
                           ADD_A_B = 0,
                           SUB_A_B = 1,
                           SUB_B_A = 2,
                           MUL_A_B = 3,
                           DIV_A_B = 4,
                           DIV_B_A = 5,
                           INC_A   = 6,
                           INC_B   = 7,
                           CLR_RES = 8,
                           ACCUM   = 9
                           } operation_e;

   `uvm_object_utils_begin(alu_pkg::item_c)
      `uvm_field_enum(operation_e, operation, UVM_ALL_ON)
      `uvm_field_int (alpha,                  UVM_ALL_ON | UVM_NOPACK | UVM_NOCOMPARE)
      `uvm_field_int (beta,                   UVM_ALL_ON | UVM_NOPACK | UVM_NOCOMPARE)
   `uvm_object_utils_end

   //------------------------------------------------------------------------------
   // Group: Fields

   // field: operation
   rand operation_e operation;

   // field: A variable
   rand bit [15:0] alpha;

   // field: B variable
   rand bit [15:0] beta;

   // ensure that all operands have legal values
   constraint protocol_cnstr {
      (operation == DIV_A_B) -> beta != 0;
      (operation == DIV_B_A) -> alpha != 0;
      (operation == SUB_A_B) -> alpha > beta;
      (operation == SUB_B_A) -> beta > alpha;
   }

   // field: result
   // This is the result of the operation, filled in by the driver and sent back with the response
   result_t result;

   //------------------------------------------------------------------------------
   // Group: Methods
   function new(string name="item");
      super.new(name);
   endfunction : new

   //------------------------------------------------------------------------------
```

```
    // func: convert2string
    // Single-line printing
    virtual function string convert2string();
       convert2string = $sformatf("%s A:%04X B:%04X", operation, alpha, beta);
    endfunction : convert2string

    ////////////////////////////////////////////
    // func: do_pack
    virtual function void do_pack(uvm_packer packer);
       super.do_pack(packer);

       if(operation inside {[ADD_A_B : INC_A], ACCUM})
          packer.pack_field_int(alpha, 16);
       if(operation inside {[ADD_A_B : DIV_B_A], INC_B})
          packer.pack_field_int(beta, 16);
    endfunction : do_pack

    ////////////////////////////////////////////
    // func: do_unpack
    virtual function void do_unpack(uvm_packer packer);
       super.do_unpack(packer);

       if(operation inside {[ADD_A_B : INC_A], ACCUM})
          alpha = packer.unpack_field_int(16);
       if(operation inside {[ADD_A_B : DIV_B_A], INC_B})
          beta = packer.unpack_field_int(16);
    endfunction : do_unpack

    ////////////////////////////////////////////
    // func: do_compare
    virtual function bit do_compare(uvm_object rhs,
                                    uvm_comparer comparer);
       item_c _rhs;
       $cast(_rhs, rhs);
       do_compare = super.do_compare(rhs, comparer);
       if(operation inside {[ADD_A_B : INC_A], ACCUM})
          do_compare &= comparer.compare_field_int("alpha", alpha, _rhs.alpha, 16);
       if(operation inside {[ADD_A_B : DIV_B_A], INC_B})
          do_compare &= comparer.compare_field_int("beta", beta, _rhs.beta, 16);
    endfunction : do_compare

endclass : item_c
```

Printing, Comparing, Packing...Recording?

You've already figured out how to customize a UVM object's comparison mechanism by overriding the `do_compare` function. You'll note that it, too, takes an optional policy class. As you've seen, UVM uses these policy classes together with the `do_*` functions and the UVM field flags such as `UVM_NOCOMPARE` to customize your classes to operate exactly the way you want them to.

UVM also includes a recording mechanism, but what does it do? The UVM reference says that it is "vendor-specific." In fact, it can be used by the simulation tool, the waveform dumper, or any other third-party tool that might come along. You could override it to write itself to a database for further analysis, for example.

Adding Coverpoints

In a coverage driven verification environment, *now*—when you've just created the transaction—is the time to create coverpoints. Start collecting functional coverage early in the project, and collect it often. Creating metrics to chart the progress of the verification environment as early as possible is very beneficial to long-term planning. Plus you may later be pleasantly surprised at what you're already covering now that you won't have to write a test for someday.

Because this is a UVM tutorial, though, we will not be going over how to create covergroups and coverpoints.

Conclusion

In this lesson, we created a transaction object that represents an ALU request, and we derived it from `uvm_sequence_item`, for reasons we'll explore later. We also briefly touched on how to customize the packing, printing, comparing, and recording of any UVM object. The level of customization that UVM provides for this is deep, and you should refer to the UVM Reference Manual when you next want to pursue these topics.

Chapter 6: Creating an Interface

A few fun facts about interfaces that you may have missed during your SystemVerilog 101 training:

- Interfaces are *not* just a bundle of wires. They are really more like classes or modules. This is because they can contain their own tasks and functions, their own member fields, assertions, procedural blocks (initial and always statements), assignments, checkers, enumerated types...the list of things you can stuff into an interface is very long. *And* they can be passed between RTL modules as a synthesizable construct (that bundle of wires analogy). Unlike classes, though, they are not dynamically created at runtime, but statically created as part of the RTL hierarchy just as modules, regs, and wires are.

- Interfaces cannot be defined in a package. As much as you'd like to define your ALU interface in your ALU package, the language won't allow it. Instead, the interface must be `` `included `` and *known* to the compiler before it compiles any packages that depend on it because...

- Testbench code (i.e. that which lives in the `verif/alutb` directory) can access hard-coded paths such as `alutb_tb_top.ctx_i.out[7:0]`. Packages—such as those which reside in *vkit* directories—cannot. Doing so would break the reusability aspect of a *vkit* because in a full-chip testbench, the path `alutb_tb_top` does not exist. Instead, *all* signals accessed by testbench code within a package must come via a *virtual interface*.

- The UVM method of getting a class in a package to see a testbench interface is to put a reference to the interface in the UVM resource database. This reference is placed in the RTL-side of the testbench and is then pulled out of the database by the UVM class's `build` phase.

- An interface may have input (or output) signals, typically a clock and a reset signal. These signals are not specific to the interface itself, because they may be shared with other devices (i.e. `tb_clk`, `tb_rst_n`). If a signal is specific to the interface (i.e. `pcie_clk`, `srio_clk`, etc.) then it *might* be a member of that interface.

- Interfaces may contain other interfaces. If a single interface has four separate 'channels', then it is often convenient to create a single channel interface, and repeat it four times in a container interface. This is more robust than naming its signals `valid0`, `valid1`, etc.

- Clocking blocks offer you the ability in one place to specify the clock edge on which signals will be sampled or driven. By using different clocking blocks for each component that will access a given signal, you can easily change these edges in one place.

- Clocking blocks also solve race conditions between the RTL and the verification environment that were previously solved in SystemVerilog by the `program`/`endprogram` constructs.

- Modports offer access protection against modules, drivers, and monitors from touching signals that they shouldn't. In a way, they act like protected access in C ++. Given that an interface can have a task or function that accesses signals, it could be easy for a monitor that shouldn't touch any signals to accidentally start affecting things.

- You don't *need* to use clocking blocks and modports, as they add additional typing when accessing signals. You'll have to judge whether their benefits outweigh this additional overhead. We'll use both of them in our example just so you can see how they are used.

Problem 6-1

Take a stab at creating an interface for the ALU signals. Use the `utg` template `intf`. The filename should be `alu_intf.sv`, and it should reside in the `verif/vkits/alu` directory. The interface itself should be named `alu_intf`. Write a `reset()` function that sets the control and input data signals to zero. A clock and an active-low reset signal should be inputs to the interface.

You can use the file `verif/vkits/ctx/ctx_intf.sv` as a guide to help you.

Extra credit:

Add assertions that ensure that there are no X's when there shouldn't be. The `` `cn_err `` macro cannot be used in an interface (it can only be used in verification environment code that derives from `uvm_object` because these macros call the `get_full_name` function). You must use the `` `cn_err_intf `` macros instead.

Solution

Admittedly this one was probably a lot to ask you for, and you don't yet have a way to simulate it to see if you're right. But hopefully the effort was worthwhile:

21. verif/vkits/alu/alu_intf.sv

```
interface alu_intf(input logic clk,
                   input logic rst_n);
   import uvm_pkg::*;

   //---------------------------------------------------------------------------------------------
   // Group: Signals

   // var: ctl
   // Asserted only on the first cycle of a new transaction, while dat contains the operation
   logic        ctl;

   // var: dat
   // The input data nibble that contains the operation and operands
   logic [7:0]  dat;

   // var: ready
   // The output of the ALU that indicates when the result data is valid
   logic        ready;

   // var: result
   // The 32-bit result data
   logic [31:0] result;

   //---------------------------------------------------------------------------------------------
   // Group: Clocking blocks

   // var: drv_cb
   // A clocking block that represents how the environment's driver sees the interface
   clocking drv_cb @(posedge clk);
      output    ctl;
      output    dat;
      input     ready;
      input     result;
      input     rst_n;
   endclocking : drv_cb

   // var: mon_cb
   // A clocking block that represents how the environment's monitor sees the interface
   clocking mon_cb @(posedge clk);
      input     ctl;
      input     dat;
      input     ready;
      input     result;
      input     rst_n;
   endclocking : mon_cb

   //---------------------------------------------------------------------------------------------
   // Group: Modports

   modport drv_mp(clocking drv_cb,
                  import reset);
   modport mon_mp(clocking mon_cb);

   //---------------------------------------------------------------------------------------------
   // Group: Methods

   // func: reset
   // Convenience function for the driver to reset its outputs
   function void reset();
      ctl = 0;
      dat = 8'b0;
```

```
    endfunction : reset

    //---------------------------------------------------------------------------------------
    // Group: Assertions

    ctl_not_x :
       assert property(@(posedge clk)
                       disable iff (~rst_n || rst_n === 1'bx)
                       (!$isunknown(ctl))) else
          `cn_err_intf(("ctl signal is an X"))

    dat_not_x :
       assert property(@(posedge clk)
                       disable iff (~rst_n || rst_n === 1'bx)
                       (!$isunknown(dat))) else
          `cn_err_intf(("dat signal is an X"))

    ready_not_x :
       assert property(@(posedge clk)
                       disable iff (~rst_n || rst_n === 1'bx)
                       (!$isunknown(ready))) else
          `cn_err_intf(("ready signal is an X"))

    result_not_x :
       assert property(@(posedge clk)
                       disable iff (~rst_n || rst_n === 1'bx || ready == 0)
                       (!$isunknown(result))) else
          `cn_err_intf(("result signal is an X"))

endinterface : alu_intf
```

The ALU Interface Explained

- Interface signals should be declared as `logic`, not `wire` or `reg`. The reason SystemVerilog needed to create the `logic` type was for this very reason: these signals could be driven either by a continuous assignment (like a `wire`) or with a procedural assignment (like a `reg`), depending upon the implementation to which it is connected. Thus the need for this new data type.

- We created 2 different types of clocking blocks and 2 modports: one for a monitor that drives no signals, and one for an environment driver that drives the `ctl` and `dat` signals. Only the driver will be allowed to call the reset function to clear the `ctl` or `dat` signals. If the RTL uses interfaces to wire itself up, more clocking blocks and modports might be added, too.

- Because the `` `cn_err_intf `` macro still refers to `uvm_top` and `uvm_report_handler`, the interface namespace must import the UVM package.

Connecting the Interface to the DUT

Now that you've defined the interface, you need to let the simulator see it. This is accomplished by adding the file to the `vkits/alu/alu.flist` file. It must be added before the ALU environment package, because the package depends on the interface, not the other way around:

22. verif/vkits/alu/alu.flist

```
+incdir+../../verif/vkits/alu
../../verif/vkits/alu/alu_intf.sv
../../verif/vkits/alu/alu_pkg.sv
```

Now we will instantiate a single ALU interface in the `alutb_tb_top` module after the CTX interface, and reset it at time zero so that we do not trip our assertions:

23. verif/alutb/alutb_tb_top.sv

```
   // field: alu_i
   // The <alu_intf> instance
   alu_intf alu_i(.clk(tb_clk), .rst_n(tb_rst_n));
   initial
      alu_i.reset();
```

You probably look at all the extra documentation comments as being unnecessary, and for such a simple testbench they probably are. But when things get more complicated or are written by others, you will appreciate having pretty documentation to look at **[Rec 2.8-2]**. These comments help create that documentation.

In this testbench, the `alu` DUT is instantiated in the `alu_wrapper` module. Add this interface as an argument to that module as the other interfaces are:

24. verif/alutb/alu_wrapper.sv

```
module alu_wrapper(input logic tb_clk,
                   tb_rst_n,
                   ctx_intf ctx_i,
                   alu_intf alu_i);
```

Strictly speaking, the wrapper module is not *necessary*. The purpose of this wrapper module is to consolidate the instantiation of the DUT and all of the wiring that is required such that the top-level testbench can be more easily read and understood.

In the future, perhaps this RTL module will just accept the SystemVerilog interfaces. But like many legacy modules, this one doesn't. Instead, you need to wire up each signal to the interface directly, or you can use emacs' verilog-mode and its auto-mode features (by pressing `C-c C-a`).

Assuming you've done everything correctly, you may now run one of your tests as a sanity check. Everything should still PASS because hooking the interface up to the DUT is (theoretically) innocuous.

Storing the Interface in the Database

The `pre_run_test()` function must appear in all testbenches **[Rule 5.1-2]**, because it will be called by the code that sits in the `verif/hdl/tbv_common.v` file to ensure that all interfaces are *set* in the resource database before any agents try to *get* them. As you can see in the `alutb_tb_top` testbench, several interfaces have already been set into the database.

You'll want to put an instance of the driver and the monitor modports (`drv_mp` and `mon_mp`, respectively) into the resource database so that the driver and monitor that will be written later can get them. You will also want to virtualize these interfaces with the `virtual` keyword, because that is what the classes will use.

The name for the interfaces here must match the name of the interfaces that the driver and monitor use to get them. As you'll see later, making that name a configuration field of those components will allow you to attach different instances of components to different interfaces, all on a configurable basis.

Problem 6-2

Put virtual driver and monitor modports into the resource database in the `alutb_tb_top.pre_run_test()` function, using the other examples as models. Name the resource for both `alu_pkg::alu_intf`, and name the interfaces `drv_vi` and `mon_vi`, respectively. See **[Rule 5.3.1-3]**.

Solution

This should do the trick:

```
uvm_resource_db#(virtual alu_intf.drv_mp)::set("alu_pkg::alu_intf", "drv_vi", alu_i.drv_mp);
uvm_resource_db#(virtual alu_intf.mon_mp)::set("alu_pkg::alu_intf", "mon_vi", alu_i.mon_mp);
```

However, these handy macros simplify the process somewhat **[Sug 5.3-2]**:

```
25.     verif/alutb/alutb_tb_top.sv
`cn_set_intf(virtual alu_intf.drv_mp, "alu_pkg::alu_intf", "drv_vi", alu_i.drv_mp)
`cn_set_intf(virtual alu_intf.mon_mp, "alu_pkg::alu_intf", "mon_vi", alu_i.mon_mp)
```

Conclusion

In SystemVerilog, testbench-related code is capable of reaching down into the hierarchy and peeking and poking at signals. All reusable components, though, are kept in packages that have no notion of the hierarchy. The bundle of signals that they can see and affect must be kept in an interface. As we'll see later, components in the environment can easily attach to a named interface via the resource database settings we just completed.

Chapter 7: Creating an Agent

In this chapter, we're going to create the ALU agent, driver, monitor, and sequencer. We will learn about the purpose of each of these types of components, and we will discuss in detail how all that configuration *magic* actually works.

Creating Multiple Files

Again, `utg` makes it a snap to create all of these files at one time:

```
verif/vkits/alu> utg agent drv mon sqr -n alu -f -q
++ Creating alu_agent.sv
++ Creating alu_drv.sv
++ Creating alu_mon.sv
++ Creating alu_sqr.sv
```

You'll want to add these files as `` `includes `` to the package **[Rec 3.4.1.3-1]**. The order in which these files are included is inconsequential. In fact, you can do it in alphabetical order or in chronological order. What matters most is that each file's dependencies are `` `included `` **[Rule 3.4.2.1-1]**, that you use `` `include `` guards **[Rule 3.4.2.1-2]**, and that you manage any cyclical dependencies (which we'll see later). If you follow these rules of order, you should never have any build ordering issues.

Objects vs. Components

Let's review a few terms and what the purpose of each of these things is before going forward:

Objects

Everything is a `uvm_object`, because it is the root class of all other UVM-based classes, but when we specifically refer to something as an object, we usually mean it is something that is not also a component. A class solely derived from `uvm_object` is dynamic in nature, being created and destroyed on-the-fly, like a transaction or a packet.

Component

A component derives from `uvm_object`, but is quasi-static in nature. While the simulator considers them to be dynamically created (as opposed to, say, modules, wires, and registers), they are created by UVM at time zero and live throughout the lifetime of the simulation.

Phases

Components have phases. First, they run their `new()` function as soon as they are created, as all classes do. Later, they run the `build_phase`, the `connect_phase`, the `end_of_elaboration_phase`, and the `start_of_simulation_phase`. All of these phases happen in zero time at time zero. Then, the `run_phase` happens, which has numerous task-based sub-phases, which consume time. See more about phases in Appendix A.

Objects that are *not* components do *not* have phases.

Driver

The driver is a component that receives transactions from the sequencer and wiggles the wires of an interface to stimulate the DUT, following the protocol of the interface it attaches to. It also re-routes responses to transactions back to the sequencer. Ideally, its purpose should be kept to that, and to that only.

Monitor

The monitor is a component that watches the interface it is attached to and reports what happens, checking the interface for protocol violations. It collects activity on the interface in the form of transactions and usually pushes them out of an analysis port. It is agnostic in nature because it monitors both the driver and the DUT looking for errors and reporting activity.

Sequencer

A sequencer is a component that arbitrates among multiple streams of stimulus (sequences), sends sequence items to a driver one at a time, and re-routes responses back to the original sequence.

Agent

An agent is merely a container class that typically holds a sequencer, driver, and monitor. Its primary purpose is to provide a single configurable, re-usable component that you can plop down into your testbench, without having to deal with multiple components. An agent may hold other components as well, but serves no other specific purpose. It usually features TLM ports and exports (see Chapter 8) that send and receive transactions to other agents or components.

With those definitions in mind, let's take a look at what `utg` created for us. The file `verif/vkits/alu/alu_agent.sv` is already chock full of most of the stuff we need. It has one configuration field, `is_active`, which instantiates the driver and sequencer when set to `UVM_ACTIVE`. Otherwise, the agent is `UVM_PASSIVE`, in which case only the monitor is created and operational. This is typical for block-level agents that monitor interfaces in a full-chip testbench, but never actually drive stimulus. The build phase is where this all happens:

```
    virtual function void build_phase(uvm_phase phase);
       super.build_phase(phase);

       mon = mon_c::type_id::create("mon", this);
       if(is_active) begin
          drv = drv_c::type_id::create("drv", this);
          sqr = sqr_c::type_id::create("sqr", this);
       end
    endfunction : build_phase
```

A monitor might perform all of the checking for the agent on-the-fly, but agents *usually* do not contain scoreboards or more complex predictors, because these might inhibit reusability. A block-level agent might be reusable from one chip design to the next, but the prediction algorithm might change. Instead, agents push monitored traffic out of TLM ports and on to other listeners, be they scoreboards, functional coverage collectors, or other agents. We'll discuss prediction more in Chapter 11.

Sequencers communicate stimulus sequence items and responses to the driver via the `seq_item_port` of the driver and the `seq_item_export` of the sequencer. `utg` has already made this connection for you:

```
    virtual function void connect_phase(uvm_phase phase);
       super.connect_phase(phase);
       if(is_active)
          drv.seq_item_port.connect(sqr.seq_item_export);
    endfunction : connect_phase
```

We'll talk about these ports and exports in the next chapter, but you should be aware that the ports, the driver, and the sequencer, are all parameterized classes. They all need to know what kind of stimulus item (called the request) the sequencer will be sending to the driver, and what kind of response item the driver will send back to the sequencer. Sometimes, a driver won't have a data item to send back if for example the interface has one-way communication. In such a case, the driver will often just send back the request itself once it has completed sending it. Other times, the response will be embedded into the transaction. You'll recall that we added the result field to our transaction for just such a purpose.

So, the driver and sequencer need to be parameterized with our transaction item (the `item_c` you created in the last chapter), and a `result_t` type that represents the result. Since we ran `utg` above with `-q`, `utg` did not prompt us for the `reqType` and `rspType` interactively, so we'll have to fill those in ourselves.

In the case of the ALU agent, the request and response types are the same thing for both the driver and sequencer: `item_c`. The base class `uvm_driver` sets the default response type to be the same as the request type (like a default argument):

```
class uvm_driver #(
   type REQ = uvm_sequence_item,
   type RSP = REQ
) extends uvm_component;
```

So you only need to provide the request type and the response type is automatically filled in:

26. verif/vkits/alu/alu_drv.sv

```
class drv_c extends uvm_driver#(item_c);
```

And likewise in the sequencer:

27. verif/vkits/alu/alu_sqr.sv

```
class sqr_c extends uvm_sequencer#(item_c);
```

You also may have run into the fact that the `alu_item.sv` needs to be `` `included `` in both the sequencer and driver files. This requirement is because they now reference `item_c`, which is now a dependency.

Problem 7-1

Instantiate the ALU agent in the `alutb_pkg::env_c`. Compile and simulate to ensure you did it correctly.

Solution

Instantiating the ALU agent in the ALUTB environment first requires that you added the `alu.flist` file before the `alutb.flist` file in your `FLISTS` setting, because of the ALUTB's dependency on the ALU package.

Then instantiate an ALU agent in the Fields group:

28. verif/vkits/alutb/alutb_env.sv

```
// field: alu_agent
// The ALU agent
alu_pkg::agent_c alu_agent;
```

Then add the following to its `build_phase`:

29. verif/vkits/alutb/alutb_env.sv

```
// create the ALU agent
uvm_config_db#(int)::set(this, "alu_agent", "is_active", is_active);
alu_agent = alu_pkg::agent_c::type_id::create("alu_agent", this);
```

We have configured the ALU agent's `is_active` field to match the one in the ALUTB environment. It is now time to learn about how configurations from the resource database trickle down to children components.

All About Build Configurations

In the following example, it is important to distinguish between the parent-child relationship of objects within the component hierarchy (i.e., an agent is the parent of a driver) versus the parent-child relationship of class inheritance (i.e., `basic_test_c` derives from `base_test_c`). To help make this clear, we'll refer to *descendants* and *ancestors* to discuss class inheritance, versus *parents* and *children* when discussing component hierarchy.

When a UVM component calls `super.build_phase()`, it calls its ancestor's `build_phase()` function, and that ancestor calls its ancestor's `build_phase()` function, and so on, until the ancestor is the base class `uvm_component`. There, the `build_phase` function looks in the configuration database for all of the fields defined by the `` `uvm_field `` macros. It uses the component's full-name in the hierarchy and the field's name and type to match against items in the database. The field will eventually be set to the matching entry that was placed by the *highest* component in the hierarchy.

The example hierarchy shows the order in which the functions are executed. At Step 1, only the test instance has been created and newed, and all other components do not exist yet.

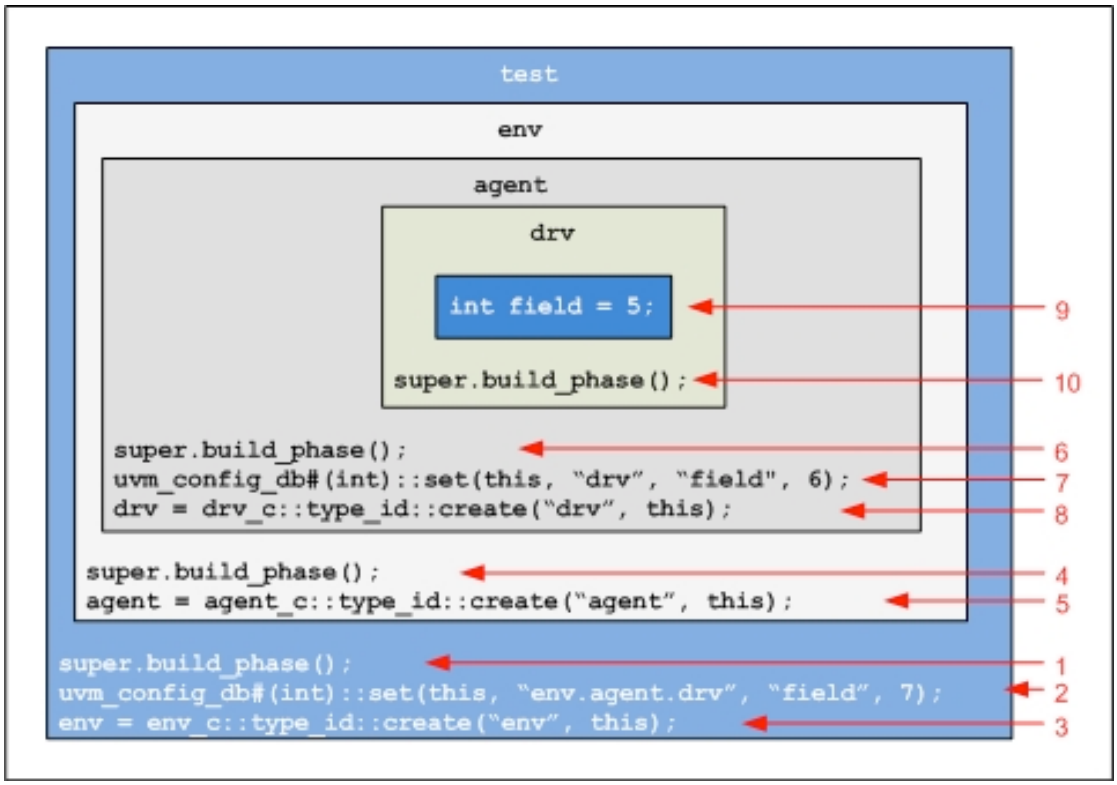


Figure 4: *Build Phases Explained*

1. When UVM enters the build phase, the test's `build_phase` function runs and it calls `super.build_phase()`.

2. The test then puts a value in a resource table that lives in the test that will later match against the `field` in the driver, and gives it a value of 7.

3. The test then creates the `env` instance, which gets newed. All of its fields are set to default values and its `new()` function runs. Its parent is a reference to the test. By creating the `env` component, UVM adds this instance to a list of components that still need to run their `build_phase`.

4. Later, UVM runs the `env`'s `build_phase` function and it also calls `super.build_phase()`.

5. The `env` creates the `agent` instance.

6. Later still, the agent's `build_phase()` function runs and it, too, calls `super.build_phase()`.

7. The agent puts a value of 6 in its resource table that will match the driver's field.

8. The driver is created.

9. Upon creation, the field is given the default value of 5, and the driver's `new()` function is then run.

10. Finally, the driver's build phase runs and it, too, calls `super.build_phase()`. Again, this is the `uvm_component::build_phase()` function, and this is where all the work happens. It calls the `apply_config_settings()` function, which is instructed by the `` `uvm_field_int(field, UVM_ALL_ON) `` macro to traverse upwards to find matching settings. The first one it finds will be the value of 6 supplied by its own parent, the agent. It then checks the agent's parent, `env`, to see if it has a matching entry. It doesn't, so it keeps going up to the test component, where it finds yet another matching entry which changes the value of the field to 7. Because the test has no parents, that will be the field's final value.

In addition to what was shown in this example is the power of using glob-style or regular expressions for either the hierarchy or the field name. Instead of specifying that the driver's field gets a value of 7, the test could have specified that any components within the agent (including the monitor and sequencer) which also have a field named 'field' will also get the value of 7.

This is accomplished instead with this call:

```
uvm_config_db#(int)::set(this, "env.agent.*", "field", 7);
```

Or, it might say that any integers in the env with a field name starting with "fi" will get a value of 7:

```
uvm_config_db#(int)::set(this, "env.*", "fi*", 7);
```

As you'll see in Chapter 12, this is how the base test distributes the `cfg` and `reg_block` instances to *all* components within the hierarchy that asks for them:

```
// push the register block and the configurations to all blocks that ask for it
uvm_config_db#(uvm_object)::set(this, "*", "reg_block", reg_block);
uvm_config_db#(uvm_object)::set(this, "*", "cfg",       cfg    );
```

Care must be exercised, though, if you start instantiating other people's components and they also happen to use field names that have names like `cfg` or `reg_block`, but do not correspond to your testbench's notion of what these objects represent. UVM offers some protection against this by also matching on a field's type, but accidents are still possible.

All About The Class Factory

In Chapter 3, you used the class factory to replace the standard clock driver instance with one of your own. The process is similar to the configuration example above. To replace components, you call one of the four available functions (shown below) to submit a class

override to the factory, and later when any sub-component tries to `create()` a class of the base type, the factory searches its database and replaces it with the descendant class you specified.

Component Factory Functions	Purpose
`set_type_override_by_type`	Replace all classes of the base type with the descendant type.
`set_inst_override_by_type`	Replace only classes whose name matches a wildcard pattern.
`set_type_override`	The same function as `set_type_override_by_type`, but use strings for the class names.
`set_inst_override`	The same function as `set_inst_override_by_type`, but use strings for the class names

Table 4: Factory Functions

To replace components, you would call these functions during the build phase of a parent component requesting the override. One important difference between the configuration example, however, is that you may need to call the override function *before* calling `super.build_phase()`.

For example, if you wanted to override the default clock driver from your base test, it stands to reason that **you must request the override before creating the clock driver**:

```
set_type_override_by_type(cn_pkg::clk_drv_c::get_type(), clk_duty_cycle_c::get_type(), .replace(1));
tb_clk_drv = cn_pkg::clk_drv_c::type_id::create("tb_clk_drv", this);
```

However, if you want to cause the override in a sub-class of base-test, such as `duty_cycle_test_c`, you must perform the override **before** `super.build_phase()`, as we did in Chapter 3. This is because calling `super.build_phase()` calls `base_test_c::build_phase()`, which will create the clock driver.

```
set_type_override_by_type(cn_pkg::clk_drv_c::get_type(), clk_duty_cycle_c::get_type(), .replace(1));
super.build_phase(phase);  // <-- calls base_test_c::build_phase and creates the clock driver.
```

If you called it after `super.build_phase()`, the clock driver would have already been created and your factory override request would be too late.

Factory Overrides on Non-Components

Class factory overrides are not limited to just components, though. They do not rely on phases to work properly. You can create your class overrides on transactions, sequences, or any other UVM object. You can even do them after time zero. For example, you could enable sending corrupted CTX transactions after the configuration phase.

So long as the factory override request is made before the create call, the override will hold.

How To See Everything

At some point in your UVM career, you might want to do something with every single component. Or, you might want to print out all of the drivers named "Fred". Whatever your strange inclinations, UVM offers a way to see everything in the component hierarchy.

`uvm_top` is the globally visible instance of `uvm_root`. Unfortunately, UVM does not adhere to our clever `_c` naming conventions (let them wallow in their ignorance!). But, because `uvm_top` is globally visible, you can refer to it from anywhere in the codebase. This is awfully nice, because it offers some nifty functions for perusing the hierarchy:

The `find` method will return a handle to the `uvm_component` with the matching hierarchical name:

```
function uvm_component find(string comp_match);
```

The `find_all` method populates a list of components that match your query, and allows for `.` and `?` wildcards.

```
function void find_all(string comp_match,
                       ref uvm_component comps[$],
                       input uvm_component comp = null);
```

Finally, the function `print_topology` can be used to print out the entire hierarchy to the logfile, starting from `uvm_top`. Printing the topology is a handy thing to do at the start of simulation, after everything has been created. Because it is such a good idea, the `global_pkg::env` will do it for you, so long as your debug level is non-zero. You can also use the `cnmake` command-line option (`TOPO`) to specify the depth of the tree to print, with the default being 4.

```
verif/alutb> cnmake sim DBG=10 TOPO=8
```

Conclusion

In this chapter, we learned all about components, their quasi-static nature, the component hierarchy, and how configurations trickle down from the top of the hierarchy. We also learned what goes into an agent and the purpose of all the different types of components. Finally, we discussed the class factory in more detail and a means to access or print all components.

Next, it's time to learn how components *talk* to one another, using Transaction Level Modeling.

Chapter 8: TLM Ports, Imps, and Exports

TLM 1.0 is the UVM equivalent of interface classes that one might use in C++ or Java. Compared to interface classes, TLM is a more formalized method of passing pointers to classes to call their functions. TLM ports simulate having hierarchical connections between components.

Some of TLM's important features are:

- A TLM `port` is always the *initiator* of the function call.
- A TLM `imp` (implementation) is always the *target* of the function call.
- A TLM `export` provides pass-through functionality, to traverse different levels of hierarchy.
- TLM provides one-to-one, many-to-one, or one-to-many calls.
- TLM initiators can perform a get or a put.
- Calls can be blocking (time-consuming tasks) or non-blocking (zero-time functions).
- Analysis ports provide a one-to-many functionality.

TLM Ports

A TLM `port` is instantiated as a parameterized class. Like all classes, they must be new'ed. It is unlikely that you'll ever override these classes with the factory, but if you prefer the safety of calling `::create`, that's ok, too. To put a transaction into the `port`, you call the port's `put()` function.

```
class producer_c extends uvm_component;
   `uvm_component_utils(producer_c)

   uvm_blocking_put_port #(item_c) put_port;

   function new(string name, uvm_component parent);
      super.new(name, this);
   endfunction : new

   virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      put_port = new("put_port", this);
   endfunction : build_phase

   task run();
      forever begin
         item_c item = item_c::type_id::create("my_item", this);
         (#50ns) put_port.put(item);
      end
   endtask : run
endclass : producer_c
```

UVM uses squares in component diagrams to represent ports:

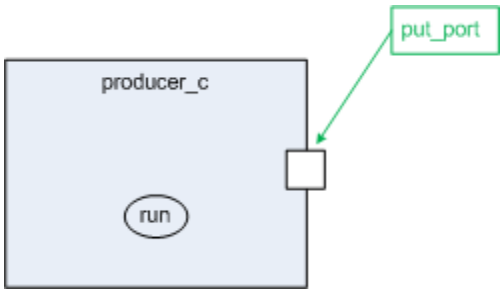


Figure 5: *Ports*

TLM Implementations (Imps)

The `imp` is the target of the `put`[3]. You must instantiate the `imp` just as you did the `port` *and* you must write the implementation function:

```
class consumer_c extends ovm_component;
   `uvm_component_utils(consumer_c)

   uvm_blocking_put_imp #(item_c, consumer_c) put_imp;

   function new(string name, uvm_component parent);
      super.new(name, this);
   endfunction : new

   virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      put_imp = new("put_imp", this);
   endfunction : build_phase

   task put(item_c _item);
      // do something with _item
   endtask : put
endclass : consumer_c
```

The `imp` class is parameterized not only with the item type but also with the class's type, so that the `imp` knows which class contains the target implementation function.

`Imps` are represented with circles:

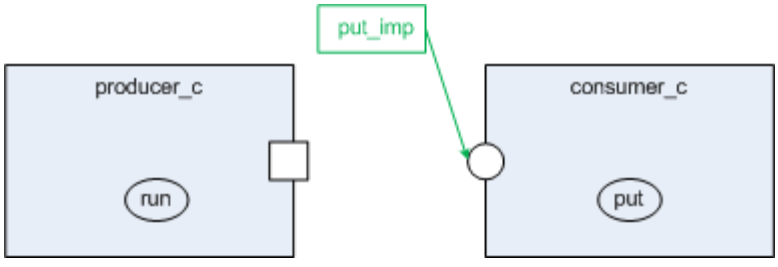


Figure 6: *Imps*

[3] More accurately, an imp is the specific functionality that the user must define for a port.

Connecting

`Ports` are connected to `imps` at a higher level of the hierarchy during the `connect_phase`. The reason this happens during the connect phase instead of during the build phase is because the two components you're trying to connect may not have yet run their build phases. And if that's the case, then they may not have created their `ports` or `imps` yet.

The whole reason for having these separate phases at time zero is because all components in the system are *guaranteed* to have run all prior phases.

```
class env_c extends uvm_env;
   `uvm_component_utils(env_c)
   producer_c producer;
   consumer_c consumer;

   virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      producer = producer_c::type_id::create("producer", this);
      consumer = producer_c::type_id::create("consumer", this);
   endfunction : build_phase

   virtual function void connect_phase(uvm_phase phase);
      super.connect_phase(phase);
      producer.put_port.connect(consumer.put_imp);
   endfunction : connect_phase

endclass : env_c
```

Connect calls are always called on the initiator's `port`, with the receiving `imp`, `port`, or `export` as the argument. This completes the diagram and allows the producer to call another class's `put` function with a single argument, `item_c`.

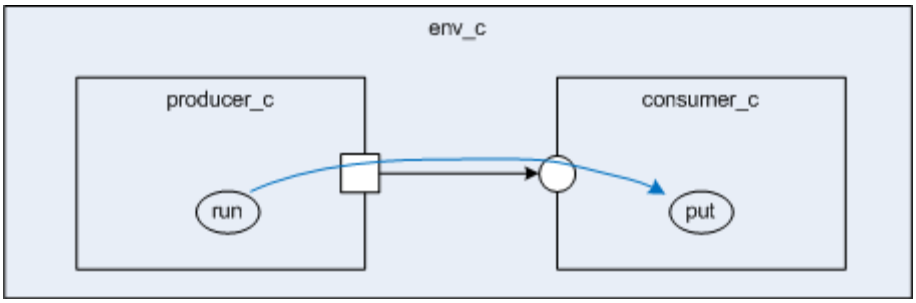


Figure 7: *Connecting Ports to Imps*

TLM Exports

TLM `exports` promote an implementation to a higher level in the hierarchy. From another component's point of view, they look exactly like an `imp`, but the real `imp` is buried someplace within the hierarchy. With `exports`, the external component need not know anything about the lower-level hierarchy.

In the Figure 8 below, `comp2` and `subcomp2` both have `exports` that promote the `imp` which is buried within `leaf2`. The `env` which makes the connection between `comp1` and `comp2` need not know the inner workings of either component.

TLM Summary

To summarize:

- TLM has unidirectional interfaces of many forms.
- Blocking ports allow time-consuming tasks to be called.
- Non-blocking ports make zero-time function calls.
- The flow of information may be *from* the port (`put`), or *to* the port (`get`).
- The `port` is *always* the initiator, and the `imp` is *always* the target.
- Calls that peek do not consume the next available data.

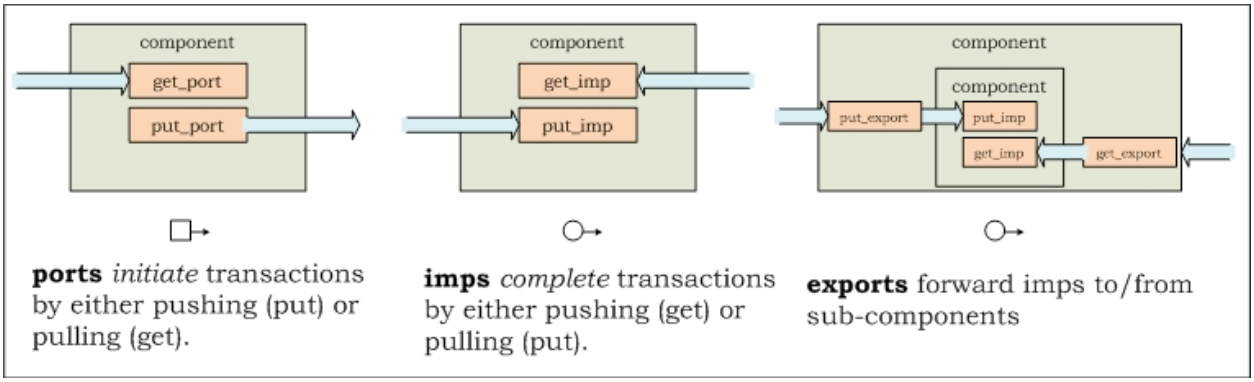


Figure 8: *TLM Cheat-Sheet*

Imp Declarations

You may have noticed something flawed about the consumer class example above. The presence of the `uvm_blocking_put_imp` requires that a task called `put`, accepting an argument of type `item_c`, be present within the class. Astute observers would note that this precludes the class from having any other put `imps`. Surely UVM doesn't think we could live on one `imp` alone!

Fortunately, they don't, and there is a way around this. As usual, UVM chooses macros to solve the problem:

```
`uvm_put_imp_decl(_monitored)
`uvm_put_imp_decl(_driven)

class consumer_c extends uvm_component;
   uvm_put_imp_monitored#(item_c) monitored_imp;
   uvm_put_imp_driven#(item_c) driven_imp;

   function void put_monitored (item_c t);
      // receives puts coming into monitored_imp
   endfunction

   function void put_driven(item_c t);
      // receives puts coming into driven_imp
   endfunction
endclass : consumer_c
```

The `` `uvm_put_imp_decl(_something) `` macro creates a new class called `uvm_put_imp_something` that allows you to create a function called `put_something()`. This class gives you the freedom to create all the `imps` you need. Preceding the argument with an underscore (_) allows you to avoid having to create the poorly-named function `putsomething()`.

The `` `uvm_*_imp_decl `` macros actually do create a new class for you. You're probably tempted to use the macro just above your class definition, as in this example. However, there may be more than one class in your package that needs to implement a `put`, `get`, or `write` for your given transaction. Using the macro twice, though, just won't do, because the compiler gets understandably cranky when a class is defined more than once. Coming up with a new name each time may not be suitable, either.

Rather than use the macros in only one class file and depend on the build order working out for you, it is advisable to instead place these declarations in the package file.

Many-to-One and One-to-Many

`Ports` and `imps` implicitly support many-to-one. That's because multiple ports can be connected to a single `imp`.

When you hear `analysis_port`, think *broadcast*. Analysis ports permit one-to-many communication and in diagrams are represented with a diamond. These are often the `ports` and `imps` of choice for broadcasting information to an indeterminate number of listeners.

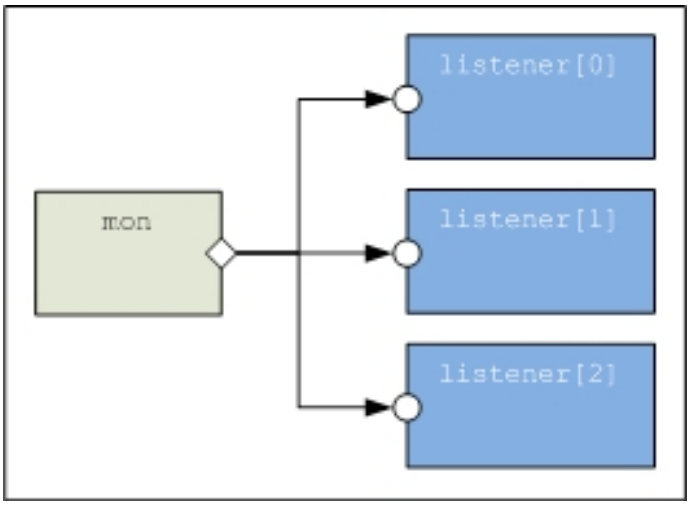


Figure 9: *Analysis Ports and Imps*

We haven't even set up our monitor yet, but we will want for it to broadcast all monitored transactions and result values that it sees to the rest of the world. So, the `mon_c` class will need an analysis `port`, and the agent should also get one, too. This allows listeners to connect directly to the agent's analysis `port` without having to dig into the implementation to find out what component broadcasts stuff and what they are called. Providing a clean and consistent API from the agent helps to promote reuse.

Problem 8-1

Create separate `analysis_ports` in the monitor and the agent to broadcast transactions and results to the rest of the world. Connect them together. You don't have any code to test them yet, but make sure you can compile it.

Solution

In the monitor, first declare the ports:

30. verif/vkits/alu/alu_mon.sv

```
//------------------------------------------------------------------------------
// Group: TLM Ports

// field: monitored_item_port
// All monitored transactions go out here
uvm_analysis_port #(item_c) monitored_item_port;

// field: monitored_result_port
// All monitored results go out here
uvm_analysis_port #(result_t) monitored_result_port;
```

Bonus points if you placed these in the TLM Ports group and used a name that ended in `_port`, to comply with the coding standards **[Rec. 2.6-4]**.

Then, in the `build_phase`:

31. verif/vkits/alu/alu_mon.sv

```
monitored_item_port = new("monitored_item_port", this);
monitored_result_port = new("monitored_result_port", this);
```

You can put the exact same lines in the agent. You also would need to remember to `` `include `` the `alu_item.sv` file as it is now a dependency of these two files (just like with the driver and sequencer earlier).

Finally, in the `connect_phase` of the agent, you make the connection unconditionally (not dependent on `is_active`), because the monitor will be present in all types of testbenches:

32. verif/vkits/alu/alu_agent.sv

```
// connect to the monitor's analysis ports
mon.monitored_item_port.connect(monitored_item_port);
mon.monitored_result_port.connect(monitored_result_port);
```

Broadcasting to a Listener

Now let's write something out of the analysis `port` and create something to listen for it. **Take your test code from `verif/alu/tests/basic.sv` and move it to the `main_phase` of the ALU monitor.** After creating a transaction, call the port's `write()` function with the transaction as the argument. This will send it out the analysis `port`.

33. verif/vkits/alu/alu_mon.sv

```
virtual task main_phase(uvm_phase phase);
   byte unsigned stream[];
   alu_pkg::item_c item;
   alu_pkg::item_c unp_item;

   phase.raise_objection(this);

   repeat(50) begin
      item = alu_pkg::item_c::type_id::create("item");
```

```
            item.randomize();
            monitored_item_port.write(item);
            `cn_info(("Created ALU transaction: %s", item.convert2string()))
            item.pack_bytes(stream);
            `cn_info(("Bytes: %s", cn_pkg::print_ubyte_array(stream)))
            unp_item = alu_pkg::item_c::type_id::create("unp_item");;
            unp_item.unpack_bytes(stream);
            `cn_info(("Unpacked: %s", unp_item.convert2string()))
            if(item.compare(unp_item) == 0)
               `cn_err(("Miscompare!"))
         end

         phase.drop_objection(this);
      endtask : main_phase
```

In the ALUTB environment, where the ALU agent is instantiated, create a `uvm_subscriber` using `utg`, naming it `alu_item_subscriber`.

uvm_subscriber

A subscriber is a specialized UVM parameterizable component that comes pre-packaged with an `analysis_imp` that can accept whatever object for which it is parameterized. All you need to do is fill in the `write()` function.

Here is how you create the subscriber component with `utg`:

```
34.
verif/vkits/alutb> utg subscriber -n alu_item_subscriber -f
++ Creating alutb_alu_item_subscriber.sv
alu_item_subscriber subscriber: Enter substitution for <description>: Listens for all monitored ALU
transactions.
alu_item_subscriber subscriber: Enter substitution for <subscription_type>: alu_pkg::item_c
```

Take special note that we used the scope operator to specify that the transaction is coming from the `alu_pkg`, because this component will be a member of the `alutb_pkg`.

Problem 8-2

Modify the `alu_item_subscriber` to print out the transactions that are written to it. Then, instantiate the component in the ALUTB environment and connect it to the ALU agent's transaction analysis `port`. Simulate and see that your subscriber is seeing all of the transactions. The subscriber uses an `export` called `analysis_export` to which you will connect.

Solution

Monitors don't often generate transactions, but this is just a temporary self-test. After removing some boilerplate code, the whole `alu_item_subscriber_c` class would look like this:

35. verif/vkits/alutb/alutb_alu_item_subscriber.sv

```
// class: alu_item_subscriber_c
// Print out all ALU transactions.
class alu_item_subscriber_c extends uvm_subscriber#(alu_pkg::item_c);
   `uvm_component_utils(alutb_pkg::alu_item_subscriber_c)

   //------------------------------------------------------------------------------
   // Group: Methods
   function new(string name="alu_item_subscriber",
                uvm_component parent=null);
      super.new(name, parent);
   endfunction : new

    ////////////////////////////////////////
   // func: write
   // Receives the alu_pkg::item_c
   virtual function void write(alu_pkg::item_c t);
      `cn_info(("Received this ALU transaction: %s", t.convert2string()))
   endfunction : write
endclass : alu_item_subscriber_c
```

And the ALUTB environment would also need to `include the `alutb_alu_item_subscriber.sv` file, instantiate and create the class as before, and then connect it during the `connect_phase` like this:

36. verif/vkits/alutb/alutb_env.sv

```
      alu_agent.monitored_item_port.connect(alu_item_subscriber.analysis_export);
```

If everything was done correctly, you should see your subscriber happily printing all the transactions it sees.

Leave it in there for now but **remove the generated transactions from the monitor**, since up next we're going to write the real stimulus sequences.

Conclusion

In this chapter we learned all about how different UVM components *talk* to one another with ports, `imps`, and `exports`. We learned how to instantiate these TLM objects and how to connect them up. We also created a subscriber that can act as a benign listener.

There are two other interesting aspects of TLM that were not covered here:

- TLM can be used to bridge across different languages. Thus, an analysis `port` can send out all monitored transactions to a UVM subscriber or to a SystemC implementation...all without having to change the UVM source code.
- TLM2 is new to UVM, but not new to the rest of the world. TLM2 offers ports that are bidirectional—called sockets. It also provides a way to generically model memory transactions using the `uvm_tlm_generic_payload` class. While the rest of UVM does not use TLM2 (yet!), later revisions of UVM will likely be driving towards it.

Chapter 9: Beginning Sequences

Sequences are intended to drive the stimulus items to the driver. They are dynamic objects that are created and destroyed (as opposed to components, which are quasi-static). They can act as generators to create new stimulus items. They can create other sequences and form a protocol stack. They can last only a short amount of time to perform a few operations, or they can be procedural blocks like an interrupt handler that last the whole life of the simulation.

The number and variety of use cases for sequences is fairly large, and their complexity may at first seem daunting to new users. This chapter is only the first on sequences, to keep the topic more digestible. In Chapters 14 and 16, you will learn more details about sequences and sequencers.

A Simple Generator

Let's create a simple sequence to generate some random transactions. **Be sure to first remove the generator code we placed in the monitor's run phase.**

Most sequences are short pieces of code, and as we will see later in Chapter 14 you can create a special library sequence that mixes and matches all (or some) of the sequences you write. For this purpose, it is often handy to put all of these sequences in one file called a sequence library file. Other sequences that you write may be larger and more specialized and may live in their own file.

Create a sequence library with `utg`. Sequences expect to eventually propagate sequence items (like our `item_c` class) as requests to a driver, and to optionally receive sequence items as responses. Since we unified our result data into the transaction class, it is ok if they are one and the same:

```
verif/vkits/alu> utg seq_lib -n alu -f
++ Creating alu_seq_lib.sv
alu seq_lib: Enter substitution for <description>:
alu seq_lib: Enter substitution for <reqType>: item_c
alu seq_lib: Enter substitution for <rspType>: item_c
```

Your sequence library now contains two pre-made classes for you. The first is a generic sequence called `alu_seq_c`, and the second is a sequence "library" called `lib_seq_c`. This special library sequence is an exerciser that randomly picks and drives sequences from those that are registered to it. **We will not need that now so comment it out.** We will discuss the library sequence further in Chapter 14.

A sequence is a class with a special time-consuming task called a `body()`. Technically, it has a `pre_body()` task as well as a `post_body()` task, but right now we'll just use the `body()` task to drive in a transaction:

37. verif/vkits/alu/alu_seq_lib.sv

```
////////////////////////////////////////////
// func: body
virtual task body();
   item_c item = item_c::type_id::create("item");
   start_item(item);
   item.randomize();
   finish_item(item);
endtask : body
```

Step-by-step, this is what this task does:

1. Create a new transaction.

2. Call `start_item`. This does a few things, but primarily waits until the sequencer is ready to send it.

3. Now randomize it. By randomizing after we've waited for the sequencer to accept it, we've achieved *late binding*. This allows us to add additional constraints, for example, based on the current conditions.

4. Call `finish_item`. This pushes it to the sequencer upon which this sequence is operating.

To see that the sequence is being driven to the driver, change the run phase of your ALU driver to get the next item, print it out, and then tell the sequencer that it is done:

38. verif/vkits/alu/alu_drv.sv

```
////////////////////////////////////////////////
// func: run_phase
virtual task run_phase(uvm_phase phase);
   // constantly poll for new transactions, printing them out
   forever begin
      seq_item_port.get_next_item(req);
      `cn_info(("Driving: %s", req.convert2string()))
      seq_item_port.item_done(req);
   end
endtask : run_phase
```

Now only one last thing remains: You have to create the sequence and *start* it on the sequencer. There are multiple ways of doing this, but putting it in the basic test's `main_phase` is the most straightforward for now:

39. verif/alutb/tests/basic.sv

```
////////////////////////////////////////////////
// func: main_phase
virtual task main_phase(uvm_phase phase);
   alu_pkg::alu_seq_c alu_seq = alu_pkg::alu_seq_c::type_id::create("basic_seq");
   alu_seq.randomize();

   phase.raise_objection(this);
   `cn_info(("Starting alu_seq."))
```

```
      alu_seq.start(alutb_env.alu_agent.sqr);
      phase.drop_objection(this);
    endtask : main_phase
```

Be sure to **delete the original code that had previously been in the `main_phase`**.

Running the basic test should produce the desired results. When a sequence's `start` task is called, it will bind itself to the sequencer specified in its argument, and then call its `pre_start`, `pre_body`, `body`, `post_body`, and finally its `post_start` tasks.

At this point, you're probably a bit underwhelmed. After all, that's an awful lot of code to write just to send one measly transaction, but things will get more interesting soon.

`uvm_do Macros

For one thing, we can clean up our sequence a little bit. UVM provides a set of macros to create, randomize, and perform a sequence or sequence item from another sequence. These are the `uvm_do` macros, and it is *very important* to know what each of them does before using them. But first, let's shorten our `alu_seq_c` body a bit:

40. verif/vkits/alu/alu_seq_lib.sv

```
    virtual task body();
      item_c item;
      `uvm_do(item)
    endtask : body
```

Much shorter. `uvm_do` does pretty much everything we had before, including randomizing the sequence. Sometimes, you want to randomize your sequence item with constraints. In this case:

41. verif/vkits/alu/alu_seq_lib.sv

```
      `uvm_do_with(item, { operation == MUL_A_B; alpha == 7; beta == 5; })
```

`uvm_do_with` takes a constraint block and applies it during the randomization call. Try running this code to ensure you get the correct results.

Other times, you want to simply create the transaction, set it precisely, and then send it:

42. verif/vkits/alu/alu_seq_lib.sv

```
      `uvm_create(item)
      item.operation = item_c::MUL_A_B;
      item.alpha = 7;
      item.beta = 5;
      `uvm_send(item)
```

This is less concise than the `uvm_do_with` call, but it is more efficient because the transaction doesn't go through the randomization process.

UVM has other macro facilities to send things with a higher priority, to send them on a different sequencer (for sequences attached to virtual sequencers, to be discussed later),

for randomizing and sending, and for combinations of all the above. **These macros can only be called from a sequence, not from within the component hierarchy.**

Learn what macros are available from the table below and use the right one.

Macro	Purpose
`uvm_create	Calls ::create on the given sequence or sequence item, and assigns it to this sequence's sequencer.
`uvm_do	Creates, starts, randomizes, and finishes the given sequence, on this sequence's sequencer.
`uvm_do_pri	The same as `uvm_do, but assigns a priority.
`uvm_do_with	The same as `uvm_do, but randomizes with the given constraint block.
`uvm_do_pri_with	The same as `uvm_do, but assigns a priority and randomizes with the given constraint block.
`uvm_create_on	Calls ::create on the given sequence, assigning it to this given sequencer.
`uvm_do_on	Creates, starts, randomizes, and finishes the given sequence, on the given sequencer.
`uvm_do_on_pri	The same as `uvm_do_on, but assigns a priority.
`uvm_do_on_with	The same as `uvm_do_on, but randomizes with the given constraint block.
`uvm_do_on_pri_with	The same as `uvm_do_on, but assigns a priority and randomizes with the given constraint block.
`uvm_send	Calls start and finish on a previously created sequence, without randomization.
`uvm_send_pri	Calls start and finish on a previously created sequence, without randomization, but with the specified priority.
`uvm_rand_send	Calls start, randomize, and finish on a previously created sequence.
`uvm_rand_send_pri	The same as `uvm_rand_send, but with the specified priority.
`uvm_rand_send_with	The same as `uvm_rand_send, but randomizes with the given constraint block.
`uvm_rand_send_pri_with	The same as `uvm_rand_send_with, but with the specified priority.

Table 5: *`uvm_do Family of Macros*

Getting a Response

Our driver doesn't do much with it yet, but it will soon enough. For the sake of our future examples, though, let's talk about how the driver would send back a response.

The response that the driver sends back to the sequencer will end up in the sequence that called it when the sequence calls its `get_response` function:

43. verif/vkits/alu/alu_seq_lib.sv

```
virtual task body();
   item_c item;
   `uvm_create(item)
   item.operation = item_c::MUL_A_B;
   item.alpha = 7;
   item.beta = 5;
   `uvm_send(item);
   get_response(rsp);
   `cn_info(("Got response result: 0x%08X", rsp.result))
endtask : body
```

What is `rsp`? Well, a sequence has two pre-defined variables in its base class—`req` and `rsp`—which are of the request and response types, respectively. In our case, they are both transactions of type `item_c`. Let's modify the ALU driver's `run_phase` to send back a real result whenever it sees a multiply operation:

44. verif/vkits/alu/alu_drv.sv

```
forever begin
   seq_item_port.get_next_item(req);
   `cn_info(("Driving: %s", req.convert2string()))
   if(req.operation == item_c::MUL_A_B)
      req.result = result_t'(req.alpha * req.beta);
   seq_item_port.item_done(req);
end
```

Again, in our ALU example, our response is just the request with the result field filled in. This is a common application, but other cases may be different. A response might be a different packet or transaction. But it must be of type `uvm_sequence_item`, and there's no sense in creating a new one just to hold a 32-bit unsigned integer.

With any luck, you should see your sequence returns the number 0x23:

```
%I-(alu_seq_lib.sv: 51) [alutb_env.alu_agent.sqr.basic_seq] { 78ns} Got response result: 0x00000023
```

Problem 9-1

Spice things up a bit by adding a random count variable to your sequence. Constrain it to between 1 and 100, and perform that many random transactions, getting each one's response. Also, rename your sequence to `exer_seq_c`, since `alu_seq_c` isn't very descriptive. In addition to renaming it in `basic_test_c`, you'll also need to randomize it before calling its `start` function.

What happens if you don't call `get_response()`?

Solution

45. verif/vkits/alu/alu_seq_lib.sv

```
// class: exer_seq_c
// Runs <count> transactions
class exer_seq_c extends uvm_sequence #(item_c, item_c);
   `uvm_object_utils_begin(alu_pkg::exer_seq_c)
      `uvm_field_int(count, UVM_ALL_ON | UVM_DEC)
   `uvm_object_utils_end

   //------------------------------------------------------------------------------
   // Group: Fields

   // field: count
   // The number of random transactions to perform
   rand int count;
   constraint reasonable_cnstr { count inside {[1:100]}; }

   //------------------------------------------------------------------------------
   // Group: Methods

   function new(string name="alu_seq");
      super.new(name);
   endfunction : new

   ////////////////////////////////////////////
   // func: body
   virtual task body();
      item_c item;

      repeat(count) begin
         `uvm_do(item)
         get_response(rsp);
         `cn_info(("Got response result: %08X", rsp.result))
      end
   endtask
endclass : exer_seq_c
```

If you do not fetch the responses, the sequence fills up with responses and starts complaining about a response queue overflow. You can manage the depth of your sequence's response queue by calling `set_response_queue_depth` and `get_response_queue_depth`, or you can turn off the error reporting with `set_response_queue_error_report_disabled`. Most often, though, it is prudent to just get the responses.

Response Queue Handler

Sequences can also launch multiple requests at a time:

```
fork
   `uvm_do(item1)
   `uvm_do(item2)
join
```

In this case, the response for `item2` could come *before* the response for `item1`. Or, depending on your architecture, responses can come out-of-order by their very nature. As an alternative to calling `get_response(rsp)` for each request, you can set up a response handler.

To do so, you create the function `response_handler` that will be called whenever a response comes in. Because this is a virtual function from the base sequence class, which doesn't know what type of response to expect, the function will receive a generic `uvm_sequence_item`, which you can then `$cast` to your own response type. To enable this function to be called, you call `use_response_handler` with a value of 1. You can enable or disable this setting on-the-fly, although this may not be a good idea.

For example:

```
class my_seq_c extends uvm_sequence#(item_c);
   `uvm_object_utils(my_seq_c)

   //------------------------------------------------------------------------------
   // Group: Fields

   // field: outstanding_requests
   // An assoc. array of all pending requests
   item_c outstanding_requests[int];

   //------------------------------------------------------------------------------
   // Group: Methods
   function new(string name="my_seq");
      super.new(name);
      use_response_handler(1);
   endfunction : new

   ////////////////////////////////////////
   // func: response_handler
   // Matches ID of each oustanding request
   virtual function void response_handler(uvm_sequence_item response);
      item_c resp, request;

      // cast the response to our transaction type
      $cast(resp, response);

      // look it up and compare
      request = outstanding_requests[resp.id];
      if(!request || !request.compare(resp))
         `cn_err(("Response miscompare!"))
      else
         outstanding_requests.delete(resp.id];
   endfunction : response_handler

   ////////////////////////////////////////
   // func: body
   // Launch 5 transactions
   task body();
      item_c item[5] = new[5];
      fork
         foreach(item[x])
            fork
               automatic item_c this_item = item[x];
               `uvm_do(this_item)
               outstanding_requests[this_item.get_sequence_id()] = this_item;
            join_none
      join
   endtask : body
endclass : my_seq_c
```

There are some interesting constructs above that you may not have seen before:

```
    item_c outstanding_requests[int];
```

This declares an associative array (or hash, map, or dictionary, if you prefer) of transactions, keyed by an integer.

```
    use_response_handler(1);
```

This tells the sequence that its `response_handler` function should be called for each response that is received by this sequence.

```
    // look it up and compare
    request = outstanding_requests[resp.id];
    if(!request || !request.compare(resp))
       `cn_err(("Response miscompare!"))
    else
       outstanding_requests.delete(resp.id);
```

Sequence items come built-in with a unique `id` field. The first line above gets the request from the associative array, using the `id`. The second line uses the short-circuit condition of the OR operator. It first checks to see if it was found and if not it prints an error. Otherwise, it checks to see if it mis-compares. The last line deletes the outstanding request from the associative array.

```
    fork
       foreach(item[x]) begin
          fork
             automatic item_c this_item = item[x];
             `uvm_do(this_item)
             outstanding_requests[this_item.get_sequence_id()] = this_item;
          join_none
       end
    join
```

This seemingly complicated bit of code sends 5 transactions in parallel, pushing each one to the `outstanding_requests` hash by its sequence id, which is a unique value that UVM sequencers assign to sequences to assist in their routing. The nested fork-join construct is explained in more detail in Appendix B.

Complex Routines

At this point, sequences must seem pretty boring, but sequences get interesting when they do more than just one transaction. Our little ALU isn't capable of too many exotic functions, but there's enough there that we can do something more complicated.

Factorials

Let's create a sequence that performs the operation:

$$f(x) = x! = 1 * 2 * 3 * \cdots * x$$

The first thing to do is create a new sequence. With `-c`, `utg` will just give you the sequence class without the file header and you can cut-and-paste it into your `alu_seq_lib.sv` file:

```
verif/vkits/alu> utg seq -n factorial -c
factorial seq: Enter substitution for <reqType>: item_c
factorial seq: Enter substitution for <rspType>: item_c
class factorial_seq_c extends uvm_sequence #(item_c, item_c);
   `uvm_object_utils_begin(alu_pkg::factorial_seq_c)
   `uvm_object_utils_end
```

Our factorial sequence should have one random field, `operand`, which is the number for which we want to compute the factorial. Then, we'll loop from 1 to `operand`, collecting the result response each time, and present the final answer. We should also squirrel away the answer in the class as a member field, so that the basic test can then print the final result.

Here is the final sequence:

46. verif/vkits/alu/alu_seq_lib.sv

```
class factorial_seq_c extends uvm_sequence #(item_c, item_c);
   `uvm_object_utils_begin(alu_pkg::factorial_seq_c)
      `uvm_field_int(operand, UVM_ALL_ON)
      `uvm_field_int(answer,  UVM_ALL_ON)
   `uvm_object_utils_end

   //------------------------------------------------------------------------------
   // Group: Fields

   // field: operand
   // The value to perform the factorial on
   rand bit [15:0] operand;
   constraint operand_cnstr { operand <= 9; }

   // field: answer
   // The final result
   result_t answer = 1;

   //------------------------------------------------------------------------------
   // Group: Methods
   function new(string name="factorial_seq");
      super.new(name);
   endfunction : new

   ////////////////////////////////////////
   // func: body
   // Loop from 1..operand and multiply all the numbers together
   virtual task body();
      item_c item;
      byte num;

      for(num = 1; num <= operand; num++) begin
         `uvm_do_with(item, { operation == MUL_A_B; alpha == num; beta == answer[15:0]; })
         get_response(rsp);
         answer = rsp.result;
         `cn_info(("num=%0d, answer=%0d", num, answer))
      end
      `cn_info(("%0d! = %0d", operand, answer))
   endtask : body
endclass : factorial_seq_c
```

If you're a math whiz, you've already figured out that our ALU has a bit of a limitation. Since it can only take 16-bit operands, our largest factorial operand is a meager 9, and the result we keep must be constrained to those 16 bits. Nonetheless, it is a good example of how sequences can be used to convert multiple transactions into one larger transaction.

Problem 9-2

Modify your `basic_test_c` to create and send in a factorial sequence with an operand of 9, and print the result.

Instead of the variable name `answer` in `factorial_seq_c`, change the name of the variable to `result` instead. What happens? How do you solve this problem?

Solution

The first part of this should have been straightforward:

47. verif/alutb/tests/basic.sv

```
////////////////////////////////////////////
// func: main_phase
virtual task main_phase(uvm_phase phase);
   alu_pkg::factorial_seq_c factorial_seq = new("factorial_seq");
   factorial_seq.operand = 9;
   phase.raise_objection(this);
   factorial_seq.start(alutb_env.alu_agent.sqr);
   `cn_info(("The factorial of 9 is %d", factorial_seq.answer))
   phase.drop_objection(this);
endtask : main_phase
```

The problem with naming the answer `result` instead is that it will conflict with the local name `result` in the transaction. When the constraint is applied, it is local to the `item_c` class, which already has a `result` field. Thus, the `beta == result;` constraint refers to the transaction's version of `result`. Worse, because we left `result` uninitialized, your simulator may complain that there are X's or Z's in a constraint value.

You might be tempted to use `beta == this.result;` to distinguish the two, but since the constraint block is in the `item_c` scope, `this` also refers to the transaction and not the sequence.

Fortunately, the architects of SystemVerilog anticipated this problem and created the **`local::` scope operator**. This does exactly what you want:

48. verif/vkits/alu/alu_seq_lib.sv

```
      `uvm_do_with(item, { operation == MUL_A_B; alpha == num; beta == local::result[15:0]; })
```

Problem 9-3

Develop the sequence `sum_array_seq_c` that creates a random array of words and add them up. To create a randomly sized array, it is always important to constrain the array's size to a reasonable value, otherwise it will probably randomize itself to an extremely large number and crash the simulator. Also, let's keep each element in the array small (less than or equal to 100) so that we do not have to deal with overflows.

49. verif/vkits/alu/alu_seq_lib.sv

```
// field: data
// An array of words to be summed
rand bit [15:0] data[];

// keep it to a reasonable size
constraint data_cnstr {
   data.size() inside {[1:50]};
   foreach(data[x]) {
      data[x] inside {[0:100]};
   }
}
```

Now when we randomize our sequence, it will create a randomly sized array of random 16-bit words. To declare our data array as a UVM configuration field, we use the following macro:

50. verif/vkits/alu/alu_seq_lib.sv

```
`uvm_field_array_int(data, UVM_ALL_ON | UVM_DEC)
```

This line declares an array of *numbers*. UVM's field macros just lump all numbers as ints, they do not distinguish between bytes, ints, or 42-bit values. Declaring this field allows you to pack, unpack, and print it correctly (if you wanted to).

Also, add the result as a field of the sequence:

51. verif/vkits/alu/alu_seq_lib.sv

```
// field: result
// The final answer
result_t result;
```

Next, write the `body()` task of the `sum_array_seq_c` sequence to send transactions into the ALU that will sum the `data` array.

Finally, modify the driver to accomodate any new instructions that you will be sending, and modify the basic test's `run_phase` to randomize and send your new sequence.

Solution

The solution below uses the ACCUM operator to sum up all of the numbers. It also makes sure that the previous result is clear by first calling the CLR_RES operator.

52. verif/vkits/alu/alu_seq_lib.sv

```
class sum_array_seq_c extends uvm_sequence #(item_c, item_c);
   `uvm_object_utils_begin(alu_pkg::sum_array_seq_c)
      `uvm_field_array_int(data, UVM_ALL_ON | UVM_DEC)
   `uvm_object_utils_end

   //------------------------------------------------------------------------------
   // Group: Fields

   // field: data
   // An array of words to be summed
   rand bit [15:0] data[];

   // constraint: data_cnstr
   // keep it to a reasonable size
   // and use small numbers only
   constraint data_cnstr {
      data.size() inside {[1:50]};
      foreach(data[x]) {
         data[x] inside {[0:100]};
      }
   }

   // field: result
   // The final answer
   result_t result;

   //------------------------------------------------------------------------------
   // Group: Methods

   function new(string name="sum_array_seq");
      super.new(name);
   endfunction : new

   ////////////////////////////////////////
   // func: body
   virtual task body();
      item_c item;

      `uvm_do_with(item, { operation == CLR_RES; })
      get_response(rsp);
      foreach(data[x]) begin
         `uvm_do_with(item, { operation == ACCUM; alpha == data[x]; })
         get_response(rsp);
      end
      result = rsp.result;
      `cn_info(("Sum of this array = 0x%0x", result))
   endtask : body
endclass : sum_array_seq_c
```

We've also used SystemVerilog's handy foreach construct. This neatly replaces iterators from C++ and creates two implicit variables: x and data[x]. **These implicit variables are local in scope only to the foreach block.**

The changes to the driver's run_phase are straightforward.

53. verif/vkits/alu/alu_drv.sv

```
   virtual task run_phase(uvm_phase phase);
      result_t prev_result;
```

```
        forever begin
           seq_item_port.get_next_item(req);
           `cn_info(("Driving: %s", req.convert2string()))
           case(req.operation)
              item_c::MUL_A_B:  req.result = result_t'(req.alpha * req.beta);
              item_c::ACCUM:    req.result = result_t'(prev_result + req.alpha);
              item_c::CLR_RES:  req.result = 0;
           endcase
           prev_result = req.result;
           seq_item_port.item_done(req);
        end
     endtask : run_phase
```

Sequencing other Sequences

So far, our little sequences have just been producing sequence items of our transaction type, `item_c`. But, the `uvm_sequence` class is derived from `uvm_sequence_item`. Therefore, **sequences can also *do* other sequences**. This allows you to have layers of sequences.

When you call `` `uvm_do `` (or any of its variants) on a sequence, things are just a little bit different. First of all, you don't need to (in fact, you shouldn't) call `get_response`. That's because the `` `uvm_do `` macros recognize that you are using a sequence and they call their `start` tasks, instead of `start_item`. When `` `uvm_do `` completes, your sequence is already done, and whatever response data that's stored in the sequence is available for your inspection.

Problem 9-4

With everything you've learned so far, this next one should present some fresh challenges. Use our `factorial_seq_c` and our `sum_array_seq_c` to provide a `sum_of_factorials_seq_c` that solves this equation:

$$\sum_{k=x}^{y} k = x! + (x+1)! + (x+2)! + ...(y-1)! + y!$$

Solution

Here is an implementation of the sum-of-factorials equation using the `sum_array_seq_c` sequence. Hopefully yours looks similar:

54. verif/vkits/alu/alu_seq_lib.sv

```
class sum_of_factorials_seq_c extends uvm_sequence #(item_c);
   `uvm_object_utils_begin(alu_pkg::sum_of_factorials_seq_c)
      `uvm_field_int(op_x, UVM_ALL_ON | UVM_DEC)
      `uvm_field_int(op_y, UVM_ALL_ON | UVM_DEC)
      `uvm_field_int(result, UVM_ALL_ON | UVM_DEC)
   `uvm_object_utils_end

   //-------------------------------------------------------------------------------------------
   // Group: Fields

   // vars: op_x, op_y
   // Operands for this summation function
   rand bit [15:0] op_x;
   rand bit [15:0] op_y;

   constraint operands_cnstr {
      op_x < op_y;
      op_x inside {[1:8]};
      op_y inside {[1:8]};
   }

   // field: result
   // The final result
   result_t result;

   //-------------------------------------------------------------------------------------------
   // Group: Methods
   function new(string name="sum_of_factorials_array_seq");
      super.new(name);
   endfunction : new

   ////////////////////////////////////////////
   // func: body
   virtual task body();
      int num;
      bit [15:0] data[];
      factorial_seq_c fact_seq;
      sum_array_seq_c sum_seq;
      int  idx;

      // fill the data array with all the factorials
      data = new[(op_y - op_x + 1)];
      idx = 0;
      for(num = op_x; num <= op_y; num++) begin
         `uvm_do_with(fact_seq, { operand == num; })
         data[idx] = fact_seq.result;
         idx++;
      end

      // now sum the array
      `uvm_create(sum_seq)
      sum_seq.data = data;
      `uvm_send(sum_seq)
      result = sum_seq.result;
      `cn_info(("The sum of factorials from %0d to %0d is %0d.",
                op_x, op_y, result))
   endtask : body
endclass : sum_of_factorials_seq_c
```

A few important notes about this implementation:

```
        data = new[(op_y - op_x + 1)];
```

This is how you size a dynamic array—by new'ing it with the number of entries. Using a queue would be a nice alternative, but the `sum_array_seq_c` expects a dynamic array of unsigned bytes, so that's what we'll give it.

```
        `uvm_create(sum_seq)
        sum_seq.data = data;
        `uvm_send(sum_seq)
```

We do not want to create a random sequence with random data. We want to use a very specific array, therefore we cannot use `` `uvm_do ``. Instead, we call `` `uvm_create `` to perform both the `new` operation and to wait for the sequencer to accept it. Then, we assign the data array to our local array. Because it is an array, this is merely copying a reference and therefore does not have the overhead of copying each byte at a time. Finally, we call `` `uvm_send `` to send it to the sequencer. When this call returns, the sequence has completed and we can fetch the answer.

Sequence Hierarchy

Allowing sequences to 'do' other sequences is the basis of a sequence hierarchy. More complex examples might involve a complete protocol hierarchy. A hefty example is an independent TCP or UDP session, living inside IPv4 packets, driven into a device using SRIO messages, all for the sake of performing a compression operation.

What's the point of all this? The point is that the compression sequence does not need to know anything about the underlying sequences, which interface it went over, or how it was broken up into many packets over a long period of time. It just deals with sending in a compression request, and when it's done, it has the compression result.

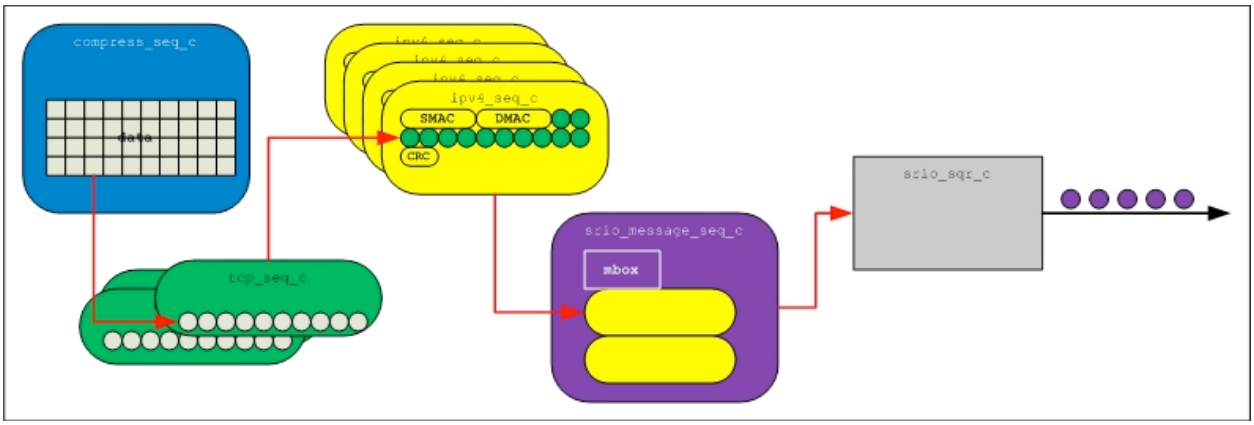


Figure 10: *Sequence Hierarchy*

The Sequencer

Take a look at your `alu_sqr.sv` sequencer file. Note that there's really nothing in it. It is actually doing quite a bit, though. Where sequencers really rock is when they are handling multiple streams of stimulus at once. So far, we've been sending it one sequence at a time. Pretty boring. Let's instead use `fork..join` to start multiple sequences at the same time. Change your basic test's `main_phase` to something more like this:

55. verif/alutb/tests/basic.sv

```
virtual task main_phase(uvm_phase phase);
   alu_pkg::sum_of_factorials_seq_c sof_seq = new("sof");
   alu_pkg::sum_array_seq_c sum_array_seq = new("sum_array");
   alu_pkg::exer_seq_c exer_seq = new("exer_seq");

   phase.raise_objection(this);
   fork
      begin
         sof_seq.randomize();
         `cn_info(("Starting:\n%s", sof_seq.sprint()))
         sof_seq.start(alutb_env.alu_agent.sqr);
         `cn_info(("The sum-of-factorials from %0d to %0d is %d",
                   sof_seq.op_x, sof_seq.op_y, sof_seq.result))
      end

      begin
         sum_array_seq.randomize() with {data.size() > 8; };
         `cn_info(("Starting:\n%s", sum_array_seq.sprint()))
         sum_array_seq.start(alutb_env.alu_agent.sqr);
         `cn_info(("The sum is %0d", sum_array_seq.result))
      end

      begin
         alu_pkg::factorial_seq_c fact_seq = new("fact_seq");
         fact_seq.randomize();
         fact_seq.start(alutb_env.alu_agent.sqr);
      end
   join

   phase.drop_objection(this);
endtask : main_phase
```

The `fork..join` concept is one you'll soon use a lot of. Here, we've spawned four separate threads that will start four separate randomized sequences on the `alutb_env.alu_agent.sqr` at the same time. The sequencer now has four sequence threads arbitrating for its attention. It has a built-in arbitrator that has several different modes of operation, shown in Table 6.

The sequencer chooses to arbitrate among the sequences when the currently chosen sequence either waits for a delay or sends an item. When the sequence calls `start_item()`, it puts its hat in the ring to try to win the next arbitration, and the item's priority and other factors will determine the winner. The arbitration occurs when the driver calls `get_next_item()`. If the sequence is chosen, then the `start_item()` task finishes and the sequence's thread takes over, performing randomization on the item, or whatever else you wish, and eventually sending it along to the driver.

Arbitration Type	Purpose
SEQ_ARB_FIFO	Requests are granted in FIFO order (default)
SEQ_ARB_WEIGHTED	Requests are granted randomly by weight
SEQ_ARB_RANDOM	Requests are granted randomly
SEQ_ARB_STRICT_FIFO	Requests at highest priority granted in fifo order
SEQ_ARB_STRICT_RANDOM	Requests at highest priority granted in randomly
SEQ_ARB_USER	Arbitration is delegated to the user-defined function, user_priority_arbitration. That function will specify the next sequence to grant.

Table 6: *Sequencer Arbitration Algorithms*

This interleaving of sequence items is the whole point of sequences and sequencers. While they have other benefits, this is what turns generic operations into a tangled web of stimulus that is better at catching bugs than directed test cases are.

Below is a more complex example of how sequencers and their sequences can be viewed. On the left is a hierarchy of independent sequences, each one minding its own business and doing its own thing. They can each *do* lower-level sequences. In the middle, the sequencer performs the arbitration and interleaving. On the rightmost edge is the random stream of transactions themselves.

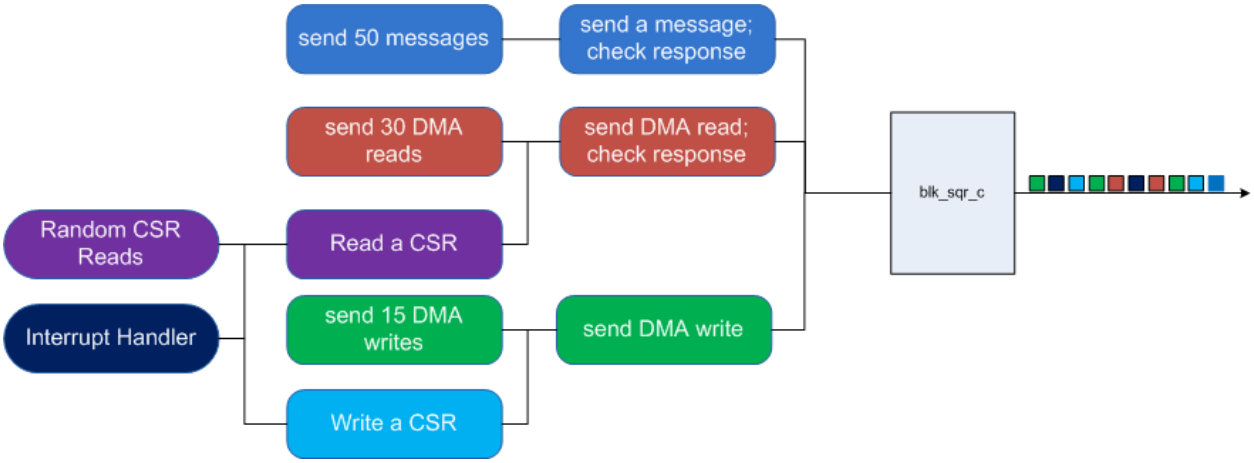


Figure 11: *Sequencer Interleaving*

And it does it all for free, which actually has two benefits:

1. First, you don't have to write this complex arbitrator yourself, for every testbench, and for every transaction type.

2. Second, these sequences essentially operate in a vacuum. They effectively separate the generation of stimulus from the underlying implementation, allowing you to focus on

one thing at a time. Creating a sum-of-factorials sequence does not need to take into account how the factorial sequence was implemented.

Locking and Grabbing

If you now run your test, you'll find that all your sequences are quite jumbled together. However, if you used your calculator to verify the output in the logfile, you'll see this leads to a problem. In order to sum an array of values using the clear and accumulate operations, these sequences need to be atomic. Otherwise, the stored result value in the ALU will become mixed up with all the other calculations.

A sequence can get exclusive access to its sequencer via `lock` or `grab`. When a sequence locks its sequencer, it first waits for it to win arbitration, and then it blocks other sequences until it calls `unlock`. Calling `grab` does the same thing, but it prioritizes the lock so that it will win arbitration next so long as other sequences do not already have exclusivity on the sequencer. You would use `grab` instead of `lock` for higher priority sequences—such as an interrupt handler.

Problem 9-5

Change the sequence body of `sum_array_seq_c` to lock the sequencer before doing the `CLR_RES` operation and to unlock it after the last `ACCUM` operation. Verify that your results are now correct.

Solution

The body of your `sum_array_seq_c` will now have this as its `body` task:

56. verif/vkits/alu/alu_seq_lib.sv

```
virtual task body();
   item_c item;

   lock();
   `uvm_do_with(item, { operation == CLR_RES; })
   get_response(rsp);
   foreach(data[x]) begin
      `uvm_do_with(item, { operation == ACCUM; alpha == data[x]; })
      get_response(rsp);
   end
   unlock();
   result = rsp.result;
   `cn_info(("Sum of this array = %0x:", result))
endtask : body
```

Conclusion

Sequences and sequencers offer dramatically more features than presented here. You'll see more of them later. What we know now should suffice to provide enough interesting stimulus for the driver we're about to write.

Chapter 10: Drivers and Monitors

We already created the empty driver class way back in Chapter 7. In this chapter, we're going to hook it up to the interface we created in Chapter 6, receive requests from the sequencer, drive the transactions, collect the results, and send them back to the sequencer. Then we will create a monitor in the same way and use analysis ports to announce what happens on the interface.

Responsibilities of Drivers and Monitors

At their heart, drivers and monitors are really very simple:

> Drivers and monitors must operate at the *lowest level of atomicity* for the interface it serves.

In our case, the ALU transaction cannot be divided up into smaller pieces, so that is the sequence item type that we developed in Chapter 5. Other interfaces allow for time-division multiplexing of cycles between larger transaction types. For example, a bus protocol in which packet data can be independently streamed from different ports should have as its sequence item the single clock cycle of data and not the packet.

> A single agent should be designed for each "action" on a bus.

Often a bus protocol that is thought of as a single interface can contain multiple "actions" happening at once. For example, data may flow in one direction while credits are returned in the other direction. These different actions should be separated into different agents, each with their own interfaces so that they can act independently. The data driver agent will receive credit notifications from the credit agent and drive data accordingly.

> Drivers must never be responsible for deciding what to do next.

Sequences and sequencers are excellent at arbitrating between multiple streams of data (such as the packet example above), at waiting on available resources (i.e. credits), and at randomly inserting idle cycles. Putting these smarts inside of a driver is a mistake that will become obvious when you learn more advanced sequencing techniques.

Fetching the Interface

You'll recall that during Chapter 6 we instantiated the `alu_intf` in the top-level testbench and pushed virtual references to its modports into the configuration database with the names `alu_pkg::drv_vi` and `alu_pkg::mon_vi`. The suffix `_vi` indicates that it is a virtual interface **[Rule 2.5-1]**.

We are trying to make a reusable agent and the interfaces won't always have those names, so let's create a configuration string in the driver called `intf_name`:

57. verif/vkits/alu/alu_drv.sv

```
class drv_c extends uvm_driver#(item_c);
   `uvm_component_utils_begin(alu_pkg::drv_c)
      `uvm_field_string(intf_name, UVM_ALL_ON)
   `uvm_component_utils_end

   //---------------------------------------------------------------------------------------
   // Group: Configuration Fields

   // field: intf_name
   // The name of the virtual interface that we'll hook up to
   string intf_name = "drv_vi";
```

We use the macro `` `uvm_field_string `` to declare this configuration field, so that it will auto-populate itself when its `super.build_phase()` is called. If no other classes have added this configuration to the database for this driver, then it will take the default name of "`drv_vi`," which is what we want in the first place. This is how you give a configuration its default value.

Next, we need to declare the virtual interface as a field in the driver, so that it can access the signals:

58. verif/vkits/alu/alu_drv.sv

```
   //---------------------------------------------------------------------------------------
   // Group: Fields

   // field: drv_vi
   // Virtual interface to drive on
   virtual alu_intf.drv_mp drv_vi;
```

We've placed this definition in the fields group, to let readers know that this is not something they'll be touching. All they need to do is set the `intf_name` string correctly. Also, the type of `drv_vi` must match **exactly** what was used to store this interface in the database in `alutb_tb_top`:

```
   `cn_set_intf(virtual alu_intf.drv_mp, "alu_pkg::alu_intf", "drv_vi", alu_i.drv_mp);
```

This will give us access to the driver modport of this interface. Now it needs to be fetched from the interface.

Problem 10-1

Fetch the interface from the resource database and assign it to `drv_vi`. In the driver's `run_phase`, call the interface's `reset()` function at the start of time, but keep the fake driver code that was there previously.

You can also **remove** the call to `reset()` that we left in the testbench earlier:

```
      initial
         alu_i.reset();
```

Afterwards, you should be able to simulate successfully.

Solution

Like the `cn_set_intf macro that was used earlier, there is a corresponding `cn_get_intf:

59. verif/vkits/alu/alu_drv.sv

```
virtual function void build_phase(uvm_phase phase);
   super.build_phase(phase);
   // get the interface
   `cn_get_intf(virtual alu_intf.drv_mp, "alu_pkg::alu_intf", intf_name, drv_vi)
endfunction : build_phase
```

The macro unrolls to this:

```
if(!uvm_resource_db#(virtual alu_intf.drv_mp)::get("alu_pkg::alu_intf", intf_name, drv_vi))
   `cn_fatal(("%s virtual interface not present.", intf_name))
```

Note that it is important to flag a fatal error if the interface is not found, because that would indicate a configuration error, and there is no purpose in continuing. The call to get the interface must take place *after* the call to `super.build_phase()` because if it happened before then the `intf_name` variable would not have been updated by the database.

Your `run_phase` should now look like this to ensure that the assertions in the interface do not fire.

60. verif/vkits/alu/alu_drv.sv

```
virtual task run_phase(uvm_phase phase);
   result_t prev_result;

   drv_vi.reset();

   // constantly poll for new transactions, printing them out
   forever begin
      seq_item_port.get_next_item(req);
```

Getting the Next Item

As you saw in Chapter 9, drivers have a built-in `port` called `seq_item_port` through which the sequencer pushes `uvm_sequence_items`. It is a special `port` of type `uvm_seq_item_pull_port` that is parameterized to your request and response types. Like the sequences, the driver also has built-in fields `req` and `rsp` corresponding to its current request and response.

To get the next request, you call a blocking task :

```
seq_item_port.get_next_item(req);
```

It will return when the next request is available. Or, you can call the function `try_next_item(req)` to see if one is available in zero-time.

Our ALU design operates in a pull-mode. The driver attempts to fetch the next item from the sequencer, and the sequencer's arbitration scheme selects a sequence, and so on. Both driver and sequencer are also offered in push-mode, where the sequencer pushes sequence items to the driver.

Driving a Transaction

There are myriad approaches to doing this, but we went through the exercise of learning how to pack and unpack transactions, so we might as well use that method.

UVM's current packing algorithms are offered in three flavors: give me a list of bits, give me a list of bytes, and give me a list of ints (32-bit numbers). Since this is the fun part, you can do it.

Problem 10-2

Write a task in the driver to fetch and drive transactions. Name it `driver`, and change the `run_phase` task to reset the interface and to call this task.

Hint:

You can use the following algorithm:

- Call the interface's reset function, to ensure that X's do not get into the design.
- Run forever:
  1. Get the next transaction.
  2. Pack the transaction into an array of bytes.
  3. For each byte:
     - Wait 1 clock cycle on the interface.
     - Assert the `ctl` line only if this is the very first cycle.
     - Drive the byte of data.
  4. Wait 1 clock, then clear the bus.
  5. Wait for the `ready` line to go high.
  6. Fetch the result from the `result` bus.
  7. Assign the result to the original request's result field.
  8. Call `item_done` and send back the original request.

Solution

The challenge here was probably dealing with the clocking block. While it may make for extra typing here, clocking blocks are worth the effort if the clock edges may change or if you later want to add sampling delays. Clocking blocks in an interface define what the sampling clock edge is, so the notion of `@(posedge clk)` has been abstracted away. If an interface suddenly became a dual-data rate interface, the only place you would need to change any code would be in the clocking block.

They also eliminate race conditions between the verification environment and the RTL, because they have a default sampling delay of `#1STEP`.

61. verif/vkits/alu/alu_drv.sv

```
// func: driver
// Drive transactions by packing into an array of bytes, then sending two 4-bit cycles
// for each byte. Then wait for the response and send it back.
task driver();
   byte unsigned stream[];

   @(posedge drv_vi.drv_cb.rst_n);
   forever begin
      seq_item_port.get_next_item(req);

      req.pack_bytes(stream);
      foreach(stream[x]) begin
         @(drv_vi.drv_cb);

         drv_vi.drv_cb.ctl <= (x == 0)? 1'b1 : 1'b0;
         drv_vi.drv_cb.dat <= stream[x];
      end

      // wait 1 clock, then clear the bus
      @(drv_vi.drv_cb);
      drv_vi.reset();

      // wait for result
      @(posedge drv_vi.drv_cb.ready);
      req.result = drv_vi.drv_cb.result;
      seq_item_port.item_done(req);
   end
endtask : driver
```

If you happened to leave the checker in the `sum_array_seq_c`, then hopefully the test still passes.

A few notes on the above design:

```
         @(drv_vi.drv_cb);
```

Because of the clocking block, the method to wait 1 clock is to just wait on the clocking block.

```
         drv_vi.drv_cb.ctl <= (x == 0)? 1'b1 : 1'b0;
         drv_vi.drv_cb.dat <= stream[x];
```

The virtual interface that the driver has is a modport of an existing interface. Recall that this modport, `drv_mp`, has only two features: `drv_cb` and the `reset` task:

```
    modport drv_mp(clocking drv_cb,
                   import reset);
```

Therefore, all signal accesses (both read and write) must take place through the clocking block, which also specifies the direction (input or output) that the driver is permitted to access. There is no such signal as `drv_vi.dat`. There is only `drv_vi.drv_cb.dat`. The same can be said for `rst_n`.

```
        @(posedge drv_vi.drv_cb.ready);
        req.result = drv_vi.drv_cb.result;
```

Notice that because of the clocking block, we are not concerned about a race condition between `ready` and `result`. Both signals are only examined on the positive edge of the clock, so the its positive edge is not perceived by the verification environment until the point shown in Figure 12.

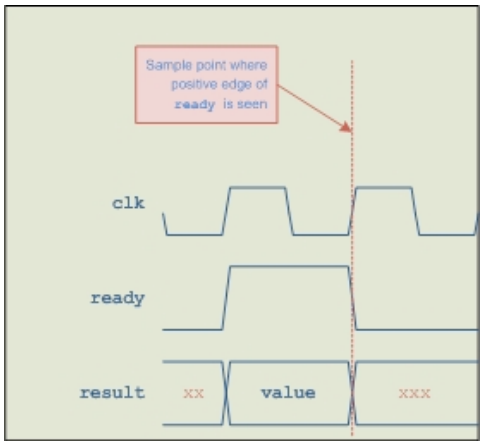


Figure 12: *Sample point due to clocking block.*

Monitoring Activity

As its name implies, the monitor is a component that watches the interface. It checks for any protocol violations and collects all activity it sees, broadcasting these as transactions to all listeners through an `analysis_port`. Unlike the driver and the sequencer, a monitor is always present in an agent whether it is active or not.

The monitor in this agent will look very much like the driver, except the exact opposite. Instead of watching the `drv_cb` clocking block, it will watch the `mon_cb`; instead of getting transactions from the sequencer, it will broadcast the ones that it sees; and instead of packing and driving transactions, it will collect data and unpack it into a transaction.

Problem 10-3

Write the ALU monitor. Use the hints below if you need to.

Hint:

As with the driver, we must start by making sure we get the interface. Then, set up the `run_phase` to launch two tasks: `monitor_item` and `monitor_result`. Use the `fork..join_any` construct so that the tasks are disabled on reset.

The `monitor_result` task should be straightforward: constantly wait for the rising edge of the `ready` signal, and broadcast the result bus out the `monitored_result_port` by calling its `write()` function.

The `monitor_item` task is a little bit trickier. Because the protocol offers no end-of-transaction signal, it must be inferred from the operation. Wait for the rising edge of `ctl`, grab the data, and based on the operation that's being driven, collect the correct number of data cycles in a dynamic array of bytes. Report an error if the `ctl` signal goes high on any cycle other than the first one.

When that's done, you should be able to create a new transaction, unpack those bytes into it, and write it out of the `monitored_item_port`. You might also want to add debug messages to see what it's doing.

Solution

Here is one possible implementation of the ALU monitor, with some explanations following:

62. verif/vkits/alu/alu_mon.sv

```
`include "alu_item.sv"

// class: mon_c
// Monitors an ALU bus and reports activity.
class mon_c extends uvm_monitor;
   `uvm_component_utils_begin(alu_pkg::mon_c)
      `uvm_field_string(intf_name, UVM_ALL_ON)
   `uvm_component_utils_end

   //---------------------------------------------------------------------------------------
   // Group: Configuration Fields

   // field: intf_name
   // The name of the virtual interface that we'll hook up to
   string intf_name = "mon_vi";

   //---------------------------------------------------------------------------------------
   // Group: TLM Ports

   // field: monitored_item_port
   // All monitored transactions go out here
   uvm_analysis_port #(item_c) monitored_item_port;

   // field: monitored_result_port
   // All monitored results go out here
   uvm_analysis_port #(result_t) monitored_result_port;

   //---------------------------------------------------------------------------------------
   // Group: Fields

   // field: mon_vi
   // Virtual interface to monitor
   virtual alu_intf.mon_mp mon_vi;

   //---------------------------------------------------------------------------------------
   // Group: Methods
   function new(string name="mon",
                uvm_component parent=null);
      super.new(name, parent);
   endfunction : new

   ////////////////////////////////////////////
   // func: build_phase
   virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);

      // get the interface
      `cn_get_intf(virtual alu_intf.mon_mp, "alu_pkg::alu_intf", intf_name, mon_vi)

      monitored_item_port = new("monitored_item_port", this);
      monitored_result_port = new("monitored_result_port", this);
   endfunction : build_phase

   ////////////////////////////////////////////
   // func: run_phase
   virtual task run_phase(uvm_phase phase);
      forever begin
         @(posedge mon_vi.mon_cb.rst_n);

         fork
            monitor_item();
```

```
            monitor_result();
            @(negedge mon_vi.mon_cb.rst_n);
         join_any

         `cn_info(("Stopping monitor due to reset."))
         disable fork;
      end

   endtask : run_phase

   //////////////////////////////////////////
   // func: monitor_item
   // Watch and broadcast the transactions on the bus
   virtual task monitor_item();
      int num_clocks;
      byte unsigned data[];
      item_c item;

      forever begin
         // wait for the rising edge of the control
         @(posedge mon_vi.mon_cb.ctl);

         // determine how many clocks are in this item
         case(mon_vi.mon_cb.dat)
            // 5-cycle transactions
            item_c::ADD_A_B, item_c::SUB_A_B, item_c::SUB_B_A,
            item_c::MUL_A_B, item_c::DIV_A_B, item_c::DIV_B_A :
               num_clocks = 5;

            // 3-cycle transactions
            item_c::INC_A, item_c::INC_B, item_c::ACCUM :
               num_clocks = 3;

            // 1-cycle transactions
            item_c::CLR_RES :
               num_clocks = 1;

            default:
               `cn_err(("Unknown operation type: %02X", mon_vi.mon_cb.dat))
         endcase

         // collect the data for each cycle
         data = new[num_clocks];
         for(int clk=0; clk < num_clocks; clk++) begin
            data[clk] = mon_vi.mon_cb.dat;
            @(mon_vi.mon_cb);
            if(mon_vi.mon_cb.ctl == 1)
               `cn_err(("The CTL signal is high during a transaction that should have been %0d
clocks.", num_clocks))
         end

         // create the transaction, unpack into it, and write it out the monitored_item_port
         item = item_c::type_id::create("mon_item");
         item.unpack_bytes(data);
         `cn_info(("Monitored: %s", item.convert2string()))
         monitored_item_port.write(item);
      end
   endtask : monitor_item

   //////////////////////////////////////////
   // func: monitor_result
   // Monitor the ready and result signal and broadcast it out the monitored_result_port
   virtual task monitor_result();
      forever begin
         @(posedge mon_vi.mon_cb.ready);
         `cn_info(("Monitored Result: %08X", mon_vi.mon_cb.result))
         monitored_result_port.write(mon_vi.mon_cb.result);
      end
   endtask : monitor_result

endclass : mon_c
```

A few explanations are in order:

```
            @(posedge mon_vi.mon_cb.ctl);
```

Here, we are waiting for the positive edge of the `ctl` signal. Because of the clocking-block, there is no concern that this event will be triggered on the rising edge of `ctl` but before the DUT drives the `dat` lines. Why is that? Because the event will sample on the clock specified by the clocking block—in this case, the positive edge of `clk`. If the clocking block had specified the negative edge of `clk`, you would find that the `posedge` event triggers on the negative edge of the `clk`, even though the `ctl` signal is driven on the rising edge.

```
            `cn_err(("Unknown operation type: %02X", mon_vi.mon_cb.dat))
```

A key element to any monitor is protocol checking. The interface itself already checks for X's. Here, we ensure that only legal values are specified on the first cycle. Later, we check that the `ctl` signal does not go high any other time during the transaction. And, at the end of the transaction, we guarantee that the data collected has a legal format because it can successfully unpack. If the unpack were to fail, it would report an error for us.

```
            data = new[num_clocks];
```

If you're coming from a C/C++ background, you might be wary by this line, because there are no free or delete operators, and this sits in a forever loop. This code looks like a memory leak, but it's not. SytemVerilog has automatic garbage-collection, so when nobody else is referencing this data, it gets freed automatically. As it happens, this data is local in scope to this task, and it is copied byte-by-byte into the transaction during the unpack operation. Therefore, the bytes pointed to by the `data` variable will have zero references the next time the `data` variable is newed.

```
            item = item_c::type_id::create("mon_item");
```

Like the `data` variable, the `item` variable is local in scope to the task, and we keep creating new ones. Are these also garbage-collected when the `item` variable creates a new one? Well, yes and no. If nobody were listening on the analysis `port`, then they would be garbage collected. In this testbench, though, we have the monitor broadcasting to the agent, which broadcasts to the `alu_item_subscriber_c` class we implemented in Chapter 8. This subscriber doesn't hold onto the transaction, though. It merely prints it out, so these transactions will be garbage collected. Other subscribers may store them in a scoreboard, or some other temporary location, and their reference count will not go to zero, so they will not be freed.

```
            item.unpack_bytes(data);
```

The `unpack_bytes` function is used in the same manner as the `pack_bytes` function we used in the driver. The `unpack_bytes` function will report an error on its own if it is

unsuccessful, and it returns an `int` which is the number of bits that were unpacked. We won't need this information, though, so we don't look at it.

```
            monitored_item_port.write(item);
```

Here we are writing that transaction out of the analysis `port`. Elsewhere, we write the result value out of the other monitored result analysis `port`. Since these are connected to the agent's analysis ports, they will likewise be broadcast out of the agent.

```
      virtual task run_phase(uvm_phase phase);
         forever begin
            @(posedge mon_vi.mon_cb.rst_n);

            fork
               monitor_item();
               monitor_result();
               @(negedge mon_vi.mon_cb.rst_n);
            join_any

            `cn_info(("Stopping monitor due to reset."))
            disable fork;
         end
```

Because your drivers and monitors should be resilient enough to handle reset conditions, the run phase handles these by disabling each of the outstanding tasks and looping back again.

However, this algorithm can be improved upon. The paper entitled "Reset Testing Made Simple with UVM Phases" discusses this topic in more detail, and the design patterns in Appendix D use these best practices already.

Conclusion

Drivers and monitors are a snap in SystemVerilog because of all the built-in constructs like events, queues, dynamic and associative arrays, and easy signal access. Hooking them up the UVM way wasn't too difficult, either. Users should first consider the design patterns presented in Appendix D.

Chapter 11: Writing a Predictor

We've been so focused on creating, driving, and monitoring the stimulus that we haven't bothered to check if it is correct or not. We'll be getting to reading and writing CSRs in Chapter 12, so for now consider the K and C values to be 1 and 0, respectively.

UVM offers a variety of options when constructing your predictor. Some are more appropriate than others, depending on the nature of the prediction being made.

Monitor Prediction

For such a small testbench, you are probably tempted to put all of the prediction in the monitor. And for something this simple, that decision is probably justified. Typically, though, this is not such a great idea. Separating the low-level pin details and interface protocol checking from the higher-level prediction algorithms is often considered the best practice.

Comparators

UVM offers several built-in comparator components that operate as simplistic versions of the PW scoreboard. The `uvm_in_order_comparator`, `uvm_in_order_built_in_comparator`, and `uvm_in_order_class_comparator` all compare two streams of data objects, both of the same type. This may be useful in situations where what goes in must come out, but for most applications probably will not be sufficient.

One place where these comparators can be useful is as an agent *self-check*. Placing a comparator in the agent between the driver and the monitor ensures that both components are in sync with one another.

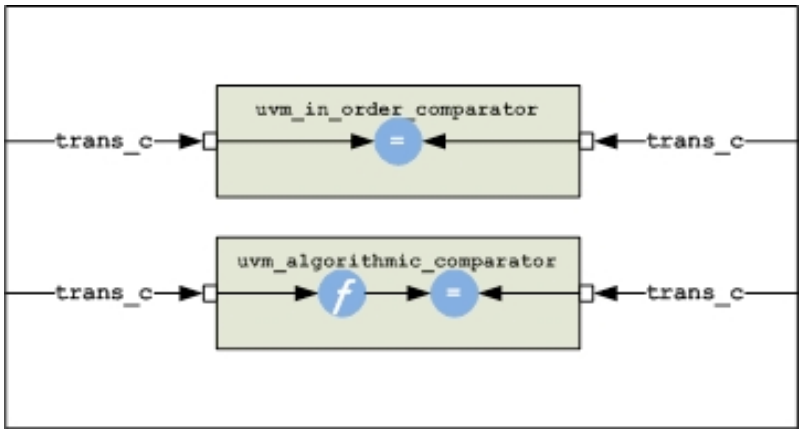


Figure 13: *UVM Comparators*

The `uvm_algorithmic_comparator` offers similar functionality, but is parameterized to work with different transaction classes as the two streams and also takes a class that is used to transform the first class into the predicted second class.

This sounds perfect for our ALU, which would have transaction sequences going in one `port`, and results going in the other, with a simple math function in between to predict expected results. What could be easier?

Unfortunately, the class that UVM 1.1d offers has one small bug—it cannot take an integer type as one of the streams. We would have to enclose the result value into a separate results class containing just a 32-bit value. This limitation is not overwhelming, though.

Scoreboards

Scoreboards are usually the avenue of choice when it is impractical to predict precisely what the design will do in all situations. Scoreboards can raise the level of abstraction to foresee what should happen to a transaction or series of transactions. They are often employed in the networking world where transactions can be broken into small pieces and re-assembled in a seemingly arbitrary manner. Here, the intermediate answers of the model are unimportant but ensuring that the final answer is correct is the most practical course of action.

UVM provides a `uvm_scoreboard` class, but it might interest you to know that in UVM 1.1 that class is just an empty component and serves no useful purpose. (As a matter of fact, so are the agent, environment, test, and a variety of other UVM classes). The reasons these "empty" classes exist are that they help identify the purpose of the derived class you're creating and they serve as placeholders for future functionality that may come along someday.

If the nature of your prediction algorithm suggests a scoreboard, this is the component you should create. Pipe monitored requests and responses into the scoreboard component via TLM interfaces and use some of SystemVerilog's handy data types such as queues or associative arrays and away you go.

Reference Modeling

A reference model is a component that receives the same input as the DUT and performs the same algorithms, providing the expected result to the testbench. Reference modeling is usually chosen when the behavior of the DUT is very well understood and unlikely to evolve during the course of development. Processor environments often use reference models to run the same instruction code and predict what the processor should do on every clock cycle.

The ALU's static algorithm and it's one-at-a-time operation is perfect for the reference model approach.

Problem 11-1

Instantiate an ALU predictor (`pred_c`) that derives from `uvm_component` in the ALU agent and provide it with two analysis `imps`, one taking an `item_c` and the other taking a `result_t` value. Use the `` `uvm_analysis_imp_decl `` macros to permit you to create both `write()` functions.

Hook up the predictor's `imps` to the monitors ports as shown in the following diagram and fire away.

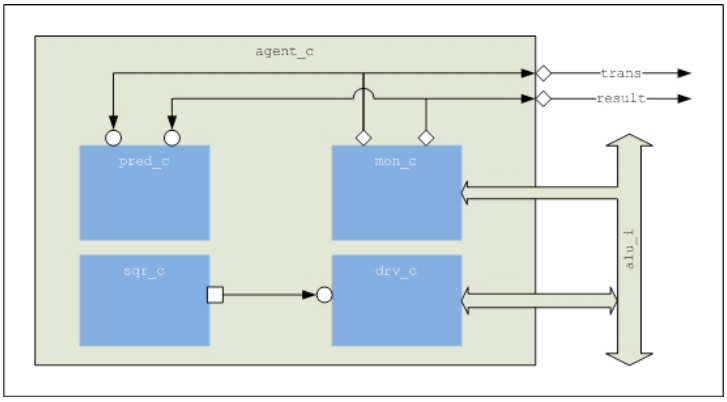


Figure 14: *Connecting the Predictor*

Solution

Hopefully this was not too challenging for you. As discussed in Chapter 8, these imp declarations can be placed in the `alu_pkg.sv` file.

63. verif/vkits/alu/alu_pkg.sv

```
   //---------------------------------------------------------------------------------------------
   // Group: Imp Declarations

   `uvm_analysis_imp_decl(_item)
   `uvm_analysis_imp_decl(_result)
```

The predictor is straightforward: receive the monitored item, calculate the expected result, and when one arrives make a comparison.

64. verif/vkits/alu/alu_pred.sv

```
`include "alu_item.sv"

class pred_c extends uvm_component;
   `uvm_component_utils(pred_c)
   //---------------------------------------------------------------------------------------------
   // Group: TLM Ports

   // field: monitored_item_imp
   uvm_analysis_imp_item #(item_c, pred_c) monitored_item_imp;

   // field: monitored_result_imp
   uvm_analysis_imp_result #(result_t, pred_c) monitored_result_imp;

   //---------------------------------------------------------------------------------------------
   // Group: Fields

   // field: result
   // The result of the monitored transaction is stored here and checked with the received result
   result_t result = 0;

   //---------------------------------------------------------------------------------------------
   // Group: Methods
   function new(string name="pred",
                uvm_component parent=null);
      super.new(name, parent);
   endfunction : new

   ////////////////////////////////////////////
   // func: build_phase
   function void build_phase(uvm_phase phase);
      super.build_phase(phase);

      monitored_item_imp = new("monitored_item_imp", this);
      monitored_result_imp = new("monitored_result_imp", this);
   endfunction : build_phase

   ////////////////////////////////////////////
   // func: write_item
   // Accepts ALU transactions and sets the next expected result
   virtual function void write_item(item_c _item);
      case(_item.operation)
         item_c::ADD_A_B :  result = _item.alpha + _item.beta;
         item_c::SUB_A_B :  result = _item.alpha - _item.beta;
         item_c::SUB_B_A :  result = _item.beta  - _item.alpha;
         item_c::MUL_A_B :  result = _item.alpha * _item.beta;
         item_c::DIV_A_B :  result = _item.alpha / _item.beta;
         item_c::DIV_B_A :  result = _item.beta  / _item.alpha;
         item_c::INC_A   :  result = _item.alpha + 1;
         item_c::INC_B   :  result = _item.beta  + 1;
```

```
            item_c::CLR_RES :  result = 0;
            item_c::ACCUM   :  result += _item.alpha;
         endcase

         `cn_dbg(30, ("Calculated result %08X on item: %s", result, _item.convert2string()))
      endfunction : write_item

      ////////////////////////////////////////
      // func: write_result
      // Called when a result is monitored
      virtual function void write_result(result_t _result);
         if(_result != result)
            `cn_err(("Actual result: %08X != Expected result: %08x",
                     _result, result))
      endfunction : write_result
   endclass : pred_c
```

Perhaps by now your only real challenge was having multiple `imps` appear in a component. This required the declaration macros, the new `uvm_analysis_imp` types that these macros created, and the naming of your implementation functions `write_item` and `write_result`.

Device Reference Modeling

It won't always be the case, but sometimes your predictor behaves very much like the DUT itself. Sometimes it is a C or SystemC reference model; or the RTL hasn't actually been written yet and you'd like something to test against; or you want to try some early performance modeling to prove the architecture; or you want to run very fast sims without the RTL to test that your functional coverage is adequate.

Whatever your reasoning, sometimes you want a *device reference model*. The predictor designed above is an excellent candidate for that.

In order to create a device reference model, we need to conditionally change the architecture of the ALU agent to something more like this diagram:

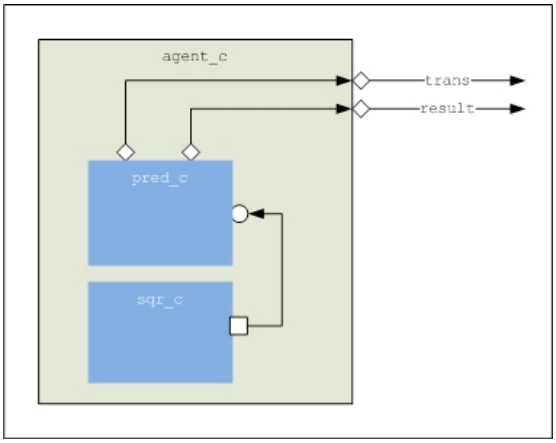


Figure 15: *Device Reference Modeling*

We could modify our predictor to contain a `seq_item_port` just like the driver, so that it can accept transactions. Then, in the main or run phase, we could have it get the next item, calculate the result just as it was doing before, and send it back to the sequencer as a response. We would also want to add analysis ports that broadcast the transactions and results as if a monitor were present. And we could have it wait for a computational delay that models the real hardware.

The rest of the verification environment is completely isolated from this change, and for all intents and purposes the RTL might as well be present.

How much practical use this method would be is a matter of debate. But in this testbench, it is so easy that it begs to be written.

Problem 11-2

Modify the ALU predictor to behave like a device reference model when a configuration field `dev_ref_model` is set. Modify the ALU agent to connect itself like a reference model when its `dev_ref_model` field is set.

Create a new test, `dev_ref_model_test_c`, that turns these bits on.

You may need to reset the ALU interface at time zero in the testbench to avoid its x-checkers.

Solution

The `alu_agent_c` modifications are fairly straightforward:

65. verif/vkits/alu/alu_agent.sv

```
`uvm_field_int(dev_ref_model, UVM_ALL_ON)
            ...
// field: dev_ref_model
// When set to 1, the predictor operates in reference mode and the monitor/driver are not enabled
bit dev_ref_model = 0;
```

In the build phase, ensure that the predictor's value of `dev_ref_model` matches the agent's by default. This way, the test(s) only have to modify the agent.

66. verif/vkits/alu/alu_agent.sv

```
//////////////////////////////////////////
// func: build_phase
virtual function void build_phase(uvm_phase phase);
   super.build_phase(phase);
   uvm_config_db#(int)::set(this, "pred", "dev_ref_model", dev_ref_model);
```

Also, don't create either the driver or the monitor when in reference model mode:

67. verif/vkits/alu/alu_agent.sv

```
   if(!dev_ref_model)
      mon = mon_c::type_id::create("mon", this);
   if(is_active) begin
      if(!dev_ref_model)
         drv = drv_c::type_id::create("drv", this);
      sqr = sqr_c::type_id::create("sqr", this);
   end
```

The connect phase should match the TLM connections of the diagram.

68. verif/vkits/alu/alu_agent.sv

```
virtual function void connect_phase(uvm_phase phase);
   super.connect_phase(phase);

   if(!dev_ref_model) begin
      // the same connections as before
   end else begin
      // as a reference model
      pred.seq_item_port.connect(sqr.seq_item_export);
      pred.monitored_item_port.connect(monitored_item_port);
      pred.monitored_result_port.connect(monitored_result_port);
   end
endfunction : connect_phase
```

The changes needed to the predictor are shown below in boldface:

69. verif/vkits/alu/alu_pred.sv

```
class pred_c extends uvm_component;
   `uvm_component_utils_begin(pred_c)
      `uvm_field_int(dev_ref_model, UVM_ALL_ON)
   `uvm_component_utils_end

   //------------------------------------------------------------------------------
   // Group: Configuration Fields

   // field: dev_ref_model
   // When set, operates in reference model mode
   bit dev_ref_model = 0;
```

```
//------------------------------------------------------------------------------
// Group: TLM Ports

// field: monitored_item_imp
uvm_analysis_imp_item #(item_c, pred_c) monitored_item_imp;

// field: monitored_result_imp
uvm_analysis_imp_result #(result_t, pred_c) monitored_result_imp;

// field: seq_item_port
// As a reference model, pulls transactions from the sequencer
uvm_seq_item_pull_port #(item_c) seq_item_port;

// field: monitored_item_port
// As a reference model, drives out the transactions that were "driven"
uvm_analysis_port #(item_c) monitored_item_port;

// field: monitored_result_port
// As a reference model, drives out the results that were "seen"
uvm_analysis_port #(result_t) monitored_result_port;

//------------------------------------------------------------------------------
// Group: Fields

// field: result
// The result of the monitored transaction is stored here and checked with the received result
result_t result = 0;

// field:
//------------------------------------------------------------------------------
// Group: Methods
function new(string name="pred",
             uvm_component parent=null);
   super.new(name, parent);
endfunction : new

////////////////////////////////////////////
// func: build_phase
function void build_phase(uvm_phase phase);
   super.build_phase(phase);

   if(dev_ref_model) begin
      seq_item_port = new("seq_item_port", this);
      monitored_item_port = new("monitored_item_port", this);
      monitored_result_port = new("monitored_result_port", this);
   end else begin
      monitored_item_imp = new("monitored_item_imp", this);
      monitored_result_imp = new("monitored_result_imp", this);
   end

endfunction : build_phase

////////////////////////////////////////////
// func: main_phase
task main_phase(uvm_phase phase);
   if(dev_ref_model) begin
      item_c item;
      forever begin
         seq_item_port.get_next_item(item);

         `cn_dbg(30, ("Dev_Ref_Model: %s", item.convert2string()))

         // create a delay that models the transmission of the transaction
         #5ns;
         monitored_item_port.write(item);

         // calculate result
         write_item(item);

         // create a delay that models the ALU calulation speed
```

```
            #15ns;

            // send back the result
            write_result(result);
            item.result = result;
            seq_item_port.item_done(item);
            monitored_result_port.write(result);
         end
      end
   endtask : main_phase

   ////////////////////////////////////////////
   // func: write_item
   // Accepts ALU transactions and sets the next expected result
   virtual function void write_item(item_c  item);
```

The big change is the `main_phase`, which now looks similar to the one in the driver. We could modify it to choose different delays based on the transaction type, or we could pull in the virtual interface that the driver uses and wait on clock edges instead.

You already know how to create the test, but just in case you've forgotten, it's here:

70. verif/alutb/tests/dev_ref_model.sv

```
`include "basic.sv"

// class: dev_ref_model_test_c
// Test the ALU using the predictor as a reference model
class dev_ref_model_test_c extends basic_test_c;
   `uvm_component_utils(dev_ref_model_test_c)

   //------------------------------------------------------------------------------
   // Group: Methods
   function new(string name="test",
                uvm_component parent=null);
      super.new(name, parent);
   endfunction : new

   ////////////////////////////////////////////
   // func: build_phase
   virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      uvm_config_db#(int)::set(this, "alutb_env.alu_agent", "dev_ref_model", 1);

      // reset the interface to avoid x-checkers
      alutb_tb_top.alu_i.reset();
   endfunction : build_phase
endclass : dev_ref_model_test_c
```

Note that we are able to reset the interface directly from the test because tests do not live in packages. The testbench is global in scope and signals or interfaces may be peeked or poked in tests as much as you wish.

A simpler alternative to this approach would be to leave the ALU agent largely the same, but disconnect the RTL from the interface. Allow the driver to drive the transaction onto the interface and the monitor to see it. Then when the monitor tells the predictor that a transaction occurred, have the predictor drive the result directly onto the interface some time later. This approach leads to fewer changes to the code, but incurs the overhead of having a driver and monitor and dissociates itself from operating only at the transaction level.

Conclusion

UVM offers a variety of choices for simple prediction schemes. TLM interfaces provide a lot of flexibility and can allow you to run without RTL at all—provided you plan ahead. Ultimately, though, prediction algorithms can be simple because of the rich set of aggregate data types and other functions that are available in the SystemVerilog language.

Chapter 12: Configuration Registers

The CSRs that are specific to this testbench are shown in the description of the ALUTB testbench. There, you'll find a CSR titled `CONST`, and it contains a read/writable field called `K_VAL`. Let's write a test that specifically configures that CSR field to a value between 9 and 20.

Register Organization

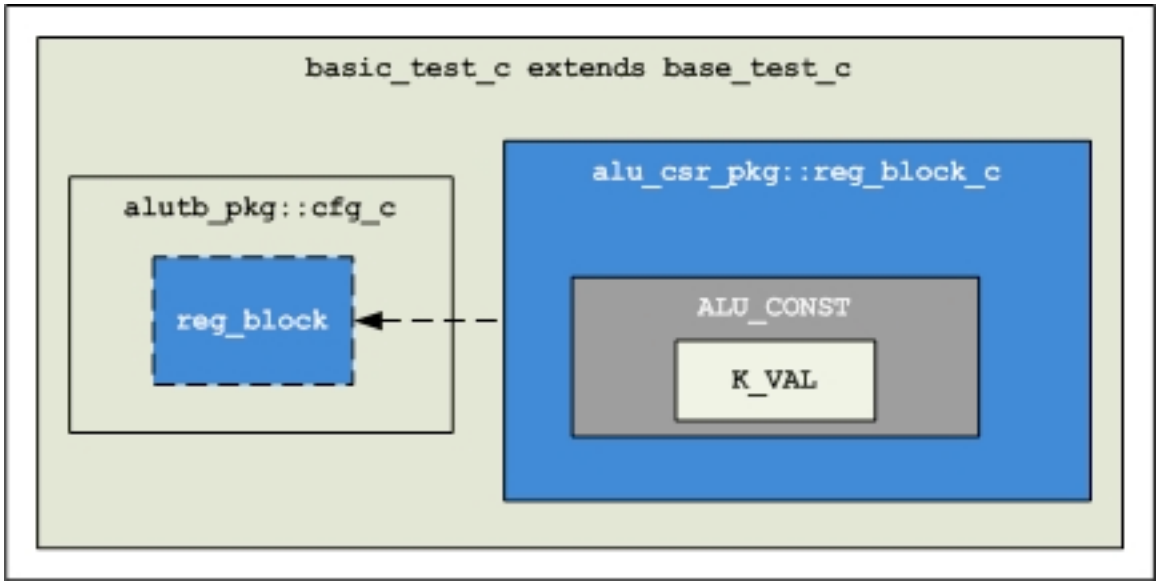


Figure 16: *Register Organization*

We should first understand where this register exists in the environment hierarchy. This picture shows that the test component instantiates two objects: `cfg` and `reg_block`. *Cfg* classes will be described in more detail in Chapter 13. The test then has the cfg class's instance of `reg_block` *reference* the one in the test (dashed lines represent handle assignments). Because it is a reference, there is only one instance of the `reg_block` in the simulation, and no extra memory is consumed. This is accomplished with the following code in `alutb/tests/base_test.sv`:

```
// create the random configurations
cfg = alutb_pkg::cfg_c::type_id::create("cfg");

// create reg_block
if(reg_block == null) begin
   reg_block = alu_csr_pkg::reg_block_c::type_id::create("reg_block", this);
   reg_block.configure(null, "alutb_tb_top.dut_wrapper.dut");
   reg_block.build_phase(phase);
   reg_block.lock_model();
end

// set configuration's reference to reg_block
cfg.reg_block = reg_block;
```

The `reg_block` is derived from `uvm_reg_block` and inside of this lives the registers and/or register files. To understand this better, let's observe the following definitions from the UVM 1.1 Reference Manual:

Register Block

> A register block represents a design hierarchy. It can contain registers, register files, memories and sub-blocks. A block has one or more address maps, each corresponding to a physical interface on the block.

Register Address Map

> An address map is a collection of registers and memories accessible via a specific physical interface. Address maps can be composed into higher-level address maps.

Register File

> A register file is a collection of register files and/or registers used to create regular repeated structures. Register files are usually instantiated as arrays.

Register

> A register represents a set of fields that are accessible as a single entity. A register may be mapped to one or more address maps, each with different access rights and policy.

Register Adapter

> This class defines an interface for converting between the generic register operation (`uvm_reg_bus_op`) and a specific bus transaction.

In short, when you hear "Register Block" think design hierarchy (such as ALU, PCIE, etc.). When you hear "Address Map" think interface (such as CTX, AMBA, or PCIE). When you hear "Register File" think of repeatable structures such as per-port statistics registers, or physical and virtual functions.

In our simple testbench, the register block contains the registers in the `alu` block. The registers should always be auto-generated by a script, but ours is already located for you in `verif/vkits/reg/gen/alu_csr_pkg.sv`. It contains the register classes and the register block.

References for Everyone!

As is usually the case, many different components and objects will want to be able to access the CSRs and the testbench's configuration class. As we saw in Chapter 7, UVM's configuration database gives us a way to push references with minimal code. We accomplish this in the following manner:

1. Put the register block into the configuration database with the field name "`reg_block`." Push it to all components by setting the `inst_name` argument to "`*`". If you have more than one register block in your testbench, then you may have to be more targeted.

2. For any component that needs access to the register block, instantiate a field of that type with the name "`reg_block`." Use the `` `uvm_field `` macros to declare this field as a member of this component. It is best to give it a flag of `UVM_REFERENCE`, so that it is not printed out multiple times in the topology report.

3. Each component will get a reference to this one instance when it calls `super.build_phase()`.

The base test uses this method to push both its `cfg` field and the `reg_block` field to any component that wishes to see them with this bit of code:

```
        // push the register block and the configurations to all blocks that ask for it
        uvm_config_db#(uvm_object)::set(this, "*", "reg_block", reg_block);
        uvm_config_db#(uvm_object)::set(this, "*", "cfg",       cfg);
```

Configuring Your Block

You'd be excused if you missed it while flipping through the reference manual, but UVM's register block class contains a nice little function called `update()`. What this function does is perform the *minimum* number of CSR writes to get the CSRs in the DUT to match the testbench's register block. So, the process of randomizing and configuring a block's CSRs is straightforward:

1. Randomize the register block. All of the CSR registers and fields are declared as `rand` member fields, and the register block is declared as a `rand` field in the `cfg` class, so randomizing the `cfg` will randomize all of the register fields.

2. Call `reg_block.update()`, preferably during the configure phase.

That was easy. Maybe too easy. It turns out that UVM will write the CSRs in an arbitrary order[4]. Maybe that's what works for your block (it works great for the `alu`), but it might require a more complicated configuration routine than that.

Configuration Sequences

In such a case, you can create a configuration sequence and set it as the *default sequence* (see Chapter 14) of a given sequencer in your environment. The registers themselves *also* have a task called `update()`, and by calling this it will perform the CSR write (if and only if one is needed).

Register blocks and registers also have a corresponding task called `mirror()` which will perform CSR reads to get the environment's version of the registers to match the one in the DUT. The mirror function can optionally perform automatic checking.

For example, imagine that before writing the `CONST` register, you wanted to also read from the `RESULT` register to ensure that it contains the correct reset value. Such a sequence body might use the `mirror` function to perform automatic checking and would be as simple as the following:

```
task body();
   uvm_status_e status;
   uvm_reg_data_t value;

   // read from the RESULT register
   reg_block.RESULT.mirror(status, UVM_CHECK);
   if(status != UVM_OK)
      `cn_err(("Read from RESULT resulted in a status of %s", status))

   // now write the CONST CSR
   reg_block.CONST.update(status);
endtask : body
```

Or, perhaps your DUT has many registers but two special registers need to be configured first before all the rest:

```
task body();
   // configure special registers first
   reg_block.SPECIAL_REG0.update(status);
   reg_block.SPECIAL_REG1.update(status);

   // configure all the rest
   reg_block.update(status);
endtask : body
```

The above relies upon the fact that calling `update` on the register block will not write to the special registers again, because they no longer need to be written.

[4] There may also be instances where the order of register writes are not seed-stable.

The kval Test

Let's put all of this together and try to accomplish the goals of this chapter. Then we'll take a look at how all of it worked.

Create a test class called `kval_test_c` and extend it from the `basic_test_c` class. In it, add this constraint:

71. verif/alutb/tests/kval.sv

```
constraint kval_cnstr {
   reg_block.CONST.K_VAL.value inside {[9:20]};
}
```

Up until now the CSRs have contained their innocuous values, due to this constraint in the `alutb_pkg::cfg_c` class:

```
// Constrain K_VAL and C_VAL to both be innocuous
constraint innocuous_cnstr {
   reg_block.CONST.K_VAL.value == 1;
   reg_block.CONST.C_VAL.value == 0;
}
```

If we were to run this test now there would be a conflict in the constraints that the simulator would be unable to solve. Go ahead and try it.

Problem 12-1

You could eliminate this problem by removing this constraint altogether. How could you fix it *without* removing the constraint or altering the configuration class?

Naturally, **your test will fail** because your predictor doesn't know anything about the K or C configurations, yet. We'll deal with that in a little bit.

Solution

Your first instinct may have been to use the factory. In that case, you could derive a class from `alutb_pkg::cfg_c`, and in its `new` function set the `constraint_mode` of the `innocuous_cnstr` to zero.

However, the `cfg` instance is accessible from the test. So as long as you turn off the constraint before the configuration class is randomized, then the constraint will not be applied. How you do that, though, is the hard part.

```
    virtual function void build_phase(uvm_phase phase);
       cfg.innocuous_cnstr.constraint_mode(0);
       super.build_phase(phase);
    endfunction : build_phase
```

If you do this, you'll find that the `cfg` instance hasn't been created yet, so you're trying to access a NULL object.

```
    virtual function void build_phase(uvm_phase phase);
       super.build_phase(phase);
       cfg.innocuous_cnstr.constraint_mode(0);
    endfunction : build_phase
```

If you do this, you'll find that the base test's build phase has already randomized the knob. This cannot be moved to the connect or end-of-elaboration phases, because it is necessary to randomize it in the build phase so that the rest of the components will build based on any configurations that might appear in `cfg`.

Fortunately, the base test was designed to solve this problem. The `cfg` randomization takes place in a virtual function called `randomize_cfg`, which you can override to do something before or after the randomization. So, in the `kval_test_c` class:

```
72.     verif/alutb/tests/kval.sv
    ////////////////////////////////////////////
    // func: randomize_cfg
    // Turn off the innocuous constraint
    virtual function void randomize_cfg();
       cfg.innocuous_cnstr.constraint_mode(0);
       super.randomize_cfg();
    endfunction : randomize_cfg
```

How It All Works

How did the register block know how to create CTX reads and writes for us? How does it know that the register isn't being written by some other interface instead?

The answer to these questions lie in the register block, the register maps, and the register adapters.

Investigate the `alu_csr_pkg::reg_block_c::build()` function (in `verif/vkits/reg/gen/alu_csr_pkg.sv`), which configures and builds each of the CSRs and builds an address map. The address map will be assigned a reference to the CTX's sequencer and register adapter by the testbench.

This happens in the base tests's connect phase. Here, the `ctx_pkg::reg_adapter_c` class is assigned to the register block's `csr_map`. It will be these classes that turn generic read and write transactions that the UVM register model knows about into the CTX reads and writes that we need.

The UVM 1.1 Class Reference and the User's Guide (section 5.5) have plenty more detail on all of these.

Question

Why couldn't the register adapter have been specified in the register block rather than in the base test?

Answer

Associating a CSR with an interface is a testbench function. In a full-chip testbench, these CSRs might not use the CTX to be written because that might be an internal interface. The writes might, instead, come from an external bus. Thus, this code does not belong in the vkit, which is meant for code that can be re-used at higher levels.

Fixing Our Predictor

Fixing the predictor to be CSR aware should be straightforward, as the predictor just needs to know the CSR values at the time it makes the prediction. Instantiate the register block in `alu_pred_c` and make it a configurable reference:

73. verif/vkits/alu/alu_pred.sv

```
`uvm_component_utils_begin(pred_c)
   `uvm_field_object(reg_block,   UVM_REFERENCE)
   `uvm_field_int(dev_ref_model,      UVM_ALL_ON)
`uvm_component_utils_end

//------------------------------------------------------------------------------
// Group: Configuration Fields

// field: reg_block
// Auto-generated Register Block
alu_csr_pkg::reg_block_c reg_block;
```

Problem 12-2

Now modify the `write_item` function to consider both the K and C values that are present in the register block.

Your `kval` test should now PASS.

Solution

Take special note of the fact that we get the current values from `uvm_reg_field` objects by looking at their `value` field.

74. verif/vkits/alu/alu_pred.sv

```
virtual function void write_item(item_c _item);
   bit [7:0] k_val = reg_block.CONST.K_VAL.value;
   bit [7:0] c_val = reg_block.CONST.C_VAL.value;

   case(_item.operation)
      item_c::ADD_A_B :  result = k_val * (_item.alpha + _item.beta) + c_val;
      item_c::SUB_A_B :  result = k_val * (_item.alpha - _item.beta) + c_val;
      item_c::SUB_B_A :  result = k_val * (_item.beta  - _item.alpha) + c_val;
      item_c::MUL_A_B :  result = k_val * (_item.alpha * _item.beta) + c_val;
      item_c::DIV_A_B :  result = k_val * (_item.alpha / _item.beta) + c_val;
      item_c::DIV_B_A :  result = k_val * (_item.beta  / _item.alpha) + c_val;
      item_c::INC_A   :  result = k_val * (_item.alpha + 1) + c_val;
      item_c::INC_B   :  result = k_val * (_item.beta  + 1) + c_val;
      item_c::CLR_RES :  result = 0;
      item_c::ACCUM   :  result += _item.alpha;
   endcase

   `cn_dbg(30, ("Calculated result %08X on item: %s", result, _item.convert2string()))
endfunction : write_item
```

Also note that when this function runs, it will fetch the predicted values based on the most recently completed CSR write. If a field is marked as being volatile, it means that it may change values between accesses (like a status register or a statistics counter).

Register Callbacks

There are many scenarios where you would want the environment to be alerted whenever a read or a write occurs on a CSR. Here are just a few:

- Aliased registers. When one register is written, another register takes a new value.
- Expecting interrupts. When a CSR is written, the write may conditionally cause an interrupt that needs to be expected.
- Soft resets. Writing a CSR causes a software reset, and some agents and predictors must be reset, too.
- Writes to the registers on-the-fly cause predictions to shift accordingly.

For cases such as these, you want to use a register callback. Register callbacks follow the same pattern that other callbacks in UVM follow:

1. Derive a class from `uvm_reg_cbs`.
2. Fill in the tasks and/or functions that supply the code you want to run.

3. Instantiate your callback class somewhere in the environment.

4. Register your callback class on the CSR(s) that it should apply to.

You may register multiple callback classes or multiple instances of a callback class on a given CSR. The following is an example of a callback that prints the read value anytime the `RESULT[SOR]` field is read from.

75. verif/vkits/alu/alu_result_reg_cb.sv

```
// class: alu_result_reg_cb_c
class alu_result_reg_cb_c extends uvm_reg_cbs;
   `uvm_object_utils(alu_result_reg_cb_c)

   //------------------------------------------------------------------------------------------------------------
   // Group: Methods
   function new(string name="alu_result_reg_cb");
      super.new(name);
   endfunction : new

   ////////////////////////////////////////////
   // func: post_read
   // Print to the logfile anytime the RESULT CSR is read from.
   virtual task post_read(uvm_reg_item rw);
      `cn_info(("Read from RESULT: %08X", rw.value[0]));
   endtask : post_read
endclass : alu_result_reg_cb_c
```

Then, the callback class is created and registered with the `RESULT` CSR any place and at any time in the environment.

76.

```
            alu_result_reg_cb_c alu_result_reg_cb;
            alu_result_reg_cb = alu_result_reg_cb_c::type_id::create("alu_result_reg_cb");
            uvm_reg_cb::add(reg_block.RESULT, alu_result_reg_cb);
```

Built-In Register Sequences

For free, UVM's register package includes a variety of register test sequences to completely exercise the CSR and memory space. These sequences can be the basis of your very first tests of a block to ensure that all CSR bugs are discovered and fixed. What's more, the sequences will automatically adapt to changes to the CSRs over time without any need to be modified.

The available sequences test the DUT's ability to withstand resets, bit-bashing, shared accesses (registers that can be accessed by more than one interface), memory walking patterns, and back-door HDL accesses.

Each sequence can be configured via the UVM resource database to turn on or turn off testing of specific CSRs or blocks.

Register Accesses

The register model holds 3 different values for each register field: the desired value, the mirrored value, and the reset value. The desired value is the value that will be written the next time the register's `update` method is run. The mirror reflects the value that the environment expects the register to be currently holding (this value obeys various access policies that a register can have, such as Write-1-to-Clear).

The following table summarizes the most commonly needed methods for accessing the values of registers and their fields.

What You Want to Do	Code That Does It
Write a new value to a register	`reg_block.REG.write(status, new_value);`
Write a new value to a field (unless it already has that value)	`reg_block.REG.FIELD.set(new_value);` `reg_block.REG.update(status);`
Write a value to a field (regardless of its current contents)	`reg_block.REG.FIELD.write(status,` `new_value);`
View the *desired* value of a register field	`future_value = reg_block.REG.FIELD.get();`

Constrain a register field before randomization	`constraint my_cnstr {`
	`reg_block.REG.FIELD.value == my_val;`
	`}`

Read from a register, update the mirrored value, and get the actual value that was read	`reg_block.REG.read(status, value);`
Read a register and view the mirrored contents of a field	`reg_block.REG.mirror(status);` `reg_block.REG.FIELD.value;`

Table 7: *Common Register Methods*

Conclusion

UVM's register package offers significantly more functionality than what is presented here. UVM also has backdoor accesses, FIFO-like CSRs, memories, indirect registers, and more. But the information presented in this lesson should provide a baseline for future lessons where these CSRs will be used.

Chapter 13: CFG Classes and Writing Tests

When an environment is written properly, writing tests to hit specific regions of the design or areas of concern is merely a matter of adjusting a few variables, tweaking a few configurations, or creating a few factory overrides. The base test instantiates everything, but all of the other tests should be easy to write.

In an environment, there are two types of configurations: testbench build configurations and random knob configurations.

Testbench Build Configurations

Many components have configuration fields that can be set by higher-level components. These configuration fields are determined and set (using `uvm_config_db::set` calls) at time zero and are used to control the build process.

Random Knob Configurations

These are set at time zero and other randomized values are generated dynamically during the simulation based on their values. They are used to control stimulus or component behaviors.

We have been using the first type since Chapter 11. When we told the predictor to act as a reference model, we used this call:

```
uvm_config_db#(int)::set(this, "alutb_env.alu_agent", "dev_ref_model", 1);
```

`dev_ref_model` is a configuration field of an ALU agent that trickles down to the same configuration field in its predictor.

Since this model is so familiar by now we're not going to spend the rest of this chapter discussing it, but suffice to say testbench build configurations are one way that you can write a test that manipulates the environment to help achieve the test's goals.

The rest of this chapter will discuss random knob configurations.

Cfg Classes

There are many places in a vkit where you can put random variables. Transactions, drivers, agents, sequences...all of the classes we've discussed so far can have random variables that change how they behave from one run to the next, and that's part of the problem: if you're writing a predictor that needs to be aware of all of these different values, then the

predictor must have references to each of these components to have visibility into these values.

The solution is that a vkit defines a *cfg* class which contains all of its random settings. Each component within that vkit needing access holds a handle to it. The cfg class is created and randomized by the base test before the environment is built so that the random configurations can subsequently influence the testbench configuration.

The base test doles out the handles to the various cfg classes to all applicable sub-components using `uvm_config_db#(uvm_object)::set` calls. The cfg class also contains handles to any register blocks associated with the vkit, so that CSR configurations and other random environment variables can have constraints upon one another. Now the vkit components can access their associated register blocks through their cfg handles.

Ted and Fred

Imagine this hypothetical testbench environment:

1. The testbench contains two DUTs, Ted and Fred. It likewise has two environments from different vkits: `ted_env` and `fred_env`.

2. Ted contains a CSR which dictates the depth of Ted's FIFO. This value dictates how many credits Ted can have. The Ted environment contains a common credit agent with a random variable, `max_credits`.

3. Fred needs to know how many credits Ted has to determine which data rate he can be programmed for.

4. The test-writer wants to have some tests run at the highest possible data rate, while others can be left unconstrained.

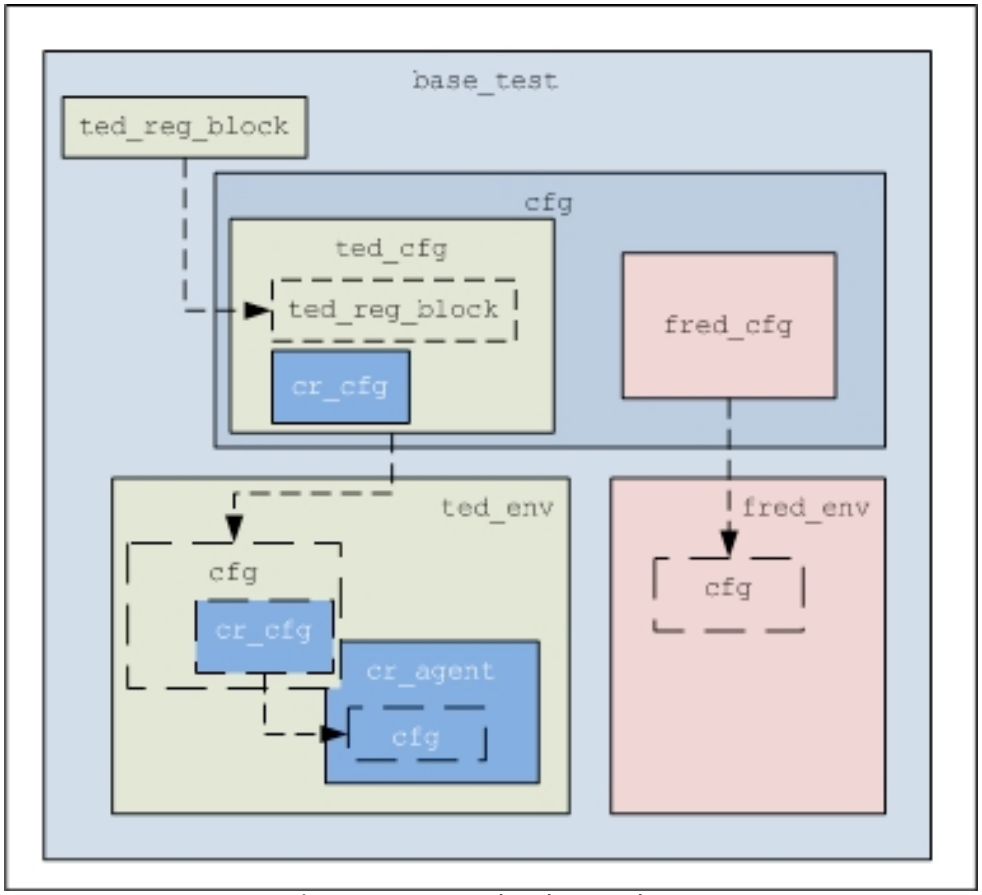


Figure 17: *Hierarchical CFG Classes*

To summarize the order in which all of these objects and components are built:

1. The base test instantiates and creates its own cfg class which in turn creates the underlying `ted_cfg` and `fred_cfg` classes.

2. The base test creates, builds, and configures the Ted's register block.

3. It then assigns the `ted_reg_block` to the handle within the `cfg.ted_cfg` class.

4. The base test then randomizes the cfg class in a virtual function called `randomize_cfg`. Derived tests can change the cfg class constraints, manipulate the cfg class by overriding `randomize_cfg`, or create derived cfg classes and use the factory, just as we did in the `kval` test from Chapter 12. Randomizing the cfg class also randomizes the register blocks.

5. The base test populates the configuration database with handles to the cfg classes, such that the build phases of later components will automatically pull them in.

```
uvm_config_db#(uvm_object)::set(this, "ted_env", "cfg", cfg.ted_cfg);
uvm_config_db#(uvm_object)::set(this, "fred_env", "cfg", cfg.fred_cfg);
uvm_config_db#(uvm_object)::set(this, "ted_env.cr_agent", "cfg", cfg.ted_cfg.cr_cfg);
```

6. The base test finally creates the Ted and Fred environments, as per the environment's architecture and the results of randomizing the cfg class.

UTG's `base_test` template does some of this automation for you by providing hints for where everything should go.

Hierarchical Constraints

Figure 17 showed how the component hierarchy of this hypothetical environment is mirrored by the cfg hierarchy. It is frequently desirable to cascade constraints in a similar manner. In this example, the `ted_cfg` must constrain the CSR containing Ted's FIFO depth to be equal to the credit agent's `max_credits` field. And the testbench must coordinate these credits with the `data_rate` field in `fred_cfg`.

The example below shows how the `ted_cfg` class coordinates the CSR value and its own credit agent's cfg class.

77. verif/vkits/ted/ted_cfg.sv

```
class cfg_c extends uvm_object;
   `uvm_object_utils_begin(ted_pkg::cfg_c)
      `uvm_field_object(reg_block, UVM_REFERENCE)
      `uvm_field_object(cr_cfg, UVM_REFERENCE)
   `uvm_object_utils_end

   //---------------------------------------------------------------------------------------
   // Group: Fields

   // var: reg_block
   // Register block for this environment
   rand reg_block_c reg_block;

   // var: cr_cfg
   // Credit config class
   rand cr_pkg::cfg_c cr_cfg;

   // constraint: credits_cnstr
   // Ensure that FIFO depth and credits are in sync
   constraint credits_cnstr {
      reg_block.FIFO_CFG.DEPTH.value == cr_cfg.max_credits;
   }

   //---------------------------------------------------------------------------------------
   // Group: Methods
   function new(string name="cfg");
```

The testbench must also ensure coordination between Ted's credits and Fred's data rates, so the following constraint might be placed in the testbench's cfg class:

78.

```
   // constraint: rate_credit_cnstr
   // Coordinate the number of credits with Fred's ability to send
   constraint rate_credit_cnstr {
      ted_cfg.cr_cfg.max_credits < 64              -> fred_cfg.data_rate == fred_pkg::MIN_RATE;
      ted_cfg.cr_cfg.max_credits inside {[65:255]} -> fred_cfg.data_rate == fred_pkg::MED_RATE;
      ted_cfg.cr_cfg.max_credits > 255             -> fred_cfg.data_rate == fred_pkg::MAX_RATE;
   }
```

Finally, the test-writer wants one test that always uses Fred's maximum data rate, so the `max_data_rate_test_c` class might simply contain this constraint:

```
79.
      // constraint: max_data_rate_cnstr
      // Constrain Fred to operate at the highest possible data rate
      constraint max_data_rate_cnstr {
         cfg.fred_cfg.data_rate == fred_pkg::MAX_RATE;
      }
```

Right now, we do not have multiple cfg classes in our simple `alutb` testbench, but we will later when we get to Chapter 15. For now, it's time that we had more control over the K and C values that we've been operating with.

Problem 13-1

The ALUTB package already contains a `cfg_c` class that, admittedly, doesn't do a whole lot just yet. We're going to add some constraints to it to modify the CSR values of K and C during the configuration phase and attempt to hit a wide swath of values.

Remove the `innocuous_cnstr` constraint from `alutb_pkg::cfg_c`. Create a configuration variable in the ALUTB `cfg_c` class that permits values of K and C to be within the following values. By default, constrain the knob to always choose `INNOCUOUS` mode:

Setting	K	C	Frequency
INNOCUOUS	1	0	20%
SMALL	2..5	2..10	50%
LARGE	6..50	11..128	15%
XLARGE	51..255	129..255	5%
UNLIMITED	1..255	0..255	10%

Table 8: *K and C Value Distribution Constraints*

Then, write a test called `exer_test_c` that varies the knob settings according to the frequencies shown above. You should find that it passes consistently.

You will also need to fix the `kval_test_c` reference to `innocuous_cnstr`.

Solution

After removing the ALU base constraints, you start by creating an enumerated type that represents the knob. By placing this type in the configuration class, you do not pollute the namespace of the environment. This should go at the top of the `cfg_c` class:

80. verif/vkits/alutb/alutb_cfg.sv

```
class cfg_c extends uvm_object;
   //---------------------------------------------------------------------------------------
   // Group: Types

   // enum: alu_const_knob_e
   // Used to constrain the K and C values for ALU
   typedef enum { INNOCUOUS, SMALL, LARGE, XLARGE, UNLIMITED } alu_const_knob_e;
```

Alternatively, you could place this in a file that contain's all of the types for this vkit.

Then, you create a random instance of your new enumerated type, and constrain it to the value `INNOCUOUS`.

81. verif/vkits/alutb/alutb_cfg.sv

```
   `uvm_object_utils_begin(alutb_pkg::cfg_c)
      `uvm_field_object(reg_block, UVM_REFERENCE)
      `uvm_field_enum(alu_const_knob_e, alu_const_knob, UVM_ALL_ON)
   `uvm_object_utils_end

   //---------------------------------------------------------------------------------------
   // Group: Configuration Fields

   // field: reg_block
   // Register block for this environment
   rand alu_csr_pkg::reg_block_c reg_block;

   // field: alu_const_knob
   // Constrains the K_VAL and C_VAL
   rand alu_const_knob_e alu_const_knob;
   constraint const_knob_cnstr { alu_const_knob == INNOCUOUS; }
```

Note the syntax of the `` `uvm_field_enum `` macro requires that you provide the type as the first argument. Now add implication constraints to K and C that match the table:

82. verif/vkits/alutb/alutb_cfg.sv

```
   // constrain K_VAL based on alu_const_knob
   constraint kval_cnstr {
      alu_const_knob == INNOCUOUS -> (reg_block.CONST.K_VAL.value == 1);
      alu_const_knob == SMALL     -> (reg_block.CONST.K_VAL.value inside {[2:5]});
      alu_const_knob == LARGE     -> (reg_block.CONST.K_VAL.value inside {[6:50]});
      alu_const_knob == XLARGE    -> (reg_block.CONST.K_VAL.value inside {[51:255]});
      alu_const_knob == UNLIMITED -> (reg_block.CONST.K_VAL.value inside {[0:255]});
   }

   // constrain C_VAL based on alu_const_knob
   constraint cval_cnstr {
      alu_const_knob == INNOCUOUS -> (reg_block.CONST.C_VAL.value == 0);
      alu_const_knob == SMALL     -> (reg_block.CONST.C_VAL.value inside {[2:10]});
      alu_const_knob == LARGE     -> (reg_block.CONST.C_VAL.value inside {[11:128]});
      alu_const_knob == XLARGE    -> (reg_block.CONST.C_VAL.value inside {[129:255]});
      alu_const_knob == UNLIMITED -> (reg_block.CONST.C_VAL.value inside {[0:255]});
   }
```

The exer_test_c class uses a distribution constraint to randomize the knob. Notice that you can add a constraint *through* the class instance hierarchy. This is essential. If you couldn't do this in SystemVerilog, the only other way to do it would be by extending the class with another one that contained the constraint, and then overriding that type using the factory. You'll find this method to be a lot less typing.

Here is the complete exer_test_c class:

83. verif/alutb/tests/exer.sv

```
// class: exer_test_c
// Turns on all possible values of K_VAL and C_VAL
class exer_test_c extends basic_test_c;
   `uvm_component_utils(exer_test_c)

   //---------------------------------------------------------------------------------------
   // Group: Configuration Fields
   constraint const_knob_cnstr {
      cfg.alu_const_knob dist { alutb_pkg::cfg_c::INNOCUOUS :/ 20,
                                alutb_pkg::cfg_c::SMALL     :/ 50,
                                alutb_pkg::cfg_c::LARGE     :/ 15,
                                alutb_pkg::cfg_c::XLARGE    :/ 5,
                                alutb_pkg::cfg_c::UNLIMITED :/ 10
      };
   }

   //---------------------------------------------------------------------------------------
   // Group: Methods
   function new(string name="alu_exer",
                uvm_component parent=null);
      super.new(name, parent);
   endfunction : new

   ////////////////////////////////////////////
   // func: randomize_cfg
   // Disable the const_knob_cnstr in cfg
   virtual function void randomize_cfg();
      cfg.const_knob_cnstr.constraint_mode(0);
      super.randomize_cfg();
   endfunction : randomize_cfg
endclass : exer_test_c
```

The exer test now creates its own const_knob_cnstr constraint and turns off the one in the cfg class. Because the enumerated type is local to the scope of the configuration class, (which is local to the scope of the ALUTB package), you need to use the scoping operator twice.

Collecting Functional Coverage

Now, a short interlude on running exercisers and collecting functional coverage. Of course, we write constrained random tests so that the constraint solver can find interesting scenarios for us, and it is important to know that your tests are doing what you think they are doing. To do this, it is important that we run as many tests as possible. The tool for this is `srand`.

Try running the following:

```
84.
verif/alutb> srand -m 20 -c 100 -a SVFCOV=8 --nopbs tests/exer.sv
```

This command will run your `exer` test 100 times, with 20 being sent to the simulation farm in parallel, and it will collect functional coverage on CSR values.

When it's complete, you can run the `covmrg` to merge the simulation results and inspect the functional coverage that you collected:

```
verif/alutb> covmrg
```

Test Overrides

Sometimes you want to create a sequence for a semi-directed test case. Perhaps you want to create an error case that you don't want intermingling with your other sequences. Or you want your driver to behave differently for a single test.

Once again, this is where the factory comes in. It is common practice to create a derived class of something else in the environment and place it in the test file, then use a factory override for that test only.

Problem 13-2

Create a new test, `div0_test_c`, that sets the denominator of a divide operation to zero 50% of the time. Do this by overriding the `alu_pkg::item_c` with a new transaction type, `div0_item_c`.

Also, have the basic test's main phase also launch an instance of `exer_seq_c`. See what happens.

Solution

Here is one solution to the problem:

85. verif/alutb/tests/div0.sv

```
`include "basic.sv"

//------------------------------------------------------------------------------
// class: div0_item_c
// Causes a divide-by-zero on 50% of all divide operations
class div0_item_c extends alu_pkg::item_c;
   `uvm_object_utils(div0_item_c)

   constraint protocol_cnstr {
      (operation == DIV_A_B) -> beta dist { 0 :/ 50,
                                            [1:'hffff] :/ 50};
      (operation == DIV_B_A) -> alpha dist { 0 :/ 50,
                                            [1:'hffff] :/ 50};
      (operation == SUB_A_B) -> alpha > beta;
      (operation == SUB_B_A) -> beta > alpha;
   }

   //---------------------------------------------------------------------------
   // Group: Methods
   function new(string name="div0_item");
      super.new(name);
   endfunction : new
endclass : div0_item_c

//------------------------------------------------------------------------------
// class: div0_test_c
// Uses the div0_item_c class instead
class div0_test_c extends basic_test_c;
   `uvm_component_utils(div0_test_c)

   //---------------------------------------------------------------------------
   // Group: Methods
   function new(string name="div0_test",
                uvm_component parent=null);
      super.new(name, parent);
   endfunction : new

   ////////////////////////////////////////////
   // func: build_phase
   virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      set_type_override_by_type(alu_pkg::item_c::get_type(), div0_item_c::get_type());
   endfunction : build_phase

endclass : div0_test_c
```

Some notes:

```
   constraint protocol_cnstr {
      (operation == DIV_A_B) -> beta dist { 0 :/ 50,
                                            [1:'hffff] :/ 50};
      (operation == DIV_B_A) -> alpha dist { 0 :/ 50,
                                            [1:'hffff] :/ 50};
      (operation == SUB_A_B) -> alpha > beta;
      (operation == SUB_B_A) -> beta > alpha;
   }
```

The constraint `protocol_cnstr` needs to have the same name as the base class. **Constraints act like virtual methods**: if the derived class has one that is the same name, it

takes precedence. This changes the constraints to have a zero in the denominator 50% of the time for division operations. Note how we've used the distribution constraint on top of an implication constraint. It was also necessary to put the subtraction constraints in because we cannot partially override a constraint block.

And finally, the factory override method should now be familiar to you:

```
set_type_override_by_type(alu_pkg::item_c::get_type(), div0_item_c::get_type());
```

If your first inclination was to only perform the factory override 50% of the time, there's really no effective way to do that. Using constraints and overriding all transactions is the way to go.

If you did everything correctly your test may PASS, but you should find that the RTL puts X's out as the result. Congratulations! You've found your first bug. The designer agrees and decides to use a denominator of one whenever a zero is seen instead.

You can patch the RTL with the `utut` script, like this:

```
verif/alutb> utut fix 1
```

Question

If you wrote your predictor's checker in the same way as the solution from Chapter 11, there's a problem in there someplace. Can you find it and fix it?

Hint:

Look in the logfile. It is a problem that also plagues designers.

Answer

Only if you looked at your logfile would you likely find the problem. It's that the predictor we wrote also doesn't have the fix and is calculating the result to be X, but the result is no longer X and yet the test still passes. What's going on here?

There are actually two problems. The first is that the predictor is predicting X, and the second is that it is not flagging an error.

If you recall, our comparison took place in the TLM `imp` function, `write_result()`:

```
    virtual function void write_result(result_t _result);
       if(_result != result)
          `cn_err(("Actual result: %08X != Expected result: %08x",
                   _result, result))
    endfunction : write_result
```

Here, we've used an `if` statement, and one of the operands is an X. Just as in RTL, verification people have to beware that the `if` statement does not propagate an X. The above test will not succeed, and no error gets printed.

Of course, we need to modify the calculations to now match the RTL, but we want to make sure that we don't miss this issue again next time.

To fix this you could use the triple-equals construct:

86. **verif/vkits/alu/alu_pred.sv**

```
       if(_result !== result)
          `cn_err(("Actual result: %08X != Expected result: %08x",
                   _result, result))
```

Or, you could use the `assert` statement which does not have the same affliction as the `if` statement:

87. **verif/vkits/alu/alu_pred.sv**

```
       assert(_result == result) else
          `cn_err(("Actual result: %08X != Expected result: %08x",
                   _result, result))
```

And to fix the ALU predictor's computation of the correct result, you would change the two division cases as follows:

88. **verif/vkits/alu/alu_pred.sv**

```
          item_c::DIV_A_B :  result = k_val * (_item.alpha / (_item.beta? _item.beta : 1)) + c_val;
          item_c::DIV_B_A :  result = k_val * (_item.beta  / (_item.alpha? _item.alpha : 1)) + c_val;
```

Conclusion

While the rest of our jobs may be overly complicated, *writing tests should be kept simple*. Each vkit should have a policy class called `cfg`, which may or may not contain other cfg classes. The highest-level cfg class is instantiated in the base test. Each cfg instance is distributed to its components through the configuration database.

Tests can be created with the template generator and extended from the base test, or any other test. Tests merely tweak some configurations and knobs, or perform a factory override or two.

Thousands of test runs are used to hit corner cases using `srand`, and functional coverage should be analyzed frequently with `covmrg`.

Chapter 14: Advanced Sequences I

Sequences and sequencers are so involved, they deserve three separate chapters! This chapter will discuss how to better launch sequences to get a more random mix of traffic, how we can use the resource database to configure sequences, and what a persistent sequence is.

Default Sequences

So far we've seen two methods that we can use to launch sequences onto a sequencer. From a component, we can create a sequence and call its start task, with a reference to the sequencer as an argument. From another sequence, we can have it launch a different sequence by calling one of the `uvm_do macros.

A third way that you can launch sequences is by calling the sequencer's task execute_item() with an instance of the sequence. This isn't a whole lot different from calling the sequence's start task, so there's not much gain here.

There's a fourth way as well. When any of a sequencer's run-time phases is started, its function start_phase_sequence() is run. If there is a configuration item in the database called "default_sequence" for this sequencer's phase, then that sequence is run automatically.

For example, to automatically launch a sum_array_seq_c during the main phase, you would put this in your test's build phase:

```
uvm_config_db#(uvm_object_wrapper)::set(this, "alutb_env.alu_agent.sqr.main_phase",
                                        "default_sequence", alu_pkg::sum_array_seq_c::type_id::get());
```

This will create, randomize, and launch a sequence of this type for you when the main phase starts. If you want a very specific instance of a sequence to run, then the call is slightly different:

```
alu_pkg::sum_of_factorials_seq_c sof_seq = new("sof");
sof_seq.randomize() with {op_x == 1; op_y == 5;};
uvm_config_db #(uvm_sequence_base)::set(this, "alutb_env.alu_agent.sqr.main_phase",
                                        "default_sequence", sof_seq);
```

Here, notice that we did not just 'set' op_x and op_y. We needed to randomize it for this to work, because UVM will randomize it for us if it hasn't already been randomized, which would then squash our settings.

Regardless of the method you choose, you're probably wondering where all the benefit is. Sure, it allows you to use the configuration database to start these sequences, but it only allows us to launch one sequence per phase. What good is that?

Well, if that sequence is one that happens to launch many other sequences, then it can be used to concisely do a whole lot more than what our basic test's `main_phase` currently does. That's where library sequences come into play.

Library Sequences

Before we get too far into library sequences, it should be mentioned that these are new to UVM 1.1 and are not considered "production-level". As such, they are also not particularly well documented yet, either.

A library sequence is a sequence that picks a random sequence, launches it, then picks another, and another, and another, until a specified count is reached. This is an ideal method for creating highly random stimulus as it allows you to create a new sequence to test some feature, write a test that focuses on that sequence, and then add it to one or more library sequences to see how well it works with others.

You might recall that when you created `alu_seq_lib.sv` with `utg`, it created a class called `lib_seq_c`, and we had you comment it out. Those few lines of code are a library sequence. You can create as many library sequences as you wish and each library can hold as many sequences as you wish. To place a sequence into a library, you use the macro `` `uvm_add_to_seq_lib ``. For example, to add our factorial sequence to the `lib_seq_c` library, you would add the following to the `factorial_seq_c` class:

```
89.     verif/vkits/alu/alu_seq_lib.sv
class factorial_seq_c extends uvm_sequence #(item_c, item_c);
   `uvm_object_utils_begin(alu_pkg::factorial_seq_c)
      `uvm_field_int(operand, UVM_ALL_ON)
      `uvm_field_int(result,  UVM_ALL_ON)
   `uvm_object_utils_end
   `uvm_add_to_seq_lib(factorial_seq_c, lib_seq_c);
```

This macro must specify the type of the sequence and which library you want it to go into. A sequence can call this macro more than once to be placed into different libraries. You can create different library sequences to mix different types of sequences. Your test can then set a library sequence as the default sequence for a particular phase, such as the main phase.

Problem 14-1

Add all of the sequences you've created so far to the `lib_seq_c` library sequence.

Clear out the main phase of your basic test, and instead launch your new library sequence as the default sequence for the main phase. Remember to uncomment the `lib_seq_c`.

Solution

To launch the sequences from your basic test, you should have altered its build phase as follows.

90. verif/alutb/tests/basic.sv

```
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    uvm_config_db#(uvm_object_wrapper)::set(this, "alutb_env.alu_agent.sqr.main_phase",
                                            "default_sequence", alu_pkg::lib_seq_c::type_id::get());
  endfunction : build_phase
```

You would also either need to move the library sequence to the top of the file, or create a forward-declaration of the library sequence:

91. verif/vkits/alu/alu_seq_lib.sv

```
typedef class lib_seq_c;
```

When you run this test, you should see 10 different random sequences chosen. Of course, most of those sequences spawn other sequences and sequence items, so a lot more traffic will be seen.

Now all of your tests that derive from `basic_test_c` will run your library sequence by default.

Library Sequence Configuration

Our library sequence only runs 10 random sequences, and it is always random. Maybe you want a more interesting pattern to choose from? Once again, UVM uses the policy class pattern to help out.

The `uvm_sequence_library_cfg` class is the policy class that you can use to customize your library sequence. It offers three different options: `selection_mode`, `min_random_count`, and `max_random_count`.

The min and max random count options do just what you think. When the sequence starts, it chooses a random number between these two numbers, and then sends that many sequences.

The selection mode is an enumerated type that offers you four choices as to how sequences will be chosen:

uvm_sequence_lib_mode	Method
UVM_SEQ_LIB_RAND	Select randomly (default)
UVM_SEQ_LIB_RANDC	Random cyclic selection
UVM_SEQ_LIB_ITEM	Send only items. Does not send sequences.
UVM_SEQ_LIB_USER	Applies a user-defined selection algorithm.

Table 9: *Library Sequence Selection Modes*

Random cyclic is a term we haven't used so far in this tutorial. This term tells SystemVerilog to choose randomly from a list of items, but don't choose an item again until the list has been exhausted.

Sending only sequence items (`UVM_SEQ_LIB_ITEM`) is a very useful scenario. Essentially, the library sequence will just send random transaction items, ignoring whatever sequences you put into it. Sequence items, though, get responses. And unfortunately, if you choose this selection mode, you will find that the library sequence's response queue fills up. To manage this, you would need to implement a response handler function that was described in Chapter 9.

The user-defined algorithm has you create a function (`select_sequence`) that returns an unsigned integer between zero and its argument, `max`. At your disposal is the member field `sequences`, which is a queue of all the different sequences in the library, stored as `uvm_object_wrapper` classes. Calling `get_type_name()` on each member gives the sequence's name as a string. The following example shows how you might distribute sequences in a library unevenly. The function `select_sequence` returns an index into the `sequences` queue that specifies which sequence to choose next.

92. verif/vkits/alu/alu_seq_lib.sv

```
class lib_seq_c extends uvm_sequence_library #(item_c);
   `uvm_object_utils(alu_pkg::lib_seq_c)
   `uvm_sequence_library_utils(lib_seq_c)

   //------------------------------------------------------------------------------
   // Group: Fields

   // field: selector
   // Distributes sequence selection unevenly when UVM_SEQ_LIB_USER is used
   rand int unsigned selector;
   constraint selector_cnstr { selector dist { 0 :/ 40, 1 :/ 20, 2 :/ 10, 3 :/ 30 }; }

   //------------------------------------------------------------------------------
   // Group: Methods
   function new(string name="seq_lib");
      super.new(name);
      init_sequence_library();
   endfunction : new

   ////////////////////////////////////////
   // func: select_sequence
```

```
    virtual function int unsigned select_sequence(int unsigned max);
       randomize(selector);
       return selector;
    endfunction : select_sequence

endclass : lib_seq_c
```

Library sequences offer several more features that allow you to modify the list of sequences dynamically, such as `add_sequence`, `remove_sequence`, and `add_typewide_sequence`, which allows you to add a sequence type to all your library sequences at one time.

To assign an instance of a library sequence configuration class to a library sequence from a test, you push it into the configuration database:

```
93.
    virtual function void build_phase(uvm_phase phase);
       // create the configuration object
       uvm_sequence_library_cfg lib_cfg = uvm_sequence_library_cfg::type_id::create("lib_cfg");

       super.build_phase(phase);

       // run lots of sequences!
       lib_cfg.min_random_count = 2500;
       lib_cfg.max_random_count = 5000;

       uvm_config_db#(uvm_object_wrapper)::set(this, "alutb_env.alu_agent.sqr.main_phase",
                                               "default_sequence", alu_pkg::lib_seq_c::type_id::get());

       // set library configuration
       uvm_config_db#(uvm_sequence_library_cfg)::set(this, "alutb_env.alu_agent.sqr.main_phase",
                                                     "default_sequence.config", lib_cfg);

    endfunction : build_phase
```

Persistent Sequences

A persistent sequence is one that starts during some phase (at time zero, perhaps) and remains active for the remainder of the simulation. Why would you want such a thing?

Well, we know that a sequencer can have multiple active sequences, so it's not as though your persistent sequence is the only thing happening. Typically, your sequence will wait some period of time, or wait for some event to happen, and then will spring into action. Here are a few possible examples of persistent sequences:

- Random Read Sequence: Picks a random register to safely read, reads it, then waits for a random period of time before doing it over again.
- Interrupt Handler: Waits for some interrupt event to occur, locks down the sequencer, and begins the process of discovery, reporting, and clearing of interrupts.
- Sequence Adapter: Let's say that you had one sequencer that was spitting out messages and was feeding requests into your ALU agent for mathematical

operations. The writer of that agent doesn't know about our `item_c` class, so is sending their own sequence items that need to be converted into `item_c`.

The ALU model has a second CSR called `RESULT`. This CSR has a 32-bit read-clear field called `SOR` which contains the sum of all results since it was last read. We're going to add a sequence that reads and clears the `RESULT` register every time we believe that it should exceed `32'h100_000`, and we will check to ensure that it does.

Here's how we'll do it:

1. Pipe all monitored results into the ALU sequencer and put each one in a `uvm_tlm_analysis_fifo`.
2. Run a sequence starting at time zero that pulls from the FIFO and accumulates the expected SOR value.
3. If the expected SOR value exceeds `32'h100_0000`, then read the CSR to clear it.
4. Ensure that the SOR field read a value greater than `32'h100_0000`.
5. Clear the expected SOR value that's been accumulated so far.

There are a whole slew of new concepts above.

The first one is the FIFO, which is a handy TLM construct. It is a parameterized object that can connect directly to any analysis `port`, and offers a `get` task to let you pull objects out one at a time. It also has zero-time functions `try_put` and `try_get`.

The next concept is having the sequence pull from its sequencer's FIFO. How can a sequence get a handle to a FIFO in the sequencer? Better yet: how will it read from a CSR?

The answer is sequencer references.

Sequencer References

The `uvm_sequence` base class comes built-in with a reference back to the sequencer it is running on. This reference is called `m_sequencer`. Unfortunately, because it is declared in the base class as being of the type `uvm_sequencer_base`, it's not very useful by itself. After all, the base class doesn't have a FIFO of results.

But, we could create our own reference and cast the `m_sequencer` back to the `alu_pkg::sqr_c` that we know it to be, like this:

```
    sqr_c my_sqr;
    $cast(my_sqr, m_sequencer);
```

It turns out that this is a pretty common thing to do, so UVM automated it. We have written plenty of sequences so far that haven't needed to reference the sequencer, so it is an optional macro that you add to the class:

```
    `uvm_declare_p_sequencer(sqr_c)
```

This macro call creates a reference, `p_sequencer`, that points to our sequencer and knows what kind it is.

Previously, all of our sequences were detached from the component hierarchy. If we wanted to interact with any other components in the environment, we were out of luck. But the **`p_sequencer` handle *anchors* our sequence into the hierarchy**. This allows us to do *lots* of nifty new things with sequences.

In addition to the tiny bit of overhead, the only drawback to declaring a `p_sequencer` handle with the macro is that now our sequence can only be run on this specific sequencer type.

Sequences can be run on different sequencers?!?! Yes, they can. And they often do. That's a topic we'll explore some more in Chapter 16. Meantime, let's see how this would all fit together.

Problem 14-2

Creating a persistent sequence to handle the sum of results register is somewhat complicated at first, so we'll do it in stages.

Add a `uvm_tlm_analysis_fifo` to your sequencer that accepts items of type `result_t`. Connect it to the analysis `port` from the monitor via an analysis `export`.

Your agent will now look like this (unless it is in device reference model mode):

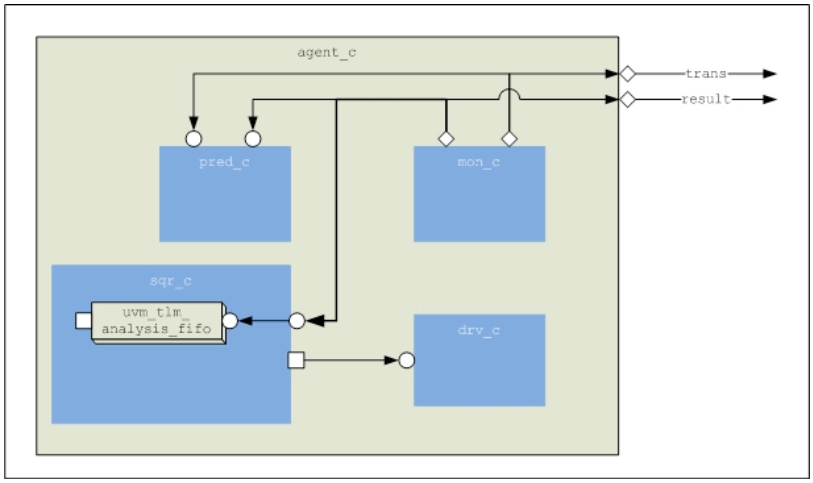


Figure 18: *Instantiating an Analysis FIFO*

Solution

An analysis FIFO receives items just like an analysis `imp`, but you do not have to create a `write` function to receive them. Instead, any running task can receive the items at its leisure. To add the FIFO to the sequencer:

94. verif/vkits/alu/alu_sqr.sv

```
//------------------------------------------------------------------------------
// Group: TLM Ports

// field: monitored_result_exp
// An export to connect to the FIFO
uvm_analysis_export#(result_t) monitored_result_exp;

//------------------------------------------------------------------------------
// Group: Fields

// field: monitored_result_fifo
// A FIFO that holds monitored results
uvm_tlm_analysis_fifo #(result_t) monitored_result_fifo;
```

Notice that we created an `export` as the exposed user interface to outside classes, and declared the FIFO as one of this component's fields. The `export` is the exposed socket that users plug into, but the FIFO is what is behind the wall--an implementation detail that can change at any time.

As with any other component you need to `new()` your FIFOs and `exports`.

95. verif/vkits/alu/alu_sqr.sv

```
virtual function void build_phase(uvm_phase phase);
   super.build_phase(phase);
   monitored_result_exp = new("monitored_result_exp", this);
   monitored_result_fifo = new("monitored_result_fifo", this);
endfunction : build_phase
```

Finally, connect the `export` to the FIFO's `analysis_export`:

96. verif/vkits/alu/alu_sqr.sv

```
virtual function void connect_phase(uvm_phase phase);
   super.connect_phase(phase);
   monitored_result_exp.connect(monitored_result_fifo.analysis_export);
endfunction : connect_phase
```

Connecting to the monitor's `port` in the agent should also be straightforward:

97. verif/vkits/alu/alu_agent.sv

```
if(!dev_ref_model) begin
   if(is_active) begin
      drv.seq_item_port.connect(sqr.seq_item_export);
      mon.monitored_result_port.connect(sqr.monitored_result_exp);
   end
```

Within the agent, you could connect the monitor's `port` directly to the sequencer's FIFO, and skip the `analysis_export` altogether. But it is considered more programmer-friendly to hide the implementation details within the sequencer and expose only an `export`.

Handling Cyclical Dependencies

In the next problem, you're going to write a sequence that declares its sequencer using `uvm_declare_p_sequencer. We're also going to start it from that very sequencer. This will create a cyclical dependency, because they both must know about each other.

Fortunately, SystemVerilog supports forward references and late binding. Simply adding a forward declaration of the sequencer above your sequences in alu_seq_lib.sv tells the compiler: "It's ok, I know what I'm doing here."

98. verif/vkits/alu/alu_seq_lib.sv

```
// Forward declaration of sequencer
typedef class sqr_c;
```

Now the compiler sees the `uvm_declare_p_sequencer(sqr_c) and knows that sqr_c refers to some class, which will later be declared.

Problem 14-3

Let's write our persistent sequence, sor_clr_seq_c. It should constantly try to pull results from its sequencer's FIFO and accumulate them. Hold off on actually performing the CSR read for now. When the sum reaches or exceeds the value 'h100_0000, announce with a print that it is time to read the CSR. Then clear your accumulated value.

Have the sequencer itself start this sequence. Run the basic test to ensure that everything is ship-shape.

Solution

Here is the sequence that does the job:

99. verif/vkits/alu/alu_seq_lib.sv

```
class sor_clr_seq_c extends uvm_sequence#(item_c);
   `uvm_object_utils(alu_pkg::sor_clr_seq_c)
   `uvm_declare_p_sequencer(sqr_c)

   //------------------------------------------------------------------------------------------
   // Group: Methods
   function new(string name="sor_clr_seq");
      super.new(name);
   endfunction : new

   ////////////////////////////////////////
   // func: body
   virtual task body();
      result_t accum_result = 0;
      result_t new_result;

      forever begin
         // get the next monitored transaction and add it to the accumulated results
         p_sequencer.monitored_result_fifo.get(new_result);
         accum_result += new_result;
         `cn_dbg(30, ("accum_result = %8X", accum_result))

         if(accum_result >= 32'h100_0000) begin
            `cn_info(("Time to read the RESULT CSR!"))
            // clear out accum_result
            accum_result = 0;
         end
      end
   endtask : body
endclass : sor_clr_seq_c
```

The solution declares the `p_sequencer` handle using the macro. It runs in a `forever` loop and calls `get` to fetch the latest result from the sequencer's FIFO, using our `p_sequencer` handle.

And here is how the sequence is launched from the sequencer's `run_phase`:

100. verif/vkits/alu/alu_sqr.sv

```
   virtual task run_phase(uvm_phase phase);
      sor_clr_seq_c sor_clr_seq = sor_clr_seq_c::type_id::create("sor_clr_seq");
      sor_clr_seq.start(this);
   endtask : run_phase
```

When you start a sequence, you have to give it a handle to the sequencer that it is supposed to run on, which just like in C++ is represented by the `this` field.

Problem 14-4

Now replace your print statement with the actual read of the CSR. Follow that up with a check that ensures that the sum actually does exceed `'h100_0000`. How do you go about accessing the CSR from a sequence?

Hint:

Declare either cfg or `reg_block` in the sequencer.

Solution

If you did all of that correctly, it's entirely possible that your test will fail. We'll discuss why it might fail and what can be done about it soon, but first a solution.

The CSR register blocks and files are all within the component hierarchy. So to use these, the sequence will have to reach down into its own sequencer via the `p_sequencer` handle. We were able to get the register block in the predictor, so we should be able to do that here, too. Start by declaring the register block in the sequencer:

101. verif/vkits/alu/alu_sqr.sv

```
// field: reg_block
// Auto-generated Register block
alu_csr_pkg::reg_block_c reg_block;
```

Declare it as a field so that it will be auto-populated during the build phase:

102. verif/vkits/alu/alu_sqr.sv

```
`uvm_field_object(reg_block, UVM_REFERENCE)
```

Now that the register block is in the sequencer the sequence can easily access it. Here is what was done to its `body` task. All of the changes are in bold:

103. verif/vkits/alu/alu_seq_lib.sv

```
virtual task body();
   result_t accum_result = 0;
   result_t new_result;
   uvm_status_e status;
   uvm_reg result_reg = p_sequencer.reg_block.RESULT;
   uvm_reg_data_t sor_value;

   forever begin
      // get the next monitored transaction and add it to the accumulated results
      p_sequencer.monitored_result_fifo.get(new_result);
      accum_result += new_result;
      `cn_dbg(30, ("accum_result = %8X", accum_result))

      if(accum_result >= 32'h100_0000) begin
         result_reg.read(status, sor_value);

         // ensure status was ok
         if(status == UVM_NOT_OK) begin
            `cn_err(("Unable to read from RESULT register."))
         end else begin
            // ensure that current value exceeds 'h100_0000
            `cn_dbg(30, ("SOR read as %08X", sor_value))
            if(sor_value < 'h100_0000)
               `cn_err(("Read from RESULT[SOR] but its value was %08X", sor_value))

            // clear out accum_result
            accum_result = 0;
         end
      end
   end
endtask : body
```

We've been using them all along, but it's worth repeating: SystemVerilog uses implicit handles all over the place. This bit of code doesn't create anything new, it just assigns a handle to the register:

```
        uvm_reg result_reg = p_sequencer.reg_block.RESULT;
```

If instead of calling `read` you called `mirror` and then look at the CSR's value, you would have seen a value of zero, which is not the value that was read. This is because the `mirror` task performs the read, then updates its expected value of the CSR. Because it's declared as read-clear ("RC"), it will now be zero.

Calling `read`, though, actually gives you the value that was read. And, it is always safe to check that the status of the read was OK.

Now if you did all of this correctly your test may still fail—under the right circumstances—with an error like this one:

```
%E-(_alu_seq_lib.sv:___312)_[alutb_env.alu_agent.sqr.sor_clr_seq_]_{__1008ns} Read from RESULT[SOR]
but its value was 000215b3
```

How is it that we did not read out something greater than `'h100_0000`?

The answer lies in the ordering of sequences and the way the RTL was written. We submitted a read to the sequencer that handles CTX transactions, but these take a certain amount of time to occur. In the meantime, other ALU traffic is going on. If a read happens in the exact same cycle that a new result occurs, then the SOR field will not go to zero, but will go to the value of the new result, which is what you would want in an accumulating CSR. Because the reads take an indeterminate amount of time, multiple results may have been put into the FIFO before the first read completes, and has cleared out the CSR value. Then, your sequence reads all of these FIFO entries in zero time and quickly accumulates beyond the threshold, but is now out of sync with the real value.

The solution to this problem is to stop the sequencer from issuing new ALU traffic while a CTX read is happening. Previously, we had called `lock()` and `unlock()` to give our sequence exclusive access. But, locking a sequencer actually isn't even fast enough, because lock can be preempted by other traffic—potentially *lots* of other traffic. When our sequence needs to read from the CSR, it needs to do so before any more ALU requests go through. The `grab()` and `ungrab()` functions will do the trick, because these are prioritized over all other traffic.

Also, because the read consumes time, it is possible that any sequences which had launched prior to the `grab()` would now have their results in the results FIFO. These should be cleared out, too. Otherwise our accumulated result will not be correct.

If you add these to your sequence just before and just after the read, you will find that your test now passes. Below is the complete solution:

104. verif/vkits/alu/alu_seq_lib.sv

```
virtual task body();
   result_t accum_result = 0;
   result_t new_result;
   uvm_status_e status;
   uvm_reg result_reg = p_sequencer.reg_block.RESULT;
   uvm_reg_data_t sor_value;

   forever begin
      // get the next monitored transaction and add it to the accumulated results
      p_sequencer.monitored_result_fifo.get(new_result);
      accum_result += new_result;
      `cn_dbg(30, ("accum_result = %8X", accum_result))

      if(accum_result >= 32'h100_0000) begin
         grab();
         result_reg.read(status, sor_value);

         // ensure status was ok
         if(status == UVM_NOT_OK) begin
            `cn_err(("Unable to read from RESULT register."))
         end else begin
            // ensure that current value exceeds 'h100_0000
            `cn_dbg(30, ("SOR read as %08X", sor_value))
            if(sor_value < 'h100_0000)
              `cn_err(("Read from RESULT[SOR] but its value was %08X", sor_value))

            // clear out accum_result
            accum_result = 0;

            // empty out FIFO in case any have been added since the read started
            p_sequencer.monitored_result_fifo.flush();
         end
         ungrab();
      end
   end
endtask : body
```

Sequence Configurations

Let's say that we didn't always want to perform the read on such a specific value as `'h100_0000`. Maybe we want to make our sequence *configurable*.

We can configure components through the configuration database, so why not sequences as well? Well, **sequences don't have a build phase like components do**, so they don't benefit from the magic call to `super.build_phase()` to automatically populate their configurable fields.

You could use the UVM *resource* database and put explicit set and get calls in your code, like this set in the `build_phase` of the basic test:

105. verif/alutb/tests/basic.sv

```
uvm_resource_db#(alu_pkg::result_t)::set("alu_pkg::sor_clr_seq", "trigger_value", 'h300_0000);
```

And then by placing the get call in the `sor_clr_seq_c`:

106. verif/vkits/alu/alu_seq_lib.sv

```
    trigger_value = uvm_resource_db#(result_t)::get_by_name("alu_pkg::sor_clr_seq",
                                                             "trigger_value").read();
```

While this works, these calls are not very attractive. Alternatively, you can have this trigger value be a field in your cfg class and populate that class in the sequencer. Using the cfg class gives you a simple and consistent method for setting variables and configurations for your sequences and provides you with all of the randomization benefits.

Conclusion

We've only really just scratched the surface of what sequences and sequencers are capable of? Wait until they're *virtualized*!

Chapter 15: A New Testbench

It shouldn't surprise you that our ALU module doesn't live all by itself. All block-level testbenches are to be reused in a higher level environment, and the ALU is no different. Fortunately, the ALU subsystem isn't a whole lot more complicated, and most of the work has already been done for you. Integrating your ALU vkit with another vkit in a new testbench is the objective of this chapter.

The ALU Framer

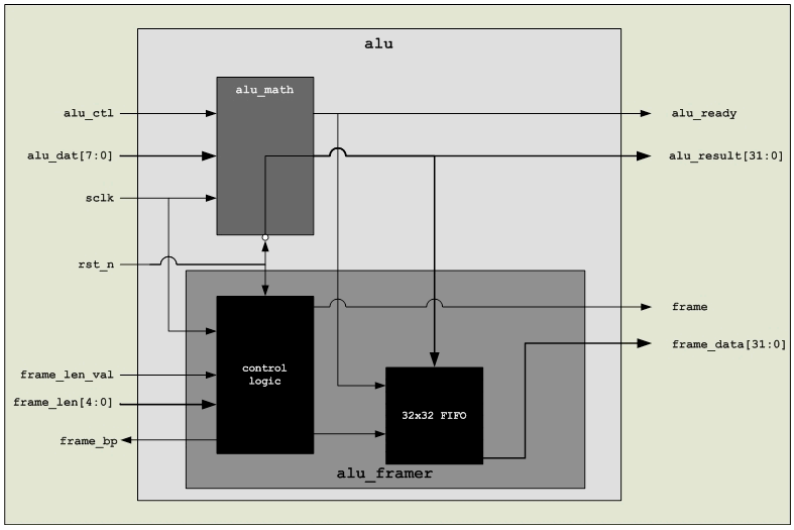


Figure 19: *The Complete ALU Block*

This diagram shows the complete ALU block, which consists of the unit you have been verifying, together with its cousin the `framer` block. ALU results are pushed into the framer's FIFO one at a time. A frame length (`frame_len`, valid on the clock cycle in which `frame_len_val` is high) is specified to the framer. When the depth of the FIFO reaches the frame length, all of the results are pushed out as a complete frame, one 32-bit result word at a time. The framer can only accept one frame length request at a time. Once the frame has been completed, a new frame length can be specified.

The FIFO is 32 quadwords deep, the maximum length of a frame. If it begins to fill up, a `frame_bp` signal will be asserted high, indicating that no more ALU transactions should be sent.

Creating the FRM Testbench

The objective of this chapter is merely to create the new testbench. Fortunately, a script will get you most of the way there. And, the framer vkit is already available to you in `verif/vkits/frm`.

But first, let's see what the environment will look like by the time we're done. We're going to look at it using the topology report that you may have seen at the beginning of your logfiles. You can set the depth of the topology report from `cnmake's TOPO` command-line argument.

Some lines have been deleted for the sake of brevity:

```
----------------------------------------------------------------------------------------------------------
Name                                        Type       Size  Value
----------------------------------------------------------------------------------------------------------
<unnamed>                                 uvm_root               -   @162
   uvm_test_top                           basic_test_c           -   @596
      env                                 frm_pkg::env_c         -   @669
         alu_agent                        alu_pkg::agent_c       -   @738
            drv                           alu_pkg::drv_c         -   @761
            mon                           alu_pkg::mon_c         -   @753
            monitored_result_port         uvm_analysis_port      -   @919
            monitored_item_port           uvm_analysis_port      -   @910
            pred                          pred_c                 -   @948
            sqr                           alu_pkg::sqr_c         -   @787
            item_logger                   vm_component           -   @931
         frm_agent                        frm_pkg::agent_c       -   @712
            drv                           frm_pkg::drv_c         -   @1134
            mon                           frm_pkg::mon_c         -   @1126
            sqr                           frm_pkg::sqr_c         -   @1160
         ctx_agent                        ctx_pkg::agent_c       -   @724
         cfg                            frm_pkg::cfg_c       -   @622
         reg_block                        alutb_pkg::reg_block_c -   @623
      global_env                          global_pkg::env_c      -   @614
         watchdog                         global_pkg::watchdog_c -   @2790
      tb_clk_drv                          clk_drv_c              -   @683
      cfg                               frm_pkg::cfg_c       -   @622
         reg_block                        alutb_pkg::reg_block_c -   @623
      reg_block                           alutb_pkg::reg_block_c -   @623
         CONST                         alu_const_reg_c        -   @629
            C_VAL                         uvm_reg_field          ...  RW CONST[7:0]=8'h2a (Mirror: 8'h00)
            K_VAL                         uvm_reg_field          ...  RW CONST[15:8]=8'h67 (Mirror: 8'h00)
            RSVD0                         uvm_reg_field          ...  RO CONST[63:16]=48'h000000000000
         RESULT                        alu_result_reg_c       -   @634
            SOR                           uvm_reg_field          ...  RC RESULT[31:0]=32'h00000000
            RSVD0                         uvm_reg_field          ...  RO RESULT[63:32]=32'h00000000
         ctx                              uvm_reg_map            -   @625
----------------------------------------------------------------------------------------------------------
```

UVM's handy topology report is an easy way to see the structure of the testbench, how random configurations were chosen, and what the values of all the CSRs will be after configuration.

UVM New Testbench (untb)

Like `utg`, the `untb` script will generate a lot of the boilerplate code for you. It's quite easy to use:

```
verif> untb frm
++ Creating Testbench verif/frm
++ Creating frm_tb_top.sv
++ Creating base_test.sv
++ Creating basic.sv
```

That was simple. Unfortunately, there's still quite a bit of work to do before you'll be able to run it.

The script creates the standard `Makefile`, flists, the top-level testbench, and the base test and basic test. The first thing to do is add the vkits this testbench will need. You'll want to change the `Makefile` so that the `FLISTS` variable is set like this:

107. **verif/frm/Makefile**

```
FLISTS= verif/vkits/cn/cn.flist \
            verif/vkits/global/global.flist \
            verif/vkits/sps/sps.flist \
            verif/vkits/csr_cn/csr_cn.flist \
            verif/vkits/reg/reg.flist \
            verif/vkits/ctx/ctx.flist \
            verif/vkits/alu/alu.flist \
            verif/vkits/frm/frm.flist \
            verif/frm/frm.flist \
            verif/frm/rtl.flist
```

Also, remove the following line from the `Makefile`:

```
TB_XDIRS = ../regs
```

Problem 16-1

Modify the top-level testbench, `frm_tb_top.sv`:

1. Create all interfaces.
2. Copy the `alu_wrapper.sv` file from the `alutb` testbench and modify it to have the frame interface as an input. Wire the frame interface up to the ALU.
3. Remove the lines from `alu_wrapper.sv` that clear the frame signals.
4. Add the `alu_wrapper.sv` path to the `frm.flist`.
5. Copy the `rtl.flist` file from the one in `verif/alutb` since both testbenches will be using the same RTL.

You will also need to make some changes to the base test:

1. The `reg_block` will be the same as the one in the ALU testbench.
2. The CTX interface names must be assigned.
3. The register block must be tied to the CTX register adapter (see the ALUTB base test's `connect_phase`).

The testbench won't do much just yet, but you should be able to compile and run it without any trouble.

Solution

In the top level testbench, you need to instantiate the CTX interface, the ALU interface, and the ALU wrapper. You also need to push the interfaces into the configuration database.

108. verif/frm/frm_tb_top.sv

```
// field: ctx_i
// The <ctx_intf> instance.
ctx_intf ctx_i(.clk(tb_clk), .rst_n(tb_rst_n));

// field: alu_i
// The <alu_intf> instance
alu_intf alu_i(.clk(tb_clk), .rst_n(tb_rst_n));

//------------------------------------------------------------------------------
// Group: DUT
// (Instantiate the DUT and other modules here)
alu_wrapper alu_wrapper(/*AUTOINST*/
                        // Interfaces
                        .ctx_i              (ctx_i),
                        .alu_i              (alu_i),
                        .frm_i              (frm_i),
                        // Inputs
                        .tb_clk             (tb_clk),
                        .tb_rst_n           (tb_rst_n));

//------------------------------------------------------------------------------
// Group: Procedural Blocks
function void pre_run_test();
   `cn_set_intf(virtual cn_clk_intf    , "cn_pkg::clk_intf" , "tb_clk_vi", tb_clk_i);
   `cn_set_intf(virtual cn_rst_intf    , "cn_pkg::rst_intf" , "tb_rst_vi", tb_rst_i);
   `cn_set_intf(virtual ctx_intf.drv_mp, "ctx_pkg::ctx_intf", "ctx_vi"   , ctx_i.drv_mp);
   `cn_set_intf(virtual ctx_intf.mon_mp, "ctx_pkg::ctx_intf", "ctx_vi"   , ctx_i.mon_mp);
   `cn_set_intf(virtual alu_intf.drv_mp, "alu_pkg::alu_intf", "drv_vi"   , alu_i.drv_mp)
   `cn_set_intf(virtual alu_intf.mon_mp, "alu_pkg::alu_intf", "mon_vi"   , alu_i.mon_mp)
   `cn_set_intf(virtual frm_intf.drv_mp, "frm_pkg::frm_intf", "drv_vi"   , frm_i.drv_mp);
   `cn_set_intf(virtual frm_intf.mon_mp, "frm_pkg::frm_intf", "mon_vi"   , frm_i.mon_mp);
endfunction : pre_run_test

// func: Reset Busses
initial begin
   alu_i.reset();
   frm_i.reset();
end
```

In the base test, the register block comes not from the `csr_frm_pkg`, but the `alu_csr_pkg::reg_block_c`. In its build phase, you need to set the interface name for the CTX interface:

109. verif/frm/tests/base_test.sv

```
// set CTX interface name
uvm_config_db#(string)::set(this, "env.ctx_agent",     "intf_name",     "ctx_vi");
```

And in the connect phase, this ties the register block to the CTX environment:

110. verif/frm/tests/base_test.sv

```
if(reg_block.get_parent() == null) begin
   ctx_pkg::reg_adapter_c ctx_adapter =
      ctx_pkg::reg_adapter_c::type_id::create("ctx_reg_adapter", , get_full_name());
   reg_block.csr_map.set_sequencer(env.ctx_agent.sqr, ctx_adapter);
   reg_block.csr_map.set_auto_predict(1);
end
```

Chapter 16: Advanced Sequences II

Virtual sequencers solve the important problem of how to create a sequence that controls more than one agent at a time. We will discuss these as well as more exotic uses of sequences such as adapter sequences.

Virtual Sequences and Sequencers

We know that a sequencer arbitrates among sequences and forwards sequence items—usually to a driver. Virtual sequencers are just a little bit different.

Virtual Sequencer

> A virtual sequencer is one that does not send sequence items. Instead, it holds references to other sequencers and forwards sequences onto those sequencers. Virtual sequencers are not parameterized with a request or response type.

Virtual Sequence

> A virtual sequence is any sequence that is running on a virtual sequencer. It may push sequence items or other sequences, but it always pushes them to a sequencer that is referenced by the virtual sequencer on which it is running.

The current `frm_pkg::env_c` environment contains the `frm_agent` and the `alu_agent`, but does not contain the virtual sequencer (`vsqr`). Our goal is to create this sequencer, and then run virtual sequences on it that are capable of touching both agents.

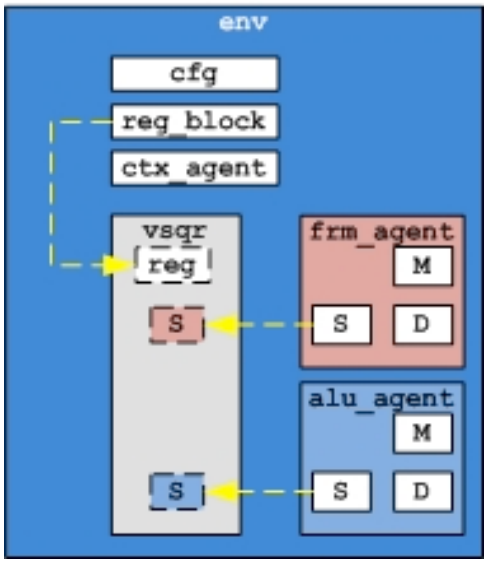


Figure 20: *The Virtual Sequencer in the Framer Environment*

As can be seen from the diagram, the `vsqr` component references the individual sequencers in `frm_agent` and `alu_agent`. These are referred to as its subsequencers. The references should be set during the environment's connect phase. You can also add the register block to the virtual sequencer, in case any of the sequences running on it need to access CSRs.

Problem 16-1

Create the `frm_pkg::vsqr_c` component using `utg` (with the template named `vsqr`), populate the references to the `frm_agent` and `alu_agent` sequencers, and the `reg_block` instance, and instantiate it within `frm_pkg::env_c`.

Uncomment the `basic_vseq_c` class in `verif/vkits/frm/frm_vseq_lib.sv`, and set it as the default sequence for the virtual sequencer's `main_phase` in the basic test.

Solution

`utg` provides most of what you need. All that needs to be done is add the references.

111. verif/vkits/frm/frm_vsqr.sv

```
`include "frm_sqr.sv"

// class: vsqr_c
// Virtual sequencer holding references to frm_sqr and alu_sqr
class vsqr_c extends uvm_sequencer;
   `uvm_component_utils_begin(frm_pkg::vsqr_c)
      `uvm_field_object(frm_sqr,         UVM_REFERENCE)
      `uvm_field_object(alu_sqr,         UVM_REFERENCE)
      `uvm_field_object(reg_block,       UVM_REFERENCE)
   `uvm_component_utils_end

   //---------------------------------------------------------------------------------------
   // Group: Sequencer references

   // field: frm_sqr
   sqr_c frm_sqr;

   // field: alu_sqr
   alu_pkg::sqr_c alu_sqr;

   //---------------------------------------------------------------------------------------
   // Group: Fields

   // field: reg_block
   // alutb register block (reference to the one in cfg)
   alu_csr_pkg::reg_block_c reg_block;

   //---------------------------------------------------------------------------------------
   // Group: Methods
   function new(string name="vsqr",
                uvm_component parent=null);
      super.new(name, parent);
   endfunction : new
endclass : vsqr_c
```

Note that **the build phase is not used to create the subsequencers**.

In the `frm_pkg::env_c`, the virtual sequencer is instantiated in the usual manner and the connect phase is used to perform the assignments. If either of the sequencers are not present (which would happen if either agent were accidentally set to `UVM_PASSIVE`), then a fatal error is reported:

112. verif/vkits/frm/frm_env.sv

```
   virtual function void connect_phase(uvm_phase phase);
      super.connect_phase(phase);

      if(!frm_agent.sqr || !alu_agent.sqr)
         `cn_fatal(("frame_agent.sqr or alu_agent.sqr are not present!"))

      vsqr.frm_sqr = frm_agent.sqr;
      vsqr.alu_sqr = alu_agent.sqr;
   endfunction : connect_phase
```

And the basic test is modified to launch the basic virtual sequence:

113. verif/frm/tests/basic.sv

```
   virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      uvm_config_db#(uvm_object_wrapper)::set(this, "env.vsqr.main_phase",
                                              "default_sequence",
                                              frm_pkg::basic_vseq_c::type_id::get());
   endfunction : build_phase
```

Virtual Sequences

Virtual sequences are generally only different from other sequences in three ways.

1. First, they launch their sequences and sequence items using the `uvm_do_on family of macros, specifying the reference to the sequencer on which to launch the item as their second argument.

2. Second, in order to refer to their sequencer's subsequencers they must call the macro `uvm_declare_p_sequencer.

3. Third, library sequences do not support virtual sequences (as of UVM 1.1). However, a pattern for library virtual sequences is provided for you in Appendix D.

Let's examine the virtual sequence in frm_vseq_lib.sv. The body task uses a fork..join construct to simultaneously send two different sequences to two different sequencers. One is a frame item, and another is an instance of alu_pkg::exer_seq_c.

114. verif/vkits/frm/frm_vseq_lib.sv

```
class basic_vseq_c extends uvm_sequence;
   `uvm_object_utils(frm_pkg::basic_vseq_c)
   `uvm_declare_p_sequencer(vsqr_c)

   //------------------------------------------------------------------------------------------
   // Group: Methods
   function new(string name="basic_vseq");
      super.new(name);
   endfunction : new

   ////////////////////////////////////////////
   // func: body
   virtual task body();
      frame_c frame;
      alu_pkg::exer_seq_c alu_exer_seq;
      // create and randomize to see how many ALU transactions to send
      `uvm_create_on(frame, p_sequencer.frm_sqr)
      frame.randomize();
      `cn_info(("Sending this frame: %s", frame.convert2string()))

      fork
         begin : send_frame
            `uvm_send(frame);
            get_response(rsp);
            `cn_info(("Frame completed: %s", rsp.convert2string()))
         end

         `uvm_do_on_with(alu_exer_seq, p_sequencer.alu_sqr, { count == frame.frame_len; })
      join
```

```
    endtask : body
endclass : basic_vseq_c
```

Some explanations are below:

```
        alu_pkg::exer_seq_c alu_exer_seq;
```

Instead of creating our own sequence which runs a number of random ALU transactions, we can just re-use sequences from other vkits that this vkit is aware of.

```
        `uvm_create_on(frame, p_sequencer.frm_sqr)
        frame.randomize();
```

We want to randomize the frame ourselves so that we know how many ALU transactions to send in. Since a sequence's sequencer must be determined at the time of its creation, UVM provides the `uvm_create_on` macro. Because we declared the `p_sequencer` variable with `uvm_declare_p_sequencer`, that is how we reference the `frm_sqr` instance.

```
            `uvm_send(frame);
```

Since the frame has already been created and randomized, we do not use `uvm_do_on`, but instead we just want to launch it.

```
            get_response(rsp);
            `cn_info(("Frame completed: %s", rsp.convert2string()))
```

We sent in an object derived from sequence item (`frame_c`), so we need to get the response. In this agent, the response frame will not return until the frame with all its data has been pushed out of the framer block.

```
        `uvm_do_on_with(alu_exer_seq, p_sequencer.alu_sqr, { count == frame.frame_len; })
```

Here, we are doing a sequence, specifying the non-virtual sequencer to do it on, and supplying the correct count of ALU transactions to run. UVM offers the `uvm_do_on_with` macro for just such an occasion.

Problem 16-2

Nothing about the framer block says that the frame length has to be specified before or after the ALU transactions associated with it. In the file `verif/vkits/frm_vseq_lib.sv`, write a virtual sequence called `basic_delay_vseq_c` that varies which one comes first.

Then, write another sequence, `exer_vseq_c`, that executes a random number of these. Set this as the default sequence of the test in `verif/frm/tests/basic.sv`.

See if you can find anything wrong with the simulation.

Solution

This implementation of `basic_delay_vseq_c` randomizes a signed integer between -100ns and 100ns. If the random delay is positive, it delays the frame first. If it is negative, it delays the ALU sequence instead.

115. verif/vkits/frm/frm_vseq_lib.sv

```
class basic_delay_vseq_c extends uvm_sequence;
   `uvm_object_utils(frm_pkg::basic_delay_vseq_c)
   `uvm_declare_p_sequencer(vsqr_c)

   //------------------------------------------------------------------------------------------
   // Group: Fields

   // frame_delay_ns
   // Delay of sending the frame, with respect to starting the ALU transactions
   // If frame_delay_ns is negative, then send the frame first, otherwise, send transactions first
   rand int frame_delay_ns;
   constraint frame_delay_cnstr { frame_delay_ns inside {[-100:100]}; }

   //------------------------------------------------------------------------------------------
   // Group: Methods
   function new(string name="basic_delay_vseq");
      super.new(name);
   endfunction : new

   ////////////////////////////////////////
   // func: body
   virtual task body();
      frame_c frame;
      alu_pkg::exer_seq_c alu_exer_seq;

      // create and randomize to see how many ALU transactions to send
      `uvm_create_on(frame, p_sequencer.frm_sqr)
      frame.randomize();

      `cn_info(("Sending this frame: %s", frame.convert2string()))

      fork
         begin
            if(frame_delay_ns > 0)
               #(frame_delay_ns * 1ns);
            `uvm_send(frame);
            get_response(rsp);
            `cn_info(("Frame completed: %s", rsp.convert2string()))
         end

         begin
            if(frame_delay_ns < 0)
               #((-frame_delay_ns) * 1ns);
            `uvm_do_on_with(alu_exer_seq, p_sequencer.alu_sqr, { count == frame.frame_len; })
         end
      join

   endtask : body
endclass : basic_delay_vseq_c
```

The exerciser sequence is a virtual sequence because it is running on a virtual sequencer. But it only launches sequences of type `basic_delay_vseq_c`, which run on the same sequencer as it does. So it does not need to use `` `uvm_do_on ``, and does not need to refer to any of the subsequencers.

116. verif/vkits/frm/frm_vseq_lib.sv

```
class exer_vseq_c extends uvm_sequence;
   `uvm_object_utils_begin(frm_pkg::exer_vseq_c)
      `uvm_field_int(count, UVM_ALL_ON | UVM_DEC)
   `uvm_object_utils_end

   //------------------------------------------------------------------------------
   // Group: Fields

   // field: count
   // The number of basic_vseq to do
   rand int count;
   constraint count_cnstr { count inside {[20:100]}; }

   //------------------------------------------------------------------------------
   // Group: Methods
   function new(string name="exer_vseq");
      super.new(name);
   endfunction : new

   ////////////////////////////////////////
   // func: body
   virtual task body();
      basic_delay_vseq_c vseq;
      `cn_info(("Transmitting %0d frames.", count))
      repeat(count)
         `uvm_do(vseq)
   endtask : body
endclass : exer_vseq_c
```

Everything about this test should seem to work. There is a problem, though. While your sequence prints that it is transmitting a certain number of frames, it will probably actually send fewer than that (or none at all).

The problem is that the main phase of the simulation can end during the body of your sequence, and your sequence will never complete. That's because UVM does not implicitly wait for a component's default sequence to finish before moving on to the next phase. After all, it might be a persistent sequence that never ends.

We'll discuss this in more detail in Appendix A. In the meantime, you can solve this problem by having the sequence explicitly raise and drop the phase's objection. Do this by adding the following code to your `exer_vseq_c` class:

117. verif/vkits/frm/frm_vseq_lib.sv

```
   virtual task body();
      basic_delay_vseq_c vseq;
      `cn_seq_raise
      `cn_info(("Transmitting %0d frames.", count))
      repeat(count)
         `uvm_do(vseq)
      `cn_seq_drop
   endtask : body
```

Adapter Sequences

With the example of virtual sequences, we have seen how block-level sequences can be re-used in higher-level testbenches by continuing to be run on the same sequencers for which they were designed. But what if, upon moving up to the full-chip level, you wanted to operate the same sequences on a different sequencer?

At a higher level your ALU agent may be passive, and the stimulus to the ALU block is instead fed by another RTL block. But, you've gone to a lot of trouble creating sequences that stimulate the ALU just the way you like them, why should you have to re-do them for a different interface? This is where adapter sequences can help.

One-to-One

A simple example would be a feeder block that received CTX write transactions that were then converted to ALU transactions. The objective would be to use the ALU stimulus that was already created for its block-level testbench in this higher-level testbench. To accomplish this, an ALU agent's sequencer could be directed to feed its ALU transactions stream to a `port` instantiated inside a feeder agent's sequencer instead of its own driver. On this sequencer is running an adapter sequence which is a persistent sequence that constantly pulls from the `port` just as a driver would. It then converts each transaction to the correct CTX transaction type, and drives them out its own sequencer.

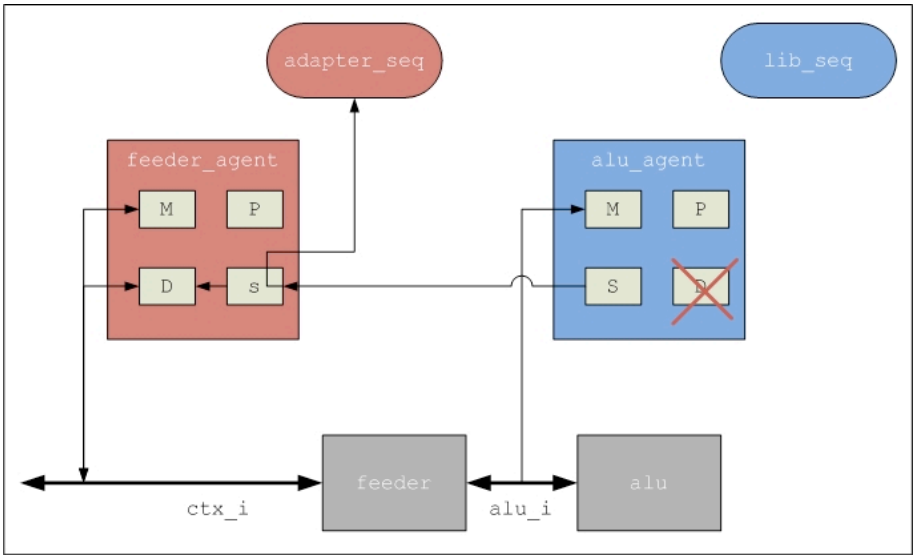


Figure 21: *Using an Adapter Sequence*

This works well when there is a one-to-one correlation between the sequence items. In this case, an `alu_pkg::item_c` is directly converted to a `ctx_pkg::item_c`. And, the `alu_pkg::result_t` is converted to a `ctx_pkg::item_c`.

The basic algorithm for the adapter sequence's body would be as follows, where creation and conversion is done in an algorithm called `convert_to_ctx`:

```
118.
    task body();
       alu_pkg::item_c alu_item;

       forever begin
          p_sequencer.alu_item_port.get_next_item(alu_item);
          convert_to_ctx(item, alu_item);
          `uvm_send(item);
          alu_item.result = read_result();
          p_sequencer.alu_item_port.item_done(alu_item);
       end
    endtask : body
```

One-to-Many

Alternatively, what if the feeder block took several CTX transactions to perform one ALU transaction? The feeder block might hypothetically operate by writing the A value to one CSR, the B value to another CSR, and the operation type to a third CSR. An adapter sequence would also be the right choice here, as it would merely send three outbound CTX transactions for every one ALU transaction.

Many-to-One

A more complex example is one that has a many-to-one mapping of sequence items. The feeder block instead might receive CTX transactions that write a block of data into it, and then an CTX request that starts summing all of the numbers to be summed, with a polling mechanism to indicate when the transaction is complete.

We already have a sequence that sums an array. It even checks the final result for us. But it transmits multiple transactions that need to be lumped into one transaction. The adapter sequence would have to be smart enough to know how to do this, and that may not be practical. Thought must be given to how and where complex scenario prediction should take place. In this example, an adapter sequence may not be the right solution, and putting the prediction within the sequence may have been a mistake because it would not be reusable in the higher-level testbench.

Conclusion

In this chapter, we learned the important concept of virtual sequences and sequencers. We also saw how adapter sequences can be re-used to take a sequence that resides on one interface and drive it on the next.

Congratulations! This concludes the final chapter. The material presented here offers a template that applies to a very specific—and rather simplistic—device under test. The scenarios for which you later apply these lessons will be more complex and very different from this one, but hopefully what is presented here will help steer you towards a clear path.

Appendix A: Phases and Heartbeats

The global environment contains both a watchdog timer and a heartbeat monitor. Together with UVM's phase objections, these components work together to ensure that simulation jobs are not running away needlessly.

Watchdog Timer

The watchdog timer will issue a fatal error when the simulation time exceeds the specified watchdog time, which can be set on a `cnmake` command-line:

```
verif/alutb> cnmake sim UTEST=basic --wdog 300000
```

Watchdog times are always specified in nanoseconds.

Phase Objections

Phase objections exist to ensure that your various independent components do not get ahead of one another. It would be inappropriate to start sending packets into a DUT before reset was complete, for example.

Each of UVM's run-time phases ends immediately when all phase objections have been dropped. **Any task that was spawned during the phase but is still active will immediately be killed.**

The heartbeat monitor works in concert with a well-constructed environment which raises and drops objections in the correct manner. Where and when your objections take place depend on the current phase.

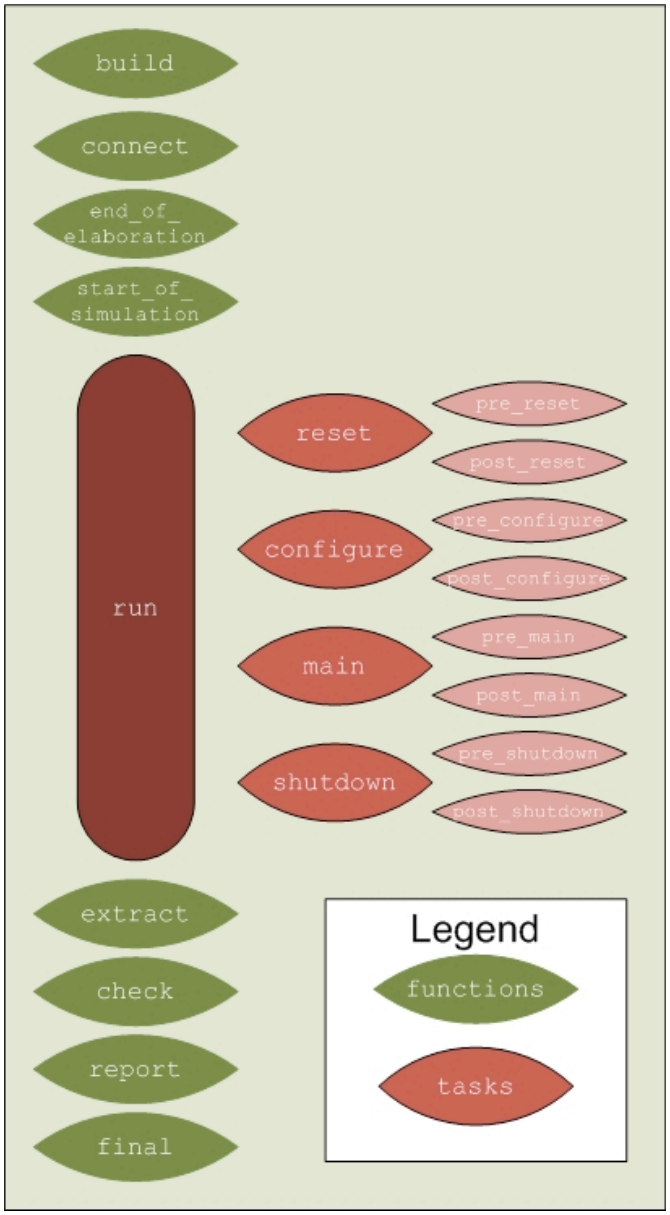


Figure 22: *UVM Phases*

Reset Phase

The `cn` package contains a standard reset driver component, `cn_pkg::rst_drv_c`. This driver may be instantiated and configured to provide a simple reset on a single reset line. When this standard driver is used, it will manage the raising and dropping of the reset phase's objections for you.

If you decide to create your own reset driver, this phase's objection must be raised at the beginning of the phase, and be dropped once the reset is complete.

Configuration Phase

As discussed in Chapter 12, the configuration phase may be as simple as calling `register_block.update()`, or as complicated as you wish it to be. Like the reset phase, objections should be raised at the beginning and dropped at the end.

Main Phase

The main phase is substantially different from both the reset and configuration phases. The purpose of the main phase is to launch sequences that inject sequence items into the DUT. The main phase does not necessarily wait for the completion of these items, it merely lasts as long as it takes to send these items to the drivers.

Therefore, the main phase should end once all sequence items have been launched. Where these objections are raised and dropped depends entirely on how your sequences are launched.

If your main phase is creating and starting your sequences directly, then you must raise and drop the objections once they are complete. In Chapter 9, your first sequence was started in this manner:

```
119.
    virtual task main_phase(uvm_phase phase);
       alu_pkg::alu_seq_c alu_seq = new("basic_seq");

       phase.raise_objection(this);
       `cn_info(("Starting alu_seq."))
       alu_seq.start(alutb_env.alu_agent.sqr);
       phase.drop_objection(this);
    endtask : main_phase
```

Alternatively, you may launch your sequences as the default sequence of the `main_phase`. In Chapter 14, this is how you launched the `sum_array_seq_c`:

```
120.
uvm_config_db#(uvm_object_wrapper)::set(this, "alutb_env.alu_agent.sqr.main_phase",
                                        "default_sequence", alu_pkg::sum_array_seq_c::type_id::get());
```

When a sequence is launched in this manner, a field in the sequence called `starting_phase` is set to the current phase. Otherwise, this field is `null`. In either the sequence's body, or in pre- and post-body tasks, you can use this field to raise and drop the objection:

```
121.
    virtual task pre_body();
       if(starting_phase)
          starting_phase.raise_objection();
    endtask : pre_body

    virtual task post_body();
       if(starting_phase)
          starting_phase.drop_objection();
    endtask : post_body
```

Since you never know when a sequence is going to be configured to be a default sequence, you would have to add these tasks to every sequence you write. A recommended procedure is to create a base sequence that contains this code, and then derive all of your other sequences from it.

You can also use the macros `cn_seq_raise and `cn_seq_drop as shown in Chapter 16.

Shutdown Phase

The shutdown phase is the DUT's opportunity to finish responding to all of the stimulus that was previously driven into it. This phase should end once all of the stimulus has *completed*. This is usually decided by a predictor whose scoreboards are empty, a monitor or driver awaiting responses, etc. Only components that can determine this information should be raising and dropping the shutdown phase's objection.

The following is a typical pattern for a predictor's shutdown phase:

```
122.
   virtual task shutdown_phase(uvm_phase phase);
      if(scoreboard.size()) begin
         phase.raise_objection(this, $sformatf("Waiting for %0d responses.", scoreboard.size()));
         while(scoreboard.size())
            @(response_received);
         phase.drop_objection(this, "All responses complete.");
      end
   endtask : shutdown_phase
```

The phase only raises an objection if there are outstanding packets, otherwise it does nothing. As shown here, an event must have been defined which is triggered whenever a response is received.

Once the shutdown phase is complete, your simulation will move to the extract and check phases.

Extract and Check Phases

The extract phase precedes the check phase and provides higher-level components with the opportunity to pull information from lower-level components in order to complete its prediction process.

The check phase is your environment's last chance to say whether or not the simulation passed or failed. This is where scoreboards must be checked to be empty, credits must be completed, etc.

These phases are not run-time phases. They are zero-time functions whose objections need not be raised or dropped. They are mentioned here because as you will see in the next

section, it is important that your predictive components report errors during the check phase for any final conditions.

Deadlock Checking

Deadlock is when stimulus has gone into the DUT, but responses are not coming out. How long to wait before deciding that it will never come out is a function of your DUT and your environment. When a simulation is deadlocked, it should immediately exit and report a failure. It is also helpful to report what the environment is still waiting for.

The global environment's heartbeat monitor, `global_pkg::heartbeat_mon_c`, is the configurable component that is responsible for detecting a deadlock scenario. During the `end_of_elaboration` phase, components that are responsible for tracking responses from the DUT (i.e. monitors, predictors, subscribers, etc.) register themselves with the heartbeat monitor. Then, after reset has completed, the heartbeat monitor periodically checks to ensure that *at least one* registered component has seen some form of activity from the DUT (the heartbeat). The length of time between these checks is called the *sample time*. The monitor will obey the *longest* sample time required by all registered components.

If no activity was seen during the last sample time, the simulation is considered deadlocked. In that scenario, all run-time phases are immediately halted, and the simulation proceeds to the extract, check, and final phases, where your environment has an opportunity to report what activity is still expected of the DUT. This is called a *phase jump*.

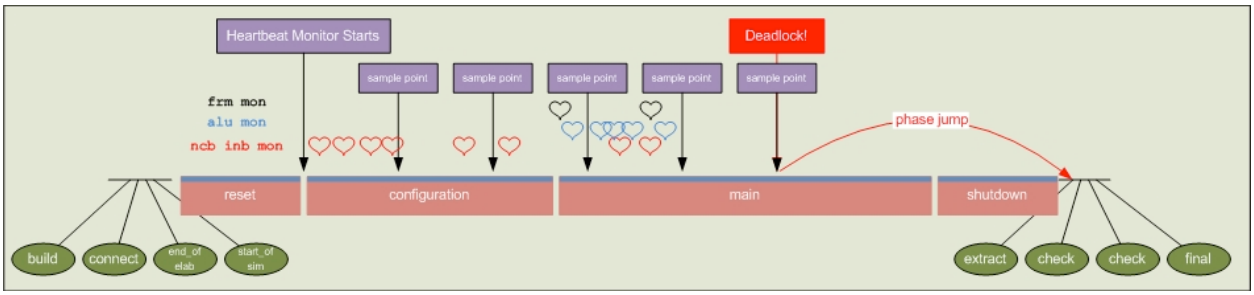


Figure 23: *Deadlock Phase Jump*

The art of getting the heartbeat monitor to work properly for your environment lies in determining which components must be registered, and how long the sample time should be. Your registered monitors will probably have different needs from one another. For example, AMBA responses might come out just a few clocks after the request, whereas a complex algorithmic unit such as a video encoder might take 50,000ns before a response comes out.

Each component registers itself and its required sample time during the `end_of_elaboration` phase using a standard macro:

```
123.
   virtual function void end_of_elaboration_phase(uvm_phase phase);
      `global_add_to_heartbeat_mon(7000ns)
   endfunction : end_of_elaboration_phase
```

When a response to *stimulus* is seen from the DUT, then your component indicates a heartbeat with another macro. It is important to distinguish between responses to stimulus and responses to random traffic, such as random CSR reads.

The ALU monitor's `monitor_result` task would add this functionality:

```
124.
   virtual task monitor_result();
      forever begin
         @(posedge mon_vi.mon_cb.ready);
         if(waiting_for_result == null)
            `cn_err(("Monitored Result does not match any outstanding transaction."))

         `cn_info(("Monitored Result: %08X", mon_vi.mon_cb.result));
         monitored_result_port.write(mon_vi.mon_cb.result);
         `global_heartbeat("Result seen")
         waiting_for_result = null;
      end
   endtask : monitor_result
```

The ALU monitor's check phase should report an error if it is still waiting for a response:

```
125.
   virtual function void check_phase(uvm_phase phase);
      super.check_phase(phase);
      if(waiting_for_result != null)
         `cn_err(("Monitor is currently waiting for a result to transaction: %s",
                  waiting_for_result.convert2string()))
   endfunction : check_phase
```

Both of these methods use the variable `waiting_for_result`, which is a reference to the outstanding request and would be set by the `monitor_item` task.

Note that the `utg` script does **not** do this for you. You will need to add these heartbeats and checks yourself.

By default, the heartbeat monitor starts in the `pre_configuration` phase, but it can be configured to start during `pre_reset` or `pre_main` instead. It stops at the beginning of the `post_shutdown` phase.

You can use the `post_shutdown` phase to perform any end-of-run testing, such as reading statistics CSRs, etc.

Appendix B: Common Recipes

Implementing a Watchdog

Often, you will want to send something into the DUT and ensure that it completes within a "reasonable" amount of time. Otherwise, you want to report an error. Depending on your scenario, this may be measured in nanoseconds, clocks, or some other triggering event.

A simple method to do this is as follows:

```
126.
fork
   do_something();
   begin
      #(5000ns);
      `cn_error(("Watchdog timeout!"))
   end
join_any
disable fork;
```

Either the `do_something()` task will complete its mission, or the error will occur in 5,000ns.

Forking Multiple Instances of a Method

There will be a host of situations where you will want to fork off multiple instances of the same method call with different arguments. Doing this simple approach will not give the desired results:

```
for(int num=0; num < 5; num++)
   fork
      do_stuff(num);
   join_none
```

Nor will this:

```
fork
   for(int num=0; num < 5; num++)
      do_stuff(num);
join
```

The problem with both of these examples is that while five tasks will be launched, they will all be launched with an argument of 4. This is due to the fact that the loop will add 4 tasks to the scheduler, and when they later launch the `num` variable will have a final value of 4.

To fix this, you need an `automatic` variable to accept the loop value:

```
127.
for(int num=0; num < 5; num++) begin
   automatic int _num = num;
   fork
      do_stuff(_num);
   join_none
end
```

Phase-Boundary Crossing Tasks

Any task that starts in a run-time phase (other than the `run_phase`), will be killed at the end of that phase. Raising and holding an objection on that phase may not be desirable if you want that task to continue into a later phase.

A way around that is to wait for a start event in the `run_phase`, launch your task, and then kill it upon receiving a finish event. In the following example, the task `do_stuff()` launches at the beginning of the main phase and continues through to the end of the shutdown phase. The `do_stuff()` task runs a `forever` loop.

```
128.
event start_event, finish_event;

////////////////////////////////////////////
virtual task run_phase(uvm_phase phase);
   @(start_event);
   fork
      do_stuff();
      @(finish_event);
   join_any
   disable fork;
endtask : run_phase

////////////////////////////////////////////
virtual task main_phase(uvm_phase phase);
   -> start_event;
endtask : main_phase

////////////////////////////////////////////
virtual task post_shutdown_phase(uvm_phase phase);
   -> finish_event;
endtask : post_shutdown_phase
```

Randomizing a Dynamic Array of Objects

If you want to generate a randomly-sized array of classes, you declare them as a dynamic array and constrain its size:

```
rand port_c ports[];
constraint ports_cnstr {
   ports.size() inside {[1:64]};
}
```

However, you still need to `new` the array and create each element before you can randomize them.

```
ports = new[??];
foreach(ports[x])
   ports[x] = port_c::type_id::create($sformatf("ports[%0d]", x), this);
```

But how do you `new` an array of classes when you don't know the size of the array beforehand? And where do you put this code?

The solution is that you `new` the maximum number of possible ports. After the size is randomized, the array will be re-sized and the extras will be garbage-collected.

```
129.
function new(string name="my_class");
   super.new(name);
   ports = new[64];
   foreach(ports[x])
      ports[x] = port_c::type_id::create($sformatf("ports[%0d]", x), this);
endfunction : new
```

While this method may initially be wasteful of resources, it does permit higher-level objects the ability to add constraints onto the items in the array:

```
constraint cfg_cnstr {
   cfg.ports.size() == 3;
   cfg.ports[0].item == 0;
   cfg.ports[1].item == 4;
   cfg.ports[2].item == 7;
}
```

Dynamic Arrays of TLM Ports

Creating a random number of TLM ports is actually pretty straightforward. A cfg class holds the number of ports that need to be created, a dynamic array is declared, and during the `build_phase` the array is newed and each of the ports is newed.

```
130.
class agent_c extends uvm_agent;
   `uvm_component_utils_begin(agent_c)
      `uvm_field_object(cfg, UVM_REFERENCE)
   `uvm_component_utils_end

   //---------------------------------------------------------------------------
   // Group: Configuration Fields

   // field: cfg
   // The agent configuration class
   cfg_c cfg;

   //---------------------------------------------------------------------------
   // Group: TLM Ports

   // field: item_ports
   uvm_blocking_put_port #(item_c) item_ports[];

   //---------------------------------------------------------------------------
   // Group: Methods
   function new(string name="agent", uvm_component parent=null);
      super.new(name, parent);
   endfunction : new

   ////////////////////////////////////////////
   // func: build_phase
```

```
   virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);

      item_ports = new[cfg.num_ports];
      foreach(item_ports[p])
         item_ports[p] = new($sformatf("item_ports[%0d]", p), this);
   endfunction : build_phase
endclass : agent_c
```

But how does one create a dynamic array of *n* `imps`? Recall that if a component contains multiple `imps`, each must be declared with the `` `uvm_blocking_put_imp_decl `` macro and each macro must have a matching `put` task. But what if you might have *n*=1,000 `imps`?

The solution involves instantiating *n* `exports`, connecting them to *n* `uvm_tlm_fifo` instances, and then spawning *n* tasks that pull from these FIFOs.

```
131.
class agent_c extends uvm_agent;
   `uvm_component_utils_begin(agent_c)
      `uvm_field_object(cfg, UVM_DEFAULT)
   `uvm_component_utils_end

   //---------------------------------------------------------------------------
   // Group: Configuration Fields

   // field: cfg
   // The agent configuration class
   cfg_c cfg;

   //---------------------------------------------------------------------------
   // Group: TLM Ports

   // field: item_exports
   uvm_blocking_put_export #(item_c) item_exports[];

   //---------------------------------------------------------------------------
   // Group: Fields

   // field: item_fifos
   // These are 'internal' to the agent and so not exposed to users
   item_fifo_t req_fifo[];

   //---------------------------------------------------------------------------
   // Group: Methods
   function new(string name="agent", uvm_component parent=null);
      super.new(name, parent);
   endfunction : new

   //////////////////////////////////////////
   // func: build_phase
   virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);

      // instantiate the exports and fifos
      item_exports = new[cfg.num_imps];
      item_fifos = new[cfg.num_imps];

      foreach(item_exports[p]) begin
         item_exports[p] = new($sformatf("item_exports[%0d]", p), this);
         item_fifos[p] = new($sformatf("item_fifos[%0d]", p), this);
      end
   endfunction : build_phase

   //////////////////////////////////////////
   // func: connect_phase
```

```
    virtual function void connect_phase(uvm_phase phase);
       super.connect_phase(phase);

       // connect each export to its assigned fifo
       foreach(req_export[p])
          req_export[p].connect(req_fifo[p].put_export);
    endfunction : connect_phase

    ////////////////////////////////////////////
    // func: run_phase
    // Spawn separate tasks for each fifo
    virtual task run_phase(uvm_phase phase);
       foreach(item_fifos[p]) begin
          automatic int _p = p;
          fork
             handle_item(_p);
          join_none
       end
    endtask : run_phase

    ////////////////////////////////////////////
    // func: handle_item
    // Continuously retrieve items from the assigned fifo
    virtual task handle_item(int _port);
       item_c item;
       forever begin
          item_fifos[_port].get(_item);
          `cn_info(("I got an item!"))
       end
    endtask : handle_item
endclass : agent_c
```

Appendix C: Vertical Reuse

The term *vertical reuse* refers to taking your block-level testbench and reusing its components at a subsystem or full-chip level. It requires that you architect your environment to be flexibly used with the other environments in the system. This appendix will describe the recommended architecture that can be used by most designs.

Suppose there are two blocks in the system, alpha and beta. They communicate with one another over the a2b interface and the b2a interface. These can collectively be referred to as the AB interface. Each of these blocks will have its own separate testbench.

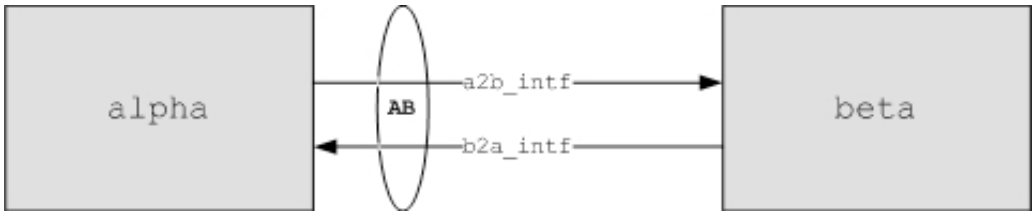


Figure 24: *Shared Interfaces*

To facilitate re-use, the preferred approach is to extract the AB interface into its own separate vkit. Separate alpha and beta block-level testbenches might appear as follows, with the components of the AB vkit given in blue. The AB vkit consists of the interface(s), an a2b agent, and a b2a agent. Because it consists of multiple agents, it is convenient to create a configurable AB environment.

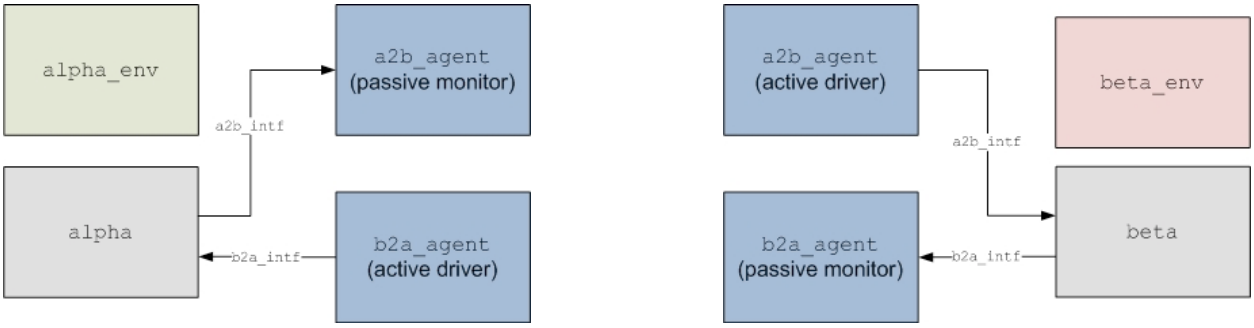


Figure 25: *Multiple Uses of An Interface Vkit*

The alpha testbench must drive on the b2a interface to stimulate alpha's inputs, but needs only to monitor the a2b interface. Therefore, it puts the b2a agent in active mode and the a2b agent in passive mode. Conversely, the beta testbench sets the a2b agent in active mode and the b2a agent in passive mode. A complete AB environment would appear as shown below.

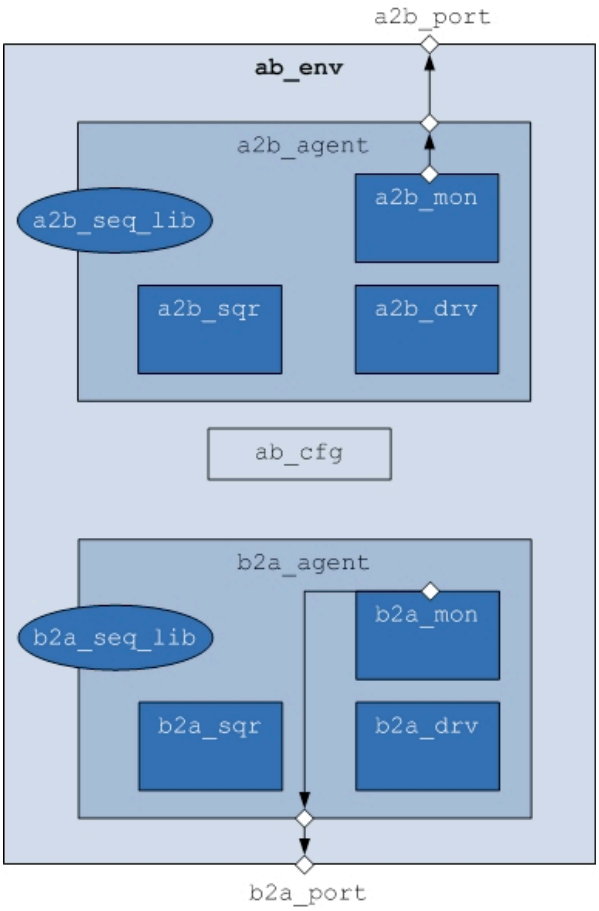


Figure 26: *The Complete AB Environment*

Now suppose that the designers of the alpha and beta environments wished to put an alpha/beta subsystem testbench together. This becomes a simple matter of instantiating and connecting the components of the three vkits and configuring them to behave in a manner suitable for this testbench.

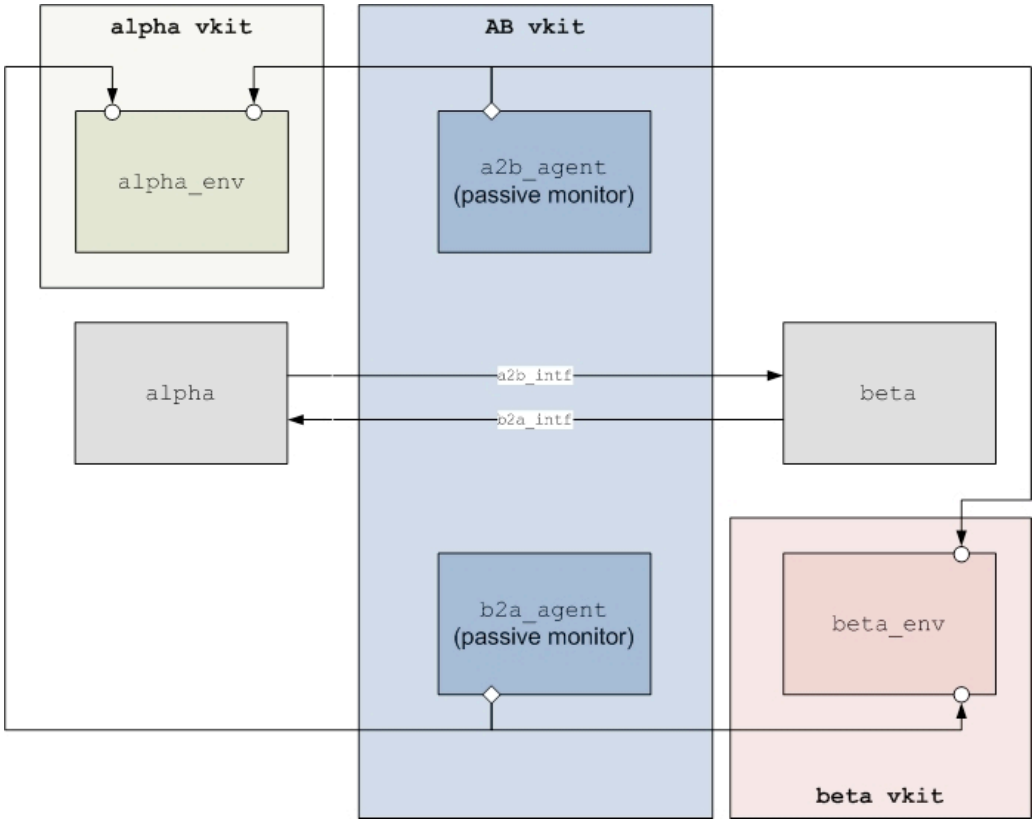


Figure 27: *An Interface Vkit in a Full-Chip Environment*

Shared Cfg Classes

The alpha testbench has the alpha environment which contains all of the required components to test the alpha block, including those components such as the AB environment that exist in other vkits. These other vkits have their own cfg classes. It is highly desirable to give higher-level components access to the configurations of lower-level components. To do so, a hierarchy of cfg classes should mirror the architecture of the components they serve.

In the case of the alpha testbench, an alpha cfg class would contain an instance of the AB cfg class. Likewise, the beta cfg class would also contain the AB cfg class.

A subsystem environment would likely have its own cfg class (`sub_pkg::cfg_c`). It would have an instance of the alpha cfg class and a beta cfg class. But, it would also contain just a single instance of the AB cfg class which it would then push down into the alpha and beta cfg classes so that only one instance exists.

In the combined subsystem `base_test` would be found a single instance of the `sub_pkg` cfg class.

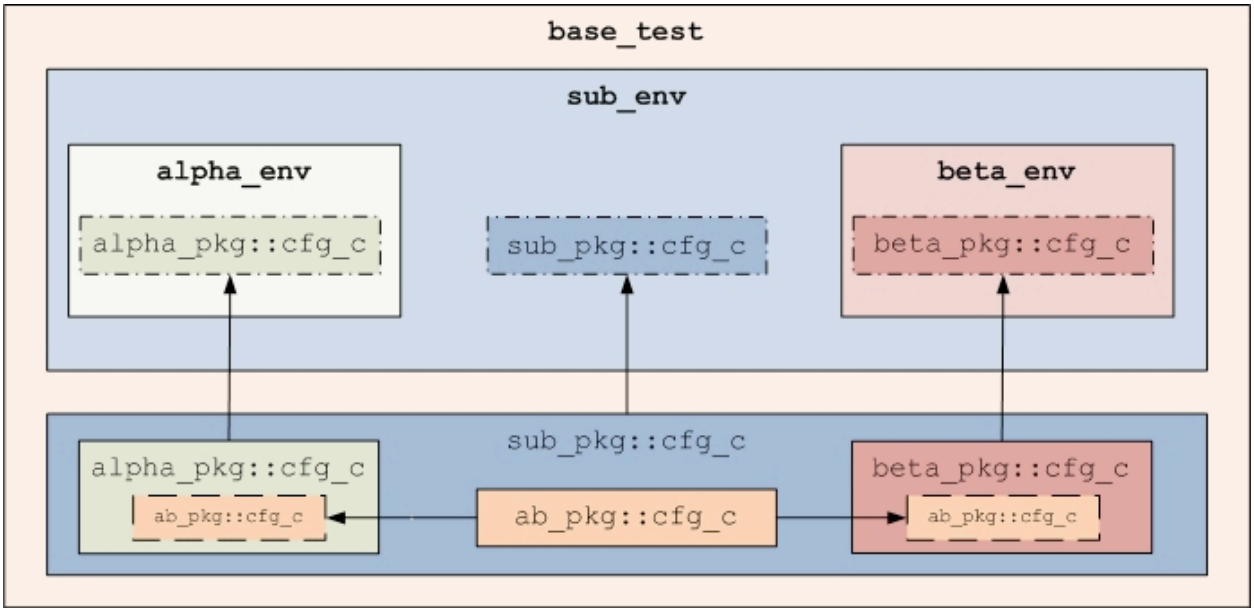


Figure 28: *CFG Classes in a Sub-System Environment*

There are a great many advantages to this approach. First, higher-level cfg classes can impose constraints and settings on lower-level cfg classes. Second, by consolidating all of the sub-system's configurations into a single hierarchical cfg class it can easily be re-used at a full-chip level. Third, by instantiating the cfg class in the `base_test` and then pushing references to it to lower-level components, the tests can tweak all of the knobs in the system by adding constraints upon them.

Example A2B and alpha cfg classes and a base test of the alpha's block-level testbench are shown below. Here, the base test calls the alpha cfg class's `create_cfg` function to create the A2B cfg class. When the alpha cfg class is randomized, the AB cfg class will also be randomized.

```
132.
//------------------------------------------------------------------------------
package ab_pkg;
   import uvm_pkg::*;

   class cfg_c extends uvm_object;
      `uvm_object_utils_begin(cfg_c)
         `uvm_field_int(my_alpha_field, UVM_ALL_ON | UVM_HEX)
         `uvm_field_int(my_beta_field, UVM_ALL_ON | UVM_HEX)
         `uvm_field_int(my_mix_field, UVM_ALL_ON | UVM_HEX)
      `uvm_object_utils_end

      rand bit[15:0] my_alpha_field;
      rand bit[15:0] my_beta_field;
      rand bit[31:0] my_mix_field;

      //------------------------------------------
      function new(string name="cfg");
         super.new(name);
      endfunction : new
   endclass : cfg_c
endpackage : ab_pkg
```

```
//------------------------------------------------------------------------------
package alpha_pkg;
   import uvm_pkg::*;

   class cfg_c extends uvm_object;
      `uvm_object_utils_begin(cfg_c)
         `uvm_field_int(my_field, UVM_ALL_ON | UVM_HEX)
         `uvm_field_object(ab_cfg, UVM_ALL_ON)
      `uvm_object_utils_end

      // field: my_field
      rand bit[15:0] my_field;

      // field: ab_cfg
      rand ab_pkg::cfg_c ab_cfg;
      constraint ab_cfg_cnstr {
         ab_cfg.my_alpha_field == my_field;
         ab_cfg.my_mix_field[15:0] == my_field;
      }

      //----------------------------------------
      function new(string name="cfg");
         super.new(name);
      endfunction : new

      //----------------------------------------
      function void create_cfg();
         ab_cfg = ab_pkg::cfg_c::type_id::create("ab_cfg");
      endfunction : create_cfg
   endclass : cfg_c
endpackage : alpha_pkg

//------------------------------------------------------------------------------
class alpha_base_test_c extends uvm_test;
   `uvm_component_utils(alpha_base_test_c)

   rand alpha_pkg::cfg_c cfg;
   constraint alpha_cnstr {
      cfg.my_field == 'haaaa;
   }

   //----------------------------------------
   function new(string name="alpha_base_test",
                uvm_component parent=null);
      super.new(name, parent);
   endfunction : new

   //----------------------------------------
   function void build_phase(uvm_phase phase);
      super.build_phase(phase);

      // create cfg
      cfg = alpha_pkg::cfg_c::type_id::create("alpha_cfg");
      cfg.create_cfg();

      // randomize all
      randomize();

      `cn_info(("cfg =\n%s", cfg.sprint()))
   endfunction : build_phase

endclass : alpha_base_test_c
```

The subsystem's base test resembles the figure above. Alpha's `create_cfg` function is **not** called because it is the base test that creates it. So long as the two blocks do not have constraints that conflict with one another on the AB cfg class, the constraints from the alpha and beta cfg classes will correctly be applied to the AB cfg class.

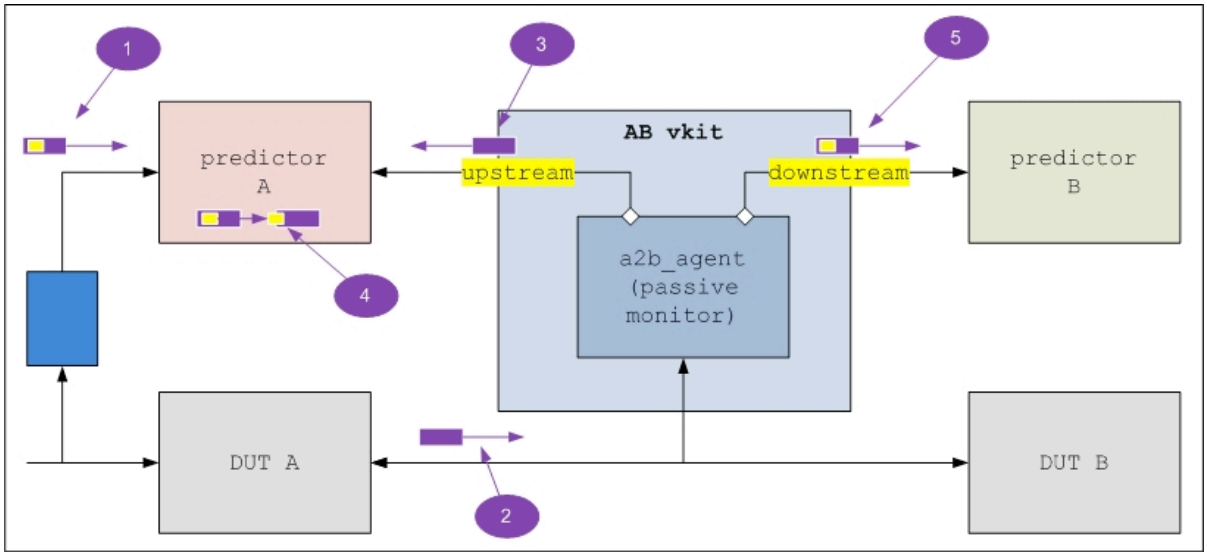
Here is the code for the sub-system's base test that builds and randomizes all of these (the environments are not shown):

```
133.
//------------------------------------------------------------------------------
class subsys_base_test_c extends uvm_test;
   `uvm_component_utils_begin(subsys_base_test_c)
   `uvm_component_utils_end

   // field: alpha_cfg
   rand alpha_pkg::cfg_c alpha_cfg;
   constraint alpha_cnstr {
      alpha_cfg.my_field == 'haaaa;
   }

   // field: beta_cfg
   rand beta_pkg::cfg_c beta_cfg;
   constraint beta_cnstr {
      beta_cfg.my_field == 'hbbbb;
   }

   // field: ab_cfg
   rand ab_pkg::cfg_c ab_cfg;

   //------------------------------------------
   function new(string name="subsys_base_test",
                uvm_component parent=null);
      super.new(name, parent);
   endfunction : new

   //------------------------------------------
   function void build_phase(uvm_phase phase);
      super.build_phase(phase);

      // create cfg
      alpha_cfg = alpha_pkg::cfg_c::type_id::create("alpha_cfg");
      beta_cfg = beta_pkg::cfg_c::type_id::create("beta_cfg");
      ab_cfg   = ab_pkg::cfg_c::type_id::create("ab_cfg");
      // don't call alpha_cfg.create_cfg()!

      // assign cfg
      alpha_cfg.ab_cfg = ab_cfg;
      beta_cfg.ab_cfg = ab_cfg;

      // randomize all
      randomize();

      `cn_info(("alpha_cfg =\n%s", alpha_cfg.sprint()))
      `cn_info(("beta_cfg =\n%s", beta_cfg.sprint()))
   endfunction : build_phase
endclass : subsys_base_test_c
```

Passing Meta-Data

The previous section showed how two different testbenches can effectively work in isolation or be bridged by sharing a configurable vkit. But what if each testbench wished to share its meta-data with each other? For example, how would predictor A tell predictor B that the packet that each of them just saw on the AB interface correlates with unique-ID `uid:0047`?

One solution is that the AB monitor pushes the transaction out not one analysis `port`, but two of them. The following diagram shows how this would work with the AB vkit.



1. Packet `uid:0047` enters the DUT and is placed in Predictor A's scoreboard.

2. Raw data flows out of DUT A and is observed by the a2b agent's monitor.

3. The a2b monitor collects the complete packet and sends it out its *upstream* port to predictor A:

```
    upstream_port.write(pkt);  // pass to predictor A
```

4. Predictor A correlates the packet received from the a2b monitor with packet `uid:0047` that it saw earlier. It then assigns a reference to this unique ID to the new packet that was found.

```
  virtual function void do_write(a2b_pkg::pkt_c _monitored_pkt);
     a2b_pkg::pkt_c found_pkt;
     found_pkt = scoreboard.find_first(item) with (item.received == 0);
     found_pkt.received = 1;
     _monitored_pkt.uid = found_pkt.uid;
  endfunction : do_write
```

5. The a2b monitor then passes **this same packet** to predictor B via its *downstream* analysis port.

```
    upstream_port.write(pkt);   // pass to predictor A
    downstream_port.write(pkt); // pass to predictor B
```

Because a handle to the packet that the `a2b_agent` created is passed to **both** predictors, predictor B will see the uid that predictor A *assigned* to it. If DUT A is expected to split packet `uid:0047` into multiple output packets, then predictor A can assign sub-ids to each of the resulting packets.

Appendix D: Design Patterns

The following design patterns incorporate the concepts of vertical reuse found in Appendix C as well as the principles of reset testing as discussed in "Reset Testing Made Simple with UVM Phases".

Design Pattern for Agents

The agent design pattern features a `pre_reset_phase` that first stops all sequences running on the sequencer, and then resets the driver.

```
134.
// class: agent_c
class agent_c extends uvm_agent;
   `uvm_component_utils_begin(my_pkg::agent_c)
      `uvm_field_enum(uvm_active_passive_enum, is_active, UVM_ALL_ON)
   `uvm_component_utils_end

   //---------------------------------------------------------------------------------------
   // Group: Configuration Fields

   // var: is_active
   // When set to UVM_ACTIVE, the corresponding sqr and drv will be present.
   uvm_active_passive_enum is_active = UVM_ACTIVE;

   //---------------------------------------------------------------------------------------
   // Group: TLM Ports

   // var: item_port
   // All monitored items go out here
   uvm_analysis_port#(item_c) item_port;

   //---------------------------------------------------------------------------------------
   // Group: Fields

   // vars: drv, mon, sqr
   // Driver, monitor, and sequencer
   drv_c drv;
   mon_c mon;
   sqr_c sqr;

   //---------------------------------------------------------------------------------------
   // Group: Methods
   function new(string name="agent",
                uvm_component parent=null);
      super.new(name, parent);
   endfunction : new

   ////////////////////////////////////////////
   // func: build_phase
   virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);

      mon = mon_c::type_id::create("mon", this);

      if(is_active) begin
         drv = drv_c::type_id::create("drv", this);
         sqr = sqr_c::type_id::create("sqr", this);
      end

      item_port = new("item_port", this);
   endfunction : build_phase
```

```
   ////////////////////////////////////////////
   // func: connect_phase
   virtual function void connect_phase(uvm_phase phase);
      super.connect_phase(phase);
      if(is_active) begin
         drv.seq_item_port.connect(sqr.seq_item_export);
         drv.downstream_port.connect(mon.upstream_imp);
      end

      mon.item_port.connect(item_port);
   endfunction : connect_phase

   ////////////////////////////////////////////
   // func: pre_reset_phase
   virtual task pre_reset_phase(uvm_phase phase);
      super.pre_reset_phase(phase);
      if(is_active) begin
         sqr.stop_sequences();
         ->drv.reset_driver;
      end
   endtask: pre_reset_phase

endclass : agent_c
```

Design Pattern for Drivers

The driver below incorporates resettability, a downstream port for vertical reuse, and a `shutdown_phase` that will raise the objection whenever a sequence is being driven.

```
135.
// class: drv_c
class drv_c extends uvm_driver#(item_c);
   `uvm_component_utils_begin(my_pkg::drv_c)
      `uvm_field_string(intf_name, UVM_DEFAULT)
   `uvm_component_utils_end

   //---------------------------------------------------------------------------------------
   // Group: Configuration Fields

   // var: intf_name
   // Interface name
   string    intf_name;

   //---------------------------------------------------------------------------------------
   // Group: TLM Ports

   // var: downstream_port
   // Push out expected beats to listeners
   uvm_analysis_port#(item_c) downstream_port;

   //---------------------------------------------------------------------------------------
   // Group: Fields

   // var: vi
   // Interface for driving sbus
   virtual my_intf.drv_mp vi;

   // var: start_driver, reset_driver
   // Used for reset testing
   event start_driver, reset_driver;

   // var: driving
   // Set to 1 while driving
   bit driving;

   //---------------------------------------------------------------------------------------
   // Group: Methods
   function new(string name="drv",
```

```
                uvm_component parent=null);
   super.new(name, parent);
endfunction : new

////////////////////////////////////////////
// func: build_phase
// Get the virtual interface and create the downstream_port
virtual function void build_phase(uvm_phase phase);
   super.build_phase(phase);
   `cn_get_intf(virtual my_intf.drv_mp, "my_pkg::intf", intf_name, vi)
   downstream_port = new("downstream_port", this);
endfunction : build_phase

////////////////////////////////////////////
// func: run_phase
// Launch the driver when out of reset
virtual task run_phase(uvm_phase phase);
   forever begin
      reset();
      @(start_driver);

      fork
         driver();
      join_none

      @(reset_driver);
      disable fork;
   end
endtask : run_phase

////////////////////////////////////////////
// func: post_reset_phase
virtual task post_reset_phase(uvm_phase phase);
   ->start_driver;
endtask : post_reset_phase

////////////////////////////////////////////
// func: shutdown_phase
// Ensure that the shutdown phase doesn't finish until we are finished driving
virtual task shutdown_phase(uvm_phase phase);
   forever begin
      if(driving) begin
         phase.raise_objection(this, "currently driving");
         @(driving);
         phase.drop_objection(this, "inactive");
      end
      @(driving);
   end
endtask : shutdown_phase

////////////////////////////////////////////
// func: reset
// Reset the interface signals and any class fields
task reset();
   vi.reset();
   driving = 0;
endtask // tx_reset

////////////////////////////////////////////
// func: driver
// Get requests, pack them, and drive them
virtual task driver();
   byte unsigned cycles[];

   forever begin
      seq_item_port.try_next_item(req);
      if(!req) begin
         vi.reset();
         seq_item_port.get_next_item(req);
         @(vi.drv_cb);
      end
```

```
            `cn_dbg(100, ("Driving: %s", req.convert2string()))
            downstream_port.write(req);
            cycles.delete();
            req.pack_bytes(cycles);
            foreach(cycles[idx]) begin
               driving = 1;
               vi.drv_cb.valid <= 1'b1;
               vi.drv_cb.data <= cycles[idx];
               `cn_dbg(150, ("Cycle: %02X", cycles[idx]))
               @(vi.drv_cb);
            end
            seq_item_port.item_done(req);
            driving = 0;
            vi.reset();
            @(vi.drv_cb);
         end
      endtask : driver
   endclass : drv_c
```

Design Pattern for Monitors

The monitor below incorporates elements that provide for a clean reset, upstream and downstream connectivity, global heartbeats, and an elegant shutdown phase.

```
136.
   // class: mon_c
   class mon_c extends uvm_monitor;
      `uvm_component_utils_begin(my_pkg::mon_c)
         `uvm_field_string(intf_name, UVM_DEFAULT)
      `uvm_component_utils_end

      //------------------------------------------------------------------------------
      // Group: Configuration Fields

      // var: intf_name
      // Interface name
      string intf_name;

      //------------------------------------------------------------------------------
      // Group: TLM Ports

      // var: item_port
      // All monitored items go out here
      uvm_analysis_port#(item_c) item_port;

      // var: upstream_analysis_imp
      // Receives the next expected item
      uvm_analysis_imp_upstream #(item_c, mon_c) upstream_imp;

      // var: downstream_port
      // Pushes out the next expected item to listeners
      uvm_analysis_port #(item_c) downstream_port;

      //------------------------------------------------------------------------------
      // Group: Fields

      // var: vi
      // Virtual interface to monitor
      virtual my_intf.mon_mp vi;

      // var: exp_items
      // The expected items from upstream
      item_c exp_items[$];

      //------------------------------------------------------------------------------
      // Group: Methods
      function new(string name="mon",
```

```
                uvm_component parent=null);
   super.new(name, parent);
endfunction : new

////////////////////////////////////////////
// func: build_phase
virtual function void build_phase(uvm_phase phase);
   super.build_phase(phase);

   `cn_get_intf(virtual my_intf.mon_mp, "my_pkg::intf", intf_name, vi)
   item_port = new("item_port", this);
   upstream_imp = new("upstream_imp", this);
   downstream_port = new("downstream_port", this);
endfunction : build_phase

////////////////////////////////////////////
// func: end_of_elaboration_phase
virtual function void end_of_elaboration_phase(uvm_phase phase);
   super.end_of_elaboration_phase(phase);
   `global_add_to_heartbeat_mon();
endfunction : end_of_elaboration_phase

////////////////////////////////////////////
// func: run_phase
// Launch the monitor and handle reset gracefully
virtual task run_phase(uvm_phase phase);
   forever begin
      @(posedge vi.rst_n);

      fork
         monitor();
      join_none

      @(negedge vi.rst_n);
      disable fork;

      exp_items.delete();
   end
endtask : run_phase

////////////////////////////////////////////
// func: shutdown_phase
// Ensure that shutdown phase doesn't end until we've seen all expected items
virtual task shutdown_phase(uvm_phase phase);
   forever begin
      if(exp_items.size()) begin
         phase.raise_objection(this, $sformatf("Expecting %0d items.", exp_items.size()));
         `cn_info(("Waiting for %0d items.", exp_items.size()))
         wait(exp_items.size() == 0);
         phase.drop_objection(this, "inactive");
      end
      wait(exp_items.size() > 0);
   end
endtask : shutdown_phase

////////////////////////////////////////////
// func: monitor
// Monitor the sbus interface and reconstruct packets
task monitor();
   byte unsigned cycles[$];

   forever begin
      @(posedge vi.mon_cb.valid);
      `global_heartbeat("activity seen")

      while(vi.mon_cb.valid) begin
      cycles.push_back(vi.mon_cb.data);
      @(vi.mon_cb);
   end

   item_rcvd(cycles);
```

```
            cycles.delete();
         end
      endtask : monitor

      ////////////////////////////////////////
      // func: item_rcvd
      // Called when a complete item has been seen
      virtual function void item_rcvd(byte unsigned _cycles[$]);
         byte unsigned stream[];
         item_c item = item_c::type_id::create("item");
         item_c exp_item = exp_items.pop_front();

         if(exp_item)
            item.uid = exp_item.uid;

         stream = new[_cycles.size()](_cycles);
         item.data = new[_cycles.size()];
         item.unpack_bytes(stream);

         `cn_dbg(100, ("Monitored: %s", item.convert2string()))
         item_port.write(item);
      endfunction : item_rcvd

      ////////////////////////////////////////
      // func: write_upstream
      // The implementation for the upstream_imp, to gather the expected items
      // (used only for UID mapping, but could be used for more)
      virtual function write_upstream(item_c _item);
         exp_items.push_back(_item);
         downstream_port.write(_item);
      endfunction : write_upstream
   endclass : mon_c
```

Design Pattern for Library Virtual Sequences

Library sequences for virtual sequences are unfortunately not supported by UVM 1.1. The design pattern below provides an example of how you might structure one. It relies upon its own cfg class being present in its virtual sequencer to provide it with the number of sequences to run, as well as their random distribution.

```
137.
   class lib_vseq_c extends uvm_sequence;
      `uvm_object_utils_begin(cmr_pkg::lib_vseq_c)
      `uvm_object_utils_end
      `uvm_declare_p_sequencer(vsqr_c)

      //------------------------------------------------------------------------------
      // Fields

      // field: outstanding
      // An assoc. array of all the outstanding sequences
      base_vseq_c outstanding[cn_pkg::uid_c];

      // field: max_outstanding
      // The most we should launch at once, to make sure we don't get too crazy
      int unsigned max_outstanding;

      // field: seq_cfg
      // Reference to sequence config component
      lib_seq_cfg_c lib_seq_cfg;

      //------------------------------------------------------------------------------
      // Methods
      function new(string name="lib_vseq");
         super.new(name);
      endfunction : new
```

```
//////////////////////////////////////////////
// func: body
virtual task body();
   lib_seq_cfg = p_sequencer.cfg.lib_seq_cfg;
   max_outstanding = lib_seq_cfg.max_outstanding;
   assert(max_outstanding) else
      `cn_fatal(("lib_seq_cfg.max_outstanding cannot be zero!"))

   `cn_seq_raise

   `cn_info(("Launching %0d sequences.", lib_seq_cfg.cnt))

   //.....................
   fork
      launcher();
      reporter();
   join_any

   disable fork;

   `cn_info(("Exerciser complete after %0d sequences.", lib_seq_cfg.cnt))
   `cn_seq_drop
endtask : body

//////////////////////////////////////////////
// func: launcher
// Launch the sequences and wait for them all to finish
virtual task launcher();
   int num_sent;
   uvm_object_wrapper wrap;
   string type_name;

   do begin
      automatic base_vseq_c vseq;

      // wait here if necessary
      wait(outstanding.size() < max_outstanding);

      // break if the number sent has reached the count
      if(num_sent >= lib_seq_cfg.cnt)
         break;

      // decide which type to send
      randcase
         lib_seq_cfg.alpha_wt   : wrap = alpha_vseq_c::type_id::get();
         lib_seq_cfg.beta_wt    : wrap = beta_vseq_c::type_id::get();
         lib_seq_cfg.charlie_wt : wrap = charlie_vseq_c::type_id::get();
      endcase
      type_name = wrap.get_type_name();

      $cast(vseq, create_item(wrap, m_sequencer, type_name));
      `cn_dbg(50, ("%s Launching %s", vseq.uid.convert2string(), type_name))
      vseq.randomize();
      outstanding[vseq.uid] = vseq;
      fork
         begin
            automatic string _type_name = type_name;
            `uvm_send(vseq)
            num_sent++;
            outstanding.delete(vseq.uid);
         end
      join_none
   end while(num_sent < lib_seq_cfg.cnt);

   // wait for all to finish
   wait(outstanding.size() == 0);

endtask : launcher

//////////////////////////////////////////////
// func: reporter
```

```
      // Report all of the outstanding vseqs whenever there is a heartbeat
      virtual task reporter();
         forever begin
            // do this everytime the heartbeat monitor samples
            @(global_pkg::env.heartbeat_mon.sampled);
            `cn_info(("These sequences remain outstanding:"))
            foreach(outstanding[uid])
               `cn_dbg(200, ("%s", uid.convert2string()))
         end
      endtask : reporter
   endclass : lib_vseq_c
```

This design pattern has a number of interesting features to note.

```
      // field: outstanding
      // An assoc. array of all the outstanding sequences
      base_vseq_c outstanding[cn_pkg::uid_c];
```

All of the virtual sequences that can be sent by this sequence must derive from a `base_vseq_c` class to ensure that they each have their own unique ID (`uid_c`). The library sequence tracks each of the outstanding sequences that it launches by keeping them in this associative array. Their unique IDs are used by the `reporter` task to help users identify which of the launched sequences are still outstanding should the simulation deadlock.

```
      uvm_object_wrapper wrap;
```

The library sequence is simplified by treating all of the sequences that it may launch the same. The `uvm_object_wrapper` is a UVM class that is passed to `create_item` so that we can assign it to the base virtual sequence, `base_vseq_c`.

```
         // decide which type to send
         randcase
            lib_seq_cfg.alpha_wt   : wrap = alpha_vseq_c::type_id::get();
            lib_seq_cfg.beta_wt    : wrap = beta_vseq_c::type_id::get();
            lib_seq_cfg.charlie_wt : wrap = charlie_vseq_c::type_id::get();
         endcase
         type_name = wrap.get_type_name();

         $cast(vseq, create_item(wrap, m_sequencer, type_name));
```

The library sequence uses a `randcase` statement and the weights supplied to it by its own cfg class to randomly select which type of sequence to launch.

```
         fork
            begin
               automatic string _type_name = type_name;
               `uvm_send(vseq)
               num_sent++;
               outstanding.delete(vseq.uid);
            end
         join_none
```

For every virtual sequence that this sequence launches, an individual thread is spawned that sends it and waits for its completion. When the sequence completes, it is deleted from the outstanding array. When all of the sequences have been launched and no more are outstanding, then the library sequence can finish.